LECTURE NOTES

ON

EMBEDDED SYSTEM DESIGN

B.Tech VI Semester (AUTONOMOUS)

(Regulation: IARE-R16)

(2018-2019)

Mr. N Nagaraju
Assistant professor

INFORMATION TECHNOLOGY

INSTITUTE OF AERONAUTICAL ENGINEERING
(AUTONOMOUS)
DUNDIGAL, HYDERABAD - 500043

UNIT-I
EMBEDDED COMPUTING

INTRODUCTION

This chapter introduces the reader to the world of embedded systems. Everything that we look around us today is electronic. The days are gone where almost everything was manual. Now even the food that we eat is cooked with the assistance of a microchip (oven) and the ease at which we wash our clothes is due to the washing machine. This world of electronic items is made up of embedded system. In this chapter we will understand the basics of embedded system right from its definition.

DEFINITION OF AN EMBEDDED SYSTEM

- An embedded system is a combination of 3 things:
  a. Hardware
  b. Software
  c. Mechanical Components

  And it is supposed to do one specific task only.

- **Example 1: Washing Machine**

  A washing machine from an embedded systems point of view has:

  a. Hardware: Buttons, Display & buzzer, electronic circuitry.
  b. Software: It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible.
  c. Mechanical Components: the internals of a washing machine which actually wash the clothes control the input and output of water, the chassis itself.

- **Example 2: Air Conditioner**

  An Air Conditioner from an embedded systems point of view has:

  a. Hardware: Remote, Display & buzzer, Infrared Sensors, electronic circuitry.
  b. Software: It has a chip on the circuit that holds the software which drives controls & monitors the various operations possible. The software monitors the external temperature through the sensors and then releases the coolant or suppresses it.
  c. Mechanical Components: the internals of an air conditioner the motor, the chassis, the outlet, etc

- An embedded system is designed to do a specific job only. Example: a washing machine can only wash clothes, an air conditioner can control the temperature in the room in which it is placed.

- The hardware & mechanical components will consist all the physically visible things that are used for input, output, etc.

- An embedded system will always have a chip (either microprocessor or microcontroller) that has the code or software which drives the system.

HISTORY OF EMBEDDED SYSTEM

- The first recognised embedded system is the Apollo Guidance Computer(AGC) developed by MIT lab.
- AGC was designed on 4K words of ROM & 256 words of RAM.
- The clock frequency of first microchip used in AGC was 1.024 MHz.
- The computing unit of AGC consists of 11 instructions and 16 bit word logic.
- It used 5000 ICs.
- The UI of AGC is known DSKY(display/keyboard) which resembles a calculator type keypad with array of numerals.
- The first mass-produced embedded system was guidance computer for the Minuteman-I missile in 1961.
- In the year 1971 Intel introduced the world's first microprocessor chip called the 4004, was designed for use in business calculators. It was produced by the Japanese company Busicom.

EMBEDDEDSYSTEM & GENERAL PURPOSE COMPUTER

The Embedded System and the General purpose computer are at two extremes. The embedded system is designed to perform a specific task whereas as per definition the general purpose computer is meant for general use. It can be used for playing games, watching movies, creating software, work on documents or spreadsheets etc.

Following are certain specific points of difference between embedded systems and general purpose computers:

Criteria	General Computer Purpose	Embedded system
Contents	It is combination of generic hardware and a general purpose OS for executing a variety of	It is combination of special purpose hardware and embedded OS for executing specific set of applications
Operating System	It contains general purpose operating system	It may or may not contain operating system.
Alterations	Applications are alterable by the user.	Applications are non-alterable by the user.

Key factor	Performance is key factor.	Application specific requirements are key factors.
Power Consumption	More	Less
Response Time	Not Critical	Critical for some applications

CLASSIFICATION OF EMBEDDED SYSTEM

The classification of embedded system is based on following criteria's:

- On generation
- On complexity & performance
- On deterministic behaviour
- On triggering

On generation

1. **First generation(1G):**
   - Built around 8bit microprocessor & microcontroller.
   - Simple in hardware circuit & firmware developed.
   - Examples: Digital telephone keypads.

2. **Second generation(2G):**
   - Built around 16-bit µp & 8-bit µc.
   - They are more complex & powerful than 1G µp & µc.
   - Examples: SCADA systems

3. **Third generation(3G):**
   - Built around 32-bit µp & 16-bit µc.
   - Concepts like Digital Signal Processors(DSPs), Application Specific Integrated Circuits(ASICs) evolved.
   - Examples: Robotics, Media, etc.

4. **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.

On complexity & performance

1. **Small-scale:**

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

- Example: an electronic toy

2. Medium-scale:

- 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.

3. Large-scale:

- 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.

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 Hard and Soft.

On triggering

- Embedded systems which are "Reactive" in nature can be based on triggering.
- Reactive systems can be:
  - ✓ Event triggered
  - ✓ Time triggered

APPLICATION OF EMBEDDED SYSTEM

The application areas and the products in the embedded domain are countless.

1. Consumer Electronics: Camcorders, Cameras.
2. Household appliances: Washing machine, Refrigerator.
3. Automotive industry: Anti-lock breaking system(ABS), engine control.
4. Home automation & security systems: Air conditioners, sprinklers, fire alarms.
5. Telecom: Cellular phones, telephone switches.
6. Computer peripherals: Printers, scanners.
7. Computer networking systems: Network routers and switches.
8. Healthcare: EEG, ECG machines.
9. Banking & Retail: Automatic teller machines, point of sales.
10. Card Readers: Barcode, smart card readers.

COMPLEX SYSTEMS AND MICROPROCESSORS

What is an ***embedded computer system***? Loosely defined, it is any device that includes a programmable computer but is not itself intended to be a general-purpose computer. Thus, a PC is not itself an embedded computing system, although PCs are often used to build embedded computing systems. But a fax machine or a clock built from a microprocessor is an embedded computing system. This means that embedded computing system design is a useful skill for many types of product design. Automobiles, cell phones, and even household appliances make extensive use of microprocessors. Designers in many fields must be able to identify where microprocessors can be used, design a hardware platform with I/O devices that can support the required tasks, and implement software that performs the required processing. Computer engineering, like mechanical design or thermodynamics, is a fundamental discipline that can be applied in many different domains. But of course, embedded computing system design does not stand alone. Many of the challenges encountered in the design of an embedded computing system are not computer engineering—for example, they may be mechanical or analog electrical problems. In this book we are primarily interested in the embedded computer itself, so we will concentrate on the hardware and software that enable the desired functions in the final product.

Embedding Computers

Computers have been embedded into applications since the earliest days of computing. One example is the Whirlwind, a computer designed at MIT in the late 1940s and early 1950s. Whirlwind was also the first computer designed to support ***real-time*** operation and was originally conceived as a mechanism for controlling an aircraft simulator. Even though it was extremely large physically compared to today's computers (e.g., it contained over 4,000 vacuum tubes), its complete design from components to system was attuned to the needs of real-time embedded computing. The utility of computers in replacing mechanical or human controllers was evident from the very beginning of the computer era—for example, computers were proposed to control chemical processes in the late 1940s [Sto95].

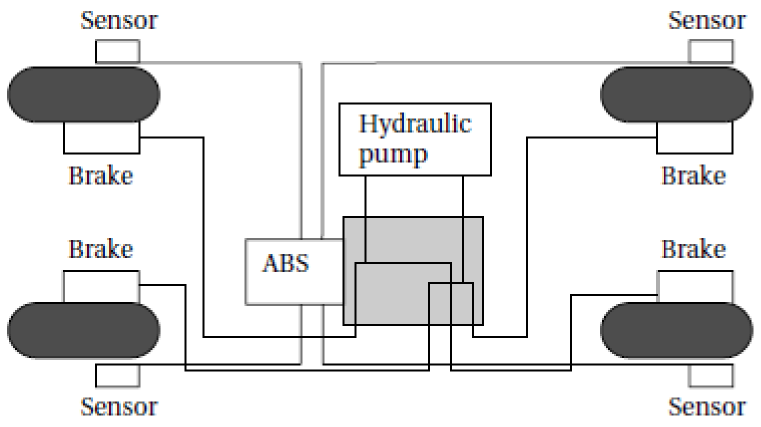
A microprocessor is a single-chip CPU. Very large scale integration (VLSI) the acronym is the name technology has allowed us to put a complete CPU on a single chip since 1970s, but those CPUs were very simple. The first microprocessor, the Intel 4004, was designed for an embedded application, namely, a calculator. The calculator was not a general-purpose computer—it merely provided basic arithmetic functions. However, Ted Hoff of Intel realized that a general-purpose computer programmed properly could implement the required function, and that the computer-on-a-chip could then be reprogrammed for use in other products as well. Since integrated circuit design was (and still is) an expensive and time consuming process, the ability to reuse the hardware design by changing the software was a key breakthrough. The HP-35 was the first handheld calculator to perform transcendental functions [Whi72]. It was introduced in 1972, so it used several chips to implement the CPU, rather than a single-chip microprocessor. However, the ability to write programs to perform math rather than having to design digital circuits to perform operations like trigonometric functions was critical to the successful design of the calculator. 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. There was a trend toward electronics in automobiles in general—electronic devices could be used to replace the mechanical distributor. But the big push toward microprocessor-based engine control came from two nearly simultaneous developments: The oil shock of the 1970s caused consumers to place much higher value on fuel economy, and fears of pollution resulted in laws restricting automobile engine emissions. The combination of low fuel consumption and low emissions is very difficult to achieve; to meet these goals without compromising engine performance, automobile manufacturers turned to sophisticated control algorithms that could be implemented only with microprocessors.

Microprocessors come in many different levels of sophistication; they are usually classified by their word size. An 8-bit ***microcontroller*** is designed for low-cost applications and includes on-board memory and I/O devices; a 16-bit microcontroller is often used for more sophisticated applications that may require either longer word lengths or off-chip I/O and memory; and a 32-bit ***RISC*** microprocessor offers very high performance for computation-intensive applications. Given the wide variety of microprocessor types available, it should be no surprise that microprocessors are used in many ways. There are many household uses of microprocessors. 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 makes extensive use of embedded processors. In some cases, specialized CPUs are designed to execute important algorithms—an example is the CPU designed for audio processing in the SGS Thomson chip set for DirecTV [Lie98]. This processor is designed to efficiently implement programs for digital audio decoding. A programmable CPU was used rather than a hardwired unit for two reasons: First, it made the system easier to design and debug; and second, it allowed the possibility of upgrades and using the CPU for other purposes. A high-end automobile may have 100 microprocessors, but even inexpensive cars today use 40 microprocessors. Some of these microprocessors do very simple things such as detect whether seat belts are in use. Others control critical functions such as the ignition and braking systems. Application Example describes some of the microprocessors used in the BMW 850i.

Application 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 following diagram. This hookup allows the ABS system to modulate the brakes in order to keep the wheels from locking. The ABS

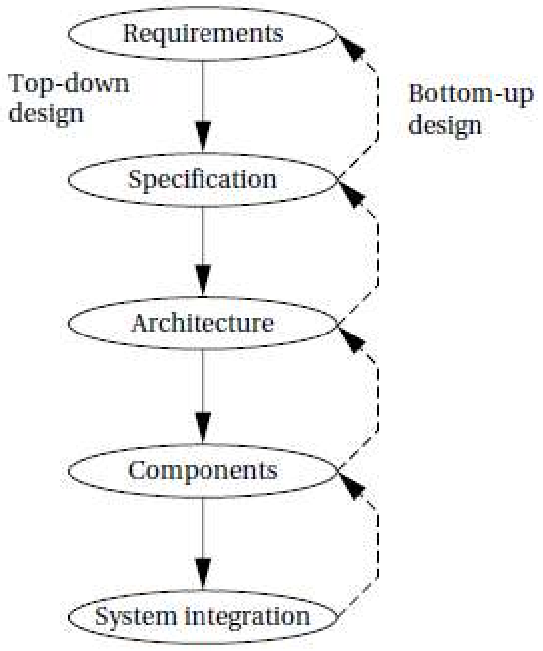
system uses sensors on each wheel to measure the speed of the wheel. The wheel speeds are used by the ABS system to determine how to vary the hydraulic fluid pressure to prevent the wheels from skidding. The ASC _ T system's job is to control the engine power and the brake to improve the car's stability during maneuvers. The ASC _ T controls four different systems: throttle, ignition timing, differential brake, and (on automatic transmission cars) gear shifting. The ASC_T can be turned off by the driver, which can be important when operating with tire snow chains. The ABS and ASC _ T must clearly communicate because the ASC _ T interacts with the brake system. Since the ABS was introduced several years earlier than the ASC _ T, it was important to be able to interface ASC _ T to the existing ABS module, as well as to other existing electronic modules. The engine and control management units include the electronically controlled throttle, digital engine management, and electronic transmission control. The ASC _ T control unit has two microprocessors on two printed circuit boards, one of which concentrates on logic-relevant components and the other on performance-specific components.



THE EMBEDDED SYSTEM DESIGN PROCESS

This section provides an overview of the embedded system design process aimed at two objectives. First,it will give us an introduction to the various steps in embedded system design before we delve into them in more detail. Second, it will allow us to consider the design ***methodology*** itself. A design methodology is important for three reasons. First, it allows us to keep a scorecard on a design to ensure that we have done everything we need to do, such as optimizing ***performance*** or performing functional tests. Second, it allows us to develop computer-aided design tools. Developing a single program that takes in a concept for an embedded system and emits a completed design would be a daunting task, but by first breaking the process into manageable steps, we can work on automating (or at least semi automating) the steps one at a time. Third, a design methodology makes it much easier for members of a design team to communicate. By defining the overall process, team members can more easily understand what they are supposed to do, what they should receive from other team members at certain times, and what they are to hand off when they complete their assigned steps. Since most embedded systems are designed by teams, coordination is perhaps the most important role of a well-defined design methodology. Figure summarizes the major steps in the embedded system design process.

In this top–down view, we start with the system ***requirements***. In the next step,



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 and conclude with concrete details. 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. Decisions at one stage of design are based upon estimates of what will happen later: How fast can we make a particular function run? How much memory will we need? How much system bus capacity do we need? If our estimates are inadequate, we may have to backtrack and amend our original decisions to take the new facts into account. In general, the less experience we have with the design of similar systems, the more we will have to rely on bottom-up design information to help us refine the system. But the steps in the design process are only one axis along which we can view embedded system design. We also 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.

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: First, we gather an informal description from the customers known as requirements, and 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. Consumers of embedded systems are usually not themselves embedded system designers or even product designers. Their understanding of the system is based on how they envision users' interactions with the system. They may have unrealistic expectations as to what can be done within their budgets; and they may also express their desires in a language very different from system architects' jargon. Capturing a consistent set of requirements from the customer and then massaging those requirements into a more formal specification is a structured way to manage the process of translating from the consumer's language to the designer's.

Requirements may be ***functional*** or ***nonfunctional***. We must of course capture the basic functions of the embedded system, but functional description is often not sufficient. Typical nonfunctional 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—the customer is unlikely to be able to describe the allowable wattage.

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 goodway to refine at least the user interface portion of a system's requirements is to build a ***mock-up***. The mock-up may use canned data to simulate functionality in a restricted demonstration, and it may be executed on a PC or a workstation. But it should give the customer a good idea of how the system will be used and how the user can react to it. Physical, nonfunctional models of devices can also give customers a better idea of characteristics such as size and weight.

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

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 toward understanding system requirements. To introduce the discipline of requirements analysis as part of system design, we will use a simple requirements methodology. Figure shows a sample ***requirements form*** that can be filled out at the start of the project. We can use the form as a checklist in considering the basic characteristics of the system. Let's consider the entries in the form:

■ *Name:* This is simple but helpful. Giving a name to the project not only simplifies talking about it to other people but can also crystallize 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? Occasional user inputs? 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 controlling physical devices or processing data coming from the physical world. In most of these cases, the computations must be performed within a certain time frame. It is essential that the performance requirements be identified early since they must be carefully measured during implementation to ensure that the system works properly.
■ *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:A machine that is meant to sell at $10 most likely has a very different internal structure than a $100 system.

■ *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.

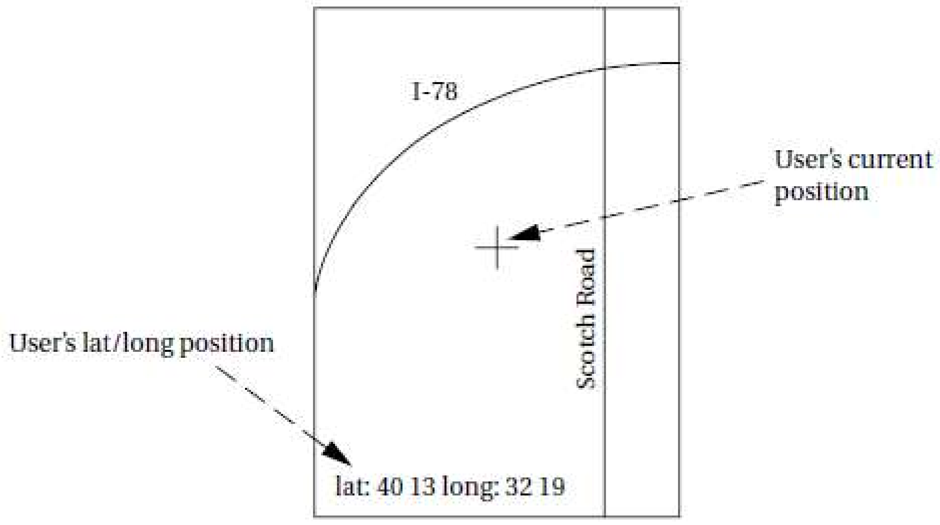
■ *Physical size and weight:* You should give some indication of the physical size of the system to help guide certain architectural decisions. A desktop machine has much more flexibility in the components used than, for example, a lapel mounted voice recorder.

A more thorough requirements analysis for a large system might use a form similar to Figure as a summary of the longer requirements document. After an introductory section containing this form, a longer requirements document could include details on each of the items mentioned in the introduction. For example, each individual feature described in the introduction in a single sentence may be described in detail in a section of the specification. After writing the requirements, you should check them for internal consistency: Did you forget to assign a function to an input or output? Did you consider all the modes in which you want the system to operate? Did you place an unrealistic number of features into a battery-powered, low-cost machine? To practice the capture of system requirements, Example creates the requirements for a GPS moving map system.

Example

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, not nautical or aviation uses that require more specialized databases and functions. The system should show major roads and other landmarks available in standard topographic databases.

■ *User interface:* The screen should have at least 400_600 pixel resolution. The device should be controlled by no more than three buttons. A menu system should pop up on the screen when buttons are pressed to allow the user to make selections to control the system.

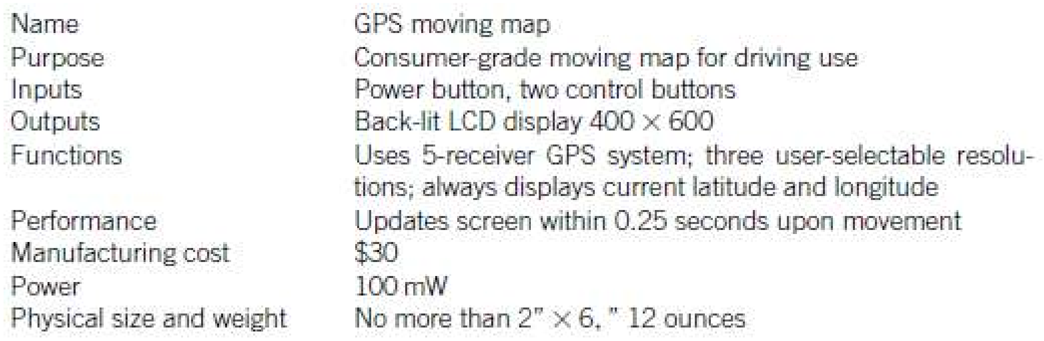
■ *Performance:* The map should scroll smoothly. Upon power-up, a display should take no more than one second to appear, and the system should be able to verify its position and display the current map within 15 s.

■ *Cost:* The selling cost (street price) of the unit should be no more than $100.

■ *Physical size and weight:* The device should fit comfortably in the palm of the hand.

■ *Power consumption:* The device should run for at least eight hours on four AA batteries.

Note that many of these requirements are not specified in engineering units—for example, physical size is measured relative to a hand, not in centimeters. Although these requirements must ultimately be translated into something that can be used by the designers, keeping a record of what the customer wants can help to resolve questions about the specification that may crop up later during design. Based on this discussion, let’s write a requirements chart for our 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 displays current latitude and longitude
Performance	Updates screen within 0.25 seconds upon movement
Manufacturing cost	$30
Power	100 mW
Physical size and weight	No more than 2” × 6, ” 12 ounces

Specification

The specification is more precise—it serves as the contract between the customer and the architects. As such, the specification must be carefully written so that it accurately reflects the customer’s requirements and does so in a way that can be clearly followed during design. Specification is probably the least familiar phase of this methodology for neophyte designers, but it is essential to creating working systems with a minimum of designer effort. Designers who lack a clear idea of what they want to build when they begin typically make faulty assumptions early in the process that aren’t obvious until they have a working system. At that point, the only solution is to take the machine apart, throw away some of it, and start again. The specification should be understandable enough so that someone can verify that it meets system requirements and overall expectations of the customer. It should also be unambiguous enough that designers know what they need to build. Designers can run into several different types of problems caused by unclear specifications. If the behavior 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:

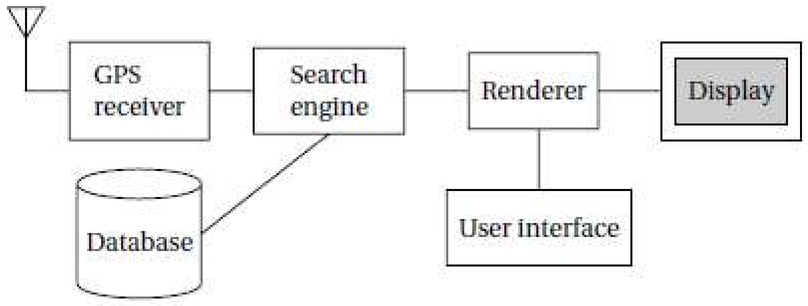
- ■ Data received from the GPS satellite constellation.
- ■ Map data.
- ■ User interface.
- ■ Operations that must be performed to satisfy customer requests.
- ■ Background actions required to keep the system running, such as operating the GPS receiver.

UML, a language for describing specifications, will be introduced later and we will use it to write a specification. We will practice writing specifications in each chapter as we work through example system designs. We will also study specification techniques in more later.

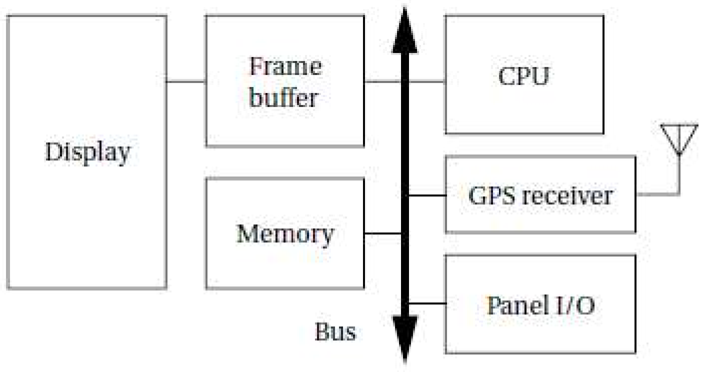
Architecture Design

The specification does not say how the system does things, only what the system does. Describing how the system implements those functions is the purpose of the architecture. The architecture is a plan for the overall structure of the system that will be used later to design the components that make up the architecture. The creation of the architecture is the first phase of what many designers think of as design. To understand what an architectural description is, let's look at sample architecture for the moving map of Example Figure shows sample system architecture in the form of a ***block diagram*** that shows major operations and data flows among them.

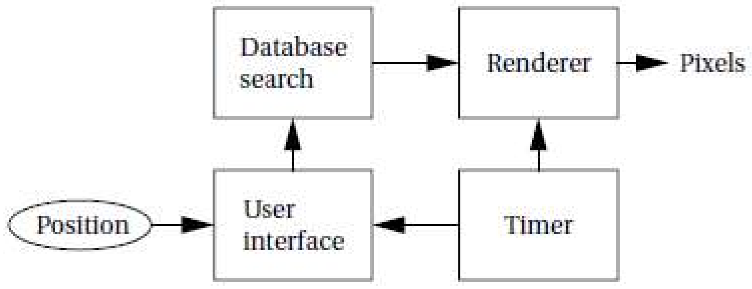
This block diagram is still quite abstract—we have not yet specified which operations will be performed by software running on a CPU, what will be done by special-purpose hardware, and so on. The diagram does, however, go a long way toward describing how to implement the functions described in the specification. We clearly see, for example, that we need to search 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.



Only after we have designed an initial architecture that is not biased toward too many implementation details should we refine that system block diagram into two block diagrams: one for hardware and another for software. 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. In particular, we have chosen to use two memories: a frame buffer for the pixels to be displayed and a separate program/data memory for general use by the CPU. The software block diagram fairly closely follows the system block diagram, but we have added a timer to control when we read the buttons on the user interface and render data onto the screen. To have a truly complete architectural description, we require more detail, such as where units in the software block diagram will be executed in the hardware block diagram and when operations will be performed in time. 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 a system architecture and refining that to hardware and software architectures



Hardware



Software

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. How do we know that our hardware and software architectures in fact meet constraints on speed, cost, and so on? We must somehow be able to estimate the properties of the components of the block diagrams, such as the search and rendering functions in the moving map system. Accurate estimation derives in part from experience, both general design experience and particular experience with similar systems. However, we can sometimes create simplified models to help us make more accurate estimates. Sound estimates of all nonfunctional constraints during the architecture phase are crucial, since decisions based on bad data will show up during the final phases of design, indicating that we did not, in fact, meet the specification.

Designing Hardware and Software Components
The architectural description tells us what components we need. The component design effort builds those components in conformance to the architecture and specification. The components will in general 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—not only is the data in a predefined format, but it is highly compressed to save storage. Using standard software for these access functions not only saves us design time, but it may give us a faster implementation for specialized functions such as the data decompression phase. You will have to design some components yourself. Even if you are using only standard integrated circuits, you may have to design the printed circuit board that connects them. You will probably have to do a lot of custom programming as well. When creating these embedded software modules, you must of course make use of your expertise to ensure that the system runs properly in real time and that it does not take up more memory space than is allowed. The power consumption of the moving map software example is particularly important. You may need to be very careful about how you read and write memory to minimize power—for example, since memory accesses are a major source of power consumption, memory transactions must be carefully planned to avoid reading the same data several times.

System Integration
Only after the components are built do we have the satisfaction of putting them together and seeing a working system. Of course, this phase usually consists of a lot more than just plugging everything together and standing back. Bugs are typically found during system integration, and good planning can help us find the bugs quickly. By building up the system in phases and running properly chosen tests, we can often find bugs more easily. 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. Only by fixing the simple bugs early will we be able to uncover the more complex or obscure bugs that can be identified only by giving the system a hard workout. We need to ensure during the architectural and component design phases that we make it as easy as possible to assemble the system in phases and test functions relatively independently.

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 stet work correctly and how they can be fixed is a challenge in itself. Careful attention to inserting appropriate debugging facilities during design can help ease system integration problems, but the nature of embedded computing means that this phase will always be a challenge.

FORMALISMS FOR SYSTEM DESIGN

As mentioned in the last section, we perform a number of different design tasks at different levels of abstraction throughout this book: creating requirements and specifications, architecting the system, designing code, and designing tests. It is often helpful to conceptualize these tasks in diagrams. Luckily, there is a visual language that can be used to capture all these design tasks: the ***Unified Modeling Language (UML).***

UML was designed to be useful at many levels of abstraction in the design process. UML is useful because it encourages design by successive refinement and progressively adding detail to the design, rather than rethinking the design at each new level of abstraction. UML is an ***object-oriented*** modeling language. We will see precisely what we mean by an object in just a moment, but object-oriented design emphasizes two concepts of importance:

■ It encourages the design to be described as a number of interacting objects, rather than a few large monolithic blocks of code.

■ At least some of those objects will correspond to real pieces of software or hardware in the system. We can also use UML to model the outside world that interacts with our system, in which case the objects may correspond to people or other machines. It is sometimes important to implement something we think of at a high level as a single object using several distinct pieces of code or to otherwise break up the object correspondence in the implementation. However, thinking of the design in terms of actual objects helps us understand the natural structure of the system. Object-oriented (often abbreviated OO) specification can be seen in two complementary ways:

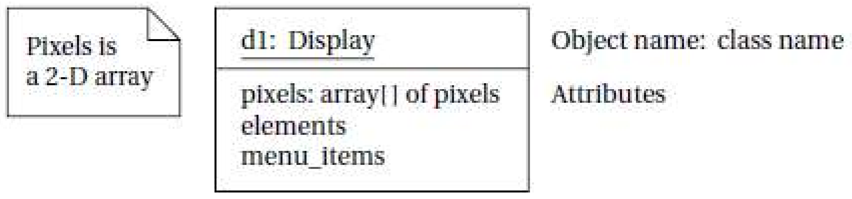
■ Object-oriented specification allows a system to be described in a way that closely models real-world objects and their interactions.

■ Object-oriented specification provides a basic set of primitives that can be used to describe systems with particular attributes, irrespective of the relationships of those systems' components to real-world objects. Both views are useful. At a minimum, object-oriented specification is a set of linguistic mechanisms. In many cases, it is useful to describe a system in terms of real-world analogs. However, performance, cost, and so on may dictate that we change the specification to be different in some ways from the real-world elements we are trying to model and implement. In this case, the object-oriented specification mechanisms are still useful. What is the relationship between an object-oriented specification and an object oriented programming language (such as C++)? A specification language may not be executable. But both object-oriented specification and programming languages provide similar basic methods for structuring large systems.

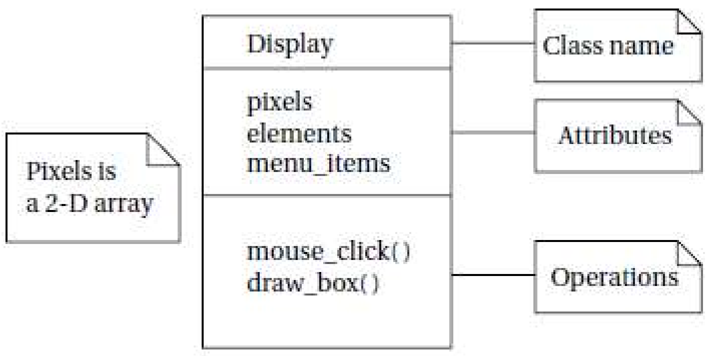
Unified Modeling Language (UML)—the acronym is the name is a large language, and covering all of it is beyond the scope of this book. In this section, we introduce only a few basic concepts. In later chapters, as we need a few more UML concepts, we introduce them to the basic modeling elements introduced here. Because UML is so rich, there are many graphical elements in a UML diagram. It is important to be careful to use the correct drawing to describe something—for instance, UML distinguishes between arrows with open and filled-in arrowheads, and solid and broken lines. As you become more familiar with the language, uses of the graphical primitives will become more natural to you. We also won't take a strict object-oriented approach. We may not always use objects for certain elements of a design—in some cases, such as when taking particular aspects of the implementation into account, it may make sense to use another design style. However, object-oriented design is widely applicable, and no designer can consider himself or herself design literate without understanding it.

Structural Description

By ***structural description***, we mean the basic components of the system; we will learn how to describe how these components act in the next section. The principal component of an object-oriented design is, naturally enough, the ***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. 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. 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; remember that when we instantiate the class an object, that object will have its own memory so that different objects of the same class have their own values for the attributes. Other classes can examine and modify class attributes; if we have to do something more complex than use the attribute directly, we define a behavior to perform that function.



An object in UML notation.



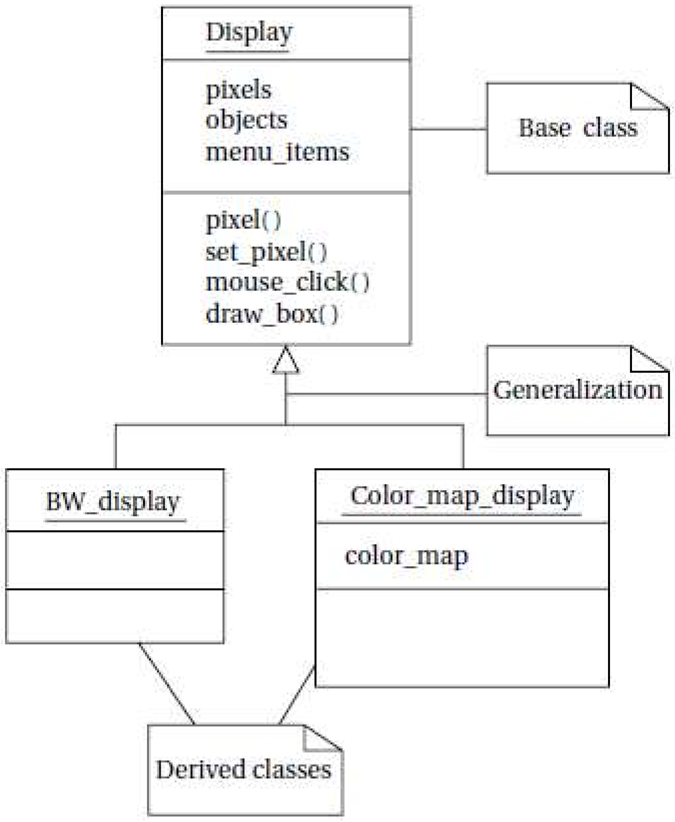
A class defines both the ***interface*** for a particular type of object and that object's ***implementation***. When we use an object, we do not directly manipulate its attributes—we can only read or modify the object's state through the operations that define the interface to the object. (The implementation includes both the attributes and whatever code is used to implement the operations.) As long as we do not change the behavior of the object seen at the interface, we can change the implementation as much as we want. This lets us improve the system by, for example, speeding up an operation or reducing the amount of memory required without requiring changes to anything else that uses the object.

Clearly, the choice of an interface is a very important decision in object-oriented design. The proper interface must provide ways to access the object's state (since we cannot directly see the attributes) as well as ways to update the state. We need to make the object's interface general enough so that we can make full use of its capabilities. However, excessive generality often makes the object large and slow. Big, complex interfaces also make the class definition difficult for designers to understand and use properly. There are several types of ***relationships*** that can exist between objects and classes:

■ ***Association*** occurs between objects that communicate with each other but have no ownership relationship between them.
■ ***Aggregation*** describes a complex object made of smaller objects.
■ ***Composition*** is a type of aggregation in which the owner does not allow access to the component objects.
■ ***Generalization*** allows us to define one class in terms of another.

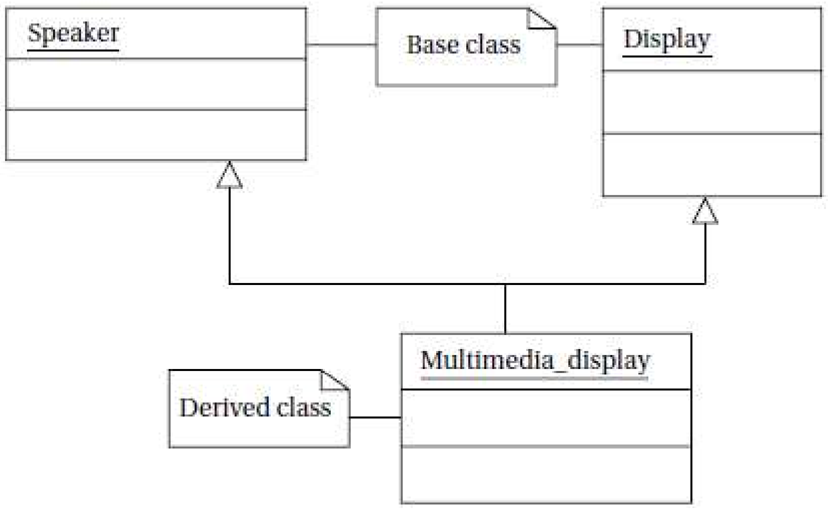
The elements of a UML class or object do not necessarily directly correspond to statements in a programming language—if the UML is intended to describe something more abstract than a program, there may be a significant gap between the contents of the UML and a program implementing it. The attributes of an object do not necessarily reflect variables in the object. An attribute is some value that reflects the current state of the object. In the program implementation, that value could be computed from some other internal variables. The behaviors of the object would, in a higher-level specification, reflect the basic things that can be done with an object. Implementing all these features may require breaking up a behavior into several smaller behaviors—for example, initialize the object before you start to change its internal state-derived classes.

