# FUTURE VISION BIE

## One Stop for All Study Materials
## & Lab Programs

### Future Vision

## By K B Hemanth Raj

## Scan the QR Code to Visit the Web Page



## Or
## Visit : https://hemanthrajhemu.github.io

## Gain Access to All Study Materials according to VTU,
## CSE – Computer Science Engineering,
## ISE – Information Science Engineering,
## ECE - Electronics and Communication Engineering
## & MORE…

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

# INTRODUCTION TO
# EMBEDDED SYSTEMS

**Shibu K V**

*Technical Architect*
*Mobility & Embedded Systems Practice*
*Infosys Technologies Ltd.,*
*Trivandrum Unit, Kerala*

# 3

# 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. The following topics give an overview of the characteristics and quality attributes of an embedded system.

## 3.1 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

### 3.1.1 Application and Domain Specific

If you closely observe any embedded system, you will find that each embedded system is having certain functions to perform and they are developed in such a manner 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 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 airconditioner 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.

### 3.1.2 Reactive and Real Time

As mentioned earlier, 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.

### 3.1.3 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.

### 3.1.4 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.

### 3.1.5  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 corns 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.

### 3.1.6  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.

## 3.2  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'.

### 3.2.1  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

*3.2.1.1 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 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 within this particular timeline.

*3.2.1.2 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.

*3.2.1.3 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.

*3.2.1.4 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 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. 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

$$A_i = MTBF/(MTBF + MTTR)$$

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

*3.2.1.5 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 unauthorised modification. 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 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.

*3.2.1.6 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).

## 3.2.2  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.

*3.2.2.1 Testability & Debug-ability* Testability 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 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.

*3.2.2.2 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.

*3.2.2.3 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 Hitachi SH3 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 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 will not function on other operating systems; whereas applications developed using 'Java' from Sun Microsystems works on any operating system that supports Java standards.

*3.2.2.4 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.

*3.2.2.5 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. 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.
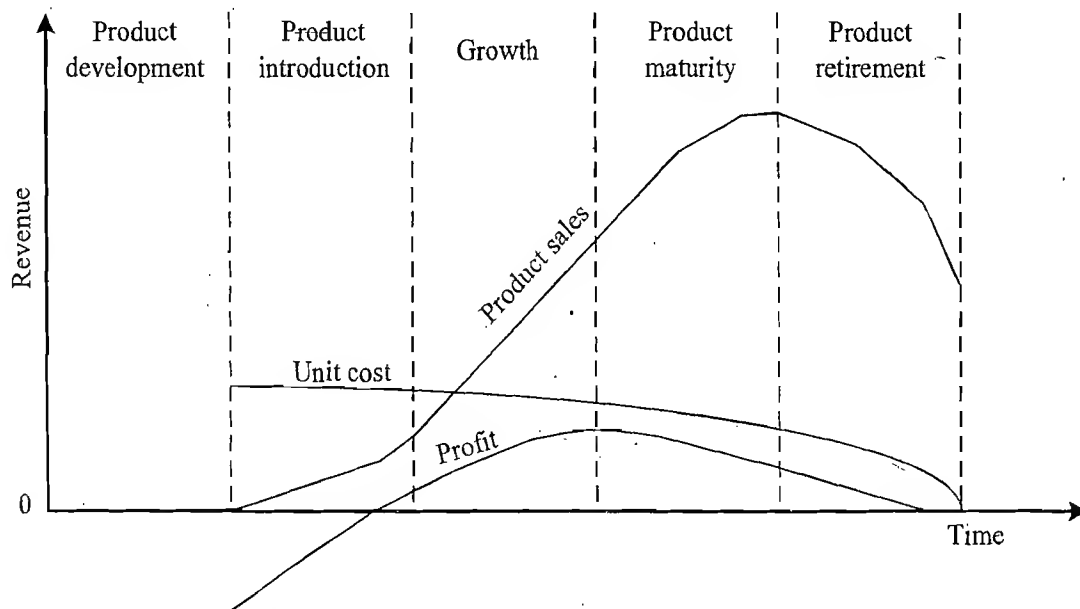


**Fig. 3.1** **Product life cycle (PLC) curve**

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 profit 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.

## Summary

✓ There exists a set of characteristics which are unique to each embedded system.
✓ Embedded systems are application and domain specific.
✓ Quality attributes of a system represents the non-functional requirements that need to be documented properly in any system design.
✓ The operational quality attributes of an embedded system refers to the non-functional requirements that needs to be considered for the operational mode of the system. Response, Throughput, Reliability, Maintainability, Security, Safety, etc. are examples of operational quality attributes.
✓ The non-operational quality attributes of an embedded system refers to the non-functional requirements that needs to be considered for the non-operational mode of the system. Testability, debug-ability, evolvability, portability, time-to-prototype and market, per unit cost and revenue, etc. are examples of non-operational quality attributes.
✓ The product life cycle curve (PLC) is the graphical representation of the unit cost, product sales and profit with respect to the various life cycle stages of the product starting from conception to disposal.
✓ For a commercial embedded product, the unit cost is peak at the introductory stage and it falls in the maturity stage.
✓ The revenue of a commercial embedded product is at the peak during the maturity stage.

## Keywords

| | | |
|---|---|---|
| **Quality attributes** | : | The non-functional requirements that need to be addressed in any system design |
| **Reactive system** | : | An embedded system which produces changes in output in response to the changes in input |
| **Real-Time system** | : | A system which adheres to strict timing behaviour and responds to requests in a known amount of time. |
| **Response** | : | It is a measure of quickness of the system |
| **Throughput** | : | The rate of production or operation of a defined process over a stated period of time |
| **Reliability** | : | It is a measure of how much % one can rely upon the proper functioning of a system |
| **MTBF** | : | Mean Time Between Failures–The frequency of failures in hours/weeks/months |
| **MTTR** | : | Mean Time To Repair–Specifies how long the system is allowed to be out of order following a failure |
| **Time-to-prototype** | : | A measure of the time required to prototype a design |
| **Product life-cycle (PLC)** | : | The representation of the different stages of a product from its conception to disposal |
| **Product life cycle curve** | : | The graphical representation of the unit cost, product sales and profit with respect to the various life cycle stages of the product starting from conception to disposal |

## Objective Questions

1. Embedded systems are application and domain specific. State True or False
   - (a) True
   - (b) False
2. Which of the following is true about Embedded Systems?
   - (a) Reactive and Real Time
   - (b) Distributed
   - (c) Operates in harsh environment
   - (d) All of these
   - (e) None of these
3. Which of the following is a distributed embedded system?
   - (a) Cell phone
   - (b) Notebook Computer
   - (c) SCADA system
   - (d) All of these
   - (e) None of these
4. Quality attributes of an embedded system are
   - (a) Functional requirements
   - (b) Non-functional requirements
   - (c) Both
   - (d) None of these
5. Response is a measure of
   - (a) Quickness of the system
   - (b) How fast the system tracks changes in Input
   - (c) Both
   - (d) None of these
6. Throughput of an embedded system is a measure of
   - (a) The efficiency of the system
   - (b) The output over a stated period of time
   - (c) Both
   - (d) None of these
7. Benchmark is
   - (a) A reference point
   - (b) A set of performance criteria
   - (c) (a) or (b)
   - (d) None of these
8. Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR) defines the reliability of an embedded system. State True or False
   - (a) True
   - (b) False
9. MTBF gives the frequency of failures of an embedded system. State True or False
   - (a) True
   - (b) False
10. Which of the following is true about the quality attribute 'maintainability'?
    - (a) The corrective maintainability requirement for a highly reliable embedded system is very less
    - (b) Availability of an embedded system is directly related to the maintainability of the system
    - (c) Both of these
    - (d) None of these
11. The Mean Time Between Failure (MTBF) for an embedded product is very high. This means:
    - (a) The product is highly reliable
    - (b) The availability of the product is very high
    - (c) The preventive maintenance requirement for the product is very less
    - (d) All of these
    - (e) None of these
12. The Mean Time Between Failure (MTBF) of an embedded product is 4 months and the Mean Time To Repair (MTTR) of the product is 2 weeks. What is the availability of the product?
    - (a) 100%
    - (b) 50%
    - (c) 89%
    - (d) 10%
13. Which of the following are the three measures of information security in embedded systems?
    - (a) Confidentiality, secrecy, integrity
    - (b) Confidentiality, integrity, availability
    - (c) Confidentiality, transparency, availability
    - (d) Integrity, transparency, availability

14. You are working on a mission critical embedded system development project for a client and the client and your company has signed a Non Disclosure Agreement (NDA) on the disclosure of the project-related information. You share the details of the project you are working with your friend. Which aspect of Information security you are violating here?
    (a) Integrity        (b) Confidentiality        (c) Availability        (d) None of these

15. Which of the following is an example of 'gradual' safety threat from an embedded system?
    (a) Product blast due to overheating of the battery     (b) UV emission from the embedded product
    (c) Both of these        (d) None of these

16. Non operational quality attributes are
    (a) Non-functional requirements        (b) Functional requirements
    (c) Quality attributes for an offline product        (d) (a) and (c)
    (e) None of these

17. Which of the following is (are) an operational quality attribute?
    (a) Testability        (b) Safety        (c) Debug-ability      (d) Portability
    (e) All of these

18. Which of the following is (are) non-operational quality attribute?
    (a) Reliability        (b) Safety        (c) Maintainability        (d) Portability
    (e) All of these        (f) None of these

19. In the Information security context, Confidentiality deals with the protection of data and application from unauthorised disclosure. State True or False
    (a) True        (b) False

20. What are the two different aspects of debug-ability in the embedded system development context?
    (a) Hardware & Firmware debug-ability        (b) Firmware & Software debug-ability
    (c) None of these

21. For an embedded system, the quality attribute 'Evolvability' refers to
    (a) The upgradability of the product        (b) The modifiability of the product
    (c) Both of these        (d) None of these

22. Portability is a measure of 'system independence'. State True or False
    (a) True        (b) False

23. For a commercial embedded product the *unit cost* is high during
    (a) Product launching        (b) Product maturity
    (c) Product growth        (d) Product discontinuing

24. For a commercial embedded product the sales volume is high during
    (a) Product launching        (b) Product maturity
    (c) Product growth        (d) Product discontinuing

## Review Questions

1. Explain the different characteristics of embedded systems in detail.
2. Explain quality attribute in the embedded system development context? What are the different Quality attributes to be considered in an embedded system design.
3. What is operational quality attribute? Explain the important operational quality attributes to be considered in any embedded system design.
4. What is non-operational quality attribute? Explain the important non-operational quality attributes to be considered in any embedded system design.
5. Explain the quality attribute *Response* in the embedded system design context.
6. Explain the quality attribute *Throughput* in the embedded system design context.

7. Explain the quality attribute *Reliability* in the embedded system design context.
8. Explain the quality attribute *Maintainability* in the embedded system design context.
9. The availability of an embedded product is 90%. The Mean Time Between Failure (MTBF) of the product is 30 days. What is the Mean Time To Repair (MTTR) in days/hours for the product?
10. Explain the quality attribute *Information Security* in the embedded system design context.
11. Explain the quality attribute *Safety* in the embedded system design context.
12. Explain the significance of the quality attributes *Testability and Debug-ability* in the embedded system design context.
13. Explain the quality attribute *Portability* in the embedded system design context.
14. Explain *Time-to-market*? What is its significance in product development?
15. Explain *Time-to-prototype*? What is its significance in product development?
16. Explain the *Product Life-cycle* curve of an embedded product development.

# 4 Embedded Systems—Application- and Domain-Specific

**LEARNING OBJECTIVES**

✓ Illustrate the domain and application specific aspect of embedded systems with examples
✓ Know the presence of embedded systems in automotive industry
✓ Learn about High Speed Electronic Control Units (HECUs) and Low Speed Electronic Control Units (LECUs) employed in automotive applications
✓ Learn about the Controller Area Network (CAN), Local Interconnect Network (LIN) and Media Oriented System Transport (MOST) communication buses used in automotive applications
✓ Know the semiconductor chip providers, tools and platform providers and solution providers for automotive embedded applications

As mentioned in the previous chapter on the characteristics of embedded systems, 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. In general purpose computing, it is possible to replace a 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 specialised 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. The following sections are intended to give the readers some idea on the application and domain specific characteristics of embedded systems.

## 4.1 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 system providing extensive support in home automation applications (Fig. 4.1).

As mentioned in an earlier chapter, 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. 4.2.
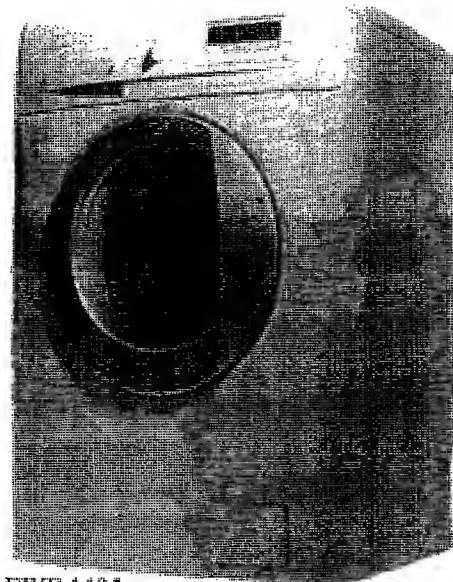
EWF 1495

Fig. 4.1  **Washing Machine – Typical example of an embedded system**
*(Photo courtesy of Electrolux Corporation (www.electrolux.com/au))*



*Picture not to scale*
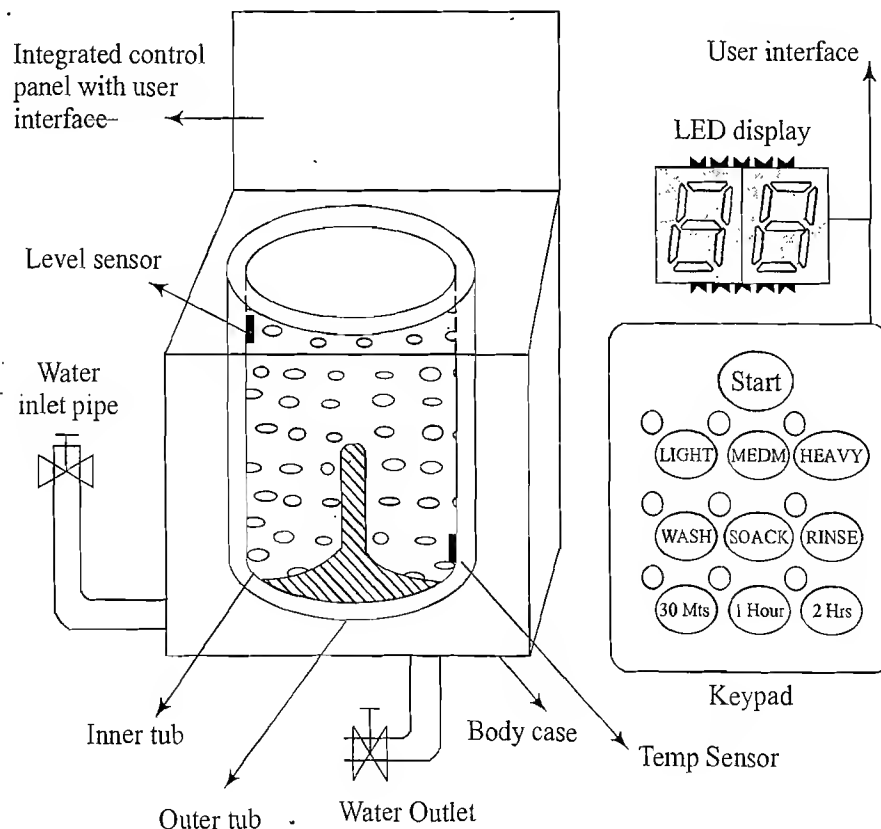
Fig. 4.2  **Washing machine – Functional block diagram**

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 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.

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 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.

## 4.2 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 4.3 gives an overview of the various types of electronic control units employed in automotive applications.

### 4.2.1 Inner Workings 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

---

*Name may vary depending on the manufacturer.

13. Airbag control

14. Power steering

12 Seat control

15. Air conditioner

11. Miror control

1. Instrumentation

10. Power windows

2. Engine control

9. Centralized locking

8. Supension control

7. Wiper control

3. Fan control

5. Headlamp control

6. ABS control

4. Fuel injection control
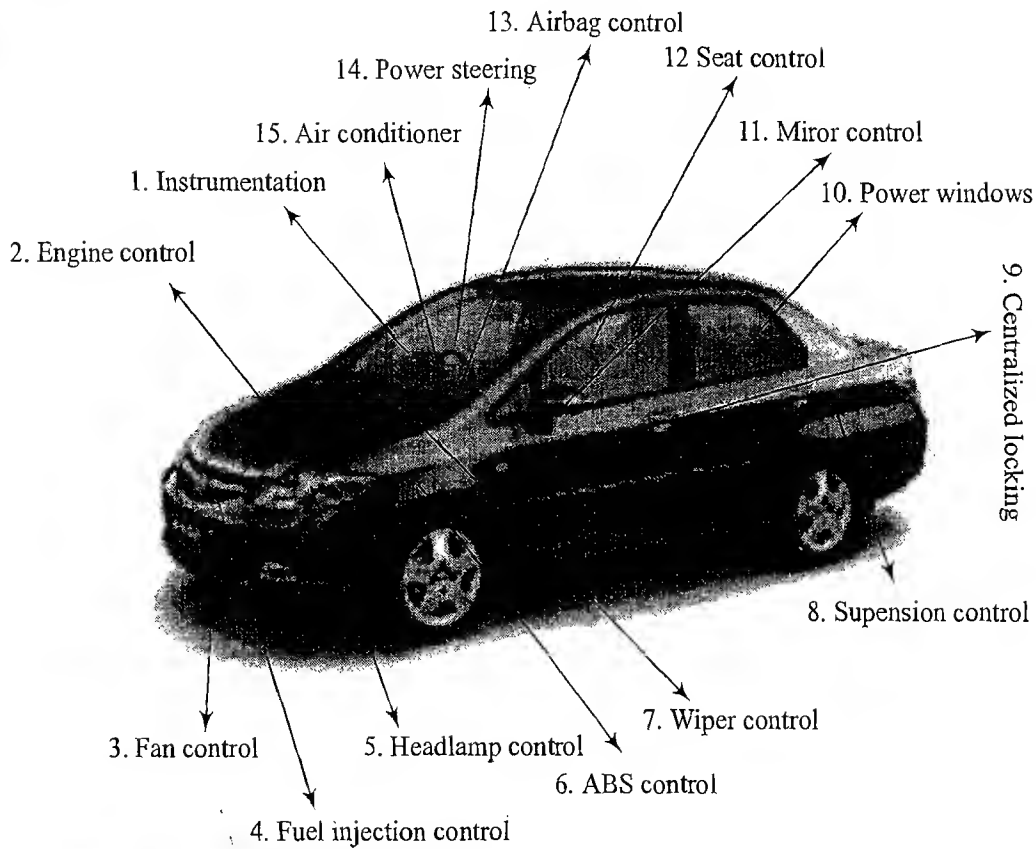
**Fig. 4.3** **Embedded system in the automotive domain**
*(Photo courtesy of Honda Siel Car India (www.hondacarindia.com))*

from 20 to 40 whereas a luxury vehicle like Mercedes S and BMW 7 may contain 75 to 100 numbers of 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.

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.

*4.2.1.1 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.

*4.2.1.2 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.

### 4.2.2 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.

*4.2.2.1 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.

*4.2.2.2 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.

*4.2.2.3 Media-Oriented System Transport (MOST) Bus* The Media-oriented system transport (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.

### 4.2.3 Key Players of the Automotive Embedded Market

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

*4.2.3.1 Silicon Providers* 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:

**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.

**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, rear seat entertainment, adaptive cruise control, voice recognition, etc.

**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], AVR®[2], 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.

**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.

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

**Renesas** (www.renesas.com): Provider of high speed microcontrollers and Large Scale Integration (LSI) technology for car navigation systems accommodating three transfer speeds: high, medium and low.

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

**Fujitsu** (www.fmal.fujitsu.com): Supplier of fingerprint sensors for security applications, graphic display controller for instrumentation application, AGPS/GPS for vehicle navigation system and different types of microcontrollers for automotive control applications.

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

**NEC** (www.nec.co.jp): Provider of high performance microcontrollers.

There are lots of other silicon manufactures which provides various automotive support systems like power supply, sensors/actuators, optoelectronics, etc. Describing all of them is out of the scope of this book. Readers are requested to use the Internet for finding more information on them.

*4.2.3.2 Tools and Platform Providers* Tools 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. Sometimes the silicon suppliers provide the development suite for application development using their chip. Some third party suppliers may also provide development kits and libraries. Some of the leading suppliers of tools and platforms in automotive embedded applications are listed below.

**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.

**The MathWorks** (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, visualisation, modelling 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.

---

[1] ARM® is the registered trademark of ARM Holdings.

[2] AVR® is the registered trademark of Atmel Corporation.

**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.

**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.

**ARTiSAN** (www.artisansw.com): Is the leading supplier of collaborative modelling tools for requirement analysis, specification, design and development of complex applications.

**Microsoft** (www.microsoft.com): It is a platform provider for automotive embedded applications. Microsoft's WindowsCE is a powerful RTOS platform for automotive applications. Automotive features are included in the new WinCE Version for providing support for automotive application developers.

*4.2.3.3 Solution Providers*  Solution providers supply 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.

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

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

**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.

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

……and many more. The list is incomplete. Describing all providers is out of the scope of this book.

**Summary**

✓ Embedded systems designed for a particular application for a specific domain cannot be replaced with another embedded system designed for another application for a different domain

✓ Consumer, industrial, automotive, telecom, etc. are the major application domains of embedded systems. Telecom and automotive industry are the two segments holding a big market share of embedded systems

✓ 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)

✓ High speed Electronic Control Units (HECUs) are deployed in critical control units requiring fast response, like fuel injection systems, antilock brake system, etc.

✓ Low speed Electronic Control Units (LECUs) are deployed in applications where response time is not so critical. They are generally built around low cost microprocessors/microcontrollers and digital signal processors. Audio controllers, passenger and driver door locks, door glass controls, etc., are examples for LECUs.

✓ Automotive applications use serial buses for communication. Controller Area Network (CAN), Local Interconnect Network (LIN), Media Oriented System Transport (MOST) bus, etc. are the important automotive communication buses.

✓ CAN is an event driven serial protocol interface with support for error handling in data transmission. It is generally employed in safety system like airbag control, powertrain systems like engine control and Antilock Brake Systems (ABS).

✓ 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 20Kbps and is used for sensor/actuator interfacing.

✓ The Media Oriented System Transport (MOST) bus is targeted for automotive audio video equipment interfacing. MOST bus is a multimedia fibre-optic point-to-point network implemented in a star, ring or daisy-chained topology over optical fibres cables

✓ The key players of the automotive embedded market can be classified into 'Silicon Providers', 'Tools and Platform Providers' and 'Solution Providers'

## Keywords

**ECU** : Electronic Control Unit. The generic term for the embedded control units in automotive application

**HECU** : High-speed Electronic Control Unit. The high-speed embedded control unit deployed in automotive applications

**LECU** : Low-speed Electronic Control Unit. The low-speed embedded control unit deployed in automotive applications

**CAN** : Controller Area Network. An event driven serial protocol interface used primarily for automotive applications

**LIN** : Local Interconnect Network. A single master multiple slave, low speed serial bus used in automotive application

**MOST** : Media Oriented System Transport Bus. A multimedia fibre-optic point-to-point network implemented in a star, ring or daisy-chained topology over optical fibres cables

## Objective Questions

1. In Automotive systems, High-speed Electronic Control Units (HECUs) are deployed in
   (a) Fuel injection systems   (b) Antilock brake systems
   (c) Power windows   (d) Wiper control   (e) Only (a) and (b)

2. In Automotive systems, Low speed electronic control units (LECUs) are deployed in
   (a) Electronic throttle   (b) Steering controls   (c) Transmission control   (d) Mirror control

3. The first embedded system used in automotive application is the microprocessor based fuel injection system introduced by _____ in 1968
   (a) BMW   (b) Volkswagen 1600   (c) Benz E Class   (d) KIA

4. CAN bus is an event driven protocol for communication. State True or False
   (a) True   (b) False

5. Which of the following serial bus is (are) used for communication in Automotive Embedded Applications?
   (a) Controller Area Network (CAN)   (b) Local Interconnect Network (LIN)
   (c) Media Oriented System Transport (MOST) bus   (d) All of these   (e) None of these

6. Which of the following is true about LIN bus?
   (a) Single master multiple slave interface   (b) Low speed serial bus
   (c) Used for sensor/actuator interfacing   (d) All of these   (e) None of these

7. Which of the following is true about MOST bus?
   (a) Used for automotive audio video system interfacing
   (b) It is a fibre optic point-to-point network
   (c) It is implemented in star, ring or daisy-chained topology
   (d) All of these          (e) None of these
8. Which of the following is (are) example(s) of Silicon providers for automotive applications?
   (a) Maxim/Dallas          (b) Analog Devices          (c) Xilinx          (d) Atmel
   (e) All of these          (f) None of these

## Review Questions

1. Explain the role of embedded systems in automotive domain.
2. Explain the different electronic control units (ECUs) used in automotive systems.
3. Explain the different communication buses used in automotive application.
4. Give an overview of the different market players of the automotive embedded application domain.

# 7

# Hardware Software Co-Design and Program Modelling

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.

During the co-design process, the product requirements captured from the customer are converted into system level needs or processing requirements. At this point of time it is not segregated as either hardware requirement or software requirement, instead it is specified as functional requirement. The system level processing requirements are then transferred into functions which can be simulated and verified against performance and functionality. The Architecture design follows the system design. The partition of system level processing requirements into hardware and software takes place during the architecture design phase. Each system level processing requirement is mapped as either hardware and/or

software requirement. The partitioning is performed based on the hardware-software trade-offs. We will discuss the various hardware software tradeoffs in hardware software co-design in a separate topic. The architectural design results in the detailed behavioural description of the hardware requirement and the definition of the software required for the hardware. The processing requirement behaviour is usually captured using computational models and ultimately the models representing the software processing requirements are translated into firmware implementation using programming languages.

## 7.1 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* 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 (which we will discuss in a later section) using a state register and two combinational circuits (we will discuss about combinational circuits in a later chapter). 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. 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 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 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.

***Partitioning System Requirements into hardware and software*** So far we discussed about the models for capturing the system requirements and the architecture for implementing the system. From an implementation perspective, it may be possible to implement the system requirements in either hardware or software (firmware). It is a tough decision making task to figure out which one to opt. Various hardware software trade-offs are used for making a decision on the hardware-software partitioning. We will discuss them in detail in a later section of this chapter.

## 7.2 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.

### 7.2.1 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 7.1 illustrates the implementation of a DFG model for implementing these requirements.
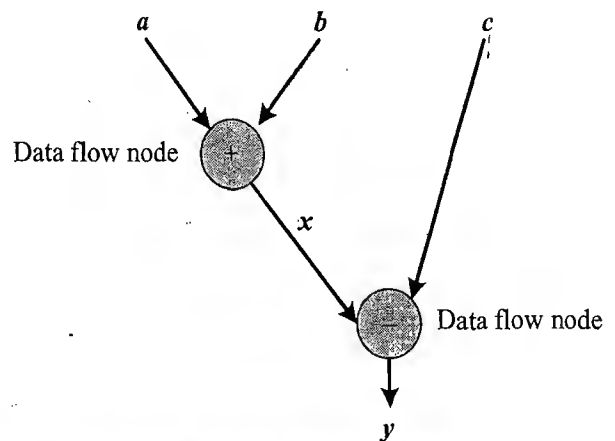


**Fig. 7.1** Data flow graph (DFG) model

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

### 7.2.2 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 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 Fig. 7.2.

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.

## 7.2.3 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



**Fig. 7.2** Control Data Flow Graph (CDFG) Model

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. 7.3.

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 Fig. 7.4.

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
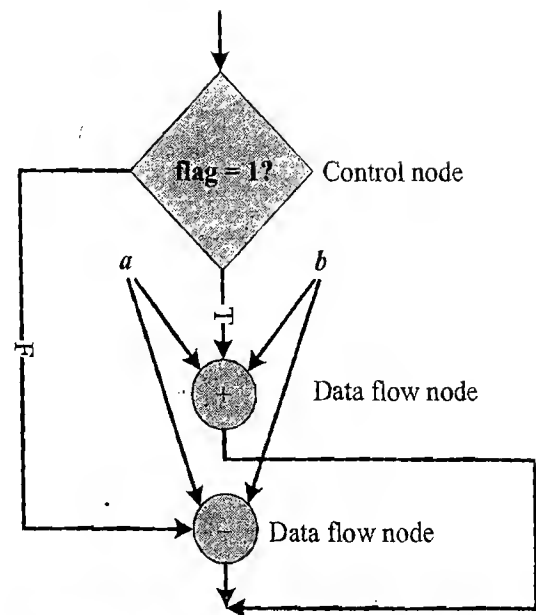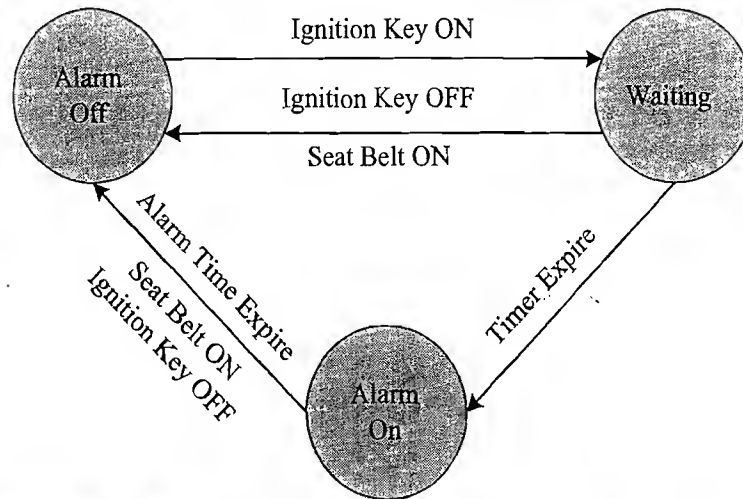
**Fig. 7.3** FSM Model for Automatic seat belt warning system
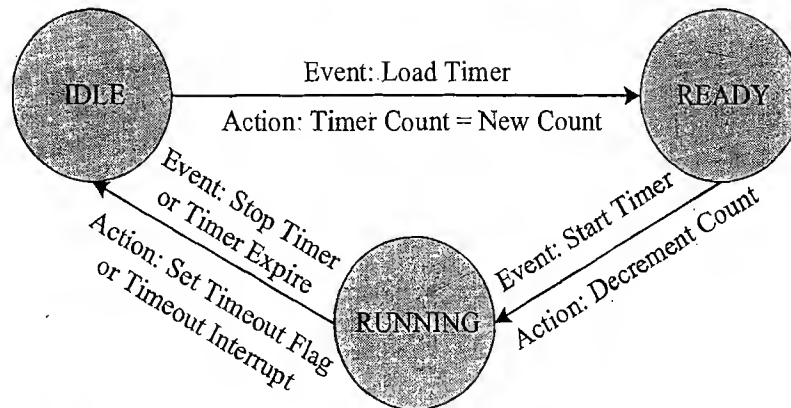


**Fig. 7.4** FSM Model for timer

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 I

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. 7.5.

