CS3691 EMBEDDED SYSTEMS AND IOT

L T P C

3 0 2 4

COURSE OBJECTIVES:

- To learn the internal architecture and programming of an embedded processor.
- To introduce interfacing I/O devices to the processor.
- To introduce the evolution of the Internet of Things (IoT).
- To build a small low-cost embedded and IoT system using Arduino/RaspberryPi/ openplatform.
- To apply the concept of Internet of Things in real world scenario.

UNIT I 8-BIT EMBEDDED PROCESSOR

8-Bit Microcontroller – Architecture – Instruction Set and Programming – Programming Parallel Ports – Timers and Serial Port – Interrupt Handling.

UNIT II EMBEDDED C PROGRAMMING

Memory And I/O Devices Interfacing – Programming Embedded Systems in C – Need For RTOS – Multiple Tasks and Processes – Context Switching – Priority Based Scheduling Policies.

UNIT III IOT AND ARDUINO PROGRAMMING

Introduction to the Concept of IoT Devices – IoT Devices Versus
Computers – IoT Configurations – Basic Components – Introduction to Arduino – Types of Arduino– Arduino Toolchain – Arduino Programming Structure – Sketches – Pins – Input/Output From Pins Using Sketches – Introduction to Arduino Shields – Integration of Sensors and Actuators with Arduino.

UNIT IV IOT COMMUNICATION AND OPEN PLATFORMS

IoT Communication Models and APIs – IoT Communication Protocols – Bluetooth – WiFi – ZigBee– GPS – GSM modules – Open Platform (like Raspberry Pi) – Architecture – Programming –Interfacing – Accessing GPIO Pins – Sending and Receiving Signals Using GPIO Pins –Connecting to the Cloud.

UNIT V APPLICATIONS DEVELOPMENT

Complete Design of Embedded Systems – Development of IoT Applications – Home Automation –Smart Agriculture – Smart Cities – Smart Healthcare.

UNIT I

8-BIT EMBEDDED PROCESSOR

- 8-Bit Microcontroller
- Architecture
- Instruction Set and Programming
- Programming Parallel Ports
- Timers and Serial Port
- Interrupt Handling.

1.1. 8051 Microcontroller

8051 microcontroller is an 8-bit microcontroller created in 1981 by Intel Corporation. It has an 8-bit processor that simply means that it operates on 8-bit data at a time. It is among the most popular and commonly used microcontroller.
As it is an 8-bit microcontroller thus has 8-bit data bus, 16-bit address bus. Along with that, it holds **4 KB ROM** with **128 bytes RAM**.

1.1.1. What a Microcontroller is?

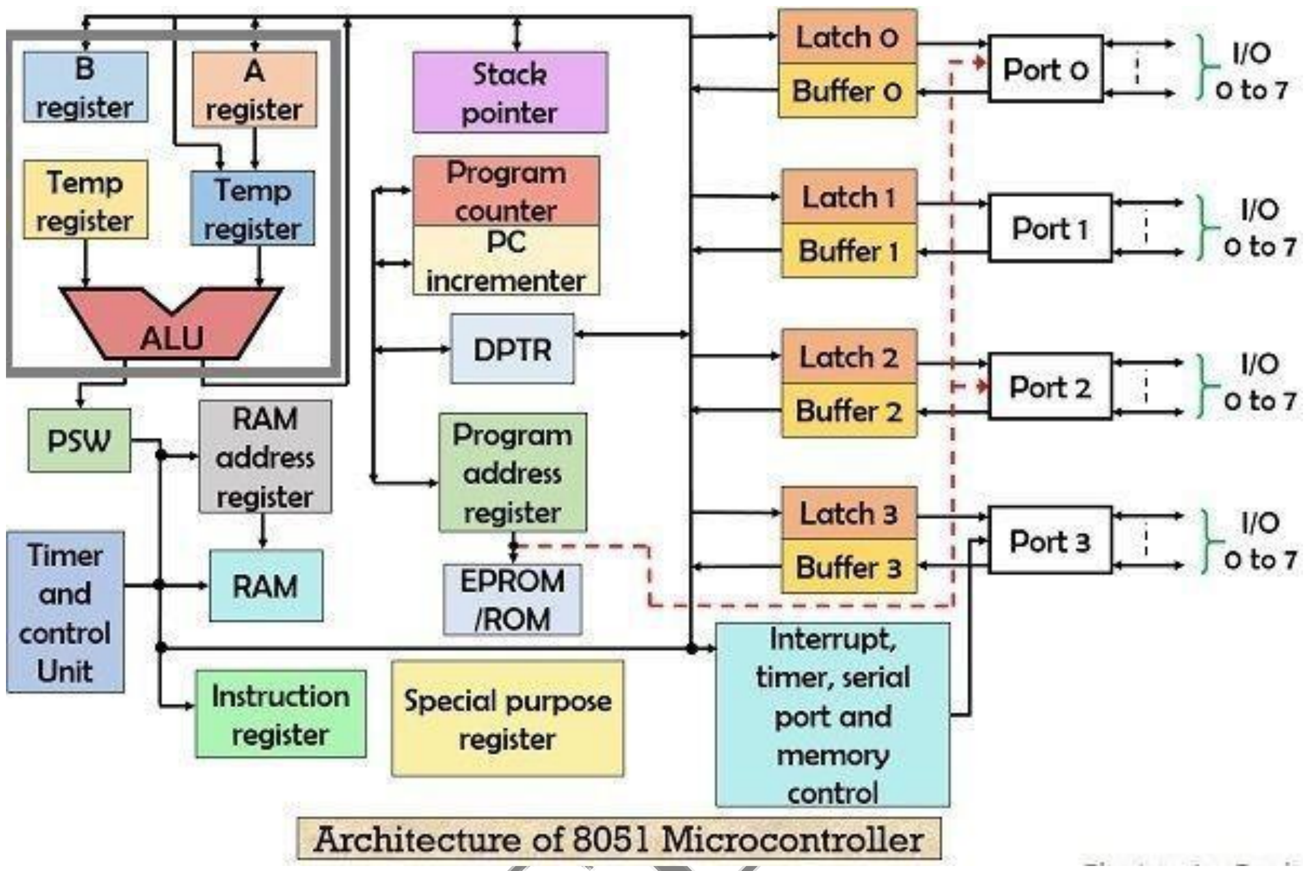
A microcontroller is an integrated chip designed under **V**ery **L**arge **S**cale **I**ntegration technique that consists of a processor with other peripheral units like memory, I/O port, timer, decoder, ADC etc. A microcontroller is basically designed in such a way that all the working peripherals are embedded in a single chip with the processor.

Any programmable device holds a processor, memory, I/O ports and timer within it. But a microcontroller contains all these components embedded in a single chip. This single-chip manages the overall operation of the device.
A microprocessor simply contains a CPU that processes the operations with the help of other peripheral units. Microprocessors are used where huge space is present to inbuilt a large motherboard like in PCs.

1.2. Architecture of 8051 Microcontroller

The figure below represents the architectural block diagram of 8051 microcontroller:
As we can see that several units are present in the above architecture. And every unit is embedded to execute the desired operation. Let us now discuss the operation of each unit present in the architecture.

1.2.1. Central processing unit (CPU): 8051 uses the 8-bit processor. This unit carries out the operation on 8-bit data. A processor is the heart of microcontroller. As the execution of the program stored in the memory is performed by the processor.
The unit performs arithmetic and logical operations on 8-bit data as it has ALU, with internal registers and program counters.
Several logical operations are performed by the ALU according to the program stored in the memory.
The processor of 8051 microcontrollers possesses a special feature by which it can process single bit or 8-bit data. This simply means that it has the ability to access each single bit data either to clear, set or move etc. for any logical computation.



Architecture of 8051 Microcontroller

1.2.2. Memory: Basically 8051 microcontroller consists of on-chip program memory i.e., ROM and on-chip data memory i.e., RAM.

Let us first understand

- **ROM**
  8051 microcontroller has 4 KB ROM with 0000H to 0FFFH as the addressable space. It is completely a program or code memory that means used by the programmer to store the programs that are to be executed by the microcontroller.
- The operations that are executed by the device in which the microcontroller is present are stored in the ROM of the memory at the time of fabrication. Hence cannot be changed or modified.

- **RAM**
  8051 holds a 128 bytes RAM. Basically, RAM is used to store data or operands for only a small time duration. It can be altered anytime according to the need of the user. It is also known as the data memory as it stores the data temporarily.
-

  Out of the 128-byte RAM, first, 32 bytes is held by the working registers. Basically, these are

4 banks which separately has 8 registers. These registers are accessed either by its name or address. It is to be noted here that at a particular time only a single register bank can be used.

- As in 8051, the data and program memory i.e., RAM and ROM hold a definite memory space. However, for some applications there exist the need for external memory to enhance the memory space, thus external RAM, ROM/EPROM is used by the 8051 microcontrollers.

1.2.3. Input/ Output port: 8051 consists of 4 parallel ports of 8 bit each thereby providing 32 input-output pins. All the 4 ports function bidirectional i.e., either input or output according to the software control.

1.2.4. Timer and Control Unit: Timers are used to create a time gap or delay between 2 events. 8051 microcontroller consists of 2 timers of 16 bit each by which the system can produce two delays simultaneously in order to generate the appropriate delay.

1.2.5. 8051 Flag Bits and PSW Register

The program status word (PSW) register is an 8-bit register, also known as flag register. It is of 8-bit wide but only 6-bit of it is used. The two unused bits are user-defined flags.

1.2.6. The Data Pointer (DPTR) is the 8051's only user-accessible 16-bit (2-byte) register. The Accumulator, R0–R7 registers and B register are 1-byte value registers. DPTR is meant for pointing to data. It is used by the 8051 to access external memory using the address indicated by DPTR.

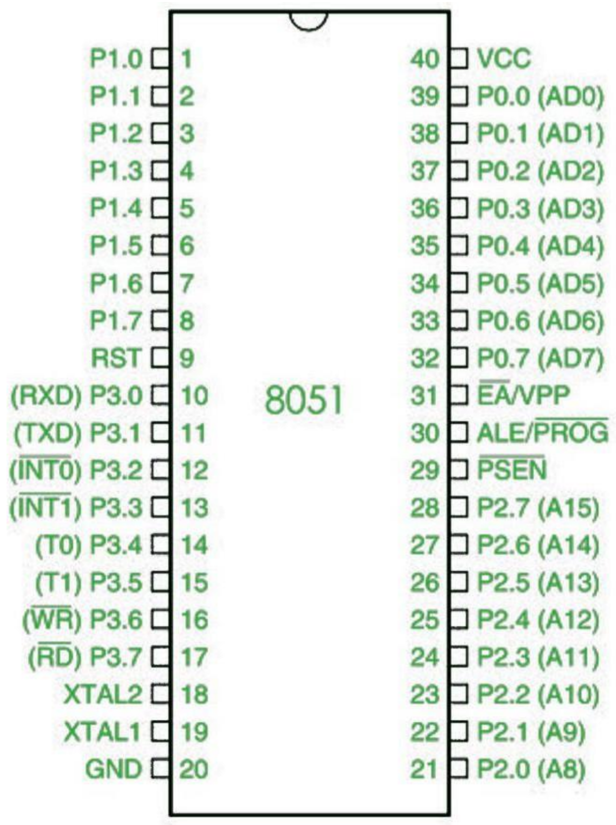
Basically, microcontrollers use **hardware delays** in which a physical device is used by the processor to produce the respective delay. And this physical device is known as a **timer**.
The timer produces the delay according to the demand of the processor and sends the signal to the processor once the respective delay gets produced.

1.2.7. Pin diagram of 8051 Microcontroller

Introduction :

The 8051 microcontroller is a popular 8-bit microcontroller widely used in embedded systems. It is a single-chip microcontroller with a Harvard architecture that includes a CPU, RAM, ROM, and several peripherals. The 8051 microcontroller has a 40-pin dual in-line package (DIP) that provides various inputs and outputs for communication with external devices.

8051 microcontroller is a 40 pin Dual Inline Package (DIP). These 40 pins serve different functions like read, write, I/O operations, interrupts etc. 8051 has four I/O ports wherein each port has 8 pins which can be configured as input or output depending upon the logic state of the pins. Therefore, 32 out of these 40 pins are dedicated to I/O ports. The rest of the pins are dedicated to VCC, GND, XTAL1, XTAL2, RST, ALE, EA' and PSEN'. Pin diagram of 8051



40 - PIN DIP

Description of the Pins :

- **Pin 1 to Pin 8 (Port 1)** – Pin 1 to Pin 8 are assigned to Port 1 for simple I/O operations. They can be configured as input or output pins depending on the logic control i.e. if logic zero (0) is applied to the I/O port it will act as an output pin and if logic one (1) is applied the pin will act as an input pin. These pins are also referred to as P1.0 to P1.7 (where P1 indicates that it is a pin in port 1 and the number after '.' tells the pin number i.e. 0 indicates first pin of the port. So, P1.0 means first pin of port 1, P1.1 means second pin of the port 1 and so on). These pins are bidirectional pins.
- **Pin 9 (RST)** – Reset pin. It is an active-high, input pin. Therefore if the RST pin is high for a minimum of 2 machine cycles, the microcontroller will reset i.e. it will close and terminate all activities. It is often referred as "power-on-reset" pin because it is used to reset the microcontroller to it's initial values when power is on (high).
- **Pin 10 to Pin 17 (Port 3)** – Pin 10 to pin 17 are port 3 pins which are also referred to as P3.0 to P3.7. These pins are similar to port 1 and can be used as universal input or output pins. These pins are bidirectional pins. These pins also have some additional functions which are as follows:

- **P3.0 (RXD) :** 10th pin is RXD (serial data receive pin) which is for serial input. Through this input signal microcontroller receives data for serial communication.
- **P3.1 (TXD) :** 11th pin is TXD (serial data transmit pin) which is serial output pin. Through this output signal microcontroller transmits data for serial communication.
- **P3.2 and P3.3 (INT0′, INT1′) :** 12th and 13th pins are for External Hardware Interrupt 0 and Interrupt 1 respectively. When this interrupt is activated(i.e. when it is low), 8051 gets interrupted in whatever it is doing and jumps to the vector value of the interrupt (0003H for INT0 and 0013H for INT1) and starts performing Interrupt Service Routine (ISR) from that vector location.
- **P3.4 and P3.5 (T0 and T1) :** 14th and 15th pin are for Timer 0 and Timer 1 external input. They can be connected with 16 bit timer/counter.
- **P3.6 (WR') :** 16th pin is for external memory write i.e. writing data to the external memory.
- **P3.7 (RD') :** 17th pin is for external memory read i.e. reading data from external memory.
- **Pin 18 and Pin 19 (XTAL2 And XTAL1) –** These pins are connected to an external oscillator which is generally a quartz crystal oscillator. They are used to provide an external clock frequency of 4MHz to 30MHz.
- **Pin 20 (GND) –** This pin is connected to the ground. It has to be provided with 0V power supply. Hence it is connected to the negative terminal of the power supply.
- **Pin 21 to Pin 28 (Port 2) –** Pin 21 to pin 28 are port 2 pins also referred to as P2.0 to P2.7. When additional external memory is interfaced with the 8051 microcontroller, pins of port 2 act as higher-order address bytes. These pins are bidirectional.
- **Pin 29 (PSEN) –** PSEN stands for Program Store Enable. It is output, active-low pin. This is used to read external memory. In 8031 based system where external ROM holds the program code, this pin is connected to the OE pin of the ROM.
- **Pin 30 (ALE/ PROG) –** ALE stands for Address Latch Enable. It is input, active-high pin. This pin is used to distinguish between memory chips when multiple memory chips are used. It is also used to de-multiplex the multiplexed address and data signals available at port 0. During flash programming i.e. Programming of EPROM, this pin acts as program pulse input (PROG).
- **Pin 31 (EA/ VPP) –** EA stands for External Access input. It is used to enable/disable external memory interfacing. In 8051, EA is connected to Vcc as it comes with on-chip ROM to store programs. For other family members such as 8031 and 8032 in which there is no on-chip ROM, the EA pin is connected to the GND.
- **Pin 32 to Pin 39 (Port 0) –** Pin 32 to pin 39 are port 0 pins also referred to as P0.0 to P0.7. They are bidirectional input/output pins. They don't have any internal pull-ups. Hence, 10 K? pull-up registers are used as external pull-ups. Port 0 is also designated as AD0-AD7 because 8051 multiplexes address and data through port 0 to save pins.
- **Pin 40 (VCC) –** This pin provides power supply voltage i.e. +5 Volts to the circuit.

Uses of pin diagram of the 8051 microcontroller :

The pin diagram of the 8051 microcontroller is used for various purposes in embedded systems. Some of the main uses of the pin diagram are:

1. **Interfacing with external devices:** The 8051 microcontroller has several input/output pins that can be used for interfacing with external devices such as sensors, actuators, displays, and communication modules. The pin diagram provides the information about the location of these pins, their functionalities, and their electrical characteristics.
2. **Programming the microcontroller:** The 8051 microcontroller can be programmed using various programming languages such as Assembly, C, and BASIC. The pin diagram provides the information about the pins that are used for programming the microcontroller, such as the PSEN pin and the ALE pin.
3. **Debugging and testing:** The pin diagram provides access to the internal signals of the microcontroller, such as the address and data buses, which can be used for debugging and testing the microcontroller. Special hardware tools such as logic analyzers and oscilloscopes can be connected to the pins to monitor the signals and diagnose any issues in the system.
4. **Expansion and customization:** The pin diagram provides the flexibility to expand and customize the functionality of the microcontroller by connecting external devices and peripherals. For example, additional memory can be added by connecting external RAM or ROM chips to the address and data buses.

1.2.8. Characteristics of 8051 Microcontroller

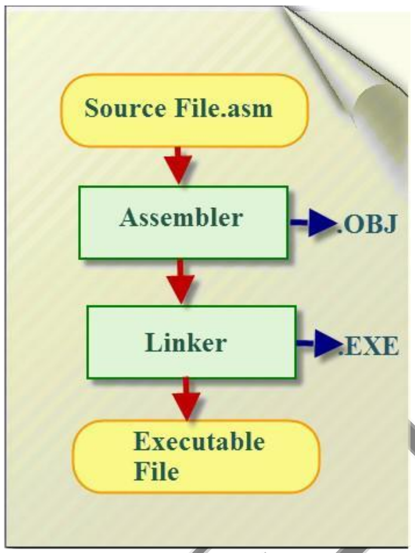
1. An 8-bit processor.
2. Data memory or RAM of 128 bytes.
3. Program memory or ROM of 4 KB.
4. 2 timers of 16 bit each.
5. 8-bit data bus.
6. 16-bit address bus.
7. Offers bit addressable format.
8. Special function registers and serial port.
9. 32 input/output lines.

1.3. 8051 Programming in Assembly Language

The assembly language is a low-level programming language used to write program code in terms of mnemonics. Even though there are many high-level languages that are currently in demand, assembly programming language is popularly used in many applications. It can be used for direct hardware manipulations. It is also used to write the 8051 programming code efficiently with less number of clock cycles by consuming less memory compared to the other high-level languages.

1.3.1. 8051 Programming in Assembly Language

The assembly language is a fully hardware related programming language. The embedded designers must have sufficient knowledge on hardware of particular processor or controllers before writing the program. The assembly language is developed by mnemonics; therefore, users cannot understand it easily to modify the program.



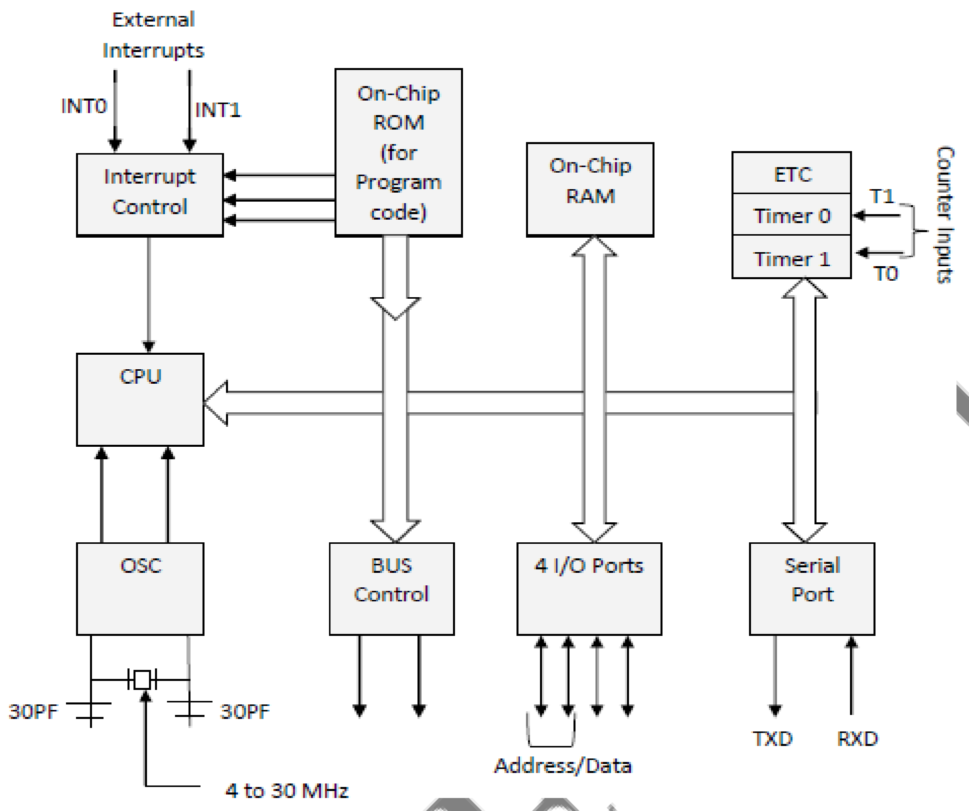
8051 Programming in Assembly Language

Assembly programming language is developed by various compilers and the “keiluvison” is best suitable for microcontroller programming development.

Microcontrollers or processors can understand only binary language in the form of ‘0s or 1s’; An assembler converts the assembly language to binary language, and then stores it in the microcontroller memory to perform the specific task.

1.3.2. 8051 Microcontroller Architecture

The 8051 microcontroller is the CISC based Harvard architecture, and it has peripherals like 32 I/O, timers/counters, serial communication and memories. The microcontroller requires a program to perform the operations that require a memory for saving and to read the functions. The 8051 microcontroller consists of RAM and ROM memories to store instructions.



8051 Microcontroller Architecuture

A Register is the main part in the processors and microcontrollers which is contained in the memory that provides a faster way of collecting and storing the data. The 8051 assembly language programming is based on the memory registers. If we want to manipulate data to a processor or controller by performing subtraction, addition, etc., we cannot do that directly in the memory, but it needs registers to process and to store the data. Microcontrollers contain several types of registers that can be classified according to their instructions or content that operate in them.

1.3.3. 8051 Microcontroller Programs in Assembly Language

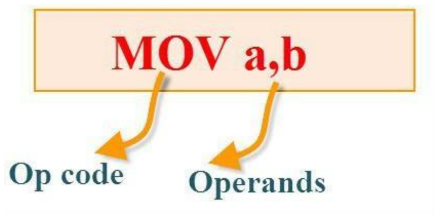
The assembly language is made up of elements which all are used to write the program in sequential manner. Follow the given rules to write programming in assembly language.

1.3.3.1. Rules of Assembly Language

- The assembly code must be written in upper case letters
- The labels must be followed by a colon (label:)
- All symbols and labels must begin with a letter

- All comments are typed in lower case
- The last line of the program must be the END directive

The assembly language mnemonics are in the form of op-code, such as MOV, ADD, JMP, and so on, which are used to perform the operations.



Op-code: The op-code is a single instruction that can be executed by the CPU. Here the op-code is a MOV instruction.

Operands: The operands are a single piece of data that can be operated by the op-code. Example, multiplication operation is performed by the operands that are multiplied by the operand.

Syntax: MUL a,b;

1.3.4. The Elements of an Assembly Language Programming:

- Assembler Directives
- Instruction Set
- Addressing Modes

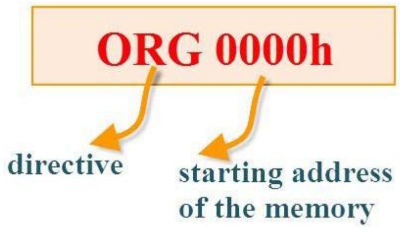
1.3.4.1. Assembler Directives:

The assembling directives give the directions to the CPU. The 8051 microcontroller consists of various kinds of assembly directives to give the direction to the control unit. The most useful directives are 8051 programming, such as:

- ORG
- DB
- EQU
- END

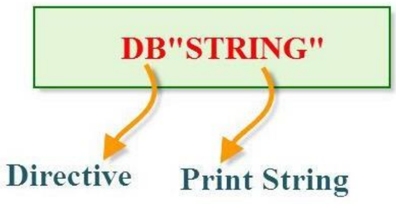
ORG(origin): This directive indicates the start of the program. This is used to set the register address during assembly. For example; ORG 0000h tells the compiler all subsequent code starting at address 0000h.

Syntax: ORG 0000h



(define byte): The define byte is used to allow a string of bytes. For example, print the "EDGEFX" wherein each character is taken by the address and finally prints the "string" by the DB directly with double quotes.

Syntax:
ORG 0000h
MOV a, #00h
————-
————-
DB"EDGEFX"



EQU (equivalent): The equivalent directive is used to equate address of the variable.

Syntax:
reg equ,09h
—————
—————
MOV reg,#2h

END: The END directive is used to indicate the end of the program.
Syntax:
reg equ,09h
—————
—————
MOV reg,#2h
END

1.3.4.2. Embedded Systems - Addressing Modes

An **addressing mode** refers to how you are addressing a given memory location. There are five different ways or five addressing modes to execute this instruction which are as follows −

- Immediate addressing mode

- Direct addressing mode
- Register direct addressing mode
- Register indirect addressing mode
- Indexed addressing mode

Immediate Addressing Mode

Let's begin with an example.

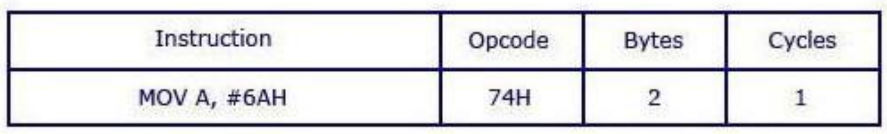
MOV A, #6AH

In general, we can write,

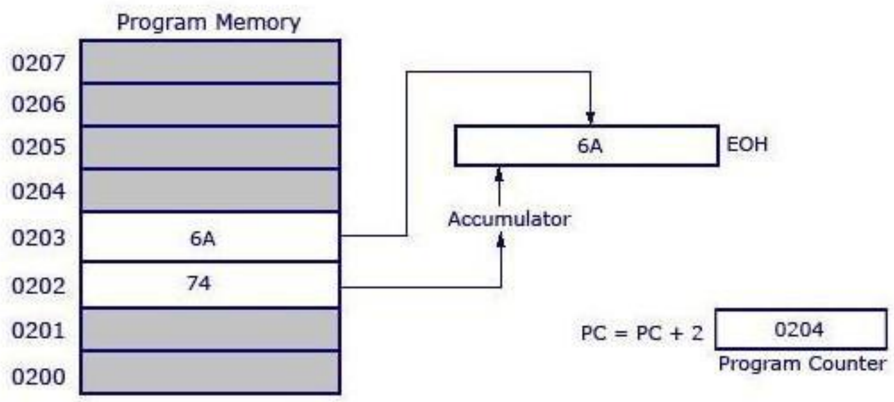
MOV A, #data

It is termed as **immediate** because 8-bit data is transferred immediately to the accumulator (destination operand).

Immediate Addressing Mode



Instruction	Opcode	Bytes	Cycles
MOV A, #6AH	74H	2	1



The above instruction and its execution. The opcode 74H is saved at 0202 address. The data 6AH is saved at 0203 address in the program memory. After reading the opcode 74H, the data at the next program memory address is transferred to accumulator A (E0H is the address of accumulator). Since the instruction is of 2-bytes and is executed in one cycle, the program counter will be incremented by 2 and will point to 0204 of the program memory.

Note − The '#' symbol before 6AH indicates that the operand is a data (8 bit). In the absence of '#', the hexadecimal number would be taken as an address.

Direct Addressing Mode

This is another way of addressing an operand. Here, the address of the data (source data) is given as an operand. Let’s take an example.

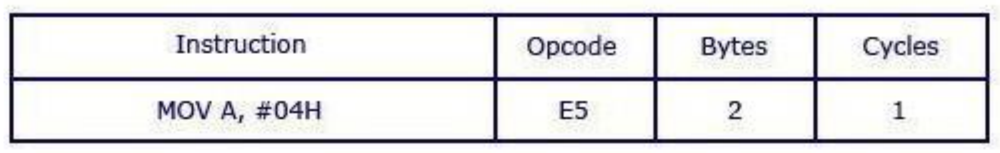
MOV A, 04H

The register bank#0 (4th register) has the address 04H. When the MOV instruction is executed, the data stored in register 04H is moved to the accumulator. As the register 04H holds the data 1FH, 1FH is moved to the accumulator.

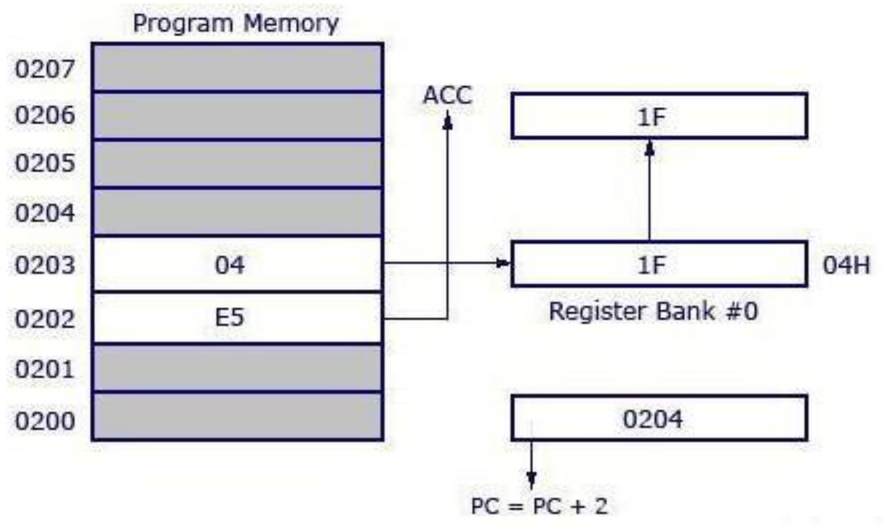
Note − We have not used '#' in direct addressing mode, unlike immediate mode. If we had used '#', the data value 04H would have been transferred to the accumulator instead of 1FH.

Now, take a look at the following illustration. It shows how the instruction gets executed.

Direct Addressing Mode



Instruction	Opcode	Bytes	Cycles
MOV A, #04H	E5	2	1



As shown in the above illustration, this is a 2-byte instruction which requires 1 cycle to complete. The PC will be incremented by 2 and will point to 0204. The opcode for the instruction MOV A, address is E5H. When the instruction at 0202 is executed (E5H), the accumulator is made active and ready to receive data. Then the PC goes to the next address as 0203 and looks up the address of the location of 04H where the source data (to be transferred to accumulator) is located. At 04H, the control finds the data 1F and transfers it to the accumulator and hence the execution is completed.

Register Direct Addressing Mode

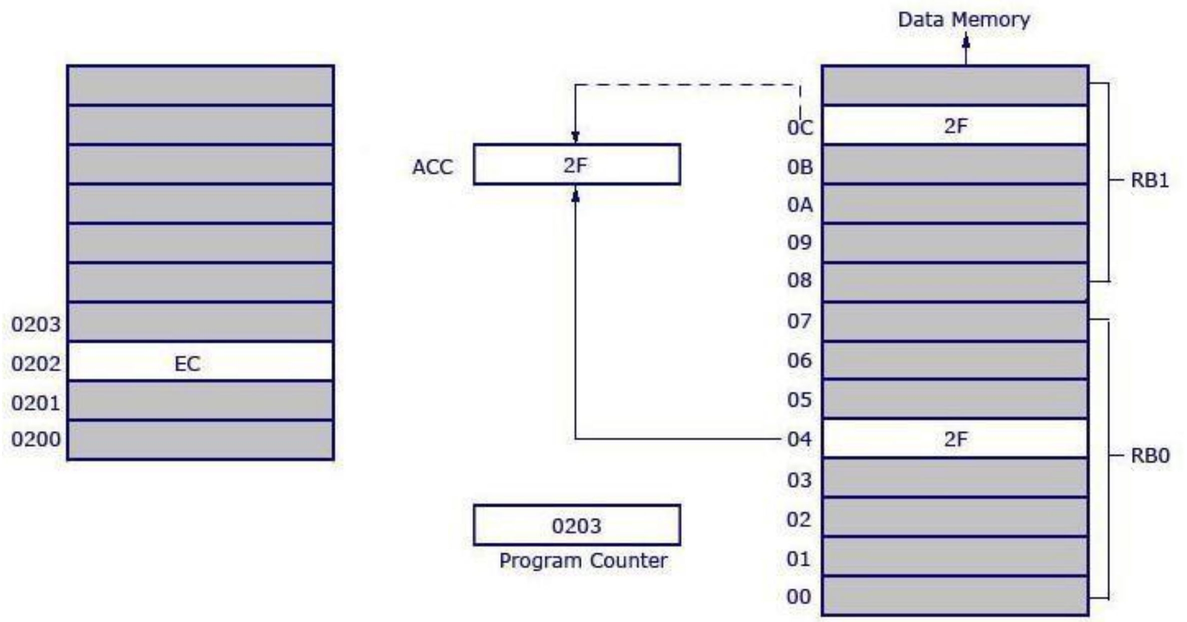
In this addressing mode, we use the register name directly (as source operand). Let us try to understand with the help of an example.

MOV A, R4

At a time, the registers can take values from R0 to R7. There are 32 such registers. In order to use 32 registers with just 8 variables to address registers, register banks are used. There are 4 register banks named from 0 to 3. Each bank comprises of 8 registers named from R0 to R7.

Register Direct Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOV A, R4	ECH	1	1



At a time, a single register bank can be selected. Selection of a register bank is made possible through a **Special Function Register** (SFR) named **Processor Status Word** (PSW). PSW is an 8-bit SFR where each bit can be programmed as required. Bits are designated from PSW.0 to PSW.7. PSW.3 and PSW.4 are used to select register banks.

Now, take a look at the following illustration to get a clear understanding of how it works.

Opcode EC is used for MOV A, R4. The opcode is stored at the address 0202 and when it is executed, the control goes directly to R4 of the respected register bank (that is selected in PSW). If register bank #0 is selected, then the data from R4 of register bank #0 will be moved to the accumulator. Here 2F is stored at 04H. 04H represents the address of R4 of register bank #0.

Data (2F) movement is highlighted in bold. 2F is getting transferred to the accumulator from data memory location 0C H and is shown as dotted line. 0CH is the address location of Register 4 (R4) of register bank #1. The instruction above is 1 byte and requires 1 cycle for complete execution. What it means is, you can save program memory by using register direct addressing mode.

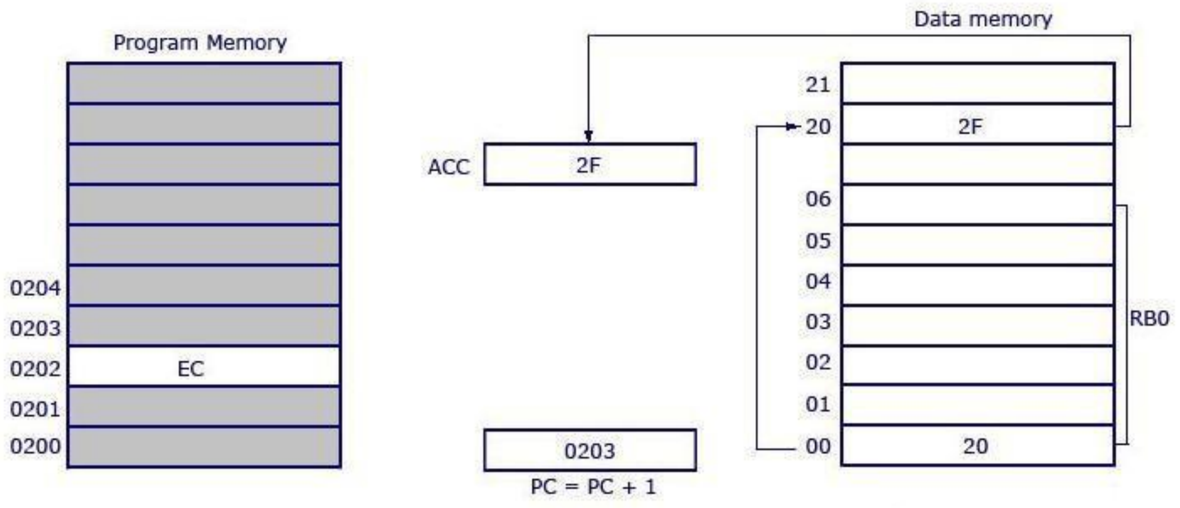
Register Indirect Addressing Mode

In this addressing mode, the address of the data is stored in the register as operand.

MOV A, @R0

Register Indirect Addressing Mode

Instruction	Opcode	Bytes	Cycles
MOV A, @ R0	E6H	1	1

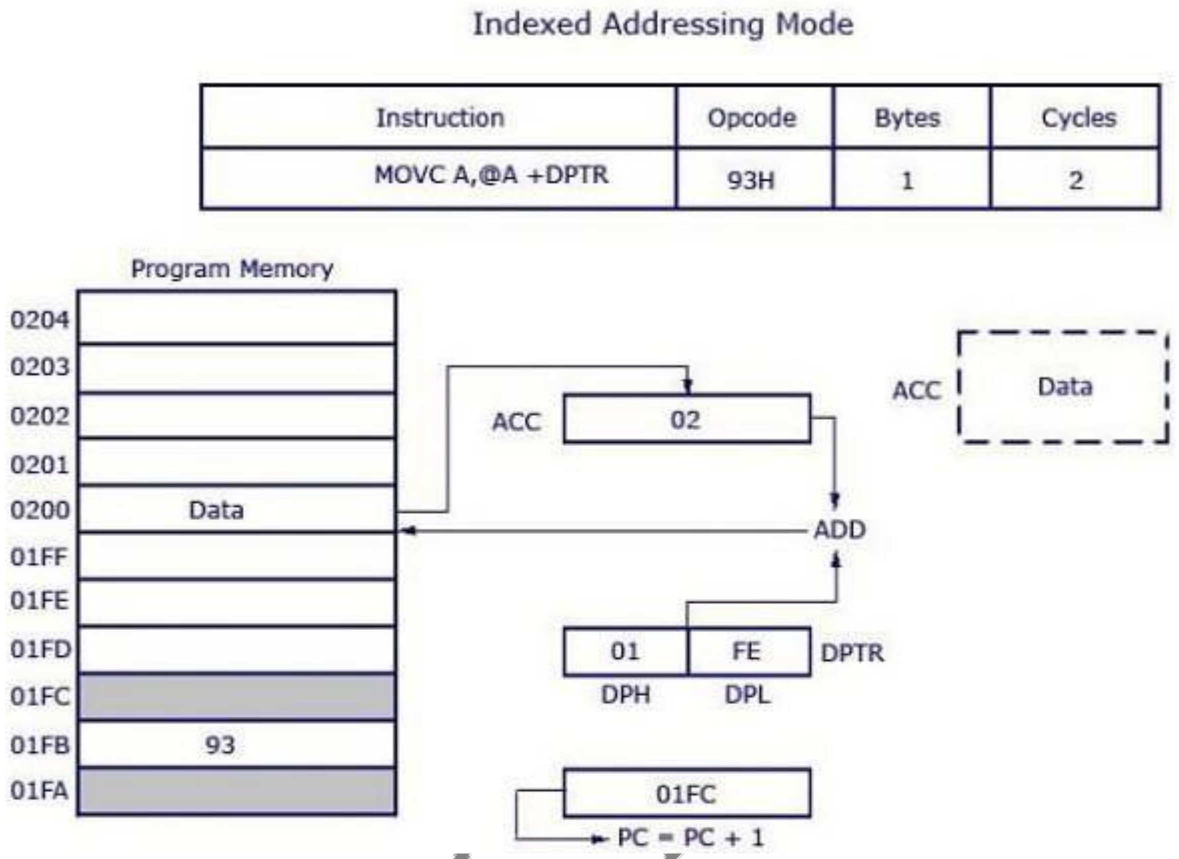


Here the value inside R0 is considered as an address, which holds the data to be transferred to the accumulator. **Example**: If R0 has the value 20H, and data 2FH is stored at the address 20H, then the value 2FH will get transferred to the accumulator after executing this instruction. See the following illustration.

So the opcode for **MOV A, @R0** is E6H. Assuming that the register bank #0 is selected, the R0 of register bank #0 holds the data 20H. Program control moves to 20H where it locates the data 2FH and it transfers 2FH to the accumulator. This is a 1-byte instruction and the program counter increments by 1 and moves to 0203 of the program memory.

Note − Only R0 and R1 are allowed to form a register indirect addressing instruction. In other words, the programmer can create an instruction either using @R0 or @R1. All register banks are allowed.

Indexed Addressing Mode



We will take two examples to understand the concept of indexed addressing mode. Take a look at the following instructions –

MOVC A, @A+DPTR

and

MOVC A, @A+PC

where DPTR is the data pointer and PC is the program counter (both are 16-bit registers). Consider the first example.

MOVC A, @A+DPTR

The source operand is @A+DPTR. It contains the source data from this location. Here we are adding the contents of DPTR with the current content of the accumulator. This addition will give a new address which is the address of the source data. The data pointed by this address is then transferred to the accumulator.

The opcode is 93H. DPTR has the value 01FE, where 01 is located in DPH (higher 8 bits) and FE is located in DPL (lower 8 bits). Accumulator has the value 02H. Then a 16-bit addition is performed and 01FE H+02H results in 0200 H. Data at the location 0200H will get transferred to the accumulator. The previous value inside the accumulator (02H) will be replaced with the new data from 0200H. The new data in the accumulator is highlighted in the illustration.

This is a 1-byte instruction with 2 cycles needed for execution and the execution time required for this instruction is high compared to previous instructions (which were all 1 cycle each).

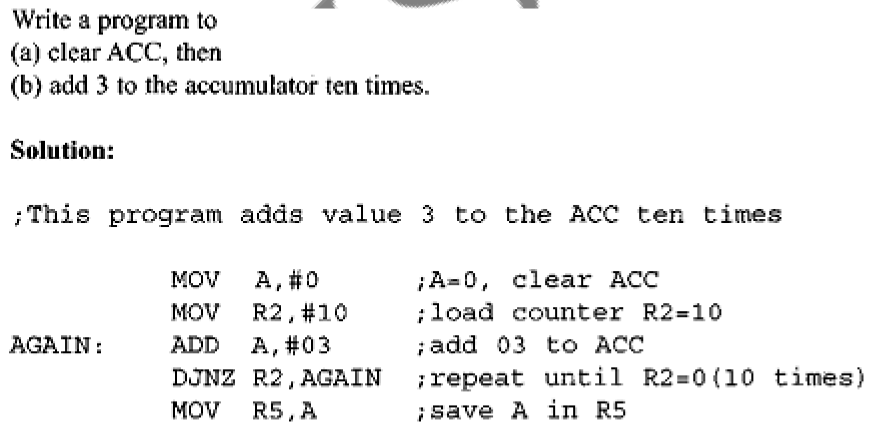
The other example **MOVC A, @A+PC** works the same way as the above example. Instead of adding DPTR with the accumulator, here the data inside the program counter (PC) is added with the accumulator to obtain the target address.

1.3.4.3. Instruction Set

In the sequence of instructions to be executed, it is often necessary to transfer program control to a different location. There are many instructions in the 8051 to achieve this. This chapter covers the control transfer instructions available in 8051 Assembly language. In the first section we discuss instructions used for looping, as well as instructions for conditional and unconditional jumps. In the second section we examine CALL instructions and their uses. In the third section, time delay subroutines are described for both the traditional 8051 and its newer generation.

Looping in the 8051

Repeating a sequence of instructions a certain number of times is called a *loop*. The loop is one of most widely used actions that any microprocessor performs. In the 8051, the loop action is performed by the instruction "DJNZ reg, label". In this instruction, the register is decremented; if it is not zero, it jumps to the target address referred to by the label. Prior to the start of the loop the register is loaded with the counter for the number of repetitions. Notice that in this instruction both the register decrement and the decision to jump are combined into a single instruction.



Write a program to
(a) clear ACC, then
(b) add 3 to the accumulator ten times.

Solution:

```
;This program adds value 3 to the ACC ten times

          MOV  A,#0       ;A=0, clear ACC
          MOV  R2,#10     ;load counter R2=10
AGAIN:    ADD  A,#03      ;add 03 to ACC
          DJNZ R2,AGAIN   ;repeat until R2=0(10 times)
          MOV  R5,A       ;save A in R5
```

In the program in Example, the R2 register is used as a counter. The counter is first set to 10. In each iteration the instruction DJNZ decrements R2 and checks its value. If R2 is not zero, it jumps to the target address associated with the label "AGAIN". This looping action continues until R2 becomes zero. After R2 becomes zero, it falls through the loop and executes the instruction immediately below it, in this case the “MOV R5 , A” instruction. Notice in the DJNZ instruction that the registers can be any of RO – R7. The counter can also be a RAM location

Loop inside a loop

As shown in Example the maximum count is 256. What happens if we want to repeat an action more times than 256? To do that, we use a loop inside a loop, which is called a *nested loop.* In a nested loop, we use two registers to hold the count.

Example

Write a program to (a) load the accumulator with the value 55H, and (b) complement the ACC 700 times.

Solution:

Since 700 is larger than 255 (the maximum capacity of any register), we use two registers to hold the count. The following code shows how to use R2 and R3 for the count.

```
          MOV  A,#55H
          MOV  R3,#10
NEXT:     MOV  R2,#70
AGAIN:    CPL  A
          DJNZ R2,AGAIN
          DJNZ R3,NEXT


;A=55H
;R3=10, the outer loop count
;R2=70, the inner loop count
;complement A register
;repeat it 70 times (inner loop)
```

In this program, R2 is used to keep the inner loop count. In the instruction “DJNZ R2 , AGAIN”, whenever R2 becomes 0 it falls through and “DJNZ R3 , NEXT” is executed. This instruction forces the CPU to load R2 with the count 70 and the inner loop starts again. This process will continue until R3 becomes zero and the outer loop is finished.

Other conditional jumps

Conditional jumps for the 8051 are summarized in Table 3-1. More details of each instruction are provided in Appendix A. In Table 3-1, notice that some of the instructions, such as JZ (jump if A = zero) and JC (jump if carry), jump only if a certain condition is met. Next we examine some conditional jump instructions with examples.

JZ (jump if A = 0)

In this instruction the content of register A is checked. If it is zero, it jumps to the target address. For example, look at the following code.

```
        ;A=R0
        ;jump if A = 0
        ;A=R1
        ;jump if A = 0
```

```
MOV   A,R0
JZ    OVER
MOV   A,R1
JZ    OVER
```

Instruction	Action
JZ	Jump if A = 0
JNZ	Jump if A ≠ 0
DJNZ	Decrement and jump if register ≠ 0
CJNE A, data	Jump if A ≠ data
CJNE reg, #data	Jump if byte ≠ #data
JC	Jump if CY = 1
JNC	Jump if CY = 0
JB	Jump if bit = 1
JNB	Jump if bit = 0
JBC	Jump if bit = 1 and clear bit

In this program,. if either RO or Rl is zero, it jumps to the label OVER. Notice that the JZ instruction can be used only for register A. It can only check to see whether the accumulator is zero, and it does not apply to any other register. More importantly, you don’t have to perform an arithmetic instruction such as decrement to use the JNZ instruction. See Example 3-4.

Example

Write a program to determine if R5 contains the value 0. If so, put 55H in it.

Solution:

```
         MOV   A,R5        ;copy R5 to A
         JNZ   NEXT        ;jump if A is not zero
         MOV   R5,#55H
NEXT:    ...
```

JNC (jump if no carry, jumps if CY = 0)

In this instruction, the carry flag bit in the flag (PSW) register is used to make the decision whether to jump. In executing “JNC label”, the processor looks at the carry flag to see if it is

raised (CY =1). If it is not, the CPU starts to fetch and execute instructions from the address of the label. If CY = 1, it will not jump but will execute the next instruction below JNC.
Note that there is also a “JC label” instruction. In the JC instruction, if CY = 1 it jumps to the target address. We will give more examples of these instructions in the context of applications in future chapters.
There are also JB (jump if bit is high) and JNB (jump if bit is low) instructions. These are discussed in Chapters 4 and 8 when bit manipulation instructions are discussed.

All conditional jumps are short jumps
It must be noted that all conditional jumps are short jumps, meaning that the address of the target must be within -128 to +127 bytes of the contents of the program counter (PC). This very important concept is discussed at the end of this section.

Unconditional jump instructions
The unconditional jump is a jump in which control is transferred unconditionally to the target location. In the 8051 there are two unconditional jumps: LJMP (long jump) and SJMP (short jump). Each is discussed below.

LJMP (long jump)
LJMP is an unconditional long jump. It is a 3-byte instruction in which the first byte is the opcode, and the second and third bytes represent the 16-bit address of the target location. The 2-byte target address allows a jump to any memory location from 0000 to FFFFH.
Remember that although the program counter in the 8051 is 16-bit, thereby giving a ROM address space of 64K bytes, not all 8051 family members have that much on-chip program ROM. The original 8051 had only 4K bytes of on-chip ROM for program space; consequently, every byte was precious. For this reason there is also an SJMP (short jump) instruction, which is a 2-byte instruction as opposed to the 3-byte LJMP instruction. This can save some bytes of memory in many applications where memory space is in short supply. SJMP is discussed next.

SJMP (short jump)
In this 2-byte instruction, the first byte is the opcode and the second byte is the relative address of the target location. The relative address range of 00 – FFH is divided into forward and backward jumps; that is, within -128 to +127 bytes of memory relative to the address of the current PC (program counter). If the jump is forward, the target address can be within a space of 127 bytes from the current PC. If the target address is backward, the target address can be within -128 bytes from the current PC. This is explained in detail next.

Calculating the short jump address
In addition to the SJMP instruction, all conditional jumps such as JNC, JZ, and DJNZ are also short jumps due to the fact that they are all two-byte instructions. In these instructions the first byte is the opcode and the second byte is the relative address. The target address is relative to the

value of the program counter. To calculate the target address, the second byte is added to the PC of the instruction immediately below the jump.

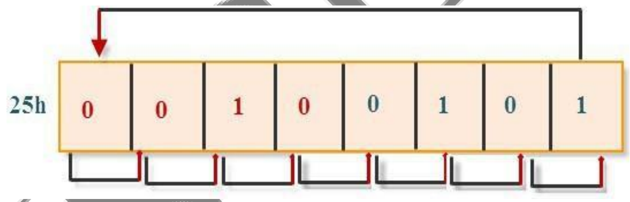
Shifting Operators

The shift operators are used for sending and receiving the data efficiently. The 8051 microcontroller consist four shift operators:

- RR —> Rotate Right
- RRC —>Rotate Right through carry
- RL —> Rotate Left
- RLC —>Rotate Left through carry

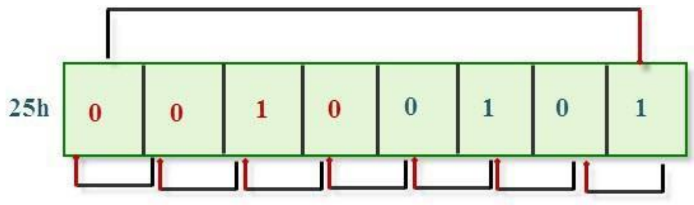
Rotate Right (RR):
In this shifting operation, the MSB becomes LSB and all bits shift towards right side bit-by-bit, serially.

Syntax:
MOV A, #25h
RR A



Rotate Left (RL):
In this shifting operation, the MSB becomes LSB and all bits shift towards Left side bit-by-bit, serially.

Syntax:
MOV A, #25h
RL A



RRC Rotate Right through Carry:
In this shifting operation, the LSB moves to carry and the carry becomes MSB, and all the bits are shift towards right side bit by bit position.

Syntax:
```
MOV A, #27h
RRC A
```

RLC Rotate Left through Carry:
In this shifting operation, the MSB moves to carry and the carry becomes LSB and all the bits shift towards left side in a bit-by-bit position.

Syntax:
```
MOV A, #27h
RLC A
```

Basic Embedded C Programs:

The microcontroller programming differs for each type of operating system. There are many operating systems such as Linux, Windows, RTOS and so on. However, RTOS has several advantages for embedded system development. Some of the Assembly levels programming examples are given below.

1.3.5. Example Program for LED blinking using with 8051 microcontroller:
- Number Displaying on 7-segment display using 8051 microcontroller
- Timer/Counter calculations and program using 8051 microcontroller
- Serial Communication calculations and program using 8051 microcontroller

LED programs with 8051 Microcontrller

1. WAP to toggle the PORT1 LEDs
```
ORG 0000H
TOGLE: MOV P1, #01    //move 00000001 to the p1 register//
CALL DELAY    //execute the delay//
MOV A, P1        //move p1 value to the accumulator//
CPL A        //complement A value //
MOV P1, A        //move 11111110 to the port1 register//
CALL DELAY    //execute the delay//
SJMP TOGLE
DELAY: MOV R5, #10H    //load register R5 with 10//
```

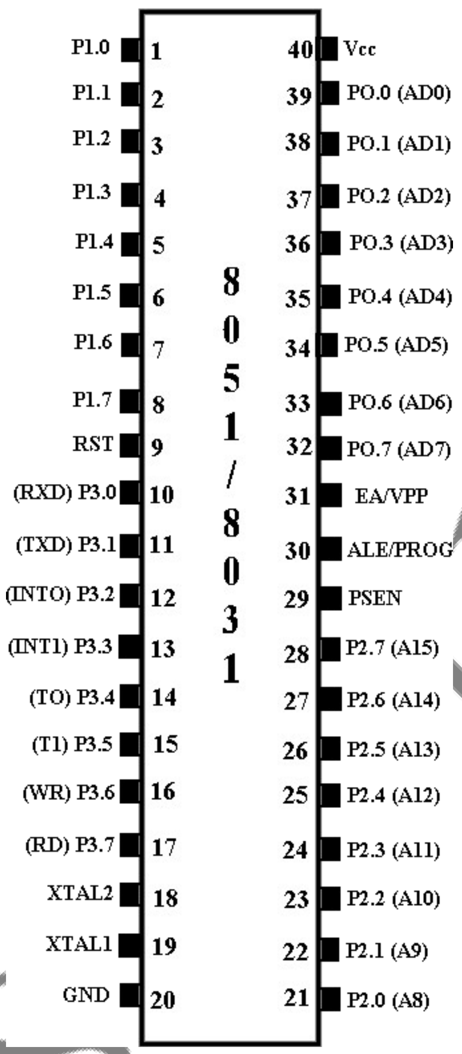
```
TWO:        MOV R6, #200    //load register R6 with 200//
ONE:        MOV R7, #200    //load register R7 with 200//
DJNZ R7, $    //decrement R7 till it is zero//
DJNZ R6, ONE    //decrement R7 till it is zero//
DJNZ R5, TWO    //decrement R7 till it is zero//
RET            //go back to the main program //
END
```

1.4. Programming Parallel port

1.4.1. 8051 Microcontroller port programming

There are four ports P0, P1, P2 and P3 each use 8 pins, making them 8-bit ports. All the ports upon RESET are configured as output, ready to be used as output ports. To use any of these ports as an input port, it must be programmed.

1.4.2. Pin configuration of 8051/8031 microcontroller.

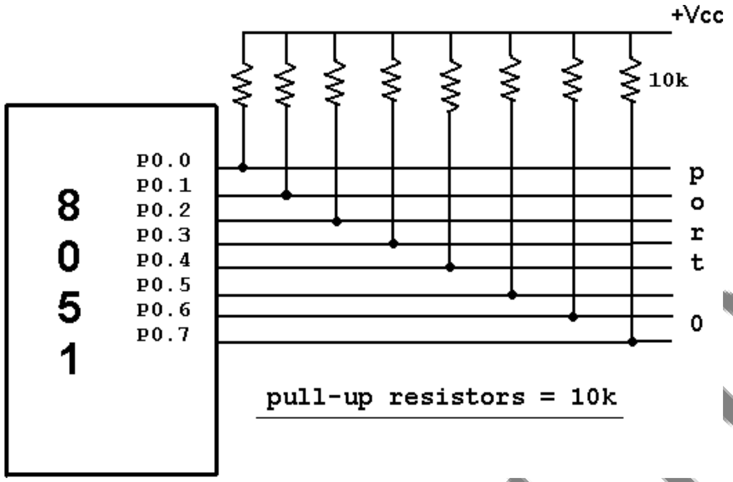


Pin configuration of 8951

Port 0: Port 0 occupies a total of 8 pins (pins 32-39) .It can be used for input or output. To use the pins of port 0 as both input and output ports, each pin must be connected externally to a 10K ohm pull-up resistor. This is due to the fact that P0 is an open drain, unlike P1, P2, and P3.Open drain is a term used for MOS chips in the same way that open collector is used for TTL chips. With external pull-up resistors connected upon reset, port 0 is configured as an output port. For example, the following code will continuously send out to port 0 the alternating values 55H and AAH

```
MOV A,#55H
BACK: MOV P0,A
ACALL DELAY
CPL A
SJMP BACK
```

Port 0 as Input : With resistors connected to port 0, in order to make it an input, the port must be programmed by writing 1 to all the bits. In the following code, port 0 is configured first as an input port by writing 1's to it, and then data is received from the port and sent to P1.



8051 I/O Ports

```
MOV A,#0FFH         ; A = FF hex
MOV P0,A            ; make P0 an input port
BACK: MOV A,P0      ;get data from P0
MOV P1,A            ;send it to port 1
SJMP BACK
```

Dual role of port 0: Port 0 is also designated as AD0-AD7, allowing it to be used for both address and data. When connecting an 8051/31 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save pins. ALE indicates if P0 has address or data. When ALE = 0, it provides data D0-D7, but when ALE =1 it has address and data with the help of a 74LS373 latch.

Port 1: Port 1 occupies a total of 8 pins (pins 1 through 8). It can be used as input or output. In contrast to port 0, this port does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset, Port 1 is configured as an output port. For example, the following code will continuously send out to port1 the alternating values 55h & AAh

```
MOV A,#55H              ; A = 55 hex
BACK: MOV P1,A          ;send it to Port 1
ACALL DELAY             ;call delay routine
CPL A                   ;make A=0
SJMP BACK
```

Port 1 as input: To make port1 an input port, it must programmed as such by writing 1 to all its bits. In the following code port1 is configured first as an input port by writing 1's to it, then data is received from the port and saved in R7 ,R6 & R5.

```
MOV A,#0FFH   ;A=FF HEX
MOV P1,A        ;make P1 an input port by writing all 1's to it
MOV A,P1        ;get data from P1
MOV R7,A        ;save it in register R7
ACALL DELAY   ;wait
MOV  A,P1       ;get another data from P1
MOV R6,A        ;save it in register R6
ACALL DELAY   ;wait
MOV  A,P1       ;get another data from P1
MOV R5,A        ;save it in register R5
```

Port 2 : Port 2 occupies a total of 8 pins (pins 21- 28). It can be used as input or output. Just like P1, P2 does not need any pull-up resistors since it already has pull-up resistors internally. Upon reset,Port 2 is configured as an output port. For example, the following code will send out continuously to port 2 the alternating values 55h and AAH. That is all the bits of port 2 toggle continuously.

```
MOV A,#55H                 ; A = 55 hex
BACK:  MOV P2,A              ;send it to Port 2
ACALL DELAY                  ;call delay routine
CPL A                       ;make A=0
SJMP BACK
```

Port 2 as input : To make port 2 an input, it must programme as such by writing 1 to all its bits. In the following code, port 2 is configured first as an input port by writing 1's to it. Then data is received from that port and is sent to P1 continuously.

```
MOV A,#0FFH        ;A=FF hex
MOV P2,A           ;make P2 an input port by writing all 1's to it
BACK: MOV A,P2       ;get data from P2
MOV P1,A           ;send it to Port1
SJMP BACK          ;keep doing that
```

Dual role of port 2 : In systems based on the 8751, 8951, and DS5000, P2 is used as simple I/O. However, in 8031-based systems, port 2 must be used along with P0 to provide the 16-bit address for the external memory. As shown in pin configuration 8051, port 2 is also designed as A8-A15, indicating the dual function. Since an 8031 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0-A7, it is the job of P2 to provide bits A8-A15 of the address. In other words, when 8031 is connected to external memory, P2 is used for the upper 8 bits of the 16 bit address, and it cannot be used for I/O.

Port 3 : Port 3 occupies a total of 8 pins, pins 10 through 17. It can be used as input or output. P3 does not need any pull-up resistors, the same as P1 and P2 did not. Although port 3 is configured as an output port upon reset. Port 3 has the additional function of providing some extremely important signals such as interrupts. This information applies both 8051 and 8031 chips.

Read-modify-write feature : The ports in the 8051 can be accessed by the read-modify-write technique. This feature saves many lines of code by combining in a single instruction all three action of (1) reading the port, (2) modifying it, and (3) writing to the port. The following code first places 01010101 (binary) into port 1. Next, the instruction "XLR P1,#0FFH" performs an XOR logic operation on P1 with 1111 1111 (binary), and then writes the result back into P1.

```
MOV P1,#55H      ;P1=01010101
AGAIN: XLR P1,#0FFH     ;EX-OR P1 with 1111 1111
ACALL DELAY
SJMP AGAIN
```

1.4.3. Addition of two 8-bit numbers in 8051 Microcontroller Using Ports Introduction:

To perform addition of two 8-bit numbers using ports in 8051 microcontroller, we need to connect the two 8-bit numbers to be added to two ports of the microcontroller. We can use any two ports of the microcontroller, for example, P1 and P2.

The first step is to load the two numbers into two different ports. For example, we can load the first number into port P1 and the second number into port P2. We can use the MOV instruction to load the numbers into the ports.

Once the numbers are loaded into the ports, we can use the ADD instruction to add the two numbers. The ADD instruction adds the contents of the accumulator and the specified operand and stores the result in the accumulator. Since the two numbers are already loaded into the ports, we can simply use the ADD instruction with the accumulator and the appropriate port.

After the addition is complete, we can retrieve the result from the accumulator and store it in another port or memory location for further processing or display. 8051 microcontroller is a microcontroller designed by Intel in 1981. It is an 8-bit microcontroller with 40 pins DIP (dual inline package), 4kb of ROM storage and 128 bytes of RAM storage, 16-bit timers. It consists of four parallel 8-bit ports, which are programmable as well as addressable as per the requirement.

Issues in Addition of two 8-bit numbers 8051 Microcontroller Using Ports :

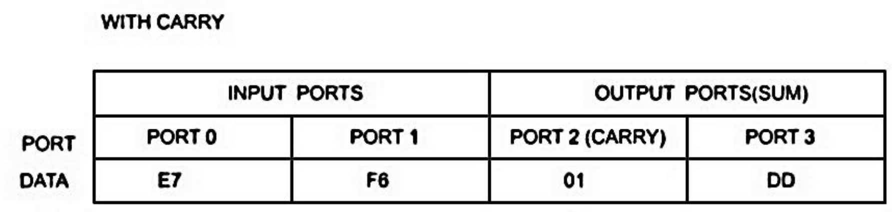
There are several issues that can arise when performing addition of two 8-bit numbers in 8051 microcontroller using ports:

1. Overflow: If the result of the addition exceeds 8 bits, the carry flag (CY) in the program status word (PSW) will be set. If this flag is not checked before storing the result, the result may be incorrect.
2. Input validation: Before performing the addition, it is important to validate the input to ensure that the numbers are within the range of 0 to 255. If the input is not validated, the result may be incorrect.

3. Port initialization: The ports used for input and output must be properly initialized before use. If the ports are not properly initialized, the data may not be transferred correctly.
4. Endianness: The order in which the bytes of the numbers are stored in memory can affect the result of the addition. It is important to ensure that the bytes are stored in the correct order before performing the addition.
5. Interrupts: If interrupts are enabled during the addition operation, the result may be affected. It is important to disable interrupts during critical operations to ensure the correct result.
6. Timing: The timing of the addition operation can affect the result. It is important to ensure that the necessary delays are added between instructions to ensure correct operation.
7. Code optimization: The code used to perform the addition should be optimized to ensure that it uses the least number of instructions and takes the least amount of time. This is important to avoid potential timing issues and to ensure that the microcontroller can perform other tasks while the addition is being performed.

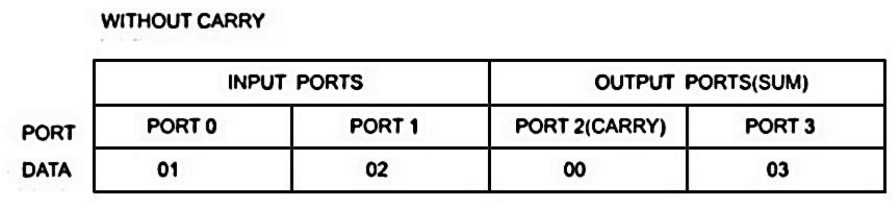
Problem: To write an assembly language program to add two 8 bit numbers in 8051 microcontroller using ports.

Example:



WITH CARRY

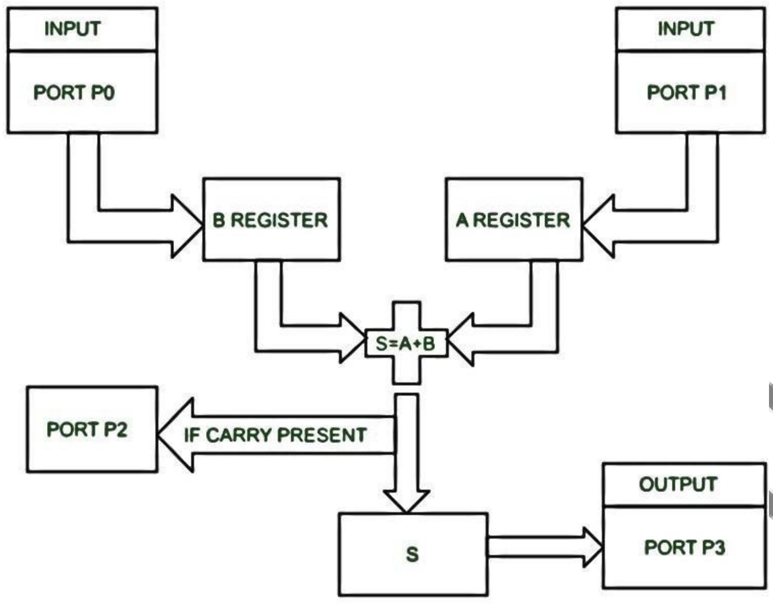
	INPUT PORTS		OUTPUT PORTS(SUM)	
PORT	PORT 0	PORT 1	PORT 2 (CARRY)	PORT 3
DATA	E7	F6	01	DD



WITHOUT CARRY

	INPUT PORTS		OUTPUT PORTS(SUM)	
PORT	PORT 0	PORT 1	PORT 2(CARRY)	PORT 3
DATA	01	02	00	03

Block diagram:



Algorithm:

- Initialize Ports P0 and P1 as input ports.
- Initialize Ports P2 and P3 as output ports.
- Initialize the R1 register.
- Move the contents from Port 0 to B register.
- Move the contents from Port 1 to A register.
- Add contents in A and B.
- If carry is present increment R1.
- Move contents in R1 to Port 2.
- Move the sum in step 6 to Port 3.

Program:

```
ORG 00H                  // Indicates starting address

MOV P0,#0FFH             // Initializes P0 as input port
MOV P1,#0FFH             // Initializes P1 as input port
MOV P2,#00H              // Initializes P2 as output port
MOV P3,#00H              // Initializes P3 as output port

L1:MOV R1, #00H          // Initializes Register R1
MOV B,P0                 // Moves content of P0 to B
MOV A,P1                 // Moves content of P1 to A
CLR C                    // Clears carry flag
ADD A,B                  // Add the content of A and B and store result in A
JNC L2                   // If carry is not set, jump to label L2
```

```
INC R1              // Increment Register R1 if carry present

L2: MOV P2, R1          // Moves the content from Register R1 to Port2
MOV P3,A              // Moves the content from A to Port3
SJMP L1              // Jumps to label L1
END
```

Explanation:

- ORG 00H is the starting address of the program.
- Giving the values as #0FFH and #00H initializes the ports as input and output ports respectively.
- R1 register is initialized to 0 so as to store any carry produced during the sum.
- MOV B, P0 moves the value present in P0 to the B register.
- MOV A, P1 moves the value present in P1 to Accumulator.
- ADD AB adds the values present in Accumulator and B register and stores the result in Accumulator.
- JNC L2 refers to jump to label L2 if no carry is present by automatically checking whether the carry bit is set or not.
- If the carry bit is set to increment register R1.
- MOV P2, R1, and MOV P3, A refers to moving the carry bit to P2 and result in Accumulator to P3.

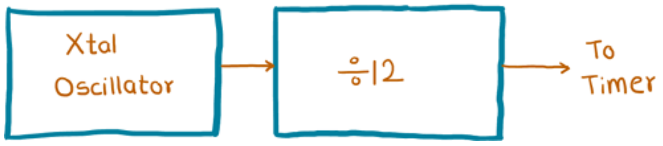
1.5. 8051 Timers

1.5.1. Introduction to 8051 Timers

8051 microcontrollers have two timers and counters which work on the clock frequency. Timer/counter can be used for time delay generation, counting external events, etc.

8051 Clock

Every Timer needs a clock to work, and 8051 provides it from an external crystal which is the main clock source for Timer. The internal circuitry in the 8051 microcontrollers provides a clock source to the timers which is 1/12th of the frequency of crystal attached to the microcontroller, also called Machine cycle frequency.



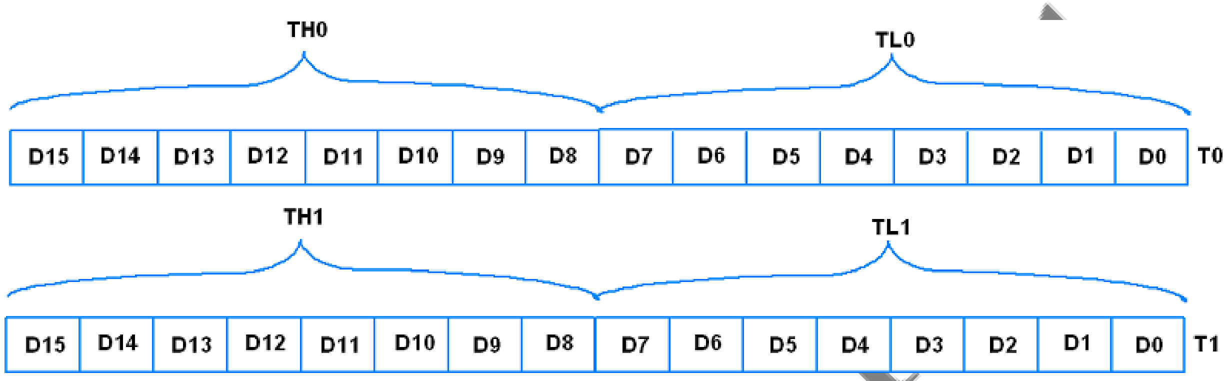
8051 Timer Clock

For example, suppose we have a crystal frequency of 11.0592 MHz then the microcontroller will provide 1/12th i.e.

Timer clock frequency= (Xtal Osc.frequency)/12 = (11.0592 MHz)/12 = 921.6 KHz
period T= 1/(921.6 kHz)=1.085 µS

1.5.2. 8051 Timer

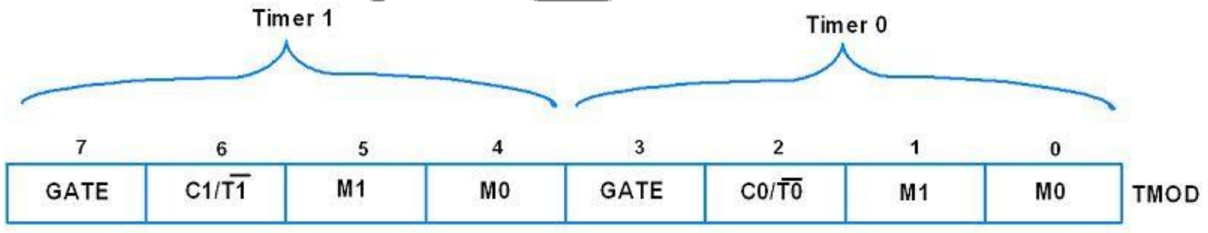
8051 has two timers Timer0 (T0) and Timer1 (T1), both are 16-bit wide. Since 8051 has 8-bit architecture, each of these is accessed by two separate 8-bit registers as shown in the figure below. These registers are used to load timer count.


TH0
TL0
D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
T0
TH1
TL1
D15 D14 D13 D12 D11 D10 D9 D8 D7 D6 D5 D4 D3 D2 D1 D0
T1

8051 has a Timer Mode Register and Timer Control Register for selecting a mode of operation and controlling purpose.
Let's see these registers,

1.5.2.1. TMOD register

TMOD is an 8-bit register used to set timer mode of timer0 and timer1.



Its lower 4 bits are used for Timer0 and the upper 4 bits are used for Timer1

Bit 7,3 – GATE:

1 = Enable Timer/Counter only when the INT0/INT1 pin is high and TR0/TR1 is set.

0 = Enable Timer/Counter when TR0/TR1 is set.

Bit 6,2 - C/$\overline{T}$ (Counter/Timer): Timer or Counter select bit

1 = Use as Counter

0 = Use as Timer

Bit 5:4 & 1:0 - M1:M0: Timer/Counter mode select bit

These are Timer/Counter mode select bit as per the below table

M1	M0	Mode	Operation
0	0	0 (13-bit timer mode)	13-bit timer/counter, 8-bit of THx & 5-bit of TLx
0	1	1 (16-bit timer mode)	16-bit timer/counter, THx cascaded with TLx
1	0	2 (8-bit auto-reload mode)	8-bit timer/counter (auto-reload mode), TLx reload with the value held by THx each time TLx overflow
1	1	3 (split timer mode)	Split the 16-bit timer into two 8-bit timers i.e. THx and TLx like two 8-bit timer

1.5.2.2. TCON Register

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

TCON is an 8-bit control register and contains a timer and interrupt flags.

Bit 7 - TF1: Timer1 Overflow Flag

1 = Timer1 overflow occurred (i.e. Timer1 goes to its max and roll over back to zero).

0 = Timer1 overflow not occurred.

It is cleared through software. In the Timer1 overflow interrupt service routine, this bit will get cleared automatically while exiting from ISR.

Bit 6 - TR1: Timer1 Run Control Bit

1 = Timer1 start.

0 = Timer1 stop.

It is set and cleared by software.

Bit 5 – TF0: Timer0 Overflow Flag

1 = Timer0 overflow occurred (i.e. Timer0 goes to its max and roll over back to zero).

0 = Timer0 overflow not occurred.

It is cleared through software. In the Timer0 overflow interrupt service routine, this bit will get cleared automatically while exiting from ISR.

Bit 4 – TR0: Timer0 Run Control Bit

1 = Timer0 start.

0 = Timer0 stop.

It is set and cleared by software.

Bit 3 - IE1: External Interrupt1 Edge Flag

1 = External interrupt1 occurred.

0 = External interrupt1 Processed.

It is set and cleared by hardware.

Bit 2 - IT1: External Interrupt1 Trigger Type Select Bit

1 = Interrupt occurs on falling edge at INT1 pin.

0 = Interrupt occur on a low level at the INT1 pin.

Bit 1 – IE0: External Interrupt0 Edge Flag

1 = External interrupt0 occurred.

0 = External interrupt0 Processed.

It is set and cleared by hardware.

Bit 0 – IT0: External Interrupt0 Trigger Type Select Bit

1 = Interrupt occurs on falling edge at INT0 pin.

0 = Interrupt occur on a low level at INT0 pin.

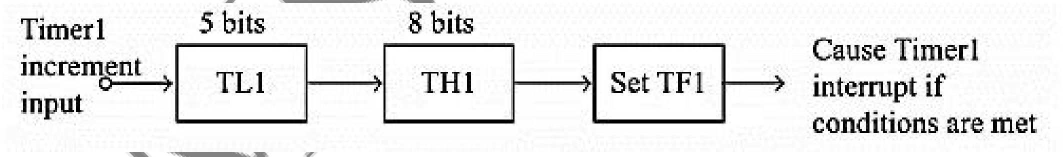
Let's see the timers modes

1.5.3. 8051 Timer Modes

Timers have their operation modes which are selected in the TMOD register using M0 & M1 bit combinations.

Mode 0 (13-bit timer mode)

The Mode 0 operation is the 8-bit timer or counter with a 5-bit pre-scaler. So it is a 13-bit timer/counter. It uses 5 bits of TL0 or TL1 and all of the 8-bits of TH0 or TH1.



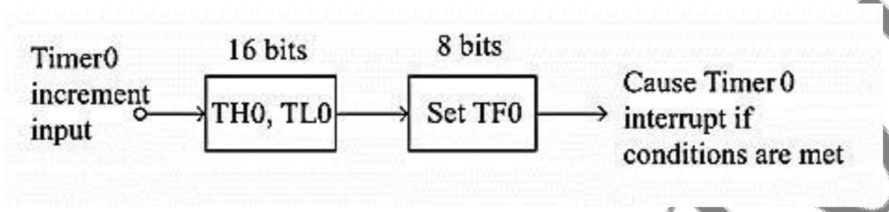
In this example the Timer1is selected, in this case, every 32 (25)event for counter operations or 32 machine cycles for timer operation, the TH1 register will be incremented by 1. When the TH1overflows from FFH to 00H, then the TF1 of TCON register will be high, and it stops the timer/counter. So for an example, we can say that if the TH1 is holding F0H, and it is in timer mode, then TF1will be high after 10H * 32 = 512 machine cycles.

```
MOVTMOD, #00H

MOVTH1, #0F0H

MOVIE, #88H

SETB TR1
```

In the above program, the Timer1 is configured as timer mode 0. In this case Gate = 0. Then the TH1 will be loaded with F0H, then enable the Timer1 interrupt. At last set the TR1 of TCON register, and start the timer.

Mode1 (16-bit timer mode)

The Mode 1 operation is the 16-bit timer or counter. In the following diagram, we are using Mode 1 for Timer0.



In this case every event for counter operations or machine cycles for timer operation, the TH0–TL0 register-pair will be incremented by 1. When the register pair overflows from FFFFH to 0000H, then the TF0 of TCON register will be high, and it stops the timer/counter. So for an example, we can say that if the TH0 – TL0 register pair is holding FFF0H, and it is in timer mode, then TF0 will be high after 10H = 16 machine cycles. When the clock frequency is 12MHz, then the following instructions generate an interrupt 16 µs after Timer0 starts running.

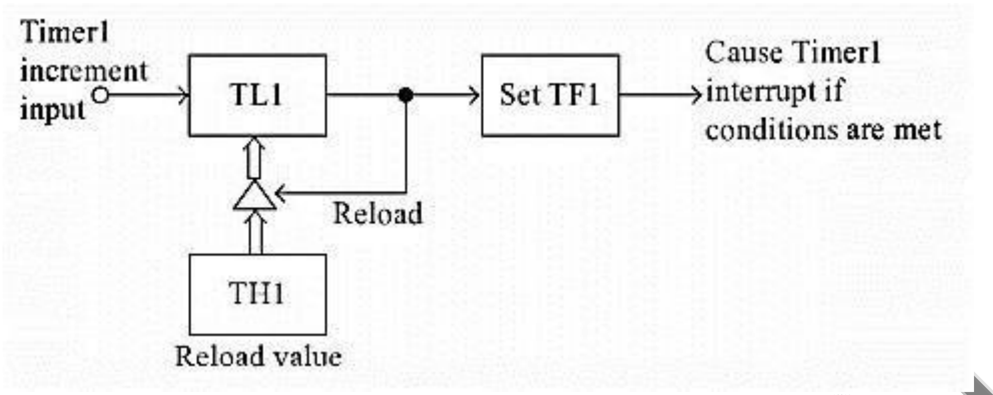
```
MOVTMOD, #01H
MOVTL0, #0F0H
MOVTH0, #0FFH
MOVIE, #82H
SETB TR0
```

In the above program, the Timer0 is configured as timer mode 1. In this case Gate = 0. Then the TL0 will be loaded with F0H and TH0 is loaded with FFH, then enable the Timer0 interrupt. At last set the TR0 of TCON register, and start the timer.

Mode2 (8-bit auto-reload timer mode)

The Mode 2 operation is the 8-bit auto reload timer or counter. In the following diagram, we are using Mode 2 for Timer1.

In this case every event for counter operations or machine cycles for timer operation, the TL1register will be incremented by 1. When the register pair overflows from FFH to 00H, then the TF1 of TCON register will be high, also theTL1 will be reloaded with the content of TH1 and starts the operation again.



So for an example, we can say that if the TH1 and TL1 register both are holding F0H and it is in timer mode, then TF1 will be high after 10H= 16 machine cycles. When the clock frequency is 12MHz this happens after 16 µs, then the following instructions generate an interrupt once every 16 µs after Timer1 starts running.

MOVTMOD, #20H

MOVTL1, #0F0H

MOVTH1, #0F0H

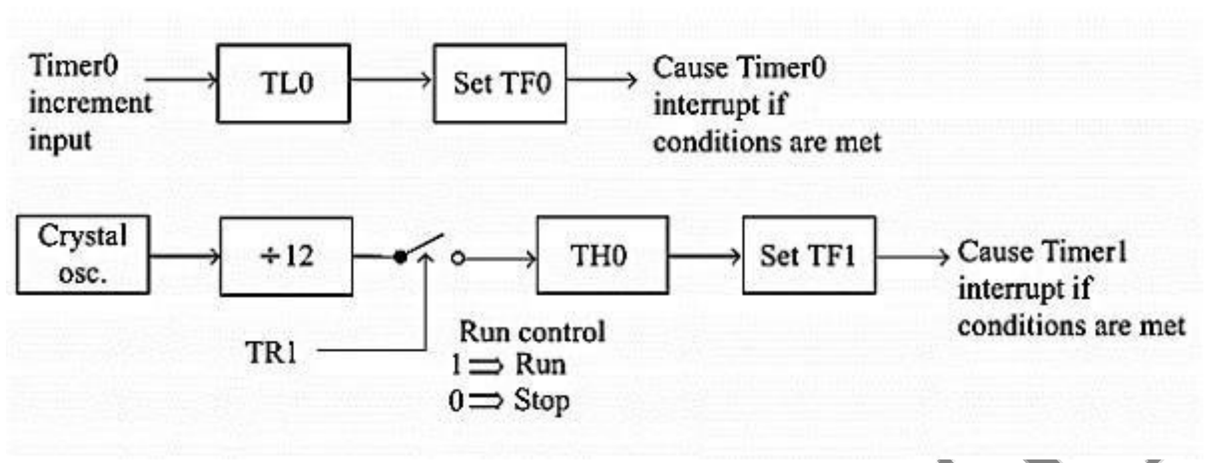
MOVIE, #88H

SETBTR1

In the above program, the Timer1 is configured as timer mode 2. In this case Gate = 0. Then the TL1 and TH1 are loaded with F0H. then enable the Timer1 interrupt. At last set the TR1 of TCON register, and start the timer.

Mode 3 of Timer/Counter

Mode 3 is different for Timer0 and Timer1. When the Timer0 is working in mode 3, the TL0 will be used as an 8-bit timer/counter. It will be controlled by the standard Timer0 control bits, T0 and INT0 inputs. The TH0 is used as an 8-bit timer but not the counter. This is controlled by Timer1 Control bit TR1. When the TH0 overflows from FFH to 00H, then TF1 is set to 1. In the following diagram, we can Timer0 in Mode 3.


Timer0 increment input
TL0
Set TF0
Cause Timer0 interrupt if conditions are met
Crystal osc.
÷12
TH0
Set TF1
Cause Timer1 interrupt if conditions are met
TR1
Run control
1⇒ Run
0⇒ Stop

When the Timer1 is working in Mode 3, it simply holds the count but does not run. When Timer0 is in mode 3, the Timer1 is configured in one of the mode 0, 1 and 2. In this case, the Timer1 cannot interrupt the microcontroller. When the TF1 is used by TH0 timer, the Timer1 is used as Baud Rate Generator.

The meaning of gate bit in Timer0 and Timer1 for mode 3 is as follows

It controls the running of 8-bit timer/counter TL0 as like Mode 0, 1, or 2. The running of TH0 is controlled by TR1 bit only. So the gate bit in this mode for Timer0 has no specific role.

The mode 3 is present for applications requiring an extra 8-bit timer/counter. In Mode 3 of Timer0, the 8051 has three timers. One 8-bit timer by TH0, another8-bit timer/counter by TL0, and one 16-bit timer/counter by Timer1.

If the Timer0 is in mode3, and Timer1 is working on either 0, 1 or 2, then the gun control of the Timer1 is activated when the gate bit is low or INT1 is high. The run control is deactivated when the gate is high and INT1 is low.

1.6. Serial Port in 8051

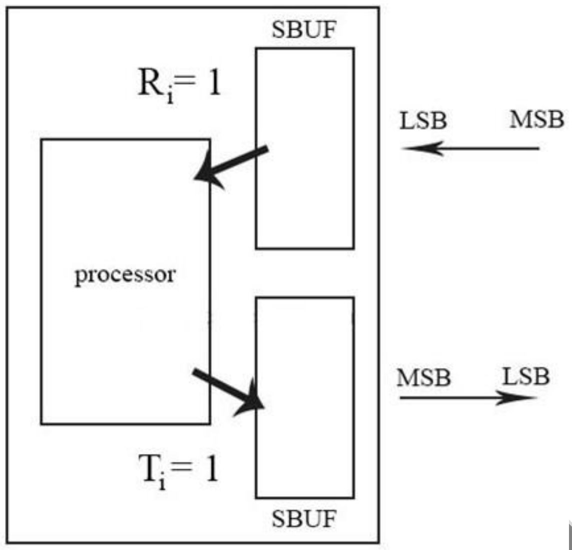
There is a serial port in 8051 as mentioned in the pin diagram of 8051. 8051 has capability to perform parallel as well as serial communication.

Parallel communication in 8051:

8051 can do 8-bit parallel communication as it has 8-bit ALU. For parallel communication, any of the ports (P0 / P1 / P2 / P3) is used as a transmission channel between transmitter and receiver.

Serial communication in 8051:

For serial communication there are two separate pins known as serial port of 8051.



TxD:
This pin basically acts as a transmitter (sending data), but in some other modes it doesn't do the job of transmitter. As it is serial communication, it sends bit by bit, the processor gives 8-bit at 1 time and those 8-bits are stored in a register named **SBUF.** Processor gives 1 byte of data that is to be transmitted to SBUF and from there bit by bit is transferred , firstly LSB and then at last MSB of the byte stored in SBUF. Once the total byte is transmitted, an interrupt is sent to the processor by making some flag 1, so that the processor can send more data for transmission as soon as the interrupt is received.

After every bit is transmitted, it requires delay for next bit transmission. So SBUF needs triggering which is provided by

- Timer T1 (here T1 only needs to trigger, T1 does not require its overflow flag , mode 3 in timers). Here we can vary the delay, so data transmission delay can be varied (frequency can be varied). It has a variable baud rate.
- There is an internal clock in 8051 (f_{osc} / 12 = 1Mhz) , where delay cannot be varied, this has fixed trigger delay. So frequency cannot be varied.It has a fixed baud rate.

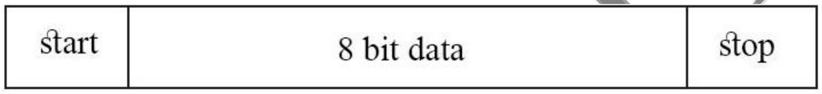
Whenever SBUF transferred 8bit of data , T_i flag becomes 1. Whenever processors go to ISR(in other interrupts the flag is auto cleared whenever processor goes to ISR) , in this the T_i flag is not auto cleared.

RxD:
This pin is basically for data reception . It received data bit by bit (as the transmitter sends LSB first, it received LSB first). There is also a register **SBUF** which stores 8 received bits. Once the 8 bits are received, instead of sending an interrupt it firstly checks for errors (errors caused due to transmission). Once there is no error in the received information R_i flag is set and an interrupt is sent to the processor. Processor goes to ISR (here also R_i is not cleared automatically).

How are SBUF in TxD and RxD different from each other ?
In SBUF of TxD, data is sent from processor to SBUF
In SBUF of RxD, data is sent from SBUF to the processor.
In this way both registers are differentiated by the processor.

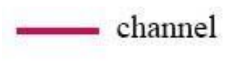
There is a bit named SM_2 ;
If $SM_2 = 1$, then after SBUF of RxD is filled there will be error check And if not then there will be no error check, directly an interrupt will be sent to the processor once SBUF in RxD is filled.

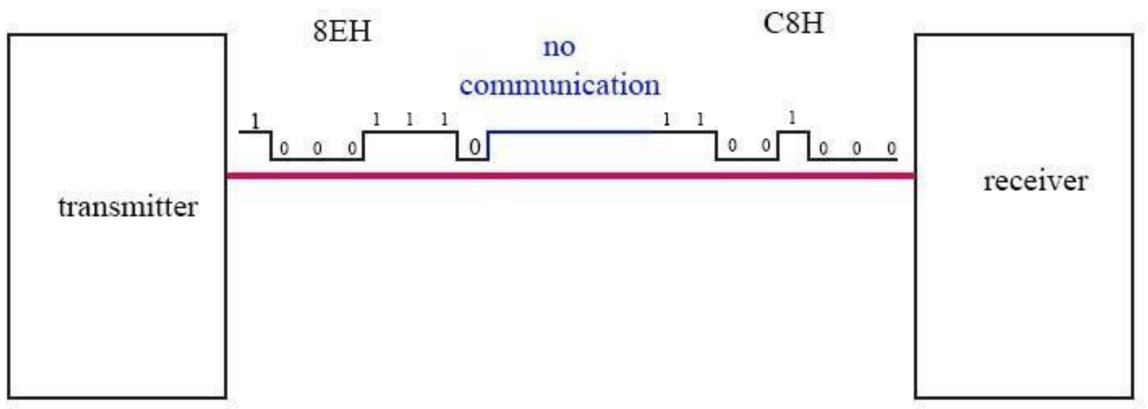
1.6.1. Modes in serial communication:
Mode 1 (8-bit UART communication):
UART stands for universal asynchronous receiver-transmitter. It means receiver and transmitter are asynchronous which mean they don’t have a common clock. Normally 8 bits are transmitted through the channel, but in this mode 10 bits are transmitted.



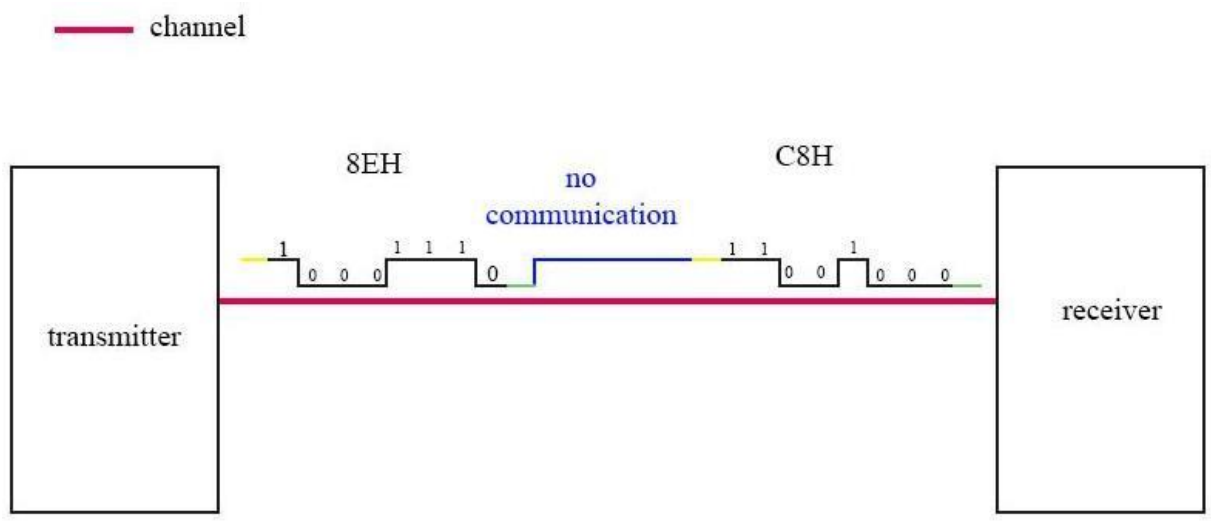
Here start and stop bits are system generated.

Significance of start and stop bit:
For eg: There is a transmitter and receiver. First C8H (1100 1000) data is transmitted and then for 10min there is no transmission and after that again 8EH (1000 1110)is transmitted.
If there is no transmission, last bit transmitted would be remained in the channel

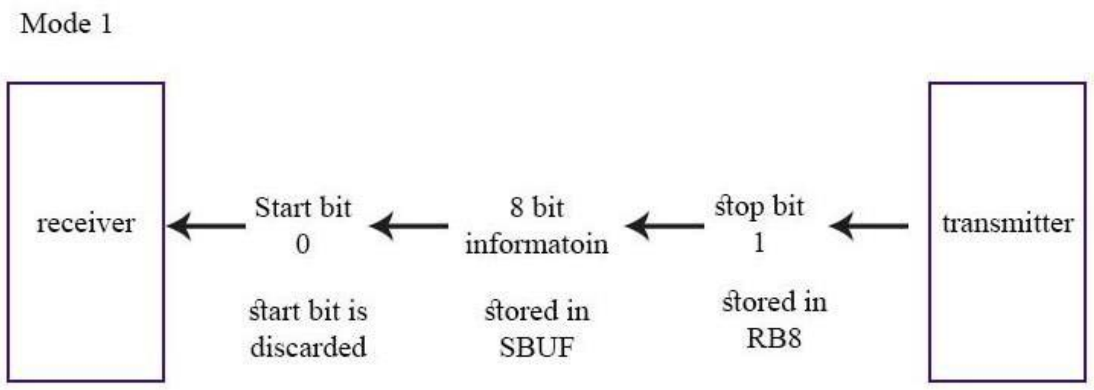




Receiver assumes blue data (when there was no communication) also as data.



Here green is the start bit which is zero. And then the yellow colored is stop bit which is 1. Whenever the first 0 bit comes, the receiver discards the start bit and accepts the next 8 bits and stores in SBUF. Then the 9th bit is 1 , this bit is stored in **R$_{B8}$** (will be discussed later). Then after this whenever the next zero bit comes (that zero bit is discarded and accepts the next 8 bits and so on).



Stop bit is also used for error checking. Whenever SM$_2$=1, It checks for error, If the RB8 = 1 (which means stop = 1 received, so the data is received correctly) and if RB8=0 (transmitter generated stop as 1, but received as 0) so there is an error. If there is an error in received data, no interrupt is sent to the processor.

This mode is variable baud rate, which means it is triggered by timer 1.

Mode 2 (9-bit UART communication):

start	8 bit data	9th bit	stop

The 9th bit is a programmable bit and it is given through TB8. Here 9th bit is 1 and it is used for error checking and stop bit for triggering the data high (so start bit gets 0 and so on).

Why the 9th bit , when the already stop bit exists?
Standard value of 9th bit is 1 and can be made 0.
Whenever SM_2 = 1(receiver accepts only errorless data) and if 9th bit is 1, then only errorless data is accepted or else discarded. Discarding data is a purpose.

Mode 3 (9-bit UART communication):
This mode is completely similar to mode 2, in mode 2 for triggering timer is used Whereas in this mode internal clock is used for triggering. It has a fixed baud rate.

Mode 0 :
Totally there were four modes in serial port of 8051, but for better understanding mode 0 is explained after three modes. In this mode data is transferred and received only through the RxD channel. TxD is used for clocks. This is synchronous mode of communication.
Such a system is also known as half duplex mode. It has fixed baud rate.

SCON register:

SM_0	SM_1	SM_2	REN	TB_8	RB_8	T_i	R_i

SM_0 and SM_1:
These are used to select the mode.

SM_0	**SM_1**	**Mode**
0	**0**	**0**
0	**1**	**1**
1	**0**	**2**
1	**1**	**3**

SM_2:
If SM_2 = 1, error is checked Or else no error checking is done.
REN:
Receiver enable, If REN=1, receiver will receive the data or else not.
TB_8:
This is the 9th bit to be transmitted.
RB_8:
This is the 9th bit to be received.
T_i :
When 8-bits are received in SBUF , then R_i = 1, that would send an interrupt to the processor.

R_i :
When 8-bits are sent from SBUF,and SBUF is empty , then R_i = 1, that would send an interrupt to the processor. Before R_i=1, it checks for error based on SM_2.

1.7. 8-Bit Interrupts

Interrupts are events detected by the MCU which cause normal program flow to be pre-empted. Interrupts pause the current program and transfer control to a specified user-written firmware routine called the Interrupt Service Routine (ISR). The ISR processes the interrupt event, then resumes normal program flow.

1.7.1. Overview of Interrupt Process

Program MCU to react to interrupts
The MCU must be programmed to enable interrupts to occur. Setting the Global Interrupt Enable (GIE) and, in many cases, the Peripheral Interrupt Enable (PEIE), enables the MCU to receive interrupts. GIE and PEIE are located in the Interrupt Control (INTCON) special function register.

Enable interrupts from selected peripherals
Each peripheral on the MCU has an individual enable bit. A peripheral's individual interrupt enable bit must be set, in addition to GIE/PEIE, before the peripheral can generate an interrupt. The individual interrupt enable bits are located in INTCON, PIE1, PIE2, and PIE3.

Peripheral asserts an interrupt request
When a peripheral reaches a state where program intervention is needed, the peripheral sets an Interrupt Request Flag (xxIF). These interrupt flags are set regardless of the status of the GIE, PEIE, and individual interrupt enable bits. The interrupt flags are located in INTCON, PIR1, PIR2, and PIR3.
The interrupt request flags are latched high when set and must be cleared by the user-written ISR.

Interrupt occurs
When an interrupt request flag is set and the interrupt is properly enabled, the interrupt process begins:

1. Global Interrupts are disabled by clearing GIE to 0.
2. The current program context is saved to the shadow registers.
3. The value of the Program Counter is stored on the return stack.
4. Program control is transferred to the interrupt vector at address 04h.

ISR runs
The ISR is a function written by the user and placed at address 04h. The ISR does the following:

1. Checks the interrupt-enabled peripherals for the source of the interrupt request.
2. Performs the necessary peripheral tasks.
3. Clears the appropriate interrupt request flag.
4. Executes the Return From Interrupt instruction (RETFIE) as the final ISR instruction.

Control is returned to the Main program

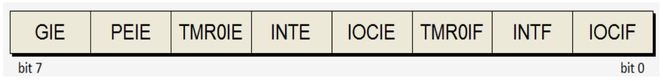
When RETFIE is executed:

1. Global Interrupts are enabled (GIE=1).
2. The program context is restored from the Shadow registers.
3. The return address from the stack is loaded into the Program Counter.
4. Execution resumes from point at which it was interrupted.

1.7.2. Registers Used to Process Interrupts

Interrupt Control Register

INTCON register



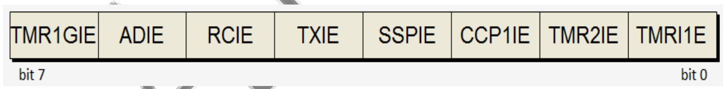
GIE	PEIE	TMR0IE	INTE	IOCIE	TMR0IF	INTF	IOCIF
bit 7							bit 0

GIE - Global Interrupt Enable
PEIE - Peripheral Interrupt Enable
TMR0IE - Timer0 Interrupt Enable
INTE - External Interrupt Enable
IOCIE -Interrupt on Change Enable
TMR0IF - Timer0 Interrupt flag
INTF - External Interrupt flag
IOCIF -Interrupt on Change flag

INTCON contains global and peripheral interrupt enable flags as well as the individual interrupt request flags and interrupt enable flags for three of the PIC16F1xxxx interrupts.

Interrupt Enable Registers

PIE1 register



TMR1GIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMRI1E
bit 7							bit 0

TMR1GIE - Timer1 Gate Interrupt Enable
ADIE - Analog-to-Digital Converter (ADC) Interrupt Enable
RCIE - Universal Synchronous Asynchronous Receiver Tranmsitter (USART) Receive Interrupt Enable
TXIE - USART Transmit Interrupt Enable
SSPIE - Synchronous Serial Port (MSSP) Interrupt Enable
CCP1IE - CCP1 Interrupt Enable

TMR2IE - Timer2 Interrupt Enable

TMR1IE - Timer1 Interrupt Enable

PIE2 register

OSFIE	C2IE	C1IE	EEIE	BCLIE	LCDIE	----	CCP2IE
bit 7							bit 0

OSFIE - Oscillator Fail Interrupt Enable

C2IE - Comparator C2 Interrupt Enable

C1IE - Comparator C1 Interrupt Enable

EEIE - EEPROM Write Completion Interrupt Enable

BCLIE - MSSP Bus Collision Interrupt Enable

LCDIE - LCD Module Interrupt Enable

--- - Unimplemented, read as 0

CCP2IE - CCP2 Interrupt Enable

PIE3 register

----	CCP5IE	CCP4IE	CCP3IE	TMR6IE	----	TMR4IE	----
bit 7							bit 0

--- - Unimplemented read as 0

CCP5IE - CCP5 Interrupt Enable

CCP4IE - CCP4 Interrupt Enable

CCP3IE - CCP3 Interrupt Enable

TMR6IE - Timer6 Interrupt Enable

--- - Unimplemented, read as 0

TMR4IE - Timer4 Interrupt Enable

--- - Unimplemented, read as 0

PIE1, **PIE2**, and **PIE3** contain the individual interrupt enable flags for the MCU's peripherals.

Interrupt Request Registers

PIR1 register

TMR1GIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMRI1F
bit 7							bit 0

TMR1GIF - Timer1 Gate Interrupt Flag

ADIF - ADC Interrupt Flag

RCIF - USART Receive Interrupt Flag

TXIF - USART Transmit Interrupt Flag

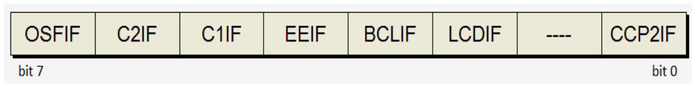
SSPIF - MSSP Interrupt Flag

CCP1IF - CCP1 Interrupt Flag

TMR2IF - Timer2 Interrupt Flag

TMR1IF - Timer1 Interrupt Flag

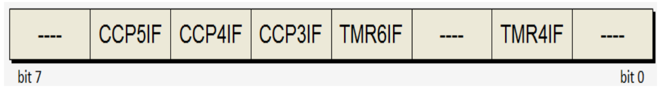
PIR2 register



bit 7							bit 0
OSFIF	C2IF	C1IF	EEIF	BCLIF	LCDIF	----	CCP2IF

OSFIF - Oscillator Fail Interrupt Flag
C2IF - Co parator C2 Interrupt Flag
C1IF - Co parator C1 Interrupt Flag
EEIF - EI PROM Write Completion Interrupt Flag
BCLIF - MSSP Bus Collision Interrupt Flag
LCDIF - LCD Module Interrupt Flag
--- - Unimplemented, read as 0
CCP2IF - CCP2 Interrupt Flag

PIR3 register



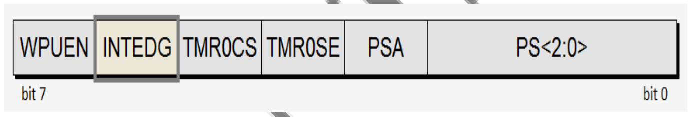
bit 7							bit 0
----	CCP5IF	CCP4IF	CCP3IF	TMR6IF	----	TMR4IF	----

--- - Unimplemented, read as 0
CCP5IF - CCP5 Interrupt Flag
CCP4IF - CCP4 Interrupt Flag
CCP3IF - CCP3 Interrupt Flag
TMR6IF - Timer6 Interrupt Flag
--- - Unimp emented, read as 0
TMR4IF - Timer4 Interrupt Flag
--- - Unimp emented, read as 0

PIR1, **PIR2**, and **PIR3** contain the individual interrupt request flags for the MCU's peripherals.

OPTION_REG

OPTION_REG



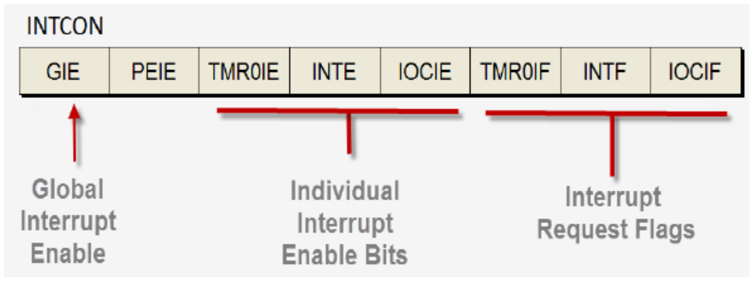
bit 7							bit 0
WPUEN	INTEDG	TMR0CS	TMR0SE	PSA	PS<2:0>		

The INTEDG flag in OPTION_REG is used to set a rising or falling edge on the INT pin as the trigger for an INTE interrupt.

Enabling Interrupts

Core Interrupts

Three interrupt sources (Timer0, External Interrupt, and Interrupt on Change) have interrupt enable bits located in INTCON. These interrupts are referred to as core interrupts.


INTCON
GIE
PEIE
TMR0IE
INTE
IOCIE
TMR0IF
INTF
IOCIF
Global Interrupt Enable
Individual Interrupt Enable Bits
Interrupt Request Flags

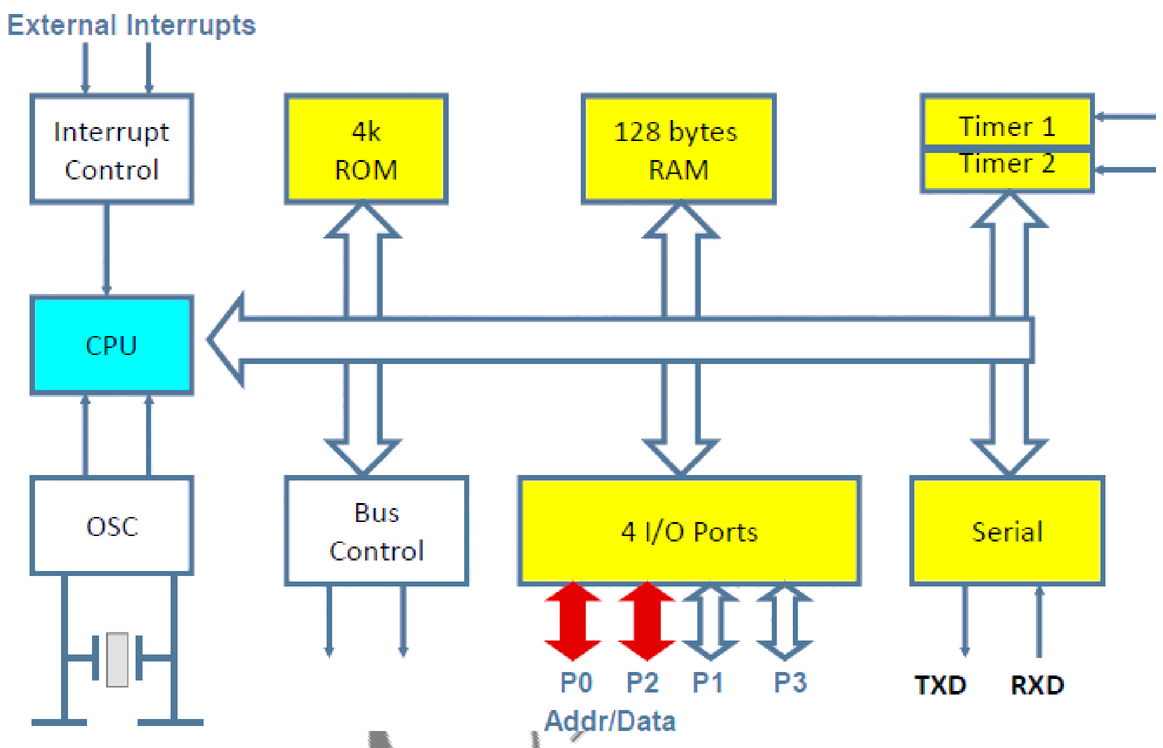
UNIT II

EMBEDDED C PROGRAMMING

- Memory And I/O Devices Interfacing
- Programming Embedded Systems in C
- Need For RTOS
- Multiple Tasks and Processes
- Context Switching
- Priority Based Scheduling Policies

2.1. Memory and I/O Interfacing

Several memory chips and I/O devices are connected to a microprocessor. The following figure shows a schematic diagram to interface memory chips and I/O devices to a microprocessor.



2.1.1. Memory Interfacing

When we are executing any instruction, the address of memory location or an I/O device is sent out by the microprocessor. The corresponding memory chip or I/O device is selected by a decoding circuit.

Memory requires some signals to read from and write to registers and microprocessor transmits some signals for reading or writing data.

The interfacing process includes matching the memory requirements with the microprocessor signals. Therefore, the interfacing circuit should be designed in such a way that it matches the memory signal requirements with the microprocessor's signals.

2.1.2. I/O interfacing

As we know, keyboard and displays are used as communication channel with outside world. Therefore, it is necessary that we interface keyboard and displays with the microprocessor. This is called I/O interfacing. For this type of interfacing, we use latches and buffers for interfacing the keyboards and displays with the microprocessor.

But the main drawback of this interfacing is that the microprocessor can perform only one function.

8051 Microcontroller Memory Organization

In the 8051 Microcontroller, we have seen the 8051 Microcontroller Introduction and Basics, Pin Diagram, Pin Description and the Architecture overview. we will continue exploring 8051 Microcontroller by understanding the 8051 Microcontroller Memory Organization, