INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

Complex systems and microprocessors—Embedded system design process—Design example: Model train controller- Instruction sets preliminaries - ARM Processor – CPU: programming input and output- supervisor mode, exceptions and traps – Co-processors- Memory system mechanisms – CPU performance- CPU power consumption.

INTRODUCTION:

SYSTEM:

• A system is a way of working, organizing or doing one or many tasks according to a fixed plan, program, or set of rules.
• A system is also an arrangement in which all its units assemble and work together according to the plan or program.
Example: watch
• A watch is a time—display system.
• Its parts are its hardware, needles and battery with the beautiful dial, chassis and strap.
• These parts organize to the real time every second and continuously update the time every second.
• The system-program updates the display using three needles after each second.
• It follows a set of rules.
• Some of these rules are as follows:
  ➢ All needles move only clockwise.
  ➢ A thin and long needle rotates every second such that it returns to same position after a minute.
  ➢ A long needle rotates every minute such that it returns to same position after an hour.
  ➢ A short needle rotates every hour such that it returns to same position after twelve hours.
  ➢ All three needles return to same inclination after twelve hours each day.

EMBEDDED SYSTEM:

• An embedded system is a system that has embedded software and computer-hardware, which makes it a system dedicated for an application(s) or specific part of an application or product or a part of a larger system.

APPLICATION AREAS OF EMBEDDED SYSTEMS:

• Embedded systems find applications in every industrial segment;

CATEGORIES OF EMBEDDED SYSTEMS:

Based on functionality and performance requirements, embedded systems can be categorized as:

1. Stand-alone embedded systems.
2. Real-time systems
3. Networked information appliances
4. Mobile device

1. Stand-alone embedded systems:
• It works in stand-alone mode. They take inputs, process them and produce the desire output.
1. **The input can be electrical signals from transducers or commands.**
2. **The output can be electrical signals to drive another system, an LED display or LCD display for displaying of information to the user.**
3. **Example: digital camera, microwave oven, etc.,**

**2. Real – time systems:**
- Real – time systems have to complete a specific task in a specified time period.
- Meeting the deadlines is the most important requirement of real–time systems.
- Real–time systems are categorized as,
  - Hard Real–time systems,
  - Soft Real – time systems

**Hard Real–time systems:** In this systems, missing a deadline may lead to a catastrophe (loss). [Missing deadline causes failure.]

**Soft Real–time systems:** In these systems, meeting the deadlines is important but missing the deadline will not lead to a catastrophe. [missing deadline results in degraded performance]

**3. Networked information appliances:**
- Embedded systems that are provided with network interfaces and accessed by networks, such as Local Area Network and the internet are called Networked information appliances.
- Such embedded systems are connected to a network, typically a network running TCP/IP Protocol suite (Transmission Control Protocol / Internet Protocol) and such the internet provider.

**4. Mobile device:**
- Mobile devices such as mobile phones, PDAs (Personal Digital Assistants), smart phones, etc. are the special category of embedded systems.
- The PDAs are now capable of supporting general purpose application software such as word processors, game, etc.

**CLASSIFICATION OF EMBEDDED SYSTEMS BASED ON APPLICATION:**
Embedded systems classify into three types as follows.

**1. Small Scale Embedded Systems:**
- These systems are designed with a single 8-bit or 16-bit microcontroller; they have little hardware and software complexities and involve board-level design.
- They may even be battery operated. When developing embedded software for these, an editor, assembler and cross assembler, specific to the microcontroller or processor used, are the main programming tools.
- Usually, ‘C’ is used for developing these systems.

**Example:**
Automatic Chocolate Vending Machine, Stepper motor controllers for a robotics system, Washing, cooking system, Multitasking Toys.

**2. Medium Scale Embedded Systems:**
- These systems are usually designed with a single or few 16- or 32-bit microcontrollers or DSPs or Reduced Instruction Set Computers (RISCs).
- These have both hardware and software complexities.
- For complex software design, there are the following programming tools: RTOS, Source code engineering tool, Simulator, Debugger and Integrated Development Environment (IDE).
Software tools also provide the solutions to the hardware complexities. An assembler is of little use as a programming tool.

These systems may also employ the readily available ASSPs (Application Specific System Processor) and IPs for the various functions—for example, for the bus interfacing, encrypting, deciphering, discrete cosine transformation and inverse transformation, TCP/IP protocol stacking and network connecting functions.

**Example:**
- Computer networking systems; for example, a router, a front-end processor in a server, a switch, a bridge, a hub and a gateway
- Entertainment systems – such as a video game and a music system.
- Banking systems for example, Bank ATM and Credit card transactions
- Signal Tracking Systems for example, an automatic signal tracker and a target tracker

**3. Sophisticated Embedded Systems:**
- Sophisticated embedded systems have enormous hardware and software complexities and may need scalable processors or configurable processors and programmable logic arrays.
- They are used for cutting edge applications that need hardware and software co-design and integration in the final system; however, they are constrained by the processing speeds available in their hardware units.
- Certain software functions such as encryption and decryption algorithms, discrete cosine transformation and inverse transformation algorithms, TCP/IP protocol stacking and network driver functions are implemented in the hardware to obtain additional speeds by saving time.
- Some of the functions of the hardware resources in the system are also implemented by the software.

**Example:**
- Embedded systems for wireless LAN and for convergent technology devices
- Embedded systems for real time video and speech or multimedia processing systems configuration in a system.
- Security products and High-speed Network security.

**COMPLEX SYSTEMS AND MICROPROCESSORS:**

**Embedded Computing System:**
- Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer.
- Thus, a PC is not itself an embedded computing system, although PCs are often used to build embedded computing systems.
- But a fax machine or a clock built from a microprocessor is an embedded computing system.
- This means that embedded computing system design is a useful skill for many types of product design.
- Automobiles, cell phones, and even household appliances make extensive use of microprocessors.

**Embedding Computers:**
- Computers have been embedded into applications since the earliest days of computing.
- One example is the Whirlwind, a computer designed at MIT in the late 1940s and early 1950s.
- Whirlwind was also the first computer designed to support **real-time operation** and was originally conceived as a mechanism for controlling an aircraft simulator.
- A microprocessor is a single-chip CPU.
EC703- EMBEDDED AND REAL TIME SYSTEMS  UNIT-I  INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

- Very large scale integration (VLSI) state the acronym is the name technology has allowed us to put a complete CPU on a single chip.
- The first microprocessor, the Intel 4004, was designed for an embedded application, namely, a calculator.
- The calculator was not a general-purpose computerit merely provided basic arithmetic functions.
- The HP-35 was the first handheld calculator to perform transcendental functions. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor.
- Integrated circuit design was an expensive and time-consuming process, the ability to reuse the hardware design by changing the software was a key breakthrough.
- Now-a-days microprocessors used in automobiles to control the engine, determining when spark plugs fire, controlling the fuel/air mixture, and so on.
- The combination of low fuel consumption and low emissions is very difficult to achieve; to meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.
- Microprocessors come in many different levels of sophistication; they are usually classified by their word size.
  - An **8-bit microcontroller** is designed for low-cost applications and includes on-board memory and I/O devices.
  - A **16-bit microcontroller** is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory.
  - A **32-bit RISC microprocessor** offers very high performance for computation-intensive applications.

**Uses of Microprocessors:**

- Household uses e.g. microwave oven
- Advanced thermostat systems
- Modern camera
- Digital television
- Automobile etc.

**EXAMPLE: BMW 850i brake and stability control system**

- The BMW 850i was introduced with a sophisticated system for controlling the wheels of the car.
- An antilock brake system (ABS) reduces skidding by pumping the brakes.
- An automatic stability control (ASC+T) system intervenes with the engine during maneuvering to improve the car’s stability.
- These systems actively control critical systems of the car; as control systems, they require inputs from and output to the automobile.
- Let’s first look at the ABS.
- The purpose of an ABS is to temporarily release the brake on a wheel when it rotates too slowly, when a wheel stops turning, the car starts skidding and becomes hard to control.
- It sits between the hydraulic pump, which provides power to the brakes, and the brakes themselves as shown in fig 1.1.
- This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking.
- The ABS system uses sensors on each wheel to measure the speed of the wheel.
- The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding.
The ASC+T system’s job is to control the engine power and the brake to improve the car’s stability during maneuvers.

The ASC+T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting.

The ASC+T can be turned off by the driver, which can be important when operating with tire snow chains.

Since the ABS was introduced several years earlier than the ASC+T, it was important to be able to interface ASC+T to the existing ABS module, as well as to the existing electronic modules.

The engine and control management units include the electronically controlled throttle, digital engine management, and electronic transmission control.

The ASC+T control unit has two microprocessors on two printed circuit boards, one of which concentrates on logic-relevant components and the other on performance-specific components.

**CHARACTERISTICS OF EMBEDDED COMPUTING APPLICATIONS:**

- Functionality is important in both general-purpose computing and embedded computing, but embedded applications must meet many other constraints as well.
- On the one hand, embedded computing systems have to provide sophisticated functionality:

**Complex algorithms:**

- The operations performed by the microprocessor may be very sophisticated.
- For example, the microprocessor that controls an automobile engine must perform complicated filtering functions to optimize the performance of the car while minimizing pollution and fuel utilization.

**User interface:**

- Microprocessors are frequently used to control complex user interfaces that may include multiple menus and many options.
- The moving maps in Global Positioning System (GPS) navigation are good examples of sophisticated user interfaces.

To make things more difficult, embedded computing operations must often be performed to meet deadlines:

**Real time:**

- Many embedded computing systems have to perform in real time, if the data is not ready by a certain deadline, the system breaks.
- In some cases, failure to meet a deadline is unsafe and can even endanger lives.
In other cases, missing a deadline does not create safety problems but does create unhappy customers missed deadlines in printers, for example, can result in scrambled pages.

**Multirate:**
- Not only must operations be completed by deadlines, but many embedded computing systems have several real-time activities going on at the same time.
- They may simultaneously control some operations that run at slow rates and others that run at high rates. Multimedia applications are prime examples of multirate behavior.
- The audio and video portions of a multimedia stream run at very different rates, but they must remain closely synchronized.
- Failure to meet a deadline on either the audio or video portions spoils the perception of the entire presentation.

**Manufacturing cost:**
- The total cost of building the system is very important in many cases.
- Manufacturing cost is determined by many factors, including the type of microprocessor used, the amount of memory required and the types of I/O devices.

**Power and energy:**
- Power consumption directly affects the cost of the hardware, since a larger power supply may be necessary.
- Energy consumption affects battery life, which is important in many applications, as well as heat consumption, which can be important even in desktop applications.

**Reason to select microprocessors or system design:**
  - Microprocessors are a very efficient way to implement digital systems.
  - Microprocessors make it easier to design families of products that can be built to provide various feature sets at different price points.
  - It extended to provide new feature keep up with rapidly changing markets.
  - It executes programs very efficiently.
  - Microprocessor make their CPUs run very fast.
  - Microprocessors are very efficient utilizers of logic.
  - Microprocessor can be used for many different algorithms simply by changing the program it executes.
  - Implementing several functions on a single processor makes much better use of the available hardware budget.

**Why not use PCs for all embedded computing?**
- Components of PCs are, used in many embedded computing systems.
- But several factors keep us from using the stock PC as the universal embedded computing platform.
  - Real-time performance is very less in PC due to different architectures in PC.
  - It increase the complexity and price of the components due to broad mix of computing requirements.
- The above two problems will be overcome by the multiprocessors.
  - By multiprocessors we can achieve best real-time performance.
  - Low power and low cost also drive toward multiprocessors.

**The Physics of Software:**
- Software performance and energy consumption are very important properties when we are connecting our embedded computers to the real world.
To understand the sources of performance and power consumption is important to meet our application’s goals.
To make very high-level decisions about the structure of our programs to greatly improve their real-time performance and power consumption.

CHALLENGES IN EMBEDDED COMPUTING SYSTEM DESIGN:
Some important problems that must be taken into account in embedded system design as follows.

Hardware:
- To control the amount of computing power we cannot only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more.
- To meet performance deadlines and manufacturing cost constraints, the choice of hardware is important.

Deadlines:
- The way of meeting a deadline is to speed up the hardware so that the program runs faster.
- It makes the system more expensive.
- It is also entirely possible that increasing the CPU clock rate may not make enough difference to execution time, since the program’s speed may be limited by the memory system.

Power consumption:
- In battery-powered non-battery applications power consumption is extremely important.
- Even in non-battery applications, excessive power consumption can increase heat dissipation.
- Digital system consume less power is to make it run more slowly, but it lead to missed deadlines.
- Careful design is required to slow down the non-critical parts of the machine for power consumption while still meeting necessary performance goals.

Design for upgradability:
- The hardware platform may be used over several product generations, or for several different versions of a product in the same generation, with few or no changes.
- We want to be able to add features by changing software.
- How can we design a machine that will provide the required performance for software that we haven’t yet written.
- Reliability is always important when selling products customers rightly expect that products they buy will work.
- Reliability is especially important in some applications, such as safety-critical systems.
Let’s consider some ways in which the nature of embedded computing machines makes their design more difficult.

Complex testing:
- Exercising an embedded system is generally more difficult than typing in some data.
- We may have to run a real machine in order to generate the proper data.
- The timing of data is often important, meaning that we cannot separate the testing of an embedded computer from the machine in which it is embedded.

Limited observability and controllability:
- Embedded computing systems usually do not come with keyboards and screens.
- So it is more difficult to see what is going on and to affect the system’s operation.
EC6703- EMBEDDED AND REAL TIME SYSTEMS    UNIT-I    INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

Restricted development environments:
- The development environments for embedded systems are often much more limited than those available for PCs and workstations.
- We generally compile code on one type of machine, such as a PC and download it onto the embedded system.
- To debug the code, we must usually rely on programs that run on the PC or workstation and then look inside the embedded system.

Performance in Embedded Computing:
- At the heart of embedded computing is real-time computing, which is the science and art of programming to deadlines.
- The program receives its input data; the deadline is the time at which a computation must be finished.
- If the program does not produce the required output by the deadline, then the program does not work, even if the output that it eventually produces is functionally correct.
- In order to understand the real-time behavior of an embedded computing system, we have to analyze the system at several different levels of abstraction.
  - **CPU:** The CPU clearly influences the behavior of the program, particularly when the CPU is a pipelined processor with a cache.
  - **Platform:** The platform includes the bus and I/O devices. The platform components that surround the CPU are responsible for feeding the CPU and can dramatically affect its performance.
  - **Program:** Programs are very large and the CPU sees only a small window of the program at a time. We must consider the structure of the entire program to determine its overall behavior.
  - **Task:** We generally run several programs simultaneously on a CPU, creating a multitasking system. The tasks interact with each other in ways that have profound implications for performance.
  - **Multiprocessor:** Many embedded systems have more than one processor; they may include multiple programmable CPUs as well as accelerators. Once again, the interaction between these processors adds yet more complexity to the analysis of overall system performance.

The Embedded System Design Process:

![Diagram of the Embedded System Design Process]

Fig1.2 Major levels of abstraction in the design process
Embedded system design process has two objectives.

- Introduction to the various steps in embedded system design before developed in detailed design
- Design methodology

A design methodology is important for following reasons.

- Design to ensure that we have done everything we need to do.
- Develop computer-aided design tools.
- Makes it much easier for members of a design team to communicate.

There are five major levels of embedded system design process. Before going to design a system the designers need to fulfil the needs to design such a system. They are

**REQUIREMENTS:**

- **First**, the designer must gather an informal description from the customers it is known as requirements.
- **Second**, then the designer must know what we are designing and then to refine the requirement into specification and to begin to designing the system architecture.
- Requirements may be functional or non-functional, we must of course capture the basic functions of the embedded system, but functional description is often not sufficient.
- Typical non-functional requirements include:

**Performance:**

- The speed of the system is often a major consideration both for the usability of the system and for its ultimate cost.
- Performance may be a combination of soft performance metrics such as approximate time to perform a user-level function and hard deadlines by which a particular operation must be completed.

**Cost:**

- The target cost or purchase price for the system is almost always a consideration. Cost typically has two major components:
  - Manufacturing cost includes the cost of components and assembly;
  - Non recurring engineering (NRE) costs include the personnel and other costs of designing the system.

**Physical size and weight:**

- The physical aspects of the final system can vary greatly depending upon the application.
- An industrial control system for an assembly line may be designed to fit into a standard-size rack with no strict limitations on weight.
- A handheld device typically has tight requirements on both size and weight that can ripple through the entire system design.

**Power consumption:**

- Power, of course, is important in battery-powered systems and is often important in other applications as well.
- Power can be specified in the requirements stage in terms of battery life.

Now we will list some requirements form collected from the user at the start of the project.

1. Name
2. Purpose
3. Inputs
4. Outputs
5. Functions
EC6703- EMBEDDED AND REAL TIME SYSTEMS

6. Performance
7. Manufacturing cost
8. Power
9. Physical size and weight

SPECIFICATION:
- The specification is more precise -it serves as the contract between the customer and the architects. It must be carefully written so that it accurately reflects the customer’s requirements.
- Specification is essential to creating working systems with a minimum of designer effort.
- The specification will guide the designers what to build when it build.
- The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer.

ARCHITECTURE DESIGN:
- The specification does not say how the system does things, only what the system does.
- The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture.
- The creation of the architecture is the first phase of what many designers think of as design.
- The architecture in the form of a block diagram that shows major operations and data flows among them.
- This block diagram is still quite abstract -we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on.
- Architectural descriptions must be designed to satisfy both functional and non-functional requirements.
- Not only must all the required functions be present, but we must meet cost, speed, power, and other non-functional constraints.
- Starting out with a system architecture and refining that to hardware and software architectures is one good way to ensure that we meet all specifications.
- We can concentrate on the functional elements in the system block diagram, and then consider the non-functional constraints when creating the hardware and software architectures.

DESIGNING HARDWARE AND SOFTWARE COMPONENTS:
- The component design effort builds those components in conformance to the architecture and specification.
- The components will in general include both hardware like FPGAs, boards, and so on and software modules.
- Some of the components will be ready-made.
- The CPU, for example, will be a standard component in almost all cases, as well as memory chips and many other components.

SYSTEM INTEGRATION:
- Only after the components are built do we have the satisfaction of putting them together and seeing a working system.
- This phase usually consists of a lot more than just plugging everything together and standing back.
- Bugs are typically found during system integration, and good planning can help us find the bugs quickly.
- By building up the system in phases and running properly chosen tests, we can often find bugs more easily.
- System integration is difficult because it usually uncovers problems.
It is often **hard to observe the system in sufficient detail to determine exactly what is wrong.**

The **debugging facilities** for embedded systems are usually much more **limited** than what you would find on desktop systems.

Now consider one example as GPS moving map and discuss about embedded system design process.

**Requirements analysis of a GPS moving map:**

- The moving map is a handheld device that displays for the user a map of the terrain around the user’s current position; the map display changes as the user and the map device change position.
- The moving map obtains its position from the GPS, a satellite-based navigation system.
- The moving map display might look something like the following functions.
  - **Functionality:** This system is designed for highway driving and similar uses, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.
  - **User interface:** The screen should have at least 400 x 600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed to allow the user to make selections to control the system.
  - **Performance:** The map should scroll smoothly. Upon power-up, a display should take no more than one second to appear, and the system should be able to verify its position and display the current map within 15 s.
  - **Cost:** The selling cost (street price) of the unit should be no more than $100.
  - **Physical size and weight:** The device should fit comfortably in the palm of the hand.
  - **Power consumption:** The device should run for at least eight hours on four AA batteries.

Based on this discussion, let’s write a requirements chart for our moving map system:

<table>
<thead>
<tr>
<th>Name</th>
<th>GPS moving map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Consumer-grade moving map for driving use</td>
</tr>
<tr>
<td>Inputs</td>
<td>Power button, two control buttons</td>
</tr>
<tr>
<td>Outputs</td>
<td>Back-lit LCD display 400 x 600</td>
</tr>
<tr>
<td>Functions</td>
<td>Uses 5-receiver GPS system; three user-selectable resolutions; always displays current latitude and longitude</td>
</tr>
<tr>
<td>Performance</td>
<td>Updates screen within 0.25 seconds upon movement</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>$30</td>
</tr>
<tr>
<td>Power</td>
<td>100mW</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>No more than 2” x 6,” 12 ounces</td>
</tr>
</tbody>
</table>

**Fig 1.3 Block diagram for the moving map**

A specification of the GPS system would include several components:

- Data received from the GPS satellite constellation.
- Map data.
- User interface.
- Operations that must be performed to satisfy customer requests.
- Background actions required to keep the system running, such as operating the GPS receiver.

**Fig 1.3.1 Hardware architecture for the moving map**

**Fig 1.3.2 Software architecture for the moving map**

- The GPS is a good example of a specialized component in almost that will nonetheless be predesigned, standard component.
- We can also make use of standard software modules one good example is that topographic database.

**FORMALISMS FOR SYSTEM DESIGN:**

- There is a visual language that can be used to capture all these design tasks known as Unified Modeling Language (UML).
- UML was designed to be useful at many levels of abstraction in the design process.
- UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction.
- UML is an object-oriented modeling language.
- Object-oriented design emphasizes two concepts of importance:
  - It encourages the design to be described as a number of interacting objects, rather than a few large monolithic blocks of code.
  - Objects will correspond to real pieces of software or hardware in the system.

Object-oriented (often abbreviated OO) specification can be seen in two complementary ways:

- Object-oriented specification allows a system to be described in a way that closely models real-world objects and their interactions.
- Object-oriented specification provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems components to real-world objects.

- The relationship between an object-oriented specification and an object-oriented programming language is a specification language may not be executable.
- But both languages provide similar basic methods for structuring large systems.
- UML is a large and rich, there are many graphical elements in a UML diagram.
It is important to be careful to use the correct drawing to describe something for instance; UML distinguishes between arrows with open and filled-in arrow heads, and solid and broken lines.

**Structural Description:**

- Structural description, gives basic components of the system and designer can learn how to describe how these components.
- The principal component of an object-oriented design is, the object.
- An object includes a set of attributes that define its internal state.
- An object describing a display is shown in UML notation in Figure 1.4.

![Fig 1.4 An object in UML notation](image)

- The object is identified in two ways: It has a unique name, and it is a member of a class. The name is underlined to show that this is a description of an object and not of a class.
- A class is a form of type definition all objects derived from the same class have the same characteristics, although their attributes may have different values.
- A class defines the attributes that an object may have.
- It also defines the operations that determine how the object interacts with the rest of the world.
- The UML description of the Display class is shown in Figure 1.4.1.
- A class defines both the interface for a particular type of object and that object’s implementation.
- When we use an object, we do not directly manipulate its attributes, we can only read or modify the object’s state through the operations that define the interface to the object.
- The choice of an interface is a very important decision in object-oriented design.
- The proper interface must provide ways to access the object’s state as well as ways to update the state.

![Fig 1.4.1 An class in UML notation](image)

- There are several types of relationships that can exist between objects and classes:
  - **Association** occurs between objects that communicate with each other but have no ownership relationship between them.
  - **Aggregation** describes a complex object made of smaller objects.
  - **Composition** is a type of aggregation in which the owner does not allow access to the component objects.
  - **Generalization** allows us to define one class in terms of another.
The elements of a UML class or object do not necessarily directly correspond to statements in a programming language, if the UML is intended to describe something more abstract than a program, there may be a significant gap between the contents of the UML and a program implementing it.

- An attribute is some value that reflects the current state of the object.
- The behaviors of the object must be in a higher-level specification, and contains basic things that can be done with an object.
- Like most object-oriented languages, UML also allows us to define one class in terms of another.
- A derived class inherits all the attributes and operations from its base class.
- A derived class is defined to include all the attributes of its base class.

![Diagram of Derived Classes as a Form of Generalization in UML](image)

**Fig 1.4.2 Derived classes as a form of generalization in UML.**

- From our example display is the base class and BW_display and Color_map_display are the two derived classes.
- Here BW_display represents black and white display, now we can draw the diagram for base and derived classes.

**Inheritance:**

![Diagram of Multiple Inheritance in UML](image)

**Fig 1.4.3 Multiple inheritance in UML.**
Unified Modeling Language considers inheritance to be one form of generalization.

A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead as shown in fig 1.4.3.

Both BW_display and Color_map_display are specific versions of Display UML also allows us to define multiple inheritance.

Multiple inheritance means which a class is derived from more than one base class.

A link describes a relationship between objects; association is to link as class is to object.

Link used to make objects to stand alone and association capture type information about these links.

![Diagram of messages and message sets](image)

**Links between objects**

**Association between classes**

Fig 1.4.4 Association between classes

**Behavioral Description:**

- We have to specify the behavior of the system as well as its structure. One way to specify the behavior of an operation is a state machine.

![Diagram of state transition](image)

**Fig 1.4.5 A state and transition in UML.**

- These state machines will not rely on the operation of a clock, as in hardware; rather, changes from one state to another are triggered by the occurrence of events.
- An event is some type of action. We have three types of events defined by UML, as illustrated in Figure 1.4.6.
  - A signal is an asynchronous occurrence. It is defined in UML by an object that is labeled as a `<<signal>>`. The object in the diagram serves as a declaration of the event’s existence. A signal may have parameters that are passed to the signal’s receiver.
  - A call event follows the model of a procedure call in a programming language.
  - A time-out event causes the machine to leave a state after a certain amount of time.
Fig 1.4.6 Signal, call, and time-out events in UML

Sequence diagram:
- It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved.
- A sequence diagram is somewhat similar to a hardware timing diagram, although the time flows vertically in a sequence diagram and horizontally in a timing diagram.
- The sequence diagram is designed to show a particular choice of events and it is not convenient for showing a number of mutually exclusive possibilities.

Fig 1.4.7 A sequence diagram in UML.
**Fig 1.5 A model train control system**

- In order to learn how to use UML to model systems, we will specify a simple system, a model train controller in Fig 1.5.
- The user sends messages to the train with a control box attached to the tracks.
- The control box may have familiar controls such as a throttle, emergency stop button, and so on.
- The train receives its electrical power from the two rails of the track, the control box can send signals to the train over the tracks by modulating the power supply voltage.
- As shown in the fig 1.5, the control panels end packets over the tracks to the receiver on the train.
- The train includes analog electronics to sense the bits being transmitted and a control system to set the train motor’s speed and direction based on those commands.
- Each packet includes an address so that the console can control several trains on the same track; the packet also includes an error correction code (ECC) to guard against transmission errors.
- This is a one-way communication system the model train cannot send commands back to the user.

**Requirements:**

The basic set of requirements for the system:

- The console shall be able to control up to eight trains on a single track.
- The speed of each train shall be controllable by a throttle to at least 63 different levels in each direction (forward and reverse).
- There shall be an emergency stop button.
- An error detection scheme will be used to transmit messages.
EC6703- EMBEDDED AND REAL TIME SYSTEMS UNIT-I

INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

➢ There shall be an inertia control that shall allow the user to adjust the responsiveness of the train to
commanded changes in speed.
➢ We can put the requirements into our chart format:

<table>
<thead>
<tr>
<th>Name</th>
<th>Model train controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Control speed of up to eight model trains</td>
</tr>
<tr>
<td>Inputs</td>
<td>Throttle, inertia setting, emergency stop, train number</td>
</tr>
<tr>
<td>Outputs</td>
<td>Train control signals</td>
</tr>
<tr>
<td>Functions</td>
<td>Set engine speed based upon inertia settings; respond to emergency stop</td>
</tr>
<tr>
<td>Performance</td>
<td>Can update train speed at least 10 times per second</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>$50</td>
</tr>
<tr>
<td>Power</td>
<td>10W (plugs into wall)</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>Console should be comfortable for two hands, approximate size of standard keyboard; weight &lt;2 pounds</td>
</tr>
</tbody>
</table>

DIGITAL COMMAND CONTROL (DCC):

➢ The DCC standard was created by the National Model Railroad Association to support interoperable
digitally-controlled model trains.
➢ DCC was created to provide a standard that could be built by any manufacturer so that hobbyists could mix
and match components from multiple vendors.
➢ The DCC standard is given in two documents:
  ➢ Standard S-9.1, the DCC Electrical Standard, defines how bits are encoded on the rails for transmission.
  ➢ Standard S-9.2, the DCC Communication Standard, defines the packets that carry information.
➢ Any DCC-conforming device must meet these specifications.
➢ The DCC standard does not specify many aspects of a DCC train system.
➢ It doesn’t define the control panel, the type of microprocessor used, the programming language to be used, or
many other aspects of a real model train system.
➢ The encoding standard must be carefully designed because the main function of the track is to carry power
to the locomotives.
➢ The signal encoding system should not interfere with power transmission either to DCC or non-DCC
locomotives.
➢ The data signal should not change the DC value of the rails.
➢ The data signal swings between two voltages around the power supply voltage such as 0 and 1 as shown in
Fig 1.5.1.
➢ The standard also describes other electrical properties of the system, such as allowable transition times for
signals.

![Fig 1.5.1 Bit encoding in DCC.](image)

➢ The DCC Communication Standard describes how bits are combined into packets and the meaning of some
important packets.
We can write the basic packet format as a regular expression:

\[ PSA(SD) + E \quad \ldots \ldots (1) \]

Where

- P is the preamble, which is a sequence of at least 101 bits.
- S is the packet start bit. It is a 0 bit.
- A is an address data byte that gives the address of the unit, with the most significant bit of the address transmitted first. An address is eight bits long. The addresses 00000000, 11111110, and 11111111 are reserved.
- S is the data byte start bit, which like the packet start bit is a 0.
- D is the data byte, which includes eight bits. A data byte may contain an address, instruction, data, or error correction information.
- E is a packet end bit, which is a 1 bit.

A packet includes one or more data byte start bit/data byte combinations.

**Baseline packet:**

- A baseline packet is the minimum packet that must be accepted by all DCC implementations.
- More complex packets are given in a Recommended Practice document.
- A baseline packet has three data bytes:
  - Address data byte that gives the intended receiver of the packet;
  - Instruction data byte provides a basic instruction;
  - Error correction data byte is used to detect and correct transmission errors.
- The instruction data byte carries several pieces of information such as
  - Bits 0-3 provide a 4-bit speed value.
  - Bit 4 has an additional speed bit.
  - Bit 5 gives direction, with 1 for forward and 0 for reverse.
  - Bits 7-8 are set at 01 to indicate that this instruction provides speed and direction.

**Conceptual Specification**

- Digital Command Control specifies some important aspects of the system, but it does not specify everything about a model train control system.
- Solution for the above problem is conceptual specification. It allows us to understand the system better.
- A train control system turns commands into packets.
- A command comes from the command unit while a packet is transmitted over the rails.
- Commands and packets may not be generated in a 1-to-1 ratio.
- In the DCC standard command units should resend packets in case a packet is dropped during transmission.
  1. Command unit and
  2. Train-board component as shown in Figure 1.5.2.
Fig 1.5.2 Class diagram for the train controller messages.

Fig 1.5.3 UML collaboration diagram for major subsystems of the train controller system.

Now we will discuss about class diagram of train subsystems and functions of each element presented in the diagram.

- The console needs to perform three functions:
  - Read the state of the front panel on the command unit,
  - Format messages, and
  - Transmit messages.

- The train receiver must also perform three major functions:
  - Receive the message,
  - Interpret the message
  - Control the motor

**Basic characteristics of classes in UML diagram:**

- The **Console** class describes the command unit’s front panel, which contains the analog knobs and hardware to interface to the digital parts of the system.
- The **Formatter** class includes behaviors that know how to read the panel knobs and creates a bit stream for the required message.
- The **Transmitter** class interfaces to analog electronics to send the message along the track.

In the class diagram some special classes that represent analog components and it is ending the name of each with an asterisk:

- **Knobs** describes the actual analog knobs, buttons, and levers on the control panel.
- **Sender** describes the analog electronics that send bits along the track.

Likewise, the train makes use of three other classes that define its components:

1. The **Receiver** class knows how to turn the analog signals on the track into digital form.
2. The **Controller** class includes behaviors that interpret the commands and figures out how to control the motor.
3. The **Motor interface** class defines how to generate the analog signals required to control the motor.

**Classes to represent analog components:**

- **Detector** detects analog signals on the track and converts them into digital form.
- **Pulser** turns digital commands into the analog signals required to control the motor speed.
We have also defined a special class, Train set, to help us remember that the system can handle multiple trains.

Fig 1.5.4 A UML class diagram for the train controller showing the composition of the subsystems.

Detailed Specification:
- We need to define the analog components in a little more detail because their characteristics will strongly influence the Formatter and Controller.
- Fig 1.5.5 Classes describing analog physical objects in the train control system. Now we will concentrate on each analog components and its functions in detailed manner with its class diagram.
- The Panel has three Knobs:
  1. Train number
  2. Speed and
  3. Inertia.
- It also has one button for emergency-stop.

<table>
<thead>
<tr>
<th>Knobs*</th>
<th>Pulser*</th>
</tr>
</thead>
<tbody>
<tr>
<td>train-knob: integer</td>
<td>pulse-width: unsigned-integer</td>
</tr>
<tr>
<td>speed-knob: integer</td>
<td>direction: boolean</td>
</tr>
<tr>
<td>inertia-knob: unsigned-integer</td>
<td>emergency-stop: boolean</td>
</tr>
<tr>
<td>emergency-stop: boolean</td>
<td></td>
</tr>
<tr>
<td>set-knobs()</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sender*</th>
<th>Detector*</th>
</tr>
</thead>
<tbody>
<tr>
<td>send-bit()</td>
<td>&lt;integer&gt; read-bit(): Integer</td>
</tr>
</tbody>
</table>
These classes specify the detailed functions of each and every component. The pulse will control the train speed using pulse width modulation.

- In the train control system we have two more components such as formatter and controller. Now we will see the class diagram of its gives the detailed operation performed by the particular component.
- Operate behavior any class also important behavior of classes is defined in state diagram for example we can see the state diagram for formatter and controller class.
- Class diagrams and state diagrams will give more detailed information and what kind of operations each class going to perform everything will be presented.

How one class is going to communicate or interface with other class to perform different kind of operations.
Fig 1.5.8 State diagram for the formatter operate behavior

Fig 1.5.9 State diagram for the Controller operate behavior.
- Sequence diagram will specifies the interface between more than one classes and its detailed operations and what ways its going to operate.
- For example we can take sequence diagram or transmitting a control input.

Fig 1.5.10 Sequences diagram for transmitting a control input.

**ARM PROCESSOR:**
- ARM is actually a family of RISC architectures that have been developed over many years.
The ARM is a 32 bit Reduced Instruction Set Computer (RISC) Instruction Set architecture developed by ARM holdings.

- It was known as the Advanced RISC machine and before that as the Ascorn RISC Machine.
- ARM processors made them suitable for lowpower applications.
- This has made them dominant in the mobile and embedded electronics market as relatively low cost and small microprocessors and microcontrollers.
- ARM is a 32 bit Processors.
- ARM architecture comes in several versions, we will concentrate on ARM version 7.

**Preliminaries:**
- It deals with the different styles of computer architecture and the nature of assembly language.

**Computer Architecture Taxonomy:**
1. Von Neumann architecture computer
2. Harvard architecture
3. Complex Instruction Set Computers (CSIC)
4. Reduced Instruction Set Computers (RSIC)

**Von Neumann architecture:**
- This kind of architecture consists of a central Processing Unit (CPU) and a memory.
- The memory holds both data and instructions, and can be read or written when given an address.
- The CPU has several internal registers that store values used internally.
- One of those registers is the program counter (PC), which holds the address in memory of an instruction.
- The program counter does not directly determine what the machine does next, but only indirectly by pointing to an instruction in memory.

**Harvard architecture:**
- Harvard machine has separate memories for data and program.
EC6703- EMBEDDED AND REAL TIME SYSTEMS  

UNIT-I  INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

• The program counter points to program memory, not data memory.
• It leads to make harder to write self-modifying programs on Harvard machines.

Advantages:
• Separation of program and data memories provides higher performance for digital signal processing.
• Data are processed at precise intervals.
• It provides higher bandwidth.
• It has streaming data it means data sets arrive continuously and periodically.

Complex Instruction Set Computers (CISC):
• These machines provided a variety of instructions that may perform very complex tasks, and it generally used a number of different instruction formats of varying lengths.

Reduced Instruction Set Computers (RISC):
• RISC is developed to high performance microprocessors and its instruction sets are executed in pipelined Processors.
• Different CPUs may have different clock speeds, cache configurations and many other changes that can make one model of CPU more attractive than another for any given application.

Assembly Language:
• Assembly languages usually share the same basic features of other languages. ARM processor has the assembly code and it includes the following functions.
  ➢ One instruction appears per line.
  ➢ Labels, which give names to memory locations, start in the first column.
  ➢ Instructions must start in the second column or after to distinguish them from labels.
  ➢ Comments run from some designated comment character (; in the case of ARM) to the end of the line.
• Assemblers must also provide some pseudo-ops to help programmers create complete assembly language programs. Pseudo-op allows data values to be loaded into memory locations.

```
label1  ADR r4,c
LDR r0,[r4]      ; a comment
               ADR r4,d
LDR r1,[r4]
SUB r0,r0,r1     ; another comment
```

INSTRUCTION SETS:
• The instruction set of the computer defines the interface between software modules and the underlying hardware; the instructions define what the hardware will do under certain circumstances.
• Instructions can have a variety of characteristics, including:
  ➢ Fixed versus variable length.
  ➢ Addressing modes.
  ➢ Numbers of operands.
  ➢ Types of operations supported.

Processor and Memory Organization
• ARM7 is a von Neumann architecture machine, and it supports two basic types of data:
  ➢ The standard ARM word is 32 bits long.
The word may be divided into four 8-bit bytes.

- ARM allows addresses upto 32 bits long. An address refers to a byte, not a word.
  - word 0 has location 0,
  - word 1 has location 4,
  - word 2 has location 8, and so on

- The ARM processor can be configured at power-up to address the bytes in a word in either little-endian mode (with the lowest-order byte residing in the low-order bits of the word) or big-endian mode (the lowest-order byte stored in the highest bits of the word).

**Data Operations:**

- In the ARM processor, arithmetic and logical operations cannot be performed directly on memory locations.
- While some processors allow such operations to directly reference main memory.
- ARM is a load-store architecture and data operands must first be loaded into the CPU and then stored back to main memory to save the results.
- The other important basic register in the programming model is the current program status register (CPSR).
- This register is set automatically during every arithmetic, logical, or shifting operation.
- The top four bits of the CPSR hold the following useful information about the results of that arithmetic/logical operation:
  - The negative (N) bit is set when the result is negative in two’s-complement arithmetic.
  - The zero (Z) bit is set when every bit of the result is zero.
  - The carry (C) bit is set when there is a carry out of the operation.
  - The overflow (V) bit is set when an arithmetic operation results in an overflow.

```c
int a, b, c, x, y, z;

x = (a b) c;
y = a^b c;
z = (a << 2) | (b & 15);
```

<table>
<thead>
<tr>
<th>For Arithmetic</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>ADC</td>
<td>Add with carry</td>
<td>Add with carry</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract</td>
<td>Subtract</td>
</tr>
<tr>
<td>SBC</td>
<td>Subtract with carry</td>
<td>Subtract with carry</td>
</tr>
<tr>
<td>RSB</td>
<td>Reverse subtract</td>
<td>Reverse subtract</td>
</tr>
<tr>
<td>RSC</td>
<td>Reverse subtract with carry</td>
<td>Reverse subtract with carry</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply</td>
<td>Multiply</td>
</tr>
<tr>
<td>MLA</td>
<td>Multiply and accumulate</td>
<td>Multiply and accumulate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For Logical</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Bit-wise and</td>
<td>Bit-wise and</td>
</tr>
<tr>
<td>ORR</td>
<td>Bit-wise or</td>
<td>Bit-wise or</td>
</tr>
<tr>
<td>EOR</td>
<td>Bit-wise exclusive-or</td>
<td>Bit-wise exclusive-or</td>
</tr>
<tr>
<td>BIC</td>
<td>Bit clear</td>
<td>Bit clear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>For Shift/rotate</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSL</td>
<td>Logical shift left (zero fill)</td>
<td>Logical shift left (zero fill)</td>
</tr>
<tr>
<td>LSR</td>
<td>Logical shift right (zero fill)</td>
<td>Logical shift right (zero fill)</td>
</tr>
<tr>
<td>ASL</td>
<td>Arithmetic shift left</td>
<td>Arithmetic shift left</td>
</tr>
<tr>
<td>ASR</td>
<td>Arithmetic shift right</td>
<td>Arithmetic shift right</td>
</tr>
<tr>
<td>ROR</td>
<td>Rotate right</td>
<td>Rotate right</td>
</tr>
<tr>
<td>RRX</td>
<td>Rotate right extended with C</td>
<td>Rotate right extended with C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ARM Comparison</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>Compare</td>
</tr>
</tbody>
</table>
Example
- The statement \( x = (a + b) - c \); can be implemented by using \( r0 \) for \( a \), \( r1 \) for \( b \), \( r2 \) for \( c \), and \( r3 \) for \( x \). We also need registers for indirect addressing.
- In this case, we will reuse the same indirect addressing register, \( r4 \), for each variable load.
- The code must load the values of \( a \), \( b \), and \( c \) into these registers before performing the arithmetic, and it must store the value of \( x \) back to memory when it is done.
- This code performs the following necessary steps:
  
  ADR r4,a ; get address for \( a \)  
  LDR r0,[r4] ; get value of \( a \)  
  ADR r4,b ; get address for \( b \), reusing \( r4 \)  
  LDR r1,[r4] ; load value of \( b \)  
  ADD r3,r0,r1 ; set intermediate result for \( x \) to \( a + b \)  
  ADR r4,c ; get address for \( c \)  
  LDR r2,[r4] ; get value of \( c \)  
  SUB r3,r3,r2 ; complete computation of \( x \)  
  ADR r4,x ; get address for \( x \)  
  STR r3,[r4] ; store \( x \) at proper location

CPU:
- CPUs that do not directly relate to their instruction sets.
- We consider a number of mechanisms that are important to interfacing to other system elements, such as interrupts and memory management.
- Also consider aspects of the CPU other than functionality, performance and power consumption are both very important attributes of programs that are only indirectly related to the instructions.