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

State A: Wait for coin
State B: Wait for user input
State C: Dispense tea
State D: Dispense coffee
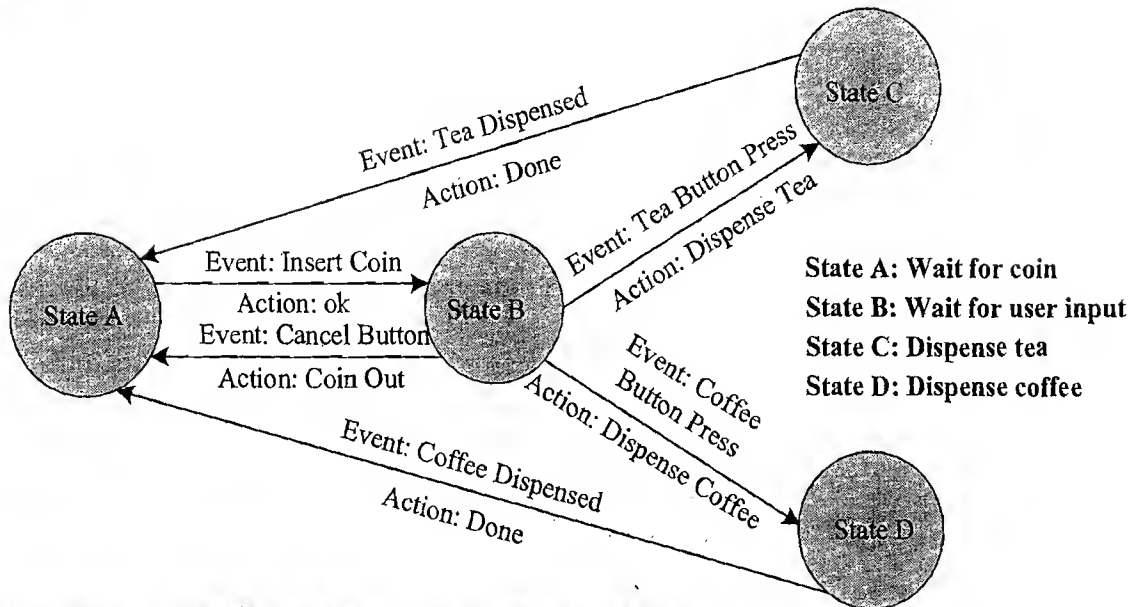
**Fig. 7.5** FSM Model for Automatic Tea\Coffee Vending Machine

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.

## 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. 7.6, 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. The Harel Statechart, UML State diagram, etc. are examples for popular statecharts used for the HCFSM modelling of embedded systems. In statecharts, 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 parenthesized condition is also labeled with the arrow. The condition specifies on what

State A: Ready
State B: Wait for coin
State C: Wait for number
State D: Dialling

State E: Call in progress
State F: Call terminated
State G: Unable to make call
State H: Invalid number input
State I: Out of order
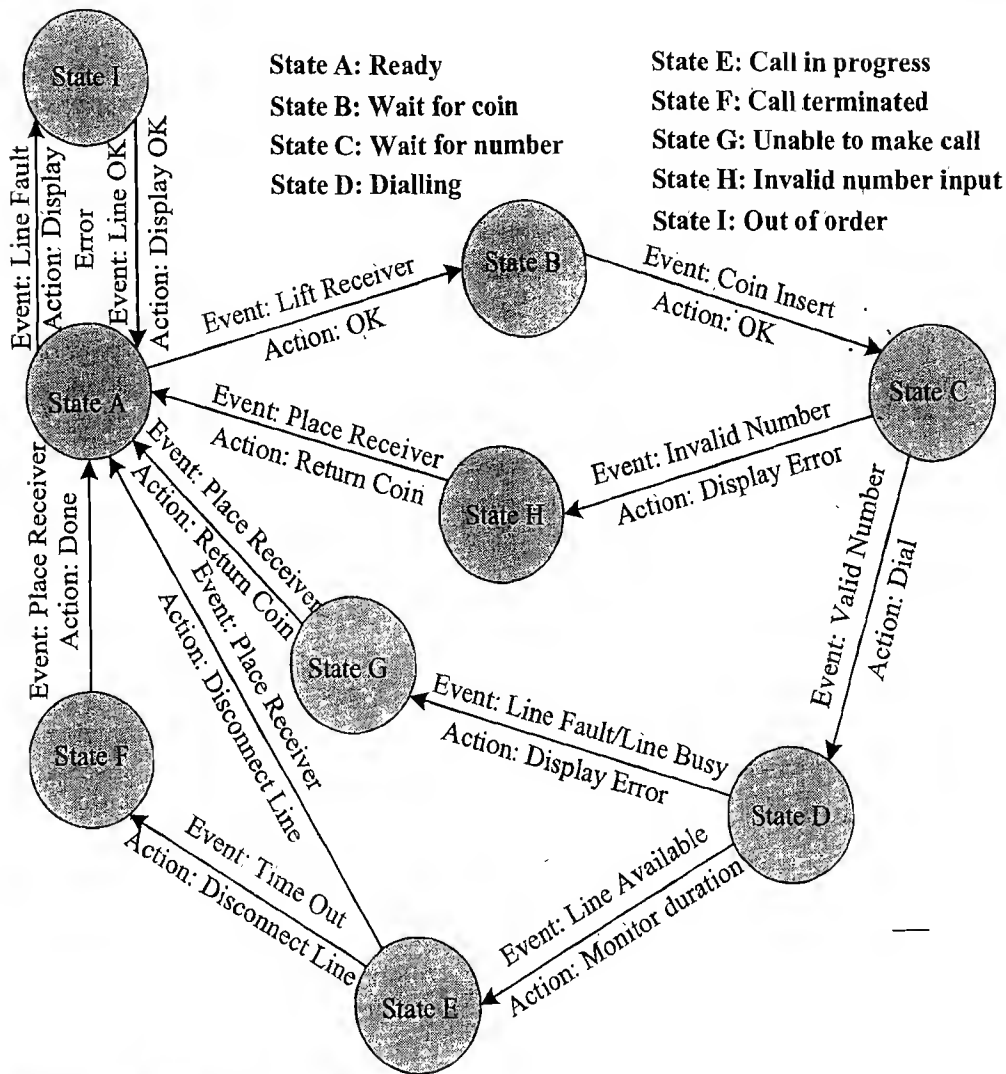
**Fig. 7.6** **FSM Model for Coin Operated Telephone System**

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. The IAR *visualSTATE* (http://www.iar.com/website1/1.0.1.0/371/1/index.php) from IAR systems is a popular visual modelling tool for embedded applications.

## 7.2.4 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.

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

## 7.2.5 Concurrent/Communicating Process Model

The concurrent or communicating process model models concurrently executing tasks/processes. So far we discussed about the sequential execution of software programs. It is easier to implement certain requirements in concurrent processing model than the conventional sequential execution. Sequential execution leads to a single sequential execution of task and thereby leads to poor processor utilisation, when the task involves I/O waiting, sleeping for specified duration etc. If the task is split into multiple subtasks, it is possible to tackle the CPU usage effectively, 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)
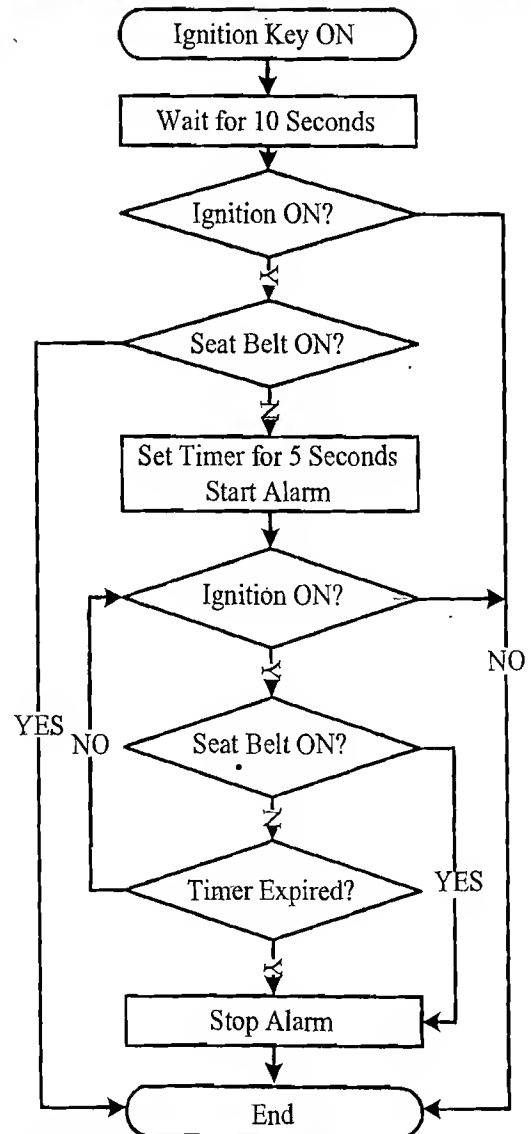


Fig. 7.7 Sequential Program Model for seat belt warning system

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 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. 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. The *alarm_timer_expire* event will be in the reset state initially and it is set when the alarm timer expires. The alarm 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_expire* signals. 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 7.8 illustrates the same.

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. We will discuss these techniques in a later chapter.

### 7.2.6 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.

Create and initialize events

*wait_timer_expire, ignition_on, ignition_off,*

*seat_belt_on, seat_belt_off,*

*alarm_timer_start, alarm_timer_expire*

Create task *Wait Timer*

Create task *Ignition Key Status Monitor*

Create task *Seat Belt Status Monitor*

Create task *Alarm Control*

Create task *Alarm Timer*

(a)

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

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

**Alarm Control Task**
Wait for the 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();
}

**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;
}
}

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

(b)

**Fig. 7.8** (a) Tasks for 'Seat Belt Warning System' (b) Concurrent processing Program model for 'Seat Belt Warning System'

## 7.3   INTRODUCTION TO UNIFIED MODELLING LANGUAGE (UML)

Unified Modelling Language (UML) is a visual modelling language for Object Oriented Design (OOD). UML helps in all phases of system design through a set of unique diagrams for requirements capturing, designing and deployment.

### 7.3.1　UML Building Blocks

'Things', 'Relationships' and 'Diagrams' are the fundamental building blocks of UML.

*7.3.1.1 Things*　A 'Thing' is an abstraction of the UML model. The 'Things' in UML are classified into:
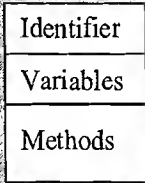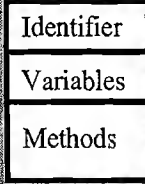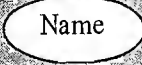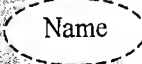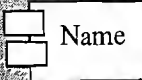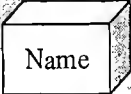
**Structural things:** Represents mostly the static parts of a UML model. They are also known as 'classifiers'. Class, interface, use case, use case realisation (collaboration), active class, component and node are the structural things in UML.
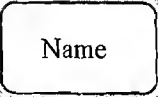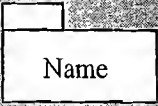
**Behavioural things:** Represents mostly the dynamic parts of a UML model. Interaction, state machine and activity are the behavioural things in UML.

**Grouping things:** Are the organisational parts of a UML model. Package and sub-system are the grouping things in UML.

**Annotational things:** Are the explanatory parts of a UML model. *Note* is the Annotational thing in UML.

The table given below gives a snapshot of various structural, behavioural, grouping and Annotational things in UML.

| Thing | Element | Description | Representation |
|---|---|---|---|
| Structural | Class | A template describing a set of objects which share the same attributes, relationships, operations and semantics. It can be considered as a blueprint of object. | Identifier / Variables / Methods |
| | Active Class | Class presenting a thread of control in the system. It can initiate control activity. Active class is represented in the same way as that of a class but with thick border lines. | Identifier / Variables / Methods |
| | Interface | A collection of externally visible operations which specify a service of a class. It is represented as a circle attached to the class. | (circle symbol) |
| | Use case | Defines a set of sequence of actions. It is normally represented with an ellipse indicating the name. | Name (ellipse) |
| | Collaboration (Use case Realisation) | Interaction diagram specifying the collaboration of different use cases. It is normally represented with a dotted ellipse indicating the name. | Name (dotted ellipse) |
| | Component | Physical packaging of classes and interfaces | Name (component symbol) |
| | Node | A computational resource existing at run time. Represented using a cube with name. | Name (cube) |

| | | | |
|---|---|---|---|
| Behavioural | Interaction | Behaviour comprising a set of objects exchanging messages to accomplish a specific purpose. Represented by arrow with name of operation | Name → |
| | State Machine | Behaviour specifying the sequence of states in response to events, through which an object traverses during its lifetime. | Name |
| Grouping | Package | Organises elements into packages. It is only a conceptual thing. Represented as a tabbed folder with name. | Name |
| Annotational | Note | Explanatory element in UML models. Contains formal informal explanatory text. May also contain embedded image. | Text |
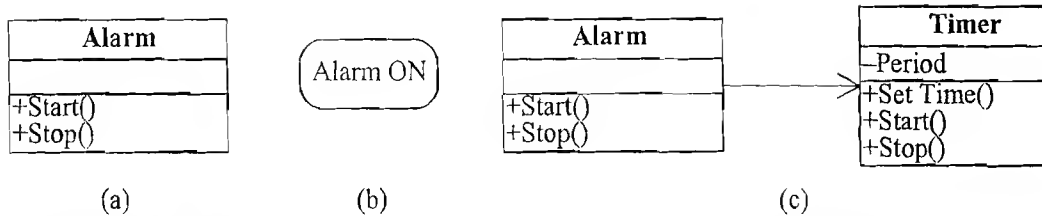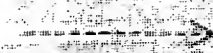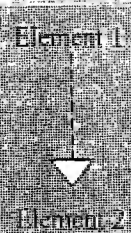
**Fig. 7.9** (a) The Alarm Class. (b) State Representation for 'Alarm ON' state. (c) Alarm – Timer Class interaction for the Seat belt Warning System

**7.3.1.2 Relationships**   As the name indicates, they express the type of relationship between UML elements (objects, classes, etc). The table given below gives a snapshot of the various relationships in UML.

| Relationship | Description | Representation |
|---|---|---|
| Association | It is a structural relationship describing the link between objects. The association can be one-to-one or one-to-many. Aggregation and Composition are the two variants of Association. | relationship → |
| Aggregation | It represents is "a part of" relationship. Represented by a line with a hollow diamond at the end. | a part of ◇ |
| Composition | Aggregation with strong ownership relation to represent the component of a complex object. Represented by a line with a solid diamond at the end. | ◆ |
| Generalisation | Represents a parent-child relationship. The parent may be more generalised and child being specialised version of the parent object. | Child ↓ Parent |

| Dependency | Represents a relationship in which one element (object, class) uses or depends on another element (object, class). Represented by a dotted arrow with head pointing to the dependent element | - - - - → |
| Realisation | The relationship between two elements in which one element realises the behaviour specified by the other element | Element 1 ... ▽ Element 2 |

| **Alarm** | **Audio Visual Indicators** | **Alarm** | **Warning System** |
|---|---|---|---|
| +Start() +Stop() | +Start() +Stop() +Set Type() | +Start() +Stop() | +Alarm() +Display() |

(a)                                          (b)
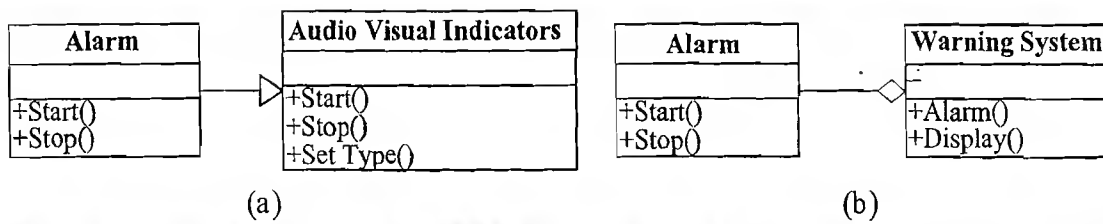
**Fig. 7.10** (a) Alarm is a special type of Audio Visual Indicator (Generalisation), (b) Alarm is a part of Warning System (Aggregation)

**7.3.1.3 UML Diagrams** UML Diagrams give a pictorial representation of the static aspects, behavioural aspects and organisation and management of different modules (classes, packages, etc.) of the system. UML diagrams are grouped into Static Diagrams and Behavioural Diagrams.

**Static Diagrams** Diagram representing the static (structural) aspects of the system. Class Diagram, Object Diagram, Component Diagram, Package Diagram, Composite Structure Diagram and Deployment Diagram falls under this category. The table given below gives a snapshot of various static diagrams.

| Diagram | Description |
|---|---|
| Object diagram | Gives a pictorial representation of a set of objects and their relationships. It represents the structural organisation between objects. |
| Class diagram | Gives a pictorial representation of the different classes in a UML model, their interfaces, the collaborations, interactions and relationship between the classes, etc. It captures the static design of the system. |
| Component diagram | It is a pictorial representation of the implementation view of a system. It comprises components (physical packaging of classes and interfaces), relationships and associations among the components. |
| Package diagram | It is a representation of the organisation of packages and their elements. Package diagrams are mostly used for organising use case diagrams and class diagrams. |
| Deployment diagram | It is a pictorial representation of the configuration of run time processing nodes and the components associated with them. |

**Behavioural Diagrams** These are diagrams representing the dynamic (behavioural) aspects of the system. Use Case Diagram (Fig. 7.11), Sequence Diagram (Fig. 7.12), State Diagram, Communication
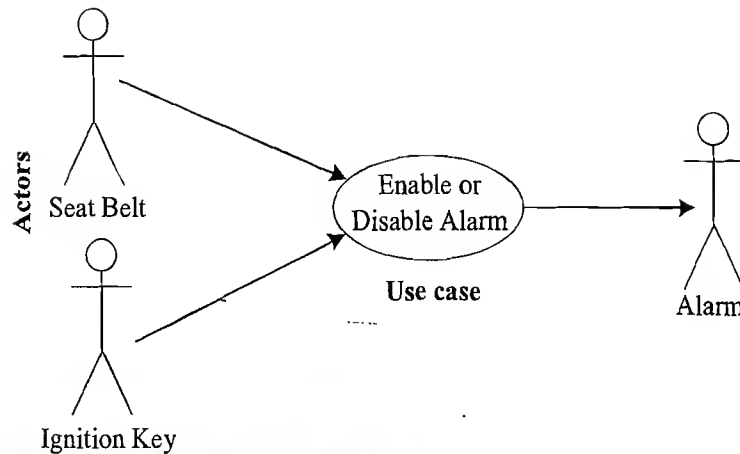
**Fig. 7.11** Use Case diagram for seat belt warning system
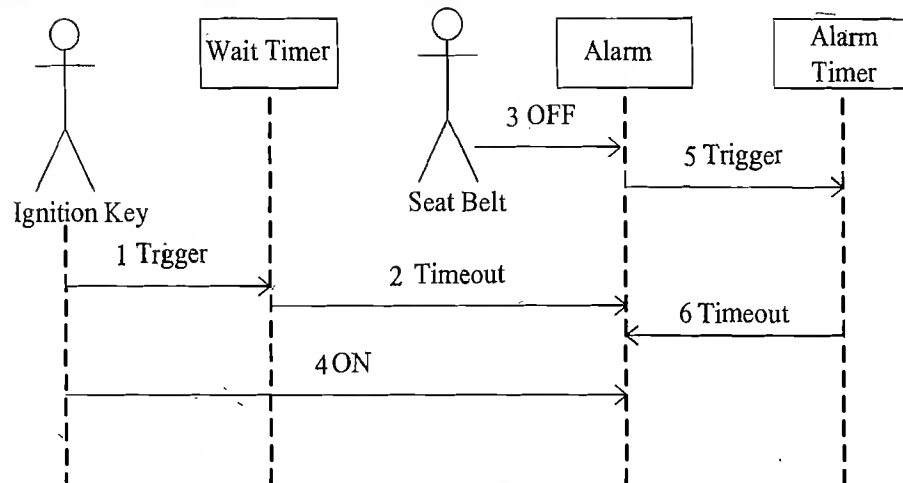


**Fig. 7.12** Sequence diagram for one possible sequence for the seat belt warning system

Diagram, Activity Diagram, Timing Diagram and Interaction Overview Diagram are the behavioural diagrams in UML model. The table given below gives a snapshot of the important behavioural diagrams.

| Diagram | Description |
|---------|-------------|
| Use Case diagram | Use Case diagrams are used for capturing system functionality as seen by users. It is very useful in system requirements capturing. Use case diagram comprise use cases, *actors* (users) and the relationship between them. In use case diagram, an actor is one (or something) who (or which) interacts with the system and use case is the sequence of interaction between the actor and system. |
| Sequence diagram | Sequence diagram is a type of interaction diagram representing object interactions with respect to time. It emphasis on the time ordering of messages. Best suited for the interaction modelling of real-time systems. |
| Collaboration (Communication) diagram | Collaboration or Communication diagram is a type of interaction diagram representing the object interaction and how they are linked together. It gives emphasis to the structural organisation of objects that send and receive messages. In short, it represents the collaboration of objects using messages. |

| State Chart diagram | A diagram showing the states, transitions, events and activities similar to a state machine representation. Best suited for modelling reactive systems. |
|---|---|
| Activity diagram | It is a special type of state chart diagram showing activity to activity transition in place of state transition. It emphasises on the flow control among objects. |

## 7.3.2 The UML Tools

The tools for building UML based models and diagrams are available from different vendors. Some of them are commercial and some of them are either free or open source. The table given below gives a summary of the popular UML modelling tools.

| Tool | Provider/Comments |
|---|---|
| Rational Rose Enterprise | IBM Software (http://www-01.ibm.com/software/awdtools/developer/rose/enterprise/index.html) |
| Rational System Developer | Eclipse based tool from IBM Software (http://www-01.ibm.com/software/awdtools/developer/systemsdeveloper/index.html) |
| Telelogic Rhapsody | UML/SysML based model-driven development tool for real-time or embedded systems from IBM Software (http://www-01.ibm.com/software/awdtools/rhapsody/) |
| Borland® Together® | Borland (www.borland.com) |
| Enterprise Architect | Spark Systems (http://www.sparxsystems.com) |
| ARTiSAN Studio | Artisan Software Tools Inc (http://www.artisansoftwaretools.com) |
| Microsoft® Visio | Microsoft Corporation. The Microsoft Visio (Part of Microsoft® Office product) supports UML model Diagram generation. www.microsoft.com/office/visio |

Some of the tools generate the skeletal code (stub) for the classes and application in programming languages like C++, C#, Java, etc.

## 7.4 HARDWARE SOFTWARE TRADE-OFFS

Certain system level processing requirements may be possible to develop in either hardware or software. The decision on which one to opt is based on the trade-offs and actual system requirement. For example, if the embedded system under consideration involves some multimedia codec[‡] requirement. The media codec can be developed in either software or using dedicated hardware chip (like ASIC or ASSP). Here the trade-off is performance and re-configurability. A codec developed in hardware may be much more efficient, optimised with low processing and power requirements. It is possible to develop the same codec in software using algorithm. But the software implementation need not be optimised for performance, speed and power efficiency. On the other hand, a codec developed in software is re-usable and re-configurable. With certain modification it can be configured for other codec implementations, whereas a codec developed in a fixed hardware (like ASIC/ASSP) is fixed and it cannot be changed.

Memory size is another important hardware software trade-off. Evaluate how much memory is required if the system requirement under consideration is implemented in software (firmware). Embedded systems are highly memory constrained and embedded designers don't have the luxury of using lavish

---

[†] Eclipse is an open source Integrated Development Environment (IDE). Visit www.eclipse.org for more details.

[‡] Multimedia codec is a compression and de-compression algorithm for compressing and de-compression of raw data media files (audio and video data)

memory for implementing requirements. On the other hand, evaluate the gate count required (Normally hardware chips are implemented using logic gates and the density of the chip is expressed in terms of the number of gates used in the design (millions of gates ☺)), if the required feature is going to implement in hardware.

Effort required in terms of man hours, if the required feature is going to build in either software or custom hardware implementation using VHDL or any other hardware description languages and the cost for each are another important hardware-software trade-off in any embedded system development.

To summarise, the important hardware-software trade-offs in embedded system design are

1. Processing speed and performance
2. Frequency of change (Re-configurability)
3. Memory size and gate count
4. Reliability
5. Man hours (Effort) and cost

## Summary

- ✓ Model selection, Architecture selection, Language selection, Hardware Software partitioning, etc. are some of the main points in the hardware-software co-design
- ✓ A model captures and describes the system characteristics. 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
- ✓ 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
- ✓ A programming language captures a 'Computational Model' and maps it into architecture. A model can be captured using multiple programming languages like C, C++, C#, Java, etc. for software implementations and languages like VHDL, System C, Verilog, etc. for hardware implementations
- ✓ The Hierarchical/Concurrent Finite State Machine Model (HCFSM) is an extension of the FSM for supporting concurrency and hierarchy. HCFSM extends the conventional state diagrams by the AND, OR decomposition of States together with inter level transitions and a broadcast mechanism for communicating between concurrent processes
- ✓ HCFSM uses statecharts for capturing the states, transitions, events and actions. The Harel Statechart, UML State diagram, etc. are examples for popular statecharts for the HCFSM modelling of embedded systems
- ✓ In the sequential model, the functions or processing requirements are executed in sequence. It is same as the conventional procedural programming
- ✓ The concurrent or communicating process model models concurrently executing tasks/processes
- ✓ The object oriented model is an object based model for modelling system requirements
- ✓ Unified Modelling Language (UML) is a visual modelling language for Object Oriented Design (OOD). Things, Relationships and Diagrams are the fundamental building blocks of UML
- ✓ UML diagrams give a pictorial representation of the static aspects, behavioural aspects and organisation and management of different modules. Static Diagrams, Object diagram, Usecase diagram, Sequence diagram, Statechart diagram etc. are the different UML diagrams
- ✓ Hardware-Software trade-offs are the parameters used in the decision making of partitioning a system requirement into either hardware or software. Processing speed, performance, frequency of changes, memory size and gate count requirements, reliability, effort, cost, etc. are examples for hardware-software trade-offs

## Keywords

| | | |
|---|---|---|
| **Hardware-Software Co-design** | : | The modern approach for the interactive 'together' design of hardware and firmware for embedded systems |
| **Controller Architecture** | : | Architecture implementing the Finite State Machine model using a state register and two combinational circuits |
| **Datapath Architecture Datapath** | : | Architecture implementing the Data Flow Graph model |
| | : | A channel between the input and output. The datapath may contain registers, counters, register files, memories and ports along with high speed arithmetic units |
| **Finite State Machine Datapath (FSMD) Architecture** | : | Architecture combining the controller architecture with datapath architecture |
| **Complex Instruction Set Computing (CISC) Architecture** | : | Architecture which uses instruction set representing complex operations |
| **Reduced Instruction Set Computing (RISC) Architecture** | : | Architecture which uses instruction set representing simple operations |
| **Very Long Instruction Word (VLIW) Architecture** | : | Architecture implementing multiple functional units (ALUs, multipliers, etc.) in the datapath |
| **Parallel Processing Architecture** | : | Architecture implementing multiple concurrent Processing Elements (PEs) |
| **Single Instruction Multiple Data (SIMD) Architecture** | : | Architecture in which a single instruction is executed in parallel with the help of the processing elements |
| **MIMD Architecture** | : | Architecture in which the processing elements execute different instructions at a given point of time |
| **Programming Language** | : | An entity for capturing a 'Computational Model' and maps it into architecture |
| **Data Flow Graph (DFG) Model** | : | A data driven model in which the program execution is determined by data |
| **Control DFG (CDFG) Model** | : | A model containing both data operations and control operations. The CDFG uses Data Flow Graph (DFG) as element and conditional (constructs) as decision makers |
| **State Machine Model** | : | A model describing the system behaviour with 'States', 'Events', 'Actions' and 'Transitions' |
| **Statechart** | : | An entity for capturing the states, transitions, events and actions |
| **Harel Statechart** | : | A type of statechart |
| **Finite State Machine (FSM) Model** | : | A state machine model with finite number of states |
| **Hierarchical/Concurrent Finite State Machine Model (HCFSM)** | : | An extension of the FSM for supporting concurrency and hierarchy |
| **Sequential Model** | : | Model for capturing sequential processing requirements |
| **Concurrent or Communicating Process Model** | : | Model for capturing concurrently executing tasks/process requirements |
| **Object Oriented Model** | : | Object based model for modelling system requirements |
| **Unified Modelling Language (UML)** | : | A visual modelling language for Object Oriented Design |
| **UML Diagram** | : | Diagram giving a pictorial representation of the static aspects, behavioural aspects and organisation and management of different modules |

| | |
|---|---|
| **Static Diagram** | : UML diagram representing the static (structural) aspects of the system |
| **Behavioural Diagram** | : UML diagram representing the dynamic (behavioural) aspects of the system |
| **Object Diagram** | : Static UML diagram giving the pictorial representation of objects and their relationships |
| **Class Diagram** | : Static UML diagram showing the pictorial representation of the different classes in a UML model, their interfaces, the collaborations, interactions and relationship between classes, etc. |
| **Usecase Diagram** | : Behavioural UML diagram for capturing system functionality as seen by users |
| **Sequence Diagram** | : Behavioural UML diagram representing object interactions with respect to time |
| **Statechart Diagram** | : Behavioural UML diagram showing the states, transitions, events and activities similar to a state machine representation |
| **Hardware-Software trade-offs** | : The parameters used in the decision making of partitioning a system requirement into either hardware or software |

## Objective Questions

1. Which of the following programming model is best suited for modelling a data driven embedded system
   (a) State Machine     (b) Data Flow Graph     (c) Harel Statechart Model
   (d) None of these

2. Which of the following programming model is best suited for modelling a Digital Signal Processing (DSP) embedded system
   (a) Finite State Machine     (b) Data Flow Graph     (c) Object Oriented Model
   (d) UML

3. Which of the following architecture is best suited for implementing a Digital Signal Processing (DSP) embedded system
   (a) Controller Architecture     (b) CISC     (c) Datapath Architecture     (d) None of these

4. Which of the following is a multiprocessor architecture?
   (a) SIMD     (b) MIMD     (c) VLIW     (d) All
   (e) (a) and (b)     (f) (b) and (c)

5. Which of the following model is best suited for modelling a reactive embedded system?
   (a) Finite State Machine (FSM)     (b) DFG     (c) Control DFG
   (d) Object Oriented Model

6. Which of the following models is best suited for modelling a reactive real time embedded system?
   (a) Finite State Machine     (b) DFG     (c) Control DFG
   (d) Hierarchical/Concurrent Finite State Machine Model

7. Which of the following model is best suited for modelling an embedded system demanding multitasking capabilities with data sharing?
   (a) Finite State Machine     (b) DFG     (c) Control DFG
   (d) Communicating Process Model

8. Which of the following is a hardware description language?
   (a) C     (b) System C     (c) VHDL     (d) C++
   (e) (b) and (c)

9. Which of the following is a structural thing in UML?
   (a) Class          (b) Interaction          (c) State Machine          (d) Activity
   (e) None of these

10. Which of the following is a behavioural thing in UML?
    (a) Class          (b) Interface          (c) State Machine          (d) Component
    (e) None of these

11. Which of the following is not a static diagram in UML?
    (a) Class Diagram          (b) Object Diagram          (c) Use case Diagram          (d) Component Diagram
    (e) None of these

12. Which of the following is not a behavioural diagram in UML?
    (a) State Diagram          (b) Object Diagram          (c) Use case Diagram          (d) Sequence Diagram
    (e) None of these

13. Which of the following UML diagrams is best suited for Requirements Capturing?
    (a) State Diagram          (b) Use case Diagram          (c) Object Diagram          (d) Sequence Diagram
    (e) None of these

14. Which of the following UML diagram represents object interactions with respect to time?
    (a) State Diagram          (b) Sequence Diagram          (c) Object Diagram          (d) Use case Diagram
    (e) None of these

15. Which of the following UML interaction diagram(s) emphasises on structural organisation of objects?
    (a) Collaboration Diagram          (b) Sequence Diagram          (c) State Diagram          (d) Use case Diagram
    (e) None of these

16. Which of the following is (are) trade-offs in hardware software partitioning?
    (a) Processing speed          (b) Memory requirement          (c) Cost          (d) All of these
    (e) None of these

## Review Questions

1. What is hardware software co-design? Explain the fundamental issues in hardware software co-design
2. Explain the difference between SIMD, MIMD and VLIW architecture
3. What is Computational model? Explain its role in hardware software co-design
4. Explain the different computational models in embedded system design
5. What is the difference between Data Flow Graph (DFG) and Control Data Flow Graph (CDFG) model? Explain their significance in embedded system design
6. What is State and State Machine? Explain the role of State Machine in embedded system design
7. What is the difference between Finite State Machine Model (FSM) and Hierarchical/Concurrent Finite State Machine Model (HCFSM)?
8. What is 'Statechart'? Explain its role in embedded system design
9. Explain the 'Sequential' Program model with an example
10. Explain the 'Concurrent/Communicating' program model. Explain its role in 'Real Time' system design
11. Explain the 'Object-Oriented' program model for embedded system design. Under which circumstances an Object-Oriented model is considered as the best suited model for embedded system design?
12. Explain the role of programming languages in system design
13. What are the building blocks of UML? Explain in detail.
14. Explain the different types of UML diagrams and their significance in each stage of the system development life cycle
15. Explain the important hardware software 'trade-offs' in hardware software partitioning?