Unified Modeling Language, like most object-oriented languages, allows us to define one class in terms of another. An example is shown in Figure, 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. The second, *Color_map_display*, uses a graphic device known as a color map to allow the user to select from a



large number of available colors even with a small number of bits per pixel. This class defines a *color_map* attribute that determines how pixel values are mapped onto display colors. A ***derived class*** inherits all the attributes and operations from its ***base class***. In this class, *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. Inheritance has two purposes. It of course allows us to succinctly describe one class that shares some characteristics with another class. Even more important, it captures those relationships between classes and documents them. If we ever need to change any of the classes, knowledge of the class structure helps us determine the reach of changes—for example, should the change affect only *Color_map_display* objects or should it change all Display objects?

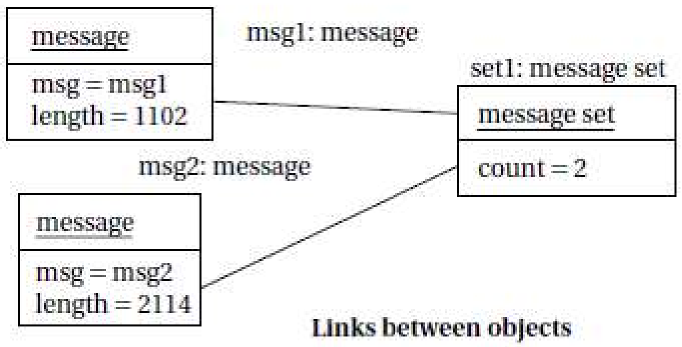
Unified Modeling Language considers inheritance to be one form of generalization. A generalization relationship is shown in a UML diagram as an arrow with an open (unfilled) arrowhead. Both *BW_display* and *Color_map_display* are specific

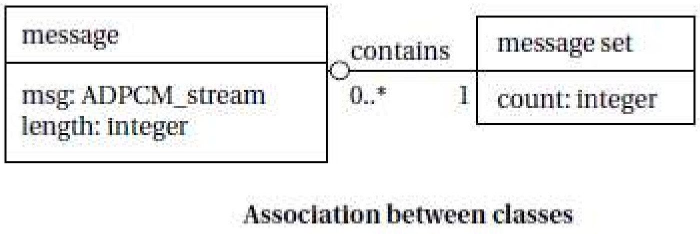


versions of *Display*, so *Display* generalizes both of them. UML also allows us to define ***multiple inheritance***, in which a class is derived from more than one base class. (Most object-oriented programming languages support multiple inheritance as well.) An example of multiple inheritance is shown in Figure; we have omitted the details of the classes' attributes and operations for simplicity. 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*. Because multiple inheritance causes the sizes of the attribute set and operations to expand so quickly, it should be used with care.

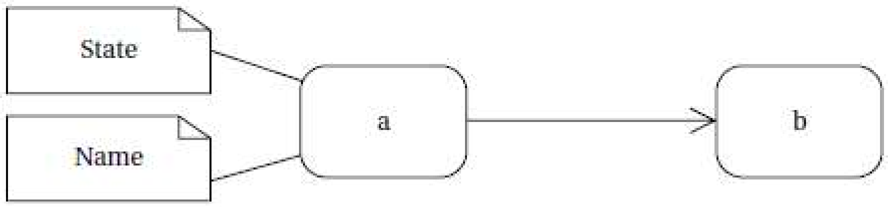
A ***link*** describes a relationship between objects; association is to link as class is to object. We need links because objects often do not stand alone; associations let us capture type information about these links. Figure 1.9 shows examples of links and an association. When we consider the actual 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. Sometimes we may want to attach data to the links themselves; we can specify this in the association by attaching a class-like box to the association's edge, which holds the association's data.

Typically,we find that we use a certain combination of elements in an object or class many times.We can give these patterns names, which are called ***stereotypes***





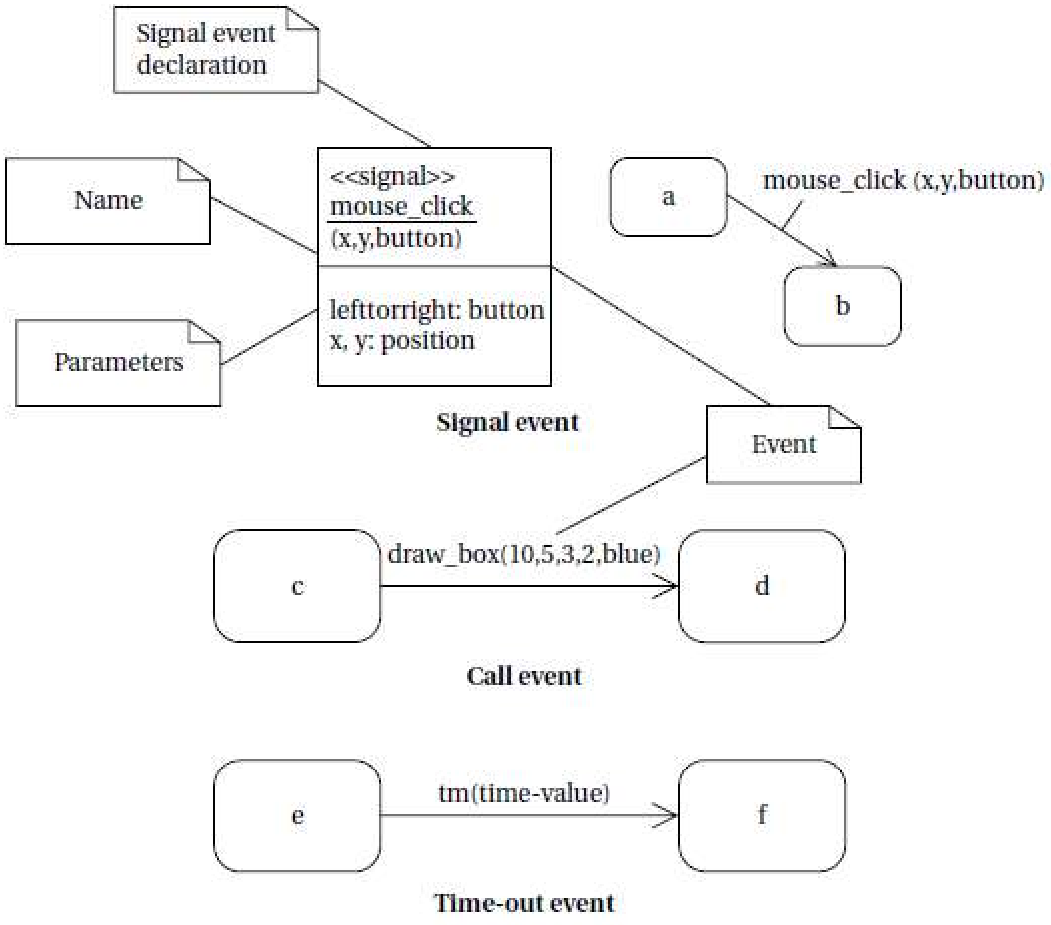
Links and association.



in UML. A stereotype name is written in the form <<signal>>. Figure shows a stereotype for a signal, which is a communication mechanism.

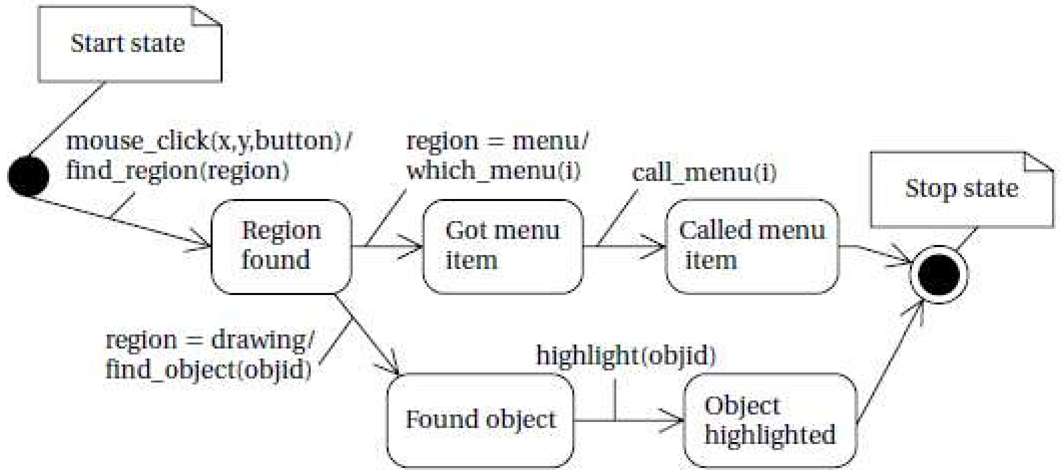
Behavioral Description

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***. Figure shows UML states; the transition between two states is shown by a skeleton 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***.



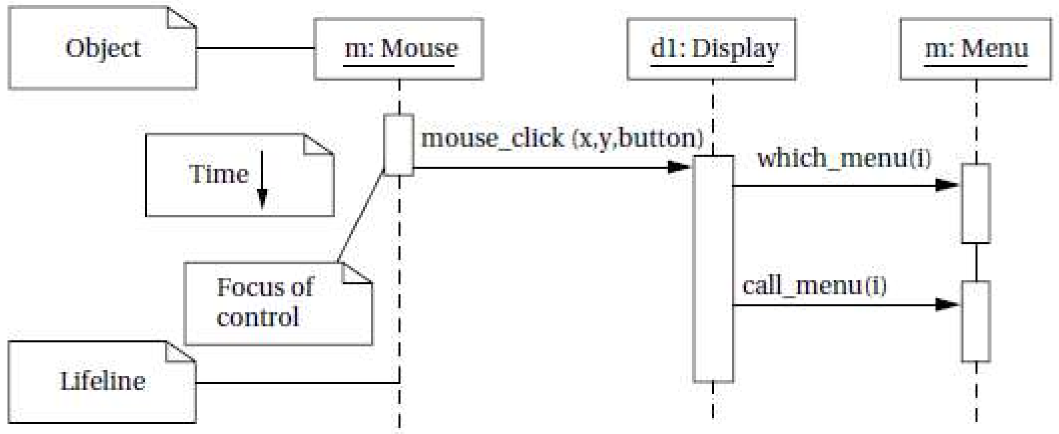
An event is some type of action. The event may originate outside the system, such as a user pressing a button. It may also originate inside, such as when one routine finishes its computation and passes the result on to another routine. We will concentrate on the following three types of events defined by UML, as illustrated in Figure.

- ■ 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



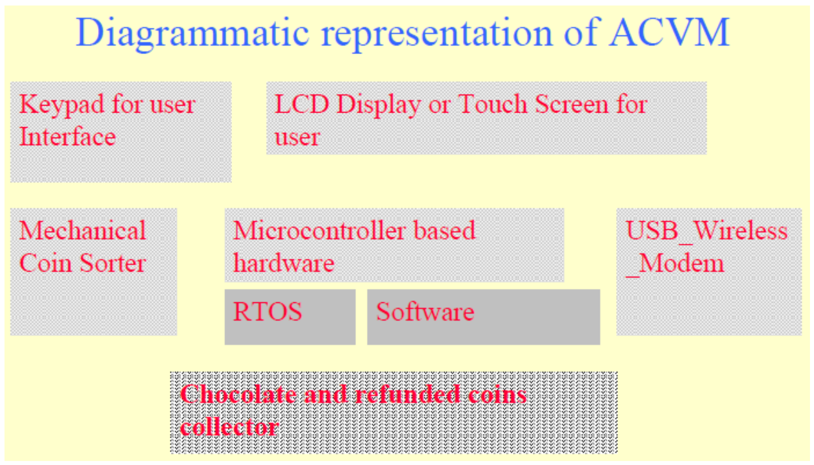
external timer. This notation simplifies the specification and allows us to defer implementation details about the time-out mechanism. We show the occurrence of all types of signals in a UML diagram in the same way— as a label on a transition.

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 Figure. The start and stop states are special states that help us to organize the flow of the state machine. 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. Splitting a complex operation into several states helps document the required steps, much as subroutines can be used to structure code. 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 Figure. 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—it is not convenient for showing a number of mutually exclusive possibilities. In this case, the sequence shows what happens when a mouse click is on the menu region. 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 PROCESS EXAMPLES
Automatic Chocolate vending machine


Diagrammatic representation of ACVM
Keypad for user Interface
LCD Display or Touch Screen for user
Mechanical Coin Sorter
Microcontroller based hardware
USB_Wireless _Modem
RTOS
Software
Chocolate and refunded coins collector

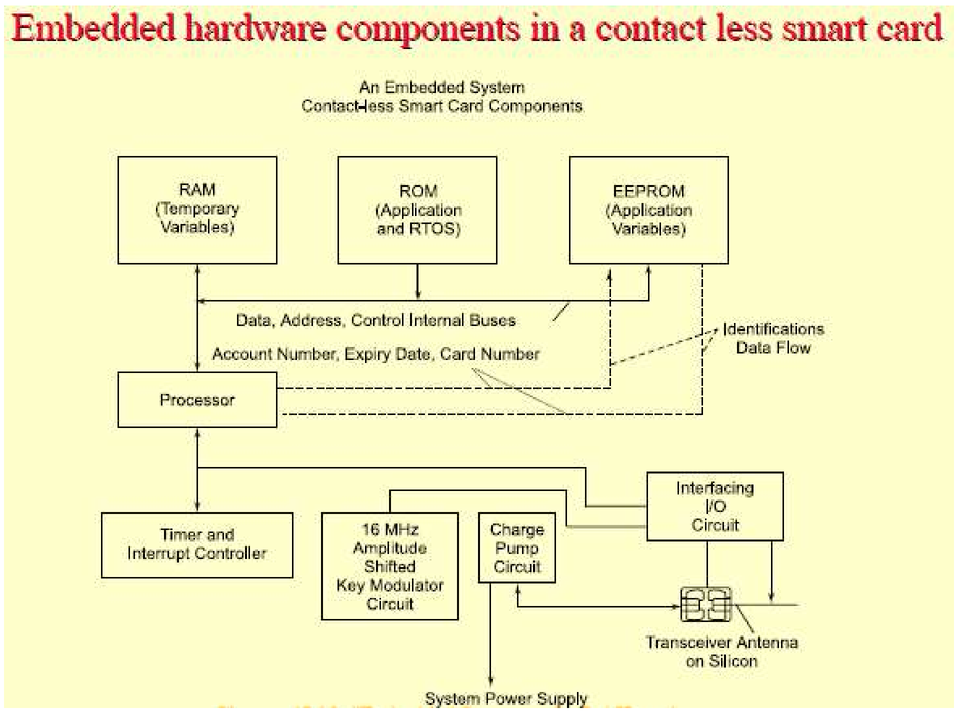
Keypad on the top of the machine. LCD display unit on the top of the machine. It displays menus, text entered into the ACVM and pictograms, welcome, thank and other messages. Graphic interactions with the machine. Displays time and date. Delivery slot so that child can collect the chocolate and coins, if refunded. Internet connection port so that owner can know status of the ACVM sales from remote.

ACVM Hardware units
Microcontroller or ASIP (Application Specific Instruction Set Processor). RAM for storing temporary variables and stack. ROM for application codes and RTOS codes for scheduling the tasks. Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, answers of FAQs. Timer and Interrupt controller. A

TCP/IP port (Internet broadband connection) to the ACVM for remote control and for getting ACVM status reports by owner. ACVM specific hardware. Power supply.

ACVM Software components
_ Keypad input read
_ Display
_ Read coins
_ Deliver chocolate
_ TCP/IP stack processing
_ TCP/IP stack communication

Smart Card
Smart card– a plastic card in ISO standard dimensions, 85.60 mm x 53.98 x 0.80 mm.
_ Embedded system on a card.
_ SoC (System-On-Chip).
_ ISO recommended standards are ISO7816 (1 to 4) for host-machine contact based cards and ISO14443 (Part A or B) for the contact-less cards.
_ Silicon chip is just a few mm in size and is concealed in-between the layers. Its very small size protects the card from bending


Embedded hardware components in a contact less smart card
An Embedded System
Contact-less Smart Card Components
RAM (Temporary Variables)
ROM (Application and RTOS)
EEPROM (Application Variables)
Data, Address, Control Internal Buses
Account Number, Expiry Date, Card Number
Identifications Data Flow
Processor
Interfacing I/O Circuit
Timer and Interrupt Controller
16 MHz Amplitude Shifted Key Modulator Circuit
Charge Pump Circuit
Transceiver Antenna on Silicon
System Power Supply

Embedded hardware components
_ Microcontroller or ASIP (Application Specific Instruction Set Processor)
_ RAM for temporary variables and stack
_ ROM for application codes and RTOS codes for scheduling the tasks
_ EEPROM for storing user data, user address, user identification codes, card number and expiry date
_ Timer and Interrupt controller

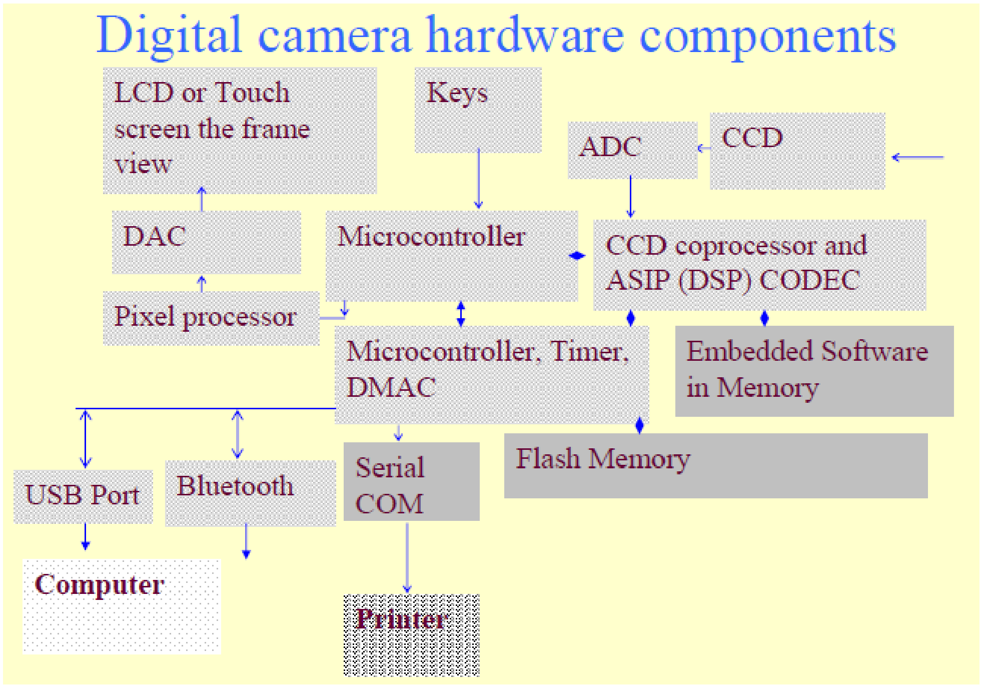
_ A carrier frequency ~16 MHz generating circuit and Amplitude Shifted Key (ASK)
_ Interfacing circuit for the I/Os
_ Charge pump

ROM
Fabrication key, Personalization key An utilization lock.
_ RTOS and application using only the logical addresses

Embedded Software
_ Boot-up, Initialisation and OS programs
_ Smart card secure file system
_ Connection establishment and termination
_ Communication with host
_ Cryptography
_ Host authentication
_ Card authentication
_ Addition parameters or recent new data sent by the host (for example, present balance left).
Smart Card OS Special features
_ Protected environment.
_ Every method, class and run time libraryshould be scalable.
_ Code-size generated be optimum.
_ Memory should not exceed 64 kB memory.
_ Limiting uses of specific data types; multidimensional arrays, long 64-bit integer and floating points

Smart Card OS Limiting features
_ Limiting uses of the error handlers, exceptions, signals, serialization, debugging and profiling. [Serialization means process of converting an object is converted into a data stream for transferring it to network or from one process to another. At receiver end there is de-serialization Smart Card OS File System and Classes
_ Three-layered file system for the data.
_ Master file to store all file headers.
_ Dedicated file to hold a file grouping and headers of the immediate successor elementary files of the group.
_ Elementary file to hold the file header and its file data.
_ Fixed-length or variable-file length management
_ Classes for the network, sockets, connections, data grams, character-input output and streams, security management, digital-certification, symmetric and asymmetric keys-based cryptography and digital signatures..

Digital Camera


Digital camera hardware components
LCD or Touch screen the frame view
Keys
ADC
CCD
DAC
Microcontroller
CCD coprocessor and ASIP (DSP) CODEC
Pixel processor
Microcontroller, Timer, DMAC
Embedded Software in Memory
USB Port
Bluetooth
Serial COM
Flash Memory
Computer
Printer

A typical Camera

_ 4 M pixel/6 M pixel still images, clear visual display (ClearVid) CMOS sensor, 7 cm wide LCD photo display screen, enhanced imaging processor, double anti blur solution and high-speed processing engine, 10X optical and 20X digital zooms

_ Record high definition video-clips. It therefore has speaker microphone(s) for high quality recorded sound.

_ Audio/video Out Port for connecting to a TV/DVD player.

Arrangements

_ Keys on the camera.

_ Shutter, lens and charge coupled device (CCD) array sensors

_ Good resolution photo quality LCD display unit

_ Displays text such as image-title, shooting data and time and serial number. It displays messages. It displays the GUI menu when user interacts with the camera.

_ Self-timer lamp for flash.

Internal units

_ Internal memory flash to store OS and embedded software and limited number of image files

_ Flash memory stick of 2 GB or more for large storage.

_ Universal Serial Bus (USB), Bluetooth and serial COM port for connecting it to computer, mobile and printer. LCD screen to display frame view.

_ Saved images display using the navigation keys.

_ Frame light falls on the CCD array, which through an ADC transmits the bits for each pixel in each row in the frame and for the dark area pixels in each row for offset correction in CCD signaled light intensities for each row.

_ The CCD bits of each pixel in each row and column are offset corrected by CCD signal processor (CCDSP).

ASIP and Single purpose processors

_ For Signals compression using a JPEG CODEC and saved in one jpg file for each frame.
_ For DSP for compression using the discrete cosine transformations (DCTs) and decompression.
_ For DCT Huffman coding for the JPEG compression.
_ For decompression by inverse DCT before the DAC sends input for display unit through pixel processor.
_ Pixel processor (for example, image contrast, brightness, rotation, translation, color adjustment)

Digital Camera Hardware units

_ Microcontroller or ASIP (Application Specific Instruction Set Processor)
_ Multiple processors (CCDSP, DSP, Pixel Processor and others)
_ RAM for storing temporary variables and stack
_ ROM for application codes and RTOS codes for scheduling the tasks Timer, Flash memory for storing user preferences, contact data, user address, user date of birth, user identification code, ADC, DAC and Interrupt controller
_ The DAC gets the input from pixel processor, which gets the inputs from JPEG file for the saved images and also gets input directly from the CCDSP through pixel processor or the frame in present view
_ USB controller Direct Memory Access controller
_ LCD controller
_ Battery and external charging circuit

Digital Camera Software components

_ CCD signal processing for off-set correction
_ JPEG coding
_ JPEG decoding
_ Pixel processing before display
_ Memory and file systems
_ Light, flash and display device drivers
_ LCD, USB and Bluetooth Port device- drivers for port operations for display, printer and Computer communication control

Light, flash and display device drivers
CCD signal processing
JPEG coding
JPEG decoding
Pixel co-processing
LCD and USB Port device drivers
LCD, Bluetooth COM and USB Port device drivers

UNIT-II
THE 8051 ARCHITECTURE

INTRODUCTION TO MICRO CONTROLLERS

INTRODUCTION:

We have noticed that Microprocessor is just not self-sufficient, and it requires other components like memory and input/output devices to form a minimum workable system configuration. To have all these components in a discrete form and to assemble them on a PCB is usually not an affordable solution for the following reasons:

1) The overall system cost of a microprocessor based system built around a CPU, memory and other peripherals is high as compared to a microcontroller based system.
2) A large sized PCB is required for assembling all these components, resulting in an enhanced cost of the system.
3) Design of such PCBs requires a lot of effort and time and thus the overall product design requires more time.
4) Due to the large size of the PCB and the discrete components used, physical size of the product is big and hence it is not handy.
5) As discrete components are used, the system is not reliable nor is it easy to trouble-shoot such a system.

Considering all these problems, Intel decided to integrate a microprocessor along with I/O ports and minimum memory into a single package. Another frequently used peripheral, a programmable timer, was also integrated to make this device a self-sufficient one. This device which contains a microprocessor and the above mentioned components has been named a microcontroller. A microcontroller is a microprocessor with integrated peripherals. Design with microcontrollers has the following advantages:

1. As the peripherals are integrated into a single chip, the overall system cost is very low.
2. The size of the product is small as compared to the microprocessor based systems thus very handy.
3. The system design requires very little efforts and is easy to troubleshoot and maintain.
4. As the peripherals are integrated with a microprocessor, the system is more reliable.
5. Though a microcontroller may have on-chip RAM, ROM and I/O ports, additional RAM, ROM and I/O ports may be interfaced externally, if required.

6. The microcontrollers with on-chip ROM provide a software security feature which is not available with microprocessor based systems using ROM/EPROM.

However, in case of a larger system design, which requires more number of I/O ports and more memory capacity, the system designer may interface external I/O ports and memory with the system. In such cases, the microcontroller based systems are not so attractive as they are in case of the small dedicated systems. Figure 17.1 shows a typical microcontroller internal block diagram.
As a microcontroller contains most of the components required to form a microprocessor system, it is sometimes called a single chip microcomputer, since it also has the ability to easily implement simple control functions.

OVERVIEW OF 8051 MICRO CONTROLLER

Let us look at Intel's 8-bit microcontroller family, popularly known as MCS-51 family. The earlier versions of Intel's microcontrollers do not have on-chip EPROM. 8031 was one such microcontroller from Intel, followed by the 8051 family. 8751 was the first microcontroller version with on-chip EPROM, followed by a number of 8751 versions with slight modifications. Recently, an electrically programmable and erasable version of 8051, named as 8951, has been introduced. Table shows the comparison between different versions of 8051. All these members of the 8051 family have identical instruction set and similar architecture with slight variations as shown in Table.

ARCHITECTURE OF 8051

The internal architecture of 8051 is presented in Fig.
The functional description of each block is presented briefly below.
Accumulator (ACC): The accumulator register (ACC or A) acts as an operand register, in case of some instructions. This may either be implicit or specified in the instruction.
B Register: This register is used to store one of the operands for multiply and divide instructions. In other instructions, it may just be used as a scratch pad.
Program Status Word (PSW): This set of flags contains the status information.
Stack Pointer (SP): This 8-bit wide register is incremented before the data is stored onto the stack using push or call instructions. This register contains 8-bit stack top address. The

stack may be defined anywhere in the on-chip 128-byte RAM. After reset, the SP register is initialised to 07. After each write to stack operation, the 8-bit contents of the operand are stored onto the stack, after incrementing the SP register by one. Thus if SP contains 07 H, the forthcoming PUSH operation will store the data at address 08H in the internal RAM. The SP content will be incremented to 08.

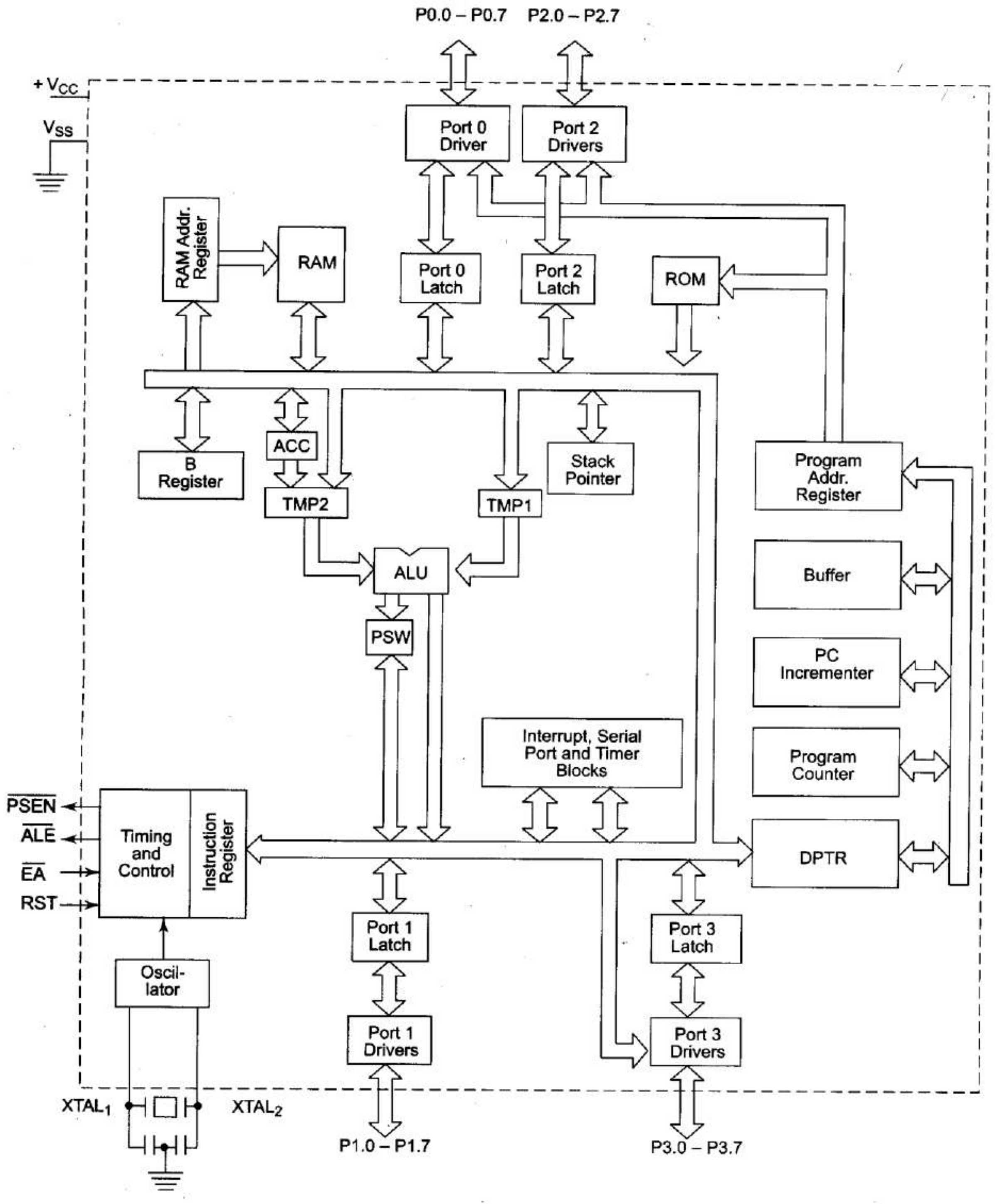


Fig. 17.2 *8051 Block Diagram (Intel Corp.)*

Data Pointer (DTPR): This 16-bit register contains a higher byte (DPH) and the lower byte (DPL) of a 16-bit external data RAM address. It is accessed as a 16-bit register or two 8-bit registers as specified above.

Port 0 to 3 Latches and Drivers: These four latches and driver pairs are allotted to each of the four on-chip I/O ports. Using the allotted addresses, the user can communicate with these ports. These are identified as P0, PI, P2 and P3.

Serial Data Buffer: The serial data buffer internally contains two independent registers. One of them is a transmit buffer which is necessarily a parallel-in serial-out register. The other is called receive buffer which is a serial-in parallel-out register. The serial data buffer is identified as SBUF.

Timer Registers: These two 16-bit registers can be accessed as their lower and upper bytes. For example, TL0 represents the lower byte of the timing register 0, while TH0 represents higher bytes of the timing register 0. Similarly, TL1 and TH1 represent lower and higher bytes of timing register 1.

Control Registers: The special function registers IP, IE, TMOD, TCON, SCON and PCON contain control and status information for interrupts, timers/counters and serial port.

Timing and Control Unit: This unit derives all the necessary timing and control signals required for the internal operation of the circuit. It also derives control signals required for controlling the external system bus.

Oscillator: This circuit generates the basic timing clock signal for the operation of the circuit using crystal oscillator.

Instruction Register: This register decodes the opcode of an instruction to be executed and gives information to the timing and control unit to generate necessary signals for the execution of the instruction.

EPROM and Program Address Register: These blocks provide an on-chip EPROM/PROM and a mechanism to internally address it. Note that EPROM is not available in all 8051 versions.

RAM and RAM Address Register: These blocks provide internal 128 bytes of RAM and a mechanism to address it internally.

ALU: The arithmetic and logic unit performs 8-bit arithmetic and logical operations over the operands held by the temporary registers TMP1 and TMP2. Users cannot access these temporary registers.

SFR Register Bank: This is a set of special function registers, which can be addressed using their respective addresses which lie in the range 80H to FFH.

Finally, the interrupt, serial port and timer units control and perform their specific functions under the control of the timing and control unit.

PIN DESCRIPTIONS OF 8051

8051 is available in a 40-pin plastic and ceramic DIP packages. The pin diagram of 8051 is shown in Fig. 17.3 followed by description of each pin.

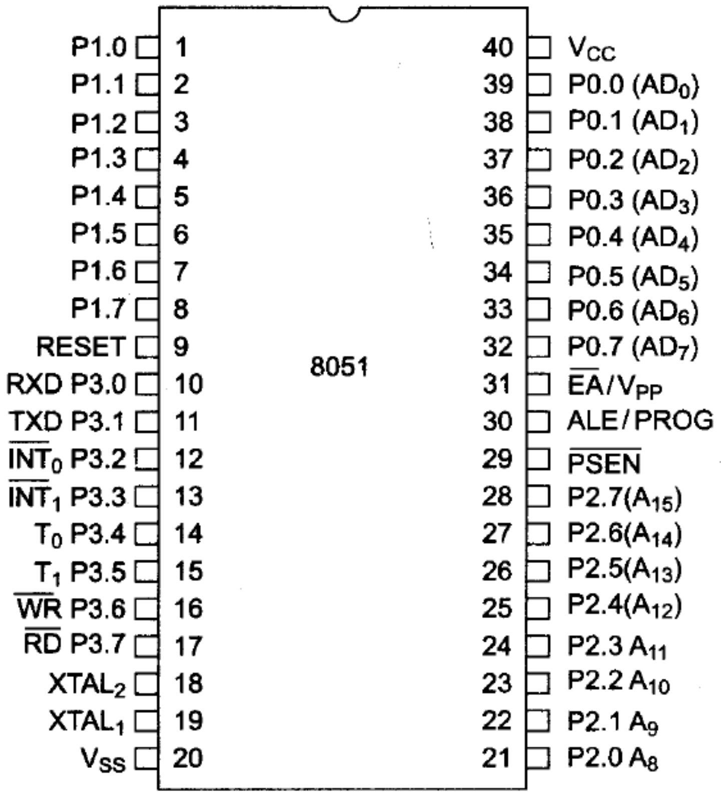


Fig. 17.3 *8051 Pin Configuration (Intel Corp.)*

V_{cc} This is a +5 V supply voltage pin

V_{ss} This is a return pin for the supply.

RESET The reset input pin resets the 8051, only when it goes high for two or more machine cycles. For a proper reinitialization after reset, the clock must be running.

ALE/PROG The address latch enable output pulse indicates that the valid address bits are available on their respective pins. This ALE signal is valid only for external memory accesses. Normally, the ALE pulses are emitted at a rate of one-sixth of the oscillator frequency. This pin acts as program pulse input during on-chip EPROM programming. ALE may be used for external timing or clocking purpose. One ALE pulse is skipped during each access to external data memory.

$\overline{EA}$ /V_{pp} External access enable pin, if tied low, indicates that the 8051 can address external program memory. In other words, the 8051 can execute a program in external memory, only if $\overline{EA}$ is tied low.