PROGRAMMING INPUT AND OUTPUT:
- The basic techniques for I/O programming can be understood relatively independent of the instruction set.
- This section, covers the basics of I/O programming of both the ARM and C55x.

Input and Output Devices:
- Input and output devices usually have some analog or non electronic component for instance, a disk drive has a rotating disk and analog read/write electronics.
• But the digital logic in the device that is most closely connected to the CPU very strongly resembles the logic.

Fig 1.7 Structure of a typical I/O device.
• Fig 1.7 shows the structure of a typical I/O device and its relationship to the CPU.
• The interface between the CPU and the device’s internals is a set of registers and communication by reading and writing the registers.
• Devices typically have several registers:
  ➢ **Data registers** hold values that are treated as data by the device, such as the data read or written by a disk.
  ➢ **Status registers** provide information about the device’s operation, such as whether the current transaction has completed. It is a read-only register.

**Example for interfacing I/O devices**

**Interfacing the 8251 UART**
• The UART is programmable for a variety of transmission and reception parameters. However, the basic format of transmission is simple. Data are transmitted as streams of characters, each of which has the following form:

![ UART Data Format Diagram ]

Fig 1.7.1
• Every character starts with a start bit (a 0) and a stop bit (a 1).
• The start bit allows the receiver to recognize the start of a new character.
• The stop bit ensures that there will be a transition at the start of the stop bit.
• The data bits are sent as high and low voltages at a uniform rate. That rate is known as the baud rate; the period of one bit is the inverse of the baud rate.

Before transmitting or receiving data, the CPU must set the UART’s mode registers to correspond to the data line’s characteristics.
  ➢ Baud rate;
  ➢ The number of bits per character (5 through 8);
  ➢ Whether parity is to be included and whether it is even or odd; and
  ➢ The length of a stop bit (1, 1.5, or 2 bits).

**Input and Output Primitives:**
EC6703- EMBEDDED AND REAL TIME SYSTEMS     UNIT-I     INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

- Microprocessors can provide programming support for input and output in two ways:
  - I/O instructions and
  - Memory-mapped I/O.
- Some architectures, such as the Intel x86, provide special instructions for input and output.
- These instructions provide a separate address space for I/O devices.
- But the most common way to implement I/O is by memory mapping even CPUs that provide I/O instructions can also implement memory-mapped I/O.
- As the name implies, memory-mapped I/O provides addresses for the registers in each I/O device. Example illustrates memory-mapped I/O on the ARM.

```
DEV1 EQU 0x1000
```

The program to read and write the device register:
```
LDR r1,#DEV1 ; set up device address
LDR r0,[r1] ; read DEV1
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write 8 to device
```

**Busy-Wait I/O:**

- The basic way to use devices in a program is busy-wait I/O.
- Typically devices are slower than the CPU and may require many cycles to complete an operation.
- If the CPU is performing multiple operations on a single device, such as writing several characters to an output device, then it must wait for one operation to complete before starting the next one.
- An I/O device whether it is finished by reading its status register is often called polling.

**Interrupts:**

- The interrupt mechanism allows devices to signal the CPU and to force execution of a particular piece of code.
- To overcome the inefficiency in busy wait I/O we go for interrupt.
- When an interrupt occurs, the program counter’s value is changed to point to an interrupt handler routine (also commonly known as a device driver) that takes care of the device: writing the next data, reading data that have just become ready, and so on.
- As shown in Fig 1.7.2, the interface between the CPU and I/O device includes the following signals for interrupting:
  - The I/O device asserts the interrupt request signal when it wants service from the CPU; and
  - The CPU asserts the interrupt acknowledge signal when it is ready to handle the I/O device’s request.

![Interrupt mechanism diagram]

**Fig 1.7.2** The interrupt mechanism

M.DEEPIKA AP/ECE
KSKCET

Downloaded From Rejinpaul Network
Above diagram illustrate the interrupt mechanism.

When the status register of the device goes into the ready state the CPU relative an interrupt.

If CPU perform some operation it may not be able to handle the interrupt immediately.

If the CPU handle the interrupt request it send the acknowledgement for those interrupt request further the data transmission done.

The program that runs when no interrupt is being handled is often called the foreground program; when the interrupt handler finishes, it returns to the foreground program, wherever processing was interrupted.

Example: Copying characters from input to output with basic interrupts and buffers.

We do not need to wait for each character, we can make this I/O program more sophisticated than the one in example of given problem.

Rather than reading a single character and then writing.

The read and write routines communicate through the following global variables:
- A character string io_buf will hold a queue of characters that have been read but not yet written.
- A pair of integers buf_start and buf_end will point to the first and last characters read.
- An integer error_buf will be set to 0 whenever io_buf overflows.

The global variables allow the input and output devices to run at different rates.

The queue io_buf acts as a wraparound buffer—we add characters to the tail when an input is received and take characters from the tail when we are ready for output.

The head and tail wrap around the end of the buffer array to make most efficient use of the array.

The head and tail are equal, we know that the queue is empty.

![Head Tail](image)

**Fig 1.7.3**

When the first character is read, the tail is incremented after the character is added to the queue, leaving the buffer and pointers looking like the following:

![a Head Tail](image)

**Fig 1.7.4**

When the buffer is full, we leave one character in the buffer unused.

![a b c d e f g Head Tail](image)

**Fig 1.7.5**

If we added another character and updated the tail buffer (wrapping it around to the head of the buffer), we would be unable to distinguish a full buffer from an empty one.

![b c d e f g h Head Tail](image)

**Fig 1.7.6**
Here is what happens when the output goes past the end of io_buf.

- A UML sequence diagram shows how input and output are interleaved with the foreground program.
- The simulation shows that the foreground program is not executing continuously, but it continues to run in its regular state independent of the number of characters waiting in the queue.

**Fig 1.7.7**

- Interrupts allow a lot of concurrency, which can make very efficient use of the CPU.
- But when the interrupt handlers are buggy, the errors can be very hard to find.
- The fact that an interrupt can occur at any time means that the same bug can manifest itself in different ways when the interrupt handler interrupts different segments of the foreground program.

**SUPERVISOR MODE, EXCEPTIONS, AND TRAPS:**

- It is a mode that provide hardware checks to ensure that the programs do not interfere with each other.
- It is provided by the CPU.
- For example, the addresses of the memory locations to be dynamically.
- Control of the memory management unit (MMU) is typically reserved for supervisor mode to avoid the obvious problems that could occur when program bugs cause inadvertent changes in the memory management registers.
- The ARM instruction that puts the CPU in supervisor mode is called SWI:
  
  ```
  SWI CODE_1
  ```
- SWI causes the CPU to go into supervisor mode and the argument to SWI is a 24-bit immediate value.

**Exceptions**

- An exception is an internally detected error.
- A simple example is division by zero.
- To handle this problem would be to check every divisor before division to be sure it is not zero, but this would both substantially increase the size of numerical programs and CPU time evaluating the divisor’s value.
- It requires both prioritization and vectoring.

**Prioritization:**

- Exceptions must be prioritized because a single operation may generate more than one exception.
- For example, an illegal operand and an illegal memory access.
EC6703- EMBEDDED AND REAL TIME SYSTEMS  UNIT-I  INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

Vectoring
- Vectoring provides a way for the user to specify the handler for the exception condition.
- The vector number for an exception is usually predefined by the architecture.

Traps:
- A trap, also known as a software interrupt, is an instruction that explicitly generates an exception condition.
- The most common use of a trap is to enter supervisor mode.

CO-PROCESSORS:
- CPU architects often want to provide flexibility in what features are implemented in the CPU.
- One way to provide such flexibility at the instruction set level is to allow co-processors, which are attached to the CPU and implement some of the instructions.
- A CPU also receive co-processor instructions even when there is no coprocessor attached.
- The ARM architecture provides support for up to 16 co-processors.
- Co-processors are able to perform load and store operations on their own registers.
- They can also move data between the co-processor registers and main ARM registers.

Example: floating-point unit. The unit occupies two co-processor units in the ARM architecture, numbered 1 and 2, but it appears as a single unit to the programmer. It provides eight 80-bit floating-point data registers, floating-point status registers, and an optional floating-point status register.

Intel Processors:
- The original IBM PC included a socket for the Intel 8087 floating point co-processor which was a popular option for people using the PC for CAD or mathematics intensive calculations. In that architecture, the co-processor speedup floating-point arithmetic on the order of fiftyfold.
- Users that only used the PC for word processing, for example saved the high cost of the co-processor, which would not have accelerated performance of text manipulations operations.
- The 8087 was tightly integrated with 8086/8088 and responded to floating-point machine code operation codes inserted in 8088 instruction stream.
- An 8088 processor without an 8087 could not interrupt these instructions, require separate versions of programs for FPU and non-FPU systems at least a test at run time to detect the FPU and select appropriate mathematical library functions.
- Another co-processor for the 8086/8088 central processor was the 8089 input/output co-processor.
- It used the same programming technique as 8087 for input/output operations, such as transfer of data from memory to a peripheral device and so reducing the load on the CPU.
- But IBM did not use it in IBM PC design and Intel stopped developed of this type of co-processor.
- During the era of 8 and 16-bit desktop computers another common source of floating-point co-processor was Weitek.
- The Intel 80386 microprocessor used an optional “math” co-processor to perform floating-point operations directly in hardware.
- The Intel 80486 DX processor included floating-point hardware on the chip released a cost reduced processor, the 80486SX that had no floating-point hardware and also sold an 80487SX co-processor that essentially disabled the main processor when installed, since the 80487SX was a complete 80486DX with a different set of pin connections.
Intel Processors later than the 80486 integrated floating-point hardware on the main processor chip; the advances in integration eliminated the cost advantage of selling the floating-point processor as an optional element.

- It would be very difficult to adapt circuit board techniques adequate at 75 MHz processor speed to meet the time delay, power consumption and radio frequency interference standards required at GHz range clock speeds.
- These on-chip floating-point processors are still referred to as co-processors because they operate in parallel with the main CPU.

**Other Co-processors:**

- The MIPS architecture supports up to four co-processor units, used for memory management, floating point arithmetic and two undefined co-processors for other tasks such as graphics accelerations.
- Using FPGA custom co-processors can be created for acceleration of particular processing tasks such as digital signal processing.
- TLS/SSL accelerators used on servers.
- Some multicore chips can be programmed so that one of their processor is the primary processor and the other processors are supporting co-processors.

**MEMORY SYSTEM MECHANISMS:**

- Microprocessor clock rates are increasing at a faster rate than memory speeds, such that memories are falling further and further behind microprocessors every day.
- As a result, computer architects resort to caches to increase the average performance of the memory system.

**Caches**

- Caches are widely used to speed up memory system performance.
- Many microprocessor architectures include caches as part of their definition.
- A cache is a small, fast memory that holds copies of some of the contents of main memory.
- Because the cache is fast, it provides higher-speed access for the CPU. Due to small size of cache, it cannot satisfy all request.

**Fig 1.8 The cache in the memory system.**

- A cache controller mediates between the CPU and the memory system comprised of the main memory.
- The cache controller sends a memory request to the cache and main memory.
- If the requested location is in the cache, the cache controller forwards the location’s contents to the CPU and aborts the main memory request; this condition is known as a cache hit.
- If the location is not in the cache, the controller waits for the value from main memory and forwards it to the CPU; this situation is known as a cache miss.

**Types of cache misses:**

- We can classify cache misses into several types depending on the situation that generated them:
A compulsory miss (also known as a cold miss) occurs the first time a location is used,
- A capacity miss is caused by a too-large working set,and
- A conflict miss happens when two locations map to the same location in the cache.

- Let \( h \) be the hit rate, the probability that a given memory location is in the cache. It follows that \( 1-h \) is the 
miss rate, or the probability that the location is not in the cache.
- Then we can compute the average memory access time as
  
  \[
  t_{ave} = ht_{cache} + (1-h)t_{main}. \quad \text{..........(1.2)}
  \]
- Where \( t_{cache} \) is the access time of the cache and \( t_{main} \) is the main memory access time. The best-case memory
  access time is \( t_{cache} \), while the worst-case access time is \( t_{main} \).
- Given that \( t_{main} \) is typically 50–60 ns for DRAM, while \( t_{cache} \) is at most a few nanoseconds, the spread
  between worst-case and best-case memory delays is substantial.
- The hit rate depends on the program being executed and the cache organization,
- Modern CPUs may use multiple levels of cache as shown in Fig 1.8.1.
- The first-level cache (commonly known as L1 cache) is closest to the CPU, the second-level cache (L2 cache)
  feeds the first-level cache, and so on.
- The second-level cache is much larger but is also slower. If \( h_1 \) is the first-level hit rate and \( h_2 \) is the rate at
  which access hit the second-level cache but not the first-level cache, then the average access time for a two-level
  cache system is
  
  \[
  t_{ave} = h_1t_{L1} + h_2t_{L2} + (1-h_1-h_2)t_{main}. \quad \text{..........(1.3)}
  \]

Fig 1.8.1 A two-level cache system.
- As the program’s working set changes, we expect locations to be removed from the cache to make way for
  new locations.
- When set-associative caches are used, we have to think about what happens when we throw out a value from
  the cache to make room for a new value.
- In direct-mapped caches because every location maps onto a unique block, but in a set-associative cache we
  must decide which set will have its block thrown out to make way for the new block.
- In least recently used (LRU), that is, throw out the block that has been used farthest in the past.
- In random replacement, which requires even less hardware to implement.

**Direct-mapped cache**
- The simplest way to implement a cache is a direct-mapped cache, as shown in Fig 1.8.2.
**Fig 1.8.2 A direct-mapped cache**

- The cache consists of cache blocks, each of which includes a tag to show which memory location is represented by this block, a data field holding the contents of that memory, and a valid tag to show whether the contents of this cache block are valid.
- An address is divided into three sections. The index is used to select which cache block to check. The tag is compared against the tag value in the block selected by the index.
- If the length of the data field is longer than the minimum addressable unit, then the lowest bits of the address are used as an offset to select the required value from the data field.
- In write-through every write changes both the cache and the corresponding main memory.
- We can reduce the number of times we write to main memory by using a write-back policy.
- If we write only when we remove a location from the cache, we eliminate the writes when a location is written several times before it is removed from the cache.

**Advantages**
- Very fast and
- Relatively Low Cost.

**Limitation**
- Caching power due to its simple scheme for mapping the cache onto main memory.

**Set-associative cache**
- The limitations of the direct-mapped cache can be reduced by going to the set-associative cache structure shown in Fig 1.8.3.

**Fig 1.8.3 A set-associative cache.**
• A set-associative cache is characterized by the number of banks or ways it uses, giving an n-way set-associative cache.
• A set is formed by all the blocks that share the same index.
• Each set is implemented with a direct-mapped cache.
• A cache request is broadcast to all banks simultaneously.
• If any of the sets has the location, the cache reports a hit.

**Memory Management Unit**
• A Memory Management Unit sometime called page memory management unit, is a computer hardware component responsible for handling accesses to memory requested by the CPU.
• Its function include translation or virtual addresses to physical addresses memory protection, cache control, bus arbitration and in simpler computer architecture bank switching.
• Memory Management is the hardware component that manages virtual memory systems
• Among the functions of such devices are the translation of virtual addresses to physical addresses memory protection, cache control, bus arbitration and in simpler bank switching.
• The MMU is the part of CPU though in some designs it is separate chip, the MMU includes a small amount of memory that holds a table matching virtual addresses and physical addresses.
• This table is called the Table Look-aside Buffer (TLB).
• All requests for data are sent to the MMU, which determines whether the data is in RAM or needs to be fetched from the mass storage device.
• If the data is not in memory, the MMU issues a page fault interrupt.

**CPU PERFORMANCE**
• To increase the performance and efficiency CPUs we consider two factors that influence program:
  ➢ Pipelining and
  ➢ Caching.

**Pipelining**
• Modern CPUs are designed as pipelined machines in which several instructions are executed in parallel. Pipelining greatly increases the efficiency of the CPU.
• Pipelining process CPU consists of three stages
  ➢ Fetch the instruction is fetched from memory.
  ➢ Decode the instruction’s opcode and operands are decoded to determine what function to perform.
  ➢ Execute the decoded instruction is executed.
• Each of these operations requires one clock cycle for typical instructions.
• Thus, a normal instruction requires three clock cycles to completely execute, known as the latency of instruction execution.
• But the pipeline has three stages, that too instruction is completed in every cycle.
• For ARM processor the RISC machines are designed to keep the pipeline busy.
• Pipelined RISC machines typically have more regular timing characteristics.
• For example consider some set of instruction performed by pipelining process.
Fig 1.8.4 Pipelined execution of ARM instructions.
- The above three instruction are executed in sequence and also these instruction are performed in few clock cycle rather than the complex execution phase.

Caching
- Caches are used to substantially reduce memory access time when the requested location is in the cache.
- However, the desired location is not always in the cache since it is considerably smaller than main memory.
- As a result, caches cause the time required to access memory to vary considerably.
- The extra time required to access a memory location not in the cache is often called the cache miss penalty.
- As we have seen, a location may not be in the cache for several reasons.
  - At a compulsory miss, the location has not been referenced before.
  - At a conflict miss, two particular memory locations are fighting for the same cache line.
  - At a capacity miss, the program’s working set is simply too large for the cache.
- The contents of the cache can change considerably over the course of execution of a program.

CPU POWER CONSUMPTION:
- Power consumption is, in some situations, as important as execution time.
- Power consumption depends on characteristics of CPU and mechanisms provided by CPU.
- First, it is important to distinguish between energy and power. Power is, of course, energy consumption per unit time. Heat generation depends on power consumption.
- The high-level power consumption characteristics of CPUs and other system components are derived from the circuits used to build those components.
- Today, virtually all digital systems are built with complementary metal oxide semi-conductor (CMOS) circuitry.
- CMOS power consumption are easily identified and briefly described below.
1. Voltage drops:
- The dynamic power consumption of a CMOS circuit is proportional to the square of the power supply voltage (V2).
- Reducing the power supply voltage to the lowest level we can obtain the reduced power consumption, parallel hardware design also reduces the power supply voltage to obtain low power consumption.
2. Toggling:
- A CMOS circuit uses most of its power when it is changing its output value.
- Reducing the speed at which the circuit operates and it will reduce its power consumption.
- By eliminating unnecessary changes to the inputs of a CMOS circuit will reduce power consumption.
3. Leakage:
EC703- EMBEDDED AND REAL TIME SYSTEMS  UNIT-I  INTRODUCTION TO EMBEDDED COMPUTING AND ARM PROCESSORS

- Even when a CMOS circuit is not active, some charge leaks out of the circuit’s nodes through the substrate.
- The only way to eliminate leakage current is to remove the power supply.

**Power-Saving Strategies used in CMOS CPUs:**
- CPUs can be used at reduced voltage levels. For example, reducing the power supply from 1 to 0.9 V causes the power consumption to drop by 12. 0.92 - 1.2X.
- The CPU can be operated at a lower clock frequency to reduce power (but not energy) consumption.
- The CPU may internally disable certain function units that are not required for the currently executing function. This reduces energy consumption.
- Some CPUs allow parts of the CPU to be totally disconnected from the powersupply to eliminate leakage currents.

**Power Management Features:**
There are two types of power management features provided by CPUs.
- **static power management**
- **dynamic power management**

- A static power management mechanism is invoked by the user but does not otherwise depend on CPU activities.
  - Example: powerdownmode intended to save energy.
- A dynamic power management mechanism takes actions to control power based upon the dynamic activity in the CPU.
  - Example, the CPU may turn off certain sections of the CPU when the instructions being executed do not need them.

![Power State Machine Diagram](image)

**Fig 1.9 A power state machine for a processor.**
- In the example each state in the machine represents a different mode of the machine and every state is labeled with its average power consumption.
- The example machine has two states: run mode with power consumption \( P_{\text{run}} \) and sleep mode with power consumption \( P_{\text{sleep}} \).
- Transitions show how the machine can go from state to state; each transition is labeled with the time required to go from the source to the destination state.

**Application- Example:**
Power-saving modes of the Strong ARM SA-1100
- The StrongARM SA-1100 is designed to provide sophisticated power management capabilities that are controlled by the on-chip power manager.
- The processor takes two power supplies, as seen in the following fig 1.20.
Fig 1.20

- VDD is the main power supply for the core CPU and is nominally 3.3 V.
- The VDDX supply is used for the pins and other logic such as the power manager; it is normally at 1.5 V.
- The system can supply two inputs about the status of the power supply.
- VDD_FAULT tells the CPU that the main power supply is not being properly regulated, while BATT_FAULT indicates that the battery has been removed or is low.
- Either of these events can cause the CPU to go into a low-power mode.
- In low-power operation, the VDD supply can be turned off.
- When resuming operation, the PWR_EN signal is used by the CPU to tell the external power supply to ramp up the VDD power supply.
- A system power manager can both monitor the CPU and other devices and control their operation to gracefully transition between power modes.
- It provides several registers that allow programs to control power modes, determine why power modes were entered, determine the current state of power management modes, and so on.
- The SA-1100 provides the three power modes described below.
  - Run mode is normal operation and has the highest power consumption.
  - Idle mode saves power by stopping the CPU clock. The system unit modules real-time clock, operating system timer, interrupt control, general-purpose I/O, and power manager—all remain operational. Idle mode is entered by executing a three-instruction sequence. The CPU returns to run mode upon receiving an interrupt from one of the internal system units or from a peripheral or by resetting the CPU. This causes the machine to restart the CPU clock and to resume execution where it left off.
  - Sleep mode shuts off most of the chip’s activity. Entering sleep mode causes the system to shut down on-chip activity, reset the CPU, and negate the PWR_EN pin to tell the external electronics that the chip’s power supply should be driven to 0 V.
- A separate I/O power supply remains on and supplies power to the power manager so that the CPU can be awakened from sleep mode; the low-speed clock keeps the power manager running at low speeds sufficient to manage sleep mode.
- The CPU software should set several registers to prepare for sleep mode. Sleep mode is entered by forcing the sleep bit in the power manager control register; it can also be entered by a power supply fault.
- The sleep shutdown sequence happens in three steps, each of which requires about 30 s. The machine wakes up from sleep state on a preprogrammed wake-up event.
- The wake-up sequence has three steps: the PWR_EN pin is asserted to turn on the external power supply and waits for about 10 ms; the 3.686-MHz oscillator is ramped up to speed; and the internal reset is negated and the CPU boot sequence begins.
Fig 1.20.1 Power State Machine of the SA-1100

- The sleep mode saves over three orders of magnitude of power consumption. However, the time required to reenter run mode from sleep is over a tenth of a second.
UNIT II
EMBEDDED COMPUTING PLATFORM DESIGN

INTRODUCTION:

- The microprocessor is an important element of the embedded computing system, but it cannot do its job without memories and I/O devices.
- We need to understand how to interconnect microprocessors and devices using the CPU bus.

THE CPU BUS:

- A computersystem encompasses much more than the CPU; it also includes memory and I/O devices.
- The bus is the mechanism by which the CPU communicates with memory and devices.
- A bus is a minimum collection of wires, but the bus also defines a protocol by which the CPU, memory, and devices communicate.
- Major roles of the bus are to provide an interface to memory. Input and output devices are connected to the bus for data transmission.

Types of Bus:

- Data bus
- Address bus
- Control bus
- System bus

Bus Protocols:

- Protocols are the set of rules and conditions for data communication.
- The basic building block of most bus protocols is the four-cycle handshake, as shown in Fig. 2.1. Fig 2.1 The four-cycle handshake
- The handshake ensures that when two devices want to communicate, one is ready to transmit and the other is ready to receive.
- The handshake uses a pair of wires dedicated to the handshake: enq (meaning enquiry) and ack (meaning acknowledge). Extra wires are used for the data transmitted during the handshake.
- The four cycles are described below.
  - Device 1 raises its output to signal an enquiry, which tells device 2 that it should get ready to listen for data.
  - When device 2 is ready to receive, it raises its output to signal an acknowledgment. At this point, devices 1 and 2 can transmit or receive.
  - Once the data transfer is complete, device 2 lowers its output, signaling that it has received the data.
  - After seeing that ack has been released, device 1 lowers its output.
- Microprocessor buses build on the handshake for communication between the CPU and other system components. The term bus is used in two ways.
  - A set of related wires, such as address wires.
  - A protocol for communicating between components.

Major Components:

- Clock provides synchronization to the bus components.
- R/W is true when the bus is reading and false when the bus is writing.
- Address is an a-bit bundle of signals that transmits the address for an access.
Data is an n-bit bundle of signals that can carry data to or from the CPU, and data ready signals when the values on the data bundle are valid.

- All transfers on this bus are controlled by the CPU.
- The CPU can read or write a device or memory, but devices or memory cannot initiate a transfer.
- Only the CPU can determine the address and direction of the transfer.

**Fig 2.1.1 A typical microprocessor bus**

- The behavior of a bus is most often specified as a timing diagram.
- A timing diagram shows how the signals on a bus vary over time, but since values like the address and data can take on many values, as shown in Fig 2.1.2.
- A stable signal has, as the name implies, a stable value that could be measured by an oscilloscope, but the exact value of that signal does not matter for purposes of the timing diagram.
- A changing signal does not have a stable value.
- Changing signals should not be used for computation.
- To be sure that signals go to their proper values at the proper times, timing diagrams sometimes show timing constraints.
- The state machine view of the bus transaction is also helpful and a useful complement to the timing diagram as shown in Fig 2.1.3.

**Fig 2.1.2 Timing diagram notation**

**Fig 2.1.3 State diagrams for the bus read transaction**

As with a timing diagram, we do not show all the possible values of address and data lines but instead concentrate on the transitions of control signals.

Some buses have data bundles that are smaller than the natural word size of the CPU. Using fewer data lines reduces the cost of the chip.
Fig 2.1.4 Bus signals for multiplexing address and data

- A more complicated protocol hides the smaller data sizes from the instruction execution unit in the CPU.
- Byte addresses are sequentially sent over the bus, receiving one byte at a time; the bytes are assembled inside the CPU’s bus logic before being presented to the CPU properly.
- Some buses use multiplexed address and data. As shown in Fig 2.1.4, additional control lines are provided to tell whether the value on the address/data lines is an address or data.
- The address comes first on the combined address/data lines, followed by the data. The address can be held in a register until the data arrive so that both can be presented to the device at the same time.

**DMA (Direct Memory Access):**
- Standard bus transactions require the CPU to be in the middle of every read and write transaction.
- There are certain types of data transfers in which the CPU does not need to be involved.
- A high-speed I/O device may want to transfer a block of data into memory. While it is possible to write a program that alternately reads the device and writes to memory, it would be faster to eliminate the CPU’s involvement and let the device and memory communicate directly.
- This capability requires that some unit other than the CPU be able to control operations on the bus.

![ DMA diagram ]

**Fig 2.1.5A bus with a DMA controller**

- **DMA** is a bus operation that allows reads and writes not controlled by the CPU.
- A DMA transfer is controlled by a DMA controller, which requests control of the bus from the CPU.
- After gaining control, the DMA controller performs read and write operations directly between devices and memory.
- The DMA requires the CPU to provide two additional bus signals:
  - The bus request is an input to the CPU through which DMA controllers ask for ownership of the bus.
  - The bus grant signals that the bus has been granted to the DMA controller.
- A device that can initiate its own bus transfers is known as a **bus master**. Devices that do not have the capability to be bus masters do not need to connect to a bus request and bus grant.
- The DMA controller uses these two signals to gain control of the bus using a classic four-cycle handshake.
The bus request is asserted by the DMA controller when it wants to control the bus, and the bus grant is asserted by the CPU when the bus is ready.

The CPU will finish all pending bus transactions before granting control of the bus to the DMA controller. When it does grant control, it stops driving the other bus signals: R/W, address, and so on.

Once the DMA controller is master, it can perform reads and writes using the same bus protocol as with any CPU-driven bus transaction.

Memory and devices do not know whether a read or write is performed by the CPU or by a DMA controller. After the transaction is finished, the DMA controller returns the bus to the CPU by deasserting the bus request, causing the CPU to deassert the bus grant.

The CPU controls the DMA operation through registers in the DMA controller.

A typical DMA controller includes the following three registers:
- A starting address register specifies where the transfer is to begin.
- A length register specifies the number of words to be transferred.
- A status register allows the DMA controller to be operated by the CPU.

The CPU initiates a DMA transfer by setting the starting address and length registers appropriately and then writing the status register to set its start transfer bit.

After the DMA operation is complete, the DMA controller interrupts the CPU to tell it that the transfer is done.

If the CPU has enough instructions and data in the cache and registers, it may be able to continue doing useful work for quite sometime and may not notice the DMA transfer.

But once the CPU needs the bus, it stalls until the DMA controller returns bus mastership to the CPU.

To prevent the CPU from idling for too long, most DMA controllers implement modes that occupy the bus for only a few cycles at a time.

The DMA controller returns control of the bus to the CPU and goes to sleep for a preset period.

System Bus Configurations:
- A microprocessor system often has more than one bus. As shown in Fig 2.1.6, high-speed devices may be connected to a high-performance bus, while lower-speed devices are connected to a different bus.

![Fig 2.1.6 A multiple bus system](image-url)

- A small block of logic known as a bridge allows the buses to connect to each other. There are several good reasons to use multiple buses and bridges:
  - Higher-speed buses may provide wider data connections.
  - A high-speed bus usually requires more expensive circuits and connectors. The cost of low-speed devices can be held down by using a lower-speed, lower-cost bus.
The bridge may allow the buses to operate independently, thereby providing some parallelism in I/O operations.

- The bridge is a slave on the fast bus and the master of the slow bus. The bridge takes commands from the fast bus on which it is a slave and issues those commands on the slow bus.
- It also returns the results from the slow bus to the fast bus.
- For example, it returns the results of a read on the slow bus to the fast bus.
- The upper sequence of states handles a write from the fast bus to the slow bus. These states must read the data from the fast bus and set up the handshake for the slow bus.
- Operations on the fast and slow sides of the bus bridge should be overlapped as much as possible to reduce the latency of bus-to-bus transfers.
- The bridge serves as a protocol translator between the two bridges as well. If the bridges are very close in protocol operation and speed, a simple state machine may be enough.
- If there are larger differences in the protocol and timing between the two buses, the bridge may need to use registers to hold some data values temporarily.

**AMBA Bus:**

- The ARM CPU is manufactured by many different vendors, the bus provided off-chip can vary from chip to chip.
- ARM has created a separate bus specification for single-chip systems. The AMBA bus [ARM99A] supports CPUs, memories, and peripherals integrated in a system-on-silicon.

**AMBA specification includes two buses:**

- AMBA high-performance bus (AHB)
- AMBA peripherals bus (APB)

- The AMBA high-performance bus (AHB) is optimized for high-speed transfers and is directly connected to the CPU as shown in fig 2.1.7.
- It supports several high-performance features: pipelining, burst transfers, split transactions, and multiple bus masters.
- A bridge can be used to connect the AHB to an AMBA peripherals bus (APB). This bus is designed to be simple and easy to implement; it also consumes relatively little power.
- The AHB assumes that all peripherals act as slaves, simplifying the logic required in both the peripherals and the bus controller. It also does not perform pipelined operations, which simplifies the bus logic.

**MEMORY DEVICES:**

- Memory is the most important component in embedded system design.
From the memory only we can perform any kinds of read and write operations, memory is a collection of memory cells arranged in specific manner.

- Bus is interfaced with memory to perform read/write operations between memory devices.
- Memory devices deal basic functions of memory and how the memories are organized in embedded systems.

**MEMORY DEVICE ORGANIZATION:**

- The most basic way to characterize a memory is by its capacity, such as 256 MB.
- However, manufacturers usually make several versions of a memory of a given size, each with a different data width.
  - For example, a 256-MB memory may be available in two versions:
    - As a 64 M × 4-bit array, a single memory access obtains an 8-bit data item, with a maximum of $2^{26}$ different addresses.
    - As a 32 M × 8-bit array, a single memory access obtains a 1-bit data item, with a maximum of $2^{23}$ different addresses.
- The height/width ratio of a memory is known as its aspect ratio.
- The best aspect ratio depends on the amount of memory required.
- Internally, the data are stored in a two-dimensional array of memory cells as shown in Fig 2.2.
- Because the array is stored in two dimensions, then bit address received by the chip is split into a row and a column address.

![Fig 2.2 Internal organization of a memory device](image)

- The row and column select a particular memory cell.
- If the memory’s external width is 1 bit, the column address selects a single bit; for wider data widths, the column address can be used to select a subset of the columns.
- Most memories include an enable signal that controls the tri-stating of data onto the memory’s pins.
- A read/write signal controls the direction of data transfer; memory chips do not typically have separate read and write data pins as shown in fig 2.2.

**Types of Memory:**

- Random-Access Memories (RAM)
- Read-only Memories (ROM)

**RANDOM-ACCESS MEMORIES:**

- Random-access memories can be both read and written. They are called random access because, unlike magnetic disks, addresses can be read in any order.

Two types of RAM:
Static RAM: It means that it is having fixed programming, it cannot be changed.

Dynamic RAM:
- It can be changed we can reprogramme.
- Most bulk memory in modern systems is dynamic RAM (DRAM).
- DRAM is very dense; it does, however, require that its values be refreshed periodically since the values inside the memory cells decay over time.
- The dominant form of dynamic RAM today is the synchronous DRAMs (SDRAMs), which uses clocks to improve DRAM performance.
- The timing diagram of read operation is shown in fig 2.2.1.
- SDRAMs use Row Address Select (RAS) and Column Address Select (CAS) signals to break the address into two parts, which select the proper row and column in the RAM array.
- Signal transitions are relative to the SDRAM clock which allows the internal SDRAM operations to be pipelined.

![Timing diagram for a read on a synchronous DRAM](image)

- SDRAMs use a separate refresh signal to control refreshing.
- It will refresh only part of DRAMs, rather than refresh the entire memory at once.
- When a section of memory is being refreshed, it cannot be accessed until the refresh is complete.
- SDRAMs include registers that control the mode in which the SDRAM operates.
- SDRAMs support burst modes that allow several sequential addresses to be accessed by sending only one address.
- SDRAMs generally also support an interleaved mode that exchanges pairs of bytes.
- Even faster synchronous DRAMs, known as double-data rate (DDR) SDRAMs or DDR2 and DDR3 SDRAMs, are now in use. It performs more operations per clock cycle.

SIMMs and DIMMs:
- Memory for PCs is generally purchased as two methods
  - Single in-line memory modules (SIMMs)
  - Double in-line memory modules (DIMMs).
- A SIMM or DIMM is a small circuit board that fits into a standard memory socket.
- A DIMM has two sets of leads compared to the SIMM’s one.

READ-ONLY MEMORIES:
Two types of ROM:

- Factory-programmed ROM (sometimes called mask-programmed ROM) and
- Field-programmable ROM.

Factory-programmed ROMs are ordered from the factory with particular programming. Factory programming is useful only when the ROMs are to be installed in some quantity. This is also known as mask Programmed ROM.

Field-programmable ROMs, on the other hand, can be programmed in the lab.

- Example: Flash memory is electrically erasable.
- Flash memory uses standard system voltage for erasing and programming, allowing it to be programmed inside a typical system.
- Flash memory has the capacity to erase only the specific content not all data is known as boot-block flash.

**INPUT / OUTPUT DEVICES:**

- Input and output devices commonly used in embedded computing systems.
- Some of these devices are often found as on-chip devices in microcontrollers; others are generally implemented separately.

**TIMERS AND COUNTERS:**

- Timers and counters are distinguished from one another based by their use, not their logic.
- Both are built from adder logic with registers to hold the current value, with an increment input that adds one value to the current register value.
- However, a timer has its count connected to a periodic clock signal to measure time intervals, while a counter has its count input connected to an aperiodic signal in order to count the number of occurrences of some external event.
- Because the same logic can be used for either purpose, the device is often called a counter/timer.

![Fig 2.3 Internals of a counter/timer](image)

- An n-bit counter/timer uses an n-bit register to store the current state of the count and an array of half subtractors to decrement the count when the count signal is asserted as shown in fig 2.3.
EC6703- EMBEDDED AND REAL TIME SYSTEMS  UNIT-II  EMBEDDED COMPUTING PLATFORM DESIGN

- Combinational logic checks when the count equals zero; the done output signals the zero count.
- It is often useful to be able to control the time-out, rather than require exactly $2^n$ events to occur.
- For this purpose, a reset register provides the value with which the count register is to be loaded.
- The counter/timer provides logic to load the reset register. Most counters provide both cyclic and acyclic modes of operation.
- In the **cyclic mode**, once the counter reaches the done state, it is automatically reloaded and the counting process continues.
- In **acyclic mode**, the counter/timer waits for an explicit signal from the microprocessor to resume counting.
- A **watchdog timer** is an I/O device that is used for internal operation of a system.
  - As shown in Fig 2.3.1, the watchdog timer is connected into the CPU bus and also to the CPU’s reset line.
  - The CPU’s software is designed to periodically reset the watchdog timer, before the timer reverses its time-out limit.
  - If the watchdog timer ever does reach that limit, its time-out action is to reset the processor.
  - In that case, the presumptionisthat either software flaw or hardware problem has caused the CPU to misbehave.
  - To avoid this problem, the system is reset to get it operational as quickly as possible.

![Fig 2.3.1 A watchdog timer](image)

**A/D AND D/A CONVERTERS:**
- Analog/digital (A/D) and digital/analog (D/A) converters (typically known as ADCs and DACs, respectively) are often used to interface nondigital devices to embedded systems.
- A/D conversion requires more complex circuitry, it requires a somewhat more complex interface.
- Analog/digital conversion requires sampling the analog input before converting it to digital form.
- A control signal causes the A/D converter to take a sample and digitize it.
- There are several different types of A/D converter circuits, some of which take a constant amount of time, while the conversion time of others depends on the sampled value.
- Variable-time converters provide a signal so that the microprocessor knows when the value is ready.
- A typical A/D interface has, in addition to its analog inputs, two major digital inputs.
- A data port allows A/D registers to be read and written, and a clock input tells when to start the next conversion.
- D/A conversion is relatively simple, so the D/A converter interface generally includes only the data value.
- The input value is continuously converted to analog form.

**KEYBOARDS:**
- A keyboard is basically an array of switches, but it may include some internal logic to help simplify the interface to the microprocessor.
- A switch uses a mechanical contact to make or break an electrical circuit.
- The major problem with mechanical switches is that they bounce.
- When the switch is depressed by pressing on the button attached to the switch’s arm, the force of the depression causes the contacts to **bounce** several times until they settle down.
- If this is not corrected, it will appear that the switch has been pressed several times, giving false inputs.
- A hardware **debouncing** circuit can be built using a one-shot timer. Software can also be used to debounce switch inputs.
- More expensive keyboards, such as those used in PCs, actually contain a microprocessor to preprocess button inputs.
- PC keyboards typically use a 4-bit microprocessor to provide the interface between the keys and the computer.
- The microprocessor can provide debouncing, but it also provides other functions as well.
- An **encoded keyboard** uses some code to represent which switch is currently being depressed.
- At the heart of the encoded keyboard is the scanned array of switches shown in Fig 2.3.2.

![Fig 2.3.2 A Scanned key array](image)

- The scanned keyboard array reads only one row of switches at a time.
- The demultiplexer at the left side of the array selects the row to be read.
- When the scan input is 1, that value is transmitted to one terminal of each key in the row. If the switch is depressed, the 1 is sensed at that switch’s column.

**LEDs:**
- Light-emitting diodes (LEDs) are often used as simple displays by themselves, and arrays of LEDs may form the basis of more complex displays.

![Fig 2.3.3 An LED connected to a digital output](image)

- Fig 2.3.3 shows the resistor is connected between the output pin and the LED to absorb the voltage difference between the digital output voltage and the 0.7V drop across the LED.
- When the digital output goes to 0, the LED voltage is in the device’s off region and the LED is not on.

**DISPLAYS:**
- A display device may be either directly driven or driven from a frame buffer.
Typically, displays with a small number of elements are driven directly by logic, while large displays use a RAM frame buffer.

A single-digit display typically consists of seven segments; each segment may be either an LED or a liquid crystal display (LCD) element.

This display relies on the digits being visible for some time after the drive to the digit is removed.

The n-digit array is a simple example of a display directly driven.

A framebuffer is a RAM that is attached to the system bus.

The microprocessor writes values into the frame buffer in whatever order is desired.

The pixels in the frame buffer are generally written to the display in raster order by reading pixels sequentially.

LCD displays present a very different interface to the system because the array of pixel LCDs can be randomly accessed.

It has two kinds of LCD panel

- Passive Matrix
- Active Matrix

Early LCD panels were called passive matrix because they relied on a two-dimensional grid of wires to address the pixels.

Modern LCD panels use an active matrix system that puts a transistor at each pixel to control access to the LCD. It provides higher contrast and a higher-quality display.

**TOUCH SCREENS:**

- A touchscreen is an input device overlaid on an output device.
- The touchscreen registers the position of a touch to its surface.
- By overlaying this on a display, the user can react to information shown on the display.