## Lab Assignments

1. Draw the Data Flow Graph (DFG) model for the FIR filter implementation $y[n] = a0 \times [n] + a1 \times [n-1] + a2 \times [n-2] + a3 \times [n-3] + \ldots + a(n-1) \times [1]$

2. Draw the sequence diagram for the automatic coffee vending machine using UML.

3. Model the following automatic teller machine (ATM) requirement using UML statecharts.

   (a) The transaction is started by user inserting a valid ATM card.

   (b) Upon detecting the card, the system prompts for a Personal Identification Number (PIN).

   (c) The card is validated against the PIN and on successful validation, the system displays the following options. 'Balance Enquiry', 'Cash Withdrawal' and 'Cancel'.

   (d) If the user selects 'Cash Withdrawal', the system prompts for entering the amount to withdraw and upon entering the amount it asks whether a printed receipt is required for the transaction. Upon receiving an input for this, the system verifies the available balance, deduct the amount to withdraw, dispense the cash, prints the transaction receipt (if the user answer to the system query 'whether a printed receipt is required for the transaction' is 'YES') and returns back to the original state to accept a new transaction request.

   (e) If the user selects the option 'Balance Enquiry' in step 'c', the system prompts the message whether a printed receipt is required for the transaction. If user response is 'YES', the balance is printed and the system returns to the original state to accept new transaction request. If the user input is 'NO' to the query 'whether a printed receipt is required for the transaction', the balance amount is displayed on the screen for 30 seconds and the system returns back to the original state to accept a new transaction request.

   (f) If the user selects the option 'Cancel' in step 'c', the system returns back to the original state to accept a new transaction request.

   (g) If there is no response from the user for a period of 30 seconds in steps 'b', 'c', 'd' and 'e' and if the time interval between the two consecutive digit entering for the PIN exceeds 30 seconds, the system displays the message 'User Failed to Respond within the specified Time' for 10 seconds and returns back to the original state to accept a new transaction request.

# 9 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. Imparting intelligence to an Embedded system is a one time process and it can happen at any stage, it can be immediately after the fabrication of the embedded hardware or at a later stage. Once intelligence is imparted to the embedded product, by embedding the firmware in the hardware, the product starts functioning properly and will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware occurs. In case of hardware breakdown, the damaged component may need to be replaced by a new component and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning. Coming back to the newborn baby example, the newborn baby is very adaptive in terms of intelligence, meaning it learns from mistakes and updates its memory each time a mistake or a deviation in expected behaviour occurs, whereas most of the embedded systems are less adaptive or non-adaptive. For most of the embedded products the embedded firmware is stored at a permanent memory (ROM) and they are nonalterable by end users. Some of the embedded products used in the Control and Instrumentation domain are adaptive. This adaptability is achieved by making use configurable parameters which are stored in the alterable permanent memory area (like NVRAM/FLASH). The parameters get updated in accordance with the deviations from expected behaviour and the firmware makes use of these parameters for creating the response next time for similar variations.

Designing embedded firmware requires understanding of the particular embedded product hardware, like various component interfacing, memory map details, I/O port details, configuration and register details of various hardware chips used and some programming language (either target processor/controller specific low level assembly language or a high level language like C/C++/JAVA).

Embedded firmware development process starts with the conversion of the firmware requirements into a program model using modelling tools 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 (Fig. 9.1). Once the program model is created, the next step is the implementation of the tasks and actions by capturing the model using a language which is understandable by the target processor/controller. The following sections are designed to give an overview of the various steps involved in the embedded firmware design and development.

## 9.1    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*' and '*Embedded Operating System (OS) Based Design*'. The conventional procedural based design is also known as '*Super Loop Model*'. We will discuss each of them in detail in the following sections.

### 9.1.1   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 (embedded systems where missing deadlines are acceptable). It is very similar to a conventional procedural programming where the code is executed task by task. The 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 initialisation 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 visualise 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 ();
}
}
```

Almost all tasks in embedded applications are non-ending and are repeated infinitely throughout the operation. 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. Here the while (1) { } 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. Whereas 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.

This type of design is deployed in low-cost embedded products and products where response time is not time critical. Some embedded products demands this type of approach if some tasks itself are sequential. 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. it

should strictly follow a specified sequence and the combination of these series of tasks constitutes a single task-namely data read/write. There is no use in putting the sub-tasks into independent tasks and running them parallel. It won't work at all.

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. The keyboard scanning and display updating happens at a reasonably high rate. 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 ☺. It is not economical to embed an OS into low cost products and it is an utter waste to do so if the response requirements are not crucial.

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 single task will affect the total system. If the program hangs up at some point while executing a task, it will remain there forever and ultimately the product stops functioning. There are remedial measures for overcoming this. Use of Hardware and software Watch Dog Timers (WDTs) helps in coming out from the loop when an unexpected failure occurs or when the processor hangs up. This, in turn, may cause additional hardware cost and firmware overheads.

Another major drawback of the *'Super loop'* 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 (That is key pressing event may not be in sync with the keypad press monitoring task within the firmware). 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. Advances in processor technology brings out low cost high speed processors/controllers, use of such processors in super loop design greatly reduces the time required to service different tasks and thereby are capable of providing a nearly real time attention to external events.

Throughout this book under the title *'Embedded Firmware Design and Development'*, we will be discussing only the *'Super loop based design'*. Again the discussion is narrowed to super loop based firmware development for 8051 controller.

## 9.1.2 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. Examples of Embedded products using Microsoft® Windows XP OS are Personal Digital Assistants (PDAs), Hand held devices/Portable devices and Point of Sale (PoS) terminals. Use of GPOS in embedded products merges the demarcation of Embedded Systems and general computing systems in terms of OS. For Developing applications on top of the OS, the OS supported APIs are used. Similar to the

different hardware specific drivers, OS based applications also require *'Driver software'* for different hardware present on the board to communicate with them.

Real Time Operating System (RTOS) based design approach 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. We will discuss the basics of RTOS based system design in a later chapter titled *'Designing with Real Time Operating Systems (RTOS)'*.

*'Windows CE'*, *'pSOS'*, *'VxWorks'*, *'ThreadX'*, *'MicroC/OS-II'*, *'Embedded Linux'*, *'Symbian'* etc are examples of RTOS employed in embedded product development. Mobile phones, PDAs (Based on Windows CE/Windows Mobile Platforms), handheld devices, etc. are examples of 'Embedded Products' based on RTOS. Most of the mobile phones are built around the popular RTOS 'Symbian'.

## 9.2  EMBEDDED FIRMWARE DEVELOPMENT LANGUAGES

As mentioned in Chapter 2, you can use either a target processor/controller specific language (Generally known as Assembly language or low level language) or a target processor/controller independent language (Like C, C++, JAVA, etc. commonly known as High Level Language) or a combination of Assembly and High level Language. We will discuss where each of the approach is used and the relative merits and de-merits of each, in the following sections.

### 9.2.1  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 (1s and 0s). Machine language is a binary representation and it consists of 1s and 0s. 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.**

Assembly Language program was the most common type of programming adopted in the beginning of software revolution. If we look back to the history of programming, we can see that a large number of programs were written entirely in assembly language. Even in the 1990s, the majority of console video games were written in assembly language, including most popular games written for the Sega Genesis and the Super Nintendo Entertainment System. The popular arcade game NBA Jam released in 1993 was also coded entirely using the assembly language.

Even today also almost all low level, system related, programming is carried out using assembly language. Some Operating System dependent tasks require low-level languages. In particular, assembly language is often used in writing the low level interaction between the operating system and the hardware, for instance in device drivers.

The general format of an assembly language instruction is an Opcode followed by Operands. The Opcode tells the processor/controller what to do and the Operands provide the data and information

required to perform the action specified by the opcode. It is not necessary that all opcode should have Operands following them. Some of the Opcode implicitly contains the operand and in such situation no operand is required. The operand may be a single operand, dual operand or more. We will analyse each of them with the *8051* ASM instructions as an 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 (single operand). The same instruction when written in machine language will look like

<center>01110100     00011110</center>

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. This instruction increments the *8051* Accumulator register content by 1.

The mnemonic *MOV A, #30* explained above is an example for single operand instruction.

*LJMP 16bit address* is an example for dual operand instruction. The machine language for the same is

```
00000010     addr_bit15 to addr_bit 8     addr_bit7 to addr_bit 0
```

The first binary data is the representation of the LJMP machine code. The first operand that immediately follows the opcode represents the bits 8 to 15 of the 16bit address to which the jump is required and the second operand represents the bits 0 to 7 of the address to which the jump is targeted.

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 + Operand). Each line of an assembly language program is split into four fields as given below

<center>**LABEL    OPCODE    OPERAND    COMMENTS**</center>

LABEL is an optional field. A 'LABEL' is an identifier used extensively in programs to reduce the reliance on programmers for remembering where data or code is located. LABEL is commonly used for representing

- A memory location, address of a program, sub-routine, code portion, etc.
- The maximum length of a label differs between assemblers. Assemblers insist strict formats for labelling. 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).

Labels are used for representing subroutine names and jump locations in Assembly language programming. It is to be noted that 'LABEL' is not a mandatory field; it is optional only.

The sample code given below using *8051* Assembly language illustrates the structured assembly language programming.

```
;############################################################################
;     SUBROUTINE FOR GENERATING DELAY
;     DELAY PARAMETR PASSED THROUGH REGISTER R1
;     RETURN VALUE NONE
;     REGISTERS USED: R0, R1
;############################################################################
```

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

The Assembly program contains a main routine which starts at address 0000H and it may or may not contain subroutines. The example given above is a subroutine, where in the main program the subroutine is invoked by the Assembly instruction

<div align="center">LCALL        DELAY</div>

Executing this instruction transfers the program flow to the memory address referenced by the 'LABEL' DELAY.

It is a good practice to provide comments to your subroutines before the beginning of it by indicating the purpose of that subroutine, what the input parameters are and how they are passed to the subroutines, which are the return values, how they are returned to the calling function, etc. While assembling the code a ';' informs the assembler that the rest of the part coming in a line after the ';' symbol is comments and simply ignore it. Each Assembly instruction should be written in a separate line. Unlike C and other high level languages, more than one ASM code lines are not allowed in a single line.

In the above example the *LABEL* DELAY represents the reference to the start of the subroutine DELAY. You can directly replace this LABEL by putting the desired address first and then writing the Assembly code for the routine as given below.

```
ORG 0100H
    MOV R0, #255        ; Load Register R0 with 50H
    DJNZ R1, 0100H      ; Decrement R1 and loop till R1= 0
    RET                 ; Return to calling program
```

The advantage of using a label is that the required address is calculated by the assembler at the time of assembling the program and it replaces the Label. Hence even if you add some code above the LABEL 'DELAY' at a later stage, it won't create any issues like code overlapping, whereas in the second method where you are implicitly telling the assembler that this subroutine should start at the specified address (in the above example 0100H). If the code written above this subroutine itself is crossing the 0100H mark of the program memory, it will be over written by the subroutine code and it will generate unexpected results☺. Hence for safety don't assign any address by yourself, let us refer the required address by using labels and let the assembler handle the responsibility for finding out the address where the code can be placed. In the above example you can find out that the label DELAY is used for calling the subroutine as well as looping (using jumping instruction based on decision-DJNZ). You can also use the normal jump instruction to jump to the label by calling *LJMP DELAY*

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. var EQU 70H)
4. Initialising variable values (e.g. val DATA 12H)

The *EQU* directive is used for allocating memory to a variable and *DATA* directive is used for initialising a variable with data. No machine codes are generated for the 'pseudo-ops'.

Till now we discussed about Assembly language and how it is used for writing programs. Now let us have a look at how assembly programs are organised and how they are translated into machine readable codes.

The Assembly language program written in assembly code is saved as *.asm* (Assembly file) file or an *.src* (source) file. Any text editor like 'notepad' or 'WordPad' from Microsoft® or the text editor provided by an Integrated Development (IDE) tool can be used for writing the assembly instructions.

Similar to 'C' and other high level language programming, you can have multiple source files called modules in assembly language programming. Each module is represented by an '*.asm*' or '*.src*' file similar to the '*.c*' files in C programming. This approach is known as 'Modular Programming'. Modular programming is employed when the program is too complex or too big. In 'Modular Programming', the entire code is divided into submodules and each module is made re-usable. Modular Programs are usually easy to code, debug and alter. Conversion of the assembly language to machine language is carried out by a sequence of operations, as illustrated below.

*9.2.1.1 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 and it is common that assemblers from multiple vendors are available in the market for the same target machines. Some target processor's/controller's assembler may be proprietary and is supplied by a single vendor only. Some assemblers are freely available in the internet for downloading. Some assemblers are commercial and requires licence from the vendor. 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 Fig. 9.1.
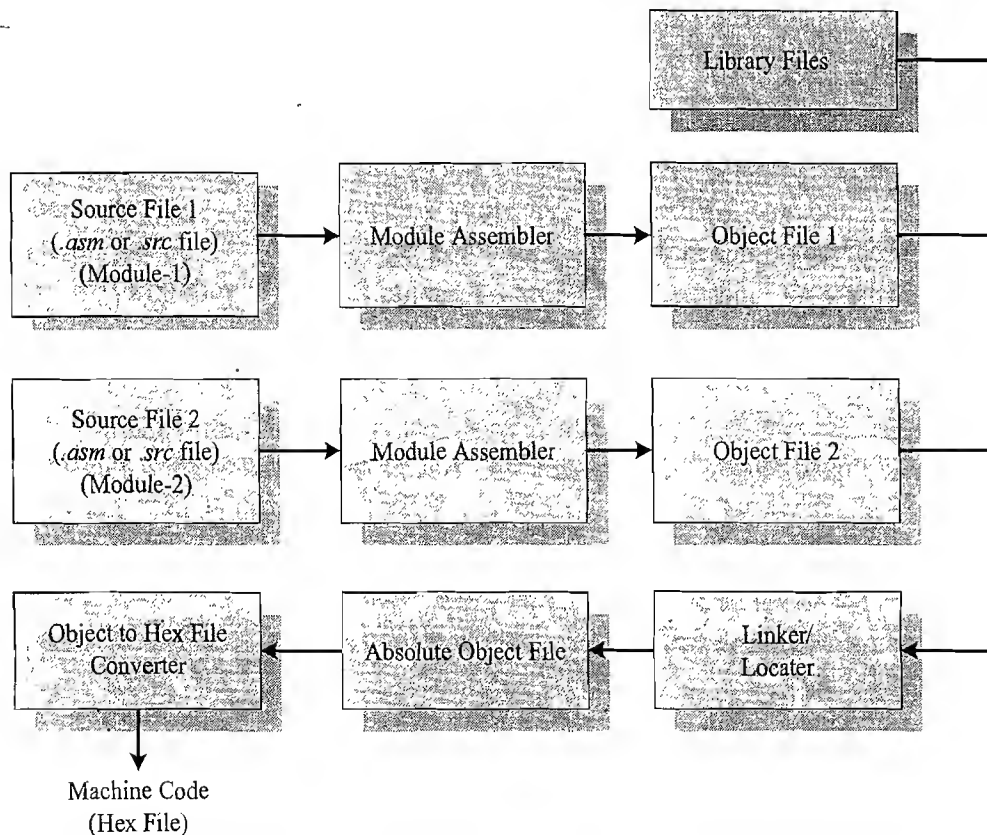


**Fig. 9.1** Assembly language to machine language conversion process

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 a 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 *EXTRN* (EXTERN) in the module where it is going to be accessed. The '*PUBLIC*' Keyword informs the assembler that the variables or functions declared as '*PUBLIC*' needs to be exported. Similarly the '*EXTRN*' Keyword tells the assembler that the variables or functions declared as '*EXTRN*' needs to be imported from some other modules. While assembling a module, on seeing variables/functions with keyword '*EXTRN*', the assembler understands that these variables or functions come from an external module and it proceeds assembling the entire module without throwing any errors, though the assembler cannot find the definition of the variables and implementation of the functions. Corresponding to a variable or function declared as '*PUBLIC*' in a module, there can be one or more modules using these variables or functions using '*EXTRN*' keyword. For all those modules using variables or functions with '*EXTRN*' keyword, there should be one and only one module which exports those variables or functions with '*PUBLIC*' keyword. If more than one module in a project tries to export variables or functions with the same name using '*PUBLIC*' keyword, it will generate 'linker' errors.

Illustrative example for A51 Assembler–Usage of '*PUBLIC*' for importing variables with same name on different modules. The target application (Simulator) contains three modules namely *ASAMPLE1. A51*, *ASAMPLE2.A51* and *ASAMPLE3.A51* (The file extension . *A51* is the .asm extension specific to A51 assembler). The modules *ASAMPLE2.A51* and *ASAMPLE3.A51* contain a function named *PUTCHAR*. Both of these modules try to export this function by declaring the function as '*PUBLIC*' in the respective modules. While linking the modules, the linker identifies that two modules are exporting the function with name *PUTCHAR*. This confuses the linker and it throws the error '*MULTIPLE PUB-LIC DEFINITIONS*'.

```
Build target 'Simulator'
assembling ASAMPLE1.A51...
assembling ASAMPLE2.A51...
assembling ASAMPLE3.A51...
linking...
*** ERROR L104: MULTIPLE PUBLIC DEFINITIONS
 SYMBOL:  PUTCHAR
 MODULE:  ASAMPLE3.obj (CHAR_IO)
```

If a variable or function declared as '*EXTRN*' in one or two modules, there should be one module defining these variables or functions and exporting them using '*PUBLIC*' keyword. If no modules in a project export the variables or functions which are declared as '*EXTRN*' in other modules, it will generate 'linker' warnings or errors depending on the error level/warning level settings of the linker.

Illustrative example for A51 Assembler–Usage of *EXTRN* without variables exported. The target application (Simulator) contains three modules, namely, *ASAMPLE1.A51*, *ASAMPLE2.A51* and *ASAMPLE3.A51* (The file extension .A51 is the .asm extension specific to A51 assembler). The modules *ASAMPLE1.A51* imports a function named *PUT_CRLF* which is declared as '*EXTRN*' in the current module and it expects any of the other two modules to export it using the keyword '*PUBLIC*'. But none of the other modules export this function by declaring the function as '*PUBLIC*' in the respective modules. While linking the modules, the linker identifies that there is no function exporting for this function. The linker generates a warning or error message '*UNRESOLVED EXTERNAL SYMBOL*' depending on the linker 'level' settings.

```
*** WARNING L1: UNRESOLVED EXTERNAL SYMBOL
    SYMBOL:   PUT_CRLF
    MODULE:   ASAMPLE1.obj (SAMPLE)
```

**9.2.1.2 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 file 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.

If you are using a commercial version of the assembler/compiler suite for your development, the vendor of the utility may provide you pre-written library files for performing multiplication, floating point arithmetic, etc. as an add-on utility or as a bonus☺.

'*LIB51*' from Keil 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.

**9.2.1.3 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 generates an absolute object module by extracting the object modules from the library, if any and those *obj* files created by the assembler, which is generated by assembling the individual modules of a project. It is the responsibility of the linker to link any external dependent variables or functions declared on various modules and resolve the external dependencies among the modules. 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.

**9.2.1.4 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 representa-

tion 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.

'*OH51*' from Keil software is an example for Object to Hex File Converter utility for A51 Assembler/ C51 Compiler for *8051* specific controller.

### 9.2.1.5 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 organisation, register sets and mnemonics is very essential for Assembly Language based development. If you master one processor architecture and its assembly instructions, you can make the processor as flexible as a gymnast. The major advantages of Assembly Language based development is listed below.

**Efficient Code Memory and Data Memory Usage (Memory Optimisation)** Since the developer is well versed with the target processor architecture and memory organisation, optimised code can be written for performing operations. This leads to less utilisation of code memory and efficient utilisation of data memory. Remember memory is a primary concern in any embedded product (Though silicon is cheaper and new memory techniques make memory less costly, external memory operations impact directly on system performance).

**High Performance** Optimised 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 'hawkers' to reveal the technology behind 'Proprietary Products'. Though most of the products employ code memory protection, if it may be possible to break the memory protection and read the code memory, it can easily be converted into assembly code using a dis-assembler program for the target machine.

### 9.2.1.6 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 organisation 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. One probable solution for this is use a readily available developer who is well versed in

the target processor architecture assembly instructions. Also more lines of assembly code are required for performing an action which can be done with a single instruction in a high-level language like 'C'.

***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 analyse this code, he/she also may not be able to understand what is done and why it is done. Hence upgrading an assembly program or modifying it on a later stage is very difficult. Well documenting the assembly code is a solution for reducing the developer dependency in assembly language programming. If the code is too large and complex, documenting all lines of code may not be productive.

***Non-Portable*** Target applications written in assembly instructions are valid only for that particular family of processors (e.g. Application written for Intel x86 family of processors) and cannot be re-used for another target processors/controllers (Say ARM11 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. This is the major drawback of assembly language programming and it makes the assembly language applications non-portable.

*"Though Assembly Language programming possesses lots of drawback, as a developer, from my personal experience I prefer assembly language based development. Once you master the internals of a processor/controller, you can really perform magic with the processor/controller and can extract the maximum out of it."*

## 9.2.2 High Level Language Based Development

As we have seen in the earlier section, 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. Here comes the role of high level languages. 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 – We will discuss cross-compilers in detail in a later section) 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'. You may be thinking why 'C' is used as the popular embedded firmware development language. The answer is "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 Fig. 9.2.

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
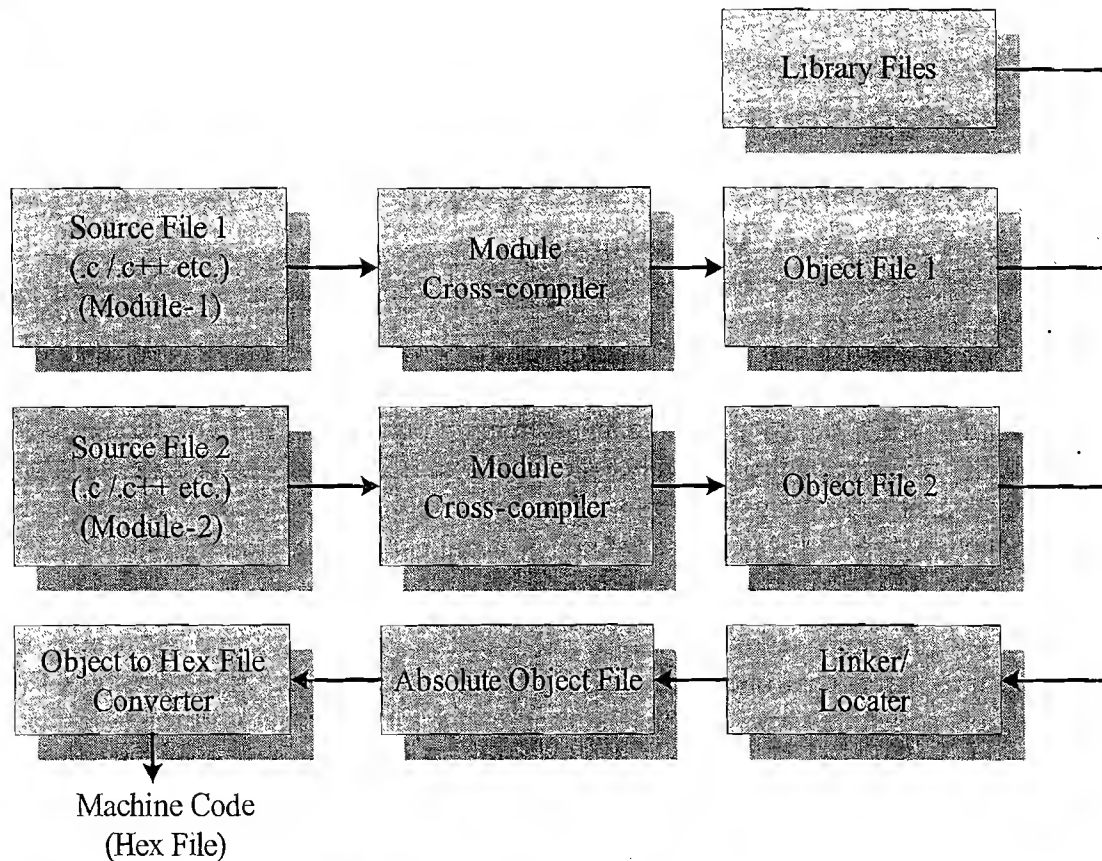
**Fig. 9.2** **High level language to machine language conversion process**

text editor provided by an Integrated Development (IDE) tool supporting the high level language in use 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. The source files corresponding to each module is represented by a file with corresponding language extension. 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. It should be noted that each high level language should have a cross-compiler for converting the high level source code into the target processor machine code. Without cross-compiler support a high level language cannot be used for embedded firmware development. C51 Cross-compiler from Keil software is an example for Cross-compiler. 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.

As an example of high level language based embedded firmware development, we will discuss how 'Embedded C' is used for embedded firmware development, in a later section of this chapter.

### 9.2.2.1 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 organisation 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. Rest of the 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 specialised 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. Certain instructions may require little knowledge of the target hardware details like register set, memory map etc. Apart from these, the high level language based firmware development makes the firmware, developer independent. High level languages always instruct certain set of rules for writing the code and commenting the piece of code. If 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 a 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 target processor/controller, provided, the cross-compiler has support for the processor/controller selected. This makes applications written in high level language highly portable. Little effort may be required in the existing code to replace the target processor specific header files with new header files, register definitions with new ones, etc. This is the major flexibility offered by high level language based design.

*9.2.2.2 Limitations of High Level Language Based Development*    The merits offered by high level language based design take advantage over its limitations. Some cross-compilers available for high level languages may not be so efficient in generating optimised target processor specific instructions. Target images created by such compilers may be messy and non-optimised in terms of performance as well as code size. For example, the task achieved by cross-compiler generated machine instructions from a high level language may be achieved through a lesser number of instructions if the same task is hand coded using target processor specific machine codes. The time required to execute a task also increases with the number of instructions. However modern cross-compilers are tending to adopt designs incorporating optimisation techniques for both code size and performance. High level language based code snippets may not be efficient in accessing low level hardware where hardware access timing is critical (of the order of nano or micro seconds).

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.

## 9.2.3    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.

*9.2.3.1 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 in use 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 optimised 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 are returned from assembly routine to 'C' and how 'Assembly routine' is invoked from the 'C' code.

Passing parameter to the assembly routine and returning values from the assembly routine to the caller 'C' function and the method of invoking the assembly routine from 'C' code is cross compiler dependent. There is no universal written rule for this. You must get these informations from the documentation of the cross compiler you are using. Different cross compilers implement these features in different ways depending on the general purpose registers and the memory supported by the target processor/controller. Let's examine this by taking Keil C51 cross compiler for 8051 controller. The objective of this example is to give an idea on 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.

As an example consider the following sample code (Extracted from Keil C51 documentation)

```
#pragma SRC
unsigned char my_assembly_func (unsigned int argument)
{
return (argument + 1);  // Insert dummy lines to access all args and
                        // retvals
}
```

This C function on cross compilation generates the following assembly *SRC* file.