For execution of programs in internal memory, the $\overline{EA}$ must be tied high. This pin also receives 21 volts for programming of the on-chip EPROM.

$\overline{\text{PSEN}}$ Program store enable is an active-low output signal that acts as a strobe to read the external program memory. This goes low during external program memory accesses.

Port 0 (P0.0-P0.7) Port 0 is an 8-bit bidirectional bit addressable I/O port. This has been allotted an address in the SFR address range. Port 0 acts as multiplexed address/data lines during external memory access, i.e. when $\overline{EA}$ is low and ALE emits a valid signal. In case of controllers with on-chip EPROM, Port 0 receives code bytes during programming of the internal EPROM.

Port 1 (P1.0-P1.7) Port 1 acts as an 8-bit bidirectional bit addressable port. This has been allotted an address in the SFR address range.

Port 2 (P2.0-P2.7) Port 2 acts as 8-bit bidirectional bit addressable I/O port. It has been allotted an address in the SFR address range of 8051. During external memory accesses, port 2 emits higher eight bits of adress (A_8-A_{15}) which are valid, if ALE goes high and EA is low. P2 also receives higher order address bits during programming of the on-chip EPROM.

Port 3 (P3.0-P3.7) Port 3 is an 8-bit bidirectional bit addressable I/O port which has been allotted an address in the SFR address range of 8051. The port 3 pins also serve the alternative functions as listed in the Table 17.2.

XTAL$_1$ and XTAL$_2$ There is an inbuilt oscillator which derives the necessary clock frequency for the operation of the controller. XTAL$_1$ is the input of amplifier and XTAL$_2$ is the output of the amplifier. A crystal is to be connected externally between these two pins to complete the feedback path to start oscillations. The controller can be operated on an external clock. In this case the external clock is fed to the controller at pin XTAL$_2$ and XTAL$_1$ pin should be grounded. Commercially available versions of 8051 run on 12 MHz to 16 MHz frequency.

REGISTER SET OF 8051

8051 has two 8-bit registers, registers A and B, which can be used to store operands, as allowed by the instruction set. Internal temporary registers of 8051 are not user accessible. Including these A and B registers, 8051 has a family of special purpose registers known as, Special Function Registers (SFRs). There are, in total, 21-bit addressable, 8-bit registers. ACC (A), B, PSW, PO, PI, P2, P3, IP, IE, TCON and SCON are all 8-bit, bit-addressable registers. The remaining registers, namely, SP, DPH, DPL, TMOD, TH0, TL0, TH1, TL1, SBUF and PCON registers are to be addressed as bytes, i.e. they are not bit-addressable. The registers DPH and DPL are the higher and lower bytes of a 16-bit register DPTR, i.e. data pointer, which is used for accessing external data memory. Starting 32-bytes of on-chip RAM may be used as general purpose registers. They have been allotted addresses in the range from 0000H to 001FH. These 32, 8-bit registers are divided into four groups of 8 registers each, called register banks. At a time only one of these four groups, i.e. banks can be accessed. The register bank to be accessed can be selected using the RS1 and RS0 bits of an internal register called program status word.

The registers TH0 and TL0 form a 16-bit counter/timer register with H indicating the upper byte and L indicating the lower byte of the 16-bit timer register T0. Similarly, TH1 and TLl form the 16-bit count for the timer Tl. The four port latches are represented by P0, P1, P2 and P3. Any communication with these ports is established using the SFR addresses to these registers. Register SP is a stack pointer register. Register PSW is a flag register and contains status information. Register IP can be programmed to control the interrupt priority. Register IE can be programmed to control interrupts, i.e. enable or disable the interrupts. TCON is called timer/counter control register. Some of the bits of this register are used to turn the timers on or off. This register also contains interrupt control flags for external interrupts $\overline{INT_0}$ and $\overline{INT_1}$. The register TMOD is used for programming the modes of operation of the timers/counters. The SCON register is a serial port mode control register and is used to control the operation of the serial port. The SBUF register acts as a serial data buffer for transmit and receive operations. The PCON register is called power control register. This register contains power down bit and idle bit which activate the power down mode and idle mode in 80C51BH. There are two power saving modes of operation provided in the CHMOS version, namely, **idle mode and power down mode.**

In the **idle mode,** the oscillator continues to run and the interrupt, serial port and timer blocks are active but the clock to the CPU is disabled. The CPU status is preserved. This mode can be terminated with a hardware interrupt or hardware reset signal. After this, the CPU resumes program execution from where it left off.

In **power down mode,** the on-chip oscillator is stopped. All the functions of the controller are held maintaining the contents of RAM. The only way to terminate this mode is hardware reset. The reset redefines all the SFRs but the RAM contents are left unchanged. Both of these modes can be entered by setting the respective bit in an internal register called PCON register using software.

All these registers are listed in Table 17.3 along with their SFR addresses and contents after reset.

IMPORTANT OPERATIONAL FEATURES OF 8051

This section describes the critical special function register formats of 8051.

1. **Program Status Word (PSW)**

This bit-addressable register has the following format as shown in Fig. 17.4. The bit descriptions are presented along with the format.

2. **Timer Mode Control Register (TMOD)**

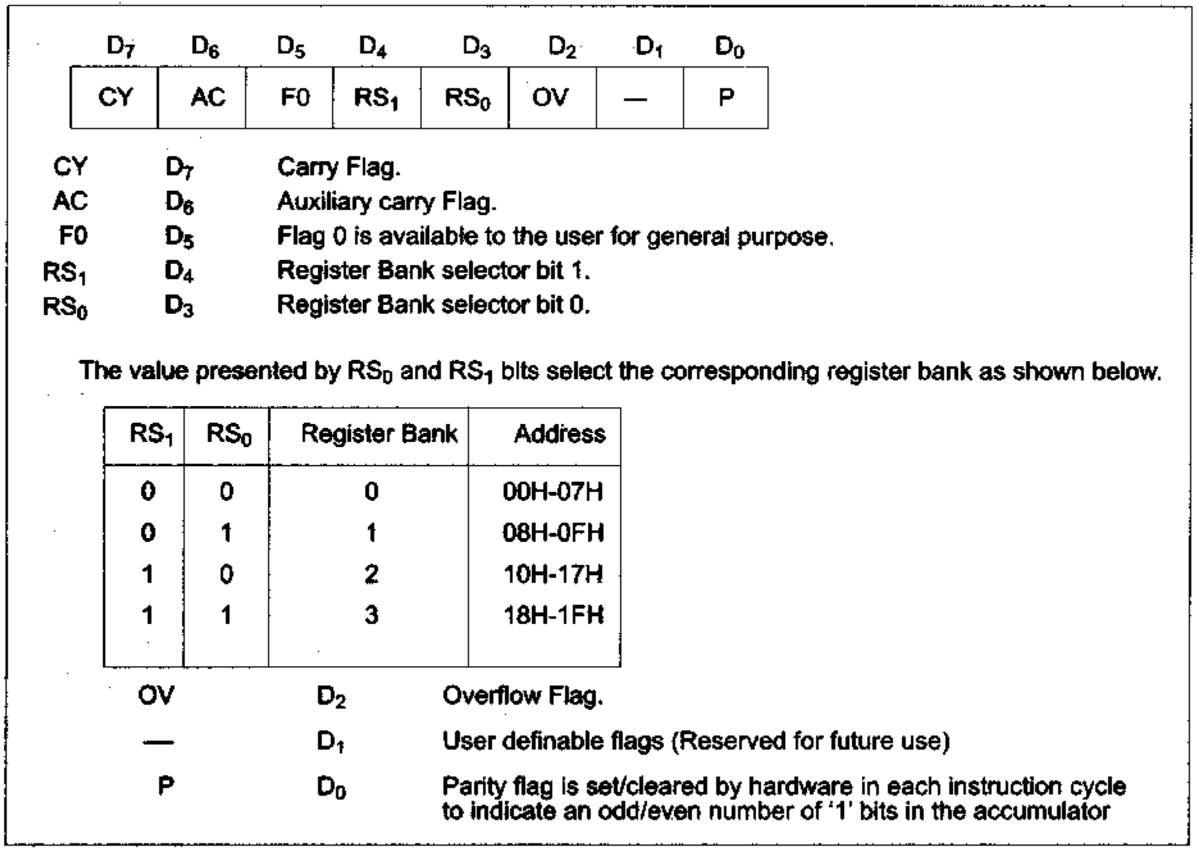
Format of this 8-bit non-bit-addressable register is shown along with its bit descriptions in Fig. 5.

3. **Timer Control Register (TCON)**

This bit-addressable register format along with its bit definitions is shown in Fig..

4. **Serial Ports Control Register (SCON)**

This 8-bit, bit-addressable register format is shown in Fig.



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CY	AC	F0	RS_1	RS_0	OV	—	P

CY	D_7	Carry Flag.
AC	D_6	Auxiliary carry Flag.
F0	D_5	Flag 0 is available to the user for general purpose.
RS_1	D_4	Register Bank selector bit 1.
RS_0	D_3	Register Bank selector bit 0.

The value presented by RS_0 and RS_1 bits select the corresponding register bank as shown below.

RS_1	RS_0	Register Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

OV	D_2	Overflow Flag.
—	D_1	User definable flags (Reserved for future use)
P	D_0	Parity flag is set/cleared by hardware in each instruction cycle to indicate an odd/even number of '1' bits in the accumulator

Fig. 17.4 *Format of PSW (Intel Corp.)*

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0
TIMER 1				TIMER 0			

GATE	When TRx (in TCON) is set and GATE = 1 TIMER/COUNTERx will run only while INTx pin is high (hardware control). When GATE = 0, TIMER/COUNTERx will run only while TRx = 1 (software control).
$C/\overline{T}$	Timer or Counter selector is cleared for Timer operation (input from internal system clock) and is set for Counter operation (input from Tx input pin).
M1	Mode selector bit.
M0	Mode selector bit.

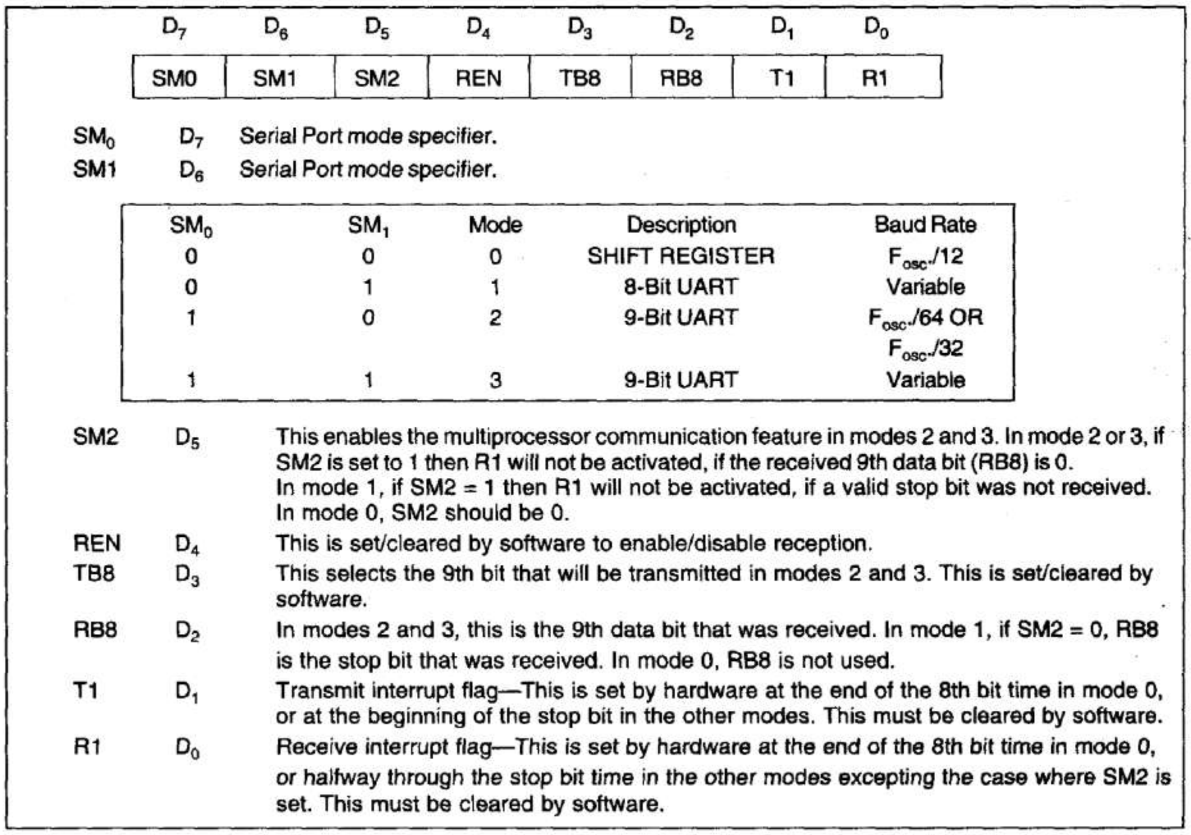
M1	M0	Operating Modes	
0	0	0	13-bit Timer (MCB-48 compatible)
0	1	1	16-bit Timer/Counter
1	0	2	8-bit Auto-Reload Timer/Counter
1	1	3	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits, TH0 is an 8-bit Timer and is controlled by Timer 1 control bits.
1	1	3	(Timer 1) Timer/Counter 1 stopped.

Fig. 17.5 *Format of TMOD Register (Intel Corp.)*

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	D_7	Timer 1 overflow flag—This is set by hardware when the Timer/Counter1 overflows, and is cleared by hardware as processor vectors to the interrupt service routine.
TR1	D_6	Timer 1 run control bit—This is set/cleared by software to turn Timer/Counter1 on/off.
TF0	D_5	Timer 0 overflow flag—This is set by hardware when the Timer/Counter 0 overflows, and is cleared by hardware as processor vectors to the service routine.
TR0	D_4	Timer 0 run control bit—This is set/cleared by software to turn Timer/Counter1 on/off.
IE1	D_3	External Interrupt1 edge flag—This is set by hardware when external interrupt edge is detected, and is cleared by hardware when the interrupt is processed.
IT1	D_2	Interrupt1 type control bit—This is set/cleared by software to specify falling edge/low level triggered external Interrupt.
IE0	D_1	External Interrupt0 edge flag—This is set by hardware when external Interrupt edge is detected, and is cleared by hardware when the interrupt is processed.
IT0	D_0	Interrupt0 type control bit—This is set/cleared by software to specify falling edge/low level triggered external Interrupt.

Fig. 17.6 *Format of TCON Register (Intel Corp.)*



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SM0	SM1	SM2	REN	TB8	RB8	T1	R1

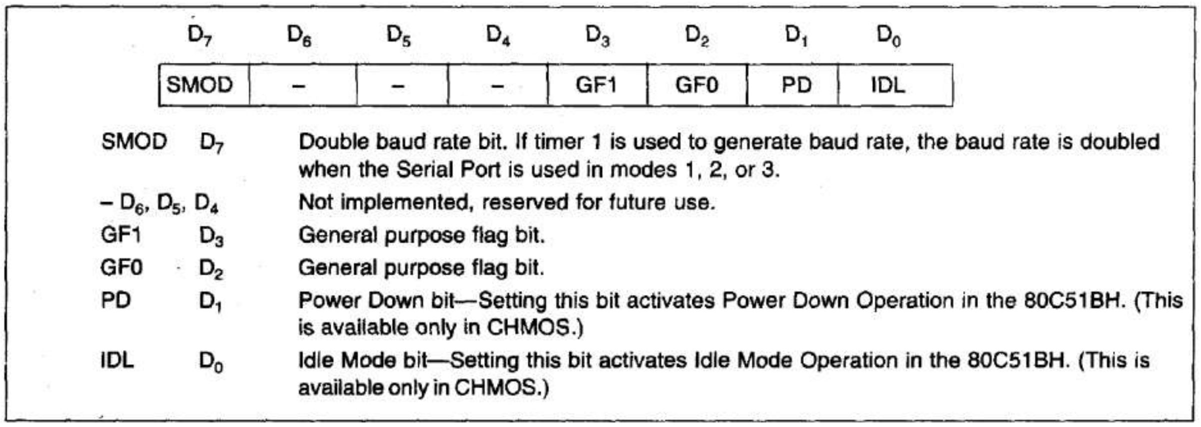
SM_0	D_7	Serial Port mode specifier.
SM1	D_6	Serial Port mode specifier.

SM_0	SM_1	Mode	Description	Baud Rate
0	0	0	SHIFT REGISTER	F_{osc}./12
0	1	1	8-Bit UART	Variable
1	0	2	9-Bit UART	F_{osc}./64 OR F_{osc}./32
1	1	3	9-Bit UART	Variable

SM2	D_5	This enables the multiprocessor communication feature in modes 2 and 3. In mode 2 or 3, if SM2 is set to 1 then R1 will not be activated, if the received 9th data bit (RB8) is 0. In mode 1, if SM2 = 1 then R1 will not be activated, if a valid stop bit was not received. In mode 0, SM2 should be 0.
REN	D_4	This is set/cleared by software to enable/disable reception.
TB8	D_3	This selects the 9th bit that will be transmitted in modes 2 and 3. This is set/cleared by software.
RB8	D_2	In modes 2 and 3, this is the 9th data bit that was received. In mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used.
T1	D_1	Transmit interrupt flag—This is set by hardware at the end of the 8th bit time in mode 0, or at the beginning of the stop bit in the other modes. This must be cleared by software.
R1	D_0	Receive interrupt flag—This is set by hardware at the end of the 8th bit time in mode 0, or halfway through the stop bit time in the other modes excepting the case where SM2 is set. This must be cleared by software.

Fig. 17.7 *Format of SCON Register (Intel Corp.)*

Power Control Register (PCON)
The format of this non-bit-addressable register is shown in Fig. 17.8.



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
SMOD	–	–	–	GF1	GF0	PD	IDL

SMOD	D_7	Double baud rate bit. If timer 1 is used to generate baud rate, the baud rate is doubled when the Serial Port is used in modes 1, 2, or 3.
– D_6, D_5, D_4		Not implemented, reserved for future use.
GF1	D_3	General purpose flag bit.
GF0	D_2	General purpose flag bit.
PD	D_1	Power Down bit—Setting this bit activates Power Down Operation in the 80C51BH. (This is available only in CHMOS.)
IDL	D_0	Idle Mode bit—Setting this bit activates Idle Mode Operation in the 80C51BH. (This is available only in CHMOS.)

Fig. 17.8 *Format of PCON Register (Intel Corp.)*

MEMORY AND I/O ADDRESSING BY 8051

1. **Memory Addressing**

The total memory of an 8051 system is logically divided into program memory and data memory. Program memory stores the programs to be executed, while data memory stores the data like intermediate results, variables and constants required for the execution of the program. Program memory is invariably implemented using EPROM, because it stores only program code which is to be executed and thus it need not be written into. However, the data memory may be read from or written to and thus it is implemented using RAM.

Further, the program memory and data memory both may be categorized as on-chip (internal) and external memory, depending upon whether the memory physically exists on the chip or it is externally interfaced. The 8051 can address 4 Kbytes on-chip program memory whose map starts from 0000H and ends at 0FFFH. It can address 64 Kbytes of external program memory under the control of $\overline{PSEN}$ signal, whose address map is from 0000H to FFFFH. Here, one may note that the map of internal program memory overlaps with that of the external program memory. However, these two memory spaces can be distinguished using the PSEN signal. In case of ROM-less versions of 8051, the $\overline{PSEN}$ signal is used to access the external program memory. Conceptually this is shown in Fig. 17.9.

8051 supports 64 Kbytes of external data memory whose map starts at 0000H and ends at FFFFH. This external data memory can be accessed under the control of register DPTR, which stores the addresses for external data memory accesses. 8051 generates $\overline{RD}$ and $\overline{WR}$ signals during external data memory accesses. The chip select line of the external data memory may be derived from the address lines as in the case of other microprocessors. Internal data memory of 8051 consists of two parts; the first is the RAM block of 128 bytes (256 bytes in case of some versions of 8051) and the second is the set of addresses from 80H to FFH, which includes the addresses allotted to the special function registers.

The address map of the 8051 internal RAM (128 bytes) starts from 00 and ends at 7FH. This RAM can be addressed by using direct or indirect mode of addressing. However, the special function register address map, i.e. from 80H to FFH is accessible only with direct addressing mode.

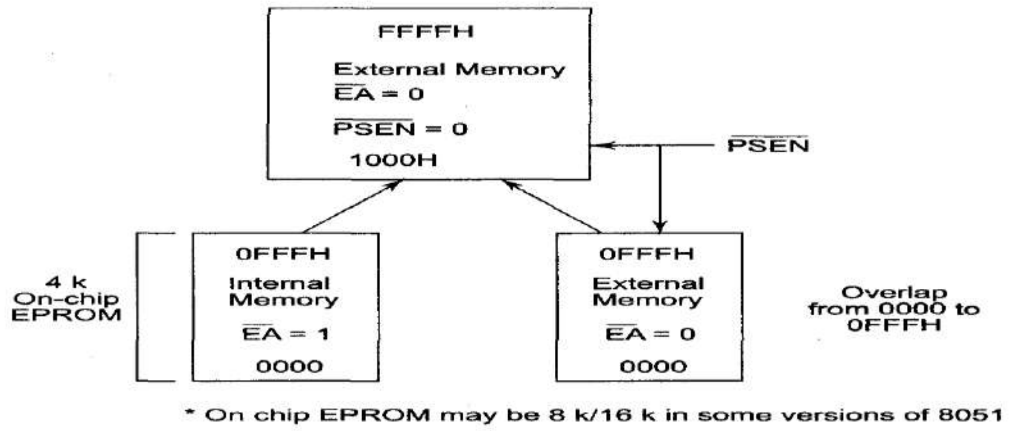


Fig. 17.9 *Program Memory Map of an 8051 System*

In case of 8051 versions with 256 bytes on-chip RAM, the map starts from 00H and ends at FFH. In this case, it may be noted that the address map of special function registers, i.e. 80H to FFH overlaps with the upper 128 bytes of RAM. However, the way of addressing, i.e. addressing mode, differentiates between these two memory spaces. The upper 128 bytes of the 256 byte on-chip RAM can be accessed only using indirect addressing, while the lower 128 bytes can be accessed using direct or indirect mode of addressing. The special function register address space can only be accessed using direct addressing. The address map of the internal RAM and SFR is shown in Fig. 17.10.

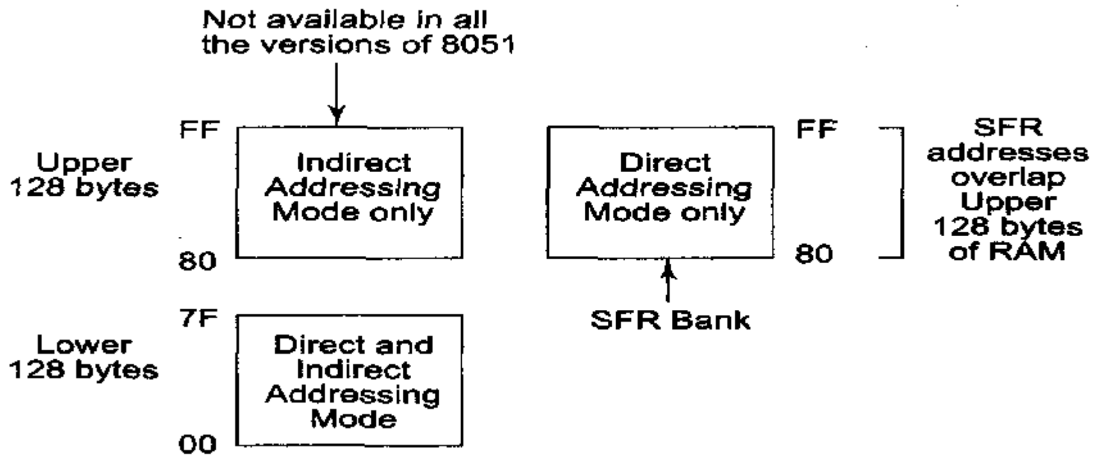

Not available in all the versions of 8051
Upper 128 bytes
FF
Indirect Addressing Mode only
80
7F
Lower 128 bytes
Direct and Indirect Addressing Mode
00
Direct Addressing Mode only
FF
80
SFR Bank
SFR addresses overlap Upper 128 bytes of RAM

Fig. 17.10 *Internal Data Memory of 8051*

The lower 128 bytes of RAM whose address map is from 00 to 7FH is functionally organized in three sections. The address block from 00 to 1FH, i.e. the lowest 32 bytes which form the first section, is divided into four banks of 8-bit registers, denoted as bank 00, 01, 10 and 11. Each of these banks contains eight 8-bit registers. The stack pointer gets initialized at address 07H, i.e. the last address of the bank 00, after reset operation. After reset bank 0 is selected by default but the actual stack data is stored from 08H onwards, i.e. bank 01, 10 and 11. These bank addressing bits of the register banks are present in PSW, to select one of these banks at a time. The second section extends from 20H to 2FH, i.e. 16 bytes, which is a bit-addressable block of memory, containing 16 x 8 = 128 bits. Each of these bits can be addressed using the addresses 00 to 7FH. Any of these bits can be accessed in two ways. In the first, its bit number is directly mentioned in the instruction while in the second the bit is mentioned with its position in the respective register byte. For example, the bits 0 to 7 can be referred directly by their numbers, i.e. 0 to 7 or using the notations 20.0 to 20.7 respectively. Note that 20 is the address of the first byte of the on-chip RAM. The third block of internal memory occupies addresses from 30H to 7FH. This block of memory is a byte addressable memory space. In general, this third block of memory is used as stack memory. All the internal data memory locations are accessed using 8-bit addresses under appropriate modes of addressing. Figure 17.11 shows the categorization of 128 bytes of internal RAM into the different sections.

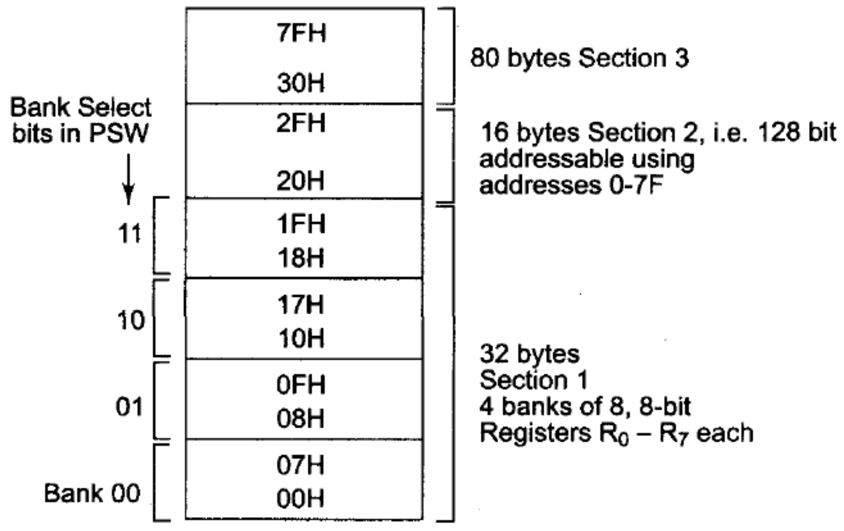

7FH
30H
80 bytes Section 3
Bank Select bits in PSW
2FH
20H
16 bytes Section 2, i.e. 128 bit addressable using addresses 0-7F
11
1FH
18H
10
17H
10H
01
0FH
08H
Bank 00
07H
00H
32 bytes Section 1 4 banks of 8, 8-bit Registers $R_0 - R_7$ each

Fig. 17.11 *Functional Description of Internal Lower 128 Bytes of RAM*

2. **I/O Addressing**

Internally, 8051 has two timers, one serial input/output port and four 8-bit, bit-addressable ports. Some complex applications may require additional I/O devices to be interfaced with 8051. Such external I/O devices are interfaced with 8051 as external memory-mapped devices. In other words, the devices are treated as external memory locations, and they consume external memory addresses. Figure 17.12 shows a system that has external RAM memory of 16 Kbytes, ROM of 16 Kbytes and one chip of 8255 interfaced externally to an 8051 family microcontroller.

Note that, the maps of external program and data memory may overlap, as the memory spaces are logically separated in an 8051 system. As the 8255 is interfaced in external data memory space its addresses are of 16-bits.

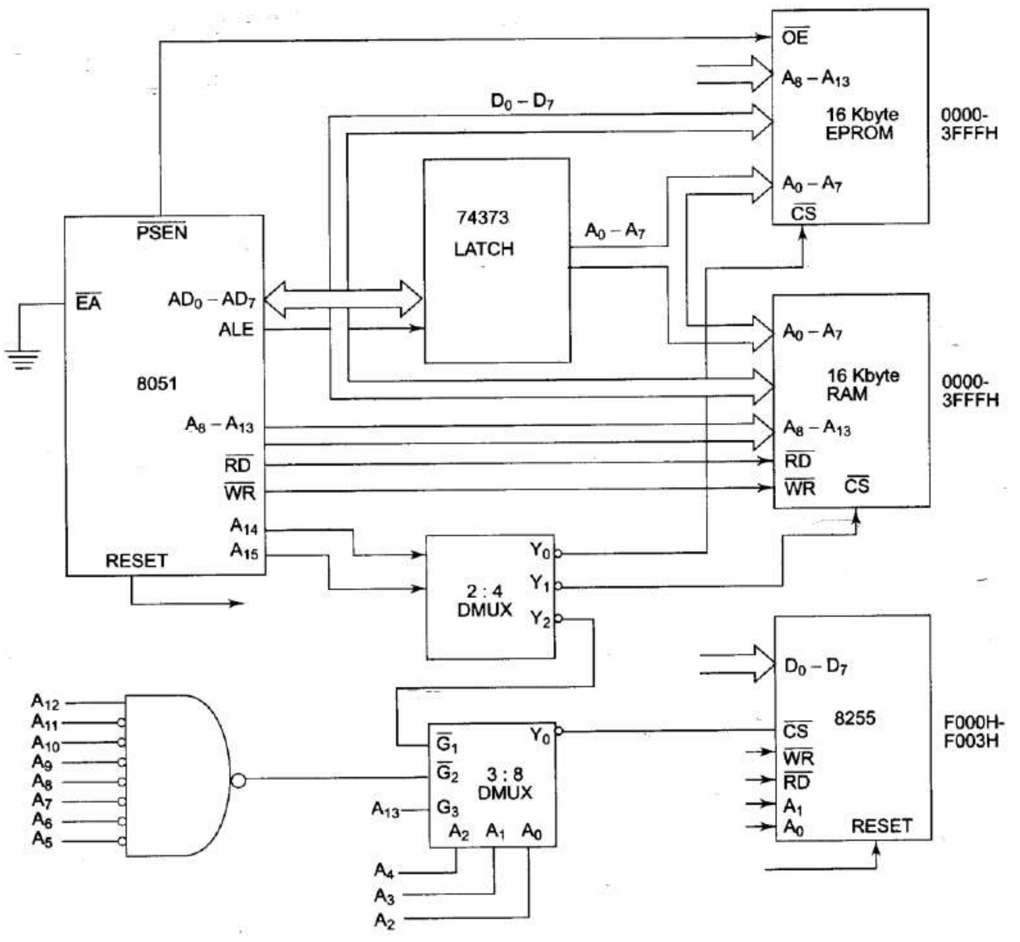


Fig. 17.12 *Interfacing External Memory and I/O with 8051*

ADRESSING MODES OF 8051

ACCESSING MEMORY USING VARIOUS ADDRESSING MODES

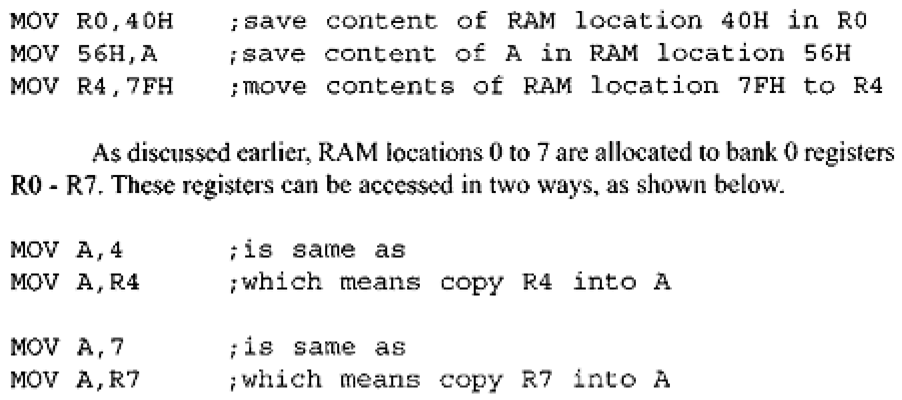
We can use direct or register indirect addressing modes to access data stored either in RAM or registers of the 8051. This topic will be discussed thoroughly in this section. We will also show how to access on-chip ROM containing data using indexed addressing mode.

Direct addressing mode

As mentioned in Chapter 2, there are 128 bytes of RAM in the 8051. The RAM has been assigned addresses 00 to 7FH. The following is a summary of the allocation of these 128 bytes.

1. RAM locations 00 - 1FH are assigned to the register banks and stack.
1. RAM locations 20 - 2FH are set aside as bit-addressable space to save single-bit data. This is discussed in Section 5.3.
2. RAM locations 30 - 7FH are available as a place to save byte-sized data.

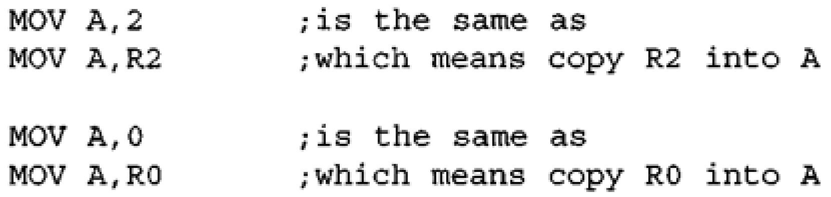
Although the entire 128 bytes of RAM can be accessed using direct addressing mode, it is most often used to access RAM locations 30 - 7FH. This is due to the fact that register bank locations are accessed by the register names of RO - R7, but there is no such name for other RAM locations. In the direct addressing mode, the data is in a RAM memory location whose address is known, and this address is given as a part of the instruction. Contrast this with immediate addressing mode, in which the operand itself is provided with the instruction. The "#" sign distinguishes between the two modes. See the examples below, and note the absence of the "#" sign.



```
MOV R0,40H    ;save content of RAM location 40H in R0
MOV 56H,A     ;save content of A in RAM location 56H
MOV R4,7FH    ;move contents of RAM location 7FH to R4
```

As discussed earlier, RAM locations 0 to 7 are allocated to bank 0 registers R0 - R7. These registers can be accessed in two ways, as shown below.

```
MOV A,4       ;is same as
MOV A,R4      ;which means copy R4 into A

MOV A,7       ;is same as
MOV A,R7      ;which means copy R7 into A
```



```
MOV A,2       ;is the same as
MOV A,R2      ;which means copy R2 into A

MOV A,0       ;is the same as
MOV A,R0      ;which means copy R0 into A
```

The above examples should reinforce the importance of the "#" sign in 8051 instructions. See the following code.

```
MOV R2,#5  ;R2 with value 5
MOV A,2    ;copy R2 to A (A=R2=05)
MOV B,2    ;copy R2 to B (B=R2=05)
MOV 7,2    ;copy R2 to R7
           ;since "MOV R7,R2" is invalid
```

Although it is easier to use the names RQ - R7 than their memory addresses, RAM locations 3 OH to 7FH cannot be accessed in any way other than by their addresses since they have no names.