**Types of touchscreens:**

- Resistive touchscreen
- Capacitive touchscreen

**COMPONENT INTERFACING:**

- Interfacing is the process of communicating two or more components.
- In any kind of design process, interfacing is the most important task.
- Interfacing can be performed in the following ways
  - Memory Interfacing
  - Device Interfacing

**MEMORY INTERFACING:**

- The memory structure is simple if we buy a memory size with exact for design process at the time no need of memory interfacing.
- The memory size is not sufficient for design process we can go for memory interfacing.
- It will obtain by placing RAMs in parallel manner.
- We also need logic to turn the bus signals into the appropriate memory signal and refresh signals.

**DEVICE INTERFACING:**

- Some I/O devices are designed to interface directly to a particular bus, forming glue less interfaces.
- But glue logic is required when a device is connected to a bus for which it is not designed.
- An I/O device typically requires a much smaller range of addresses than a memory, so addresses must be decoded much more finely.
- Some additional logic is required to cause the bus to read and write the device’s registers.
Example: A glue logic interface

![Diagram of glue logic interface]

**Fig 2.4**

- The above diagram is an interfacing scheme for a simple I/O device.

1. The device has four registers that can be read and written by presenting the register number on the Regid pins, asserting R/W as required, and reading or writing the value on the regval pins.
2. To interface to the bus, the bottom two bits of the address are used to refer to registers within the device, and the remaining bits are used to identify the device itself.
3. The top bits of the address are sent to a comparator for testing against the device address.
4. The device’s address can be set with switches to allow the address to be easily changed. When the bus address matches the device’s, the result is used to enable a transceiver for the data pins.
5. When the transceiver is disabled, the regval pins are disconnected from the data bus.
6. The comparator’s output is also used to modify the R/W signal.
7. The device’s R/W pin is given the value (bus R/W not-equal address), so that when the comparator’s result is not 1, the device’s R/W pin always receives a 1 to avoid inadvertently writing the device registers.

**DESIGNING WITH MICROPROCESSORS:**

- A microprocessor or microcontroller incorporates most or all of the functions of a central processing unit on a single integrated circuit.
- CPU is a logical machine that can execute computer programs.
- The mainfunction of the CPU is to execute a sequence of instructions called a program stored in some kind of computer memory.
- To make any design with microprocessor is important to know the system architecture and kind of hardware design used.

**SYSTEM ARCHITECTURE:**

- An architecture is a set of elements and the relationships between them that together form a single unit.
- The architecture of an embedded computing systemis the blueprint for implementing that system.
- An architecture tells what are the components need and how to put them together to get complete design.
- The architecture of an embedded computing system includes both hardware and software elements.

**Hardware Elements of Architecture:**

- The hardware architecture of an embedded computing system is the more obvious manifestation of the architecture since you can touch it and feel it.
- It includes several elements, some of which may be less obvious than others.
CPU:
- An embedded computing system clearly contains a microprocessor and has much different architecture, within the architecture itself we can select between models that vary in clock speed, bus data width, integrated peripherals, and so on.
- The choice of the CPU is one of the most important, but it cannot be made without considering the software that will execute on the machine.

Bus:
- The choice of a bus is closely tied to that of a CPU, since the bus is an integral part of the microprocessor.
- But in applications that make intensive use of the bus due to I/O or other data traffic, the bus may become one of the limiting factors more than the CPU.

Memory:
- The most important characteristic of memory, total size, which depends on both the required data volume and the size of the program instructions.
- The ratio of ROM to RAM and selection of DRAM versus SRAM can have a significant influence on the cost of the system.
- The speed of the memory will play a large part in determining system performance.

Input and output devices:
- Many kinds of input and output devices are available for connecting the devices to the microprocessor.
- The input devices that vary based on the cost and kind of its job.
- The difficulty of using a particular device is it requires some amount of glue logic required to interface it.
- So based on the microprocessors they will select the input and output devices.

Software elements of architecture:
- Not only hardware, software also has architecture.
- Software is a collection of programs each and every program having different kind of architecture.
- A fundamental task in software architecture design is partitioning breaking the functionality into pieces in a way that makes it easy to implement, test, and modify.
- Different types of functionality often require different programming styles, and it have different procedure in the code.
- The code that must satisfy the deadline and to measure the performance of that code.

Hardware Design:
- Hardware design is having the high complexity, because in hardware design parts different kinds of components with different kinds of functions.
- In hardware design many levels are involved in that board level is the most important because it contains different kinds of hardware components grouped.
- At the board level, the first step is to consider the evaluation board supplied by the microprocessor manufacturer.
- Evaluation boards are mainly used from microprocessor systems; which include the CPU, some memory, a serial link for downloading programs, and some minimal number of I/O devices.
- The evaluation board may be a complete solution for slight modifications.
- If the evaluation board is supplied by the microprocessor vendor, its design may be available from the vendor.
- The other major task is the choice of memory and peripheral components.
- In the case of I/O devices, there are two alternatives for each device:
  - Selecting a component from a catalog
• Simple peripheral logic can be implemented in **programmable logic devices (PLDs)**, while more complex units can be built from **field-programmable gate arrays (FPGAs)**.

**The PC as a Platform:**

- Personal computers are often used as platforms for embedded computing.
- A PChascometoapplytoavarietyofmachines, including IBM-compatibles, Macs and others.
- A typical PC includes several major hardware components
  - The CPU provides basic computational facilities.
  - RAM is used for program storage.
  - ROM holds the boot program.
  - A DMA controller provides DMA capabilities.
  - Timers are used by the operating system for a variety of purposes.
  - A high-speed bus, connected to the CPU bus through a bridge, allows fast devices to communicate efficiently with the rest of the system.
  - A low-speed bus provides an inexpensive way to connect simpler devices and may be necessary for backward compatibility as well.

![Diagram of PC hardware architecture](image)

**Fig 2.5 Hardware architecture of a typical PC**

- A PC also provides a standard software platform that provides interfaces to the underlying hardware as well as more advanced services.
- At the bottom of the software platform structure in most PCs is a minimal set of software in ROM.
- This software is designed to load the complete operating system from some other device, and it may also provide low-level hardware interfaces.
- In the IBM-compatible PC, the low-level software is known as the **basic input/output system (BIOS)**.
- The BIOS provides low-level hardware drivers, control of executing processes, user interfaces, and so on.
- The PC software environment is so rich, developing embedded code for a PC target is much easier than when a host must be connected to a CPU in a development target.
- However, if the software is delivered directly on a standard version of the operating system, the resulting software package will require significant amounts of RAM as well as occupy a large disk image.
- Both the IBM-compatible PC and the Mac provide a combination of hardware and software that allows devices to provide their own configuration information.
- On the IBM-compatible PC, this is known as the Plug-and-Play standard developed by Microsoft.
- These standards make it possible to plug in a device and have it work directly, without hardware or software intervention from the user.
EC6703- EMBEDDED AND REAL TIME SYSTEMS

UNIT-II

• A single-chip PC makes the development of certain types of embedded systems much easier, providing the rich software development of a PC with the low cost of a single-chip hardware platform.

Advantages:

➢ It is pre-designed hardware platform
➢ It has great many features
➢ A wide variety of I/O devices can be purchased for it
➢ It provides a rich programming environment

Disadvantages:

➢ It does not use custom hardware for PC based system
➢ It is large more power hungry and more expensive than custom hardware Platform.

PCI (Peripheral Component Interconnect):

• PCI is the dominant high performance system bus today.
• PCI uses high-speed data transmission techniques and efficient protocols to achieve high throughput.
• The original PCI standard allowed operation up to 33 MHz; at that rate, it could achieve a maximum transfer rate of 264 MB/s using 64-bit transfers.
• The revised PCI standard allows the bus to run up to 66 MHz, giving a maximum transfer rate of 524 MB/s with 64-bit wide transfers.
• PCI uses wide buses with many data and address bits along with multiple control bits.
• If the width of the bus increases it also increases the cost and more complicated.
• To avoid the above problem serial buses are used and it will provide high-speed transfers and it will reduce the cost.
• USB (Universal Serial Bus) and IEEE1394 are the two major high-speed serial buses.
• Both of these buses offer high transfer rates using simple connectors.
• They also allow devices to be chained together so that users don’t have problem on details of connection. It is used in PC circuit board.

Application Example:

System organization of the Intel StrongARM SA-1100 and SA-1111

![Diagram](Fig 2.5.1)

• The StrongARM SA-1100 provides a number of functions besides the ARM CPU:
• The chip contains two on-chip buses: a high-speed system bus and a lower-speed peripheral bus.
• The chip also uses two different clocks.
• A 3.686MHz clock is used to drive the CPU and high-speed peripherals, anda32.768kHzclockisaninputtothesystemcontrolmodule.
• The system control module contains the following peripheral devices:
  ➢ A real-time clock
  ➢ An operating system timer
The 32.768kHz clock’s frequency is chosen to be useful in timing real-time events.
- The slower clock is also used by the power management to provide continued operation of the manager at a lower clock rate and therefore lower power consumption.
- The SA-1111 is a companion chip that provides a suite of I/O functions.
- It connects to the SA-1100 through its system bus and provides several functions: a USB host controller; PS/2 ports for keyboards, mice, and so on; a PCMCIA interface; pulse-width modulation outputs; a serial port for digital audio; and an SSP serial port for telecom interfacing.

**DEVELOPMENT AND DEBUGGING:**

- Development process involved series of actions used to make a complete design of the system.
- It guides the developers how to design a system and what are the process involved in the tasks.
- First we have to consider how to build an effective programming for embedded system.
- Once the program is build next important task is testing, we then see how hosts and other techniques can be used for debugging embedded systems.

**DEVELOPMENT ENVIRONMENTS:**

- A typical embedded computing system has all the components in a small manner such as CPU, memory, I/O devices, and so forth.
- Development of embedded system have both hardware and software development. Software plays an important role in embedded system.
- Software development on a PC or workstation known as a host as illustrated in Fig 2.6. The hardware on which the code will finally run is known as the target.
- The host and target are frequently connected by a USB link, but a higher-speed link such as Ethernet can also be used.
- The target must include a small amount of software to talk to the host system.
- That software will take up some memory, interrupt vectors, and so on.
- The host should be able to do the following:
  - Load programs into the target.
EC6703- EMBEDDED AND REAL TIME SYSTEMS  UNIT-II  EMBEDDED COMPUTING PLATFORM DESIGN

➢ Start and stop program execution on the target, and
➢ Examine memory and CPU registers.

COMPILER:
• A compiler is a kind of software it translate one form of program into another form of program or language.
• In the embedded system design cross-compiler will be used.
• A cross-compiler is a compiler that runs on one type of machine but generates code for another.
• After compilation, the executable code is downloaded to the embedded system by a serial link.
• A PC or workstation offers a programming environment that is in many ways much friendlier than the typical embedded computing platform.
• But it has one problem such as the embedded code cannot be run as is on the host due its device configuration.
• The solution for the above problem is, a test-bench program can be built to help debug the embedded code.
• The testbench generates inputs to simulate the actions of the input devices; it may also take the output values and compare them against expected values.
• The test bench will slightly modify the embedded code to perform the debugging process but the changes will not produce any bugs.

DEBUGGING TECHNIQUES:
• Debugging is the process of checking the errors and correcting those errors.
• Software debugging can be done by compiling and executing the code on a PC or workstation.
• Debugging can be performed in two sides, one is the software side and another one is the hardware side for both sides many debugging tools are involved.

Types of software debugging tools:
➢ Serial port tool
➢ Break point tool
• The serial port is one of the most important debugging tools
• This tool will perform the debugging process from the initial state of embedded system design.
• The serial port can be used not only for development but also for diagnosing problems in the field.
• Breakpoint is the most important tool in which user to specify an address where the program’s execution is to break.
• Once the PC reaches that address, control is returned to the monitor program.
• From the monitor program the user can examine and/or modify CPU registers, after that the process can be continued.

Advantage:
• Implementation breakpoints does not require any external devices.

Types of hardware debugging tools:
• When software tools are insufficient to debug the system, hardware tools will be used.
• Hardware tools clearly mention what is happening in the system and when the system is running.
• As like software tools it has having the following tools.
➢ Microprocessor in-circuit emulator
➢ Logic analyzer
• The MICROPROCESSOR IN-CIRCUIT EMULATOR (ICE) is a specialized hardware tool that can help debug software in a working embedded system.
In-circuit emulator is a special version of the microprocessor that has internal registers; it will be read out when it is stopped.

The CPU provides as much debugging functionality without any memory utilization.

**Drawbacks:**
- In-circuit emulator is specific to a particular microprocessor.
- Several microprocessors used means maintaining a fleet of in-circuit emulator to match can be very expensive.

**LOGIC ANALYZER:**
- The logic analyzer can sample many different signals simultaneously (tens to hundreds) but can display only 0, 1, or changing values for each.
- The logic analyzer records the values on the signals into an internal memory and then displays the results on a display once the memory is full or the run is aborted.
- The logic analyzer can capture thousands or even millions of samples of data.

**Data modes of Logic analyzer:**
- State mode and Timing mode.
- **State mode** represents different ways of sampling the values.
- State mode uses the system’s own clock to control sampling.
- It samples each signal only once per clock cycle.
- State mode having less memory to store a given number of system clock.
- It is used for sequentially oriented problems.
- **Timing mode** represents different ways of sampling the values.
- Timing mode uses an internal clock and it is fast to take several samples per clock period in a typical system.
- Timing mode requires more memory to store a given number of system clock cycles and it is used for glitch oriented debugging problems.

![Fig 2.6.1 Architecture of a logic analyzer](image)

**Format of data:**
- Logic analyzers provide a number of formats for viewing data.
- One format is a timing diagram format.

**Drawback:**
- The logic analyzer does not provide access to the internal state of the components.
- But it gives a very good view of the externally visible signals that information can be used for both functional and timing debugging.

**Debugging Challenges:**
Logical errors in software can be hard to track down and it will create many problems in real-time code.

Real-time programs are required to finish their work within a certain amount of time.

Real-time programs run too long means it creates very unexpected behavior.

The exact reason for missing real-time deadlines depends on the detailed characteristics of the I/O devices.

Missing deadlines makes debugging process as difficult.

**PROGRAM DESIGN:**

- The process of programming embedded processors is the most important tasks.
- The creation of embedded programs is at the heart of embedded system design.
- Designing and implementing embedded program is different form of PC programs.
- High-level languages are used for embedded programs.
- For embedded programs mainly ‘C’ language is used.
- To create embedded software three different kinds of structure or components will be used such as follows:
  - State machine
  - Circular buffer
  - Queue

**STATE MACHINES:**

- Once the input is given to the system it will react based on the given input.
- That reaction of systems can be characterized state machine.
- The state machine describes the behavior of the system.
- The current state of the system characteristics is known as **finite-state machine**.
- The state machine style of programming is also an efficient implementation of such computations.
- Finite-state machines are usually first encountered in the context of hardware design.

**CIRCULAR BUFFER:**

- Circular buffer is a data structure that handles streaming data in an efficient way
- It contains two kinds of elements known as head and end.
- Head indicates starting position of the circular buffer.
- If head = null buffer is empty. Head position is incremented.
- End indicates the last position of the buffer.
- Buffer can have fixed size to store current data.
- Consider data stream having four values at time t, as shown in fig 2.7 (a).
- Circular buffers stores that values in the following format with fixed storage size as four, as shown in fig 2.7 (b).
- Now we need to enter new value 5 in the above circular buffer.
- Window slides throw out old values no longer needed and new values, the old value is thrown out and new value 5 is entered, as shown in fig 2.7 (c).
- To avoid constantly copying the data within the same location of buffer we will move the head of the buffer to the next position or in time.
- Now enter the new value 6 means the oldest value 2 will be thrown out and 6 will be entered, as shown in fig 2.7 (d).
- When the pointer reaches the end of the buffer it wrap around to the top.
Fig 2.7 A circular buffer for streaming data

**QUEUES:**
- Queue is also a data structure that will perform the process first in first out manner.
- Queues are also used in signal processing and event processing.
- Queues are mainly used at variable amounts of data may arrive.
- A queue is often referred to as an elastic buffer.
- Queues are designed in two ways: one is dynamically allocating memory and another way is array to hold all the data.

**MODELS OF PROGRAMS:**
- Programs are collections of instructions to execute a specific task.
- Models for programs are more general than source code.
- We cannot use the source code directly, because we have different types of source code such as assembly languages, C code, and so on.
- We must use a single model to describe all of them.
- Single model usage can be performed many useful analyses on the model more easily.
- Fundamental model of programs is the control/dataflow graph (CDFG).
- The CDFG has constructs that model both data operations and control operations.
- To understand the CDFG in clear format, first we must understand data descriptions.

**DATA FLOW GRAPHS:**
- A data flow graph is a model of a program with no conditionals.
- In a high-level programming language, a code segment with no condition having only one entry and exit point is known as a basic block.
- For example, basic block in C:

  \[
  W = a + b; \\
  X = a - c; \\
  Y = X + d; \\
  X = a + c; \\
  Z = Y + e;
  \]

- In the above code, X is having two assignments so we need to rewrite the code with single assignment.
- Because if any variable is having two assignments, means it contains only the latest assigned value.
- The modified format for above code is
- For example, basic block in Single assignment form:

  \[
  W = a + b; \\
  X_1 = a - c; \\
  Y = X_1 + d; \\
  X_2 = a + c; \\
  Z = Y + e;
  \]
The single-assignment form is more important because it clearly mention a unique location of variable in the code.

- In single assignment form the data flow graph, is acyclic

Types of nodes in data flow graph:

- Round nodes represent operators
- Square nodes represent operators

Advantages:

- Order of execution operation is mention.
- It reduces pipeline process
- It is used to determine feasible recordings of the operations.

Control/Data Flow Graphs:

- A CDFG uses a data flow graph as an element, adding constructs to describe control.
- In a basic CDFG, we have two types of nodes:
  - Decision nodes and
  - Data flow nodes.
- A data flow node encapsulates a complete data flow graph to represent a basic block.
- Decision node are used to describe all the types of control in a sequential program.
- Now we have one C code and CDFG code for C.

```c
for (i = 0; i < N; i++)
{
    loop_body();
}
```

is equivalent to

```c
i = 0;
while (i < N) {
    loop_body();
    i++;
}
```

if (cond1)
```
basic_block_1();
```
else
```
basic_block_2();
```

Fig 2.8 An extended data flow graph for our sample basic block.
Now compute the CDFG code for above C code, it follows

- In CDFG construction it has two kinds of nodes.
  - Rectangular nodes in the graph represent the basic blocks.
  - Diamond-shaped nodes represent the conditionals.

![CDFG Diagram](image)

This is the CDFG of the given C code.

- For a complete CDFG model, we can use a data flow graph to model each data flow node.
- The CDFG is a hierarchical representation of a data flow CDFG can be expanded to reveal a complete data flow graph.
- We can also build a CDFG for an assembly language program.
- ARM and many VLIW processors support predicated execution of instructions, which may be represented by special constructs in the CDFG.

**ASSEMBLY, LINKING, AND LOADING:**

- Assembly and linking are the last steps in the compilation process.
- They turn a list of instructions into an image of the program’s bits in memory.
- Fig 2.9 shows how to generate an executable program.
- It will perform the following sequence of process.
  - Compiler is converting high-level language code into machine code. But most compilers do not directly generate machine code. It will generate human-readable assembly language.
  - The assembler’s job is to translate symbolic assembly language statements into bit-level representations of instructions known as object code. Assembler has to translate labels into addresses.
Program may be built from many files, the final steps in determining the addresses of instructions and data are performed by the linker, which produces an executable binary file.

Loader will load the program into memory for execution.

**Fig 2.9 Program generation from compilation through loading.**

**Types of address:**
- Absolute addresses
- Relative addresses

- The starting address of the assembly language program has been specified by the programmer. The addresses in such a program are known as absolute addresses.
- In relative address the origin of the assembly language module is to be computed later. These are the basic functions of converting high level language into executable code.

**Assembler:**
- Assembler not only translate assembly code into object code, it also translate opcodes and format the bits in each instruction, and translate labels into addresses.
- Labels make the assembly process more complex.
- Label processing requires making two passes through the assembly source code as follows:
  - The first pass scans the code to determine the address of each label.
  - The second pass assembles the instructions using the label values computed in the first pass.
- In the first pass, the name of each symbol and its address is stored in a symbol table.
- The symbol table is built by scanning from the first instruction to the last.
- During scanning, the current location in memory is kept in a program location counter (PLC). The PLC is not used to execute the program, only to assign memory locations to labels.
- PLC and PC are same but PLC always makes exactly one pass through the program, whereas the program counter makes many passes over code in a loop.
- Thus, at the start of the first pass, the PLC is set to the program’s starting address and the assembler looks at the first line.
- After examining the line, the assembler updates the PLC to the next and looks at the next instruction.
- If the instruction begins with a label, a new entry is made in the symbol table, which includes the label name and its value. The value of the label is equal to the current value of the PLC.
- At the end of the first pass, the assembler rewinds to the beginning of the assembly language file to make the second pass.
- During the second pass, when a label name is found, the label is looked up in the symbol table and its value substituted into the appropriate place in the instruction.

```
add r0 , r1 , r2
```
Assembly code Symbol table

- Assemblers allow labels to be added to the symbol table without occupying space in the program memory. A typical name of this pseudo-op is EQU for equate.
- The ARM assembler supports one pseudo-op that is particular to the ARM instruction set. In other architectures, an address would be loaded into a register by reading it from a memory location.
- The assembler produces an object file that describes the instructions and data in binary format. A commonly used object file format, originally developed for Unix but now used in other environments as well, is known as COFF (common object file format).
- The object file must describe the instructions, data, and any addressing information and also usually carries along the symbol table for later use in debugging.
- To understand the details of turning relocatable code into executable code, we must understand the linking process described in the next section.

**Linking:**

- Many assembly language programs are written as several smaller pieces rather than as a single large file.
- The linker allows a program to be stitched together out of several smaller pieces.
- The linker operates on the object files created by the assembler and modifies the assembled code to make the necessary links between files.
  - Some labels will be both defined and used in the same file.
  - Other labels will be defined in a single file but used elsewhere.
  - Label is defined is known as an entry point and label used place is known as an external reference.
  - The main job of the loader is to resolve external references based on available entry points.
  - Even if the entire symbol table is not kept for later debugging purposes, it must at least pass the entry points.

**Phases of linker:**

- First phase
- Second phase

- In first phase, it determines the address of the start of each object file.
- In second phase, the loader merges all symbol tables from the object files into a single large table.
- Workstations and PCs provide dynamically linked libraries and some embedded computing environment also provides it.
- Dynamically linked libraries allow them to be linked in at the start of program execution.

**Advantages of Dynamically linked libraries:**

- It saves storage space.
- Programs can be easily updated.

---

**BASIC COMPILATION TECHNIQUES:**

- Implementing an embedded computing system often requires controlling the instruction sequences used to handle interrupts, placement of data and instructions in memory, and so on.
- It is useful to understand how a high-level language program is translated into instructions.
- Many applications are also performance sensitive, understanding how code is generated can help you meet your performance goals.

**Compilation = Translation + Optimization.**
The high-level language program is translated into the lower-level form of instructions; optimizations try to generate better instruction sequences.

Optimization techniques focus on more of the program to ensure that compilation decisions that appear to be good.

For one statement are not unnecessarily problematic for other parts of the program.

![Diagram showing the compilation process]

**Fig 2.10 The compilation process**

- Compilation begins with high-level language code such as C and generally produces assembly code.
- The high-level language program is parsed to break it into statements and expressions. A symbol table is generated, which includes all the named objects in the program.
- Simplifying arithmetic expressions is one example of a machine-independent optimization. All compilers do not perform this kind of optimizations.
- Compilers perform two kind of optimizations one is instruction-level and other machine-independent optimizations.

**Statement Translation:**

- A large amount of the code in a typical application consists of arithmetic and logical expressions.
- Converting these expressions are more complex tasks of compiler.
- To understand how to compile a single expression, let us consider the following example.

**Compiling an arithmetic expression:**

In the following arithmetic expression,

\[ a * b + 5 * (c - d) \]

- The variable is written in terms of program variables.
- In some machines we may be able to perform memory-to-memory arithmetic directly on the locations corresponding to those variables.
- In many machines, such as the ARM, we must first load the variables into registers.
- This requires choosing which registers receive not only the named variables but also intermediate results such as \((c - d)\).
- The code for the expression can be built by walking the data flow graph.
- The data flow graph for the expression appears is shown in fig 2.10.1.
- The temporary variables for the intermediate values and final result have been named \(w, x, y, \) and \(z\).
- To generate code, we walk from the tree’s root by traversing the nodes in post order.
- During the walk, we generate instructions to cover the operation at every node.
• The nodes are numbered in the order in which code is generated.
• Since every node in the data flow graph corresponds to an operation that is directly supported by the instruction set.

**ARM code follows:**

```
; operator 1 (+
ADR r4,a : get address for a
MOV r1,[r4] ; load a
ADR r4,b ;get address for b
MOV r2,[r4] ;load b
ADD r3,r1,r2 ; put w into r3
; operator 2 (-):
ADR r4,c ; get address for c
MOV r4,[r4] ; load c
ADR r4,d ;get address for d
MOV r5,[r4] ; load d
SUB r6,r4,r5 ; put x into r6
; operator 3 (*)
MUL r7,r6,#5 ; operator 3, puts y into r7
; operator 4 (+)
ADD r8,r7,r3 ; operator 4, puts z into r8
```

• Optimization is to reuse a register whose value is no longer needed.
• In the case of the intermediate values w, x, and y, we know that they cannot be used after the end of the expression.
• The final result z may in fact be used in a C assignment and the value reused later in the program.
  ➢ We have large programs with multiple expression we must allocate registers more carefully, because CPUs have a limited number of registers.
  ➢ Compiler also able to translate control structures, drawing a control flow graph based on the while form of the loop helps us understand how to translate it into instructions.
• C compilers can generate assembler source, which some compilers intersperse with the C code.
• Such code is a very good way to learn about both assembly language programming and compilation.
Procedures:
- Creation of procedures is the major problem in code generation.
- Generating code for procedures is relatively straightforward procedure definition, must handle the procedure call and return.
- In modern programming language the CPU’s subroutine call mechanism is usually not sufficient to directly support procedures.
- Procedure stack and procedure linkage are different kinds of functions performed on procedure.
- Procedure linkage mechanism provides a way for the program to pass parameters into the program and for the procedure to return a value.
- It also provides help in restoring the values of registers that the procedure has modified.
- All procedures in a given programming language use the same linkage mechanism. The mechanism can also be used to call handwritten assembly language routines from compiled code.
- Procedure stacks are typically built to grow down from high addresses. A stack pointer (sp) defines the end of the current frame, while a frame pointer (fp) defines the end of the last frame.
- The procedure can refer to an element in the frame by addressing relative to sp.
- When a new procedure is called, the sp and fp are modified to push another frame onto the stack.
- The ARM Procedure Call Standard (APCS) is a good illustration of a typical procedure linkage mechanism. Although the stack frames are in main memory, how registers are used will be explained below.
- r0 - r3 are used to pass parameters into the procedure. r0 is also used to hold the return value. If more than four parameters are required, they are put on the stack frame.
- r4 - r7 hold register variables.
- r11 is the frame pointer and r13 is the stack pointer.
- r10 holds the limiting address on stack size, which is used to check for stack overflows.
- Other registers have additional uses in the protocol.

Data Structures:
- Data structure is the way of organizing the data.
- The compiler must also translate references to data structures into references to raw memories.
- In general, this requires address computations. Some of these computations can be done at compile time while others must be done at run time.
- Some examples of data Structures are linked list, array, queue, structure, union and so on.
- Arrays are interesting because the address of an array element must in general be computed at run time.
arrays have three kinds such as
  ➢ one-dimensional array
  ➢ Two-dimensional arrays
  ➢ Multi-dimensional arrays

one-dimensional array:
  ➢ Consider one dimensional array which is having following format.
  ➢ a[i] it contains I number of values.
  ➢ The layout of the array in memory is shown in Fig 2.10.3. Fig 2.10.3 One-dimensional array
  ➢ The zer0th element is stored as the first element of the array, the first element directly below, and so on.
  ➢ We can create a pointer for the array. Pointer is a variable it contains the address of another variable.
  ➢ Array pointer points to the array head namely, a[0].
  ➢ If we call that pointer aptr for convenience, then we can rewrite the reading of a[i] as
  *(aptr + i)

two-dimensional arrays:
  ➢ Two-dimensional arrays are more challenging.
  ➢ There are multiple possible ways to lay out a two-dimensional array in memory, as shown in Fig 2.10.4.
  ➢ One form of memory layout for two-dimensional array is row major.
  ➢ In the row-major the inner variable of the array (j in a[i][j]) varies most quickly.
  ➢ Two-dimensional arrays also require more sophisticated addressing.
  ➢ First we must know the size of array. Fig 2.10.4 Two-dimensional array
  ➢ If the a[] array is of size N x M, then we can turn the two-dimensional array access into a one-dimensional array access. Thus, a[i][j] becomes a[i*M + j]
  ➢ Where the maximum value for j is M - 1.

program-level energy and power analysis and optimization:
  ➢ Power consumption is a particularly important design metric for battery-powered systems because the battery has a very limited lifetime.
  ➢ However, power consumption is increasingly important in systems that run off the power grid.
  ➢ Fast chips run hot and controlling power consumption is an important element of increasing reliability and reducing system cost.
  ➢ We must consume the energy required to perform necessary computations.
  ➢ For example, consider the following technique to reduce power consumption.
    ⚫ To replace the algorithms with other algorithm which contains less power consume.
    ⚫ Memory accesses are a major component of power consumption in many applications. By optimizing memory accesses we may be able to significantly reduce power.
  ➢ The first step in optimizing a program’s energy consumption is knowing how much energy the program consumes.
  ➢ It is possible to measure power consumption for an instruction or a small code fragment.
  ➢ Several factors contribute to the energy consumption of the program.
    ⚫ Energy consumption varies somewhat from instruction to instruction.
    ⚫ The sequence of instructions has some influence.
    ⚫ The opcode and the locations of the operands also matter.
    ⚫ Memory system also reduces the power consumption.
Caches are an important factor in energy consumption.

PROBLEMS ON CACHE MEMORY:
- Cache hit saves a costly main memory access, and on the other hand, the cache itself is relatively power hungry because it is built from SRAM, not DRAM.
- If the cache is too small, the program runs slowly and the system consumes a lot of power due to the high cost of main memory accesses.
- If the cache is too large, the power consumption is high without a corresponding payoff in performance. At intermediate values, the execution time and power consumption are both good.

STEPS TO IMPROVE ENERGY CONSUMPTION:
- Try to use registers efficiently.
- Analyze cache behavior to find major cache conflicts.
- Make use of page mode accesses in the memory system whenever possible.
- Metha presents some additional observations about energy optimization as follows:
  - Moderate loop unrolling eliminates some loop control overhead. However, when the loop is unrolled too much, power increases due to the lower hit rates of straight-line code.
  - Software pipelining reduces pipeline stalls, thereby reducing the average energy per instruction.
  - Eliminating recursive procedure calls where possible saves power by getting rid of function call overhead. Tail recursion can often be eliminated; some compilers do this automatically.

ANALYSIS AND OPTIMIZATION OF PROGRAM SIZE:
- The memory footprint of a program is determined by the size of its data and instructions. Both must be considered to minimize program size.
- Data provide an excellent opportunity for minimizing size because the data are most highly dependent on programming style.
- Because inefficient programs often keep several copies of data, identifying and eliminating duplications can lead to significant memory savings usually with little performance penalty.
- A very low-level technique for minimizing data is to reuse values.
- Data buffers can often be reused at several different points in the program. It is also used to reduce program size.
- Minimizing the size of the instruction text of a program requires a mix of high-level program transformations and careful instruction selection.
- Encapsulating functions in subroutines can reduce program size when done carefully.
- Architectures that have variable-size instruction lengths are particularly good candidates for careful coding to minimize program size.
- When reducing the number of instructions in a program, one important technique is the proper use of subroutines.
- Proper instruction selection may reduce code size; this is particularly true in CPUs that use variable-length instructions.
- Some microprocessor architectures support dense instruction sets, specially designed instruction sets that use shorter instruction formats to encode the instructions.
- The ARMThumb instruction set and the MIPS-16 instruction set for the MIPS architecture are two examples of this type of instruction set.
- Microprocessor that supports the dense instruction set also supports the normal instruction set.
- Special compilation modes produce the program in terms of the dense instruction set.
**PROGRAM VALIDATION AND TESTING:**

- Complex systems need testing to ensure that they work as they are intended. But bugs can be subtle, particularly in embedded systems, where specialized hardware and real-time responsiveness make programming more challenging.
  - There are many available techniques for software testing that can help us generate a comprehensive set of tests to ensure that our system works properly.
  - Breaking the testing problem into subproblems and analyzing each subproblem, we can identify testing methods that provide reasonable amounts of testing while keeping the testing time within reasonable bounds.
- The **two major types of testing strategies**:
  - **Black-box** methods generate tests without looking at the internal structure of the program.
  - **Clear-box** (also known as white-box) methods generate tests based on the program structure.

**CLEAR-BOX TESTING:**

- The control/data flow graph extracted from a program’s source code is an important tool in developing clear-box tests for the program.
- To adequately test the program, we must exercise both its control and data operations.
- In order to execute and evaluate these tests, we must be able to control variables in the program is most important.
- The following three things in a test:
  - Provide the program with inputs that exercise the test we are interested in.
  - Execute the program to perform the test.
  - Examine the outputs to determine whether the test was successful.
- In the testing process, the next task is to determine the set of tests to be performed. We need to perform many different types of tests to be confident that we have identified a large fraction of the existing bugs.
- If we test the program using one criterion, that criterion ignores other aspects of the program.
- The most fundamental concept in clear-box testing is the path of execution through a program. To test the program by forcing the program to execute along chosen paths.
- The test path will be executed by giving it inputs that will cause it to take the appropriate branches.
- Execution of a path exercises both the control and data aspects of the program.
- To execute every complete path in an arbitrary program is not possible. For that relatively select a small number of paths should be able to cover most practical programs.
- Graph theory helps us get a quantitative handle on the different paths required. In an undirected graph, we can form any path through the graph from combinations of basis paths.

**Graph**

![Graph](image)

**Incidence matrix**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Basis set**

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Fig 2.11** The matrix representation of a graph and its basis set
• Incidence matrix contains each row and column represents a node; a 1 is entered for each node pair connected by an edge.
• Next method is cyclomatic complexity, which allows us to measure the control complexity of a program. It is an upper bound on the size of the basis set.
• Then the cyclomatic complexity is given by
  \[ M = e - n + 2p \]  
  \[ \ldots \ldots (1) \]
  Where
  e is the number of edges in the flow graph,
  n the number of nodes, and
  p the number of components in the graph

**TYPES OF TESTING STRATEGY:**
  - Branch testing
  - Domain testing
  - Data flow testing

**BRANCH TESTING:**
• A simple condition testing strategy is known as branch testing.
• This strategy requires the true and false branches of a conditional and every simple condition in the conditional’s expression to be tested at least once.

**Example:**
  ```c
  if ((a == b) || (c >= d))
  {
      ...
      printf(“OK”);
  }
  ```

**DOMAIN TESTING:**
• Another more sophisticated strategy for testing conditionals is known as domain testing.
• Domain testing concentrates on linear inequalities.
• The program must use the test condition is \( j \leq i + 1 \).
• We test the inequality with three test points:
  - Two on the boundary of the valid region
  - Third test on the outside the region but between the \( i \) values of the other two points.

**DATA FLOW TESTING:**
• The important testing strategy is known as data flow testing, makes use of def-use analysis (short for definition-use analysis).
• It selects paths that have some relationship to the program’s function.
• The terms def and use come from compilers, which use def-use analysis for optimization
• A variable’s value is defined when an assignment is made to the variable; it is used when it appears on the right side of an assignment.

**BLACK-BOX TESTING:**
• Black-box tests are generated without knowledge of the code being tested.
• Black-box testing can be performed in two ways:
  - Black-box test alone
  - Black-box testing with clear box tests
• When usedalone,black-box testshave a low probability of finding all the bugs in a program.
• When black-box tests used in conjunction with clear-box tests they help provide a well-rounded test set.
• One important technique is to take tests directly from the specification for the code under design.
• The specification should state which outputs are expected for certain inputs.
• We can’t test every possible input combination, but some rules of thumb help us select reasonable sets of inputs.
• When an input can range across a set of values, it is a very good idea to test at the ends of the range.

RANDOM TESTS:
• Random tests form one category of black-box test.
• Random values are generated with a given distribution.
• The expected values are computed independently of the system, and then the test inputs are applied.

REGRESSION TESTS:
• Regression tests form an extremely important category of tests.
• When tests are created during earlier stages in the system design, these tests should be saved to apply to the later versions of the system.
• The system specification changed, the new system should be able to pass old tests.
• Regression tests simply exercise the code in different ways than would be done for the current version of the code and therefore possibly exercise different bugs.
• In digital signal processing systems, we used to perform numerical analysis and signal processing algorithms are frequently implemented with limited-range arithmetic to save hardware costs.
• Aggressive data sets can be generated to stress the numerical accuracy of the system.
CONSUMER ELECTRONICS ARCHITECTURE:

- Not all devices have all features, depending on the way the device is to be used, but most devices select features from a common menu.
- Similarly, there is no single platform for consumer electronics devices, but the architectures in use are organized around some common themes.
- This convergence is possible because these devices implement a few basic types of functions in various combinations: multimedia, communications, and data storage and management.
- The style of multimedia or communications may vary, and different devices may use different formats, but this causes variations in hardware and software components within the basic architectural templates.

FUNCTIONAL REQUIREMENTS:

- Consumer electronics devices provide several types of services in different combinations:

Multimedia:
- The media may be audio, still images, or video (which includes both motion pictures and audio).
- These multimedia objects are generally stored in compressed form and must be uncompressed to be played (audio playback, video viewing, etc.).
- A large and growing number of standards have been developed for multimedia compression: MP3, Dolby Digital(TM), etc. for audio; JPEG for still images; MPEG-2, MPEG-4, H.264, etc. for video.

Data storage and management:
- Because people want to select what multimedia objects they save or play, data storage goes hand-in-hand with multimedia capture and display.
- Many devices provide PC-compatible file systems so that data can be shared more easily.

Communications:
- Communications may be relatively simple, such as a USB interface to a host computer.
- The communications link may also be more sophisticated, such as an Ethernet port or a cellular telephone link.

NON-FUNCTIONAL REQUIREMENTS:

- Many devices are battery-operated means, which that they must operate under strict energy budgets.
- A typical battery for a portable device provides only about 75mW, which must support not only the processors and digital electronics but also the display, radio, etc.
- Consumer electronics must also be very inexpensive.
- These devices must also provide very high performance; sophisticated networking and multimedia compression require huge amounts of computation.

USE CASES:

![Use case diagram](image)

**Fig 2.12 Use case for playing multimedia**

- Fig 2.12 shows a use case for selecting and playing a multimedia object (an audio clip, a picture, etc.).
- Selecting an object makes use of both the user interface and the file system.
- Playing also makes use of the file system as well as the decoding subsystem and I/O subsystem.
Fig 2.12.1 Use case of synchronizing with a host system

- Fig 2.12.1 shows a use case for connecting to a client.
- The connection may be either over a local connection like USB or over the Internet.
- While some operations may be performed locally on the client device, most of the work is done on the host system while the connection is established.

Platforms and Operating Systems:

- Given these types of usage scenarios, we can deduce a few basic characteristics of the underlying architecture of these devices.
- The storage system provides bulk, permanent storage as shown in Fig 2.12.2.
- The network interface may provide a simple USB connection or a full-blown Internet connection.
- Multiprocessor architectures are common in many consumer multimedia devices.
- Fig 2.12.2 shows, two-processor architecture; if more computation is required, more DSPs and CPUs may be added.
- The RISC CPU runs the operating system, runs the user interface, maintains the file system, etc.
- The DSP performs signal processing.
- The DSP may be programmable in some systems; in other cases, it may be one or more hardwired accelerators.

Fig 2.12.2 Hardware architecture of a generic consumer electronics device

Operating System:

- The operating system that runs on the CPU must maintain processes and the file system.
- Processes are necessary to provide concurrency—for example, the user wants to be able to push a button while the device is playing back audio.
- Depending on the complexity of the device, the operating system may not need to create tasks dynamically.
- If all tasks can be created using initialization code, the operating system can be made smaller and simpler.

FILE SYSTEMS:

DOS File system:

- DOS file allocation table (FAT) file systems refer to the file system developed by Microsoft for early versions of the DOS operating system.
- FAT can be implemented on flash storage devices as well as magnetic disks, wear-leveling algorithms for flash memory can be implemented without disturbing the basic operation of the file system.
• FAT can be implemented in a relatively small amount of code.

Flash memory:
• Many consumer electronics devices use flash memory for mass storage.
• Flash memory is a type of semiconductor memory that, unlike DRAM or SRAM, provides permanent storage.
• Values are stored in the flash memory cell as electric charge using a specialized capacitor that can store the charge for years.
• The flash memory cell does not require an external power supply to maintain its value.
• Furthermore, the memory can be written electrically and, unlike previous generations of electrically-erasable semiconductor memory, can be written using standard power supply voltages and so does not need to be disconnected during programming.

