CHAPTER

8 Networks

- Why we build networked embedded systems.
- General network architectures and the ISO network layers.
- Several networks: I^2C, CAN, and Ethernet.
- Internet-enabled embedded systems.
- Sensor networks.
- Elevator controller design example.

INTRODUCTION

In this chapter we study ***networks*** that can be used to build ***distributed embedded systems***. In a distributed embedded system, several ***processing elements (PEs)*** (either microprocessors or ASICs) are connected by a network that allows them to communicate. The application is distributed over the PEs, and some of the work is done at each node in the network.

There are several reasons to build network-based embedded systems. When the processing tasks are physically distributed, it may be necessary to put some of the computing power near where the events occur. Consider, for example, an automobile: the short time delays required for tasks such as engine control generally mean that at least parts of the task are done physically close to the engine. Data reduction is another important reason for distributed processing. It may be possible to perform some initial signal processing on captured data to reduce its volume—for example, detecting a certain type of event in a sampled data stream. Reducing the data on a separate processor may significantly reduce the load on the processor that makes use of that data. Modularity is another motivation for network-based design. For instance, when a large system is assembled out of existing components, those components may use a network port as a clean interface that does not interfere with the internal operation of the component in ways that using the microprocessor bus would. A distributed system can also be easier to debug—the microprocessors in one part of the network can be used to probe components in another part of the network. Finally, in some cases, networks are used to build fault tolerance into systems. Distributed embedded system design is another example of hardware/software co-design, since we must design the network topology as well as the software running on the network nodes.

Of course, the microprocessor bus is a simple type of network. However, we use the term *network* to mean an interconnection scheme that does not provide shared memory communication. In the next section, we develop the basic principles of hardware and software architectures for networks. Section 8.2 examines several different networking systems. Section 8.3 considers techniques for the design of distributed embedded systems. Section 8.4 focuses on how embedded systems can be designed to talk to the Internet. Section 8.5 looks at the networked electronics in automobiles and airplanes. Section 8.6 introduces some basic principles of wireless sensor networks. Section 8.7 presents an elevator system as an example of network-based design.

8.1 DISTRIBUTED EMBEDDED ARCHITECTURES

A distributed embedded system can be organized in many different ways, but its basic units are the PE and the network as illustrated in Figure 8.1. A PE may be an instruction set processor such as a *DSP*, *CPU*, or *microcontroller*, as well as a nonprogrammable unit such as the *ASICs* used to implement *PE 4*. An I/O device such as *PE 1* (which we call here a ***sensor*** or ***actuator***, depending on whether it provides input or output) may also be a PE, so long as it can speak the network protocol to communicate with other PEs. The network in this case is a bus, but other network topologies are also possible. It is also possible that the system can use more than one network, such as when relatively independent functions require relatively little communication among them. We often refer to the connection between PEs provided by the network as a ***communication link***.

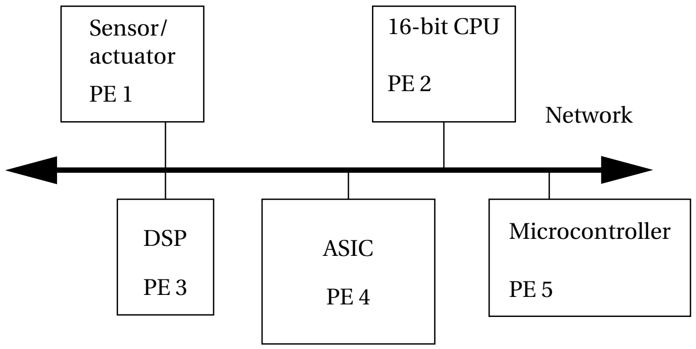


FIGURE 8.1

An example of a distributed embedded system.

The system of PEs and networks forms the ***hardware platform*** on which the application runs.

However, unlike the system bus of Chapter 4, the distributed embedded system does not have memory on the bus (unless a memory unit is organized as an I/O device that speaks the network protocol). In particular, PEs do not fetch instructions over the network as they do on the microprocessor bus. We take advantage of this fact when analyzing network performance—the speed at which PEs can communicate over the bus would be difficult if not impossible to predict if we allowed arbitrary instruction and data fetches as we do on microprocessor buses.

8.1.1 Why Distributed?

Building an embedded system with several PEs talking over a network is definitely more complicated than using a single large microprocessor to perform the same tasks. So why would anyone build a distributed embedded system? All the reasons for designing accelerator systems also apply to distributed embedded systems, and several more reasons are unique to distributed systems.

In some cases, distributed systems are necessary because the devices that the PEs communicate with are physically separated. If the deadlines for processing the data are short, it may be more cost-effective to put the PEs where the data are located rather than build a higher-speed network to carry the data to a distant, fast PE.

An important advantage of a distributed system with several CPUs is that one part of the system can be used to help diagnose problems in another part. Whether you are debugging a prototype or diagnosing a problem in the field, isolating the error to one part of the system can be difficult when everything is done on a single CPU. If you have several CPUs in the system, you can use one to generate inputs for another and to watch its output.

8.1.2 Network Abstractions

Networks are complex systems. Ideally, they provide high-level services while hiding many of the details of data transmission from the other components in the system. In order to help understand (and design) networks, the International Standards Organization has developed a seven-layer model for networks known as Open Systems Interconnection (OSI) models [Sta97A]. Understanding the OSI layers will help us to understand the details of real networks.

The seven layers of the **OSI model**, shown in Figure 8.2, are intended to cover a broad spectrum of networks and their uses. Some networks may not need the services of one or more layers because the higher layers may be totally missing or an intermediate layer may not be necessary. However, any data network should fit into the OSI model. The OSI layers from lowest to highest level of abstraction are described below.

- *Physical:* The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires),

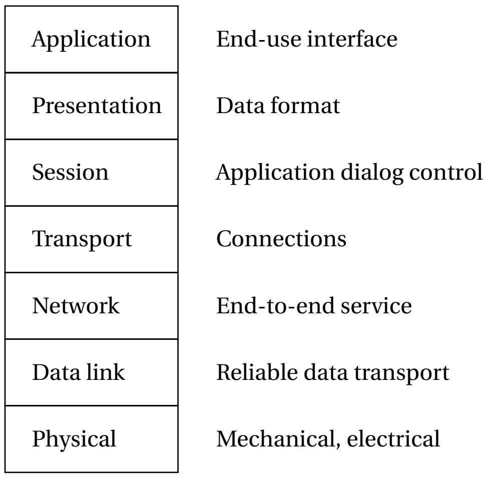


FIGURE 8.2

The OSI model layers.

electrical properties, basic functions of the electrical and physical components, and the basic procedures for exchanging bits.

- *Data link:* The primary purpose of this layer is error detection and control across a single link. However, if the network requires multiple hops over several data links, the data link layer does not define the mechanism for data integrity between hops, but only within a single hop.
- *Network:* This layer defines the basic end-to-end data transmission service. The network layer is particularly important in multihop networks.
- *Transport:* The transport layer defines connection-oriented services that ensure that data are delivered in the proper order and without errors across multiple links. This layer may also try to optimize network resource utilization.
- *Session:* A session provides mechanisms for controlling the interaction of end-user services across a network, such as data grouping and checkpointing.
- *Presentation:* This layer defines data exchange formats and provides transformation utilities to application programs.
- *Application:* The application layer provides the application interface between the network and end-user programs.

Although it may seem that embedded systems would be too simple to require use of the OSI model, the model is in fact quite useful. Even relatively simple embedded networks provide physical, data link, and network services. An increasing number of embedded systems provide Internet service that requires implementing the full range of functions in the OSI model.

8.1.3 Hardware and Software Architectures

Distributed embedded systems can be organized in many different ways depending upon the needs of the application and cost constraints. One good way to understand possible architectures is to consider the different types of interconnection networks that can be used.

A ***point-to-point*** link establishes a connection between exactly two PEs. Point-to-point links are simple to design precisely because they deal with only two components. We do not have to worry about other PEs interfering with communication on the link.

Figure 8.3 shows a simple example of a distributed embedded system built from point-to-point links. The input signal is sampled by the input device and passed to the first digital filter, $F1$, over a point-to-point link. The results of that filter are sent through a second point-to-point link to filter $F2$. The results in turn are sent to the output device over a third point-to-point link. A digital filtering system requires that its outputs arrive at strict intervals, which means that the filters must process their inputs in a timely fashion. Using point-to-point connections allows both $F1$ and $F2$ to receive a new sample and send a new output at the same time without worrying about collisions on the communications network.

It is possible to build a ***full-duplex***, point-to-point connection that can be used for simultaneous communication in both directions between the two PEs. (A half-duplex connection allows for only one-way communication.)

A bus is a more general form of network since it allows multiple devices to be connected to it. Like a microprocessor bus, PEs connected to the bus have addresses. Communications on the bus generally take the form of ***packets*** as illustrated in Figure 8.4. A packet contains an address for the destination and the

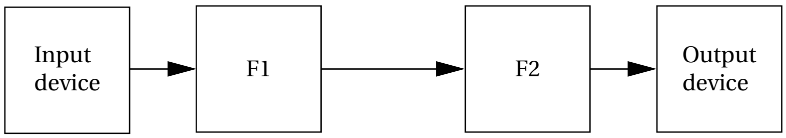


FIGURE 8.3

A signal processing system built from print-to-point links.

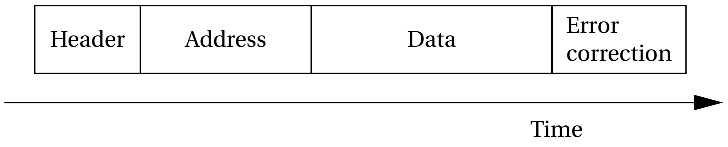


FIGURE 8.4

Format of a typical message on a bus.

data to be delivered. It frequently includes error detection/correction information such as parity. It also may include bits that serve to signal to other PEs that the bus is in use, such as the header shown in the figure. The data to be transmitted from one PE to another may not fit exactly into the size of the ***data payload*** on the packet. It is the responsibility of the transmitting PE to divide its data into packets; the receiving PE must of course reassemble the complete data message from the packets.

Distributed system buses must be arbitrated to control simultaneous access, just as with microprocessor buses. Arbitration scheme types are summarized below.

- *Fixed-priority arbitration* always gives priority to competing devices in the same way. If a high-priority and a low-priority device both have long data transmissions ready at the same time, it is quite possible that the low-priority device will not be able to transmit anything until the high-priority device has sent all its data packets.
- *Fair arbitration* schemes make sure that no device is starved. ***Round-robin arbitration*** is the most commonly used of the fair arbitration schemes. The PCI bus requires that the arbitration scheme used on the bus must be fair, although it does not specify a particular arbitration scheme. Most implementations of PCI use round-robin arbitration.

A bus has limited available bandwidth. Since all devices connect to the bus, communications can interfere with each other. Other network topologies can be used to reduce communication conflicts. At the opposite end of the generality spectrum from the bus is the ***crossbar*** network shown in Figure 8.5. A crossbar not only allows any input to be connected to any output, it also allows all combinations of input/output connections to be made. Thus, for example, we can simultaneously

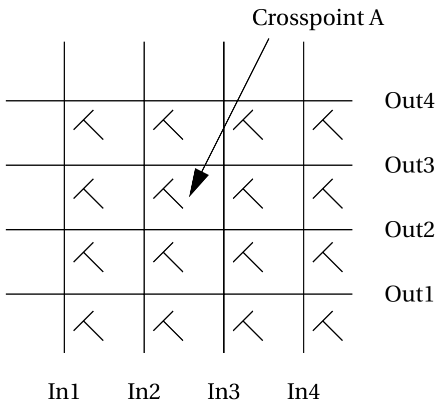


FIGURE 8.5

A crossbar network.

connect *in1* to *out4*, *in2* to *out3*, *in3* to *out2*, and *in4* to *out1* or any other combinations of inputs. (Multicast connections can also be made from one input to several outputs.) A ***crosspoint*** is a switch that connects an input to an output. To connect an input to an output, we activate the crosspoint at the intersection between the corresponding input and output lines in the crossbar. For example, to connect *in2* and *out3* in the figure, we would activate crossbar *A* as shown. The major drawback of the crossbar network is expense: The size of the network grows as the square of the number of inputs (assuming the numbers of inputs and outputs are equal).

Many other networks have been designed that provide varying amounts of parallel communication at varying hardware costs. Figure 8.6 shows an example ***multistage network***. The crossbar of Figure 8.5 is a ***direct network*** in which messages go from source to destination without going through any memory element. Multistage networks have intermediate routing nodes to guide the data packets.

Most networks are ***blocking***, meaning that there are some combinations of sources and destinations for which messages cannot be delivered simultaneously. A bus is a maximally blocking network since any message on the bus blocks messages from any other node. A crossbar is non-blocking.

In general, networks differ from microprocessor buses in how they implement communication protocols. Both need handshaking to ensure that PEs do not interfere with each other. But in most networks, most of the protocol is performed in software. Microprocessors rely on bus hardware for fast transfers of instructions and data to and from the CPU. Most embedded network ports on microprocessors implement the basic communication functions (such as driving the communications medium) in hardware and implement many other operations in software.

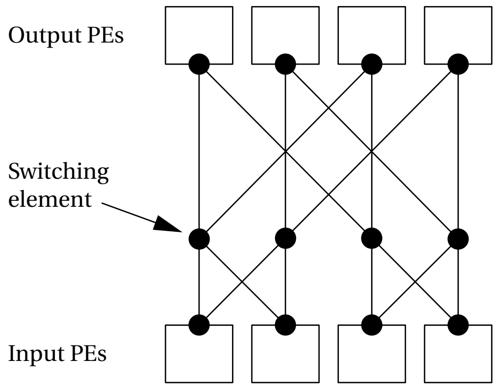


FIGURE 8.6

A multistage network.

An alternative to a non-bus network is to use multiple networks. As with PEs, it may be cheaper to use two slow, inexpensive networks than a single high-performance, expensive network. If we can segregate critical and noncritical communications onto separate networks, it may be possible to use simpler topologies such as buses. Many systems use serial links for low-speed communication and CPU buses for higher speed and volume data transfers.

8.1.4 Message Passing Programming

Distributed embedded systems do not have shared memory, so they must communicate by passing *messages*. We will refer to a message as the natural communication unit of an algorithm; in general, a message must be broken up into packets to be sent on the network. A procedural interface for sending a packet might look like the following:

```
send_packet(address,data);
```

The routine should return a value to indicate whether the message was sent successfully if the network includes a handshaking protocol. If the message to be sent is longer than a packet, it must be broken up into packet-size data segments as follows:

```
for (i = 0; i < message.length; i = i + PACKET_SIZE)
     send_packet(address,&message.data[i]);
```

The above code uses a loop to break up an arbitrary-length message into packet-size chunks. However, clever system design may be able to recast the message to take advantage of the packet format. For example, clever encoding may reduce the length of the message enough so that it fits into a single packet. On the other hand, if the message is shorter than a packet or not an even multiple of the packet data size, some extra information may be packed into the remaining bits of a packet.

Reception of a packet will probably be implemented with interrupts. The simplest procedural interface will simply check to see whether a received message is waiting in a buffer. In a more complex RTOS-based system, reception of a packet may enable a process for execution.

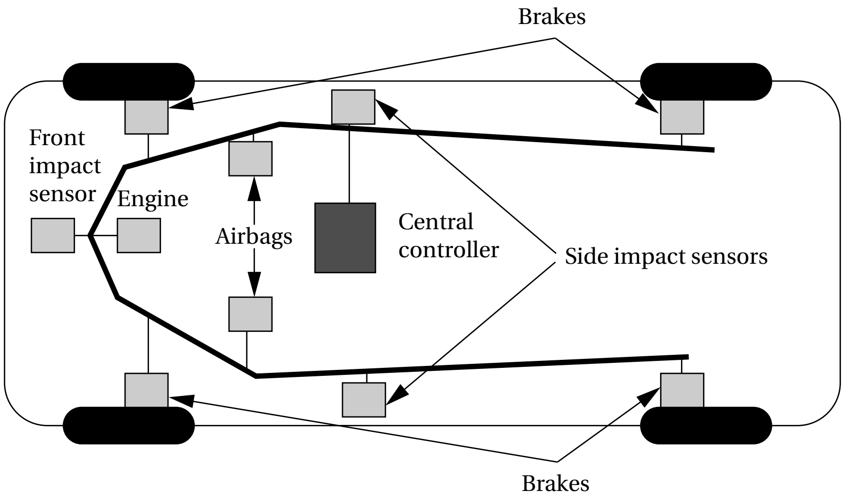
As seen in Section 6.4, communication may be blocking or non-blocking. Of course, the simplest implementation of message passing is blocking, with the routine not returning until it has transmitted or received. A non-blocking network interface requires a queue of data to be sent, with the network driver sending packets off the head of the queue and placing received packets on the tail of the queue. A non-blocking communication mechanism makes sense only when concurrency is available between computing and data transfer.

Network protocols may encourage a ***data-push*** design style for the system built around the network. In a single-CPU environment, a program typically initiates a read whenever it wants data. In many networked systems, nodes send values out without any request from the intended user of the system. Data-push programming makes sense for periodic data—if the data will always be used at regular intervals, we can reduce data traffic on the network by automatically sending it when it is needed. Example 8.1 shows an application that can make good use of the data-push architecture.

Example 8.1

Data-push network architectures

Consider the following automobile in which distributed sensors and actuators talk to a central controller:



The sensors generally need to be sampled periodically. In such a system, it makes sense for sensors to transmit their data automatically rather than waiting for the controller to request it.

8.2 NETWORKS FOR EMBEDDED SYSTEMS

Networks for embedded computing span a broad range of requirements; many of those requirements are very different from those for general-purpose networks. Some networks are used in safety-critical applications, such as automotive control. Some networks, such as those used in consumer electronics systems, must be very inexpensive. Other networks, such as industrial control networks, must be extremely rugged and reliable.