SFR registers and their addresses

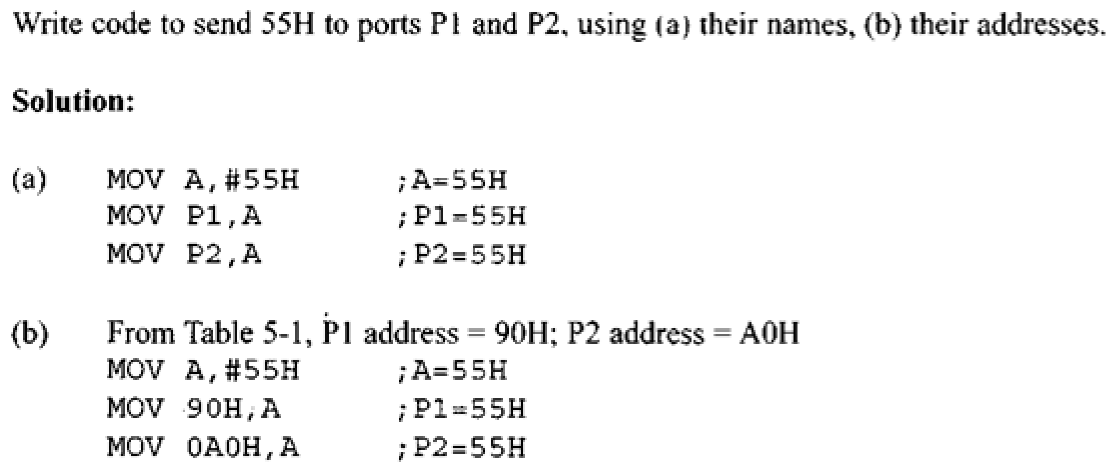
Among the registers we have discussed so far, we have seen that RO - R7 are part of the 128 bytes of RAM memory. What about registers A, B, PSW, and DPTR? Do they also have addresses? The answer is yes. In the 8051, registers A, B, PSW, and DPTR are part of the group of registers commonly referred to as SFR (special function registers). There are many special function registers and they are widely used, as we will discuss in future chapters. The SFR can be accessed by their names (which is much easier) or by their addresses. For example, register A has address EOH, and register B has been designated the address FOH, as shown in Table 5-1. Notice how the following pairs of instructions mean the same thing.

```
MOV 0E0H,#55H  ;is the same as
MOV A,#55H     ;which means load 55H into A (A=55H)

MOV 0F0H,#25H  ;is the same as
MOV B,#25H     ;which means load 25H into B (B=25H)

MOV 0E0H,R2    ;is the same as
MOV A,R2       ;which means copy R2 into A

MOV 0F0H,R0    ;is the same as
MOV B,R0       ;which means copy R0 into B

MOV P1, A      ;is the same as
MOV 90H,A      ;which means copy reg A to P1
```

Table lists the 8051 special function registers (SFR) and their addresses. The following two points should be noted about the SFR addresses.

1. The special function registers have addresses between 80H and FFH. These addresses are above 80H, since the addresses 00 to 7FH are addresses of RAM memory inside the 8051.
2. Not all the address space of 80 to FF is used by the SFR. The unused locations 80H to FFH are reserved and must not be used by the 8051 programmer.

Regarding direct addressing mode, notice the following two points: (a) the address value is limited to one byte, 00 - FFH, which means this addressing mode is limited to accessing RAM locations and registers located inside the 8051. (b) if you examine the 1st file for an Assembly language program, you will see that the SFR registers' names are replaced with their addresses as listed in Table 5-1.

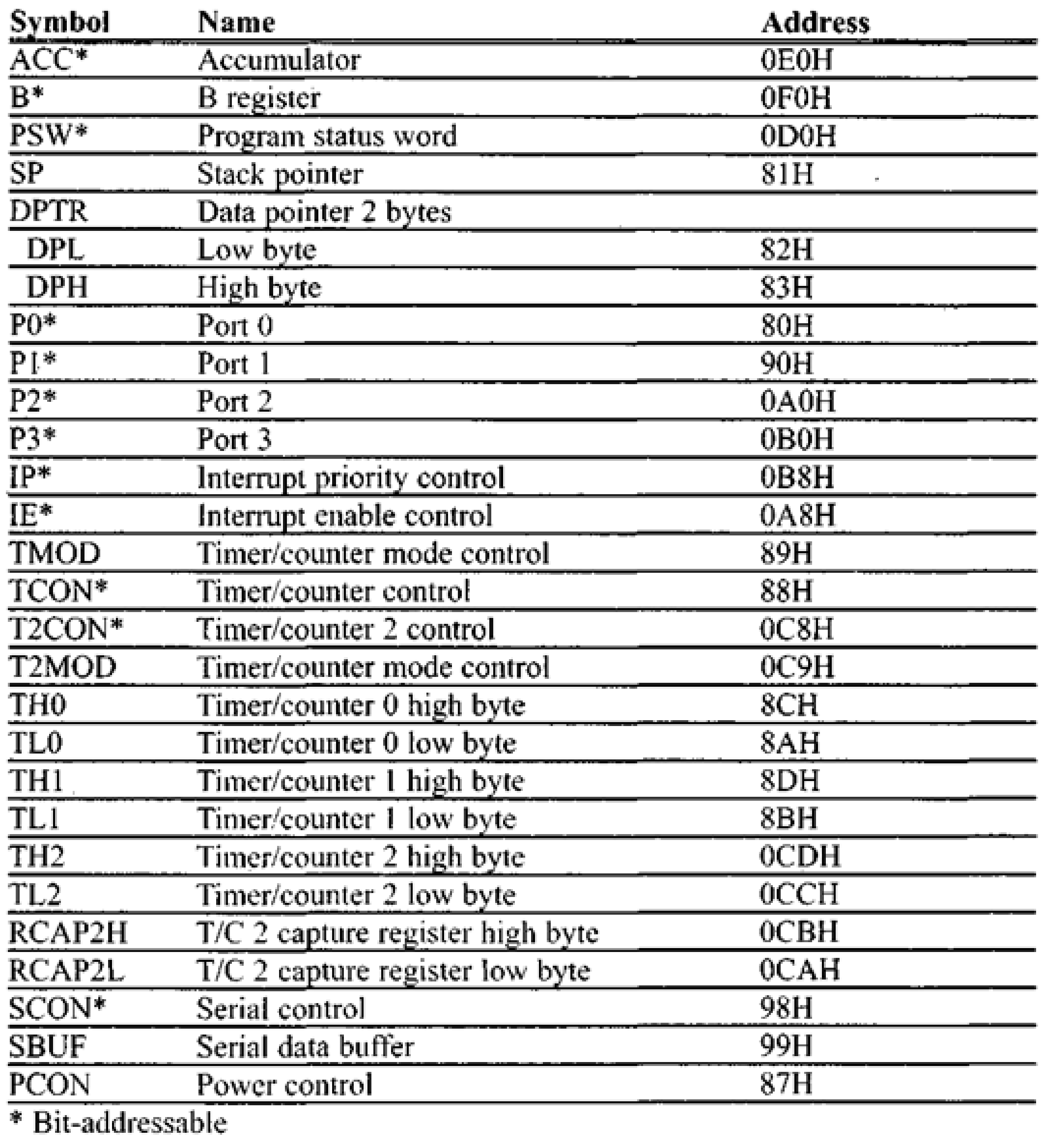

Write code to send 55H to ports P1 and P2, using (a) their names, (b) their addresses.

Solution:

(a)
```
MOV A,#55H        ;A=55H
MOV P1,A          ;P1=55H
MOV P2,A          ;P2=55H
```

(b) From Table 5-1, P1 address = 90H; P2 address = A0H
```
MOV A,#55H        ;A=55H
MOV 90H,A         ;P1=55H
MOV 0A0H,A        ;P2=55H
```

Table : 8051 Special Function Register (SFR) Addresses



Symbol	Name	Address
ACC*	Accumulator	0E0H
B*	B register	0F0H
PSW*	Program status word	0D0H
SP	Stack pointer	81H
DPTR	Data pointer 2 bytes	
DPL	Low byte	82H
DPH	High byte	83H
P0*	Port 0	80H
P1*	Port 1	90H
P2*	Port 2	0A0H
P3*	Port 3	0B0H
IP*	Interrupt priority control	0B8H
IE*	Interrupt enable control	0A8H
TMOD	Timer/counter mode control	89H
TCON*	Timer/counter control	88H
T2CON*	Timer/counter 2 control	0C8H
T2MOD	Timer/counter mode control	0C9H
TH0	Timer/counter 0 high byte	8CH
TL0	Timer/counter 0 low byte	8AH
TH1	Timer/counter 1 high byte	8DH
TL1	Timer/counter 1 low byte	8BH
TH2	Timer/counter 2 high byte	0CDH
TL2	Timer/counter 2 low byte	0CCH
RCAP2H	T/C 2 capture register high byte	0CBH
RCAP2L	T/C 2 capture register low byte	0CAH
SCON*	Serial control	98H
SBUF	Serial data buffer	99H
PCON	Power control	87H

* Bit-addressable

Example 1

Stack and direct addressing mode

Another major use of direct addressing mode is the stack. In the 8051 family, only direct addressing mode is allowed for pushing onto the stack. Therefore, an instruction such as "PUSH A" is invalid. Pushing the accumulator onto the stack must be coded as "PUSH OEOH" where OEOH is the address of register A. Similarly, pushing R3 of bank 0 is coded as "PUSH 03". Direct addressing mode must be used for the POP instruction as well. For example, "POP 04" will pop the top of the stack into R4 of bank 0.

Example 2

Show the code to push R5, R6, and A onto the stack and then pop them back them into R2, R3, and B, where register B = register A, R2 = R6, and R3 = R5.

Solution:

```
PUSH 05         ;push R5 onto stack
PUSH 06         ;push R6 onto stack
PUSH 0E0H       ;push register A onto stack
POP  0F0H       ;pop top of stack into register B
                ;now register B = register A
POP  02         ;pop top of stack into R2
                ;now R2 = R6
POP  03         ;pop top of stack into R3
                ;now R3 = R5
```

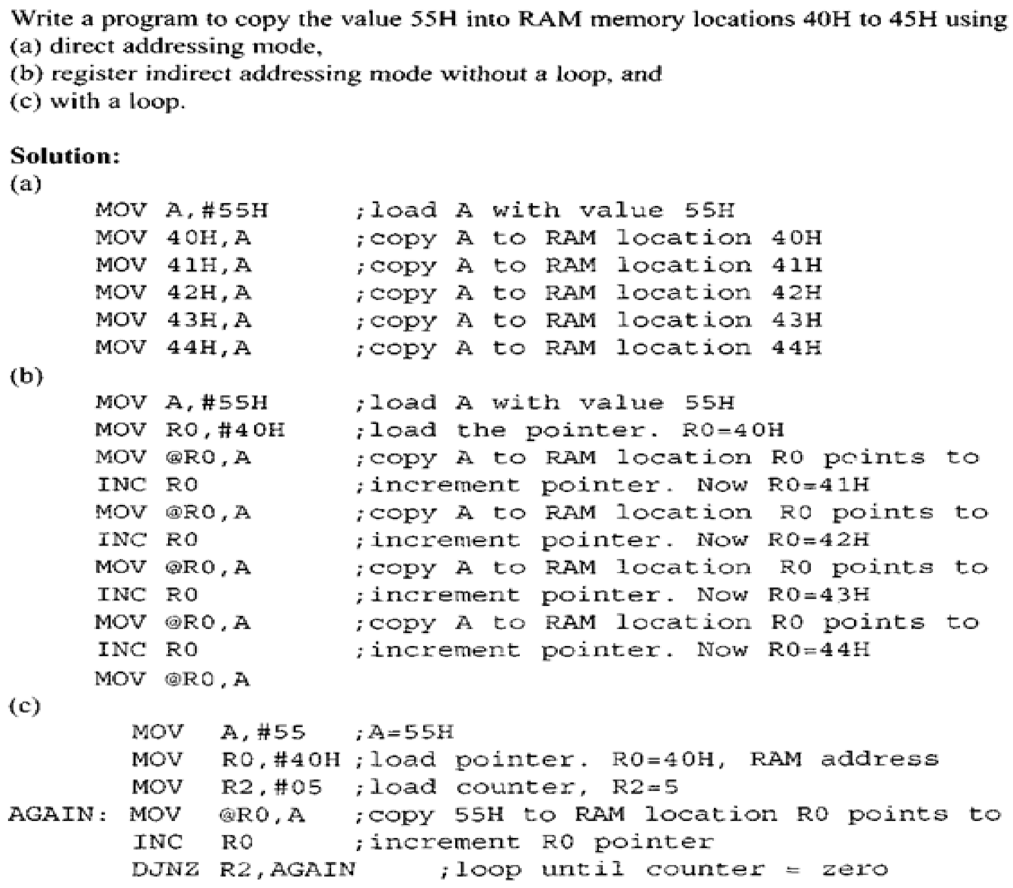
Register indirect addressing mode

In the register indirect addressing mode, a register is used as a pointer to the data. If the data is inside the CPU, only registers RO and Rl are used for this purpose. In other words, R2 - R7 cannot be used to hold the address of an operand located in RAM when using this addressing mode. When RO and Rl are used as **pointers, that is, when they hold the addresses of RAM locations, they must be preceded by the "@" sign, as shown below.**

```
MOV A,@R0 ;move contents of RAM location whose
          ;address is held by R0 into A
MOV @R1,B ;move contents of B into RAM location
          ;whose address is held by R1
```

Notice that RO (as well as Rl) is preceded by the "@" sign. In the absence of the "@" sign, MOV will be interpreted as an instruction moving the contents of register RO to A, instead of the contents of the memory location pointed to by RO.

Example 3



Write a program to copy the value 55H into RAM memory locations 40H to 45H using
(a) direct addressing mode,
(b) register indirect addressing mode without a loop, and
(c) with a loop.

Solution:

(a)

```
      MOV A,#55H      ;load A with value 55H
      MOV 40H,A       ;copy A to RAM location 40H
      MOV 41H,A       ;copy A to RAM location 41H
      MOV 42H,A       ;copy A to RAM location 42H
      MOV 43H,A       ;copy A to RAM location 43H
      MOV 44H,A       ;copy A to RAM location 44H
```

(b)

```
      MOV A,#55H      ;load A with value 55H
      MOV R0,#40H     ;load the pointer. R0=40H
      MOV @R0,A       ;copy A to RAM location R0 points to
      INC R0          ;increment pointer. Now R0=41H
      MOV @R0,A       ;copy A to RAM location  R0 points to
      INC R0          ;increment pointer. Now R0=42H
      MOV @R0,A       ;copy A to RAM location  R0 points to
      INC R0          ;increment pointer. Now R0=43H
      MOV @R0,A       ;copy A to RAM location R0 points to
      INC R0          ;increment pointer. Now R0=44H
      MOV @R0,A
```

(c)

```
        MOV   A,#55   ;A=55H
        MOV   R0,#40H ;load pointer. R0=40H, RAM address
        MOV   R2,#05  ;load counter, R2=5
AGAIN:  MOV   @R0,A   ;copy 55H to RAM location R0 points to
        INC   R0      ;increment R0 pointer
        DJNZ  R2,AGAIN     ;loop until counter = zero
```

Advantage of register indirect addressing mode

One of the advantages of register indirect addressing mode is that it makes accessing data dynamic rather than static as in the case of direct addressing mode. Example 5-3 shows two cases of copying 55H into RAM locations 40H to 45H. Notice in solution (b) that there are two instructions that are repeated numerous times. We can create a loop with those two instructions as shown in solution (c). Solution (c) is the most efficient and is possible only because of register indirect addressing mode. Looping is not possible in direct addressing mode. This is the main difference between the direct and register indirect addressing modes.

Example 5-4

Write a program to clear 16 RAM locations starting at RAM address 60H.

Solution:

```
       CLR     A          ;A=0
       MOV     R1,#60H    ;load pointer. R1=60H
       MOV     R7,#16     ;load counter, R7=16 (10 in hex)
AGAIN: MOV     @R1,A      ;clear RAM location R1 points to
       INC     R1         ;increment R1 pointer
       DJNZ    R7,AGAIN   ;loop until counter = zero
```

An example of how to use both RO and Rl in the register indirect addressing mode in a block transfer is given in Example 5.

Example 5-5

Write a program to copy a block of 10 bytes of data from RAM locations starting at 35H to RAM locations starting at 60H.

Solution:

```
          MOV  R0,#35H   ;source pointer
          MOV  R1,#60H   ;destination pointer
          MOV  R3,#10    ;counter
BACK:     MOV  A,@R0     ;get a byte from source
          MOV  @R1,A     ;copy it to destination
          INC  R0        ;increment source pointer
          INC  R1        ;increment destination pointer
          DJNZ R3,BACK   ;keep doing it for all ten bytes
```

Limitation of register indirect addressing mode in the 8051

As stated earlier, RO and Rl are the only registers that can be used fo^r pointers in register indirect addressing mode. Since RO and Rl are 8 bits wide, their use is limited to accessing any information in the internal RAM (scratch pad memory of 30H - 7FH, or SFR). However, there are times when we need to access data stored in external RAM or in the code space of on-chip ROM. Whether accessing externally connected RAM or on-chip ROM, we need a 16-bit pointer. In such cases, the DPTR register is used, as shown next.

Indexed addressing mode and on-chip ROM access

Indexed addressing mode is widely used in accessing data elements of look-up table entries located in the program ROM space of the 8051. The instruction used for this purpose is "MOVC A, @A+DPTR". The 16-bit register DPTR and register A are used to form the address of the data element stored in on-chip ROM. Because the data elements are stored in the program (code) space ROM of the 8051, the instruction MOVC is used instead of MOV. The "C" means code. In this instruction the contents of A are added to the 16-bit register DPTR to form the 16-bit address of the needed data. See Example 5-6.

Example 6

In this program, assume that the word "USA" is burned into ROM locations starting at 200H, and that the program is burned into ROM locations starting at 0. Analyze how the program works and state where "USA" is stored after this program is run.

Solution:

```
      ORG   0000H             ;burn into ROM starting at 0
      MOV   DPTR,#200H        ;DPTR=200H look-up table address
      CLR   A                 ;clear A(A=0)
      MOVC  A,@A+DPTR         ;get the char from code space
      MOV   R0,A              ;save it in R0
      INC   DPTR              ;DPTR=201 pointing to next char
      CLR   A                 ;clear A(A=0)
      MOVC  A,@A+DPTR         ;get the next char
      MOV   R1,A              ;save it in R1
      INC   DPTR              ;DPTR=202 pointing to next char
      CLR   A                 ;clear A(A=0)
      MOVC  A,@A+DPTR         ;get the next char
      MOV   R2,A              ;save it in R2
HERE:SJMP   HERE              ;stay here

;Data is burned into code space starting at 200H
            ORG 200H
MYDATA:     DB "USA"
            END               ;end of program
```

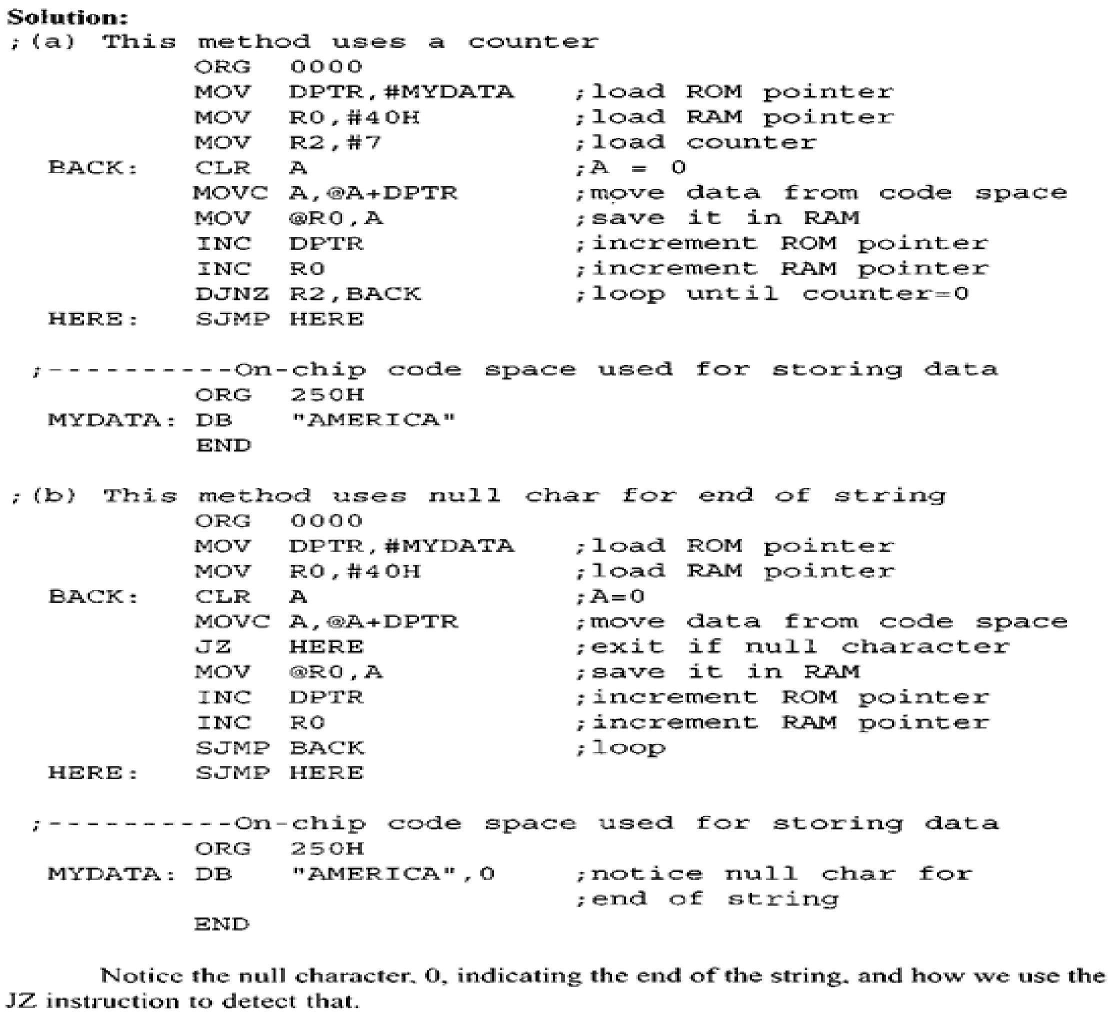
In the above program ROM locations 200H - 202H have the following contents. 200=('U') 201=('S') 202=('A")

We start with DPTR = 200H, and A = 0. The instruction "MOVC A, @A+DPTR" moves the contents of ROM location 200H (200H + 0 = 200H) to register A. Register A contains 55H, the ASCII value for "U". This is moved to RO. Next, DPTR is incremented to make DPTR = 201H. A is set to 0 again to get the contents of the next ROM location

201H, which holds character "S". After this program is run, we have RO = 55H, Rl = 53H, and R2 = 41H, the ASCII values for the characters "U", "S" and "A".

Example 7

Assuming that ROM space starting at 250H contains "America", write a program to transfer the bytes into RAM locations starting at 40H.



Solution:

```
;(a) This method uses a counter
          ORG   0000
          MOV   DPTR,#MYDATA     ;load ROM pointer
          MOV   R0,#40H          ;load RAM pointer
          MOV   R2,#7            ;load counter
  BACK:   CLR   A                ;A = 0
          MOVC  A,@A+DPTR        ;move data from code space
          MOV   @R0,A            ;save it in RAM
          INC   DPTR             ;increment ROM pointer
          INC   R0               ;increment RAM pointer
          DJNZ  R2,BACK          ;loop until counter=0
  HERE:   SJMP  HERE

 ;----------On-chip code space used for storing data
          ORG   250H
  MYDATA: DB    "AMERICA"
          END

;(b) This method uses null char for end of string
          ORG   0000
          MOV   DPTR,#MYDATA     ;load ROM pointer
          MOV   R0,#40H          ;load RAM pointer
  BACK:   CLR   A                ;A=0
          MOVC  A,@A+DPTR        ;move data from code space
          JZ    HERE             ;exit if null character
          MOV   @R0,A            ;save it in RAM
          INC   DPTR             ;increment ROM pointer
          INC   R0               ;increment RAM pointer
          SJMP  BACK             ;loop
  HERE:   SJMP  HERE

 ;----------On-chip code space used for storing data
          ORG   250H
  MYDATA: DB    "AMERICA",0      ;notice null char for
                                 ;end of string
          END
```

Notice the null character, 0, indicating the end of the string, and how we use the JZ instruction to detect that.

Look-up table and the MOVC instruction

The look-up table is a widely used concept in microprocessor programming. It allows access to elements of a frequently used table with minimum operations.

Example 8

Write a program to get the *x* value from PI and send x^2 to P2, continuously.

Solution:

```
            ORG   0
            MOV   DPTR,#300H        ;load look-up table address
            MOV   A,#0FFH           ;A=FF
            MOV   P1,A              ;configure P1 as input port
BACK:       MOV   A,P1              ;get X
            MOVC  A,@A+DPTR         ;get X squared from table
            MOV   P2,A              ;issue it to P2
            SJMP  BACK              ;keep doing it

            ORG   300H
XSQR_TABLE:
            DB    0,1,4,9,16,25,36,49,64,81
            END
```

Notice that the first instruction could be replaced with "MOV DPTR,#XSQR_TABLE"

Example 5-9

Answer the following questions for Example 5-8.
(a) Indicate the content of ROM locations 300 - 309H.
(b) At what ROM location is the square of 6, and what value should be there?
(c) Assume that P1 has a value of 9: What value is at P2 (in binary)?

Solution:

(a) All values are in hex.

```
      300  =  (00)       301  =  (01)    302  =  (04)    303  =  (09)
      304  =  (10)       4 × 4  =  16  =  10  in  hex
      305  =  (19)       5 × 5  =  25  =  19  in  hex
      306  =  (24)       6 × 6  =  36  =  24H
      307  =  (31)       308  =  (40)    309  =  (51)
```

(b) 306H; it is 24H

(c) 01010001B, which is 51H and 81 in decimal ($9^2 = 81$).

In addition to being used to access program ROM, DPTR can be used to access memory externally connected to the 8051. Another register used in indexed addressing mode is the program counter.

In many of the examples above, the MOV instruction was used for the sake of clarity, even though one can use any instruction as long as that instruction supports the addressing mode. For example, the instruction "ADD A, @RO" would add the contents of the memory location pointed to by RO to the contents of register A. We will see more examples of using addressing modes with various instructions in the next few chapters.

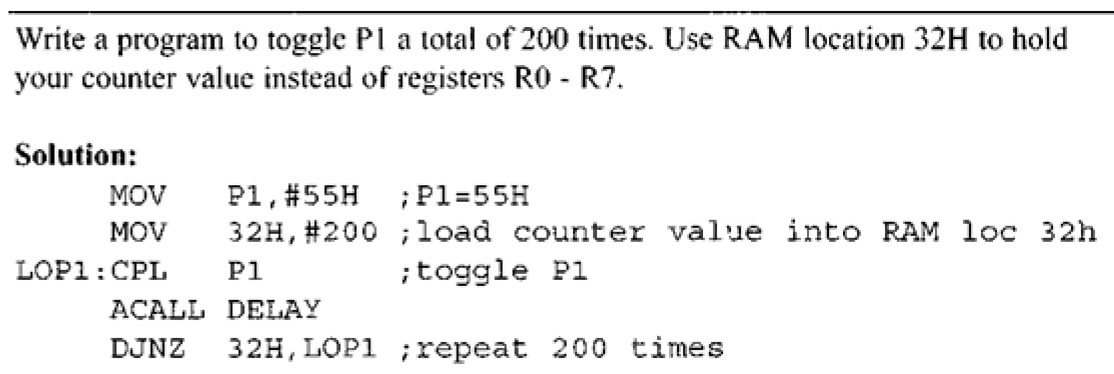
Indexed addressing mode and MOVX instruction

As we have stated earlier, the 8051 has 64K bytes of code space under the direct control of the Program Counter register. We just showed how to use the MOVC instruction to access a portion of this 64K-byte code space as data memory space. In many applications the size of program code does not leave any room to share the 64K-byte code space with data. For this reason the 8051 has another 64K bytes of memory space set aside exclusively for data storage. This data memory space is referred to as *external memory* and it is accessed only by the MOVX instruction. In other words, the 8051 has a total of 128K bytes of memory space since 64K bytes of code added to 64K bytes of data space gives us 128K bytes. One major difference between the code space and data space is that, unlike code space, the data space cannot be shared between code and data. This is such an important topic that we have dedicated an entire chapter to it: Chapter 14.

Accessing RAM Locations 30 - 7FH as scratch pad

As we have seen so far, in accessing registers RO - R7 of various banks, it is much easier to refer to them by their RO - R7 names than by their RAM locations. The only problem is that we have only 4 banks and very often the task of bank switching and keeping track of register bank usage is tedious and prone to errors. For this reason in many applications we use RAM locations 30 - 7FH as scratch pad and leave addresses 8 - 1FH for stack usage. That means that we use RO - R7 of bank 0, and if we need more registers we simply use RAM locations 30-7FH. Look at Example 5-10.

Example 10



Write a program to toggle P1 a total of 200 times. Use RAM location 32H to hold your counter value instead of registers R0 - R7.

Solution:

```
      MOV    P1,#55H   ;P1=55H
      MOV    32H,#200  ;load counter value into RAM loc 32h
LOP1:CPL    P1        ;toggle P1
      ACALL  DELAY
      DJNZ   32H,LOP1  ;repeat 200 times
```

INTERRUPTS OF 8051

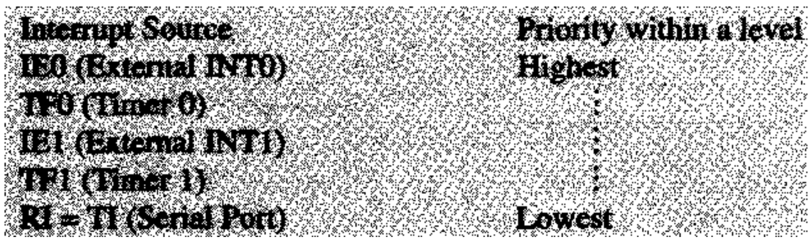
8051 provides five sources of interrupts. $\overline{INT_0}$ and $\overline{INT_1}$ are the two external interrupt inputs. These can either be edge-sensitive or level-sensitive, as programmed with bits IT_0 and IT_1 register TCON. These interrupts are processed internally by the flags IE_0 and IE_1. If the interrupts are programmed as edge-sensitive, these flags are automatically cleared after the control is transferred to the respective vector. On the other hand, if the interrupts are programmed level-sensitive, these flags are controlled by the external interrupts sources themselves. Both timers can be used in timer or counter mode. In counter mode, it counts the pulses at T_0 or T_1 pin. In timer mode, oscillator clock is divided by a pre-scalar (1/32) and then given to the timer. So clock frequency for timer is 1/32th of the controller operating frequency. The timer is an up-counter and generates an interrupt when the count has reached FFFFH. It can be operated in four different modes that can be set by TMOD register.

The timer 0 and timer 1 interrupt sources are generated by TF_0 and TF_1 bits of the register TCON, which are set, if a rollover takes place in their respective timer registers, except timer 0 in mode 3. When these interrupts are generated, the respective flags are automatically cleared after the control is transferred to the respective interrupt service routines.

The serial port interrupt is generated, if at least one of the two bits RI and TI is set. Neither of the flags is cleared, after the control is transferred to the interrupt service routine. The RI and TI flags need to be cleared using software, after deciding, which one of these two caused the interrupt? This is accomplished in the interrupt service routine.

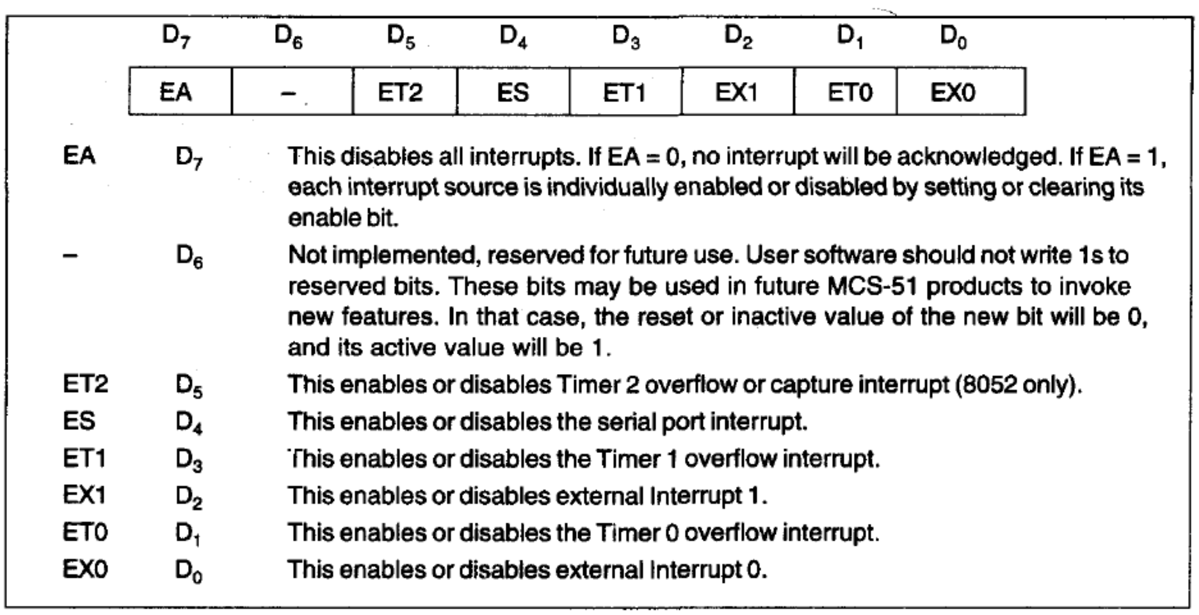
In addition to these five interrupts, 8051 also allows single step interrupts to be generated with help of software. The external interrupts, if programmed level-sensitive, should remain high for at least two machine cycles for being sensed. If the external interrupts are programmed edge-sensitive, they should remain high for at least one machine cycle and low for at least one machine cycle, for being sensed.

The interrupt structure of 8051 provides two levels of the interrupt priorities for its sources of interrupt. Each interrupt source can be programmed to have one of these two levels using the interrupt priority register IP. The different sources of interrupts programmed to have the same level of priority, further follow a sequence of priority under that level as shown:



Interrupt Source	Priority within a level
IE0 (External INT0)	Highest
TF0 (Timer 0)	⋮
IE1 (External INT1)	⋮
TF1 (Timer 1)	⋮
RI = TI (Serial Port)	Lowest

All these interrupts are enabled using a special function register called **interrupt enable register** (IE) and their priorities are programmed using another special function register called **interrupt priority register** (IP). Formats of both of these registers are shown in Fig. 17.13 and Fig. 17.14.



D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
EA	–	ET2	ES	ET1	EX1	ET0	EX0

EA	D_7	This disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
–	D_6	Not implemented, reserved for future use. User software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.
ET2	D_5	This enables or disables Timer 2 overflow or capture interrupt (8052 only).
ES	D_4	This enables or disables the serial port interrupt.
ET1	D_3	This enables or disables the Timer 1 overflow interrupt.
EX1	D_2	This enables or disables external Interrupt 1.
ET0	D_1	This enables or disables the Timer 0 overflow interrupt.
EX0	D_0	This enables or disables external Interrupt 0.

Fig. 17.13 *Format of IE Register*

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
–	–	PT2	PS	PT1	PX1	PT0	PX0

If the bit is 0, the corresponding interrupt is disabled. If the bit is 1 the corresponding interrupt is enabled.

–	D_7	Not implemented, reserved for future use.*
–	D_6	Not implemented, reserved for future use.*
PT2	D_5	This defines the Timer 2 interrupt priority level (8052 only).
PS	D_4	This defines the Serial Port interrupt priority level.
PT1/PT0	D_3/D_1	This defines the Timer 1/Timer 0 interrupt priority level.
PX1/PX0	D_2/D_0	This defines External $\overline{\text{INT1}}/\overline{\text{INT0}}$ priority level.

* The software should not write 1s to reserved bits. These bits may be used in future MCS-51 products to invoke new features. In that case, the reset or inactive value of the new bit will be 0, and its active value will be 1.

