LECTURE NOTES

ON

EMBEDDED SYSTEMS

B.Tech VII semester (AEC016)

(Autonomous-IARE-R16)

Mr. S Lakshmanachari
(Assistant Professor)

Dr. S China Venkateswarlu
(Professor)

ELECTRONICS AND COMMUNICATION ENGINEERING

INSTITUTE OF AERONAUTICAL ENGINEERING
(AUTONOMOUS)
DUNDIGAL, HYDERABAD - 500 043

SYALLABUS

Unit-I	EMBEDDED COMPUTING

Definition of embedded system, embedded systems vs. general computing systems, history of embedded systems, complex systems and microprocessor, classification, major application areas, the embedded system design process, characteristics and quality attributes of embedded systems, formalisms for system design, design examples

Unit-II	INTRODUCTION TO EMBEDDED C AND APPLICATIONS

C looping structures, register allocation, function calls, pointer aliasing, structure arrangement, bit fields, unaligned data and endianness, inline functions and inline assembly, portability issues; Embedded systems programming in C, binding and running embedded C program in Keil IDE, dissecting the program, building the hardware; Basic techniques for reading and writing from I/O port pins, switch bounce; Applications: Switch bounce, LED interfacing, interfacing with keyboards, displays, D/A and A/D conversions, multiple interrupts, serial data communication using embedded C interfacing

Unit-III	RTOS FUNDAMENTALS AND PROGRAMMING

Operating system basics, types of operating systems, tasks and task states, process and threads, multiprocessing and multitasking, how to choose an RTOS ,task scheduling, semaphores and queues, hard real-time scheduling considerations, saving memory and power. Task communication: Shared memory, message passing, remote procedure call and sockets; Task synchronization: Task communication synchronization issues, task synchronization techniques, device drivers.

Unit-IV	EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Host and target machines, linker/locators for embedded software, getting embedded software into the target system; Debugging techniques: Testing on host machine, using laboratory tools, an example system.

Unit-V	INTRODUCTION TO ADVANCED PROCESSORS

Introduction to advanced architectures: ARM and SHARC, processor and memory organization and instruction level parallelism; Networked embedded systems: Bus protocols, I2C bus and CAN bus; Internet-EnAnalyzed systems, design example-Elevator controller.

Text Books:

1. Shibu K.V, –Introduction to Embedded Systems‖, Tata McGraw Hill Education Private Limited, 2 nd Edition, 2009.
2. Raj Kamal, –Embedded Systems: Architecture, Programming and Design‖, Tata McGraw-Hill Education, 2 nd Edition, 2011.
3. Andrew Sloss, Dominic Symes,Wright, –ARM System Developer's Guide Designing and Optimizing System Software‖, 1st Edition, 2004.

Reference Books:

1. Wayne Wolf, ― Computers as Components, Principles of Embedded Computing Systems Design‖, Elsevier, 2 nd Edition, 2009.
2. Dr. K. V. K. K. Prasad, ― Embedded / Real-Time Systems: Concepts, Design & Programming‖, dreamtech publishers, 1 st Edition, 2003.
3. Frank Vahid, Tony Givargis, –Embedded System Design‖, John Wiley & Sons, 3 rd Edition, 2006.
4. Lyla B Das, –Embedded Systems‖ , Pearson Education, 1 st Edition, 2012.
5. David E. Simon, –An Embedded Software Primer‖, Addison-Wesley, 1 st Edition, 1999.
6. Michael J. Pont, –Embedded C‖, Pearson Education, 2nd Edition, 2008.

UNIT-I
EMBEDDED COMPUTING

SYLLABUS:

Definition of embedded system, embedded systems vs. general computing systems, history of embedded systems, complex systems and microprocessor, classification, major application areas, the embedded system design process, characteristics and quality attributes of embedded systems, formalisms for system design, design examples.

INTRODUCTION:

System Definition:

A way of working, organizing or performing one or many tasks according to a fixed set of rules, program or plan.
Also an arrangement in which all units assemble and work together according to a program or plan.
Examples of Systems:

- Time display system – A watch
- Automatic cloth washing system – A washing machine

Embedded System Definitions:

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

(Or)

An embedded system is one that has dedicated purpose software embedded in computer hardware.

(Or)

It is a dedicated computer based system for an application(s) or product. It may be an independent system or a part of large system. Its software usually embeds into a ROM (Read Only Memory) or flash."

(Or)

It is any device that includes a programmable computer but is not itself intended to be a general purpose computer."

In simple words, Embedded System = (Hardware + Software) dedicated for a particular task with its own memory.

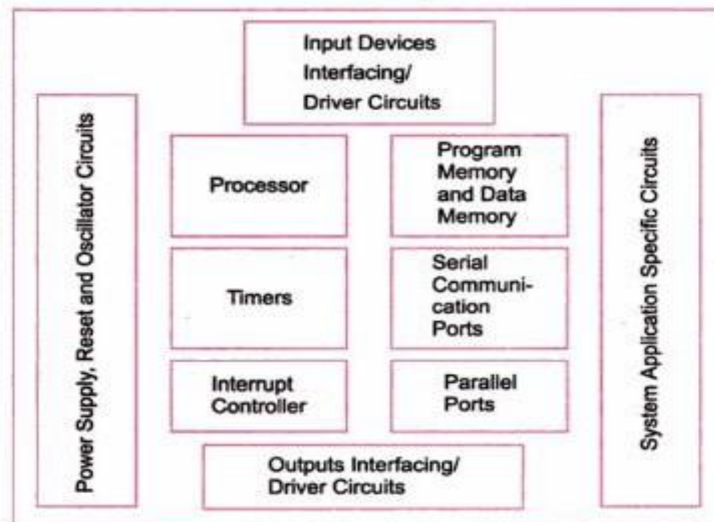
The components of embedded system hardware:

MICROPROCESSOR:

- Microprocessor is a multipurpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output.

or

- A microprocessor is a multipurpose, programmable, clock-driven, register-based electronic device that reads binary instructions from a storage device called memory accepts binary data as input and processes data according to instructions, and provides result as output.



MICROCONTROLLER:

- A **microcontroller** (sometimes abbreviated µC, uC or MCU) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory in the form of NOR flash or OTP ROM is also often included on chip, as well as a typically small amount of RAM.

or

- CPUs with integrated memory or peripheral interfaces

DIGITAL SIGNAL PROCESSOR:

- Dedicated processors .A digital signal processor (DSP) is a specialized microprocessor (or a SIP block), with its architecture optimized for the operational needs of digital signal processing

IMAGE PROCESSOR:

- An image processor, image processing engine, also called media processor, is a specialized digital signal processor (DSP) used for image processing in digital cameras, mobile phones or other devices.

EMBEDDED COMPUTING SYSTEM:

- An embedded system is a special-purpose system in which the computer is completely encapsulated by the device it controls. Unlike a general-purpose computer, such as a personal computer, an embedded system performs pre-defined tasks, usually with very specific requirements. Since the system is dedicated to a specific task, design engineers can optimize it, reducing the size and cost of the product.

- Some examples of embedded systems include ATMs, cell phones, printers, thermostats, calculators, and videogame consoles.

THE CLASSIFICATION OF EMBEDDED SYSTEM IS BASED ON FOLLOWING CRITERIA'S:

1. On generation
2. On complexity & performance
3. On deterministic behaviour
4. On triggering

1. On generation:

(i) First generation (1G):

- Built around 8bit microprocessor & microcontroller.
- Simple in hardware circuit & firmware developed.

Examples: Digital telephone keypads.

(ii) Second generation (2G):

- Built around 16-bit µp & 8-bit µc.
- They are more complex & powerful than 1G µp & µc.

Examples: SCADA systems

(iii) Third generation (3G):

- Built around 32-bit µp & 16-bit µc.
- Concepts like Digital Signal Processors (DSPs), Application Specific Integrated Circuits (ASICs) solved. Examples: Robotics, Media, etc.

(iv) Fourth generation:

- Built around 64-bit µp & 32-bit µc.
- The concept of System on Chips (SoC), Multicore Processors evolved.
- Highly complex & very powerful. Examples: Smart Phones.

2. On complexity & performance

(i) Small-scale Embedded Systems:

- Simple in application need
- Performance not time-critical.
- Built around low performance & low cost 8 or 16 bit µp/µc.

Example: an electronic toy

(ii) Medium-scale Embedded Systems:

- Slightly complex in hardware & firmware requirement.
- Built around medium performance & low cost 16 or 32 bit µp/µc.
- Usually contain operating system. Examples: Industrial machines.

(iii) Large-scale Embedded Systems:

- Highly complex hardware & firmware.
- Built around 32 or 64 bit RISC µp/µc or PLDs or Multicore Processors.
- Response is time-critical. Examples: Mission critical applications.

3. On deterministic behavior

This classification is applicable for "Real Time" systems. The task execution behavior for an embedded system may be deterministic or non-deterministic. Based on execution behavior Real Time embedded systems are divided into two types

- Hard Real Time embedded systems
- Soft Real Time embedded systems

4 On triggering

Embedded systems which are "Reactive" in nature can be based on triggering. Reactive systems can be:

- Event triggered
- Time triggered

COMPLEX SYSTEM AND MICROPROCESSORS:

➢ **Three main tasks or components in embedded system design:**

- Selecting and integrating hardware to give computer like functionalities
- Dumping main application software generally into flash or ROM and the application software performs concurrently the number of tasks.
- Integrating with a real time operating system (RTOS), this supervises the application software tasks running on the hardware and organizes the accesses to system resources according to priorities and timing constraints of tasks in the system.

Embedding Computers:

➢ **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**. It was extremely large physically compared to today's computers (e.g., it contained over 4,000 vacuum tubes).

- Very-large-scale integration (VLSI) is the process of creating an integrated circuit (IC) by combining thousands of transistors into a single chip. VLSI began in the 1970s. A microprocessor is a single-chip CPU. Very large scale integration (VLSI) technology allowed us to put a complete CPU on a single chip since 1970s, but those CPUs were very simple.
- In 1971 the **first microprocessor the Intel 4004 invented by Ted Hoff**, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it 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.
- **Automobile designers** started making use of the microprocessor soon after single-chip CPUs became available. The most important and sophisticated use of microprocessors in automobiles was to **control the engine**: determining when spark plugs fire, controlling the fuel/air mixture, and so on.
- **Microprocessors** are usually classified according to their word length.
    - An 8-bit ***microcontroller*** is designed for low-cost applications and includes on-board memory and I/O devices
    - 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory;
    - 32-bit ***RISC*** microprocessor offers very high performance for computation-intensive applications.
- **House Hold uses of microprocessor:**
    - The typical **microwave oven** has at least one microprocessor to control oven operation.
    - Many houses have **advanced thermostat systems**, which change the temperature level at various times during the day.
    - The **modern camera** is a prime example of the powerful features that can be added under microprocessor control.
    - **Digital Television** uses embedded processors

APPLICATIONS OF EMBEDDED SYSTEMS IN VARIOUS SECTORS:

We can find applications of embedded systems in following sectors:

- Daily Life Electronic appliances(Lift, Microwave Oven, Refrigerator, Washing Machine)
- Health Care(X-ray, ECG, Cardiograph, diseases diagnosis devices etc)
- Education (Laptop or desktop, projector, printer, calculator, lab equipments etc)
- Communication(Mobile phone, satellite, Modem, Network Hub, Router, Telephone,

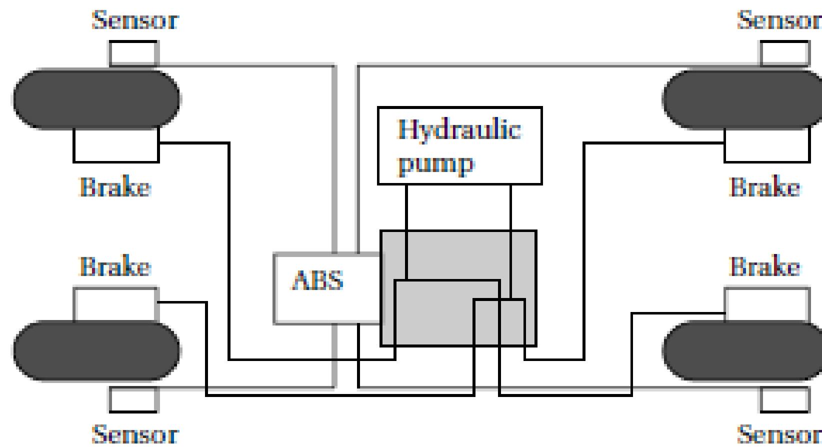
Fax)

- Security System(CC Camera, X ray Scanner, RFID System, Password protected door, Face detection)
- Entertainment(Television etc)
- Banking System(ATM etc)
- Automation
- Navigation
- Consumer Electronics: Camcorders, Cameras
- Household appliances: Washing machine, Refrigerator.
- Automotive industry: Anti-lock breaking system(ABS), engine control
- Home automation & security systems: Air conditioners, sprinklers, fire alarms.
- Telecom: Cellular phones, telephone switches.
- Computer peripherals: Printers, scanners.
- Computer networking systems: Network routers and switches.
- Healthcare: EEG, ECG machines.
- Banking & Retail: Automatic teller machines, point of sales.
- Card Readers: Barcode, smart card readers

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 seen in the below diagram. 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. The ASC _ T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting.

Characteristics of Embedded Computing Applications:

a. Complex Algorithms
b. User Interface
c. Real Time
d. Multirate
e. Manufacturing Cost
f. Power

- ***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*** behaviour. 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.

Costs of various sorts are also very important:

- ***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.

Why Use Microprocessors?

- There are many ways to design a digital system: custom logic, field-programmable gate arrays (FPGAs), and so on.
- Why use microprocessors? There are two answers:
    - 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 and can be extended to provide new features to keep up with rapidly changing markets.

Other reasons are

- Predesigned instruction set processor may in fact result in **faster implementation of your application** than designing your own custom logic.
- But there are two factors that work together to make microprocessor-based designs fast.
    - First, microprocessors execute programs very efficiently. Modern RISC processors can execute one instruction per clock cycle most of the time and high performance processors can execute several instructions per cycle.
    - Second, microprocessor manufacturers spend a great deal of money to make their CPUs run very fast. With the slight changes designer can make the microprocessor to run at the highest possible speed.

- Microprocessors are **efficient utilizers of logic**
- Microprocessors can be used for many different algorithms simply by changing the program it executes.
- The microprocessors allow program design to be separated from the design of hardware on which programs will be running.

Challenges in Embedded Computing System Design:

1. *How much hardware do we need?*
2. *How do we meet deadlines?*
3. *How do we minimize power consumption?*
4. *How do we design for upgradability?*
5. *Does it really work?*
6. *Complex testing*
7. *Limited observability and controllability*
8. *Restricted development environments*

External constraints are one important source of difficulty in embedded system design. Let's consider some important problems that must be taken into account in embedded system design.

How much hardware do we need?

We have a great deal of control over the amount of computing power we apply to our problem. We cannot only select the type of microprocessor used, but also select the amount of memory, the peripheral devices, and more. Since we often must meet both performance deadlines and manufacturing cost constraints, the choice of hardware is important—too little hardware and the system fails to meet its deadlines, too much hardware and it becomes too expensive.

How do we meet deadlines?

The brute force way of meeting a deadline is to speed up the hardware so that the program runs faster. Of course, that 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.

How do we minimize power consumption?

In battery-powered applications, power consumption is extremely important. Even in non battery applications, excessive power consumption can increase heat dissipation. One way to make a digital system consume less power is to make it run more slowly, slowing down the system can obviously lead to missed deadlines. Careful design is

required to slow down the noncritical parts of the machine for power consumption while still meeting necessary performance goals.

How do we 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. However, we want to be able to add features by changing software.

Does it really work?

Reliability is always important when selling products—customers rightly expect that products they buy will work. Reliability is especially important in some applications. If we wait until we have a running system and try to eliminate the bugs, we will be too late—we won't find enough bugs, it will be too expensive to fix them, and it will take more time.

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. This makes it more difficult to see what is going on and to affect the system's operation. We may be forced to watch the values of electrical signals on the microprocessor bus, for example, to know what is going on inside the system. Moreover, in real-time applications we may not be able to easily stop the system to see what is going on inside.

Restricted development environments: The development environments for embedded systems (the tools used to develop software and hardware) 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.

THE EMBEDDED SYSTEM DESIGN PROCESS

- The embedded system design process aimed at two objectives.
  - First, it will give us an introduction to the various steps in embedded system design Second, it will allow us to consider the design ***methodology*** itself
- A design methodology is important for **three** reasons.
  - **First**, to ensure that we have done everything we need.
  - **Second**, it allows us to develop computer-aided design tools.
  - **Third**, it makes members of a design team to

communicate easily Designing can be done in two ways.
They are

■ Top down

■ Bottom –up

Figure 1.1 summarizes the major steps in the embedded system design process. In this top–down view, we start from the system ***requirements.*** In bottom up approach we start with components. ***Specification***, we create a more detailed description of what we want. But the specification states only how the system behaves, not how it is built.

The details of the system's internals begin to take shape when we develop the architecture, which gives the system structure in terms of large components. Once we know the components we need, we can design those components, including both software modules and any specialized hardware we need. Based on those components, we can finally build a complete system. In this section we will consider design from the ***top–down***—we will begin with the most abstract description of the system.

The alternative is a **bottom–up** view in which we start with components to build a system. Bottom–up design steps are shown in the figure as dashed-line arrows. We need bottom–up design because we do not have perfect insight into how later stages of the design process will turn out.

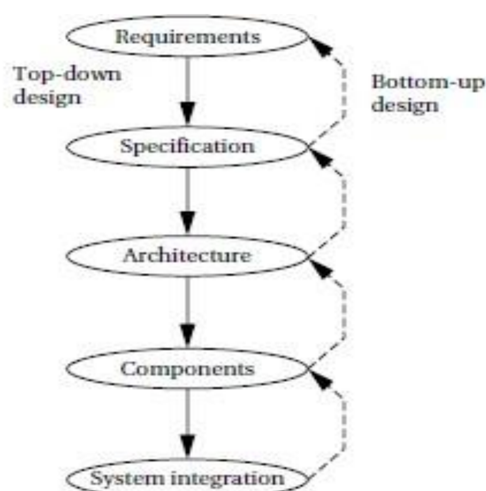


FIGURE 1.1
Major levels of abstraction in the design process.

We need to consider the major goals of the design:

- Manufacturing cost;
- Performance (both overall speed and deadlines); and
- Power consumption.

We must also consider the tasks we need to perform at every step in the design process. At each step in the design, we add detail:

- We must *analyze* the design at each step to determine how we can meet the specifications.
- We must then *refine* the design to add detail.
- And we must *verify* the design to ensure that it still meets all system goals, such as cost, speed, and so on.

1. Requirements:

Clearly, before we design a system, we must know what we are designing. The initial stages of the design process capture this information for use in creating the architecture and components. We generally proceed in two phases:

1. First, we gather an informal description from the customers known as requirements;
2. Second we refine the requirements into a specification that contains enough information to begin designing the system architecture.

   Separating out requirements analysis and specification is often necessary because of the large gap between what the customers can describe about the system they want and what the architects need to design the system. Requirements may be ***functional*** or ***non functional***.

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. As we have noted, 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
  - ***Nonrecurring 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.

➢ Validating a set of requirements is ultimately a psychological task since it requires understanding both what people want and how they communicate those needs. One good way to refine at least the user interface portion of a system's requirements is to build a ***mock-up***. The mock-up may use scanned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation.

➢ Requirements analysis for big systems can be complex and time consuming. However, capturing a relatively small amount of information in a clear, simple format is a good start towards understanding system requirements. As part of system design to analyze requirements, we will use a simple requirements methodology. Figure 1.2 shows a sample ***requirements form*** that can be filled out at the start of the project. Let's consider the entries in the form:

Name
Purpose
Inputs
Outputs
Functions
Performance
Manufacturing cost
Power
Physical size and weight

FIGURE 1.2
Sample requirements form.

■***Name:*** This is simple but helpful. Giving a name to the project should tell the purpose of the machine.

■***Purpose:*** This should be a brief one- or two-line description of what the system is supposed to do. If you can't describe the essence of your system in one or two lines, chances are that you don't understand it well enough.

■***Inputs and outputs:*** These two entries are more complex than they seem. The inputs and outputs to the system encompass a wealth of detail:

— *Types of data:* Analog electronic signals? Digital data? Mechanical inputs?

— *Data characteristics:* Periodically arriving data, such as digital audio samples? How many bits per data element?

— *Types of I/O devices:* Buttons? Analog/digital converters? Video displays?

■***Functions:*** This is a more detailed description of what the system does. A good way to approach this is to work from the inputs to the outputs: When the system receives an input, what does it do? How do user interface inputs affect these functions? How do different functions interact?

■***Performance:*** Many embedded computing systems spend at least some time to control physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time.

■***Manufacturing cost:*** This includes primarily the cost of the hardware components. Even if you don't know exactly how much you can afford to spend on system components, you should have some idea of the eventual cost range. Cost has a substantial influence on architecture.

■***Power:*** Similarly, you may have only a rough idea of how much power the system can consume, but a little information can go a long way. Typically, the most important decision is whether the machine will be battery powered or plugged into the wall. Battery-powered machines must be much more careful about how they spend energy.

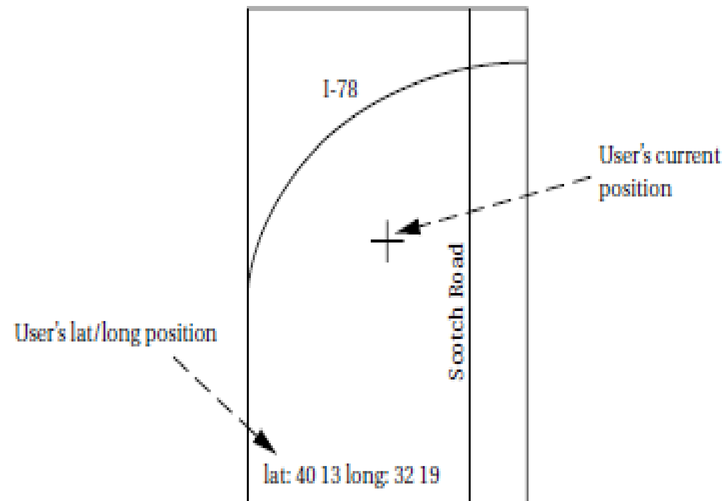
■***Physical size and weight:*** You should give some indication of the physical size of the system that helps to take architectural decisions.

After writing the requirements, you should check them for internal consistency. To practice the capture of system requirements, Example 1.1 creates the requirements for a GPS moving map system.

Example 1.1

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 figure.



What requirements might we have for our GPS moving map? Here is an initial list:

■*Functionality:* This system is designed for highway driving and similar uses. The system should show major roads and other landmarks available in standard topographic databases.

■*User interface:* The screen should have at least 400_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 sec.

■*Cost:* The selling cost 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 batteries.

Requirements form for GPS moving map system:

Name	GPS moving map
Purpose	Consumer-grade moving map for driving use
Inputs	Power button, two control buttons
Outputs	Back-lit LCD display 400 _ 600

Functions	Uses 5-receiver GPS system. Three user-selectable resolutions: always display current latitude and l longitude
Performance	Updates screen within 0.25 seconds upon movement
Manufacturing cost	$30
Power	100mW
Physical size and weight	No more than 2” _ 6,” 12 ounces

The selling price is four to five times the ***cost of goods sold*** (the total of all the component costs).

2. Specification:

- The specification is more precise—it serves as the contract between the customer and the architects.
- The specification must be carefully written so that it accurately reflects the customer’s requirements
  and that can be clearly followed during design.
- An unclear specification leads different types of problems.
- If the behaviour of some feature in a particular situation is unclear from the specification, the designer may implement the wrong functionality.
- If global characteristics of the specification are wrong or incomplete, the overall system architecture derived from the specification may be inadequate to meet the needs of implementation.
- 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.

3. Architecture Design:

- 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.
- To understand what an architectural description is, let’s look at sample architecture for the moving map of Example 1.1.
- Figure 1.3 shows a sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them.

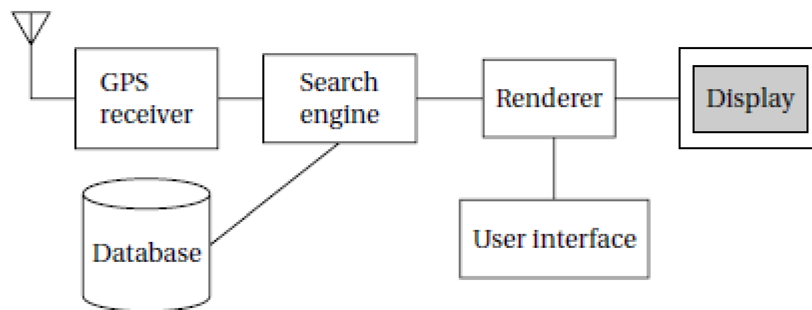
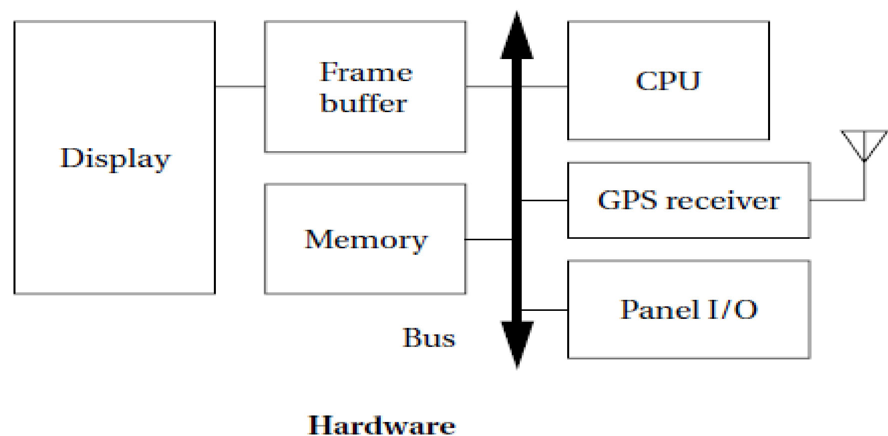
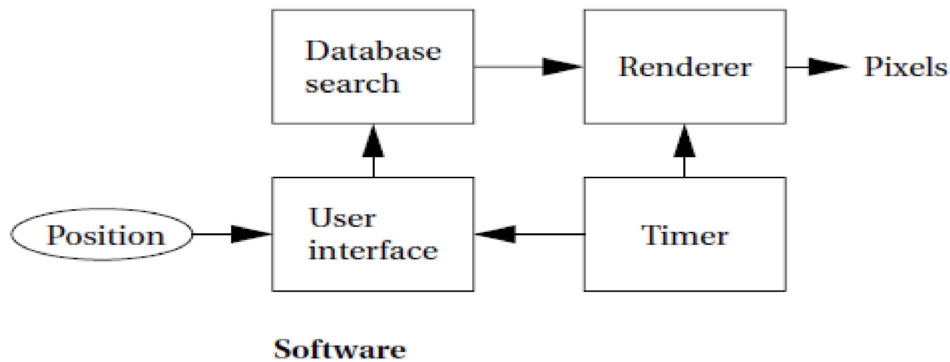


FIGURE 1.3

Block diagram for the moving map.

- The topographic database and to render (i.e., draw) the results for the display.
- We have chosen to separate those functions so that we can potentially do them in parallel— performing rendering separately from searching the database may help us update the screen more fluidly.
- For more implementation details we should refine that system block diagram into two block diagrams:
  - Hardware block diagram (Hardware architecture)
  - Software block diagram(Software architecture)
- These two more refined block diagrams are shown in Figure 1.4
- The hardware block diagram clearly shows that we have one central CPU surrounded by memory and I/O devices.
- We have chosen to use two memories:
  - A frame buffer for the pixels to be displayed
  - A separate program/data memory for general use by the CPU
- The software block diagram fairly closely follows the system block diagram.
- We have added a timer to control when we read the buttons on the user interface and render data onto the screen.





Software

- Architectural descriptions must be designed to satisfy both functional and nonfunctional requirements.
- Not only must all the required functions be present, but we must meet cost, speed, power and other nonfunctional constraints.
- Starting out with 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 nonfunctional constraints when creating the hardware and software architectures.

4. Designing Hardware and Software Components

- The architectural description tells us what components we need.
- In general the components will include both hardware—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 will memory chips and many other components.
- In the moving map, the GPS receiver is a good example of a specialized component that will nonetheless be a predesigned, standard component.
- We can also make use of standard software modules. One good example is the topographic database.
- Standard topographic databases exist, and you probably want to use standard routines to access the database—the data in a predefined format and it is highly compressed to save storage.
- Using standard software for these access functions not only saves us design time.

5. System Integration:

- Putting hardware and software components together will give complete working system.
- Bugs are typically found during system integration, and good planning can help us to find the bugs quickly.
- If we debug only a few modules at a time, we are more likely to uncover the simple bugs and able to easily recognize them.

- 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. As a result, determining why things do not work correctly and how they can be fixed is a challenge in itself.

4. FORMALISMS FOR SYSTEM DESIGN

- We perform a number of different design tasks at different levels of abstraction: creating requirements and specifications, architecting the system, designing code, and designing tests. It is often helpful to conceptualize these tasks in diagrams.
- The ***Unified Modeling Language (UML)***. UML was designed to be useful at many levels of abstraction in the design process. UML is an ***object-oriented*** modeling language.
- The design in terms of actual objects helps us to understand the natural structure of the system.
- Object-oriented 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.
- What is the relationship between an object-oriented specification and an object oriented programming language?
- A specification language may not be executable. But both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

Structural Description:

- By ***structural description***, we mean the basic components of the system.
- The principal component of an object-oriented design is ***object***. An object includes a set of ***attributes*** that define its internal state.
- When implemented in a programming language, these attributes usually become variables or constants held in a data structure. In some cases, we will add the type of the attribute after the attribute name for clarity, but we do not always have to specify a type for an attribute.
- An object describing a display (such as a CRT screen) is shown in UML notation in Figure 1.5.

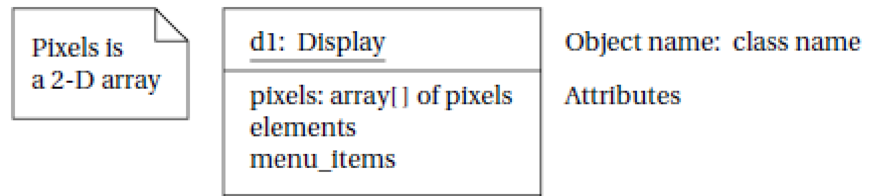


FIGURE 1.5

An object in UML notation.

- The text in the folded-corner page icon is a ***note***; it does not correspond to an object in the system and only serves as a comment.
- The attribute is, in this case, an array of pixels that holds the contents of the display.
- 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.
- In a programming language, the operations would become pieces of code used to manipulate the object.
- The UML description of the *Display* class is shown in Figure 1.6.

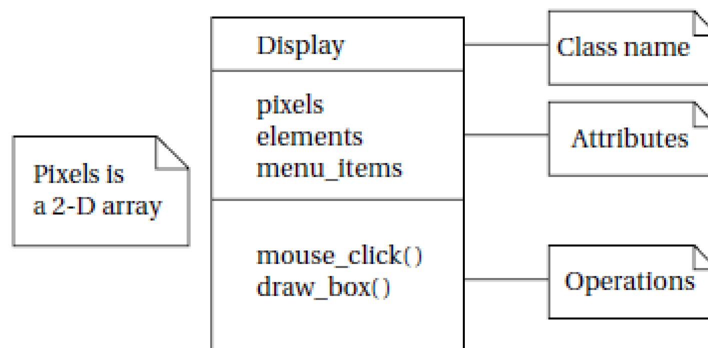


FIGURE 1.6

A class in UML notation.

- The class has the name that we saw used in the *d*1 object since *d*1 is an instance of class *Display*. The

  Display class defines the *pixels* attribute seen in the object;
- A class defines both the ***interface*** for a particular type of object and that object's ***implementation***.
- 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

Derived class:

- *Unified Modeling Language*, like most object-oriented languages, allows us to define one class in terms of another.
- An example is shown in Fig1.7, where we ***derive*** two particular types of displays. The first, *BW_display*, describes a black and- white display. This does not require us to add new attributes or operations, but we can specialize both to work on one-bit pixels.
- A ***derived class*** inherits all the attributes and operations from its ***base class***.
- Here *Display* is the base class for the two derived classes. A derived class is defined to include all the attributes of its base class. This relation is transitive—if *Display* were derived from another class, both *BW_display* and *Color_map_display* would inherit all the attributes and operations of *Display's* base class as well.

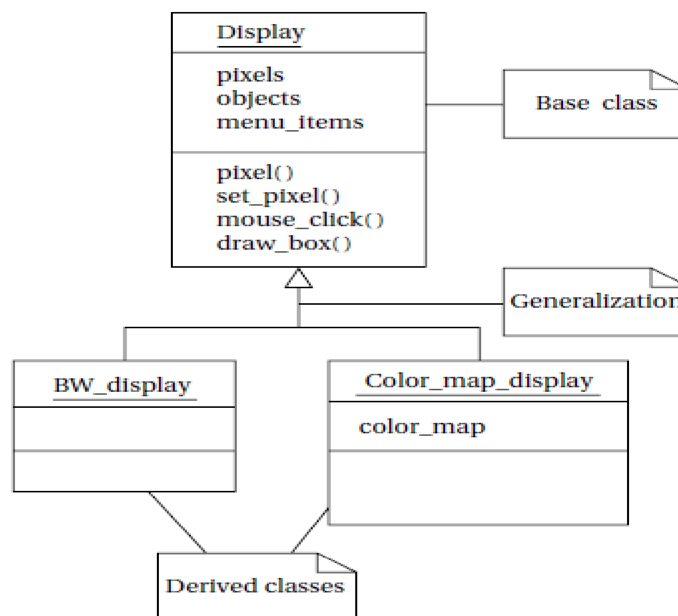


FIGURE 1.7
Derived classes as a form of generalization in UML.

- Inheritance has two purposes.
  - It allows us to describe one class that shares some characteristics with another class.
  - It captures those relationships between classes and documents them

- **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. Both *BW_display* and *Color_map_display* are specific versions of *Display*, so *Display* generalizes both of them.

Multiple inheritances:

- In which a class is derived from more than one base class.
- An example of multiple inheritances is shown in Figure 1.8; In this case, we have created a *Multimedia display* class by combining the *Display* class with a *Speaker* class for sound.
- The derived class inherits all the attributes and operations of both its base classes, *Display* and *Speaker*.

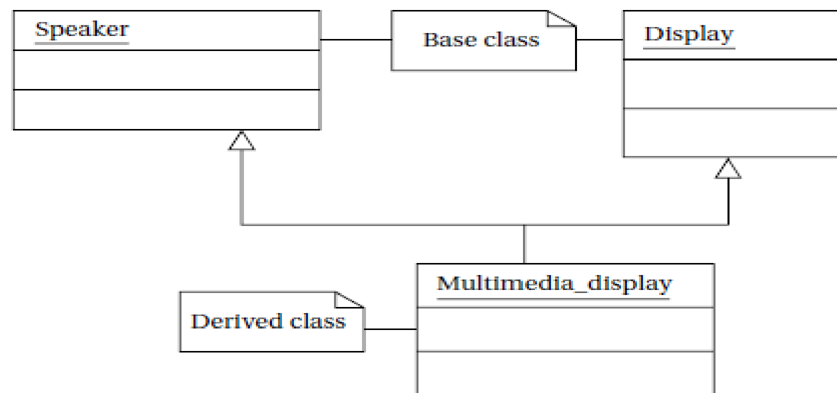
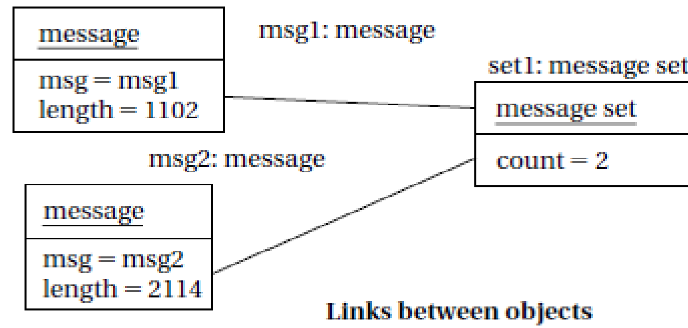


FIGURE 1.8
Multiple inheritance in UML.

Link:

- A ***link*** describes a relationship between objects; association is to link as class is to object.
- Fig1.9 shows an example of links and an association.



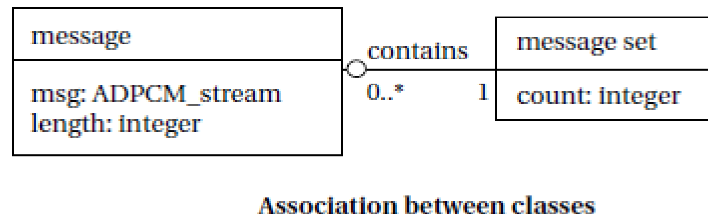


FIGURE 1.9

Links and association.

- When we consider the actual objects in the system, there is a set of messages that keeps track of the current number of active messages (two in this example) and points to the active messages. In this case, the link defines the *contains* relation.
- When generalized into classes, we define an association between the message set class and the message class. The association is drawn as a line between the two labeled with the name of the association, namely, *contains*. The ball and the number at the message class end indicate that the message set may include zero or more message objects.

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***.
- Fig1.10 shows UML states; the transition between two states is shown by arrow. 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***.

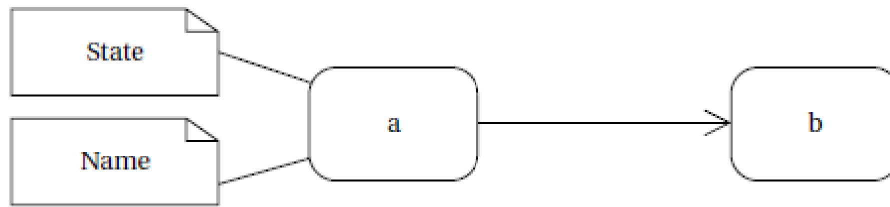


FIGURE 1.10

A state and transition in UML.

- An event is some type of action. Events are divided into two categories. They are:
  - External events: The event may originate outside the system, such as a user pressing a button.
  - Internal events: It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine.
- We will concentrate on the following three types of events defined by UML, as illustrated in figure 1.11(signal and call event) and (Time out event)
  - 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.

    Because it is an object, 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. The label *tm (time-value)* on the edge gives the amount of time after which the transition occurs. A time-out is generally implemented with an external timer.

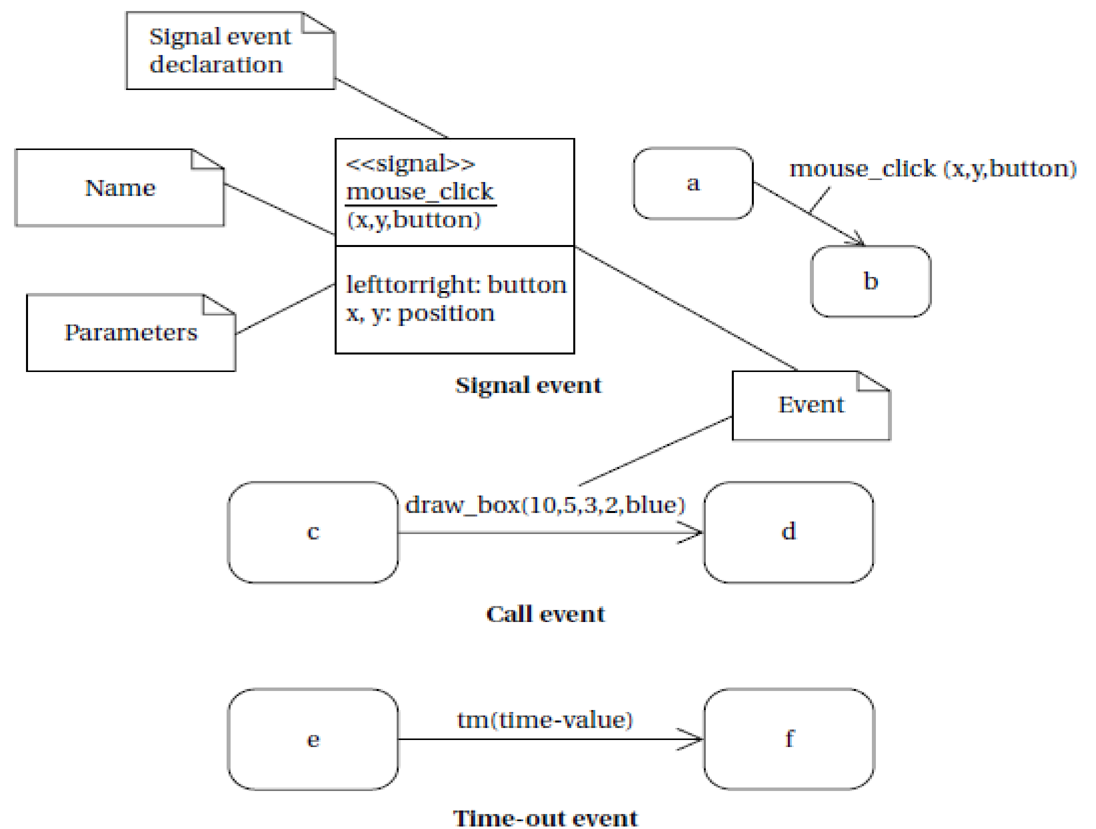


FIGURE 1.11

Signal, call, and time-out events in UML.

Unconditional and conditional transitions:

- The states in the state machine represent different conceptual operations.
- In some cases, we take conditional transitions out of states based on inputs or the results of some computation done in the state.
- In other cases, we make an unconditional transition to the next state. Both the unconditional and conditional transitions make use of the call event.
- Let’s consider a simple state machine specification to understand the semantics of UML state machines. A state machine for an operation of the display is shown in Fig1.12. The start and stop states are special states that help us to organize the flow of the state machine.

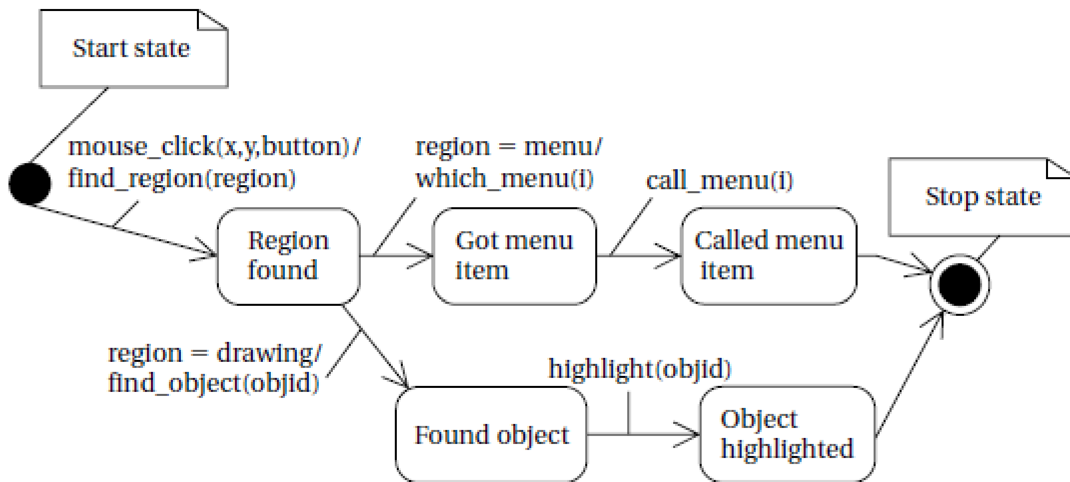


FIGURE 1.12

A state machine specification in UML.

Sequence diagram:

- It is sometimes useful to show the sequence of operations over time, particularly when several objects are involved.
- In this case, we can create a sequence diagram, like the one for a mouse click scenario shown in Fig1.13.
- A ***sequence diagram*** is somewhat similar to a hardware timing diagram, although the time flows vertically in a sequence diagram, whereas time typically flows horizontally in a timing diagram.
- The sequence diagram is designed to show a particular scenario or choice of events. In this case, the sequence shows what happens when a mouse click is on the menu region.

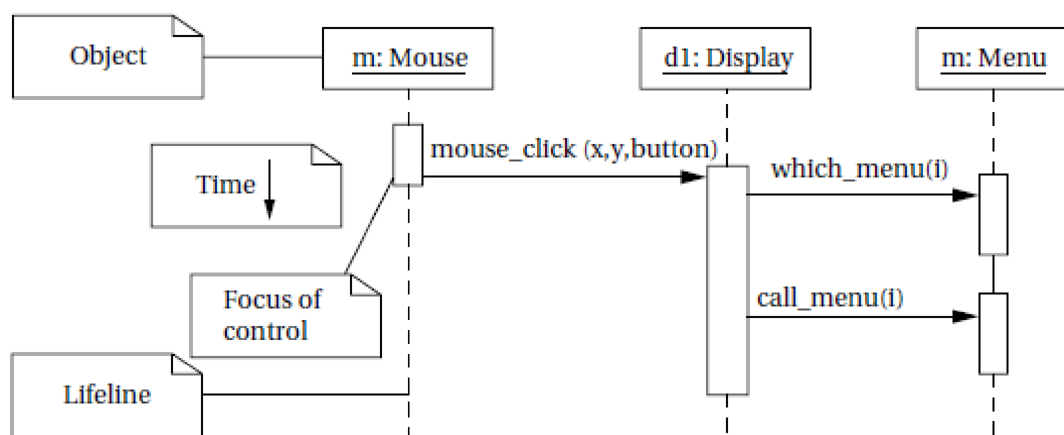


FIGURE 1.13

A sequence diagram in UML.

- Processing includes three objects shown at the top of the diagram. Extending below each object is its *lifeline*, a dashed line that shows how long the object is alive. In this case, all the objects remain alive for the entire sequence, but in other

cases objects may be created or destroyed during processing.

- The boxes along the lifelines show the *focus of control* in the sequence, that is, when the object is actively processing.
- In this case, the mouse object is active only long enough to create the *mouse_click* event. The display object remains in play longer; it in turn uses call events to invoke the menu object twice: once to determine which menu item was selected and again to actually execute the menu call.
- The find region ( ) call is internal to the display object, so it does not appear as an event in the diagram.

DESIGN EXAMPLE: MODEL TRAIN CONTROLLER:

- The model train controller, which is shown in the below figure.

i. The user sends messages to the train with the control box attached to the tracks.

ii. The control box may have familiar controls such as throttle, emergency stop button and so on.

iii. Since train receives its electrical power from the track, the control box can send a signal to the train over the track by modulating the power supply voltage.

iv. As shown Fig1.14, the control panel sends packet over the tracks to the receiver on the train. 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:

- Here is a 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 inertia control that shall allow the user to adjust the responsiveness of the train to commanded changes in speed. Higher inertia means that the train responds more slowly to a change in the throttle, simulating the inertia of a large train. The inertia control will provide at least eight different levels.
  - There shall be an emergency stop button.
  - An error detection scheme will be used to transmit messages.

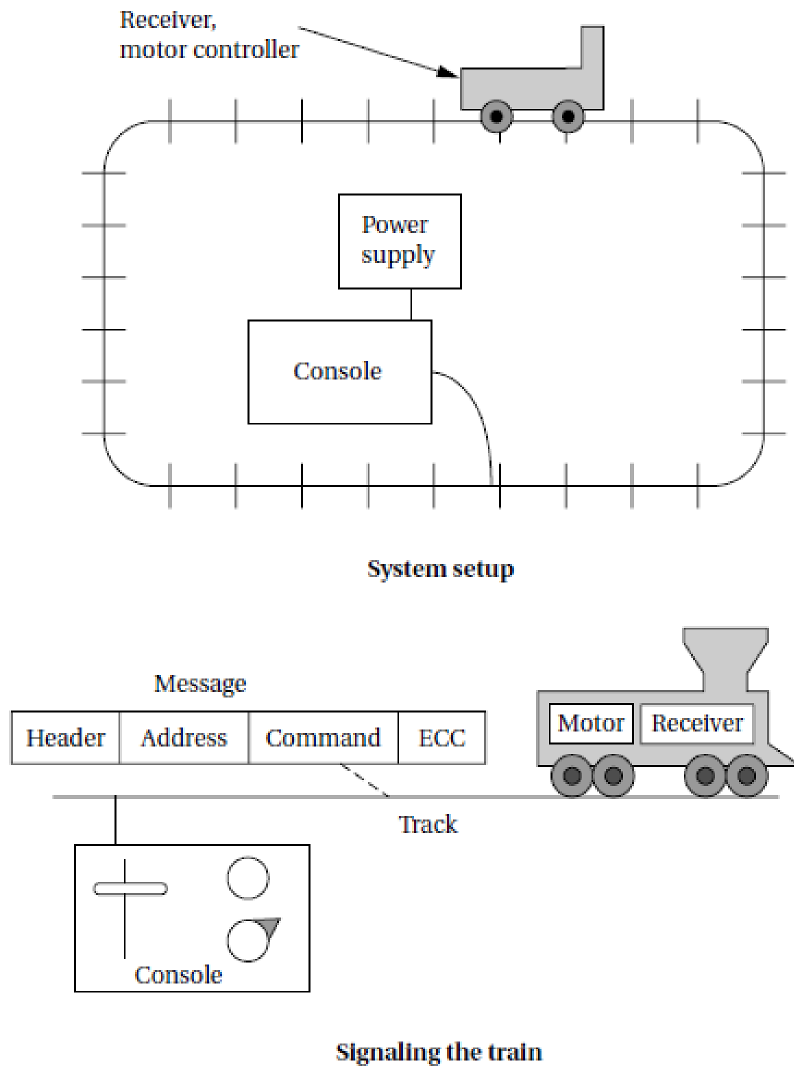


FIGURE 1.14

A model train control system.

➢ We can put the requirements into our chart format:

Name	Model train controller
Purpose	Control speed of up to eight model trains
Inputs	Throttle, inertia setting, emergency stop, train number
Outputs	Train control signal
Functions	Set engine speed based upon inertia settings, Respond to emergency stop
Performance	Can update train speed at least 10 times per second
Manufacturing cost	$50
Power	10W (plugs into walls)
Physical size and weight	Console should be comfortable for two hands, approximate size of standard keyboard. Weight less than 2 pounds

CONCEPTUAL SPECIFICATION OF MODEL TRAIN CONTROLLER:

1. Objects: Console , Train
2. Commands: set speed, set inertia, Estop.
3. Console: panel, formatter, transmitter
4. Train: receiver, controller, motor interface

- The conceptual specification allows us to understand the system little better. Writing of conceptual specification will help us to write a detailed specification. Defining the messages will help us understand the functionality of the components. The set of commands that we can use to implement the requirements placed on the system.
- The system console controls the train by sending messages on to the tracks. The transmissions are packetized: each packet includes an address and a message. A typical sequence of train control commands is shown as a UML sequence diagram.