Several interconnect networks have been developed especially for distributed embedded computing:

- The I^{2}C bus is used in microcontroller-based systems.
- The Controller Area Network (CAN) bus was developed for automotive electronics. It provides megabit rates and can handle large numbers of devices.
- Ethernet and variations of standard Ethernet are used for a variety of control applications.

In addition, many networks designed for general-purpose computing have been put to use in embedded applications as well.

In this section, we study some commonly used embedded networks, including the I^{2}C bus and Ethernet; we will also briefly discuss networks for industrial applications.

8.2.1 The I^{2}C Bus

The ***I^{2}C bus*** [Phi92] is a well-known bus commonly used to link microcontrollers into systems. It has even been used for the command interface in an MPEG-2 video chip [van97]; while a separate bus was used for high-speed video data, setup information was transmitted to the on-chip controller through an I^{2}C bus interface.

I^{2}C is designed to be low cost, easy to implement, and of moderate speed (up to 100 KB/s for the standard bus and up to 400 KB/s for the extended bus). As a result, it uses only two lines: the ***serial data line (SDL)*** for data and the ***serial clock line (SCL)***, which indicates when valid data are on the data line. Figure 8.7 shows the structure of a typical I^{2}C bus system. Every node in the network is connected to both SCL and SDL. Some nodes may be able to act as bus masters and the bus

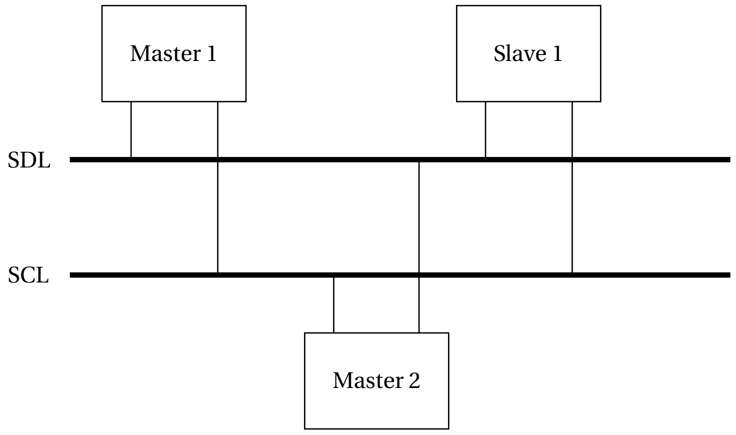


FIGURE 8.7

Structure of an I^{2}C bus system.

may have more than one master. Other nodes may act as slaves that only respond to requests from masters.

The basic electrical interface to the bus is shown in Figure 8.8. The bus does not define particular voltages to be used for high or low so that either bipolar or MOS circuits can be connected to the bus. Both bus signals use open collector/open drain circuits.[1] A pull-up resistor keeps the default state of the signal high, and transistors are used in each bus device to pull down the signal when a 0 is to be transmitted. Open collector/open drain signaling allows several devices to simultaneously write the bus without causing electrical damage.

The open collector/open drain circuitry allows a slave device to stretch a clock signal during a read from a slave. The master is responsible for generating the SCL clock, but the slave can stretch the low period of the clock (but not the high period) if necessary.

The I^2C bus is designed as a multimaster bus—any one of several different devices may act as the master at various times. As a result, there is no global master to generate the clock signal on *SCL*. Instead, a master drives both *SCL* and *SDL* when it is sending data. When the bus is idle, both *SCL* and *SDL* remain high. When two devices try to drive either *SCL* or *SDL* to different values, the open collector/open drain circuitry prevents errors, but each master device must listen to the bus while transmitting to be sure that it is not interfering with another message—if the device receives a different value than it is trying to transmit, then it knows that it is interfering with another message.

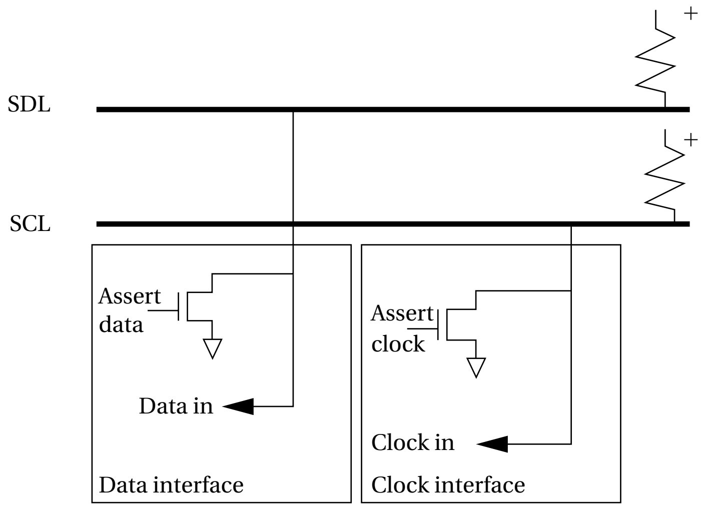


FIGURE 8.8

Electrical interface to the I^2C bus.

[1]An open collector uses a bipolar transistor, while an open drain circuit uses an MOS transistor.

Every I^{2}C device has an address. The addresses of the devices are determined by the system designer, usually as part of the program for the I^{2}C driver. The addresses must of course be chosen so that no two devices in the system have the same address. A device address is 7 bits in the standard I^{2}C definition (the extended I^{2}C allows 10-bit addresses). The address 0000000 is used to signal a ***general call*** or bus broadcast, which can be used to signal all devices simultaneously. The address 11110XX is reserved for the extended 10-bit addressing scheme; there are several other reserved addresses as well.

A ***bus transaction*** comprised a series of 1-byte ***transmissions*** and an address followed by one or more data bytes. I^{2}C encourages a data-push programming style. When a master wants to write a slave, it transmits the slave's address followed by the data. Since a slave cannot initiate a transfer, the master must send a read request with the slave's address and let the slave transmit the data. Therefore, an address transmission includes the 7-bit address and 1 bit for data direction: 0 for writing from the master to the slave and 1 for reading from the slave to the master. (This explains the 7-bit addresses on the bus.) The format of an address transmission is shown in Figure 8.9.

A bus transaction is initiated by a start signal and completed with an end signal as follows:

- A start is signaled by leaving the SCL high and sending a 1 to 0 transition on SDL.
- A stop is signaled by setting the SCL high and sending a 0 to 1 transition on SDL.

However, starts and stops must be paired. A master can write and then read (or read and then write) by sending a start after the data transmission, followed by another address transmission and then more data. The basic state transition graph for the master's actions in a bus transaction is shown in Figure 8.10.

The formats of some typical complete bus transactions are shown in Figure 8.11. In the first example, the master writes 2 bytes to the addressed slave. In the second, the master requests a read from a slave. In the third, the master writes 1 byte to the slave, and then sends another start to initiate a read from the slave.

Figure 8.12 shows how a data byte is transmitted on the bus, including start and stop events. The transmission starts when SDL is pulled low while SCL remains high.

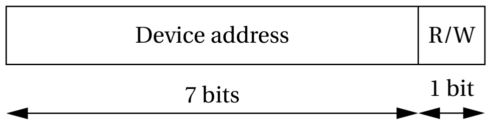


FIGURE 8.9

Format of an I^{2}C address transmission.

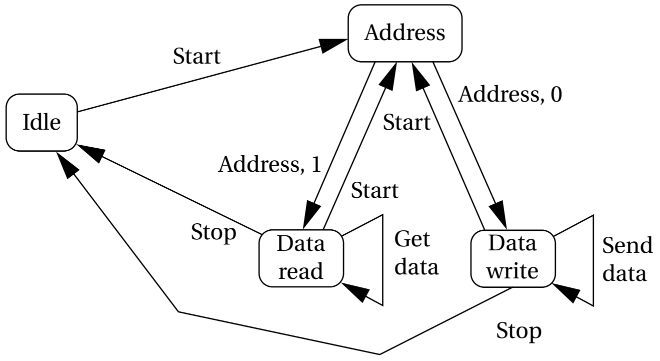


FIGURE 8.10

State transition graph for an I^{2}C bus master.

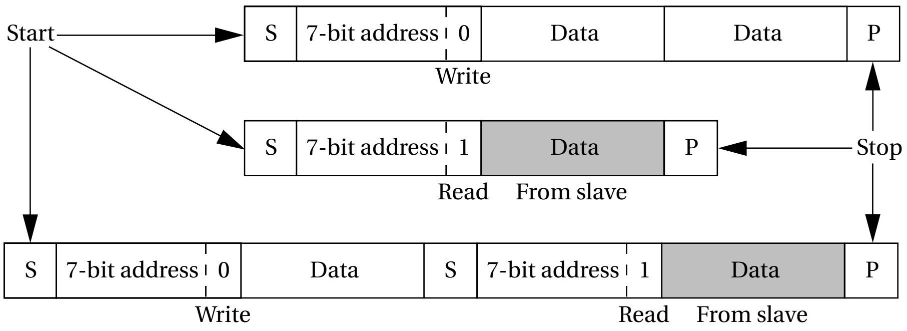


FIGURE 8.11

Typical bus transactions on the I^{2}C bus.

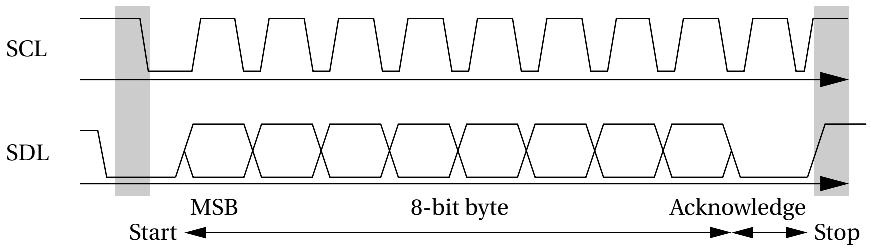


FIGURE 8.12

Transmitting a byte on the I^{2}C bus.

After this start condition, the clock line is pulled low to initiate the data transfer. At each bit, the clock line goes high while the data line assumes its proper value of 0 or 1. An acknowledgment is sent at the end of every 8-bit transmission, whether it is an address or data. For acknowledgment, the transmitter does not pull down the SDL, allowing the receiver to set the SDL to 0 if it properly received the byte. After acknowledgment, the SDL goes from low to high while the SCL is high, signaling the stop condition.

The bus uses this feature to arbitrate on each message. When sending, devices listen to the bus as well. If a device is trying to send a logic 1 but hears a logic 0, it immediately stops transmitting and gives the other sender priority. (The devices should be designed so that they can stop transmitting in time to allow a valid bit to be sent.) In many cases, arbitration will be completed during the address portion of a transmission, but arbitration may continue into the data portion. If two devices are trying to send identical data to the same address, then of course they never interfere and both succeed in sending their message.

The I^2C interface on a microcontroller can be implemented with varying percentages of the functionality in software and hardware [Phi89]. As illustrated in Figure 8.13, a typical system has a 1-bit hardware interface with routines for byte-level functions. The I^2C device takes care of generating the clock and data. The application code calls routines to send an address, send a data byte, and so on, which then generates the SCL and SDL, acknowledges, and so forth. One of the microcontroller's timers is typically used to control the length of bits on the bus. Interrupts may be used to recognize bits. However, when used in master mode, polled I/O may be acceptable if no other pending tasks can be performed, since masters initiate their own transfers.

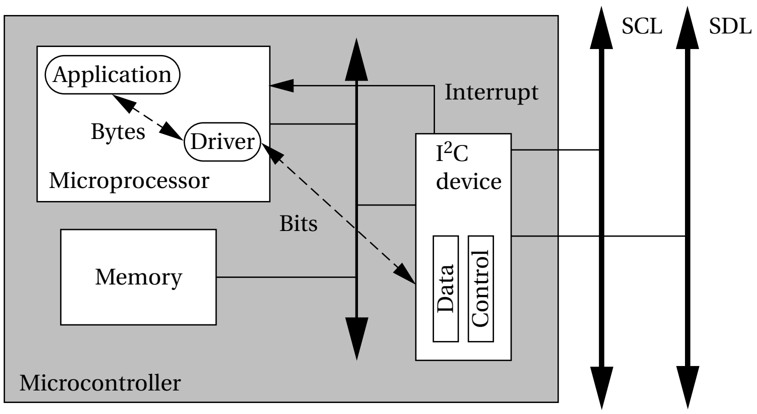


FIGURE 8.13

An I^2C interface in a microcontroller.

8.2.2 Ethernet

Ethernet is very widely used as a local area network for general-purpose computing. Because of its ubiquity and the low cost of Ethernet interfaces, it has seen significant use as a network for embedded computing. Ethernet is particularly useful when PCs are used as platforms, making it possible to use standard components, and when the network does not have to meet rigorous real-time requirements.

The physical organization of an Ethernet is very simple, as shown in Figure 8.14. The network is a bus with a single signal path; the Ethernet standard allows for several different implementations such as twisted pair and coaxial cable.

Unlike the I^2C bus, nodes on the Ethernet are not synchronized—they can send their bits at any time. I^2C relies on the fact that a collision can be detected and quashed within a single bit time thanks to synchronization. But since Ethernet nodes are not synchronized, if two nodes decide to transmit at the same time, the message will be ruined. The Ethernet arbitration scheme is known as ***Carrier Sense Multiple Access*** with ***Collision Detection (CSMA/CD)***. The algorithm is outlined in Figure 8.15. A node that has a message waits for the bus to become silent and then starts transmitting. It simultaneously listens, and if it hears another transmission that interferes with its transmission, it stops transmitting and waits to retransmit. The waiting time is random, but weighted by an exponential function of the number of times the message has been aborted. Figure 8.16 shows the exponential backoff function both before and after it is modulated by the random wait time. Since a message may be interfered with several times before it is successfully transmitted, the ***exponential backoff*** technique helps to ensure that the network does not become overloaded at high demand factors. The random factor in the wait time minimizes the chance that two messages will repeatedly interfere with each other.

The maximum length of an Ethernet is determined by the nodes' ability to detect collisions. The worst case occurs when two nodes at opposite ends of the bus are transmitting simultaneously. For the collision to be detected by both nodes, each node's signal must be able to travel to the opposite end of the bus so that it can be heard by the other node. In practice, Ethernets can run up to several hundred meters.

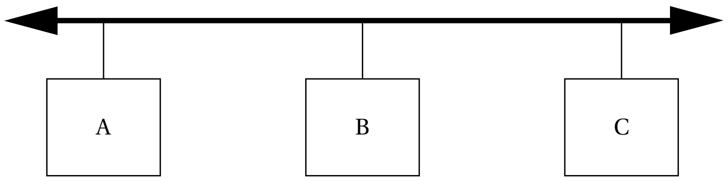


FIGURE 8.14

Ethernet organization.

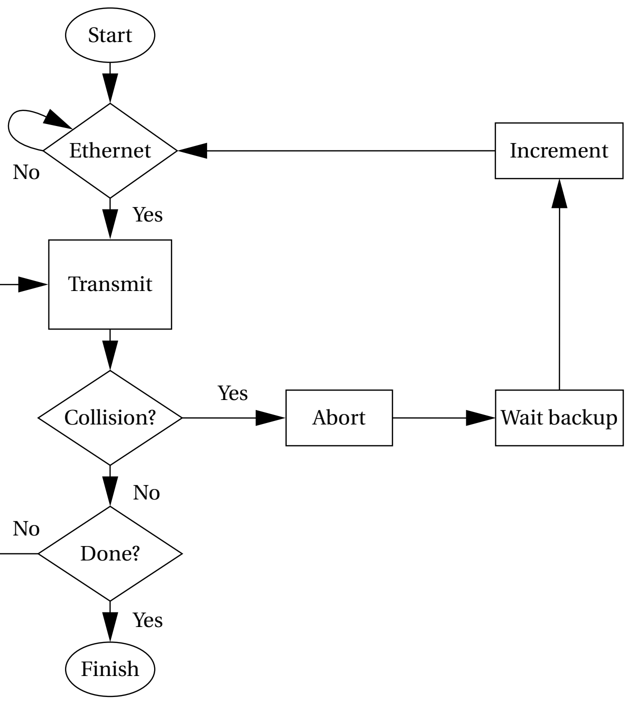


FIGURE 8.15

The Ethernet CSMA/CD algorithm.

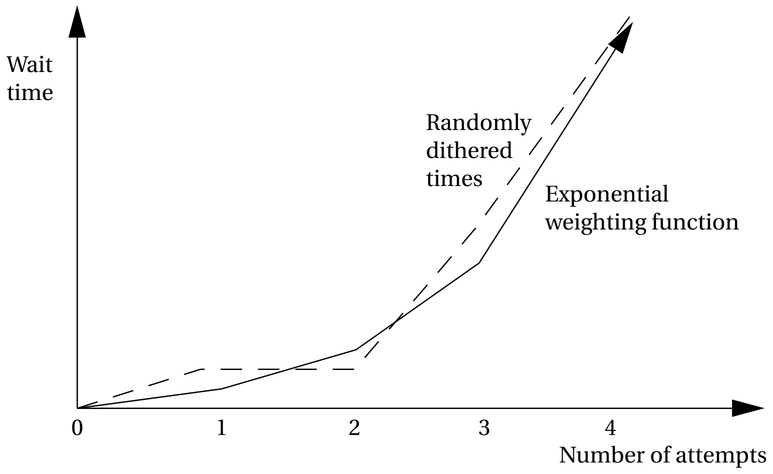


FIGURE 8.16

Exponential backoff times.

Preamble	Start frame	Destination address	Source address	Length	Data	Padding	CRC

FIGURE 8.17

Ethernet packet format.

Figure 8.17 shows the basic format of an Ethernet packet. It provides addresses of both the destination and the source. It also provides for a variable-length data payload.

The fact that it may take several attempts to successfully transmit a message and that the waiting time includes a random factor makes Ethernet performance difficult to analyze. It is possible to perform data streaming and other real-time activities on Ethernets, particularly when the total network load is kept to a reasonable level, but care must be taken in designing such systems.

Ethernet was not designed to support real-time operations; the exponential backoff scheme cannot guarantee delivery time of any data. Because so much Ethernet hardware and software is available, many different approaches have been developed to extend Ethernet to real-time operation; some of these are compatible with the standard while others are not. As Decotignie points out [Dec05], there are three ways to reduce the variance in Ethernet's packet delivery time: suppress collisions on the network, reduce the number of collisions, or resolve collisions deterministically. Felser [Fel05] describes several real-time Ethernet architectures.

8.2.3 Fieldbus

Manufacturing systems require networked sensors and actuators. Fieldbus (http://www.fieldbus.org) is a set of standards for industrial control and instrumentation systems.

The H1 standard uses a twisted-pair physical layer that runs at 31.25 MB/s. It is designed for device integration and process control.

The High Speed Ethernet standard is used for backbone networks in industrial plants. It is based on the 100 MB/s Ethernet standard. It can integrate devices and subsystems.

8.3 NETWORK-BASED DESIGN

Designing a distributed embedded system around a network involves some of the same design tasks we faced in accelerated systems. We must schedule computations in time and allocate them to PEs. Scheduling and allocation of communication are important additional design tasks required for many distributed networks. Many embedded networks are designed for low cost and therefore do not provide excessively high communication speed. If we are not careful, the network can become the bottleneck in system design. In this section we concentrate on design tasks unique to network-based distributed embedded systems.

We know how to analyze the execution time of programs and systems of processes on single CPUs, but to analyze the performance of networks we must know how to determine the delay incurred by transmitting messages. Let us assume for the moment that messages are sent reliably—we do not have to retransmit a message. The ***message delay*** for a single message with no contention (as would be the case in a point-to-point connection) can be modeled as

$$t_m = t_x + t_n + t_r \tag{8.1}$$

where t_x is the transmitter-side overhead, t_n is the network transmission time, and t_r is the receiver-side overhead. In I^{2}C, t_x and t_r are negligible relative to t_n, as illustrated by Example 8.2.

Example 8.2

Simple message delay for an I^{2}C message

Let's assume that our I^{2}C bus runs at the rate of 100 KB/s and that we need to send one 8-bit byte. Based on the message format shown in Figure 8.9, we can compute the number of bits in the complete packet:

$$\begin{aligned} n_{\text{packet}} &= \text{startbit} + \text{address} + \text{data} + \text{stopbit} \\ &= 1 + 8 + 8 + 1 = 18 \text{ bits} \end{aligned}$$

The time required, then, to transmit the packet is

$$t_n = n_{\text{packet}} \times t_{\text{bit}} = 1.8 \times 10^{-4} \text{ s.}$$

Some of the instructions in the transmitter and receiver drivers—namely, the loops that send bytes to and receive bytes from the network interface—will run concurrently with the message transmission. If we assume that 20 instructions outside of these loops are executed by the transmitter and receiver, overheads on an 8 MHz microcontroller would be as follows:

$$t_x = t_r = 20 \times 0.125 \times 10^{-6} = 2.5 \times 10^{-6}.$$

The total message delay is:

$$t_m = 2.5 \times 10^{-6} + 1.8 \times 10^{-4} + 2.5 \times 10^{-6} = 1.85 \times 10^{-4}.$$

Overhead is <3% of the total message time in this case.

If messages can interfere with each other in the network, analyzing communication delay becomes difficult. In general, because we must wait for the network to become available and then transmit the message, we can write the **message delay** as

$$t_y = t_d + t_m \tag{8.2}$$

where t_d is the ***network availability delay*** incurred waiting for the network to become available. The main problem, therefore, is calculating t_d. That value depends on the type of arbitration used in the network.

- If the network uses fixed-priority arbitration, the network availability delay is unbounded for all but the highest-priority device. Since the highest-priority device always gets the network first, unless there is an application-specific limit on how long it will transmit before relinquishing the network, it can keep blocking the other devices indefinitely.
- If the network uses fair arbitration, the network availability delay is bounded. In the case of round-robin arbitration, if there are N devices, then the worst-case network availability delay is $N(t_x + t_{\text{arb}})$, where t_{arb} is the delay incurred for arbitration. t_{arb} is usually small compared to transmission time.

Even when round-robin arbitration is used to bound the network availability delay, the waiting time can be very long. If we add acknowledgment and data corruption into the analysis, figuring network delay is more difficult. Assuming that errors are random, we cannot predict a worst-case delay since every packet may contain an error. We can, however, compute the probability that a packet will be delayed for more than a given amount of time. However, such analysis is beyond the scope of this book.

Arbitration on networks is a form of prioritization. Therefore, we can use the techniques we learned for process scheduling in Chapter 6 to help us schedule communications. In a rate-monotonic communication scheme, the task with the shortest deadline should be assigned the highest priority in the network.

Our process scheduling model assumed that we could interrupt processes at any point. But network communications are organized into packets. In most networks we cannot interrupt a packet transmission to take over the network for a higher-priority packet. As a result, networks exhibit priority inversion like that introduced in Chapter 6. When a low-priority message is on the network, the network is effectively allocated to that low-priority message, allowing it to block higher-priority messages. This cannot cause deadlock since each message has a bounded length, but it can slow down critical communications. The only solution is to analyze network behavior to determine whether priority inversion causes some messages to be delayed for too long.

Of course, a round-robin arbitrated network puts all communications at the same priority. This does not eliminate the priority inversion problem because processes still have priorities.

Thus far we have assumed a ***single-hop network:*** A message is received at its intended destination directly from the source, without going through any other network node. It is possible to build ***multihop networks*** in which messages are routed through network nodes to get to their destinations. (Using a multistage network does not necessarily mean using a multihop network—the stages in a multistage network are generally much smaller than the network PEs.) Figure 8.18 shows an example of a multihop communication. The hardware platform has two separate networks (perhaps so that communications between subsets of the PEs do not interfere), but there is no direct path from $M1$ to $M5$. The message is therefore routed through $M3$, which reads it from one network and sends it on to the other one. Analyzing delays

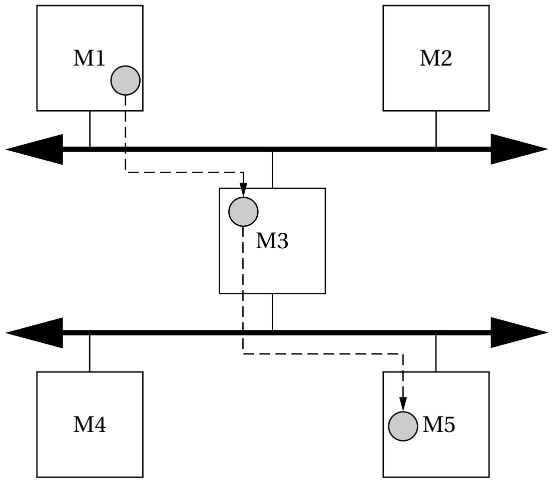


FIGURE 8.18

A multihop communication.

through multihop systems is very difficult. For example, the time that the message is held at $M3$ depends on both the computational load of $M3$ and the other messages that it must handle.

If there is more than one network, we must allocate communications to the networks. We may establish multiple networks so that lower-priority communications can be handled separately without interfering with high-priority communications on the primary network.

Scheduling and allocation of computations and communications are clearly interrelated. If we change the allocation of computations, we change not only the scheduling of processes on those PEs but also potentially the schedules of PEs with which they communicate. For example, if we move a computation to a slower PE, its results will be available later, which may mean rescheduling both the process that uses the value and the communication that sends the value to its destination.

8.4 INTERNET-ENABLED SYSTEMS

Some very different types of distributed embedded system are rapidly emerging—the ***Internet-enabled embedded system*** and ***Internet appliances***. The Internet is not well suited to the real-time tasks that are the bread and butter of embedded computing, but it does provide a rich environment for non–real-time interaction. In this section we will discuss the Internet and how it can be used by embedded computing systems.

8.4.1 Internet

The **Internet Protocol (IP)** [Los97, Sta97A] is the fundamental protocol on the ***Internet***. It provides connectionless, packet-based communication. Industrial automation has long been a good application area for Internet-based embedded systems. Information appliances that use the Internet are rapidly becoming another use of IP in embedded computing.

Internet protocol is not defined over a particular physical implementation—it is an ***internetworking*** standard. Internet packets are assumed to be carried by some other network, such as an Ethernet. In general, an Internet packet will travel over several different networks from source to destination. The IP allows data to flow seamlessly through these networks from one end user to another. The relationship between IP and individual networks is illustrated in Figure 8.19. IP works at the network layer. When node A wants to send data to node B, the application's data pass through several layers of the protocol stack to send to the IP. IP creates packets for routing to the destination, which are then sent to the *data link* and *physical* layers. A node that transmits data among different types of networks is known as a ***router***. The router's functionality must go up to the IP layer, but since it is not running applications, it does not need to go to higher levels of the OSI model. In general, a packet may go through several routers to get to its destination. At the destination, the IP layer provides data to the transport layer and ultimately the receiving application. As the data pass through several layers of the protocol stack, the IP packet data are encapsulated in packet formats appropriate to each layer.

The basic format of an IP packet is shown in Figure 8.20. The header and data payload are both of variable length. The maximum total length of the header and data payload is 65,535 bytes.

An Internet address is a number (32 bits in early versions of IP, 128 bits in IPv6). The IP address is typically written in the form xxx.xx.xx.xx. The names by which users and applications typically refer to Internet nodes, such as foo.baz.com,

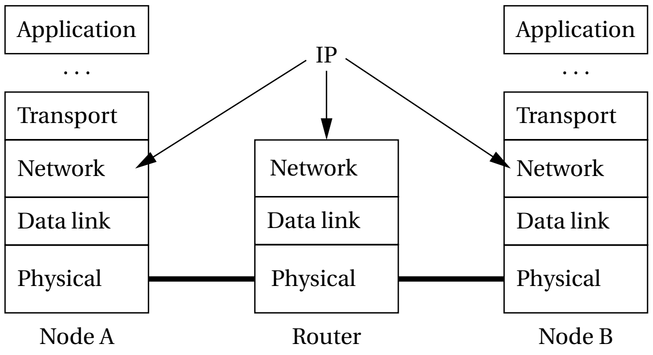


FIGURE 8.19

Protocol utilization in Internet communication.

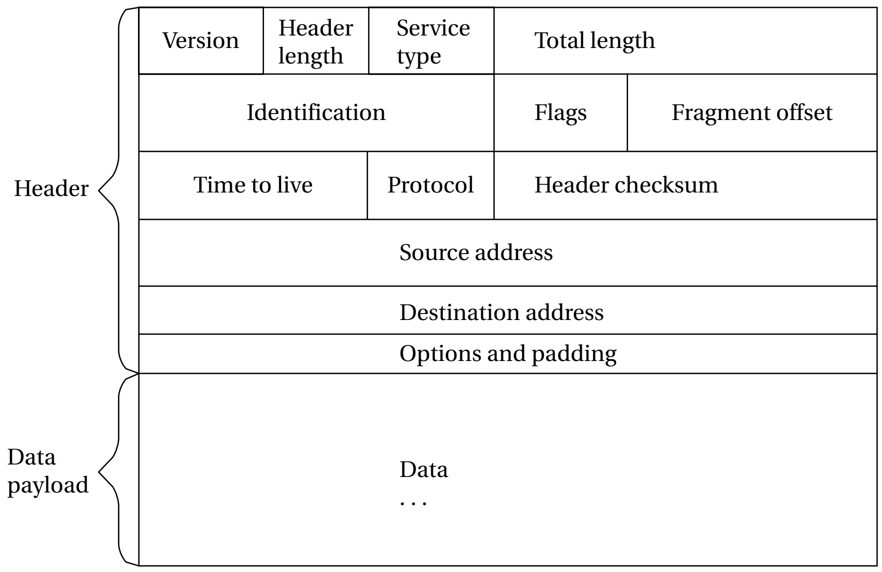


FIGURE 8.20

IP packet structure.

are translated into IP addresses via calls to a ***Domain Name Server***, one of the higher-level services built on top of IP.

The fact that IP works at the network layer tells us that it does not guarantee that a packet is delivered to its destination. Furthermore, packets that do arrive may come out of order. This is referred to as ***best-effort routing***. Since routes for data may change quickly with subsequent packets being routed along very different paths with different delays, real-time performance of IP can be hard to predict. When a small network is contained totally within the embedded system, performance can be evaluated through simulation or other methods because the possible inputs are limited. Since the performance of the Internet may depend on worldwide usage patterns, its real-time performance is inherently harder to predict.

The Internet also provides higher-level services built on top of IP. The ***Transmission Control Protocol (TCP)*** is one such example. It provides a connection-oriented service that ensures that data arrive in the appropriate order, and it uses an acknowledgment protocol to ensure that packets arrive. Because many higher-level services are built on top of TCP, the basic protocol is often referred to as TCP/IP.

Figure 8.21 shows the relationships between IP and higher-level Internet services. Using IP as the foundation, TCP is used to provide ***File Transport Protocol*** for batch file transfers, ***Hypertext Transport Protocol (HTTP)*** for World Wide Web service, ***Simple Mail Transfer Protocol*** for email, and Telnet for virtual terminals. A separate transport protocol, ***User Datagram Protocol***, is used as

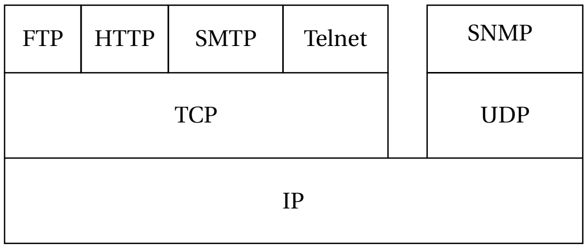


FIGURE 8.21

The Internet service stack.

the basis for the network management services provided by the ***Simple Network Management Protocol***.

8.4.2 Internet Applications

The Internet provides a standard way for an embedded system to act in concert with other devices and with users, such as:

- One of the earliest Internet-enabled embedded systems was the laser printer. High-end laser printers often use IP to receive print jobs from host machines.
- Portable Internet devices can display Web pages, read email, and synchronize calendar information with remote computers.
- A home control system allows the homeowner to remotely monitor and control home cameras, lights, and so on.

Although there are higher-level services that provide more time-sensitive delivery mechanisms for the Internet, the basic incarnation of the Internet is not well suited to hard real-time operations. However, IP is a very good way to let the embedded system talk to other systems. IP provides a way for both special-purpose and standard programs (such as Web browsers) to talk to the embedded system. This non–real-time interaction can be used to monitor the system, set its configuration, and interact with it.

As seen in Section 8.4.1, the Internet provides a wide range of services built on top of IP. Since code size is an important issue in many embedded systems, one architectural decision that must be made is to determine which Internet services will be needed by the system. This choice depends on the type of data service required, such as connectionless versus connection oriented, streaming vs. non-streaming, and so on. It also depends on the application code and its services: does the system look to the rest of the Internet like a terminal, a Web server, or something else?

Application Example 8.1 describes an Internet appliance that runs Java to provide useful services.

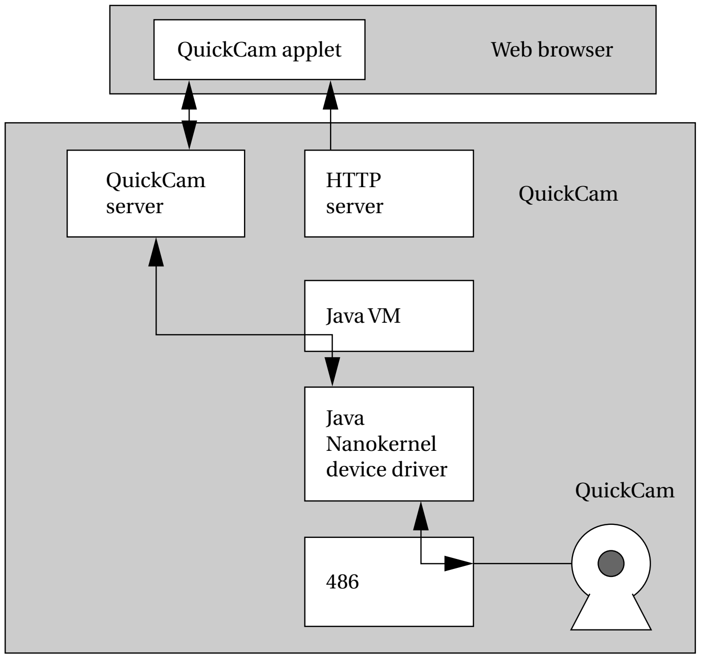
Application Example 8.1

An Internet video camera

Javacam [McD98] is an Internet-accessible video camera that was designed as a demonstration of a Java Nanokernel. The Java Nanokernel is designed to require very little memory. As a result, Javacam can provide an Internet interface with a National Semiconductor NS486SXF microprocessor and 1.5 MB of memory.

Javacam was built from a Connectix QuickCam, a widely available, low-cost video camera for PCs that can send and receive data on a standard PC parallel port.

The illustration below shows how QuickCam operates as a Java applet.



From [McD98].

The HTTP server returns a page containing a piece of Java code that acts as an applet to talk to the device. That Java code running on the Web browser requests an image from the QuickCam server on the QuickCam. The QuickCam server, which executes on top of the Java virtual machine and Java Nanokernel, grabs an image from the QuickCam, performs required transformations, and returns the data to the applet running on the Web browser.

The QuickCam driver communicates with the camera over a parallel port. It provides three basic functions: qc_initialize(); qc_send_command(), which sends commands to the camera; and qc_take_picture(), which returns a picture. Those functions are implemented in C.

8.4.3 Internet Security

Connecting an embedded system to the Internet opens up the system to the same sorts of attacks that are made on PCs and servers every day. However, attacks on embedded systems can destroy not only information but also the physical devices connected to the embedded processor. Dzung et al. [Dzu05] listed several example attacks that caused significant damage:

- A work infected the computer network of the CSX railway, causing all trains in the Washington, DC area to be shut down for a half day.
- A worm disabled the computer-based safety monitoring system at the Davis-Besse nuclear power plant in Ohio.
- A former consultant to a waste water plant in Australia used its computers to release one million liters of sewage into the area waterways.

They point out that security can be enforced at all levels of the network stack. General network security principles can be applied to Internet-enabled embedded systems; various industrial standards also deal with measures specific to industrial networks.

8.5 VEHICLES AS NETWORKS

Modern cars and planes rely on electronics to operate. About one-third of the total cost of an airplane or car comes from its electronics. Electronic systems are used in all aspects of the vehicle—safety-critical control, navigation and systems monitoring, and passenger comfort. These electronic devices are connected using data networks.

Networks are used for a variety of purposes in vehicles, with varying requirements on reliability and performance:

- Vehicle control (steering and brakes in cars, flight control surfaces in airplanes) is the most critical operation in the vehicle since it determines vehicle stability.
- Instruments for the driver or pilot must be reliable but often operate at higher data rates than do the vehicle control systems.
- Crew information systems may provide intercom functions, etc.
- Passenger systems provide entertainment, Internet access, etc.

Early vehicle networks assigned a separate processor to each physical device. Today, network designers tend to combine several functions onto one processor. In cars, the engine controller is the prime candidate for system compute server. This trend plays out more slowly in automobiles, but modern systems assign multiple tasks to a CPU in order to reduce the number of processors and their associated support hardware.

15 The Embedded Product Development Life Cycle (EDLC)

LEARNING OBJECTIVES

- ✓ *Learn about the life-cycle stages in embedded product development, the modelling of the life cycle stages and the objectives for modelling the life cycle*
- ✓ *Learn about the project management and its objectives in embedded product development*
- ✓ *Learn about the techniques for productivity improvement in embedded product development*
- ✓ *Learn the different phases (Need, Conceptualisation, Analysis, Design, Development, Testing, Deployment, Support, Upgrades and Disposal) of the product development life cycle and the activities happening in each stage of the life cycle*
- ✓ *Learn about the different modelling techniques for modelling the stages involved in the embedded product development life cycle*
- ✓ *Learn about the application, advantages and limitations of Linear/Waterfall Model in embedded product development life cycle management*
- ✓ *Learn about the application, advantages and limitations of Iterative/Incremental or Fountain Model in embedded product development life cycle management*
- ✓ *Learn about the application, advantages and limitations of prototyping/evolutionary model in embedded product development life cycle management*
- ✓ *Learn about the application, advantages and limitations of Spiral Model in embedded product development life cycle management*

Just imagine about your favourite chicken dish that your mom served during a holiday lunch. Have you ever thought of the effort behind preparing the delicious chicken dish? Frankly speaking No, Isn't it? Okay let's go back and analyse how the yummy chicken in the chicken stall became the delicious chicken dish served on the dining table. One fine morning Mom thinks "Wohh!! It's Sunday and why can't we have a special lunch with our favourite chicken dish." Mom tells this to Papa and he agrees on it and together they decide how much quantity of chicken is required for the four-member family (may be based on past experience) and lists out the various ingredients required for preparing the dish. Now the list is ready and papa checks his purse to ensure whether the item list is at the reach of the money in his purse. If not, both of them sit together and make some reductions in the list. Papa goes to the market to buy the items in the list. Finds out a chicken stall where quality chicken is selling at a nominal rate. Procures all the items and returns back home with all the items in the list and hands over the packet to Mom. Mom starts cleaning the chicken pieces and chopping the ingredient/vegetables and then puts them in a vessel and starts boiling them. Mom may go on to preparing other side dishes or rice and comes back to the chicken dish preparation at regular intervals to add the necessary ingredients (like chicken masala, chilly powder, coriander powder, tamarind, coconut oil, salt, etc.) in time and checks whether all ingredients are in correct proportion, if not adjust them to the required proportion. Papa gives an overall supervision to the preparation and informs mom if she forgot to add any ingredients and also helps her in verifying the proportion of the ingredients. Finally the fragrance of spicy boiled chicken comes out of the vessel and mom realises that chicken dish is ready. She takes it out from the stove and adds the final taste building ingredients. Mom tastes a small piece from the dish and ensures that it meets the regular quality. Now the dish is ready to serve. Mom takes the responsibility of serving the same to you and what you need to do is taste it and tell her how tasty it is. If you closely observe these steps with an embedded product developer's mind, you can find out that this simple chicken dish preparation contains various complex activities like overall management, development, testing and releasing. Papa is the person who did the entire management activity starting from item procurement to giving overall supervision. Mom is responsible for developing the dish, testing the dish to certain extent and serving the dish (release management) and finally you are the one who did the actual field test. All embedded products will have a design and development life cycle similar to our chicken dish preparation example, with management, design, development, test and release activities.

15.1 WHAT IS EDLC?

Embedded Product Development Life Cycle (Let us call it as EDLC, though it is not a standard and universal term) is an 'Analysis-Design-Implementation' based standard problem solving approach for Embedded Product Development. In any product development application, the first and foremost step is to figure out what product needs to be developed (analysis), next you need to figure out a good approach for building it (design) and last but not least you need to develop it (implementation).

15.2 WHY EDLC

EDLC is essential for understanding the scope and complexity of the work involved in any embedded product development. EDLC defines the interaction and activities among various groups of a product development sector including project management, system design and development (hardware, firmware and enclosure design and development), system testing, release management and quality assurance. The standards imposed by EDLC on a product development makes the product, developer independent in terms of standard documents and it also provides uniformity in development approaches.

15.3 OBJECTIVES OF EDLC

The ultimate aim of any embedded product in a commercial production setup is to produce marginal benefit. Marginal benefit is usually expressed in terms of Return on Investment (ROI). The investment for a product development includes initial investment, manpower investment, infrastructure investment, etc. A product is said to be profitable only if the turnover from the selling of the product is more than that of the overall investment expenditure. For this, the product should be acceptable by the end user and it should meet the requirements of end user in terms of quality, reliability and functionality. So it is very essential to ensure that the product is meeting all these criteria, throughout the design, development, implementation and support phases. Embedded Product Development Life Cycle (EDLC) helps out in ensuring all these requirements. EDLC has three primary objectives, namely

1. Ensure that high quality products are delivered to end user.
2. Risk minimisation and defect prevention in product development through project management.
3. Maximise the productivity.

15.3.1 Ensuring High Quality for Products

The primary definition of quality in any embedded product development is the Return on Investment (ROI) achieved by the product. The expenses incurred for developing the product may fall in any of the categories; initial investment, developer recruiting, training, or any other infrastructure requirement related. There will be some budgetary and cost allocation given to the development of the product and it will be allocated by some higher officials based on the assumption that the product will produce a good return or a return justifying the investment. The budget allocation might have done after studying the market trends and requirements of the product, competition, etc. EDLC must ensure that the development of the product has taken account of all the qualitative attributes of the embedded system. The qualitative attributes are discussed separately in an earlier chapter of this book. Please refer back for the same.

15.3.2 Risk Minimisation and Defect Prevention through Management

You may be thinking what is the significance of project management, or why project management is essential in product development. Nevertheless it is an additional expenditure to the project! If we look back to the chicken dish example, we can find out that the management activity from dad is essential in the beginning phase but in the preparation phase it can be handled by mom itself. There are projects in embedded product development which requires 'loose' or 'tight' project management. If the product development project is a simple one, a senior developer itself can take charge of the management activity and no need for a skilled project manager to look after this with dedicated effort throughout the development process, but there should be an overall supervision from a skilled project management team for ensuring that the development process is going in the right direction. Projects which are complex and requires timeliness should have a dedicated and skilled project management part and hence they are said to be "tightly" bounded to project management. '*Project management is essential for predictability, co-ordination and risk minimisation*'. Whenever a product development request comes, an estimate on the duration of the development and deployment activity should be given to the end user/client. The timeframe may be expressed in number of person days PDS (The effort in terms of single person working for this much days) or 'X person for X week (e.g. 2 person 2 week) etc. This is one aspect of predictability. The management team might have reached on this estimate based on past experience in handling similar project or on the analysis of work summary or data available for a similar project, which was logged in using an activity tool. Resource (Developer) allocation is another aspect of predictability in management. Resource allocations like how many resources should work for this project for how many days, how many resources are critical with respect to the work handling by them and how many backups required for the resources to overcome a critical situation where a resource is not available (Risk minimisation). Resource allocation is critical and it is having a direct impact on investment.

The communication aspect of the project management deals with co-ordination and interaction among resources and client from which the request for the product development aroused. Project management adds an extra cost on the budget but it is essential for ensuring the development process is going in the right direction and the schedules of the development activity are meeting. Without management, the development work may go beyond the stipulated time frame (schedule slipping) and may end up in a product which is not meeting the requirements from the client side, as a result re-works should be done to rectify the possible deviations occurred and it will again put extra cost on the development. Project management makes the proverb "A stitch in time saves nine" meaningful in an embedded product development. Apart from resource allocation, project management also covers activities like task allocation, scheduling, monitoring and project tracking. Computer Assisted Software Engineering (CASE) Tools and Gantt charts help the manager in achieving this. Microsoft® Project Tool is a typical example of CASE tool for project management.

15.3.3 Increased Productivity

Productivity is a measure of efficiency as well as Return on Investment (ROI). One aspect of productivity covers how many resources are utilised to build the product, how much investment required, how much time is taken for developing the product, etc. For example, the productivity of a system is said to be doubled if a product developed by a team of 'X' members in a period of 'X' days is developed by another team of 'X/2' members in a period of 'X' days or by a team of 'X' members in a period of 'X/2' days. This productivity measurement is based on total manpower efficiency. Productivity in terms of Returns is said to be increased, if the product is capable of yielding maximum returns with reduced investment. Saving manpower effort will definitely result in increased productivity. Usage of automated tools, wherever possible, is recommended for this. The initial investment on tools may be an additional burden in terms of money, but it will definitely save efforts in the next project also. It is a one-time investment. "Pay once use many time". Another important factor which can help in increased productivity is "re-usable effort". Some of the works required for the current product development may have some common features which you built for some of the other product development in the projects you executed before. Identify those efforts and design the new product in such a way that it can directly be plugged into the new product without any additional effort. (For example, the current product you are developing requires an RS-232C serial interface and one of the product you already developed have the same feature. Adapt this part directly from the existing product in terms of the hardware and firmware required for the same.) This will definitely increase the productivity by reducing the development effort. Another advised method for increasing the productivity is by using resources with specific skill sets which matches the exact requirement of the entire or part of the product (e.g. Resource with expertise in Bluetooth technology for developing a Bluetooth interface for the product). This reduces the learning time taken by a resource, who does not have prior expertise in the particular feature or domain. This is not possible in all product development environments, since some of the resources with the desired skill sets may be engaged with some other work and releasing them from the current work is not possible. Recruiting people with desired skill sets for the current product development is another option; this is worth only if you expect to have more work on the near future on the same technology or skill sets. Use of Commercial Off-the-Shelf Components (COTS) wherever possible in a product is a very good way of reducing the development effort and thereby increasing the productivity (Refer back to the topic 'Commercial Off-the-shelf components' given in Chapter 2 for more details on COTS). COTS component is a ready to use component and you can use the same as plug-in modules in your product. For example, if the product under development requires a 10 base T Ethernet connectivity, you can either implement the same in your product by using the TCP/IP chip and related components or can use a readily available TCP/IP full functional plug-in module. The second approach will save effort and time. EDLC should take all these aspects into consideration to provide maximum productivity.

15.4 DIFFERENT PHASES OF EDLC

The life cycle of a product development is commonly referred to as models. Model defines the various phases involved in the life cycle. The number of phases involved in a model may vary according to the complexity of the product under consideration. A typical simple product contains five minimal phases namely: 'Requirement Analysis', 'Design', 'Development and Test', 'Deployment' and 'Maintenance'. The classic Embedded Product Life Cycle Model contains the phases: 'Need', 'Conceptualisation', 'Analysis', 'Design', 'Development and Testing', 'Deployment', 'Support', 'Upgrades' and 'Retirement/ Disposal' (Fig. 15.1). In a real product development environment, the phases given in this model may vary and can have submodels or the models contain only certain important phases of the classic model. As mentioned earlier, the number of phases involved in the EDLC model depends on the complexity of the product. The following section describes each phases of the classic EDLC model in detail.

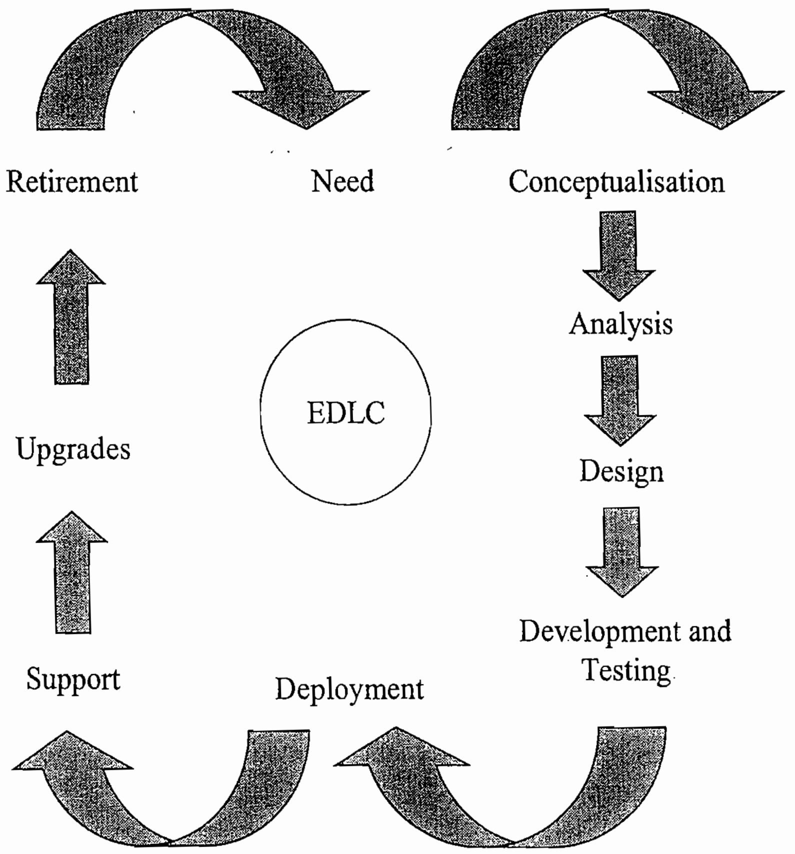
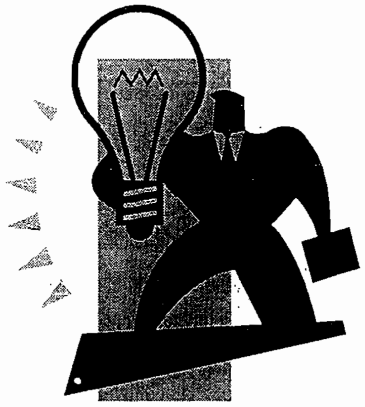


Fig. 15.1 **Classic Embedded Product Development Life Cycle Model**

15.4.1 Need

Any embedded product evolves as an output of a *'Need'*. The need may come from an individual or from the public or from a company (Generally speaking from an end user/client). 'Need' should be articulated to initiate the Product Development Life Cycle and based on the need for the product, a 'Statement of Need' or 'Concept Proposal' is prepared. The 'Concept Proposal' must be reviewed by the senior management and funding agency and should get necessary approval. Once the proposal gets approval, it goes to a product development team, which can either be an organisation from which the need arise or a third party product development/service company (If a product development company approaches a client/sponsor with the idea of a product, the business needs for the product will be prepared by the company itself). The product development need can be visualised in any one of the following three needs.



15.4.1.1 New or Custom Product Development The need for a product which does not exist in the market or a product which acts as competitor to an existing product in the current market will lead to the development of a completely new product. The product can be a commercial requirement or an individual requirement or a specific organisation's requirement. Example for an Embedded Product which acts a competitor in the current market is 'Mobile handset' which is going to be developed by a new company apart from the existing manufacturers. A PDA tailored for any specific need is an example for a custom product.

15.4.1.2 Product Re-engineering The embedded product market is dynamic and competitive. In order to sustain in the market, the product developer should be aware of the general market trends, technology changes and the taste of the end user and should constantly improve an existing product by incorporating new technologies and design changes for performance, functionalities and aesthetics. Re-engineering a product is the process of making changes in an existing product design and launching it as a new version. It is generally termed as product upgrade. Re-engineering an existing product comes as a result of the following needs.

1. Change in Business requirements
2. User Interface Enhancements
3. Technology Upgrades

15.4.1.3 Product Maintenance Product maintenance 'need' deals with providing technical support to the end user for an existing product in the market. The maintenance request may come as a result of product non-functioning or failure. Product maintenance is generally classified into two categories; Corrective maintenance and Preventive maintenance. Corrective maintenance deals with making corrective actions following a failure or non-functioning. The failure can occur due to damages or non-functioning of components/parts of the product and the corrective maintenance replaces them to make the product functional again. Preventive maintenance is the scheduled maintenance to avoid the failure or non-functioning of the product.

15.4.2 Conceptualisation

Conceptualisation is the 'Product Concept Development Phase' and it begins immediately after a Concept Proposal is formally approved. Conceptualisation phase defines the scope of the concept, performs cost benefit analysis and feasibility study and prepares project management and risk management plans. Conceptualisation can be viewed as a phase shaping the “Need” of an end-user or convincing the end user, whether it is a feasible product and how this product can be realised (Fig. 15.2).

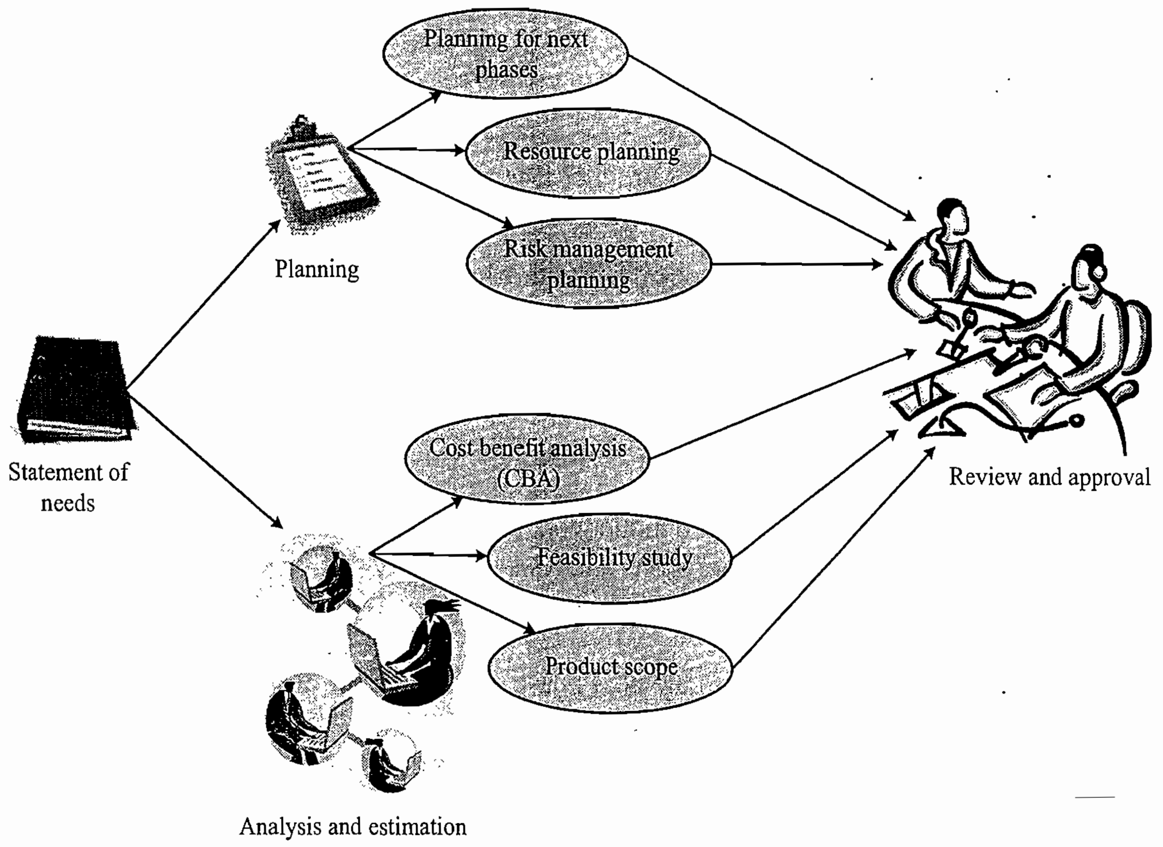


Fig. 15.2 **Various activities involved in Conceptualisation Phase**

The ‘Conceptualisation’ phase involves two types of activities, namely; ‘Planning Activity’ and ‘Analysis and Study Activity’. The Analysis and Study Activities are performed to understand the opportunity of the product in the market (for a commercial product). The following are the important ‘Analysis and Study activities’ performed during ‘Conceptualisation Phase’.

15.4.2.1 Feasibility Study Feasibility study examines the need for the product carefully and suggests one or more possible solutions to build the need as a product along with alternatives. Feasibility study analyses the Technical (Technical Challenges, limitations, etc.) as well as financial feasibility (Financial requirements, funding, etc.) of the product under consideration.

15.4.2.2 Cost Benefit Analysis (CBA) The Cost Benefit Analysis (CBA) is a means of identifying, revealing and assessing the total development cost and the profit expected from the product. It is somewhat similar to the loss-gain analysis in business term except that CBA is an assumption oriented approach based on some rule of thumb or based on the analysis of data available for similar products developed in the past. In summary, CBA estimates and totals up the equivalent money value of the benefits and costs of the product under consideration and give an idea about the worthwhile of the product. Some common underlying principles of Cost Benefit Analysis are illustrated below.

Common Unit of Measurement In order to make the analysis sensible and meaningful, all aspects of the product, either plus points or minus points should be expressed in terms of a common unit. Since the ultimate aim of a product is "Marginal Profit" and the marginal profit is expressed in terms of money in most cases, 'money' is taken as the common unit of measurement. Hence all benefits and costs of the product should be expressed in terms of money. Money in turn may be represented by a common currency. It can be Indian Rupee (INR) or US Dollar (USD), etc.

Market choice based benefit measurement Ensure that the product cost is justifying the money (value for money), the end user is spending on the product to purchase, for a commercial product.

Targeted end users For a commercial product it is very essential to understand the targeted end-users for the product and their tastes to give them the best in the industry.

15.4.2.3 Product Scope Product scope deals with what is in scope (what all functionalities or tasks are considered) for the product and what is not in scope (what all functionalities or tasks are not considered) for the product. Product scope should be documented properly for future reference.

15.4.2.4 Planning Activities The planning activity covers various plans required for the product development, like the plan and schedule for executing the next phases following conceptualisation, **Resource Planning** – How many resources should work on the project, **Risk Management Plans** – The technical and other kinds of risks involved in the work and what are the mitigation plans, etc. At the end of the conceptualisation phase, the reports on 'Analysis and Study Activities' and 'Planning Activities' are submitted to the client/project sponsor for review and approval, along with any one of the following recommendation.

1. The product is feasible; proceed to the next phase of the product life cycle.
2. The product is not feasible; scrap the project

The executive committee, to which the various reports are submitted, will review the same and will communicate the review comments to the officials submitted the conceptualisation reports and will give the green signal to proceed, if they are satisfied with the analysis and estimates.

15.4.3 Analysis

Requirements Analysis Phase starts immediately after the documents submitted during the 'Conceptualisation' phase is approved by the client/sponsor of the project. Documentation related to user requirements from the Conceptualisation phase is used as the basis for further user need analysis and the development of detailed user requirements. Requirement analysis is performed to develop a detailed functional model of the product under consideration. During the Requirements Analysis Phase, the product is defined in detail with respect to the inputs, processes, outputs, and interfaces at a functional level. Requirement Analysis phase gives emphasis on determining 'what functions must be performed by the product' rather than how to perform those functions. The various activities performed during 'Requirement Analysis' phase is illustrated in Fig. 15.3.

15.4.3.1 Analysis and Documentation The analysis and documentation activity consolidates the business needs of the product under development and analyses the purpose of the product. It addresses various functional aspects (product features and functions) and quality attributes (non-functional requirements) of the product, defines the functional and data requirements and connects the function-

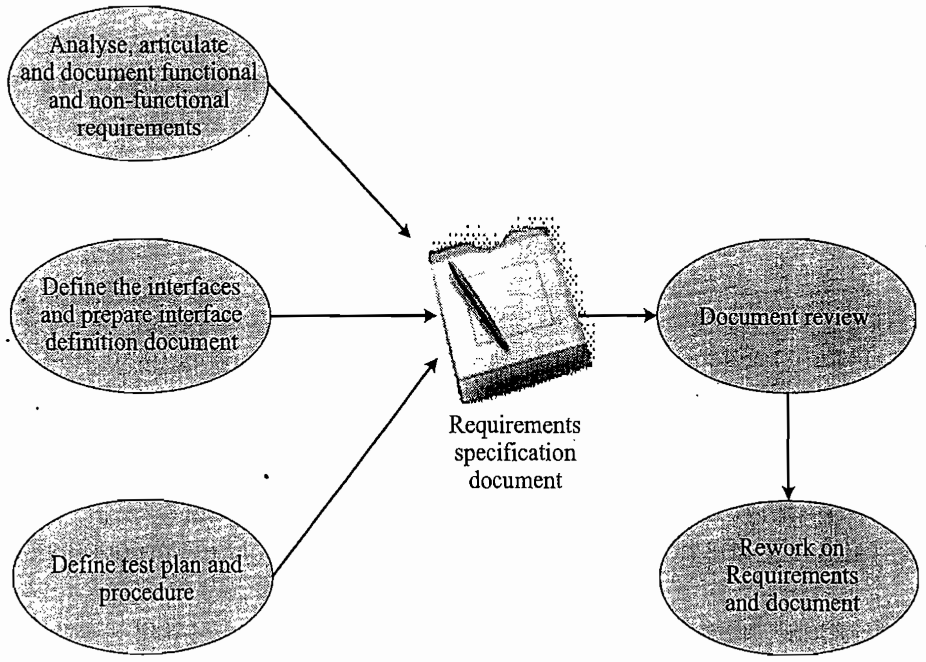

Analyse, articulate and document functional and non-functional requirements
Define the interfaces and prepare interface definition document
Define test plan and procedure
Requirements specification document
Document review
Rework on Requirements and document

Fig. 15.3 **Various activities involved in Requirements Analysis Phase**

al requirements with the data requirements. In summary, the analysis activities address all business functionalities of the product and develop a logical model describing the fundamental processes and the data required to support the functionalities. The various requirements that need to be addressed during the requirement analysis phase for a product under consideration are listed below.

1. Functional capabilities like performance, operational characteristics (Refer to Chapter 3 for details on operational characteristics), etc.
2. Operational and non-operational quality attributes (Refer to Chapter 3 for Quality attributes of embedded products)
3. Product external interface requirements
4. Data requirements
5. User manuals
6. Operational requirements
7. Maintenance requirements
8. General assumptions

15.4.3.2 Interface Definition and Documentation The embedded product under consideration may be a standalone product or it can be a part of a large distributed system. If it is a part of another system there should be some interface between the product and the other parts of the distributed system. Even in the case of standalone products there will be some standard interfaces like serial, parallel etc as a provision to connect to some standard interfaces. The interface definition and documentation activity should clearly analyse on the physical interface (type of interface) as well as data exchange through these interfaces and should document it. It is essential for the proper information flow throughout the life cycle.

15.4.3.3 Defining Test Plan and Procedures Identifies what kind of tests are to be performed and what all should be covered in testing the product. Define the test procedures, test setup and test environment. Create a master test plan to cover all functional as well as other aspects of the product and document the scope, methodology, sequence and management of all kinds of tests. It is essential for ensuring the total quality of the product. The various types of testing performed in a product development are listed below.

1. **Unit testing**—Testing each unit or module of the product independently for required functionality and quality aspects
2. **Integration testing**—Integrating each modules and testing the integrated unit for required functionality
3. **System testing**—Testing the functional aspects/product requirements of the product after integration. System testing refers to a set of different tests and a few among them are listed below.
   - **Usability testing**—Tests the usability of the product
   - **Load testing**—Tests the behaviour of the product under different loading conditions.
   - **Security testing**—Testing the security aspects of the product
   - **Scalability testing**—Testing the scalability aspect of the product
   - **Sanity testing**—Superficial testing performed to ensure that the product is functioning properly
   - **Smoke testing**—Non exhaustive test to ensure that the crucial requirements for the product are functioning properly. This test is not giving importance to the finer details of different functionalities
   - **Performance testing**—Testing the performance aspects of the product after integration
   - **Endurance testing**—Durability test of the product
4. **User acceptance testing**—Testing the product to ensure it is meeting all requirements of the end user.

At the end, all requirements should be put in a template suggested by the standard (e.g. ISO-9001) Process Model (e.g. CMM Level 5) followed for the Quality Assurance. This document is referred as 'Requirements Specification Document'. It is sent out for review by the client and the client, after reviewing the requirements specification document, communicates back with the review comments. According to the review comments more requirement analysis may have to be performed and re-work required is performed on the initial 'Requirement Specification Document'.

15.4.4 Design

Product 'Design phase' deals with the entire design of the product taking the requirements into consideration and it focuses on 'how' the required functionalities can be delivered to the product. The design phase identifies the application environment and creates an overall architecture for the product. Product design starts with '*Preliminary Design/High Level Design*'. Preliminary design establishes the top-level architecture for the product, lists out the various functional blocks required for the product, and defines the inputs and outputs for each functional block. The functional block will look like a 'black box' at this design point, with only inputs and outputs of the block being defined. On completion, the 'Preliminary Design Document (PDD) is sent for review to the end-user/client who come up with a need for the product. If the end-user agrees on the preliminary design, the product design team can take the work to the next level – *'Detailed Design'*. Detailed design generates a detailed architecture, identifies and lists out the various components for each functional block, the inter connection among various functional blocks, the control algorithm requirements, etc. The 'Detailed Design' also needs to be reviewed and get approved by the end user/customer. If the end user wants modifications on the design, it can be informed to the design team through review comments.

An embedded product is a real mix of hardware, embedded firmware and enclosure. Hence both *Preliminary Design* and *Detailed Design* should take the design of these into consideration. For traditional embedded system design approach, the functional requirements are classified into either hardware or firmware at an early stage of the design and the hardware team works on the design of the required hardware, whereas the software team works on the design of the embedded firmware. Today the scenario is totally changed and a hardware software co-design approach is used. In this approach, the functional requirements are not broken down into hardware and software, instead they are treated as system requirements and the partitioning of system requirements into either hardware or software is performed at a later stage of the design based on hardware-software trade-offs for implementing the required functionalities. The hardware and software requirements are modelled using different modelling techniques. Please refer to Chapter 7 for more details on hardware-software co-design and program models used in hardware-software co-design. The other activities performed during the design phase are 'Design of Operations and maintenance manual' and 'Design of Training materials' (Fig. 15.4). The operations manual is necessary for educating the end user on 'how to operate the product' and the maintenance manual is essential for providing information on how to handle the product in situations like non-functioning, failure, etc.

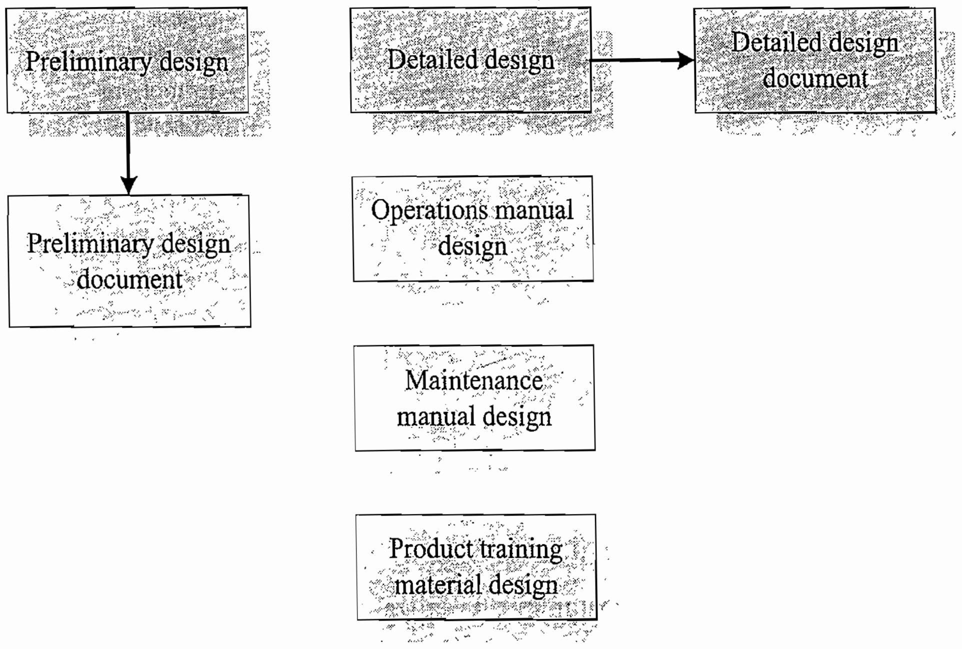

Preliminary design
Preliminary design document
Detailed design
Detailed design document
Operations manual design
Maintenance manual design
Product training material design

Fig. 15.4 **Various Activities involved in Design Phase**

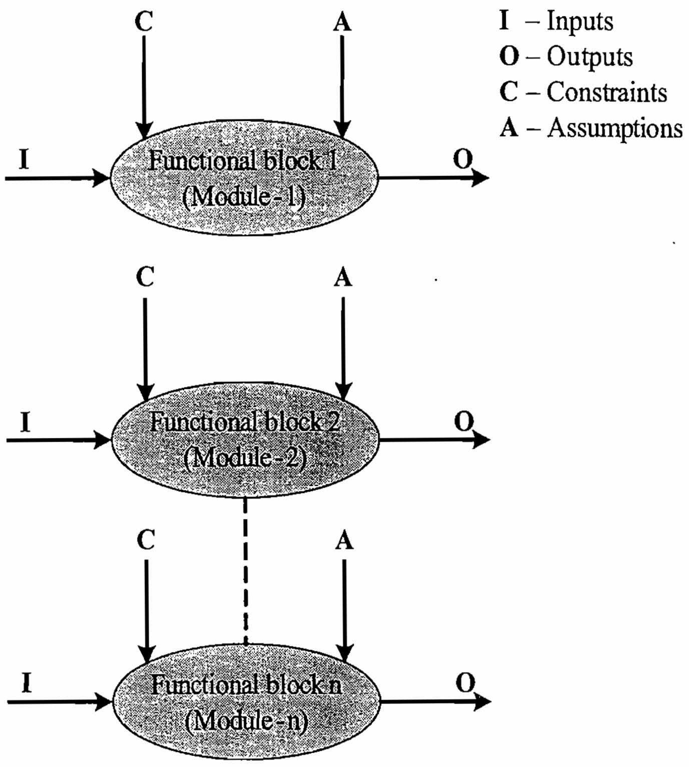

C
A
I
O
Functional block 1 (Module-1)
Functional block 2 (Module-2)
Functional block n (Module-n)
I – Inputs
O – Outputs
C – Constraints
A – Assumptions

Fig. 15.5 **Preliminary Design Illustration**

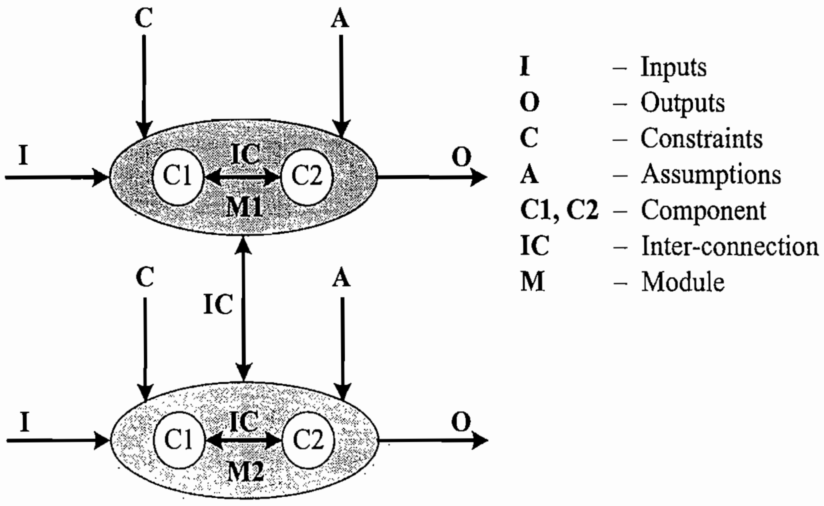
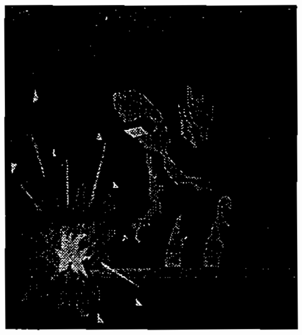


Fig. 15.6 **Detailed Design Illustration**

15.4.5 Development and Testing



The 'Development Phase' transforms the design into a realizable product. For this the detailed specifications generated during the design phase are translated into hardware and firmware. During development phase, the installation and setting up of various development tools is performed and the product hardware and firmware is developed using different tools and associated production setup. The development activities can be partitioned into embedded hardware development, embedded firmware development and product enclosure development. Embedded hardware development refers to the development of the component placement platform – PCB using CAD tools and its fabrication using CAM Tools. Embedded firmware development is performed using the embedded firmware development tools. The mechanical enclosure design is performed with CAD Tools like Solid Works, AutoCAD, etc. "Look and Feel" of a commercial product plays an important role on its market demand. So the entire product design and development should concentrate on the product aesthetics (Size, Weight, Appearance, etc.) and special requirements like protection shielding, etc.

Component procurement also carried out during this phase. As mentioned in one of the earlier chapters, some of the components may have "*lead time* – The time required for the component to deliver after placing the purchase order for the component" and the component procurement should be planned well in advance to avoid the possible delay in the product development. The other important activities carried out during this phase are preparation of test case procedures, preparation of test files and development of detailed user, deployment, operations and maintenance manual. The embedded firmware development may be divided into various modules and the firmware (code) review for each module as well as the schematic diagram and layout file review for hardware is carried out during the development phase and the changes required should be incorporated in the development from time to time.

The testing phase can be divided into independent testing of firmware and hardware (Unit Testing), testing of the product after integrating the firmware and hardware (Integration Testing), testing of the whole system on a functionality and non-functionality basis (System Testing) and testing of the product against all acceptance criteria mentioned by the client/end user for each functionality (User Acceptance Testing). The Unit testing deals with testing the embedded hardware and its associated modules, the different modules of the firmware for their functionality independently. A test plan is prepared for the unit testing (Unit Test plan) and test cases (Unit Test cases) are identified for testing the functionality of the product as per the design. Unit test is normally performed by the hardware developer or firmware developer as part of the development phase. Depending on the product complexity, rest of the tests can be considered as the activities performed in the 'Testing phase' of the product. Once the hardware and firmware are unit tested, they are integrated using various firmware integration techniques and the integrated product is tested against the required functionalities. An integration test plan is prepared for this and the test cases for integration testing (Integration test cases) is developed. The Integration testing ensures that the functionality which is tested working properly in an independent mode remains working properly in an integrated product (Hardware + Firmware). The integration test results are logged in and the firmware/hardware is reworked against any flaws detected during the Integration testing. The purpose of integration testing is to detect any inconsistencies between the firmware modules units that are integrated together or between any of the firmware modules and the embedded hardware. Integration testing of various modules, hardware and firmware is done at the end of development phase depending on the readiness of hardware and firmware.

The system testing follows the Integration testing and it is a set of various tests for functional and non-functional requirements verification. System testing is a kind of black box testing, which doesn't require any knowledge on the inner design logic or code for the product. System testing evaluates the product's compliance with the requirements. User acceptance test or simply Acceptance testing is the product evaluation performed by the customer as a condition of the product purchase. During user Acceptance testing, the product is tested against the acceptance value set by customer/end user for each requirement (e.g. The loading time for the application running on the PDA device is less than 1 millisecond ☺). The deliverables from the 'Design and Testing' phase are firmware source code, firmware binaries, finished hardware, various test plans (Hardware and Firmware Unit Test plans, Integration Test plan, System Test plan and Acceptance Test plan), test cases (Hardware and Firmware Unit Test cases, Integration Test cases, System Test cases and Acceptance Test cases) and test reports (Hardware and Firmware Unit Test Report, Integration Test Report, System Test Report and Acceptance Test Report).

15.4.6 Deployment

Deployment is the process of launching the first fully functional model of the product in the market (for a commercial embedded product) or handing over the fully functional initial model to an end user/client. It is also known as *First Customer Shipping* (*FCS*). During this phase, the product modifications as per the various integration tests are implemented and the product is made operational in a production environment. The 'Deployment Phase' is initiated after the system is tested and accepted (User Acceptance Test) by the end user. The important tasks performed during the Deployment Phase are listed below.

15.4.6.1 Notification of Product Deployment Whenever the product is ready to launch in the market, the launching ceremony details should be communicated to the stake holders (all those who are related to the product in a direct or indirect way) and to the public if it is a commercial product. The notifications can be sent out through e-mail, media, etc. mentioning the following in a few words.

1. Deployment Schedule (Date Time and Venue)
2. Brief description about the product
3. Targeted end users
4. Extra features supported with respect to an existing product (for upgrades of existing model and new products having other competitive products in the market)
5. Product support information including the support person name, contact number, contact mail ID, etc.

15.4.6.2 Execution of Training Plan Proper training should be given to the end user to get them acquainted with the new product. Before launching the product, conduct proper training as per the training plan developed during the earlier phases. Proper training will help in reducing the possible damages to the product as well as the operating person, including personal injuries and product mal-functioning due to inappropriate usage. User manual will help in understanding the product, its usage and accessing its functionalities to certain extend.

15.4.6.3 Product Installation Instal the product as per the installation document to ensure that it is fully functional. As a typical example take the case of a newly purchased mobile handset. The user manual accompanying it will tell you clearly how to instal the battery, how to charge the battery, how many hours the battery needs to be charged, how the handset can be switched on/off, etc.

15.4.6.4 Product post-Implementation Review Once the product is launched in the market, conduct a post-implementation review to determine the success of the product. The ultimate aim behind post-implementation review is to document the problems faced during installation and the solutions adopted to overcome them which will act as a reference document for future product development. It also helps in understanding the customer needs and the expectations of the customer on the next version of the product.

15.4.7 Support

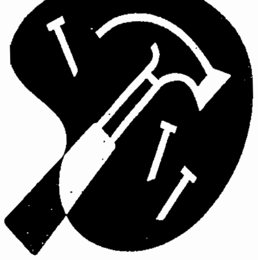
The support phase deals with the operations and maintenance of the product in a production environment. During this phase all aspects of operations and maintenance of the product are covered and the

product is scrutinised to ensure that it meets the requirements put forward by the end user/client. 'Bugs (product mal-functioning or unexpected behaviour or any operational error)' in the products may be observed and reported during the operations phase. Support should be provided to the end user/client to fix the bugs in the product. The support phase ensures that the product meets the user needs and it continues functioning in the production environment. The various activities involved in the 'Support' phase are listed below.

15.4.7.1 Set up a Dedicated Support Wing The availability of certain embedded products in terms of product functioning in production environment is crucial and they may require 24x7 support in case of product failure or malfunctioning. For example the patient monitoring system used in hospitals is a very critical product in terms of functioning and any malfunctioning of the product requires immediate technical attention/support from the supplier. Set up a dedicated support wing (customer care unit) and ensure high quality service is delivered to the end user. 'After service' plays significant role in product movement in a commercial market. If the manufacturer fails to provide timely and quality service to the end user, it will naturally create the talk '*the product is good but the after service is very poor*' among the end users and people will refrain from buying such products and will opt for competitive products which provides more or less same functionalities and high quality service. The support wing should be set up in such a way that they are easily reachable through e-mail, phone, fax, etc.

15.4.7.2 Identify Bugs and Areas of Improvement None of the products are bug-free. Even if you take utmost care in design, development and implementation of the product, there can be bugs in it and the bugs reveal their real identity when the conditions are favouring them, similar to the harmless microbes living in human body getting turned into harmful microbes when the body resistance is weak. You may miss out certain operating conditions while design or development phase and if the product faces such an operating condition, the product behaviour may become unexpected and it is addressed with the familiar nick name '*Bug*' by a software/product engineer. Since the embedded product market is highly competitive, it is very essential to stay connected with the end users and know their needs for the survival of the product. Give the end user a chance to express their views on the product and suggestions, if any, in terms of modifications required or feature enhancements (areas of improvement), through user feedbacks. Conduct product specific surveys and collect as much data as possible from the end user.

15.4.8 Upgrades



The upgrade phase of product development deals with the development of upgrades (new versions) for the product which is already present in the market. Product upgrade results as an output of major bug fixes or feature enhancement requirements from the end user. During the upgrade phase the system is subject to design modification to fix the major bugs reported or to incorporate the new feature addition requirements aroused during the support phase. For embedded products, the upgrades may be for the product resident firmware or for the hardware embedding the firmware. Some bugs may be easily fixed by modifying the firmware and it is known as firmware up-gradation. Some feature enhancements can also be performed easily by mere firmware modification. The product resident firmware will have a version number which starts with version say 1.0 and after each firmware modification or bug fix, the firmware version is changed accordingly (e.g. version 1.1). Version numbering is essential for backward traceability. Releasing of upgrades is managed through release management. (Embedded Product Engineers might have used the term "Today I have a release" at least once in their life). Certain feature enhancements and bug fixes require hardware modification and they are generally termed as hardware upgrades. Firmware also needs to be modified according to the hardware enhancements. The upgraded product appears in the market with the same name as that of the initial product with a different version number or with a different name.

15.4.9 Retirement/Disposal

We are living in a dynamic world where everything is subject to rapid changes. The technology you feel as the most advanced and best today may not be the same tomorrow. Due to increased user needs and revolutionary technological changes, a product cannot sustain in the market for a long time. The disposal/ retirement of a product is a gradual process. When the product manufacture realises that there is another powerful technology or component available in the market which is most suitable for the production of the current product, they will announce the current product as obsolete and the newer version/upgrade of the same is going to be released soon. The retirement/disposition phase is the final phase in a product development life cycle where the product is declared as obsolete and discontinued from the market. There is no meaning and monetary benefit in continuing the production of a device or equipment which is obsolete in terms of technology and aesthetics. The disposition of a product is essential due to the following reasons.

1. Rapid technology advancement
2. Increased user needs

Some products may get a very long 'Life Time' in the market, beginning from the launch phase to the retirement phase and during this time the product will get reasonably good amount of maturity and stability and it may be able to create sensational waves in the market (e.g. Nokia 3310 Mobile Handset; Launched in September 2000 and discontinued in early 2005, more than 4 years of continued presence in the market with 126 million pieces sold). Some products get only small 'Life Time' in the market and some of them even fails to register their appearance in the market.

By now we covered all the phases of embedded product development life cycle. We can correlate the various phases of the product development we discussed so far to the various activities involved in our day-today life. Whenever you are a child you definitely have the ambition to become an earning person like your mom/dad (There may be exceptions too ☺). The need for a job arises here. To get a good job you must have a good academic record and hence you should complete the schooling with good marks and should perform very well in interviews. Finally you are managed to get a good job and your professional career begins here. You may get promotions to next senior level depending on your performance. At last that day comes—Your retirement day from your professional life.

15.5 EDLC APPROACHES (MODELING THE EDLC)

The term 'Modelling' in Embedded Product Development Life Cycle refers to the interconnection of various phases involved in the development of an embedded product. The various approaches adopted or models used in Modelling EDLC are described below.

15.5.1 Linear or Waterfall Model

Linear or waterfall approach is the one adopted in most of the olden systems and in this approach each phase of EDLC is executed in sequence. The linear model establishes a formal analysis and design methodology with highly structured development phases. In linear model, the various phases of EDLC are executed in sequence and the flow is unidirectional with output of one phase serving as the input to the next phase. In the linear model all activities involved in each phase are well documented, giving an insight into what should be done in the next phase and how it can be done. The feedback of each phase is available locally and only after they are executed. The linear model implements extensive review mechanisms to ensure the process flow is going in the right direction and validates the effort during a phase. One significant feature of linear model is that even if you identify bugs in the current design, the corrective actions are not implemented immediately and the development process proceeds with the current design. The fixes for the bugs are postponed till the support phase, which accompanies the deployment phase. The major advantage of 'Linear Model' is that the product development is rich in terms of documentation, easy project management and good control over cost and schedule. The major drawback of this approach is that it assumes all the analysis can be done and everything will be in right place without doing any design or implementation. Also the risk analysis is performed only once throughout the development and risks involved in any changes are not accommodated in subsequent phases, the working product is available only at the end of the development phase and bug fixes and corrections are performed only at the maintenance/support phase of the life cycle. 'Linear Model' (Fig. 15.7) is best

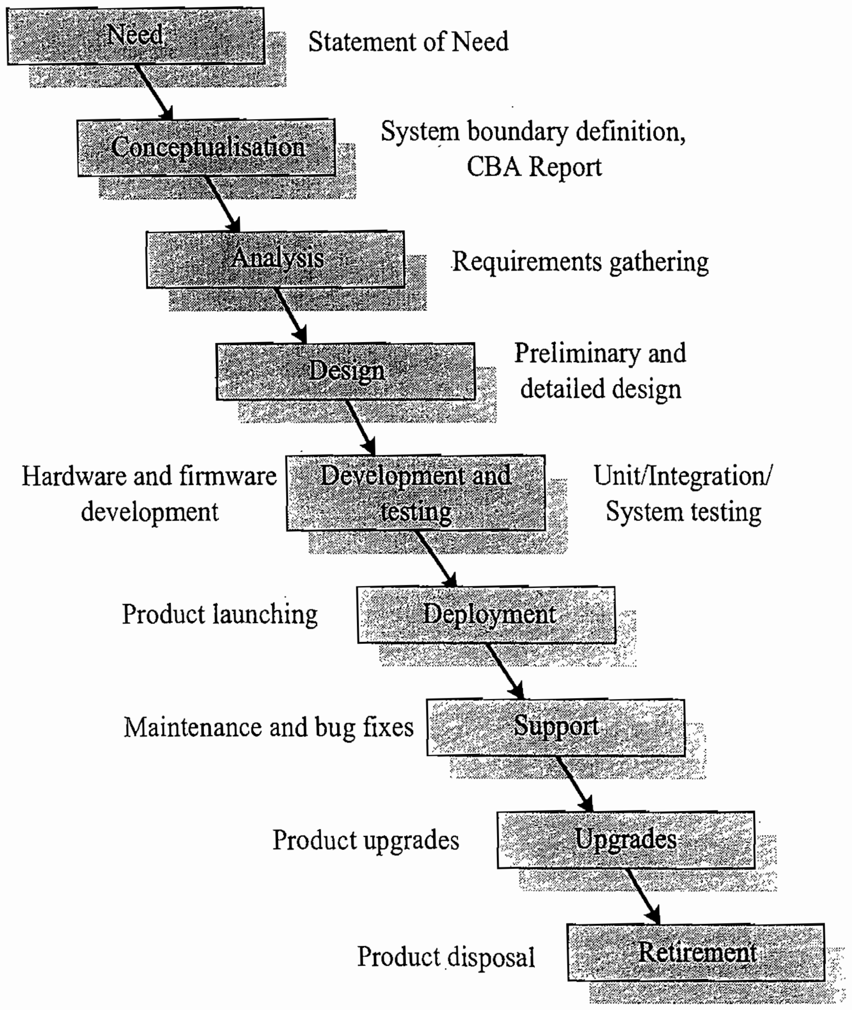


Fig. 15.7 Linear (Waterfall) EDLC Model

suited for product developments, where the requirements are well defined and within the scope, and no change requests are expected till the completion of the life cycle.