Fig. 17.14 *Format of IP Register*

INTERRUPTS

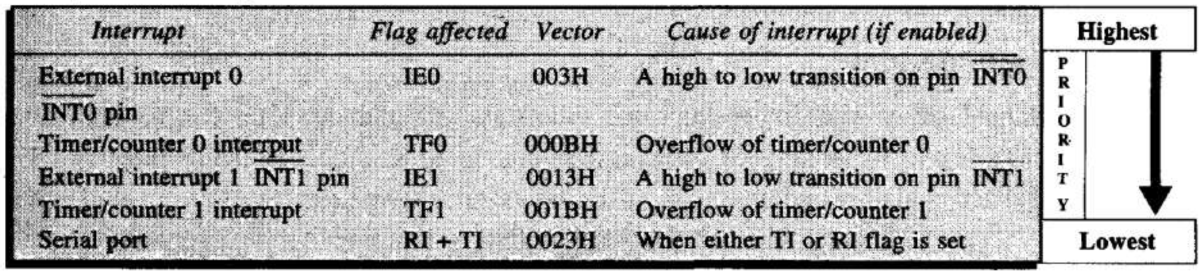
Interrupt is an input to a processor that indicates the occurrence of an event. In case of external events, the status of a microprocessor pin is altered. Interrupts are also generated due to the events occurring inside the machine like timer overflow or transmission/reception of a byte through the serial port, etc. The processor responds to an interrupt by saving the current machine status and branching to execute a subprogram called 'interrupt service subroutine'. When an interrupt occurs, the CPU jumps to the location associated with that interrupt, in the program memory and starts executing from there. This location is called 'vector' and the interrupt is called vectored interrupt. After serving the interrupt, the processor restores the original machine status and continues with the original program.

INTERRUPTS IN MCS-51

MCS-51 supports five vectored interrupt sources. These are external interrupt 0, external interrupt 1, timer/counter 0 interrupt, timer/counter 1 interrupt and serial port interrupts. When an interrupt is generated, the program counter (PC) is pushed onto a stack. Vectored address is loaded in the program counter. As the vectoring takes place, that particular interrupt flag corresponding to the interrupt source (e.g. external interrupt 1) is cleared by the hardware. In MCS-51, these flags are bits IE0, IE1, TF0, TF1, RI and TI.

The program now starts executing from the vectored location. This subroutine is called as the interrupt service subroutine (ISS). The ISS ends with RETI instruction. The interrupt vector locations in 8051 are spaced out at every 8 bytes, so technically it is possible to put ISS there if it were no longer than 8 bytes, including RETI instruction. Otherwise and in almost all the cases, a jump instruction is written at the vectored address (3 bytes maximum), and the remaining part ISS is located somewhere else. The vector addresses are listed in the following Table 6.1 in the order of priority. Consider the external interrupt 1. Assume that this interrupt is initialized properly in program. While the CPU is busy with the main program, if a '1' to '0' transition occurs at pin number 12, ($\overline{\text{INT0}}$ pin), the program counter (PC) current contents are stored onto the stack and the PC is then loaded with the vectored address 0003H. Thus, the next instruction at 0003H would be fetched and executed. Now there are only 8 bytes available to write the interrupt service subroutine, as seen above. Therefore, normally a JMP instruction is written at this vectored location 0003H. The interrupt service subroutine lying somewhere else in the program memory, ends with RETI instruction. This RETI instruction will get the program counter contents from the stack and the CPU will again start executing from where the main program was interrupted. Thus, any external event, which causes a change in the status of the interrupt pin, can be taken care of by the interrupt service subroutine. The external interrupts may be configured as either level-triggered or edge-triggered. If the interrupt is level-triggered, the signal must stay low until the interrupt is generated. In case of an edge-triggered interrupt, a transition from high to low at the interrupt pin is sufficient. It is further necessary that proper settings in the SFR called interrupt enable (IE) register is made to initialize the MCS-51 interrupts.

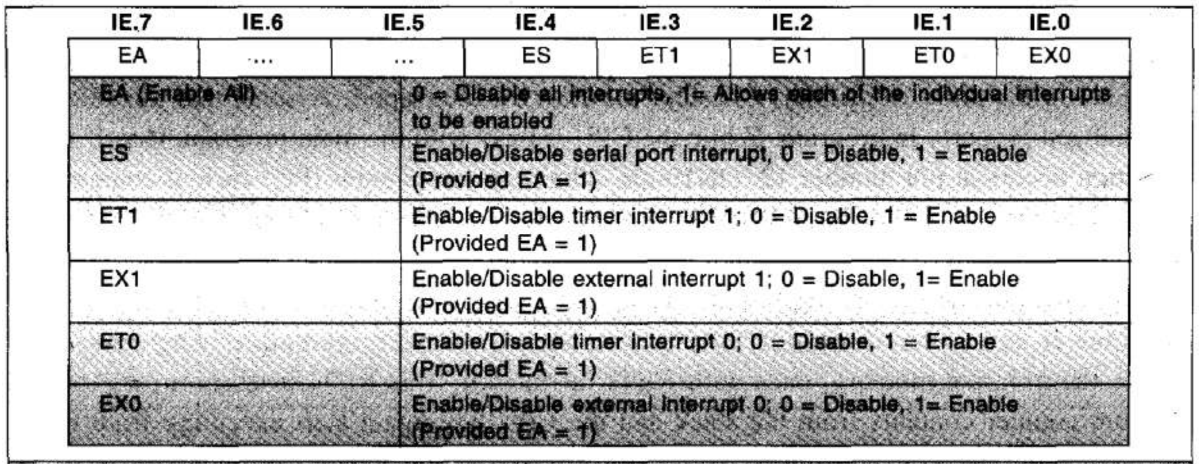
Table 6.1 Interrupts in 8051



Interrupt	*Flag affected*	*Vector*	*Cause of interrupt (if enabled)*	Highest
External interrupt 0 $\overline{\text{INT0}}$ pin	IE0	003H	A high to low transition on pin $\overline{\text{INT0}}$	P R I O R I T Y
Timer/counter 0 interrupt	TF0	000BH	Overflow of timer/counter 0	
External interrupt 1 $\overline{\text{INT1}}$ pin	IE1	0013H	A high to low transition on pin $\overline{\text{INT1}}$	
Timer/counter 1 interrupt	TF1	001BH	Overflow of timer/counter 1	
Serial port	RI + TI	0023H	When either TI or RI flag is set	Lowest

Initializing 8051 Interrupts

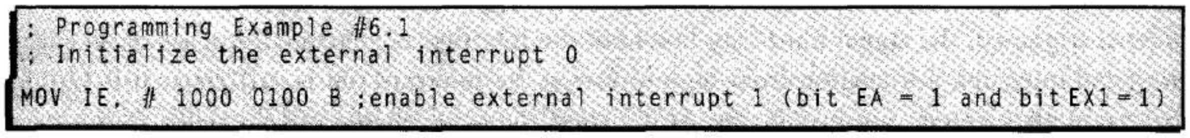
The interrupt enable (IE) register allows the programmer to enable interrupts as needed. This register IE is bit addressable and is shown in Fig. 6.1. Enable All (EA) bit allows disabling the whole interrupt operation, if cleared. Thus, it acts as a master control bit for any of the interrupts. For any particular interrupt to occur, bit EA and the corresponding bit must be set. For example, in case of serial interrupt, bit EA and bit ES must be set. ES is the serial port interrupt, useful in serial transmission, if set, enables the serial interrupts TI or RI Similarly, bits ET1, ET0 are for timer 1 and timer 0 interrupts, respectively. EX1 and EX0 are external interrupt enable bits for external interrupts 1 and 0, respectively. Programming Example #6.1 shows initialization of external interrupt 1.



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

EA (Enable All)	0 = Disable all interrupts, 1= Allows each of the individual interrupts to be enabled
ES	Enable/Disable serial port interrupt, 0 = Disable, 1 = Enable (Provided EA = 1)
ET1	Enable/Disable timer interrupt 1; 0 = Disable, 1 = Enable (Provided EA = 1)
EX1	Enable/Disable external interrupt 1; 0 = Disable, 1= Enable (Provided EA = 1)
ET0	Enable/Disable timer interrupt 0; 0 = Disable, 1 = Enable (Provided EA = 1)
EX0	Enable/Disable external interrupt 0; 0 = Disable, 1= Enable (Provided EA = 1)

Fig. 6.1 Interrupt Enable Register (Bit Addressable)



```
; Programming Example #6.1
; Initialize the external interrupt 0
MOV IE, # 1000 0100 B ;enable external interrupt 1 (bit EA = 1 and bit EX1=1)
```

This instruction will enable the external interrupt 1. If now this is followed by CLR EA instruction, whole interrupt operation is disabled. To initialize the serial interrupt, one may load the IE register with 10010000B.

Interrupt Priorities

Let us consider the case, when more than one interrupts are enabled. User can program the interrupt priority levels by setting or clearing the bits in SFR called interrupt priority (IP) register. IP register is also bit addressable. If the bit is set, that particular interrupt will have high priority.

A high-priority interrupt can interrupt the low-priority interrupt, but a high-priority interrupt will not be interrupted by the interrupt having low priority. Now, if the request of interrupts of two different priority levels occur simultaneously, naturally the interrupt having the high priority will be served. However, if the same priority level interrupts request simultaneously, then within each priority level there is a polling structure due to the inherent priority in the order shown in Table 6.1 by the arrow. Note that the priority within level structure is used only to distinguish the requests of the same priority levels. Programming Example #6.2 shows the assignment of interrupt priority to timer 1 interrupt.

```
; Programming Example #6.2
; Assigning Interrupt Priorities
     MOV IE, #1000 1100H          ;Enable EX1 and ET1
     SETB PT1                     ;Timer 1 interrupt has high priority.
```

The first instruction enables both interrupts, namely, the external interrupt 1 and the timer 1 interrupt. The instruction SETB PT1 assigns high priority to the timer interrupt. So, if both of them request simultaneously, then the timer interrupt will be served. However, let us see what happens when one more instruction is added to this program. This is shown in Programming Example #6.3. Both external interrupt 1 and timer 1 interrupt have the same priorities. Now, if either interrupt requests occur simultaneously, the external interrupt will be served as per the priority order mentioned in the Table 6.1 and Fig. 6.2.

IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
X	X	PT2	PS	PT1	PX1	PT0	PX0

IP.5	PT2	Timer 2 priority in case of 8032/8052 only
IP.4	PS	Serial interrupt priority
IP.3	PT1	Timer 1 interrupt
IP.2	PX1	External interrupt 1
IP.1	PT0	Timer 0 interrupt
IP.0	PX0	External interrupt 0

Fig. 6.2 Interrupt Priority Register (Bit Addressable)

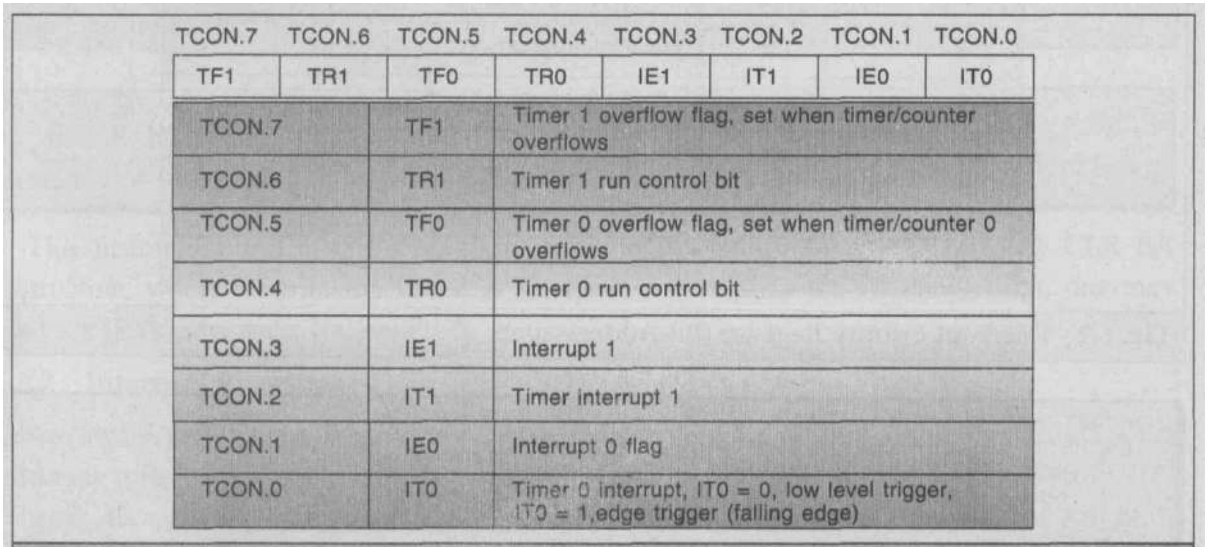
```
; Programming Example #6.3
; Assigning Interrupt Priorities
     MOV IE, #1000 1100H          ;Enable EX1 and ET1
     SETB PT1                     ;Timer 1 interrupt has high priority
     SETB PX1                     ;External interrupt also has high priority
```

TIMERS AND COUNTERS

On-chip timing/counting facility has proved the capabilities of the micrcontrollers for implementing the real time applications. These include pulse counting, frequency measurement, pulse width measurement, baud rate generation, etc. Having sufficient number of timer/counters may be a need in a certain design application. As seen in the first chapter, 8051 has two 16-bit timer/counters. Before discussing 8051 timer/counters, it is necessary to see the exact difference between a timer and a counter. A timer counts machine cycles and provides a reference time delay or a clock. A machine cycle of 8051 consists of 12 oscillator periods or the counting rate is 1/12 of the oscillator frequency. At 12 MHz, the clocking period will be equal to 1 µs. Let us now see, the counting function. A counter of 8051 is incremented in response to a transition from '1' to '0' at its corresponding external pin (either T0 or T1). Thus, the counter output will be a count or a number representing the occurrence of such '1' to '0' transitions at the external pin. For counting function, 8051 takes 2 machine cycles or 24 oscillator periods to detect a '1' to '0' transition at Pin T0 or T1. When a timer or counter overflows from FFFFH to 0000H, it sets a flag and generates an interrupt. The 16 bits of timer are referred as higher byte THx and the lower byte TLx. Thus, TH1 is the higher byte of timer 1 and TL1 is the lower byte of timer 1. 'x' can be 0 or1 (or 2 in case of 8032/52).

Timer/Counter Modes

There are four timer modes in 8051. A timer or counter function and modes are selected by writing appropriate bits in the SFR, called the timer mode register (TMOD), whereas the control of timer/counter operation is done through the SFR, called the timer control register (TCON). These SFRs are shown in Figs. 6.3 and 6.4. Now, the question is how exactly to

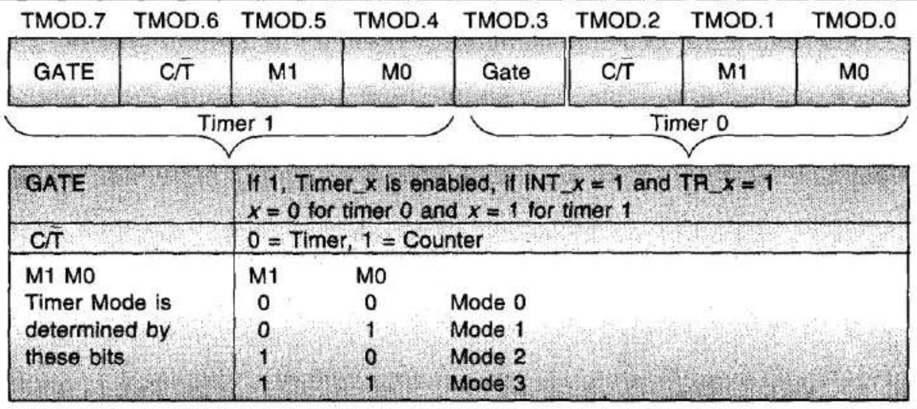


TCON.7	TCON.6	TCON.5	TCON.4	TCON.3	TCON.2	TCON.1	TCON.0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TCON.7	TF1	Timer 1 overflow flag, set when timer/counter overflows
TCON.6	TR1	Timer 1 run control bit
TCON.5	TF0	Timer 0 overflow flag, set when timer/counter 0 overflows
TCON.4	TR0	Timer 0 run control bit
TCON.3	IE1	Interrupt 1
TCON.2	IT1	Timer interrupt 1
TCON.1	IE0	Interrupt 0 flag
TCON.0	IT0	Timer 0 interrupt, IT0 = 0, low level trigger, IT0 = 1,edge trigger (falling edge)

Fig. 6.3 Timer Control Register (TCON) (Bit Addressable)

Configure timer/counters as a timer or counter. As seen from Fig. 6.4, the TMOD bit $C/\overline{T}$, defines this operation.



GATE	If 1, Timer_x is enabled, if INT_x = 1 and TR_x = 1 x = 0 for timer 0 and x = 1 for timer 1
C/$\overline{T}$	0 = Timer, 1 = Counter
M1 M0 Timer Mode is determined by these bits	M1 M0 0 0 Mode 0 0 1 Mode 1 1 0 Mode 2 1 1 Mode 3

Fig. 6.4 Timer Mode Control Register (TMOD)

A). Mode 0

In mode 0, the timer is 13-bit wide. This mode is same for timer 0 and timer 1. When the count overflows, it sets the timer interrupt flag (TF1 for timer 1 and TF0 for timer 0). To start timer 0, TR0 bit in TCON is required to be set. Using the upper byte TH1 (or TH0) and the lower 5 bits of TL1 (or TL0) forms the 13 bits. This is shown in Fig. 6.5 and Fig. 6.6.

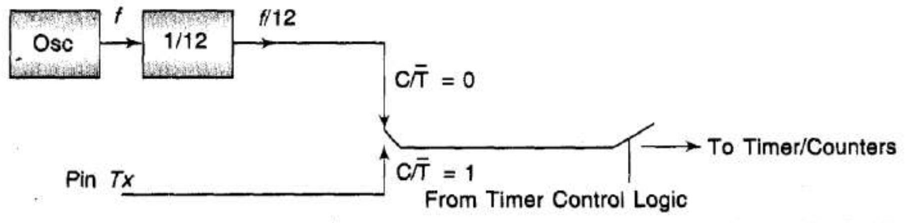


Fig. 6.5 C/$\overline{T}$ bit in TMOD Decides Timer or Counter Operation

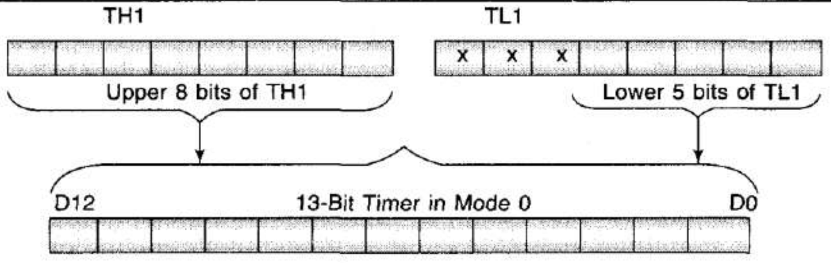


Fig. 6.6 Mode 0, 13-bit Timer/Counter

```
; Programming Example #6.4
; Initializing timer 1 in mode 0
MOV TMOD, # 1000 0000 B
; Timer 1 in mode 0, Timer 0 in mode 0, both are configured as timers.
; Timer 1 is controlled by the external Pin 13 INT1 (Note GATE=1),
whereas the system clock clocks timer 0:
SETB TR1           ; Start timer 1
SETB TR0           ; Start timer 0
CLR TR1            ; Stop timer 1
SJMP $             ; Infinite loop
```

Programming Example #6.4 initializes timer 1 in mode 0. In the above program, timer 1 is configured as a timer in mode 0. Observe that bit TMOD.7 in TMOD is set to 1. This is the GATE bit. If this is set to 1 and TR1 is 1, then the timer 1 is controlled by the external input at Pin 13 ($\overline{\text{INT1}}$). This can be seen from Fig. 6.7. When GATE is 0, then, it is only TR1 which enables the timer.

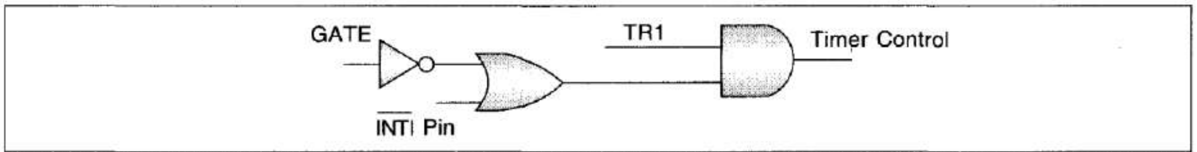


Fig. 6.7 Timer Control Logic in Mode 0 or 1

B). Mode 1

Mode 1 is same as mode 0, except the timers are 16 bits wide. Mode 1 is again the same for timer 0 and timer 1. The maximum count in this mode is FFFFH. To initialize timer 1 in mode 1, see Programming Example #6.5.

```
; Programming Example #6.5
; Initializing timer1 in mode 1
MOV TMOD, # 0001 0000 B        ; Timer 1 in mode 1
SETB TR1                       ; As the GATE bit is zero, TR1 can fully:
                               ; control the timer operation
                               ; Hence to start timer 1, TR1 is set to 1.
SJMP $                         ; Infinite loop
```

If initialized, the timer overflow can generate an interrupt. Consider one such program segment (Programming Example #6.6) to initialize the timer 1 interrupt. Note that it is always advisable to initialize the stack pointer before going for a main program, because the default value of SP 07H may not be suitable in general. This is also the address of register R7, and if any register bank switching is done, it can overwrite some useful register contents.

```
; Programming Example #6.6
; Program to initialize timer 1 in mode 1
MOV SP, #54H                    ; Initialize the stack pointer
MOV TMOD, # 0001 0000 B         ; Timer1 in mode 1
SETB ET1                        ; Enable timer 1 interrupt
SETB TR1                        ; Start timer 1
SETB EA                         ; Enable all
SJMP $                          ; Infinite loop (Jump here)
```

Note that simply setting only ET1 bit in IE register will not enable the timer interrupt. In addition, it is necessary to set the EA bit in IE. This program will start timer 1, and when it overflows, timer 1 interrupt is generated, which will cause the program counter to jump to the vector location 001B H.

C). Mode 2

This operation is again the same for timer 0 and timer 1. Consider timer 1 in mode 2. Timer register is configured as an 8-bit counter TL1. Overflow from TL1 sets the flag TF1, and it loads TL1 with the contents of TH1. The software can preload TH1. This mode of timer 1 or timer 0 thus supports the automatic reload operation. Mode 2 auto-reload mechanism is shown in Fig. 6.8. Timer control logic is again the same as that of mode 0 or 1.

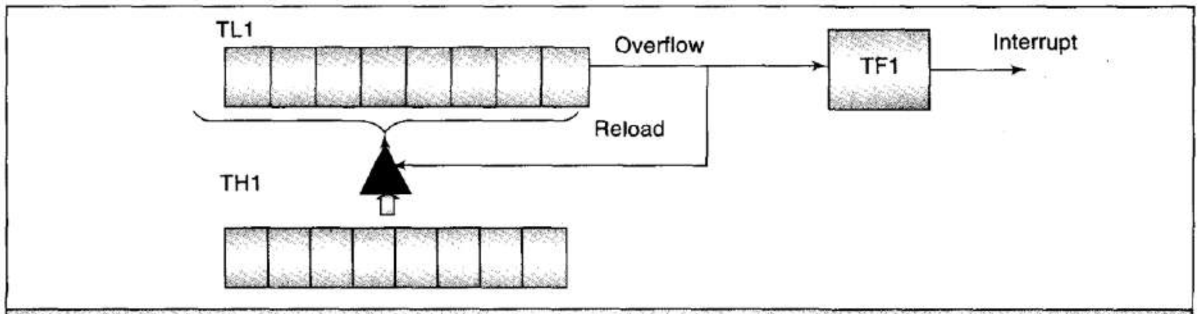


Fig. 6.8 Mode 2 Auto-reload Mechanism of Timer

Let us now write the initialization program for timer 0 in mode 2 as shown in the Programming Example #6.7. The program must load TMOD and then the auto-reload value must be written in the timer high byte. Further, the starting count will also be the same as that of the reload value in general, but it is not so strict since this is applicable for the very first overflow.

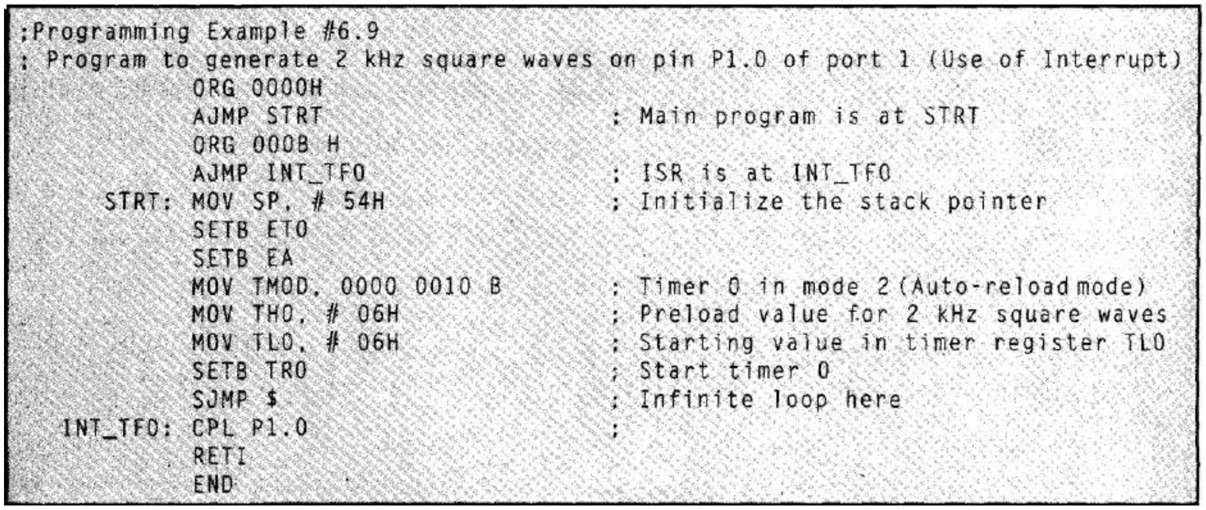
However, it is very essential to load the timer high byte with the auto-reload value, otherwise the timer after each overflow will start from 00H.

```
; Programming Example #6.7
; Initializing timer 0 in mode 2
MOV TMOD, # 0000 0010 B      ; Load TMOD for timer 0 in mode 2
MOV THO, # 33H               ; Load THO with preset value to be reloaded
MOV TLO, # 33 H              ; Starting count = preset value
SETB TRO                     ; Start timer 0
SJMP $
```

Mode 2 is very commonly used for baud rate generation for serial port operation, or where a constant frequency square wave output is needed. The frequency or baud rate can be controlled using the preloaded value in THx register. The maximum delay generated using mode 2 will be corresponding to the auto-reload value of 00H. Thus, at 12 MHz clock, this would generate the maximum delay of 256 µs. If one can write an instruction to toggle any of the port pins, a square wave output on that pin can be seen on the oscilloscope. Consider this program to generate a 2 kHz (0.5 ms period) square waveform on pin Pl.0 (Pin 1), as shown in Programming Example #6.8. The reload count will be corresponding to 0.25mS. At 12 MHz, this will be (256-250) equal to 06H.

```
;Programming Example #6.8
; Program to generate 2 kHz square waves on pin P1.0 of port 1
             MOV SP, # 54H: Initialize the stack pointer
             MOV TMOD, 0000 0010 B: Timer 0 in mode 2 (Auto-reload mode)
             MOV THO, # 06H: Preload value for 2 kHz square waves
             MOV TLO, # 06H: Starting value in timer register TLO
             SETB TRO: Start timer 0
      LOOP:  JB TFO, COMPLMNT
             SJMP LOOP
  COMPLMNT:  CPL P1.0: Toggle bit P1.0
             SJMP LOOP
```

The same program could be written using timer interrupt also. Let us see how to do it! Note that timer 0 interrupt has been enabled at the time of starting the timer. The timer 0 is initialized in mode 2 or auto-reload mode. TH0 and TL0 both are initialized to the count 06H corresponding to the 2 kHz frequency of square waves. The main program is over once timer 0 is started. But notice the instruction SJMP $. This instruction is to jump at the same address and generate an infinite loop. The effect is same as the instruction "LABEL: SJMP LABEL". This program is shown in the Programming Example #6.9.



```
;Programming Example #6.9
; Program to generate 2 kHz square waves on pin P1.0 of port 1 (Use of Interrupt)
           ORG 0000H
           AJMP STRT                    ; Main program is at STRT
           ORG 000B H
           AJMP INT_TF0                 ; ISR is at INT_TF0
     STRT: MOV SP, # 54H                ; Initialize the stack pointer
           SETB ET0
           SETB EA
           MOV TMOD, 0000 0010 B        ; Timer 0 in mode 2(Auto-reload mode)
           MOV TH0, # 06H               ; Preload value for 2 kHz square waves
           MOV TL0, # 06H               ; Starting value in timer register TL0
           SETB TR0                     ; Start timer 0
           SJMP $                       ; Infinite loop here
  INT_TF0: CPL P1.0                     ;
           RETI
           END
```

Further, we have written an interrupt service routine (ISR) in which we just complement the bit Pl.0. The ISR ends with RETI instruction. After the execution of this instruction, the CPU will again be in the infinite loop, from where it was interrupted. Note that timer 0, once started, is not made off or on afterwards, which is not needed also due to auto-reload feature.

D). Mode 3

In this mode, timer 1 has a passive role of holding its count. In effect, it looks like as the one who keeps TR1 = 0. Now, timer 0 bytes TH0 and TL0 are used as two separate timers. Because of this, mode 3 is also called as split timer mode. TH0 is locked into timer operation and simply counts the machine cycles. After overflowing, it sets the flag TF1.

TL0 can be configured and controlled by using C/$\overline{T}$, GATE, TR0, INT0, and TF0. Note that TR1 controls the operation of TH0 timer, now the question remains how to control timer 1? Timer 1 can be used in a different manner, for any application that does not require the interrupt operation; like for generating the baud rate for serial port operation. When timer 0 is in mode 3, one can just control its operation by switching it out or into its mode 3 using TMOD settings. Thus, in mode 3 it resembles like 8051 having 3 timer/counters.

Note that in timer mode 3, timer 1 is a 16-bit timer and TH0, TL0 two 8-bit timers.

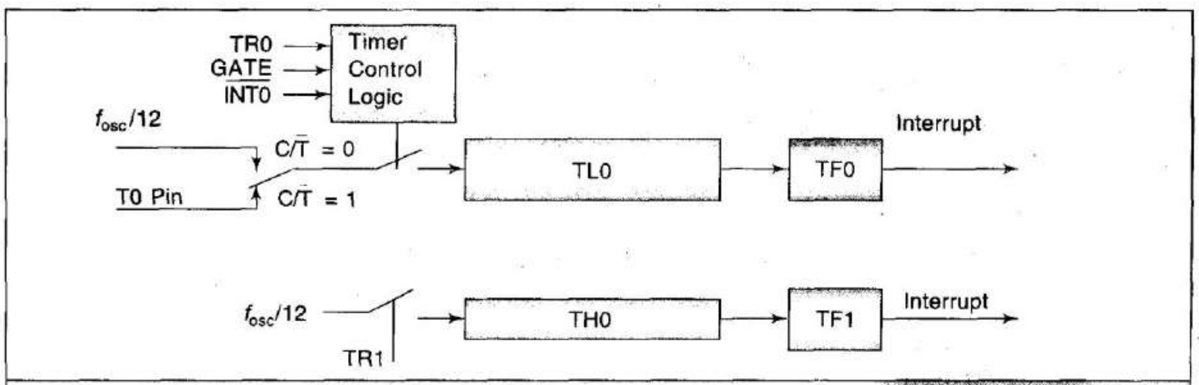
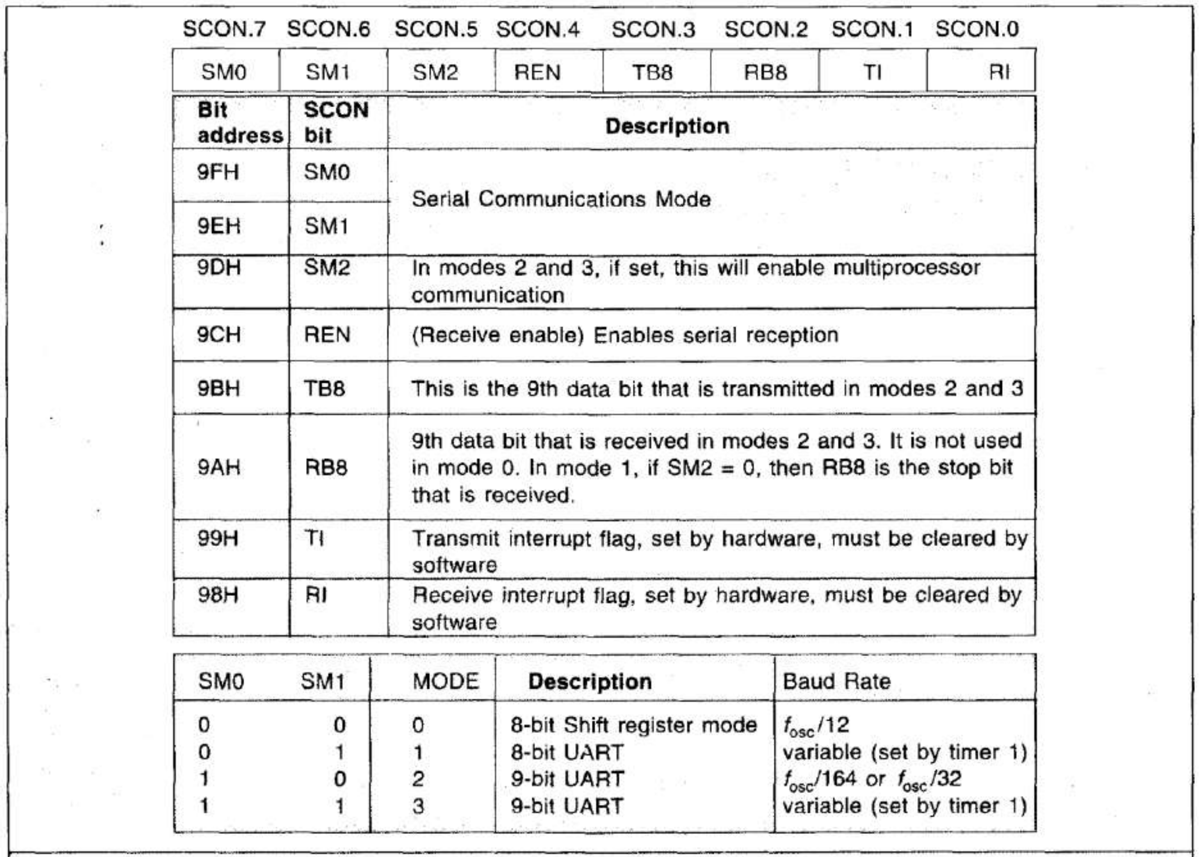


Fig. 6.9 Timer/Counter in Mode 3

SERIAL COMMUNICATION

Serial data transmission is very commonly used for digital data communication. Its main advantage is that the number of wires needed is reduced as compared to that in parallel communication. 8051 supports a full duplex serial port. Full duplex means, it can transmit and receive a byte simultaneously. 8051 has TXD and RXD pins for transmission and reception of serial data respectively. The 8051 serial communication is supported by RS232 standard. The term "RS" stands for Recommended Standard. Communication between two microcontrollers and multiprocessor communication is also possible. The start and stop bits are used to synchronize the serial receivers. The data byte is always transmitted with least-significant-bit first. For error checking purpose, it is possible to include a parity bit as well, just prior to the stop bit. Thus, the bits are transmitted at specific time intervals determined by the baud rate. For error-free serial communication, it is necessary that the baud rate, the number of data bits, the number of stop bits, and the presence or absence of a parity bit along with its status be the same at the transmitter and receiver ends.

The basic mechanism of serial transmission is that a data byte in parallel form is converted into serial data stream. Along with some more bits like start, stop and parity bits, a serial data frame is sent over a line. There are four modes of serial data transmission in 8051. In each of these modes, it is important to decide the baud rate, the way in which serial data frame is sent and any other information, etc.