Flash file Systems:
• Flash memory has one important limitation that must be taken into account.
• Writing a flash memory cell causes mechanical stress that eventually wears out the cell.
• Today’s flash memories can reliably be written a million times but at some point they will fail.
• While a million write cycles may sound like enough to ensure that the memory will never wear out, creating a single file may require many write operations, particularly to the part of the memory that stores the directory information.
• A wear-leveling flash file system manages the use of flash memory locations to equalize wear while maintaining compatibility with existing file systems.
• A simple model of a standard file system has two layers: the bottom layer handles physical reads and writes on the storage device; the top layer provides a logical view of the file system.
• A flash file system imposes an intermediate layer that allows the logical-to-physical mapping of files to be changed.
• This layer keeps track of how frequently different sections of the flash memory have been written and allocates data to equalize wear.
• It may also move the location of the directory structure while the file system is operating.
• Because the directory system receives the most wear, keeping it in one place may cause part of the memory to wear out before the rest, unnecessarily reducing the useful life of the memory device.
• Several flash file systems have been developed, such as Yet Another Flash Filing System (YAFFS).

PLATFORM-LEVEL PERFORMANCE ANALYSIS:
• Bus-based systems add another layer of complication to performance analysis.
• Platform-level performance involves much more than the CPU.
• We often focus on the CPU because it processes instructions, but any part of the system can affect total system performance.
• More precisely, the CPU provides an upper bound on performance, but any other part of the system can slow down the CPU.
• Merely counting instruction execution times is not enough.
Consider the simple system of Fig 2.13. We want to move data from memory to the CPU to process it.

To get the data from memory to the CPU we must:

- Read from the memory;
- Transfer over the bus to the cache; and
- Transfer from the cache to the CPU

The time required to transfer from the cache to the CPU is included in the instruction execution time, but the other two times are not.

**Bandwidth as performance:**

- Bandwidth: The rate at which we can move data.
- Ultimately, if we are interested in real-time performance, measured in seconds.
- But often the simplest way to measure performance is in units of clock cycles.
- However, different parts of the system will run at different clock rates.

**Bus Bandwidth:**

- When we are transferring large blocks of data, consider the bandwidth provided by only one system component, the bus.
- Consider an image of $320 \times 240$ pixels, with each pixel composed of 3 bytes of data.
- This gives a grand total of 230, 400 bytes of data.
- If these images are video frames, we want to check if we can push one frame through the system within the 1/30 sec that we have to process a frame before the next one arrives.
- Let us assume that we can transfer one byte of data every microsecond, which implies a bus speed of 1MHz.
- In this case, we would require 230, 400 bytes =0.23sec to transfer one frame.
- That is more than the 0.033sec allotted to the data transfer.
- We would have to increase the transfer rate by 7× to satisfy our performance requirement.
- **We can increase bandwidth in two ways:** We can increase the clock rate of the bus or we can increase the amount of data transferred per clock cycle.
- For example, if we increased the bus to carry four bytes or 32 bits per transfer, we would reduce the transfer time to 0.058 sec.
- If we could also increase the bus clock rate to 2 MHz, then we would reduce the transfer time to 0.029 sec, which is within our time budget for the transfer.

**Bus bandwidth characteristics:**

- How do we know how long it takes to transfer one unit of data? To determine that, we have to look at the data sheet for the bus.
- A bus transfer generally takes more than one bus cycle.
- Burst transfers, which move to contiguous locations, may be more efficient per byte.
- We also need to know the width of the bus, how many bytes per transfer.
- Finally, we need to know the bus clock period, which in general will be different from the CPU clock period.

**Bus bandwidth formulas:**

- Let’s assume the bus clock period $P$ and the bus width $W$.
- Write the basic formulas in units of bus cycles $T$, then convert those bus cycle counts to real time $t$ using the bus clock period $P$:

  \[ t = TP \quad \text{.... (1)} \]

- As shown in Fig 2.13.1, a basic bus transfer transfers a $W$-wide set of bytes.
Fig 2.13.1 Times and data volumes in a basic bus transfer

Fig 2.13.2 Times and data volumes in a burst bus transfer.

- The data transfer itself takes D clock cycles. (Ideally, D = 1, but a memory that introduces wait states is one example of a transfer that could require D > 1 cycles.)
- Addresses, handshaking, and other activities constitute overhead that may occur before (O1) or after (O2) the data.
- For simplicity, we will lump the overhead into O = O1 + O2.
- This gives a total transfer time in clock cycles of:
  \[ T = (W + W) + (W + W) \]  
  \[ T = (W + W) + (W + W) \]  

- As shown in Fig 2.13.2, a burst transaction performs B transfers of W bytes each.
- Each of those transfers will require D clock cycles.
- The bus also introduces O cycles of overhead per burst.
  \[ T = (W + W) + (W + W) \]  
  \[ T = (W + W) + (W + W) \]  

Component Bandwidth:

- The width of a memory determines the number of bits we can read from the memory in one cycle.
- That is a form of data bandwidth.
- We can change the types of memory components we use to change the memory bandwidth; we may also be able to change the format of our data to accommodate the memory components.

Memory aspect ratios:

- A single memory chip is not solely specified by the number of bits it can hold.

Fig 2.13.3 Memory aspect ratios

- As shown in Fig 2.13.3, memories of the same size can have different aspect ratios.
- For example, a 64-MB memory that is 1-bit-wide will present 64 million addresses of 1-bit data.
- The same size memory in a 4-bit-wide format will have 16 distinct addresses and an 8-bit-wide memory will have 8 million distinct addresses.
• Memory chips do not come in extremely wide aspect ratios but we can build wider memories by using several memories in parallel.
• Rather than buy memory chips individually, we may buy memory as SIMMs or DIMMs.
• These memories are wide but generally only come in fairly standard widths.
• Which aspect ratio is preferable for the overall memory system depends in part on the format of the data that we want to store in the memory and the speed with which it must be accessed, giving rise to bandwidth analysis.

Memory access times and Bandwidth:
• We also have to consider the time required to read or write a memory.
• Once again, we refer to the component data sheets to find these values.
• Access times depend quite a bit on the type of memory chip used.
• Page modes operate similarly to burst modes in buses.
• If the memory is not synchronous, we can still refer the times between events back to the bus clock cycle to determine the number of clock cycles required for an access.
• The basic form of the equation for memory transfer time is that of Equ(3), where O is determined by the page mode overhead and D is the time between successive transfers.
• However, the situation is slightly more complex if the data types do not fit naturally into the width of the memory.
• Let’s say that we want to store color video pixels in our memory.
• A standard pixel is three 8-bit color values (red, green, blue, for example).
• A 24-bit-wide memory would allow us to read or write an entire pixel value in one access.
• An 8-bit-wide memory, in contrast, would require three accesses for the pixel.
• If we have a 32-bit-wide memory, we have two main choices: We could waste one byte of each transfer or use that byte to store unrelated data, or we could pack the pixels.
• In the latter case, the first read would get all of the first pixel and one byte of the second pixel; the second transfer would get the last two bytes of the second pixel and the first two bytes of the third pixel; and so forth.
• The total number of accesses required to read E data elements of w bits each out of a memory of width W is:

$$\phi = (\lceil \frac{w}{8} \rceil) \cdot \phi + 1$$  \hspace{1cm} \ldots (4)$$

PROGRAM-LEVEL PERFORMANCE ANALYSIS:
• Because embedded systems must perform functions in real time, we often need to know how fast a program runs.
• The techniques we use to analyze program execution time are also helpful in analyzing properties such as power consumption.
• We also examine how to optimize programs to improve their execution times; of course, optimization relies on analysis.
• It is important to keep in mind that CPU performance is not judged in the same way as program performance.
• As illustrated in Fig 2.14, the CPU pipeline and cache act as windows into our program.
• In order to understand the total execution time of our program, we must look at execution paths, which in general are far longer than the pipeline and cache windows.
• The pipeline and cache influence execution time, but execution time is a global property of the program.
Fig 2.14 Execution time is a global property of a program

- While we might hope that the execution time of programs could be precisely determined, this is in fact difficult to do in practice:
  - **The execution time of a program often varies with the input data values** because those values select different execution paths in the program. For example, loops may be executed a varying number of times, and different branches may execute blocks of varying complexity.
  - **The cache has a major effect on program performance**, and once again, the cache’s behavior depends in part on the data values input to the program.
  - **Execution times may vary even at the instruction level.** Floating-point operations are the most sensitive to data values, but the normal integer execution pipeline can also introduce data-dependent variations. In general, the execution time of an instruction in a pipeline depends not only on that instruction but also the instructions around it in the pipeline.

Measuring execution speed:

- **Some microprocessor manufacturers supply simulators for their CPUs:**
  - The simulator runs on a workstation or PC, takes as input an executable for the microprocessor along with input data, and simulates the execution of that program.
  - Simulation is clearly slower than executing the program on the actual microprocessor, but it also provides much greater visibility during execution.
  - Be careful—some microprocessor performance simulators are not 100% accurate, and simulation of I/O-intensive code may be difficult.
- **A timer connected to the microprocessor bus** can be used to measure performance of executing sections of code.
  - The code to be measured would reset and start the timer at its start and stop the timer at the end of execution.
  - The length of the program that can be measured is limited by the accuracy of the timer.
- **A logic analyzer can be connected to the microprocessor bus** to measure the start and stop times of a code segment.
  - This technique relies on the code being able to produce identifiable events on the bus to identify the start and stop of execution.
  - The length of code that can be measured is limited by the size of the logic analyzer’s buffer.
- **We are interested in the following three different types of performance measures on programs:**
  - **Average-case execution time:** This is the typical execution time we would expect for typical data. Clearly, the first challenge is defining typical inputs.
  - **Worst-case execution time:** The longest time that the program can spend on any input sequence is clearly important for systems that must meet deadlines. In some cases, the input set that causes the worst-case execution time is obvious, but in many cases it is not.
  - **Best-case execution time:** This measure can be important in multirate real time systems.

**ELEMENTS OF PROGRAM PERFORMANCE:**

- Program execution time can be seen as
  
  \[
  \text{Execution time} = \text{program path} + \text{instruction timing}
  \]
The path is the sequence of instructions executed by the program.
- The instruction timing is determined based on the sequence of instructions traced by the program path, which takes into account data dependencies, pipeline behavior, and caching.
- Luckily, these two problems can be solved relatively independently.
- Although we can trace the execution path of a program through its high-level language specification, it is hard to get accurate estimates of total execution time from a high-level language program.
- This is because there is not a direct correspondence between program statements and instructions.
- The number of memory locations and variables must be estimated, and results may be either saved for reuse or recomputed on the fly, among other effects.
- These problems become more challenging as the compiler puts more and more effort into optimizing the program.
- For example, if a C program contains a loop with a large, fixed iteration bound or if one branch of a conditional is much longer than another, we can get at least a rough idea that these are more time-consuming segments of the program.

**Example: data-dependent program paths.**
- In the code below, the assignment in the loop is performed exactly N times.
  
  ```c
  for (i = 0; i < N; i++)
  a[i] = b[i]*c[i];
  ```
- However, we can’t forget the code executed to set up the loop and to test the iteration variable.

**Instruction Timing:**
- Once we know the execution path of the program, we have to measure the execution time of the instructions executed along that path.
- The simplest estimate is to assume that every instruction takes the same number of clock cycles, which means we need only count the instructions and multiply by the per-instruction execution time to obtain the program’s total execution time.
- However, even ignoring cache effects, this technique is simplistic for the reasons summarized below.

- **Not all instructions take the same amount of time.** Although RISC architectures tend to provide uniform instruction execution times in order to keep the CPU’s pipeline full, even many RISC architectures take different amounts of time to execute certain instructions. Multiple load-store instructions are examples of longer-executing instructions in the ARM architecture. Floating point instructions show especially wide variations in execution time, while basic multiply and add operations are fast, some transcendental functions can take thousands of cycles to execute.

- **Execution times of instructions are not independent.** The execution time of one instruction depends on the instructions around it. For example, many CPUs use register bypassing to speed up instruction sequences when the result of one instruction is used in the next instruction. As a result, the execution time of an instruction may depend on whether its destination register is used as a source for the next operation (or vice versa).

- **The execution time of an instruction may depend on operand values.** This is clearly true of floating-point instructions in which a different number of iterations may be required to calculate the result.
- We can handle the first two problems more easily than the third.
- We can look up instruction execution time in a table; the table will be indexed by opcode and possibly by other parameter values such as the registers used.
- To handle interdependent execution times, we can add columns to the table to consider the effects of nearby instructions.
Since these effects are generally limited by the size of the CPU pipeline, we know that we need to consider a relatively small window of instructions to handle such effects.

Handling variations due to operand values is difficult to do without actually executing the program using a variety of data values, given the large number of factors that can affect value-dependent instruction timing.

Luckily, these effects are often small.

**Caching Effects:**
- Thus far we have not considered the effect of the cache.
- Because the access time for main memory can be 10-100 times larger than the cache access time, caching can have huge effects on instruction execution time by changing both the instruction and data access times.
- Caching performance inherently depends on the program’s execution path since the cache’s contents depend on the history of accesses.

**MEASUREMENT-DRIVEN PERFORMANCE ANALYSIS:**
- This approach is appealing, but it does have some drawbacks.
- First, in order to cause the program to execute its worst-case execution path, we have to provide the proper inputs to it.
- Furthermore, in order to measure the program’s performance on a particular type of CPU, we need the CPU or its simulator.
- Despite these drawbacks, measurement is the most commonly used way to determine the execution time of embedded software.
- Worst-case execution time analysis algorithms have been used successfully in some areas, such as flight control software, but many system design projects determine the execution time of their programs by measurement.

**Program Traces:**
- We refer to the record of the execution path of a program as a program trace (or more succinctly, a trace).
- Traces can be valuable for other purposes, such as analyzing the cache behavior of the program.

**Measurement Issues:**
- Perhaps the biggest problem in measuring program performance is figuring out a useful set of inputs to provide to the program.
- This problem has two aspects.
- First, we have to determine the actual input values.
- We may be able to use benchmark data sets or data captured from a running system to help us generate typical values.
- For simple programs, we may be able to analyze the algorithm to determine the inputs that cause the worst-case execution time.
- The software testing methods of can help us generate some test values and determine how thoroughly we have exercised the program.
- The other problem with input data is the **software scaffolding** that we may need to feed data into the program and get data out.
- When we are designing a large system, it may be difficult to extract out part of the software and test it independently of the other parts of the system.
- We may need to add new testing modules to the system software to help us introduce testing values and to observe testing outputs.
- We can measure program performance either directly on the hardware or by using a simulator.
- Each method has its advantages and disadvantages.
Profiling:
- Profiling is a simple method for analyzing software performance.
- A profiler does not measure execution time, instead it counts the number of times that procedure or basic blocks in the program are executed.
- There are two major ways to profile a program: we can modify the executable program by adding instructions that increment a location every time the program passes that point in the program; or we can sample the program counter during execution and keep track of the distribution of PC values.
- Profiling adds relatively little overhead to the program and it gives us some information about where the program spends most of its time.

Physical Performance Measurement:
- Physical measurement requires some sort of hardware instrumentation.
- The most direct method of measuring the performance of a program would be to watch the program counter’s value: start a timer when the PC reaches the program’s start, stop the timer when it reaches the program’s end.
- However, it is possible in many cases to modify the program so that it starts a timer at the beginning of execution and stops the timer at the end.
- While this doesn’t give us direct information about the program trace, it does give us execution time.
- If we have several timers available, we can use them to measure the execution time of different parts of the program.
- A logic analyzer or an oscilloscope can be used to watch for signals that mark various points in the execution of the program.
- However, because logic analyzers have a limited amount of memory, this approach doesn’t work well for programs with extremely long execution times.

Hardware Traces:
- Some CPUs have hardware facilities for automatically generating trace information.
- For example, the Pentium family microprocessors generate a special bus cycle, a branch trace message, that shows the source and/or destination address of a branch
- If we record only traces, we can reconstruct the instructions executed within the basic blocks while greatly reducing the amount of memory required to hold the trace.

Simulation-Based Performance Measurement:
- The alternative to physical measurement of execution time is simulation.
- A CPU simulator is a program that takes as input a memory image for a CPU and performs the operations on that memory image that the actual CPU would perform, leaving the results in the modified memory image.
- For purposes of performance analysis, the most important type of CPU simulator is the cycle-accurate simulator, which performs a sufficiently detailed simulation of the processor’s internals so that it can determine the exact number of clock cycles required for execution.
- Cycle-accurate simulators are slower than the processor itself, but a variety of techniques can be used to make them surprisingly fast, running only hundreds of times slower than the hardware itself.
- A cycle-accurate simulator has a complete model of the processor, including the cache.
- It can therefore provide valuable information about why the program runs too slowly.
- A simulator that functionally simulates instructions but does not provide timing information is known as an instruction-level simulator.
Loop Optimizations:
- Loops are important targets for optimization because programs with loops tend to spend a lot of time executing those loops.
- There are three important techniques in optimizing loops: code motion, induction variable elimination, and strength reduction.
- Code motion lets us move unnecessary code out of a loop.
- If a computation’s result does not depend on operations performed in the loop body, then we can safely move it out of the loop.
- A simple example of code motion is also common.

Consider the following loop:
```
for (i = 0; i < N*M; i++) {
    z[i] = a[i] + b[i];
}
```
- The code motion opportunity becomes more obvious when we draw the loop’s CDFG as shown in Fig 2.15.

**Fig 2.15 Code motion in a loop**

- The loop bound computation is performed on every iteration during the loop test, even though the result never changes.
- We can avoid $N \times M - 1$ unnecessary executions of this statement by moving it before the loop, as shown in the fig 2.15.
- An induction variable is a variable whose value is derived from the loop iteration variable’s value.
- The compiler often introduces induction variables to help it implement the loop.
- Properly transformed, we may be able to eliminate some variables and apply strength reduction to others.
- A nested loop is a good example of the use of induction variables.

Here is a simple nested loop:
```
for (i = 0; i < N; i++)
    for (j = 0; j < M; j++)
        z[i][j] = b[i][j];
```
- The compiler uses induction variables to help it address the arrays.
- Let us rewrite the loop in C using induction variables and pointers.
In the above code, zptr and bptr are pointers to the heads of the z and b arrays and zbinduct is the shared induction variable.

However, we do not need to compute zbinduct afresh each time.

Since we are stepping through the arrays sequentially, we can simply add the update value to the induction variable:

\[
\begin{align*}
z\text{binduct} &= 0; \\
&\text{for } (i = 0; i < N; i++) \\
&\quad \text{for } (j = 0; j < M; j++) \\
&\quad \quad * (z\text{ptr} + z\text{binduct}) = * (b\text{ptr} + z\text{binduct}); \\
&\quad z\text{binduct}++; \\
&\end{align*}
\]

This is a form of strength reduction since we have eliminated the multiplication from the induction variable computation.

**Strength reduction** helps us reduce the cost of a loop iteration.

Consider the following assignment:

\[
y = x \times 2;
\]

In integer arithmetic, we can use a left shift rather than a multiplication by 2.

If the shift is faster than the multiply, we probably want to perform the substitution.

This optimization can often be used with induction variables because loops are often indexed with simple expressions.

Strength reduction can often be performed with simple substitution rules since there are relatively few interactions between the possible substitutions.

**Cache Optimizations:**

- A loop nest is a set of loops, one inside the other.
- Loop nests occur when we process arrays.
- A large body of techniques has been developed for optimizing loop nests.
- Rewriting a loop nest changes the order in which array elements are accessed.
- This can expose new parallelism opportunities that can be exploited by later stages of the compiler, and it can also improve cache performance.

**Performance Optimization Strategies:**

- Performance analysis and measurement will give you a baseline for the execution time of the program.
- Knowing the overall execution time tells you how much it needs to be improved.
- Knowing the execution time of various pieces of the program helps you to identify the right locations for changes to the program.
- You may be able to redesign your algorithm to improve efficiency.
- Examining asymptotic performance is often a good guide to efficiency.
- Doing fewer operations is usually the key to performance.
- In a few cases, however, brute force may provide a better implementation.
• A seemingly simple high-level language statement may in fact hide a very long sequence of operations that slows down the algorithm.
• Using dynamically allocated memory is one example, since managing the heap takes time but is hidden from the programmer.
• For example, a sophisticated algorithm that uses dynamic storage may be slower in practice than an algorithm that performs more operations on statically allocated memory.
• Finally, you can look at the implementation of the program itself.

**Program Implementation:**

➢ **Try to use registers efficiently.** Group accesses to a value together so that the value can be brought into a register and kept there.
➢ **Make use of page mode accesses in the memory system whenever possible.** Page mode reads and writes eliminate one step in the memory access. You can increase use of page mode by rearranging your variables so that more can be referenced contiguously.
➢ **Analyze cache behavior to find major cache conflicts.** Restructure the code to eliminate as many of these as you can as follows:
  • For **instruction conflicts**, if the offending code segment is small, try to rewrite the segment to make it as small as possible so that it better fits into the cache. Writing in assembly language may be necessary. For conflicts across larger spans of code, try moving the instructions or padding with NOPs.
  • For **scalar data conflicts**, move the data values to different locations to reduce conflicts.
  • For **array data conflicts**, consider either moving the arrays or changing your array access patterns to reduce conflicts.
UNIT III

PROCESSES AND OPERATING SYSTEMS

Introduction – Multiple tasks and multiple processes – Multirate systems- Preemptive real-time operating systems- Priority based scheduling- Interprocess communication mechanisms – Evaluating operating system performance- power optimization strategies for processes – Example Real time operating systems-POSIX-Windows CE.

INTRODUCTION:

- Simple applications can be programmed on a microprocessor by writing a single piece of code.
- But for complex application we cannot execute with simple program of code.
- There are two fundamental abstractions that allow us to build complex applications on microprocessors:
  - Process: It defines the state of an executing program
  - Operating System (OS): It provides the mechanism for switching execution between the processes.
- These two mechanisms together to build applications with more complex functionality and much greater flexibility to satisfy timing requirements.
- In real-time operating systems (RTOSs), which are OSs that provide facilities for satisfying real-time requirements.
- A RTOS allocates resources using algorithms that take real time into account.
- General-purpose OSs, in contrast, generally allocate resources using other criteria like fairness.
- Trying to allocate the CPU equally to all processes without regard to time can easily cause processes to miss their deadlines.

MULTIPLE TASKS AND MULTIPLE PROCESSES:

- Most of the embedded systems are too complex hence the programming and system also complex.
- To reduce the complexity we break the system into multiple tasks.

Tasks and Processes:

Tasks:
- Task is nothing but different parts of functionality in a single system.
- For example, when designing a telephone answering machine, we can define recording a phone call and operating the user’s control panel as distinct tasks.
- The different functions (i.e.) answering a call, recording a call and operating are the various applications.
- Thus the various applications in a system and the each application is a task.

Processes:
- A process is a single execution of a program.
- If we run the same program two different times, we have created two different processes.
- Each process has its own state that includes not only its registers but all of its memory.
- In some OSs, the memory management unit is used to keep each process in a separate address space.
- In others, particularly lightweight RTOSs, the processes run in the same address space.

Threads:
- Processes that share the same address space are often called threads.

Example: Consider how we would build a stand-alone compression unit based on the compression algorithm can be implemented.
- In fig 3.1, this device is connected to serial ports on both ends.
- The input to the box is an uncompressed stream of bytes.
- The box emits a compressed string of bits on the output serial line, based on a predefined compression table.
- Such a box may be used, for example, to compress data being sent to a modem.

Variable data rates:
- The program’s need to receive and send data at different rates, for example, the program may emit 2 bits for the first byte and then 7 bits for the second byte, will obviously find itself reflected in the structure of the code.
- It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets.
- But beyond the need to create a clean data structure that simplifies the control structure of the code, we must also ensure that we process the inputs and outputs at the proper rates.
- For example, if we spend too much time in packaging and emitting output characters, we may drop an input character.
Asynchronous input:
- A control panel on a machine provides an example of a different type of rate control problem, the asynchronous input.
- The control panel of the compression box may, for example, include a compression mode button that disables or enables compression, so that the input text is passed through unchanged when compression is disabled.
- Keeping up with the input and output data while checking on the button can introduce some very complex control code into the program.
- Sampling the button’s state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently and duplicating a data value can cause the machine to incorrectly compress data.
- One solution is to introduce a counter into the main compression loop, so that a subroutine to check the input button is called once every n times the compression loop is executed.
- But this solution does not work when either the compression loop or the button-handling routine has highly variable execution times—if the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost.
- We need to be able to keep track of these two different tasks separately, applying different timing requirements to each.
- This is the sort of control that processes allow.
- The above two examples illustrate how requirements on timing and execution rate can create major problems in programming.
- When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution become very complex very quickly.

MULTIRATE SYSTEMS:
- The system which are embedded with more than one application are called multirate system.
- Multirate embedded computing systems are very common, including automobile engines, printers, and cell phones.
- Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled.
- In all these systems, certain operations must be executed periodically, and each operation is executed at its own rate.

Example: Automotive engine control
- The simplest automotive engine controllers, such as the ignition controller for a basic motor- cycle engine, perform only one task, timing the firing of the spark plug, which takes the place of a mechanical distributor.
- The spark plug must be fired at a certain point in the combustion cycle, but to obtain better performance, the phase relationship between the piston’s movement and the spark should change as a function of engine speed.
- Using a microcontroller that senses the engine crankshaft position allows the spark timing to vary with engine speed.
- Firing the spark plug is a periodic process.
Hence the need of microprocessor is much greater in the complex system design.

Automobile engine with control algorithm must meet strict requirements on both emissions and fuel economy.

On the other hand, the engines must still satisfy customers not only in terms of performance but also in terms of ease of starting in extreme cold and heat, low maintenance, and so on.

Automobile engine controllers use additional sensors, including the gas pedal position and an oxygen sensor used to control emissions. They also use a multimode control scheme.

For example, one mode may be used for engine warm-up, another for cruise, and yet another for climbing steep hills, and so forth.

The larger number of sensors and modes increases the number of discrete tasks that must be performed.

The engine controller takes a variety of inputs that determine the state of the engine.

It then controls two basic engine parameters:
- Spark plug firing.
- Fuel/air mixture.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Time to move full range (ms)</th>
<th>Update period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine spark timing</td>
<td>300</td>
<td>2</td>
</tr>
<tr>
<td>Throttle</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>Airflow</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>Battery voltage</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>Fuel flow</td>
<td>250</td>
<td>10</td>
</tr>
<tr>
<td>Recycled exhaust gas</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Set of status switches</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>Air temperature</td>
<td>Seconds</td>
<td>500</td>
</tr>
<tr>
<td>Barometric pressure</td>
<td>Seconds</td>
<td>1000</td>
</tr>
<tr>
<td>Spark/dwell</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Fuel adjustments</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>Carburetor adjustments</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Mode actuators</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 3.1 Different tasks performed by the engine controller unit

**Timing Requirements on Processes:**

Processes can have several different types of timing requirements imposed on them by the application.

A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid.

There are two important timing requirements on processes:
- Initiation time and
- Deadline

**Initiation Time:**

The initiation time is the time at which the process goes from waiting to the ready state.

An aperiodic process is by definition initiated by an event, such as external data arriving or data computed by another process.

The initiation time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself.

For a periodic process it is defined initiated by event and periodically executed process.

**Deadline:**

A deadline specifies when a computation must be finished.

The deadline for an aperiodic process is generally measured from the initiation time, because that is the only reasonable time reference.

The deadline for a periodic process may in general occur at some time other than the end of the period.
The period of a process is the time between successive executions. Then the process rate is defined by the inverse of its period.

Jitter:
- Jitter of a task, which is the allowable variation in the completion of a task.
- For example, in the playback of multimedia data to avoid audio gaps or jerky images; in the control of machines to ensure that the control signal is applied at the right time.
- The timing constraints between processes may be constrained when the processes pass data between each other.

Communication among processes that run at different rates cannot be represented by data dependencies because there is no one-to-one relationship between data coming out of the source process and going into the destination process.
Fig 3.2.3 illustrates the communication required among three elements of an MPEG audio/video decoder.

- Data come into the decoder in the system format, which multiplexes audio and video data.
- The system decoder process demultiplexes the audio and video data and distributes it to the appropriate processes.

**Multirate communication** is necessarily one way—for example, the system process writes data to the video process, but a separate communication mechanism must be provided for communication from the video process back to the system process.

**CPU usage Metrics:**

- The CPU utilization is given by

  \[
  \text{CPU Utilization} = \frac{\text{CPU Time Used}}{\text{CPU Time Available}}
  \]

  This ratio **ranges between 0 and 1**, with 1 meaning that all of the available CPU time is being used for system purposes.

- The utilization is often expressed as a **percentage**.

**Process State and Scheduling:**

- The first job of the OS is to determine the process that runs next.
- The work of choosing the order of running processes is known as scheduling.
- There are three basic scheduling states
  1. Waiting
  2. Ready
  3. Executing

  Upon these, there is at most one process executing on the CPU at any time.
- If any process that could execute is in the **ready state** then the OS chooses among the ready processes to select the next executing process.

**Fig 3.2.4 Scheduling states of a process**

- Fig 3.2.4 shows the possible transitions between states available to a process.
- A process goes into the **waiting state** when it needs data that it has not yet received or when it has finished all its work for the current period.
- A process goes into the ready state when it receives its required state.
- Finally, a process goes to its **executing state** only when it has all its data, is ready to run, and the scheduler selects the process as the next process to run.

**Scheduling Policy:**

- A scheduling policy defines **how processes are selected for promotion from the ready state to the running state**.
- Every multitasking OS implements some type of scheduling policy.
- Choosing the right scheduling policy not only ensures that the system will meet all its timing requirements, but it also has a profound influence on the CPU horsepower required to implement the system’s functionality.

- Utilization is one of the key metrics in evaluating a scheduling policy.

**Scheduling Overhead:**

- The execution time required to choose the next execution process, which is incurred in addition to any context switching overhead.
- The final decision on a scheduling policy must take into account both theoretical utilization and practical scheduling overhead.

**Running Periodic Processes:**

- Let’s assume that process as a subroutine; we will call them p1(), p2(), etc.
- Here is a very simple program that runs our process subroutines repeatedly:
First Step: While loop

```c
while (TRUE) {
    p1();
    p2();
}
```

- This program has several problems.
- First, it does not control the rate at which the processes execute—the loop runs as quickly as possible, starting a new iteration as soon as the previous iteration has finished.
- Second, all the processes run at the same rate.

A timed loop:
- Before worrying about multiple rates, let’s first make the processes run at a controlled rate.
- A timer is a much more reliable way to control execution of the loop and to generate periodic interrupts.
- Let’s assume for the moment that the pall( ) function is called by the timer’s interrupt handler.
- Then this code will execute each process once after a timer interrupt:

```c
void pall( ) {
    p1();
    p2();
}
```

- But what happens when a process runs too long? The timer’s interrupt will cause the CPU’s interrupt system to mask its interrupts, so the interrupt will not occur until after the pall( ) routine returns.
- As a result, the next iteration will start late.

Multiple timers:
- Our next problem is to execute different processes at different rates.
- If we have several timers, we can set each timer to a different rate.
- We could then use a function to collect all the processes that run at that rate:

```c
void pA( ) {
    /* processes that run at rate A*/
    p1();
    p3();
}
void pB( ) {
    /* processes that run at rate B */
    p2();
    p4();
    p5();
}
```

- This works, but it does require multiple timers, and we may not have enough timers to support all the rates required by a system.

Timer plus counters:
- An alternative is to use counters to divide the counter rate.
- If, for example, process p2( ) must run at 1/3 the rate of p1( ), then we can use this code:

```c
static int p2count = 0; /* use this to remember count across timer interrupts */
void pall( ) {
    p1();
    if (p2count >= 2) { /* execute p2() and reset count */
        p2();
        p2count = 0;
    } else p2count++; /* just update count in this case */
}
```

- However, when the rates aren’t related by a simple ratio, the counting process becomes more complex and more likely to contain bugs.
- If the tasks we executed in timer_handler( ) but ran past the period, the timer would interrupt again and stop execution of the previous iteration.
- The interrupted task may be left in an inconsistent state. So we cannot put the tasks into the timer handler.
1. A RTOS executes processes based upon timing constraints provided by the system designer.
2. The most reliable way to meet timing constraints accurately is to build a **preemptive OS** and to use **priorities** to control what process runs at any given time.

**Preemption:**
- Preemption is an alternative to the C function to control execution of processes.
- To make the effect of preemption we need to the process as something more than a function call in the program and also create new routines that allow us to jump from one subroutine to another at any point in the program.
- That, together with the timer, will allow us to move between functions whenever necessary based upon the system’s timing constraints.

![Fig 3.3 Sequence diagram for preemptive execution](image_url)

**Kernel:**
- We want to share the CPU across two processes.
- The kernel is the part of the OS that determines what process is running.
- The kernel is activated periodically by the timer.
- The length of the timer period is known as the **time quantum** because it is the smallest increment in which we can control CPU activity.
- On the next timer interrupt, the kernel may pick the same process or another process to run.
- Before, we used the timer to control loop iterations, with one loop iteration including the execution of several complete processes.
- Here, the time quantum is in general smaller than the execution time of any of the processes.

**Context Switching:**
- The timer interrupt causes control to change from currently executing process to the kernel.
- The assembly language can be used to save and restore registers.
- We can similarly use assembly language to restore registers from any process we want.
- The **set of registers that define a process** are known as **context** and **switching from one process register to another** is known as **context switching**.
- The data structure that holds the state of process is known as the **record**

**Process Priorities:**
- If we assign each task a numerical priority, then the kernel can simply look at the processes and their priorities, see which ones actually want to execute (some may be waiting for data or for some event), and select the highest priority process that is ready to run.
- This mechanism is both flexible and fast.

**Process and Context:**
- FreeRTOS.org kernel as an example; in particular, we will use version 4.7.0 for the ARM7 AVR32 platform.
- A process is known in FreeRTOS.org as a task.
- Let as assume that the set of tasks is in steady state: Everything has been initialized, the OS is running, and we are ready for a timer interrupt.
- Fig 3.3.1 shows the application tasks, the hardware timer, and all the functions in the kernel that are involved in the context switch:
  - `vPreemptiveTick()` is called when the **timer ticks**.
  - `SIG_OUTPUTCOMPARE1A` responds to the **timer interrupt** and uses `portSAVE_CONTEXT()` to swap out the current task context.
  - `vTaskIncrementTick()` updates the **time** and `vTaskSwitchContext()` chooses a new task.
  - `portRESTORE_CONTEXT()` swaps in the new context.
Fig 3.3.1 Sequence diagram for a FreeRTOS.org context switch

Here is the code for vPreemptiveTick() in the file portISR.c:

```c
void vPreemptiveTick( void )
{
    /* Save the context of the interrupted task. */
    portSAVE_CONTEXT( );
    /* Increment the tick count, this may wake a task */
    vTaskIncrementTick( );
    /* Find the highest priority task that is ready to run. */
    vTaskSwitchContext( );
    /* End the interrupt in the AIC */
    AT91C_BASE_AIC->AIC_EOICR = AT91C_BASE_PITC->PITC_PIVR; 
    /* Restore the context of the new task */
    portRESTORE_CONTEXT( );
}
```

Processes and Object-Oriented Design:
- We need to design systems with processes as components.

UML active objects:
- UML often refers to processes as active objects, that is, objects that have independent threads of control.
- The class that defines an active object is known as an active class.

Fig 3.3.2 An active class in UML

Fig 3.3.3 A collaboration diagram with active and normal objects

- Fig 3.3.3 shows a simple collaboration diagram in which an object is used as an interface between two processes: p1 uses the w object to manipulate its data before the data is sent to the master process.
PRIORITY - BASED SCHEDULING:
- The operating system's fundamental job is to allocate resources in the computing system among programs that request them.
- Naturally, the CPU is the scarcest resource, so scheduling the CPU is the operating system's most important job.
- To determine an algorithm by which to assign priorities to processes.
- After assigning priorities, the OS takes care of the rest by choosing the highest-priority ready process.
- There are two major ways to assign priorities:
  - Static priorities that do not change during execution and
  - Dynamic priorities that do change during execution.
- Depending on the static and dynamic way of assigning priority there are two methods to schedule the process. They are
  - Rate-monotonic scheduling (RMS)
  - Earliest-Deadline-First scheduling (EDF)

ROUND-ROBIN SCHEDULING:
- A common scheduling algorithm in general-purpose operating systems is round robin.
- All the processes are kept on a list and scheduled one after the other.
- This is generally combined with preemption so that one process does not grab all the CPU time.
- Round-robin scheduling provides a form of fairness in that all processes get a chance to execute.
- However, it does not guarantee the completion time of any task; as the number of processes increases, the response time of all the processes increases.
- Real-time systems, in contrast, require their notion of fairness to include timeliness and satisfaction of deadlines.

PROCESS PRIORITIES:
- A common way to choose the next executing process in an RTOS is based on process priorities.
- Each process is assigned a priority, an integer-valued number.
- The next process to be chosen to execute is the process in the set of ready processes that has the highest-valued priority.

PRIORITY-DRIVEN SCHEDULING:
For this example, we will adopt the following simple rules:
- Each process has a fixed priority that does not vary during the course of execution.
- The ready process with the highest priority (with 1 as the highest priority of all) is selected for execution.
- A process continues execution until it completes or it is preempted by a higher-priority process.

Let’s define a simple system with three processes as seen below.

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>20</td>
</tr>
</tbody>
</table>

We assume that P2 is ready to run when the system is started, P1’s data arrive at time 15, and P3’s data arrive at time 18.

When the system begins execution, P2 is the only ready process, so it is selected for execution.
At time 15, P1 becomes ready; it preempts P2 and begins execution since it has a higher priority.
Since P1 is the highest-priority process in the system, it is guaranteed to execute until it finishes.
P3’s data arrive at time 18, but it cannot preempt P1.
Even when P1 finishes, P3 is not allowed to run.
P2 is still ready and has higher priority than P3.
Only after both P1 and P2 finish can P3 execute.
**RATE-MONOTONIC SCHEDULING:**

- Rate-monotonic scheduling (RMS) was one of the first scheduling policies developed for real-time systems and is still very widely used.
- RMS is a **static scheduling policy** because it assigns fixed priorities to processes.
- To schedule the process using RMS we need to analyze the process known as **rate-monotonic analysis (RMA).**
- This theory, as summarized below, uses a relatively simple model of the system.
  - All processes run periodically on a single CPU.
  - Context switching time is ignored.
  - There are no data dependencies between processes.
  - The execution time for a process is constant.
  - All deadlines are at the ends of their periods.
  - The highest-priority ready process is always selected for execution.
- Priorities are assigned by rank order of period, with the process with the shortest period being assigned the highest priority.
- This fixed-priority scheduling policy is the optimum assignment static priorities to the process, and also it provides the highest CPU utilization.
- Example: A set of process and their characteristics given below

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>12</td>
</tr>
</tbody>
</table>

- For above processes apply the principles of RMA and assign the priorities.
- Then P1 has the highest priority, P2 has the middle priority and P3 has the lowest priority.
- After assigning priority we need to construct a timeline equal in length to the LCM of the process periods, which is 12 in this case.

![Timeline of Processes](image)

**Fig 3.4**

- All three periods start at time zero.
- P1’s data arrive first.
- Since P1 is the highest-priority process, it can start to execute immediately.
- After one time unit, P1 finishes and goes out of the ready state until the start of its next period.
- At time 1, P2 starts executing as the highest-priority ready process.
- At time 3, P2 finishes and P3 starts executing. P1’s next iteration starts at time 4, at which point it interrupts P3.
- P3 gets one more time unit of execution between the second iterations of P1 and P2, but P3 does not get to finish until after the third iteration of P1.
- In the above case the period of execution is within the timeline.
- In the some other case, if the period of execution is beyond the time line then the scheduling is by optimal analysis called critical-instant analysis.

**Response time:**
- Response time of a process as the time at which the process finishes.

**Critical instant:**
- It is defined as the instant during execution at which the task has the largest response time.
- The proof that the critical instant for any process P, under the RMA model, occurs when it is ready and all higher-priority processes are also ready.
- We can use critical-instant analysis to determine whether there is any feasible schedule for the system.
- The first step in the analysis is to imply that priorities should be assigned in order of periods.
- Let the periods and computation times of two processes P1 and P2 be \( T_1 \) and \( T_2 \) and \( C_1 \), with \( C_1 < T_2 \).
We can generalize the result to show the total CPU requirements for the two processes in two cases.

In the first case, let P1 have the higher priority.

In the worst case we then execute P2 once during its period and as many iterations of P1 as fit in the same interval.

Since there are $$\left\lfloor \frac{t_2}{t_1} \right\rfloor$$ iterations of P1 during a single period of P2, the required constraint on CPU time, ignoring context switching overhead, is

$$\left\lfloor \frac{t_2}{t_1} \right\rfloor \phi + \phi \leq t_2$$

In the worst case P2 has highest priority

$$\phi + \phi \leq t_2$$

This is the relationship of the period of execution and total timeline.

According to CPU utilization for a set of n tasks is given by,

$$\phi = \sum_{i=1}^{n} \phi_i$$

When there are m tasks with fixed priorities, the maximum processor utilization is

$$\phi = \left(2^{1/m} - 1\right)$$

**SHARED RESOURCES:**

A process may need to do more than read and write values memory.

For example, it may need to communicate with an I/O device.

And it may use shared memory locations to communicate with other processes.

When dealing with shared resource special care must be taken.

**Race condition:**

Consider the case in which an I/O device has a flag that must be tested and modified by a process.

Problems can arise when other processes may also want to access the device.

If combinations of events from the two tasks operate on the device in the wrong order, we may create a critical timing race or race condition that causes erroneous operation.

For example:

1. Task 1 reads the flag location and sees that it is 0.
2. Task 2 reads the flag location and sees that it is 0.
3. Task 1 sets the flag location to 1 and writes data to the I/O device's data register.
4. Task 2 also sets the flag to 1 and writes its own data to the device data register, overwriting the data from task 1.

In this case, both devices thought they were able to write to the device, but the task 1's write was never completed because it was overridden by task 2.

**Critical Sections:**

A critical section of code, also called a critical region, is code that needs to be treated indivisibly.

Once the section of code starts executing, it must not be interrupted.

To ensure this, interrupts are typically disabled before the critical codes is executed and enabled when the critical code is finished.

**Semaphore:**

The semaphore is used to guard a resource.

We start a critical section by calling a semaphore function that does not return until the resource is available.

When we are done with the resource we use another semaphore function to release it.

The semaphore names are, by tradition, P() to gain access to the protected resource and V() to release it.

/* some non protected operations here */

P(); /* wait for semaphore */

/* do protected work here */,

V(); /* release semaphore */

**Test-and-set:**

To implement P() and V(), the microprocessor bus must support an atomic read/write operation, which is available on a number of microprocessors.

These types of instructions first read a location and then set it to a specified value, returning the result of the test.

If the location was already set, then the additional set has no effect but the instruction returns a false result.
If the location was not set, the instruction returns true and the location is in fact set.

- The bus supports this as an atomic operation that cannot be interrupted.
- The test-and-set allows us to implement semaphores.
- The P() operation uses a test and-set to repeatedly test a location that holds a lock on the memory block.
- The P() operation does not exit until the lock is available; once it is available, the test-and-set automatically sets the lock.
- Once past the P() operation, the process can work on the protected memory block.
- The V() operation resets the lock, allowing other processes access to the region by using the P() function.

Critical sections and timing:
- It poses some problems for real-time systems.
- Because the interrupt system is shut off during the critical section, the timer cannot interrupt and other processes cannot start to execute.
- The kernel may also have its own critical sections that keep interrupts from being serviced and other processes from executing.

Priority inversion:
- Shared resources cause a new and subtle scheduling problem: a low-priority process blocks execution of a higher-priority process by keeping hold of its resource, a phenomenon known as priority inversion.
- A system with three processes: P1 has the highest priority, P3 has the lowest priority, and P2 has a priority in between that of P1 and P3.
- P1 and P3 both use the same shared resource.
- Processes become ready in this order:
  1. P3 becomes ready and enters its critical region, reserving the shared resource.
  2. P2 becomes ready and preempts P3.
  3. P1 becomes ready. It will preempt P2 and start to run but only until it reaches its critical section for the shared resource. At that point, it will stop executing.
- For P1 to continue, P2 must completely finish, allowing P3 to resume and finish its critical section.
- Only when P3 is finished with its critical section can P1 resume.

Priority inheritance:
- The most common method for dealing with priority inversion is priority inheritance: promote the priority of any process when it requests a resource from the operating system.
- The priority of the process temporarily becomes higher than that of any other process that may use the resource.
- This ensures that the process will continue executing once it has the resource so that it can finish its work with the resource, return it to the operating system, and allow other processes to use it.
- Once the process is finished with the resource, its priority is demoted to its normal value.

Earliest-Deadline-First Scheduling (EDF):
- EDF is a dynamic priority scheme, it changes process priorities during execution based on initiation times.
- As a result, it can achieve higher CPU utilizations than RMS.
- It is also very simple and assigns priorities in order of deadline.
- The highest-priority process is the one whose deadline is nearest in time and the lowest-priority process is the one whose deadline is farthest away.
- Clearly, priorities must be recalculated at every completion of a process.
- The final step of the OS during the scheduling procedure is the same as for RMS and the highest-priority ready process is chosen for execution.
- Example: EDF scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Execution time</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

- The LCM is 60.
- According to the above system, P1 has the highest priority, P2 has the middle priority and P3 has the lowest priority.
- There is one time slot left at t = 59, giving a CPU utilization of 59/60.
- Hence EDF achieved nearly 100% utilization of CPU.
- Then the deadline table is written as,
<table>
<thead>
<tr>
<th>Time</th>
<th>Running process</th>
<th>Deadlines</th>
<th>Time</th>
<th>Running process</th>
<th>Deadlines</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P₁</td>
<td></td>
<td>30</td>
<td>P₁</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>P₂</td>
<td>31</td>
<td>P₃</td>
<td>P₂</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>P₁ P₁</td>
<td>32</td>
<td>P₃</td>
<td>P₁ P₁</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>P₃ P₂</td>
<td>33</td>
<td>P₁</td>
<td>P₂</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>P₁ P₁</td>
<td>34</td>
<td>P₂</td>
<td>P₁ P₁</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>P₂ P₁</td>
<td>35</td>
<td>P₃</td>
<td>P₁ P₂</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>P₁</td>
<td>36</td>
<td>P₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>P₃ P₂</td>
<td>37</td>
<td>P₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>P₃ P₁</td>
<td>38</td>
<td>P₃</td>
<td>P₁ P₂</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>P₁ P₁</td>
<td>39</td>
<td>P₁</td>
<td>P₂ P₃</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>P₂</td>
<td>40</td>
<td>P₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>P₃ P₁ P₂</td>
<td>41</td>
<td>P₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>P₃</td>
<td>42</td>
<td>P₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>P₁</td>
<td>43</td>
<td>P₁ P₃ P₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>P₂ P₁ P₃</td>
<td>44</td>
<td>P₃</td>
<td>P₁ P₃ P₃</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>P₁ P₂</td>
<td>45</td>
<td>P₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>P₂</td>
<td>46</td>
<td>P₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>P₃ P₁</td>
<td>47</td>
<td>P₃</td>
<td>P₂ P₃</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>P₂</td>
<td>48</td>
<td>P₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>P₁ P₂ P₃</td>
<td>49</td>
<td>P₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>P₂ P₁</td>
<td>50</td>
<td>P₂</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>P₁</td>
<td>51</td>
<td>P₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>P₃</td>
<td>52</td>
<td>P₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>P₃ P₁ P₂</td>
<td>53</td>
<td>P₃</td>
<td>P₁ P₃ P₃</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>P₁ P₁</td>
<td>54</td>
<td>P₂</td>
<td>P₁ P₃ P₃</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>P₂</td>
<td>55</td>
<td>P₁</td>
<td>P₂ P₃</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>P₃ P₁</td>
<td>56</td>
<td>P₂</td>
<td>P₂ P₃</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>P₃ P₂</td>
<td>57</td>
<td>P₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>P₁</td>
<td>58</td>
<td>P₃</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>P₂ P₁ P₃</td>
<td>59</td>
<td>idle</td>
<td>P₁ P₂ P₃</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S.No</th>
<th>RMS</th>
<th>EDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Less utilization of CPU</td>
<td>More utilization of CPU</td>
</tr>
<tr>
<td>2</td>
<td>It is a static priority</td>
<td>It is a dynamic priority</td>
</tr>
<tr>
<td>3</td>
<td>It can diagnose the overload</td>
<td>It cannot</td>
</tr>
<tr>
<td>4</td>
<td>It is very easier to ensure all deadline</td>
<td>Much difficult compared to RMS</td>
</tr>
<tr>
<td>5</td>
<td>While the CPU utilization is less, this gives less problem to complete the deadline</td>
<td>It is more problematic</td>
</tr>
</tbody>
</table>

**INTERPROCESS COMMUNICATION MECHANISMS:**
- Processes often need to communicate with each other.
- Interprocess communication mechanisms are provided by the operating system as part of the process abstraction.
- In general, a process can send a communication in one of two ways: blocking or non blocking.

**Blocking Communication:**
- The process goes into the waiting state until it receives a response is called blocking communication.

**Non Blocking communication:**
- It allows the process to continue execution after sending the communication.
- Both types of communication are useful depending on the application requirement.
- There are two major styles of interprocess communication:
  - Shared memory and
  - Message passing

**SHARED MEMORY COMMUNICATION:**
- Fig 3.5, illustrates how shared memory communication works in a bus-based system.
- Two components, such as a CPU and an I/O device, communicate through a shared memory location.
- The software on the CPU has been designed to know the address of the shared location; the shared location has also been loaded into the proper register of the I/O device.
- If, as in the fig 3.5, the CPU wants to send data to the device, it writes to the shared location.
- The I/O device then reads the data from that location.
• The read and write operations are standard and can be encapsulated in a procedural interface.

Fig 3.5 Shared memory communication implemented on a bus.

• Fig 3.5.1 shows the text compressor uses the CPU to compress incoming text, which is then sent on a serial line by a UART.

Fig 3.5.1 Elastic buffers as shared memory

• The input data arrive at a constant rate and are easy to manage.
• But because the output data are consumed at a variable rate, these data require an elastic buffer.
• The CPU and output UART share a memory area—the CPU writes compressed characters into the buffer and the UART removes them as necessary to fill the serial line.
• Because the number of bits in the buffer changes constantly, the compression and transmission processes need additional size information.
• In this case, coordination is simple—the CPU writes at one end of the buffer and the UART reads at the other end.
• The only challenge is to make sure that the UART does not overrun the buffer.

MESSAGE PASSING:
• Message passing communication complements the shared memory model.

Fig 3.5.2 Message passing communication

• As shown in fig 3.5.2, each communicating entity has its own message send/receive unit.
• The message is not stored on the communications link, but rather at the senders/ receivers at the end points.
• For example, a home control system has one microcontroller per household device—lamp, thermostat, faucet, appliance, and so on.
• The devices must communicate relatively infrequently; furthermore, their physical separation is large enough that we would not think about the shared memory location, hence we go for passing communication.

Queue:
• A queue is a common form of message passing.
• The queue uses a FIFO discipline and holds records that represent messages.
• The FreeRTOS.org system provides a set of queue functions.
• It allows queues to be created and deleted so that the system may have as many queues as necessary.
• A queue is described by the data type xQueueHandle and created using xQueueCreate:
  xQueueHandle q1;
  q1 = xQueueCreate(MAX_SIZE,sizeof(msg_record)); /* maximum number of records in queue size of each record */
  if (q1 == 0) /* error */
  
  ……..
• The queue is created using the vQueueDelete( ) function.
A message is put into the queue using xQueueSend( ) and received using xQueueReceive( ):

```c
xQueueSend(q1,( void *)&msg,(portTickType),0); /*queue to send, final parameter controls
   timeout */
if(xQueueReceive(q2,&(in_msg),0); /*queue message received timeout */
```

The final parameter in these functions determines how long the queue waits to finish.

In the case of a send, the queue may have to wait for something to leave the queue to make room.

In the case of the receive, the queue may have to wait for data to arrive.

**Signals:**
- Another form of interprocess communication commonly used in Unix is the signal.
- A signal is simple because it does not pass data beyond the existence of the signal itself.
- A signal is analogous to an interrupt, but it is entirely a software creation.
- A signal is generated by a process and transmitted to another process by the operating system.
- Fig 3.5.2 shows the use of a signal in UML.
- The sigbehavior( ) behavior of the class is responsible for throwing the signal, as indicated by <<send>>.
- The signal object is indicated by the <<signal>> stereotype.

**MAILBOXES:**
- The mailbox is a simple mechanism for asynchronous communication.
- Some architectures define mailbox registers.
- These mailboxes have a fixed number of bits and can be used for small messages.
- We can also implement a mailbox using P( ) and V( ) using main memory for the mailbox storage.
- A very simple version of a mailbox, one that holds only one message at a time.
- In order for the mailbox to be most useful, we want it to contain two items: the message itself and a mail ready flag.
- The flag is true when a message has been put into the mailbox and cleared when the message is removed.
- Here is a simple function to put a message into the mailbox, assuming that the system supports only one mailbox used for all messages;

```c
void post(message *msg) {
P(mailbox.sem); /*wait for the mailbox */ copy(mailbox.data,msg); /*copy the data into the mailbox */ mailbox.flag = TRUE; /*set the
   flag to indicate a message is ready */ V(mailbox.sem); /*release the mailbox */
}
```

Here is a function to read from the mailbox:

```c
boolean pickup(message *msg) {
   boolean pickup = FALSE; /*local copy of the ready flag */
P(mailbox.sem); /*wait for the mailbox */
pickup = mailbox.flag; /*get the flag */
mailbox.flag = FALSE; /*remember that this message was received */
copy(msg,mailbox.data); /*copy the data into the caller's buffer */
V(mailbox.sem); /*release the flag:- -can't get the mail if we keep the mailbox */
return(pickup); /*return the flag value */
}
```

- The semaphores in pickup( ) ensure that a post( ) cannot interleave between the memory reads of the pickup operation.

**EVALUATING OPERATING SYSTEM PERFORMANCE:**
- The scheduling policy does not tell us all that we would like to know about the performance of a real system running processes.
- Our analysis of scheduling policies makes some simplifying assumptions:

---

**Fig 3.5.2 Use of a UML signal**

**MAILBOXES:**
- The mailbox is a simple mechanism for asynchronous communication.
- Some architectures define mailbox registers.
- These mailboxes have a fixed number of bits and can be used for small messages.
- We can also implement a mailbox using P( ) and V( ) using main memory for the mailbox storage.
- A very simple version of a mailbox, one that holds only one message at a time.
- In order for the mailbox to be most useful, we want it to contain two items: the message itself and a mail ready flag.
- The flag is true when a message has been put into the mailbox and cleared when the message is removed.
- Here is a simple function to put a message into the mailbox, assuming that the system supports only one mailbox used for all messages:

```c
void post(message *msg) {
P(mailbox.sem); /*wait for the mailbox */ copy(mailbox.data,msg); /*copy the data into the mailbox */ mailbox.flag = TRUE; /*set the
   flag to indicate a message is ready */ V(mailbox.sem); /*release the mailbox */
}
```

Here is a function to read from the mailbox:

```c
boolean pickup(message *msg) {
   boolean pickup = FALSE; /*local copy of the ready flag */
P(mailbox.sem); /*wait for the mailbox */
pickup = mailbox.flag; /*get the flag */
mailbox.flag = FALSE; /*remember that this message was received */
copy(msg,mailbox.data); /*copy the data into the caller's buffer */
V(mailbox.sem); /*release the flag:- -can't get the mail if we keep the mailbox */
return(pickup); /*return the flag value */
}
```

- The semaphores in pickup( ) ensure that a post( ) cannot interleave between the memory reads of the pickup operation.

**EVALUATING OPERATING SYSTEM PERFORMANCE:**
- The scheduling policy does not tell us all that we would like to know about the performance of a real system running processes.
- Our analysis of scheduling policies makes some simplifying assumptions:
We have assumed that context switches require zero time. Although it is often reasonable to neglect context switch time when it is much smaller than the process execution time, context switching can add significant delay in some cases.

- We have assumed that we know the execution time of the processes, that program time is not a single number, but can be bounded by worst-case and best-case execution times.
- We probably determined worst-case or best-case times for the processes insolation. But, in fact, they interact with each other in the cache. Cache conflicts among processes can drastically degrade process execution time.

We need to examine the validity of all these assumptions.

**Context switching time:**

It depends on several factors:

- The amount of CPU context that must be saved;
- Scheduler execution time.

- The execution time of the scheduler can of course be affected by coding practices.
- However, the choice of scheduling policy also affects the time required by the schedule to determine the next process to run.
- We can classify scheduling complexity as a function of the number of tasks to be scheduled.
- For example, Round-robin is often referred to as an $O(1)$ scheduling algorithm because its execution time is a constant independent of the number of tasks.
- Earliest deadline first scheduling, in contrast, requires sorting deadlines, which $O(n \log n)$ activity.

**Interrupt latency:**

- Interrupt latency for an RTOS is the duration of time from the assertion of a device interrupt to the completion of the device's requested operation.
- Interrupt latency is critical because data may be lost when an interrupt is not serviced in a timely fashion.

---

**Fig 3.6 Sequence diagram for RTOS interrupt latency**

- Fig 3.6 shows a sequence diagram for RTOS interrupt latency.
- A task is interrupted by a device.
- The interrupt goes to the kernel, which may need to finish a protected operation.
- Once the kernel can process the interrupt, it calls the interrupt service routine (ISR), which performs the required operations on the device.
- Once the ISR is done, the task can resume execution.
- Several factors in both hardware and software affect interrupt latency:
  - The **processor interrupt latency** was chosen when the hardware platform was selected; this is often not the dominant factor in overall latency.
  - The **execution time of the handler** depends on the device operation required, assuming that the interrupt handler code is not poorly designed.
  - This leaves **RTOS scheduling delays**, which can be the dominant component of RTOS interrupt latency, particularly if the operating system was not designed for low interrupt latency.

**Critical sections and Interrupt latency:**

- The RTOS can delay the execution of an interrupt handler in two ways.
- First, critical sections in the kernel will prevent the RTOS from taking interrupts.
- A critical section may not be interrupted, so the semaphore code must turn off interrupts.
Some operating systems have very long critical sections that disable interrupt handling for very long periods. Linux is an example of this phenomenon. Longer critical sections can improve performance for some types of workloads because it reduces the number of context switches. However, long critical sections cause major problems for interrupts.

**Fig 3.6.1 Interrupt latency during a critical section.**

- Fig 3.6.1 shows the effect of critical sections on interrupt latency.
- If a device interrupts during a critical section, that critical section must finish before the kernel can handle the interrupt.
- The longer the critical section, the greater the potential delay.
- Critical sections are one important source of scheduling jitter because a device may interrupt at different points in the execution of processes and hit critical sections at different points.

**Interrupt priorities and Interrupt latency:**

- Second, a higher-priority interrupt may delay a lower-priority interrupt.
- The priorities for interrupts are determined by hardware, not the RTOS.
- Furthermore, any interrupt handler preempts all user threads because interrupts are part of the CPU's fundamental operation.
- We can reduce the effects of hardware preemption by dividing interrupt handling into two different pieces of code.
- First, a very simple piece of code usually called an *interrupt service handler* (ISH) performs the minimal operations required to respond to the device.
- The rest of the required processing, which may include updating user buffers or other more complex operations, is performed by a user-mode thread known as an *interrupt service routine* (ISR).
- Because the ISR runs as a thread, the RTOS can use its standard policies to **ensure that all the tasks in the system receive their required resources**.

**RTOS Performance Evaluation tools:**

- Some RTOSs provide simulators or other tools that allow you to **view the operation of the processes in the system**.
- These tools will show not only abstract events such as processes but also context switching time, interrupt response time, and other overheads.
- This sort of view can be helpful in both functional and performance debugging.

**Windows CE** provides several performance analysis tools: **ILTiming**, an instrumentation routine in the kernel that measures both interrupt service routine and interrupt service thread latency; **OS Bench** measures the timing of operating system tasks such as critical section access, signals, and so on; **Kernel Tracker** provides a graphical user interface for RTOS events.

**Caches and RTOS Performance:**

- Many real-time systems have been designed based on the assumption that there is no cache present, even though one actually exists.
- Because the system architects lack tools that permit them to analyze the effect of caching.
- Because they do not know where caching will cause problems, they simply assume that there is no cache.
- By **analyzing the effects of the cache**, we can make much better use of the available hardware.
- Li and Wolf developed a model for estimating the performance of multiple processes sharing a cache.
In the model, some processes can be given reservations in the cache, such that only a particular process can inhabit a reserved section of the cache; other processes are left to share the cache.

We generally want to use cache partitions only for performance-critical processes because cache reservations are wasteful of limited cache space.

Performance is estimated by constructing a schedule, taking into account not just execution time of the processes but also the state of the cache.

Each process in the shared section of the cache is modeled by a binary variable: 1 if present in the cache and 0 if not.

Each process is also characterized by three total execution times: assuming no caching, with typical caching, and with all code always resident in the cache.

The always-resident time is unrealistically optimistic, but it can be used to find a lower bound on the required schedule time.

During construction of the schedule, we can look at the current cache state to see whether the no-cache or typical-caching execution time should be used at this point in the schedule.

We can also update the cache state if the cache is needed for another process.

**POWER OPTIMIZATION STRATEGIES FOR PROCESSES:**

- The RTOS and system architecture can use static and dynamic power management mechanisms to help manage the system’s power consumption.
- A power management policy is a strategy for determining when to perform certain power management operations.
- It is used to examine the state of the system to determine when to take actions.
- When the power consumption is low, then it takes more time to shut off and it makes longer the delay to perform certain action.
- Because power-down and power-up are not free, modes should be changed carefully.
- Determining when to switch into and out of a power-up mode requires an analysis of the overall system activity.
  - Avoiding a power-down mode can cost unnecessary power.
  - Powering down too soon can cause severe performance penalties.
  - Re-entering run mode typically costs a considerable amount of time.
  - A straight forward method is to power up the system when a request is received.

To avoid the long delay in handling the shut down request, we go for predictive shutdown.

**Predictive Shutdown:**

- It is a technique, they make guesses or prediction of the activity patterns based on a probabilistic model of expected behavior.
- This can cause two types of problems:
  - The requestor may have to wait for an activity period. In the worst case, the requestor may not make a deadline due to the delay incurred by system start-up.
  - The system may restart itself when no activity is imminent. As a result, the system will waste power.
- Although it has some problem, it can be used in several operations.
- Several predictive techniques are possible.
- A very simple technique is to use fixed times.
- For instance, if the system does not receive inputs during an interval of length $T_{on}$, it shuts down; a powered-down system waits for a period $T_{off}$ before returning to the power-on mode.
- They plotted the observed idle time ($T_{sid}$) of a graphics terminal versus the immediately preceding active time ($T_{on}$). The result was an L-shaped distribution as illustrated in Fig 3.7.

![Fig 3.7 An L-shaped usage distribution.](image-url)
In this distribution, the idle period after a long active period is usually very short, and the length of the idle period after a short active period is uniformly distributed.

Based on this distribution, they proposed a shut down threshold that depended on the length of the last active period, they shutdown when the active period length was below a threshold, putting the system into the vertical portion of the L distribution.

**Fig 3.7.1 Architecture of a power-managed system**

- In fig 3.7.1, the **service provider** is the machine whose power is being managed; the **service requestor** is the machine or person making requests of that power-managed system; a **queue** is used to hold pending requests (e.g., while waiting for the service provider to power up); and the **power manager** is responsible for sending power management commands to the provider.

- The power manager can observe the behavior of the requestor, provider, and queue.

- Each of these elements can be modeled by a Markov model which is a probabilistic state machine.

- Bemmel et al. showed that a power management policy that maximizes system performance while meeting a power consumption limitation can be solved in polynomial time.

**The Advanced Configuration and Power Interface (ACPI)**

- It is an open industry standard for power management services.

- It is designed to be compatible with a wide variety of OSs.

- It was targeted initially to PCs.

- In fig 3.7.2, ACPI provides some basic power management facilities and abstracts the hardware layer, the OS has its own power management module that determines the policy, and the OS then uses ACPI to send the required controls to the hardware and to observe the hardware’s state as input to the power manager.

**Fig 3.7.2 The advanced configuration and power interface and its relationship to a complete system**

- ACPI supports the following five basic global power states:
  
  i. **G3, the mechanical off state**, in which the system consumes no power.
  
  ii. **G2, the soft off state**, which requires a full OS reboot to restore the machine to working condition.

- This state has four substates:
  
  a. **S1**, a low wake-up latency state with no loss of system context;
  
  b. **S2**, a low wake-up latency state with a loss of CPU and system cache state;
  
  c. **S3**, a low wake-up latency state in which all system state except for main memory is lost; and
  
  d. **S4**, the lowest-power sleeping state, in which all devices are turned off.

- **G1**, the sleeping state, in which the system appears to be off and the time required to return to working condition is inversely proportional to power consumption.

- **G0, the working state**, in which the system is fully usable.

- The **legacy state**, in which the system does not comply with ACPI.

- The power manager typically includes an observer, which receives messages through the ACPI interface that describe the system behavior.

- It also includes a decision module that determines power management actions based on those observations.
EC6703 - EMBEDDED AND REAL TIME SYSTEMS

UNIT III

PROCESSES AND OPERATING SYSTEMS

EXAMPLE REAL TIME OPERATING SYSTEMS:

- POSIX
- Windows CE

WINDOWS CE:
- Windows CE supports devices such as smart phones, electronic instruments, etc.
- Windows CE is designed to run on multiple hardware platforms and instruction set architectures.
- Some aspects of Windows CE, such as details of the interrupt structure, are determined by the hardware architecture and not by the operating system itself.

WinCE architecture:

- Fig 3.8 shows a layer diagram for Windows CE.
- Applications run under the shell and its user interface.
- The Win32 APIs manage access to the operating system.
- A variety of services and drivers provide the core functionality.
- The OEM Adaption Layer (OAL) provides an interface to the hardware in much the same way that a HAL does in other software architecture.
- As shown in Fig 3.8.1, the hardware provides certain primitives such as a real-time clock and an external connection.
- The OAL itself provides services such as a real-time clock, power management, interrupts, and a debugging interface.
- A Board Support Package (BSP) for a particular hardware platform includes the OAL and drivers.

WinCE memory space:
- Windows CE provides support for virtual memory with a flat 32-bit virtual address space.
- A virtual address can be statically mapped into main memory for key kernel-mode code; an address can also be dynamically mapped, which is used for all user-mode and some kernel-mode code.
- Flash as well as magnetic disk can be used as a backing store.

Fig 3.8.2 Kernel and user address spaces in Windows CE
- Fig 3.8.2 shows the division of the address space into kernel and user with 2GB for the operating system and 2 GB for the user.
Fig 3.8.3 User address space in Windows CE

- Fig 3.8.3 shows the organization of the user address space.
- The top 1GB is reserved for system elements such as DLLs, memory mapped files, and shared system heap.
- The bottom 1GB holds user elements such as code, data, stack, and heap.

**WinCE threads and drivers:**
- WinCE supports two kernel-level units of execution: the thread and the driver.
- **Threads** are defined by executable files while **drivers** are defined by dynamically-linked libraries (DLLs).
  - A process can run multiple threads.
  - All the threads of a process share the same execution environment.
  - Threads in different processes run in different execution environments.
  - Threads are scheduled directly by the operating system.
  - Threads may be launched by a process or a device driver.
  - A driver may be loaded into the operating system or a process.
  - Drivers can create threads to handle interrupts.

**WinCE scheduling:**
- Each thread is assigned an **integer priority**.
- Lower-valued priorities signify higher priority. **0** is the highest priority and **255** is the lowest possible priority.
- **Priorities 248 through 255 are used for non-real-time threads** while the higher priorities are used for various categories of real-time execution.
- The operating system maintains a queue of ready processes at each priority level.
- Execution is divided into time quantum.
- Each thread can have a separate quantum, which can be changed using the API.
- If the running process does not go into the waiting state by the end of its time quantum, it is suspended and put back into the queue.
- Execution of a thread can also be blocked by a higher-priority thread.
- Tasks may be scheduled using either of two policies: a thread runs until the end of its quantum; or a thread runs until a higher-priority thread is ready to run.
- Within each priority level, round-robin scheduling is used.
- WinCE supports priority inheritance.
- When priorities become inverted, the kernel temporarily boosts the priority of the lower-priority thread to ensure that it can complete and release its resources.
- However, the kernel will apply priority inheritance to only one level.
- If a thread that suffers from priority inversion in turn causes priority inversion for another thread, the kernel will not apply priority inheritance to solve the nested priority inversion.

**WinCE interrupts:**
- Interrupt handling is divided among three entities:
  - **The interrupt service handler (ISH)** is a kernel service that provides the first response to the interrupt.
  - The ISH selects an **interrupt service routine (ISR)** to handle the interrupt.
  - The ISH runs in the kernel with interrupts turned off; as a result, it should be designed to do as little direct work as possible.
  - The ISR in turn calls an **interrupt service thread (IST)** which performs most of the work required to handle the interrupt. The IST runs in the OAL and so can be interrupted by a higher-priority interrupt.
- Fig 3.8.4 shows an response to an interrupt.
- The interrupt causes the ISH to vector the interrupt to the appropriate ISR.
The ISR in turn determines which IST to use to handle the interrupt and requests the kernel to schedule that thread.

The ISH then performs its work and signals the application about the updated device status as appropriate.

Windows CE 6.0 also supports kernel mode drivers.

These drivers run inside the kernel protection layer and so eliminate some of the overhead of user mode drivers.

However, kernel-mode and user-mode drivers use the same API.

![Fig 3.8.4 Sequence diagram for an interrupt](image)

**POSIX:** (Portable Operating System Interface)

- POSIX is a version of the Unix operating system created by a standards organization.
- While Unix was not originally designed as a real-time operating system, POSIX has been extended to support real-time requirements.
- The POSIX standard has many options; particular implementations do not have to support all options.
- The existence of features is determined by C preprocessor variables; for example, the F00 option would be available if the _POSIX_F00 preprocessor variable were defined.
- All these options are defined in the system include file unistd.h.

**Linux:**

- Linux is a POSIX-compliant operating system that is available as open source.
- Some versions of Linux may exhibit long interrupt latencies, primarily due to large critical sections in the kernel that delays interrupt processing.
- Two methods have been proposed to improve interrupt latency.
- A dual-kernel approach uses a specialized kernel, the co-kernel, for real-time processes and the standard kernel for non-real-time processes.
- All interrupts must go through the co-kernel to ensure that real-time operations are predictable.
- The other method is a kernel patch that provides priority inheritance to reduce the latency of many kernel operations.
- These features are enabled using the PREEMPT_RT mode.

**Processes in POSIX:**

- In POSIX, a new process is created by calling the fork() function to make a copy of an existing process.
- That function causes the operating system to create a new process (the child process) which is a nearly exact copy of the process that called fork() (the parent process).
- They both share the same code and the same data values with one exception, the return value of fork(): the parent process is returned the process ID number of the child process, while the child process gets a return value of 0.
- We can therefore test the return value of fork() to determine which process is the child:

  ```c
  childid = fork();
  if (childid == 0) {
      /* must be the child */
      /* do child process here */
  }
  ```
- POSIX provides the exec facility for overloading the code in a process.
So here is the process call with an overlay of the child's code on the child process:

```
childid = fork( );
if (childid == 0) {/*must be the child */  
  exec("mychild", childargs);  
  perror("execv");  
  exit(1);  
}
```

The `execv` function takes as argument the name of the file that holds the child's code and the array of arguments.

It overlays the process with the new code and starts executing it from the main( ) function.

In the absence of an error, `execv` should never return.

The code that follows the call to `perror( )` and `exit( )`, take care of the case where `execv( )` fails and returns to the parent process.

The `exit( )` function is a C function that is used to leave a process; it relies on an underlying POSIX function that is called `_exit( )`.

The parent process should use one of the POSIX wait functions before calling `exit( )` for itself.

The wait functions not only return the child process's status, in many implementations of POSIX they make sure that the child's resources (namely memory) are freed.

So we can extend our code as follows:

```
childid = fork( );
if (childid == 0) {/*must be the child */  
  exec("mychild", childargs);  
  perror("execv");  
  exit (1) ;  
}
else { /* is the parent */  
  parent_stuff( ); /* execute parent functionality */  
  wait(& cstatus) ;  
  exit (0) ;  
}
```

The `parent_stuff( )` function performs the work of the parent function.

The `wait( )` function waits for the child process; the function sets the integer `cstatus` variable to the return value of the child process.

**The POSIX process model:**

- POSIX does not implement lightweight processes.
- Each POSIX process runs in its own address space and cannot directly access the data or code of other processes.

**Real-time Scheduling in POSIX:**

- POSIX supports real-time scheduling in the `_POSIX_PRIORITY_SCHEDULING` resource.
- Not all processes have to run under the same scheduling policy.
- The `sched_setscheduler()` function is used to determine a process's scheduling policy and other parameters:
  ```c
  #include <sched.h>
  int i, my_process_id;
  struct sched_param my_sched_params;
  i = sched_setscheduler (my_process_id, SCHED_FIFO, & sched_params);
  ```
- POSIX supports rate-monotonic scheduling in the `SCHED_FIFO` scheduling policy.
- It is a strict priority-based scheduling scheme in which a process runs until it is preempted or terminates.
- The term FIFO simply refers to the fact that, within a priority, processes run in first-come first-served order.
- Two other useful functions allow a process to determine the minimum and maximum priority values in the system:
  ```c
  minval = sched_get_priority_min(SCHED RR);
  maxval = sched_get_priority_max(SCHED RR);
  ```
- The `sched_getparam( )` function returns the current parameter values for a process and `sched_setparams( )` changes the parameter values.
EC6703 - EMBEDDED AND REAL TIME SYSTEMS UNIT-III PROCESSES AND OPERATING SYSTEMS

- SCHED_RR is a combination of real-time and interactive scheduling techniques: within a priority level, the processes are time sliced.
- The SCHED_OTHER is defined to allow non-real-time processes to intermix with real time processes.
- Different processes in a system can run with different policies, so some processes may run SCHED_FIFO while others run SCHED_RR.

POSIX Semaphores:

- POSIX supports semaphores but it also supports a direct shared memory mechanism.
- POSIX supports counting semaphores in the _POSIX_SEMAPHORES option.
- A counting semaphore allows more than one process access to a resource at a time.
- If the semaphore allows up to N resources, then it will not block until N processes have simultaneously passed the semaphore; at that point, the blocked process can resume only after one of the processes has given up its semaphore.
- Counting semaphores is that they count down to 0, when the semaphore value is 0, the process must wait until another process gives up the semaphore and increments the count.
- Because there may be many semaphores in the system, each one is given a name.
- Names are similar to file names except that they are not arbitrary paths-they should always start with "/" and should have no other "/".
- The POSIX names for P and V are sem_wait( ) and sem_post( ) respectively.
- POSIX also provides a sem_trywait( ) function that tests the semaphore but does not block.

Here are examples of their use:

```c
int i;
  i = sem_wait(my_semaphore); /* P */
/* do useful work */
  i = sem_post(my_semaphore);  /* V */
/* sem_trywait tests without blocking */
  i = sem_trywait(my_semaphore);
```

- POSIX shared memory is supported under the _POSIX_SHARED_MEMORY_OBJECTS option.
- The shared memory functions create blocks of memory that can be used by several processes.
- The shm_open( ) function opens a shared memory object:
  ```c
  objdesc = shm_open("/memobj1",0_RDWR);
  ```
- This code creates a shared memory object called /memobj1 with read/write access; the 0_RDWR mode allows reading only.
- Before using the shared memory object, we must map it into the process memory space using the mmap( ) function.
- The munmap( ) function to unmap the memory when the process is done with it:
  ```c
  Only one process calls shm_open( ) to create the shared memory object and close( ) to destroy it; every process (including the one that created the object) must use mmap( ) and munmap( ) to map it into their address space.
```

POSIX pipes:

- The pipe is very familiar to Unix users from its shell syntax:
  ```c
  % foo file1 | baz > file2
  ```
- In this command, the output of foo is sent directly to the baz program's standard input by the operating system.
- The vertical bar (|) is the shell's notation for a pipe; programs use the pipe( ) function to create pipes.
- A parent process uses the pipe( ) function to create a pipe to talk to a child.
- It must do so before the child itself is created or it won't have any way to pass a pointer to the pipe to the child.
- Each end of a pipe appears to the programs as a file—the process at the head of the pipe writes to one file descriptor while the tail process reads from another file descriptor.
- Here is an example:
  ```c
  if (pipe(pipe_ends) < 0) { /* create the pipe, check for errors */
    perror("pipe");
    break;
  }
  /* create the process */
  ```
POSIX message queues:

- POSIX also supports message queues under the _POSIX_MESSAGE_PASSING facility.
- The advantage of a queue over a pipe is that, because queues have names, we don't have to create the pipe descriptor before creating the other process using it, as with pipes.
- The name of a queue follows the same rules as for semaphores and shared memory: it starts with a "/" and contains no other "/" characters.
- In this code, the 0_CREAT flag to mq_open() causes it to create the named queue if it doesn't yet exist and just opens the queue for the process if it does already exist:
- Messages can be prioritized, with a priority value between 0 and M0_PRI0_MAX (there are at least 32 priorities available).
- Messages are inserted into the queue such that they are after all existing messages of equal or higher priority and before all lower priority messages.
- When a process is done with a queue, it calls mq_close():
  
i = mq_close(myq);
UNIT IV
SYSTEM DESIGN TECHNIQUES AND NETWORKS
Design methodologies- Design flows - Requirement Analysis – Specifications-System analysis and architecture design – Quality Assurance techniques- Distributed embedded systems – MPSoCs and shared memory multiprocessors.

**DESIGN METHODOLOGIES:**
- Design methodology is employed in technology fields, including internet, software and information systems development.

**Why Design Methodologies?**
- **Process is important because without it, we can’t reliably deliver the products we want to create.**
- Thinking about the sequence of steps necessary to build something may seem superfluous.
- If you are designing embedded systems in your basement by yourself, having your own work habits is fine.
- But when several people work together on a project, they need to agree on who will do things and how they will get done.
- Therefore, since many embedded computing systems are too complex to be designed and built by one person, we have to think about design processes.

**PRODUCT METRICS:**
- The obvious goal of a design process is to create a product that does something useful.
- Typical specifications for a product will include functionality (e.g., Personal Digital Assistant), manufacturing cost (must have a retail price below $200), performance (must power up within 3 s), power consumption (must run for 12 hours on two AA batteries), or other properties.
- Design process has several important goals beyond function, performance, and power.

**Time-to-market:**
- Customers always want new features.
- The product that comes out first can win the market, even setting customer preferences for future generations of the product.
- The profitable market life for some products is 3.6 months. If you are 3 months late, you will never make money.
- Calculators, for example, are disproportionately sold just before school starts in the fall.
- If you miss your market window, you have to wait a year for another sales season.

**Design cost:**
- Many consumer products are very cost sensitive.
- Industrial buyers are also increasingly concerned about cost.
- The costs of designing the system are distinct from manufacturing cost, the cost of engineers salaries, computers used in design, and so on must be spread across the units sold.
- In some cases, only one or a few copies of an embedded system may be built, so design costs can dominate manufacturing costs.

**Quality:**
- Customers not only want their products fast and cheap, they also want them to be right.
- Correctness, reliability, and usability must be explicitly addressed from the beginning of the design job to obtain a high-quality product at the end.
- Processes evolve over time.
- They change due to external and internal forces.
- Customers may change, requirements change, products change, and available components change.
Internally, people learn how to do things better, people move onto other projects and others come in, and companies are bought and sold to merge and shape corporate cultures.

Software engineers have spent a great deal of time thinking about software design processes.

Much of this thinking has been motivated by mainframe softwaresuch as databases.

A good methodology is critical to build systems that work properly.

Delivering buggy systems to customers always causes dissatisfaction.

But in some applications, such as medical and automotive systems, bugs create serious safety problems that can endanger the lives of users.

**Example: Loss of the Mars Climate Observer**

In September 1999, the Mars Climate Observer, an unmanned U.S. spacecraft designed to study Mars, was lost—it most likely exploded as it heated up in the atmosphere of Mars after approaching the planet too closely.

The spacecraft came too close to Mars because of a series of problems, according to an analysis by IEEE Spectrum and contributing editor James Oberg.

From an embedded systems perspective, the first problem is best classified as a requirements problem.

The contractors who built the spacecraft at Lockheed Martin calculated values for flight controllers at the Jet Propulsion Laboratory (JPL).

**JPL did not specify the physical units to be used**, but they expected them to be in newtons.

**The Lockheed Martin engineers returned values in units of pound force.**

This discrepancy resulted in trajectory adjustments being **4.45 times larger** than they should have been.

The error was **not caught by a software configuration process nor was it caught by manual inspections.**

Although there were concerns about the spacecraft’s trajectory, errors in the calculation of the spacecraft’s position were not caught in time.

**DESIGN FLOWS:**

A design flow is a sequence of steps to be followed during a design.

Some of the steps can be performed by tools, such as compilers or CAD systems; other steps can be performed by hand.

**WATERFALL MODEL:**

![The waterfall model of software development](image)

Fig 4.1 The waterfall model of software development

Fig 4.1 shows the waterfall model introduced by Royce, the first model proposed for the software development process.

The waterfall development model consists of five major phases: **requirements** analysis determines the basic characteristics of the system; **architecture** design decomposes the functionality into major components; **coding** implements the pieces and integrates them; **testing** uncovers bugs; and **maintenance** entails deployment in the field, bug fixes, and upgrades.
The waterfall model gets its name from the largely one-way flow of work and information from higher levels of abstraction to more detailed design steps.

Although top–down design is ideal since it implies good foreknowledge of the implementation during early design phases, most designs are clearly not quite top–down.

Most design projects entail experimentation and changes that require bottom–up feedback.

As a result, the waterfall model is today cited as an unrealistic design process.

**SPIRAL MODEL:**

- Fig 4.1.1 illustrates an alternative model of software development called the spiral model.

  ![Spiral Model Diagram](image)

  **Fig 4.1.1** The spiral model of software design

- While the waterfall model assumes that the system is built once in its entirety, the spiral model assumes that several versions of the system will be built.
- As design progresses, more complex systems will be constructed.
- At each level of design, the designers go through requirements, construction, and testing phases.
- At later stages when more complete versions of the system are constructed, each phase requires more work, widening the design spiral.
- This successive refinement approach helps the designers understand the system they are working on through a series of design cycles.
- The first cycles at the top of the spiral are very small and short, while the final cycles at the spiral’s bottom add detail learned from the earlier cycles of the spiral.
- The spiral model is more realistic than the waterfall model because multiple iterations are often necessary to add enough detail to complete a design.
- However, a spiral methodology with too many spirals may take too long when design time is a major requirement.