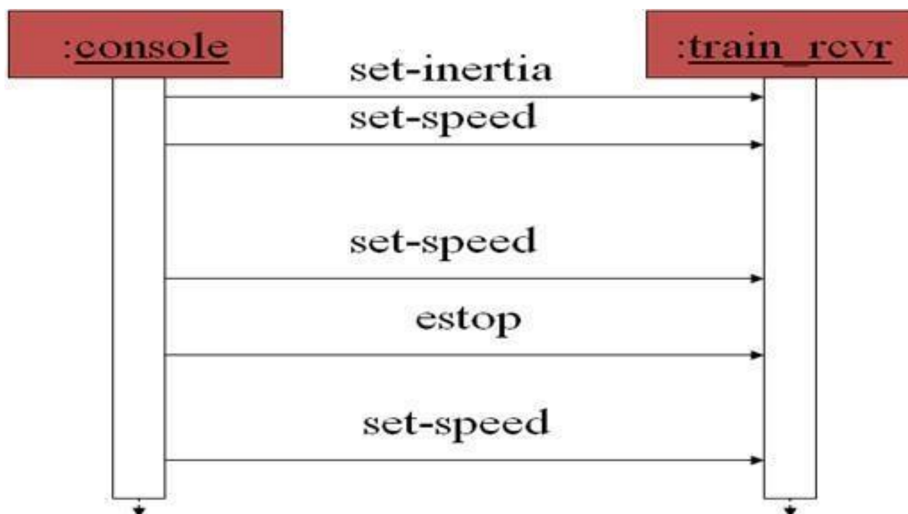


Fig: A UML sequence diagram for a typical sequence of train control commands

- The focus of the control bars shows the both the console and receiver run continuously. The packets can be sent at any time—there is no global clock controlling when the console sends and the train receives, we do not have to worry about detecting collisions among the packets.
- Set- inertia message will send infrequently. Most of the message commands are speed commands. When a train receives speed command, it will speed up and slow down the train smoothly at rate determined by the set-inertia command.
- An emergency stop command may be received, which causes the train receiver to immediately shut down the train motor.
- We can model the **commands in UML** with two level class hierarchy as shown in the Fig1.16. Here we have one base class command, there are three sub classes set-speed, set-inertia, Estop, derived from base class. One for each specific type of command.

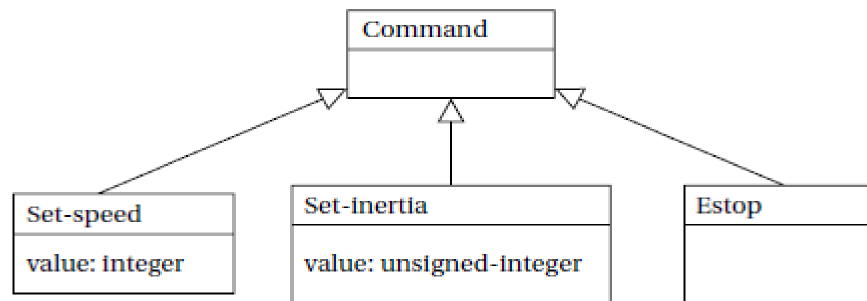


FIGURE 1.16

Class diagram for the train controller messages.

- We now need to model the train control system itself. There are clearly two major subsystems: the control-box and the train board component. Each of these subsystems has its own internal structure.
- The figure 1.17 Shows relationship between console and receiver (ignores role of track):

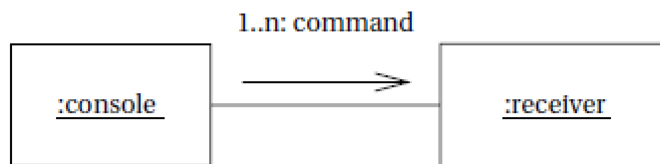


FIGURE 1.17

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

- The console and receiver are each represented by objects: the console sends a sequence of packets to the train receiver, as illustrated by the arrow. The notation on the arrow provides both the type of message sent and its sequence in a flow of messages .we have numbered the arrow's messages as 1…n .
- Let's break down the console and receiver into three major components.
- The console needs to perform three functions
  - **Console**:
    - Read state of front panel
    - Format messages
    - Transmit messages.
- The train receiver must also perform three major functions
  - **Train receiver:**
    - receive message
    - interpret message
    - control the train
- The UML class diagram is show in the below figure 1.18
- **Console class roles:**

- **Panel:** Describes the console front panel, which contains analog knobs and interface hardware to interface to the digital parts of the system.

- **Formatter:** It knows how to read the panel knobs and creates bit stream for message.
- **Transmitter:** Send the message along the track.

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

➢ **Train class roles:**

- **Receiver:** It knows how to turn the analog signal on the track into digital form.

- **Controller:** Interprets received commands and figures out how to control the motor.

- **Motor interface:** Generates the analog signals required to control the motor.

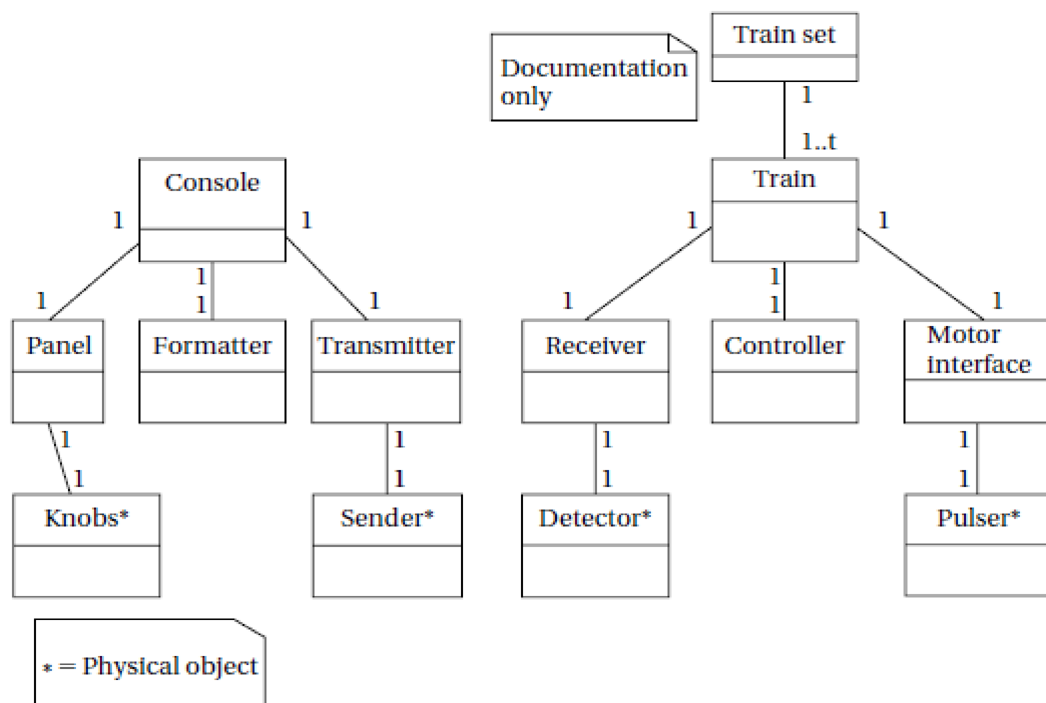


FIGURE 1.18

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

➢ We define two 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.

DETAILED SPECIFICATION:

- Conceptual specification that defines the basic classes, let's refine it to create a more detailed specification. We won't make a complete specification. But we will add details to the class. We can now fill in the details of the conceptual specification. Sketching out the spec first helps us understand the basic relationships in the system.
- We need to define the analog components in a little more detail because there characteristics will strongly influence the formatter and controller. Fig1.19 shows a little more detail than Fig 1.18, It includes attributes and behavior of these classes. The panel has three knobs: *train* number (which train is currently being controlled), *speed* (which can be positive or negative), and *inertia*. It also has one button for *emergency-stop*.
- The *Sender* and *Detector* classes are relatively simple: They simply put out and pick up a bit, respectively.

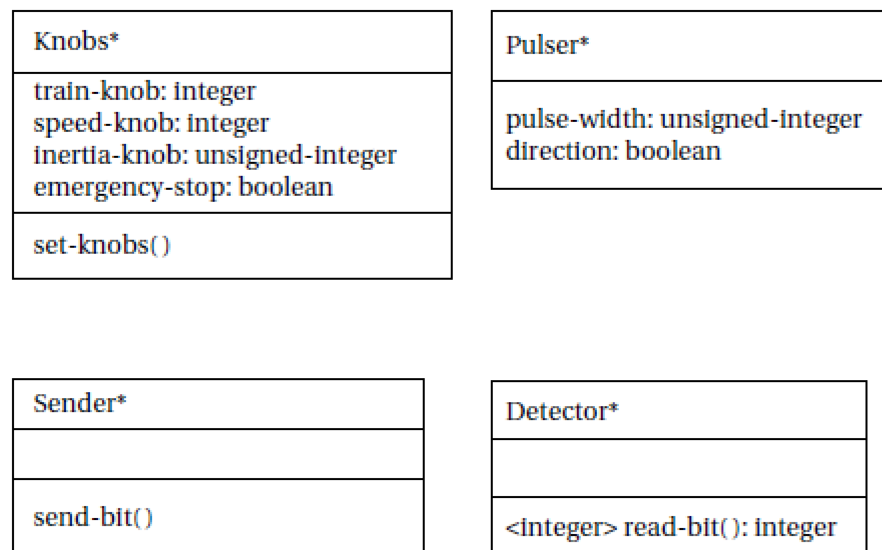


FIGURE 1.19

Classes describing analog physical objects in the train control system.

- To understand the *Pulser* class, let's consider how we actually control the train motor's speed. As shown in Figure 1.20, the speed of electric motors is commonly controlled using pulse-width modulation: Power is applied in a pulse for a fraction of some fixed interval, with the fraction of the time that power is applied determining the speed.

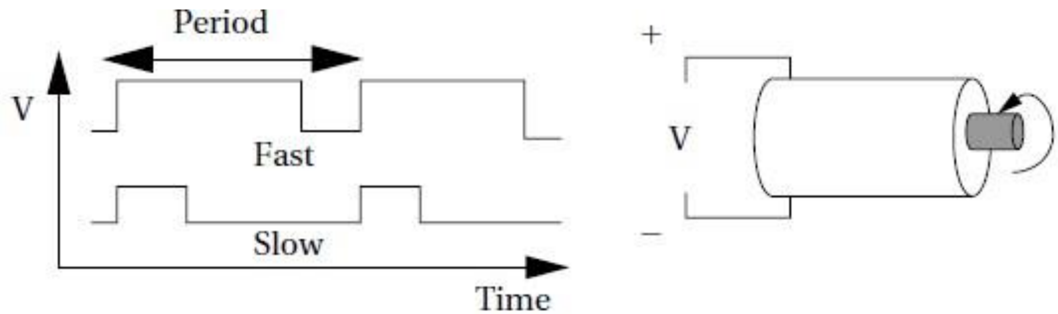


FIGURE 1.20

Controlling motor speed by pulse-width modulation.

- Figure 1.21 shows the classes for the panel and motor interfaces. These classes form the software interfaces to their respective physical devices.

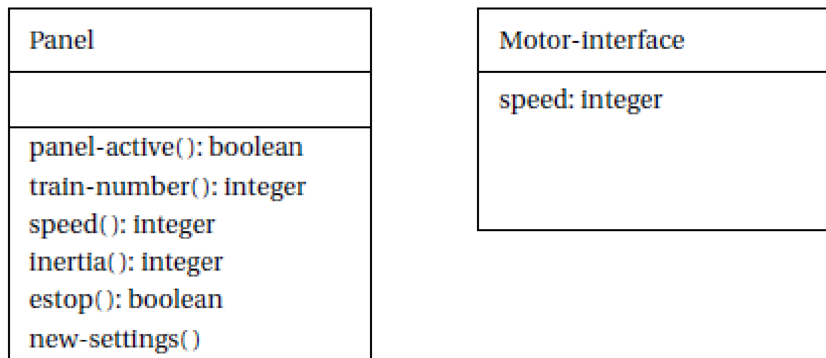


FIGURE 1.21

Class diagram for the Panel and Motor interface.

- The *Panel* class defines a behavior for each of the controls on the panel;
- The *new-settings* behavior uses the *set-knobs* behavior of the *Knobs** class to change the knobs settings whenever the train number setting is changed.
- The *Motor-interface* defines an attribute for speed that can be set by other classes.
- The *Transmitter* and *Receiver* classes are shown in Figure 1.22.They provides the software interface to the physical devices that send and receive bits along the track.

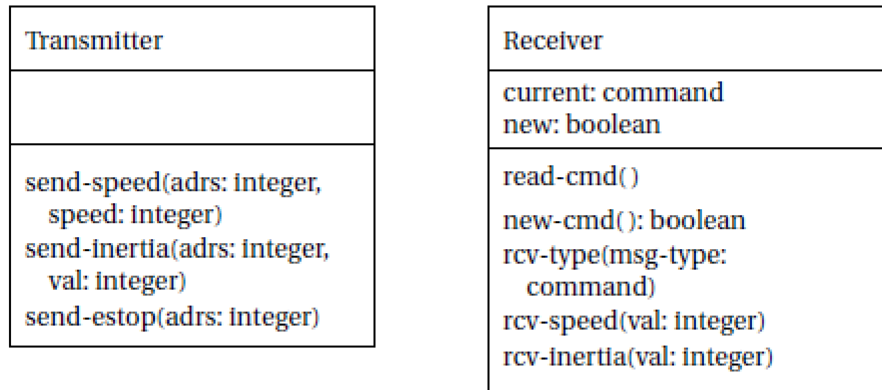


FIGURE 1.22

Class diagram for the Transmitter and Receiver.

- The *Transmitter* provides a distinct behavior for each type of message that can be sent; it internally takes care of formatting the message.
- The *Receiver* class provides a *read-cmd* behavior to read a message off the tracks.
- The *Formatter* class is shown in Figure 1.23. The formatter holds the current control settings for all of the trains.
- The *send-command* method is a utility function that serves as the interface to the transmitter.
- The *operate* function performs the basic actions for the object.
- The *panel-active* behaviour returns true whenever the panel's values do not correspond to the
  current values.

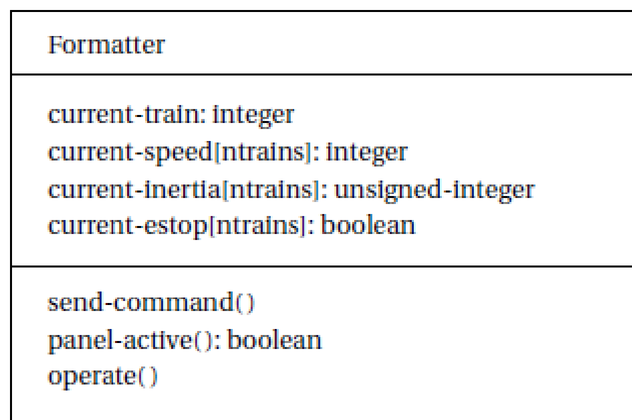


FIGURE 1.23

Class diagram for the Formatter class.

- The role of the formatter during the panel's operation is illustrated by the sequence diagram of Figure 1.24.

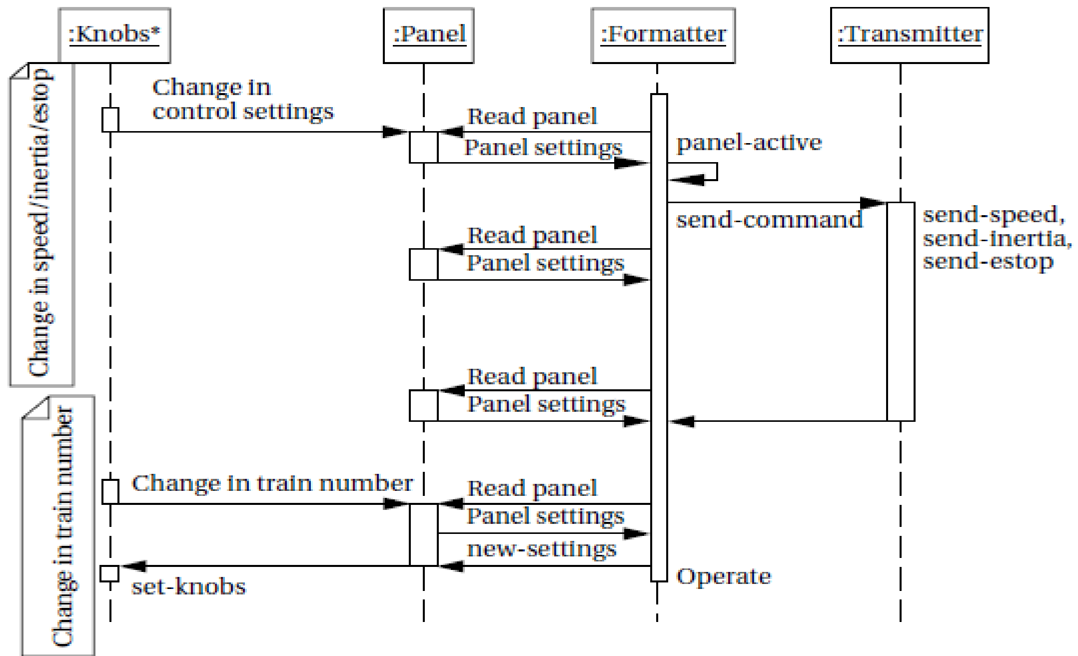


FIGURE 1.24

Sequences diagram for transmitting a control input.

- The figure shows two changes to the knob settings: first to the throttle, inertia, or emergency stop; then to the train number.
- The panel is called periodically by the formatter to determine if any control settings have changed. If a setting has changed for the current train, the formatter decides to send a command, issuing a *send- command* behavior to cause the transmitter to send the bits.
- Because transmission is serial, it takes a noticeable amount of time for the transmitter to finish a command; in the meantime, the formatter continues to check the panel's control settings.
- If the train number has changed, the formatter must cause the knob settings to be reset to the proper values for the new train.
- The state diagram for a very simple version of the *operate* behavior of the *Formatter* class is shown in Figure 1.25.
- This behavior watches the panel for activity: If the train number changes, it updates the panel display; otherwise, it causes the required message to be sent.

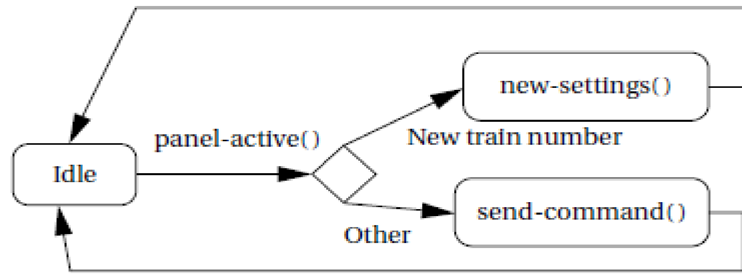


FIGURE 1.25

State diagram for the formatter operate behavior.

- Figure 1.26 shows a state diagram for the *panel-active* behavior.

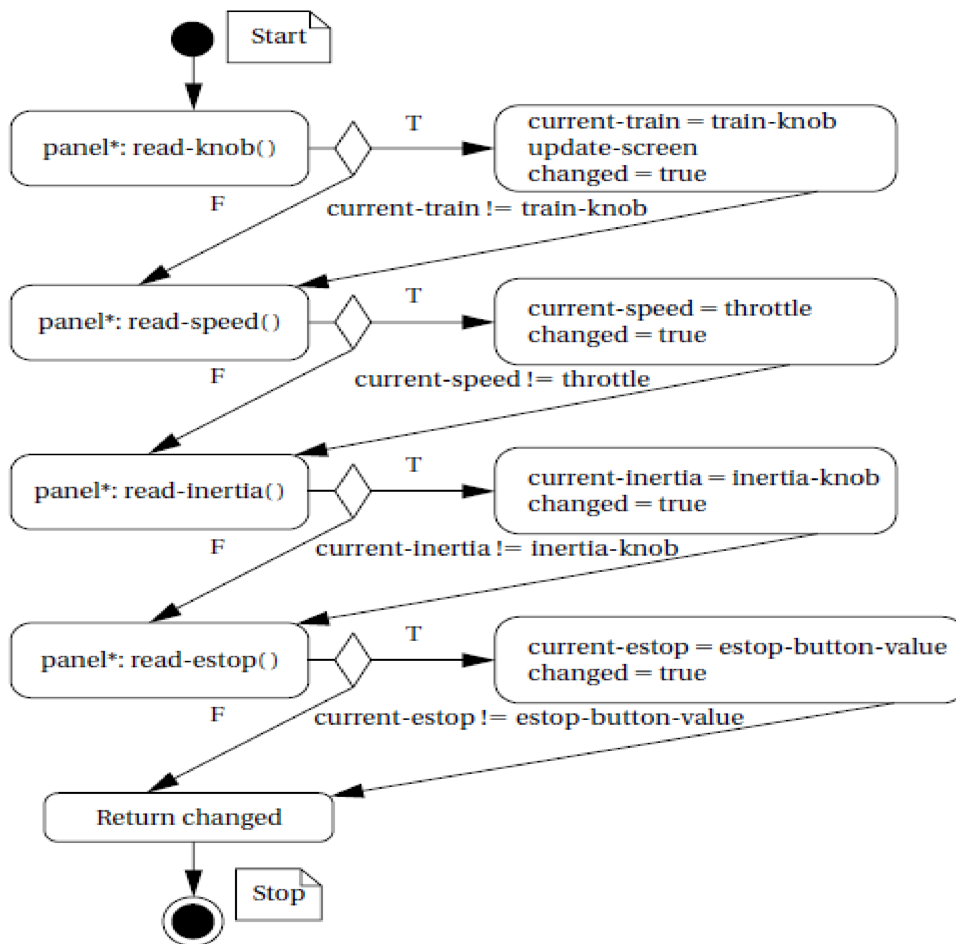


FIGURE 1.26

State diagram for the panel-active behavior.

- The definition of the train's *Controller* class is shown in Figure 1.27
- The *operate* behavior is called by the receiver when it gets a new command; *operate* looks at the contents of the message and uses the *issue-command* behavior to change the speed, direction, and inertia settings as necessary.

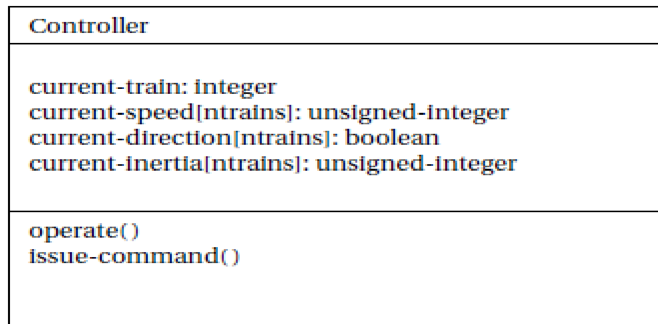


FIGURE 1.27
Class diagram for the Controller class.

- A specification for *operate* is shown in Figure 1.28.

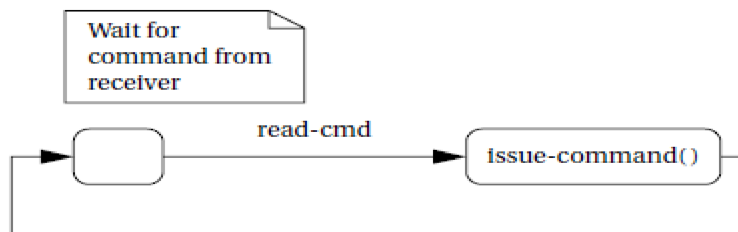


FIGURE 1.28
State diagram for the Controller operate behavior.

- The operation of the *Controller* class during the reception of a *set-speed* command is illustrated in Figure 1.29.

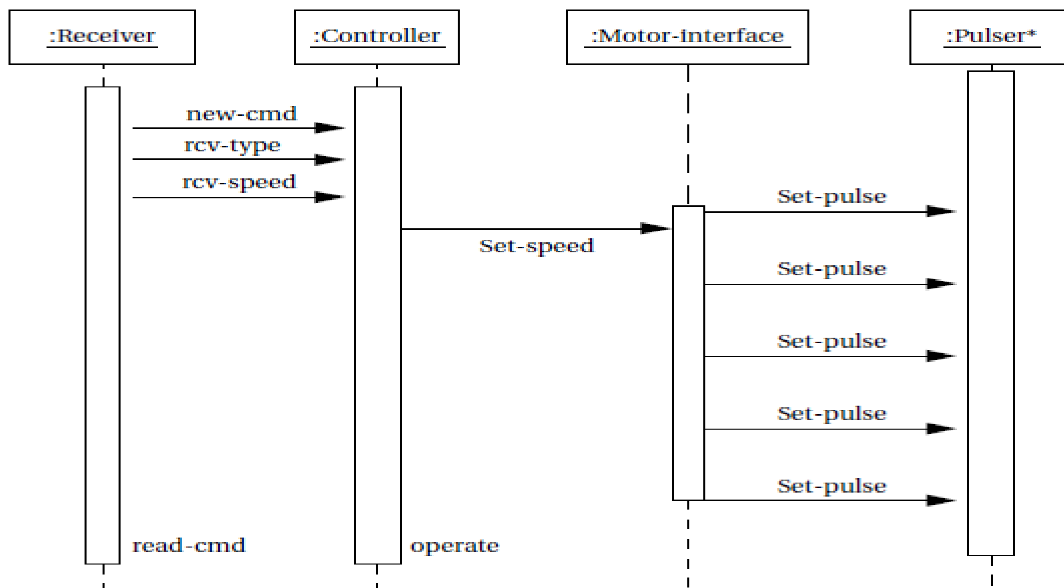


FIGURE 1.29
Sequence diagram for a set-speed command received by the train.

UNIT-II

INTRODUCTION TO EMBEDDED C AND APPLICATIONS

SYLLABUS:

C looping structures, register allocation, function calls, pointer aliasing, structure arrangement, bit fields, unaligned data and endianness, inline functions and inline assembly, portability issues; Embedded systems programming in C, binding and running embedded C program in Keil IDE, dissecting the program, building the hardware; Basic techniques for reading and writing from I/O port pins, switch bounce; Applications: Switch bounce, LED interfacing, interfacing with keyboards, displays, D/A and A/D conversions, multiple interrupts, serial data communication using embedded C interfacing.

2.1 C Looping Structures

This section looks at the most efficient ways to code for and while loops on the ARM. We start by looking at loops with a fixed number of iterations and then move on to loops with a variable number of iterations. Finally we look at loop unrolling.

2.1.1 Loops with a Fixed Number of Iterations

What is the most efficient way to write a **for** loop on the ARM? Let's return to our checksum example and look at the looping structure.

Here is the last version of the 64-llword packet checksum routine we studied in Section 5.2. This shows how the compiler treats a loop with incrementing count i++.

```
int checksum_v5(int *data)
{
unsigned int i;
int sum=0;

for (i=0; i<64; i++)
{
sum += *(data++);
}
return sum;
}
```

This compiles to

```
checksum_v5
      MOV     r2,r0             ; r2 = data
      MOV     r0,#0             ; sum = 0
      MOV     r1,#0             ;i=0
checksum_v5_loop
      LDR     r3,[r2],#4        ; r3 = *(data++)
```

```
ADD     r1,r1,#1            ; i++
CMP     r1,#0x40            ; compare i, 64
ADD     r0,r3,r0            ; sum += r3
BCC     checksum_v5_loop    ; if (i<64) goto loop
MOV     pc,r14              ; return sum
```

It takes three instructions to implement the for loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i< 64

This is not efficient. On the ARM, a loop should only use two instructions:

- A subtract to decrement the loop counter, which also sets the condition code flags on the result
- A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags. Since we are no longer using i as an array index, there is no problem in counting down rather than up.

2.1.2 Loops Using a Variable Number of Iterations

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable N giving the number of words in the data packet. Using the lessons from the last section we count down until N 0 and don't require an extra loop counter i.

The checksum_v7 example shows how the compiler handles a for loop with a variable number of iterations N.

```
int checksum_v7(int *data, unsigned int N)
{
  int sum=0;

  for (; N!=0; N--)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v7
        MOV     r2,#0               ; sum = 0
```

```
        CMP     r1,#0                 ; compare N, 0
        BEQ     checksum_v7_end       ; if (N==0) goto end

checksum_v7_loop
        LDR     r3,[r0],#4            ; r3 = *(data++)
        SUBS    r1,r1,#1              ; N-- and set flags
        ADD     r2,r3,r2              ; sum += r3
        BNE     checksum_v7_loop      ; if (N!=0) goto loop
checksum_v7_end
        MOV     r0,r2                 ; r0 = sum
        MOV     pc,r14                ; return r0
```

Notice that the compiler checks that N is nonzero on entry to the function. Often this check is unnecessary since you know that the array won't be empty. In this case a do-while loop gives better performance and code density than a for loop.

2.1.3 LOOP UNROLLING

We saw in Section 5.3.1 that each loop iteration costs two instructions in addition to the body of the loop: a subtract to decrement the loop count and a conditional branch.

We call these instructions the *loop overhead*. On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.

You can save some of these cycles by *unrolling* a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion. For example, let's unroll our packet checksum example fourtimes.

EXAMPLE4

The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet Nis a multiple of four.

```
int checksum_v9(int *data, unsigned int N)
{
  int sum=0;

  do
  {
  sum += *(data++);
  sum += *(data++);
  sum += *(data++);
  sum += *(data++);
  N -= 4;
  } while (N!=0);
```

```
return sum;
}
```

This compiles to

```
checksum_v9
        MOV     r2,#0               ; sum = 0
checksum_v9_loop
        LDR     r3,[r0],#4          ; r3 = *(data++)
        SUBS    r1,r1,#4            ; N -= 4 & set flags
        ADD     r2,r3,r2            ; sum += r3
        LDR     r3,[r0],#4          ; r3 = *(data++)
        ADD     r2,r3,r2            ; sum += r3
        LDR     r3,[r0],#4          ; r3 = *(data++)
        ADD     r2,r3,r2            ; sum += r3
        LDR     r3,[r0],#4          ; r3 = *(data++)
        ADD     r2,r3,r2            ; sum += r3
        BNE     checksum_v9_loop    ; if (N!=0) goto
                                      loop
        MOV     r0,r2               ; r0 = sum
        MOV     pc,r14              ; return r0
```

We have reduced the loop overhead from 4N cycles to (4N)/4 = N cycles. On the ARM7TDMI, this accelerates the loop from 8 cycles per accumulate to 20/4 = 5 cycles per accumulate, nearly doubling the speed! For the ARM9TDMI, which has a faster load instruction, the benefit is even higher.

There are two questions you need to ask when unrolling a loop:

- How many times should I unroll the loop?
- What if the number of loop iterations is not a multiple of the unroll amount? For example, what if N is not a multiple of four in checksum_v9?

To start with the first question, only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.

Suppose the loop is important, for example, 30% of the entire application. Suppose you unroll the loop until it is 0.5 KB in code size (128 instructions). Then the loop overhead is at most 4 cycles compared to a loop body of around 128 cycles. The loop overhead cost is 3/128, roughly 3%. Recalling that the loop is 30% of the entire application, overall the loop overhead is only 1%. Unrolling the code further gains little extra performance, but has a significant impact on the cache contents. It is usually not worth unrolling further when the gain is less than 1%.

For the second question, try to arrange it so that array sizes are multiples of your unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases.

2.2 REGISTER ALLOCATION

The compiler attempts to allocate a processor register to each local variable you use in a C function. It will try to use the same register for different local variables if the use of the variables do not overlap. When there are more local variables than available registers, the compiler stores the excess variables on the processor stack. These variables are called spilled or swapped out variables since they are written out to memory (in a similar way virtual memory is swapped out to disk). Spilled variables are slow to access compared to variables allocated to registers.
To implement a function efficiently, you need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

First let's look at the number of processor registers the ARM C compilers have avail- able for allocating variables. Table 5.3 shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

Provided the compiler is not using software stack checking or a frame pointer, then the C compiler can use registers *r0* to *r12* and *r14* to hold variables. It must save the callee values of *r4* to *r11* and *r14* on the stack if using these registers.

In theory, the C compiler can assign 14 variables to registers without spillage. In practice, some compilers use a fixed register such as *r12* for intermediate scratch working and do not assign variables to this register. Also, complex expressions require intermediate working registers to evaluate. Therefore, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.

If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use. A variable used inside a loop counts multiple times. You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

The register keyword in C hints that a compiler should allocate the given variable to a register. However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM). Therefore we recommend that you avoid using register and rely on the compiler's normal register allocation routine.

Table 2.1 C compiler register usage.

Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	
r9	*v6 sb*	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.
r10	*v7 sl*	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.
r11	*v8 fp*	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	*ip*	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
r13	*sp*	The stack pointer, pointing to the full descending stack.
r14	*lr*	The link register. On a function call this holds the return address.
r15	*pc*	The program counter.

SUMMARY **Efficient Register Allocation**

- Try to limit the number of local variables in the internal loop of functions to 12. The compiler should be able to allocate these to ARM registers.
- You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.

2.3 FUNCTION CALLS

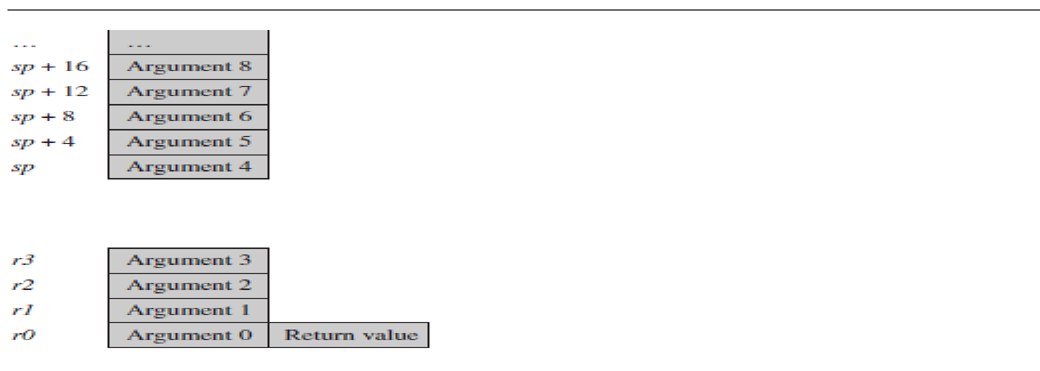
The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers. The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.

The first four integer arguments are passed in the first four ARM registers: *r0, r1, r2*, and *r3*. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in Figure 5.1. Function return integer values are passed in *r0*.

This description covers only integer or pointer arguments. Two-word arguments such as longlongor doubleare passed in a pair of consecutive argument registers and returned in *r0, r1*. The compiler may pass structures in registers or by reference according to command line compiler options.

The first point to note about the procedure call standard is the *four-register rule*. Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments. For functions with four or fewer arguments, the compiler can pass all the arguments in registers. For functions with more arguments, both the caller and callee must access the stack for some arguments. Note that for C++ the first argument to an object method is the *this* pointer. This argument is implicit and additional to the explicit arguments.

If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures. Group related arguments into structures, and pass a structure pointer rather than mul- tiple arguments. Which arguments are related will depend on the structure of your software.



ATPCS argument passing.

The next example illustrates the benefits of using a structure pointer. First we show a typical routine to insert Nbytes from array datainto a queue. We implement the queue using a cyclic buffer with start address Q_start(inclusive) and end address Q_end(exclusive).

```
char *queue_bytes_v1(
char *Q_start,            /* Queue buffer start address */
char *Q_end,              /* Queue buffer end address */
char *Q_ptr,              /* Current queue pointer position */
char *data,               /* Data to insert into the queue */
unsigned int N)           /* Number of bytes to insert */
{
do
{
*(Q_ptr++) = *(data++);

if (Q_ptr == Q_end)
{
Q_ptr = Q_start;
}
} while (--N); return Q_ptr;
}
```

This compiles to

queue_bytes_v1

```
        STR     r14,[r13,#-4]!   ; save lr on the stack
        LD      r12,[r13,#4]     ; r12 = N
        R
queue_v1_loop
        LDRB    r14,[r3],#1      ; r14 = *(data++)
        STRB    r14,[r2],#1      ; *(Q_ptr++) = r14
        CMP     r2,r1            ; if (Q_ptr ==
                                   Q_end)
        MOVE    r2,r0            ;     {Q_ptr =
        Q                              Q_start;}
        SUBS    r12,r12,#1       ; --N and set flags
        BNE     queue_v1_l       ; if (N!=0) goto loop
                oop
        MOV     r0,r2            ; r0 = Q_ptr
        LDR     pc,[r13],#4      ; return r0
```

Compare this with a more structured approach using three function arguments.

POINTER ALIASING:

Two pointers are said to alias when they point to the same address. If you write to one pointer, it will affect the value you read from the other pointer.

In a function, the compiler often doesn't know which pointers can alias and which pointers can't. The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

UNALIGNED DATA AND ENDIANNESS:

Unaligned data and endianness are two issues that can complicate memory accesses and portability. In computing endianness is the ordering or sequencing of bytes of a word of digital data in computer memory storage or during transmission. A big-endian system stores the most significant byte of a word at the smallest memory address and the least significant byte at the largest memory address.

A memory access is said to be aligned when the data being accessed is n bytes long and the datum address is n-byte aligned. ... A memory pointer that refers to primitive data that is n bytes long is said to be aligned if it is only allowed to contain addresses that are n-byte aligned, otherwise it is said to be unaligned.

INLINE FUNCTIONS AND INLINE ASSEMBLY:

Generally the inline term is used to instruct the compiler to insert the code of a function into the code of its caller at the point where the actual call is made. Such functions are called "inline functions". ... It is just a set of assembly instructions written as inline functions.

EMBEDDED SYSTEMS PROGRAMMING IN C:

The embedded firmware is responsible for controlling the various peripherals of the embedded hard-ware and generating response in accordance with the functional requirements mentioned in the requirements for the particular embedded product.
Firmware is considered as the master brain of the embedded system.
Imparting intelligence to an embedded system is a onetime process and it can happen at any stage, it can be immediately after the fabrication of the embedded hardware or at a later stage.

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

'C' v/s. 'Embedded C':

'C' is a well structured, well defined and standardized general purpose programming language with extensive bit manipulation support.
'C' offers a combination of the features of high level language and assembly and helps in hardware access programming (system level programming) as well as business package developments (Application developments like pay roll systems, banking applications, etc).
The conventional 'C' language follows ANSI(American National Standards Institute) standard and it incorporates various library files for different operating systems.
A platform (operating system) specific application, known as, compiler is used for the conversion of programs written in 'C' to the target processor (on which the OS is running) specific binary files. Hence it is a platform specific development.
Embedded 'C' can be considered as a subset of conventional 'C' language. Embedded 'C' supports all 'C' instructions and incorporates a few target processor specific functions/instructions.
It should be noted that the standard ANSI 'C' library implementation is always tailored to the target processor/controller library files in Embedded 'C'.
A software program called '**Cross-compiler**' is used for the conversion of programs written in Embedded 'C' to target processor/controller specific instructions (machine language).

Compiler vs. Cross-Compiler:

Compiler is a software tool that converts a source code written in a high level language on top of a particular operating system running on a specific target processor architecture (e.g. Intel x86/Pentium).

Here the operating system, the compiler program and the application making use of the source code run on the same target processor. The source code is converted to the target processor specific **machine instructions**.

The development is platform specific (OS as well as target processor on which the OS is running). Compilers are generally termed as 'Native Compilers'. A native compiler generates machine code for the same machine (processor) on which it is running.

Cross-compilers are the software tools used in cross-platform development applications. In cross-platform development, the compiler running on a particular target processor/OS converts the source code to machine code for a target.

Embedded system development is a typical example for cross-platform development where embedded firmware is developed on a machine with Intel/AMD or any other target processors and the same is converted into machine code for any other target processor architecture (e.g. 8051, PIC, ARM, etc).

Keil C51 is an example for cross-compiler. The term 'Compiler' is used interchangeably with 'Cross-compiler' in embedded firmware applications. Whenever you see the term 'Compiler' related to any embedded firmware application, please understand that it is referring to the cross-compiler.

COMPILER	CROSS COMPILER
A software that translates the computer code written in high-level programming language to machine language	A software that can create executable code for platforms other than the one on which the compiler is running
Helps to convert the high-level source code into machine understandable machine code	A type of compiler that can create executable code for different machines other than the machine it runs on

Using 'C' in 'Embedded C':

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

Keywords and Identifiers:

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

All 'C' supported keywords should be written in 'lowercase' letters.

C Keywords are predefined, reserved words used in programming that have special meanings to the compiler.

Keywords in C Language

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

Identifiers are user defined names and labels. Identifiers can contain letters of English alphabet (both upper and lower case) and numbers. The starting character of an identifier should be a letter. The only special character allowed in identifier is underscore (_).
Ex: Root, _getchar, _sin, x_1, x1, If

Data Types:

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

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

Arithmetic and Relational Operations:

The list of arithmetic operations supported by 'C' are listed below.

arithmetic operators		relation operators	
operator	effect	operator	effect
−	Subtraction	>	Greater than
+	Addition	>=	Greater or equal
·	Multiplication	<	Less than
/	Division	<=	Less or equal
% or MOD	Modulus division	==	Equal
−−	Decrement	! =	Not equal
++	Increment		

Logical Operations:

Logical operations are usually performed for decision making and program control transfer.

Logical Operators		
Operator	Description	Example
&&	AND	x=6 y=3 x<10 && y>1 Return True
||	OR	x=6 y=3 x==5 || y==5 Return False
!	NOT	x=6 y=3 !(x==y) Return True

Looping Instructions:

Looping instructions are used for executing a particular block of code repeatedly till a condition is met or wait till an event is fired.

Embedded programming often uses the looping instructions for checking the status of certain I/O ports, registers, etc. and also for producing delays. Certain devices allow write/read operations to and from some registers of the device only when the device is ready and the device ready is normally indicated by a status register or by setting/clearing certain bits of status registers.

Hence the program should keep on reading the status register till the device ready indication comes. The reading operation forms a loop. The looping instructions supported by are listed below.

Looping Instructions:

//while statement
While (expression)
{
Body of while loop
}

```
//do while statement
do
{
Body of do loop
}
While (expression)
```

```
//for loop
for (initialization; test for condition; update variable)
{
Body of for loop
}
```

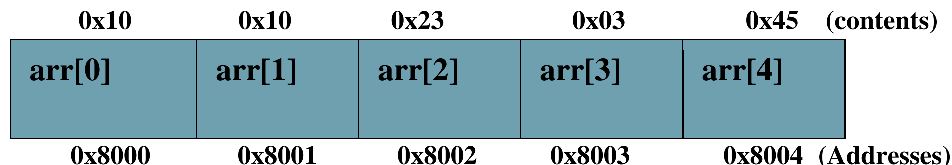
Arrays and Pointers:

Array is a collection of related elements (data types).

Arrays are usually declared with data type of array, name of the array and the number of related elements to be placed in the array.

For example the following array declaration declares a character array with name 'arr' and reserves space for 5 character elements in the memory as below figure.

char arr [5]



The elements of an array are accessed by using the array index or subscript.

The index of the first element is '0'. For the above example the first element is accessed by arr[0], second element by arr[1], and so on. In the above example, the array starts at memory location 0x8000 (arbitrary value taken for illustration) and the address of the first element is 0x8000.

The `address of operator (&) returns the address of the memory location where the variable is stored. Hence &arr[0] will return 0x8000 and &arr[1] will return 0x8001, etc.. The name of the array itself with no index (subscript) always returns the address of the first element. If we examine the first element arr[0] of the above array, we can see that the variable arr[0] is allocated a memory location 0x8000 and the contents of that memory location holds the value for arr[0].

Pointers:
Pointer is a flexible at the same time most dangerous feature, capable of creating potential damages leading to firmware crash, if not used properly.
Pointer is a memory pointing based technique for variable access and modification. Pointers are very helpful in
1. Accessing and modifying variables
2. Increasing speed of execution
3. Accessing contents within a block of memory
4. Passing variables to functions by eliminating the use of a local copy of variables
5. Dynamic memory.

BINDING AND RUNNING EMBEDDED C PROGRAM IN KEIL IDE:

Embedded system means some combination of computer hardware and programmable software which is specially designed for a particular task like displaying message on LCD. If you are still wondering about an embedded system, just take a look at these circuit applications using 8051 microcontroller. You can call these applications embedded systems as it involves hardware (8051 microcontroller) and software (the code written in assembly language).

Some real life examples of embedded systems may involve ticketing machines, vending machines, temperature controlling unit in air conditioners etc. Microcontrollers are nothing without a Program in it.

One of the important part in making an embedded system is loading the software/program we develop into the microcontroller. Usually it is called "***burning software***" into the controller. Before "burning a program" into a controller, we must do certain prerequisite operations with the program. This includes writing the program in assembly language or C language in a text editor like notepad, compiling the program in a compiler and finally generating the hex code from the compiled program. Earlier people used different softwares/applications for all these 3 tasks. Writing was done in a text editor like notepad/WordPad, compiling was done using a separate software (probably a dedicated compiler for a particular controller like 8051), converting the assembly code to hex code was done using another software etc. It takes lot of time and work to do all these separately, especially when the task involves lots of error debugging and reworking on the source code.
Keil MicroVision is free software which solves many of the pain points for an embedded program developer. This software is an integrated development environment (IDE), which integrated a text editor to write programs, a compiler and it will convert your source code to hex files too.
Here is simple guide to start working with Keil uVision which can be used for

- Writing programs in C/C++ or Assembly language
- Compiling and Assembling Programs
- Debugging program
- Creating Hex and Axf file
- Testing your program without Available real Hardware (Simulator Mode)

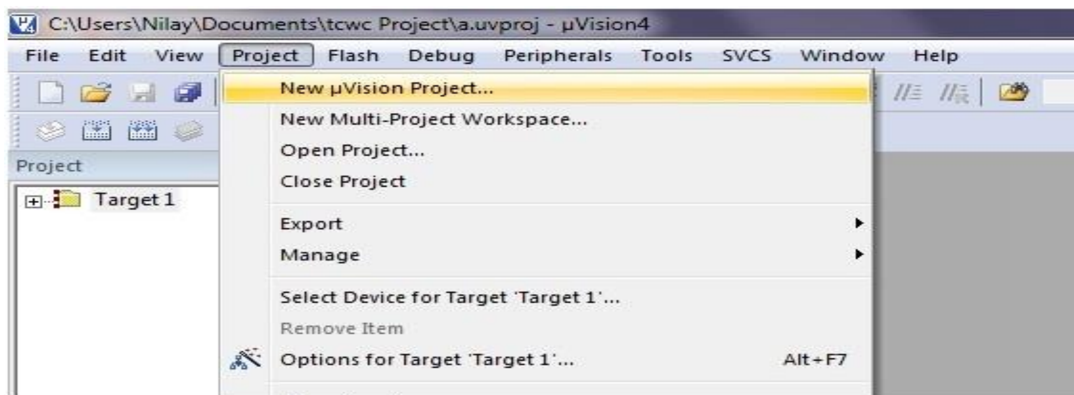
This is simple guide on Keil uVision 4 though also applicable on previous versions also.

These are the simple steps to get off the mark your inning!

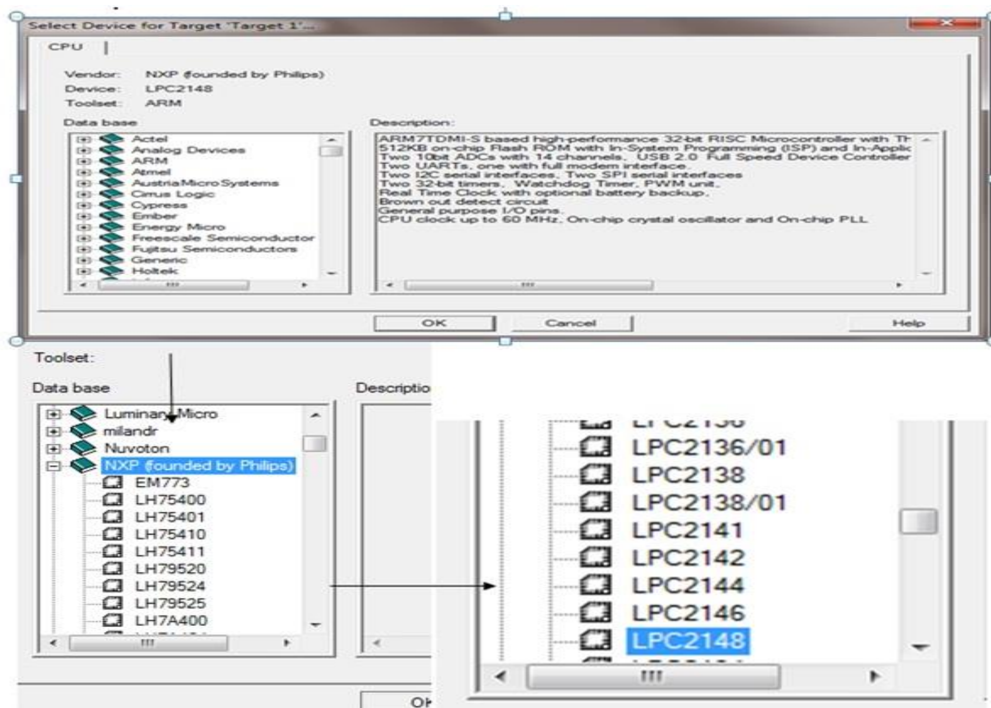
Step 1: After opening Keil uV4, Go to **Project** tab and

Create new uVision project

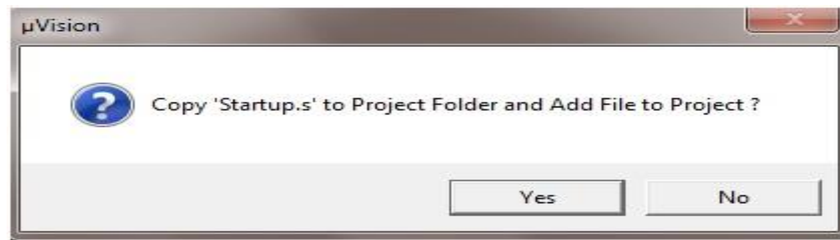
Now Select new folder and give name to Project.



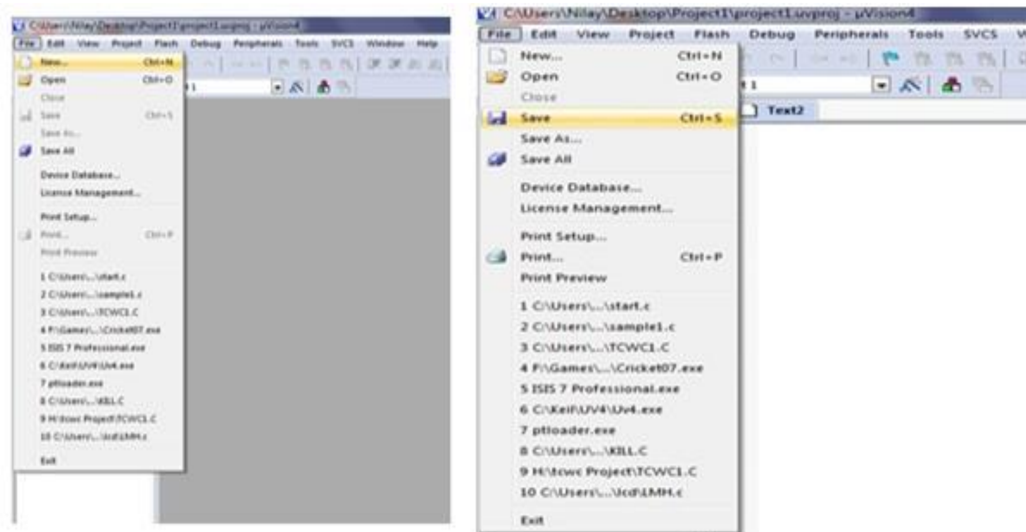
Step 2: After Creating project now **Select your device model**. Example.NXP-LPC2148 [You can change it later from project window.]