SCON.7	SCON.6	SCON.5	SCON.4	SCON.3	SCON.2	SCON.1	SCON.0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Bit address	SCON bit	Description
9FH	SM0	Serial Communications Mode
9EH	SM1	
9DH	SM2	In modes 2 and 3, if set, this will enable multiprocessor communication
9CH	REN	(Receive enable) Enables serial reception
9BH	TB8	This is the 9th data bit that is transmitted in modes 2 and 3
9AH	RB8	9th data bit that is received in modes 2 and 3. It is not used in mode 0. In mode 1, if SM2 = 0, then RB8 is the stop bit that is received.
99H	TI	Transmit interrupt flag, set by hardware, must be cleared by software
98H	RI	Receive interrupt flag, set by hardware, must be cleared by software

SM0	SM1	MODE	Description	Baud Rate
0	0	0	8-bit Shift register mode	$f_{osc}/12$
0	1	1	8-bit UART	variable (set by timer 1)
1	0	2	9-bit UART	$f_{osc}/164$ or $f_{osc}/32$
1	1	3	9-bit UART	variable (set by timer 1)

Fig. 6.10 Serial Control Register (SCON)

What is common in all these modes is the use of the SFR called "SBUF", for transmission as well as reception. The data to be transmitted must be transferred to SBUF. One more SFR that controls the serial communication operation is the serial control register SCON. Details of SCON are shown in Fig. 6.10. Bits SM0 and SM1 in SCON define serial port mode. Bit SM2 enables the multiprocessor communication in modes 2 and 3. Transmission is initiated by the execution of any instruction that uses SBUF as the destination.

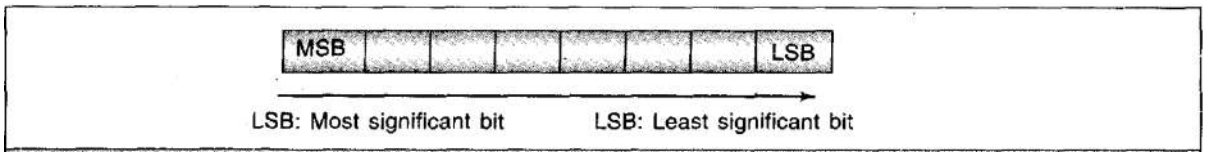


Fig. 6.11 A Frame of 8 bits Transmitted Serially in Mode 0, with LSB First

Serial Communication Modes

There are four modes in which 8051 serial port can be configured.

A. Mode 0

This is also called as shift register mode. Only RXD is the pin through which data enter or exits. TXD pin outputs the shift clock only. Eight data bits are transmitted or received. The baud rate is fixed and is totally determined by the system clock frequency. If f_{osc} is the clock frequency, then $f_{osc}/12$ will be the baud rate.

To see exactly how the operation of serial data transfer takes place in mode 0, see Programming Example #6.11.

```
; Programming Example #6.11
; Serial transmission mode 0
          ORG 0000H             ; Program starts at 0000H
          MOV SCON, #0000 0000B ; Mode 0
; Now write the data byte to be transmitted in SBUF
          MOV SBUF, #44H        ; Transmit 0100 0100 binary
; After transmission, TI flag in SCON will be set by hardware, this can be
; tested for assuring the transmission operation
          Here: JNB TI, Here
                                ; Wait till all 8 bits are transmitted
                                ; Remember TI flag must be cleared
          CLR TI; TI flag is reset
```

B. Mode 1

In mode 1, 10 bits are transmitted through TXD pin or received through RXD pin. There is a start bit (0), then 8 data bits (LSB first) and a stop bit (1). This is shown in Fig. 6.12. On receiving, the stop bit goes into RB8 in SCON. The baud rate is variable and is determined by the timer 1 overflow rate. Therefore, before using this mode, one has to initialize timer 1. A simple program to initialize serial port in mode 1 is given in Programming Example #6.12. The baud rate is calculated using the formula:

Baud rate = 2SMOD/32 x (Timer 1 overflow rate) (6.1)

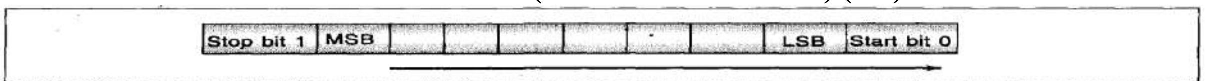


Fig. 6.12 Ten Bits Transmitted in Mode 1 of Serial Transmission

```
:Programming Example #6.12
: Initializing the serial port in mode 1
ORG 0000H
MOV SP, #51H
; Note that SMOD is '0' after RESET
MOV SCON, #0100 0000B         ; Serial port in mode 1
MOV TMOD, #0010 0000B         ; Timer 1 in auto-reload mode
MOV TH1, #230 D               ; Baud rate =1200 at 12 MHz
SETB TR1                      ; Start timer
MOV SBUF, #56H
JNB TI, $                     ; Wait till the transmission is over
CLR TI                        ; Reset bit TI after transmission
```

If timer 1 is configured in auto-reload mode (or mode 2), with reload value in TH1, after each overflow, contents of TH1 will be loaded into TL1. This is convenient for generating baud rate. In this mode, TMOD high nibble will be 0010B. At 12 MHz oscillator frequency, the timer clocking time is 1μs. Now, the baud rate formula is simplified to

Baud rate = [2SMOD/32] x (oscillator frequency) / [12 x (256 - (TH1)] (6.2)

For example, if TH1 contents are 230D, and SMOD bit in PCON is 0, then the baud rate at 12 MHz is 1201 baud or 1.2K approximately. To get exactly 1200 baud, the oscillator frequency must be 11.059 MHz This shows the degree of dependency of the baud rate on the operating frequency. Thus, to be precise, the actual oscillator frequency must be measured on the oscilloscope.

To receive a byte in mode 1, the RI bit in SCON is tested for 1. Similarly, the REN bit in SCON must be'1'.

The following Programming Example #6.13 will receive a byte through pin RXD.

```
;Programming Example #6.13
; Receiving a serial byte through RXD
MOV SCON, #0101 0000B        ; Serial port mode 1 and REN bit is set
                             ;SMOD is 0 after RESET
MOV TMOD, #0010 0000B        ; Timer 1 in mode 2
MOV TH1, #230D               ; Baud rate 1.2K at 12 MHz
SETB TR1                     ; Start timer 1
CLR R1                       ; Ready to receive
JNB R1, $                    ; Wait till a byte is received in SBUF
MOV A, SBUF                  ; Get the received byte in accumulator.89
```

C. Mode 2

In mode 2, 11 bits are transmitted, with a low start bit, then 8 data bits, a 9th bit and a stop bit T. This is shown in Fig. 6.13.

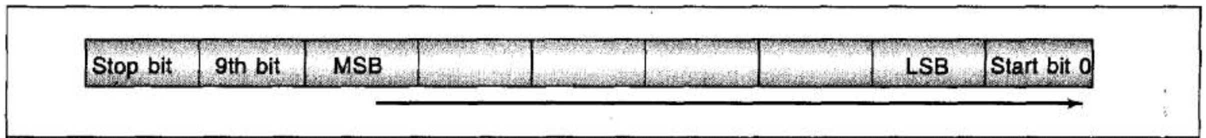
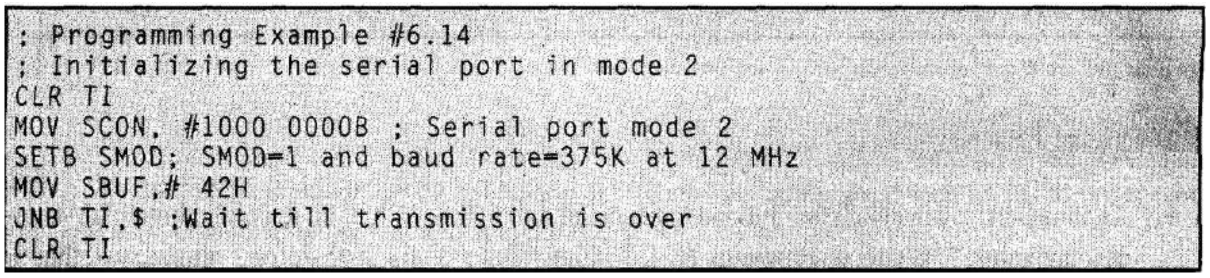


Fig. 6.13 Eleven Bits Transmitted in 8051 Serial Communication Mode 2

The 9th bit is programmable. User program can define 9th bit as TB8 in SCON. It may be the parity of data byte. On reception, this 9th data bit goes into RB8 in SCON. In mode 2, the bit SMOD in PCON and the oscillator frequency defines the baud rate and is given by

Baud rate = [2SMOD/64] x (oscillator frequency) (6.3)

Now consider Programming Example #6.14 to initialize the serial port in mode 2. At 12 MHz oscillator frequency, if SMD bit is 1, then the baud rate will be 375,000 or 375K.



```
: Programming Example #6.14
: Initializing the serial port in mode 2
CLR TI
MOV SCON, #1000 0000B ; Serial port mode 2
SETB SMOD; SMOD=1 and baud rate=375K at 12 MHz
MOV SBUF,# 42H
JNB TI,$ ;Wait till transmission is over
CLR TI
```

D. Mode3

Again 11 bits are transmitted as shown in Fig. 6.13, this is almost same as mode 2, except that the baud rate is defined by the timer 1 overflow rate. The baud rate calculations are exactly same as that of mode 1.

UNIT-III
INTRODUCTION TO EMBEDDED C AND APPLICATIONS

KEYBOARD INTERFACING

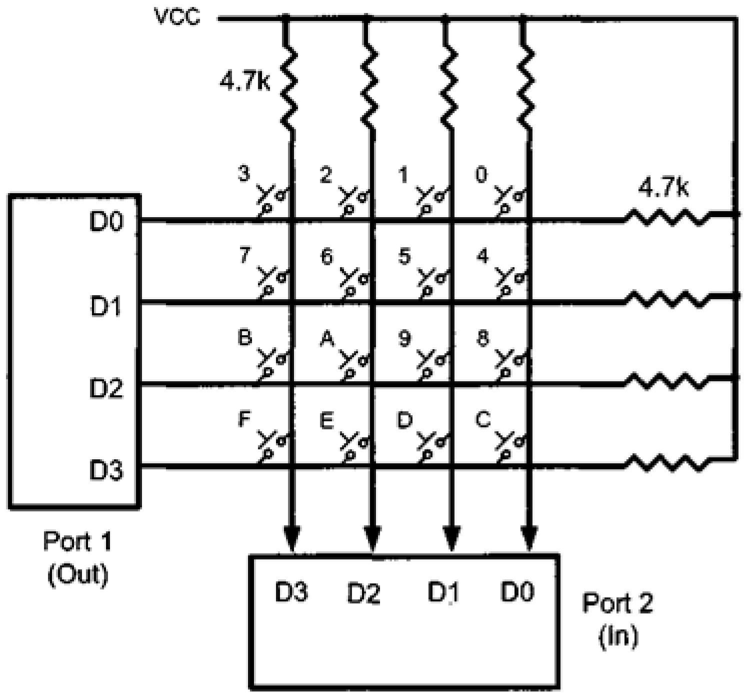
Keyboards and LCDs are the most widely used input/output devices of the 8051, and a basic understanding of them is essential. In this section, we first discuss keyboard fundamentals, along with key press and key detection mechanisms. Then we show how a keyboard is interfaced to an 8051.

Interfacing the keyboard to the 8051

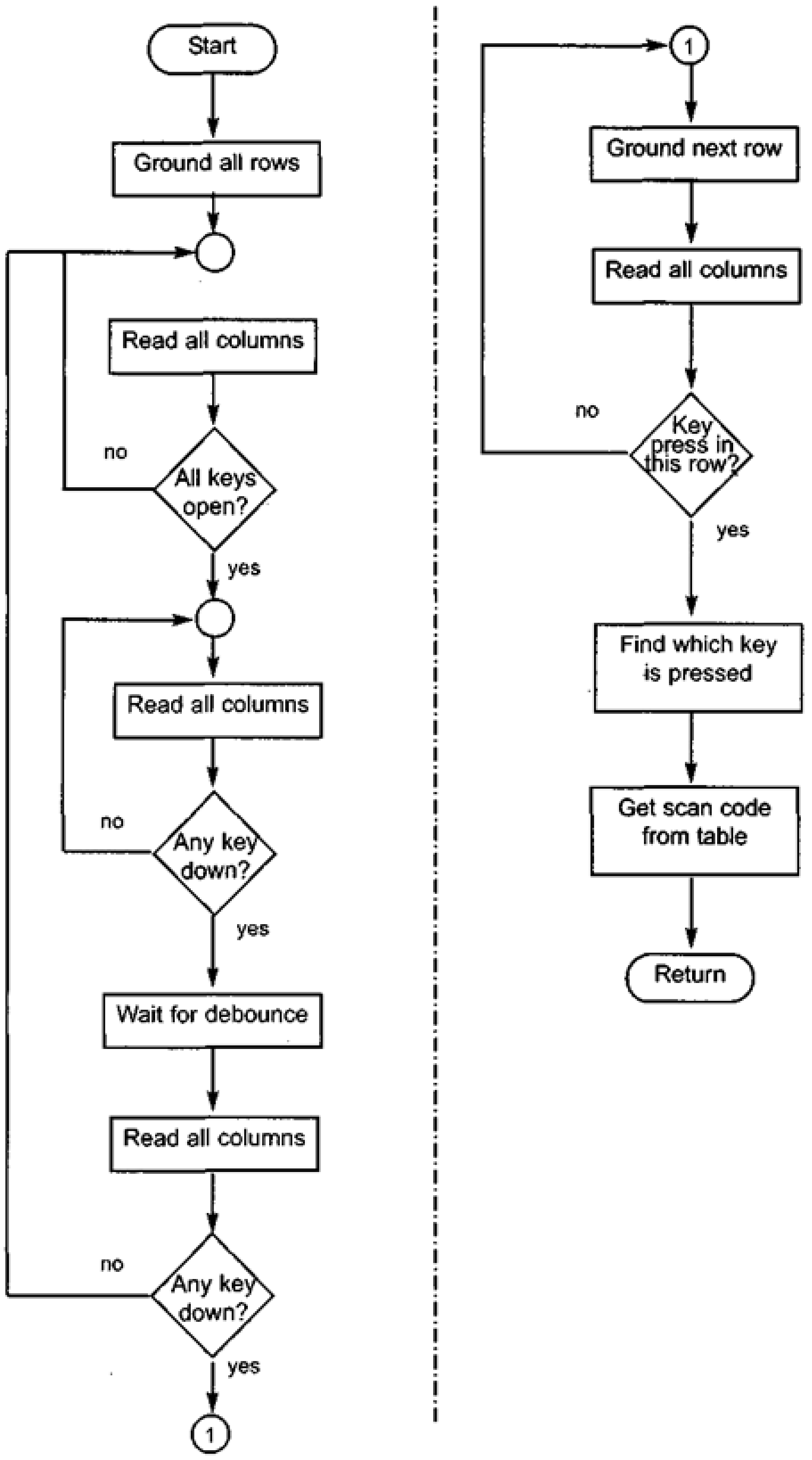
At the lowest level, keyboards are organized in a matrix of rows and columns. The CPU accesses both rows and columns through ports; therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor. When a key is pressed, a row and a column make a contact; otherwise, there is no connection between rows and columns. In IBM PC keyboards, a single microcontroller (consisting of a microprocessor, RAM and EPROM, and several ports all on a single chip) takes care of hardware and software interfacing of the keyboard. In such systems, it is the function of programs stored in the EPROM of the microcontroller to scan the keys continuously, identify which one has been activated, and present it to the motherboard. In this section we look at the mechanism by which the 8051 scans and identifies the key.

Scanning and identifying the key

Figure 12-6 shows a 4 x 4 matrix connected to two ports. The rows are connected to an output port and the columns are connected to an input port. If no key has been pressed, reading the input port will yield 1 s for all columns since they are all connected to high (V_{cc}). If all the rows are grounded and a key is pressed, one of the columns will have 0 since the key pressed provides the path to ground. It is the function of the microcontroller to scan the keyboard continuously to detect and identify the key pressed. How it is done is explained next.



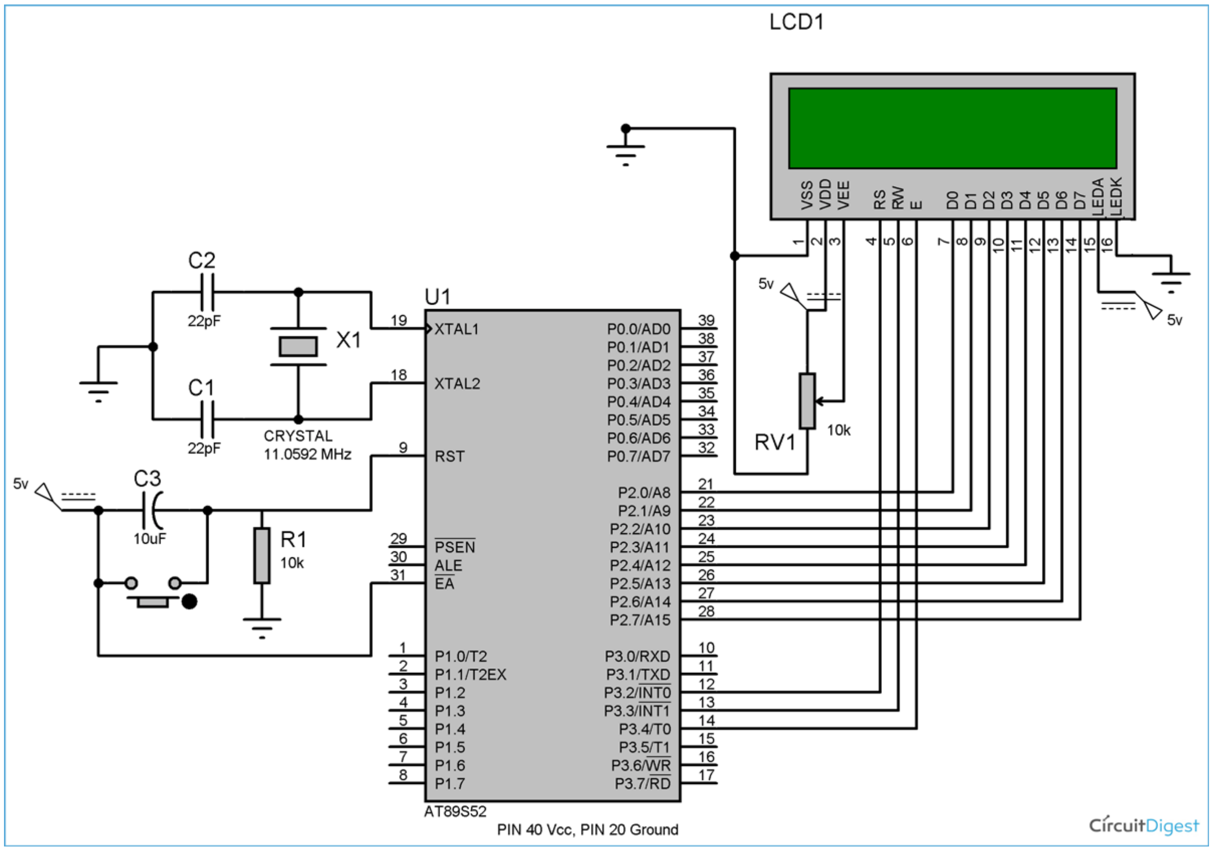
VCC
4.7k
4.7k
3
2
1
0
7
6
5
4
B
A
9
8
F
E
D
C
D0
D1
D2
D3
Port 1
(Out)
D3
D2
D1
D0
Port 2
(In)



Start
Ground all rows
Read all columns
All keys open?
no
yes
Read all columns
Any key down?
no
yes
Wait for debounce
Read all columns
Any key down?
no
yes
1
1
Ground next row
Read all columns
Key press in this row?
no
yes
Find which key is pressed
Get scan code from table
Return

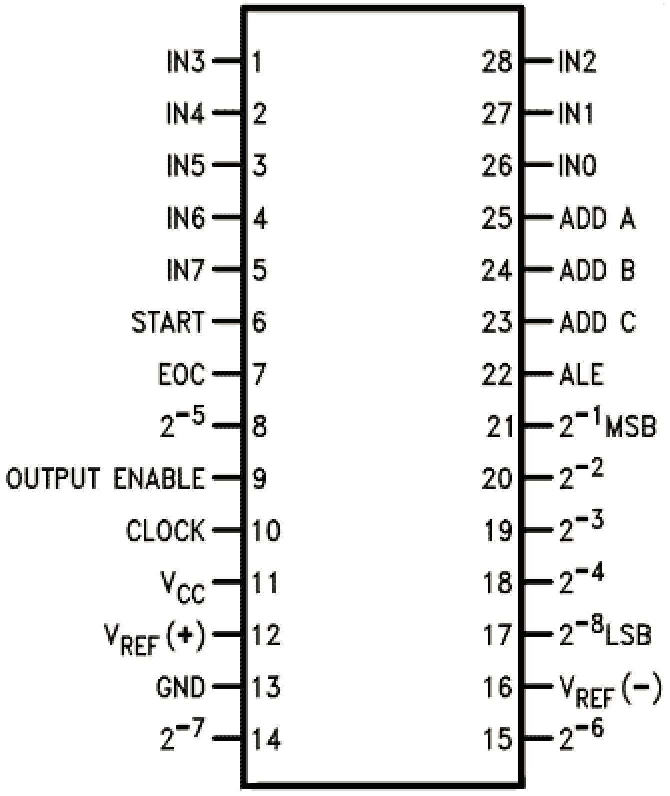
```
;Keyboard subroutine. This program sends the ASCII code
;for pressed key to P0.1
;P1.0-P1.3 connected to rows P2.0-P2.3 connected to columns
        MOV    P2,#0FFH          ;make P2 an input port
K1:     MOV    P1,#0             ;ground all rows at once
        MOV    A,P2        ;read all col. ensure all keys open
        ANL    A,#00001111B      ;masked unused bits
        CJNE   A,#00001111B,K1   ;check til all keys released
  :     ACALL DELAY              ;call 20 ms delay
        MOV    A,P2              ;see if any key is pressed
        ANL    A,#00001111B      ;mask unused bits
        CJNE   A,#00001111B,OVER ;key pressed, await closure
        SJMP   K2                ;check if key pressed
OVER:   ACALL DELAY              ;wait 20 ms debounce time
        MOV    A,P2              ;check key closure
        ANL    A,#00001111B      ;mask unused bits
        CJNE   A,#00001111B,OVER1 ;key pressed, find row
        SJMP   K2                ;if none, keep polling
OVER1:  MOV    P1,#11111110B     ;ground row 0
        MOV    A,P2              ;read all columns
        ANL    A,#00001111B      ;mask unused bits
        CJNE   A,#00001111B,ROW_0 ;key row 0, find the col.
        MOV    P1,#11111101B     ;ground row 1
        MOV    A,P2              ;read all columns
        ANL    A,#00001111B      ;mask unused bits
        CJNE   A,#00001111B,ROW_1 ;key row 1, find the col.
        MOV    P1,#11111011B     ;ground row 2
        MOV    A,P2              ;read all columns
        ANL    A,#00001111B      ;mask unused bits
        CJNE   A,#00001111B,ROW_2 ;key row 2, find the col.
        MOV    P1,#11110111B     ;ground row 3
        MOV    A,P2              ;read all columns
        ANL    A,#00001111B      ;mask unused bits
        CJNE   A,#00001111B,ROW_3 ;key row 3, find the col.
        LJMP   K2                ;if none, false input, repeat

ROW_0:  MOV    DPTR,#KCODE0      ;set DPTR=start of row 0
        SJMP   FIND              ;find col. key belongs to
ROW_1:  MOV    DPTR,#KCODE1      ;set DPTR=start of row 1
        SJMP   FIND              ;find col. key belongs to
ROW_2:  MOV    DPTR,#KCODE2      ;set DPTR=start of row 2
        SJMP   FIND              ;find col. key belongs to
ROW_3:  MOV    DPTR,#KCODE3      ;set DPTR=start of row 3
FIND:   RRC    A                 ;see if any CY bit is low
        JNC    MATCH             ;if zero, get the ASCII code
        INC    DPTR              ;point to next col. address
        SJMP   FIND                  ;keep searching
MATCH:  CLR    A                     ;set A=0 (match is found)
        MOVC   A,@A+DPTR             ;get ASCII code from table
        MOV    P0,A                  ;display pressed key
        LJMP   K1
;ASCII LOOK-UP TABLE FOR EACH ROW
        ORG    300H
KCODE0: DB     '0','1','2','3'               ;ROW 0
KCODE1: DB     '4','5','6','7'               ;ROW 1
KCODE2: DB     '8','9','A','B'               ;ROW 2
KCODE3: DB     'C','D','E','F'               ;ROW 3
       END
```

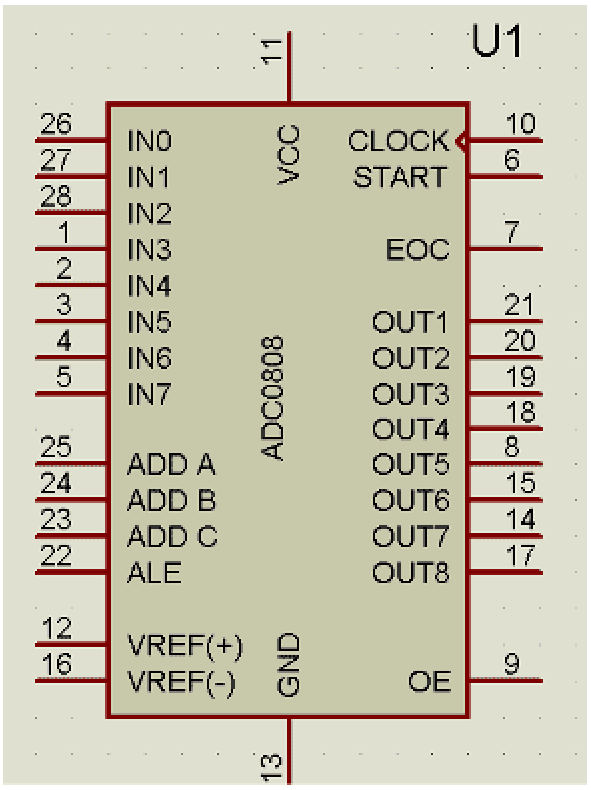
LCD INTERFACING USING 8051



ADC is the Analog to Digital converter, which converts analog data into digital format; usually it is used to convert **analog voltage** into digital format. Analog signal has infinite no of values like a sine wave or our speech, ADC converts them into particular levels or states, which can be measured in numbers as a physical quantity. Instead of continuous conversion, ADC converts data periodically, which is usually known as sampling rate. **Telephone modem** is one of the examples of ADC, which is used for internet, it converts analog data into digital data, so that computer can understand, because computer can only understand Digital data. The major advantage, of using ADC is that, we noise can be efficiently eliminated from the original signal and digital signal can travel more efficiently than analog one. That's the reason that digital audio is very clear, while listening.

ADC0808/0809 is a monolithic CMOS device and microprocessor compatible control logic and has 28 pin which gives 8-bit value in output and 8- channel ADC input pins (IN0-IN7). Its resolution is 8 so it can encode the analog data into one of the **256 levels** (2^8). This device has three channel address line namely: ADDA, ADDB and ADDC for selecting channel. Below is the Pin Diagram for ADC0808:





ADC0808/0809 **requires a clock pulse** for conversion. We can provide it by using oscillator or by using microcontroller. In this project we have applied frequency by using microcontroller.

We can select the any input channel by using the Address lines, like we can select the input line IN0 by keeping all three address lines (ADDA, ADDB and ADDC) Low. If we want to select input channel IN2 then we need to keep ADDA, ADDB low and ADDC high. For selecting all the other input channels, have a look on the given table:

ADC Channel Name	**ADDC PIN**	**ADDB PIN**	**ADDA PIN**
IN0	LOW	LOW	LOW
IN1	LOW	LOW	HIGH
IN2	LOW	HIGH	LOW
IN3	LOW	HIGH	HIGH
IN4	HIGH	LOW	LOW
IN5	HIGH	LOW	HIGH
IN6	HIGH	HIGH	LOW
IN7	HIGH	HIGH	HIGH

```
# include<reg51.h>
#include<stdio.h>
sbit ale=P3^3;
sbit oe=P3^6;
sbit sc=P3^4;
sbit eoc=P3^5;
sbit clk=P3^7;
sbit ADDA=P3^0;  //Address pins for selecting input channels.
sbit ADDB=P3^1;
sbit ADDC=P3^2;
#define lcdport P2  //lcd
sbit rs=P2^0;
sbit rw=P2^2;
sbit en=P2^1;
#define input_port P1  //ADC
int result[3],number;
```

```
# include<reg51.h>
#include<stdio.h>
sbit ale=P3^3;
sbit oe=P3^6;
sbit sc=P3^4;
sbit eoc=P3^5;
sbit clk=P3^7;
sbit ADDA=P3^0;  //Address pins for selecting input channels.
sbit ADDB=P3^1;
sbit ADDC=P3^2;
#define lcdport P2  //lcd
sbit rs=P2^0;
sbit rw=P2^2;
sbit en=P2^1;
#define input_port P1  //ADC
```

```
int result[3],number;
void timer0() interrupt 1  // Function to generate clock of frequency 500KHZ using Timer 0
interrupt.
{
clk=~clk;
}
void delay(unsigned int count)
{
int i,j;
for(i=0;i<count;i++)
  for(j=0;j<100;j++);
}
void daten()
{
    rs=1;
    rw=0;
    en=1;
    delay(1);
    en=0;
}
void lcd_data(unsigned char ch)
{
    lcdport=ch & 0xF0;
    daten();
    lcdport=ch<<4 & 0xF0;
    daten();
}
void cmden(void)
{
    rs=0;
    en=1;
    delay(1);
    en=0;
}
void lcdcmd(unsigned char ch)
{
    lcdport=ch & 0xf0;
    cmden();
    lcdport=ch<<4 & 0xF0;
    cmden();
}
lcdprint(unsigned char *str)  //Function to send string data to LCD.
{
    while(*str)
    {
        lcd_data(*str);
        str++;
    }
}
void lcd_ini()  //Function to inisialize the LCD
{
    lcdcmd(0x02);
    lcdcmd(0x28);
    lcdcmd(0x0e);
    lcdcmd(0x01);
}
void show()
{
   sprintf(result,"%d",number);
   lcdprint(result);
   lcdprint("  ");
}
void read_adc()
{
   number=0;
   ale=1;
   sc=1;
   delay(1);
   ale=0;
   sc=0;
   while(eoc==1);
   while(eoc==0);
   oe=1;
   number=input_port;
   delay(1);
   oe=0;
}
void adc(int i)  //Function to drive ADC
{
switch(i)
  {
   case 0:
    ADDC=0;  // Selecting input channel IN0 using address lines
    ADDB=0;
    ADDA=0;
    lcdcmd(0xc0);
    read_adc();
    show();
    break;
   case 1:
    ADDC=0;  // Selecting input channel IN1 using address lines
    ADDB=0;
    ADDA=1;
    lcdcmd(0xc6);
    read_adc();
    show();
   break;
   case 2:
    ADDC=0;  // Selecting input channel IN2 using address lines
    ADDB=1;
    ADDA=0;
    lcdcmd(0xcc);
    read_adc();
    show();
    break;
  }
}
void main()
{
 int i=0;
 eoc=1;
 ale=0;
 oe=0;
 sc=0;
 TMOD=0x02;
 TH0=0xFD;
lcd_ini();
lcdprint(" ADC 0808/0809  ");
lcdcmd(192);
lcdprint("  Interfacing   ");
delay(500);
lcdcmd(1);
lcdprint("Circuit Digest ");
lcdcmd(192);
lcdprint("System Ready...    ");
delay(500);
lcdcmd(1);
lcdprint("Ch1   Ch2   Ch3 ");
 IE=0x82;
 TR0=1;
while(1)
{
   for(i=0;i<3;i++)
   {
     adc(i);
     number=0;
   }
}
}
```

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: binary weighted and 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)

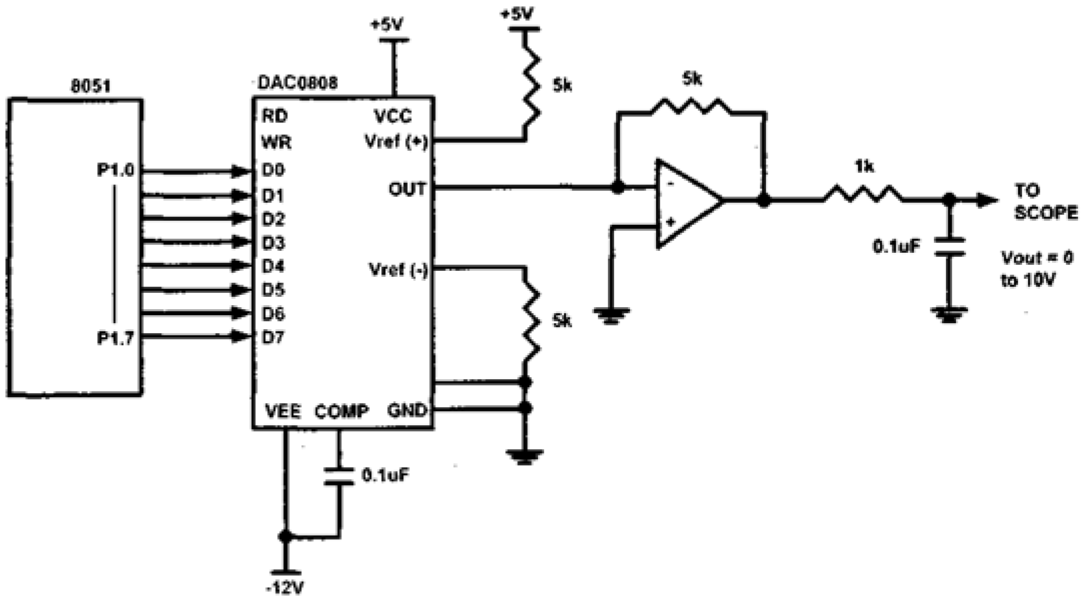
In the MC1408 (DAC0808), the digital inputs are converted to current (I_{out}), and by connecting a resistor to the I_{out}pin, we convert the result to voltage.

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_{re}f), and is as follows:

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

where DO is the LSB, D7 is the MSB for the inputs, and I_{re}f is the input current that must be applied to pin 14. The I_{re}f current is generally set to 2.0 mA. Figure 13-18 shows the generation of current reference (setting I_{re}f = 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



To find the value sent to the DAC for various angles, we simply multiply the V_{out} voltage by 25.60 because there are 256 steps and full-scale V_{out} is 10 volts. Therefore, 256 steps /10 V = 25.6 steps per volt. To further clarify this, look at the following code. This program sends the values to the

DAC continuously (in an infinite loop) to produce a crude sine wave. See Figure 13-19.

```
AGAIN:      MOV DPTR,#TABLE
            MOV R2,#COUNT
BACK:       CLR A
            MOVC A,@A+DPTR
            MOV P1,A
            INC DPTR
            DJNZ R2,BACK
            SJMP AGAIN
            ORG 300
TABLE:      DB 128,192,238,255,238,192 ;see Table 13-7
            DB 128,64,17,0,17,64,128
```

UNIT-IV

INTRODUCTION TO REAL – TIME OPERATING SYSTEMS

- **Introduction**
    - A more complex software architecture is needed to handle multiple tasks, coordination, communication, and interrupt handling – an RTOS architecture
    - **Distinction:**
        - Desktop OS – OS is in control at all times and runs applications, OS runs in different address space
        - RTOS – OS and embedded software are integrated, ES starts and activates the OS – both run in the same address space (RTOS is less protected)
        - RTOS includes only service routines needed by the ES application
        - RTOS vendors: VsWorks (we got it!), VTRX, Nucleus, LynxOS, uC/OS
        - Most conform to POSIX (IEEE standard for OS interfaces)
        - Desirable RTOS properties: use less memory, application programming interface, debugging tools, support for variety of microprocessors, already-debugged network drivers

What Is an O.S?

- ☐ A piece of software
- ☐ It provides tools to manage (for embedded systems)
    - Processes, (or tasks)
    - Memory space

What Is an Operating System?

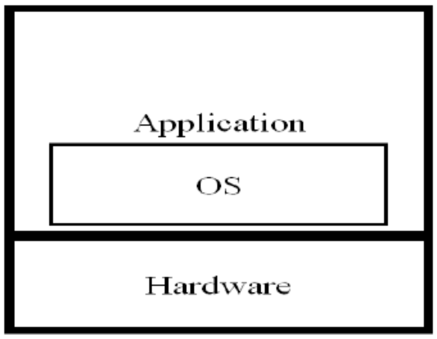
- ☐ What? It is a program (software) that acts as an intermediary between a user of a computer and the computer hardware.
- ☐ Why? Make the use of a computer CONVENIENT and EFFICIENT.