**SUCCESSIVE REFINEMENT:**

- A first system is used as a rough prototype, and successive models of the system are further refined.
- This methodology makes sense when you are relatively unfamiliar with the application domain for which you are building the system.
- Refining the system by building several increasingly complex systems allows you to test out architecture and design techniques.
**Fig 4.1.2 A successive refinement development model**
- The various iterations may also be only partially completed; for example, continuing an initial system only through the detailed design phase may teach you enough to help you avoid many mistakes in a second design iteration that is carried through to completion.
- Embedded computing systems often involve the design of hardware as well as software.
- Even if you aren’t designing a board, you may be selecting boards and plugging together multiple hardware components as well as writing code.

**Fig 4.1.3A simple hardware/software design methodology**
- In Fig 4.1.3 the front-end activities such as specification and architecture simultaneously consider hardware and software aspects.
- Similarly, back-end integration and testing consider the entire system.
- In the middle, however, development of hardware and software components can go on relatively independently—while testing of one will require stubs of the other, most of the hardware and software work can proceed relatively independently.

**HIERARCHICAL DESIGN FLOWS:**
- In fact, many complex embedded systems are themselves built of smaller designs.
- The complete system may require the design of significant software components, application-specific integrated circuits (ASICs), and so on.
- The design flow for these complex systems resembles the flow shown in Fig 4.1.4.
- The implementation phase of a flow is itself a complete flow from specification through testing.
- In such a large project, each flow will probably be handled by separate people or teams.
- The teams must rely on each other’s results.
- Good communication is vital in such large projects.
Fig 4.1.4A Hierarchical design flow for an embedded system

CONCURRENT ENGINEERING:
- When designing a large system along with many people, it is easy to lose track of the complete design flow and have each designer take a narrow view of his or her role in the design flow.
- Attempts to take a broader approach and optimize the total flow.
- Reduced design time is an important goal.
- To eliminate the wall between design and manufacturing.
- Concurrent engineering efforts are comprised of several elements:
  - **Cross-functional teams** include members from various disciplines involved in the process, including manufacturing, hardware and software design, marketing, and so forth.
  - **Concurrent product realization** process activities are at the heart of concurrent engineering. Doing several things at once, such as designing various subsystems simultaneously, is critical to reducing design time.
  - **Incremental information sharing** and use helps minimize the chance that concurrent product realization will lead to surprises. As soon as new information becomes available, it is shared and integrated into the design. Cross-functional teams are important to the effective sharing of information in a timely fashion.
  - **Integrated project management** ensures that someone is responsible for the entire project, and that responsibility is not abdicated once one aspect of the work is done.
  - **Early and continual supplier involvement** helps make the best use of suppliers’ capabilities.
  - **Early and continual customer focus** helps ensure that the product best meets customers’ needs.

**Example: Concurrent engineering applied to telephone systems**
- A group at AT&T applied concurrent engineering to the design of PBXs (telephone switchingsystems).
- The company had a large existing organization and methodology for designing PBXs; their goal was to reengineer their process to reduce design time and make other improvements to the end product.
- They used the seven-step process described below.

1. **Benchmarking:**
   - They compared themselves to competitors and found that it took them 30% longer to introduce a new product than their best competitors.
   - Based on this study, they decided to shoot for a 40% reduction in design time.

2. **Breakthrough improvement:**
   - Next, they identified the factors that would influence their effort.
Three major factors were identified: increased partnership between design and manufacturing; continued existence of the basic organization of design labs and manufacturing; and support of managers at least two levels above the working level.

As a result, three groups were established to help manage the effort.

A steering committee was formed by midlevel managers to provide feedback on the project.

A project office was formed by an engineering manager and an operations analyst from the AT&T internal consulting organization.

Finally, a core team of engineers and analysts was formed to make things happen.

3. Characterization of the current process:

The core team built flowcharts and used other techniques to understand the current product development process.

The existing design and manufacturing process resembled this process as shown in fig 4.1.5.

![Fig 4.1.5](image-url)

- The core team identified several root causes of delays that had to be remedied.
- First, too many design and manufacturing tasks were performed sequentially.
- Second, groups tended to focus on intermediate milestones related to their narrow job descriptions, rather than trying to take into account the effects of their decisions on other aspects of the development process.
- Third, too much time was spent waiting in queues—jobs were handed off from one person to another very frequently.
- Fixing this problem was deemed to be fundamentally a managerial problem, not a technical one.
- Finally, the team found that too many groups had their own design databases, creating redundant data that had to be maintained and synchronized.

4. Create the target process:

- Based on its studies, the core team created a model for the new development process, which is reproduced below.

5. Verify the new process:

- The team undertook a pilot product development project to test the new process.
- Some challenges were identified; for example, in the sequential project the design of circuit board took longer than that of the mechanical enclosures, while in the new process the enclosures ended up taking longer, pointing out the need to start designing them earlier.

6. Implement across the product line:

- After the pilot project, the new methodology was rolled out across the product lines.
• This activity required training of personnel, documentation of the new standards and procedures, and improvements to information systems.

7. Measure results and improve:
• The performance of the new design flow was measured.
• The team found that product development time had been reduced from 18–30 months to 11 months.

**Fig 4.1.6**

**REQUIREMENT ANALYSIS:**
• **Requirements** are informal descriptions of what the customer wants, while **specifications** are more detailed, precise, and consistent descriptions of the system that can be used to create the architecture.
• Both requirements and specifications are, however, directed to the outward behavior of the system, not its internal structure.
• The overall goal of creating a requirements document is effective communication between the customers and the designers.
• We have two types of requirements: **functional and non-functional requirements**.
  • A **functional requirement** states what the system must do, such as compute an FFT.
  • A **non-functional requirement** can be any number of other attributes, including physical size, cost, power consumption, design time, reliability, and so on.
• A good set of requirements should meet several tests:
  1. **Correctness**: The requirements should not mistakenly describe what the customer wants. Part of correctness is avoiding over-requiring, the requirements should not add conditions that are not really necessary.
  2. **Unambiguousness**: The requirements document should be clear and have only one plain language interpretation.
  3. **Completeness**: All requirements should be included.
  4. **Verifiability**: There should be a cost-effective way to ensure that each requirement is satisfied in the final product.
  5. **Consistency**: One requirement should not contradict another requirement.
  6. **Modifiability**: The requirements document should be structured so that it can be modified to meet changing requirements without losing consistency, verifiability, and so forth.
  7. **Traceability**: Each requirement should be traceable in the following ways:
    - We should be able to trace backward from the requirements to know why each requirement exists.
We should also be able to trace forward from documents created before the requirements to understand how they relate to the final requirements.

We should also be able to trace forward to understand how each requirement is satisfied in the implementation.

We should also be able to trace backward from the implementation to know which requirements they were intended to satisfy.

**How do you determine requirements?**

- If the product is a continuation of a series, then many of the requirements are well understood.
- But even in the most modest upgrade, talking to the customer is valuable.
- In a large company, marketing or sales departments may do most of the work of asking customers what they want, but a surprising number of companies have designers talk directly with customers.
- Direct customer contact gives the designer an unfiltered sample of what the customers say.
- Talking to the customer may also include conducting surveys, organizing focus groups, or asking selected customers to test a mock-up prototype.

**SPECIFICATIONS:**

**Control-Oriented Specification Languages:**

**SDL: (Specification and Description Language)**

- SDL, language, was developed by the communications industry for specifying communication protocols, telephone systems, and so forth.
- As illustrated in Fig. 4.2, SDL specifications include states, actions, and both conditional and unconditional transitions between states.
- SDL is an event-oriented state machine model since transitions between states are caused by internal and external events.

**Statecharts:**

- The Statechart notation uses an event-driven model.
- Statecharts allow states to be grouped together to show common functionality.
- There are two basic groupings: OR and AND.
- Fig 4.2.1 shows an example of an OR state by comparing a traditional state transition diagram with a Statechart described via an OR state.
The statemachine specifies that the machine goes to state s4 from any of s1, s2, or s3 when they receive the input i2.

The Statechart denotes this commonality by drawing an OR state around s1, s2, and s3 (the name of the OR state is given in the small box at the top of the state).

A single transition out of the OR state s123 specifies that the machine goes into state s4 when it receives the i2 input while in any state included in s123.

Fig 4.2.1 An OR state in Statecharts

- The OR state still allows interesting transitions between its member states.
- There can be multiple ways to get into s123 (via s1 or s2), and there can be transitions between states within the OR state (such as from s1 to s3 or s2 to s3).
- The OR state is simply a tool for specifying some of the transitions relating to these states.

Fig 4.2.2 An AND state in Statecharts

- Fig 4.2.2 shows an example of an AND state specified in Statechart notation as compared to the equivalent in the traditional state machine model.
- In the traditional model, there are numerous transitions between the states; there is also one entry point into this cluster of states and one exit transition out of the cluster.
- In the Statechart, the AND state sab is decomposed into two components, sa and sb.
- When the machine enters the AND state, it simultaneously inhabits the states s1 of component sa and the state s3 of component sb.
- We can think of the system’s state as multidimensional.
- When it enters sab, knowing the complete state of the machine requires examining both sa and sb.
- The names of the states in the traditional state machine reveal their relationship to the AND state components.
Thus, state s1-3 corresponds to the Statechart machine having its sa component in s1 and its sb component in s3, and so forth.

We can exit this cluster of states to go to state s5 only when, in the traditional specification, we are in state s2-4 and receive input r.

In the AND state, this corresponds to state s2, sb in state s4, and the machine receiving the r input while in this composite state.

Although the traditional and Statechart models describe the same behavior, each component has only two states, and the relationships between these states are much simpler to see.

**AND/OR tables:**

**Boolean expression: cond1 or (cond2 and !cond3)**

![AND/OR table](image)

**Fig 4.2.3 AND/OR table**

- The rows in the AND/OR table are labeled with the basic variables in the expression.
- Each column corresponds to an AND term in the expression.
- For example, the AND term (cond2 and not cond3) is represented in the second column with a T for cond2, an F for cond3, and a dash (don’t-care) for cond1; this corresponds to the fact that cond2 must be T and cond3 F for the AND term to be true.
- We use the table to evaluate whether a given condition holds in the system.
- The current states of the variables are compared to the table elements.
- A column evaluates to true if all the current variable values correspond to the requirements given in the column.
- If any one of the columns evaluates to true, then the table’s expression evaluates to true, as we would expect for an AND/OR expression.

- The most important difference between this notation and Statecharts is that don’t-cares are explicitly represented in the table, which was found to be of great help in identifying problems in a specification table.

**Advanced Specifications:**

- The specification techniques developed to ensure the correctness and safety of this system can also be used in many applications, particularly in systems where much of the complexity goes into the control structure.

**Example: The TCAS II specification**

- TCAS II (Traffic Alert and Collision Avoidance System) is a collision avoidance system (CAS) for aircraft.
- Based on a variety of information, a TCAS unit in an aircraft keeps track of the position of other nearby aircraft.
- If TCAS decides that a mid-air collision may be likely, it uses audio commands to suggest evasive action—for example, a prerecorded voice may warn -DESCEND! DESCEND!! if TCAS believes that an aircraft above poses a threat and that there is room to maneuver below.
- TCAS makes sophisticated decisions in real time and is clearly safety critical.
- On the one hand, it must detect as many potential collision events as possible.
- On the other hand, it must generate as few false alarms as possible, since the extreme maneuvers it recommends are themselves potentially dangerous.
The TCAS II specification was written in their RSML language.
They use a modified version of State-chart notation for specifying states, in which the inputs to and outputs of the state are made explicit as shown in fig 4.3.

- They also use a transition bus to show sets of states in which there are transitions between all (or almost all) states. In the following example, there are transitions from a, b, c, or d to any of the other states as shown in fig 4.3.1.

The top-level description of the CAS as shown in fig 4.3.2.

- In fig 4.3.2, shows that the system has Power-off and Power-on states.
- In the power-on state, the system may be in Standby or Fully operational mode.
- In the Fully operational mode, three components are operating in parallel, as specified by the AND State: the own-aircraft subsystem, a subsystem to keep track of up to 30 other aircraft, and a subsystem to keep track of up to 15 Mode S ground stations, which provide radar information.
- The next diagram shows a specification of the Own-Aircraft AND state.
- Once again, the behavior of Own-Aircraft is an AND composition of several subbehaviors.
- The Effective-SL and Alt-SL states are two ways to control the sensitivity level (SL) of the system, with each state representing a different sensitivity level.
- Differing sensitivities are required depending on distance from the ground and other factors.
- The Alt-Layer state divides the vertical airspace into layers, with this state keeping track of the current layer.
- Climb-Inhibit and Descent-Inhibit states are used to selectively inhibit climbs (which may be difficult at high altitudes) or descents (clearly dangerous near the ground), respectively.
- Similarly, the Increase-Climb-Inhibit and Increase-Descent-Inhibit states can inhibit high-rate climbs and descents.
- Because the Advisory-Status state is rather complicated, its details are not shown here.

**Fig 4.3.3**

**SYSTEM ANALYSIS AND ARCHITECTURE DESIGN:**

**CRC Cards:** (Class Responsibility and Collaboration)
- The CRC card methodology is a well-known and useful way to help analyze a system's structure.
- It is particularly well suited to object-oriented design since it encourages the encapsulation of data and functions.
- The acronym CRC stands for the following three major items that the methodology tries to identify:
  - **Classes** define the logical groupings of data and functionality.
  - **Responsibilities** describe what the classes do.
  - **Collaborators** are the other classes with which a given class works.
- An example card is shown in Fig 4.4; it has space to write down the class name, its responsibilities and collaborators, and other information.

**Fig 4.4 Layout of a CRC card**

- The essence of the CRC card methodology is to have people write on these cards, talk about them, and update the cards until they are satisfied with the results.

**Advantages:**
- First, it is easy to get non-computerpeople to create CRC cards.
- Getting the advice of domain experts is very important in system design.
- Second, it aids even computer specialists by encouraging them to work in a group and analyze scenarios.
- The walkthrough process used with CRC cards is very useful in scoping out a design and determining what parts of a system are poorly understood.
- This informal technique is valuable to tool-based design and coding.
- If you still feel a need to use tools to help you practice the CRC methodology, software engineering tools are available that automate the creation of CRC cards.

**Class:**
- A class may represent a real-world object or it may describe an object that has been created solely to help architect the system.
- A class has both an internal state and a functional interface; the functional interface describes the class’s capabilities.
- CRC card class is easily transformable into a class definition in an object-oriented design.

**Responsibility:**
- The responsibility set is an informal way of describing that functional interface.
- The responsibilities provide the class’s interface, not its internal implementation.
- The responsibilities may be described informally in English.

**Collaborators:**
- The collaborators of a class are simply the classes that it talks to, that is, classes that use its capabilities or that it calls upon to help it do its work.
- CRC card analysis is performed by a team of people.
- As you are working in your group, you will be writing on these cards; you will probably discard many of them and rewrite them as the system evolves.
- The CRC card methodology is informal, but you should go through the following steps when using it to analyze a system:

1. **Develop an initial list of classes:**
   - Write down the class name and perhaps a few words on what it does.
   - A class may represent a real-world object or an architectural object.
   - Identifying which category the class falls into is helpful.
Each person can be responsible for handling a part of the system, but team members should talk during this process to be sure that no classes are missed and that duplicate classes are not created.

2. Write an initial list of responsibilities and collaborators:
   - The responsibilities list helps describe in a little more detail what the class does.
   - The collaborators list should be built from obvious relationships between classes.
   - Both the responsibilities and collaborators will be refined in the later stages.

3. Create some usage scenarios:
   - These scenarios describe what the system does.
   - Scenarios probably begin with some type of outside stimulus, which is one important reason for identifying the relevant real-world objects.

4. Walk through the scenarios:
   - This is the heart of the methodology.
   - During the walk-through, each person on the team represents one or more classes.
   - The scenario should be simulated by acting: people can call out what their class is doing, ask other classes to perform operations, and so on.
   - Moving around, for example, to show the transfer of data, may help you visualize the system’s operation.
   - During the walk-through, all of the information created so far is targeted for updating and refinement, including the classes, their responsibilities and collaborators, and the usage scenarios. Classes may be created, destroyed, or modified during this process.
   - You will also probably find many holes in the scenario itself.

5. Refine the classes, responsibilities, and collaborators:
   - Some of this will be done during the course of the walkthrough, but making a second pass after the scenarios is a good idea.
   - The longer perspective will help you make more global changes to the CRC cards.

6. Add class relationships:
   - Once the CRC cards have been refined, subclass and superclass relationships should become clearer and can be added to the cards.

Example: CRC card analysis of the elevator system

- First, we need the following basic set of classes:
  - Real-world classes: elevator car, passenger, floor control, car control, and car sensor.
  - Architectural classes: car state, floor control reader, car control reader, car control sender, and scheduler.

- For each class, we need the following initial set of responsibilities and collaborators. (An asterisk, *, is used to remind ourselves which classes represent real-world objects.)

<table>
<thead>
<tr>
<th>Class</th>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elevator car*</td>
<td>Moves up and down</td>
<td>Car control, car sensor, car control sender</td>
</tr>
<tr>
<td>Passenger*</td>
<td>Pushes floor control and car control buttons</td>
<td>Floor control, car control</td>
</tr>
<tr>
<td>Floor control*</td>
<td>Transmits floor requests</td>
<td>Passenger, floor control reader</td>
</tr>
<tr>
<td>Car control*</td>
<td>Transmits car requests</td>
<td>Passenger, car control reader</td>
</tr>
<tr>
<td>Car sensor*</td>
<td>Senses car position</td>
<td>Scheduler</td>
</tr>
<tr>
<td>Car state</td>
<td>Records current position of car</td>
<td>Scheduler, car sensor</td>
</tr>
<tr>
<td>Floor control reader</td>
<td>Interface between floor control and rest of system</td>
<td>Floor control, scheduler</td>
</tr>
<tr>
<td>Car control reader</td>
<td>Interface between car control and rest of system</td>
<td>Car control, scheduler</td>
</tr>
</tbody>
</table>
Several usage scenarios define the basic operation of the elevator system as well as some unusual scenarios:

1. One passenger requests a car on a floor, gets in the car when it arrives, requests another floor, and gets out when the car reaches that floor.
2. One passenger requests a car on a floor, gets in the car when it arrives, and requests the floor that the car is currently on.
3. A second passenger requests a car while another passenger is riding in the elevator.
4. Two people push floor buttons on different floors at the same time.
5. Two people push car control buttons in different cars at the same time.

- At this point, we need to walk through the scenarios and make sure they are reasonable.
- Find a set of people and walk through these scenarios.
- Do the classes, responsibilities, collaborators, and scenarios make sense? How would you modify them to improve the system specification?
QUALITY ASSURANCE:
- The quality of a product or service can be judged by how well it satisfies its intended function.
- A product can be of low quality for several reasons, such as
  - It was shoddily manufactured,
  - Its components were improperly designed,
  - Its architecture was poorly conceived, and
  - The product’s requirements were poorly understood.
- The quality assurance (QA) process is vital for the delivery of a satisfactory system.

Example: The Therac-25 medical imaging system
- In the course of six known accidents, these machines delivered massive radiation overdoses, causing deaths and serious injuries.
- Leveson and Turner analyzed the Therac-25 system and the causes for these accidents.
- The Therac-25 was controlled by a PDP-11 minicomputer and it was responsible for controlling a radiation gun that delivered a dose of radiation to the patient.
- It also runs a terminal that presents the main user interface.

![Diagram of Therac-25](image)

**Fig 4.5**
- The machine’s software was developed by a single programmer in PDP-11 assembly language over several years.
- The software includes four major components:
  - stored data,
  - a scheduler,
  - a set of tasks, and
  - Interrupt services.
- The three major critical tasks in the system follow:
  - A treatment monitor controls and monitors the setup and delivery of the treatment in eight phases.
  - A servo task controls the radiation gun, machine motions, and so on.
  - A housekeeper task takes care of system status interlocks and limit checks.
  - Let’s examine the software problems responsible for one series of accidents.
• Treat is the treatment monitor task, divided into eight subroutines (Reset, Datent, and so on).
• **Tphase** is a variable that controls which of these subroutines is currently executing.
• **Treat** reschedules itself after the execution of each subroutine.
• The Datent subroutine communicates with the keyboard entry task via the data entry completion flag, which is a shared variable.
• Datent looks at this flag to determine when it should leave the data entry mode and go to the **Setup test mode**.
• The **Mode/energy offset** variable is a shared variable: The top byte holds offset parameters used by the Datent subroutine, and the low-order byte holds mode and energy offset used by the Hand task.
• When the machine is run, the operator is forced to enter the mode and energy, but the operator can later edit the mode and energy separately.
• The software’s behavior is timing dependent.
• If the keyboard handler sets the completion variable before the operator changes the Mode/energy data, the Datent task will not detect the change—once Treat leaves Datent, it will not enter that subroutine again during the treatment.
• However, the Hand task, which runs concurrently, will see the new Mode/energy information.
• Apparently, the software included no checks to detect the incompatible data.
• After the Mode/energy data are set, the software sends parameters to a digital/analog converter and then calls a Magnet subroutine to set the bending magnets.
• Setting the magnets takes about 8 seconds and a subroutine called **Ptime** is used to introduce a time delay.
• Due to the way that Datent, Magnet, and Ptime are written, it is possible that changes to the parameters made by the user can be shown on the screen but will not be sensed by Datent.
• One accident occurred when the operator initially entered Mode/energy, went to the command line, changed Mode/energy, and returned to the command line within 8 s.
• The error therefore depended on the typing speed of the operator.
• Leveson and Turner emphasize that the following poor design methodologies and flawed architectures were at the root of the particular bugs that led to the accidents:
  - The designers performed a very limited safety analysis.
  - Mechanical backups were not used to check the operation of the machine.
  - Programmers created overly complex programs based on unreliable coding styles.

**QUALITY ASSURANCE TECHNIQUES:**
• The **International Standards Organization (ISO)** has created a set of quality standards known as **ISO 9000**.
• **ISO 9000** was created to apply to a broad range of industries, including but not limited to embedded hardware and software.
• The processes used to satisfy **ISO 9000** affect the entire organization as well as the individual steps taken during design and manufacturing.
• We can, however, make the following observations about quality management based on **ISO 9000**:
**Process is crucial:**
  - Knowing what steps are to be followed to create a high quality product is essential to ensuring that all the necessary steps are in fact followed.
**Documentation is important:**
  - The creation of the documents describing processes helps those involved understand the processes.
Documentation helps internal quality monitoring groups to ensure that the required processes are actually being followed and
It also helps outside groups to understand the processes and how they are being implemented.

Communication is important:
- Quality ultimately relies on people.
- Good documentation is an aid for helping people to understand the total quality process.
- The people in the organization should understand not only their specific tasks but also how their jobs can affect overall system quality.
- Metrics are used in quality control process to know the levels of quality, the execution speed of programs or the coverage of test patterns and the rate at which bugs are found.

Role of ISO 9000:
- To help organizations to study their total process, not just particular segments that may appear to be important at a particular time.
- One well-known way of measuring the quality of an organization’s software development process is the Capability Maturity Model (CMM).
- The CMM provides a model for judging an organization.
- It defines the following five levels of maturity:
  1. Initial: A poorly organized process, with very few well-defined processes. Success of a project depends on the efforts of individuals, not the organization itself.
  2. Repeatable: This level provides basic tracking mechanisms that allow management to understand cost, scheduling, and how well the systems under development meet their goals.
  3. Defined: The management and engineering processes are documented and standardized. All projects make use of documented and approved standard methods.
  4. Managed: This phase makes detailed measurements of the development process and product quality.
  5. Optimizing: At the highest level, feedback from detailed measurements is used to continually improve the organization’s processes.

VERIFYING THE SPECIFICATION:
- Verifying the requirements and specification is very important for the simple reason that bugs in the requirements or specification can be extremely expensive to fix later on.
- Fig 4.5.1 shows how the cost of fixing bugs grows over the course of the design process.

![Fig 4.5.1 Long-lived bugs are more expensive to fix](attachment:fig4.5.1.png)
- The longer a bug survives in the system, the more expensive it will be to fix.
- A coding bug, if not found until after system deployment, will cost money to recall and reprogram existing systems, among other things.
• Discovering bugs early is crucial because it prevents bugs from being released to customers, minimizes design costs, and reduces design time.

Requirements Validation:
• **Prototypes** are a very useful tool when dealing with end users—rather than simply describe the system to them in broad.
• It will not be fully functional since the design work has not yet been done.
• It can help the end user critique numerous functional and non-functional requirements, such as data displays, speed of operation, size, weight, and so forth.
• Certain programming languages, sometimes called **prototyping languages** or **specification languages**, are especially well suited to prototyping.
• Very high-level languages (such as Matlab in the signal processing domain) may be able to perform functional attributes, but not non-functional attributes such as the speed of execution.
• **Preexisting systems** can also be used to help the end user articulate his or her needs.
• In some cases, it may be possible to construct a prototype of the new system from the preexisting system.

Validation of Specifications:
• Building prototypes, specification languages, and comparisons to preexisting systems are as useful to system analysis and designers as they are to end users.
• Auditing tools may be useful in verifying consistency, completeness, and so forth.
• Working through **usage scenarios** often helps designers fillout the details of a specification and ensure its completeness and correctness.
• In some cases, **formal techniques** (that is, design techniques that make use of mathematical proofs) may be useful.
• Proofs may be done either manually or automatically.
• Automated proofs are particularly useful in certain types of complex systems.

**DESIGN REVIEWS:**
• It is a simple, low-cost way to catch bugs early in the design process.
• A design review is simply a meeting in which team members discuss a design, reviewing how a component of the system works.
• Some bugs are caught simply by preparing for the meeting, as the designer is forced to think through the design in detail.
• Other bugs are caught by people attending the meeting, who will notice problems that may not be caught by the unit’s designer.
• By catching bugs early and not allowing them to propagate into the implementation, we reduce the time required to get a working system.
• We can also use the design review to improve the quality of the implementation and make future changes easier to implement.

**Design Review Format:**
• A design review is held to review a particular component of the system.
• A design review team has the following members:
  ➢ The **designers of the component** being reviewed are, of course, central to the design process. They present their design to the rest of the team for review and analysis.
  ➢ The **review leader** coordinates the pre-meeting activities, the design review itself, and the post-meeting follow-up.
The review scribe records the minutes of the meeting so that designers and others know which problems need to be fixed.

The review audience studies the component. Audience members will naturally include other members of the project for which this component is being designed and they add valuable perspective and they may notice problems that team members have missed.

- The design review process begins before the meeting itself.
- The design team prepares a set of documents (code listings, flowcharts, specifications, etc.) that will be used to describe the component.
- These documents are distributed to other members of the review team in advance of the meeting, so that everyone has time to become familiar with the material.
- The review leader coordinates the meeting time, distribution of handouts, and so forth.
- During the meeting, the leader is responsible for ensuring that the meeting runs smoothly, while the scribe takes notes about what happens.
- The designers are responsible for presenting the component design.
- A top–down presentation often works well, beginning with the requirements and interface description, followed by the overall structure of the component, the details, and then the testing strategy.
- The audience should look for all types of problems listed below.
  - Is the design team’s view of the component’s specification consistent with the overall system specification, or has the team misinterpreted something?
  - Is the interface specification correct?
  - Does the component’s internal architecture work well?
  - Are there coding errors in the component?
  - Is the testing strategy adequate?
- The notes taken by the scribe are used in meeting follow-up.
- The design team should correct bugs and address concerns raised at the meeting.
- While doing so, the team should keep notes describing what they did.
- The design review leader coordinates with the design team, both to make sure that the changes are made and to distribute the change results to the audience.
- If the changes are straightforward, a written report of them is probably adequate.
- If the errors found during the review caused a major reworking of the component, a new design review meeting for the new implementation may be useful.

**DISTRIBUTED EMBEDDED SYSTEMS:**

**Distributed System:**

- Distributed System is more than two CPUs communicated in a tightly coupled manner.
- The main reason to implement distributed system in the embedded field is, it will provide high performance to perform complicated task in an easy manner, high processing rate.

**Reason for Distributed System in an embedded system:**

- In distributed systems the PEs(Processing Elements such as sensor, CPU, DSP) communicates with physically separated manner.
- Deadlines for processing the data are short.
- It has more cost-effective performance.
- One part of the system can be used to help diagnose problems in another part.
- Error identification is more easy.
- Several CPUs involved in the processing.
One CPU use to generate inputs for another CPU and to watch its output.

**NETWORK ABSTRACTIONS:**
- Networks are complex systems, network provide high-level services and 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.
- The OSI layers from lowest to highest level of abstraction are shown in Fig 4.6.

![Fig 4.6 The OSI model layers](image)

**Physical:**
- The physical layer defines the basic properties of the interface between systems, including the physical connections (plugs and wires), 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.
- This layer divides the message into packet form.

**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 check pointing.

**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.
- Simple embedded networks provide physical, data link and network services.
- Many embedded systems provide Internet service that requires implementing the full range of functions in the OSI model.
**CAN (Controller Area Network) BUS:**

- The CAN bus was designed for automotive electronics and was first used in production cars in 1991.
- It uses bit-serial transmission.
- CAN runs at rates of 1 MB/s over a twisted pair connection of 40 m.
- An optical link can also be used.
- The bus protocol supports multiple masters on the bus.

**Physical Layer:**

- As shown in Fig 4.6.1, each node in the CAN bus has its own electrical drivers and receivers that connect the node to the bus in wired-AND fashion.
- In CAN terminology, a **logical 1 on the bus is called recessive** and a **logical 0 is dominant**.
- When all nodes are transmitting 1s, the bus is said to be in the recessive state; when a node transmits a 0, the bus is in the dominant state.
- Data are sent on the network in packets known as **data frames**.
- CAN is a synchronous bus—all transmitters must send at the same time for bus arbitration to work.

![Fig 4.6.1 Physical and electrical organization of a CAN bus](image)

**Data frame:**

- The format of a CAN data frame is shown in Fig 4.6.2.

![Fig 4.6.2 The CAN data frame format](image)

- A data frame starts with a 1 and ends with a string of seven zeroes.
The first field in the packet contains the packet’s destination address and is known as the **arbitration field**.

The destination identifier is 11 bits long.

The trailing **remote transmission request (RTR)** bit is set to 0 if the data frame is used to request data from the device specified by the identifier.

When RTR1, the packet is used to write data to the destination identifier.

The **control field** provides an identifier extension and a 4-bit length for the data field with a 1 in between.

The **data field** is from 0 to 64 bytes, depending on the value given in the control field.

A **cyclic redundancy check (CRC)** is sent after the data field for error detection.

The **acknowledge field** is used to let the identifier signal whether the frame was correctly received:

- The sender puts a recessive bit (1) in the ACK slot of the acknowledge field;
- If the receiver detected an error, it forces the value to a dominant (0) value.

If the sender sees a 0 on the bus in the ACK slot, it knows that it must retransmit.

The ACK slot is followed by a single bit delimiter followed by the end-of-frame field.

**Arbitration:**

Control of the CAN bus is arbitrated using a technique known as **Carrier Sense Multiple Access with Arbitration on Message Priority (CSMA/AMP)**.

Network nodes transmit synchronously, so they all start sending their identifier fields at the same time.

When a node hears a dominant bit in the identifier when it tries to send a recessive bit, it stops transmitting.

By the end of the arbitration field, only one transmitter will be left.

The identifier field acts as a priority identifier, with the all-0 identifier having the highest priority.

**Remote Frames:**

A remote frame is used to request data from another node.

The requestor sets the RTR bit to 0 to specify a remote frame; it also specifies zero data bits.

The node specified in the identifier field will respond with a data frame that has the requested value.

Note that there is no way to send parameters in a remote frame.

**Error Handling:**

An error frame can be generated by any node that detects an error on the bus.

Upon detecting an error, a node interrupts the current transmission with an error frame, which consists of an error flag field followed by an error delimiter field of 8 recessive bits.

The **error delimiter field** allows the bus to return to the quiescent state so that data frame transmission can resume.

The bus also supports an overload frame, which is a special error frame sent during the interframe quiescent period.

An **overload frame** signals that a node is overloaded and will not be able to handle the next message.

The **CRC field** can be used to check a message’s data field for correctness.

If a transmitting node does not receive an acknowledgment for a data frame, it should retransmit the data frame until the frame is acknowledged.

This action corresponds to the **data link layer** in the OSI model.

As shown in fig 4.6.3, the controller implements the physical and data link layers; since CAN is a bus, it does not need network layer services to establish end-to-end connections.
The protocol control block is responsible for determining when to send messages, when a message must be resent due to arbitration losses, and when a message should be received.

![Fig 4.6.3 Architecture of a CAN controller](image)

**Other Automotive Networks:**
- The **FlexRay network** has been designed as the next generation of system buses for cars.
- FlexRay provides high data rates—up to 10 MB/s—with deterministic communication.
- It is also designed to be fault-tolerant.
- The **Local Interconnect Network (LIN)** bus was created to connect components in a small area, such as a single door.
- The physical medium is a single wire that provides data rates of up to 20 KB/s for up to 16 bus subscribers.
- All transactions are initiated by the master and responded to by a frame.
- Several buses have come into use for **passenger entertainment**.
- **Bluetooth** is becoming the standard mechanism for cars to interact with consumer electronics devices such as audio players or phones.
- The **Media Oriented Systems Transport (MOST) bus** was designed for entertainment and multimedia information.
- The basic MOST bus runs at 24.8 MB/s and is known as MOST 25; 50 and 150 MB/s versions have also been developed.
- MOST can support up to 64 devices and the network is organized as a ring.
- Data transmission is divided into channels.
- A **control channel** transfers control and system management data.
- **Synchronous channels** are used to transmit multimedia data; MOST 25 provides up to 15 audio channels.
- An **asynchronous channel** provides high data rates but without the quality-of-service guarantees of the synchronous channels.

**DISTRIBUTED COMPUTING IN CARS AND AIRPLANES:**
- Cars provide an ideal opportunity for message passing style multiprocessing.
- A car is controlled by a network of processors that each has its own responsibility but must communicate with other processors to make sure that the different subsystems act together.

**Car subsystems:**
- Fig 4.6.4 shows a car network and three of the major subsystems in the car: the engine, the transmission, and the anti-lock braking system (ABS).
Fig 4.6.4 Major elements of an automobile network

- Each of these is a mechanical system that is controlled by a processor.
- First consider the roles of the mechanical systems, all of which are mechanically coupled together:
  - The engine provides power to drive the wheels.
  - The transmission mechanically transforms the engine’s rotational energy into a form most useful by the wheels.
  - The ABS system controls how the brakes are applied to each of the four wheels. ABS can separately control the brake on each wheel.
- Now consider the roles of the associated processors:
  - The engine controller accepts commands from the driver via the gas pedal. It also takes several measurements. Based on the commands and measurements, it determines the spark and fuel timing on every engine cycle.
  - The transmission controller determines when to change gears.
  - The ABS system takes braking commands from the driver via the brake pedal. It also takes measurements from the wheels about their rotating speed.

Car subsystem interactions:
- The engine controller may change the spark timing during gear shifting to reduce shocks during shifting.
- The transmission controller must receive the throttle position from the engine controller to help it determine the proper shifting pattern for the transmission.
- The ABS system tells the transmission when brakes are being applied in case the gear needs to be shifted.

Example: Free scale MPC5676R
The MPC5676R is a dual-processor platform for power train systems.
- The two main processors are members of the Power Architecture™ Book E architecture and user-mode compatible with PowerPC.
- They provide short vector instructions for use in signal processing.
- Each processor has its own 16-K data and instruction caches.
- The time processing unit can be used to generate and read waveforms.
- Interfaces to the CAN, UN, and FlexRay networks are supported.

**Avionics:**
- Aircraft electronics are known as avionics.
- The certification process for production aircraft is two-fold: first, the design is certified in a process known as **type certification**; then, the manufacture of each aircraft is certified during production.
- The traditional architecture for an avionics system has a separate unit for each function: artificial horizon, engine control, flight surfaces, etc.
- These units are known as **line replaceable units (LRUs)** and are designed to be easily plugged and unplugged into the aircraft during maintenance.
- A more sophisticated system is bus-based.
- The Boeing 777 avionics, for example, is built from a series of racks.
- Each rack is a set of core processor modules (CPMs), I/O modules, and power supplies.
- The CPMs may implement one or more functions.
- A bus known as SAFE bus connects the modules.
- Cabinets are connected together using serial bus known as ARINC 629.
- A more distributed approach to avionics is the **federated network**.
- In this architecture, a function or several functions have their own network.
- The networks share data as necessary for the interaction of these functions.
- A federated architecture is designed so that a failure in one network will not interfere with the operation of the other networks.
- The **Genesis Platform** is a next-generation architecture for avionics and safety-critical systems; it is used on the Boeing 787 Dreamliner.
- It does not require a one-to-one correspondence between application groups and network units.
- It defines a virtual system for the avionics applications that are then mapped onto a physical network that may have a different topology.

**The I²C Bus:**
- The I²C bus is a simple and well-known bus commonly used to link microcontrollers into systems.
- Example for I²C is used for the command interface in an MPEG-2 video chip.
- In this, a separate bus was used for high-speed video data and setup information was transmitted to the on-chip controller through an I²C bus interface.

**Physical layer:**
- I²C is designed to meet the low cost and easy to implement.
- The data transmission speed is moderate of 100-400 KB/s
- As a result, it uses only two lines:
  - **Serial Data Line (SDL)** for data transmission and
  - **Serial Clock Line (SCL)**, which indicates when valid data are on the data line.
- Fig 4.7 shows the basic I²C bus, 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 may have more than one master.
• Other nodes may act as slaves that only respond to requests from masters.

**Fig 4.7 Structure of an I^2C bus system**

**Electrical interface:**

- The basic electrical interface to the bus is shown in Fig 4.7.1.
- 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.
- 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 and also it allows a slave device to drag a clock signal when respective data is to be read.
- The master is responsible for generating the SCL clock, but the slave can drag the low period of the clock not the high period.
- The I^2C bus is designed as a multi-master bus for several different devices may act as the master.
- A master drives both SCL and SDL when it is sending data.
- When the bus is idle, both SCL and SDL remain high.
- When the SCL and SDL are try to drive two different values, the open collector/open drain circuitry prevents errors and sending the information status to the masters and slaves.
- If the device receives a different value without error then the transmission is known it is interfering with another message not the original message.
Data Link Layer:
- Every I^2C device has separate address.
- These addresses consist of 7 bits of device address and one bit of read/write data.
- Totally 8 bit of address in extended I^2C device allows 10-bit addresses.
- To generate 0000000 is used for 8 bit device to signal all devices simultaneously and for extended 10-bit device generate call address is 11110XX.
- The format of an address transmission is shown in Fig 4.7.2.

![Fig 4.7.2 Format of an I^2C address transmission.](image)

![Fig 4.7.3 State transition graph for an I^2C bus master](image)

- A bus transaction comprised a series of 1-byte transmissions and an address followed by one or more data bytes.
- The transaction is between the master and slave.
- 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.
- 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.
- 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 by sending a start after the data transmission, followed by another address transmission and then more data.
- The Typical bus transaction is shown in Fig 4.7.4.

![Fig 4.7.4 Typical bus transactions on the I^2C bus](image)
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.

**Byte Format:**

![Fig 4.7.5 Transmitting a byte on the \( \text{I}^2\text{C} \) bus](image)

- Fig 4.7.5 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.
- 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.

**Bus arbitration:**

- 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.
- In many cases, arbitration will be completed during the address portion of a transmission, but arbitration may continue into the data portion.

**Application Interface:**

- The \( \text{I}^2\text{C} \) interface on a microcontroller can be implemented with varying percentages of the functionality in software and hardware.
- This system has a 1-bit hardware interface with routines for byte level functions.
- The \( \text{I}^2\text{C} \) device takes care of generating the clock and data.

![Fig 4.7.6 An \( \text{I}^2\text{C} \) interface in a microcontroller](image)
• 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 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.

ETHERNET:
• Ethernet is very widely used as a local area network for general-purpose computing.
• Because the low cost of Ethernet interfaces, it has significantly used as a network for embedded computing.
• Ethernet is particularly useful
  ➢ When PCs are used as platforms,
  ➢ Use of standard components,
  ➢ The network does not have to meet rigorous real-time requirements.
• The physical organization of an Ethernet is very simple, as shown in Fig 4.8.

![Fig 4.8 Ethernet organization](image)

• 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.
• The main problem in the Ethernet network is the node does not have synchronization as like I²C bus.
• In the Ethernet two nodes transmit the data at the same time, it will cause collision on the network.
• To avoid this problem, the Ethernet arbitration scheme is known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD) used.
• The algorithm is outlined in Fig 4.8.1.

![Fig 4.8.1 The Ethernet CSMA/CD algorithm](image)
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.

![Exponential backoff times](image)

**Fig 4.8.2 Exponential backoff times**
- This 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.
- Collision to be detected by both nodes.
- Each node’s signal must be able to travel from one end of the bus to the another end of the bus.
- Ethernets can run up to several hundred meters.
- The basic Ethernet packet format shown in fig 4.8.3.
- It provides addresses of both the destination and the source.
- It also provides for a variable-length data payload.

**Fig 4.8.3 Ethernet packet format**
- 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.

**Real-time operations over Ethernet:**
- Ethernet was not designed to support real-time operations; because it will not deliver data in proper time.
- There are three ways to reduce the variance in Ethernet’s packet delivery time
  - Suppress collisions on the network,
  - Reduce the number of collisions,
  - Resolve collisions deterministically.

**Other industrial networks:**
**Field bus:**
- Field bus 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 (HSE) based on the 100 MB/s Ethernet.
INTERNET:
- The Internet Protocol (IP) 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 for embedded computing.

Internetworking:
- Internet protocol is not defined over a particular physical implementation—it is an internetworking standard.
- Internet protocol transmits the message in the form of packets.
- Internet packets carry some information from one network to another network.
- Internet packet will travel over several different networks from source to destination.

![Diagram of protocol utilization in Internet communication](image)

**Fig 4.9 Protocol utilization in Internet communication**
- The relationship between IP and individual networks is illustrated in Fig 4.9.
- 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 those data and routing the data to node B that is destination node.
- A node that transmits data among different types of networks is known as a router.
- All the packets are transmitting data via router only.
- Each router will contain routing table which contains all the information about packet, router, routing path etc.
- The basic format of an IP packet is shown in Fig 4.9.1.

![Diagram of IP packet structure](image)

**Fig 4.9.1 IP packet structure**
- 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.xxx.xxx.xxx.
- The names by which users and applications typically refer to Internet nodes, such as foo.baz.com, 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.
- The Internet also provides higher-level services built on top of IP.

**Example: Transmission Control Protocol (TCP)**

![Image of the Internet service stack]

- 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.
- Fig 4.9.2 shows the relationships between IP and higher-level Internet services.
- Internet protocol provides two kinds of services:
  - Transmission Control Protocol (TCP)
  - User Datagram Protocol (UDP)
- TCP is used to provide many services:
  1. FTP- File Transport Protocol for batch file transfers,
  2. HTTP- Hypertext Transport Protocol for World Wide Web service,
  3. SMTP-Simple Mail Transfer Protocol for email, and
  4. Telnet for virtual terminals.

**UDP: User Datagram Protocol**

- A separate transport protocol which provides connectionless services.
- It is used as the basis for the network management services provided by the Simple Network Management Protocol (SNMP).

**MPSOCS AND SHARED MEMORY MULTIPROCESSORS:**

**Multiprocessors:**

- A multiprocessor is any computer system with two or more processors coupled together.
- It is used for scientific or business applications to have regular architecture.
Reason to use multiprocessor in all embedded system design:
- Real time performance
- Power consumption
- Cost
- Shared memory processors are well-suited to signal processing systems applications that require a large amount of data to be processed.
- Most MPSoCs are shared memory systems.
- Shared memory allows for processors to communicate with varying patterns.
- If the pattern of communication is very fixed and if the processing of different steps is performed in different units, then a networked multiprocessor may be most appropriate.
- Shared memory provides flexibility for one processing element is used for several different steps and also for varying communication patterns between steps.

Heterogeneous shared memory multiprocessors:
- Many high-performance embedded platforms are heterogeneous multiprocessors.
- The PEs may be programmable processors with different instruction sets Provide
- Accelerators provide
  1. Little or no programmability 2. Even faster and 3. Lower-power operation for a narrow range of functions

Example: TI TMS320DM816x DaVinci digital media processor
- The DaVinci 816x is designed for high-performance video applications.

![Diagram of DaVinci 816x](image)

**Fig 4.10**
- It includes both a CPU, a DSP, and several specialized units:
- The 816x has two main programmable processors.
- The ARM Cortex A8 includes the Neon multimedia instructions and in-order dual-issue machine.
- The C674x is a VLIW DSP, has six ALUs and 64 general-purpose registers.
- The HD video coprocessor subsystem (HDVICP2) provides image and video acceleration.
- It natively supports several standards, such as H.264 (used in BluRay), MPEG-4, MPEG-2, and JPEG.
- It includes specialized hardware for major image and video operations, including transform and quantization, motion estimation, and entropy coding.
- It also has its own DMA engine.
- It can operate at resolutions up to 1080P/I at 60 frames/sec.
- The HD video processing subsystem (HDVPSS) provides additional video processing capabilities.
- It can process up to three high-definition and one standard-definition video streams simultaneously.
- It can perform operations such as scan rate conversion, chromakey, and video security.
- The graphics unit is designed for 3D graphics operations that can process up to 30 M triangles/sec

**ACCELERATORS:**
- An accelerator is attached to CPU buses to quickly execute certain key functions.
Accelerators can provide large performance increases for applications with computational kernels.
Accelerators can also provide critical speedups for low-latency I/O functions.

**Accelerator system design:**

- The simultaneous design of hardware and software to meet system objectives.
- To take a given computing platform by adding accelerators, can customize the embedded platform to better meet our application’s demands.
- As illustrated in Fig 4.10.1, a CPU accelerator is attached to the CPU bus and the CPU is often called the **host**.
- The CPU talks to the accelerator through data and control registers in the accelerator.
- These registers allow the CPU to monitor the accelerator’s operation and to give the accelerator commands.
- The CPU and accelerator may also communicate via shared memory.
- The accelerator operates on a large volume of data with efficient data in memory.
- Accelerator will perform read and write operation directly.
- The CPU and accelerator use synchronization mechanisms to ensure that they do not destroy each other’s data.
- An accelerator is not a co-processor.
- A co-processor is connected to the internals of the CPU and processes instructions as defined by opcodes.
- An accelerator interacts with the CPU through the programming model interface; it does not execute instructions.
- Its interface is functionally equivalent to an I/O device, although it usually does not perform input or output.
- Both CPUs and accelerators perform computations for specification.

**Analysis:**
- The first task in designing an accelerator is determining that our system actually needs one.
- Next task is to identify, after having accelerator in the design of system must run more quickly.
- To ensure that the accelerator must increase the system speed up.

**Design:**
- Once we have analyzed the system, we need to design the accelerator itself.
- In the design of accelerator, many things to be identified first.
  - To understand the algorithm for the accelerator is more important and it is in the form of a high-level language program.
- To translate the algorithm description into a hardware design, a considerable task in itself.
- To make interface between the accelerator core and the CPU bus.
- The interface includes more than bus handshaking logic.
- To determine how the application software on the CPU will communicate with the accelerator and provide the required registers.
- We have to implement shared memory synchronization operations.
- We can add address generation logic to read and write large amounts of data from system memory.

Finally, we will have to design the CPU-side interface to the accelerator.
- The application software will have to talk to the accelerator, providing it data and telling it what to do.
- Accelerator operation is synchronized for the application.
- From the synchronization an accelerator knows when it has the required data and the CPU knows when it has received the desired results.

- **Field-programmable gate arrays (FPGAs)** provide one useful platform for custom accelerators.
- An FPGA has a fabric with both programmable logic gates and programmable interconnect that can be configured to implement a specific function.
- Most FPGAs also provide on-board memory that can be configured with different ports for custom memory systems.
- Some FPGAs provide on-board CPUs to run software that can talk to the FPGA fabric.
- Small CPUs can also be implemented directly in the FPGA fabric; the instruction sets of these processors can be customized for the required function.
- **Example: Xilinx Zynq-7000 platform** combines a processor system augmented with an FPGA fabric.

The main CPU is a two-processor ARM MP Core with two Cortex-A9 CPUs.
- The CPU platform includes a variety of typical I/O devices connected to the processor by an AMBA bus.
- The AMBA bus also provides a number of ports to the FPGA fabric which can be used to build custom logic, programmable processors, or embedded RAM.

**Accelerators Performance analysis:**
- The main reason to have accelerator is speedup: how much faster is the system with the accelerator than the system without it.
- The speedup factor of accelerator will depend upon the following factors:
  - **Single threaded or multithreaded**
  - **Blocking vs non-blocking.**
- The CPU sits idle while the accelerator runs in the single-threaded case and the CPU can do useful work in parallel with the accelerator in the multithreaded case.
- Blocking means the CPU’s scheduler block other operations and wait for the accelerator call to complete.
- Non-blocking means the CPU allow some other process to run in parallel with the accelerator.
As shown in Fig 4.10.3, data dependencies allow P2 and P3 to run independently on the CPU, but P2 relies on the results of the A1 process that is implemented by the accelerator.

However, in the single-threaded case, the CPU blocks to wait for the accelerator to return the results of its computation.

As a result, it does not matter whether P2 or P3 runs next on the CPU.

In the multithreaded case, the CPU continues to do useful work while the accelerator runs, so the CPU can start P3 just after starting the accelerator and finish the task earlier.

**Accelerator execution time:**

As shown in Fig 4.10.4, the execution time for the accelerator depends on more than just the time required to execute the accelerator's function.

It also depends on the time required to get the data into the accelerator and back out of it.

The total execution time may be written as

\[ t_a = t_{read} + t_{write} + t_{accelerator} \]  

Where \( t_a \) is the execution time of the accelerator assuming all data are available, and \( t_{read} \) and \( t_{write} \) are the times required for reading and writing the required variables, respectively.

The values for \( t_{read} \) and \( t_{write} \) must reflect the time required for the bus transactions, including the following factors:
- The time required to flush any register or cache values to main memory, if those values are needed in main memory to communicate with the accelerator; and
- The time required for transfer of control between the CPU and accelerator.
- A more sophisticated accelerator could try to overlap input and output with computation.
- **Example of overlapped I/O and computation** is streaming data applications such as digital filtering.
- As illustrated in Fig 4.10.5, an accelerator may take in one or more streams of data and output a stream.
- **Latency requirements** generally require that outputs be produced on the fly rather than storing up all the data and then computing; furthermore, it may be impractical to store long streams at all.
- In this case, the $\Delta t_{\text{in}}$ and $\Delta t_{\text{out}}$ terms are determined by the amount of data read in before starting computation and the length of time between the last computation and the last data output.

![Fig 4.10.5 Streaming data in and out of an accelerator](image)

- The total speedup $S$ for a kernel can be written
  \[ S = \Delta t_{\text{soft}} - \Delta t_{\text{alt}} = \Delta t_{\text{soft}} - \Delta t_{\text{alt}} + \Delta t_{\text{alt}} \]
  \[ = \Delta t_{\text{soft}} - \Delta t_{\text{alt}} + \Delta t_{\text{alt}} \quad \cdots (2) \]
- Where $\Delta t_{\text{soft}}$ is the execution time of the equivalent function in software on the CPU and $n$ is the number of times the function will be executed.

**System speedup:**
- In a single-threaded system, the evaluation of the accelerator’s speedup to the total system speedup is simple: The system execution time is reduced by $S$.

![Fig 4.10.6 Evaluating system speedup in a single-threaded implementation](image)

- The reason is illustrated in Fig 4.10.6, the single thread of control gives us a single path whose length we can measure to determine the new execution speed.
- Evaluating system speedup in a multithreaded environment requires more than one execution path as shown in Fig 4.10.7
- The total system execution time depends on the longest path from the beginning of execution to the end of execution.
In this case, the system execution time depends on the relative speeds of P3 and P2 plus A1.

![Flow of control](image)

**Fig 4.10.7 Evaluating system speedup in a multithreaded implementation**

- If P2 and A1 together take the most time, P3 will not play a role in determining system execution time.
- If P3 takes longer, then P2 and A1 will not be a factor.
- To determine system execution time, we must label each node in the graph with its execution time.
- Efficient graph algorithms can also be used to compute the longest path.
- This analysis shows the importance of selecting the proper functions to be moved to the accelerator.

**Scheduling and Allocation:**

- We must **schedule** operations in time, including communication on the network and computations on the processing elements.
  - Clearly, the scheduling of operations on the PEs and the communications between the PEs are linked.
  - If one PE finishes its computations too late, it may interfere with another communication on the network as it tries to send its result to the PE that needs it.
  - This is bad for both the PE that needs the result and the other PEs whose communication is interfered with.
- We must **allocate** computations to the processing elements.
  - The allocation of computations to the PEs determines what communications are required-if a value computed on one PE is needed on another PE, it must be transmitted over the network.

**Example: Scheduling and Allocating Processes on a Distributed Embedded System**

- We can specify the system as a task graph.
- However, different processes may end up on different processing elements.

![Diagram](image)

**Fig 4.10.8**

- We have labeled the data transmissions on each arc so that we can refer to them later.
- We want to execute the task on the platform as shown in fig 4.10.9

![Diagram](image)

**Fig 4.10.9**

- The platform has two processing elements and a single bus connecting both PEs.
- To make decisions about where to allocate and when to schedule processes, we need to know how fast each process runs on each PE.
- Here are the process speeds:
- The dash entry signifies that the process cannot run on that type of processing element.
- In practice, a process may be excluded from some PEs for several reasons.
- If we use an ASIC to implement a special function, it will be able to implement only one process.
- A small CPU such as a microcontroller may not have enough memory for the process's code or data; it may also simply run too slowly to be useful.
- If two processes are allocated to the same PE, they can communicate using the PE's internal memory and incur no network communication time.
- Each edge in the task graph corresponds to a data communication that must be carried over the network.
- Because all PEs communicate at the same rate, the data communication rate is the same for all transmissions between PEs.
- We need to know how long each communication takes.
- In this case, d1 is a short message requiring 2 time units and d2 is a longer communication requiring 4 time units.
- As an initial design, let us allocate P1 and P2 to M1 and P3 to M2.
- This allocation would, on the surface, appear to be a good one because P1 and P2 are both placed on the processor that runs them the fastest.
- This schedule shows what happens on all the processing elements and the network:

<table>
<thead>
<tr>
<th></th>
<th>M1</th>
<th>M2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>P2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>P3</td>
<td>-</td>
<td>5</td>
</tr>
</tbody>
</table>

![Fig 4.10.10](image)

- The schedule has length 19.
- The d1 message is sent between the processes internal to P1 and does not appear on the bus.
- Let's try a different allocation: P1 on M1 and P2 and P3 on M2. This makes P2 run more slowly.
- Here is the new schedule:

![Fig 4.10.11](image)

- The length of this schedule is 18, or one time unit less than the other schedule.
- The increased computation time of P2 is more than made up for by being able to transmit a shorter message on the bus.
- If we had not taken communication into account when analyzing total execution time, we could have made the wrong choice of which processes to put on the same processing element.
Data compressor - Alarm Clock - Audio player - Software modem - Digital still camera - Telephone answering machine - Engine control unit – Video accelerator

**DATA COMPRESSOR:**
- Compression is the most important technique to transmit data with minimum amount of data.
- Data compression can be carried out by data compressor.
- Data compressor technique is process of reducing the amount of data needed for storage or transmission of a given piece of information using encoding techniques.
- It is mainly used in graphics, video, sound, text etc.
- It can be classified into two types.
  - Lossy compression
  - Lossless compression

**Lossy compression:**
- It is frequently used for photograph, video and sound files where the loss of some detail is generally unnoticeable.

**Loseless compression:**
- It scans the data for respective sequences or regions and replaces them with a single token.
- Example: ZIP and GIF

**Requirements and Algorithm:**
- In data compressor, **Huffman coding** technique is used.
- To understand how compression code fits into a larger system is more important.
- Fig 5.1 shows a collaboration diagram for the data compression process.

![Fig 5.1 UML collaboration diagram for the data compressor](image)

- The data compressor takes in a sequence of input symbols and then produces a stream of output symbols.
- For example consider the input symbols are one byte in length and the output symbols are variable length, so we have to choose a format in which to deliver the output data.
- Delivering each coded symbol separately is tedious, so have to supply the length of each symbol and use external code to pack them into words.
- Bit-by-bit delivery is almost certainly too slow so the data compressor to pack the coded symbols into an array.
- There is not a one-to-one relationship between the input and output symbols, and we may have to wait for several input symbols before a packed output word comes out.

**Example: Huffman coding for text compression**
- Huffman coding, which makes use of information on the frequency of characters to assign variable-length codes to characters.
- If shorter bit sequences are used to identify more frequent characters, then the length of the total sequence will be reduced.
- In order to be able to decode the incoming bit string, the code characters must have unique prefixes: No code may be a prefix of a longer code for another character.
- Assume that these characters have the following probabilities P of appearance in a message:
We build the code from the bottom up.
After sorting the characters by probability, we create a new symbol by adding a bit.
We then compute the joint probability of finding either one of those characters and re-sort the table.
The result is a tree that we can read top down to find the character codes as shown in fig 5.1.1.

Reading the codes off the tree from the root to the leaves, we obtain the following coding of the characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>P</th>
<th>Character</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.45</td>
<td>d</td>
<td>0.08</td>
</tr>
<tr>
<td>b</td>
<td>0.24</td>
<td>e</td>
<td>0.07</td>
</tr>
<tr>
<td>c</td>
<td>0.11</td>
<td>f</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Once the code has been constructed, the codes can be stored in a table for encoding.
This makes encoding simple, but the encoded bit rate can vary significantly depending on the input character sequence.
On the decoding side, since we do not know a priori the length of a character’s bit sequence, the computation time required to decode a character can vary significantly.

**Requirement:**

<table>
<thead>
<tr>
<th>S.No</th>
<th>Name</th>
<th>Data compression module</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Purpose</td>
<td>Code module for Huffman data compression</td>
</tr>
<tr>
<td>2</td>
<td>Inputs</td>
<td>Encoding table, uncoded byte-size input symbols</td>
</tr>
<tr>
<td>3</td>
<td>Outputs</td>
<td>Packed compressed output symbols</td>
</tr>
<tr>
<td>4</td>
<td>Functions</td>
<td>Huffman coding</td>
</tr>
<tr>
<td>5</td>
<td>Performance</td>
<td>Requires fast performance</td>
</tr>
<tr>
<td>6</td>
<td>Manufacturing cost</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>7</td>
<td>Power</td>
<td>Not Applicable</td>
</tr>
<tr>
<td>8</td>
<td>Physical size and weight</td>
<td>Not Applicable</td>
</tr>
</tbody>
</table>

**Specification:**

- For a fully functional system, we have to provide the following additional behavior.
  - We have to be able to provide the compressor with a new symbol table.
  - We should be able to flush the symbol buffer to cause the system to release all pending symbols that have been partially packed.
- A class description for this refined understanding of the requirements on the module is shown in Fig 5.1.2.
• The class’s buffer and current-bit behaviors keep track of the state of the encoding, and the table attribute provides the current symbol table.

• The class has three methods as follows:
  ➢ **Encode** performs the basic encoding function. It takes in a 1-byte input symbol and returns two values: a boolean showing whether it is returning a full buffer.
  ➢ **New-symbol-table** installs a new symbol table into the object and throws away the current contents of the internal buffer.
  ➢ **Flush** returns the current state of the buffer, including the number of valid bits in the buffer.

• We also need to define classes for the data buffer and the symbol table, are shown in Fig 5.1.3.

**Fig 5.1.2 Definition of the Data-compressor class**

**Fig 5.1.3 Additional class definitions for the data compressor**

• The **data-buffer** will be used to hold both packed symbols and unpacked ones.
  ➢ Data buffer class diagram defines the buffer itself and the length of the buffer. Data types definition is important because the longest encoded symbol is longer than an input symbol.
  ➢ The longest Huffman code for an eight-bit input symbol is 256 bits. The insert function packs a new symbol into the upper bits of the buffer; it also puts the remaining bits in a new buffer if the current buffer is overflowed.

• The **Symbol-table** class indexes the encoded version of each symbol.

• The class defines an access behavior for the table; it also defines a **load** behavior to create a new symbol table.

**Fig 5.1.4 Relationships between classes in the data compressor**

• The relationships between these classes are shown in Fig 5.1.4, a data compressor object includes one buffer and one symbol table.

• Fig 5.1.5 shows a state diagram for the **encode behavior**.
It shows that most of the effort goes into filling the buffers with variable-length symbols.

**Fig 5.1.5 State diagram for encode behavior**

Fig 5.1.6 shows a state diagram for **insert**.

It shows that we must consider two cases, the new symbol does not fill the current buffer or it does.

**Fig 5.1.6 State diagram for insert behavior**

**Program Buffer:**
- Program design for encoder is very simple.
- We can make two ways to make a program design:
  - object-oriented design implementation
  - non-object oriented implementation
- First is the object-oriented implementation C++ implementations most closely mirrors the specification.
- The first step is to design the data buffer.
- The data buffer needs to be as long as the longest symbol.
- The code for data_buffer is relatively complex, and not all of its complexity was reflected in the state diagram of Fig 5.1.6.
- That does not mean the specification was bad, but only that it was written at a higher level of abstraction.
- We have to create code for all the element presented in the module, for sample we can create code for data compressor.

```cpp
typedef char boolean; /* for clarity */
class data_compressor
{
    data_buffer buffer;
    int current_bit;
    symbol_table table;
    public:
    boolean encode(char, data_buffer&);
    void new_symbol_table(symbol_table newtable)
    {
        table = newtable;
        current_bit = 0;
        buffer = empty_buffer;
    }
    int flush(data_buffer& buf)
    {
        int temp = current_bit;
```
```c
buf = buffer;
buffer = empty_buffer;
current_bit = 0;
return temp;
}
data_compressor()
{
    /* C++ constructor */
    
    ~data_compressor()
    {
        /* C++ destructor */
    }
}

OO design in C:
- In C implementation we can support multiple simultaneous data compressors we may not have the luxury of coding the algorithm in C++.
- The C is almost universally supported on embedded processors, the object orientation language such as C++ or Java is not so universal.
- We create a structure that holds the data part of the object as follows:
  ```c
  struct data_compressor_struct
  {
    data_buffer buffer;
    int current_bit;
    sym_table table;
  }
  ```
- The efficiency of the code depends on the instruction and good compiler.
- If we have good compiler it can select the right instructions to efficiently implement the required operations.

Testing:
- Testing is the process to ensure the program module will work correctly.
- Testing can be performed in many different ways.
- One way to test the code is to run it and look at the output without considering how the code is written.
- In this case, symbol table will be used, run some symbols through it, and see whether we get the correct result.
- We should test several different symbol tables.
- We can get the symbol table from outside sources or by writing a small program to generate it ourselves.
- We have to test enough symbols for each symbol table.
- One way to help automate testing is to write a Huffman decoder.
- As shown in Fig 5.1.7, first we can run a set of symbols through the encoder, and then through the decoder.
• Finally make sure that the input and output are the same.

![Diagram of encoder and decoder]

**Fig 5.1.7 A test of the encoder**

• If the input and output are not the same means we have to check both the encoder and decoder to locate the problem.
• Another way to test the code is to examine the code itself and try to identify potential problem areas in code.
• At the time of reading the code itself, we must know where data operations take place and performed properly or not.
• Some condition to be identified to perform different cases on the code such as follows.
  ❚ Is it possible to run past the end of the symbol table?
  ❚ What happens when the next symbol does not fill up the buffer?
  ❚ What happens when the next symbol exactly fills up the buffer?
  ❚ What happens when the next symbol overflows the buffer?
  ❚ Do very long encoded symbols work properly? How about very short ones?
  ❚ Does flush() work properly?
• Testing the internals of code often requires building **scaffolding code**, if the programming language comes with an interpreter, building such scaffolding is easier because we do not have to create a complete executable code for it.
• Finally using data compressor we can reduce the size of memory for data storage.
• It will improve the performance of any kind of systems.

**ALARM CLOCK:**

• We use a microprocessor to read the clock’s buttons and update the time display.

**Requirements:**

![Front panel of the alarm clock]

**Fig 5.2 Front panel of the alarm clock**

• Fig 5.2 shows the front panel design for the alarm clock.
• The time is shown as four digits in 12-h format; we use a light to distinguish between AM and PM.
• We use several buttons to set the clock time and alarm time.
• When we press the hour and minute buttons, we advance the hour and minute, respectively, by one.
When setting the time, we must hold down the set time button while we hit the hour and minute buttons; the set alarm button works in a similar fashion.

- We turn the alarm on and off with the alarm on and alarm off buttons.
- When the alarm is activated, the alarm ready light is on.
- A separate speaker provides the audible alarm.

**Requirements:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Alarm clock.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A 24-h digital clock with a single alarm.</td>
</tr>
<tr>
<td>Inputs</td>
<td>Six push buttons: set time, set alarm, hour, minute, alarm on, alarm off.</td>
</tr>
<tr>
<td>Functions</td>
<td>Default mode: The display shows the current time. PM light is on from noon to midnight. Hour and minute buttons are used to advance time and alarm, respectively. Pressing one of these buttons increments the hour/minute once. Depress set time button: This button is held down while hour/minute buttons are pressed to set time. New time is automatically shown on display. Depress set alarm button: While this button is held down, display shifts to current alarm setting; depressing hour/minute buttons sets alarm value in a manner similar to setting time. Alarm on: puts clock in alarm-on state, causes clock to turn on buzzer when current time reaches alarm time, turns on alarm ready light. Alarm off: turns off buzzer, takes clock out of alarm-on state, turns off alarm ready light.</td>
</tr>
<tr>
<td>Performance</td>
<td>Displays hours and minutes but not seconds. Should be accurate within the accuracy of a typical microprocessor clock signal.</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>Consumer product range. Cost will be dominated by the microprocessor system, not the buttons or display.</td>
</tr>
<tr>
<td>Power</td>
<td>Powered by AC through a standard power supply.</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>Small enough to fit on a nightstand with expected weight for an alarm clock.</td>
</tr>
</tbody>
</table>

**Specification:**

- Fig 5.2.1 shows the Mechanism class that handles the basic clock operation.
- We have three classes that represent physical elements: Lights* for all the digits and lights, Buttons* for all the buttons, and Speaker* for the sound output.
- The Buttons* class can easily be used directly by Mechanism.
- The details of the low-level user interface classes are shown in Fig 5.2.2.
- The Buzzer* class allows the buzzer to be turned off; we will use analog electronics to generate the buzz tone for the speaker.
- The Buttons* class provides read-only access to the current state of the buttons.
- The Lights* class allows us to drive the lights.
Fig 5.2.2 Details of low-level class for the alarm clock

- However, to save pins on the display, Lights* provides signals for only one digit, along with a set of signals to indicate which digit is currently being addressed.
- We generate the display class by scanning the digits periodically.
- The Mechanism class is described in Fig 5.2.3.

Fig 5.2.3 The Mechanism class

- This class keeps track of the current time, the current alarm time, whether the alarm has been turned on, and whether it is currently buzzing.
- The clock shows the time only to the minute, but it keeps internal time to the second.
- The time is kept as discrete digits rather than a single integer to simplify transferring the time to the display.
- The class provides two behaviors, both of which run continuously.
  - First, scan-keyboard is responsible for looking at the inputs and updating the alarm and other functions as requested by the user and the state diagram is shown in fig 5.2.4.
  - Second, update-time keeps the current time accurate and the state diagram is shown in fig 5.2.5.
- The scan-keyboard function is called periodically, frequently enough so that all the user’s button presses are caught by the system.
- Because the keyboard will be scanned several times per second, we do not want to register the same button press several times.
- To make the buttons respond more reasonably, the function computes button activations, it compares the current state of the button to the button’s value on the last scan.
• Once computing the activation values for all the buttons, it looks at the activation combinations and takes the appropriate actions.
• Before exiting, it saves the current button values for computing activations the next time this behavior is executed.

![State diagram for scan-keyboard](image1)

![State diagram for update-time](image2)

• This update-time behavior is activated once per second and must update the seconds clock.
• If it has counted 60 s, it must then update the displayed time; when it does so, it must roll over between
digits and keep track of AM-to-PM and PM-to-AM transitions.
• It sends the updated time to the display object.
• It also compares the time with the alarm setting and sets the alarm buzzing under proper conditions.

System Architecture:
• The software and hardware architectures of a system are always hard to completely separate, but let’s
first consider the software architecture and then its implications on the hardware.
• The system has both periodic and aperiodic components—the current time must obviously be updated
periodically, and the button commands occur occasionally.
• It seems reasonable to have the following two major software components:
  ➢ An interrupt-driven routine can update the current time. The current time will be kept in a variable in
    memory. A timer can be used to interrupt periodically and update the time.
  ➢ A foreground program can poll the buttons and execute their commands. It will read the button values
    and then use simple conditional tests to implement the commands, including setting the current time,
    setting the alarm, and turning off the alarm. Another routine called by the foreground program will turn the
    buzzer on and off based on the alarm time.
• The foreground code will be implemented as a while loop:
  while (TRUE) {
    read_buttons (button_values); /* read inputs */
    process_command (button_values); /* do commands */
    check_alarm( ); /* decide whether to turn on the alarm */
  }
• The loop first reads the buttons using read_buttons( ).
• In addition to reading the current button values from the input device, this routine must preprocess
  the current button values from the input device, so that the user interface code will respond properly.
• The process_command( ) function is responsible for responding to button events.
• The check_alarm( ) function checks the current time against the alarm time and decides when to turn on
  the buzzer.
• This routine is kept separate from the command processing code since the alarm must go on when the
  proper time is reached, independent of the button inputs.
• As shown in Fig 5.2.6, when the button is depressed, a simple edge detection is determined on the button
  input—the button event value is 1 for one sample period when the button is depressed and then goes back to
  0 and does not return to 1 until the button is depressed and then released.

![Fig 5.2.6 Preprocessing button inputs](image)

• We have determined from the software architecture that we will need a timer connected to the CPU.
• We will also need logic to connect the buttons to the CPU bus.
• The final step before starting to write code and build hardware is to draw the state transition graph for
  the clock’s commands.
Component Design and Testing:
- The two major software components, the interrupt handler and the foreground code, can be implemented relatively straight forwardly.
- Since most of the functionality of the interrupt handler is in the interruption process itself, that code is best tested on the microprocessor platform.
- The foreground code can be more easily tested on the PC or workstation used for code development.
- We can create a testbench for this code that generates button depressions to exercise the state machine.
- We will also need to simulate the advancement of the system clock.
- A better testing strategy is to add testing code that updates the clock, perhaps once per four iterations of the foreground while loop.
- The timer will probably be a stock component, so we would then focus on implementing logic to interface to the buttons, display, and buzzer.
- The buttons will require debouncing logic.
- The display will require a register to hold the current display value in order to drive the display elements.

System Integration and Testing:
- Because this system has a small number of components, system integration is relatively easy.
- The software must be checked to ensure that debugging code has been turned off.
- Three types of tests can be performed.
- First, the clock’s accuracy can be checked against a reference clock.
- Second, the commands can be exercised from the buttons.
- Finally, the buzzer’s functionality should be verified.

AUDI0 PLAYERS:
Theory of operation and requirements:
- Audio players are often called MP3 players after the popular audio data format, although a number of audio compression formats have been developed and are in regular use.
- The earliest portable MP3 players were based on compact disc mechanisms.
- Modem MP3 players use either flash memory or disk drives to store music.
- An MP3 player performs three basic functions: audio storage, audio decompression, and user interface.

Audio decompression:
- Although audio compression is computationally intensive, audio decompression is relatively lightweight.
- The incoming bit stream has been encoded using a Huffman style code, which must be decoded.
- Audio compression is a lossy process that relies on perceptual coding.
- It tries to eliminate features that are not easily perceived by the human audio system.

Masking is one perceptual phenomenon that is exploited by perceptual coding.
- Some audio features can also be masked if they occur too close in time after another feature.
- The term MP3 comes from MPEG-I, layer 3; the MP3 spec is part of the MPEG-1 standard.
- That standard defined three layers of audio compression:
  - Layer 1 (MP1) uses a lossless compression of subbands and an optional, simple masking model.
  - Layer 2 (MP2) uses a more advanced masking model.
  - Layer 3 (MP3) performs additional processing to provide lower bit rates.
- The various layers support several different input sampling rates, output bit rates, and modes (mono, stereo, etc.).
- Fig 5.3 gives a block diagram of a Layer 1 encoder.
- The main processing path includes the filter bank and the quantizer /encoder.
The filter bank splits the signal into a set of 32 subbands that are equally spaced in the frequency domain and together cover the entire frequency range of the audio.

Audio signals tend to be more correlated within a narrower band, so splitting into subbands helps the encoder reduce the bit rate.

The quantizer first scales each subband so that it fits within 6 bits of dynamic range, then quantizes based upon the current scale factor for that subband.

The masking model selects the scale factors for the subbands, which can change along with the audio stream.

It is driven by a separate fast Fourier transform (FFT).

The multiplexer at the output of the encoder passes along all the required data.

MPEG data streams are divided into frames.

A frame carries the basic MPEG data, error correction codes, and additional information as shown in Fig 5.3.1.

MPEG audio decoding is a relatively straightforward process as shown in Fig 5.3.2.

After disassembling the data frame, the data are unsealed and inverse quantized to produce sample streams for the subband.

An inverse filter bank then reassembles the subbands into the uncompressed signal.

User interface:

The user interface of an MP3 player is usually kept simple to minimize both the physical size and power consumption of the device.

Many players provide only a simple display and a few buttons.

File system:

The file system of the player generally must be compatible with PCs.

CD/MP3 players used compact discs that had been created on PCs.

Today's players can be plugged into USB ports and treated as disk drives on the host processor.
Name | Audio Player  
---|---  
Purpose | Play audio from files  
Inputs | Flash memory socket, on/off, play/stop, menu up/down  
Outputs | Speaker  
Functions | Display list of files in flash memory, select file to play, play file  
Performance | Sufficient to play audio files at required rate  
Manufacturing cost | Approximately $25  
Power | 1 AAA battery  
Physical size and weight | Approx. 1 in x 2 in, less than 2 oz  

**Specification:**  
- Fig 5.3.3 shows the major classes in the audio player.  
- The File ID class is an abstraction of a file in the flash file system.  
- The controller class provides the method that operates the player.

![Fig 5.3.3 Classes in the audio player](image)

- Fig 5.3.4 shows a state diagram for file display/selection.  
- This specification assumes that all files are in the root directory and that all files are playable audio.

![Fig 5.3.4 State diagram for file display and selection](image)

- Fig 5.3.5 shows the state diagram for audio playback.  
- This state diagram refers to sending the samples to the audio system rather than explicitly sending them because playback and reading the next data frame must be overlapped to ensure continuous operation.
System architecture:

Audio processors:
- The Cirrus CS741O is an audio controller designed for CD/MP3 players as shown in fig 5.3.6.

- It includes two processors.
  - The 32-bit RISC processor is used to perform system control and audio decoding.
  - The 16-bit DSP is used to perform audio effects such as equalization.
- The memory controller can be interfaced to several different types of memory:
  - Flash memory can be used for data or code storage;
  - DRAM can be used as a buffer to handle temporary disruptions of the CD data stream.
- The audio interface unit puts out audio in formats that can be used by A/D converters.
- General-purpose I/O pins can be used to decode buttons, run displays, etc.
- The software architecture of this system is relatively simple.
The only major complication occurs from the requirements for DMA or other method to overlap audio playback and file access.

**File systems:**
- Home music collectors adopted MP3 files as a medium for storing compressed music.
- They passed around collections of MP3 files and playlists.
- These files were often shared as files written on CDs.
- Over time, consumer electronics manufacturers noticed the trend and made players for MP3 files.
- Because this approach was independently practiced by many different users, there are no standards for organizing MP3 files and playlists into directories.
- As a result, MP3 players must be able to navigate arbitrary user hierarchies, be able to find playlist files in any directory, etc.

**Component design and testing:**
- The audio decompression object can be implemented from existing code or created as new software.
- In the case of an audio system that does not conform to a standard, it may be necessary to create an audio compression program to create test files.
- The file system can either implement a known standard such as DOS FAT or can implement a new file system.
- The file system and user interface can be tested independently of the audio decompression system.
- The audio output system should be tested separately from the compression system.
- Testing of audio decompression requires sample audio files.

**System integration and debugging:**
- The most challenging part of system integration and debugging is ensuring that audio plays smoothly and without interruption.
- Any file access and audio output that operate concurrently should be separately tested, ideally using an easily recognizable test signal.
- Simple test signals such as tones will more readily show problems such as missed or delayed samples.

**SOFTWARE MODEM:**
- A software modem or a soft modem is a modem with minimal hardware capacities, designed to use a host computer’s resource to perform most of the task performed by a dedicated hardware in a traditional modem.
- A soft modem is also referred as win modem because the first commercially soft modem mostly worked only with the Microsoft windows operating system.
- As PSTN modem technology advanced, the modulation and encoding scheme will be used.
- This becomes increasingly more complex and thus forcing the hardware used by the modems.
- Hardware modem also increases the complexity and this gives rise to the software modem.

**Theory of Operation and Requirements:**
- Soft modem will use frequency-shift keying (FSK), a technique used in 1200-baud modems.

![Fig 5.4 Frequency-shift keying](image-url)
As shown in Fig 5.4, the FSK scheme transmits sinusoidal tones, with 0 and 1 assigned to different frequencies.
Sinusoidal tones are much better suited to transmission over analog phone lines.
The 01 bit patterns create the chirping sound characteristic of modems.
The FSK scheme used to translate the audio input into a bit stream is illustrated in Fig 5.4.1.

![Fig 5.4.1 The FSK detection scheme](image)

- The analog input is sampled and the resulting stream is sent to two digital FIR filters.
- One filter passes frequencies in the range that represents a 0 and rejects the 1-band frequencies, and the other filter does the converse.
- The outputs of the filters are sent to detectors, which compute the average value of the signal over the past n samples.
- When the energy goes above a threshold value, the appropriate bit is detected.
- We will send data in units of 8-bit bytes.
- The transmitting and receiving modems agree in advance on the length of time during which a bit will be transmitted (otherwise known as the baud rate).

But the transmitter and receiver are physically separated and therefore are not synchronized in any way.
The receiving modem does not know when the transmitter has started to send a byte.
We can avoid this problem by reducing the chances for error by sending the waveforms for a longer time.
The receiving process is illustrated in Fig 5.4.2.

![Fig 5.4.2 Receiving bits in the modem](image)

The receiver will detect the start of a byte by looking for a start bit, which is always 0.
By measuring the length of the start bit, the receiver knows where to look for the start of the first bit.

### Requirements:

<table>
<thead>
<tr>
<th>Name</th>
<th>Modem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>A fixed baud rate frequency-shift keyed modem.</td>
</tr>
<tr>
<td>Inputs</td>
<td>Analog sound input, reset button</td>
</tr>
<tr>
<td>Outputs</td>
<td>Analog sound output, LED bit display</td>
</tr>
<tr>
<td>Functions</td>
<td><strong>Transmitter:</strong> Sends data stored in microprocessor memory in 8-bit bytes. Sends start bit for each byte equal in length to one bit. <strong>Receiver:</strong> Automatically detects bytes and stores results in main memory. Displays currently received bit on LED.</td>
</tr>
</tbody>
</table>
Performance | 1200 baud.
---|---
Manufacturing cost | Dominated by microprocessor and analog I/O.
Power | Powered by AC through a standard power supply.
Physical size and weight | Small and light enough to fit on a desktop

**Specification:**
- The basic classes for the modem are shown in Fig 5.4.3.

![Fig 5.4.3 Class diagram for the modem](image)

**System Architecture:**
- The modem consists of
  - One small subsystem (the interrupt handlers for the samples) and
  - Two major subsystems (transmitter and receiver).
- Two sample interrupt handlers are required, one for input and another for output, but they are very simple.
- The transmitter is simpler, so let’s consider its software architecture first.
- The best way to generate waveforms that retain the proper shape over long intervals is **table lookup**.
- Software oscillators can be used to generate periodic signals.
- Table lookup can be combined with interpolation to generate high-resolution waveforms without excessive memory costs.
- This is more accurate than oscillators because no feedback is involved.
- Fig 5.4.4 shows an analog waveform with sample points and the C code for these samples.

![Fig 5.4.4 Waveform generation by table lookup](image)

- The required number of samples for the modem can be found by experimentation with the analog/digital converter and the sampling code.
- The structure of the receiver is considerably more complex.
  - The filters and detectors of Fig 5.4.1 can be implemented with circular buffers. So it must feed a state machine that recognizes the bits.
  - The recognizer state machine must use a timer to determine when to start and stop computing.
  - To determine the nature of the bit at the proper interval.
  - To detect the start bit and measure it using the counter.
- The hardware architecture is relatively simple.
In addition to the analog/digital and digital/analog converters, a timer is required. The amount of memory required to implement the algorithms is relatively small.

**Component Design and Testing:**
- The transmitter and receiver can be tested on the host platform since the timing-critical code only delivers data samples.
- The transmitter’s output is easy to verify, when the data are plotted.
- Testing in receiver is complex task.
- A testbench can be constructed to feed the receiver code sinusoidal inputs and test its bit recognition rate.
- It is a good idea to test the bit detectors first before testing the complete receiver operation.
- Receiver testing process is more critical in host-based testing if the receiver has library code.
- So the receiver must then be retested when moved to the target system to ensure that it work properly with the library code.
- Next important task is receiver does not run too long and miss its deadline.

**System Integration and Testing:**
- Modem can be tested in two ways such as
  - By having the modem’s transmitter send bits to its receiver, and
  - By connecting two different modems.
- The most important testing method is the second one.
- But single-unit testing, is called loop-back testing.
- It can be applied in telecommunications industry.
- Loop-back can be performed in two ways.
  - First, a shared variable can be used to directly pass data from the transmitter to the receiver.
  - Second, an audio cable can be used to plug the analog output to the analog input.

**Advantages:**
- It can be easily upgradable of newer modem.
- It can be programmed other than a modem, for example it could emulate an answering machine or a signal generator.
- It is less cost and less weight.
- Low power consumption.
- It can be used as a portable modem in laptops and PDAs.

**Disadvantages:**
- In certain stage of usage it will slow down their host computer systems.
- Now a day with modern CPUs and better drivers, their most serious drawback is their operating system and machine dependent. They cannot be used on other operating system.
- They consume some CPU cycle on the computer to which they are attached, which can slow down the application software on the computer.

**DIGITAL STILL CAMERA (DSC):**
- Video cameras share some similarities with DSCs but are different in several ways, most notably their emphasis on streaming media.

**Theory of operation and requirements:**
- A modern digital camera performs a great many steps;
  - Determine the exposure and focus.
  - Capture image.
  - Develop the image.
Compress the image.
Generate and store the image as a file.

- In addition to actually taking the photo, the camera must perform several other important operations, often simultaneously with the picture-taking process.
- It must, for example, update the camera's electronic display.
- It must also listen for button presses from the user, which may command operations that modify the current operation of the camera.
- The camera should also provide a browser with which the user can review the stored images.

**Imaging terminology:**
- An image is divided into pixels; a pixel's brightness is often referred to as its *luminance*; a color pixel's brightness in a particular color is known as *chrominance*.

**Imaging algorithms:**
- The camera can use the image sensor data to drive the exposure-setting process.
- Two major types of image sensors are used in modern cameras: *charged coupled devices (CCDs)* and *CMOS*.
- Developing the image involves both putting the image in usable form and improving its quality.
- Several algorithms are possible but all involve starting with a test exposure and using some search algorithm to select the final exposure.
- The camera may evaluate some function of several points in the image.
- It may also evaluate the image's histogram.
- The histogram is composed by sorting the pixels into bins by luminance; 256 bins is a common choice for the resolution of the histogram.
- The histogram gives us more information than does a single average.
- Three major approaches are used to determine focus: active range finding, phase detection, contrast detection.
  - **Active range finding** uses a pulse that is sent out and the time-of-flight of the reflected and returned pulse is measured to determine distance.
    - Ultrasound was used in one early autofocus system, the Polaroid SX-70, but today infrared pulses are more commonly used.
  - **Phase detection** compares light from opposite sides of the lens, creating an optical rangefinder.
  - **Contrast detection** relies on the fact that out-of-focused edges do not display the sharp changes in luminance of in-focus edges.
- Most image sensors capture color images using a *color filter array*, color filters that cover a single pixel.
- The filters usually capture one of the primary colors-red, green, and blue and are arranged in a two-dimensional pattern.
- The first such color filter array was proposed by Bayer.
  - His pattern is still widely used and known as the **Bayer pattern**.
  - As shown in Fig 5.5, the Bayer pattern is a 2 X 2 array with two green, one blue, and one red pixels.
  - More green pixels are used because the eye is most sensitive to green, which Bayer viewed as a simple luminance signal.
  - Because each image sensor pixel captured only one of the primaries, the other two primaries must be interpolated for each pixel.
  - This process is known as **Bayer pattern interpolation** or **demosaicing**.
  - The simplest interpolation algorithm is a simple average, which can also be used as a low-pass filter.
Fig 5.5 A color filter array arranged in a Bayer pattern.

- For example, we can interpolate the missing values from the green pixel G2.1 as:
  \[
  \text{G2.1} = (\text{G1.0} + \text{G1.1})/2
  \]
- We can use more information to interpolate missing green values.
- For example, the green value associated with red pixel R1.1 can be interpolated from the four nearest green pixels:
  \[
  (G1.1 + G2.1 + G2.2 + G1.4)/4
  \]
- However, this simple interpolation algorithm introduces color fringes at edges.
- Development also determines and corrects for color temperature in the scene.
- A variety of algorithms exist to determine the color temperature.
- A set of chrominance histograms is often used to judge color temperature.
- Once a color correction is determined, it can be applied to the pixels in the image.
- Many cameras also apply sharpening algorithms to reduce some of the effects of pixellation in digital image capture.
- A variety of sharpening algorithms are used to apply filters to sets of adjacent pixels.
- Some cameras offer a RAW mode of image capture.
- RAW capture allows the user to develop the image manually and offline using sophisticated algorithms and programs.

**Image compression:**

- Compression reduces the amount of storage required for the image.
- Some **lossless compression** methods are used that do not throw away information from the image.
- However, most images are stored using **lossy compression**.
- A lossy compression algorithm throws away information in the image, so that the decompression process cannot reproduce an exact copy of the original image.
- A lossy compression is used to reduce the storage space required for an image without noticeably affecting image quality.
- The most common family of compression algorithms is JPEG (**JPEG stands for Joint Photographic Experts Group**.)
- The JPEG standard was extended as JPEG2000 but classic JPEG is still widely used.

---

Fig 5.5.1 The typical JPEG compression process.
As shown in Fig 5.5.1, the typical compression process used for JPEG images has five main steps:

- Color space conversion.
- Color down sampling.
- Block-based discrete cosine transform (DCT).
- Quantization.
- Entropy coding.

- The first step puts the color image into a form that allows optimizations less likely to reduce image quality.
- Color can be represented by a number of color spaces or combinations of colors that, when combined, form the full range of colors.
- We saw the red/green/blue (RGB) color space in the color filter array.
- For compression, we convert to the Y′C_R C_B color space: Y′ is a luminance channel, C_R is a red channel, and C_B is a blue channel.

- The conversion between these two color spaces is defined by the JFIF standard:

\[
\begin{align*}
Y' &= + 0.6931* R + 0.5460* G + 0.1304* B \\
C_R &= + 0.1422* R - 0.2877* G - 0.1972* B \\
C_B &= + 0.0693* R + 0.5155* G - 0.3863* B \\
\end{align*}
\]

- Once in Y′C_R C_B form, the C_R and C_B is generally reduced by down sampling.
- The 4:2:2 method down samples both C_R and C_B to half their normal resolution only in the horizontal direction.
- The 4:2:0 method down samples both C_R and C_B both horizontally and vertically.
- Downsampling reduces the amount of data required and is justified by the human visual system's lower acuity in chrominance.

- The three color channels, Y′, C_R, and C_B, are processed separately.
- The color channels are next separately broken into 8 X 8 blocks.
- The discrete cosine transform is applied to each block.
- DCT is a frequency transform that produces an 8 X 8 block of transform coefficients.
- It is reversible, so that the original values can be reconstructed from the transform coefficients.
- DCT does not itself reduce the amount of information in a block.
- Many highly optimized algorithms exist to compute the DCT, and in particular an 8 X 8 DCT.
- Specifically, because the DCT breaks up the block according to spatial frequencies, quantization often reduces the high frequency content of the block.
- This strategy tends to significantly reduce the data set size during compression while causing less noticeable visual artifacts.
- The quantization is defined by an 8 X 8 quantization matrix Q.
- A DCT coefficient G_{i,j} is quantized to a value B_{i,j} using the Q_{i,j} value from the quantization matrix:

\[
G_{i,j} = \frac{B_{i,j}}{Q_{i,j}}
\]

- The JPEG standard allows for different quantization matrices but gives a typical matrix that is widely used.
<table>
<thead>
<tr>
<th>16</th>
<th>11</th>
<th>10</th>
<th>16</th>
<th>24</th>
<th>40</th>
<th>51</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>12</td>
<td>14</td>
<td>19</td>
<td>26</td>
<td>58</td>
<td>60</td>
<td>55</td>
</tr>
<tr>
<td>14</td>
<td>13</td>
<td>16</td>
<td>24</td>
<td>40</td>
<td>57</td>
<td>69</td>
<td>56</td>
</tr>
<tr>
<td>14</td>
<td>17</td>
<td>22</td>
<td>29</td>
<td>51</td>
<td>87</td>
<td>80</td>
<td>62</td>
</tr>
<tr>
<td>18</td>
<td>22</td>
<td>37</td>
<td>56</td>
<td>68</td>
<td>109</td>
<td>103</td>
<td>77</td>
</tr>
<tr>
<td>24</td>
<td>35</td>
<td>55</td>
<td>64</td>
<td>81</td>
<td>104</td>
<td>113</td>
<td>92</td>
</tr>
<tr>
<td>49</td>
<td>64</td>
<td>78</td>
<td>87</td>
<td>103</td>
<td>121</td>
<td>120</td>
<td>101</td>
</tr>
<tr>
<td>72</td>
<td>92</td>
<td>95</td>
<td>98</td>
<td>112</td>
<td>100</td>
<td>103</td>
<td>99</td>
</tr>
</tbody>
</table>

Fig 5.5.2
• This matrix tends to zeroes in the lower-right coefficients.
• Higher spatial frequencies are represented by the coefficients in the lower and right parts of the matrix.
• Putting these to zero eliminates some fine detail from the block.
• Quantization does not directly provide for a smaller representation of the image.
• Entropy coding (lossless encoding) recodes the quantized blocks in a form that requires fewer bits.
• Example: Huffman coding.
• The encoding can be represented as a table that maps a fixed number of bits into a variable number of bits.
• This step encodes the difference between the current coefficient and the previous coefficient, not the coefficient itself.
• Several different styles of encoding are possible: baseline sequential codes one block at a time; baseline progressive encodes corresponding coefficients of every block.
• Coefficients are read from the coefficient matrix in the zig-zag pattern shown in Fig 5.5.3.

![Fig 5.5.3 Zig-zag pattern for reading coefficients.](image)

• This pattern reads along diagonals from the upper left to the lower right, which corresponds to reading from lowest spatial frequency to highest spatial frequency.
• If fine detail is reduced equally in both horizontal and vertical dimensions, then zero coefficients are also arranged diagonally.
• The zig-zag pattern increases the length of sequences of zero coefficients in such cases.
• Long strings of zeroes can be coded in a very small number of bits.

**Requirements:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Digital still camera</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Digital still camera with JPEG compression</td>
</tr>
<tr>
<td>Inputs</td>
<td>Image sensor, shutter button</td>
</tr>
<tr>
<td>Outputs</td>
<td>Display, flash memory</td>
</tr>
<tr>
<td>Functions</td>
<td>Determine exposure and focus, capture image, perform Bayer pattern interpolation, JPEG compression, store in flash file system</td>
</tr>
<tr>
<td>Performance</td>
<td>Take one picture in 2 sec.</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>Approximately $75</td>
</tr>
<tr>
<td>Power</td>
<td>Two AA batteries</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>Approx 4 in x 4 in x 1 in, less than 4 ounces.</td>
</tr>
</tbody>
</table>

**Specification:**

**File formats:**
• The Tagged Image File Format (TIFF) is often used to store uncompressed images, although it also supports several compression methods as well.
- Baseline TIFF specifies a basic format that also provides flexibility on image size, bits per pixel, compression, and other aspects of image storage.
- The JPEG standard itself allows a number of options that can be followed in various combinations.
- A set of operations used to generate a compressed image is known as a **process**.
- The JFIF standard is a widely used file interchange format.
- The **Exchangeable Image File Format (EXIF)** standard is widely used to further extend the information stored in an image file.
- As shown in Fig 5.5.4, an EXIF file holds several types of data:

![EXIF File Structure](image)

**Fig 5.5.4 Structure of an EXIF file**

- The **metadata** section provides a wide range of information: date, time, location, and so on.
- The metadata is defined as attribute/value pairs.
- An EXIF file need not contain all possible attributes.
- A **thumbnail** is a smaller version of a file used for quick display.
- **JPEG compression** data includes tables, such as entropy coding and quantization tables, used to encode the image.
- The compressed JPEG image itself takes up the bulk of the space in the file.
- The entire image storage process is defined by yet another standard, the **Design rule for Camera File (DCF) standard**.
  - DCF specifies three major steps: JPEG compression, EXIF file generation, and DOS FAT image storage.
  - DCF specifies a number of aspects of file storage:
    - The DCF image root directory is kept in the root directory and is named DCIM (for "digital camera images").
    - The directories within DCIM have names of eight characters, the first three of which are numbers between 100 and 999, giving the directory number.
    - The remaining five characters are required to be upper-case alphanumeric.
    - File names in DCF are eight characters long. The first four characters are uppercase alphanumeric, followed by a four digit number between 0001 and 9999.
    - **Basic files** in DCF are in EXIF version 2 format. The standard specifies a number of properties of these EXIF files.
  - The Digital Print Order Format (DPOF) standard provides a standard way for camera users to order prints of selected photographs.
  - Print orders can be captured in the camera, a home computer, or other devices and transmitted to a photo finisher or printer.

**Camera operating modes**:
- Two basic operations are fundamental: display a live view of the current image and capture an image.
- Fig 5.5.5 shows a state diagram for the display.
• In normal operation, the camera repeatedly displays the latest image from the image sensor.

Fig 5.5.5 State diagram for display operation

• After an image is captured, it is briefly shown on the display.
• Fig 5.5.6 shows a state diagram for the picture-taking process.

For 5.5.6 State diagram for picture taking

• Depressing the shutter button half-way signals the camera to evaluate exposure and focus.
• Once the shutter is fully depressed, the image is captured, developed, compressed, and stored.

System Architecture:
• It uses a controller to provide the basic sequencing for picture taking and camera operation.
• The controller calls on a number of other units to perform the various steps in each process.

Fig 5.5.7 Basic classes in the digital still camera
• Fig 5.5.7 shows the controller class implements the state diagrams for camera operation.
• The buttons and display classes provide an abstraction of the physical user interface.
• The image sensor abstracts the operation of the sensor.
• The picture developer provides algorithms for mosaicing, sharpening, etc.
• The compression unit generates compressed image data.
• The file generator takes care of aspects of the file generation beyond compression.
• The file system performs basic file functions.
• Cameras may also provide other communication ports such as USB or Firewire.
• Figure 5.5.8 shows a typical block diagram for digital still camera hardware.

![Block Diagram](image)

**Fig 5.5.8 Computing platform for a digital still camera**

- While some cameras will use the same processor for both system control and image processing, many cameras rely on separate processors for these two tasks.
- The DSP may be programmable or custom hardware.
- A sequence diagram for taking a photo is shown in Fig 5.5.9.

![Sequence Diagram](image)

**Fig 5.5.9 Sequence diagram for taking a picture with a digital still camera.**
• This sequence diagram maps the basic operations onto the units in the hardware architecture.
• The design of buffering is very important in a digital still camera.
• Buffering affects the rate at which pictures can be taken, the energy consumed, and the cost of the camera.
• The image exists in several versions at different points in the process: the raw image from the image sensor; the developed image; the compressed image data; and the file.
• Most of this data is buffered in system RAM.
• However, displays may have their own memory to ensure adequate performance.

Component design and testing:
• Components based on standards, such as JPEG or FAT, may be implemented using modules developed elsewhere.
• JPEG compression in particular may be implemented with special-purpose hardware.
• Some digital still camera engines use hardware units that produce a complete JFIF file from an image.
• The multiple buffering points in the picture taking process can help to simplify testing.
• Test file inputs can be introduced into the buffer by test scaffolding, then run with results in the output buffer checked against reference output.

System integration and testing:
• Buffers help to simplify system integration, although care must be taken to ensure that the buffers do not overlap in main memory.
• Some tests can be performed by substituting pixel value streams for the sensor data.
• Final tests should make use of a target image so that qualities such as sharpness and color fidelity can be judged.

TELEPHONE ANSWERING MACHINE:
• The system will store messages in digital form rather than on an analog tape.
• A simple algorithm is used to compress the voice data, so that we can make more efficient use of the limited amount of available memory.

Theory of Operation and Requirements:
• The compression scheme we will use is known as Adaptive Differential Pulse Code Modulation (ADPCM) is illustrated in Fig 5.6.
• This technique is relatively simple but can yield 2X compression ratios on voice data.

![ADPCM Coding Scheme](image)

**Fig 5.6 The ADPCM coding scheme**
• Unlike traditional sampling, in which each sample shows the magnitude of the signal at a particular time, ADPCM encodes changes in the signal.
• The samples are expressed in a coding alphabet, whose values are in a relatively small range that spans both negative and positive values.
• In this case, the value range is {-3,-2,-1, 1, 2, 3}.  

28
• Each sample is used to predict the value of the signal at the current instant from the previous value.
• At each point in time, the sample is chosen such that the error between the predicted value and the actual signal value is minimized.
• An ADPCM compression system, including an encoder and decoder, is shown in Fig 5.6.1.

![ADPCM compression system diagram]

**Fig 5.6.1 An ADPCM compression system**

• The encoder is more complex, but both the encoder and decoder use an integrator to reconstruct the waveform from the samples.
• The encoder compares the incoming waveform to the predicted waveform.
• The quantizer encodes this difference as the best predictor of the next waveform value.
• The inverse quantizer allows us to map bit-level symbols onto real numerical values.
• The decoder simply uses an inverse quantizer and integrator to turn the differential samples into the waveform.
• The answering machine will ultimately be connected to a telephone subscriber line.
• At the other end of the subscriber line is the **central office**.
• All information is carried on the phone line in analog form over a pair of wires.
• In addition to analog/digital and digital/analog converters to send and receive voice data, we need to sense two other characteristics of the line.

**Ringing:**
• The central office sends a ringing signal to the telephone when a call is waiting.
• The ringing signal is in fact a 90V RMS sinusoid, but we can use analog circuitry to produce 0 for no ringing and 1 for ringing.

**Off-hook:** means the telephone in use.
**On-hook:** means the telephone is not in use.

**Requirements:**
• **Out Going message (OGM)term** is used to refer the message recorded by the owner of the machine and played at the start of every phone call.

<table>
<thead>
<tr>
<th>Name</th>
<th>Digital telephone answering machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Telephone answering machine with digital memory, using speech compression.</td>
</tr>
<tr>
<td>Inputs</td>
<td><strong>Telephone:</strong> voice samples, ring indicator.</td>
</tr>
<tr>
<td></td>
<td><strong>User interface:</strong> microphone, play messages button, record OGM button.</td>
</tr>
<tr>
<td>Outputs</td>
<td><strong>Telephone:</strong> voice samples, on-hook/off-hook command.</td>
</tr>
<tr>
<td></td>
<td><strong>User interface:</strong> speaker, # messages indicator, message light.</td>
</tr>
</tbody>
</table>
**Functions**

**Default mode:** When machine receives ring indicator, it signals off-hook, plays the OGM, and then records the incoming message. Maximum recording length for incoming message is 30 s, at which time the machine hangs up. If the machine runs out of memory, the OGM is played and the machine then hangs up without recording.

**Playback mode:** When the play button is depressed, the machine plays all messages. If the play button is depressed again within five seconds, the messages are played again. Messages are erased after playback.

**OGM editing mode:** When the user hits the record OGM button, the machine records an OGM of up to 10 s. When the user holds down the record OGM button and hits the play button, the OGM is played back.

**Performance**

Should be able to record about 30 min of total voice, including incoming and OGMs. Voice data are sampled at the standard telephone rate of 8 kHz.

**Manufacturing cost**

Consumer product range: approximately $50.

**Power**

Powered by AC through a standard power supply

**Physical size and weight**

Comparable in size and weight to a desk telephone

**Specification:**

![Class diagram for answering machine](image)

**Fig 5.6.2 Class diagram for the answering machine**

- In addition to the classes that perform the major functions, we also use classes to describe the incoming and OGMs.
- The definitions of the physical interface classes are shown in Fig 5.6.3.

![Physical class interfaces for answering machine](image)

**Fig 5.6.3 Physical class interfaces for the answering machine**

- The buttons and lights simply provide attributes for their input and output values.
- The phone line, microphone, and speaker are given behaviors that let us sample their current values.
- The message classes are defined in Fig 5.6.4.
Since incoming and OGM types share many characteristics, we derive both from a more fundamental message type.

The major operational classes—Controls, Record, and Playback—are defined in Fig 5.6.5.

The Controls class provides an operate() behavior that oversees the user-level operations.

The Record and Playback classes provide behaviors that handle writing and reading sample sequences.

The state diagram for the Controls activate behavior is shown in Fig 5.6.6.

Most of the user activities are relatively straightforward.

The most complex is answering an incoming call.

State diagrams for record-msg and playback-msg are shown in Fig 5.6.7.

We have parameterized the specification for record-msg so that it can be used either from the phone line or from the microphone.
• This requires parameterizing the source itself and the termination condition.

![State diagrams](image)

**Fig 5.6.7 State diagrams for the record-msg and playback-msg behaviors**

**System Architecture:**
- The machine consists of two major subsystems from the user’s point of view: the user interface and the telephone interface.
- The user and telephone interfaces both appear internally as I/O devices on the CPU bus with the main memory serving as the storage for the messages.
- The software splits into the following seven major pieces:
  - The **front panel module** handles the buttons and lights.
  - The **speaker module** handles sending data to the user’s speaker.
  - The **telephone line module** handles off-hook detection and on-hook commands.
  - The **telephone input and output modules** handle receiving samples from and sending samples to the telephone line.
  - The **compression module** compresses data and stores it in memory.
  - The **decompression module** uncompresses data and sends it to the speaker module.
- We can determine the execution model for these modules based on the rates at which they must work and the ways in which they communicate.
- The front panel and telephone line modules must regularly test the buttons and phone line, but this can be done at a fairly low rate.
- As seen below, they can therefore run as polled processes in the software’s main loop.
  ```c
  while (TRUE) {
    check_phone_line();
    run_front_panel();
  }
  ```
- The speaker and phone input and output modules must run at higher, regular rates and are natural candidates for interrupt processing.
- The compression and decompression modules run at the same rate as the speaker and telephone I/O modules, but they are not directly connected to devices.
- The hardware architecture is straightforward and illustrated in Fig 5.6.8.
- The speaker and telephone I/O devices appear as standard A/D and D/A converters.
- The telephone line appears as a one-bit input device (ring detect) and a one bit output device.
- The compressed data are kept in main memory.
Component Design and Testing:
- Performance analysis is important in this case because we want to ensure that we don’t spend so much time compressing that we miss voice samples.
- In a real consumer product, we would carefully design the code so that we could use the slowest, cheapest possible CPU that would still perform the required processing in the available time between samples.
- In this case, we will choose the microprocessor in advance for simplicity and simply ensure that all the deadlines are met.
- An important class of problems that should be adequately tested is memory overflow.
- The system can run out of memory at any time, not just between messages.
- The modules should be tested to ensure that they do reasonable things when all the available memory is used up.

System Integration and Testing:
- We can test partial integrations of the software on our host platform.
- Final testing with real voice data must wait until the application is moved to the target platform.
- You can build a telephone line simulator to test the hardware independently of a real telephone line.
- The phone line simulator consists of A/D and D/A converters plus a speaker and microphone for voice data, an LED for off-hook/on-hook indication, and a button for ring generation.
- The telephone line interface can easily be adapted to connect to these components, and for purposes of testing the answering machine the simulator behaves identically to the real phone line.

ENGINE CONTROL UNIT:
- The Engine control unit controls the operation of a fuel-injected engine based on several measurements taken from the running engine.

Theory of operation and requirements:
- As shown in Fig 5.7, the throttle is the command input.

The engine measures throttle, RPM, intake air volume, and other variables.
• The engine controller computes injector pulse width and spark.

**Requirements:**

<table>
<thead>
<tr>
<th>Name</th>
<th>ECU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Engine controller for fuel-injected engine</td>
</tr>
<tr>
<td>Inputs</td>
<td>Throttle, RPM, intake air volume, intake manifold pressure</td>
</tr>
<tr>
<td>Outputs</td>
<td>Injector pulse width, spark advance angle</td>
</tr>
<tr>
<td>Functions</td>
<td>Compute injector pulse width and spark advance angle as function of throttle, RPM, intake air volume, intake manifold pressure</td>
</tr>
<tr>
<td>Performance</td>
<td>Injector pulse updated at 2-ms period, spark advance angle updated at 1-ms period</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>Approximately $50</td>
</tr>
<tr>
<td>Power</td>
<td>Powered by engine generator</td>
</tr>
<tr>
<td>Physical size and weight</td>
<td>Approx 4 in x4 in, less than 1 pound.</td>
</tr>
</tbody>
</table>

**Specification:**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Variable name</th>
<th>In/out</th>
<th>Update period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throttle</td>
<td>T</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>RPM</td>
<td>NE</td>
<td>input</td>
<td>2</td>
</tr>
<tr>
<td>Intake air volume</td>
<td>VS</td>
<td>input</td>
<td>25</td>
</tr>
<tr>
<td>Injector pulse width</td>
<td>PW</td>
<td>output</td>
<td>2</td>
</tr>
<tr>
<td>Spark advance angle</td>
<td>S</td>
<td>output</td>
<td>1</td>
</tr>
<tr>
<td>Intake air temperature</td>
<td>THA</td>
<td>input</td>
<td>500</td>
</tr>
<tr>
<td>Exhaust oxygen</td>
<td>OX</td>
<td>input</td>
<td>25</td>
</tr>
<tr>
<td>Battery voltage</td>
<td>+B</td>
<td>input</td>
<td>4</td>
</tr>
</tbody>
</table>

**Fig 5.7.1 Periods for data in the engine controller**

• Fig 5.7.1 shows the update periods for the different signals.
• We will use ΔNE and ΔT to represent the change in RPM and throttle position, respectively.
• Our controller computes two output signals, injector pulse width PW and spark advance angle S.
• It first computes initial values for these variables:

\[
\Delta = \Delta \times \Delta \times \Delta \quad \ldots (1)
\]

\[
\Delta = \Delta \times \Delta \Delta \quad \ldots (2)
\]

• The controller then applies corrections to these initial values:
  ➢ As the intake air temperature (THA) increases during engine warm-up, the controller reduces the injection duration.
  ➢ As the throttle opens, the controller temporarily increases the injection frequency.
  ➢ The controller adjusts duration up or down based upon readings from the exhaust oxygen sensor (OX).
  ➢ The injection duration is increased as the battery voltage (+ B) drops.

**System architecture:**

• Fig 5.7.2 shows the class diagram for the engine controller.
• The two major processes, pulse-width and advance-angle, compute the control parameters for the spark plugs and injectors.
• The control parameters rely on changes in some of the input signals.
• We will use the physical sensor classes to compute these values.
• Each change must be updated at the variable's sampling rate.
• The update process is simplified by performing it in a task that runs at the required update rate.
Fig 5.7.2 diagram for the engine controller

- Fig 5.7.3 shows the state diagram for throttle sensing, which saves both the current value and change in value of the throttle.

Fig 5.7.3 State diagram for throttle position sensing

- We can use similar control flow to compute changes to the other variables.
- Fig 5.7.4 shows the state diagram for injector pulse width and Fig 5.7.5 shows the state diagram for spark advance angle.
- In each case, the value is computed in two stages, first an initial value followed by a correction.

Fig 5.7.4 State diagram for injector pulse width

- The pulse-width and advance-angle processes do not, however, generate the waveforms to drive the spark and injector waveforms.
• These waveforms must be carefully timed to the engine's current state.
• Each spark plug and injector must fire at exactly the right time in the engine cycle, taking into account the engine's current speed as well as the control parameters.
• Some engine controller platforms provide hardware units that generate high-rate, changing waveforms.
• **Example:** MPC5602D is a PowerPC processor.
• The enhanced modular IO subsystem (eMIOS) provides 28 input and output channels controlled by timers.
• Each channel can perform a variety of functions.
• The output pulse width and frequency modulation buffered mode (OPWFMB) will automatically generate a waveform whose period and duty cycle can be varied by writing registers in the eMIOS.
• The details of the waveform timing are then handled by the output channel hardware.
• Because these objects must be updated at different rates, their execution will be controlled by an RTOS.
• Depending on the RTOS latency, we can separate the I/O functions into interrupt service handlers and threads.

**Component design and testing:**
• The various tasks must be coded to satisfy the requirements of RTOS processes.
• The RTOS initialization phase is used to set up the task periods.
• Because some of the output variables depend on changes in state, these tasks should be tested with multiple input variable sequences to ensure that both the basic and adjustment calculations are performed correctly.
• The Society of Automotive Engineers (SAE) has several standards for automotive software: J2632 for coding practices for C code, J2516 for software development lifecycle, J2640 for software design requirements, J2734 for software verification and validation.

**System integration and testing:**
• Engines generate huge amounts of electrical noise that can cripple digital electronics.
• They also operate over very wide temperature ranges: hot during engine operation, potentially very cold before the engine is started.
• Any testing performed on an actual engine must be conducted using an engine controller that has been designed to withstand the harsh environment of the engine compartment.

**VIDEO ACCELERATOR:**

![Diagram](image)

**Fig 5.8 Block diagram of MPEG-2 compression algorithm**
• Fig 5.8 shows the block diagram for MPEG-2 video compression, forms the basis for U.S. HDTV broadcasting.
• This compression uses several component algorithms together in a feedback loop to further improve image quality.
• As in still image compression, the DCT of a block of pixels, is quantized for lossy compression and then subjected to lossless variable-length coding to further reduce the number of bits required to represent the block.

**Motion-based coding:**
• However, JPEG-style compression alone does not reduce video bandwidth enough for many applications.
• MPEG uses motion to encode one frame in terms of another.
• Rather than send each frame separately, as in motion JPEG, some frames are sent as modified forms of other frames using a technique known as **block motion estimation**.
• During encoding, the frame is divided into **macroblocks**.
• Macroblocks from one frame are identified in other frames using correlation.
• The frame can then be encoded using the vector that describes the motion of the macroblock from one frame to another without explicitly transmitting all of the pixels.
• The encoder uses the encoding information to recreate the lossily-encoded picture, compares it to the original frame, and generates an error signal that can be used by the receiver to fix smaller errors.
• The decoder must keep some recently decoded frames in memory so that it can retrieve the pixel values of macroblocks.
• This internal memory saves a great deal of transmission and storage bandwidth.
• The concept of block motion estimation is illustrated in Fig 5.8.1.

![Fig 5.8.1 Block motion estimation](image)

• The goal is to perform a two-dimensional correlation to find the best match between regions in the two frames.
• We divide the current frame into 16 x16 macroblocks.
• For every macroblock in the frame, we want to find the region in the previous frame that most closely matches the macroblock.
• Searching over the entire previous frame would be too expensive, so we usually limit the search to a given area, centered around the macroblock and larger than the macroblock.
• We measure similarity using the following sum-of-differences measure:

\[
\sum_{i,j} | f(i,j) - \hat{f}(i,j) | = \sum_{i,j} (| f(i,j) - \hat{f}(i,j) |)
\]
\[ 1 \leq \phi \leq \]
- Where \( M(i, j) \) is the intensity of the macroblock at pixel \( i, j \), \( S(i, j) \) is the intensity of the search region, \( n \) is the size of the macroblock in one dimension, and \( (\theta_\phi, \phi_\phi) \) is the offset between the macroblock and search region.
- We choose the macroblock position relative to the search area that gives us the smallest value for this metric.
- The offset at this chosen position describes a vector from the search area center to the macroblock’s center that is called the **motion vector**.

**Algorithm and Requirements:**
- More advanced algorithms are generally used in practice but we choose full motion search here to concentrate on some basic issues in relationship between the accelerator and the rest of the system.
- A good way to describe the algorithm is in C.
- Some basic parameters of the algorithm are illustrated in Fig 5.8.2

![Fig 5.8.2 Block motion search parameters](image)

**Fig 5.8.2 Block motion search parameters**
- Here is the C code for a single search, which assumes that the search region does not extend past the boundary of the frame.
  ```c
  bestx = 0; besty = 0; /* initialize best location-none yet */
  bestsad = MAXSAD; /* best sum-of-difference thus far */
  for (ox = -SEARCHSIZE; ox < SEARCHSIZE; ox++)
    {
      /* x search ordinate */
      for (oy = -SEARCHSIZE; oy < SEARCHSIZE; oy++)
        {
          /* y search ordinate */
          int result = 0;
          for (i = 0; i < MBsize; i++)
            {
              for (j = 0; j < MBsize; j++)
                {
                  result = result + iabs(mb[i][j] - search[i - ox + XCENTER] [j - oy + YCENTER]);
                }
            }
          if (result <= bestsad)
            {
              bestx = ox;
              besty = oy;
              bestsad = result;
            }
        }
    }
  ```
The arithmetic on each pixel is simple, but we have to process a lot of pixels.

- If MBSIZE is 16 and SEARCHSIZE is 8, and remembering that the search distance in each dimension is $8 + 1 + 8$, then we must perform

$$\text{difference operations} = 16 \times 16 \times 17 \times 17 = 73,984$$

- Adjacent blocks have overlapping search areas, so we will try to avoid reloading pixels we already have.
- One relatively low-resolution standard video format, common intermediate format (CIF), has a frame size of 352×288, which gives an array of 22×18 macroblocks.
- If we want to encode video, we would have to perform motion estimation on every macroblock of most frames.

**Requirements:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Block motion estimator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Perform block motion estimation within a PC system</td>
</tr>
<tr>
<td>Inputs</td>
<td>Macro blocks and search areas</td>
</tr>
<tr>
<td>Outputs</td>
<td>Motion vectors</td>
</tr>
<tr>
<td>Functions</td>
<td>Compute motion vectors using full search algorithms</td>
</tr>
<tr>
<td>Performance</td>
<td>As fast as we can get</td>
</tr>
<tr>
<td>Manufacturing cost</td>
<td>Hundreds of dollars</td>
</tr>
<tr>
<td>Power</td>
<td>Powered by PC power supply</td>
</tr>
<tr>
<td>Physical size</td>
<td>Packaged as PCI card for PC</td>
</tr>
</tbody>
</table>

**Specification:**

- Fig 5.8.3 defines some classes that describe basic data types in the system: the motion vector, the macroblock, and the search area.

**Fig 5.8.3 Classes describing basic data types in the video accelerator.**

- These definitions are straightforward.
- Because the behavior is simple, we need to define only two classes to describe it: the accelerator itself and the PC.

**Fig 5.8.4 Basic classes for the video accelerator**
• These classes are shown in Fig 5.8.4.
• The PC makes its memory accessible to the accelerator.
• The accelerator provides a behavior compute-mv( ) that performs the block motion estimation algorithm.
• Fig 5.8.5 shows a sequence diagram that describes the operation of compute-mv( ).

![Sequence diagram for the video accelerator](image)

**Fig 5.8.5 Sequence diagram for the video accelerator**

• After initiating the behavior, the accelerator reads the search area and macroblock from the PC; after computing the motion vector, it returns it to the PC.

**Architecture:**
• The accelerator will be implemented in an FPGA on a card connected to a PC’s PCI slot.
• The architecture for the accelerator requires some thought because of the large amount of data required by the algorithm.
• The macroblock has $16 \times 16 = 256$; the search area has $(8+8+1+8+8)^2 = 1089$ pixels.
• The FPGA probably will not have enough memory to hold 1,089 8-bit values.
• We have to use a memory external to the FPGA but on the accelerator board to hold the pixels.

![An architecture for the motion estimation accelerator](image)

**Fig 5.8.6 An architecture for the motion estimation accelerator**

• As shown in Fig 5.8.6, the machine has two memories, one for the macroblock and another for the search memories.
- It has 16 PEs that perform the difference calculation on a pair of pixels; the comparator sums them up and selects the best value to find the motion vector.
- This architecture can be used to implement algorithms other than a full search by changing the address generation and control.
- Depending on the number of different motion estimation algorithms that you want to execute on the machine, the networks connecting the memories to the PEs may also be simplified.
- Fig 5.8.7 shows how we can schedule the transfer of pixels from the memories to the PEs in order to efficiently compute a full search on this architecture.

<table>
<thead>
<tr>
<th>t</th>
<th>M</th>
<th>S</th>
<th>S9</th>
<th>PE0</th>
<th>PE1</th>
<th>PE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M(0,0)</td>
<td>S(0,0)</td>
<td></td>
<td>[M(0,0) – S(0,0)]</td>
<td>[M(0,0) – S(0,1)]</td>
<td>[M(0,0) – S(0,2)]</td>
</tr>
<tr>
<td>1</td>
<td>M(0,1)</td>
<td>S(0,1)</td>
<td></td>
<td>[M(0,1) – S(0,1)]</td>
<td>[M(0,1) – S(0,2)]</td>
<td>[M(0,1) – S(0,3)]</td>
</tr>
<tr>
<td>2</td>
<td>M(0,2)</td>
<td>S(0,2)</td>
<td></td>
<td>[M(0,2) – S(0,2)]</td>
<td>[M(0,2) – S(0,3)]</td>
<td>[M(0,2) – S(0,4)]</td>
</tr>
<tr>
<td>3</td>
<td>M(0,3)</td>
<td>S(0,3)</td>
<td></td>
<td>[M(0,3) – S(0,3)]</td>
<td>[M(0,3) – S(0,4)]</td>
<td>[M(0,3) – S(0,5)]</td>
</tr>
<tr>
<td>4</td>
<td>M(0,4)</td>
<td>S(0,4)</td>
<td></td>
<td>[M(0,4) – S(0,4)]</td>
<td>[M(0,4) – S(0,5)]</td>
<td>[M(0,4) – S(0,6)]</td>
</tr>
<tr>
<td>5</td>
<td>M(0,5)</td>
<td>S(0,5)</td>
<td></td>
<td>[M(0,5) – S(0,5)]</td>
<td>[M(0,5) – S(0,6)]</td>
<td>[M(0,5) – S(0,7)]</td>
</tr>
<tr>
<td>6</td>
<td>M(0,6)</td>
<td>S(0,6)</td>
<td></td>
<td>[M(0,6) – S(0,6)]</td>
<td>[M(0,6) – S(0,7)]</td>
<td>[M(0,6) – S(0,8)]</td>
</tr>
<tr>
<td>7</td>
<td>M(0,7)</td>
<td>S(0,7)</td>
<td></td>
<td>[M(0,7) – S(0,7)]</td>
<td>[M(0,7) – S(0,8)]</td>
<td>[M(0,7) – S(0,9)]</td>
</tr>
<tr>
<td>8</td>
<td>M(0,8)</td>
<td>S(0,8)</td>
<td></td>
<td>[M(0,8) – S(0,8)]</td>
<td>[M(0,8) – S(0,9)]</td>
<td>[M(0,8) – S(10)]</td>
</tr>
<tr>
<td>9</td>
<td>M(0,9)</td>
<td>S(0,9)</td>
<td></td>
<td>[M(0,9) – S(0,9)]</td>
<td>[M(0,9) – S(10)]</td>
<td>[M(0,9) – S(10)]</td>
</tr>
<tr>
<td>10</td>
<td>M(0,10)</td>
<td>S(0,10)</td>
<td></td>
<td>[M(0,10) – S(0,10)]</td>
<td>[M(0,10) – S(0,10)]</td>
<td>[M(0,10) – S(0,10)]</td>
</tr>
<tr>
<td>11</td>
<td>M(0,11)</td>
<td>S(0,11)</td>
<td></td>
<td>[M(0,11) – S(0,11)]</td>
<td>[M(0,11) – S(0,11)]</td>
<td>[M(0,11) – S(0,11)]</td>
</tr>
<tr>
<td>12</td>
<td>M(0,12)</td>
<td>S(0,12)</td>
<td></td>
<td>[M(0,12) – S(0,12)]</td>
<td>[M(0,12) – S(0,12)]</td>
<td>[M(0,12) – S(0,12)]</td>
</tr>
<tr>
<td>13</td>
<td>M(0,13)</td>
<td>S(0,13)</td>
<td></td>
<td>[M(0,13) – S(0,13)]</td>
<td>[M(0,13) – S(0,13)]</td>
<td>[M(0,13) – S(0,13)]</td>
</tr>
<tr>
<td>14</td>
<td>M(0,14)</td>
<td>S(0,14)</td>
<td></td>
<td>[M(0,14) – S(0,14)]</td>
<td>[M(0,14) – S(0,14)]</td>
<td>[M(0,14) – S(0,14)]</td>
</tr>
<tr>
<td>15</td>
<td>M(0,15)</td>
<td>S(0,15)</td>
<td></td>
<td>[M(0,15) – S(0,15)]</td>
<td>[M(0,15) – S(0,15)]</td>
<td>[M(0,15) – S(0,15)]</td>
</tr>
<tr>
<td>16</td>
<td>M(1,0)</td>
<td>S(1,0)</td>
<td>S(0,16)</td>
<td>[M(1,0) – S(1,0)]</td>
<td>[M(1,0) – S(1,0)]</td>
<td>[M(1,0) – S(1,0)]</td>
</tr>
<tr>
<td>17</td>
<td>M(1,1)</td>
<td>S(1,1)</td>
<td>S(0,17)</td>
<td>[M(1,1) – S(1,1)]</td>
<td>[M(1,1) – S(1,1)]</td>
<td>[M(1,1) – S(1,1)]</td>
</tr>
</tbody>
</table>

**Fig 5.8.7** A schedule of pixel fetches for a full search

- The schedule fetches one pixel from the macro block memory and two pixels from the search area memory per clock cycle.
- The pixels are distributed to the PEs in a regular pattern as shown by the schedule.
- This schedule computes 16 correlations between the macro block and search area simultaneously.
- Based on our understanding of efficient architectures for accelerating motion estimation, we can derive a more detailed definition of the architecture in UML, which is shown in Fig 5.8.8.

**Fig 5.8.8** Object diagram for the video accelerator

- The system includes the two memories for pixels, one a single-port memory and the other dual ported.
- A bus interface module is responsible for communicating with the PCI bus and the rest of the system.
• The estimation engine reads pixels from the M and S memories, and it takes commands from the bus interface and returns the motion vector to the bus interface.

**Component Design:**
• Once we have verified that the accelerator board has the required structure, we can concentrate on designing the FPGA logic.
• Because the logic for the accelerator is very regular, we can improve the FPGA’s clock rate by properly placing the logic in the FPGA to reduce wire lengths.
• If we are designing our own accelerator board, we have to design both the video accelerator design proper and the interface to the PCI bus.
• We can create and exercise the video accelerator architecture in a hardware description language like VHDL or Verilog and simulate its operation.
• Designing the PCI interface requires somewhat different techniques since we may not have a simulation model for a PCI bus.
• We may want to verify the operation of the basic PCI interface before we finish implementing the video accelerator logic.
• The accelerator board will have its own driver that is responsible for talking to the board.
• Since most of the data transfers are performed directly by the board using DMA, the driver can be relatively simple.

**System Testing:**
• Testing video algorithms requires a large amount of data.
• Luckily, the data represents images and video, which are plentiful.
• Because we are designing only a motion estimation accelerator and not a complete video compressor, it is probably easiest to use images, not video, for test data.
• You can use standard video tools to extract a few frames from a digitized video and store them in JPEG format.
• Open source for JPEG encoders and decoders is available.
• These programs can be modified to read JPEG images and put out pixels in the format required by your accelerator.
• If you want to be adventurous and try motion estimation on video, open source MPEG encoders and decoders are also available.