```
NAME     TESTCODE
?PR?_my_assembly_func?TESTCODE          SEGMENT CODE
         PUBLIC   _my_assembly_func
; #pragma SRC
; unsigned char my_assembly_func (
```

```
        RSEG  ?PR?_my_assembly_func?TESTCODE
        USING  0
my_assembly_func:
;---- Variable 'argument?040' assigned to Register 'R6/R7' ----
; SOURCE LINE # 2
;  unsigned int argument)
; {
; SOURCE LINE # 4
; return (argument + 1);   // Insert dummy lines to access all args
; and retvals
; SOURCE LINE # 5
        MOV    A,R7
        INC    A
        MOV    R7,A
; }
; SOURCE LINE # 6
?C0001:
        RET
; END OF _my_assembly_func
        END
```

The special compiler directive *SRC* generates the Assembly code corresponding to the 'C' function and each lines of the source code is converted to the corresponding Assembly instruction. You can easily identify the Assembly code generated for each line of the source code since it is implicitly mentioned in the generated *.SRC* file. By inspecting this code segments you can find out which registers are used for holding the variables of the 'C' function and you can modify the source code by adding the assembly routine you want.

### 9.2.3.2 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 optimised code, optimal performance, efficient code memory utilisation 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 16bit 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'.

Most often the functions written in 'C' use parameter passing to the function and returns value/s to the calling functions. The major question that needs to be addressed in mixing a 'C' function with Assembly is that how the parameters are passed to the function and how values are returned from the function and how the function is invoked from the assembly language environment. Parameters are passed to the function and values are returned from the function using CPU registers, stack memory and fixed memory. Its implementation is cross compiler dependent and it varies across cross compilers. A typical example is given below for the Keil C51 cross compiler

C51 allows passing of a maximum of three arguments through general purpose registers R2 to R7. If the three arguments are *char* variables, they are passed to the function using registers R7, R6 and R5

respectively. If the parameters are *int* values, they are passed using register pairs (R7, R6), (R5, R4) and (R3, R2). If the number of arguments is greater than three, the first three arguments are passed through registers and rest is passed through fixed memory locations. Refer to C51 documentation for more details. Return values are usually passed through general purpose registers. R7 is used for returning *char* value and register pair (R7, R6) is used for returning *int* value. The 'C' subroutine can be invoked from the assembly program using the subroutine call Assembly instruction (Again cross compiler dependent).

```
E.g.  LCALL      _Cfunction
```

Where *Cfunction* is a function written in 'C'. The prefix _ informs the cross compiler that the parameters to the function are passed through registers. If the function is invoked without the _ prefix, it is understood that the parameters are passed through fixed memory locations.

*9.2.3.3 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 (If the Assembly instructions to be inserted are put in a subroutine as mentioned in the section mixing assembly with 'C'). Special keywords are used to indicate that the start and end of Assembly instructions. The keywords are cross-compiler specific. C51 uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

```
E.g.   #pragma asm
        MOV A, #13H
       #pragma endasm
```

**Important Note:**
*The examples used for illustration throughout the section **Mixing Assembly & High Level Language** is Keil C51 cross compiler specific. The operation is cross compiler dependent and it varies from cross compiler to cross compiler. The intention of the author is just to give an overall idea about the mixing of Assembly code and High level language 'C' in writing embedded programs. Readers are advised to go through the documentation of the cross compiler they are using for understanding the procedure adopted for the cross compiler in use.*

## 9.3 PROGRAMMING IN EMBEDDED C

Whenever the conventional 'C' Language and its extensions are used for programming embedded systems, it is referred as '*Embedded C*' programming. Programming in 'Embedded C' is quite different from conventional Desktop application development using 'C' language for a particular OS platform. Desktop computers contain working memory in the range of Megabytes (Nowadays Giga bytes) and storage memory in the range of Giga bytes. For a desktop application developer, the resources available are surplus in quantity and s/he can be very lavish in the usage of RAM and ROM and no restrictions are imposed at all. This is not the case for embedded application developers. Almost all embedded systems are limited in both storage and working memory resources. Embedded application developers should be aware of this fact and should develop applications in the best possible way which optimises the code memory and working memory usage as well as performance. In other words, the hands of an embedded application developer are always tied up in the memory usage context☺.

### 9.3.1 'C' v/s. 'Embedded C'

'C' is a well structured, well defined and standardised 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 the supported cross-compiler for the particular Embedded 'C' language. A software program called 'Cross-compiler' is used for the conversion of programs written in Embedded 'C' to target processor/controller specific instructions (machine language).

### 9.3.2 Compiler vs. Cross-Compiler

Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (e.g. Intel x86/Pentium). Here the operating system, the compiler program and the application making use of the source code run on the same target processor. The source code is converted to the target processor specific machine instructions. 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 the software tools used in cross-platform development applications. In cross-platform development, the compiler running on a particular target processor/OS converts the source code to machine code for a target processor whose architecture and instruction set is different from the processor on which the compiler is running or for an operating system which is different from the current development environment OS. Embedded system development is a typical example for cross-platform development where embedded firmware is developed on a machine with Intel/AMD or any other target processors and the same is converted into machine code for any other target processor architecture (e.g. 8051, PIC, ARM etc). Keil C51 is an example for cross-compiler. 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.

### 9.3.3 Using 'C' in 'Embedded C'

The author takes the privilege of assuming the readers are familiar with 'C' programming. Teaching 'C' is not in the scope of this book. If you are not familiar with 'C' language syntax and 'C' programming technique, please get a handle on the same before you proceed. Readers are advised to go through books by 'Brian W. Kernighan and Dennis M. Ritchie (K&R)' or 'E. Balagurusamy' on 'C' programming.

This section is intended only for giving readers a basic idea on how 'C' Language is used in embedded firmware development.

Let us brush up whatever we learned in conventional 'C' programming. Remember we will only go through the peripheral aspects and will not go in deep.

*9.3.3.1 Keywords and Identifiers* *Keywords* are the reserved names used by the 'C' language. All *keywords* have a fixed meaning in the 'C' language context and they are not allowed for programmers for naming their own variables or functions. ANSI 'C' supports 32 keywords and they are listed below. All 'C' supported keywords should be written in '*lowercase*' letters.

| auto | Double | int | struct |
|---|---|---|---|
| break | else | long | switch |
| case | enum | register | typedef |
| char | extern | return | union |
| const | float | short | unsigned |
| continue | for | signed | void |
| default | goto | sizeof | volatile |
| do | if | static | while |

Identifiers are user defined names and labels. Identifiers can contain letters of English alphabet (both upper and lower case) and numbers. The starting character of an identifier should be a letter. The only special character allowed in identifier is underscore (_).

*9.3.3.2 Data Types* Data type represents the type of data held by a variable. The various data types supported, their storage space (bits) and storage capacity for 'C' language are tabulated below.

| Data Type | Size (Bits) | Range | Comments |
|---|---|---|---|
| char | 8 | –128 to +127 | Signed character |
| signed char | 8 | –128 to +127 | Signed character |
| unsigned char | 8 | 0 to +255 | Unsigned character |
| short int | 8 | –128 to +127 | Signed short integer |
| signed short int | 8 | –128 to +127 | Signed short integer |
| unsigned short int | 8 | 0 to +255 | Unsigned short integer |
| int | 16 | –32,768 to +32,767 | Signed integer |
| signed int | 16 | –32,768 to +32,767 | Signed integer |
| unsigned int | 16 | 0 to +65,535 | Unsigned integer |
| long int | 32 | –2147,483,648 to +2,147,483,647 | Signed long integer |
| signed long int | 32 | –2147,483,648 to +2,147,483,647 | Signed long integer |
| unsigned long int | 32 | 0 to +4,294,967,295 | Unsigned long integer |
| float | 32 | 3.4E-38 to 3.4E+38 | Signed floating point |
| double | 64 | 1.7E-308 to 1.7E+308 | Signed floating point (Double precision) |
| long double | 80 | 3.4E-4932 to 3.4E+4932 | Signed floating point (Long Double precision) |

The data type size and range given above is for an ordinary 'C' compiler for 32 bit platform. It should be noted that the storage size may vary for data type depending on the cross-compiler in use for embedded applications.

Since memory is a big constraint in embedded applications, select the optimum data type for a variable. For example if the variable is expected to be within the range 0 to 255, declare the same as an 'unsigned char' or 'unsigned short int' data type instead of declaring it as 'int' or 'unsigned int'. This will definitely save considerable amount of memory.

### 9.3.3.3 Storage Class

Keywords related to storage class provide information on the scope (visibility or accessibility) and life time (existence) of a variable. 'C' supports four types of storage classes and they are listed below.

| Storage class | Meaning | Comments |
|---|---|---|
| auto | Variables declared inside a function. Default storage class is auto | Scope and accessibility is restricted within the function where the variable is declared. No initialization. Contains random values at the time of creation |
| register | Variables stored in the CPU register of processor. Reduces access time of variable | Same as auto in scope and access. The decision on whether a variable needs to be kept in CPU register of the processor depends on the compiler |
| static | Local variable with life time same as that of the program | Retains the value throughout the program. By default initialises to zero on variable creation. Accessibility depends on where the variable is declared |
| extern | Variables accessible to all functions in a file and all files in a multiple file program | Can be modified by any function within a file or across multiple files (variable needs to be exported by one file and imported by other files using the same) |

Apart from these four storage classes, 'C' literally supports storage class 'global'. An 'auto or static' variable declared in the public space of a file (declared before the implementation of all functions including main in a file) is accessible to all functions within that file. There is no explicit storage class for 'global'. The way of declaration of a variable determines whether it is global or not.

### 9.3.3.4 Arithmetic Operations

The list of arithmetic operators supported by 'C' are listed below

| Operator | Operation | Comments |
|---|---|---|
| + | Addition | Adds variables or numbers |
| – | Subtraction | Subtracts variables or numbers |
| * | multiplication | multiplies variables or numbers |
| / | Division | Divides variables or numbers |
| % | Remainder | Finds the remainder of a division |

### 9.3.3.5 Logical Operations

Logical operations are usually performed for decision making and program control transfer. The list of logical operations supported by 'C' are listed below

| Operator | Operation | Comments |
|---|---|---|
| && | Logical AND | Performs logical AND operation. Output is true (logic 1) if both operands (left to and right to of && operator) are true |
| \|\| | Logical OR | Performs logical OR operation. Output is true (logic 1) if either operand (operands to left or right of \|\| operator) is true |
| ! | Logical NOT | Performs logical Negation. Operand is complemented (logic 0 becomes 1 and vice versa) |

*9.3.3.6 Relational Operations*  Relational operations are normally performed for decision making and program control transfer on the basis of comparison. Relational operations supported by 'C' are listed below.

| Operator | Operation | Comments |
|---|---|---|
| < | less than | Checks whether the operand on the left side of '<' operator is less than the operand on the right side. If yes return logic one, else return logic zero |
| > | greater than | Checks whether the operand on the left side of '>' operator is greater than the operand on the right side. If yes return logic one, else return logic zero |
| <= | less than or equal to | Checks whether the operand on the left side of '<=' operator is less than or equal to the operand on the right side. If yes return logic one, else return logic zero |
| >= | greater than | Checks whether the operand on the left side of '>=' operator is greater than or equal to the operand on the right side. If yes return logic one, else return logic zero |
| == | Checks equality | Checks whether the operand on the left side of '==' operator is equal to the operand on the right side. If yes return logic one, else return logic zero |
| != | Checks non-equality | Checks whether the operand on the left side of '!=' operator is not equal to the operand on the right side. If yes return logic one, else return logic zero |

*9.3.3.7 Branching Instructions*  Branching instructions change the program execution flow conditionally or unconditionally. Conditional branching depends on certain conditions and if the conditions are met, the program execution is diverted accordingly. Unconditional branching instructions divert program execution unconditionally.

Commonly used conditional branching instructions are listed below

| Conditional branching instruction | Explanation |
|---|---|
| //if statement<br><br>if (expression)<br>{<br>statement1;<br>statement2;<br>.............;<br>}<br>statement 3;<br>.............; | Evaluates the expression first and if it is true executes the statements given within the { } braces and continue execution of statements following the closing curly brace (}). Skips the execution of the statements within the curly brace { } if the expression is false and continue execution of the statements following the closing curly brace (}).<br><br>One way branching |
| //if else statement<br>if (expression)<br>{<br>if_statement1;<br>if_statement2;<br>..............;<br>}<br>else<br>{<br>else_statement1; | Evaluates the expression first and if it is true executes the statements given within the { } braces following if (expression) and continue execution of the statements following the closing curly brace (}) of else block. Executes the statements within the curly brace { } following the else, if the expression is false and continue execution of statements following the closing curly brace (}) of else. |

```
else statement2;                    Two way branching
..................;
}
statement 3;
```

| | |
|---|---|
| `//switch case statement`<br>`switch (expression)`<br>`{`<br>`    case value1:`<br>`        break;`<br>`    case value2:`<br>`        break;`<br>`    default:`<br>`        break;`<br>`}` | Tests the value of a given expression against a list of case values for a matching condition. The expression and case values should be integers. value1, value2, etc. are integers. If a match found, executes the statement following the case and breaks from the switch. If no match found, executes the default case.<br><br>Used for multiple branching. |
| `//Conditional operator`<br>`// ?exp1 : exp2`<br><br>`(expression) ?exp1: exp2`<br><br>`E.g`<br>`    if (x>y)`<br>`    a=1;`<br>`    else`<br>`    a=0;`<br>`can be written using conditional`<br>`operator as`<br>`a=(x>y)? 1:0` | Used for assigning a value depending on the (expression). (expression) is calculated first and if it is greater than 0, evaluates exp1 and returns it as a result of operation else evaluate exp2 and returns it as result. The return value is assigned to some variable.<br><br>It is a combination of if else with assignment statement.<br><br>Used for two way branching |
| `//unconditional branching`<br><br>`goto label` | goto is used as unconditional branching instruction. goto transfers the program control indicated by a label following the goto statement. The label indicated by goto statement can be anywhere in the program either before or after the goto label instruction.<br><br>goto is generally used to come out of deeply nested loops in abnormal conditions or errors. |

*9.3.3.8 Looping Instructions*   Looping instructions are used for executing a particular block of code repeatedly till a condition is met or wait till an event is fired. Embedded programming often uses the looping instructions for checking the status of certain I/o ports, registers, etc. and also for producing delays. Certain devices allow write/read operations to and from some registers of the device only when the device is ready and the device ready is normally indicated by a status register or by setting/clearing certain bits of status registers. Hence the program should keep on reading the status register till the device ready indication comes. The reading operation forms a loop. The looping instructions supported by 'C' are listed below.

| Looping instruction | Explanation |
|---|---|
| //while statement<br>while (expression)<br>{<br><br>body of while loop<br><br><br>} | Entry controlled loop statement.<br>The expression is evaluated first and if it is true the body of the loop is entered and executed. Execution of 'body of while loop' is repeated till the expression becomes false. |
| // do while loop<br><br>do<br>{<br><br>body of do loop<br><br>}<br><br>while (expression); | The 'body of the loop' is executed at least once. At the end of each execution of the 'body of the loop', the while condition (expression) is evaluated and if it is true the loop is repeated, else loop is terminated. |
| //for loop<br><br>for (initialisation; test for condition; update variable)<br>{<br>body of for loop<br>} | Entry controlled loop. Enters and executes the 'body of loop' only if the test for condition is true. *for* loop contains a loop control variable which may be initialised within the initialisation part of the loop. Multiple variables can be initialised with ',' operator. |
| //exiting from loop<br><br><br>break;<br><br>goto label | Loops can be exited in two ways. First one is normal exit where loop is exited when the expression/test for condition becomes false. Second one is forced exit. break and goto statements are used for forced exit.<br>break exits from the innermost loop in a deeply nested loop, whereas goto transfers the program flow to a defined label. |
| //skipping portion of a loop<br><br>while (expression)<br>{<br>.............;<br>if (condition);<br>continue;<br>.............;<br><br>}<br>//do while with skipping<br>do<br>{<br>..............;<br>if (condition)<br>continue;<br>..............;<br><br>}<br>while (expression); | Certain situation demands the skipping of a portion of a loop for some conditions. The 'continue' statement used inside a loop will skip the rest of the portion following it and will transfer the program control to the beginning of the loop.<br>//for loop with skipping<br><br><br><br>for (initialisation; test for condition; update variable)<br>{<br>.................;<br>if (condition)<br>continue;<br>.................;<br>} |

*Every 'for' loop can be replaced by a 'while' loop with a counter.*

Let's consider a typical example for a looping instruction in embedded C application. I have a device which is memory mapped to the processor and I'm supposed to read the various registers of it (except status register) only after the contents of its status register, which is memory mapped at 0x3000 shows device is ready (say value 0x01 means device is ready). I can achieve this by different ways as given below.

```
//################################################################
//using while loop
//################################################################

char *status_reg = (char *) 0x3000; //Declares memory mapped register

while (*status_reg!=0x01);        //Wait till status_reg = 0x01


//################################################################
//using do while loop
//################################################################

char *status_reg = (char*) 0x3000;


do
{

} while (*status_reg!=0x01); Loop till status_reg = 0x01


//################################################################
//using for loop
//################################################################

char *status_reg = (char*) 0x3000;

for (;(*status_reg!=0x01););
```

The instruction *char \*status_reg = (char\*) 0x3000;* declares *status_reg* as a character pointer pointing to location 0x3000. The character pointer is used since the external device's register is only 8bit wide. We will discuss the pointer based memory mapping technique in a later section. In order to avoid compiler optimisation, the pointer should be declared as volatile pointer. We will discuss the same also in another section.

*9.3.3.9 Arrays and Pointers*   Array is a collection of related elements (data types). Arrays are usually declared with data type of array, name of the array and the number of related elements to be placed in the array. For example the following array declaration

```
char arr [5];
```

declares a character array with name 'arr' and reserves space for 5 character elements in the memory as in Fig. 9.3†.

Contents of each memory location†

| 0x10 | 0x00 | 0x23 | 0x03 | 0x45 |
|------|------|------|------|------|

| arr[0] | arr[1] | arr[2] | arr[3] | arr[4] |
|--------|--------|--------|--------|--------|

| 0x8000 | 0x8001 | 0x8002 | 0x8003 | 0x8004 |
|--------|--------|--------|--------|--------|

Address of the memory location where the array elements are stored†
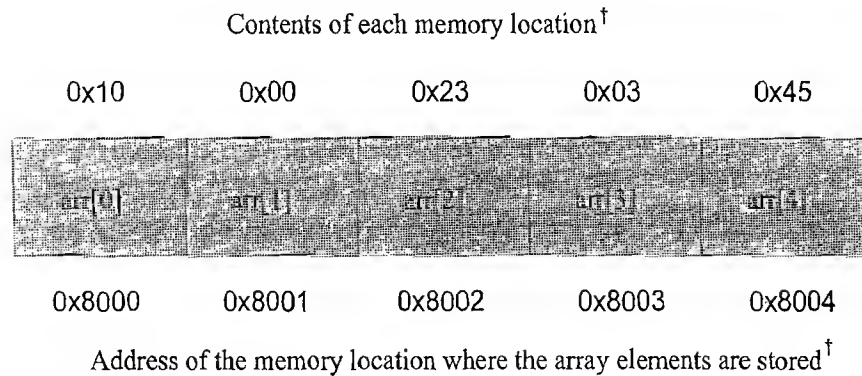
**Fig. 9.3** **Array representation in memory**

The elements of an array are accessed by using the array index or subscript. The index of the first element is '0'. For the above example the first element is accessed by arr[0], second element by arr[1], and so on. In the above example, the array starts at memory location 0x8000 (arbitrary value taken for illustration) and the address of the first element is 0x8000. The *address of* operator (&) returns the address of the memory location where the variable is stored. Hence &arr[0] will return 0x8000 and &arr[1] will return 0x8001, etc. The name of the array itself with no index (subscript) always returns the address of the first element. If we examine the first element arr[0] of the above array, we can see that the variable arr[0] is allocated a memory location 0x8000 and the contents of that memory location holds the value for arr[0] (Fig. 9.4).

&arr[0]  0x8000

arr[0]  0x10
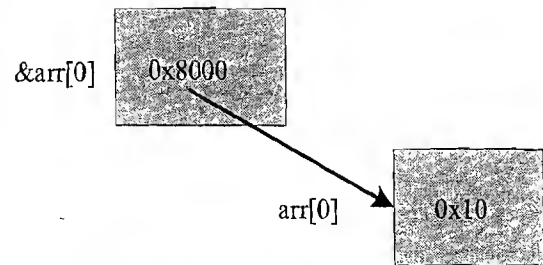
**Fig. 9.4** **Array element address and content relationship**

Arrays can be initialised by two methods. The first method is initialising the entire array at the time of array declaration itself. Second method is selective initialisation where any member can be initialised or altered with a value.

```
//Initialization of array at the time of declaration

unsigned char arr[5] = {5, 10, 20, 3, 2};
unsigned char arr[ ] = {5, 10, 20, 3, 2};

//Selective initialization

unsigned char arr[5];

arr[0] = 5;
arr[1] = 10;
arr[2] = 20;
arr[3] = 3;
arr[4] = 2;
```

† Arbitrary value taken for illustration.

### A few important points on Arrays

1. The '*sizeof()*' operator returns the size of an array as the number of bytes. E.g. '*sizeof (arr)*' in the above example returns 5. If arr[ ] is declared as an integer array and if the byte size for integer is 4, executing the '*sizeof (arr)*' instruction for the above example returns 20 ($5 \times 4$).

2. The '*sizeof()*' operator when used for retrieving the size of an array which is passed as parameter to a function will only give the size of the data type of the array.

```
E.g.
    void test (char *p);          //Function declaration
    char arr [5] = {5, 10, 20, 3, 2};    //Array data type char

    void main ( )
    {
    test (arr);
    }

    void test (char *p)
    {
    printf ("%d", sizeof (*p));
    }
```

This code snippet will print '1' as the output, though the user expects 5 (size of arr) as output. The output is equivalent to sizeof (char), size of the data type of the array.

3. Use the syntax '*extern array type array name [ ]*' to access an array which is declared outside the current file. For example '*extern char arr[ ]*' for accessing the array '*arr[ ]*' declared in another file

4. Arrays are not equivalent to pointers. But the expression '*array name*' is equivalent to a pointer, of type specified by the array, to the first element of an array, e.g. '*arr*' illustrated for sizeof () operator is equivalent to a character pointer pointing to the first element of array '*arr*'.

5. Array subscripting is commutative in 'C' language and '*arr[k]*' is same as '*\*((arr)+(k))*' where '*arr[k]*' is the content of $k^{th}$ element of array '*arr*' and '*(arr)*' is the starting address of the array arr and $k$ is the index of the array or offset address for the '$k^{th}$' element from the base address of array. '*\*((arr) + (k))*' is the content of array for the index $k$.

**Pointers**  *Pointer* is a flexible at the same time most dangerous feature, capable of creating potential damages leading to firmware crash, if not used properly. Pointer is a memory pointing based technique for variable access and modification. Pointers are very helpful in

1. Accessing and modifying variables
2. Increasing speed of execution
3. Accessing contents within a block of memory
4. Passing variables to functions by eliminating the use of a local copy of variables
5. Dynamic memory allocation (Will be discussed later)

To understand the pointer concept, let us have a look at the data memory organisation of a processor. For a processor/controller with 128 bytes user programmable internal RAM (e.g. AT89C51), the memory is organised as
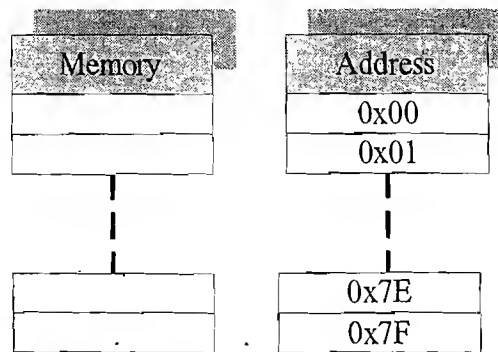
Fig. 9.5 **Data memory organisation for 8051**

If we declare a character variable, say *char input*, the compiler assigns a memory location to the variable anywhere within the internal memory 0x00 to 0x7F. The allocation is left to compiler's choice unless specified explicitly (Let it be 0x45 for our example). If we assign a value (say 10) to the variable *input* (input=10), the memory cell representing the variable input is loaded with 10.



| Variable Name | Memory Address | Content |
|:---:|:---:|:---:|
| input | 0x45 | 10 |
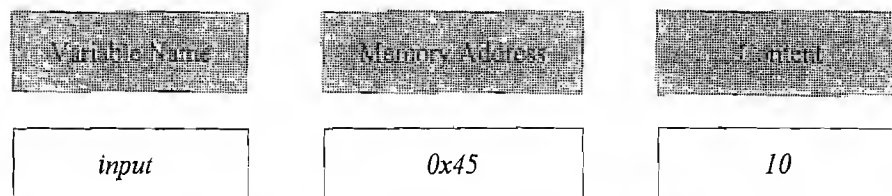
Fig. 9.6 **Relationship between variable name, address and data held by variable**

The contents of memory location 0x45 (representing the variable *input*) can be accessed and modified by using a pointer of type same as the variable (*char* for the variable *input* in the example). It is accomplished by the following method.

```
char input;     //Declaring input as character variable
char *p; //Declaring a character pointer p (* denotes p is a pointer)
p = &input      //Assigns the address of input as content to p
```

The same is diagrammatically represented as



| Variable Name | Memory Address | Content |
|:---:|:---:|:---:|
| input | 0x45 | 10 |
| p | 0x00 | 0x45 |
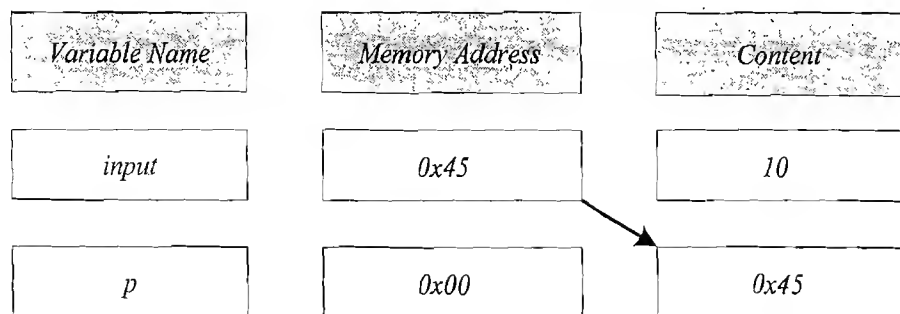
Fig. 9.7 **Pointer based memory access technique**

The compiler assigns a memory to the character pointer variable '*p*'. Let it be 0x00 (Arbitrary value chosen for illustration) and the memory location holds the memory address of variable *input* (0x45)

as content. In 'C' the address assignment to pointer is done using the address of operator '&' or it can be done using explicitly by giving a specific address. The pointer feature in 'C' is same as the indirect addressing technique used in *8051* Assembly instructions. The code snippet

```
MOV R0, #45H
MOV A,@R0 ; R0 & R1 are the indirect addressing registers.
```

is an example for 8bit memory pointer usage in *8051* Assembly and the code snippet

```
MOV DPTR, #0045H
MOV A,@DPTR ; DPTR is the 16bit indirect addressing register
```

is an example for 16bit memory pointer operation. The general form of declaring a pointer in 'C' is

```
data type *pointer; //'data type' is the standard data type like
                    //int, char, float etc.. supported by 'C' language.
```

The * (asterisk) symbol informs the compiler that the variable *pointer* is a pointer variable. Like any other variables, pointers can also be initialised in its declaration itself.

```
E.g.      char x, y;
          char *ptr = &x;
```

The contents pointed by a pointer is modified/retrieved by using * as prefix to the pointer.