What Is an Operating System?*For an Embedded System*

- ☐ Provides software tools for a convenient and prioritized control of tasks.

- Provides tools for task (process) synchronization.
- Provides a simple memory management system

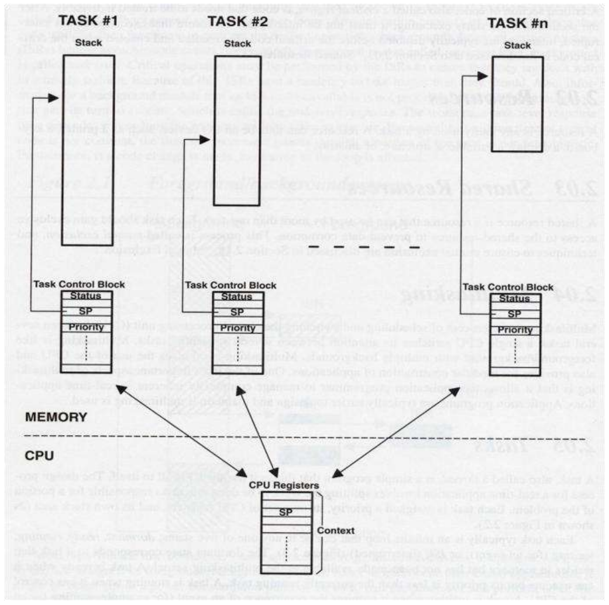
Abstract View of A System (*Embedded System*):



Process/Task Concept:

- Process is a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

Multitasking:



Process/Task Concept:

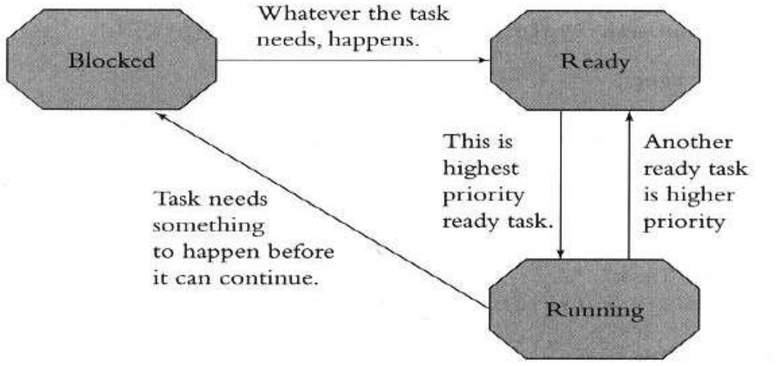
- Task States:
  - Running: Instructions are being executed
  - Ready: The process is waiting to be assigned to a process
  - Blocked: The process is waiting for some event to occur
  - terminated: The process has finished execution
  - new: The process is being created

Task states:

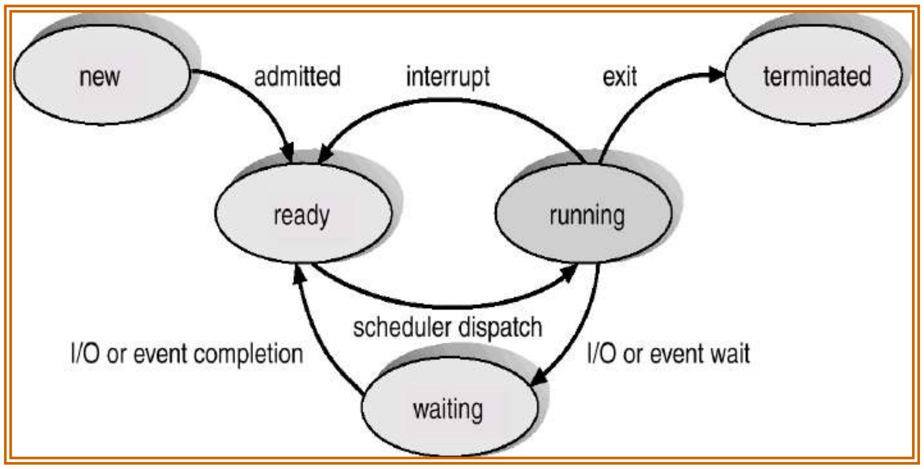
Tasks and Task States:

- A task – a simple subroutine
- ES application makes calls to the RTOS functions to start tasks, passing to the OS, start address, stack pointers, etc. of the tasks
- Task States:
  - Running
  - Ready (possibly: suspended, pended)
  - Blocked (possibly: waiting, dormant, delayed)
  - [Exit]
  - Scheduler – schedules/shuffles tasks between Running and Ready states
  - Blocking is self-blocking by tasks, and moved to Running state via other tasks' interrupt signaling (when block-factor is removed/satisfied)
  - When a task is unblocked with a higher priority over the 'running' task, the scheduler 'switches' context immediately (for all pre-emptive RTOSs)

Figure 6.1 Task States


Blocked
Whatever the task needs, happens.
Ready
This is highest priority ready task.
Another ready task is higher priority
Task needs something to happen before it can continue.
Running

Task State Transitions:



Tasks – 1:

- Issue – Scheduler/Task signal exchange for block-unblock of tasks via function calls
- Issue – All tasks are blocked and scheduler idles forever (not desirable!)
- Issue – Two or more tasks with same priority levels in Ready state (time-slice, FIFO)
- Example: scheduler switches from processor-hog vLevelsTask to vButtonTask (on user interruption by pressing a push-button), controlled by the main() which initializes the RTOS, sets priority levels, and starts the RTOS

Figure 6.2 Uses for Tasks

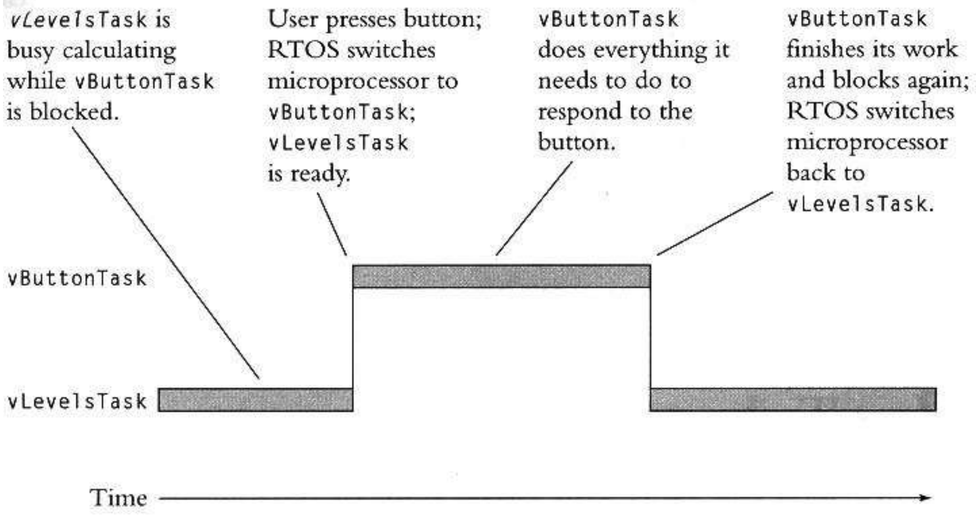
```
/* "Button Task" */
void vButtonTask (void)    /* High priority */
{
   while (TRUE)
   {
      !! Block until user pushes a button
      !! Quick: respond to the user
   }
}

/* "Levels Task" */
void vLevelsTask (void)    /* Low priority */
{
   while (TRUE)
   {
      !! Read levels of floats in tank
      !! Calculate average float level
```

(continued)

Figure 6.2 *(continued)*

```
      !! Do some interminable calculation
      !! Do more interminable calculation
      !! Do yet more interminable calculation

      !! Figure out which tank to do next
   }
}
```

Figure 6.3 Microprocessor Responds to a Button under an RTOS

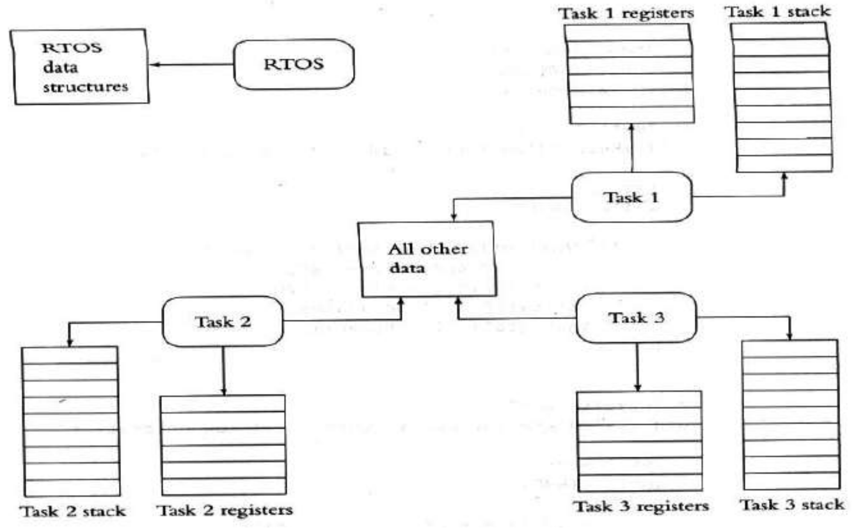


vLevelsTask is busy calculating while vButtonTask is blocked.
User presses button; RTOS switches microprocessor to vButtonTask; vLevelsTask is ready.
vButtonTask does everything it needs to do to respond to the button.
vButtonTask finishes its work and blocks again; RTOS switches microprocessor back to vLevelsTask.
vButtonTask
vLevelsTask
Time

Tasks and Data:

- Each tasks has its won context - not shared, private registers, stack, etc.
- In addition, several tasks share common data (via global data declaration; use of ‘extern’ in one task to point to another task that declares the shared data
- Shared data caused the ‘shared-data problem’ without solutions discussed in Chp4 or use of ‘Reentrancy’ characterization of functions

Figure 6.5 Data in an RTOS-Based Real-Time System



Semaphores and Shared Data – A new tool for atomicity

- Semaphore – a variable/lock/flag used to control access to shared resource (to avoid shared-data problems in RTOS)
- Protection at the start is via primitive function, called ***take***, indexed by the semaphore
- Protection at the end is via a primitive function, called ***release***, also indexed similarly
- Simple semaphores – Binary semaphores are often adequate for shared data problems in RTOS

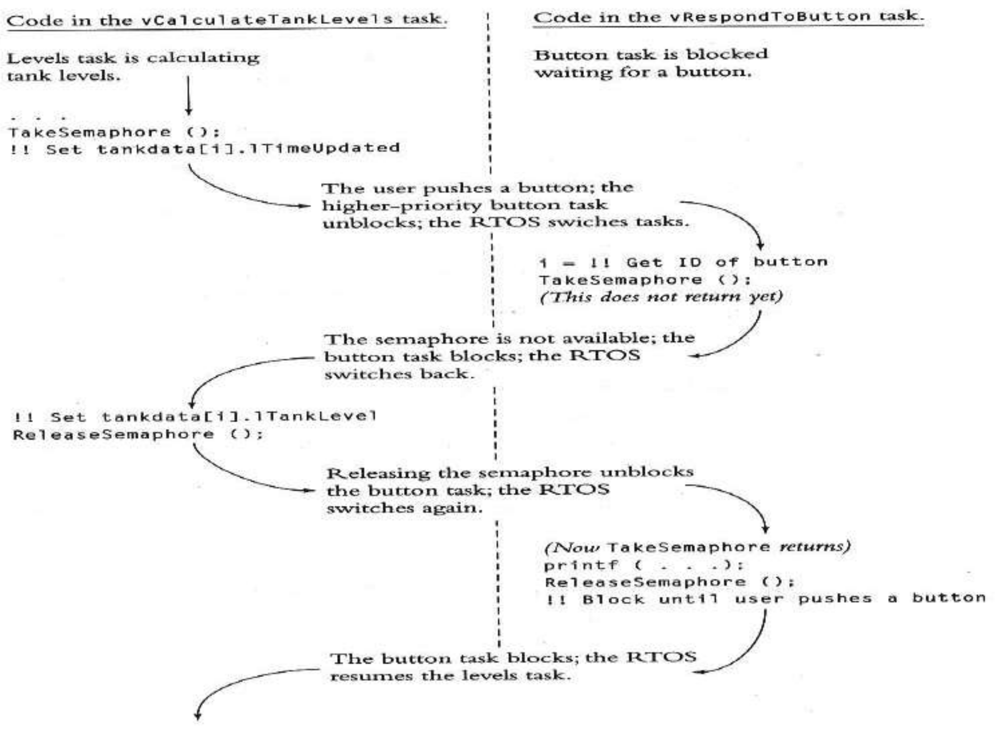


Figure 6.13 Execution Flow with Semaphores

Code in the vCalculateTankLevels task.

Levels task is calculating tank levels.

```
. . .
TakeSemaphore ();
!! Set tankdata[i].lTimeUpdated
```

The user pushes a button; the higher-priority button task unblocks; the RTOS swiches tasks.

Code in the vRespondToButton task.

Button task is blocked waiting for a button.

```
i = !! Get ID of button
TakeSemaphore ();
```

(This does not return yet)

The semaphore is not available; the button task blocks; the RTOS switches back.

```
!! Set tankdata[i].lTankLevel
ReleaseSemaphore ();
```

Releasing the semaphore unblocks the button task; the RTOS switches again.

(Now TakeSemaphore *returns)*

```
printf ( . . .);
ReleaseSemaphore ();
!! Block until user pushes a button
```

The button task blocks; the RTOS resumes the levels task.

Semaphores and Shared Data – 1:

- RTOS Semaphores & Initializing Semaphores
- Using binary semaphores to solve the ‘tank monitoring’ problem
- The nuclear reactor system: The issue of initializing the semaphore variable in a dedicated task (not in a ‘competing’ task) before initializing the OS – timing of tasks and priority overrides, which can undermine the effect of the semaphores
- Solution: Call OSSemInit() before OSInit()

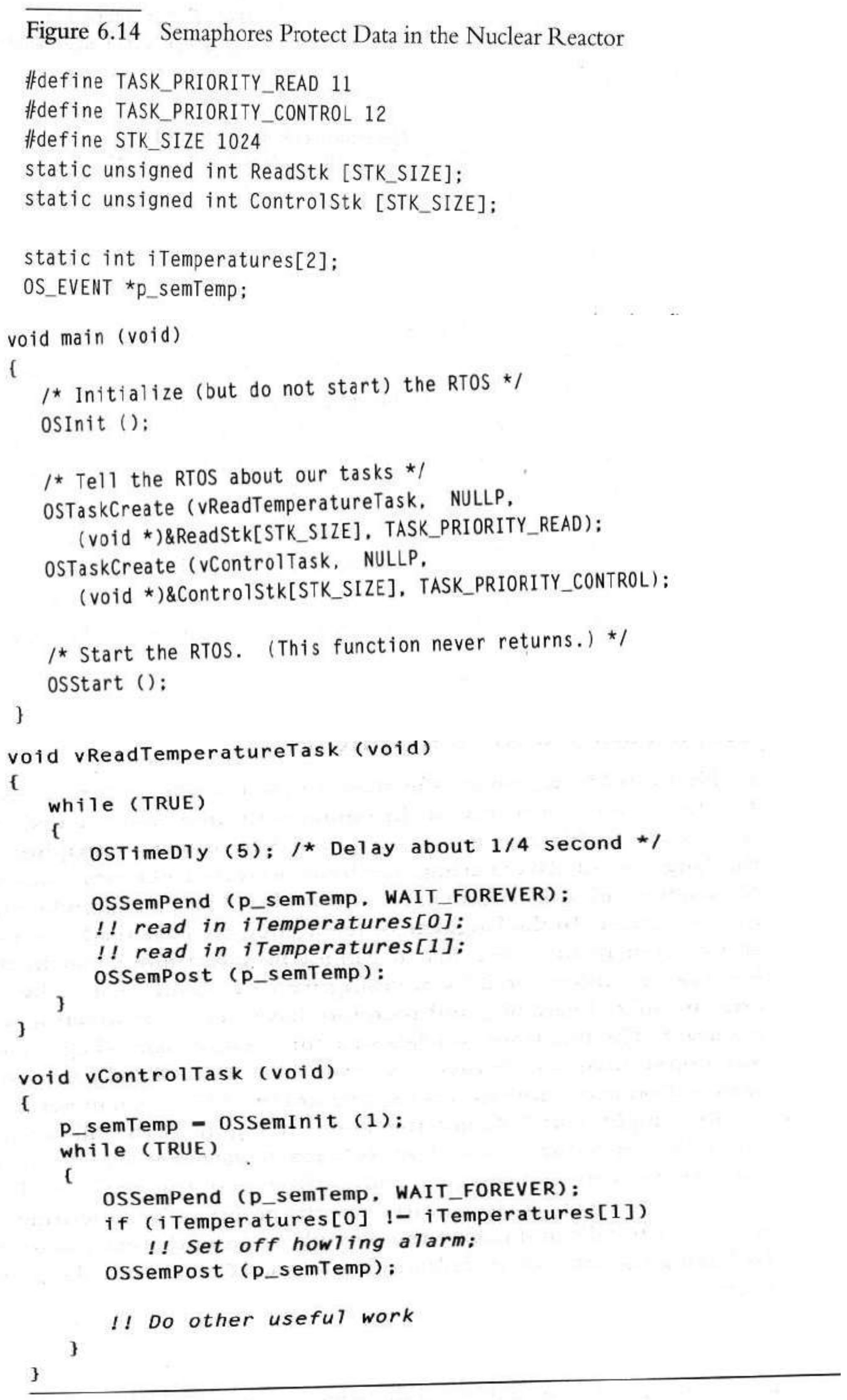


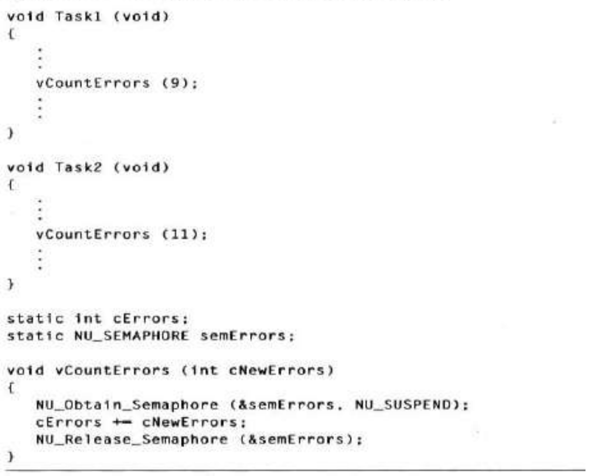
Figure 6.14 Semaphores Protect Data in the Nuclear Reactor

```
 #define TASK_PRIORITY_READ 11
 #define TASK_PRIORITY_CONTROL 12
 #define STK_SIZE 1024
 static unsigned int ReadStk [STK_SIZE];
 static unsigned int ControlStk [STK_SIZE];

 static int iTemperatures[2];
 OS_EVENT *p_semTemp;

void main (void)
{
   /* Initialize (but do not start) the RTOS */
   OSInit ();

   /* Tell the RTOS about our tasks */
   OSTaskCreate (vReadTemperatureTask, NULLP,
      (void *)&ReadStk[STK_SIZE], TASK_PRIORITY_READ);
   OSTaskCreate (vControlTask, NULLP,
      (void *)&ControlStk[STK_SIZE], TASK_PRIORITY_CONTROL);

   /* Start the RTOS. (This function never returns.) */
   OSStart ();
}

void vReadTemperatureTask (void)
{
   while (TRUE)
   {
      OSTimeDly (5); /* Delay about 1/4 second */

      OSSemPend (p_semTemp, WAIT_FOREVER);
      !! read in iTemperatures[0];
      !! read in iTemperatures[1];
      OSSemPost (p_semTemp);
   }
}

void vControlTask (void)
{
   p_semTemp = OSSemInit (1);
   while (TRUE)
   {
      OSSemPend (p_semTemp, WAIT_FOREVER);
      if (iTemperatures[0] != iTemperatures[1])
         !! Set off howling alarm;
      OSSemPost (p_semTemp);

      !! Do other useful work
   }
}
```

Semaphores and Shared Data – 2

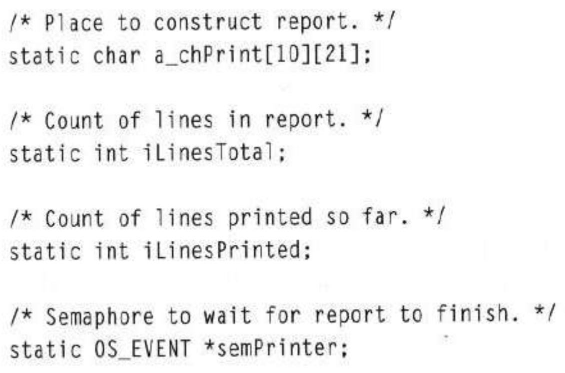
- Reentrancy, Semaphores, Multiple Semaphores, Device Signaling,
- a reentrant function, protecting a shared data, cErrors, in critical section
- Each shared data (resource/device) requires a separate semaphore for individual protection, allowing multiple tasks and data/resources/devices to be shared exclusively, while allowing efficient implementation and response time
- example of a printer device signaled by a report-buffering task, via semaphore signaling, on each print of lines constituting the formatted and buffered report

Figure 6.15 Semaphores Make a Function Reentrant



```
void Task1 (void)
{
   :
   vCountErrors (9);
   :
}

void Task2 (void)
{
   :
   vCountErrors (11);
   :
}

static int cErrors;
static NU_SEMAPHORE semErrors;

void vCountErrors (int cNewErrors)
{
   NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
   cErrors += cNewErrors;
   NU_Release_Semaphore (&semErrors);
}
```

Figure 6.16 Using a Semaphore as a Signaling Device



```
/* Place to construct report. */
static char a_chPrint[10][21];

/* Count of lines in report. */
static int iLinesTotal;

/* Count of lines printed so far. */
static int iLinesPrinted;

/* Semaphore to wait for report to finish. */
static OS_EVENT *semPrinter;
```

```
void vPrinterTask(void)
{
   BYTE byError;   /* Place for an error return. */
   Int wMsg;

   /* Initialize the semaphore as already taken. */
   semPrinter = OSSemInit(0);

   while (TRUE)
   {
      /* Wait for a message telling what report to format. */
      wMsg = (int) OSQPend (QPrinterTask, WAIT_FOREVER, &byError);

      !! Format the report into a_chPrint
      iLinesTotal = !! count of lines in the report

      /* Print the first line of the report */
      iLinesPrinted = 0;
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);

      /* Wait for print job to finish. */
      OSSemPend (semPrinter, WAIT_FOREVER, &byError);
   }
}
```

(continued)

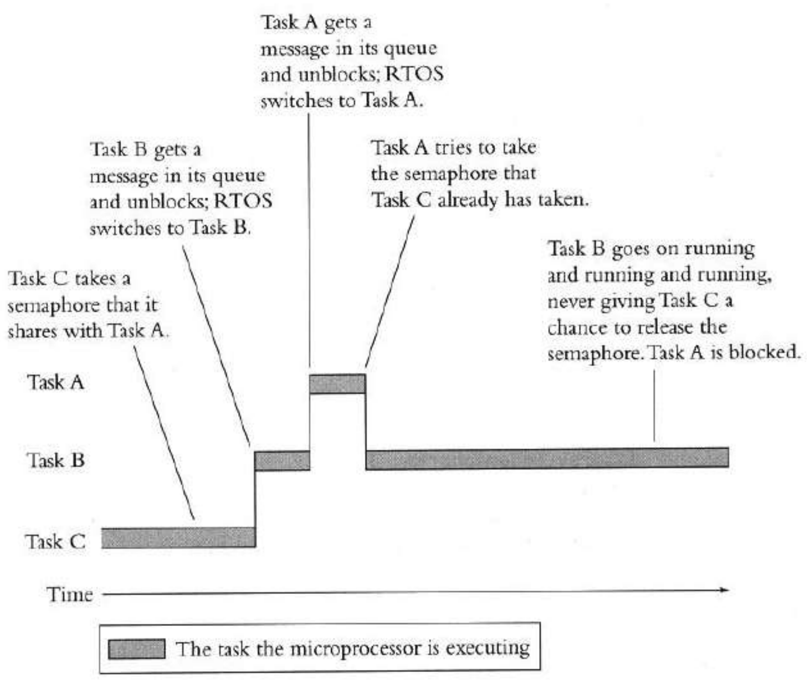
Figure 6.16 *(continued)*

```
void  vPrinterInterrupt (void)
{
   if (iLinesPrinted == iLinesTotal)
      /* The report is done.  Release the semaphore. */
      OSSemPost (semPrinter);

   else
      /* Print the next line. */
      vHardwarePrinterOutputLine (a_chPrint[iLinesPrinted++]);
}
```

semaphores and Shared Data – 3:

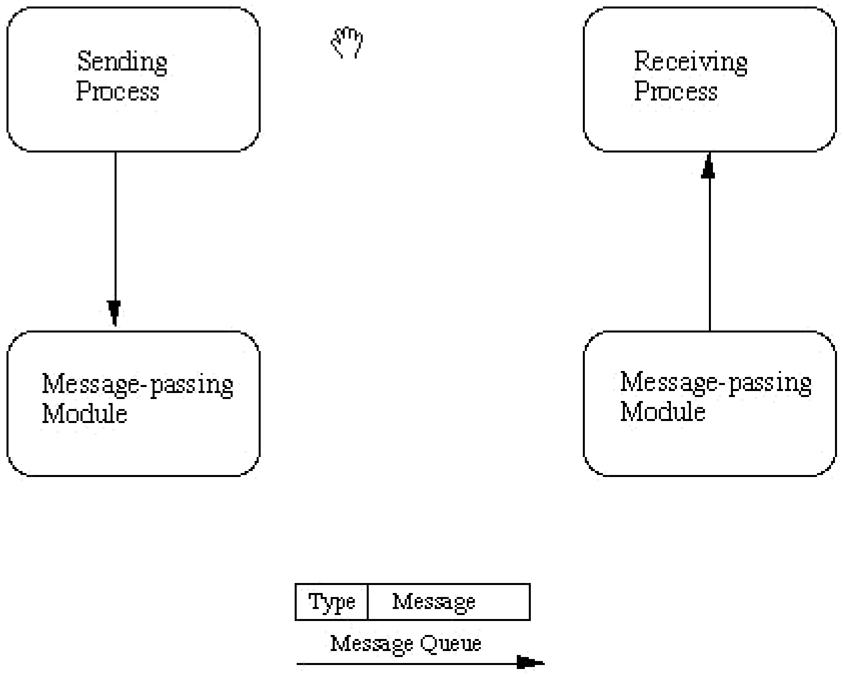
- Semaphore Problems – ‘Messing up’ with semaphores
  - The initial values of semaphores – when not set properly or at the wrong place
  - The ‘symmetry’ of takes and releases – must match or correspond – each ‘take’ must have a corresponding ‘release’ somewhere in the ES application
  - ‘Taking’ the wrong semaphore unintentionally (issue with multiple semaphores)
  - Holding a semaphore for too long can cause ‘waiting’ tasks’ deadline to be missed
  - Priorities could be ‘inverted’ and usually solved by ‘priority inheritance/promotion’

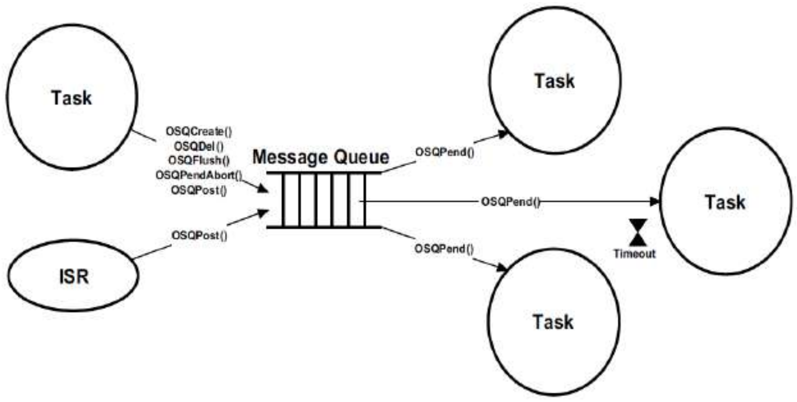
Figure 6.17 Priority Inversion



message queue :

Two (or more) processes can exchange information via access to a common system message queue. The *sending* process places via some (OS) message-passing module a message onto a queue which can be read by another process (Figure)Each message is given an identification or `type` so that processes can select the appropriate message. Process must share a common `key` in order to gain access to the queue in the first place (subject to other permissions -- see below).





Basic Message Passing IPC messaging lets processes send and receive messages, and queue messages for processing in an arbitrary order. Unlike the file byte-stream data flow of pipes, each IPC message has an explicit length. Messages can be assigned a specific type. Because of this, a server process can direct message traffic between clients on its queue by using the client process PID as the message type. For single-message transactions, multiple server processes can work in parallel on transactions sent to a shared message queue.

Before a process can send or receive a message, the queue must be initialized (through the `msgget` function see below) Operations to send and receive messages are performed by the `msgsnd()` and `msgrcv()` functions, respectively.

When a message is sent, its text is copied to the message queue. The `msgsnd()` and `msgrcv()` functions can be performed as either blocking or non-blocking operations. Non-blocking operations allow for asynchronous message transfer -- the process is not suspended as a result of sending or receiving a message. In blocking or synchronous message passing the sending process cannot continue until the message has been transferred or has even been acknowledged by a receiver. IPC signal and other mechanisms can be employed to implement such transfer. A blocked message operation remains suspended until one of the following three conditions occurs:

- The call succeeds.
- The process receives a signal.
- The queue is removed.

Initialising the Message Queue :

The `msgget()` function initializes a new message queue:

```
int msgget(key_t key, int msgflg)
```

It can also return the message queue ID (`msqid`) of the queue corresponding to the key argument. The value passed as the `msgflg` argument must be an octal integer with settings for the queue's permissions and control flags.

The following code illustrates the `msgget()` function.

```
#include <sys/ipc.h>;
#include <sys/msg.h>;

...


key_t key; /* key to be passed to msgget() */
int msgflg /* msgflg to be passed to msgget() */
int msqid; /* return value from msgget() */

...
key = ...
msgflg = ...

if ((msqid = msgget(key, msgflg)) == &ndash;1)
  {
    perror("msgget: msgget failed");
    exit(1);
   } else
    (void) fprintf(stderr, &ldquo;msgget succeeded");
```

Mailbox:

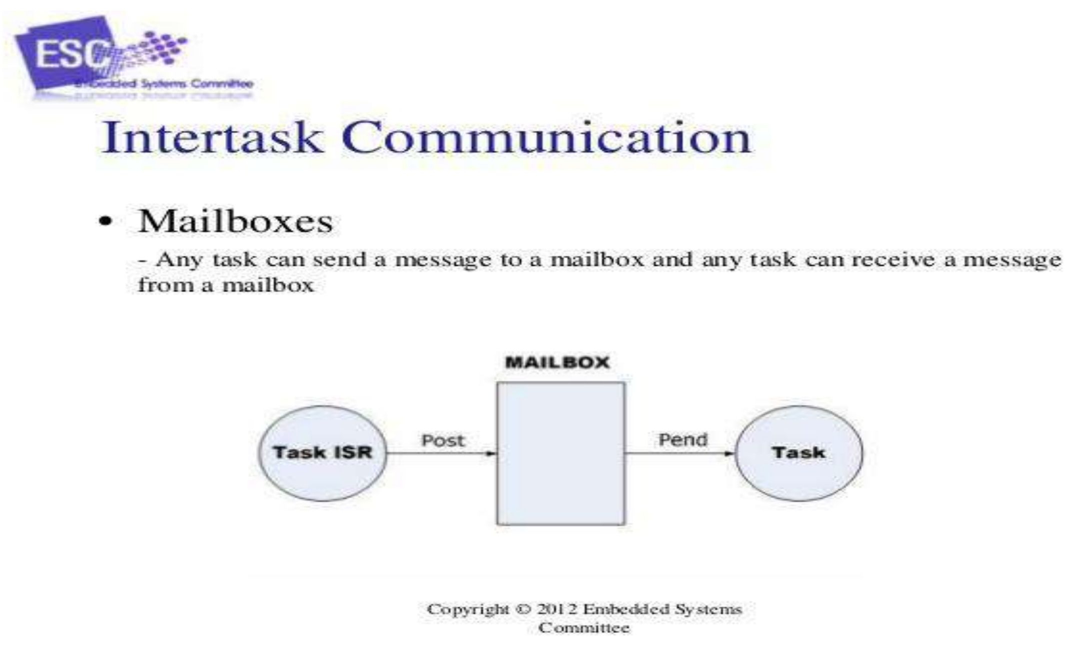
- Mailbox (for message) is an IPC through a message-block at an OS that can be used only by a single destined task.
-
- A task on an OS function call puts (means post and also send) into the mailbox nly a pointer to a mailbox message

- Mailbox message may also include a header to identify the message-type specification.

Mailbox IPC features:

•OS provides for inserting and deleting message into the mailbox message- pointer. Deleting eans message-pointer pointing to Null.

•Each mailbox for a message need initialization (creation) before using the functions in the scheduler for the message queue and message pointer pointing to null

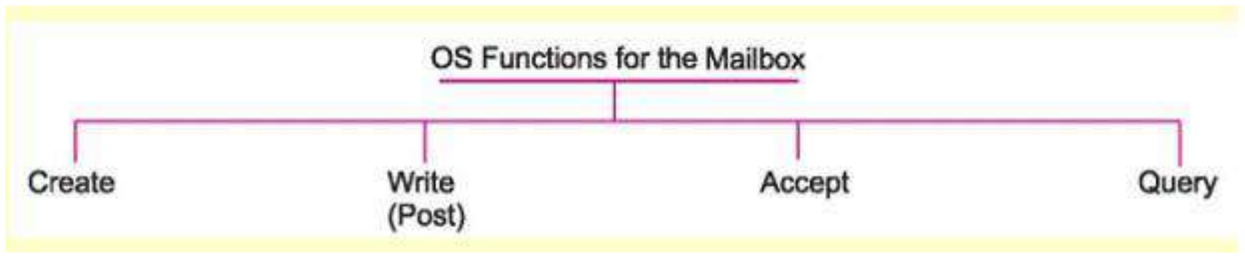


Intertask Communication

- Mailboxes
  - Any task can send a message to a mailbox and any task can receive a message from a mailbox

Copyright © 2012 Embedded Systems Committee

Mailbox Related Functions at the OS:



Pipe Function:

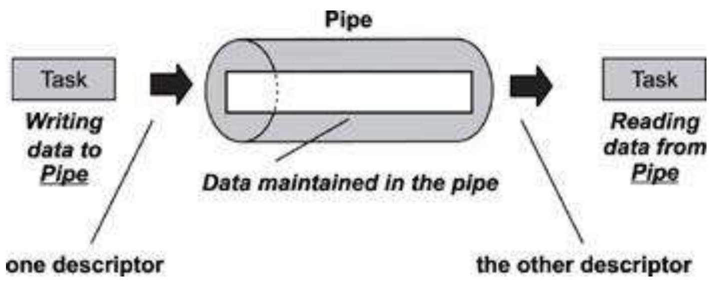
Pipe

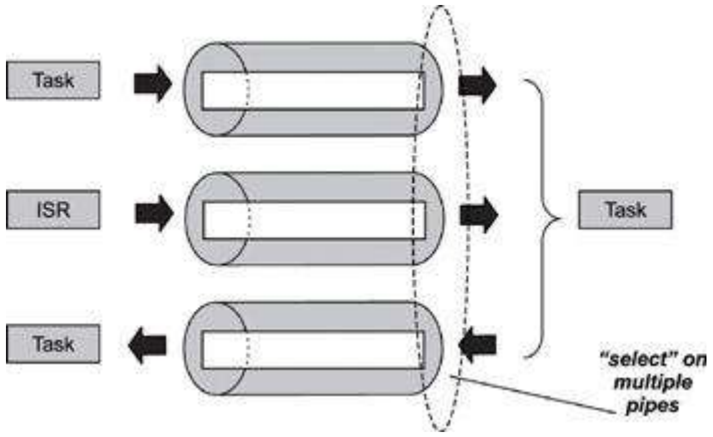
- Pipe is a device used for the interprocess communication
- Pipe has the functions create, connect and delete and functions similar to a device driver

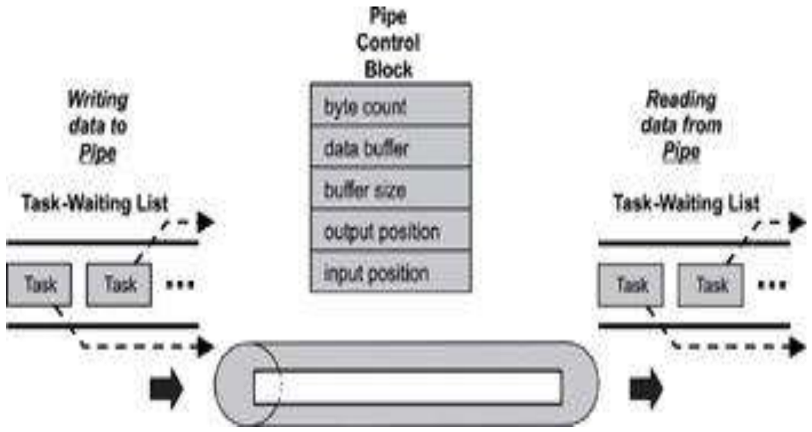
Writing and reading a Pipe:

• A message-pipe— a device for inserting (writing) and deleting (reading) from that between two given inter-connected tasks or two sets of tasks.

• Writing and reading from a pipe is like using a C command*fwrite with a file name* to write into a named file, and C command *fread with a file name*to read into a named







Pipe function calls:

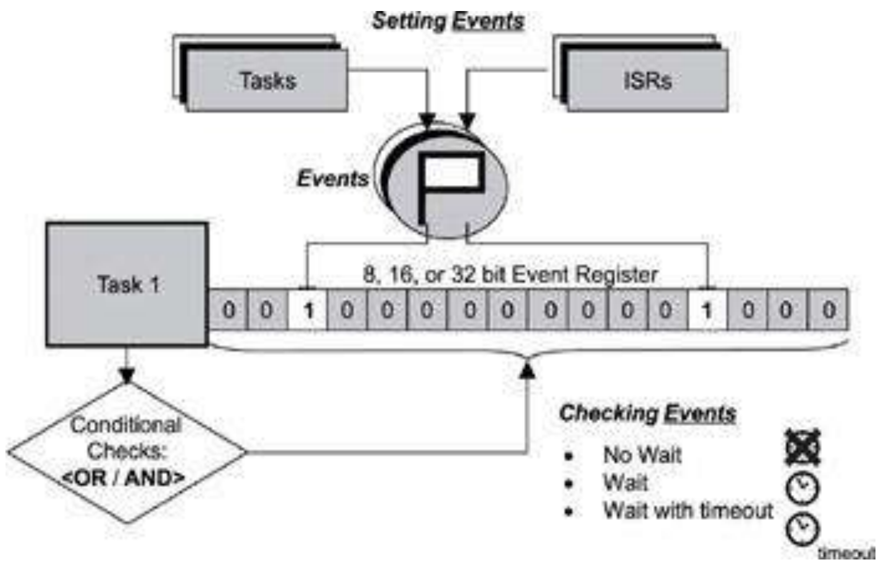
- Create a pipe
- Open pipe
- Close pipe
- Read from the pipe
- Write to the pipe

Event Functions:

- Wait for only one event (semaphore or mailboxmessage posting event)
- Event related OS functions can wait for number of events before initiating an action or wait for any of the predefined set of events
- Events for wait can be from different tasks or the ISRs

Event functions at OS:

Some OSes support and some don't support event functions for a group of event



Event registers function calls:

- Create an event register
- Delete an event register
- Query an event register
- Set an event register
- Clear an event register

- Each bit I an event register can be used to obtain the states of an event .
- A task can have an event register and other tasks can set/clear the bits in the event register

Signal:

- one way for messaging is to use an OS function signal ( ).
- Provided in Unix, Linux and several RTOSes.
- Unix and Linux OSes use signals profusely and have thirty-one different types of signals for the various events.
- A signal is the software equivalent of the flag at a register that sets on a hardware interrupt. Unless masked by a signal mask, the signal allows the execution of the Signal handling function and allows the handler to run just as a hardware interrupt allows the execution of an ISR
- Signal provides the shortest communication.

Signal management fuction calls:

- Install a signal handler
- Remove an installed signal handler
- Send a signal to another task
- Block a signal from being delivered
- Unblock a blocked signal
- Ignore a signal

Timers:

- Real time clock ─ system clock, on each tick SysClkIntr interrupts
- Based on each SysClkIntr interrupts─ there are number of OS timer functions
- Timer are used to message the elasped time of events for instance , the kernel has to keep track of different times

The following functions calls are provided to manage the timer

- Get time
- Set time
- Time delay(in system clock)
- Time delay(in sec.)
- Reset timer

Memory management:

Memory allocation:

- Memory allocation When a process is created, the memory manager allocates the memory addresses (blocks) to it by mapping the process address space.
- Threads of a process share the memory space of the process

Memory Managing Strategy for a system

- Fixed

- blocks allocation
- Dynamic
- blocks Allocation
- Dynamic Page
- Allocation
- Dynamic Data memory Allocation

Interrupt service routine (ISR):

- Interrupt is a hardware signal that informs the cpu that an important event has occurred when interrupt occured, cpu saves its content and jumps to the ISR
- In RTOS
    - Interrupt latency
    - Interrupt response
    - Interrupt recovery

Mutex:

Mutex standards for mutual exclusion ,mutex is the general mechanism used for both rsource synchronization as well as task synchronization

It has following mechanisms

- Disabling the scheduler
- Disabling the interrupts
- By test and set operations
- Using semaphore

UNIT-V
INTRODUCTION TO ADVANCED ARCHITECTURES

SHARC Processor Architectural Overview

Super Harvard Architecture

Analog Devices' 32-Bit Floating-Point SHARC® Processors are based on a Super Harvard architecture that balances exceptional core and memory performance with outstanding I/O throughput capabilities. This "Super" Harvard architecture extends the original concepts of separate program and data memory busses by adding an I/O processor with its associated dedicated busses. In addition to satisfying the demands of the most computationally intensive, real-time signal-processing applications, SHARC processors integrate large memory arrays and application-specific peripherals designed to simplify product development and reduce time to market.

The SHARC processor portfolio currently consists of four generations of products providing code-compatible solutions ranging from entry-level products priced at less than $10 to the highest performance products offering fixed- and floating-point computational power to 450 MHz/2700 MFLOPs. Irrespective of the specific product choice, all SHARC processors provide a common set of features and functionality useable across many signal processing markets and applications. This baseline functionality enables the SHARC user to leverage legacy code and design experience while transitioning to higher-performance, more highly integrated SHARC products.

Common Architectural Features

- 32/40-Bit IEEE Floating-Point Math
- 32-Bit Fixed-Point Multipliers with 64-Bit Product & 80-Bit Accumulation
- No Arithmetic Pipeline; All Computations Are Single-Cycle
- Circular Buffer Addressing Supported in Hardware
- 32 Address Pointers Support 32 Circular Buffers
- Six Nested Levels of Zero-Overhead Looping in Hardware
- Rich, Algebraic Assembly Language Syntax
- Instruction Set Supports Conditional Arithmetic, Bit Manipulation, Divide & Square Root, Bit Field Deposit and Extract
- DMA Allows Zero-Overhead Background Transfers at Full Clock Rate Without Processor Intervention

First Generation SHARC products offer performance to 66 MHz/ 198 MFLOPs and form the cornerstone of the SHARC processor family. Their easy-to-use Instruction Set Architecture that supports both 32-bit fixed-point and 32/40-bit floating data formats combined with large memory arrays and sophisticated communications ports make them suitable for a wide array of parallel processing applications including consumer audio, medical imaging, military, industrial, and instrumentation.

Second Generation SHARC products double the level of signal processing performance (100MHz / 600MFLOPs) offered by utilizing a Single-Instruction, Multiple-Data (SIMD) architecture. This hardware extension to first generation SHARC processors doubles the number of computational resources available to the system programmer. Second generation products contain dual multipliers, ALUs, shifters, and data register files - significantly increasing overall system performance in a variety of applications. This capability is especially relevant in consumer, automotive, and professional audio where the algorithms related to stereo channel processing can effectively utilize the SIMD architecture.

Third Generation SHARC products employ an enhanced SIMD architecture that extends CPU performance to 450 MHz/2700 MFLOPs. These products also integrate a variety of ROM memory configurations and audio-centric peripherals design to decrease time to market and reduce the overall bill of materials costs. This increased level of performance and peripheral integration allow third generation SHARC processors to be considered as single-chip solutions for a variety of audio markets.

The fourth generation of SHARC® Processors, now includes the ADSP-21486, ADSP-21487, ADSP-21488, ADSP-21489 and offers increased performance, hardware-based filter accelerators, audio and application-focused peripherals, and new memory configurations capable of supporting the latest surround-sound decoder algorithms. All devices are pin-compatible with each other and completely code-compatible with all prior SHARC Processors. These newest members of the fourth generation SHARC Processor family are based on a single-instruction, multiple-data (SIMD) core, which supports both 32-bit fixed-point and 32-/40-bit floating-point arithmetic formats making them particularly suitable for high-performance audio applications

Fourth-generation SHARC Processors also integrate application-specific peripherals designed to simplify hardware design, minimize design risks, and ultimately reduce time to market. Grouped together, and broadly named the Digital Applications Interface (DAI), these functional blocks may be connected to each other or to external pins via the software-programmable Signal Routing Unit (SRU). The SRU is an innovative architectural feature that enables complete and flexible routing amongst DAI blocks. Peripherals connected through the SRU include but are not limited to serial ports, IDP, S/PDIF Tx/Rx, and an 8-Channel asynchronous sample rate converter block. The fourth generation SHARC allows data from the serial ports to be directly transferred to external memory by the DMA controller. Other peripherals such as SPI,UART and Two-Wire Interface are routed through a Digital Peripheral Interface (DPI).

Instruction-level parallelism (ILP)

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel.

The amount of parallelism available within a basic block (a straight-line code sequence with no branches in and out except for entry and exit) is quite small. The average dynamic branch frequency in integer programs was measured to be about 15%, meaning that about 7 instructions execute between a pair of branches.

Since the instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than 7.

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism.

Example 1

```
for (i=1; i<=1000; i= i+1)
  x[i] = x[i] + y[i];
```

This is a parallel loop. Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little opportunity for overlap.

Example 2

```
for (i=1; i<=100; i= i+1){
  a[i] = a[i] + b[i];         //s1
  b[i+1] = c[i] + d[i];       //s2
}
```

Is this loop parallel? If not how to make it parallel?
Statement s1 uses the value assigned in the previous iteration by statement s2, so there is a loop-carried dependency between s1 and s2. Despite this dependency, this loop can be made parallel because the dependency is not circular:

- neither statement depends on itself;

- while s1 depends on s2, s2 does not depend on s1.

A loop is parallel unless there is a cycle in the dependencies, since the absence of a cycle means that the dependencies give a partial ordering on the statements.

To expose the parallelism the loop must be transformed to conform to the partial order. Two observations are critical to this transformation:

There is no dependency from s1 to s2. Then, interchanging the two statements will not affect the execution of s2.

On the first iteration of the loop, statement s1 depends on the value of b[1] computed prior to initiating the loop.
This allows us to replace the loop above with the following code sequence, which makes possible overlapping of the iterations of the loop:

```
a[1] = a[1] + b[1];
for (i=1; i<=99; i= i+1){
  b[i+1] = c[i] + d[i];
  a[i+1] = a[i+1] + b[i+1];
}
```

```
b[101] = c[100] + d[100];
```

Example 3

```
for (i=1; i<=100; i= i+1){
  a[i+1] = a[i] + c[i];       //S1
  b[i+1] = b[i] + a[i+1];     //S2
}
```

This loop is not parallel because it has cycles in the dependencies, namely the statements S1 and S2 depend on themselves!

There are a number of techniques for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop.

An important alternative method for exploiting loop-level parallelism is the use of vector instructions on a vector processor, which is not covered by this tutorial.

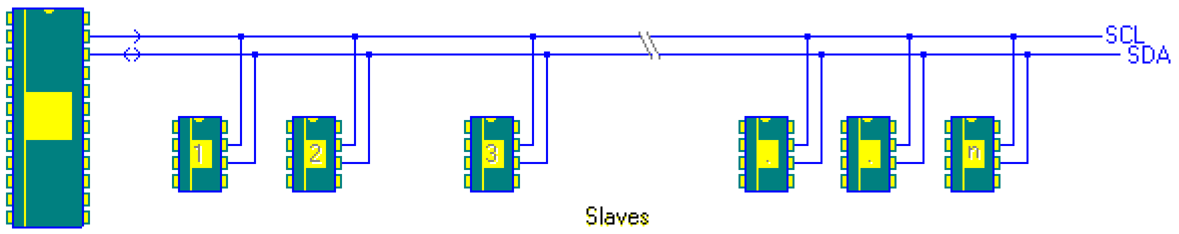
I2C Bus Protocol

The I2C bus physically consists of 2 active wires and a ground connection. The active wires, called SDA and SCL, are both bi-directional. SDA is the Serial Data line, and SCL is the Serial CLock line.

Every device hooked up to the bus has its own unique address, no matter whether it is an MCU, LCD driver, memory, or ASIC. Each of these chips can act as a receiver and/or transmitter, depending on the functionality. Obviously, an LCD driver is only a receiver, while a memory or I/O chip can be both transmitter and receiver.

The I2C bus is a multi-master bus. This means that more than one IC capable of initiating a data transfer can be connected to it. The I2C protocol specification states that the IC that initiates a data transfer on the bus is considered the Bus Master. Consequently, at that time, all the other ICs are regarded to be Bus Slaves.

As bus masters are generally microcontrollers, let's take a look at a general 'inter-IC chat' on the bus. Let’s consider the following setup and assume the MCU wants to send data to one of its slaves (also see here for more information; click here for information on how to receive data from a slave).



First, the MCU will issue a START condition. This acts as an 'Attention' signal to all of the connected devices. All ICs on the bus will listen to the bus for incoming data.

Then the MCU sends the ADDRESS of the device it wants to access, along with an indication whether the access is a Read or Write operation (Write in our example). Having received the address, all IC's will compare it with their own address. If it doesn't match, they simply wait until the bus is released by the stop condition (see below). If the address matches, however, the chip will produce a response called the ACKNOWLEDGE signal.

Once the MCU receives the acknowledge, it can start transmitting or receiving DATA. In our case, the MCU will transmit data. When all is done, the MCU will issue the STOP condition. This is a signal that the bus has been released and that the connected ICs may expect another transmission to start any moment.

We have had several states on the bus in our example:

START, ADDRESS, ACKNOWLEDGE, DATA, STOP. These are all unique conditions on the bus. Before we take a closer look at these bus conditions we need to understand a bit about the physical structure and hardware of the bus.

Controller Area Network (CAN) interface in embedded systems:

CAN or Controller Area Network or CAN-bus is an ISO standard computer network protocol and bus standard, designed for microcontrollers and devices to communicate with each other without a host computer. Designed earlier for industrial networking but recently more adopted to automotive applications, CAN have gained widespread popularity for embedded control in the areas like industrial automation, automotives, mobile machines, medical, military and other harsh environment network applications.

Development of the CAN-bus started originally in 1983 at Robert Bosch GmbH. The protocol was officially released in 1986. And the first CAN controller chips, produced by Intel and Philips, introduced in the market in the year of 1987.

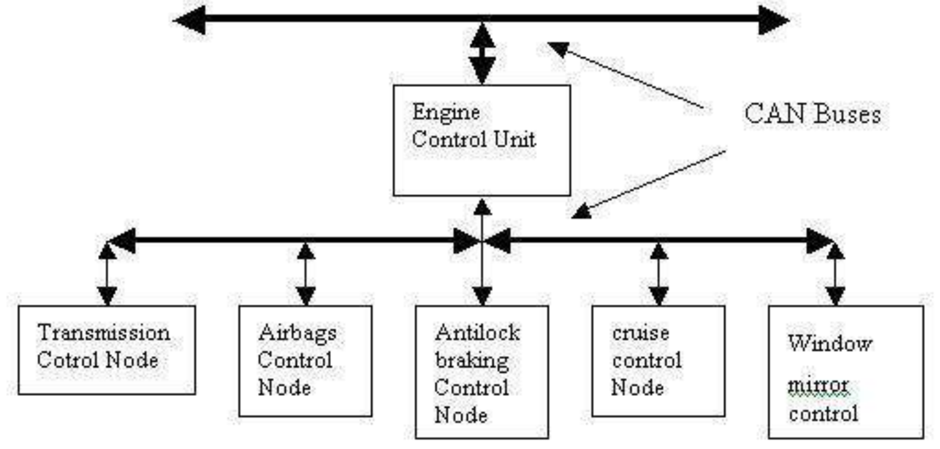
Introduction:
The CAN is a "broadcast" type of bus. That means there is no explicit address in the messages. All the nodes in the network are able to pick-up or receive all transmissions. There is no way to send a message to just a specific node. To be more specific, the messages transmitted from any node on a CAN bus does not contain addresses of either the transmitting node, or of any intended receiving node. Instead, an identifier that is unique throughout the network is used to label the content of the message. Each message carries a numeric value, which controls its priority on the bus, and may also serve as an identification of the contents of the message. And each of the receiving nodes performs an acceptance test or provides local filtering on the identifier to determine whether the message, and thus its content, is relevant to that particular node or not, so that each node may react only on the intended messages. If the message is relevant, it will be processed; otherwise it is ignored.

How do they communicate?

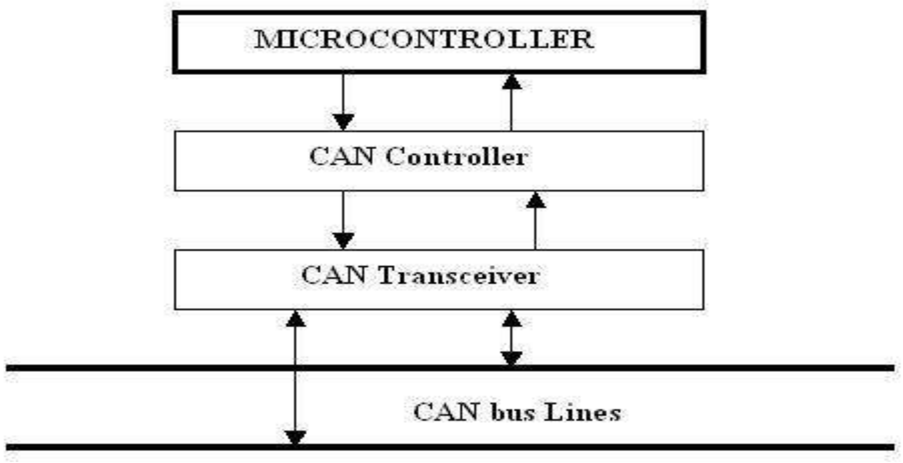
If the bus is free, any node may begin to transmit. But what will happen in situations where two or more nodes attempt to transmit message (to the CAN bus) at the same time. The identifier field, which is unique throughout the network helps to determine the priority of the message. A "non-destructive arbitration technique" is used to accomplish this, to ensure that the messages are sent in order of priority and that no messages are lost. The lower the numerical value of the identifier, the higher the priority. That means the message with identifier having more dominant bits (i.e. bit 0) will overwrite other nodes' less dominant identifier so that eventually (after the arbitration on the ID) only the dominant message remains and is received by all nodes.

As stated earlier, CAN do not use address-based format for communication, instead uses a message-based data format. Here the information is transferred from one location to another by sending a group of bytes at one time (depending on the order of priority). This makes CAN ideally suited in applications requiring a large number of short messages (e.g.: transmission of temperature and rpm information). by more than one location and system-wide data consistency is mandatory. (The traditional networks such as USB or Ethernet are used to send large blocks of data, point-to-point from node A to node B under the supervision of a central bus master).

Let us now try to understand how these nodes are interconnected physically, by pointing out some examples. A modern automobile system will have many electronic control units for various subsystems (fig1-a). Typically the biggest processor will be the engine control unit (or the host processor). The CAN standard even facilitates the subsystem to control actuators or receive signals from sensors. A CAN message never reaches these devices directly, but instead a host-processor and a CAN Controller (with a CAN transciever) is needed between these devices and the bus. (In some cases, the network need not have a controller node; each node can easily be connected to the main bus directly.)
The CAN Controller stores received bits (one by one) from the bus until an entire message block is available, that can then be fetched by the host processor (usually after the CAN Controller has triggered an interrupt). The Can transciever adapts signal levels from the bus, to levels that the CAN Controller expects and also provides a protective circuitry for the CAN Controller. The host-processor decides what the received messages mean, and which messages it wants to transmit itself.



It is likely that the more rapidly changing parameters need to be transmitted more frequently and, therefore, must be given a higher priority. How this high-priority is achieved? As we know, the priority of a CAN message is determined by the numerical value of its identifier. The numerical value of each message identifier (and thus the priority of the message) is assigned during the initial phase of system design. To determine the priority of messages (while communication), CAN uses the established method known as CSMA/CD with the enhanced capability of non-destructive bit-wise arbitration to provide collision resolution and to exploit the maximum available capacity of the bus. "Carrier Sense" describes the fact that a transmitter listens for a carrier wave before trying to send. That is, it tries to detect the presence of an encoded signal from another station before attempting to transmit. If a carrier is sensed, the node waits for the transmission in progress to finish before initiating its own transmission. "Multiple Access" describes the fact that multiple nodes send and receive on the same medium. All other nodes using the medium generally receive transmissions by one node. "Collision Detection" (CD) means that collisions are resolved through a bit-wise arbitration, based on a preprogrammed priority of each message in the identifier field of a message.



Let us now try to understand how the term "priority" becomes more important in the network. Each node can have one or more function. Different nodes may transmit messages at different times (Depends how the system is configured) based on the function(s) of each node. For example:

1) Only when a system failure (communication failure) occurs.
2) Continually, such as when it is monitoring the temperature.
3) A node may take action or transmit a message only when instructed by another node, such as when a fan controller is instructed to turn a fan on when the temperature-monitoring node has detected an elevated temperature.

Note:
When one node transmits the message, sometimes many nodes may accept the message and act on it (which is not a usual case). For example, a temperature-sensing node may send out temperature data that are accepted & acted on only by a temperature display node. But if the temperature sensor detects an over-temperature situation, then many nodes might act on the information.

CAN use "Non Return to Zero" (NRZ) encoding (with "bit-stuffing") for data communication on a "differential two wire bus". The two-wire bus is usually a twisted pair (shielded or unshielded). Flat pair (telephone type) cable also performs well but generates more noise itself, and may be more susceptible to external sources of noise.

Main Features:

a) A two-wire, half duplex, high-speed network system mainly suited for high-speed applications using "short messages". (The message is transmitted serially onto the bus, one bit after another in a specified format).

b) The CAN bus offers a high-speed communication rate up to 1 M bits / sec, for up to 40 feet, thus facilitating real-time control. (Increasing the distance may decrease the bit-rate).

c) With the message-based format and the error-containment followed, it's possible to add nodes to the bus without reprogramming the other nodes to recognize the addition or changing the existing hardware. This can be done even while the system is in operation. The new node will start receiving messages from the network immediately. This is called "hot-plugging"
d) Another useful feature built into the CAN protocol is the ability of a node to request information from other nodes. This is called a remote transmit request, or RTR.

e) The use of NRZ encoding ensures compact messages with a minimum number of transitions and high resilience to external disturbance.

f) CAN protocol can link up to 2032 devices (assuming one node with one identifier) on a single network. But accounting to the practical limitations of the hardware (transceivers), it may only link up to 110 nodes on a single network.

g) Has an extensive and unique error checking mechanisms.

h) Has High immunity to Electromagnetic Interference. Has the ability to self-diagnose & repair data errors.

i) Non-destructive bit-wise arbitration provides bus allocation on the basis of need, and delivers efficiency benefits that cannot be gained from either fixed time schedule allocation (e.g. Token ring) or destructive bus allocation (e.g. Ethernet.)

j) Fault confinement is a major advantage of CAN. Faulty nodes are automatically dropped from the bus. This helps to prevent any single node from bringing the entire network down, and thus ensures that bandwidth is always available for critical message transmission.

k) The use of differential signaling (a method of transmitting information electrically by means of two complementary signals sent on two separate wires) gives resistance to EMI & tolerance of ground offsets.

l) CAN is able to operate in extremely harsh environments. Communication can still continue (but with reduced signal to noise ratio) even if:

1. Either of the two wires in the bus is broken

2. Either wire is shorted to ground

3. Either wire is shorted to power supply.

CAN protocol Layers & message Frames:

Like any network applications, Can also follows layered approach to the system implementation. It conforms to the Open Systems Interconnection (OSI) model that is defined in terms of layers. The ISO 11898 (For CAN) architecture defines the lowest two layers of the seven layers OSI/ISO model as the data-link layer and physical layer. The rest of the layers (called Higher Layers) are left to be implemented by the system software developers (used to adapt and optimize the protocol on multiple media like twisted pair. Single wire, optical, RF or IR). The Higher Level Protocols (HLP) is used to implement the upper five layers of the OSI in CAN.

CAN use a specific message frame format for receiving and transmitting the data. The two types of frame format available are:

a) Standard CAN protocol or Base frame format
b) Extended Can or Extended frame format

The following figure (Fig 2) illustrates the standard CAN frame format, which consists of seven different bit-fields.

a) A Start of Frame (SOF) field - indicates the beginning of a message frame.

b) An Arbitration field, containing a message identifier and the Remote Transmission Request (RTR) bit. The RTR bit is used to discriminate between a transmitted Data Frame and a request for data from a remote node.

c) A Control Field containing six bits in which two reserved bits (r0 and r1) and a four bit Data Length Code (DLC). The DLC indicates the number of bytes in the Data Field that follows.

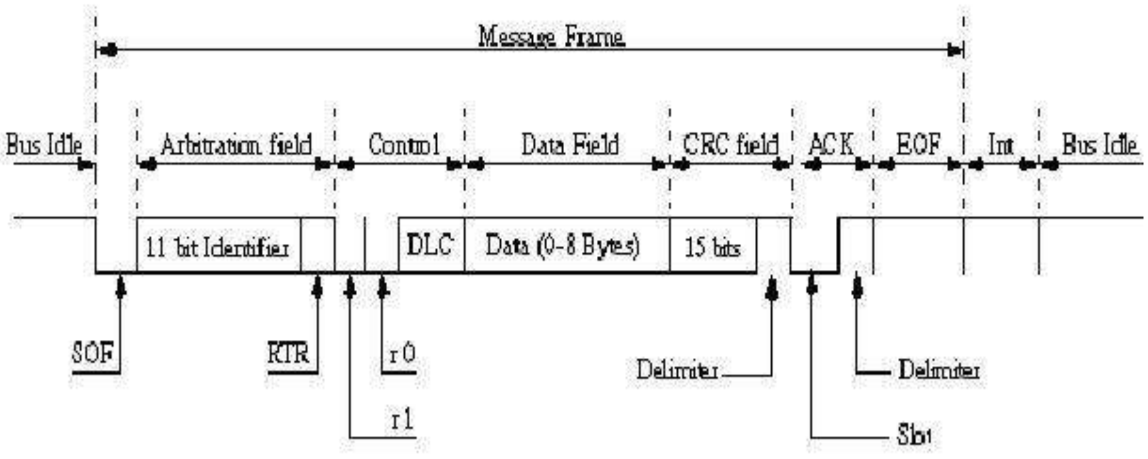
d) A Data Field, containing from zero to eight bytes.

e) The CRC field, containing a fifteen-bit cyclic redundancy check-code and a recessive delimiter bit.

f) The Acknowledge field, consisting of two bits. The first one is a Slot bit which is transmitted as recessive, but is subsequently over written by dominant bits transmitted from any node that successfully receives the transmitted message. The second bit is a recessive delimiter bit.

g) The End of Frame field, consisting of seven recessive bits.

An Intermission field consisting of three recessive bits is then added after the EOF field. Then the bus is recognized to be free.



The Extended Frame format provides the Arbitration field with two identifier bit fields. The first (the base ID) is eleven (11) bits long and the second field (the ID extension) is eighteen (18) bits long, to give a total length of twenty nine (29) bits. The distinction between the two formats is made using an Identifier Extension (IDE) bit. A Substitute Remote Request (SRR) bit is also included in the Arbitration Field.

Error detection & correction:
This mechanism is used for detecting errors in messages appearing on the CAN bus, so that the transmitter can retransmit message. The CAN protocol defines five different ways of detecting errors. Two of these works at the bit level, and the other three at the message level.

1. Bit Monitoring.

2. Bit Stuffing.

3. Frame Check.

4. Acknowledgement Check.

5. Cyclic Redundancy Check

1. Each transmitter on the CAN bus monitors (i.e. reads back) the transmitted signal level. If the signal level read differs from the one transmitted, a Bit Error is signaled. Note that no bit error is raised during the arbitration process.

2. When five consecutive bits of the same level have been transmitted by a node, it will add a sixth bit of the opposite level to the outgoing bit stream. The receivers will remove this extra bit. This is done to avoid excessive DC components on the bus, but it also gives the receivers an extra opportunity to detect errors: if more than five consecutive bits of the same level occurs on the bus, a Stuff Error is signaled.

3. Some parts of the CAN message have a fixed format, i.e. the standard defines exactly what levels must occur and when. (Those parts are the CRC Delimiter, ACK Delimiter, End of Frame, and also the Intermission). If a CAN controller detects an invalid value in one of these fixed fields, a Frame Error is signaled.

4. All nodes on the bus that correctly receives a message (regardless of their being "interested" of its contents or not) are expected to send a dominant level in the so-called Acknowledgement Slot in the message. The transmitter will transmit a recessive level here. If the transmitter can't detect a dominant level in the ACK slot, an Acknowledgement Error is signaled.

5. Each message features a 15-bit Cyclic Redundancy Checksum and any node that detects a different CRC in the message than what it has calculated itself will produce a CRC Error.

Error confinement:
Error confinement is a technique, which is unique to CAN and provides a method for discriminating between temporary errors and permanent failures in the communication network. Temporary errors may be caused by, spurious external conditions, voltage spikes, etc. Permanent failures are likely to be caused by bad connections, faulty cables, defective transmitters or receivers, or long lasting external disturbances.

Let us now try to understand how this works.

Each node along the bus will be having two error counters namely the transmit error counter (TEC) and the receive error counter (REC), which are used to be incremented and/or decremented in accordance with the error detected. If a transmitting node detects a fault, then it will increments its TEC faster than the listening nodes increments its REC because there is a good chance that it is the transmitter who is at fault.

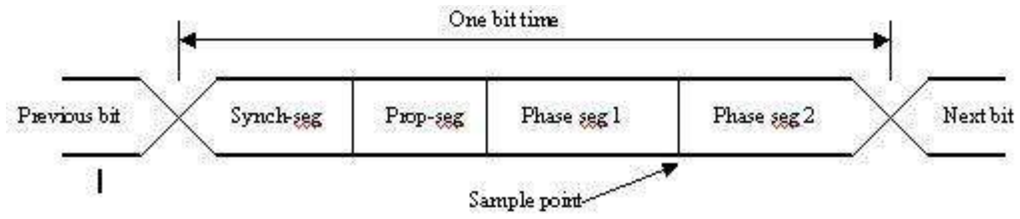
A node usually operates in a state known as "Error Active" mode. In this condition a node is fully functional and both the error count registers contain counts of less than 127. When any one of the two error counters raises above 127, the node will enter a state known as "Error Passive". That means, it will not actively destroy the bus traffic when it detects an error. The node which is in error passive mode can still transmit and receive messages but are restricted in relation to how they flag any errors that they may detect. When the Transmit Error Counter rises above 255, the node will enter the Bus Off state, which means that the node doesn't participate in the bus traffic at all. But the communications between the other nodes can continue unhindered.

To be more specific, an "Error Active" node will transmit "Active Error Flags" when it detects errors, an "Error Passive" node will transmit "Passive Error Flags" when it detects errors and a node, which is in "Bus Off" state will not transmit "anything" on the bus at all. The transmit errors give 8 error points, and receive errors give 1 error point. Correctly transmitted and/or received messages cause the counter(s) to decrease. The other nodes will detect the error caused by the Error Flag (if they haven't already detected the original error) and take appropriate action, i.e. discard the current message.

Let's assume that whenever node-A (for example) on a bus tries to transmit a message, it fails (for whatever reason). Each time this happens, it increases its Transmit Error Counter by 8 and transmits an Active Error Flag. Then it will attempt to retransmit the message and suppose the same thing happens again. When the Transmit Error Counter rises above 127 (i.e. after 16 attempts), node A goes Error Passive. It will now transmit passive error flags on the bus. A Passive Error Flag comprises 6 recessive bits, and will not destroy other bus traffic - so the other nodes will not hear the node-A complaining about bus errors. However, A continues to increase its TEC. When it rises above 255, node-A finally stops and goes to "Bus Off" state.

What does the other nodes think about node A? - For every active error flag that A transmitted, the other nodes will increase their Receive Error Counters by 1. By the time that A goes Bus Off, the other nodes will have a count in their Receive Error Counters that is well below the limit for Error Passive, i.e. 127. This count will decrease by one for every correctly received message. However, node A will stay bus off. Most CAN controllers will provide status bits and corresponding interrupts for two states: "Error Warning" (for one or both error counters are above 96) and "Bus Off".

Bit Timing and Synchronization:
The time for each bit in a CAN message frame is made up of four non-overlapping time segments as shown below.



The following points may be relevant as far as the "bit timing" is concerned.

1. Synchronization segment is used to synchronize the nodes on the bus. And it will always be of one quantum long.

2. One time quanta (which is also known as the system clock period) is the period of the local oscillator, multiplied by the value in the Baud Rate Pre-scaler (BRP) register in the CAN controller.

3. A bit edge is expected to take place during this synchronization segment when the data changes on the bus.

4. Propagation segment is used to compensate for physical delay times within the network bus lines. And is programmable from one to eight time quanta long.

5. Phase-segment1 is a buffer segment that can be lengthened during resynchronization to compensate for oscillator drift and positive phase differences between the oscillators of the transmitting and receiving nodes. And is also programmable from one to eight time quanta long.

6. Phase-segment2 can be shortened during resynchronization to compensate for negative phase

errors and oscillator drift. And is the maximum of Phase-segment1 combined with the Information Processing Time.

7. The Sample point will always be at the end of Phase-seg1. It is the time at which the bus level is read and interpreted as the value of the current bit.

8. The Information Processing Time is less than or equal to 2 time quanta.

This bit time is programmable at each node on a CAN Bus. But be aware that all nodes on a single CAN bus must have the same bit time regardless of transmitting or receiving. The bit time is a function of the period of the oscillator local to each node, the value that is user-programmed into BRP register in the controller at each node, and the programmed number of time quanta per bit.

How do they synchronize:

Suppose a node receives a data frame. Then it is necessary for the receiver to synchronize with the transmitter to have proper communication. But we don't have any explicit clock signal that a CAN system can use as a timing reference. Instead, we use two mechanisms to maintain synchronization, which is explained below.

Hard synchronization:

It occurs at the Start-of-Frame or at the transition of the start bit. The bit time is restarted from that edge.

Resynchronization:

To compensate for oscillator drift, and phase differences between transmitter and receiver oscillators, additional synchronization is needed. The resynchronization for the subsequent bits in any received frame occurs when a bit edge doesn't occur within the Synchronization Segment in a message. The resynchronization is automatically invoked and one of the Phase Segments are shortened or lengthened with an amount that depends on the phase error in the signal. The maximum amount that can be used is determined by a user-programmable number of time quanta known as the Synchronization Jump Width parameter (SJW).

Higher Layer Protocols:

Higher layer protocol (HLP) is required to manage the communication within a system. The term HLP is derived from the OSI model and its seven layers. But the CAN protocol just specifies how small packets of data may be transported from one point to another safely using a shared communications medium. It does not contain anything on the topics such as flow control, transportation of data larger than CAN fit in an 8-byte message, node addresses, establishment of communication, etc. The HLP gives solution for these topics.