15.5.2 Iterative/Incremental or Fountain Model

Iterative or fountain model follows the sequence—Do some analysis, follow some design, then some implementation. Evaluate it and based on the shortcomings, cycle back through and conduct more analysis, opt for new design and implementation and repeat the cycle till the requirements are met completely. The iterative/fountain model can be viewed as a cascaded series of linear (waterfall) models. The incremental model is a superset of iterative model where the requirements are known at the beginning and they are divided into different groups. The core set of functions for each group is identified in the first cycle and is built and deployed as the first release. The second set of requirements along with the bug fixes and modification for first release is carried out in the second cycle and the process is repeated until all functionalities are implemented and they are meeting the requirements. It is obvious that in an iterative/incremental model (Fig. 15.8), each development cycle act as the maintenance phase for the previous cycle (release). Another approach for the iterative/interactive model is the 'Overlapped' model where development cycles overlap; meaning subsequent iterative/incremental cycle may begin before the completion of previous cycle.

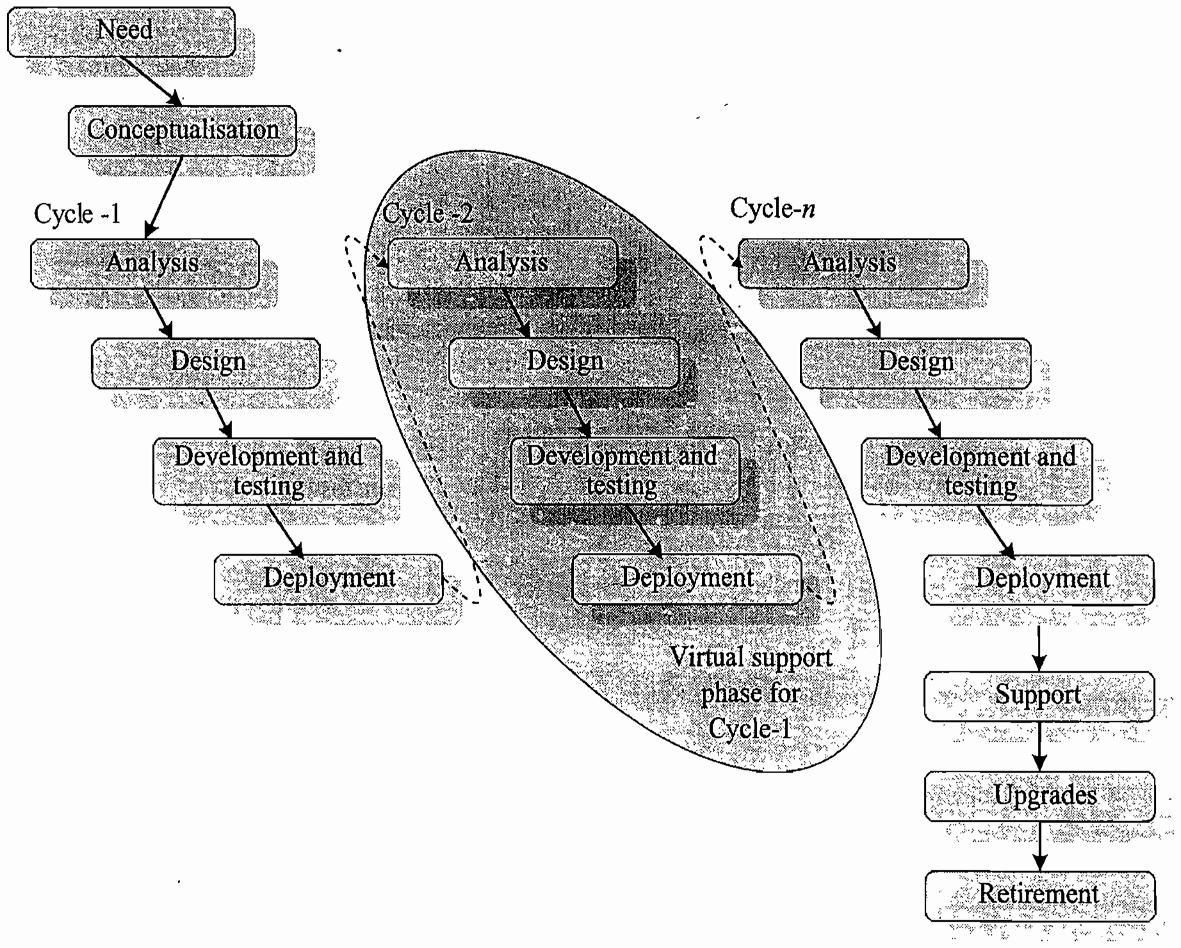


Fig. 15.8 **Iterative/Incremental/Fountain EDLC Model**

If you closely observe this model you can see that each cycle is interconnected in a similar fashion of a fountain, where water first moves up and then comes down, again moves up and comes down. The major advantage of iterative/fountain model is that it provides very good development cycle feedback at each function/feature implementation and hence the data can be used as a reference for similar product development in future. Since each cycle can act as a maintenance phase for previous cycle, changes in feature and functionalities can be easily incorporated during the development and hence more responsive to changing user needs. The iterative model provides a working product model with at least minimum features at the first cycle itself. Risk is spread across each individual cycle and can be minimised easily. Project management as well as testing is much simpler compared to the linear model. Another major advantage is that the product development can be stopped at any stage with a bare minimum working product. Though iterative model is a good solution for product development, it possess lots of drawbacks like extensive review requirement at each cycle, impact on operations due to new releases, training requirement for each new deployment at the end of each development cycle, structured and well documented interface definition across modules to accommodate changes. The iterative/incremental model is deployed in product developments where the risk is very high when the development is carried out by linear model. By choosing an iterative model, the risk is spread across multiple cycles. Since each cycle produces a working model, this model is best suited for product developments where the continued funding for each cycle is not assured.

15.5.3 Prototyping/Evolutionary Model

Prototyping/evolutionary model is similar to the iterative model and the product is developed in multiple cycles. The only difference is that this model produces a more refined prototype of the product at the end of each cycle instead of functionality/feature addition in each cycle as performed by the iterative model. There won't be any commercial deployment of the prototype of the product at each cycle's end. The shortcomings of the proto-model after each cycle are evaluated and it is fixed in the next cycle.

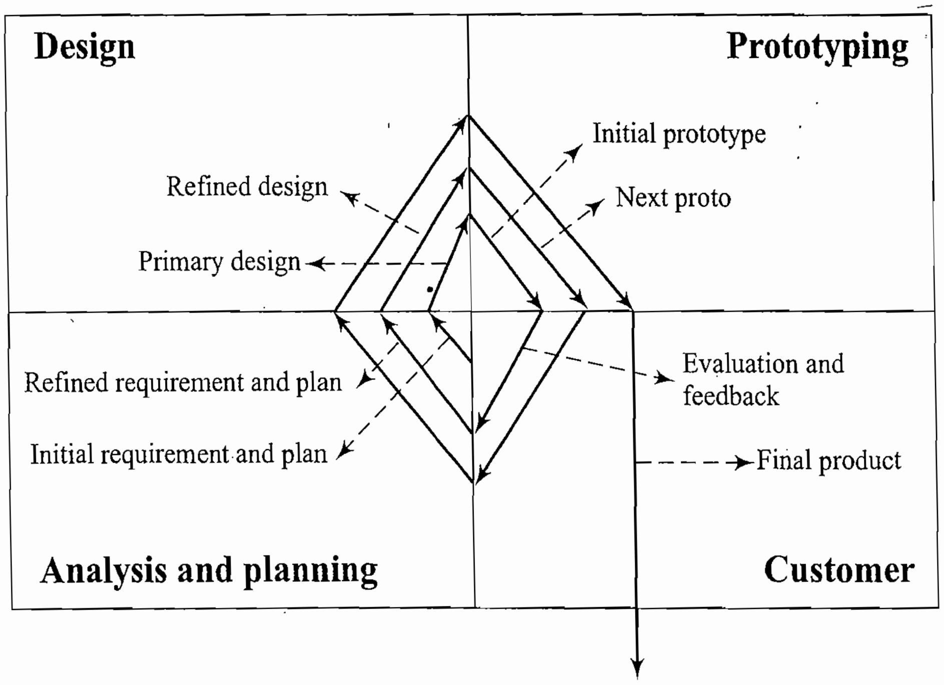


Fig. 15.9 **Proto-typing/Evolutionary EDLC Model**

After the initial requirement analysis, the design for the first prototype is made, the development process is started. On finishing the prototype, it is sent to the customer for evaluation. The customer evaluates the product for the set of requirements and gives his/her feedback to the developer in terms of shortcomings and improvements needed. The developer refines the product according to the customer's exact expectation and repeats the proto development process. After a finite number of iterations, the final product is delivered to the customer and launches in the market/operational environment. In this approach, the product undergoes significant evolution as a result of periodic shuttling of product information between the customer and developer. The prototyping model follows the approach – 'Requirements definition, proto-type development, proto-type evaluation and requirements refining'. Since the requirements undergo refinement after each proto model, it is easy to incorporate new requirements and technology changes at any stage and thereby the product development process can start with a bare minimum set of requirements. The evolutionary model relies heavily on user feedback after each implementation and hence finetuning of final requirements is possible. Another major advantage of proto-typing model is that the risk is spread across each proto development cycle and it is well under control. The major drawbacks of proto-typing model are

1. Deviations from expected cost and schedule due to requirements refinement
2. Increased project management
3. Minimal documentation on each prototype may create problems in backward prototype traceability
4. Increased Configuration Management activities

Prototyping model is the most popular product development model adopted in embedded product industry. This approach can be considered as the best approach for products, whose requirements are not fully available and are subject to change. This model is not recommended for projects involving the upgradation of an existing product. There can be slight variations in the base prototyping model depending on project management.

15.5.4 Spiral Model

Spiral model (Fig. 15.10) combines the elements of linear and prototyping models to give the best possible risk minimised EDLC Model. Spiral model is developed by Barry Boehm in 1988. The product development starts with project definition and traverse through all phases of EDLC through multiple phases. The activities involved in the Spiral model can be associated with the four quadrants of a spiral and are listed below.

1. Determine objectives, alternatives, constraints.
2. Evaluate alternatives. Identify and resolve risks.
3. Develop and test.
4. Plan.

Spiral model is best suited for the development of complex embedded products and situations where requirements are changing from customer side. Customer evaluation of prototype at each stage allows addition of requirements and technology changes. Risk evaluation in each stage helps in risk planning and mitigation. The proto model developed at each stage is evaluated by the customer against various parameters like strength, weakness, risk, etc. and the final product is built based on the final prototype on agreement with the client.

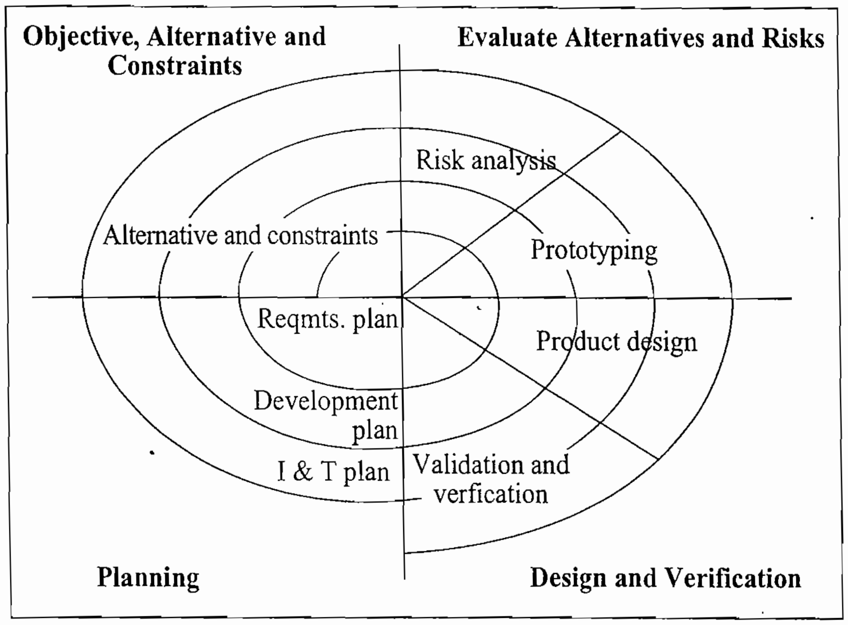

Objective, Alternative and Constraints
Evaluate Alternatives and Risks
Risk analysis
Alternative and constraints
Prototyping
Reqmts. plan
Product design
Development plan
I & T plan
Validation and verfication
Planning
Design and Verification

Fig. 15.10 **Spiral Model**

Summary

- ✓ Embedded Product Development Life Cycle (EDLC) is an 'Analysis-Design-Implementation' based standard problem solving approach for Embedded Product Development
- ✓ EDLC defines the interaction and activities among various groups of a product development sector including project management, system design and development (hardware, firmware and enclosure design and development), system testing, release management and quality assurance
- ✓ Project management, productivity improvement and ensuring quality of the product are the primary objectives of EDLC
- ✓ *Project Management* is essential for predictability, co-ordination and risk minimisation in product development
- ✓ The Life Cycle of a product development is commonly referred as Models and a *Model* defines the various phases involved in a product's life cycle
- ✓ The classic Embedded Product Life Cycle Model contains the phases: *Need, Conceptualisation, Analysis, Design, Development and Testing, Deployment, Support, Upgrades* and *Retirement/Disposal*
- ✓ The product development *need* falls into three categories namely, New or custom product development, Product Re-engineering and product maintenance
- ✓ *Conceptualisation phase* is the phase dealing with product concept development. Conceptualisation phase includes activities like product feasibility analysis, cost–benefit analysis, product scoping and planning for next phases
- ✓ The *Requirements analysis* phase defines the inputs, processes, outputs, and interfaces of the product at a functional level. Analysis and documentation, interface definition and documentation, high level test plan and procedure definition, etc. are the activities carried out during requirement analysis phase
- ✓ Product design phase deals with the implementation aspects of the required functionalities for the product.
- ✓ The *Preliminary design* establishes the top-level architecture for the product, lists out the various functional blocks required for the product, and defines the inputs and outputs for each functional block
- ✓ *Detailed Design* generates a detailed architecture, identifies and lists out the various components for each functional block, the inter-connection among various functional blocks, the control algorithm requirements, etc.

16 Trends in the Embedded Industry

EPILOGUE

The embedded industry has evolved a lot from the first mass produced embedded system, Autonetics D-17, to the recently launched Apple iPhone (Apple iPhone was the sensational gadget in the embedded device market at the time of writing this book (Year 2008-09)) in terms of miniaturisation, performance, features, development cost and time to market. Revolutions in processor technology is a prime driving factor in the embedded development arena. The embedded industry has witnessed the improvements over processor design from the 8bit processors of the 1970s to today's system on chip and multicore processors. The embedded operating system vendors, standards bodies, alliances, and open source communities are also playing a vital role in the growth of Embedded domain.

I hope this book was able to give you the fundamentals of embedded system, the steps involved in their design and development. I know this book itself is not complete in all aspects. It is quite difficult to present all the aspects of embedded design in a single book with limited page count. Moreover, the embedded industry is the one growing at a tremendous pace and whatever we feel superior in terms of performance and features may not be the same tomorrow. Wait for the rest of the books planned under this series to get an in-depth knowledge on the various aspects of embedded system design

So what next? Get your hands dirty with the design and development of Embedded Systems☺.

Wishing you good luck and best wishes in your new journey....

From the first mass produced embedded system, Autonetics D-17, to the recently launched Apple iPhone†, the embedded industry has evolved a lot, in terms of miniaturisation, performance, features, development cost and time to market. This book will not be complete without throwing some light into the current trends and bottlenecks in the embedded development.

16.1 PROCESSOR TRENDS IN EMBEDDED SYSTEM

Indeed, the advances in the processor technology are the prime driving factor in the embedded development arena. In the 1970s we started developing our embedded devices based on the 8bit microprocessors/controllers. Later we moved to the 16 and 32bit processor based designs, depending on our

†Apple iPhone was the sensational gadget in the embedded device market at the time of writing this book (Year 2008–09).

system requirement, as and when they appear in the market. Even today the 8bit processors/controllers are widely used in low-end applications. The only difference between the olden days 8bit processors/controllers and today's 8bit processors/controllers is that today's 8bit processors/controllers are more power and feature packed. In the olden days we used '*n*' number of ICs for building a device, but today, with the high degree of integration and IP Core re-use techniques, processors/controllers are integrating multiple IC functionalities into a single chip (System on Chip). In the olden days we required a processor/controller, Brown out circuit, reset circuit, watch dog timer ICs, and ADC/DAC ICs separately for building a simple Data Acquisition System, today a single microcontroller with all these components integrated is available in the market and it leads to the concept of miniaturisation and cost saving. Embedded industry has also witnessed the architectural changes for processors. In the beginning of the processor revolution, the speed at which an *8085* microprocessor executing a piece of code was awesome to us. Because it was beyond our imaginations. As we experienced the performance enhancements offered by general computing processors like x86, P-I, P-II, P-III, P-IV, Celeron, Centrino and Core 2 Duo, today we are unsatisfied with the performance offered by even the most powerful processor, because we have witnessed the rapid growth in the processor technology and our expectations are sky high today. Reduced Instruction Set Computing (RISC) and execution pipelining contributed a lot to the improvement on processor performance. From the procedural execution, the processor architecture moved to the single stage instruction pipelining and today processor families like ARM are supporting 8-stage pipelining (The latest ARM family ARM11 at the time of writing this book) with instruction and data cache.

The operating clock frequency was a bottleneck in processor designs. The earlier versions of processors/controllers were designed to operate at a few MHz. Advances in the semiconductor technology has elevated the bar on the operating frequency limitations. Nowadays processors/controllers with operating frequency in the range of Giga hertz (GHz) are available. Indeed, increase in operating frequency increases the total power consumption. Another trend in the embedded processor market is the 'Device oriented' design of processors. As mentioned earlier, in the beginning of the embedded revolution, we started building our embedded products around the available processors/controllers. As embedded industry started gaining momentum, the need for device specific processor design is flagged and processor manufactures started thinking in that direction. Today, when we think about developing a product, say Set Top Box or Handheld Device, we have a bunch of processors to select for the design. Intel is offering a variety of device specific processors. Intel PXA family of StrongARM processor (Now owned by Marvell Technology Group) is specifically designed for PDA applications and Intel has an x86 version for Set Top Box devices. The following section gives an overview of the key processor architecture/trends in embedded development.

16.1.1 System on Chip (SoC)

As the name indicates, a System on Chip (SoC) makes a system on a single chip. The System on Chip technology places multiple function 'systems' on a single chip. As mentioned earlier, in the beginning of the embedded revolution, we used separate ICs in an interconnected fashion for implementing a system. Innovations in the semiconductor area introduced the concept of integration of multiple function 'systems' on a single chip. With this it is possible to integrate almost all functional systems required to build an embedded product into a single chip. SoCs changed the concept of embedded processor from general purpose design to device specific design. Nowadays SoCs are available for diverse application needs like Set Top Boxes, Portable media players, PDAs, etc. iMX31 SoC from freescale semiconductor is a typical example for SoC targeted for multimedia applications. It integrates a powerful ARM 11 Core and other system functions like USB OTG interface, Multimedia and human interface (Graphics Accelerator, MPEG 4, Keypad Interface), Standard Interfaces (Timers, Watch Dog Timer, general purpose I/O), Image Processing Unit (Camera Interface, Display/TV control, Image Inversion and rotation, etc.) etc. on a single silicon wafer. On a first look SoC and Application Specific Integrated Circuit (ASIC) resembles the same. But they are not. SoCs are built by integrating the proven reusable IPs of different sub-systems, whereas ASICs are IPs developed for a specific application. By integrating multiple functions into a single chip, SoCs greatly save the board space in embedded hardware development and thereby leads to the concept of miniaturisation. SoCs are also cost effective.

16.1.2 Multicore Processors/Chiplevel Multi Processor (CMP)

One way of achieving increased performance is to increase the operating clock frequency. Indeed it will increase the speed of execution with the cost of high power consumption. In today's world most embedded devices are demanding battery power source for their operation. Here we don't have the luxury to offer high performance with the cost of reduced battery life. Here comes the role of Multicore processors. Multicore processors incorporate multiple processor cores on the same chip and works on the same clock frequency supplied to the chip. Based on the number of cores, the processors are known as dual core (2 cores), tri core (3 cores), quad core (4 cores), etc. Multicore processors implement multiprocessing (Simultaneous execution. Don't confuse it with multitasking). Each core of the CMP implements pipelining, multithreading and superscalar execution. Current implementations[†] of Intel multicore processors support cores up to 4, whereas Freescale multicore processors support cores up to 2. It is amazing to note that the multicore processor OCTEON™ CN3860, developed by Cavium Networks (http://www.caviumnetworks.com) is supporting 16 MIPS processor cores capable of operating at a clock frequency of 1GHz.

16.1.3 Reconfigurable Processors

Reconfigurable processor is a microprocessor/controller with reconfigurable hardware features. They contain an array of Programming Elements (PE) along with a microprocessor (Typically a RISC processor). The PE can be either a computational engine or a memory element. The hardware feature of the reconfigurable processor can be changed statically or dynamically. The dynamic changing of hardware configuration brings the advantage of making the chip adaptable to the firmware running on the processor. Depending on the situational need, the reconfigurable processor can entirely change their functionality to adapt to the new requirements. For example, a chip can configure itself to function as the heart of a camera system or a media player in a single device by downloading the appropriate firmware. Billions of Operations and Chameleon Systems are the key players of reconfigurable processor market. Reconfigurable SoCs (RSoCs) implement the IP for different subsystems through a reconfigurable matrix. This enables configurable hardware functionality for an SoC. The SoC needs to be configured to the required hardware functionality, through software support at the time of system initialisation (startup task). The hardware configuration can also be changed on the fly. A typical example for this is configuring the hardware codec for the required compression like MPEG1, MPEG 2, etc. by a media player application on a need basis. The Field Programmable System Level Integrated Circuit (FPSLIC)

[†]The statistics shown here is based on the data available till Dec 2008. Since processor technology is undergoing rapid changes, this data may not be relevant in future.

from Atmel Corporation is a typical example for RSoC. It integrates an 8bit AVR processor and a dynamically reconfigurable Field Programmable Gate Array (FPGA). The system is reconfigured from a library of pre-compiled IP cores stored in a FLASH memory on a need basis. RSoCS bring the concept of silicon sharing and thereby leads to less silicon usage and low power consumption.

16.2 EMBEDDED OS TRENDS

The Embedded OS industry is also undergoing revolutionary changes to take advantage of the potential offered by the trends in processor technologies. Today we have lot of options to select from a bunch of commercial and open source embedded OSs for building embedded devices. Most of the embedded OSs are trying to bring the virtualisation concept to the embedded device industry by adopting the microkernel architecture in place of the monolithic architecture. In microkernel architecture, the kernel contains very limited and essential features/services and rest of the features or services are installed as service and they run at the user space. Another noticeable trend adopting by OS suppliers is the 'Device oriented' design similar to the processor trends. In the olden days for building a device, we have to select an OS matching the device requirements and then customise it. Today OS suppliers are providing off-the-shelf OS customised for the device. Microsoft Embedded OS product line is a typical example of this. Microsoft has a range of OSs targeted for a group of device families like point of sale terminal, media player, set top box, etc. The Windows Mobile OS from Microsoft is specifically designed for mobile handsets. Open source embedded OS (Mostly centered around Embedded Linux) are gaining attention in the embedded market and the open source community is also offering off the shelf OS customised for specific group/family of devices. The Ubuntu MID edition which is specifically designed for Mobile Internet Device (MID) is a typical example for this. Since the processor technology is shifting the paradigm of computing from single core to multicore processors, the OS suppliers are also forced to change their strategies for supporting multicore processors. The latest version of the VxWorks (VxWorks 6.6SMP[†]) RTOS is designed to support the latest market-leading multicore processors.

16.3 DEVELOPMENT LANGUAGE TRENDS

When we talk about languages for embedded development, there are two aspects. One is the system side application (embedded device platform development) and the other one is the user application development. The system side application is responsible for managing the embedded device, interacting with low level hardware, scheduling and executing user applications, memory management, etc. whereas the user application runs on top of the system applications and requires the intervention of system application for performing system resource access (Like hardware access, memory access, etc.). In Embedded terminology the system side applications are referred as '*Embedded Firmware*' and the user applications are known as '*Embedded Software*'. The embedded firmware always comes as embedded into the program memory of the device and it is unalterable by the end user. Embedded software comes as either embedded with the firmware or user can install the software applications later on the device (A typical example is PDA, where the OS comes as embedded in the device and along with the OS certain application software like document viewer, mail client, etc. also comes as embedded in the device. Users can download and install rest of the applications in the device).

Whenever we think of development languages for embedded firmware, the first and foremost language that comes into our mind is 'C'. The reason being historic, the flexibility provided by 'C'

[†] At the time of writing this book (Year 2008–09).

language in hardware access and the bunch of cross-compilers and IDEs available for different platforms. We lived in an era where imagining a language beyond 'C' and Assembly (ALP) for embedded firmware was impossible. Now the scenario is totally changed. We have object-oriented languages like C++ for embedded firmware development. Efficient cross platform development tools and compilers from providers like Microsoft® played a significant role in this transformation. Advances in the compiler implementations and processor support brings java as one of the emerging language for system software development. Though java lacks lot of features essential for system software development, a move towards improving the missing factors is in progress. Still it is in the infancy stage. We will discuss the java based development under a separate thread. When it comes to embedded software development, a bunch of languages, including 'C', 'C++', Microsoft C#, ASP.NET, VB, Java, etc. are available. The following sections illustrate the role of Java and Microsoft .NET framework supported languages for embedded development.

16.3.1 Java for Embedded Development

Java being considered as the popular language for enterprise applications due to its platform independent implementations, but when it comes to embedded application development (firmware and software), it is not so popular due to its inherent shortcomings in real time system development support. In a basic java development, the java code is compiled by a java class compiler to platform independent code called *java bytecode*. The *java bytecode* is converted into processor specific object code by a utility called Java Virtual Machine (JVM). JVM abstracts the processor dependency from Java applications. JVM performs the operation of converting the bytecode into the processor specific machine code implementations. During run time, the bytecode is interpreted by the JVM for execution. The interpretation technique makes java applications slower than the other (cross) compiled applications. JVM can be implemented as either a piece of software code or hardware unit. Nowadays processors are providing built-in support for Java bytecode execution. The ARM processor implements hardware JVM called jazelle for supporting java. Figure 16.1 given below illustrates the implementation of Java applications in an embedded device with Operating System.

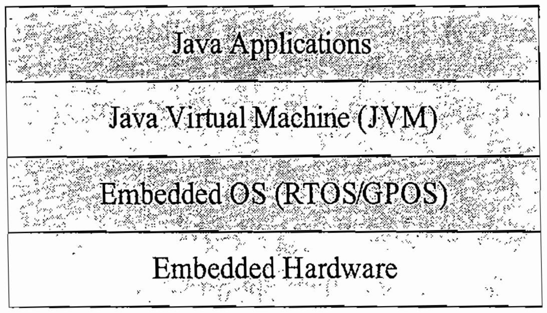


Fig. 16.1 **Java based embedded application development**

As mentioned earlier, JVM interprets the bytecode and thereby greatly affects the speed of execution. Another technique called Just In Time (JIT) compiler speeds up the Java program execution by caching all the previously interpreted bytecode. This avoids the delay in interpreting a bytecode which was already interpreted earlier.

The limitations of standard Java in embedded application development are:

1. The interpreted version of Java is quite slower and is not meeting the real-time requirement demanding by most embedded systems (For real-time applications)
2. The garbage collector of Java is non-deterministic in execution behaviour. It is not acceptable for a hard real-time system.
3. Processors which don't have a built-in JVM support or an off-the-shelf software JVM, require the JVM ported for the processor architecture.
4. The resource access (Hardware registers, memory, etc.) supported by Java is limited.
5. The runtime memory requirement for JVM and Java class libraries or JIT is a bit high and embedded systems which are constrained on memory cannot afford this.

Some movements to overcome these limitations are in the process and lot of novel ideas are emerging out to make Java suitable for embedded applications. The Ahead-of-Time (AOT) compilers for Java is intended for converting the Java bytecodes to target processor specific assembly code during compile time. It eliminates the need for a JVM/JIT for executing Java code. However there should be a piece of code for implementing the garbage collection in place of JVM/JIT. AOT compiler also brings the advantage of linking the compiled bytecode with other modules developed in languages like C/C++ and Assembly.

Java provides an interface named Java Native Interface (JNI), which can be used for invoking the functions written in other languages like C and Assembly. The JNI feature of Java can be explored for implementing system software like, user mode device drivers which has access to the JVM. Since JVM runs on top of the OS, device drivers which are statically linked with the OS kernel during OS image building or drivers which are loaded as shared object cannot be implemented using Java. In the Java based drivers, the low level hardware access is implemented in native languages like C.

The Java Community Process within the Safety Critical Java Technology expert group has come up with a real-time version for the JVM which is capable of providing comparable execution speed with C and also with predictable latency and reduced memory footprint.

Java Platform Micro Edition (Java ME), which was known as J2ME earlier, is a customised Java version specifically designed for 'Embedded application software' development for devices like mobile phones, PDAs, Set Top Boxes, printers, etc. Java ME applications are portable across devices. Java ME provides flexible User Interface (UI), built-in network protocols, security, etc. Sun Microsystems has also released an Embedded version of the Java Standard Edition (Java SE) with optimised memory footprint requirements for 'embedded application software' development. The embedded edition of the Java SE supports multicore processors.

16.3.2 .NET CF for Embedded Development

.NET is a framework for application development for desktop Windows Operating Systems (The UNIX version is also under development) from Microsoft®. It is a collection of pre-coded libraries and it acts as the run time component for .NET supported languages (C++, C#, VB, J#, etc.). It contains class libraries for User Interface, database connectivity, network connectivity, image editing, cryptography etc. The runtime environment of .NET is known as Common Language Runtime (CLR). The CLR resembles JVM in operation. It abstracts the target processor from application programmer. Like JVM, CLR also provides memory management (garbage collection), exception handling, etc. The CLR provides a language neutral platform for application development and execution. Applications written in .NET supported languages are compiled to a platform neutral intermediate language called Common Intermediate Language (CIL). For actual execution, the CIL is converted to the target processor specific machine code by the Common Language Runtime (CLR). Fig. 16.2 illustrates the concept of .NET framework based program execution.

The .NET framework is targeted for enterprise application development for desktop systems. The .NET framework consumes significant amount of memory. When it comes to embedded application development, the .NET framework is not a viable choice due to its memory requirement. For embedded and small devices running on Microsoft Embedded Operating Systems (like Windows Mobile and Windows CE), Microsoft® is providing a stripped down customised version of .NET framework called .NET Compact Framework or simply .NET CF for application development. Obviously .NET CF doesn't support all features of the .NET Framework. But it supports all necessary components required for application development for small devices. Users can develop applications in any .NET supported language including C++, C#, VB, J# etc.

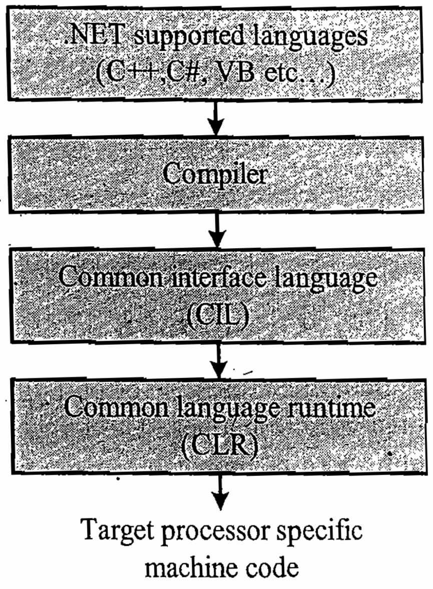

.NET supported languages (C++,C#, VB etc...)
Compiler
Common interface language (CIL)
Common language runtime (CLR)
Target processor specific machine code

Fig. 16.2 **.NET based Embedded Application Development**

16.4 OPEN STANDARDS, FRAMEWORKS AND ALLIANCES

Certain areas of the embedded industry are highly 'hot' and competitive. It demands strategic alliances to sustain in the market and bring innovations through combined R&D. A typical example is the handset industry. The combined R&D helps hardware manufacturers to come up with new designs and software developers to support the new hardware. With diverse market players it is essential to have formal specifications to ensure interoperability in operations. The Open Alliances and standards body are aimed towards defining and formulating the standards. Time to market is a critical factor in the sensational embedded market segments. Open sources and frameworks reduce the time to market in product development. The following sections highlight the popular strategic alliances, open source standards and frameworks in the mobile handset industry.

16.4.1 Open Mobile Alliance (OMA)

The Open Mobile Alliance (OMA) is a standards body for developing open standards for *mobile phone industry*. The OMA alliance was formed in 2002 by stakeholders from the world's leading network operators, device manufacturers, Information technology companies and content providers. The mission of OMA is 'To create interoperable services across countries, operators and mobile terminals by acting as the centre of the mobile service enabler specification work'. Please visit http://www.openmobilealliance.org/ for more details on OMA.

16.4.2 Open Handset Alliance (OHA)

The Open Handset Alliance (OHA) is a business alliance formed by various handset manufacturers (HTC, Samsung, LG, Motorola etc), chip manufacturers (Intel, Nvidia, Qualcomm, etc.), platform providers (Google, Wind River Systems, etc.) and carriers (T-Mobile, China Mobile, Sprint Nextel, etc.) for developing standards for *mobile devices*. Please visit http://www.openhandsetalliance.com/ for more details on OHA.

16.4.3 Android

Android is an open source software platform and operating system for mobile devices. It was developed by Google Corporation (www.google.com) and retained by the Open Handset Alliance (OHA). The Android operating system is based on the Linux kernel. Google has developed a set of Java libraries and developers can make use of these libraries for application development in java. For more details on android, please visit the website http://www.android.com/

16.4.4 Openmoko

Openmoko is a project for building open hardware and firmware (OS) standards for a family of open source mobile phones. The target operating system under consideration by openmoko is Embedded Linux and it is known as Openmoko Linux. It also supplies the hardware details (schematics and CAD drawings) of the phone as reference design for developers. Developers can customise the software stack and hardware to suit their product needs. For more details on Openmoko, please visit the website http://wiki.openmoko.org/wiki/Main_Page

16.5 BOTTLENECKS

So far we discussed about the positives of the embedded technology. Now let's have a look at the bottlenecks faced by the embedded industry today.

16.5.1 Memory Performance

Memory performance is a real road blocker in embedded development. Though processor technologies are supporting high speed operations through increased clock, the current memory technology is not yet up to the mark to catch up the speed offered by processors. It is high time to think about alternate memory techniques that can at least offer a performance somewhat near to that of processors.

16.5.2 Lack of Standards/Conformance to Standards

Though we talk about various alliances and open standards for specific areas of embedded system, there are no specific standards in place to ensure interoperability in all areas of embedded development. Even though, standards are in place for certain areas of the embedded development like mobile handset market, only a minor proportion of the players are sticking on to these standards and majority of the players are still sticking on to their own proprietary architecture and designs. To bring innovations and cutting edge feature rich products, it is essential to have a combined effort and standardisation. It is high time to think about strategic alliances in all areas of embedded development.

16.5.3 Lack of Skilled Resources

This is the most crucial problem faced by the embedded industry today. 'Design of Embedded System is an Art' and it requires highly skilled, self motivated and talented people to meet the time criticalities of embedded product development. Though most of our engineering graduates are highly talented, they lack proper orientation towards the design and development of embedded systems. Lack of good books on the embedded fundamentals is also a major reason for it. It is high time to think about teaching embedded system as a special branch of undergraduate course in all major universities.