```
E.g.      char x=5, y=56;
          char *ptr=&x;        //ptr holds address of x
          *ptr = y;            //x = y
```

***Pointer Arithmetic and Relational Operations***   'C' language supports the following Arithmetic and relational operations on pointers.

1. Addition of integer with pointer. e.g. ptr+2 (It should be noted that the pointer is advanced forward by the storage length supported by the compiler for the data type of the pointer multiplied by the integer. For example for integer pointer where storage size of int = 4, the above addition advances the pointer by $4 \times 2 = 8$)
2. Subtraction of integer from pointer, e.g. ptr–2 (Above rule is applicable)
3. Incremental operation of pointer, e.g. ++ptr and ptr++ (Depending on the type of pointer, the ++ increment context varies). For a character pointer ++ operator increments the pointer by 1 and for an integer pointer the pointer is incremented by the storage size of the integer supported by the compiler (e.g. pointer ++ results in pointer + 4 if the size for integer supported by compiler is 4)
4. Decrement operation of pointer, e.g. – –ptr and ptr– – (Context rule for decrement operation is same as that of incremental operation)
5. Subtraction of pointers, e.g. ptr1– ptr2
6. Comparison of two pointers using relational operators, e.g. ptr1 > ptr2, ptr1 < ptr2, ptr1 = = ptr2, ptr1! = ptr2 etc (Comparison of pointers of same type only will give meaningful results)

> ***Note:***
> 1. Addition of two pointers, say *ptr1* + *ptr2* is illegal
> 2. Multiplication and division operations involving pointer as numerator or denominator are also not allowed

### A few important points on Pointers

1. The instruction *ptr++ increments only the pointer not the content pointed by the pointer '*ptr*' and *ptr– – decrements the pointer not the contents pointed by the pointer '*ptr*'

2. The instruction (*ptr) ++ increments the content pointed by the pointer '*ptr*' and not the pointer '*ptr*'. (*ptr)-- decrements the content pointed by the pointer '*ptr*' and not the pointer '*ptr*'

3. A type-casted pointer cannot be used in an assignment expression and cannot be incremented or decremented, e.g. ((int *ptr))++; will not work in the expected way

4. '*Null Pointer*' is a pointer holding a special value called '*NULL*' which is not the address of any variable or function or the start address of the allocated memory block in dynamic memory allocations

5. Pointers of each type can have a related null pointer viz. there can be character type null pointer, integer type null pointer, etc.

6. '*NULL*' is a preprocessor macro which is literally defined as zero or ((void *) 0). #define NULL 0 or #define NULL ((void *) 0)

7. '*NULL*' is a constant zero and both can be used interchangeably as Null pointer constants

8. A '*NULL*' pointer can be checked by the operator if ( ). See the following example

```
if (ptr)   //ptr is a pointer declared
printf ("ptr is not a NULL pointer");
else
printf ("ptr is a NULL pointer");
```

The statement if (ptr) is converted to if (ptr! =0) by the (cross) compiler. Alternatively you can directly use the statement if (ptr! =0) in your program to check the 'NULL' pointer.

9. Null pointer is a 'C' language concept and whose internal value does not matter to us. '*NULL*' always guarantee a '0' to you but Null pointer need not be.

**Pointers and Arrays—Are they related?** Arrays are not equivalent to pointers and vice versa. But the expression array 'name [ ]' is equivalent to a pointer, of type specified by the array, to the first element of an array, e.g. for the character array '*char arr[5]*', '*arr[ ]*' is equivalent to a character pointer pointing to the first element of array '*arr*' (This feature is referred to as '*equivalence of pointers and arrays*'). You can achieve the array features like accessing and modifying members of an array using a pointer and pointer increment/decrement operators. Arrays and pointer declarations are interchangeable when they are used as parameters to functions. Point 2 discussed under the section '**A few important points on Arrays**' for *sizeof()* operator usage explains this feature also.

*9.3.3.10 Characters and Strings*   Character is a one byte data type and it can hold values ranging from 0 to 255 (unsigned character) or -128 to +127 (signed character). The term character literally refers to the alpha numeric characters (English alphabet A to Z (both small letters and capital letters) and number representation from '0' to '9') and special characters like *, ?, !, etc. An integer value ranging from 0 to 255 stored in a memory location can be viewed in different ways. For example, the hexadecimal number 0x30 when represented as a character will give the character '0' and if it is viewed as a decimal number it will give 48. String is an array of characters. A group of characters defined within a double quote represents a constant string.

'*H*' is an example for a character, whereas "*Hello*" is an example for a string. String always terminates with a '\0' character. The '\0' character indicates the string termination. Whenever you declare a string using a character array, allocate space for the null terminator '\0' in the array length.

```
E.g.    char name [ ] = "SHIBU" ;
or      char name [6] = {'S', 'H', 'I', 'B', 'U', '\0'};
```

String operations are very important in embedded product applications. Many of the embedded products contain visual indicators in the form of alpha numeric displays and are used for displaying text. Though the conventional alpha numeric displays are giving way to graphic displays, they are deployed widely in low cost embedded products. The various operations performed on character strings are explained below.

**Input & Output operations** Conventional 'C' programs running on Desktop machines make use of the standard string inputting (*scanf()*) and string outputting (*printf()*) functions from the platform specific I/O library. Standard keyboard and monitor are used as the input and output media for desktop application. This is not the case for embedded systems. Embedded systems are compact and they need not contain a standard keyboard or monitor screen for I/O functions. Instead they incorporate application specific keyboard and display units (Alpha numeric/graphic) as user interfaces. The standard string input instruction supported by 'C' language is *scanf( )* and a code snippet illustrating its usage is given below.

```
char name [6];
scanf ("%s", name);
```

A standard ANSI C compiler converts this code according to the platform supported I/O library files and waits for inputting a string from the keyboard. The scanf function terminates when a white space (blank, tab, carriage return, etc) is encountered in the input string. Implementation of the *scanf()* function is (cross) compiler dependent. For example, for 8051 microcontroller all I/O operations are supposed to be executed by the serial interface and the C51 cross compiler implements the *scanf()* function in such a way to expect and receive a character/string (according to the scanf usage context (character if first parameter to *scanf()* is "%c" and string if first parameter is "%s")) from the serial port.

*printf()* is the standard string output instruction supported by 'C' language. A code snippet illustrating the usage of *printf( )* is given below.

```
char name [ ] = "SHIBU";
printf ("%s", name);
```

Similar to *scanf()* function, the standard ANSI C compiler converts this code according to the platform supported I/O library files and displays the string in a console window displayed on the monitor. Implementation of *printf()* function is also (cross) compiler specific. For 8051 microcontroller the C51 cross compiler implements the *printf()* function in such a way to send a character/string (according to the printf usage context (character if first parameter to *printf()* is "%c" and string if first parameter is "%s")) to the serial port.

**String Concatenation** Concatenation literally means '*joining together*'. String concatenation refers to the joining of two or more strings together to form a single string. 'C' supports built in functions for string operations. To make use of the built in string operation functions, include the header file '*string. h*' to your '*.c*' file. '*strcat()*' is the function used for concatenating two strings. The syntax of '*strcat()*' is illustrated below.

```
strcat (str1, str2); //'str1' and 'str2' are the strings to
                     //be concatenated.
```

On executing the above instruction the null character ('\0') from the end of '*str1*' is removed and '*str2*' is appended to '*str1*'. The string '*str2*' remains unchanged. As a precautionary measure, ensure that str1 (first parameter of '*strcat( )*' function) is declared with enough size to hold the concatenated string. '*strcat()*' can also be used for appending a constant string to a string variable.

```
E.g.    strcat(str1, "Hello!");
```

> **Note:**
> 1. Addition of two strings, say *str1+str2* is not allowed
> 2. Addition of a string variable, say str1, with a constant string, say "*Hello*" is not allowed

**String Comparison**    Comparison of strings can be performed by using the '*strcmp()*' function supported by the string operation library file. Syntax of *strcmp()*' is illustrated below.

```
strcmp (str1, str2); //str1 and str2 are two character strings
```

If the two strings which are passed as arguments to '*strcmp()*' function are equal, the return value of '*strcmp()*' will be zero. If the two strings are not equal and if the ASCII value of the first non-matching character in the string *str1* is greater than that of the corresponding character for the second string *str2*, the return value will be greater than zero. If the ASCII value of first non-matching character in the string *str1* is less than that of the corresponding character for the second string *str2*, the return value will be less than zero. '*strcmp()*' function is used for comparing two string variables, string variable and constant string or two constant strings.

```
E.g.    char str1 [ ] = "Hello world";
        char str2 [ ] = "Hello World!"  ;
        int n;
        n= strcmp(str1, str2);
```

Executing this code snippet assigns a value which is greater than zero to *n*. (since ASCII value of '*w*' is greater than that of '*W*'). *n= strcmp(str2, str1);* will assign a value which is less than zero to *n*. (since ASCII value of '*W*' is less than that of '*w*').

The function *stricmp()* is another version of *strcmp()*. The difference between the two is, *stricmp()* is not case sensitive. There won't be any differentiation between upper and lowercase letters when *stricmp()* is used for comparison. If *stricmp()* is used for comparing the two strings str1 and str2 in the above example, the return value of the function *stricmp* (str1, str2) will be zero.

> **Note:**
> 1. Comparison of two string variables using '= =' operator is invalid, e.g. if (*str1* == *str2*) is invalid
> 2. Comparison of a string variable and a string constant using '= =' operator is also invalid, e.g. if (*str1* == "*Hello*") is invalid

**Finding String length**    String length refers to the number of characters except the null terminator character '\0' present in a string. String length can be obtained by using a counter combined with a search operation for the '\0' character. 'C' supplies a ready to use string function **strlen()** for determining the length of a string. Its syntax is explained below.

```
strlen (str1);          //where str1 is a character string
```

```
e.g.    char str1 [ ] = "Hello World!"  ;
        int n;
        n = strlen (str1);
```

Executing this code snippet assigns the numeric value 12 to integer variable '*n*'.

**Copying Strings**    *strcpy()* function is the 'C' supported string operation function for copying a string to another string. Syntax of *strcpy( )* function is

```
strcpy (str1, str2);    //str1, str2 are character strings.
```

This function assigns the contents of *str2* to *str1*. The original content of str1 is overwritten by the contents of *str2*. *str2* remains unchanged. The size of the character array which is passed as the first argument the *strcpy( )* function should be large enough to hold the copied string. *strcpy()* function can also be used to assign constant string to string variables.

```
e.g.    strcpy(str1, "Hello!")
```

> **Note:**
> A string variable cannot be copied to another string variable using the '=' operator e.g. *str1 = str2* is illegal

### A few important points on Characters and Strings

1. The function *strcat()* is used for concatenating two string variables or string variable and string constants. Characters cannot be appended to a string variable using *strcat()* function. For example *strcat (str1, 'A')* may not give the expected result. The same can be achieved by *strcat (str1, "A")*

2. Strings are character arrays and they cannot be assigned directly to a character array (except the initialisation of arrays using string constants at the time of declaring character arrays)

```
//##############################################################
E.g.    unsigned char str1 [] = 'Hello"; // is valid;
        unsigned char str1 [6];
        str1= "Hello"; //is invalid.
```

3. Whenever a character array is declared to hold a string, allocate size for the null terminator character '\0' also in the character array

*9.3.3.11 Functions*    Functions are the basic building blocks of modular programs. A function is a self-contained and re-usable code snippet intended to perform a particular task. 'Embedded *C*' supports two different types of functions namely, *library functions* and *user defined functions*.

*Library functions* are the built in functions which is either part of the standard 'Embedded C' library or user created library files. 'C' provides extensive built in library file support and the library files are categorised into various types like I/O library functions, string operation library functions, memory allocation library functions etc. *printf()*, *scanf ()*, etc. are examples of I/O library functions. *strcpy()*, *strcmp()*, etc. are examples for string operations library functions. *malloc()*, *calloc()* etc are examples of memory allocation library functions supported by 'C'. All library functions supported by a particular library is implemented and exported in the same. A corresponding header ('*.h*') file for the library file provides information about the various functions available to user in a library file. Users should include the header file corresponding to a particular library file for calling the functions from that library in the

'C' source file. For example, if a programmer wants to use the standard I/O library function *printf()* in the source file, the header file corresponding to the I/O library file ("*stdio.h*" meant for standard i/o) should be included in the 'c' source code using the #include preprocessor directive.

```
E.g.
        #include <stdio.h>
        void main ()
        {
            printf("Hello World");
        }
```

"*string.h*" is the header file corresponding to the built in library functions for string operations and "*malloc.h*" is the header file for memory allocation library functions. Readers are requested to get info on header files for other required libraries from the standard ANSI library. As mentioned earlier, the standard 'C' library function implementation may be tailored by cross-compilers, keeping the function name and parameter list unchanged depending on Embedded 'C' application requirements. *printf()* function implemented by C51 cross compiler for *8051* microcontroller is a typical example. The library functions are standardised functions and they have a unique style of naming convention and arguments. Users should strictly follow it while using library functions.

*User defined functions* are programmer created functions for various reasons like modularity, easy understanding of code, code reusability, etc. The generic syntax for a function definition (implementation) is illustrated below.

```
Return type   function name (argument list)
{
    //Function body (Declarations & statements)
    //Return statement
}
```

Return type of a function tells—what is the data type of the value returning by the function on completion of its execution. The return type can be any of the data type supported by 'C' language, viz. *int*, *char*, *float*, *long*, etc. *void* is the return type for functions which do not return anything. Function name is the name by which a function is identified. For user defined functions users can give any name of their interest. For library functions the function name for doing certain operations is fixed and the user should use those standard function names. Parameters are the list of arguments to be passed to the function for processing. Parameters are the inputs to the functions. If a function accepts multiple parameters, each of them are separated using ',' in the argument list. Arguments to functions are passed by two means, namely; pass by value and pass by reference. Pass by value method passes the variables to the function using local copies whereas in pass by reference variables are passed to the function using pointers. In addition to the above-mentioned attributes, a function definition may optionally specify the function's linkage also. The linkage of the function can be either '*external*' or '*internal*'.

The task to be executed by the function is implemented within the body of the function. In certain situations, you may have a single source ('*.c*') file containing the entire variable list, user defined functions, function *main()*, etc. There are two methods for writing user defined functions if the source code is a single '*.c*' file. The first method is writing all user defined functions on top of the *main* function and other user defined functions calling the user defined function.

```
E.g.

Int xyz (int i)
{
..........;
}

void abc (void)
{
..........;
xyz (a)
}

void main ()
{
    abc ();
}
```

If you are writing the user defined functions after the entry function *main()* and calling the same inside *main*, you should specify the function prototype (function declaration) of each user defined functions before the function *main()*. Otherwise the compiler assumes that the user defined function is an extern function returning integer value and if you define the function after *main()* without using a function declaration/prototype, compiler will generate error complaining the user defined function is redefining (Compiler already assumed the function which is used inside *main()* without a function prototype as an extern function returning an integer value). The function declaration informs the compiler about the format and existence of a function prior to its use. Implicit declaration of functions is not allowed: every function must be explicitly declared before it is called. The general form of function declaration is

```
Linkage Type Return type  function name (arguments);
E.g. static int add(int a, int b);
```

The '*Linkage Type*' specifies the linkage for the function. It can be either '*external*' or '*internal*'. The '*static*' keyword for the '*Linkage Type*' specifies the linkage of the function as internal whereas the '*extern*' '*Linkage Type*' specifies 'external' linkage for the function. It is not mandatory to specify the name of the argument along with its type in the argument list of the function declaration. The declarations can simply specify the types of parameters in the argument list. This is called function *prototyping*. A function prototype consists of the function return type, the name of the function, and the parameter list. The usage is illustrated below.

```
static int add(int, int);
```

Let us have a look at the examples for the different scenarios explained above on user defined functions.

```
//###############################################################
//Example for Source code with function prototype for user defined-
//functions. Source file test.c
//###############################################################
```

```
#include <stdio.h>
//function prototype for user defined function test
void test (void);

        void main ( )
        {
        test ();          //Calling user defined function from main
        }
//Implementation of user defined function test after function main
void test (void)
{
printf ("Hello World!");
return;
}


//###############################################################
//Example for Source code without function prototype for user defined
//functions. Source file test.c
//###############################################################
#include <stdio.h>
//function prototype for user defined function test not declared
        void main ()
        {
        test ();          //Calling user defined function from main
        }

//Implementation of user defined function test after function main
        void test (void)
        {
        printf ("Hello World!");
        return;
        }
```

*Compiler Output:*

Compiling...
test.c
test.c(5): warning c4013: 'test' undefined; assuming extern returning int
test.c(9): error C2371: 'test': redefinition; different basic types Error executing cl.exe.
test.exe-1 error(s), 1 warning(s)

There is another convenient method for declaring variables and functions involved in a source file. This technique is a header ('.h') file and source ('.c') file based approach. In this approach, corresponding to each 'c' source file there will be a header file with same name (not necessarily) as that of the 'c' source file. All functions and global/extern variables for the source file are declared in the header file instead of declaring the same in the corresponding source file. Include the header file where the functions are declared to the source file using the "#include" pre-processor directive. Functions declared in a file can be either global (extern access) in scope or static in scope depending on the declaration of the

function. By default all functions are global in scope (accessible from outside the file where the function is declared). If you want to limit the scope (accessibility) of the function within the file where it is declared, use the keyword '*static*' before the return type in the function declaration.

### 9.3.3.12 Function Pointers

A function pointer is a pointer variable pointing to a function. When an application is compiled, the functions which are part of the application are also get converted into corresponding processor/compiler specific codes. When the application is loaded in primary memory for execution, the code corresponding to the function is also loaded into the memory and it resides at a memory address provided by the application loader. The function name maps to an address where the first instruction (machine code) of the function is present. A function pointer points to this address.

The general form of declaration of a function pointer is given below.

```
return_type (*pointer_name) (argument list)
```

where, '*return_type*' represents the return type of the function, '*pointer_name*' represents the name of the pointer and 'argument list' represents the data type of the arguments of the function. If the function contains multiple arguments, the data types are separated using ','. Typical declarations of function pointer are given below.

```
//Function pointer to a function returning int and takes no parameter
int (*fptr) ();
//Function pointer to a function returning int and takes 1 parameter
int (*fptr) (int)
```

The parentheses () around the function pointer variable differentiates it as a function pointer variable. If no parentheses are used, the declaration will look like

```
int *fptr();
```

The cross compiler interprets it as a function declaration for function with name '*fptr*' whose argument list is void and return value is a pointer to an integer. Now we have declared a function pointer, the next step is 'How to assign a function to a function pointer?'.

Let us assume that there is a function with signature

```
int function1(void);
```

and a function pointer with declaration

```
int (*fptr) ();
```

We can assign the address of the function '*function1()*' to our function pointer variable '*fptr*' with the following assignment statement:

```
fptr = &function1;
```

The '*&*' operator gets the address of function '*function1*' and it is assigned to the pointer variable '*fptr*' with the assignment operator '='. The address of operator '*&*' is optional when the name of the function is used. Hence the assignment operation can be re-written as:

```
fptr = function1;
```

Once the address of the right sort of function is assigned to the function pointer, the function can be invoked by any one of the following methods.

```
(*fptr)();
fptr();
```

Function pointers can also be declared using *typedef*. Declaration and usage of a function pointer with *typedef* is illustrated below.

```
//Function pointer to a function returning int and takes no parameter
    typedef int (*funcptr)();
    funcptr fptr;
```

The following sample code illustrates the declaration, definition and usage of function pointer.

```
#include <stdio.h>
void square(int x);
void main()
{
    //Declare a function pointer
    void (*fptr)(int);
    //Define the function pointer to function square
    fptr = square;
    //Style 1: Invoke the function through function pointer
    fptr(2);
    //Style 2: Invoke the function through function pointer
    (*fptr)(2);
}
//###############################################################
//Function for printing the square of a number
void square(int x)
{
    printf("Square of %d = %d\n", x, x*x);
}
```

Function pointer is a helpful feature in late binding. Based on the situational need in the application you can invoke the required function by binding the function with the right sort of function pointer (The function signature and function pointer signature should be matching). This reduces the usage of 'if' and 'switch – case' statements with function names. Function pointers are extremely useful for handling situations which demand passing of a function as argument to another function. Function pointers are often used within functions where the function should be able to work with a number of functions whose names are not known until the program is running. A typical example for this is callback functions, which requires the information about the function which needs to be called. The following sample piece of code illustrates the usage of function pointer as parameter to a function.

```
#include <stdio.h>
//###############################################################
//Function prototype declaration
void square(int x);
void cube(int x);
void power(void (*fptr)(int), int x);
```

```
void main()
{
      //Declare a function pointer
      void (*fptr)(int);
      //Define the function pointer to function square
      fptr = square;
      //Invoke the function 'square' through function pointer
      power (fptr,2);
      //Define the function pointer to function cube
      fptr = cube;
      //Invoke the function 'cube' through function pointer
      power (fptr,2);
}
//###################################################################
//Interface function for invoking functions through function pointer

void power(void (*fptr)(int), int x)
{
      fptr(x);

}
//###################################################################
//Function for printing the square of a number

void square(int x)
{
      printf("Square of %d = %d\n", x, x*x);

}
//###################################################################
//Function for printing the third power (cube) of a number
void cube(int x)
{
      printf("Cube of %d = %d\n", x, x*x*x);
}
```

**Arrays of Function Pointers**  An array of function pointers holds pointers to functions with same type of signature. This offers the flexibility to select a function using an index. Arrays of function pointers can be defined using either direct function pointers or the '*typedef*' qualifier.

```
//Declare an array of pointers to functions, which returns int and
//takes no parameters, using direct function pointer declaration
int (*fnptrarr[5])();

//Declare and initialise an array of pointers to functions, which
//return int and takes no parameters, using direct function pointer-
//declaration
int (*fnptrarr[])()= {/*initialisation*/};
```

```
//Declare and initialize to 'NULL' an array of pointers to functions,
//which return int and takes no parameters, using direct function
//pointer declaration
int (*fnptrarr[5])()= {NULL};

//Declare an array of pointers to functions, which returns int and
//takes no parameters, using typedef function pointer declaration
typedef int (*fncptr)();
fncptr fnptrarr[5])();

//Declare and initialize an array of pointers to functions, which
//return int and takes no parameters, using typedef function pointer-
//declaration
typedef int (*fncptr)();
fncptr fnptrarr[] ()= {/*initialisation*/};

//Declare and initialise to 'NULL' an array of pointers to functions,
//which return int and takes no parameters, using typedef function
//pointer declaration
typedef int (*fncptr)();
fncptr fnptrarr[5]()= {NULL};
```

The following piece of code illustrates the usage of function pointer arrays:

```
#include <stdio.h>
//################################################################
//Function prototype definition
void square(int x);
void cube(int x);

void main()
{
        //Declare a function pointer array of size 2 and initialize
        void (*fptr[2])(int)= {NULL};
        //Define the function pointer 0 to function square
        fptr[0] = square;
        //Invoke the function square
        fptr[0](2);
        //Define the function pointer 1 to function cube
        fptr[1] = cube;
        //Invoke the function cube through function pointer
        fptr[1](2);
}
//################################################################
//Function for printing the square of a number
void square(int x)
{
        printf("Square of %d = %d\n", x, x*x);
}
```

```
//#########################################################################
//Function for printing the third power (cube) of a number

void cube(int x)
{
    printf("Cube of %d = %d\n", x, x*x*x);
}
```

### 9.3.3.13 Structures and Unions

*'structure'* is a variable holding a collection of data types (int, float, char, long etc). The data types can be either unique or distinct. The tag *'struct'* is used for declaring a structure. The general form of a structure declaration is given below.

```
struct   struct_name
        {
            //variable 1 declaration
            //variable 2 to declaration
            //............
            //variable n declaration
        };
```

*struct_name* is the name of the structure and the choice of the name is left to the programmer.

Let us examine the details kept in an employee record for an employee in the employee database of an organisation as example. A typical employee record contains information like *'Employee Name'*, *'Employee Code'*, and *'Date of Birth'*. This information can be represented in 'C' using a structure as given below.

```
struct   employee
        {
        char emp_name [20]; // Allowed maximum length for name = 20
        int emp_code;
        char DOB [10]; // DD-MM-YYYY Format (10 character)
        };
```

Hence in 'C', the employee record is represented by two string variables (character arrays) and an integer variable. Since these three variables are relevant to each employee, they are grouped together in the form of a structure. Literally structure can be viewed as a collection of related data. The above structure declaration does not allocate memory for the variables declared inside the structure. It's a mere representation of the data inside a structure. To allocate memory for the structure we need to create a variable of the structure. The structure variable creation is illustrated below.

```
struct employee emp1;
```

Keyword *'struct'* informs the compiler that the variable *'emp1'* is a structure of type *'employee'*. The name of the structure is referred as *'structure tag'* and the variables declared inside the structure are called *'structure elements'*. The definition of the structure variable only allocates the memory for the different elements and will not initialise the members. The members need to be initialised explicitly. Members of the structure variable can be initialised altogether at the time of declaration of the structure variable itself or can be initialised (modified) independently using the '.' operator (member operator).

```
struct employee emp1= {"SHIBU K V", 42170, "11-11-1977"};
```

This statement declares a structure variable with name 'emp1' of type employee and each elements of the structure is initialised as

```
emp_name = "SHIBU K V"
emp_code = 42170
DOB= "11-11-1977"
```

It should be noted that the variables should be initialised in the order as per their declaration in the structure variable. The selective method of initialisation/modification is used for initialising /modifying each element independently.

```
E.g.        struct employee emp1;
            emp1.emp_code = 42170;
```

All members of a structure, except character string variables can be assigned values using '.' Operator and assignment ('=') operator (character strings are character arrays and character arrays cannot be initialised altogether using the assignment operator). Character string variables can be assigned using string copy function (*strcpy*).

```
        strcpy (emp1.emp_name,"SHIBU K V");
        strcpy (emp1.DOB,"11-11-1977");
```

Declaration of a structure variable requires the keyword '*structure*' as the first word and it may sound awkward from a programmer point of view. This can be eliminated by using '*typedef*' in defining the structure. The above 'employee' structure example can be re-written using typedef as

```
typedef struct
    {
    char emp_name [20];// Allowed maximum length for name = 20
    int emp_code;
    char DOB [10];      // DD-MM-YYYY Format (10 character)
    } employee;
employee emp1; //No need to add struct before employee
```

This approach eliminates the need for adding the keyword '*struct*' each time a structure variable is declared.

***Structure operations***     The following table lists the various operations supported by structures

| Operator | Operation | Example |
|---|---|---|
| = (Assignment) | Assigns the values of one structure to another structure of same type | employee emp1,emp2;<br>emp1.emp_code = 42170;<br>strcpy(emp1.emp_name,"SHIBU");<br>strcpy(emp1.DOB,"11/11/1977");<br>emp2=emp1; |
| == (Checking the equality of all members of two structures) | Compare individual members of two structures of same type for equality. Return 1 if all members are identical in both structures else return 0 | employee emp1,emp2;<br>emp1.emp_code = 42170;<br>strcpy(emp1.emp_name,"SHIBU");<br>strcpy(emp1.DOB,"11/11/1977");<br>if (emp2==emp1) |

| | Compare individual members of two structures of same type for non equality. Return 1 if all members are not identical in both structures else return 0 | employee emp1,emp2; emp1.emp_code = 42170; strcpy(emp1.emp_name,"SHIBU"); strcpy(emp1 DOB,"11/11/1977"); if(emp2!=emp1) |
|---|---|---|
| != (Checking the equality of all members of two structures) | | |
| sizeof() | Returns the size of the structure (memory allocated for the structure variable in bytes) | employee emp1; sizeof (emp1); |

> **Note:**
> 1. The assignment and comparison operation is valid only if both structure variables are of the same type.
> 2. Some compilers may not support the direct assignment and comparison operation. In such situation the individual members of structure should be assigned or compared separately.

**Structure pointers**   Structure pointers are pointers to structures. It is easy to modify the memory held by structure variables if pointers are used. Functions with structure variable as parameter is a very good example for it. The structure variable can be passed to the function by two methods; either as a copy of the structure variable (pass by value) or as a pointer to a structure variable (pass by reference/address). Pass by value method is slower in operation compared to pass by pointers since the execution time for copying the structure parameter also needs to be accounted. Pass by pointers is also helpful if the structure which is given as input parameter to a function need to be modified and returned on execution of the calling function. Structure pointers can be declared by prefixing the structure variable name with '*'. The following structure pointer declaration illustrates the same.

```
struct employee *emp1;    //structure defined using the structure tag
employee *emp1;           //structure defined with typedef structure
```

For structure variables declared as pointers to a structure, the member selection of the structure is performed using the '→' operator.

```
E.g.    struct employee *emp1, emp2;
        emp1 = &emp2; //Obtain a pointer
        emp1→ emp_code = 42170;
        strcpy (emp1→DOB, "11-11-1977");
```

**Structure pointers at absolute address**   Most of the time structures are used in Embedded C applications for representing the memory locations or registers of chip whose memory address is fixed at the time of hardware designing. Typical example is a Real Time Clock (RTC) which is memory mapped at a specific address and the memory address of the registers of the RTC is also fixed. If we use a structure to define the different registers of the RTC, the structure should be placed at an absolute address corresponding to the memory mapped address of the first register given as the member variable of the structure. Consider the structure describing RTC registers.

```
typedef struct
{
//RTC Control register (8bit) memory mapped at 0x4000
unsigned char control;
//RTC Seconds register (8bit) memory mapped at 0x4001
unsigned char seconds;
```

```
//RTC Minutes register (8bit) memory mapped at 0x4002
unsigned char minutes;
//RTC Hours register (8bit) memory mapped at 0x4003
unsigned char hours;
//RTC Day of Week register (8bit) memory mapped at 0x4004
unsigned char day;
//RTC Date register (8bit) memory mapped at 0x4005
unsigned char date;
//RTC Month register (8bit) memory mapped at 0x4006
unsigned char month;
//RTC Year register (8bit) memory mapped at 0x4007
unsigned char year;
} RTC;
```

To read and write to these hardware registers using structure member variable manipulation, we need to place the RTC structure variable at the absolute address 0x4000, which is the address of the RTC hardware register, represented by the first member of structure RTC.

The implementation of structures using pointers to absolute memory location is cross-compiler dependent. Desktop applications may not give permission to explicitly place structures or pointers at an absolute address since we are not sure about the address allocated to general purpose RAM by the linker. Any attempt to explicitly allocate a structure at an absolute address may result in *access violation exception*. More over there is no need for a general purpose application to explicitly place structure or pointers at an absolute address, whereas embedded systems most often requires it if different hardware registers are memory mapped to the processor/controller memory map. The C51 cross compiler for 8051 microcontroller supports a specific Embedded C keyword '*xdata*' for external memory access and the structure absolute memory placement can be done using the following method.

```
RTC xdata *rtc_registers = (void xdata *) 0x4000;
```

**Structure Arrays**    In the above employee record example, three important data is associated with each employee, namely; employee name (emp_name), employee code (emp_code) and Date of Birth (DOB). This information is held together as a structure for associating the same with an employee. Suppose the organisation contains 100 employees then we need 100 such structures to hold the data related to the 100 employees. This is achieved by array of structures. For the above employee structure example a structure array with 100 structure variables can be declared as

```
struct employee emp [100]; //structure declared using struct keyword
        or
employee emp [100]; //structure declared using typedef struct
```

emp [0] holds the structure variable data for the first employee and emp [99] holds the structure variable data corresponding to the 100th employee. The variables corresponding to each structure in an array can be initialised altogether at the time of defining the structure array or can be initialised/modified using the corresponding array subscript for the structure and the '.' Operator as explained below.

```
typedef struct
        {
        char emp_name [20]; // Allowed maximum length for name = 20
        int emp_code;
```

```
        char DOB [10]; // DD-MM-YYYY Format (10 character)
        } employee;

//Initialisation at the time of defining variable
employee emp [3] = {{"JOHN", 1, "01-01-1988"}, {"ALEX", 2, "10-01-1976"},
{"SMITH", 3, "07-05-1985"}};
//Selective initialization
emp [0].emp_code = 1;
strcpy (emp [0].emp_name, "JOHN");
strcpy (emp [0].DOB, "01-01-1988");
```

**Structure in Embedded 'C' programming**    Simple structure variables and array of structures are widely used in 'embedded 'C' based firmware development. Structures and structure arrays are used for holding various configuration data, exchanging information between Interrupt Service Routine (ISR) and application etc. A typical example for the structure usage is for holding the various register configuration data for setting up a particular baudrate for serial communication. An array of such configuration data holding structures can be used for setting different baudrates according to user needs. It is interesting to note that more than 90% of the embedded C programmers use '*typedef*' to define the structures with meaningful names.

**Structure padding (Packed structure)**    Structure variables are always stored in the memory of the target system with structure member variables in the same order as they are declared in the structure definition. But it is not necessary that the variables should be placed in continuous physical memory locations. The choice on this is left to the compiler/cross-compiler. For multi byte processors (processors with word length greater than 1 byte (8 bits)), if the structure elements are arranged in memory in such a way that they can be accessed with lesser number of memory fetches, it definitely speeds up the operation. The process of arranging the structure elements in memory in a way facilitating increased execution speed is called *structure padding*. As an example consider the following structure:

```
    typedef struct
            {
            char x;
            int y;
            } exmpl;
```

Let us assume that the storage space for '*int*' is 4 bytes (32 bits) and for '*char*' it is 1 byte (8 bits) for the target embedded system under consideration. Suppose the target processor for embedded application, where the above structure is making use is with a 4 byte (32 bit) data bus and the memory is byte accessible. Every memory fetch operation retrieves four bytes from the memory locations $4x$, $4x + 1$, $4x + 2$ and $4x + 3$, where $x = 0, 1, 2$, etc. for successive memory read operations. Hence a 4 byte (32 bit) variable can be fully retrieved in a single memory fetch if it is stored at memory locations with starting address $4x$ ($x = 0, 1, 2$, etc.). If it is stored at any other memory location, two memory fetches are required to retrieve the variable and hence the speed of operation is reduced by a factor of 2.

Let us analyse the various possibilities of storing the above structure within the memory.

Method-1 member variables of structure stored in consecutive data memory locations.

In this model the member variables are stored in consecutive data memory locations (*Note:* the member variable storage need not start at the address mentioned here, the address given here is only

| Memory Address | 4x + 3 | 4x + 2 | 4x + 1 | 4x |
|---|---|---|---|---|
| Data | Byte 2 of exmpl.y | Byte 1 of exmpl.y | Byte 0 of exmpl.y | exmpl.x |
| Data | | | | Byte 3 of exmpl.y |
| Memory Address | 4(x + 1) + 3 | 4(x + 1) + 2 | 4(x + 1) + 1 | 4(x + 1) |

**Fig. 9.8** **Memory representation for structure without padding**

for illustration) and if we want to access the character variable exmpl.x of structure exmpl, it can be achieved with a single memory fetch. But accessing the integer member variable exmpl.y requires two memory fetches.

Method-2 member variables of structure stored in data memory with padding

In this approach, a structure variable with storage size same as that of the word length (or an integer multiple of word length) of the processor is always placed at memory locations with starting address as multiple of the word length so that the variable can be retrieved with a single data memory fetch. The memory locations coming in between the first variable and the second variable of the structure are filled with extra bytes by the compiler and these bytes are called 'padding bytes' (Fig. 9.9). The structure padding technique is solely dependent on the cross-compiler in use. You can turn ON or OFF the padding of structure by using the cross-compiler supported padding settings. Structure padding is applicable only for processors with word size more than 8bit (1 byte). It is not applicable to processors/controllers with 8bit bus architecture.

| Memory Address | 4x + 3 | 4x + 2 | 4x + 1 | 4x |
|---|---|---|---|---|
| Data | Padding | Padding | Padding | exmpl.x |
| Data | Byte 3 of exmpl.y | Byte 2 of exmpl.y | Byte 1 of exmpl.y | Byte 0 of exmpl.y |
| Memory Address | 4(x + 1) + 3 | 4(x + 1) + 2 | 4(x + 1) + 1 | 4(x + 1) |

**Fig. 9.9** **Memory representation for structure with padding**

***Structure and Bit fields*** Bit field is a useful feature supported by structures for bit manipulation operation and flag settings in embedded applications. Most of the processors/controllers used in embedded application development provides extensive Boolean (bit) operation support and a similar support in the firmware environment can directly be used to explore the Boolean processing capability of the target device.

For most of the application development scenarios we use integer and character to hold data items even though the variable is expected to vary in a range far below the upper limit of the data type used for holding the variable. A typical example is flags. Flags are used for indicating a 'TRUE' or 'FALSE' condition. Practically this can be done using a single bit and the two states of the bit (1 and 0) can be used for representing 'TRUE' or 'FALSE' condition. Unfortunately 'C' does not support a built in data type for holding a single bit and it forces us to use the minimum sized data type (quite often char (8bit data type) and short int) for representing the flag. This will definitely lead to the wastage of memory. Since memory is a big constraint in embedded applications we cannot tolerate the memory wastage. 'C' indirectly supports bit data types through '***Bit fields***' of structures. Structure bit fields can be directly accessed and manipulated. A set of bits in the structure bit field forms a '*char*' or '*int*' variable. The general format of declaration of bit fields within structure is illustrated below.

```
struct  struct_name
        {
        data type (char or int) bit_var_1_name : bit_size,
        bit_var_2_name : bit_size,
        .....................:,
        .....................:,
        bit_var_n_name : bit_size;
        };
```

'*struct_name*' represents the name of the bit field structure. '*data type*' is the data type which will be formed by packing several bits. Only character (char) and integer (int/short int) data types are allowed in bit field structures in Embedded C applications. Some compilers may not support the 'char' data type. However our illustrative cross-compiler C51 supports 'char' data type as data type. '*bit_var_1_name*' denotes the bit variable and '*bit_size*' gives the number of bits required by the variable '*bit_var_1_name*' and so. The operator ':' associates the number of bits required with the bit variable. Bit variables that are packed to form a data type should be separated inside the structure using ',' operator. A real picture of bit fields in structures for embedded applications can be provided by the Program Status Word (PSW) register representation of *8051* controller. The PSW register of 8051 is bit addressable and its bit level representation is given below.

| PSW.7 | PSW.6 | PSW.5 | PSW.4 | PSW.3 | PSW.2 | PSW.1 | PSW.0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| CY | AC | F0 | RS1 | RS0 | OV |  | P |

Using structure and bit fields the same can be represented as

*struct PSW*
```
        {
char            P:1,     /* Bit 0 of PSW  : Parity Flag */
                :1,      /* Bit 1 of PSW  : Unused */
                OV:1,    /* Bit 2 of PSW  : Overflow Flag */
                RS0:1,   /* Bit 3 of PSW  : Register Bank Select 0 bit */
                RS1:1,   /* Bit 4 of PSW  : Register Bank Select 1 bit */
                F0:1,    /* Bit 5 of PSW  : User definable Flag */
                AC:1,    /* Bit 6 of PSW  : Auxiliary Carry Flag */
                C:1;     /* Bit 7 of PSW  : Carry Flag */
```

*/\*Note that the operator ';' is used after the last bit field to
indicate end of bit field\*/*

```
};
```

In the above structure declaration, each bit field variable is defined with a name and an associated bit size representation. If some of the bits are unused in a packed fashion, the same can be skipped by merely giving the number of bytes to be skipped without giving a name for that bit variables. In the above example Bit 1 of PSW is skipped since it is an unused bit in the PSW register.

It should be noted that the total number of bits used by the bit field variables defined for a specific data type should not exceed the maximum number of allocated bits for that specific data type. In the above example the bit field data type is '*char*' (8 bits) and 7 bit field variables each of size 1 are declared and one bit variable is declared as unused bit. Hence the total number of bits consumed by all the bit variables including the non declared bit variable sums to 8, which is same as the bit size for the data type '*char*'. The internal representation of structure bit fields depends on the size supported by the cross-compiler data type (char/int) and the ordering of bits in memory. Some processors/controllers store the bits from left to right while others store from right to left (Here comes the significance of endianness).

***Unions*** Union is a concept derived from structure and *union* declarations follow the same syntax as that of structures (*structure* tag is replaced by *union* tag). Though *union* looks similar to structure in declaration, it differs from structure in the memory allocation technique for the member variables. Whenever a *union* variable is created, memory is allocated only to the member variable of *union* requiring the maximum storage size. But for structures, memory is allocated to each member variables. This helps in efficient data memory usage. Even if a *union* variable contains different member variables of different data types, existence is only for a single variable at a time. The size of a *union* returns the storage size of its member variable occupying the maximum storage size. The syntax for declaring *union* is given below

```
union    union_name
    {
        //variable 1 declaration
        //variable 2 to declaration
        //.............
        //variable n declaration
    };
```

or

```
typedef union
    {
        //variable 1 declaration
        //variable 2 to declaration
        //.............
        //variable n declaration
    } union_name;
```

'*union_name*' is the name of the *union* and programmers can use any name according to their programming style. As an illustrative example let's declare a *union* variable consisting of an integer member variable and a character member variable.

```
typedef union
{
    int y;      //Integer variable
    char z;     //Character variable
} example;
example ex1;
```

Assuming the storage location required for '*int*' as 4 bytes and for '*char*' as 1 byte, the memory allocated to the *union* variable ex1 will be as shown below

| Memory Address | 4x + 3 | 4x + 2 | 4x + 1 | ex1.z 4x |
|---|---|---|---|---|
| Data | Byte 3 of ex1.y | Byte 2 of ex1.y | Byte 1 of ex1.y | Byte 0 of ex1.y |

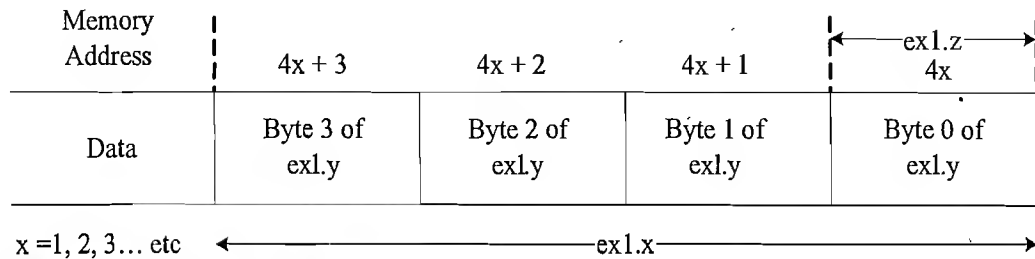x =1, 2, 3... etc ←————————————— ex1.x —————————————→

**Fig. 9.10** **Memory representation for union**

*Note:* The start address is chosen arbitrarily for illustration, it can be any data memory. It is obvious from the figure that by using *union* the same physical address can be accessed with different data type references. Hence *union* is a convenient way of '*variant access*'.

In Embedded C applications, *union* may be used for fast accessing of individual bytes of 'long' or 'int' variables, eliminating the need for masking the other bytes of 'long' or 'int' variables which are of no interest, for checking some conditions. Typical example is extracting the individual bytes of 16 or 32 bit counters.

### A few important points on Structure and Union

1. The *offsetof()* macro returns the offset of a member variable (in bytes) from the beginning of its parent structure. The usage is *offsetof (structName, memberName);* where '*structName*' is the name of the parent structure and '*memberName*' is the name of the member in the parent data structure whose offset is to be determined. For using this macro use the header file '*stddef.h*'

2. If you declare a *structure* just before the *main ()* function in your source file, ensure that the structure is terminated with the structure definition termination indicator ';'. Otherwise function *main ()* will be treated as a structure and the application may crash with exceptions.

3. A *union* variable can be initialised at the time of creation with the first member variable value only.

### 9.3.3.14 Pre-processors and Macros

Pre-processor in 'C' is compiler/cross-compiler directives used by compiler/cross-compiler to filter the source code before compilation/cross-compilation. The pre-processor directives are mere directions to the compilers/cross compilers on how the source file should be compiled/cross compiled. No executable code is generated for pre-processor directives on compilation. They are same as pseudo ops in assembly language. Pre-processors are very useful for selecting target processor/controller dependent pieces of code for different target systems and allow a single source code to be compiled and run on several different target boards. The syntax for pre-processor directives is different from the syntax of 'C' language. Each pre-processor directive starts with the

'#' symbol and ends without a semicolon (;). Pre-processor directives are normally placed before the entry point function *main()* in a source file. Pre-processor directives are grouped into three categories; namely

1. File inclusion pre-processor directives
2. Compile control pre-processor directives
3. Macro substitution pre-processor directives

**File inclusion pre processor directives**   The file inclusion pre processor directives include external files containing macro definitions, function declarations, constant definitions etc to the current source file. '*#include*' is the pre processor directive used for file inclusion. '*#include*' pre processor instruction reads the entire contents of the file specified by the '*#include*' directive and inserts it to the current source file at a location where the '*#include*' statement is invoked. This is generally used for reading header files for library functions and user defined header files or source files. Header files contain details of functions and types used within the library and user created files. They must be included before the program uses the library functions and other user defined functions. Library header file names are always enclosed in angle brackets, < >. This instructs the pre-processor to look the standard include directory for the header file, which needs to be inserted to the current source file.

e.g. *#include <stdio.h>* is the directive to insert the standard library file '*stdio.h*'. This file is available in the standard include directory of the development environment. (Normally inside the folder '*inc*'). If the file to be included is given in double quotes (" "), the pre-processor searches the file first in the current directory where the current source code file is present and if not found will search the standard include directory. Usually user defined files kept in the project folder are included using this technique. E.g. *#include "constants.h"* where 'constants' is a user defined header file with name *constants.h*, kept in the local project folder. An include file can include another include file in it (Nesting of include files). An include file is not allowed to include the same file itself. Source files (.c) file can also be used as include files.

**Compile control pre-processor directives**   Compile control pre-processor directives are used for controlling the compilation process such as skipping compilation of a portion of code, adding debug features, etc. The conditional control pre-processor directives are similar to the conditional statement if else in 'C'. *#ifdef, #ifndef, #else, #endif, #undef*, etc are the compile control pre-processor directives.

*#ifdef* uses a name as argument and returns true if the argument (name) is already defined. *#define* is used for defining the argument (name).
*#else* is used for two way branching when combined with *#ifdef* (same as *if else* in 'C').
*#endif* is used for indicating the end of a block following *#ifdef* or *#else*

Usage of *#ifdef*, *#else* and *#endif* is given below.

```
#ifdef

#else   (optional)

#endif
```

The pre-processor directive *#ifndef* is complementary to *#ifdef*. It is used for checking whether an argument (e.g. macro) is not defined. Pre-processor directive *#undef* is used for disabling the definition of the argument or macro if it is defined. It is complementary to *#define*. Pre-processor directives are a powerful option in source code debugging. If you want to ensure the execution of a code block, for

debug purpose you can define a debug variable and define it. Within the code wherever you want to ensure the execution, use the *#ifdef* and *#endif* pre-processors. The argument to *#ifdef* should be the debug variable. Insert a *printf()* function within the *#ifdef #endif* block. If no debugging is required comment or remove the definition of debug variable.

E.g.

```
#define DEBUG 1
//Inside code block
#ifdef DEBUG
printf ("Debug Enabled");
#endif
```

**The #error pre-processor directive**   The *#error* pre-processor generates error message in case of an error and stops the compilation on accounting an error condition. The syntax of *#error* directive is given below

```
        #error   error message
```

**Macro substitution pre-processor directives**   *Macros* are a means of creating portable inline code. '*Inline*' means wherever the macro is called in a source file it is replaced directly with the code defined by the macro. In-line code improves the performance in terms of execution speed. Macros are similar to subroutines in functioning but they differ in the way in which they are coded in the source code. Functions are normally written only once with arguments. Whenever a function needs to be executed, a call to the function code is made with the required parameters at the point where it needs to be invoked. If a macro is used instead of functions, the compiler inserts the code for macro wherever it is called. From a code size viewpoint macros are non-optimised compared to functions. Macros are generally used for coding small functions requiring execution without latency. The '*#define*' pre-processor directive is used for coding macros. Macros can be either simple definitions or functions with arguments.

*#define   PI  3.1415* is an example for a simple macro definition.

Macro definition can contain arithmetic operations also. Proper care should be taken in giving right syntax while defining macros, keeping an eye on their usage in the source code. Consider the following example

```
        #define A 12+25
        #define B 45-10
```

Suppose the source contains a statement *multiplier = A\*B;* the pre-processor directly replaces the macros A and B and the statement becomes

```
        multiplier = 12+25*45-10;
```

Due to the operator precedence criteria, this won't give the expected result. Expected result can be obtained by a simple re-writing of the macro with necessary parentheses as illustrated below.

```
        #define A (12+25)
        #define B (45-10)
```

Proper care should be given to parentheses the macro arguments. As mentioned earlier macros can also be defined with arguments. Consider the following example

```
#define CIRCLE_AREA(a) (3.14 * a*a)
```

This defines a macro for calculating the area of a circle. It takes an argument and returns the area. It should be noted that there is no space between the name of the macro (macro identifier) and the left bracket parenthesis.

Suppose the source code contains a statement like *area=CIRCLE_AREA(5);* it will be replaced as

```
area = (3.14*5*5);
```

Suppose the call is like this, *area=CIRCLE_AREA(2+5);* the pre-processor will translate the same as

```
area = (3.14*2+5*2+5);
```

Will it produce the expected result? Obviously no. This shortcoming in macro definition can be eliminated by using parenthesis to each occurrence of the argument. Hence the ideal solution will be;

```
#define CIRCLE_AREA(a) (3.14 * (a)*(a))
```

*9.3.3.15 Constant Declarations in Embedded 'C'*    In embedded applications the qualifier (keyword) *'const'* represents a 'Read only' variable. Use of the keyword *'const'* in front of a variable declares that the value of the variable cannot be altered by the program. This is a kind of defensive programming in which any attempt to modify a variable declared as *'const'* is reported as an access violation by the cross-compiler. The different types of constant variables used in embedded 'C' application are explained below.

**Constant data**    Constant data informs that the data held by a variable is always constant and cannot be modified by the application. Constants used as scaling variables, ratio factors, various scientific computing constants (e.g. Plank's constant), etc. are represented as constant data. The general form of declaring a constant variable is given below.

```
const  data type  variable name;
```
            or
```
data type  const  variable name;
```

*'const'* is the keyword informing compiler/cross compiler that the variable is constant. *'data type'* gives the data type of the variable. It can be 'int', 'char', 'float', etc. *'variable name'* is the constant variable.

```
E.g.   const float PI = 3.1417;
       float const PI = 3.1417;
```

Both of these statements declare PI as floating point constant and assign the value 3.1417 to it. Constant variable can also be defined using the *#define* pre-processor directive as given below.

```
#define PI 3.1417
/*No assignment using = operator and no ';' at end*/
```

The difference between both approaches is that the first one defines a constant of a particular data type (int, char, float, etc.) whereas in the second method the constant is represented using a mere symbol (text) and there is no implicit declaration about the data type of the constant. Both approaches are used in declaring constants in embedded C applications. The choice is left to the programmer.

***Pointer to constant data***    Pointer to constant data is a pointer which points to a data which is read only. The pointer pointing to the data can be changed but the data is non-modifiable. Example of pointer to constant data

```
const int* x;      //Integer pointer x to constant data
int const* x       //Same meaning as above definition
```

***Constant pointer to data***    Constant pointer has significant importance in Embedded C applications. An embedded system under consideration may have various external chips like, data memory, Real Time Clock (RTC), etc interfaced to the target processor/controller, using memory mapped technique. The range of address assigned to each chips and their registers are dependent on the hardware design. At the time of designing the hardware itself, address ranges are allocated to the different chips and the target hardware is developed according to these address allocations. For example, assume we have an RTC chip which is memory mapped at address range 0x3000 to 0x3010 and the memory mapped address of register holding the time information is at 0x3007. This memory address is fixed and to get the time information we have to look at the register residing at location 0x3007. But the content of the register located at address 0x3007 is subject to change according to the change in time. We can access this data using a constant pointer. The declaration of a constant pointer is given below.

```
// Constant character pointer x to constant/variable data
char *const x;

/*Explicit declaration of character pointer pointing to 8bit memory location,
mapped at location 0x3007; RTC example illustrated above*/

char *const x= (char*) 0x3007;
```

***Constant pointer to constant data***    Constant pointers pointing to constant data are widely used in embedded programming applications. Typical uses are reading configuration data held at ROM chips which are memory mapped at a specified address range, Read only status registers of different chips, memory mapped at a fixed address. Syntax of declaring a constant pointer to constant data is given below.

```
/*Constant character pointer x pointing to constant data*/
const char *const x;
char const* const x;    //Equivalent to above declaration

/*Explicit declaration of constant character pointer* pointing to constant
data/
char const* const x = (char*) 0x3007;
```

***9.3.3.16 The 'Volatile' Type Qualifier in Embedded 'C'***    The keyword *'volatile'* prefixed with any variable as qualifier informs the cross-compiler that the value of the variable is subject to change at any point of time (subject to asynchronous modification) other than the current statement of code where the control is at present.

Examples of variables which are subject to asynchronous modifications are

1. Variables common to Interrupt Service Routines (ISR) and other functions of a file
2. Memory mapped hardware registers

3. Variables shared by different threads in a multi threaded application (Not applicable to Super loop Firmware development approach)

The '*volatile*' keyword informs the cross-compiler that the variable with '*volatile*' qualifier is subject to asynchronous change and there by the cross compiler turns off any optimisation (assumptions on the variable) for these variables. The general form of declaring a volatile variable is given below.

```
volatile data type variable name; or
data type volatile variable name;
```

'data type' refers to the data type of the variable. It can be *int, char, float,* etc. 'variable name' is the user defined name for the *volatile* variable of the specified type.

```
E.g.      volatile unsigned char x;
          unsigned char volatile x;
```

**What is the catch in using 'volatile' variable?**    Let's examine the following code snippet.

```
//Declare a memory mapped register
char* status_reg = (char*) 0x3000;

while (*status_reg!=0x01);    //Wait till status_reg = 0x01
```

On cross-compiling the code snippet, the cross-compiler converts the code to a read operation from the memory location mapped at address 0x3000 and it will assume that there is no point where the variable is going to modify (sort of over smartness) and may keep the data in a register to speed up the execution. The actual intention of the programmer, with the above code snippet is to read a memory mapped hardware status register and halt the execution of the rest of the code till the status register shows a ready status. Unfortunately the program will not produce the expected result due to the oversmartness of the cross-compiler in optimising the code for execution speed. Re-writing the code as given below serves the intended purpose.

```
//Declares volatile variable
volatile char *status_reg = (char *) 0x3000;

while (*status_reg!=0x01);    //Wait till status_reg = 0x01
```

In embedded applications all memory mapped device registers which are subject to asynchronous modifications (e.g. status, control and general purpose registers of memory mapped external devices) should be declared with '*volatile*' keyword to inform the cross-compiler that the variables representing these registers/locations are subject to asynchronous changes and do not optimise them. Another area which needs utmost care in embedded applications is variables shared between ISR and functions (variables which can be modified by both Interrupt Sub Routines and functions). These include structure variable, union variable and other ordinary variables. To avoid unexpected behaviour of the application, always declare such variables using '*volatile*' keyword.

**The 'constant volatile' Variable**    Some variables used in embedded applications can be both '*constant*' and '*volatile*'. A '***Read only***' status register of a memory mapped device is a typical example for this. From a user point of view the '***Read only***' status registers can only be read but cannot modify. Hence it is a constant variable. From the device point the contents can be modified at any time by the device. So it is a volatile variable. Typical declarations are given ahead.

```
volatile const int a;      // Constant volatile integer
volatile const int* a;     //Pointer to a Constant volatile integer
```

**Volatile pointer**   Volatile pointers is subject to change at any point after they are initialised. Typical examples are pointer to arrays or buffers modifiable by Interrupt Service Routines and pointers in dynamic memory allocation. Pointers used in dynamic memory allocation can be modified by the *realloc()* function. The general form of declaration of a volatile pointer to a non-volatile variable is given below.

```
data type* volatile variable name;
e.g.   unsigned char* volatile a;
```

Volatile pointer to a volatile variable is declared with the following syntax.

```
data type volatile* volatile variable name;
e.g.   unsigned char volatile* volatile a;
```

*9.3.3.17 Delay Generation and Infinite Loops in Embedded C*   Almost every embedded application involves delay programming. Embedded applications employ delay programming for waiting for a fixed time interval till a device is ready, for inserting delay between displays updating to give the user sufficient time to view the contents displayed, delays involved in bit transmission and reception in asynchronous serial transmissions like I2C, 1-Wire data transfer, delay for key de-bouncing etc. Some delay requirements in embedded application may be critical, meaning delay accuracy should be within a very narrow tolerance band. Typical example is delay used in bit data transmission. If the delay employed is not accurate, the bits may lost while transmission or reception. Certain delay requirements in embedded application may not be much critical, e.g. display updating delay.

It is easy to code delays in desktop applications under DOS or Windows operating systems. The library function *delay()* in DOS and *Sleep ()* in Windows provides delays in milliseconds with reasonable accuracy. Coding delay routines in embedded applications is bit difficult. The major reason is delay is dependent on target system's clock frequency. So we need to have a trial and error approach to code delays demanding reasonably good accuracy. Refer to the code snippet given for 'Performance Analyser' in *Chapter 14* for getting a handle on how to code delay routine in embedded applications using IDEs. Delay codes are generally non-portable. Delay routine requires a complete re-work if the target clock frequency is changed. Normally '*for loops*' are used for coding delays. Infinite loops are created using various loop control instructions like while (), do while (), for and goto labels. The super loop created by while (1) instruction in a traditional super loop based embedded firmware design is a typical example for infinite loop in embedded application development.

**Infinite loop using while**   The following code snippet illustrates '*while*' for infinite loop implementation.

```
while (1)
{
}
```

**Infinite loop using do while**

```
do
{
} while (1);
```

### Infinite loop using for

```
for (; ; ;)
{

}
```

### Infinite loop using goto
'*goto*' when combined with a '*label*' can create infinite loops.

```
label: //Task to be repeated
       //................
       //................
       goto label;
```

**Which technique is the best?** According to all experienced Embedded 'C' programmers *while()* loop is the best choice for creating infinite loops. There is no technical reason for this. The clean syntax of while loop entitles it for the same. The syntax of *for* loop for infinite loop is little puzzling and it is not capable of conveying its intended use. '*goto*' is the favorite choice of programmers migrating from Assembly to Embedded C ☺.

*break;* statement is used for coming out of an infinite loop. You may think why we implement an infinite loop and then quitting it? Answer – There may be instructions checking some condition inside the infinite loop. If the condition is met the program control may have to transfer to some other location.

### 9.3.3.18 Bit Manipulation Operations
Though Embedded 'C' does not support a built in Boolean variable (Bit variable) for holding a 'TRUE (Logic 1)' or 'FALSE (Logic 0)' condition, it provides extensive support for Bit manipulation operations. Boolean variables required in embedded application are quite often stored as variables with least storage memory requirement (obviously *char* variable). Indeed it is wastage of memory if the application contains large number of Boolean variables and each variable is stored as a *char* variable. Only one bit (Least Significant bit) in a *char* variable is used for storing Boolean information. Rest 7 bits are left unused. This will definitely lead to serious memory bottle neck. Considerable amount of memory can be saved if different Boolean variables in an application are packed into a single variable in 'C' which requires less memory storage bytes. A character variable can accommodate 8 Boolean variables. If the Boolean variables are packed for saving memory, depending upon the program requirement each variable may have to be extracted and some manipulation (setting, clearing, inverting, etc.) needs to be performed on the bits. The following Bit manipulation operations are employed for the same.

**Bitwise AND** Operator '&' performs Bitwise AND operations. Please note that the Bitwise AND operator '&' is entirely different from the Logical AND operator '&&'. The '&' operator acts on individual bits of the operands. Bitwise AND operations are usually performed for selective clearing of bits and testing the present state of a bit (Bitwise ANDing with '1').

**Bitwise OR** Operator '|' performs Bitwise OR operations. Logical OR operator '||' is in no way related to the Bitwise OR operator '|'. Bitwise OR operation is performed on individual bits of the operands. Bitwise OR operation is usually performed for selectively setting of bits and testing the current state of a bit (Bitwise ORing with '0').

**Bitwise Exclusive OR- XOR** Bitwise XOR operator '^' acts on individual operand bits and performs an 'Excusive OR' operation on the bits. Bitwise XOR operation is used for toggling bits in embedded applications.

***Bitwise NOT***   Bitwise NOT operations negates (inverts) the state of a bit. The operator '~' (tilde) is used as the Bitwise NOT operator in C.

***Setting and Clearing and Bits***   Setting the value of a bit to '1' is achieved by a Bitwise OR operation. For example consider a character variable (8bit variable) flag. The following instruction sets its $0^{th}$ bit always 1.

```
flag = flag | 1;
```

Brief explanation about the above operation is given below.

Using 8 bits, 1 is represented as 00000001. Upon a Bitwise OR each bit is ORed with the corresponding bit of the operand as illustrated below.

> Bit 0 of flag is ORed with 1 and Resulting o/p bit=1
> Bit 1 of flag is ORed with 0 and Resulting o/p bit= Bit 1 of flag
> Bit 2 of flag is ORed with 0 and Resulting o/p bit= Bit 2 of flag
> Bit 3 of flag is ORed with 0 and Resulting o/p bit= Bit 3 of flag
> Bit 4 of flag is ORed with 0 and Resulting o/p bit= Bit 4 of flag
> Bit 5 of flag is ORed with 0 and Resulting o/p bit= Bit 5 of flag
> Bit 6 of flag is ORed with 0 and Resulting o/p bit= Bit 6 of flag
> Bit 7 of flag is ORed with 0 and Resulting o/p bit= Bit 7 of flag

Bitwise OR operation combined with left shift operation of '1' is used for selectively setting any bit in a variable. For example the following operation will set bit 6 of *char* variable flag.

```
//Sets 6th bit of flag. Bit numbering starts with 0.
flag = flag | (1<<6);
```

Re-writing the above code for a neat syntax will give

```
flag |= (1<<6);      //Equivalent to flag = flag | (1<<6);
```

The same can also be achieved by bitwise ORing the variable flag with a mask with $6^{th}$ bit '1' and all other bits '0', i.e. mask with 01000000 in Binary representation and 0x40 in hex representation.

```
flag |= 0x40;   //Equivalent to flag = flag | (1<<6);
```

Clearing a desired bit is achieved by Bitwise ANDing the bit with '0'. Bitwise AND operation combined with left shifting of '1' can be used for clearing any desired bit in a variable.

```
Example:
        flag &= ~ (1<<6);
```

The above instruction will clear the $6^{th}$ bit of the character variable *flag*. The operation is illustrated below.

Execution of (1<<6) shifts '1' to six positions left and the resulting output in binary will be 01000000. Inverting this using the Bitwise NOT operation (~ (1<<6)) inverts the bits and give 10111111 as output. When *flag* is Bitwise ANDed with 10111111, the $6^{th}$ bit of *flag* is cleared (set to '0') and all other bits of *flag* remain unchanged.

From the above illustration it can be inferred that the same operation can also be achieved by a direct Bitwise ANDing of the variable *flag* and a mask with binary representation 10111111 or hex representation 0xBF.

```
flag &= 0xBF;        //Equivalent to flag = flag & ~(1<<6);
```

Shifting the mask '1' for setting or clearing a desired bit works perfectly, if the operand on which these operations are performed is 8bit wide. If the operand is greater than 8 bits in size, care should be taken to adjust the mask as wide as the operand. As an example let us assume *flag* as a 32bit operand. For clearing the 6th bit of *flag* as illustrated in the previous example, the mask 1 should be re-written as '1L' and the instruction becomes

```
flag &= ~ (1L<<6);
```

### Toggling Bits

Toggling a bit is performed to negate (toggle) the current state of a bit. If current state of a specified bit is '1', after toggling it becomes '0' and vice versa. Toggling is also known as inverting bits. The Bitwise XOR operator is used for toggling the state of a desired bit in an operand.

```
flag ^= (1<<6);        //Toggle bit 6 of flag
```

The above instruction toggles bit 6 of *flag*.

*Adding '1' to the desired bit position (In the above example 0x40 for 6th bit) will also toggle the current state of the desired bit. This approach has the following drawback. If the current state of the bit which is to be inverted is '1', adding a '1' to that bit position inverts the state of that bit and at the same time a carry is generated and it is propagated to the next most significant bit (MSB) and change the status of some of the other bits falling to the left of the bit which is toggled.*

### Extracting and Inserting Bits

Quite often it is meaningful to store related information as the bits of a single variable instead of saving them as separate variables. This saves considerable amount of memory in an embedded system where data memory is a big bottleneck. Typical scenario in embedded applications where information can be stored using the bits of single variable is, information provided by Real Time Clock (RTC). RTC chip provides various data like current date/month/year, day of the week, current time in hours/minutes/ seconds, etc. If an application demands the storage of all these information in the data memory of the embedded system for some reason, it can be achieved in two ways;

1. Use separate variables for storing each data (date, month, year, etc.)
2. Club the related data and store them as the bits of a single variable

As an example assume that we need to store the date information of the RTC in the data memory in D/M/Y format. Where 'D' stands for date varying from 1 to 31, 'M' stands for month varying from 1 to 12 and 'Y' stands for year varying from 0 to 99. If we follow the first approach we need 3 character variables to store the date information separately. In the second approach, we need only 5 bits for date (With 5 bits we can represent 0 to $2^5 - 1$ numbers (0 to 31), 4 bits for month and 7 bits for year. Hence the total number of bits required to represent the date information is 16. All these bits can be fitted into a 16 bit variable as shown in Fig. 9.11.
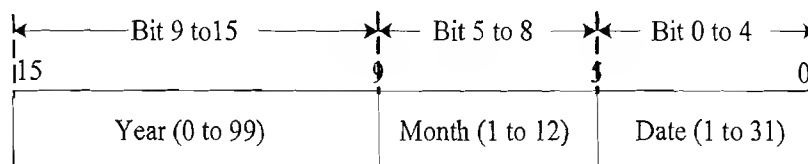
| ← Bit 9 to15 → | ← Bit 5 to 8 → | ← Bit 0 to 4 → |
|---|---|---|
| 15 | 9 | 0 |
| Year (0 to 99) | Month (1 to 12) | Date (1 to 31) |

**Fig. 9.11** Packed Bits for data representation

Suppose this is arranged in a 16bit integer variable date, for any calculation requiring 'Year', 'Month' or 'Date', each should be extracted from the single variable date. The following code snippet illustrates the extraction of 'Year'

```
char year = (date>>9) & 0x7F;
```

(*date*>>9) shifts the contents 9 bits to the left and now 'Year' is stored in the variable *date* in bits 0 to 6 (including both). The contents of other bits (7 to 15) are not relevant to us and we need only the first 7 bits (0 to 6) for getting the 'Year' value. ANDing with the mask 0x7F (01111111 in Binary) retains the contents of bits 0 to 6 and now the variable year contains the 'Year' value extracted from variable '*date*'.

Similar to the extracting of bits, we may have to insert bits to change the value of certain data. In the above example if the program is written in such a way that after each 1 minute, the RTC's year register is read and the contents of bit fields 9 to 15 in variable '*date*' needs to be updated with the new value. This can be achieved by inserting the bits corresponding to the data into the desired bit position. Following code snippet illustrates the updating of 'Year' value for the above example. To set the new 'Year' value to the variable '*date*' all the corresponding bits in the variable for 'Year' should be cleared first. It is illustrated below.

```
date = date & ~ (0x7F<<9);
```

The mask for 7 bits is 0x7F (01111111 in Binary). Shifting the mask 9 times left aligns the mask to that of the bits for storing 'Year'. The ~ operator inverts the 7 bits aligning to the bits corresponding to 'Year'. Bitwise ANDing with this negated mask clears the current bits for 'Year' in the variable '*date*'. Now shift the new value which needs to be put at the 'Year' location, 9 times for bitwise alignment for the bits corresponding to 'Year'. Bitwise OR the bit aligned new value with the '*date*' variable whose bits corresponding to 'Year' is already cleared. The following instruction performs this.

```
date |= (new_year <<9);
```

where 'new_year' is the new value for 'Year'.

In order to ensure the inserted bits are not exceeding the number of bits assigned for the particular bit variable, bitwise AND the new value with the mask corresponding to the number of bits allocated to the corresponding variable (Above example 7 bits for 'Year' and the mask is 0x7F). Hence the above instruction can be written more precisely as

```
date |= (new_year & 0x7F) <<9;
```

*If all the bits corresponding to the bit field to be modified are not set to zero prior to Bitwise ORing with the new value, any existing bit with value '1' will remain as such and expected result will not be obtained.*

**Testing Bits**    So far we discussed the Bitwise operators for changing the status of a selected bit. Bitwise operators can also be used for checking the present status of a bit without modifying it for decision making operations.

```
if (flag & (1<<6))    //Checks whether 6th bit of flag is '1'
```

This instruction examines the status of the 6th bit of variable *flag*. The same can also be achieved by using a constant bit mask as

```
if (flag & 0x40)    //Checks whether 6th bit of flag is '1'
```

### 9.3.3.19 Coding Interrupt Service Routines (ISR)

Interrupt is an event that stops the current execution of a process (task) in the CPU and transfers the program execution to an address in code memory where the service routine for the event is located. The event which stops the current execution can be an internal event or an external event or a proper combination of the external and internal event. Any trigger signal coming from an externally interfaced device demanding immediate attentions is an example for external event whereas the trigger indicating the overflow of an internal timer is an example for internal event. Reception of serial data through the serial line is a combination of internal and external events. The number of interrupts supported, their priority levels, interrupt trigger type and the structure of interrupts are processor/controller architecture dependent and it varies from processor to processor. Interrupts are generally classified into two: Maskable Interrupts and Non-maskable Interrupts (NMI). Maskable interrupt can be ignored by the CPU if the interrupt is internally not enabled or if the CPU is currently engaged in processing another interrupt which is at high priority. Non-maskable interrupts are interrupts which require urgent attention and cannot be ignored by the CPU. Reset (RST) interrupt and TRAP interrupt of *8085* processor are examples for Non-maskable interrupts.

Interrupts are considered as boon for programmers and their main motto is *"give real time behaviour to applications"*. In a processor/controller based simple embedded system where the application is designed in a super loop model, each interrupt supported by the processor/controller will have a fixed memory location assigned in the code memory for writing its corresponding service routine and this address is referred as *Interrupt Vector Address*. The vector address for each interrupt will be pre-defined and the number of code memory bytes allocated for each Interrupt Service Routine, starting from the Interrupt Vector Address may also be fixed. For example the interrupt vector address for interrupt 'INT0' of *8051* microcontroller is '0003H' and the number of code memory bytes allowed for writing its Service routine is 8 bytes.

The function written for serving an Interrupt is known as Interrupt Service Routine (ISR). ISR for each interrupt may be different and they are placed at the Interrupt Vector Address of corresponding Interrupt. ISR is essentially a function that takes no parameters and returns no results. But, unlike a regular function, the ISR can be active at any time since the triggering of interrupts need not be in sync with the internal program execution (e.g. An external device connected to the external interrupt line can assert external interrupt at any time regardless at what stage the program execution is currently). Hence special care must be taken in writing these functions keeping in mind; they are not going to be executed in a pre-defined order. What all special care should be taken in writing an ISR? The following section answers this query.

Imagine a situation where the application is doing some operations and some registers are modified and an interrupt is triggered in between the operation. Indeed the program flow is diverted to the Interrupt Service Routine, if the interrupt is an enabled interrupt and the operation in progress is not a service routine of a high priority interrupt, and the ISR is executed. After completing the ISR, the program flow is re-directed to the point where it got interrupted and the interrupted operation is continued. What happens if the ISR modifies some of the registers used by the program? No doubt the application will produce unexpected results and may go for a toss. How can this situation be tackled? Such a situation can be avoided if the ISR is coded in such a way that it takes care of the following:

1. Save the current context (Important Registers which the ISR will modify)
2. Service the Interrupt
3. Retrieve the saved context (Retrieve the original contents of registers)
4. Return to the execution point where the execution is interrupted

Normal functions will not incorporate code for saving the current context before executing the function and retrieve the saved context before exiting the function. ISR should incorporate code for perform-

ing these operations. Also it is known to the programmer at what point a normal function is going to be invoked and the programmer can take necessary precautions before calling the function, it is not the case for an ISR. Interrupt Service Routines can be coded either in Assembly language or High level language in which the application is written. Assembly language is the best choice if the ISR code is very small and the program itself is written in Assembly. Assembly code generates optimised code for ISR and user will have a good control on deciding what all registers needs to be preserved from alteration. Most of the modern cross-compilers provide extensive support for efficient and optimised ISR code. The way in which a function is declared as ISR and what all context is saved within the function is cross-compiler dependent.

Keil C51 Cross-complier for 8051 microcontroller implements the Interrupt Service Routine using the keyword *interrupt* and its syntax is illustrated below.

```
void interrupt_name (void)       interrupt x using y
{
/*Process Interrupt*/
}
```

*interrupt_name* is the function name and programmers can choose any name according to his/her taste. The attribute '*interrupt*' instructs the cross compiler that the associated function is an interrupt service routine. The *interrupt* attribute takes an argument $x$ which is an integer constant in the range 0 to 31 (supporting 32 interrupts). This number is the interrupt number and it is essential for placing the generated hex code corresponding to the ISR in the corresponding Interrupt Vector Address (e.g. For placing the ISR code at code memory location 0003H for Interrupt 0 – External Interrupt 0 for *8051* microcontroller). *using* is an optional keyword for indicating which register bank is used for the general purpose Registers R0 to R7 (For more details on register banks and general purpose registers, refer to the hardware description of *8051*). The argument $y$ for *using* attribute can take values from 0 to 3 (corresponding to the register banks 0 to 3 of 8051). The *interrupt* attribute affects the object code of the function in the following way:

1. If required, the contents of registers ACC, B, DPH, DPL, and PSW are saved on the stack at function invocation time.
2. All working registers (R0 to R7) used in the interrupt function are stored on the stack if a register bank is not specified with the *using* attribute.
3. The working registers and special registers that were saved on the stack are restored before exiting the function.
4. The function is terminated by the 8051 RETI instruction.

Typical usage is illustrated below

```
void external_interrupt0 (void )    interrupt 0 using 0
{
adc_control = *adc_control_reg //Read memory mapped ADC
                                // control Register
}
```

If the cross-compiler you are using don't have a built in support for writing ISRs, What shall you do? Don't be panic you can implement the ISR feature with little tricky coding. If the cross-compiler provides support for mixing high level language-C and Assembly, write the ISR in Assembly and place the ISR code at the corresponding Interrupt Vector address using the cross compiler's support for placing the code in an absolute address location of code memory (Using keywords like _at. Refer to your

cross compiler's documentation for getting the exact keyword). If the ISR is too complicated, you can place the body of the ISR processing in a normal C function and call it from a simple assembly language wrapper. The assembly language wrapper should be installed as the ISR function at the absolute address corresponding to the Interrupt's Vector Address. It is responsible for executing the current context saving and retrieving instructions. Current context saving instructions are written on top of the call to the 'C' function and context retrieving instructions are written just below the 'C' function call. It is little puzzling to find answers to the following questions in this approach.

1. Which registers must be saved and restored since we are not sure which registers are used by the cross compiler for implementing the 'C' function?
2. How the assembly instructions can be interfaced with high-level language like 'C'?

Answers to these questions are cross compiler dependent and you need to find the answer by referring the documentation files of the cross-compiler in use. Context saving is done by 'Pushing' the registers (using PUSH instructions) to stack and retrieving the saved context is done by 'Poping' (using POP instructions) the pushed registers. Push and Pop operations usually follow the Last In First Out (LIFO) method. While writing an ISR, always keep in mind that the primary aim of an interrupt is to provide real time behaviour to the program. Saving the current context delays the execution of the original ISR function and it creates Interrupt latency (Refer to the section on Interrupt Latency for more details) and thereby adds lack of real time behaviour to an application. As a programmer, if you are responsible for saving the current context by Pushing the registers, avoid Pushing the registers which are not used by the ISR. If you deliberately avoid the saving of registers which are going to be modified by the ISR, your application may go for a toss. Hence Context saving is an unavoidable evil in Interrupt driven programming. If you go for saving unused registers, it will increase interrupt latency as well as stack memory usage. So always take a judicious decision on the context saving operation. If the cross compiler offers the context saving operation by supporting ISR functions, always rely on it. Because most modern cross compilers are smart and capable of taking a judicious decision on context saving. In case the cross compiler is not supporting ISR function and as a programmer you are the one writing ISR functions either in Assembly or by mixing 'C' and Assembly, ensure that the size of ISR is not crossing the size of code memory bytes allowed for an Interrupt's ISR (8 bytes for *8051*). If it exceeds, it will overlap with the location for the next interrupt and may create unexpected behaviour on servicing the Interrupt whose Vector address got overlapped.

*9.3.3.20 Recursive Functions* A function which calls itself repeatedly is called a Recursive Function. Using recursion, a complex problem is split into its single simplest form. The recursive function only knows how to solve that simplest case. Recursive functions are useful in evaluating certain types of mathematical function, creating and accessing dynamic data structures such as linked lists or binary trees. As an example let us consider the factorial calculation of a number.

By mathematical definition

*n factorial* $= 1 \times 2 \times \ldots\ldots (n-2) \times (n-1) \times n$; where $n = 1, 2$, etc… and $0$ factorial $= 1$

Using 'C' the function for finding the factorial of a number '*n*' is written as

```
int factorial (int n)
{
int count;
int factorial=1;
for (count=1; count<=n; count++)
        factorial*=count;
```

```
        return factorial;
    }
```

This code is based on iteration. The iteration of calculating the repeated products is performed until the count value exceeds the value of the number whose factorial is to be calculated. We can split up the task performed inside the function as *count \* (count +1)* till count is one short of the number whose factorial is to be calculated. Using recursion we can re-write the above function as

```
int factorial (int n)
{
    if (n==0)
        return 1;
    else
        return (n*factorial (n-1));
}
```

Here the function *factorial (n)* calls itself, with a changed version of the parameter for each call, inside the same for calculating the factorial of a given number. The 'if' statement within the recursive function forces the function to stop recursion when a certain criterion is met. Without an 'if' statement a recursive function will never stop the recursion process. Recursive functions are very effective in implementing solutions expressed in terms of a successive application of the same solution to the solution subsets. Recursion is a powerful at the same time most dangerous feature which may lead to application crash. The local variables of a recursive function are stored in the stack memory and new copies of each variable are created for successive recursive calls of the function. As the recursion goes in a deep level, stack memory may overflow and the application may bounce.

***Recursion vs. Iteration: A comparison***    Both recursion and iteration is used for implementing certain operations which are self repetitive in some form.

- Recursion involves a lot of call stack overhead and function calls. Hence it is slower in operation compared to the iterative method. Since recursion implements the functionality with repeated self function calls, more stack memory is required for storing the local variables and the function return address
- Recursion is the best method for implementing certain operations like certain mathematical operation, creating and accessing of dynamic data structures such as linked lists or binary trees
- A recursive solution implementation can always be replaced by iteration. The process of converting a recursive function to iterative method is called '*unrolling*'

***Benefits of Recursion***    Recursion brings the following benefits in programming:

- Recursive code is very expressive compared to iterative code. It conveys the intended use.
- Recursion is the most appropriate method for certain operations like permutations, search trees, sorting, etc.

***Drawbacks of Recursion***    Though recursion is an effective technique in implementing solutions expressed in terms of a successive application of the same solution to the solution subsets, it possesses the following drawbacks.

- Recursive operation is slow in operation due to call stack overheads. It involves lot of stack operations like local variable storage and retrieval, return address storage and retrieval, etc.
- Debugging of recursive functions are not so easy compared to iterative code debugging

*9.3.3.21 Re-entrant Functions*    Functions which can be shared safely with several processes concurrently are called re-entrant functions. When a re-entrant function is executing, another process can interrupt the execution and can execute the same re-entrant function. The "*another process*" referred here can be a thread in a multithreaded application or can be an Interrupt Service Routine (ISR). Re-entrant function is also referred as '*pure*' function. Embedded applications extensively make use of re-entrant functions. Interrupt Service Routines (ISR) in Super loop based firmware systems and threads in RTOS based systems can change the program's control flow to alter the current context at any time. When an interrupt is asserted, the current operation is put on hold and the control is transferred to another function or task (ISR). Imagine a situation where an interrupt occurs while executing a *function x* and the ISR also contain the task of executing the *function x*. What will happen? - Obviously the ISR will modify the shared variables of *function x* and when the control is switched back to the point where the execution got interrupted, the function will resume its execution with the data which is modified by the ISR. What will be the outcome of this action? - Unpredictable result causing data corruption and potential disaster like application break-down. Why it happens so? Due to the corruption of shared data in function which is unprotected. How this situation can be avoided? By carefully controlling the sharing of data in the function.

In embedded applications, a function/subroutine is considered re-entrant if and only if it satisfies the following criteria.

1. The function should not hold static data over successive calls, or return a pointer to static data.
2. All shared variables within the function are used in an atomic way.
3. The function does not call any other non-reentrant functions within it.
4. The function does not make use of any hardware in a non-atomic way

Rule# 1 deals with variable usage and function return value for a reentrant function. In an operating system based embedded system, the embedded application is managed by the operating system services and the '*memory manager*' service of the OS kernel is responsible for allocating and de-allocating the memory required by an application for its execution. The working memory required by an application is divided into three groups namely; stack memory, heap memory and data memory (Refer to *the Dynamic Memory Allocation* section of this chapter for more details). The life time of static variables is same as that of the life time of the application to which it belongs and they are usually held in the data memory area of the memory space allocated for the application. All static variables of a function are allocated in the data memory area space of the application and each instance of the function shares this, whereas local (auto) variables of a function are stored in the stack memory and each invocation of the function creates independent copies of these variables in the stack memory. For a function to be reentrant, it should not keep any data over successive invocations (the function should not contain any static storage). If a function needs some data to be kept over successive invocations, it should be provided through the caller function instead of storing it in the function in the form of static variables. If the function returns a pointer to a static data, each invocation of the function makes use of this pointer for returning the result. This can be tackled by using caller function provided storage for passing the data back to the caller function. The 'callee' function needs to be modified accordingly to make use of the caller function provided storage for passing the data back to the caller.

Rule# 2 deals with '*atomic*' operations. So what does '*atomic*' mean in reentrancy context? Meaning the operation cannot be interrupted. If an embedded application contains variables which are shared between various threads of a multitasking system (Applicable to Operating System based embedded systems) or between the application and ISR (In Non-RTOS based embedded systems), and if the operation on the shared variable is non-atomic, there is a possibility for corruption of the variable due

to concurrent access of the variable by multiple threads or thread and ISR. As an example let us assume that the variable counter is shared between multiple threads or between a thread and ISR, the instruction,

```
counter++;
```

need not operate in an '*atomic*' way on the variable *counter*. The compiler/cross-compiler in use converts this high level instruction into machine dependent code (machine language) and the objective of incrementing the variable *counter* may be implemented using a number of low level instructions by the compiler/cross compiler. This violates the 'atomic' rule of operation. Imagine a situation where an execution switch (context switch) happens when one thread is in the middle of executing the low level instructions corresponding to the high level instruction *counter++*, of the function and another thread (which is currently in execution after the context switch) or an ISR calls the same function and executes the instruction *counter++*, this will result in inconsistent result. Refer to the section '*Racing*' under the topic '*Task Synchronisation Issues*' of the chapter '*Designing with Real Time Operating Systems*' for more details on this. Eliminating shared global variables and making them as variables local to the function solves the problem of modification of shared variables by concurrent execution of the function.

Rule# 3 is selfexplanatory. If a re-entrant function calls another function which is non-reentrant, it may create potential damages due to the unexpected modification of shared variables if any. What will happen if a reentrant function calls a standard library function at run time? By default most of the run time library is reentrant. If a standard library function is not reentrant the function will no longer be reentrant.

Rule# 4 deals with the atomic way of accessing hardware devices. The term '*atomic*' in hardware access refers to the number of steps involved in accessing a specific register of a hardware device. For the hardware access to be atomic the number of steps involved in hardware access should be one. If access is achieved through multiple steps, any interruption in between the steps may lead to erroneous results. A typical example is accessing the hardware register of an I/O device mapped to the host CPU using paged addressing technique. In order to Read/Write from/to any of the hardware registers of the device a minimum of two steps is required. First write the page address, corresponding to the page in which the hardware register belongs, to the page register and then read the register by giving the address of the register within that page. Now imagine a situation where an interrupt occurs at the moment executing the page setting instruction in progress. The ISR will be executed after finishing this instruction. Suppose ISR also involves a Read/Write to another hardware register belonging to a different page. Obviously it will modify the page register of the device with the required value. What will be its impact? On finishing the ISR, the interrupted code will try to read the hardware register with the page address which is modified by the ISR. This yields an erroneous result.

***How to declare a function as Reentrant***   The way in which a function is declared reentrant is cross-compiler dependent. For Keil C51 cross-compiler for *8051* controller, the keyword '*reentrant*' added as function attribute treats the function as reentrant. For each reentrant function, a reentrant stack area is simulated in internal or external memory depending whether the data memory is internal or external to the processor/controller. A typical reentrant function implementation for C51 cross-compiler for *8051* controller is given below.

```
int multiply (char i, int b) reentrant
{
    int x;
    x = table [i];
    return (x * b);
}
```

The simulated stack used by reentrant functions has its own stack pointer which is independent of the processor stack and stack pointer. For C51 cross compiler the simulated stack and stack pointers are declared and initialised in the startup code in STARTUP.A51 which can be found in the LIB subdirectory. You must modify the startup code to specify which simulated stack(s) to initialize in order to use re-entrant functions. You can also modify the starting address for the top of the simulated stack(s) in the startup code. When calling a function with a re-entrant stack, the cross-compiler must know that the function has a re-entrant stack. The cross-compiler figures this out from the function prototype which should include the reentrant keyword (just like the function definition). The cross compiler must also know which reentrant stack to stuff the arguments for it. For passing arguments, the cross-compiler generates code that decrements the stack pointer and then "pushes" arguments onto the stack by storing the argument indirectly through R0/R1 or DPTR (*8051* specific registers). Calling reentrant function decrements the stack pointer (for local variables) and access arguments using the stack pointer plus an offset (which corresponds to the number of bytes of local variables). On returning, reentrant function adjusts the stack pointer to the value before arguments were pushed. So the caller does not need to perform any stack adjustment after calling a reentrant function.

**Reentrant vs. Recursive Functions** The terms Reentrant and Recursive may sound little confusing. You may find some amount of similarity among them and ultimately it can lead to the thought are Reentrant functions and Recursive functions the same? The answer is—it depends on the usage context of the function. It is not necessary that all recursive functions are reentrant. But a Reentrant function can be invoked recursively by an application.

For *8051* Microcontroller, the internal data memory is very small in size (128 bytes) and the stack as well user data memory is allocated within it. Normally all variables local to a function and function arguments are stored in fixed memory locations of the user data memory and each invocation of the function will access the fixed data memory and any recursive calls to the function use the same memory locations. And, in this case, arguments and locals would get corrupted. Hence the scope for implementing recursive functions is limited. A reentrant function can be called recursively to a recursion level dependent on the simulated stack size for the same reentrant function.

C51 Cross-compiler does not support recursive calls if the functions are non-reentrant.

*9.3.3.22 Dynamic Memory Allocation* Every embedded application, regardless of whether it is running on an operating system based product or a non-operating system based product (Super loop based firmware Architecture) contains different types of variables and they fall into any one of the following storage types namely; *static, auto, global* or *constant* data. Regardless of the storage type each variable requires memory locations to hold them physically. The storage type determines in which area of the memory each variable needs to be kept. For an Assembly language programmer, handling memory for different variables is quite difficult. S/he needs to assign a particular memory location for each variable and should recollect where s/he kept the variable for operations involving that variable.

Certain category of embedded applications deal with fixed number of variables with fixed length and certain other applications deal with variables with fixed memory length as well as variable with total storage size determined only at the runtime of application (e.g. character array with variable size). If the number of variables are fixed in an application and if it doesn't require a variable size at run time, the cross compiler can determine the storage memory required by the application well in advance at the run time and can assign each variable an absolute address or relative (indirect) address within the data memory. Here the memory required is fixed and allocation is done before the execution of the application. This type of memory allocation is referred as '*Static Memory Allocation*'. The term '*Static*' mentioned

here refers '*fixed*' and it is no way related to the storage class static. As mentioned, some embedded applications require data memory which is a combination of fixed memory (Number of variables and variable size is known prior to cross compilation) and variable length data memory. As an example, let's take the scenario where an application deals with reading a stream of character data from an external environment and the length of the stream is variable. It can vary between any numbers (say 1 to 100 bytes). The application needs to store the data in data memory temporarily for some calculation and it can be ignored after the calculation. This scenario can be handled in two ways in the application program. In the first approach, allocate fixed memory with maximum size (say.100 bytes) for storing the incoming data bytes from the stream. In the second approach allocate memory at run time of the application and de-allocate (free) the memory once the data memory storage requirement is over. In the first approach if the memory is allocated fixedly, it is locked forever and cannot re-used by the application even if there is no requirement for the allocated number of bytes and it will definitely create memory bottleneck issues in embedded systems where memory is a big constraint. Hence it is not advised to go for fixed memory allocations for applications demanding variable memory size at run time. Allocating memory on demand and freeing the memory at run time is the most advised method for handling the storage of dynamic (changing) data and this memory allocation technique is known as '*Dynamic Memory Allocation*'.

Dynamic memory allocation technique is employed in Operating System (OS) based embedded systems. Operating system contains a '*Memory Management Unit*' and it is responsible for handling memory allocation related operations. The memory management unit allocates memory to hold the code for the application and the variables associated with the application. The conceptual view of storage of an application and the variables related to the application is represented in Fig. 9.12.
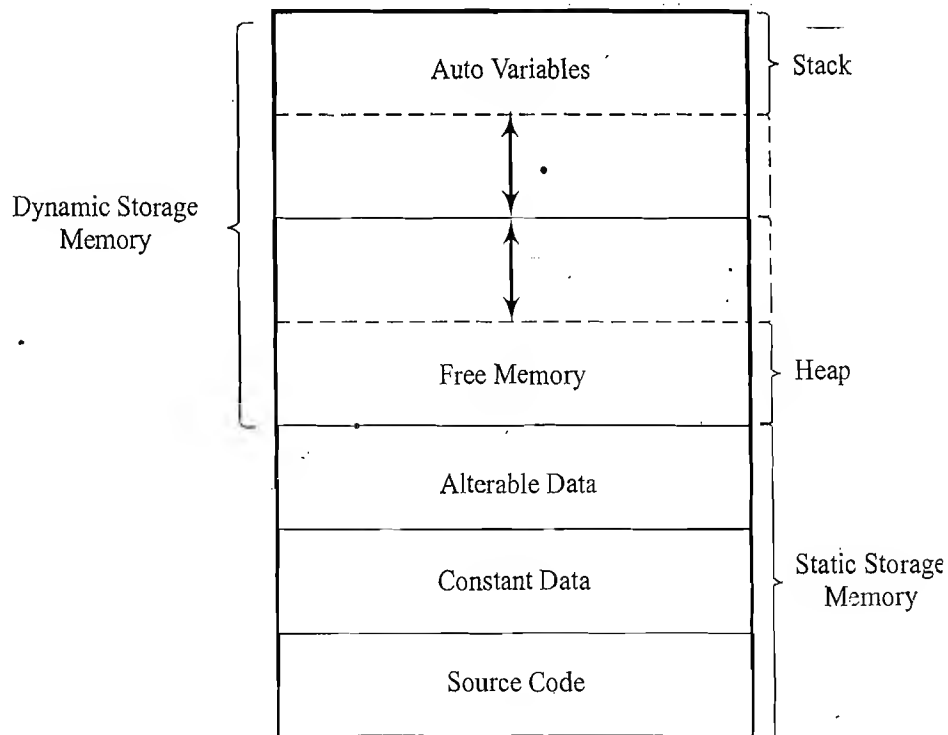


**Fig. 9.12** **Static and Dynamic Memory Storage Allocation**

Memory manager allocates a memory segment to each application and the memory segment holds the source code of the application as well as data related to the application. The actual program code (executable machine code instructions) is stored at the lower address of the memory (at the beginning address of the memory segment and stores upward). Constant data related to the application (e.g. const *int x* = 10;) is stored at the memory area just above the executable code. The alterable data (global and static variables; e.g. static int *j* = 10; *int k* = 2; (global declaration), etc.) are stored at the '*Alterable Data*' memory area. The size of the executable code, the number of constant data and alterable data in an application is always fixed and hence they occupy only a fixed portion of the memory segment. Memory allocated for the executable code, constant data and alterable data together constitute the '*Static storage memory (Fixed storage memory)*'. The storage memory available within the memory segment excluding the fixed memory is the '*Dynamic storage memory*'. Dynamic storage memory area is divided into two separate entities namely '*stack*' and '*heap*'. '*Stack memory*' normally starts at the high memory area (At the top address) of the memory segment and grows down. Within a memory segment, fixed number of storage bytes are allocated to stack memory and the stack can grow up to this maximum allocated size. Stack memory is usually used for holding auto variables (variables local to a function), function parameters and function return values. Depending upon the function calls/return and auto variable storage, the size of the stack grows and shrinks dynamically. If at any point of execution the stack memory exceeds the allocated maximum storage size, the application may crash and this condition is called '*Stack overflow*'. The free memory region lying in between the stack and fixed memory area is called '*heap*'. Heap is the memory pool where storage for data is done dynamically as and when the application demands. Heap is located immediately above the fixed memory area and it grows upward. Any request for dynamic memory allocation by the program increases the size of *heap* (depending on the availability of free memory within heap) and the free memory request decrements the size of *heap* (size of already occupied memory within the heap area). *Heap* can be viewed as a 'bank' in real life application, where customers can demand for loan. Depending on the availability of money, bank may allot loan to the customer and customer re-pays the loan to the bank when he/she is through with his/her need. Bank uses the money repaid by a customer to allocate a loan to another customer—some kind of rolling mechanism. 'C' language establishes the dynamic memory allocation technique through a set of '*Memory management library routines*'. The most commonly used 'C' library functions for dynamic memory allocation are '*malloc*', '*calloc*', '*realloc*' and '*free*'. The following sections illustrate the use of these memory allocation library routines for allocating and de-allocating (freeing) memory dynamically.

***malloc()***    *malloc()* function allocates a block of memory dynamically. The *malloc()* function reserves a block of memory of size specified as parameter to the function, in the *heap* memory and returns a pointer of type void. This can be assigned to a pointer of any valid type. The general form of using *malloc()* function is given below.

```
pointer =  (pointer_type *) malloc (no. of bytes);
```

where '*pointer*' is a pointer of type 'pointer_type'. 'pointer_type' can be '*int*', '*char*', '*float*' etc. *malloc()* function returns a pointer of type 'pointer_type' to a block of memory with size 'no. of bytes'. A typical example is given below.

```
ptr= (char *) malloc(50);
```

This instruction allocates 50 bytes (If there is 50 bytes of memory available in the *heap* area) and the address of the first byte of the allocated memory in the *heap* area is assigned to the pointer *ptr* of type *char*. It should be noted that the *malloc()* function allocates only the requested number of bytes and it

will not allocate memory automatically for the units of the pointer in use. For example, if the programmer wants to allocate memory for 100 integers dynamically, the following code

```
x= (int *) malloc(100);
```

will not allocate memory size for 100 integers, instead it allocates memory for just 100 bytes. In order to make it reserving memory for 100 integer variables, the code should be re-written as

```
x= (int *) malloc(100 * sizeof (int));
```

*malloc()* function can also be used for allocating memory for complex data types such as structure pointers apart from the usual data types like 'int', 'char', 'float' etc. *malloc()* allocates the requested bytes of memory in *heap* in a continuous fashion and the allocation fails if there is no sufficient memory available in continuous fashion for the requested number of bytes by the *malloc()* functions. The return value of *malloc()* will be NULL (0) if it fails. Hence the return value of *malloc()* should be checked to ensure whether the memory allocation is successful before using the pointer returned by the *malloc()* function.

```
E.g.   int * ptr;
       if ((ptr= (int *) malloc(50 * sizeof (int))))
              printf ("Memory allocated successfully");
       else
              printf ("Memory allocation failed");
```

*Remember malloc() only allocates required bytes of memory and will not initialise the allocated memory. The allocated memory contains random data.*

**calloc()**   The library function *calloc()* allocates multiple blocks of storage bytes and initialises each allocated byte to zero. Syntax of *calloc()* function is illustrated below.

```
pointer = (pointer_type *) calloc (n, size of block);
```

where 'pointer' is a pointer of type 'pointer_type'. 'pointer_type' can be 'int', 'char', 'float' etc. '*n*' stands for the number of blocks to be allocated and 'size of block' tells the size of bytes required per block. The *calloc(n*, size of block) function allocates continuous memory for '*n*' number of blocks with 'size of block' number of bytes per block and returns a pointer of type 'pointer_type' pointing to the first byte of the allocated block of memory. A typical example is given below.

```
ptr= (char *) calloc (50,1);
```

Above instruction allocates 50 contiguous blocks of memory each of size one byte in the *heap* memory and assign the address of the first byte of the allocated memory region to the character pointer '*ptr*'. Since *malloc()* is capable of allocating only fixed number of bytes in the *heap* area regardless of the storage type, *calloc()* can be used for overcomming this limitation as discussed below.

```
ptr= (int *) calloc(50,sizeof(int));
```

This instruction allocates 50 blocks of memory, each block representing an integer variable and initialises the allocated memory to zero. Similar to the *malloc()* function, *calloc()* also returns a 'NULL' pointer if there is not enough space in the *heap* area for allocating storage for the requested number of memory by the *calloc()* function. Hence it is advised to check the pointer to which the *calloc()* assigns

the address of the first byte in the allocated memory area. If *calloc()* function fails, the return value will be 'NULL' and the pointer also will be 'NULL'. Checking the pointer for 'NULL' immediately after assigning with pointer returned by *calloc()* function avoids possible errors and application crash.

Features provided by *calloc()* can be implemented using *malloc()* function as

```
pointer = (pointer type *) malloc (n * size of block);
memset(pointer, 0, n * size of block);
```

For example

```
ptr= (int *) malloc (10 * sizeof (int));
memset(ptr, 0, n * size of block);
```

The function *memset (ptr, 0, n \* size of block)* sets the memory block of size (*n* \* size of block) with starting address pointed by '*ptr*', to zero.

**free()** The 'C' memory management library function *free()* is used for releasing or de-allocating the memory allocated in the *heap* memory by *malloc()* or *calloc()* functions. If memory is allocated dynamically, the programmer should release it if the dynamically allocated memory is no longer required for any operation. Releasing the dynamically allocated memory makes it ready to use for other dynamic allocations. The syntax of *free()* function is given below.

```
free (ptr);
```

'*ptr*' is the valid pointer returned by the *calloc()* or *malloc()* function on dynamic memory allocation. Use of an invalid pointer with function *free()* may result in the unexpected behaviour of the application.

*Note:*
1. *The dynamic memory allocated using malloc () or calloc() functions should be released (deallocated) using free () function.*
2. *Any use of a pointer which refers to freed memory space results in abnormal behaviour of application.*
3. *If the parameter to free () function is not a valid pointer, the application behaviour may be unexpected.*

**realloc()** *realloc()* function is used for changing the size of allocated bytes in a dynamically allocated memory block. You may come across situations where the allocated memory is not sufficient to hold the required data or it is surplus in terms of allocated memory bytes. Both of these situations are handled using *realloc()* function. The *realloc()* function changes the size of the block of memory pointed to, by the *pointer* parameter to the number of bytes specified by the *modified size* parameter and it returns a new pointer to the block. The pointer specified by the *pointer* parameter must have been created with the malloc, calloc, or realloc subroutines and not been de-allocated with the free or realloc subroutines. Function *realloc()* may shift the position of the already allocated block depending on the new size, with preserving the contents of the already allocated block and returns a pointer pointing to the first byte of the re-allocated memory block. *realloc()* returns a void pointer if there is sufficient memory available for allocation, else return a 'NULL' pointer. Syntax of *realloc* is given below.

```
realloc (pointer, modified size);
```

Example illustrating the use of *realloc()* function is given below.

```
char *p;
p= (char*) malloc (10); //Allocate 10 bytes of memory
```

```
p= realloc(p,15);        //Change the allocation to 15 bytes
```

*realloc(p,0) is same as free(p), provided 'p' is a valid pointer.*

## Summary

✓ Embedded firmware can be developed to run with the help of an embedded operating system or without an OS. The non-OS based embedded firmware execution runs the tasks in an infinite loop and this approach is known as 'Super loop based' execution

✓ Low level language (Assembly code) or High Level language (C,C++ etc.) or a mix of both are used in embedded firmware development

✓ In Assembly language based design, the firmware is developed using the Assembly language Instructions specific to the target processor. The Assembly code is converted to machine specific code by an assembler.

✓ In High Level language based design, the firmware is developed using High Level language like 'C/C++' and it is converted to machine specific code by a compiler or cross-compiler

✓ Mixing of Assembly and High Level language can be done in three ways namely; mixing assembly routines with a high level language like 'C', mixing high level language functions like 'C' functions with application written in assembly code and invoking assembly instructions inline from the high level code

✓ Embedded 'C' can be considered as a subset of conventional 'C' language. Embedded C supports almost all 'C' instructions and incorporates a few target processor specific functions/instructions. The standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded C

✓ Compiler is a tool for native platform application development, whereas cross-compiler is a tool for cross platform application development

✓ Embedded 'C' supports all the keywords, identifiers and data types, storage classes, arithmetic and logical operations, array and branching instructions supported by standard 'C'

✓ *structure* is a collection of data types. Arrays of structures are helpful in holding configuration data in embedded applications. The *bitfield* feature of structures helps in bit declaration and bit manipulation in embedded applications

✓ *Pre-processor* in 'C' is compiler/cross-compiler directives used by compiler/cross-compiler to filter the source code before compilation/cross-compilation. Preprocessor directives falls into one of the categories: file inclusion, compile control and macro substitution

✓ 'Read only' variable in embedded applications are represented with the keyword *const. Pointer to constant data* is a pointer which points to a data which is read only. A *constant pointer* is a pointer to a fixed memory location. *Constant pointer to constant data* is a pointer pointing to a fixed memory location which is read only

✓ A variable or memory location in embedded application, which is subject to asynchronous modification should be declared with the qualifier *volatile* to prevent the compiler optimisation on the variable

✓ A *constant volatile pointer* represents a Read only register (memory location) of a memory mapped device in embedded application

✓ *while(1) { }; do { }while (1); for (;;){}* are examples for infinite loop setting instructions in embedded 'C'.

✓ The ISR contains the code for saving the current context, code for performing the operation corresponding to the interrupt, code for retrieving the saved context and code for informing the processor that the processing of interrupt is completed

✓ *Recursive function* is a function which calls it repeatedly. Functions which can be shared safely with several processes concurrently are called *re-entrant function.*

✓ *Dynamic memory allocation* is the technique for allocating memory on a need basis for tasks. *malloc(), calloc(), realloc()* and *free()* are the 'C' library routines for dynamic memory management

## Keywords

| | | |
|---|---|---|
| **Super Loop Model** | : | An embedded firmware design model which executes tasks in an infinite loop at a predefined order |
| **Assembly Language** | : | The human readable notation of 'machine language' |
| **Machine Language** | : | Processor understandable language made up of 1s and 0s |
| **Mnemonics** | : | Symbols used in Assembly language for representing the machine language |
| **Assembler** | : | A program to translate Assembly language program to object code |
| **Library** | : | Specially formatted, ordered program collections of object modules |
| **Linker** | : | A software for linking the various object files of a project |
| **Hex File** | : | ASCII representation of the machine code corresponding to an application |
| **Inline Assembly** | : | A technique for inserting assembly instructions in a C program |
| **Embedded C** | : | 'C' Language for embedded firmware development. It supports 'C' instructions and incorporates a few target processor specific functions/instructions along with tailoring of the standard library functions for the target embedded system |
| **Compiler** | : | A software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture |
| **Cross-compiler** | : | The software tools used in cross-platform development applications. It converts the application to a target processor specific code, which is different from the processor architecture on which the compiler is running |
| **Function** | : | A self-contained and re-usable code snippet intended to perform a particular task |
| **Function Pointer** | : | Pointer variable pointing to a function |
| *structure* | : | Variable holding a collection of data types (int, float, char, long, etc.) in C language |
| **structure Padding** | : | The act of arranging the structure elements in memory in a way facilitating increased execution speed |
| *union* | : | A derived form of structure, which allocates memory only to the member variable of union requiring the maximum storage size on declaring a union variable |
| **Pre-processor** | : | A compiler/cross-compiler directives used by compiler/ cross-compiler to filter the source code before compilation/cross-compilation in 'C' language |
| **Macro** | : | The 'C' pre-processor for creating portable inline code |
| *const* | : | A keyword used in 'C' language for informing the compiler/cross-compiler that the variable is constant. It represents a 'Read only' variable |
| **Dynamic Memory Allocation** | : | The technique for allocating memory on a need basis at run time |
| **Stack** | : | The memory area for storing local variables, function parameters and function return values and program counter, in the memory model for an application/ task |
| **Static Memory Area** | : | The memory area holding the program code, constant variables, static and global variables, in the memory model for an application/task |
| **Heap Memory** | : | The free memory lying in between stack and static memory area, which is used for dynamic memory allocation |

## Objective Questions

1. Which of the following is a processor understandable language?
   - (a) Assembly language
   - (b) Machine language
   - (c) High level language
2. Assembly language is the human readable notation of?
   - (a) Machine language
   - (b) High level language
   - (c) None of these
3. Consider the following piece of assembly code

```
ORG 0000H
LJMP MAIN
```

   Here '*ORG*' is a
   - (a) Pseudo-op
   - (b) Label
   - (c) Opcode
   - (d) Operand
4. Translation of assembly code to machine code is performed by the
   - (a) Assembler
   - (b) Compiler
   - (c) Linker
   - (d) Locater
5. A cross-compiler converts an embedded 'C' program to
   - (a) The machine code corresponding to the processor of the PC used for application development
   - (b) The machine code corresponding to a processor which is different from the processor of the PC used for application development
6. '*ptr*' is an *integer* pointer holding the address of an *integer* variable say $x$ which holds the value 10. Assume the address of the *integer* variable $x$ as 0x12ff7c. What will be the output of the below piece of code? Assume the storage size of *integer* is 4

```
ptr+=2;
//Print the address holding by the pointer
printf("0x%x\n", ptr);
```

   - (a) 0x12ff7c
   - (b) 0x12ff7e
   - (c) 0x12ff84
   - (d) None
7. '*ptr*' is a *char* pointer holding the address of a *char* variable say $x$ which holds the value 10. Assume the address of the *char* variable $x$ as 0x12ff7c. What will be the output of the below piece of code?

```
//Print the address holding by the pointer
printf("0x%x\n", ptr++);
```

   - (a) 0x12ff7c
   - (b) 0x12ff7d
   - (c) 0x12ff80
   - (d) None
8. '*ptr*' is a *char* pointer holding the address of a *char* variable say $x$ which holds the value 10. Assume the address of the *char* variable $x$ as 0x12ff7c. What will be the output of the below piece of code?

```
//Print the address holding by the pointer
printf("0x%x\n", ++ptr);
```

   - (a) 0x12ff7c
   - (b) 0x12ff7d
   - (c) 0x12ff80
   - (d) None
9. '*ptr1*' is a *char* pointer holding the address of the *char* variable say $x$ which holds the value 10. '*ptr2*' is a *char* pointer holding the address of the *char* variable say $y$ which holds the value 20. Assume the address of *char* variable $x$ as 0x12ff7c and *char* variable $y$ as 0x12ff78. What will be the output of the following piece of code?

```
//Print the address holding by the pointer
printf("%x\n", (ptr1+ptr2));
```

   - (a) 30
   - (b) 4
   - (c) Compile error (cannot add two pointers)
   - (d) 0x25fef4

10. '*ptr1*' is a *char* pointer holding the address of the *char* variable say *x* which holds the value 10. Assume the address of *char* variable *x* as 0x12ff7c. What will be the output of the following piece of code?

```
++*ptr1;
printf("%x\n", *ptr1);
```

    (a) 0x0a          (b) 0x0b          (c) 0x12ff7c          (d) 0x12ff7d

11. '*ptr1*' is a *char* pointer holding the address of the *char* variable say *x* which holds the value 10. Assume the address of *char* variable *x* as 0x12ff7c. What will be the output of the following piece of code?

```
++*ptr1;
printf("%x\n", ptr1);
```

    (a) 0x0a          (b) 0x0b          (c) 0x12ff7c          (d) 0x12ff7d

12. '*ptr1*' is a *char* pointer holding the address of the *char* variable say *x* which holds the value 10. Assume the address of *char* variable *x* as 0x12ff7c. What will be the output of the following piece of code?

```
*ptr1++;
printf("%x\n", *ptr1);
```

    (a) 0x0b          (b) The contents of memory location 0x12ff7d          (c) 0x12ff7c
    (d) 0x12ff7d

13. '*ptr1*' is a *char* pointer holding the address of the *char* variable say *x* which holds the value 10. Assume the address of *char* variable *x* as 0x12ff7c. What will be the output of the following piece of code?

```
*ptr1++;
printf("%x\n", ptr1);
```

    (a) 0x0b          (b) The contents of memory location 0x12ff7d          (c) 0x12ff7c
    (d) 0x12ff7d

14. Which of the following is the string termination character?
    (a) '\n'          (b) '\t'          (c) '\0'          (d) '\a'

15. What is the output of the following piece of code?

```
char name[6] = {'H','E','L','L','O','\0'};
printf("%d",strlen(name));
```

    (a) 6          (b) 5          (c) 7          (d) None of the above

16. What is the output of the following piece of code?

```
char str1[] = "Hello "
char str2[] = "World!";
str1+= str2;
printf("%s\n",str1);
```

    (a) Hello          (b) Hello World!          (c) Compile error          (d) World!

17. What is the output of the following piece of code?

```
char str1 [ ] = "Hello world!";
char str2 [ ] = "Hello World!" ;
int n;
n= stricmp(str1, str2);
printf("%d", n);
```

    (a) 1          (b) 0          (c) −1

18. What is the output of the following piece of code?

```
char str1[] = "Hello "
char str2[] = "World!";
strcpy(str1,str2);
printf("%s\n",str1);
```

    (a) Hello              (b) Hello World!         (c) Compile error        (d) World!

19. What is the output of the following piece of code?

```
char str1[] = "Hello "
char str2[] = "World!";
str1 = str2;
printf("%s\n",str1);
```

    (a) Hello              (b) Hello World!         (c) Compile error        (d) World!

20. Consider the following structure declaration

```
typedef struct
    {
        unsigned char command;  // command to pass to device
        unsigned char status;   //status of command execution
        unsigned char BytesToSend;  //No. of bytes to send
        unsigned char BytesReceived; //No. of bytes received
    }Info;
```

Assuming the size of *unsigned char* as 1 byte, what will be the memory allocated for the structure?

    (a) 1 byte             (b) 2 bytes           (c) 4 bytes          (d) 0 bytes

21. Consider the following structure declaration

```
typedef struct
    {
        unsigned char hour;     // command to pass to device
        unsigned char minute;   //status of command execution
        unsigned char seconds;  //No. of bytes to send
    }RTC_Time;
```

Assuming the size of *unsigned char* as 1 byte, what will be the output of the following piece of code when compiled for Keil C51 cross compiler

```
static volatile RTC_Time xdata *current_time = (void xdata *) 0x7000;
void main()
{
unsigned char test;
test = current_time->minute;
printf("%d", test);
}
```

    (a) 0x7000          (b) 0x7001         (c) Content of memory location 0x7000
    (d) Content of memory location 0x7001

22. Consider the following structure declaration

```
typedef struct
{
    unsigned char  hour;     // Hour Reg value
    unsigned char  minute;   // Minute value
    unsigned char  seconds;  // Seconds value
}RTC_Time;
```

Assuming the size of *unsigned char* as 1 byte, what will be the output of the following piece of code when compiled for Keil C51 cross compiler

```
void main(void)
{
    unsigned char test;
    test = offsetof(RTC_Time, seconds);
    printf("%d",test);
}
```

(a) 1          (b) 2          (c) Compile error          (d) 0

23. Consider the following union definition

```
typedef union
{
    int intVal;
    unsigned char  charVal[3];
}union_ichar;
```

What will be the output of the following piece of code? Assume the storage size of *int* as 2 and *unsigned char* as 1

```
union_ichar int_char;
void main(void)
{
    unsigned char test;
    test = sizeof (int_char);
    printf("%d",test);
}
```

(a) 0          (b) 2          (c) 3          (d) 5

24. The default initialiser for a *union* with static *storage* is the default for
    (a) The first member variable
    (b) The last member variable
    (c) The member variable with the highest storage requirement

25. Which of the following is (are) True about pre-processor directives?
    (a) compiler/cross-compiler directives
    (b) executable code is generated for pre-processor directives on compilation
    (c) No executable code is generated for pre-processor directives on compilation
    (d) Start with # symbol (e) (a), (b) and (d) (f) (a), (c) and (d)

26. The 'C' pre-processor directive instruction always ends with a semicolon (;). State '*True*' or '*False*'
    (a) True          (b) False

27. Which of the following is the file inclusion pre-processor directive?
    (a) #define          (b) #include          (c) #ifdef          (d) None of these

28. Which of the following pre-processor directive is used for indicating the end of a block following *#ifdef* or *#else*?

   (a) #define          (b) #undef          (c) #endif          (d) #ifndef

29. Which of the following preprocessor directive is used for coding macros?

   (a) #ifdef          (b) #define          (c) #undef          (d) #endif

30. What will be the output of the following piece of code?

```
#define A          2+8
#define B          2+3
void main(void)
{
Unsigned char result ;
result = A/B ;
printf("%d", result) ;
}
```

   (a) 0          (b) 2          (c) 9          (d) 8

31. The instruction

```
const unsigned char* x;
```

   represents:
   (a) Pointer to constant data      (b) Constant pointer to data
   (c) Constant pointer to constant data      (d) None of these

32. The instruction

```
unsigned char* const x;
```

   represents:
   (a) Pointer to constant data      (b) Constant pointer to data
   (c) Constant pointer to constant data      (d) None of these

33. The instruction

```
const unsigned char* const x;
```

   represents:
   (a) Pointer to constant data      (b) Constant pointer to data
   (c) Constant pointer to constant data      (d) None of these

34. The instruction

```
volatile unsigned char* x;
```

   represents:
   (a) Volatile pointer to data      (b) Pointer to volatile data
   (c) Volatile pointer to constant data      (d) None of these

35. The instruction

```
volatile const unsigned char* x;
```

   represents:
   (a) Volatile pointer to data      (b) Pointer to volatile data
   (c) Pointer to constant volatile data      (d) None of these

36. The constant volatile variable in Embedded application represents a
   (a) Write only memory location/register      (b) Read only memory location/register
   (c) Read/Write memory location/register

37. What will be the output of the following piece of code? Assume the data bus width of the controller on which the program is executed as 8 bits.

```
void main(void)
{
unsigned char flag = 0x00;
flag |= (1<<7)
printf("%d", flag);
}
```

(a) 0x00         (b) 0x70         (c) 0x80         (d) 0xFF

38. The variable 'x' declared with the following code statement

```
const int x = 5;
```

will be stored in which section of the memory allocated to the program?

(a) Constant Data Memory    (b) Heap Memory      (c) Alterable Data Memory
(d) Stack Memory           (e) Register

39. What will be the memory allocated on successful execution of the following memory allocation request? Assume the size of *int* as 2 bytes

```
x = (int *) malloc(100);
```

(a) 2 Bytes        (b) 100 Bytes        (c) 200 Bytes     (d) 4 Bytes

40. Which of the following memory management routine is used for changing the size of allocated bytes in a dynami-.cally allocated memory block

(a) *malloc()*        (b) *realloc()*        (c) *calloc()*        (d) *free()*

## Review Questions

1. Explain the different 'embedded firmware design' approaches in detail
2. What is the difference between 'Super loop' based and '*OS*' based embedded firmware design? Which one is the better approach?
3. What is 'Assembly Language' Programming?
4. Explain the format of assembly language instruction
5. What is 'pseudo-ops'? What is the use of it in Assembly Language Programming?
6. Explain the various steps involved in the assembling of an assembly language program
7. What is relocatable code? Explain its significance in assembly programming
8. Explain 'library file' in assembly language context. What is the benefit of 'library file'?
9. What is absolute object file?
10. Explain the advantages of 'Assembly language' based Embedded firmware development
11. Explain the limitations/drawbacks of 'Assembly language' based Embedded firmware development
12. What is the difference between compiler and cross-compiler?
13. Explain the 'High Level language' based 'Embedded firmware' development technique
14. Explain the advantages of 'High Level language' based 'Embedded firmware' development
15. Give examples for situations demanding mixing of assembly with 'C'. Explain the techniques for mixing assembly with 'C'.
16. Give examples for situations demanding mixing of 'C' with assembly. Explain the techniques for mixing 'C' with assembly.

17. What is 'inline Assembly'? How is it different from mixing assembly language with 'C'?
18. What is 'pointer' in embedded C programming? Explain its role in embedded application development.
19. Explain the different arithmetic and relational operations supported by pointers
20. What is 'NULL' Pointer? Explain its significance in embedded C programming
21. Explain the similarities and differences between strings and character arrays
22. Explain function in the Embedded C programming context. Explain the generic syntax of function declaration and implementation
23. What is static function? What is the difference between static and global functions?
24. Explain the similarities and differences between function prototype and function declaration
25. What is function pointer? How is it related to function? Explain the use of function pointers
26. Explain *structure* in the 'Embedded C' programming context. Explain the significance of *structure* over normal variables
27. Explain the declaration and initialisation of structure variables
28. Explain the different operations supported by *structures*
29. What is *structure pointer*? What is the advantage of using structure pointers?
30. Explain 'structure placement at absolute memory location' and its advantage in embedded application development. Will it be possible to place a structure at absolute memory location in desktop application development? Explain
31. What is structure padding? What are the merits and demerits of structure padding?
32. What is bit field? How bit field is useful in variant data access?
33. Explain the use of *offsetof()* macro in structure operations.
34. What is *union*? What is the difference between union and structure?
35. Explain how *union* is useful in variant data access.
36. Explain the use of *union* in 'Embedded C' applications
37. What is pre-processor directive? How is a pre-processor directive instruction differentiated from normal program code?
38. What are the different types of pre-processor directives available in 'Embedded C'? Explain them in detail
39. What is *macro* in 'Embedded C' programming?
40. What is the difference between *macros* and *functions*?
41. What are the merits and drawbacks of *macros*?
42. Write a *macro* to multiply two numbers
43. Explain the different methods of '*constant data*' declaration in 'Embedded C'. Explain the differences between the methods.
44. Explain the difference between '*pointer to constant data*' and '*constant pointer to data*' in 'Embedded C' programming. Explain the syntax for declaring both.
45. What is *constant pointer* to *constant data*? Where is it used in embedded application development?
46. Explain the significance of '*volatile*' type qualifier in Embedded C applications. Which all variables need to be declared as '*volatile*' variables in Embedded C application?
47. What is '*pointer to constant volatile data*'? What is the syntax for defining a 'pointer to constant volatile data'? What is its significance in embedded application development?
48. What is *volatile pointer*? Explain the usage of '*volatile pointer*' in 'Embedded C' application
49. Explain the different techniques for *delay* generation in 'Embedded C' programming. What are the limitations of delay programming for super loop based embedded applications?
50. Explain the different bit manipulation operations supported by 'Embedded C'
51. What is *Interrupt*? Explain its properties? What is its role in embedded application development?
52. What is *Interrupt Vector Address* and *Interrupt Service Routine* (ISR)? How are they related?
53. What is the difference between *Interrupt Service Routine* and Normal Service Routine?
54. Explain *context switching*, *context saving* and *context retrieval* in relation to Interrupts and Interrupt Service Routine (ISR)

55. What all precautionary measures need to be implemented in an Interrupt Service Routine (ISR)?
56. What is '*recursion*'? What is the difference between *recursion* and *iteration*? Which one is better?
57. What are the merits and drawbacks of '*recursion*'?
58. What is '*reentrant*' function? What is its significance in embedded applications development?
59. What is the difference between '*recursive*' and '*reentrant*' function?
60. Explain the different criteria that need to be strictly met by a function to consider it as '*reentrant*' function.
61. What is the difference between *Static* and *Dynamic memory* allocation?
62. Explain the different sections of a memory segment allocated to an application by the memory manager
63. Explain the different '*Memory management* library routines' supported by C
64. Explain the difference between the library functions *malloc()* and *calloc()* for dynamic memory allocation

## Lab Assignments

1. Write a 'C' program to create a 'reentrant' function which removes the white spaces from a string which is supplied as input to the function and returns the new string without white spaces. The main 'C' function passes a string to the reentrant function and accepts the string with white spaces removed.
2. Write a C program to place a character variable at memory location 0x000FF and load it with 0xFF. Compile the application using Microsoft Visual Studio compiler and run it on a desktop machine with Windows Operating System. Record the output and explain the reason behind the output behaviour
3. Write a small embedded C program to complement bit 5 (Assume bit numbering starts at 0) of the status register of a device, which is memory mapped to the CPU. The status register of the device is memory mapped at location 0x3000. The data bus of the controller and the status register of the device is 8bit wide
4. Write a small embedded C program to set bit 0 and clear bit 7 of the status register of a device, which is memory mapped to the CPU. The status register of the device is memory mapped at location 0x8000. The data bus of the controller and the status register of the device is 8bit wide. The application should illustrate the usage of bit manipulation operations.
5. Write a small embedded C program to test the status of bit 5 of the status register and reset it if it is 1, of a device, which is memory mapped to the CPU. The status register of the device is memory mapped at location 0x7000. The data bus of the controller and the status register of the device is 8bit wide. The application should illustrate the usage of bit manipulation operations.
6. Write an 'Embedded C' program for Keil C51 cross-compiler for transmitting a string data through the serial port of *8051* microcontroller as per the following requirements
   (a) Use structure to hold the communication parameters
   (b) Use a structure array for holding the configurations corresponding to various baudrates (say 2400, 4800, 9600 and 195200)
   (c) Write the code for sending a string to the serial port as function. The main routine invokes this function with the string to send. Use polling of Transmit Interrupt for ensuring the sending of a character