Step 3: so now your project is created and **Message** window will appear to add startup file of your Device click on **Yes** so it will be added to your project folder



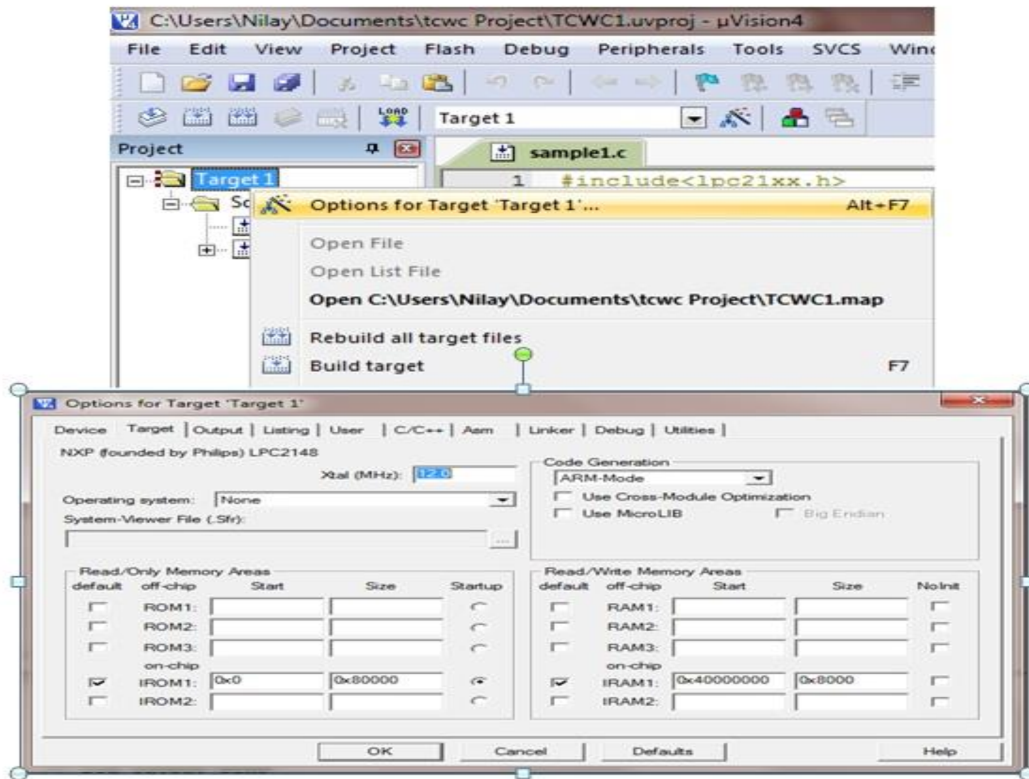
Step 4: Now go to File and create new file and save it with **.C** extension if you will write program in C language or save with **.asm** for **assembly** language.
i.e., **Led.c**



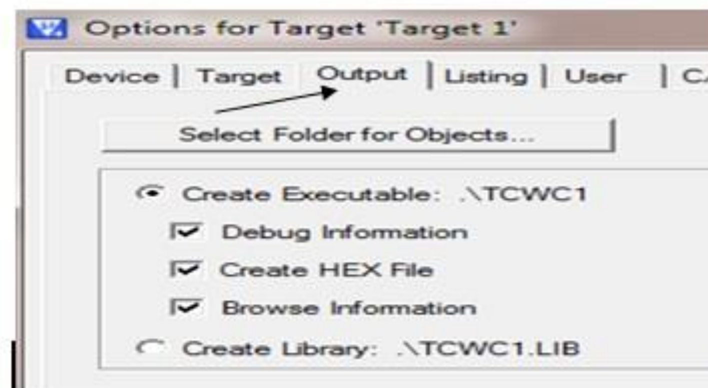
Step 5: Now write your program and save it again. You can try example given at end of this tutorial.

Step 6: After that on left you see project window [if it's not there….go to View tab and click on project window]
Now come on Project window.

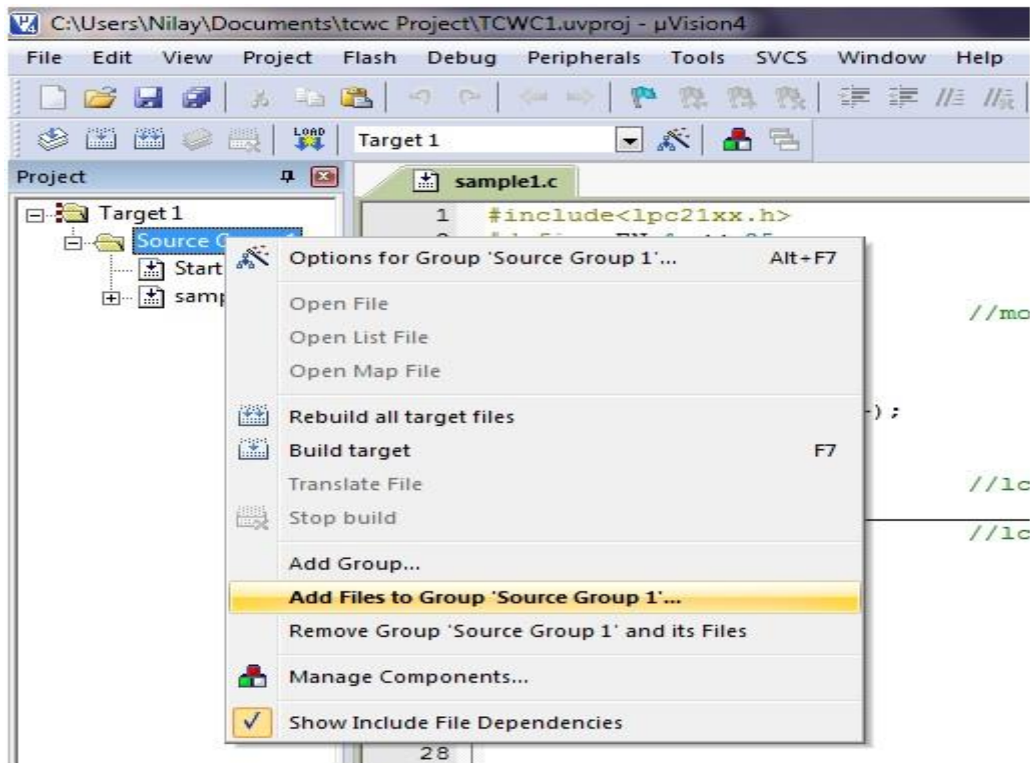
Right click on target and click on **options for target**
Here you can change your device also.



Click **output** tab here & check **create Hex file** if you want to generate hex file
Now click on ok so it will save changes



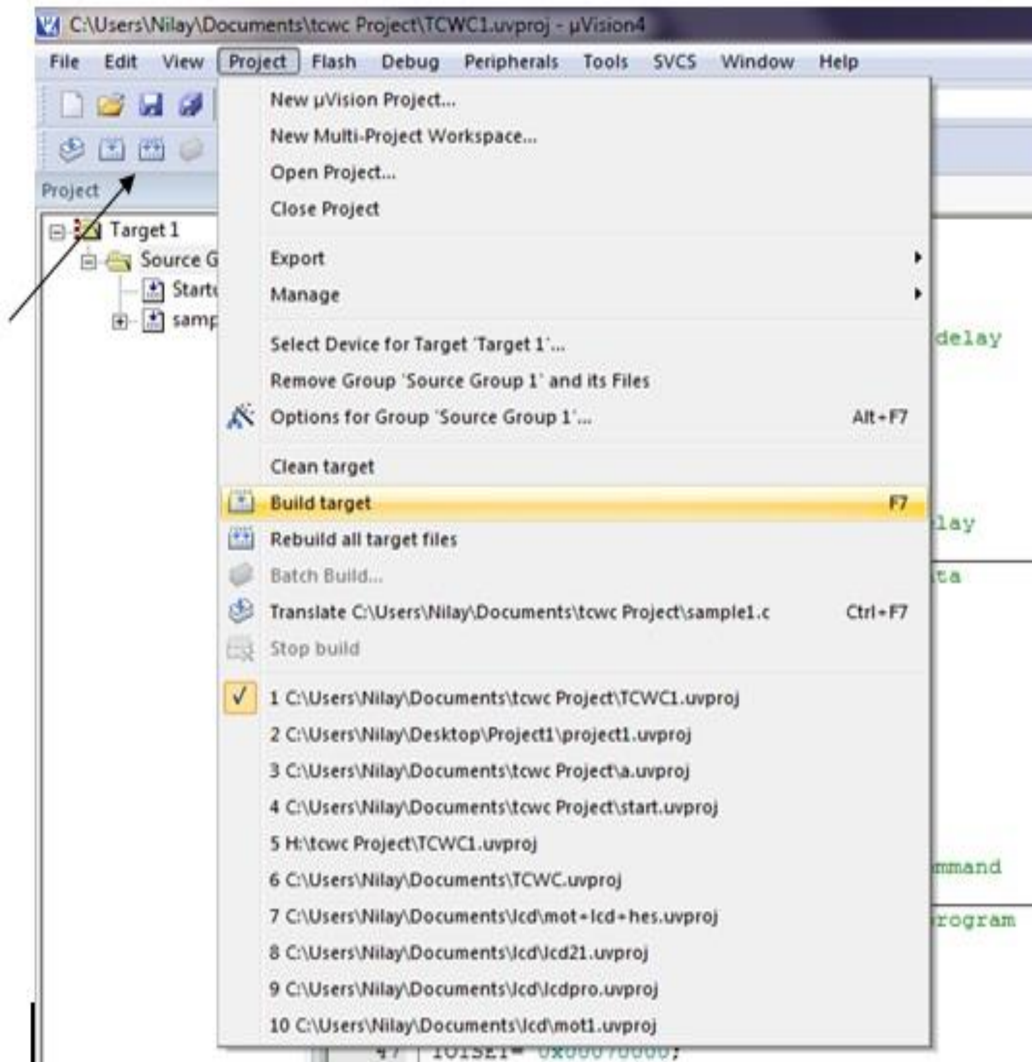
<u>**Step 7**</u>**:** Now Expand target and you will see source group
Right click on group and click on **Add files to source group**



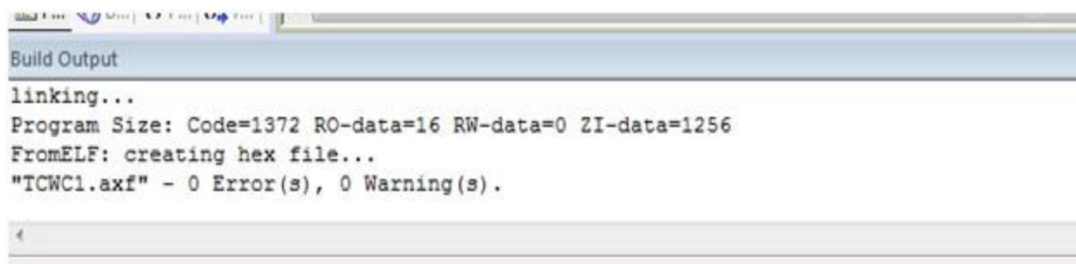
Now add your program file which you have written in C/assembly.

You can see program file added under source group.

Step 8: Now Click on **Build target**.You can find it under Project tab or in toolbar.It can also be done by pressing **F7** key.



Step 9: you can see Status of your program in **Build output** window
[If it’s not there go to view and click on Build output window]



Now you are done with your program.

BASIC TECHNIQUES FOR READING FROM PORT PINS:

As we saw in Chapter 3, control of the 8051 ports is carried out using 8-bit latches (SFRs). We can send some data to Port 1 as follows:

```
sfr P1 = 0x90; // Usually in header file
P1 = 0x0F; // Write 00001111 to Port 1
```

In exactly the same way, we can read from Port 1 as follows:

```
unsigned char Port_data;
P1 = 0xFF; // Set the port to 'read mode'
Port_data = P1; // Read from the port
```

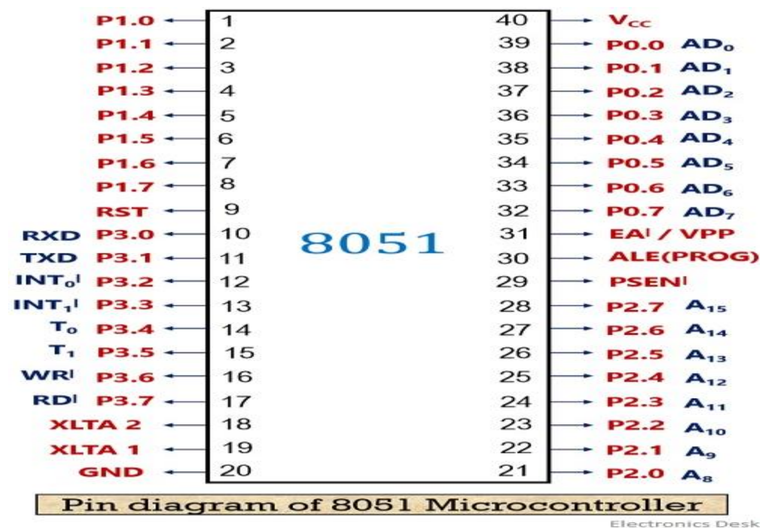
After the 8051 microcontroller is reset, the port latches all have the value 0xFF (11111111 in binary): that is, all the port-pin latches are set to values of '1'. It is tempting to assume that writing data to the port is therefore unnecessary, and that we can get away with the following version:

```
unsigned char Port_data;
// Assume nothing written to port since reset
// – DANGEROUS!!!
Port_data = P1;
```

The problem with this code is that, in simple test programs it works: this can lull the developer into a false sense of security. If, at a later date, someone modifies the program to include a routine for writing to all or part of the same port, this code will not generally work as required:

```
unsigned char Port_data;
P1 = 0x00;
. . .
// Assumes nothing written to port since reset
// – WON'T WORK
Port_data = P1;
```

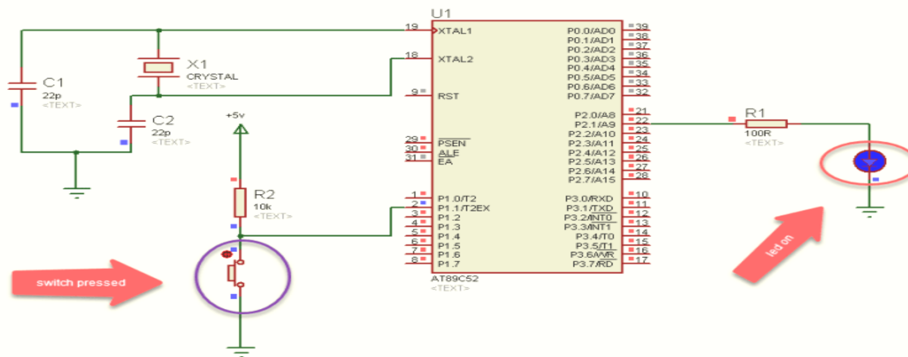
In most cases, initialization functions are used to set the port pins to a known state at the start of the program. Where this is not possible, it is safer to always write '1' to any port pin before reading from it.



;Toggle all bits of continuously.

```
      MOV A,#55
BACK: MOV P2,A
      ACALL DELAY
      CPL A ;complement(inv) reg.A
      SJMP BACK
```

Reading and writing bits:

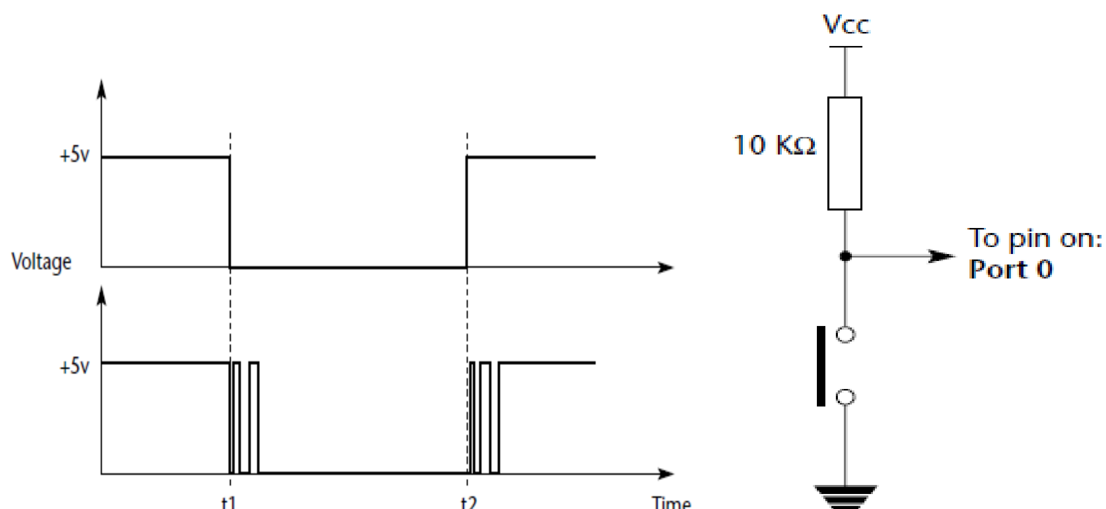
- Demonstrated how to read from or write to an entire port. However, suppose we have a **switch connected to Pin 1.1** and an **LED connected to Pin 2.1.**
- We might also have input and output devices connected to the other pins on Port 1.
- These pins may be used by totally different parts of the same system, and the code to access them may be produced by other team members, or other companies.
- It is therefore essential that we are able to read-from or write-to individual port pins without altering the values of other pins on the same port.
- We provided a simple example to illustrates how we can read from Pin 1.1, and write to Pin 2.1, without disrupting any other pins on this (or any other) port.
- 

```
#include<reg51.h>
sbit Led = P2^1;      //pin connected to toggle Led
sbit Switch =P1^1;  //Pin connected to toggle led
void main(void)
{
        Led = 0;          //configuring as output pin
        Switch = 1;      //Configuring as input pin
        while(1)        //Continuous monitor the status of the switch.
        {
        if(Switch == 0)
                {
                Led =1;         //Led On
                }
        else
                {
                Led =0; //Led Off
                }
        }
        return 0;
}
```

SWITCH BOUNCE:

In an ideal world, this change in voltage obtained by connecting a switch to the port pin of an 8051 microcontroller would take the form illustrated in Figure 4.8 (top). In practice, all mechanical switch contacts bounce (that is, turn on and off, repeatedly, for a short period of time) after the switch is closed or opened. As a result, the actual input waveform looks more like that shown in Figure 4.8 (bottom). Usually, switches bounce for less than 20 ms: however large mechanical switches exhibit bounce behaviour for 50 ms or more.

When you turn on the lights in your home or office with a mechanical switch, the switches will bounce. As far as humans are concerned, this bounce is imperceptible.

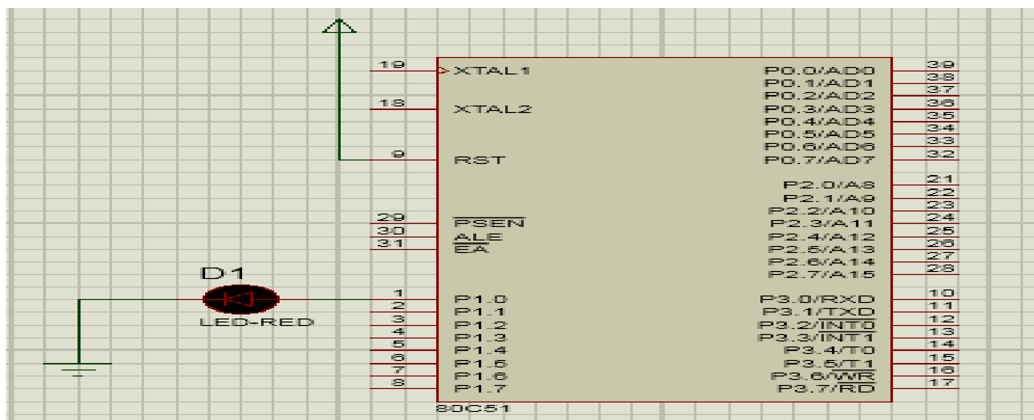


However, as far as the microcontroller is concerned, each ‘bounce’ is equivalent to one press and release of an ‘ideal’ switch. Without appropriate software design, this can give rise to a number of problems, not least:

_ Rather than reading ‘A’ from a keypad, we may read ‘AAAAA’.
_ Counting the number of times that a switch is pressed becomes extremely difficult.
_ If a switch is depressed once, and then released some time later, the ‘bounce’ may make it appear as if the switch has been pressed again (at the time of release).

APPLICATIONS:

LED INTERFACING WITH 8051 TO A SINGLE PIN:

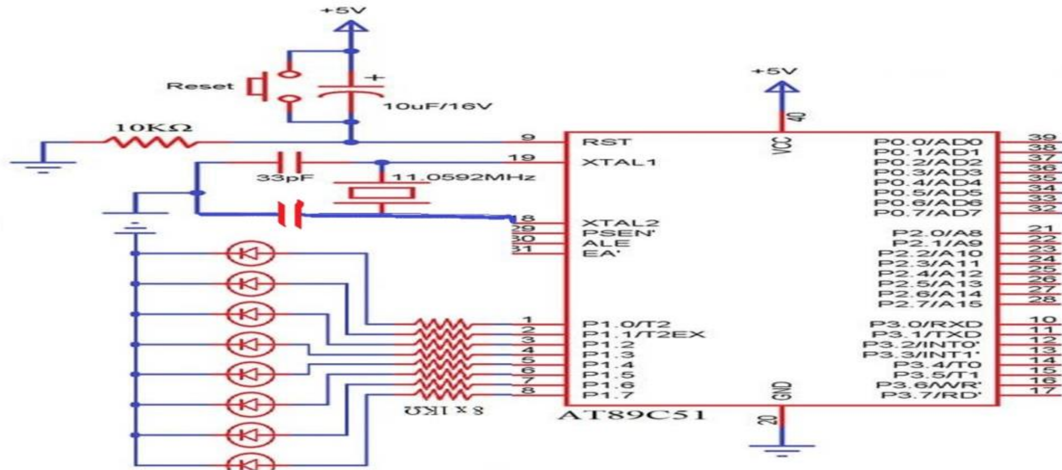


Program:

```
#include<reg51.h>     // special function register declarations
sbit LED = P2^0;      // Defining LED pin
void Delay(void);     // Function prototype declaration
void main (void)
{
while(1)              // infinite loop
{
LED = 0;              // LED ON
Delay();
LED = 1;              // LED OFF
Delay();
}
}
void Delay(void)
{
int j;
int i;
for(i=0;i<10;i++)
{
for(j=0;j<10000;j++)
{
```

```
}
    }
      }
```

LED's interfacing with Port, P1 of 8051:



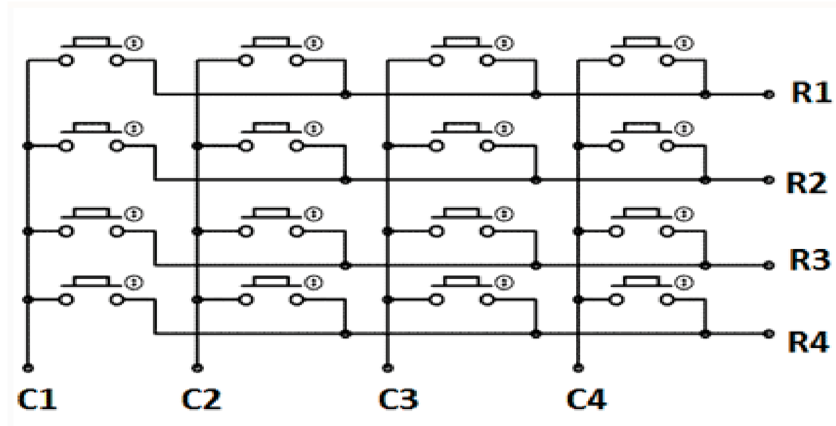
Program:

```
#include<REG51.H>
#define LEDPORT P1
void delay(unsigned int);
void main(void)
{
LEDPORT =0x00;
while(1)
{
LEDPORT = 0X00;
delay(250);
LEDPORT = 0xff;
delay(250);
}
}
void delay(unsigned int itime)
{
unsigned int i,j;
for(i=0;i<itime;i++)
{
for(j=0;j<250;j++);
}
}
```

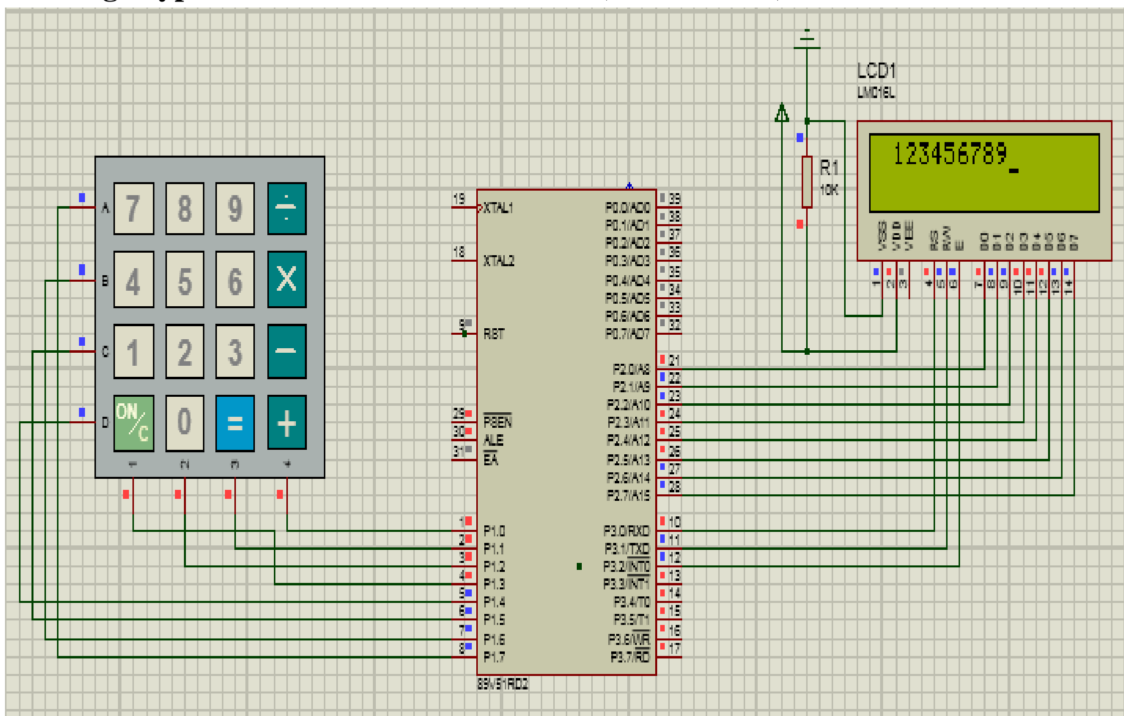
4X4 MATRIX KEYPAD INTERFACING WITH 8051 MICROCONTROLLER:

Keypads/Keyboards are widely used input devices being used in various electronics and embedded projects. They are used to take inputs in the form of numbers and alphabets, and feed the same into system for further processing. In this tutorial we are going to **interface a 4x4 matrix keypad/Keyboard with 8051 microcontroller**.

Before we interface the keypad with microcontroller, first we need to understand how it works. Matrix keypad consists of set of Push buttons, which are interconnected. Like in our case we are using 4X4 matrix keypad, in which there are 4 push buttons in each of four rows. And the terminals of the push buttons are connected according to diagram. In first row, one terminal of all the 4 push buttons are connected together and another terminal of 4 push buttons are representing each of 4 columns, same goes for each row. So we are getting 8 terminals to connect with a microcontroller.



Interfacing keypad with 8051 microcontroller (P89V51RD2)

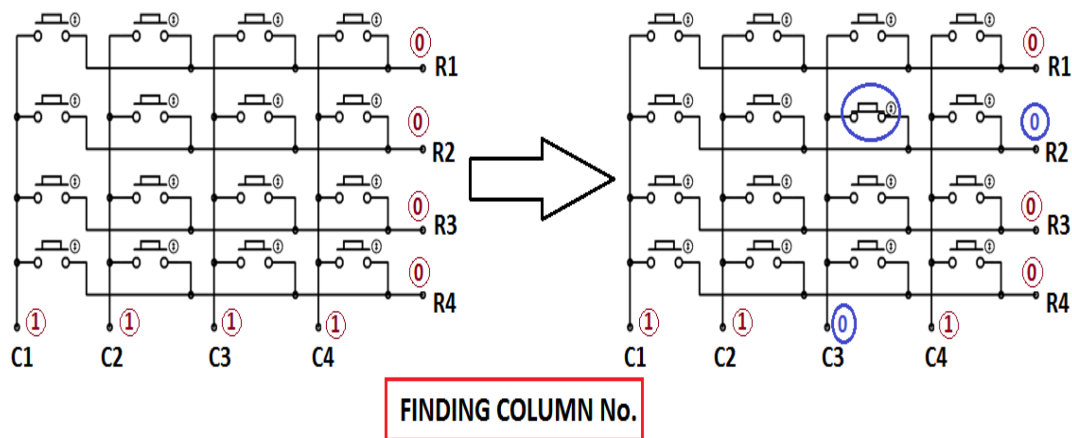


As shown in above circuit diagram, to interface Keypad, we need to connect 8 terminals of the keypad to any port (8 pins) of the microcontroller. Like we have connected keypad terminals to Port 1 of 8051. Whenever any button is pressed we need to get the location of the button, means the corresponding ROW an COLUMN no. Once we get the location of the button, we can print the character accordingly.

Now the question is how to get the location of the pressed button? I am going to explain this in below steps and also want you to look at the code:

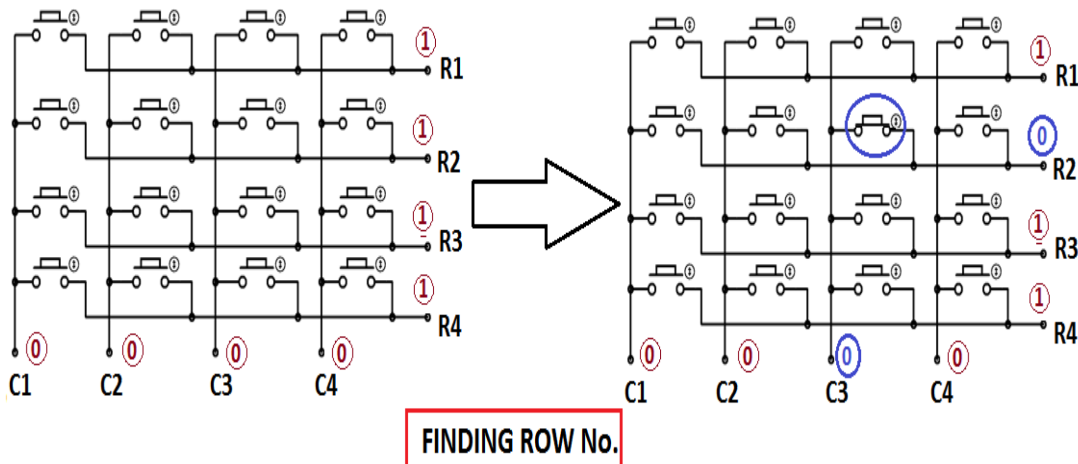
1. First we have made all the Rows to Logic level 0 and all the columns to Logic level 1.

2. Whenever we press a button, column and row corresponding to that button gets shorted and makes the corresponding column to logic level 0. Because that column becomes connected (shorted) to the row, which is at Logic level 0. So we get the column no. See main() function.



3. Now we need to find the Row no., so we have created four functions corresponding to each column. Like if any button of column one is pressed, we call function row_finder1(), to find the row no.

4. In row_finder1() function, we reversed the logic levels, means now all the Rows are 1 and columns are 0. Now Row of the pressed button should be 0 because it has become connected (shorted) to the column whose button is pressed, and all the columns are at 0 logic. So we have scanned all rows for 0.



5. So whenever we find the Row at logic 0, means that is the row of pressed button. So now we have column no (got in step 2) and row no., and we can print no. of that button using lcd_data function.

Same procedure follows for every button press, and we are using while(1), to continuously check, whether button is pressed or not.

Code:

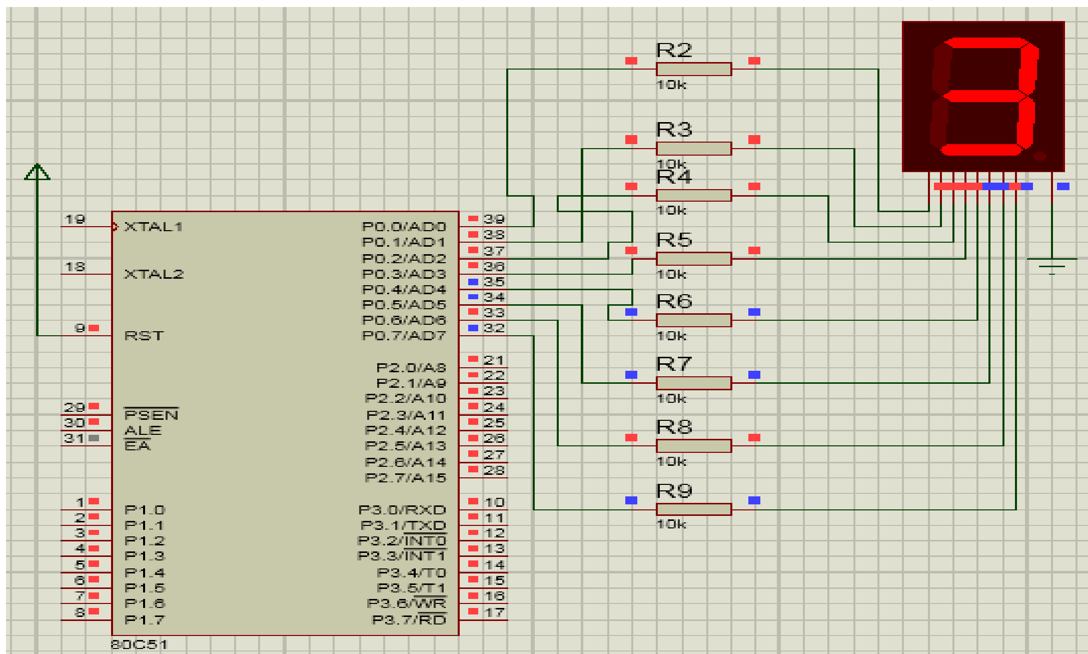
```
#include<reg51.h>
#define display_port P2          //Data pins connected to port 2 on microcontroller
sbit rs = P3^0;                  //RS pin connected to pin 2 of port 3
sbit rw = P3^1;                 // RW pin connected to pin 3 of port 3
sbit e =  P3^2;                 //E pin connected to pin 4 of port 3

sbit C4 = P1^0;                  // Connecting keypad to Port 1
sbit C3 = P1^1;
sbit C2 = P1^2;
sbit C1 = P1^3;
sbit R4 = P1^4;
sbit R3 = P1^5;
sbit R2 = P1^6;
sbit R1 = P1^7;

void msdelay(unsigned int time)  // Function for creating delay in milliseconds.
{
   unsigned i,j ;
   for(i=0;i<time;i++)
   for(j=0;j<1275;j++);
}
void lcd_cmd(unsigned char command)  //Function to send command instruction to LCD
{
   display_port = command;
```

```
   rs= 0;
   rw=0;
   e=1;
   msdelay(1);
   e=0;
}
void lcd_data(unsigned char disp_data)  //Function to send display data to LCD
{
   display_port = disp_data;
   rs= 1;
   rw=0;
   e=1;
   msdelay(1);
   e=0;
}
 void lcd_init()    //Function to prepare the LCD  and get it ready
{
   lcd_cmd(0x38);  // for using 2 lines and 5X7 matrix of LCD
   msdelay(10);
   lcd_cmd(0x0F);  // turn display ON, cursor blinking
   msdelay(10);
   lcd_cmd(0x01);  //clear screen
   msdelay(10);
   lcd_cmd(0x81);  // bring cursor to position 1 of line 1
   msdelay(10);
}

void row_finder1() //Function for finding the row for column 1
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('7');
if(R2==0)
lcd_data('4');
if(R3==0)
lcd_data('1');
if(R4==0)
lcd_data('N');
}

void row_finder2() //Function for finding the row for column 2
{
```

```
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('8');
if(R2==0)
lcd_data('5');
if(R3==0)
lcd_data('2');
if(R4==0)
lcd_data('0');
}
void row_finder3() //Function for finding the row for column 3
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('9');
if(R2==0)
lcd_data('6');
if(R3==0)
lcd_data('3');
if(R4==0)
lcd_data('=');
}

void row_finder4() //Function for finding the row for column 4
{
R1=R2=R3=R4=1;
C1=C2=C3=C4=0;

if(R1==0)
lcd_data('%');
if(R2==0)
lcd_data('*');
if(R3==0)
lcd_data('-');
if(R4==0)
lcd_data('+');
}

void main()
{
```

```
lcd_init();
while(1)
{
   msdelay(30);
   C1=C2=C3=C4=1;
     R1=R2=R3=R4=0;
     if(C1==0)
     row_finder1();
     else if(C2==0)
      row_finder2();
      else if(C3==0)
   row_finder3();
   else if(C4==0)
   row_finder4();
}

}
```

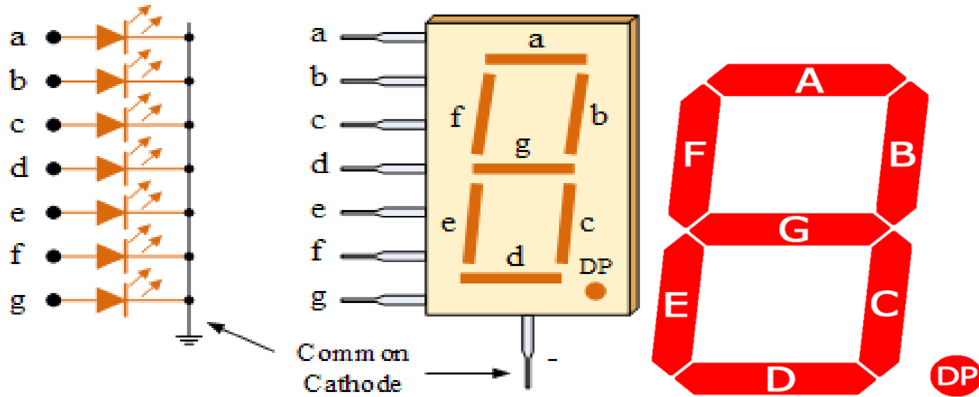
7 SEGMENT DISPLAY INTERFACING WITH 8051 MICROCONTROLLER:



This is how to interface a seven segment LED display to an 8051 microcontroller. 7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters. Knowledge about how to interface a seven segment display to a micro controller is very essential in designing embedded systems. Seven segment displays are of two types, ***common cathode and common anode.***

In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as '**com**' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) .

In common anode type, the anodes of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins.



Numbers	Common Cathode		Common Anode	
	(DP)GFEDCBA	HEX Code	(DP)GFEDCBA	HEX Code
0	00111111	0x3F	11000000	0xC0
1	00000110	0x06	11111001	0xF9
2	01011011	0x5B	10100100	0xA4
3	01001111	0x4F	10110000	0xB0
4	01100110	0x66	10011001	0x99
5	01101101	0x6D	10010010	0x92
6	011111101	0x7D	10000010	0x82
7	00000111	0x07	11111000	0xF8
8	01111111	0x7F	10000000	0x80
9	01101111	0x6F	10010000	0x90

Program:

```
/*Program to interface seven segment display unit.*/
#include <REG51.H>
#define LEDPORT P0
#define ZERO 0x3f
#define ONE 0x06
#define TWO 0x5b
#define THREE 0x4f
#define FOUR 0x66
#define FIVE 0x6d
#define SIX 0x7d
#define SEVEN 0x07
#define EIGHT 0x7f
#define NINE 0x6f
```

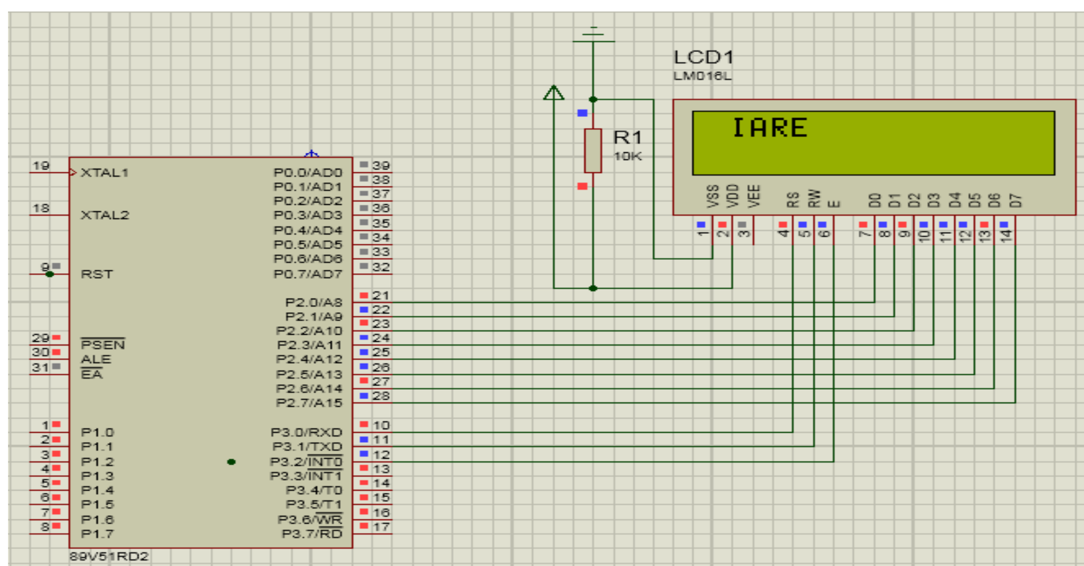
```
#define TEN 0x77
#define ELEVEN 0x7c
#define TWELVE 0x39
#define THIRTEEN 0x5e
#define FOURTEEN 0x79
#define FIFTEEN 0x71
void Delay(void);
void main (void)
{
while(1)
{
LEDPORT = ZERO;
Delay();
LEDPORT = ONE;
Delay();
LEDPORT = TWO;
Delay();
LEDPORT = THREE;
Delay();
LEDPORT = FOUR;
Delay();
LEDPORT = FIVE;
Delay();
LEDPORT = SIX;
Delay();
LEDPORT = SEVEN;
Delay();
LEDPORT = FOURTEEN;
Delay();
LEDPORT = FIFTEEN;
Delay();
}
}
void Delay(void)
{
int j; int i;
for(i=0;i<30;i++)
{
for(j=0;j<10000;j++)
{
}
}
}
```

LCD DISPLAY INTERFACING WITH 8051 MICROCONTROLLER:

In this, we will have brief discussion on how to interface 16×2 LCD module to P89V51RD2, which is an 8051 family microcontroller. We use LCD display for the displaying messages in a more interactive way to operate the system or displaying error messages etc. Interfacing 16×2 LCD with 8051 microcontroller is very easy if you understanding the working of LCD. 16×2 Liquid Crystal Display which will display the 32 characters at a time in two rows (16 characters in one row). Each character in the display is of size 5×7 pixel matrix.

PIN NO	NAME	FUNCTION
1	**VSS**	Ground pin
2	**VCC**	Power supply pin of 5V
3	**VEE**	Used for adjusting the contrast commonly attached to the potentiometer.
4	**RS**	RS is the register select pin used to write display data to the LCD (characters), this pin has to be high when writing the data to the LCD. During the initializing sequence and other commands this pin should low.
5	**R/W**	Reading and writing data to the LCD for reading the data R/W pin should be high (R/W=1) to write the data to LCD R/W pin should be low (R/W=0)
6	**E**	Enable pin is for starting or enabling the module. A high to low pulse of about 450ns pulse is given to this pin.
7	**DB0**	DB0-DB7 Data pins for giving data(normal data like numbers characters or command data) which is meant to be displayed
8	**DB1**	DB0-DB7 Data pins for giving data
9	**DB2**	DB0-DB7 Data pins for giving data

10	**DB3**	DB0-DB7 Data pins for giving data
11	**DB4**	DB0-DB7 Data pins for giving data
12	**DB5**	DB0-DB7 Data pins for giving data
13	**DB6**	DB0-DB7 Data pins for giving data
14	**DB7**	DB0-DB7 Data pins for giving data
15	**LED+**	Back light of the LCD which should be connected to Vcc
16	**LED-**	Back light of LCD which should be connected to ground.



Follow these simple steps for displaying a character or data
E=1; enable pin should be high
RS=1; Register select should be high
R/W=0; Read/Write pin should be low.

To send a command to the LCD just follows these steps:
E=1; enable pin should be high
RS=0; Register select should be low
R/W=0; Read/Write pin should be low.

Program:

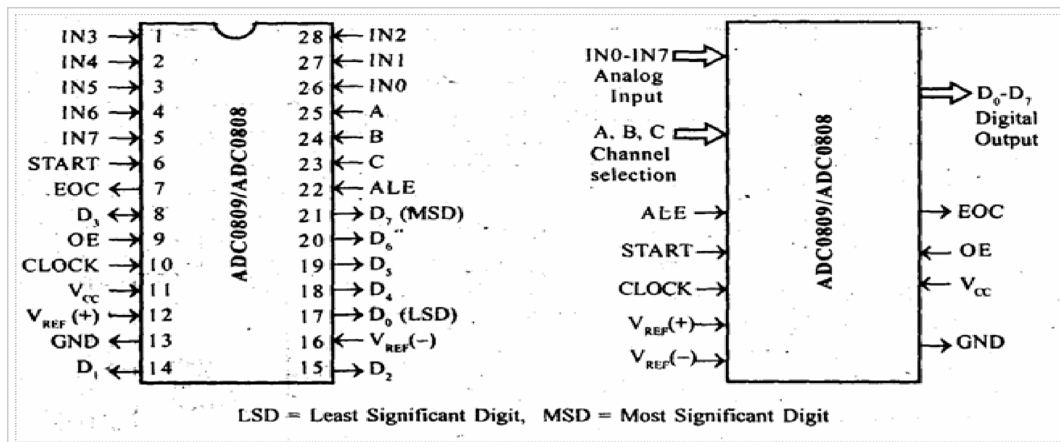
```
#include<reg51.h>
sbit rs=P3^0;
sbit rw=P3^1;
sbit en=P3^2;
```

```
void lcdcmd(unsigned char);
void lcddat (unsigned char);
void delay();
void main()
{
P2=0x00;
while(1)
{
lcdcmd(0x38);

delay();
lcdcmd(0x01);

delay();
lcdcmd(0x10);

delay();
lcdcmd(0x0c);

delay();
lcdcmd(0x81);

delay();
lcddat('I');
delay();
lcddat('A');

delay();
lcddat('R');

delay();
lcddat('E');

delay();
}
}
void lcdcmd(unsigned char val)
{
P2=val;
rs=0;
rw=0;
en=1;
delay();
en=0;
```

```
}
void lcddat(unsigned char val)
{
P2=val;
rs=1;
rw=0;
en=1;
delay();
en=0;
}
void delay()
{unsigned int i;
for(i=0;i<6000;i++);
}
```

ADC (ADC0808) INTERFACING WITH 8051 MICROCONTROLLER:

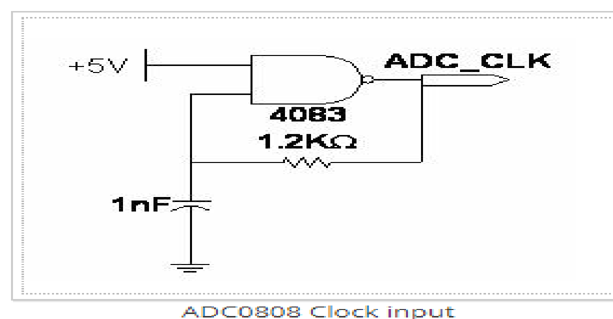
ADC0808/ADC0809 is an 8 channel 8-bit analog to digital converter. Unlike ADC0804 which has one Analog channel, this ADC has 8 multiplexed analog input channels. This tutorial will provide you basic information regarding this ADC, testing in free run mode and interfacing example with 8051 with sample program in C and assembly.



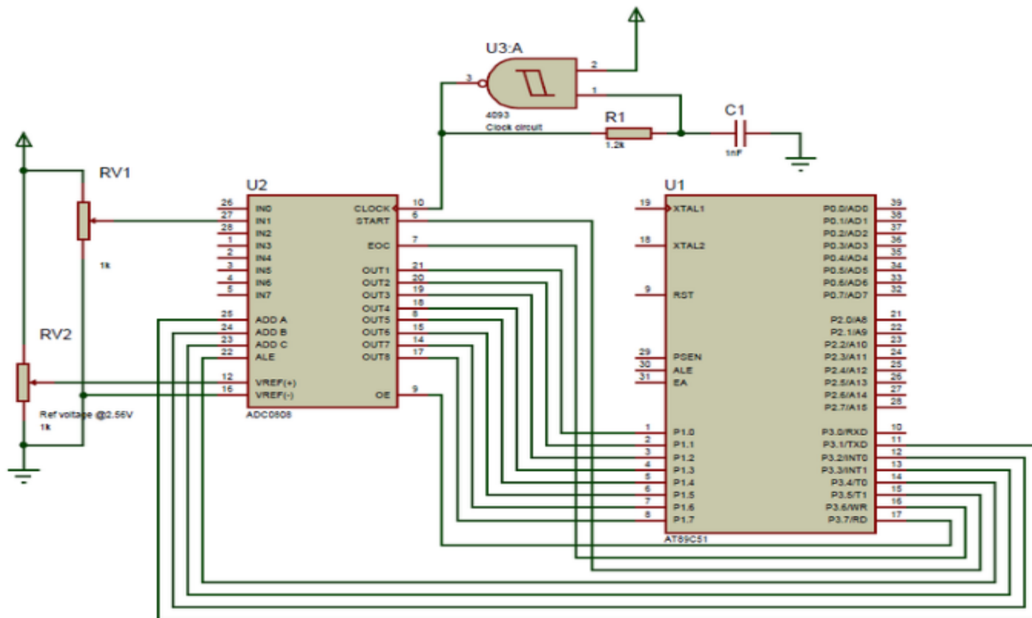
- IN0-IN7: Analog Input channels
- D0-D7: Data Lines
- A, B, C: Analog Channel select lines; A is LSB and C is MSB
- OE: Output enable signal
- ALE: Address Latch Enable
- EOC: End of Conversion signal
- V_{ref+}/V_{ref-}: Differential Reference voltage input
- Clock: External ADC clock input
- Normally analogue-to-digital converter (ADC) needs interfacing through a microprocessor to convert analogue data into digital format. This requires hardware and necessary

software, resulting in increased complexity and hence the total cost. The circuit of A-to-D converter shown here is configured around ADC 0808, avoiding the use of a microprocessor. The ADC 0808 is an 8-bit A-to-D converter, having data lines D0-D7. It works on the principle of successive approximation. It has a total of eight analogue input channels, out of which any one can be selected using address lines A, B and C. Here, in this case, input channel IN0 is selected by grounding A, B and C address lines.

- Usually the control signals EOC (end of conversion), SC (start conversion), ALE (address latch enable) and OE (output enable) are interfaced by means of a microprocessor. However, the circuit shown here is built to operate in its continuous mode without using any microprocessor. Therefore the input control signals ALE and OE, being active-high, are tied to Vcc (+5 volts). The input control signal SC, being active-low, initiates start of conversion at falling edge of the pulse, whereas the output signal EOC becomes high after completion of digitization. This EOC output is coupled to SC input, where falling edge of EOC output acts as SC input to direct the ADC to start the conversion.
- As the conversion starts, EOC signal goes high. At next clock pulse EOC output again goes low, and hence SC is enabled to start the next conversion. Thus, it provides continuous 8-bit digital output corresponding to instantaneous value of analogue input. The maximum level of analogue input voltage should be appropriately scaled down below positive reference (+5V) level.
- The ADC 0808 IC requires clock signal of typically 550 kHz, which can be easily derived from an Astable multivibrator, constructed using 7404 inverter gates. In order to visualize the digital output, the row of eight LEDs (LED1 through LED8) have been used, where in each LED is connected to respective data lines D0 through D7. Since ADC works in the continuous mode, it displays digital output as soon as analogue input is applied. The decimal equivalent digital output value D for a given analogue input voltage Vin can be calculated from the relationship.



ADC0808 Clock input



Program:

```
#include <reg51.h>
#define ALE P3_4
#define OE P3_7
#define START P3_5
#define EOC P3_6
#define SEL_A P3_1
#define SEL_B P3_2
#define SEL_C P3_3
#define ADC_DATA P1
void main()
{
unsigned char adc_data;
 /* Data port to input */
ADC_DATA = 0xFF;

EOC = 1;        /* EOC as input */
ALE = OE = START = 0;
 while (1)
{
/* Select channel 1 */
SEL_A = 1;    /* LSB */
SEL_B = 0;
SEL_C = 0;    /* MSB */

/* Latch channel select/address */
ALE = 1;
```

```
 /* Start conversion */
START = 1;
 ALE = 0;
START = 0;
/* Wait for end of conversion */
while (EOC == 1);
while (EOC == 0);
  /* Assert Read signal */
OE = 1;
/* Read Data */
adc_data = ADC_DATA;
OE = 0;
 /* Now adc data is stored */
/* start over for next conversion */
}
}
```

DAC INTERFACING WITH 8051 MICROCONTROLLER:

This section will show how to interface a DAC (digital-to-analog converter) to the 8051. Then we demonstrate how to generate a sine wave on the scope using the DAC.

Digital-to-analog (DAC) converter

The digital-to-analog converter (DAC) is a device widely used to convert digital pulses to analog signals. In this section we discuss the basics of interfacing a DAC to the 8051.

Recall from your digital electronics book the two methods of creating a DAC:

1. Binary weighted.
2. R/2R ladder.

The vast majority of integrated circuit DACs, including the MC1408 (DAC0808) used in this section use the R/2R method since it can achieve a much higher degree of precision. The first criterion for judging a DAC is its resolution, which is a function of the number of binary inputs. The common ones are 8, 10, and 12 bits. The number of data bit inputs decides the resolution of the DAC since the number of analog output levels is equal to 2″, where *n* is the number of data bit inputs.

Therefore, an 8-input DAC such as the DAC0808 provides 256 discrete voltage (or current) levels of output. Similarly, the 12-bit DAC provides 4096 discrete voltage levels. There are also
16-bit DACs, but they are more expensive.

MC1408 DAC (or DAC0808)

In the MC1408 (or DAC0808), the digital inputs are converted to current (I_{out}), and by connecting a resistor to the I_{out} pin, we convert the result to voltage.

The total current provided by the I_{out} pin is a function of the binary numbers at the DO – D7 inputs of the DAC0808 and the reference current (I_{ref}), and is as follows:

$$I_{out} = I_{ref}(\frac{D7}{2} + \frac{D6}{4} + \frac{D5}{8} + \frac{D4}{16} + \frac{D3}{32} + \frac{D2}{64} + \frac{D1}{128} + \frac{D0}{256})$$

Where DO is the LSB, D7 is the MSB for the inputs, and I_{ref} is the input current that must be applied to pin 14. The I_{ref} current is generally set to 2.0 mA. Figure shows the generation of current reference (setting I_{ref} = 2 mA) by using the standard 5-V power supply and IK and 1.5K-ohm standard resistors. Some DACs also use the zener diode (LM336), which overcomes any fluctuation associated

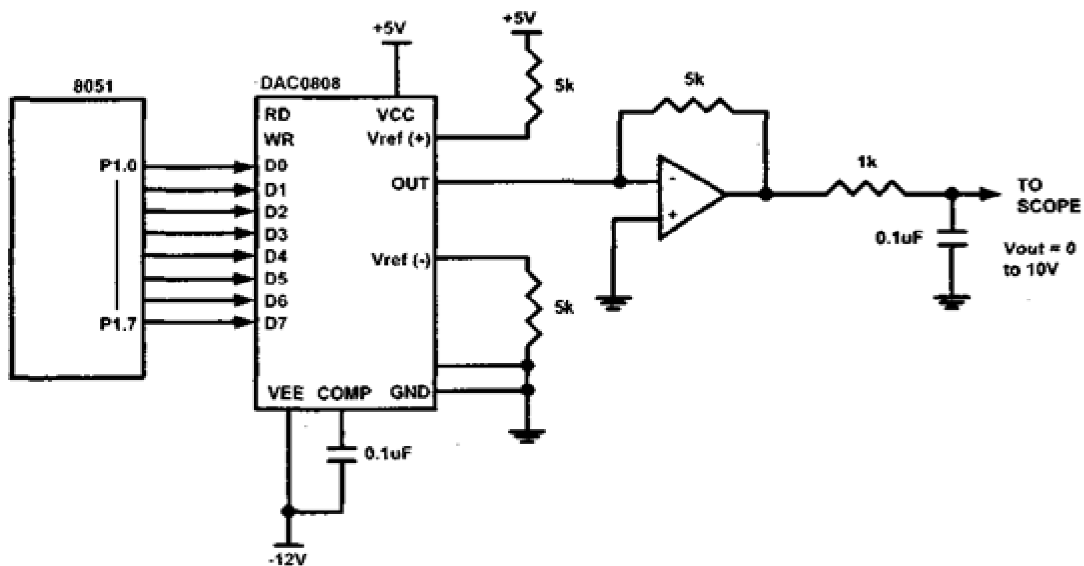


Figure: 8051 Connection to DAC808

Converting I_{out} to voltage in DAC0808

Ideally we connect the output pin I_{out} to a resistor, convert this current to voltage, and monitor the output on the scope. In real life, however, this can cause inaccuracy since the input resistance of the load where it is connected will also affect the output voltage. For this reason, the I_{ref} current output is isolated by connecting it to an op-amp such as the 741 with R_f = 5K ohms for the feedback resistor. Assuming that R = 5K ohms, by changing the binary input, the output voltage changes.

Generating a sine wave:

To generate a sine wave, we first need a table whose values represent the magnitude of the sine of angles between 0 and 360 degrees. The values for the sine function vary from -1.0 to +1.0 for 0- to 360-degree angles. Therefore, the table values are integer numbers representing the voltage magnitude for the sine of theta. This method ensures that only integer numbers are output to the DAC by the 8051 microcontroller. Table shows the angles, the sine values, the voltage magnitudes, and the integer values representing the voltage magnitude for each

angle (with 30-degree increments). To generate Table 13-7, we assumed the full-scale voltage of 10 V for DAC output. Full-scale output of the DAC is achieved when all the data inputs of the DAC are high. Therefore, to achieve the full-scale 10 V output, we use the following equation.

$$V_{out} = 5\text{ V} + (5 \times \sin\theta)$$

V_{out} of DAC for various angles is calculated and shown in Table 13-7. See Example 13-5 for verification of the calculations.

Angle θ (degrees)	**Sin θ**	**V_{out} (Voltage Magnitude) 5 V + (5 V x sin θ)**	**Values Sent to DAC (decimal) (Voltage Mag. X 25.6)**
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128

Program:

```
#include <reg51.h>
sfr DACDATA = Pl;
void main ()
{
        unsigned char WAVEVALUE [12]={128,192,238,255, 238,192,128,64, 17,0,17,64} ;
        unsigned char x ,
        while (1)
            {
              for(x=0;x<12;x++)
              {
              DACDATA = WAVEVALUE[x];
              }
            }

}
```

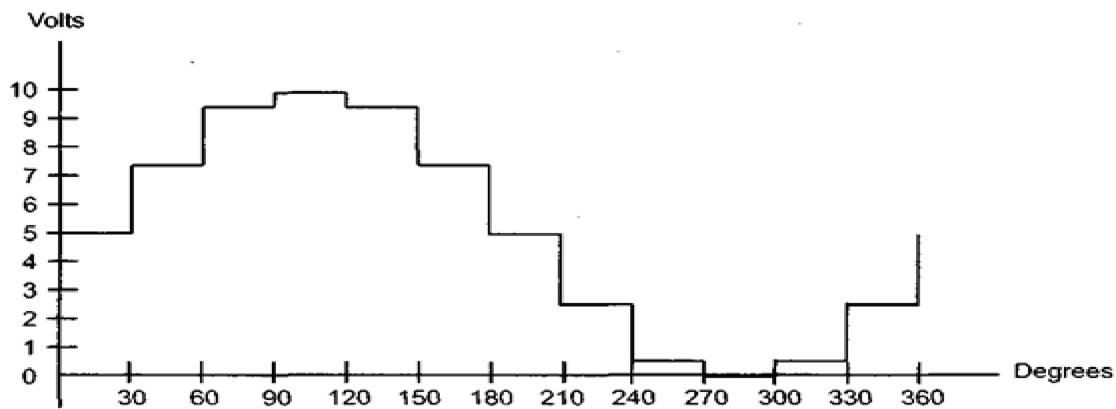


Figure: Angle vs. Voltage Magnitude for Sine Wave

MULTIPLE INTERRUPTS IN 8051 MICROCONTROLLER:

Interrupts vs. polling:

A single microcontroller can serve several devices. There are two ways to do that: interrupts or polling. In the ***interrupt method****, whenever any device needs* its service the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device.

The program associated with the interrupt is called the ***interrupt service routine (ISR)*** *or* ***interrupt handler.***

In polling, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller.

The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it.

The polling method cannot assign priority since it checks all devices in a round robin fashion.

SIX INTERRUPTS IN THE 8051 MICROCONTROLLER:

In reality, only five interrupts are available to the user in the 8051, but many manufacturers data sheets state that there are six interrupts since they include reset. The six interrupts in the 8051 are allocated as follows.

1. Reset. When the reset pin is activated, the 8051 jumps to address location 0000. This is the power-up reset.

2. Two interrupts are set aside for the timers: one for **Timer 0** and one for **Timer1**. Memory locations 000BH and 001BH in the interrupt vector table belong to Timer 0 and Timer 1, respectively.

3. Two interrupts are set aside for hardware **external hardware interrupts**, Pin numbers 12

(P3.2) and 13 (P3.3) in port 3 are for the external hardware interrupts **INT 0** and **INT 1**, respectively. These external interrupts are also referred to as EX 1 and EX 2. Memory locations 0003H and 0013H In the interrupt vector table are assigned to INT0 and INT1, respectively.
4. Serial communication has a single interrupt that belongs to both receive and transmit. The interrupt vector table location 0023H belongs to this interrupt.

Interrupts	Memory Location	Pin	Flag Clearing
Reset	0000	9	Auto
Timer0	000B		Auto
Timer1	001B		Auto
INT0	0003	12	Auto
INT1	0013	13	Auto
Serial com	0023		Cleared by programmer

Table: Interrupt Vector Table for the 8051

Enabling and Disabling an interrupt:
Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them. There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts.
Figure shows the IE register. Note that IE is a bit-addressable register.
From figure notice that bit D7 in the IE register is called EA (enable all). This must be set to 1 in order for the rest of the register to take effect. D6 is unused. D5 is used by the 8052. The D4 bit is for the serial interrupt, and so on.

Steps in enabling an interrupt:
To enable an interrupts, we take the following steps:
1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take the effect.
2. If EA =1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA=0, no interrupt will be responded to, even if the associated bit in the IE register is high.

IE (Interrupt Enable) Register:

- This register is responsible for enabling and disabling the interrupt.
- EA register is set to 1 for enabling interrupts and
- EA register is set to 0 for disabling the interrupts.
- Its bit sequence and their meanings are shown in the following figure.

EA(MSB)			ES	ET1	EX1	ET0	EX0(LSB)
IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0

INTERRUPT ENABLE REGISTER AT A8H

EA	IE.7	It disables all interrupts. When EA = 0 no interrupt will be acknowledged and When EA = 1 enables the interrupt individually.
-	IE.6	Reserved for future use.
-	IE.5	Reserved for future use.
ES	IE.4	Enables/disables serial port interrupt.
ET1	IE.3	Enables/disables timer1 overflow interrupt.
EX1	IE.2	Enables/disables external interrupt1.
ET0	IE.1	Enables/disables timer0 overflow interrupt.
EX0	IE.0	Enables/disables external interrupt0.

SERIAL COMMUNICATION PROGRAMMING:

Serial Communication can be

- Asynchronous
- Synchronous

Synchronous Communication:
Synchronous methods transfer a block of data (characters) at a time
The events are referenced to a clock
Example: SPI bus, I2C bus
Asynchronous Communication:
Asynchronous methods transfer a single byte at a time
There is no clock. The bytes are separated by start and stop bits.
Example: UART

1. Serial port programming in assembly

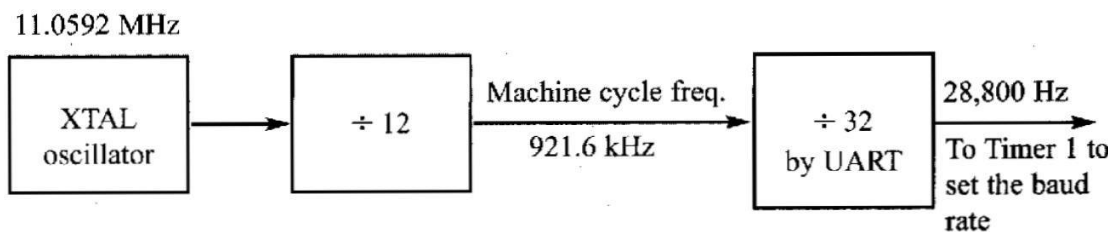
Since IBM PC/compatible computers are so widely used to communicate with 8051-based systems, serial communications of the 8051 with the COM port of the PC will be

emphasized. To allow data transfer between the PC and an 8051 system without any error, we must make sure that the baud rate of the 8051 system matches the baud rate of the PC"s COM port.

Baud rate in the 8051

The 8051 transfers and receives data serially at many different baud rates. Serial communications of the 8051 is established with PC through the COM port. It must make sure that the baud rate of the 8051 system matches the baud rate of the PC's COM port/ any system to be interfaced. The baud rate in the 8051 is programmable. This is done with the help of Timer. When used for serial port, the frequency of timer tick is determined by (XTAL/12)/32 and 1 bit is transmitted for each timer period (the time duration from timer start to timer expiry).

The Relationship between the crystal frequency and the baud rate in the 8051 is that the 8051 divides the crystal frequency by 12 to get the machine cycle frequency which is shown in figure1. Here the oscillator is XTAL = 11.0592 MHz, the machine cycle frequency is 921.6 kHz. 8051's UART divides the machine cycle frequency of 921.6 kHz by 32 once more before it is used by Timer 1 to set the baud rate. 921.6 kHz divided by 32 gives 28,800 Hz. Timer 1 must be programmed in mode 2, that is 8-bit, auto-reload.



Baud rate supported by Pentium / IBM 486 PC are

110
150
300
600
1200
2400
4800
9600
19200

Calculation of baud rate:

In serial communication if data transferred with a baud rate of 9600 and XTAL used is 11.0592 then following is the steps followed to find the TH1 value to be loaded.

Clock frequency of timer clock: f = (11.0592 MHz / 12)/32 = 28,800Hz

Time period of each clock tick: T0 = 1/f = 1/28800
Duration of timer : n*T0 (n is the number of clock ticks)
9600 baud ->duration of 1 symbol: 1/9600
1/9600 = n*T0 = n*1/28800

n = f/9600 = 28800/9600 = 3 ->TH1 =-3

Similarly, for baud 2400
n = f/2400 = 12 ->TH1 = -12

Example: set baud rate at 9600
MOV TMOD, #20H ; timer 1,mode 2(auto reload) MOV
TH1, #-3 ; To set 9600 baud rate
SETB TR1; start timer 1

Baud rate selection

Baud rate is selected by timer1 and when Timer 1 is used to set the baud rate it must be programmed in mode 2 that is 8-bit, auto-reload. To get baud rates compatible with the PC, we must load TH1 with the values shown in Table 1.

Table.1 Timer 1 THI register values for different baud rates

Baud Rate	TH1 (Decimal)	TH1 (Hex)
9600	–3	FD
4800	–6	FA
2400	–12	F4
1200	–24	E8

Note: XTAL = 11.0592 MHz.

Registers for serial communication

SBUF (serial buffer) register:

It is an 8 bit register used solely for serial communication in the 8051.A byte of data to be transferred via the TxD line must be placed in the SBUF register. SBUF holds the byte of data when it is received by the RxD line. It can be accessed like any other register

MOV SBUF, #'D'   ; load SBUF=44H, ASCII for 'D,,

MOV SBUF, A   ; copy accumulator into

SBUF MOV A, SBUF   ; copy SBUF into accumulator
when a byte is written, it is framed with the start and stop bits and transferred serially via the TxD pin. when the bits are received serially via RxD, it is deframed by eliminating the stop and start bits, making a byte out of the data received, and then placing it in the SBUF.

SCON (serial control) register:

It is an 8 bit register used to program start bit, stop bit, and data bits of data framing, among other things.

Figure 2. SCON register

The first two bits are SM0, SM1 which is the serial port mode bits. It is used to specify framing format, how to calculate baud. For example if (SM0, SM1) = (0,1), mode 1: 8-bit data, 1 start bit, 1 stop bit, variable baud rate can be set by timer. The other three modes are rarely used and they are (SM0,SM1) = (0,0), mode 0: fixed baud = XTAL/12,(SM0, SM1) = (1,0), mode 2: 9-bit data, fixed baud,(SM0, SM1) = (1, 1), mode 3: 9-bit data, variable baud. The third bit is used to select the type of processor used for communication. If SM2 is 0 means it is single processor communication. If SM2 is 1, then it is multiprocessor communication. The fourth bit REN is Receive Enable which is used to enable/disable reception. If REN=1,then 8051 will accept incoming data from serial port. If REN=0, then the receiver is disabled. E.g. SETB REN,CLR REN, SETB SCON.4, CLR SCON.4.

The fifth bit is TB8 which is used by modes 2 and 3 for the 8-bit transmission. When mode 1 is used the pin TB8 should be cleared. The sixth bit RB8 is used by modes 2 and 3 for the reception of bit 8. It is used by mode1 to store the stop bit. The seventh bit is TI which is the Transmit Interrupt. When 8051 finishes the transfer of the 8-bit character, it sets TI to "1" to indicate that it is ready to transfer the next character. The TI is raised at the beginning of the stop bit. The last bit is the RI which is the receive interrupt. When 8051 receives a character,the UART removes start bit and stop bit. The UART puts the 8-bit character in SBUF. RI is set to „1" to indicate that a new byte is ready to be picked up in SBUF.RI is raised halfway through the stop bit

Steps to send data serially:

1. Set baud rate by loading TMOD register with the value 20H, this indicating timer 1 in mode
   2 (8-bit auto-reload) to set baud rate
2. The TH1 is loaded with proper values to set baud rate for serial data transfer
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is
   framed with start and stop bits
4. TR1 is set to 1 to start timer 1
5. TI is cleared by CLR TI instruction
6. The character byte to be transferred serially is written into SBUF register
7. The TI flag bit is monitored with the use of instruction JNB TI,xx to see if the character has been transferred completely
8. To transfer the next byte, go to step 5.

Program to transfer letter “D” serially at 9800baud, continuously:

```
MOV TMOD,#20H              ; timer 1,mode 2(auto reload)
MOV TH1, #-3               ; 9600 baud rate
MOV SCON, #50H             ; 8-bit, 1 stop, REN enabled
SETB TR1                   ; start timer 1
AGAIN: MOV SBUF, #”D”      ; letter “D” to transfer
HERE: JNB TI, HERE         ; wait for the last bit
CLR TI                     ;clear TI for next char
SJMP AGAIN                 ; keep sending A
```

Importance of the TI flag:

Check the TI flag bit, we know whether or not 8051 is ready to transfer another byte. TI flag bit is raised by the 8051 after transfer of data. TI flag is cleared by the programmer by instruction like “CLR TI”. When writing a byte into SBUF, before the TI flag bit is raised, it may lead to loss of a portion of the byte being transferred.

Steps to receive data serially:

1. Set baud rate by loading TMOD register with the value 20H, this indicating timer 1 in mode 2 (8-bit auto-reload) to set baud rate
2. The TH1 is loaded with proper values to set baud rate
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits
4. TR1 is set to 1 to start timer 1
5. RI is cleared by CLR RI instruction
6. The RI flag bit is monitored with the use of instruction JNB RI,xx to see if an entire character has been received yet
7. When RI is raised, SBUF has the byte; its contents are moved into a safe place
8. To receive next character, go to step 5

Program to receive bytes of data serially, and put them in P2, set the baud rate at 9600, 8-bit data, and 1 stop bit:

```
        MOV TMOD, #20H     ; timer 1,mode 2(auto reload)
        MOV TH1, #-3       ; 9600 baud rate
        MOV SCON, #50H     ; 8-bit, 1 stop, REN enabled
        SETB TR1           ; start timer 1
HERE: JNB RI, HERE         ; wait for char to come in
MOV A, SBUF                ; saving incoming byte in A
        MOV P2, A          ; send to port 1
```

```
        CLR RI              ; get ready to receive next byte
        SJMP HERE           ; keep getting data
```

Importance of the RI flag bit:

It receives the start bit, next bit is the first bit of the character about to be received. When the last bit is received, a byte is formed and placed in SBUF. when stop bit is received, it makes RI = 1 indicating entire character byte has been received and can be read before overwritten by next data. When RI=1, received byte is in the SBUF register, copy SBUF contents to a safe place. After the SBUF contents are copied the RI flag bit must be cleared to 0.

Increasing the baud rate:

Baud rate can be increase by two ways-

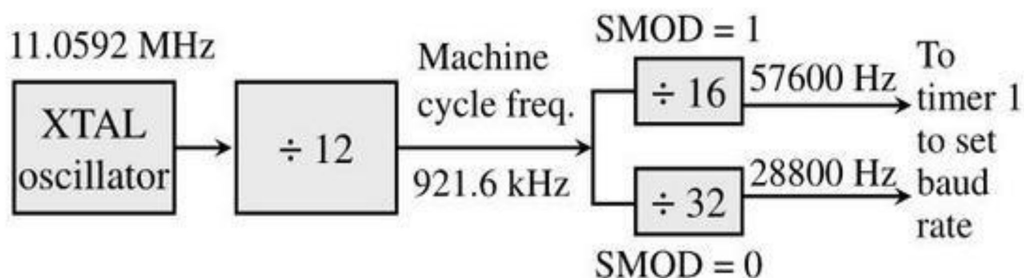
1. Increasing frequency of crystal
2. Change bit in PCON register

PCON

It is 8-bit register. When 8051 is powered up, SMOD is zero. By setting the SMOD, baud rate can be doubled. If SMOD = 0 (which is its value on reset), the baud rate is 1/64 the oscillator frequency. If SMOD = 1, the baud rate is 1/32 the oscillator frequency.

Table 2. Comparison of Baud rate

TH1(decimal)	TH1 (hex)	SMOD=0	SMOD=1
-3	FD	9600	19200
-6	FA	4800	9600
-12	F4	2400	4800



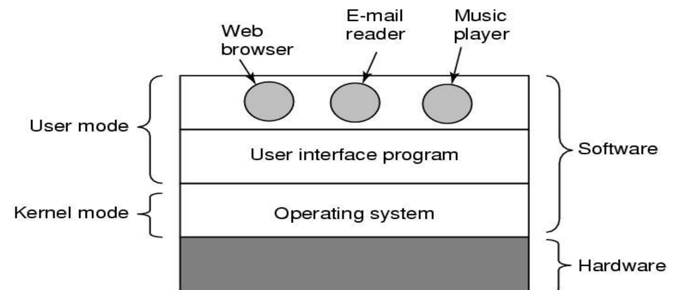
UNIT-III

RTOS BASED EMBEDDED SYSTEM DESIGN

Operating System Basics:

- The Operating System acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services

- OS manages the system resources and makes them available to the user applications/tasks on a need basis



The primary functions of an Operating system is

- Make the system convenient to use
- Organize and manage the system resources efficiently andcorrectly

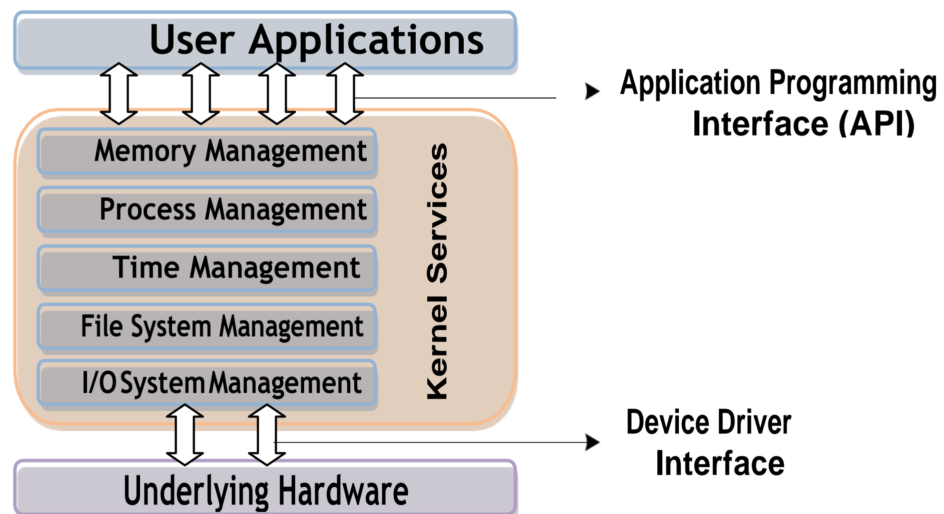


Figure 1: The Architecture of Operating System

The Kernel:

- The kernel is the core of the operating system
- It is responsible for managing the system resources and the communication among the hardware and other system services
- Kernel acts as the abstraction layer between system resources and user applications
- Kernel contains a set of system libraries and services.
- For a general purpose OS, the kernel contains different services like
    - Process Management
    - Primary Memory Management
    - File System management
    - I/O System (Device) Management
    - Secondary Storage Management
    - Protection
    - Time management
    - Interrupt Handling

Kernel Space and User Space:

- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorized access by user programs/applications
- The memory space at which the kernel code is located is known as '*Kernel Space*'
- All user applications are loaded to a specific area of primary memory and this memory area is referred as '*User Space*'
- The partitioning of memory into kernel and user space is purely Operating System dependent
- An operating system with virtual memory support, loads the user applications into its corresponding virtual memory space with demand paging technique. Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory

Monolithic Kernel:

- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a single kernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application
- LINUX, SOLARIS, MS-DOS kernels are examples of monolithic kernel

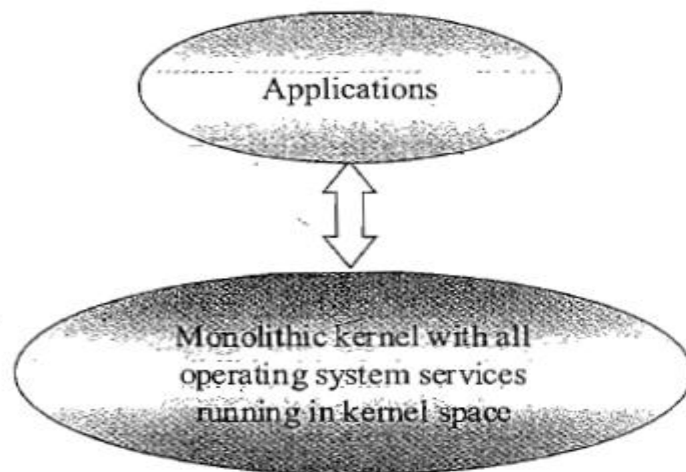


Figure 2: The Monolithic Kernel Model

Microkernel:

The microkernel design incorporates only the essential set of Operating System services into the kernel.

- The rest of the Operating System services are implemented in programs known as 'Servers' which runs in user space.
- The kernel design is highly modular provides OS-neutral abstraction.
- Memory management, process management, timer systems and interrupt handlers are

examples of essential services, which forms the part of the microkernel.

Examples for microkernel: **QNX, Minix 3 kernels.**

Benefits of Microkernel:

- **Robustness:** If a problem is encountered in any services in server can reconfigured and re-started without the need for re-starting the entire OS.
- **Configurability:** Any services , which run as 'server' application can be changed without need to restart the whole system.

Types of Operating Systems:

Depending on the type of kernel and kernel services, purpose and type of computing systems where the OS is deployed and the responsiveness to applications, Operating Systems are classified into

1. **General Purpose Operating System (GPOS).**
2. **Real Time Purpose Operating System (RTOS).**

1. General Purpose Operating System (GPOS):

i. Operating Systems, which are deployed in general computing systems

ii. The kernel is more generalized and contains all the required servicesto execute generic applications

iii. Need not be deterministic in execution behavior

iv. May inject random delays into application software and thus causeslow responsiveness of an application at unexpected times

v. Usually deployed in computing systems where deterministic behavior isnot an important criterion

vi. Personal Computer/Desktop system is a typical example for a systemwhere GPOSs are deployed.

vii. Windows XP/MS-DOS etc are examples of General Purpose Operating System

2. Real Time Purpose Operating System (RTOS):

i. Operating Systems, which are deployed in embedded systems demanding real-time response

ii. Deterministic in execution behavior. Consumes only known amount of time for kernel applications

iii. Implements scheduling policies for executing the highest priority task/application always

iv. Implements policies and rules concerning time-critical allocation of a system's resources

v. Windows CE, QNX, VxWorks , MicroC/OS-II etc are examples of Real Time Operating Systems (RTOS)

The Real Time Kernel: The kernel of a Real Time Operating System is referred as Real Time kernel. In complement to the conventional OS kernel, the Real Time kernel is highly specialized and it contains only the minimal set of services required for running the user applications/tasks. The basic functions of a Real Time kernel are

a) Task/Process management

b) Task/Process scheduling

c) Task/Process synchronization

d) Error/Exception handling

e) Memory Management

f) Interrupt handling

g) Time management

- **Real Time Kernel Task/Process Management:** Deals with setting up the memory space for the tasks, loading the task's code into the memory space, allocating system resources, setting up a Task Control Block (TCB) for the task and task/process termination/deletion. A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information

  - *Task ID:* Task Identification Number

- ❖ *Task State:* The current state of the task. (E.g. State= 'Ready' for atask which is ready to execute)

- ❖ *Task Type*: Task type. Indicates what is the type for this task. The taskcan be a hard real time or soft real time or background task.

- ❖ *Task Priority*: Task priority (E.g. Task priority =1 for task with priority =1)

- ❖ *Task Context Pointer*: Context pointer. Pointer for context saving
- ❖ *Task Memory Pointers*: Pointers to the code memory, data memoryand stack memory for the task

- ❖ *Task System Resource Pointers*: Pointers to system resources (semaphores, mutex etc) used by the task

- ❖ *Task Pointers:* Pointers to other TCBs (TCBs for preceding, next and waiting tasks)

- ❖ *Other Parameters* Other relevant task parameters

The parameters and implementation of the TCB is kernel dependent. The TCB parameters vary across different kernels, based on the task management implementation

- **Task/Process Scheduling:** Deals with sharing the CPU among various tasks/processes. A kernel application called '*Scheduler*' handles the task scheduling. Scheduler is nothing but an algorithm implementation, which performs the efficient and optimal scheduling of tasks to provide a deterministic behavior.

- **Task/Process Synchronization:** Deals with synchronizing the concurrent access of a resource, which is shared across multiple tasks and the communication between various tasks.

- **Error/Exception handling:** Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks. Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions. Errors/Exceptions can happen at the kernel level services or at task level. Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception. The OS kernel gives the information about the error in the form of a system call (API).

Memory Management:

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems. The memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than un- initialized memory block)

- Since predictable timing and deterministic behavior are the primary focus for an RTOS, RTOS achieves this by compromising the effectiveness of memory allocation

- RTOS generally uses '*block*' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

- RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a '*Free buffer Queue*'.

- Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads

- RTOS kernels assume that the whole design is proven correct and protection is unnecessary. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe* mode when an illegal memory access occurs

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems

- A few RTOS kernels implement *Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).

- In the '*block*' based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

- The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behavior of the RTOS kernel untouched.

Interrupt Handling:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.

- Interrupts can be either *Synchronous* or *Asynchronous*.
- Interrupts which occurs in sync with the currently executing task is known as *Synchronous* interrupts. Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.
- For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.
- Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.
- The interrupts generated by external devices (by asserting the Interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, and serial data reception / transmission interrupts etc are examples for asynchronous interrupts.
- For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.
- Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually.
- Most of the RTOS kernel implements '*Nested Interrupts*' architecture. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a higher priority interrupt.

Time Management:

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- Accurate time management is essential for providing precise time reference for all applications
- The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (hardware timer)
- The hardware timer is programmed to interrupt the processor/controller at a fixed rate. This timer interrupt is referred as 'Timer tick'
- The 'Timer tick' is taken as the timing reference by the kernel. The 'Timer tick'

interval may vary depending on the hardware timer. Usually the 'Timer tick' varies in the microseconds range

- ❖ The time parameters for tasks are expressed as the multiples of the 'Timer tick'
- ❖ The System time is updated based on the 'Timer tick'
- ❖ If the System time register is 32 bits wide and the 'Timer tick' interval

  is 1microsecond, the System time register will reset in

  232 * 10-6/ (24 * 60 * 60) = 49700 Days =~ 0.0497 Days = 1.19 Hours

If the 'Timer tick' interval is 1 millisecond, the System time register will reset in

232 * 10-3 / (24 * 60 * 60) = 497 Days = 49.7 Days =~ 50 Days

The '*Timer tick*' interrupt is handled by the 'Timer Interrupt' handler of kernel. The '*Timer tick*' interrupt can be utilized for implementing the following actions.

- Save the current context (Context of the currently executing task)
- Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register
- Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = '*count up*' and decrement registers with count direction setting = '*count down*')
- Activate the periodic tasks, which are in the idle state
- Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
- Delete all the terminated tasks and their associated data structures (TCBs)
- Load the context for the first task in the ready queue. Due to the re- scheduling, the ready task might be changed to a new one from the task, which was pre-empted by the 'Timer Interrupt' task

Hard Real-time System:

- ❖ A Real Time Operating Systems which strictly adheres to the timing constraints for a task.
- ❖ A Hard Real Time system must meet the deadlines for a task without any slippage

- Missing any deadline may produce catastrophic results for Hard Real Time Systems, including permanent data lose and irrecoverable damages to the system/users

- Emphasize on the principle '*A late answer is a wrong answer*'

- Air bag control systems and Anti-lock Brake Systems (ABS) of vehicles are typical examples of Hard Real Time Systems

- As a rule of thumb, Hard Real Time Systems does not implement the virtual memory model for handling the memory. This eliminates the delay in swapping in and out the code corresponding to the task to and from the primary memory

- The presence of *Human in the loop* (*HITL*) for tasks introduces un- expected delays in the task execution. Most of the Hard Real Time Systems are automatic and does not contain a 'human in the loop'

- **Soft Real-time System:**

  - Real Time Operating Systems that does not guarantee meeting deadlines, but, offer the best effort to meet the deadline

  - Missing deadlines for tasks are acceptable if the frequency of deadline missing is within the compliance limit of the Quality of Service(QoS)

  - A Soft Real Time system emphasizes on the principle '*A late answer is an acceptable answer, but it could have done bit faster'*

  - Soft Real Time systems most often have a '*human in the loop (HITL)*'
  - Automatic Teller Machine (ATM) is a typical example of Soft Real Time System. If the ATM takes a few seconds more than the ideal operation time, nothing fatal happens.

  - An audio video play back system is another example of Soft Real Time system. No potential damage arises if a sample comes late by fraction of a second, for play back.

Tasks, Processes & Threads:

- In the Operating System context, a task is defined as the program in execution and the related information maintained by the Operating system for the program

- Task is also known as '*Job*' in the operating system context

- A program or part of it in execution is also called a '*Process*'

- The terms '*Task*', '*job*' and '*Process*' refer to the same entity in the Operating System context and most often they are used interchangeably

- A process requires various system resources like CPU for executing the process, memory for storing the code corresponding to the process and associated variables, I/O devices for information exchange etc

The structure of a Processes

- The concept of '*Process*' leads to concurrent execution (pseudo parallelism) of tasks and thereby the efficient utilization of the CPU and other system resources
- Concurrent execution is achieved through the sharing of CPU among the processes.
- A process mimics a processor in properties and holds a set of registers, process status, a Program Counter (PC) to point to the next executable instruction of the process, a stack for holding the local variables associated with the process and the code corresponding to the process
- A process, which inherits all the properties of the CPU, can be considered as a virtual processor, awaiting its turn to have its properties switched into the physical processor

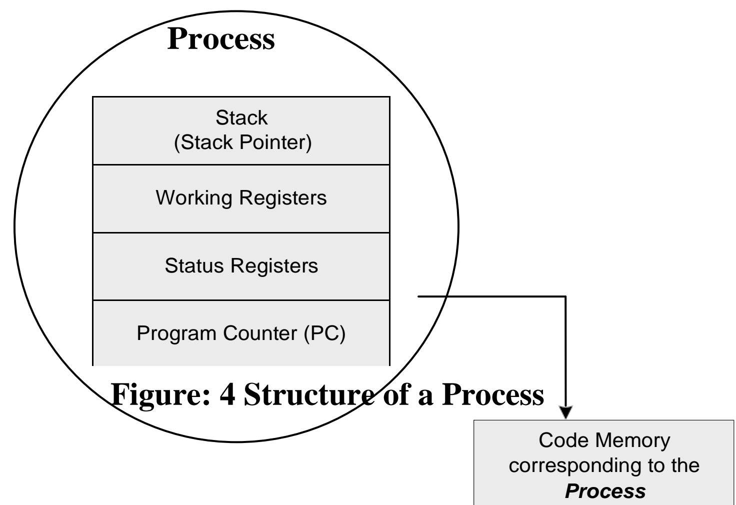


Figure: 4 Structure of a Process

- When the process gets its turn, its registers and Program counterregister becomes mapped to the physical registers of the CPU

Memory organization of Processes:

The memory occupied by the process is segregated into three regions namely; Stack memory, Data memory and Code memory.

The Stack memory holds all temporary data such as variables local to the process

Data memory holds all global data for the process

The Code memory contains the program code (instructions) corresponding to the process

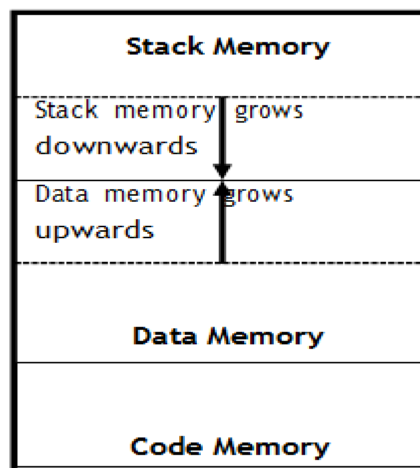


Fig: Memory organization of a Process

Process States & State Transition

- The creation of a process to its termination is not a single step operation
- The process traverses through a series of states during its transition from the newly created state to the terminated state
- The cycle through which a process changes its state from '*newly created*' to '*execution completed*' is known as '*Process Life Cycle*'. The various states through which a process traverses through during a Process Life Cycle indicates the current status of the process with respect to time and also provides information on what it is allowed to do next

Process States & State Transition:

- **Created State:** The state at which a process is being created is referred as 'Created State'. The Operating System recognizes a process in the '*Created State*' but no resources are allocated to the process

Ready State: The state, where a process is incepted into the memory and awaiting the processor time for execution, is known as '*Ready State*'. At this stage, the process is placed in the '*Ready list*' queue maintained by the OS

- **Running State:** The state where in the source code instructions corresponding to the process is being executed is called '*Running State*'. Running state is the state at which the process executionhappens

- . **Blocked State/Wait State:** Refers to a state where a running process is temporarily suspended from execution and does not have immediate access to resources. The blocked state might have invoked by various conditions like- the process enters a wait state for an event to occur (E.g. Waiting for user inputs such as keyboard input) or waiting for getting access to a shared resource like semaphore, mutex etc

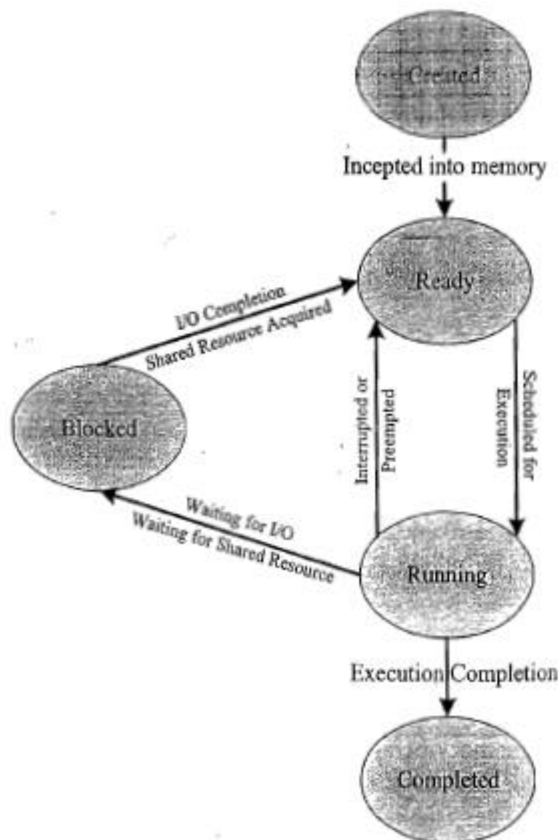


Figure 6.Process states and State transition

- **Completed State:** A state where the process completes its execution

- The transition of a process from one state to another is knownas '*State transition*'
- When a process changes its state from Ready to running orfrom running to blocked or terminated or from blocked to running, the CPU allocation for the process may also change

Threads

- A *thread* is the primitive that can execute code
- A *thread* is a single sequential flow of control within a process
- '*Thread*' is also known as lightweight process
- A process can have many threads of execution
- Different threads, which are part of a process, share the same address space; meaning they share the data memory, code memory and heap memory area
- Threads maintain their own thread status (CPU register values), Program Counter (PC) and stack

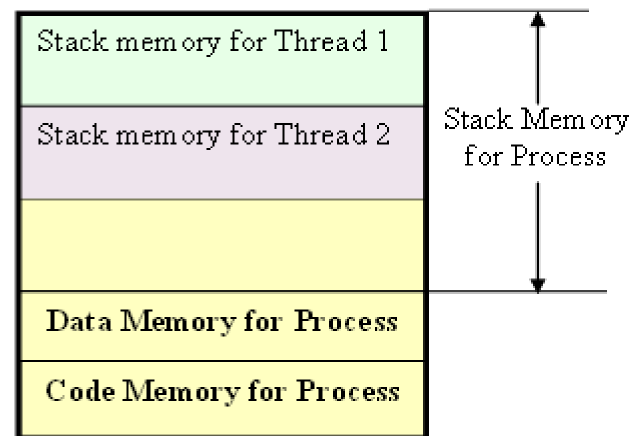


Figure 7 Memory organization of process and its associated Threads

Thread V/s Process

Thread	Process
Thread is a single unit of execution and is part of process.	Process is a program in execution and contains one or more threads.
A thread does not have its own data memory and heap memory. It shares the data memory and heap memory with other threads of the same process.	Process has its own code memory, data memory and stack memory.
A thread cannot live independently; it lives within the process.	A process contains at least one thread.

There can be multiple threads in a process. The first thread (main thread) calls the main function and occupies the start of the stack memory of the process.	Threads within a process share the code, data and heap memory. Each thread holds separate memory area for stack (shares the total stack memory of the process).
Threads are very inexpensive to create	Processes are very expensive to create. Involves many OS overhead.
Context switching is inexpensive and fast	Context switching is complex and involves lot of OS overhead and is comparatively slower.
If a thread expires, its stack is reclaimed by the process.	If a process dies, the resources allocated to it are reclaimed by the OS and all the associated threads of the process also dies.

Advantages of Threads:

1. **Better memory utilization:** Multiple threads of the same process share the address space for data memory. This also reduces the complexity of inter thread communication since variables can be shared across the threads.

2. **Efficient CPU utilization:** The CPU is engaged all time.

3. **Speeds up the execution of the process:** The process is split into different threads, when one thread enters a wait state, the CPU can be utilized by other threads of the process that do not require the event, which the other thread is waiting, for processing.

Multiprocessing & Multitasking

- The ability to execute multiple processes simultaneously is referred as

  multiprocessing

- Systems which are capable of performing multiprocessing are known as

  multiprocessor systems

- *Multiprocessor* systems possess multiple CPUs and can execute multiple processes simultaneously

- The ability of the Operating System to have multiple programs in memory, which are ready for execution, is referred as *multiprogramming*

- *Multitasking* refers to the ability of an operating system to hold multiple processes in memory and switch the processor (CPU) from executing one process to another process

- *Multitasking* involves '*Context switching*', '*Context saving*' and '*Context retrieval*'

- *Context switching* refers to the switching of execution context from task to other

- When a task/process switching happens, the current context of execution should be saved to (*Context saving*) retrieve it at a later point of time when the CPU executes the process, which is interrupted currently due to execution switching

- During context switching, the context of the task to be executed is retrieved from the saved context list. This is known as *Context retrieval.*

Multitasking – Context Switching:

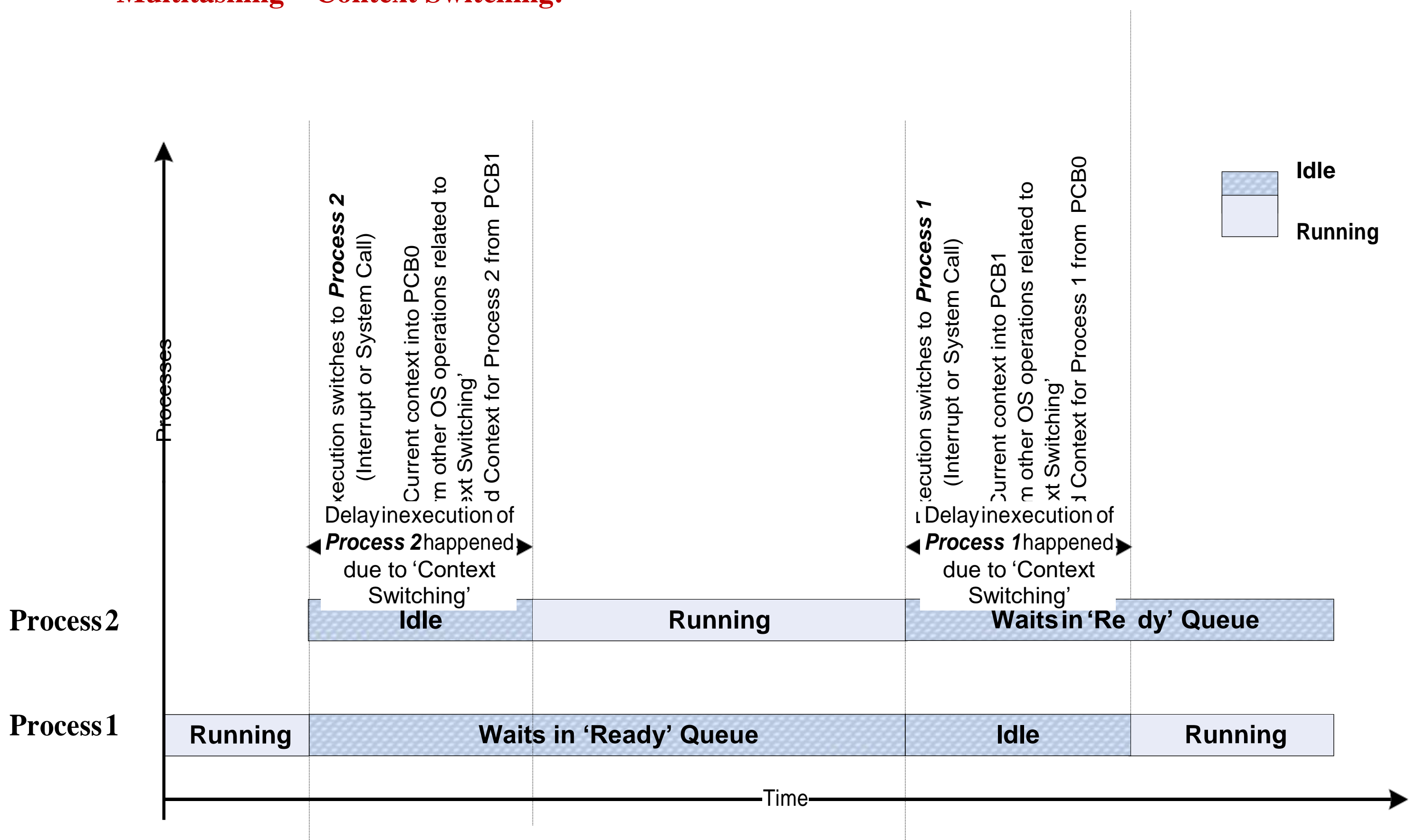


Figure 9 Context Switching

- **Multiprogramming:** The ability of the Operating System to have multiple Programs in memory, which are ready for execution, is referred as multiprogramming.

Types of Multitasking:

Depending on how the task/process execution switching act is implemented, multitasking can is classified into

- **Co-operative Multitasking:** Co-operative multitasking is the most primitive form of multitasking in which a task/process gets a chance to execute only when the currently executing task/process voluntarily relinquishes the CPU. In this method, any task/process can avail the CPU as much time as it wants. Since this type of implementation involves the mercy of the tasks each other for getting the CPU time for execution, it is known as co-operative multitasking. If the currently executing task is non-cooperative, the other tasks may have to wait for a long time to get the CPU

- **Preemptive Multitasking:** Preemptive multitasking ensures that every task/process gets a chance to execute. When and how much time a process gets is dependent on the implementation of the preemptive scheduling. As the name indicates, in preemptive multitasking, the currently running task/process is preempted to give a chance to other tasks/process to execute. The preemption of task may be based on time slots or task/process priority

- **Non-preemptive Multitasking:** The process/task, which is currently given the CPU time, is allowed to execute until it terminates (enters the 'Completed' state) or enters the 'Blocked/Wait' state, waiting for an I/O. The co- operative and non-preemptive multitasking differs in their behavior when they are in the 'Blocked/Wait' state. In co-operative multitasking, the currently executing process/task need not relinquish the CPU when it enters the 'Blocked/Wait' sate, waiting for an I/O, or a shared resource access or an event to occur whereas in non-preemptive multitasking the currently executing task relinquishes the CPU when it waits for an I/O.

Task Scheduling:

- In a multitasking system, there should be some mechanism in place to share the CPU among the different tasks and to decide which process/task is to be executed at a given point of time

- Determining which task/process is to be executed at a given point of time is known as task/process scheduling

- Task scheduling forms the basis of multitasking

- Scheduling policies forms the guidelines for determining which task is to be executed when

- The scheduling policies are implemented in an algorithm and it is run by the kernel as a service

- The kernel service/application, which implements the scheduling algorithm, is known as '*Scheduler*'

- The task scheduling policy can be *pre-emptive*, *non-preemptive* or *co- operative*

- Depending on the scheduling policy the process scheduling decision may take place when a process switches its state to

  - '*Ready*' state from '*Running*' state
  - '*Blocked/Wait*' state from '*Running*' state
  - '*Ready*' state from '*Blocked/Wait*' state
  - '*Completed*' state

Task Scheduling - Scheduler Selection:

The selection of a scheduling criteria/algorithm should consider the following factors:

- **CPU Utilization:** The scheduling algorithm should always make the CPU utilization high. CPU utilization is a direct measure of how much percentage of the CPU is being utilized.
- **Throughput:** This gives an indication of the number of processes executed per unit of time. The throughput for a good scheduler should always be higher.
- **Turnaround Time:** It is the amount of time taken by a process for completing its execution. It includes the time spent by the process for waiting for the main memory, time spent in the ready queue, time spent on completing the I/O operations, and the time spent in execution. The turnaround time should be a minimum for a good scheduling algorithm.
- **Waiting Time:** It is the amount of time spent by a process in the '*Ready*' queue waiting to get the CPU time for execution. The waiting time should be minimal for a good scheduling algorithm.
- **Response Time:** It is the time elapsed between the submission of a process and the first response. For a good scheduling algorithm, the response time should be as least as possible.

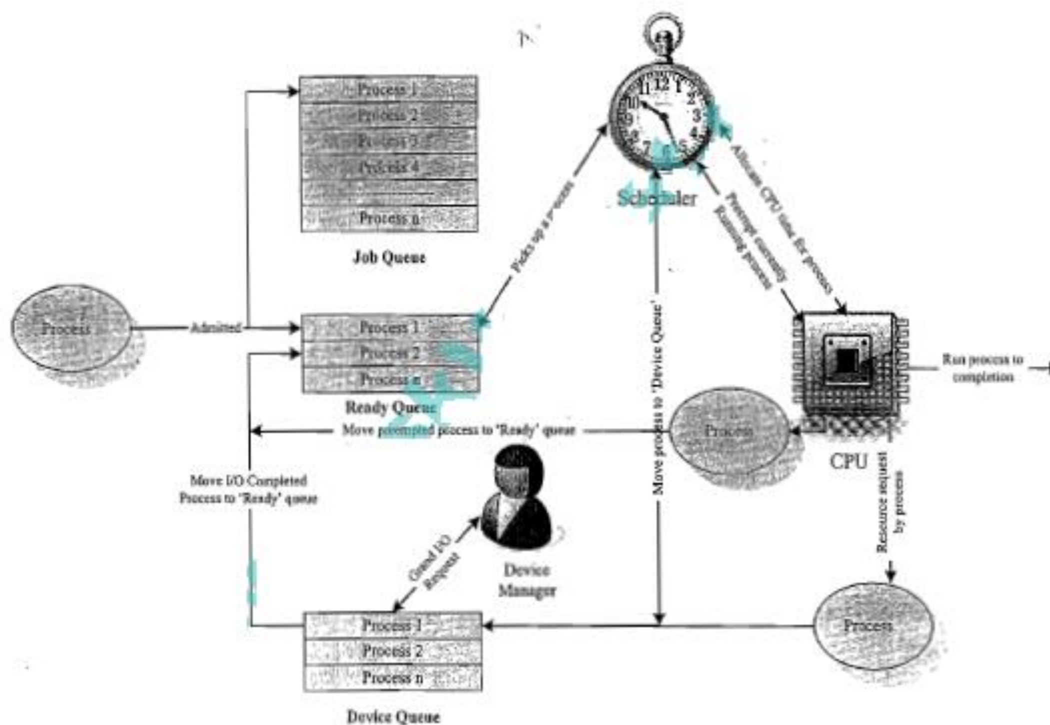
To summarize, a good scheduling algorithm has high CPU utilization, minimum Turn around Time (TAT), maximum throughput and least response time.

Task Scheduling - Queues

The various queues maintained by OS in association with CPU scheduling are:

- Job Queue: Job queue contains all the processes in the system
- Ready Queue: Contains all the processes, which are ready for execution and waiting for CPU to get their turn for execution. The Ready queue is empty when there is no process ready for running.
- Device Queue: Contains the set of processes, which are waiting for an I/O device

Task Scheduling – Task transition through various Queues



Non-preemptive scheduling – First Come First Served (FCFS)/First In First Out (FIFO) Scheduling:

- Allocates CPU time to the processes based on the order in which they enters the '*Ready*' queue
- The first entered process is serviced first
- It is same as any real world application where queue systems are used; E.g. Ticketing

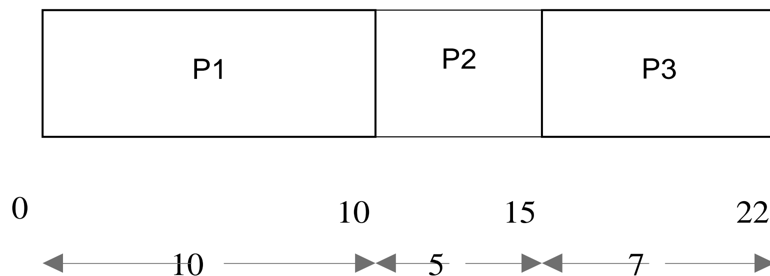
Drawbacks:

- Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task

- In general, FCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- The average waiting time is not minimal for FCFS scheduling algorithm

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes).

Solution: The sequence of execution of the processes by the CPU is represented as



Assuming the CPU is readily available at the time of arrival of P1, P1 starts executing without any waiting in the 'Ready' queue. Hence the waiting time for P1 is zero.

Waiting Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P2 = 10 ms (P2 starts executing after completing P1)

Waiting Time for P3 = 15 ms (P3 starts executing after completing P1 and P2)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P2+P3)) / 3

= (0+10+15)/3 = 25/3 = 8.33 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 15 ms (-

Do-) Turn Around Time (TAT) for P3 = 22 ms (-

Do-)

Average Turn around Time= (Turn around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P2+P3)) / 3

= (10+15+22)/3 = 47/3

= 15.66 milliseconds

Non-preemptive scheduling – Last Come First Served (LCFS)/Last In First Out (LIFO) Scheduling:

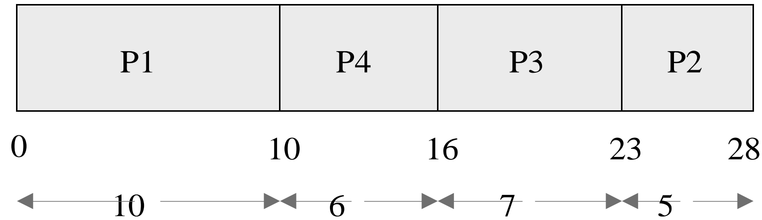
- Allocates CPU time to the processes based on the order in which theyare entered in the '*Ready*' queue
- The last entered process is serviced first
- LCFS scheduling is also known as Last In First Out (LIFO) where the process, which is put last into the '*Ready*' queue, is servicedfirst

Drawbacks:

- ➢ Favors monopoly of process. A process, which does not contain any I/O operation, continues its execution until it finishes its task
- ➢ In general, LCFS favors CPU bound processes and I/O bound processes may have to wait until the completion of CPU bound process, if the currently executing process is a CPU bound process. This leads to poor device utilization.
- ➢ The average waiting time is not minimal for LCFS scheduling algorithm

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together in the order P1, P2, P3 (Assume only P1 is present in the '*Ready*' queue when the scheduler picks up it and P2, P3 entered '*Ready*' queue after that). Now a new process P4 with estimated completion time 6ms enters the 'Ready' queue after 5ms of scheduling P1. Calculate the waiting time and Turn around Time (TAT) for each process and the Average waiting time and Turn around Time (Assuming there is no I/O waiting for the processes).Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution: Initially there is only P1 available in the Ready queue and the scheduling sequence will be P1, P3, P2. P4 enters the queue during the execution of P1 and becomes the last process entered the '*Ready*' queue. Now the order of execution changes to P1, P4, P3, and P2 as given below.



The waiting time for all the processes are given as Waiting

Time for P1 = 0 ms (P1 starts executing first)

Waiting Time for P4 = 5 ms (P4 starts executing after completing P1. But P4 arrived after 5ms of execution of P1. Hence its waiting time = Execution start time – Arrival Time = 10-5 = 5)

Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)

Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all processes) / No. of Processes

= (Waiting time for (P1+P4+P3+P2)) / 4

= (0 + 5 + 16 + 23)/4 = 44/4

= 11 milliseconds

Turn around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn around Time (TAT) for P4 = 11 ms (Time spent in Ready Queue + Execution Time = (Execution Start Time – Arrival Time) + Estimated Execution Time = (10-5) + 6 = 5 +6)

Turn around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time)

Average Turn Around Time = (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P4+P3+P2)) / 4

= (10+11+23+28)/4 = 72/4

= 18 milliseconds

Non-preemptive scheduling – Shortest Job First (SJF) Scheduling.

- Allocates CPU time to the processes based on the execution completion time for tasks
- The average waiting time for a given set of processes is minimal in SJF scheduling
- Optimal compared to other non-preemptive scheduling like FCFS

Drawbacks:

- A process whose estimated execution completion time is high may not get a chance to execute if more and more processes with least estimated execution time enters the '*Ready*' queue before the process with longest estimated execution time starts its execution
- May lead to the 'Starvation' of processes with high estimated completion time
- Difficult to know in advance the next shortest process in the '*Ready*' queue for scheduling since new processes with different estimated execution time keep entering the '*Ready*' queue at any point of time.

Non-preemptive scheduling – Priority based Scheduling

- A priority, which is unique or same is associated with each task
- The priority of a task is expressed in different ways, like a priority number, the time required to complete the execution etc.
- In number based priority assignment the priority is a number ranging from 0 to the maximum priority supported by the OS. The maximum level of priority is OS dependent.
- Windows CE supports 256 levels of priority (0 to 255 priority numbers, with 0 being the highest priority)
- The priority is assigned to the task on creating it. It can also be changed dynamically (If the Operating System supports this feature)
- The non-preemptive priority based scheduler sorts the 'Ready' queue based on the priority and picks the process with the highest level of priority for execution.

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 0, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in priority based scheduling algorithm.

Solution: The scheduler sorts the '*Ready*' queue based on the priority and schedules the process with the highest priority (P1 with priority number 0) first and the next high priority process (P3 with priority number 2) as second and so on. The order in which the processes are scheduled for execution is represented as

The waiting time for all the processes are given as Waiting
Time for P1 = 0 ms (P1 starts executing first)
Waiting Time for P3 = 10 ms (P3 starts executing after completing P1)
Waiting Time for P2 = 17 ms (P2 starts executing after completing P1 and P3)
Average waiting time = (Waiting time for all processes) / No. of Processes
= (Waiting time for (P1+P3+P2)) / 3

= (0+10+17)/3 = 27/3

= 9 milliseconds

Turn Around Time (TAT) for P1 = 10 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P3 = 17 ms (-
Do-) Turn Around Time (TAT) for P2 = 22 ms (-
Do-)
Average Turn Around Time= (Turn Around Time for all processes) / No. of Processes

= (Turn Around Time for (P1+P3+P2)) / 3

= (10+17+22)/3 = 49/3

= 16.33 milliseconds

Drawbacks:

- Similar to SJF scheduling algorithm, non-preemptive priority based algorithm also possess the drawback of '*Starvation*' where a process whose priority is low may not get a chance to execute if more and more processes with higher priorities enter the '*Ready*' queue before the process with lower priority starts its execution.
- '*Starvation*' can be effectively tackled in priority based non-preemptive scheduling by dynamically raising the priority of the low priority task/process which is under starvation (waiting in the ready queue for a longer time for getting the CPU time)
- The technique of gradually raising the priority of processes which are waiting in the 'Ready' queue as time progresses, for preventing '*Starvation*', is known as '*Aging*'.

Preemptive scheduling:

- Employed in systems, which implements preemptive multitasking model
- Every task in the '*Ready*' queue gets a chance to execute. When and how often each process gets a chance to execute (gets the CPU time) is dependent on the type of preemptive scheduling algorithm used for scheduling the processes
- The scheduler can preempt (stop temporarily) the currently executing task/process and select another task from the '*Ready*' queue forexecution
- When to pre-empt a task and which task is to be picked up from the '*Ready*' queue for execution after preempting the current task is purely dependent on the scheduling algorithm
- A task which is preempted by the scheduler is moved to the '*Ready*' queue. The act of moving a '*Running*' process/task into the '*Ready*' queue by the scheduler, without the processes requesting for it is known as'*Preemption*'
- Time-based preemption and priority-based preemption are the two important approaches adopted in preemptive scheduling

Preemptive scheduling – Preemptive SJF Scheduling/ Shortest Remaining Time (SRT):

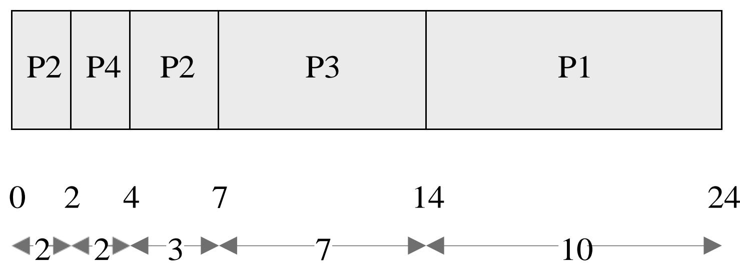
- The *non preemptive SJF* scheduling algorithm sorts the 'Ready' queue only after the current process completes execution or enters wait state, whereas the *preemptive SJF* scheduling algorithm sorts the 'Ready' queue when a new process enters the 'Ready' queue and checks whether the execution time of the new process is shorter than the remaining of the total estimated execution time of the currently executing process
- If the execution time of the new process is less, the currently executing process is

preempted and the new process is scheduled for execution

- Always compares the execution completion time (ie the remaining execution time for the new process) of a new process entered the 'Ready' queue with the remaining time for completion of the currently executing process and schedules the process with shortest remaining time for execution.

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds respectively enters the ready queue together. A new process P4 with estimated completion time 2ms enters the 'Ready' queue after 2ms. Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution: At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the SRT scheduler picks up the process with the Shortest remaining time for execution completion (In this example P2 with remaining time 5ms) for scheduling. Now process P4 with estimated execution completion time 2ms enters the '*Ready*' queue after 2ms of start of execution of P2. The processes are re-scheduled for execution in the following order



The waiting time for all the processes are given as

Waiting Time for P2 = 0 ms + (4 -2) ms = 2ms (P2 starts executing first and is interrupted by P4 and has to wait till the completion of P4 to get the next CPU slot)

Waiting Time for P4 = 0 ms (P4 starts executing by preempting P2 since the execution time for completion of P4 (2ms) is less than that of the Remaining time for execution completion of P2 (Here it is 3ms))

Waiting Time for P3 = 7 ms (P3 starts executing after completing P4 and P2)

Waiting Time for P1 = 14 ms (P1 starts executing after completing P4, P2 and P3)

Average waiting time = (Waiting time for all the processes) / No. of Processes

= (Waiting time for (P4+P2+P3+P1)) / 4

= (0 + 2 + 7 + 14)/4 = 23/4

= 5.75 milliseconds

Turn around Time (TAT) for P2 = 7 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 2 ms (Time spent in Ready Queue + Execution Time

= (Execution Start Time – Arrival Time) + Estimated Execution Time = (2-2) + 2)

Turn around Time (TAT) for P3 = 14 ms (Time spent in Ready Queue +Execution Time)

Turn around Time (TAT) for P1 = 24 ms (Time spent in Ready Queue +Execution Time)

Average Turn around Time = (Turn around Time for all the processes) / No. of Processes

= (Turn Around Time for (P2+P4+P3+P1)) / 4

= (7+2+14+24)/4 = 47/4

= 11.75 milliseconds

Preemptive scheduling – Round Robin (RR) Scheduling:

The term Round Robin is very popular among the sports and games activities. You might have heard about 'Round Robin' league or 'Knock out' league associated with any football or cricket tournament. In the 'Round Robin' league each team in a group gets an equal chance to play against the rest of the teams in the same group whereas in the 'Knock out' league the losing team in a match moves out of the tournament .

In Round Robin scheduling, each process in the 'Ready' queue is executed for a pre-defined time slot.

The execution starts with picking up the first process in the 'Ready' queue. It is executed for a pre-defined time and when the pre-defined time elapses or the process completes (before the pre-defined time slice), the next process in the 'Ready' queue is selected for execution.

This is repeated for all the processes in the 'Ready' queue. Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.

The sequence is repeated. This reveals that the Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the `Ready' queue.

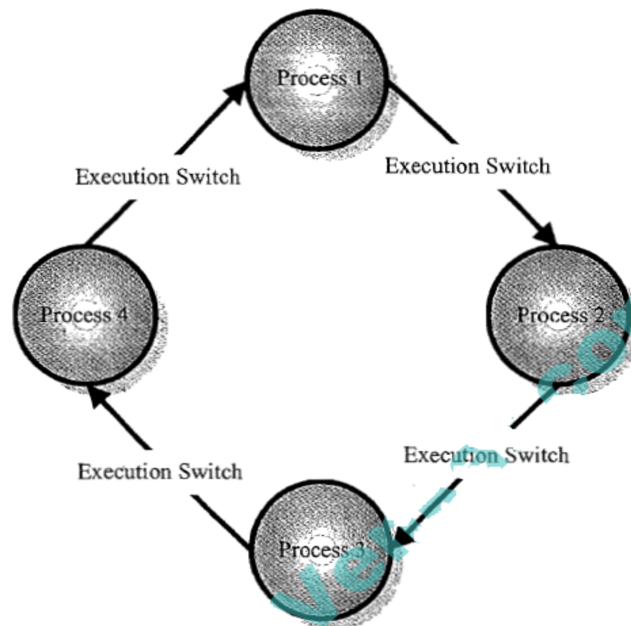


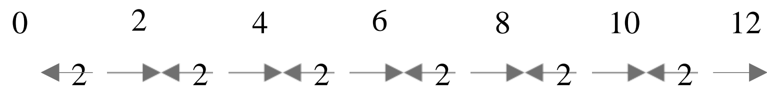
Figure: Round Robin Scheduling

- This is repeated for all the processes in the 'Ready' queue
- Once each process in the 'Ready' queue is executed for the pre-defined time period, the scheduler comes back and picks the first process in the 'Ready' queue again for execution.
- Round Robin scheduling is similar to the FCFS scheduling and the only difference is that a time slice based preemption is added to switch the execution between the processes in the 'Ready' queue

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 6, 4, 2 milliseconds respectively, enters the ready queue together in the order P1, P2, P3. Calculate the waiting time and Turn Around Time (TAT) for each process and the Average waiting time and Turn Around Time (Assuming there is no I/O waiting for the processes) in RR algorithm with Time slice= 2ms.

Solution: The scheduler sorts the '*Ready*' queue based on the FCFS policy and picks up the first process P1 from the 'Ready' queue and executes it for the time slice 2ms. When the time slice is expired, P1 is preempted and P2 is scheduled for execution. The Time slice expires after 2ms of execution of P2. Now P2 is preempted and P3 is picked up for execution. P3 completes its execution within the time slice and the scheduler picks P1 again for execution for the next time slice. This procedure is repeated till all the processes are serviced. The order in which the processes are scheduled for execution is represented as

P1	P2	P3	P1	P2	P1



The waiting time for all the processes are given as

Waiting Time for P1 = 0 + (6-2) + (10-8) = 0+4+2= 6ms (P1 starts executing first and waits for two time slices to get execution back and again 1 time slice for getting CPU time)

Waiting Time for P2 = (2-0) + (8-4) = 2+4 = 6ms (P2 starts executing after P1 executes for 1 time slice and waits for two time slices to get the CPU time)

Waiting Time for P3 = (4 -0) = 4ms (P3 starts executing after completing the first time slices for P1 and P2 and completes its execution in a single time slice.)

Average waiting time = (Waiting time for all the processes) / No. of Processes

= (Waiting time for (P1+P2+P3)) / 3

= (6+6+4)/3 = 16/3

= 5.33 milliseconds

Turn around Time (TAT) for P1 = 12 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P2 = 10 ms (-Do-)

Turn Around Time (TAT) for P3 = 6 ms (-Do-)

Average Turn around Time = (Turn around Time for all the processes) / No. of Processes

= (Turn Around Time for (P1+P2+P3)) / 3

= (12+10+6)/3 = 28/3

= 9.33 milliseconds.

Preemptive scheduling – Priority based Scheduling

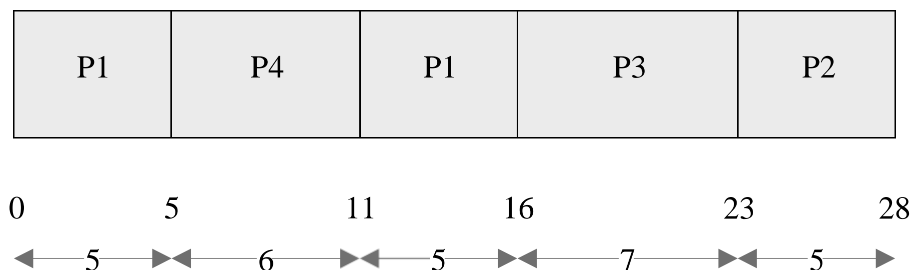
- Same as that of the ***non-preemptive priority*** based scheduling except for the switching of execution between tasks

- In ***preemptive priority*** based scheduling, any high priority process entering the 'Ready' queue is immediately scheduled for execution whereas in the ***non-preemptive*** scheduling any high priority process entering the 'Ready' queue is scheduled only after the currently executing process completes its execution or only when it voluntarily releases the CPU

- The priority of a task/process in preemptive priority based scheduling is indicated in the same way as that of the mechanisms adopted for non- preemptive multitasking.

EXAMPLE: Three processes with process IDs P1, P2, P3 with estimated completion time 10, 5, 7 milliseconds and priorities 1, 3, 2 (0- highest priority, 3 lowest priority) respectively enters the ready queue together. A new process P4 with estimated completion time 6ms and priority 0 enters the 'Ready' queue after 5ms of start of execution of P1. Assume all the processes contain only CPU operation and no I/O operations are involved.

Solution: At the beginning, there are only three processes (P1, P2 and P3) available in the '*Ready*' queue and the scheduler picks up the process with the highest priority (In this example P1 with priority 1) for scheduling. Now process P4 with estimated execution completion time 6ms and priority 0 enters the '*Ready*' queue after 5ms of start of execution of P1. The processes are re-scheduled for execution in the following order



The waiting time for all the processes are given as

Waiting Time for P1 = 0 + (11-5) = 0+6 =6 ms (P1 starts executing first and gets Preempted by P4 after 5ms and again gets the CPU time after completion of P4)

Waiting Time for P4 = 0 ms (P4 starts executing immediately on entering the 'Ready' queue, by preempting P1)

Waiting Time for P3 = 16 ms (P3 starts executing after completing P1 and P4)

Waiting Time for P2 = 23 ms (P2 starts executing after completing P1, P4 and P3)

Average waiting time = (Waiting time for all the processes) / No. of Processes

= (Waiting time for (P1+P4+P3+P2)) / 4

= (6 + 0 + 16 + 23)/4 = 45/4

= 11.25 milliseconds

Turn Around Time (TAT) for P1 = 16 ms (Time spent in Ready Queue + Execution Time)

Turn Around Time (TAT) for P4 = 6ms (Time spent in Ready Queue + Execution Time
= (Execution Start Time – Arrival Time) + Estimated Execution Time = (5-5) + 6 = 0 + 6)

Turn Around Time (TAT) for P3 = 23 ms (Time spent in Ready Queue + Execution Time) Turn Around Time (TAT) for P2 = 28 ms (Time spent in Ready Queue + Execution Time) Average Turn Around Time= (Turn Around Time for all the processes) / No. of Processes

= (Turn Around Time for (P2+P4+P3+P1)) / 4

= (16+6+23+28)/4 = 73/4

= 18.25 milliseconds

How to choose RTOS:

- The decision of an RTOS for an embedded design is very critical.
- A lot of factors need to be analyzed carefully before making a decisionon the selection of an RTOS.

These factors can be either

1. **Functional**
2. **Non-functional requirements.**

1. Functional Requirements:

1. Processor support:

- It is not necessary that all RTOS's support all kinds of processor architectures.
- It is essential to ensure the processor support by the RTOS

2. Memory Requirements:

- The RTOS requires ROM memory for holding the OS files and it is normally stored in a non-volatile memory like FLASH.
- OS also requires working memory RAM for loading the OS service.
- Since embedded systems are memory constrained, it is essential to evaluate the minimal RAM and ROM requirements for the OS under consideration.

3. Real-Time Capabilities:

- It is not mandatory that the OS for all embedded systems need to be Real-Time and all embedded OS's are 'Real-Time' in behavior.
- The Task/process scheduling policies plays an important role in the Real-Time behavior of an OS.

4. Kernel and Interrupt Latency:

- The kernel of the OS may disable interrupts while executing certain services and it may lead to interrupt latency.
- For an embedded system whose response requirements are high, this latency should be minimal.

5. Inter process Communication (IPC) and Task Synchronization:

- The implementation of IPC and Synchronization is OS kernel dependent.

6. Modularization Support:

- Most of the OS's provide a bunch of features.
- It is very useful if the OS supports modularization where in which the developer can choose the essential modules and re-compile the OS image for functioning.

7. Support for Networking and Communication:

- The OS kernel may provide stack implementation and driver support for a bunch of communication interfaces and networking.
- Ensure that the OS under consideration provides support for all the interfaces required by the embedded product.

8. Development Language Support:

- Certain OS's include the run time libraries required for running applications written in languages like JAVA and C++.
- The OS may include these components as built-in component, if not; check the availability of the same from a third party.

2. Non-Functional Requirements:

1. Custom Developed or Off the Shelf:

- It is possible to go for the complete development of an OS suiting the embedded system needs or use an off the shelf, readily available OS.
- It may be possible to build the required features by customizing an open source OS.
- The decision on which to select is purely dependent on the development cost, licensing fees for the OS, development time and availability of skilled resources.

2. Cost:

- The total cost for developing or buying the OS and maintaining it in terms of commercial product and custom build needs to be evaluated before taking a decision on the selection of OS.

3. Development and Debugging tools Availability:

- The availability of development and debugging tools is a critical decision making factor in the selection of an OS for embedded design.
- Certain OS's may be superior in performance, but the availability of tools for supporting the development may be limited.

4. Ease of Use:

- How easy it is to use a commercial RTOS is another important feature that needs to be considered in the RTOS selection.

5. After Sales:

- For a commercial embedded RTOS, after sales in the form of e-mail, on-call services etc. for bug fixes, critical patch updates and support for production issues etc. should be analyzed thoroughly.

3.2 TASK COMMUNICATION:

In a multitasking system, multiple tasks/processes run concurrently (in pseudo parallelism) and each process may or may not interact between. Based on the degree of interaction, the processes running on an OS are classified as,

1. Co-operating Processes: In the co-operating interaction model one process requires the inputs from other processes to complete its execution.

2. Competing Processes: The competing processes do not share anything among themselves but they share the system resources. The competing processes compete for the system resources such as file, display device, etc.

Co-operating processes exchanges information and communicate through the following methods.

Co-operation through Sharing: The co-operating process exchange data through some shared resources.

Co-operation through Communication: No data is shared between the processes. But they communicate for synchronization.

The mechanism through which processes/tasks communicate each other is known as "**Inter Process/Task Communication (IPC)**". Inter Process Communication is essential for process co-ordination. The various types of Inter Process Communication (IPC) mechanisms adopted by process are kernel (Operating System) dependent. Some of the important IPC mechanisms adopted by various kernels are explained below.

3.2.1 Shared Memory:

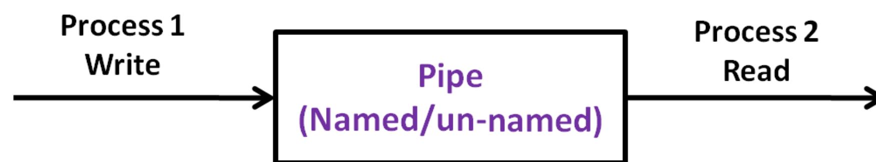
Processes share some area of the memory to communicate among them. Information to be communicated by the process is written to the shared memory area. Other processes which require this information can read the same from the shared memory area. It is same as the real world example where 'Notice Board' is used by corporate to publish the public information among the employees (The only exception is; only corporate have the right to modify the-information published on the Notice board and employees are given 'Read' only access, meaning it is only a one way channel).

Figure: Concept of shared memory

The implementation of shared memory concept is kernel dependent. Different mechanisms are adopted by different kernels for implementing this. A few among them are:

3.2.1.1 Pipes:

'Pipe' is a section of the shared memory used by processes for communicating. Pipes follow the client-server architecture. A process which creates a pipe is known as a pipe server and a process which connects to a pipe is known as pipe client. A pipe can be considered as a conduit for information flow and has two conceptual ends. It can be unidirectional, allowing information flow in one direction or bidirectional allowing bi-directional information flow. A unidirectional pipe allows the process connecting at one end of the pipe to write to the pipe and the process connected at the other end of the pipe to read the data, whereas a bi-directional pipe allows both reading and writing at one end. The unidirectional pipe can be visualized as



The implementation of 'pipes' is also OS dependent. Microsoft® Windows Desktop Operating Systems support two types of 'Pipes' for Inter Process Communication. They are:

Anonymous Pipes: The anonymous pipes-are unnamed, unidirectional pipes used for data transfer between two processes.

Named Pipes: Named pipe is a named, unidirectional or bi-directional pipe for data exchange between processes. Like anonymous pipes, the process which creates the named pipe is known as pipe server. A process which connects to the named pipe is known as pipe client.

With named pipes, any process can act as both client and server allowing point-to-point communication. Named pipes can be used for communicating between processes running on the same machine or between processes running on different machines connected to a network.

Please refer to the Online Learning Centre for details on the Pipe implementation under Windows Operating Systems.

Under VxWorks kernel, pipe is a special implementation of message queues. We will discuss the same in a latter chapter.

3.2.1.2 Memory Mapped Objects:

Memory mapped object is a shared memory technique adopted by certain Real-Time Operating Systems for allocating a shared block of memory which can be accessed by multiple process simultaneously (of course certain synchronization techniques should be applied to prevent inconsistent results). In this approach a mapping object is created and physical storage for it is reserved and committed. A process can map the entire committed physical area or a block of it to its virtual address space. All read and write operation to this virtual address space by a process is directed to its committed physical area. Any process

which wants to share data with other processes can map the physical memory area of the mapped object to its virtual memory space and use it for sharing the data.

3.2.2 Message Passing:

Message passing-is an (a) synchronous information exchange mechanism used for Inter Process/Thread Communication. The major difference between shared memory and message passing technique is that, through shared memory lots of data can be shared whereas only limited amount of info/data is passed through message passing. Also message passing is relatively fast and free from the synchronization overheads compared to shared memory. Based on the message passing operation between the processes, message passing is classified into:

- Message Queue.
- Mailbox.
- Signaling.

3.2.2.1 Message Queue: Usually the process which wants to talk to another process posts the message to a First-In-First-Out (FIFO) queue called 'Message queue', which stores the messages temporarily in a system defined memory object, to pass it to the desired process (Fig. 10.20). Messages are sent and received through send (Name of the process to which the message is to be sent,-message) and receive (Name of the process from which the message is to be received, message) methods. The messages are exchanged through a message queue. The implementation of the message queue, send and receive methods are OS kernel dependent. The Windows XP OS kernel maintains a single system message queue and one process/thread (Process and threads are used interchangeably here, since thread is the basic unit of process in windows) specific message queue. A thread which wants to communicate with another thread posts the message to the system message queue. The kernel picks up the message from the system message queue one at a time and examines the message for finding the destination thread and then posts the message to the message queue of the corresponding thread. For posting a message to a thread's message queue, the kernel fills a message structure MSG and copies it to the message queue of the thread. The message structure MSG contains the handle of the process/thread for which the message is intended, the message parameters, the time at which the message is posted, etc. A thread can simply post a message to another thread and can continue its operation or it may wait for a response from the thread to which the message is posted. The messaging mechanism is classified into synchronous and asynchronous based on the behaviour of the message posting thread. In asynchronous messaging, the message posting thread just posts the message to the queue and it will not wait for an acceptance (return) from the thread to which the message is posted, whereas in synchronous messaging, the thread which posts a message enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the send message becomes blocked and the scheduler will not pick it up for scheduling. The PostMessage (HWND hWnd, UINT Msg, WPARAM wParam, LPARAM /Param) or PostThreadMessage (DWORD idThread, UNT Msg, WPARAM wParam, LPARAM IParam)

API is used by a thread in Windows for posting a message to its own message queue or to the message queue of another thread.

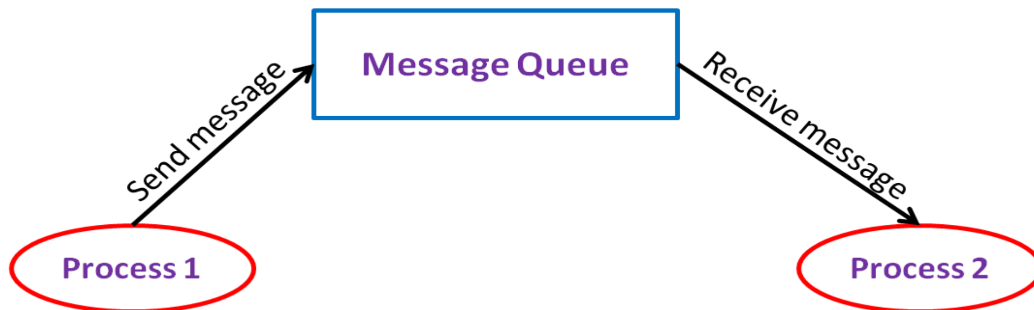


Figure: Concept of message queue based indirect messaging for IPC.

The PostMessage API does not always guarantee the posting of messages to message queue. The PostMessage API will not post a message to the message queue when the message queue is full. Hence it is recommended to check the return value of PostMessage API to confirm the posting of message. The SendMessage (HWND hWnd, U1NT Msg, WPARAM wParam, LPARAM 1Param) API call sends a message to the thread specified by the handle hWnd and waits for the callee thread to process the message. The thread which calls the SendMessage API enters waiting state and waits for the message result from the thread to which the message is posted. The thread which invoked the SendMessage API call becomes blocked and the scheduler will not pick it up for scheduling.

The Windows CE operating system supports a special Point-to-Point Message queue implementation. The OS maintains a First In First Out (FIFO) buffer for storing the messages and each process can access this buffer for reading and writing messages. The OS also maintains a special queue, with single message storing capacity, for storing high priority messages Werlmessages).

3.2.2.2 Mailbox:

Mailbox is an alternate form of 'Message queues' and it is used in certain Real-Time Operating Systems for IPC. Mailbox technique for IPC in RTOS is usually used for one way messaging. The task/thread which wants to send a message to other tasks/threads creates a mailbox for posting the messages. The threads which are interested in receiving the messages posted to the mailbox by the mailbox creator thread can subscribe to the mailbox.

The thread which creates the mailbox is known. as 'mailbox server' and the threads which subscribe to the mailbox are known as 'mailbox clients'. The mailbox server posts messages to the mailbox and notifies it to the clients which are subscribed to the mailbox. The clients read the message from the mailbox on receiving the notification.

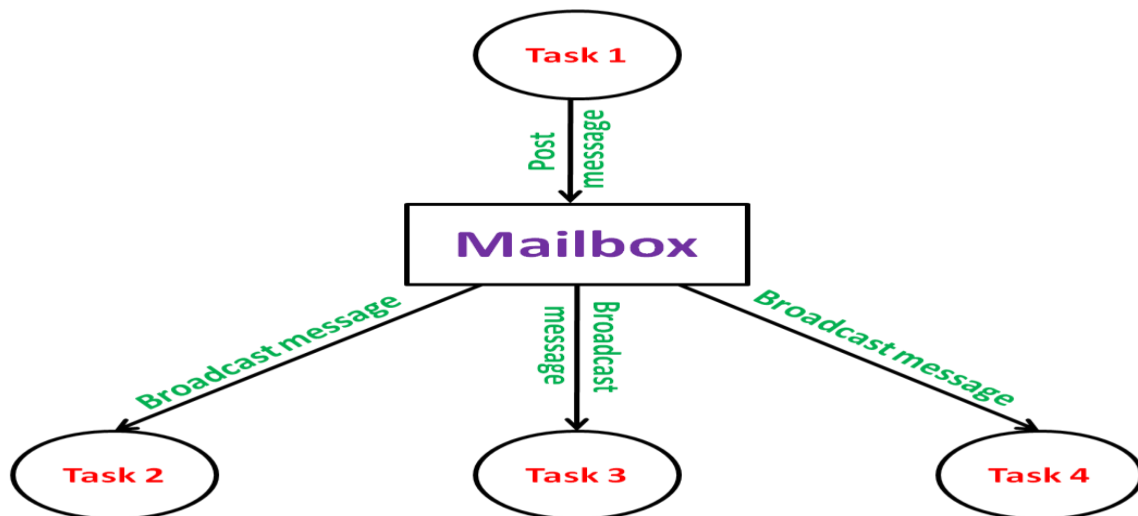


Figure: Concept of mailbox based indirect messaging for IPC.

The mailbox creation, subscription, message reading and writing are achieved through OS kernel provided API calls. Mailbox and message queues are same in functionality. The only difference is in the number of messages supported by them. Both of them are used for passing data in the form of message(s) from a task to another task(s).

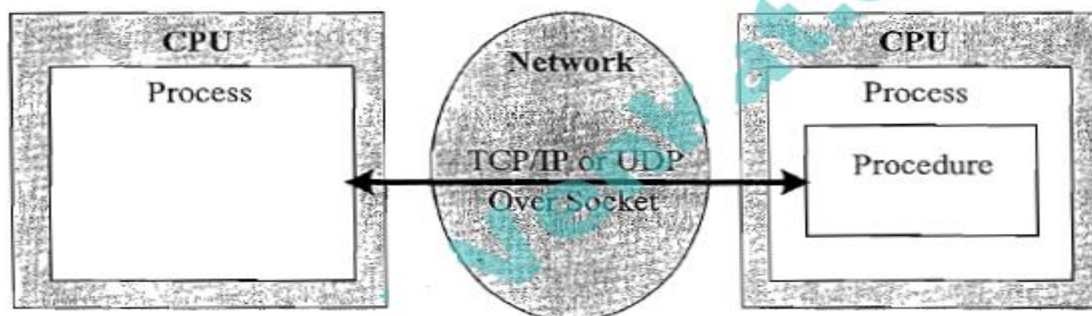
Mailbox is used for exchanging a single, message between two tasks or between an Interrupt Service Routine (ISR) and a task. Mailbox associates a pointer pointing to the mailbox and a wait list to hold the tasks waiting for a message to appear in the mailbox. The implementation of mailbox is OS kernel dependent. The MicroC/OS-II implements mailbox as a mechanism for inter-task communication.

3.2.2.3 Signaling:

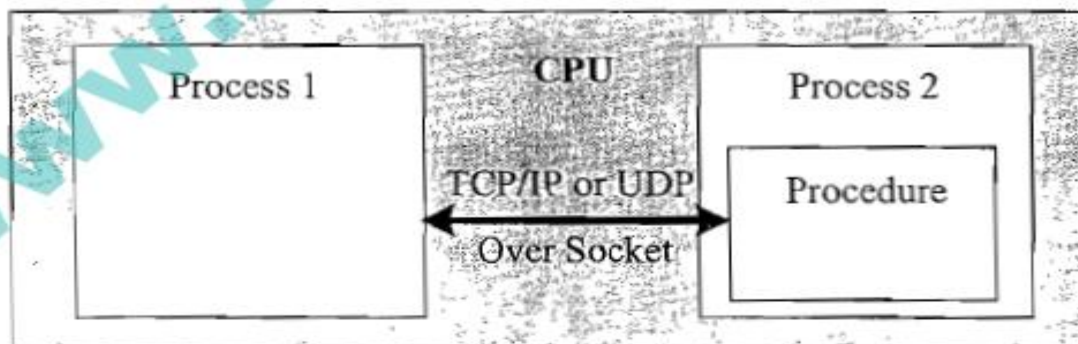
Signaling is a primitive way of communication between process-es/threads. Signals are used for asynchronous notifications where one process/thread fires a signal, indicating the occurrence of a scenario which the other process(es)/thread(s) is waiting. Signals are not queued and they do not carry any data. The communication mechanisms used in RTX51 Tiny OS is an example for Signaling. The amend signal kernel call under RTX 51 sends a signal from one task to a specified task. Similarly the os_wait kernel call waits for a specified signal. The VxWorks RTOS kernel also implements 'signals' for inter process communication. Whenever a signal occurs it is handled in a signal handler associated with the signal.

3.2.3 Remote Procedure Call (RPC) and Sockets:

Remote Procedure Call or RPC is the Inter Process Communication (IPC) mechanism used by a process to call a procedure of another process running on the same CPU or on a different CPU which is interconnected in a network. In the object oriented language terminology RPC is also known as Remote Invocation or Remote Method Invocation (RMI). RPC is mainly used for distributed applications like client-server applications. With RPC it is possible to communicate over a heterogeneous network (i.e. Network where Client and server applications are running on different Operating systems). The CPU/process containing the procedure which needs to be invoked remotely is known as server. The CPU/process which initiates an RPC request is known as client.



Processes running on different CPUs which are networked



Processes running on same CPU

Figure: Concept of Remote Procedure Call (RPC) for IPC

It is possible to implement RPC communication with different invocation interfaces. In order to make the RPC communication compatible across all platforms, it should stick on to certain standard formats. Interface Definition Language (IDL) defines the interfaces for RPC.

Microsoft Interface Definition Language (MIDL) is the IDL implementation from Microsoft for all Microsoft platforms. The RPC communication can be either Synchronous (Blocking) or Asynchronous (Non-blocking). In the Synchronous communication, the process which calls the remote procedure is blocked until it receives a response back from the other process. In asynchronous RPC calls, the calling process continues its execution while the remote process performs the execution of the procedure. The result from the remote procedure is returned back to the caller through mechanisms like callback functions.

On security front, RPC employs authentication mechanisms to protect the systems against vulnerabilities. The client applications (processes)-should authenticate themselves with the server for getting access. Authentication mechanisms like IDs, public-key cryptography, etc. are used by the client for authentication. Without authentication, any client can access the remote procedure. This may lead to potential security risks.

Sockets are used for RPC communication. The socket is a logical endpoint in a two-way communication link between two applications running on a network. A port number is associated with a socket so that the network layer of the communication channel can deliver the data to the designated application. Sockets are of different types, namely, Internet sockets (INET), UNIX sockets, etc. The INET socket works on internet communication protocol TCP/IP, UDP (User Datagram Protocol), etc. are the communication protocols used by INET sockets. INET sockets are classified into:

1. Stream sockets
2. Datagram sockets

Stream sockets are connection-oriented and they use TCP to establish liable connection. On the other hand, Datagram sockets rely on UDP for establishing a connection. The UDP connection is unreliable when compared to TCP. The client-server communication model uses a socket at the client-side and a socket at the server-side. A port number is assigned to both of these sockets. The client and server should be aware of the port number associated with the socket. In order to start the communication, the client needs to send a connection request to the server at the specified port number.
The client should be aware of the name of the server along with its port number. The server always listens to the specified port number on the network. Upon receiving a connection request from the client, based on the success of authentication, the server grants the connection request and a communication channel is established between the client and server. The client uses the hostname and port number of the server for sending requests and the server uses the client's name and port number for sending responses.

3.3 TASK SYNCHRONISATION:

In a multitasking environment, multiple processes run concurrently (in pseudo parallelism) and share the system resources. Apart from this, each process has its own boundary wall and they communicate with each other with different IPC mechanisms including shared memory and variables. Imagine a situation where two processes try to access display hardware connected to the system or two processes try to access a shared memory area where one process tries to write to a memory location when the other process is trying to read from this.

What could be the result in these scenarios? Obviously unexpected results. How these issues can be addressed? The solution is, make each process aware of the access of a shared resource either directly or indirectly. The act of making processes aware of the access of shared resources by each process to avoid conflicts is known as `Task/Process Synchronization'. Various synchronization issues may arise in a multitasking environment if processes are not synchronized properly.

The following sections describe the major task communication/ synchronization issues observed in multitasking and the commonly adopted synchronization techniques to overcome these issues.

3.3.1 Task Communication/Synchronization Issues:

3.3.1.1 Racing: Let us have a look at the following piece of code.

```
#include <windows.h>
#include <stdio.h>
//****************************************************************
//counter is an integer variable and Buffeger arrbyte array shared
//between two processes Process A and Prcesse..
char Buffer[10] = {1,2,3,4,5,6,7,8,9, } = {1,2
short int counter = 0;
//***************************************************************
// Process A
void Process_A (void) {
int i;
  for (i =0; i<5; i++)
{
    if (Buffer[i] > 0) {
       counter++;
}
 }
```

```
//*****************************************************************
// Process B
void Process_B(void) {
int j;
  for (j =5; j<10; j++)
{
    if (Buffer[j] > 0)
       counter++;
}
}
//*****************************************************************
//Main Thread.
int main() {
       DWORD id;
       CreateThread(NULL,0,
                      (LPTHREAD_START_ROUTINE)Process_A,
                      (LPVOID)0,0,&id);
       CreateThread(NULL,0,
                      (LPTHREAD_START_ROUTINE)Process_B,
                      (LPVOID)0,0,&id);
       Sleep(100000);
       return 0;
};
```

From a programmer perspective, the value of the counter will be 10 at the end of the execution of processes A & B. But 'it need not be always' in a real-world execution of this piece of code under a multitasking kernel. The results depend on the process scheduling policies adopted by the OS kernel. The program statement counter++; looks like a single statement from a high-level programming language (`C' language) perspective. The low-level implementation of this statement is dependent on the underlying processor instruction set and the (cross) compiler in use. The low-level implementation of the high-level program statement counter++; under Windows XP operating system running on an Intel Centrino Duo processor is given below.

mov eax, dword ptr [ebp-4] ; Load counter in Accumulator

add eax,1 ; Increment Accumulator by 1

mov dword ptr [ebp-4], eax ; Store counter with Accumulator

At the processor instruction level, the value of the variable counter is loaded to the Accumulator register (EAX register). The memory variable counter is represented using a pointer. The base pointer register (EBP register) is used for pointing to the memory variable counter. After loading the contents of the variable-counter to the Accumulator, the Accumulator content is incremented by one using the add instruction. Finally the content of Accumulator is loaded to the memory location which represents the variable counter. Both

the processes Process A and Process B contain the program statement counter++; Translating this into the machine instruction.

Process A	Process A
mov eax, dword ptr [ebp-4]	mov eax, dword ptr [ebp-4]
add eax,1	add eax,1
mov dword ptr [ebp-4], eax	mov dword ptr [ebp-4], eax

Imagine a situation where a process switching (context switching) happens from Process A to Process B when Process A is executing the counter++; statement. Process A accomplishes the counter++; statement through three different low-level instructions. Now imagine that the process switching happened at the point where Process A executed the low-level instruction, `mov eax,dword ptr [ebp-4]' and is about to execute the next instruction 'add eax,1'.

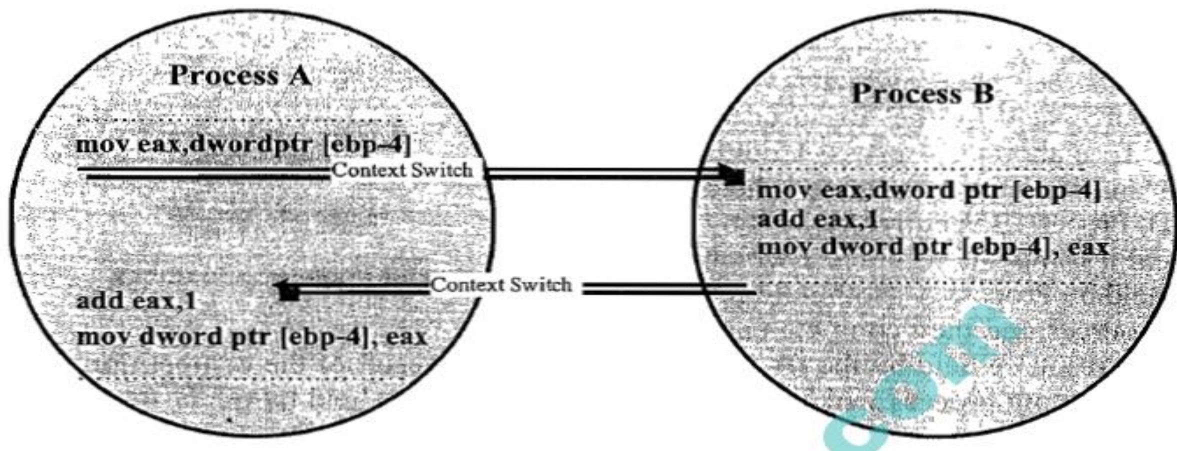


Figure: Race condition

Though the variable counter is incremented by Process B, Process A is unaware of it and it increments the variable with the old value. This leads to the loss of one increment for the Variable counter. This problem occurs due to non-atomic Operation on variables. This issue wouldn't have been occurred if the underlying actions corresponding to the program statement counter++; is finished in a single CPU execution cycle. The best way to avoid this situation is make the access and modification of shared variables mutually exclusive; meaning when one process accesses a shared variable, prevent the other processes from accessing it.

To summarize, Racing or Race condition is the situation in which multiple processes compete (race) each other to access and manipulate shared data concurrently. In a Race condition, the final value of the shared data depends on the process which acted on the data finally.

3.3.1.2 Deadlock:

A race condition produces incorrect results whereas a deadlock condition creates a situation where none of the processes are able to make any progress in their execution, resulting in a get of deadlocked processes. A situation very similar to our traffic jam issues in a junction.

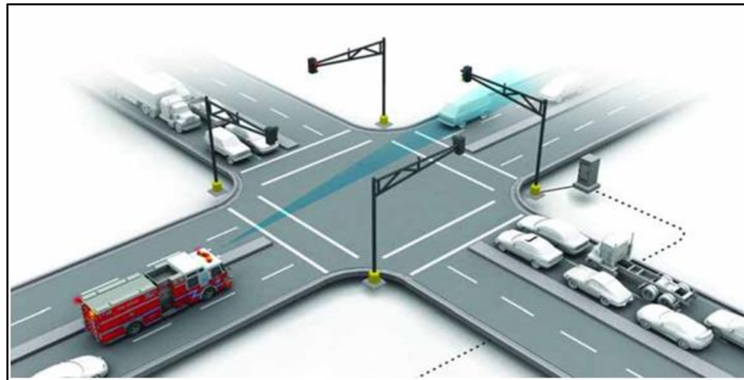


Figure: Deadlock Visualization

In its simplest form 'deadlock' is the condition in which a process is waiting for a resource held by another process which is waiting for a resource held by the first process.

To elaborate: Process A holds a resource x and it wants a resource y held by Process B. Process B is currently holding resource y and it wants the resource x which is currently held by Process A. Both hold the respective resources and they compete each other to get the resource held by the respective processes. The result of the competition is 'deadlock'. None of the competing processes will be able to access the resources held by other processes since they are locked by the respective processes.

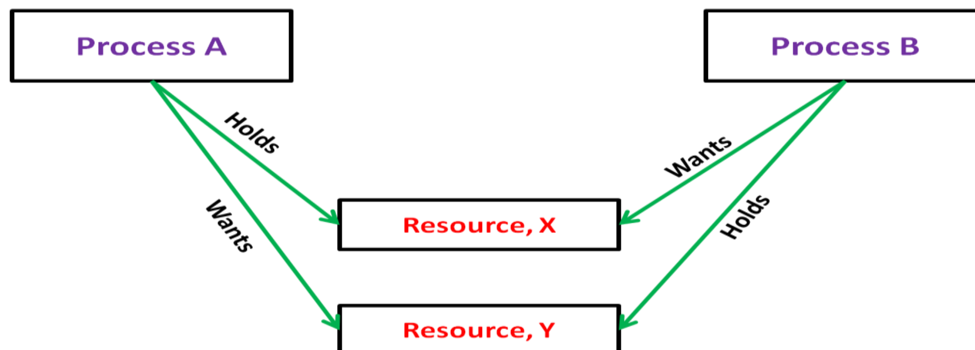


Figure: Scenarios leading to deadlock

The different conditions favoring a deadlock situation are listed below.

Mutual Exclusion: The criteria that only one process can hold resource at a time. Meaning processes should access shared resources with mutual exclusion. Typical example is the accessing of display hardware in an embedded device.

Hold and Walt: The condition in which a process holds a shared resource by acquiring the lock controlling the shared access and waiting for additional resources held by other processes.

No Resource Preemption: The criteria that operating system cannot take back a resource from a process which is currently holding it and the resource can only be released voluntarily by the process holding it.

Circular Wait: A process is waiting for a resource which is currently held by another process which in turn is waiting for a resource held by the first process. In general, there exists a set of waiting process P0, P1, Pn with P0 is waiting for a resource held by P1 and P1 is waiting for a resource held P0, Pn is waiting for a resource held by P0 and P0 is waiting for a resource held by Pn and so on... This forms a circular wait queue.

Deadlock Handling: A smart OS may foresee the deadlock condition and will act proactively to avoid such a situation. Now if a deadlock occurred, how the OS responds to it? The reaction to deadlock condition by OS is nonuniform. The OS may adopt any of the following techniques to detect and prevent deadlock conditions.

(i).Ignore Deadlocks: Always assume that the system design is deadlock free. This is acceptable for the reason the cost of removing a deadlock is large compared to the chance of happening a deadlock. UNIX is an example for an OS following this principle. A life critical system cannot pretend that it is deadlock free for any reason.

(ii). Detect and Recover: This approach suggests the detection of a deadlock situation and recovery from it. This is similar to the deadlock condition that may arise at a traffic junction.

When the vehicles from different directions compete to cross the junction, deadlock (traffic jam) condition is resulted. Once a deadlock (traffic jam) is happened at the junction, the only solution is to back up the vehicles from one direction and allow the vehicles from opposite direction to cross the junction. If the traffic is too high, lots of vehicles may have to be backed up to resolve the traffic jam. This technique is also known as `back up cars' technique.

Operating systems keep a resource graph in their memory. The resource graph is updated on each resource request and release.

Avoid Deadlocks: Deadlock is avoided by the careful resource allocation techniques by the Operating System. It is similar to the traffic light mechanism at junctions to avoid the traffic jams.

Prevent Deadlocks: Prevent the deadlock condition by negating one of the four conditions favoring the deadlock situation.

• Ensure that a process does not hold any other resources when it requests a resource. This can be achieved by implementing the following set of rules/guidelines in allocating resources to processes.

1. A process must request all its required resource and the resources should be allocated before the process begins its execution.
2. Grant resource allocation requests from processes only if the process does not hold a resource currently.

• Ensure that resource preemption (resource releasing) is possible at operating system level. This can be achieved by implementing the following set of rules/guidelines in resources allocation and releasing.

1. Release all the resources currently held by a process if a request made by the process for a new resource is not able to fulfil immediately.
2. Add the resources which are preempted (released) to a resource list describing the resources which the process requires to complete its execution.
3. Reschedule the process for execution only when the process gets its old resources and the new resource which is requested by the process.

Imposing these criterions may introduce negative impacts like low resource utilization and starvation of processes.

Livelock: The Livelock condition is similar to the deadlock condition except that a process in livelock condition changes its state with time. While in deadlock a process enters in wait state for a resource and continues in that state forever without making any progress in the execution, in a livelock condition a process always does something but is unable to make any progress in the execution completion. The livelock condition is better explained with the real world example, two people attempting to cross each other in a narrow corridor. Both the persons move towards each side of the corridor to allow the opposite person to cross. Since the corridor is narrow, none of them are able to cross each other. Here both of the persons perform some action but still they are unable to achieve their target, cross each other. We will make the livelock, the scenario more clear in a later section—The Dining Philosophers ' Problem, of this chapter.

Starvation: In the multitasking cont on is the condition in which a process does not get the resources required to continue its execution for a long time. As time progresses the process starves on resource. Starvation may arise due to various conditions like byproduct of preventive measures of deadlock, scheduling policies favoring high priority tasks and tasks with shortest execution time, etc.

3.3.1.3 The Dining Philosophers' Problem: The *'Dining philosophers 'problem'* is an interesting example for synchronization issues in resource utilization. The terms 'dining', 'philosophers', etc. may sound awkward in the operating system context, but it is the best way to explain technical things abstractly using non-technical terms. Now coming to the problem definition:

Five philosophers (It can be 'n'. The number 5 is taken for illustration) are sitting around a round table, involved in eating and brainstorming. At any point of time each philosopher will be in any one of the three states: eating, hungry or brainstorming. (While eating the philosopher is not involved in brainstorming and while brainstorming the philosopher is not involved in eating). For eating, each philosopher requires 2 forks. There are only 5 forks available on the dining table ('n' for 'n' number of philosophers) and they are arranged in a fashion one fork in between two philosophers. The philosopher can only use the forks on his/her immediate left and right that too in the order pickup the left fork first and then the right fork. Analyze the situation and explain the possible outcomes of this scenario.

Let's analyze the various scenarios that may occur in this situation.

Scenario 1: All the philosophers involve in brainstorming together and try to eat together. Each philosopher picks up the left fork and is unable to proceed since two forks are required for eating the spaghetti present in the plate. Philosopher 1 thinks that Philosopher 2 sitting to the right of him/her will put the fork down and waits for it. Philosopher 2 thinks that Philosopher 3' sitting to the right of him/her will

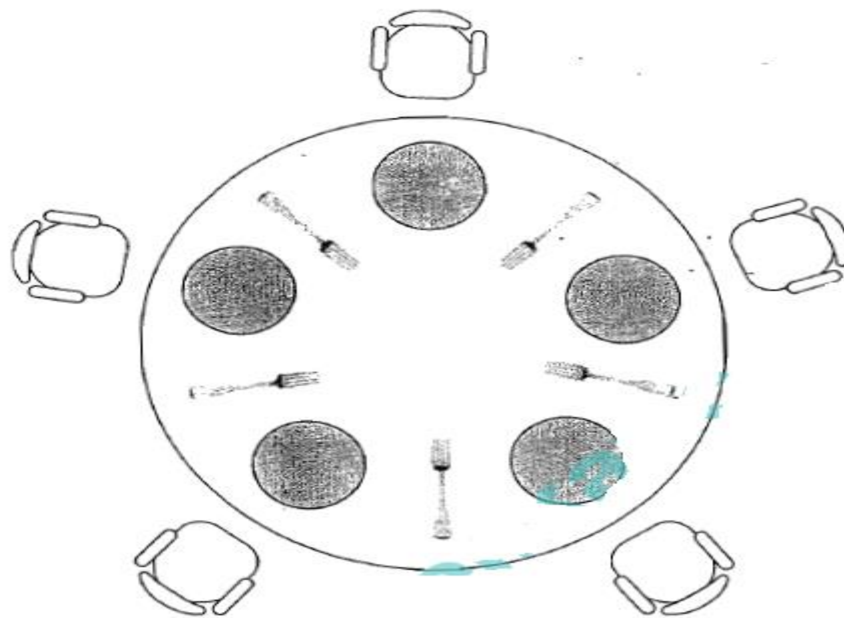


Figure: Visualization of the 'Dining Philosophers' Problem'

put the fork down and waits for it, and so on. This forms a circular chain of un-granted requests. If the philosophers continue in this state waiting for the fork from the philosopher sitting to the right of each, they will not make any progress in eating and this will result in starvation of the philosophers and deadlock.

Scenario 2: All the philosophers start brainstorming together. One of the philosophers is hungry and he/ she picks up the left fork. When the philosopher is about to pick up the right fork, the philosopher sitting. to his right also become hungry and tries to grab the left fork which is the right fork of his neighboring philosopher who is trying to lift it, resulting in a 'Race condition'..

Scenario 3: All the philosophers involve in brainstorming together and by to eat together. Each philosopher picks up the left fork and is unable to proceed, since two forks are required for eating the spaghetti present in the plate. Each of them anticipates that the adjacently sitting philosopher will put his/her fork down and waits for a fixed duration grid after this puts the fork down. Each of them again tries to lift the fork after a fixed duration of time. Since all philosophers are trying to lift the fork at the same time, none of them will be able to grab two forks. This condition leads to livelock and starvation of philosophers, where each philosopher tries to do something, but they are unable to make any progress in achieving the target.

Figure illustrates these scenarios.

Solution: We need to find out alternative solutions to avoid the.deadlock, livelock, racing and starvation condition that may arise due to the concurrent access of forks by philosophers. This situation can be handled in many ways by allocating the forks in different allocation techniques including round Robin allocation, FIFO allocation: etc.

But the requirement is that the solution should be optimal, avoiding deadlock and starvation of the philosophers and allowing maximum number of philosophers to eat at a time. One solution that we could think of is:

• Imposing rules in accessing the forks by philosophers, like: The philosophers should put down the fork he/she already have in hand (left fork) after waiting for a fixed duration for the second fork (right fork) and should wait for a fixed time before making the next attempt.

This solution works fine to some extent, but, if all the philosophers try to lift the forks at the same time, a livelock situation is resulted.

Another solution which gives maximum concurrency that can be thought of is each philosopher ac-quires a semaphore (mutex) before picking up any fork. When a philosopher feels hungry he/she checks whether the philosopher sitting to the left and right of him is already using the fork, by checking the state of the associated semaphore. If the forks are in use by the neighboring philosophers, the philosopher waits till the forks are available. A philosopher when finished eating puts the forks down and informs the philosophers sitting to his/her left and right, who are hungry (waiting for the forks), by signaling the semaphores associated with the forks.

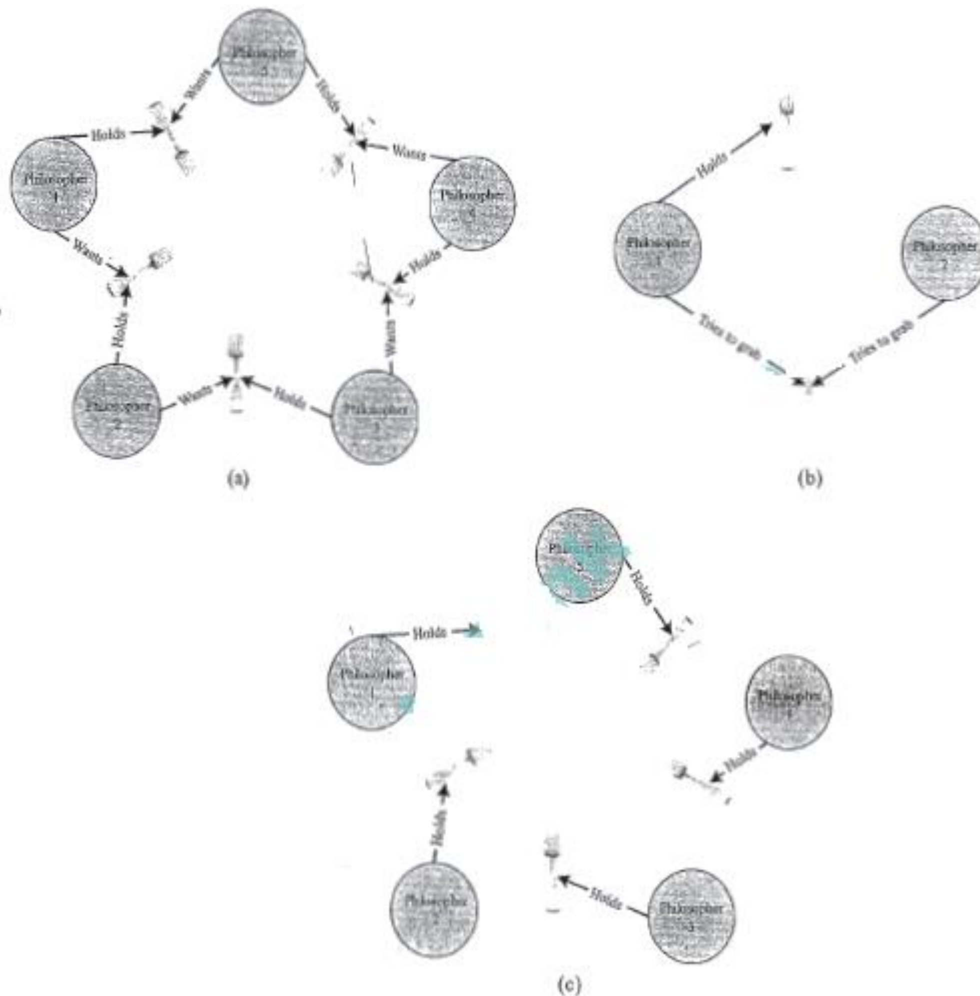


Figure: The 'Real Problems' in the 'Dining Philosophers problem' (a) Starvation and Deadlock (b) Racing (c) Livelock and Starvation

We will discuss about semaphores and mutexes at a latter section of this chapter. In the operating system context, the dining philosophers represent the processes and forks represent the resources. The dining philosophers' problem is an analogy of processes competing for shared resources and the different problems like racing, deadlock, starvation and livelock arising from the competition.

3.3.1.4 Producer-Consumer/Bounded Buffer Problem: Producer-Consumer problem is a common data sharing problem where two processes concurrently access a shared buffer with fixed size. A thread/process which produces data is called 'Producer thread/process' and a thread/process which consumes the data produced by a producer thread/process is known as 'Consumer thread/process'. Imagine a situation where the producer thread keeps on producing data and puts it into the buffer and the consumer thread keeps on consuming the data from the buffer and there is no synchronization between the two. There may be chances where in which the producer produces data at a faster rate than the rate at which it is consumed by the consumer. This will lead to 'buffer overrun' where the producer tries to put data to a full buffer. If the consumer consumes data at a faster rate than the rate at which it is produced by

the producer, it will lead to the situation `buffer under-run' in which the consumer tries to read from an empty buffer. Both of these conditions will lead to inaccurate data and data loss. The following code snippet illustrates the producer-consumer problem

```
#include <windows.h>
#include <stdio.h>
#define N 20 //Define buffer size as 20
int buffer[N];//Shared buffer for producer & consumer
//***************************************************************
//Producer thread
void producer_thread(void) {
        int x;
        while(true) {
                for(x=0;x<N;x++)
                {
                //Fill buffer with random data
                buffer[x]= rand()%1000;
                printf("Produced : Buffer [%d] = %4d\n", x,
                buffer[x]);
                Sleep(25);
                }
        }
}
//***************************************************************
//Consumer thread
void consumer_thread(void) {
        int y=0,value;
        while(true) {
                for(y=0;y<N;y++)


                {
                value=buffer[y];
                printf("Consumed : Buffer [%d] = %4d\n", y, value);
                Sleep(20);
                }
        }
}
//***************************************************************
```

www.Ask

```
//Main Thread
int main()
{
        DWORD thread_id;
        //Create Producer thread
        CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)producer_thread,
                        NULL,0,&thread_id);
        //Create Consumer thread
        CreateThread(NULL,0,
                        (LPTHREAD_START_ROUTINE)consumer_thread,
                        NULL,0,&thread_id);
        //Wait for some time and exit
        Sleep(500);
        return 0;
}
```

Here the 'producer thread' produces random numbers and puts it in a buffer of size 20. If the 'producer thread' fills the buffer fully it re-starts the filling of the buffer from the bottom. The 'consumer thread' consumes the data produced by, the 'producer thread'. For consuming the data, the 'consumer thread' reads the buffer which is shared with the 'producer thread'. Once the 'consumer thread' consumes all the data, it starts consuming the data from the bottom of the buffer. These two threads run independently and are scheduled for execution based on the scheduling policies adopted by the OS. The different situations that may arise based on the scheduling of the 'producer thread' and 'consumer thread' is listed below.

1. 'Producer thread' is scheduled more frequently than the 'consumer thread': There are chances for overwriting the data in the buffer by the 'producer thread'. This leads to inaccurate data.

2. Consumer thread' is scheduled more frequently than the 'producer thread': There are chances for reading the old data in the buffer again by the 'consumer thread'. This will also lead to inaccurate data.

The output of the above program when executed on a Windows XP machine is shown in Fig. 10.29. The output shows that the consumer thread runs faster than the producer thread and most often leads to buffer under-run and thereby inaccurate data.

Note
It should be noted that the scheduling of the threads *'producer_thread'* ,and '*consumer_thread*' is OS kernel scheduling policy dependent and you may not get the same output all the time when you run this piece of code in Windows XP.

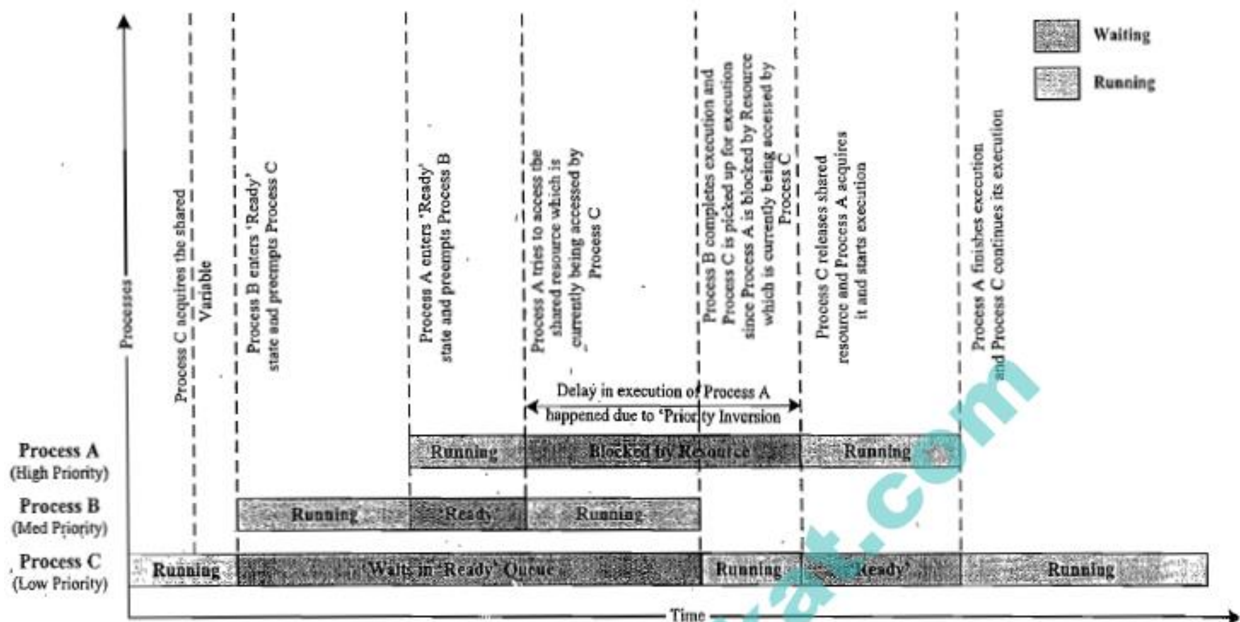
The producer-consumer problem can be rectified in various methods. One simple solution is the *`sleep and wake-up'*. The 'sleep and wake-up' can be implemented in various process synchronization techniques like semaphores, mutex, monitors, etc. We will discuss it in a latter section of this chapter.

```
"C:\Program Files\Microsoft Visual Studio\MyProjects\consume\Debug\consume.exe"
Produced : Buffer [0] =   41
Consumed : Buffer [0] =   41
Consumed : Buffer [1] =    0
Produced : Buffer [1] =  467
Consumed : Buffer [2] =    0
Produced : Buffer [2] =  334
Consumed : Buffer [3] =    0
Produced : Buffer [3] =  500
Consumed : Buffer [4] =    0
Produced : Buffer [4] =  169
Consumed : Buffer [5] =    0
Consumed : Buffer [6] =    0
Produced : Buffer [5] =  724
Consumed : Buffer [7] =    0
Produced : Buffer [6] =  478
Consumed : Buffer [8] =    0
Produced : Buffer [7] =  358
Consumed : Buffer [9] =    0
Produced : Buffer [8] =  962
Consumed : Buffer [10] =    0
Consumed : Buffer [11] =    0
Produced : Buffer [9] =  464
Consumed : Buffer [12] =    0
Produced : Buffer [10] =  705
Consumed : Buffer [13] =    0
Produced : Buffer [11] =  145
Consumed : Buffer [14] =    0
Produced : Buffer [12] =  281
Consumed : Buffer [15] =    0
Consumed : Buffer [16] =    0
Produced : Buffer [13] =  827
Consumed : Buffer [17] =    0
Produced : Buffer [14] =  961
```

Figure: Output of win32 program illustrating producer-consumer problem

3.3.1.5 Readers-Writers Problem: Tire Readers-Writers problem is a common issue observed in processes competing for limited shared resources. The Readers-Writers problem is characterized by multiple processes trying to read and write shared data concurrently. A typical real-world example for the Readers-Writers problem is the banking system where one process tries to read the account information like available balance and the other process tries to update the available balance for that account. This may result in inconsistent results. If multiple processes try to read a shared data concurrently it may not create any impacts, whereas when multiple processes try to write and read concurrently it will definitely create inconsistent results. Proper synchronization techniques should be applied to avoid the readers-writers problem. We will discuss about the various synchronization techniques in a later section of this chapter.

3.3.1.6 Priority Inversion: Priority inversion is the byproduct of the combination of blocking based (lock based) process synchronization and pre-emptive priority scheduling. 'Priority inversion' is the condition in which a high priority task needs to wait for a low priority task to release a resource which is shared between the high priority task and the low priority task, and a medium priority task which doesn't require the shared resource continue its execution by preempting the low priority task. Priority based preemptive scheduling technique ensures that a high priority task is always executed first, whereas the lock based process synchronization mechanism (like mutex, semaphore, etc.) ensures that a process will not access a shared resource, which is currently in use by another process. The synchronization technique is only interested in avoiding conflicts that may arise due to the concur-rent access of the shared resources and not at all bothered about the priority of the process which tries to access the shared resource. In fact, the priority based preemption and lock based synchronization are the two contradicting OS primitives. Priority inversion is better explained with the following scenario: Let Process A, Process B and Process C be three processes with priorities High, Medium and Low respectively. Process A and Process C share a variable 'X' and the access to this variable is synchronized through a mutual exclusion mechanism like Binary Semaphore S.



Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'. Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted, and 'Process B' starts executing. Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A' is of higher priority than 'Process B', 'Process B' is preempted, and 'Process A' is scheduled for execution. 'Process A' involves accessing of shared variable 'X' which is currently being accessed by 'Process C'. Since 'Process C' acquired the semaphore for signaling the access of the shared variable 'X', 'Process A' will not be able to access it. Thus 'Process A' is put into

blocked state (This condition is called Pending on resource). Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task. The highest priority process 'Process A' has to wait till 'Process C' gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was 'Ready'. Priority inversion may be sporadic in nature but can lead to potential damages as a result f missing critical deadlines. Literally speaking, priority inversion 'inverts' the priority of a high priority task with that of a low priority task. Proper workaround mechanism should be adopted for handling the priority inversion problem. The commonly adopted priority inversion workarounds are:

through a mutual exclusion mechanism like Binary Semaphore S. Imagine a situation where Process C is ready and is picked up for execution by the scheduler and 'Process C' tries to access the shared variable 'X'. 'Process C' acquires the 'Semaphore S' to indicate the other processes that it is accessing the shared variable 'X'. Immediately after 'Process C' acquires the 'Semaphore S', 'Process B' enters the 'Ready' state. Since 'Process B' is of higher priority compared to 'Process C', 'Process C' is preempted, and 'Process B' starts executing. Now imagine 'Process A' enters the 'Ready' state at this stage. Since 'Process A' is of higher priority than 'Process B', 'Process B' is preempted, and 'Process A' is scheduled for execution. 'Process A' involves accessing of shared variable 'X' which is currently being accessed by 'Process C'. Since 'Process C' acquired the semaphore for signaling the access of the shared variable 'X', 'Process A' will not be able to access it. Thus 'Process A' is put into blocked state (This condition is called Pending on resource). Now 'Process B' gets the CPU and it continues its execution until it relinquishes the CPU voluntarily or enters a wait state or preempted by another high priority task. The highest priority process 'Process A' has to wait till 'Process C' gets a chance to execute and release the semaphore. This produces unwanted delay in the execution of the high priority task which is supposed to be executed immediately when it was 'Ready'. Priority inversion may be sporadic in nature but can lead to potential damages as a result f missing critical deadlines. Literally speaking, priority inversion 'inverts' the priority of a high priority task with that of a low priority task. Proper workaround mechanism should be adopted for handling the priority inversion problem. The commonly adopted priority inversion workarounds are:

Priority Inheritance: A low-priority task that is currently accessing (by holding the lock) a shared resource requested by a high-priority task temporarily 'inherits' the priority of that high-priority task, from the moment the high-priority task raises the request. Boosting the priority of the low priority task to that of the priority of the task which requested the shared resource holding by the low priority task eliminates the preemption of the low priority task by other tasks whose priority are below that of the task requested the shared resource 'and thereby reduces the delay in waiting to get the resource requested by the high priority task. The priority of the low priority task which is temporarily boosted to high is brought to the original value when it releases the shared resource. Implementation of Priority inheritance workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Figure.

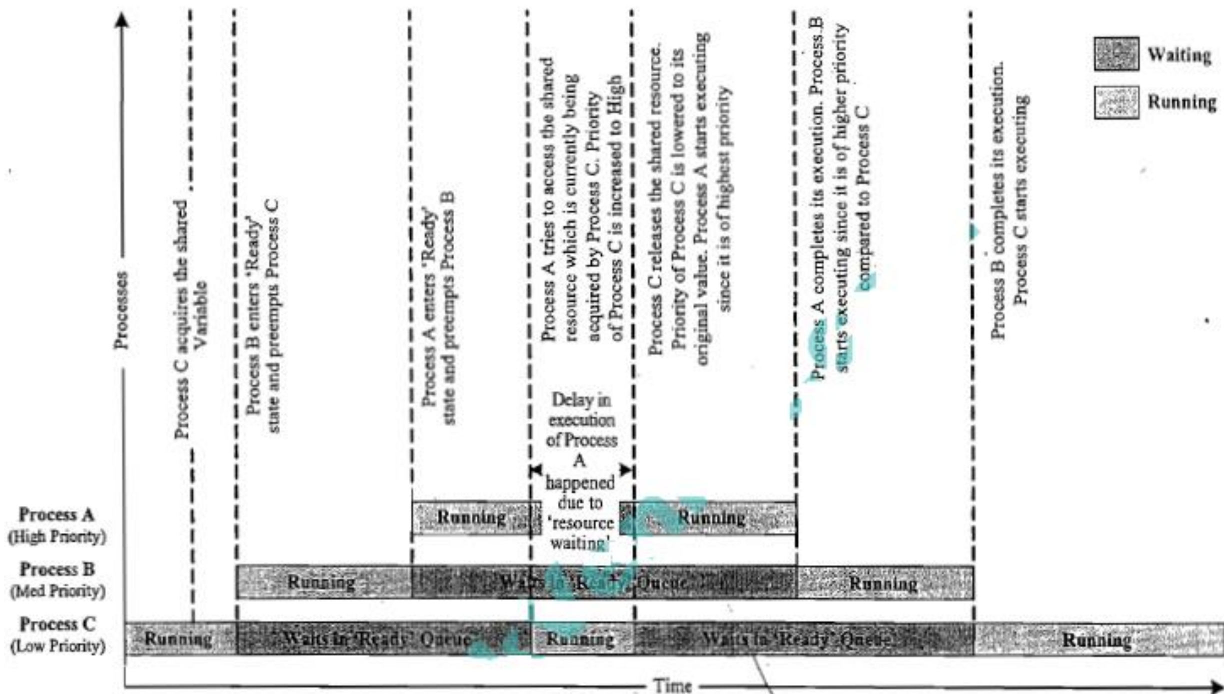


Figure: Handling Priority Inversion problem with priority Inheritance.

Priority inheritance is only a work around and it will not eliminate the delay in waiting the high priority task to get the resource from the low priority task. The only thing is that it helps the low priority task to continue its execution and release the shared resource as soon as possible. The moment, at which the low priority task releases the shared resource, the high priority task kicks the low priority task out and grabs the CPU - A true form of selfishness. Priority inheritance handles priority inversion at the cost of run-time overhead at scheduler. It imposes the overhead of checking the priorities of all tasks which tries to access shared resources and adjust the priorities dynamically.

Priority Ceiling: In *'Priority Ceiling'*, a priority is associated with each shared resource. The priority associated to each resource is the priority of the highest priority task which uses this shared resource. This priority level is called 'ceiling priority'. Whenever a task accesses a shared resource, the scheduler elevates the priority of the task to that of the ceiling priority of the resource. If the task which accesses the shared resource is a low priority task, its priority is temporarily boosted to the priority of the highest priority task to which the resource is also shared. This eliminates the pre-emption of the task by other medium priority tasks leading to priority inversion. The priority of the task is brought back to the original level once the task completes the accessing of the shared resource. 'Priority Ceiling' brings the added advantage of sharing resources without the need for synchronization techniques like locks. Since the priority of the task accessing a shared resource is boosted to the highest priority of the task among which the resource is shared, the concurrent access of shared resource is automatically handled. Another advantage of 'Priority Ceiling' technique is that all the overheads are at compile time instead of run-time. Implementation of 'priority ceiling' workaround in the priority inversion problem discussed for Process A, Process B and Process C example will change the execution sequence as shown in Figure.

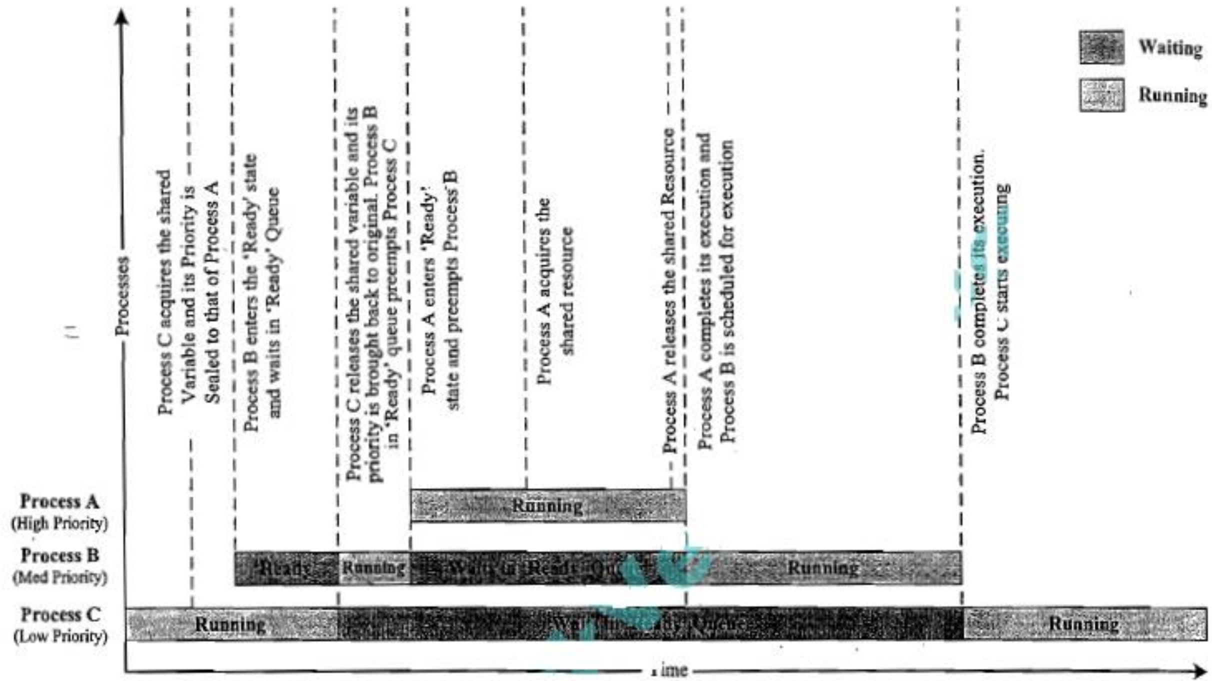


Figure: Handling Priority Inversion problem with priority Ceiling.

The biggest drawback of 'Priority Ceiling' is that it may produce hidden priority inversion. With 'Priority Ceiling' technique, the priority of a task is always elevated no matter another task wants the shared resources. This unnecessary priority elevation always boosts the priority of a low priority task to that of the highest priority tasks among which the resource is shared and other tasks with priorities higher than that of the low priority task is not allowed to preempt the low priority task when it is accessing a shared resource. This always gives the low priority task the luxury of running at high priority when accessing shared resources.

3.3.2 Task Synchronization Techniques

So far we discussed about the various task/process synchronization issues encountered in multitasking systems due to concurrent resource access. Now let's have a discussion on the various techniques used for synchronization in concurrent access in multitasking. Process/Task synchronization is essential for

1. Avoiding conflicts in resource access (racing, deadlock, starvation, livelock, etc.) in a multitasking environment.

2. Ensuring proper sequence of operation across processes. The producer consumer problem is a typical example for processes requiring proper sequence of operation. In producer consumer problem, accessing the shared buffer by different processes is not the issue; the issue is the writing process should write to the shared buffer only if the buffer is not full and the consumer thread should not read from the buffer if it is empty. Hence proper synchronization should be provided to implement this sequence of operations.

3. Communicating between processes.

The code memory area which holds the program instructions (piece of code) for accessing a shared resource (like shared memory, shared variables, etc.) is known as 'critical section'. In order to synchronize the access to shared resources, the access to the critical section should be exclusive. The exclusive access to critical section of code is provided through mutual exclusion mechanism. Let us have a look at how mutual exclusion is important in concurrent access. Consider two processes Process A and Process B running on a multitasking system. Process A is currently running and it enters its critical section. Before Process A completes its operation in the critical section, the scheduler preempts Process A and schedules Process B for execution (Process B is of higher priority compared to Process A). Process B also contains the access to the critical section which is already in use by Process A. If Process B continues its execution and enters the critical section which is already in use by Process A, a racing condition will be resulted. A mutual exclusion policy enforces mutually exclusive access of critical sections. Mutual exclusions can be enforced in different ways. Mutual exclusion blocks a process. Based on the behaviour of the blocked process, mutual exclusion methods can be classified into two categories. In the following section we will discuss them in detail.

3.3.2.1 Mutual Exclusion through Busy Waiting/Spin Lock: 'Busy waiting' is the simplest method for enforcing mutual exclusion. The following code snippet illustrates how 'Busy waiting' enforces mutual exclusion.

```
//Inside parent thread/main thread corresponding to a process
bool bFlag; //Global declaration of lock Variable.
bFlag= FALSE; //Initialise the lock to indicate it is available.
//..............................................................................
//Inside the child threads/threads of a process
while(bFlag == TRUE); //Check the lock for availability
bFlag=TRUE; //Lock is available. Acquire the lock
//Rest of the source code dealing with shared resource access
```

The 'Busy waiting' technique uses a lock variable for implementing mutual exclusion. Each process/ thread checks this lock variable before entering the critical section. The lock is set to '1' by a process/ thread if the process/thread is already in its critical section; otherwise the lock is set to '0'. The major challenge in implementing the lock variable based synchronization is the non-availability of a single atomic instruction which combines the reading, comparing and setting of the lock variable. Most often the three different operations related to the locks, viz. the operation of Reading the lock variable, checking its present value, and setting it are achieved with multiple low-level instructions. The low-level implementation of these operations are dependent on the underlying processor instruction set and the (cross) compiler in use. The low-level implementation of the 'Busy waiting' code snippet, which we discussed earlier, under Windows XP operating system running on an Intel Centrino Duo processor is given below. The code snippet is compiled with Microsoft Visual Studio 6.0 compiler.

```
--- D:\Examples\counter.cpp -------------------------------------
1:    #include <stdio.h>
2:    #include <windows.h>

3:
4:    int main()
5:    {
//Code memory         Opcode       Operand
00401010              push         ebp
00401011              mov          ebp,esp
00401013              sub          esp,44h
00401016              push         ebx
00401017              push         esi
00401018              push         edi
00401019              lea          edi,[ebp-44h]
0040101C              mov          ecx,11h
00401021              mov          eax,0CCCCCCCCh
00401026              rep stos     dword ptr [edi]
6:    //Inside parent thread/ main thread corresponding to a process
7:    bool bFlag; //Global declaration of lock Variable.
8:    bFlag= FALSE; //Initialise the lock to indicate it is //available.
00401028    mov         byte ptr [ebp-4],0
```

```
9:      //...........................................................................
10:     //Inside the child threads/ threads of a process
11:     while(bFlag == TRUE); //Check the lock for availability
0040102C    mov         eax,dword ptr [ebp-4]
0040102F    and         eax,0FFh
00401034    cmp         eax,1
00401037    jne         main+2Bh (0040103b)
00401039    jmp         main+1Ch (0040102c)
12:     bFlag=TRUE; //Lock is available. Acquire the lock
0040103B    mov         byte ptr [ebp-4],1
```

The assembly language instructions reveals that the two high level instructions (while(bFlag==false); and bFlag=true;), corresponding to the operation of reading the lock variable, checking its present value and setting it is implemented in the processor level using six low level instructions. Imagine a situation where 'Process 1' read the lock variable and tested it and found that the lock is available and it is about to set the lock for acquiring the critical section. But just before 'Process 1' sets the lock variable, 'Process 2' preempts 'Process 1' and starts executing. 'Process 2' contains a critical section code and it tests the lock variable for its availability. Since 'Process 1' was unable to set the lock variable, its state is still '0' and 'Process 2' sets it and acquires the critical section. Now the scheduler preempts 'Process 2' and schedules 'Process 1' before 'Process 2' leaves the critical section. Remember, `Process l' was preempted at a point just before setting the lock variable ('Process 1' has already tested the lock variable just before it is preempted and found that the lock is available). Now 'Process 1' sets the lock variable and enters the critical section. It violates the mutual exclusion policy and may pro-duce unpredicted results.

Device Driver

Device driver is a piece of software that acts as a bridge between the operating system and the hardware. In an operating system based product architecture, the user applications talk to the Operating System kernel for all necessary information exchange including communication with the hardware peripherals. The architecture of the OS kernel will not allow direct device access from the user application. All the device related access should flow through the OS kernel and the OS kernel mutes it to the concerned hardware peripheral. OS provides interfaces in the form of Application Programming Interfaces (APIs) for accessing the hardware. The device driver abstracts the hardware from user applications. The topology of user applications and hardware interaction in an RTOS based system is depicted in Fig.

Device drivers are responsible for initiating and managing the communication with the hardware peripherals. They are responsible for establishing the connectivity, initializing the hardware (setting up various registers of the hardware device) and transferring data. An embedded product may contain different types of hardware components like Wi-Fi module, File systems, Storage device interface, etc. The initialization of these devices and the protocols required for communicating with these devices may be different. All these requirements are implemented in drivers and a single driver will not be able to satisfy all these. Hence each hardware (more specifically each class of hardware) requires a unique driver component.

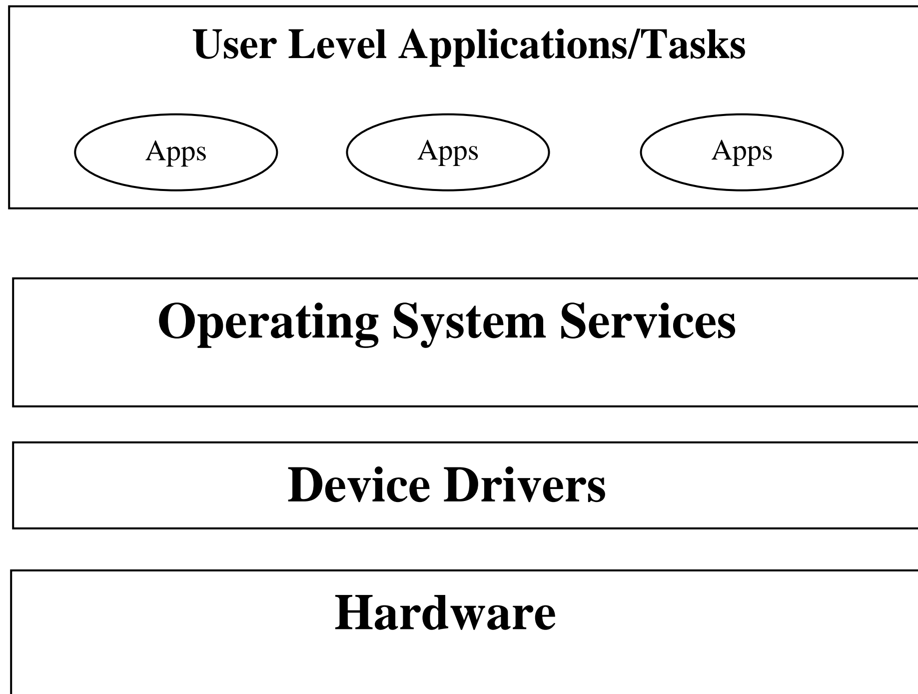


Figure: Role of device driver in Embedded OS based products

Certain drivers come as part of the OS kernel and certain drivers need to be installed on the fly. For example, the program storage memory for an embedded product, say NAND Flash memory requires a NAND Flash driver to read and write data from/to it. This driver should come as part of the OS kernel image. Certainly the OS will not contain the drivers for all devices and peripherals under the Sun. It contains only the necessary drivers to communicate with the onboard devices (Hardware devices which are part of the platform) and for certain set of devices supporting standard protocols and device class (Say USB Mass storage device or HID devices like Mouse/keyboard). If an external device, whose driver software is not available with OS kernel image, is connected to the embedded device (Say a medical device with custom USB class implementation is connected to the USB port of the embedded product), the OS prompts the user to install its driver manually. Device drivers which are part of the OS image are known as 'Built-in drivers' or 'On-board drivers'. These drivers are loaded by the OS at the time of booting the device and are always kept in the RAM. Drivers which need to be installed for accessing a device are known. as 'Installable

drivers'. These drivers are loaded by the OS on a need basis. Whenever the device is connected, the OS loads the corresponding driver to memory. When the device is removed, the driver is unloaded from memory. The Operating system maintains a record of the drivers corresponding to each hardware.

The implementation of driver is OS dependent. There is no universal implementation for a driver. How the driver communicates with the kernel is dependent on the OS structure and implementation. Different Operating Systems follow different implementations.

It is very essential to know the hardware interfacing details like the memory address assigned to the device, the Interrupt used, etc. of on-board peripherals for writing a driver for that peripheral. It varies on the hardware design of the product. Some Real-Time operating systems like 'Windows CE' support a layered architecture for the driver which separates out the low level implementation from the OS specific interface. The low level implementation part is generally known as Platform Dependent Device (PDD) layer. The OS specific interface part is known as Model Device Driver (MDD) or Logical Device Driver (LDD). For a standard driver, for a specific operating system, the MDD/LDD always remains the same and only the PDD part needs to be modified according to the target hardware for a particular class of devices.

Most of the time, the hardware developer provides the implementation for all on board devices for a specific OS along with the platform. The drivers are normally shipped in the form of Board Support Package. The Board Support Package contains low level driver implementations for the onboard peripherals and OEM Adaptation Layer (OAL) for accessing the various chip level functionalities and a bootloader for loading the operating system. The OAL facilitates communication between the Operating System (OS) and the target device and includes code to handle interrupts, timers, power management, bus abstraction; generic I/O control codes (IOCTLs), etc. The driver files are usually in the form of a dll file. Drivers can run on either user space or kernel space. Drivers which run in user space are known as user mode drivers and the drivers which run in kernel space are known as kernel mode drivers. User mode drivers are safer than kernel mode drivers. If an error or exception occurs in a user mode driver, it won't affect the services of the kernel. On the other hand, if an exception occurs in the kernel mode driver, it may lead to the kernel crash. The way how a device driver is written and how the interrupts are handled in it are operating system and target hardware specific. However regardless of the OS types, a device driver implements the following:

1. Device (Hardware) Initialization and Interrupt configuration
2. Interrupt handling and processing
3. Client interfacing (Interfacing with user applications)

The Device (Hardware) initialisation part of the driver deals with configuring the different registers of the device (target hardware). For example configuring the I/O port line of the processor as Input or output line and setting its associated registers for building a General Purpose IO (GPIO) driver. The interrupt configuration part deals with configuring the interrupts that needs to be associated with the hardware. In the case of the GPIO driver, if the intention is to generate an interrupt when the Input line is asserted, we need to configure the interrupt associated with the I/O port by modifying its associated registers. The basic Interrupt configuration involves the following.

1. Set the interrupt type (Edge Triggered (Rising/Flailing) or Level Triggered (Low or High)), enable the interrupts and set the interrupt priorities.
2. Bind the Interrupt with an Interrupt Request (IRQ). The processor identifies an interrupt through IRQ. These IRQs are generated by the Interrupt Controller. In order to identify an interrupt the interrupt needs to be bonded to an IRQ.
3. Register an Interrupt Service Routine (ISR) with an Interrupt Request (IRQ). ISR is the handler for an Interrupt. In order to service an interrupt, an ISR should be associated with an IRQ. Registering an ISR with an IRQ takes care of it.

With these the interrupt configuration is complete. If an interrupt occurs, depending on its priority, it is serviced and the corresponding ISR is invoked. The processing part of an interrupt is handled in an ISR. The whole interrupt processing can be done by the ISR itself or by invoking an Interrupt Service Thread (IST). The IST performs interrupt processing on behalf of the ISR. To make the ISR compact and short, it is always advised to use an IST for interrupt processing. The intention of an interrupt is to send or receive command or data to and from the hardware device and make the received data available to user programs for application specific processing. Since interrupt processing happens at kernel level, user applications may not have direct access to the drivers to pass and receive data. Hence it is the responsibility of the Interrupt processing routine or thread to inform the user applications that au interrupt is occurred and data is available for further processing. The client interfacing part of the device driver takes care of this. The client interfacing implementation makes use of the Inter Process communication mechanisms supported by the embedded OS for communicating and synchronising with user applications and drivers. For example, to inform a user application that an interrupt is occurred and the data received from the device is placed in I shared buffer, the client interfacing code can signal (or set) an event. The user application creates the event, registers it and waits for the driver to signal it. The driver can share the received data through shared memory techniques. IOCTLs, shared buffers, etc. can be used for data sharing. The story line is incomplete without performing an interrupt done (Interrupt processing completed) functionality in the driver. Whenever an interrupt is asserted, while vectoring to its corresponding ISR, all interrupts of equal and low'5riorities are disabled. They are re-enable only on executing the interrupt done function (Same as the Return from Interrupt RETI instruction execution for 8051) by the driver. The interrupt done function can be invoked at the end of corresponding ISR or IST.

UNIT-IV

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

SYLLABUS:

Host and target machines, linker/locators for embedded software, getting embedded software into the target system; Debugging techniques: Testing on host machine, using laboratory tools, an example system.

I.HOST AND TARGET MACHINES:

- **Host:**
– A computer system on which all the programming tools run
– Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device.
- **Target:**
– After writing the program, compiled, assembled and linked, it is moved to target
– After development, the code is cross-compiled, translated – cross-assembled, linked into target processor instruction set and located into the target.

Host System	Target Computer System
Writing, editing a program, compiling it, linking it, debugging it are done on host system	After the completion of programming work, it is moved from host system to target system.
It is also referred as Work Station	No other name
Software development is done in host system for embedded system	Developed software is shifted to customer from host
Compiler, linker, assembler, debugger are used	Cross compiler is also used
Unit testing on host system ensures software is working properly	By using cross compiler, unit testing allows to recompile code ,execute, test on target system
Stubs are used	Real libraries
Programming centric	Customer centric

Cross Compilers:

- A **cross compiler that runs on host system** and produces the binary instructions that will be understood by your target microprocessor.
- *A **cross compiler** is a compiler capable of creating executable code for a*

platform other than the one on which the compiler is running. For example, a compiler that runs on aWindows 7 PC but generates code that runs on Android smartphone is a cross compiler.

- Most desktop systems are used as hosts come with compilers, assemblers, linkers that will run on the host. These tools are called native tools.

- Suppose the native compiler on a Windows NT system is based on Intel Pentium. This compiler may possible if target microprocessor is also Intel Pentium. This is not possible if the target microprocessor is other than Intel i.e. like MOTOROLA, Zilog etc.

- A cross compiler that runs on host system and produces the binary instructions that will be understood by your target microprocessor. This cross compiler is a program which will do the above task. If we write C/C++ source code that could compile on native compiler and run on host, we could compile the same source code through cross compiler and make run it run on target also.

- That may not possible in all the cases since there is no problem with if, switch and loops statements for both compilers but there may be an error with respect to the following:

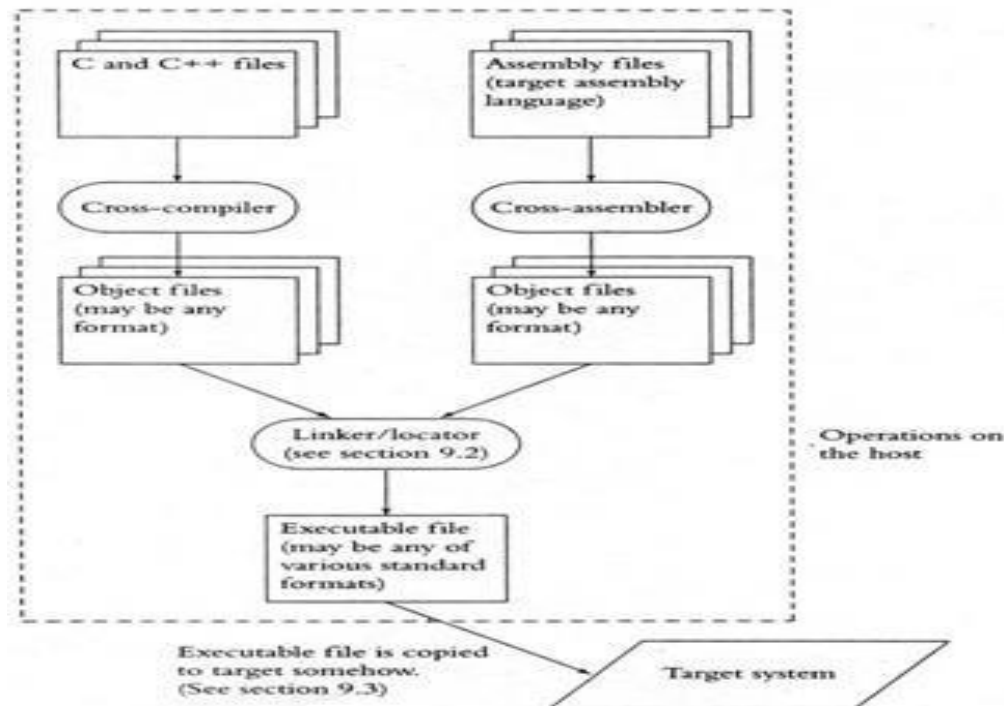
    - In Function declarations
    - The size may be different in host and target
    - Data structures may be different in two machines.
    - Ability to access 16 and 32 bit entries reside at two machines.

  Sometimes cross compiler may warn an error which may not be warned by native complier.

Cross Assemblers and Tool Chains:

- Cross assembling is necessary if target system cannot run an assembler itself.
- **A cross assembler is a program that runs on host produces binary instructions appropriate for the target.** The input to the cross assembler is assembly language file (.asm file) and output is binary file.
- A cross-assembler is just like any other assembler except that it runs on some CPU other than the one for which it assembles code.

Tool chain for building embedded software shown below:



The figure shows the process of building software for an embedded system.

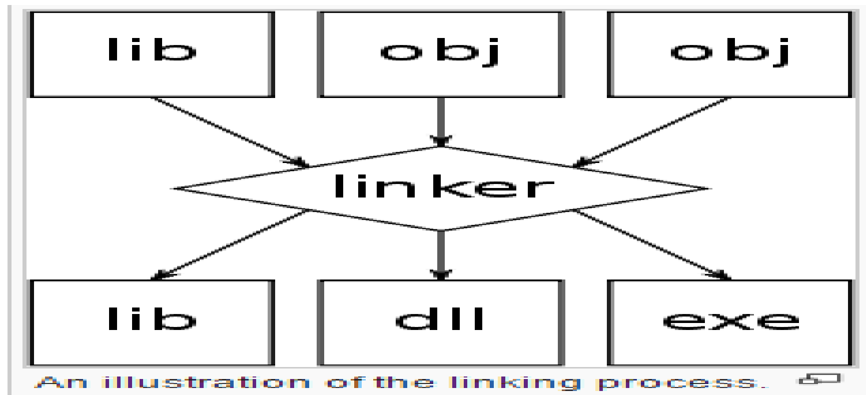
As you can see in figure the output files from each tool become the input files for the next. Because of this the tools must be compatible with each other.

A set of tools that is compatible in this way is called tool chain. Tool chains that run on various hosts and builds programs for various targets.

II. LINKER/LOCATORS FOR EMBEDDED SOFTWARE:

- **Linker:**
  - a linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.
- **Locator:**
  - *locate* embedded binary code into target processors
  - produces target machine code (which the locator glues into the RTOS) and the combined code (called map) gets copied into the target ROM

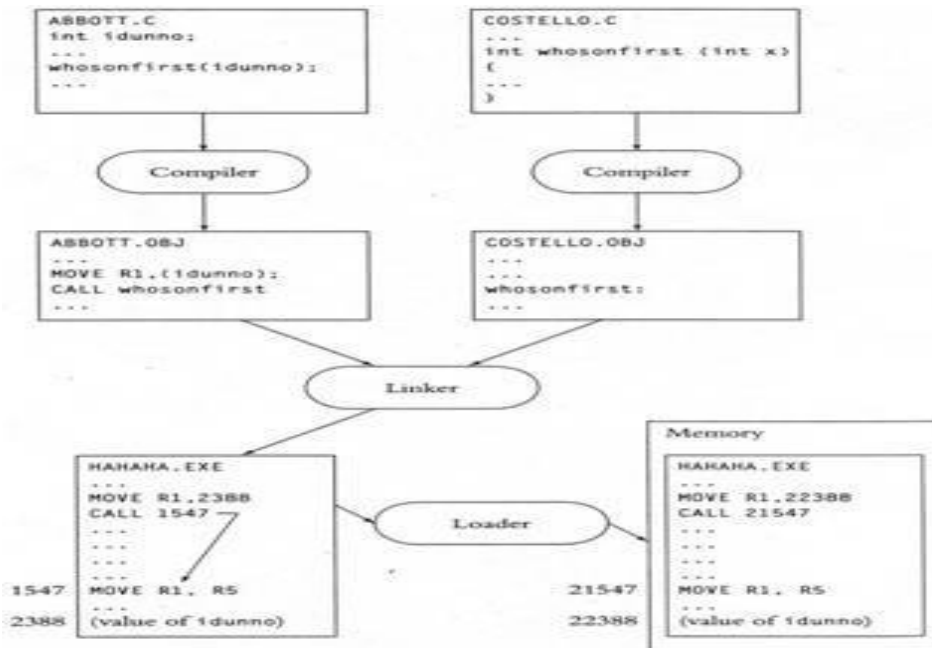
Linking Process shown below:



An illustration of the linking process.

- The **native linker** creates a file on the disk drive of the host system that is read by a part of operating system called the loader whenever the user requests to run the programs.
- **The loader** finds memory into which to load the program, copies the program from the disk into the memory
- Address Resolution:

Native Tool Chain:



Explanation for above native tool chain figure:

- Above Figure shows the process of building application software with native tools. One problem in the tool chain must solve is that many microprocessor instructions contain the addresses of their operands.

- the above figure MOVE instruction in ABBOTT.C will load the value of variable idunno into register R1 must contain the address of the variable. Similarly CALL instruction must contain the address of the whosonfirst. This process of solving problem is called address resolution.

- When abbott.c file compiling,the compiler does not have any idea what the address of idunno and whosonfirst() just it compiles both separately and leave them as object files for linker.

- Now linker will decide that the address of idunno must be patched to whosonfirst() call instructoin. When linker puts the two object files together, it figures out idunno and whosonfirst() are in relation for execution and places in executable files.

- After loader copies the program into memory and exactly knows where idunno and whosonfirst() are in memory. This whole process called as **address resolution.**

Output File Formats:

In most embedded systems there is no loader, when the locator is done then output will be copied to target.

Therefore the locator must know where the program resides and fix up all memories.

Locators have mechanism that allows you to tell them where the program will be in the target system. Locators use any number of different output file formats.

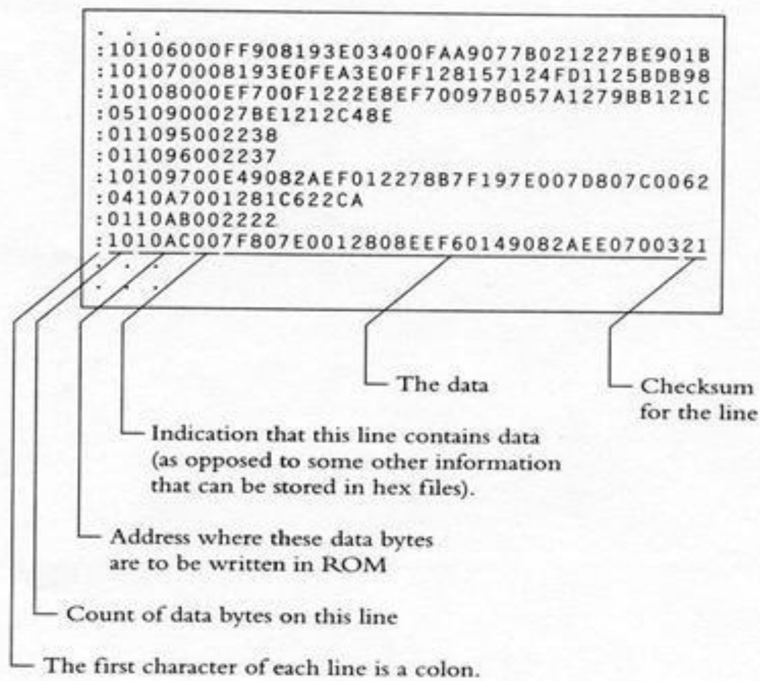
The tools you are using to load your program into target must understand whatever file format your locator produces.

1. intel Hex file format

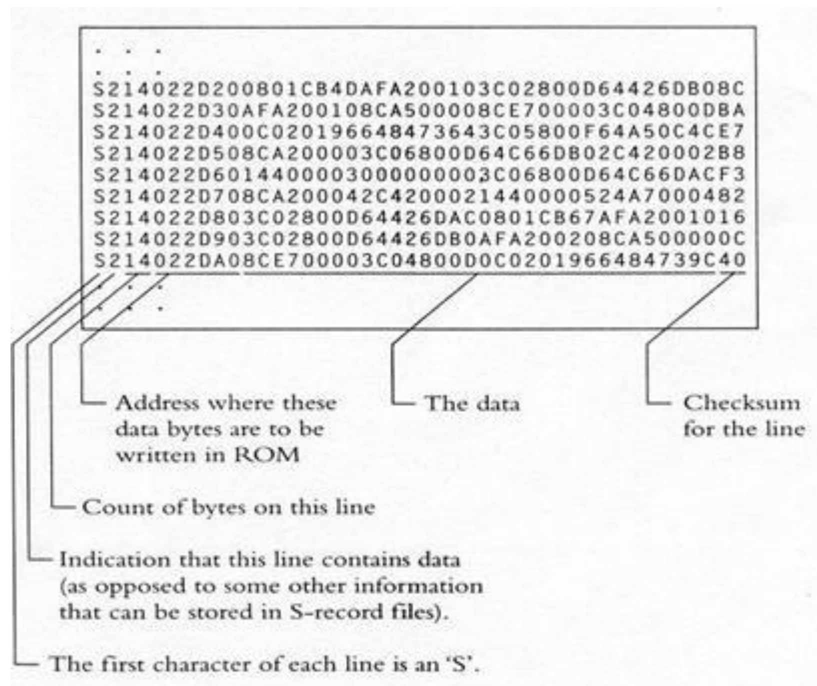
2. Motorola S-Record format

1. Intel Hex file format:

below figure shows Intel Hex file format



2. Motorola S-Record format



Loading program components properly:

Another issue that locators must resolve in the embedded environment is that some parts of the program need to end up in the ROM and some parts need to end up in RAM.

For example whosonfirst() end up in ROM and must be remembered even power is off. The variable idunno would have to be in RAM, since it data may be changed.

This issue does not arise with application programming, because the loader copies the entire program into RAM.

Most tools chains deal with this problem by **dividing the programs into segments**. Each **segment is a piece of program that the locator can place it in memory independently of other segments.**

Segments solve other problems like when processor power on, embedded system programmer must ensure where the first instruction is at particular place with the help of segments.

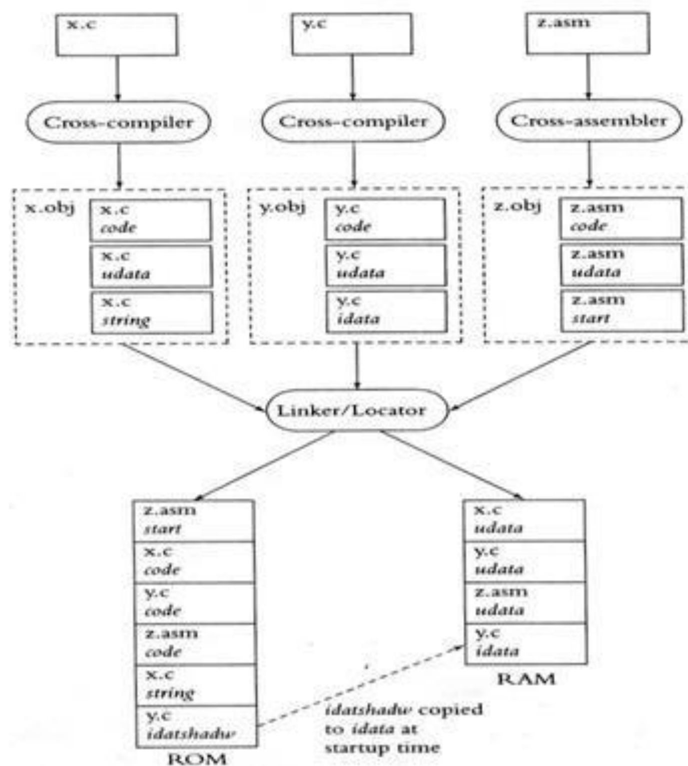


Figure: How the tool chain uses segments

Figure shows how a tool chain might work in a system in hypothetical system that contains three modules X.c, Y.c and Z.asm.The code X.c contains some instructions, some uninitialized data and some constant strings. The Y. c contains some instructions, some uninitialized and some initialized data. The Z.asm contains some assembly language function, start up code and uninitialized code

.The cross compiler will divide X.c into 3 segments in the object file

First segment: code

Second segment: udata

Third segment: constant strings

- The cross compiler will divide Y.c into 3 segments in the object file First segment: code Second segment: udata Third segment: idata
- The cross compiler Z.asm divides the segments into First Segment: assembly language functions
  Second Segment: start up code
  Third Segment t: udata

The linker/ Locator reshuffle these segments and places Z.asm start up code at where processor begins its execution, it places code segment in ROM and data segment in RAM. Most compilers automatically divide the module into two or more segments: The instructions (code), uninitialized code, Initialized, Constant strings. Cross assemblers also allow you to specify the segment or segments into which the output from the assembler should be placed. Locator places the segments in memory. The following two lines of instructions tells one commercial locator how to build the program.

- The –Z at the beginning of each line indicates that this line is a list of segments.

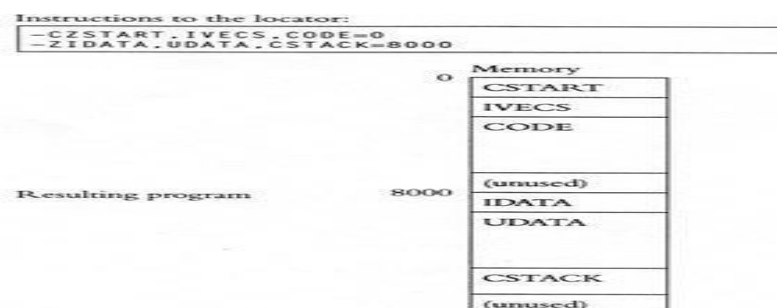


Fig 6: Locator places segments in memory

- At the end of each line is the address where the segment should be placed.
- The locator places the segments one after other in memory, starting with the given address.
- The segments CSTART, IVECS, CODE one after other must be placed at address 0.
- The segments IDATA, UDATA AND CTACK at address at 8000.

Some other features of locators are:

- We can specify the address ranges of RAM and ROM, the locator will warn you if program does not fit within those functions.
- We can specify the address at which the segment is to end, then it will place the segment below that address which is useful for stack memory.
- We can assign each segment into group, and then tell the locator where the group go and deal with individual segments.

Where the variable ifreq must be stored. In the above code, in the first case ifreq the initial value must reside in the ROM (this is the only memory that stores the data while the power is off).In the second case the ifreq must be in RAM, because setfreq () changes it frequently.

The only solution to the problem is to store the variable in RAM and store the initial value in ROM and copy the initial value into the variable at startup. Loader sees that each initialized variable has the correct initial value when it loads the program. But there is no loader in embedded system, so that the application must itself arrange for initial values to be copied into variables.

The locator deals with this is to create a shadow segment in ROM that contains all of the initial values, a segment that is copied to the real initialized - data segment at start up. When an embedded system is powdered on the contents of the RAM are garbage. They only become all zeros if some start up code in the embedded system sets them as zeros.

Locator Maps:

- Most **locators will create an output file, called map**, that lists where the locator placed each of the segments in memory.
- A **map** consists of **address of all public functions and global variables**.
- These are useful for debugging an 'advanced' locator is capable of running a startup code in ROM, which load the embedded code from ROM into RAM to execute quickly since RAM is faster

Locator MAP IS SHOWN BELOW:

```
LINK MAP OF MODULE:  XYZ

   TYPE     BASE      LENGTH    RELOCATION    SEGMENT NAME

-------------------------------------------------------

   * * * * * * *  X D A T A   M E M O R Y  * * * * * * *

            0000H     8100H                   *** GAP ***
   XDATA    8100H     0001H     UNIT          ?XD?PROGFLSH
   XDATA    8101H     000CH     UNIT          ?XD?VPROG?PROGFLSH
   XDATA    810DH     0006H     UNIT          ?XD?CHKSM?PROGFLSH
   XDATA    8113H     0080H     UNIT          ?C_LIB_XDATA
   XDATA    8193H     0002H     UNIT          ?XD?MAIN?PAD
   XDATA    8195H     0002H     UNIT          ?XD?RXCALLBACK?PAD
   :

   * * * * * * *   C O D E   M E M O R Y   * * * * * * *

            0000H     0017H                   *** GAP ***
   CODE     0080H     000FH     UNIT          PROGFLSTSTA
   CODE     008FH     0055H     UNIT          PROGFLSA
   CODE     00E4H     01ADH     UNIT          ?PR?VPROG?PROGFLSH
   CODE     0291H     0073H     UNIT          ?PR?SEND?PROGFLSH
   CODE     0304H     001DH     UNIT          ?PR?RX?PROGFLSH
   CODE     0321H     0072H     UNIT          ?PR?CHKSM?PROGFLSH
   CODE     0393H     007EH     INBLOCK       SCC_INIT
   CODE     0411H     082EH     UNIT          ?C_LIB_CODE
   :
```

Executing out of RAM:

RAM is faster than ROM and other kinds of memory like flash. The fast microprocessors (RISC) execute programs rapidly if the program is in RAM than ROM. But they store the programs in ROM, copy them in RAM when system starts up.

The start-up code runs directly from ROM slowly. It copies rest of the code in RAM for fast processing. The code is compressed before storing into the ROM and start up code decompresses when it copies to RAM.

The system will do all this things by locator, locator must build program can be stored at

one collection of address ROM and execute at other collection of addresses at RAM.

Getting embedded software into the target system:

- The locator will build a file as an image for the target software. There are few ways to getting the embedded software file into target system.
  - PROM programmers
  - ROM emulators
  - In circuit emulators
  - Flash
  - Monitors

PROM Programmers:

- The classic way to get the software from the locator output file into target system by creating file in ROM or PROM.

- Creating ROM is appropriate when software development has been completed, since cost to build ROMs is quite high. Putting the program into PROM requires a device called PROM programmer device.

- PROM is appropriate if software is small enough, if you plan to make changes to the software and debug. To do this, place PROM in socket on the Target than being soldered directly in the circuit (the following figure shows). When we find bug, you can remove the PROM containing the software with the bug from target and put it into the eraser (if it is an erasable PROM) or into the waste basket. Otherwise program a new PROM with software which is bug fixed and free, and put that PROM in the socket. We need small tool called chip puller (inexpensive) to remove PROM from the socket. We can insert the PROM into socket without any tool than thumb (see figure8). If PROM programmer and the locator are from different vendors, its upto us to make them compatible.

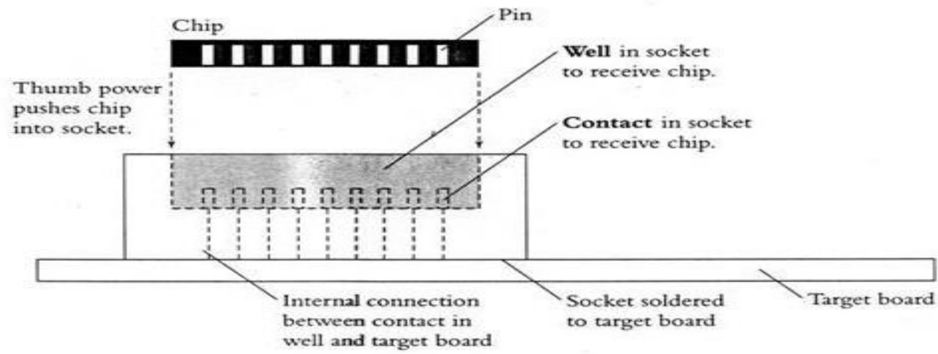


Fig : Semantic edge view of socket

ROM Emulators:

Other mechanism is ROM emulator which is used to get software into target. ROM emulator is a device that replaces the ROM into target system. It just looks like ROM, as shown figure9; ROM emulator consists of large box of electronics and a serial port or a network connection through which it can be connected to your host. Software running on your host can send files created by the locator to the ROM emulator. Ensure the ROM emulator understands the file format which the locator creates.

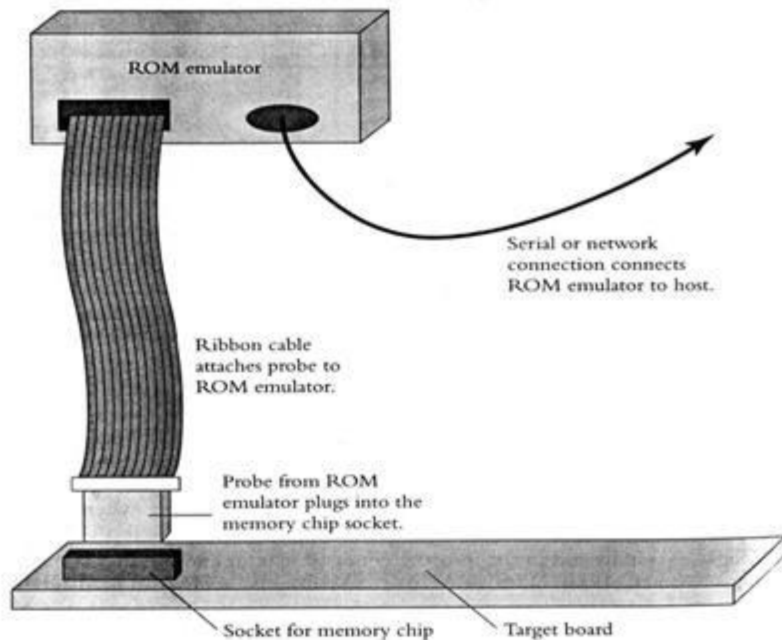


Fig: ROM emulator

In circuit emulators:

If we want to debug the software, then we can use overlay memory which is a common feature of in-circuit emulators. In-circuit emulator is a mechanism to get software into target for debugging purposes.

Flash:

If your target stores its program in flash memory, then one option you always have is to place flash memory in socket and treat it like an EPROM .However, If target has a serial port, a network connection, or some other mechanism for communicating with the outside world, link then target can communicate with outside world, flash memories open up another possibility: you can write a piece of software to receive new programs from your host across the communication link and write them into the flash memory. Although this may seem like difficult

The reasons for new programs from host:

- You can load new software into your system for debugging, without pulling chip out of socket and replacing.
- Downloading new software is fast process than taking out of socket, programming and returning into the socket.
- If customers want to load new versions of the software onto your product.

The following are some issues with this approach:

- Here microprocessor cannot fetch the instructions from flash.
- The flash programming software must copy itself into the RAM, locator has to take care all these activities how those flash memory instructions are executing.
- We must arrange a foolproof way for the system to get flash programming software into the target i.e target system must be able to download properly even if earlier download crashes in the middle.
- To modify the flash programming software, we need to do this in RAM and then copy to flash.

Monitors:

It is a program that resides in target ROM and knows how to load new programs onto the system. A typical monitor allows you to send the data across a serial port, stores the software in the target RAM, and then runs it. Sometimes monitors will act as locator also,

offers few debugging services like setting break points, display memory and register values. You can write your own monitor program.

DEBUGGING TECHNIQUES

I. Testing on host machine
II. using laboratory tools
III. an example system

Introduction:

While developing the embedded system software, the developer will develop the code with the lots of bugs in it. The testing and quality assurance process may reduce the number of bugs by some factor. But only the way to ship the product with fewer bugs is to write software with few fewer bugs. The world extremely intolerant of buggy embedded systems. The testing and debugging will play a very important role in embedded system software development process.

Testing on host machine :

- **Goals of Testing process are**
  - **Find bugs early in the development process**
  - **Exercise all of the code**
  - **Develop repeatable , reusable tests**
  - **Leave an audit trail of test results**

Find the bugs early in the development process:

This saves time and money. Early testing gives an idea of how many bugs you have and then how much trouble you are in.

BUT: the target system is available early in the process, or the hardware may be buggy and unstable, because hardware engineers are still working on it.

Exercise all of the code:

Exercise all exceptional cases, even though, we hope that they will never happen, exercise them and get experience how it works.

BUT: It is impossible to exercise all the code in the target. For example, a laser printer may have code to deal with the situation that arise when the user presses the one of the buttons just as a paper jams, but in the real time to test this case. We have to make paper to jam and then press the button within a millisecond, this is not very easy to do.

Develop reusable, repeatable tests:

It is frustrating to see the bug once but not able to find it. To make refuse to happen again, we need to repeatable tests.

BUT: It is difficult to create repeatable tests at target environment.

Example: In bar code scanner, while scanning it will show the pervious scan results every time, the bug will be difficult to find and fix.

Leave an "Audit trail" of test result:

Like telegraph "seems to work" in the network environment as it what it sends and receives is not easy as knowing, but valuable of storing what it is sending and receiving.

BUT: It is difficult to keep track of what results we got always, because embedded systems do not have a disk drive.

Conclusion: Don't test on the target, because it is difficult to achieve the goals by testing software on target system. The alternative is to test your code on the host system.

Basic Technique to Test:

The following figure shows the basic method for testing the embedded software on the development host. The left hand side of the figure shows the target system and the right hand side shows how the test will be conducted on the host. The hardware independent code on the two sides of the figure is compiled from the same source.

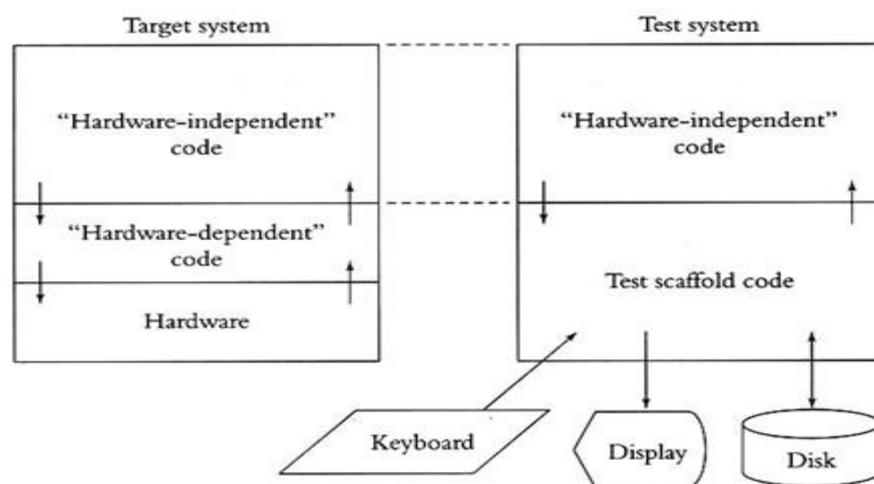


Figure: Test System

The hardware and hardware dependent code has been replaced with test scaffold software on the right side. The scaffold software provides the same entry points as does the hardware dependent code on the target system, and it calls the same functions in the hardware independent code. The scaffold software takes its instructions from the keyboard or from a file; it produces output onto the display or into the log file.

Conclusion: Using this technique you can design clean interface between hardware independent software and rest of the code.

Calling Interrupt Routines by scaffold code:

Based on the occurrence of interrupts tasks will be executed. Therefore, to make the system do anything in the test environment, the test scaffold must execute the interrupt routines. Interrupts have two parts one which deals with hardware (by hardware dependent interrupt calls) and other deals rest of the system (hardware independent interrupt calls).

Calling the timer interrupt routine:

One interrupt routine your test scaffold should call is the timer interrupt routine. In most embedded systems initiated the passage of time and timer interrupt at least for some of the activity. You could have the passage of time in your host system call the timer interrupt routine automatically. So time goes by your test system without the test scaffold software participation. It causes your test scaffold to lose control of the timer interrupt routine. So your test scaffold must call Timer interrupt routine directly.

Script files and Output files:

A test scaffold that calls the various interrupt routines in a certain sequence and with certain data. A test scaffold that reads a script from the keyboard or from a file and then makes calls as directed by the script. Script file may not be a project, but must be simple one.

Example: script file to test the bar code scanner

```
#frame arrives
# Dst                         Src    Ctrl mr/56 ab
#Backoff timeout expires Kt0
#timeout expires again Kt0
#sometime pass Kn2
Kn2
```

#Another beacon frame arrives

Each command in this script file causes the test scaffold to call one of the interrupts in the hardware independent part.

In response to the kt0 command the test scaffold calls one of the timer interrupt routines. In response to the command kn followed by number, the test scaffold calls a different timer interrupt routine the

indicated number of times. In response to the command mr causes the test scaffold to write the data into memory.

Features of script files:

- The commands are simple two or three letter commands and we could write the parser more quickly.
- Comments are allowed, comments script file indicate what is being tested, indicate what results you expect, and gives version control information etc.
- Data can be entered in ASCII or in Hexadecimal.

Most advanced Techniques:

These are few additional techniques for testing on the host. **It is useful to have the test scaffold software do something automatically**. For example, when the hardware independent code for the underground tank monitoring system sends a line of data to the printer, the test scaffold software must capture the line, and it must call the printer interrupt routine to tell the hardware independent code that the printer is ready for the next line.

There may be a need that test scaffold a switch control because there may be button interrupt routine, so that the test scaffold must be able to delay printer interrupt routine. **There may be low, medium, high priority hardware independent requests, then scaffold switches as they appear**. Some Numerical examples of test scaffold software: In Cordless bar code scanner, when H/W independent code sends a frame the scaffold S/W calls the interrupt routine to indicate that the frame has been sent. When H/W independent code sets the timer, then test scaffold code call the timer interrupt after some period. The scaffold software acts as communication medium, which contains multiple instances of H/W independent code with respect to multiple systems in the project.

Bar code scanner Example:

Here the scaffold software generate an interrupts when ever frame send and receive. Bar

code Scanner A send data frame, captures by test scaffold and calls frame sent interrupt. The test scaffold software calls receive frame interrupt when it receives frame. When any one of the H/W independent code calls the function to control radio, the scaffold knows which instances have turned their radios, and at what frequencies.

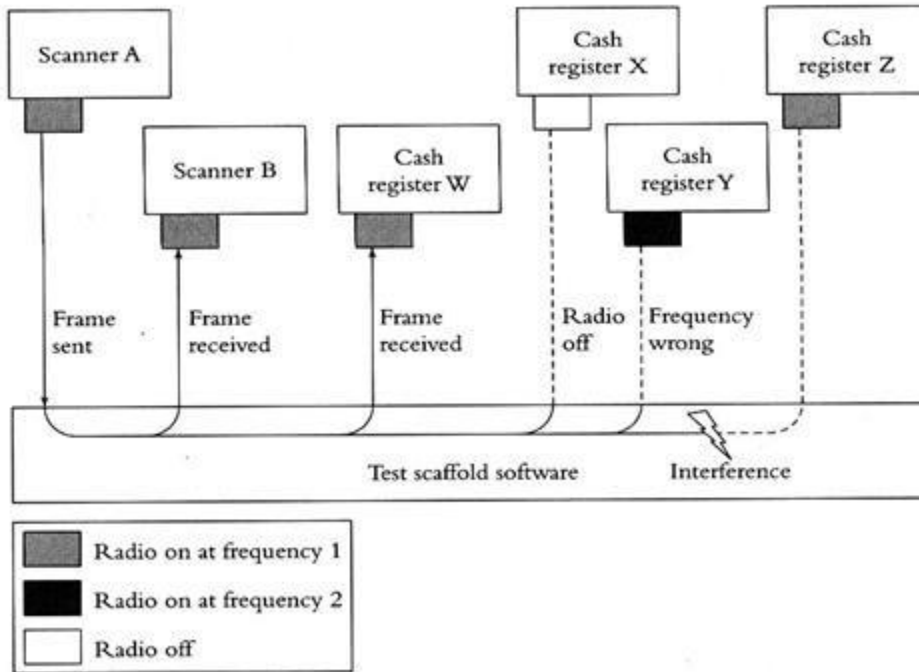


Fig2: Test scaffold for the bar- code scanner software

Targets that have their radios turned off and tuned to different frequencies do not receive the frame.

The scaffold simulates the interference that prevents one or more stations from receiving the data. In this way the scaffold tests various pieces of software communication properly with each other or not.(see the above figure).

OBJECTIONS, LIMITATIONS AND SHORT COMINGS:

Engineers raise many objections to testing embedded system code on their host system, Because many embedded systems are hardware dependent. Most of the code which is tested at host side is hardware dependent code.

To test at host side embedded systems interacts only with the microprocessor, has no direct contact with the hardware. As an example the Telegraph software huge percentage of software is hardware independent i.e. this can be tested on the host with an appropriate scaffold. There are few objections to scaffold: Building a scaffold is more trouble, making

compatible to RTOS is other tedious job.

Using laboratory Tools:

- Volt meters and Ohm Meters
- Oscilloscopes
- Logic Analyzers
- Logic Analyzers in Timing mode
- Logic Analyzers in State Mode
- In-circuit Emulators
- Getting " Visibility" into the Hardware
- Software only Monitors
- Other Monitors

Volt meters:

Volt meter is for measuring the voltage difference between two points. The common use of voltmeter is to determine whether or not chip in the circuit have power. A system can suffer power failure for any number of reasons- broken leads, incorrect wiring, etc. the usual way to use a volt meter It is used to turn on the power, put one of the meter probes on a pin that should be attached to the VCC and the other pin that should be attached to ground. If volt meter does not indicate the correct voltage then we have hardware problem to fix.

Ohm Meters:

Ohm meter is used for measuring the resistance between two points, the most common use of Ohm meter is to check whether the two things are connected or not. If one of the address signals from microprocessors is not connected to the RAM, turn the circuit off, and then put the two probes on the two points to be tested, if ohm meter reads out 0 ohms, it means that there is no resistance between two probes and that the two points on the circuit are therefore connected. The product commonly known as Multimeter functions as both volt and Ohm meters.

Oscilloscopes:

It is a device that graphs voltage versus time, time and voltage are graphed horizontal and vertical axis respectively. It is analog device which signals exact voltage but not low or high.

Features of Oscilloscope:

- You can monitor one or two signals simultaneously.
- You can adjust time and voltage scales fairly wide range.
- You can adjust the vertical level on the oscilloscope screen corresponds to ground. With the use of trigger, oscilloscope starts graphing. For example we can tell the oscilloscope to start graphing when signal reaches 4.25 volts and is rising.

Oscilloscopes extremely useful for Hardware engineers, but software engineers use them for the following purposes:

1. Oscilloscope used as volt meter, if the voltage on a signal never changes, it will display horizontal line whose location on the screen tells the voltage of the signal.
2. If the line on the Oscilloscope display is flat, then no clocking signal is in Microprocessor and it is not executing any instructions.
3. Use Oscilloscope to see as if the signal is changing as expected.
4. We can observe a digital signal which transition from VCC to ground and vice versa shows there is hardware bug.

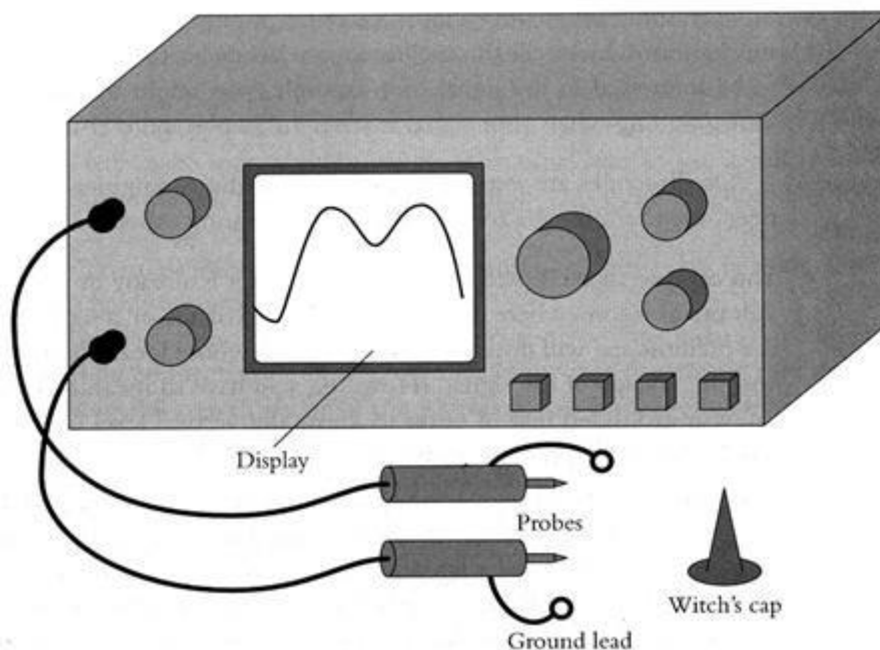
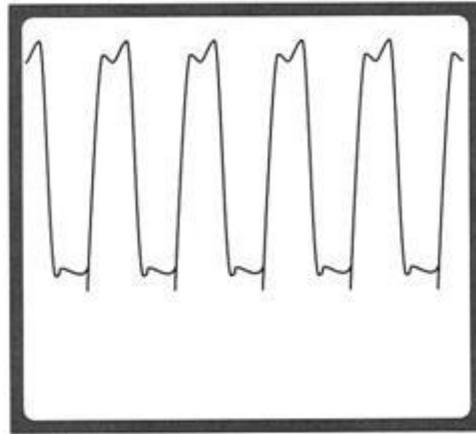


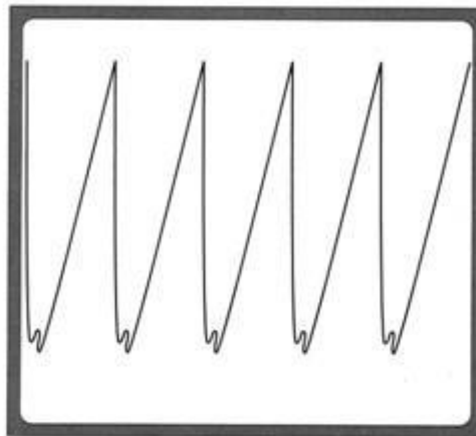
Fig3: Typical Oscilloscope

Figure3 is a sketch of a typical oscilloscope, consists of probes used to connect the oscilloscope to the circuit. The probes usually have sharp metal ends holds against the signal on the circuit. Witch's caps fit over the metal points and contain little clip that hold the probe in the circuit. Each probe has ground lead a short wire that extends from the head of

the probe, it can easily attach to the circuit. It is having numerous adjustment knobs and buttons allow you to control. Some may have on screen menus and set of function buttons along the side of the screen.

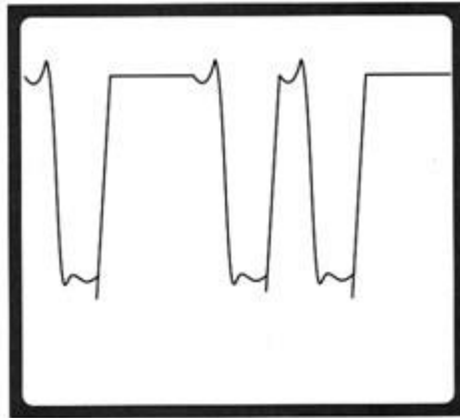


4(a): A Reasonable clock signal



4(b): A Questionable clock signal

4 (c): A dead clock signal



4(d): A ROM chip selection signal

Figure4 (a) to 4(d) shows some typical oscilloscope displays.fig (a) shows a microprocessor input clock signal. Fig (b) shows a questionable clock signal, it differs from 4(a) in that it does not go from lo to high cleanly and stay high for a period of time. Instead it draft from low to high .fig(c) shows a clock circuit that is not working at all.fig(d) shows chip enable signal.

Logic Analyzers:

This tool is similar to oscilloscope, which captures signals and graphs them on its screen. But it differs with oscilloscope in several fundamental ways

- A logic analyzer track many signals simultaneously.
- The logic analyzer only knows 2 voltages, VCC and Ground. If the voltage is in between VCC and ground, then logical analyzer will report it as VCC or Ground but not like exact voltage.
- All logic analyzers are storage devices. They capture signals first and display them later.
- Logic analyzers have much more complex triggering techniques than oscilloscopes.
- Logical analyzers will operate in state mode as well as timing mode.

Logical analyzers in Timing Mode:

Some situations where logical analyzers are working in Timing mode

- If certain events ever occur.
- Example: In bar code scanner software ever turns the radio on, we can attach logic analyzer to the signals that controls the power to the radio.
- We can measure how long it takes for software to respond.
- We can see software puts out appropriate signal patterns to control the hardware. The

underground tank monitoring system to find out how long it will takes the software to turn off the bell when you push a button shown in fig5.

Example: After finishing the data transmitting, we can attach the logical analyzer to RTS and its signal to find out if software lowers RTS at right time or early or late. We can also attach the logical analyzer, to ENABLE/ CLK and DATA signals to EEPROM to find if it works correctly or not.(see fig6).

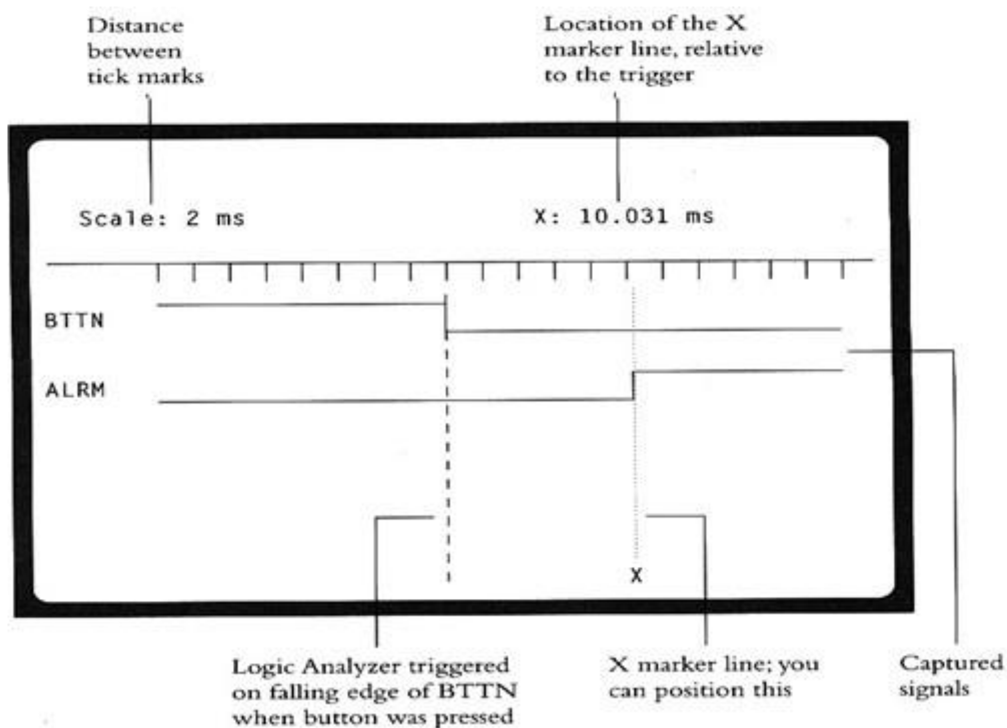


Fig5 : Logic analyzer timing display: Button and Alarm signal

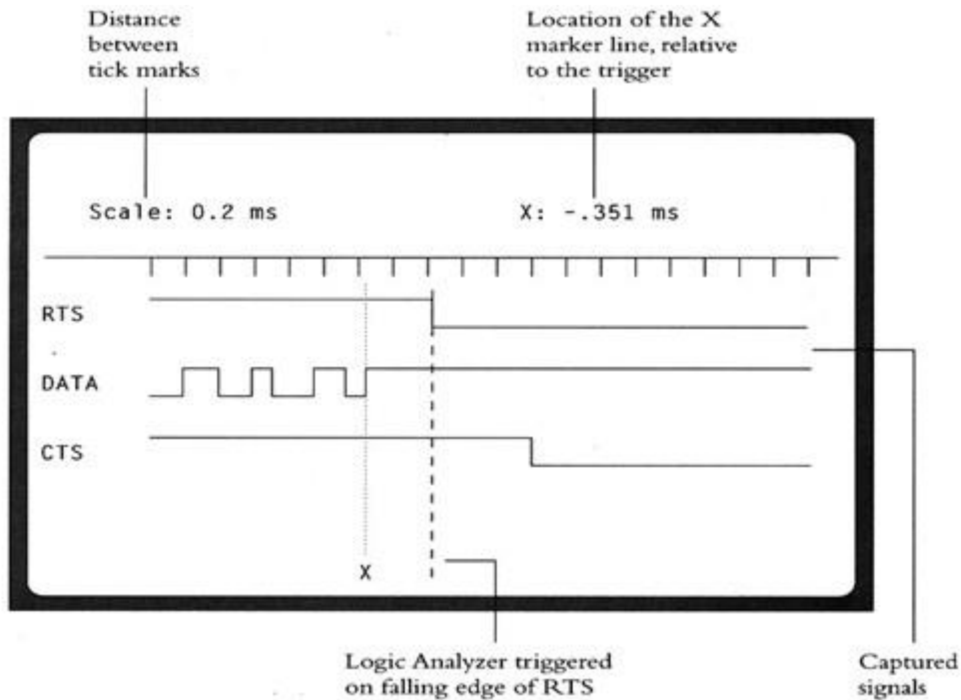


Fig6 : Logic Analyzer timing Display: Data and RTS signal

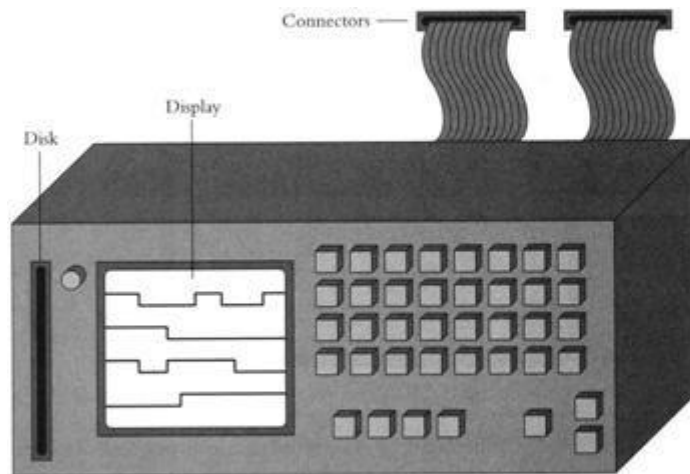
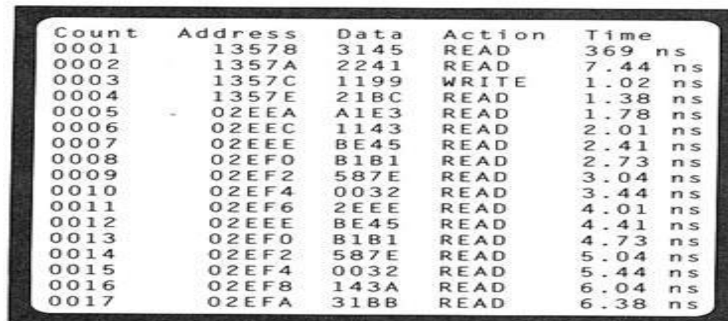


Fig7 : Logic analyzer

Figure7 shows a typical logic analyzer. They have display screens similar to those of oscilloscopes. Most logic analyzers present menus on the screen and give you a keyboard to enter choices, some may have mouse as well as network connections to control from work stations. Logical analyzers include hard disks and diskettes. It can be attached to many signals through ribbons. Since logic analyzer can attach to many signals simultaneously, one or more ribbon cables typically attach to the analyzer.

Logical Analyzer in State Mode:

In the timing mode, logical analyzer is self clocked. That is, it captures data without reference to any events on the circuit. In state mode, they capture data when some particular event occur, called a clock occurs in the system. In this mode the logical analyzer see what instructions the microprocessor fetched and what data it read from and write to its memory and I/O devices. To see what instructions the microprocessor fetched, you connect logical analyzer probes to address and data signals of the system and RE signal on the ROM. Whenever RE signal raise then logical analyzer capture the address and data signals. The captured data is called as trace. The data is valid when RE signal raise. State mode analyzers present a text display as state of signals in row as shown in the below figure.



Count	Address	Data	Action	Time
0001	13578	3145	READ	369 ns
0002	1357A	2241	READ	7.44 ns
0003	1357C	1199	WRITE	1.02 ns
0004	1357E	21BC	READ	1.38 ns
0005	02EEA	A1E3	READ	1.78 ns
0006	02EEC	1143	READ	2.01 ns
0007	02EEE	BE45	READ	2.41 ns
0008	02EF0	B1B1	READ	2.73 ns
0009	02EF2	587E	READ	3.04 ns
0010	02EF4	0032	READ	3.44 ns
0011	02EF6	2EEE	READ	4.01 ns
0012	02EEE	BE45	READ	4.41 ns
0013	02EF0	B1B1	READ	4.73 ns
0014	02EF2	587E	READ	5.04 ns
0015	02EF4	0032	READ	5.44 ns
0016	02EF8	143A	READ	6.04 ns
0017	02EFA	31BB	READ	6.38 ns

Fig8 : Typical logic analyzer state mode display

The logical analyzer in state mode extremely useful for the software engineer,

1. Trigger the logical analyzer, if processor never fetch if there is no memory.
2. Trigger the logical analyzer, if processor writes an invalid value to a particular address in RAM.
3. Trigger the logical analyzer, if processor fetches the first instruction of ISR and executed.
4. If we have bug that rarely happens, leave processor and analyzer running overnight and check results in the morning.
5. There is filter to limit what is captured.

Logical analyzers have short comings:

Even though analyzers tell what processor did, we cannot stop, break the processor, even if it did wrong. By the analyzer the processors registers are invisible only we know the contents of memory in which the processors can read or write. If program crashes, we cannot examine anything in the system. We cannot find if the processor executes out of cache. Even if the

program crashes, still emulator let make us see the contents of memory and registers. Most emulators capture the trace like analyzers in the state mode. Many emulators have a feature called overlay memory, one or more blocks of memory inside the emulator, emulated microprocessor can use instead of target machine.

In circuit emulators:

In-circuit emulators also called as emulator or ICE replaces the processor in target system. Ice appears as processor and connects all the signals and drives. It can perform debugging, set break points after break point is hit we can examine the contents of memory, registers, see the source code, resume the execution. Emulators are extremely useful, it is having the power of debugging, acts as logical analyzer. Advantages of logical analyzers over emulators:

- Logical analyzers will have better trace filters, more sophisticated triggering mechanisms.
- Logic analyzers will also run in timing mode.
- Logic analyzers will work with any microprocessor.
- With the logic analyzers you can hook up as many as or few connections as you like. With the emulator you must connect all of the signal.
- Emulators are more invasive than logic analyzers.

Software only Monitors:

One widely available debugging tool often called as Monitor .monitors allow you to run software on the actual target, giving the debugging interface to that of In circuit emulator. Monitors typically work as follows:

- One part of the monitor is a small program resides in ROM on the target, this knows how to receive software on serial port, across network, copy into the RAM and run on it. Other names for monitor are target agent, monitor, debugging kernel and so on.
- Another part the monitor run on host side, communicates with debugging kernel, provides debugging interface through serial port communication network.
- You write your modules and compile or assemble them.
- The program on the host cooperates with debugging kernel to download compiled module into the target system RAM. Instruct the monitor to set break points, run the system and so on.
- You can then instruct the monitor to set breakpoints.

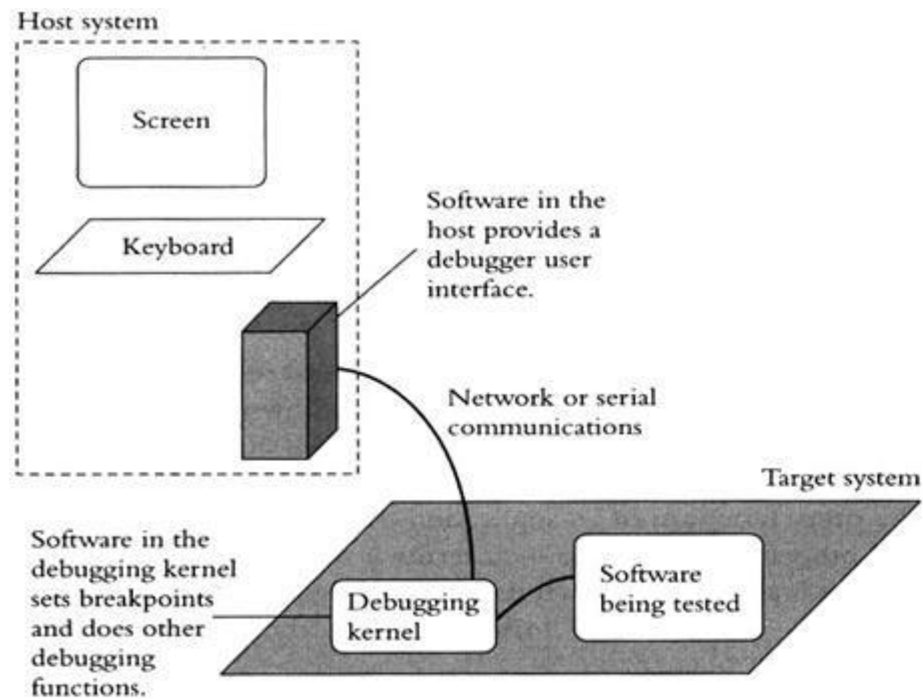


Fig 9: software only the monitor

See the above figure, Monitors are extraordinarily valuable, gives debugging interface without any modifications.

Disadvantages of Monitors:

- The target hardware must have communication port to communicate the debugging kernel with host program. We need to write the communication hardware driver to get the monitor working.
- At some point we have to remove the debugging kernel from your target system and try to run the software without it.
- Most of the monitors are incapable of capturing the traces like of those logic analyzers and emulators.
- Once a breakpoint is hit, stop the execution can disrupt the real time operations so badly.

Other Monitors:

The other two mechanisms are used to construct the monitors, but they differ with normal monitor in how they interact with the target. The first target interface is with through a ROM emulator. This will do the downing programs at target side, allows the host program to set break points and other various debugging techniques.

UNIT V

INTRODUCTION TO ADVANCED PROCESSORS

SYLLABUS:

Introduction to advanced architectures: ARM and SHARC, processor and memory organization and instruction level parallelism; Networked embedded systems: Bus protocols, I2C bus and CAN bus; Internet-Enabled systems, design example-Elevator controller.

Unit V contents at a glance:

I. Introduction to advanced architectures
II. ARM ,
III. SHARC,
IV. processor and memory organization and instruction level parallelism;

Networked embedded systems:

I. bus protocols,
II. I2C bus and CAN bus;
III. internet-enabled systems,
IV. design example-elevator controller.

I. INTRODUCTION TO ADVANCED ARCHITECTURES:

Two Computing architectures are available:

1. von Neumann architecture computer
2. Harvard architecture

von Neumann architecture computer:

- The memory holds both data and instructions, and can be read or written when given an address. A computer whose memory holds both data and instructions is known as a von Neumann machine
- 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 CPU fetches the instruction from memory, decodes the instruction, and executes it.
- The program counter does not directly determine what the machine does next, but only indirectly by pointing to an instruction in memory.

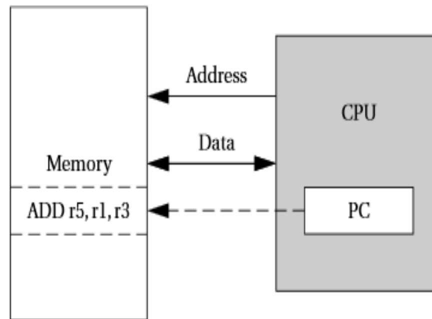


FIGURE 2.1

A von Neumann architecture computer.

2. Harvard architecture:

- Harvard machine has separate memories for data and program.
- The program counter points to program memory, not data memory.
- As a result, it is harder to write self-modifying programs (programs that write data values, then use
  Those values as instructions) on Harvard machines.

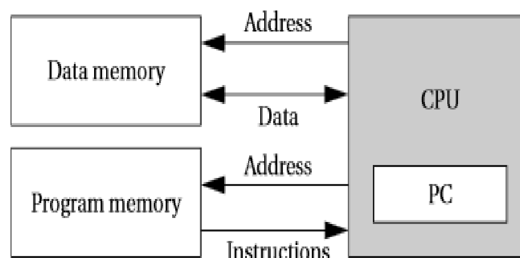


FIGURE 2.2

A Harvard architecture.

Advantage:

- The separation of program and data memories provides higher performance for digital signal processing.

Differences between Von neumann and harvard architecture:

VON NEUMANN	HARVARD ARCHITECTURE
Same memory holds data, instructions	Separate memories for data and instructions
A single set of address/data buses between CPU and memory	Two sets of address/data buses between CPU and memory
Single memory fetch operation	Harvard allows two simultaneous memory fetches
The code is executed serially and takes more clock cycles	The code is executed in parallel
Not exactly suitable for DSP	Most DSPs use Harvard architecture for

	streaming data: • greater memory bandwidth; • more predictable bandwidth
There is no exclusive Multiplier	It has MAC (Multiply Accumulate)
No Barrel Shifter is there	Barrel Shifter help in shifting and rotating operations of the data
The programs can be optimized in lesser size	The program tend to grow big in size
Used in conventional processors found in PCs and Servers, and embedded systems with only control functions.	Used in DSPs and other processors found in latest embedded systems and Mobile communication systems, audio, speech, image processing systems

RISC and CISC Processors:

RISC	CISC
RISC stands for Reduced Instruction Set Computer	CISC stands for Complex Instruction Set Computer
Hardware plays major role in CISC processors	Software plays major role in CISC processors
RISC processors use single clock to execute an instruction	CISC processors use multiple clocks for execution.
Memory-to-memory access is used for data manipulations is RISC processors	intermediate registers are used for data manipulation
In RISC processors, single word instructions are given as inputs	In CISC processors, instructions of variable lengths are given as input, based upon the task to be performed
More lines of code and large memory footprint	High code density
Compact, uniform instructions and hence facilitate pipelining	Many addressing modes and long instructions
Allow effective compiler optimization	Often require manual optimization of assembly code for embedded systems
These machines provided a variety of instructions that may perform very complex tasks, such as string searching	These computers tended to provide somewhat fewer and simpler instructions.

II. ARM(Advanced RISC Machine) Processor:

- **ARM** uses **RISC** architecture
- **ARM** uses **assembly language** for writing programs
- ARM instructions are written one per line, starting after the first column.
- Comments begin with a semicolon and continue to the end of the line.
- A label, which gives a name to a memory location, comes at the beginning of the line, starting in the first column.

Here is an example:

```
        LDR r0,[r8]; a comment
label   ADD r4,r0,r1
```

Memory Organization in ARM Processor:

The ARM architecture supports two basic types of data:

- The standard ARM word is 32 bits long.
- The word may be divided into four 8-bit byte
- ARM allows addresses up to 32 bits long
- 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**

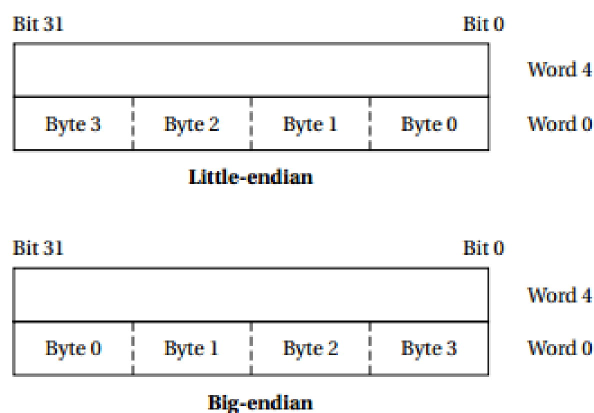


FIGURE 2.6

Byte organizations within an ARM word.

Data Operations in ARM:

- In the ARM processor, arithmetic and logical operations cannot be performed directly on memory locations.

- ARM is a **load-store architecture**—data operands must first be loaded into the CPU and then stored back to main memory to save the results

ARM Programming Model:

1. Programming model gives information about various registers supported by ARM
2. ARM has **16 general-purpose registers, r0 to r15**
3. Except for r15, they are identical—any operation that can be done on one of them can be done on the other one also
4. **r15** register is also used as **program counter(PC)**
5. **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.

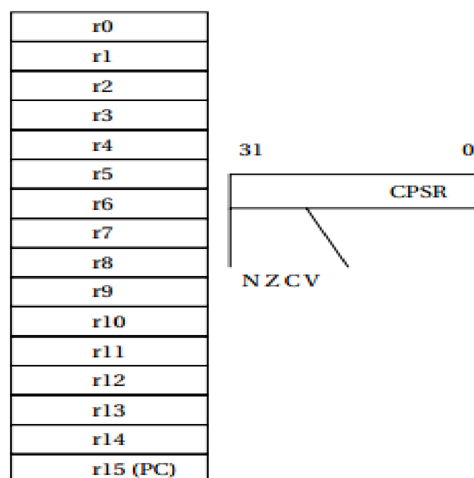


FIGURE 2.8
The basic ARM programming model.

Types of Instructions supported by ARM Processor:
1. Arithmetic Instructions
2. Logical Instructions
3. shift / rotate Instructions

4. Comparison Instructions
5. move instructions
6. Load store instructions

ADD	Add
ADC	Add with carry
SUB	Subtract
SBC	Subtract with carry
RSB	Reverse subtract
RSC	Reverse subtract with carry
MUL	Multiply
MLA	Multiply and accumulate

Arithmetic

AND	Bit-wise and
ORR	Bit-wise or
EOR	Bit-wise exclusive-or
BIC	Bit clear

Logical

LSL	Logical shift left (zero fill)
LSR	Logical shift right (zero fill)
ASL	Arithmetic shift left
ASR	Arithmetic shift right
ROR	Rotate right
RRX	Rotate right extended with C

Shift/rotate

CMP	Compare
CMN	Negated compare
TST	Bit-wise test
TEQ	Bit-wise negated test

FIGURE 2.10
ARM comparison instructions.

MOV	Move
MVN	Move negated

FIGURE 2.11
ARM move instructions.

LDR	Load
STR	Store
LDRH	Load half-word
STRH	Store half-word
LDRSH	Load half-word signed
LDRB	Load byte
STRB	Store byte
ADR	Set register to address

FIGURE 2.12
ARM load-store instructions and pseudo-operations.

Instructions examples:

ADD r0,r1,r2
This instruction sets register r0 to the sum of the values stored in r1 and r2.
ADD r0,r1,#2 (immediate operand are allowed during addition)
RSB r0, r1, r2 sets r0 to be r2-r1.
bit clear: BIC r0, r1, r2 sets r0 to r1 and not r2.
Multiplication:

no immediate operand is allowed in multiplication
two source operands must be different registers

MLA: The MLA instruction performs a multiply-accumulate operation, particularly useful in matrix operations and signal processing

MLA r0,r1,r2,r3 sets r0 to the value **r1x r2+r3.**

Shift operations:

Logical shift(LSL, LSR)
Arithmetic shifts (ASL, ASR)

- A **left shift** moves bits up toward the most-significant bits,
- **right shift** moves bits down to the least-significant bit in the word.
- The LSL and LSR modifiers perform left and right logical shifts, filling the least-significant bits of the operand with zeroes.
- The **arithmetic shift left** is equivalent to an LSL, but the ASR copies the sign bit—if the sign is 0, a 0 is copied, while if the sign is 1, a 1 is copied.

Rotate operations: (ROR, RRX)

- The rotate modifiers always rotate right, moving the bits that fall off the least-significant bit up to the most-significant bit in the word.
- The RRX modifier performs a 33-bit rotate, with the CPSR's C bit being inserted above the sign bit of the word; this allows the carry bit to be included in the rotation

Compare instructions: (CMP, CMN)

- **compare instruction modifies flags values (Negative flag, zero flag, carry flag, Overflow flag)**
- **CMP r0, r1 computes r0 – r1**, sets the status bits, and throws away the result of the subtraction.
- CMN uses an addition to set the status bits.
- TST performs a bit-wise AND on the operands,
- while TEQ performs an exclusive-or

Load store instructions:

- ARM uses register-indirect addressing
- The value stored in the register is used as the address to be fetched from memory; the result of that fetch is the desired operand value.
- LDR r0,[r1] sets r0 to the value of memory location 0x100.
- Similarly, STR r0,[r1] would store the contents of r0 in the memory location whose address is given in r1

LDR r0,[r1, – r2]

ARM Register indirect addressing:

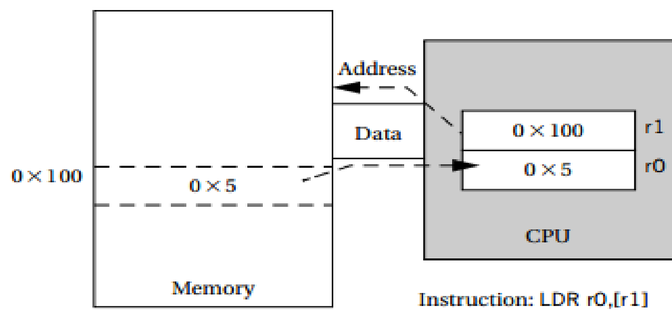


FIGURE 2.13

Register-indirect addressing in the ARM.

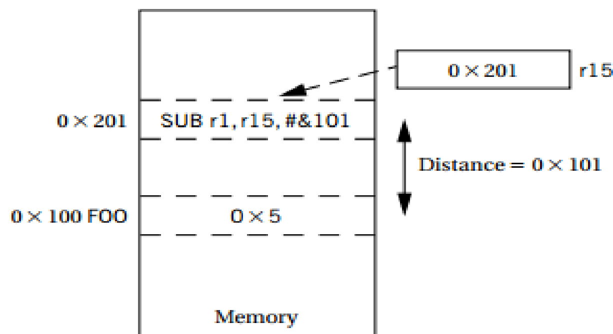


FIGURE 2.14

Computing an absolute address using the PC.

LDR r0,[r1, #4] loads r0 from the address r1+ 4.

Sample programs using ARM instruction set:

Expression: x = (a+b)-c

program:

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
```

Expression:

y=a*(b+c)

can be coded similarly, but in this case we will reuse more registers by using r0 for both *a* and *b*, r1 for *c*, and r2 for *y*. Once again, we will use r4 to store addresses for indirect addressing. The resulting code is

```
ADR r4,b      ; get address for b
LDR r0,[r4]   ; get value of b
ADR r4,c      ; get address for c
LDR r1,[r4]   ; get value of c
ADD r2,r0,r1 ; compute partial result of y
ADR r4,a      ; get address for a
```

```
LDR r0,[r4]  ; get value of a
MUL r2,r2,r0 ; compute final value of y
ADR r4,y     ; get address for y
STR r2,[r4]  ; store value of y at proper location
```

program 3:

The C statement

$$z = (a << 2) \mid (b \& 15);$$

can be coded using r0 for *a* and *z*, r1 for *b*, and r4 for addresses as follows:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
MOV r0,r0,LSL 2   ; perform shift
ADR r4,b          ; get address for b
LDR r1,[r4]       ; get value of b
AND r1,r1,#15     ; perform logical AND
ORR r1,r0,r1      ; compute final value of z
ADR r4,z          ; get address for z
STR r1,[r4]       ; store value of z
```

ARM Base plus offset addressing mode:

the register value is added to another value to form the address.

For instance, **LDR r0,[r1,#16]** loads r0 with the value stored at location r1+16.(r1-base address, 16 is offset)

Base plus offset addressing mode → auto indexing

Base plus offset addressing mode → post indexing

Auto-indexing updates the base register, such that LDR r0,[r1,#16]!   ----first adds 16 to the value of r1, and then uses that new value as the address. The ! operator causes the base register to be updated with the computed address so that it can be used again later.

Post-indexing does not perform the offset calculation until after the fetch has been performed. Consequently,
LDR r0,[r1],#16 will load r0 with the value stored at the memory location whose address is given by r1, and then add 16 to r1 and set r1 to the new value.

FLOW OF CONTROL INSTRUCTIONS (Branch Instructions):

Branch Instructions

1. **conditional instructions(BGE-- B is branch, GE is condition)**
2. **unconditional instructions(B)**

the following **branch instruction B #100 will add 400 to the current PC value**

EQ	Equals zero	Z = 1
NE	Not equal to zero	Z = 0
CS	Carry set	C = 1
CC	Carry clear	C = 0
MI	Minus	N = 1
PL	Nonnegative (plus)	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 1 and Z = 0
LS	Unsigned lower or same	C = 0 or Z = 1
GE	Signed greater than or equal	N = V
LT	Signed less than	N V
GT	Signed greater than	Z = 0 and N = V
LE	Signed less than or equal	Z = 1 or N V

FIGURE 2.15

Condition codes in ARM.

example for flow of control programs:

Implementing an* `if` *statement in ARM

We will use the following if statement as an example:

```
if (a < b) {
      x = 5;
      y = c + d;
      }
else x = c - d;
```

The implementation uses two blocks of code, one for the true case and another for the false case. A branch may either fall through to the true case or branch to the false case:

```
; compute and test the condition
       ADR r4,a         ; get address for a
       LDR r0,[r4]      ; get value of a
       ADR r4,b         ; get address for b
       LDR r1,[r4]      ; get value of b
       CMP r0, r1       ; compare a < b
       BGE fblock       ; if a >= b, take branch
; the true block follows
       MOV r0,#5        ; generate value for x
       ADR r4,x         ; get address for x
       STR r0,[r4]      ; store value of x
       ADR r4,c         ; get address for c
       LDR r0,[r4]      ; get value of c
       ADR r4,d         ; get address for d
       LDR r1,[r4]      ; get value of d
       ADD r0,r0,r1     ; compute c + d
       ADR r4,y         ; get address for y
       STR r0,[r4]      ; store value of y
       B after          ; branch around the false block
; the false block follows
fblock ADR r4,c         ; get address for c
       LDR r0,[r4]      ; get value of c
       ADR r4,d         ; get address for d
       LDR r1,[r4]      ; get value of d
       SUB r0,r0,r1     ; compute c - d
       ADR r4,x         ; get address for x
       STR r0,[r4]      ; store value of x
after ... ; code after the if statement
```

Branch and Link instruction (BL) for implementing functions or sub routines or procedures:

Procedure calls in ARM

We use as an example one of the functions from Figure 2.16:

```
void f1(int a) {
     f2(a);
}
```

The ARM C compiler's convention is to use register r13 to point to the top of the stack. We assume that the argument a has been passed into f1() on the stack and that we must push the argument for f2 (which happens to be the same value) onto the stack before calling f2().

Here is some handwritten code for f1(), which includes a call to f2():

```
f1 LDR r0,[r13]    ; load value of a argument into r0 from stack
   ; call f2()
   STR r14,[r13]!       ; store f1's return address on the stack
   STR r0,[r13!]        ; store argument to f2 onto stack
   BL f2                ; branch and link to f2
   ; return from f1()
   SUB r13,#4           ; pop f2's argument off the stack
   LDR r13!,r15         ; restore registers and return
```

We use base-plus-offset addressing to load the value passed into f1() into a register for use by r1. To call f2(), we first push f1()'s return address, stored in r14 by the branch-and-link instruction executed to get into f1(), onto the stack. We then push f2()'s parameter onto the stack. In both cases, we use autoincrement addressing to both store onto the stack and adjust the stack pointer. To return, we must first adjust the stack to get rid of f2()'s parameter that

*****Note : for more programs, refer class notes.***

III . SHARC Processor:

Features of SHARC processor:

1.SHARC stands for **Super Harvard Architecture Computer**

2.The ADSP-21060 SHARC chip is made by Analog Devices, Inc.

3.It is a **32-bit signal processor** made mainly for **sound, speech, graphics, and imaging applications**.

4.It is a high-end digital signal processor designed with **RISC techniques**.

5. Number formats:

i. **32-bit Fixed Format**

Fractional/Integer

Unsigned/Signed

ii. **Floating Point**

32-bit single-precision IEEE floating-point data format

40-bit version of the IEEE floating-point data format.

16-bit shortened version of the IEEE floating-point data format.

6. **32 Bit floating point, with 40 bit extended floating point capabilities**.

7. **Large on-chip memory**.

8. Ideal for scalable multi-processing applications.

9. Program memory can store data.

10. Able to simultaneously read or write data at one location and get instructions from another place in memory.

11. 2 buses

Data memory bus.

Program bus.

12. Either two separate memories or a single dual-port memory

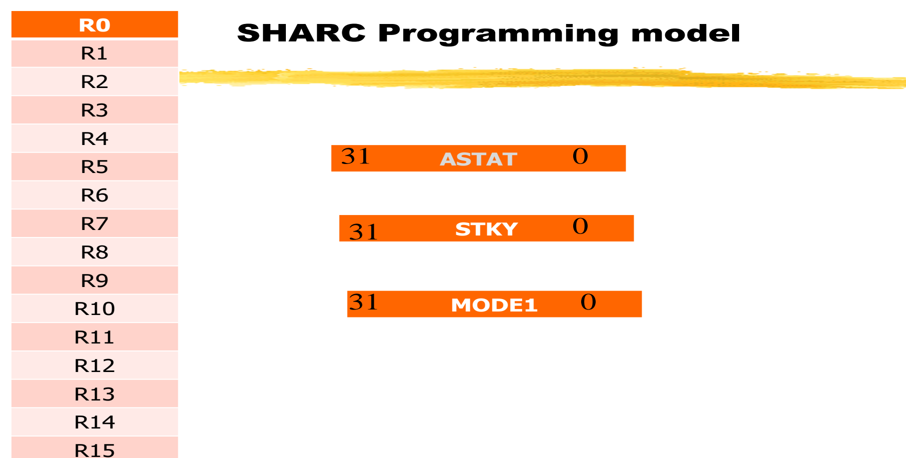
13. The SHARC incorporates features aimed at optimizing such loops.

14. High-Speed Floating Point Capability
15. Extended Floating Point
16. The **SHARC supports floating, extended-floating and non-floating point**.
17. No additional clock cycles for floating point computations.
18. Data automatically truncated and zero padded when moved between 32-bit memory and internal registers.

SHARC PROCESSOR PROGRAMMING MODEL:

Programming model gives the registers details. The following registers are used in SHARC processors for various purposes:

- Register files: R0-R15 (aliased as F0-F15 for floating point)
- Status registers.
- Loop registers.
- Data address generator registers(DAG1 and DAG2)
- Interrupt registers.
- 16 primary registers (R0-R15)
- 16 alternate registers (F0-F15)
- each register can hold 40 bits
- R0 – R15 are for Fixed-Point Numbers
- F0 – F15 are for Floating-Point Numbers



Status registers:

ASTAT: arithmetic status.
STKY: sticky.
MODE 1: mode 1.

- The **STKY register** is a sticky version of **ASTAT register**, the STKY bits are set along with ASTAT register bits but not cleared until cleared by an instruction.
- The SHARC perform saturation arithmetic on fixed point values, saturation mode is controlled by ALUSAT bit in MODE1 register.
- All ALU operations set AZ (zero), AN (negative), AV (overflow), AC (fixed-point carry), AI (floating-point invalid) bits in ASTAT.

Data Address Generators(DAG)

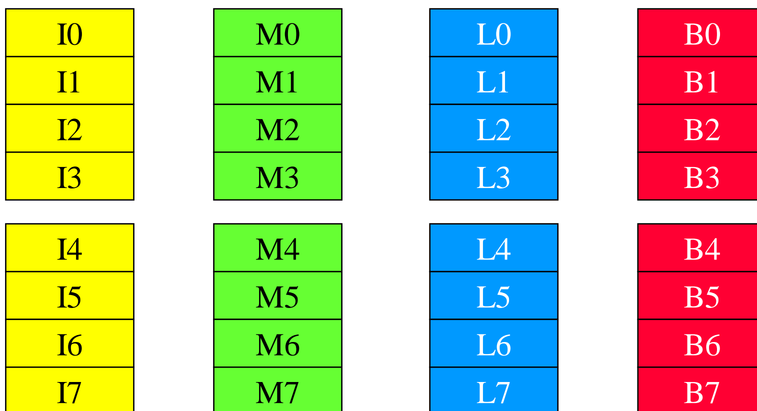
Data Address Generators

- There are two data address generators (DAG1 & DAG2) for addressing memory indirectly (with pre-modify or post-modify).
- Data address generator 1 (DAG1) generates 32-bit addresses on the Data Memory Address Bus.
- Data address generator 2 (DAG2) generates 24-bit addresses on the Program Memory Address Bus.
- Each DAG has four types of registers:
  - The Index (I) register acts as a pointer to memory.
  - The Modify (M) register contains the increment value for advancing the pointer.
  - Base and Limit Registers (More on the next page).

- Two data address generators (DAGs):

  program memory and data memory.

DAG1 registers



DAG2 registers

I	M	L	B
I8	M8	L8	B8
I9	M9	L9	B9
I10	M10	L10	B10
I11	M11	L11	B11
I12	M12	L12	B12
I13	M13	L13	B13
I14	M14	L14	B14
I15	M15	L15	B15

Multifunction computations or instruction level parallel processing:

Can issue some computations in parallel:

- dual add-subtract;
- fixed-point multiply/accumulate and add, subtract, average
- floating-point multiply and ALU operation
- multiplication and dual add/subtract

Pipelining in SHARC processor:

Instructions are processed in three cycles:

- Fetch instruction from memory
- Decode the opcode and operand
- Execute the instruction

- SHARC supports delayed and non-delayed branches
- Specified by bit in branch instruction
- 2 instruction branch delay slot
- Six Nested Levels of Looping in Hardware

Bus Architecture:

Twin Bus Architecture:

- 1 bus for Fetching Instructions
- 1 bus for Fetching Data

Improves multiprocessing by allowing more steps to occur during each clock

Addressing modes provided by DAG in SHARC Processor:

1. The Simplest addressing mode
2. Absolute address
3. post modify with update mode
4. base-plus-offset mode
5. Circular Buffers
6. Bit reversal addressing mode

1. The **Simplest addressing mode** provides an immediate value that can represent the address.

Example : R0=DM(0X200000)

R0=DM(_a) i.e load R0 with the contents of the variable a

2. An **Absolute address** has entire address in the instruction, space inefficient, address occupies the more space.

3. A **post modify with update mode** allows the program to sweep through a range of address. This uses I register and modifier, I registers shows the address value and modifier (M register value or Immediate value) is update the value.

For load
R0=DM(I3,M1)
For store : DM(I3,M1)=R0

4. The **base-plus-offset mode** here the address computed as I+M where I is the base and M modifier or offset.

Example: R0=DM(M1, I0)

I0=0x2000000 and M0= 4 then the value for R0 is loaded from 0x2000004

5. **Circular Buffers** is an array of n elements is n+1th element is referenced then the location is 0. It is wrapping around from end to beginning of the buffer.

This mode uses L and B registers, L registers is set with +ve and nonzero value at staring point, B register is stored with same value as the I register is store with base address.

If I register is used in post modify mode, the incremental value is compared to the sum of L and B registers, if end of the buffer is reached then I register is wrapped around.

6. **Bit reversal addressing mode** : this is used in Fast Fourier Transform (FFT). Bit reversal can be performed only in I0 and I8 and controlled by BR0 and BR8 bits in the MODE1 register.

SHARC allows two fetches per cycle.

F0=DM(M0,I0); FROM DATA MEMORY

F1=PM(M8,I8); FROM PROGRAM MEMORY

BASIC addressing:

Immediate value:

R0 = DM(0x20000000);

Direct load:

R0 = DM(_a); ! Loads contents of _a

Direct store:

DM(_a)= R0; ! Stores R0 at _a

SHARC programs examples:

expression:

x = (a + b) - c;

program:

```
R0 = DM(_a) ! Load a

R1 = DM(_b); ! Load b

R3 = R0 + R1;

R2 = DM(_c); ! Load c

R3 = R3-R2;

DM(_x) = R3; ! Store result in x
```

expression :

y = a*(b+c);

program:

```
R1 = DM(_b) ! Load b

R2 = DM(_c); ! Load c

R2 = R1 + R2;

R0 = DM(_a); ! Load a

R2 = R2*R0;

DM(_y) = R23; ! Store result in y
```

note: for programs , refer class notes

SHARC jump:

Unconditional flow of control change:

JUMP foo

Three addressing modes:

direct;

indirect;

PC-relative.

ARM vs. SHARC

• ARM7 is von Neumann architecture

• ARM9 is Harvard architecture

• SHARC is modified Harvard architecture. – On chip memory (> 1Gbit) evenly split between program memory (PM) and data memory (DM) – Program memory can be used to store some data. – Allows data to be fetched from both memory in parallel

The SHARC ALU operations:

1. Fixed point ALU operations
2. Floating point ALU operations
3. Shifter operations in SHARC

Fixed point ALU operations

Rn=Rx+Ry	Add
Rn=Rx- Ry	Subtract
Rn= Rx+Ry+CI	Add with carry
Rn=Rx-Ry+CI-1	Subtract with borrow
Rn=(Rx+Ry)/2	Average
COMP(Rx,Ry)	Compare
Rx=Rx+CI	Add carry
Rn=Rx+CI-1	Add borrow
Rn=Rx+1	Increment
Rn=Rx-1	Decrement
Rn=-Rx	Negate

Rn= ABS Rx	Absolute Value
Rn=PASS Rx	Copy Rx to Rn
Rn=Rx AND Ry	Logical AND
Rn=Rx OR Ry	Logical OR
Rn=Rx XOR Ry	Logical exclusive OR
Rn= NOT Rx	Logical Negate
Rn= MIN (Rx, Ry)	Minimum of Rx, Ry
Rn=MAX (Rx, Ry)	Maximum of Rx, Ry
Rn= CLIP Rx by Ry	Clip Rx within range [-Ry, Ry]

Floating point ALU operations:

Fn=Fx +Fy	Add
Fn=Fx - Fy	Subtract
Fn=ABS (Fx+Fy)	Absolute value of sum
Fn=ABS (Fx-Fy)	Absolute value of difference
Fn=(Fx+Fy)/2	Average
COMP(Fx,Fy)	Compare
Fn=-Fx	Negate
Fn=ABS Fx	Absolute value
Fn=PASS Fx	Copy Fx to Fn
Fn =RND Fx	Round
Fn= SCALB Fx by Ry	Scale exponent of Fx by Ry

Rn = MANT Fx	Extract mantissa of Fx
Rn= LOGB Fx	Convert exponent of Fx into Integer
Rn=FIX Fx, Rn= TRUNC Fx	Convert floating point to integer
Fn= FLOAT Rx by Ry, FLOAT Rx	Convert Integer to Floating Point
Fn= RECIPS Fx	Create seed of Reciprocal
Fn=RSQRTS Fx	Create seed for reciprocal square root
Fn=Fx COPYSIGN Fy	Copy sign of Fy to Fx
Fn=MIN(Fx, Fy)	Minimum of Fx, Fy
Fn=MAX (Fx, Fy)	Maximum of Fx, Fy
Fn= CLIP Fx by Fy	Clip Fx within RANGE [-Fy,Fy]

SHARC Shift operations

Rn= LSHIFT Rx by Ry	Logical shift distance Ry
Rn= Rn OR LSHIFT Rx by Ry	Logical Shift and logical OR
Rn=ASHIFT Rx by Ry	Arithmetic shift
Rn= Rn OR ASHIFT Rx by Ry	Arithmetic shift and logical OR
Rn= ROT Rx by Ry	Rotate distance Ry
Rn= BCLR Rx by Ry	Clear one bit in Rx
Rn=BSET Rx by Ry	Set one bit in Rx
Rn=BTGL Rx by Ry	Toggle one bit in Rx
BTST Rx by Ry	Text one bit in Rx
Rn=FDEP Rx by Ry	Deposit field from Rx into Rn
Rn=Rn OR FDEP Rx by Ry	Deposit field from Rx using OR
Rn=FDEP Rx by Ry	Deposit and sign extend field from Rx
Rn=Rn OR FDEP Rx by Ry	Deposit and sign extend using OR

SHARC Shift operations

Rn=FEXT Rx by Ry	Extract field from Rx
Rn=FEXT Rx by Ry	Extract and sign extend field from Rx
Rn = EXP Rx	Extract exponent field from Rx
Rn = EXP Rx (EX)	Extract exponent field from ALU
Rn = LEFTZ Rx	Extract number of leading zeros
Rn = LEFTO Rx	Extract number of leading ones
Rn = FPACK Rx	Convert 32 bit floating point to 16 bit floating point
Rn = FUNPACK RN	Convert 16 bit floating point to 32 bit floating point

UNIT V - part II

Network Embedded System

Contents:

I. bus protocols,
II. I2 C bus ,
III. CAN bus;
IV. internet enabled systems,
V. design example elevator controller.

I. BUS PROTOCOLS:

For serial data communication between different peripherals components , the following standards are used :

- VME
- PCI
- ISA etc

For distributing embedded applications, the following interconnection network protocols are there:

- I^2C
- CAN etc

I^2C :

- The I 2 C bus is a well-known bus commonly used to link microcontrollers into systems
- I 2C is designed to be low cost, easy to implement, and of moderate speed up to 100 KB/s for the standard bus and up to 400 KB/s for the extended bus
- it uses only two lines: the serial data line (SDL) for data and the serial clock line (SCL), which indicates when valid data are on the data line

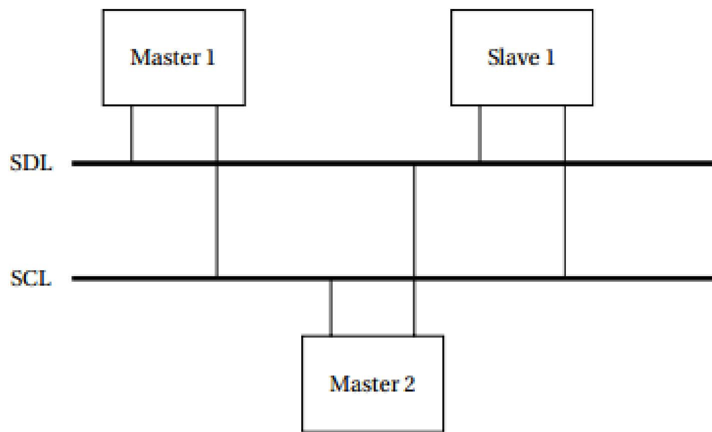


FIGURE 8.7

Structure of an I^2C bus system.

The basic **electrical interface of I2C** to the bus is shown in Figure

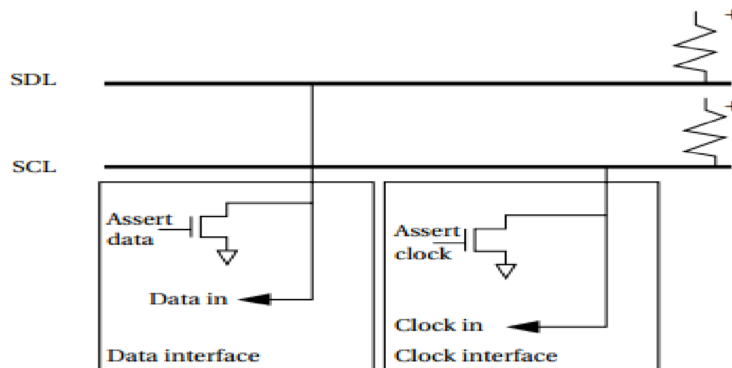


FIGURE 8.8

Electrical interface to the I^2C bus.

- A **pull-up resistor** keeps the default state of the signal high, and transistors are used in each bus device to pull down the signal when a 0 is to be transmitted.
- **Open collector/open drain** signaling allows several devices to simultaneously write the bus without causing electrical damage.
- The open collector/open drain circuitry allows a slave device to stretch a clock signal during a read from a slave.
- The master is responsible for generating the SCL clock, but the slave can stretch the low period of the clock
- The **I2C** bus is designed as a **multimaster bus**—any one of several different devices may act as the master at various times.
- As a result, there is no global master to generate the clock signal on SCL. Instead, a master drives both SCL and SDL when it is sending data. When the bus is idle, both SCL and SDL remain high.

- When **two devices try to drive either SCL or SDL to different values**, the **open collector/ open drain circuitry prevents errors**

Address of devices:

- A device address is 7 bits in the standard I2C definition (the extended I2C allows 10-bit addresses).
- The address 0000000 is used to signal a general call or bus broadcast, which can be used to signal all devices simultaneously**.** A bus transaction comprised a series of 1-byte transmissions and an address followed by one or more data bytes.

data-push programming :

- I2C encourages a data-push programming style. When a master wants to write a slave, it transmits the slave's address followed by the data.
- Since a slave cannot initiate a transfer, the master must send a read request with the slave's address and let the slave transmit the data.
- Therefore, an address transmission includes the 7-bit address and 1 bit for data direction: 0 for writing from the master to the slave and 1 for reading from the slave to the master

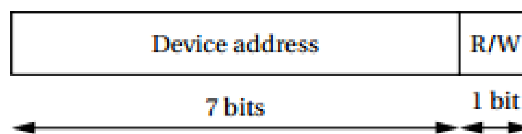


FIGURE 8.9

Format of an I^2C address transmission.

Bus transaction or transmission process:

1) start signal (SCL high and sending 1 to 0 in SDL)

2) followed by device address of 7 bits

3) RW(read / write bit) set to either 0 or 1

4) after address, now the data will be sent

5) after transmitting the complete data, the transmission stops.

The below figure is showing write and read bus transaction:

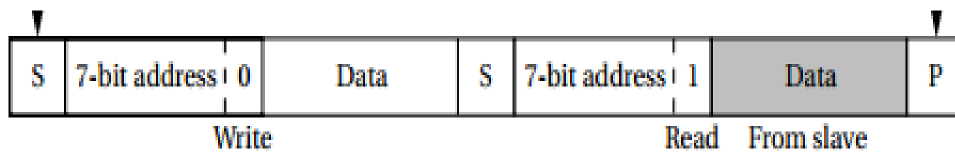


FIGURE 8.11

Typical bus transactions on the I^2C bus.

State transition graph:

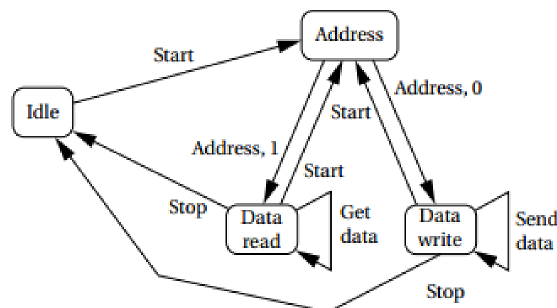


FIGURE 8.10

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

Transmitting byte in I2C Bus (Timing Diagram):

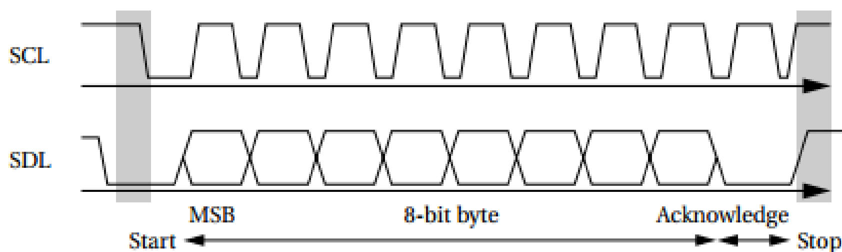


FIGURE 8.12

Transmitting a byte on the I^2C bus.

1. initially, SCL will be high, SDL will be low.

2. data byte will be transmitted.

3. after transmitting every 8 bits, an Acknowledgement will come

4. then stop signal is issued by setting both SCL and SDL high.

I2C interface on a microcontroller:

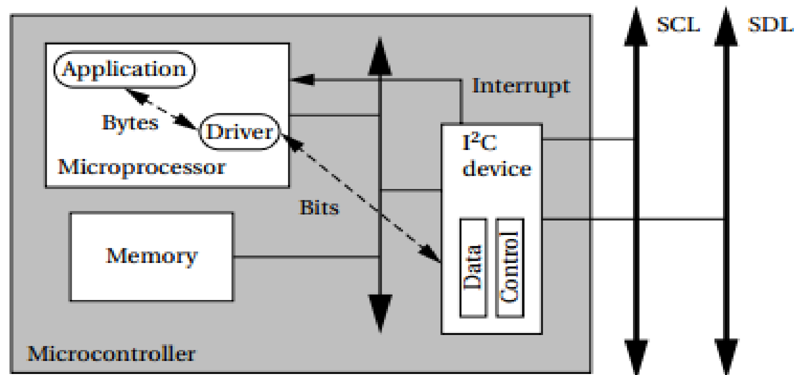


FIGURE 8.13

An I²C interface in a microcontroller.

Controlled Area Network:

The CAN bus was designed for automotive electronics and was first used in production cars in 1991.

The CAN bus 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.

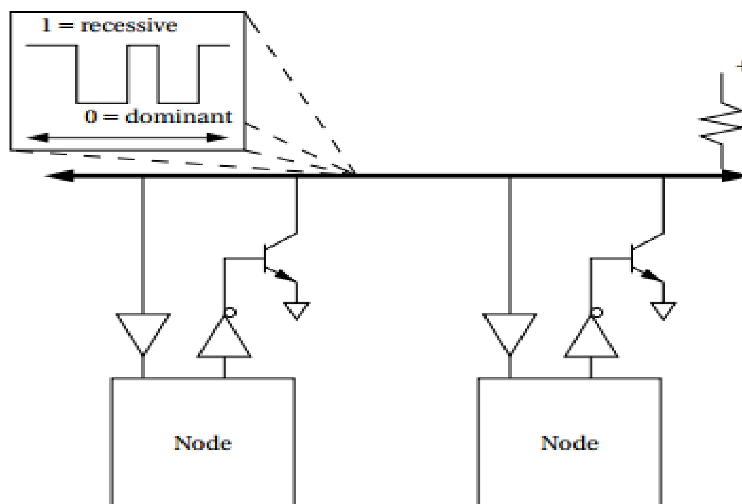


FIGURE 8.22

Physical and electrical organization of a CAN bus.

The above figure shows CAN electrical interface:

- 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.**
- The driving circuits on the bus cause the bus to be pulled down to 0 if any node on the bus pulls the bus down (making 0 dominant over 1).
- 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 DATA FRAME:

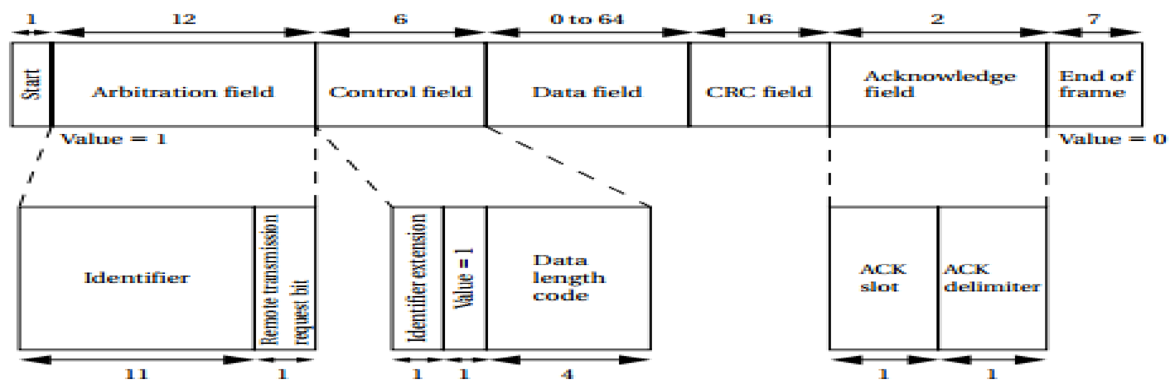


FIGURE 8.23
The CAN data frame format.

Explanation for data frame :

- A data frame starts with a 1 and ends with a string of seven zeroes. (There are at least three bit fields between data frames.)
- 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 RTR 1, 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.

Architecture of CAN controller:

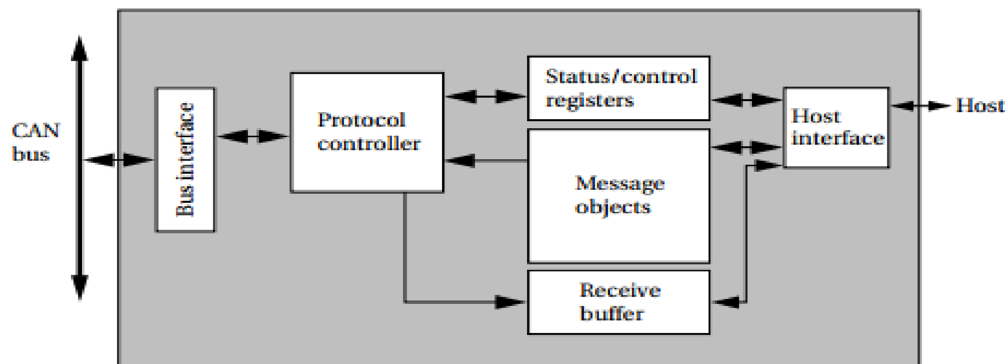


FIGURE 8.24

Architecture of a CAN controller.

- 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.

INTERNET ENABLED SYSTEMS:

IP Protocol:

- The Internet Protocol (IP) is the fundamental protocol on the Internet.
- It provides connectionless, packet-based communication.
- it is an internetworking standard.
- an Internet packet will travel over several different networks from source to destination.
- The IP allows data to flow seamlessly through these networks from one end user to another
- **Figure 8.19 explanation:**
- IP works at the network layer.

- When node A wants to send data to node B, the application's data pass through several layers of the protocol stack to send to the IP.
- IP creates packets for routing to the destination, which are then sent to the data link and physical layers.
- A node that transmits data among different types of networks is known as a router.

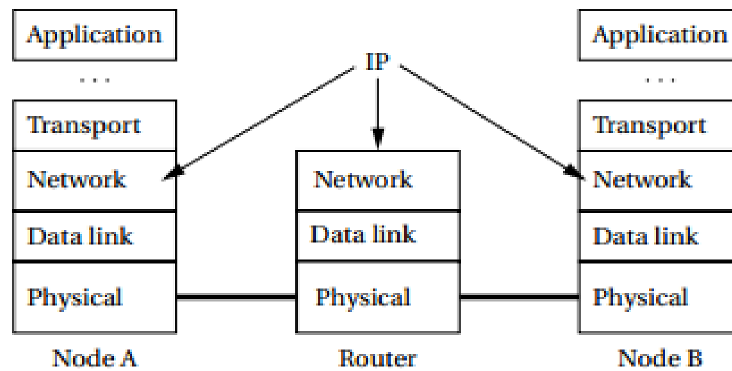


FIGURE 8.19
Protocol utilization in Internet communication.

IP Packet Format:

- The header and data payload are both of variable length.
- The maximum total length of the header and data payload is 65,535 bytes.
- An Internet address is a number (32 bits in early versions of IP, 128 bits in IPv6). The IP address is typically written in the form xxx.xx.xx.xx.
- 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.

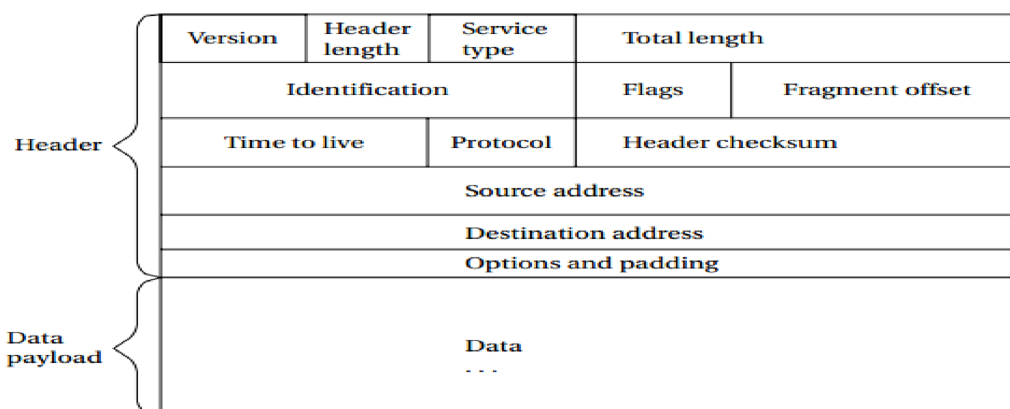


FIGURE 8.20
IP packet structure.

relationships between IP and higher-level Internet services:

Using IP as the foundation, TCP is used to provide File Transport Protocol for batch file transfers, Hypertext Transport Protocol (HTTP) for World Wide Web service, Simple Mail Transfer Protocol for email, and Telnet for virtual terminals. A separate transport protocol, User Datagram Protocol, is used as the basis for the network management services provided by the Simple Network Management Protocol

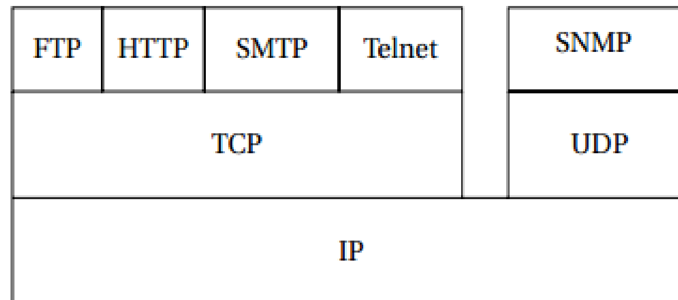


FIGURE 8.21

The Internet service stack.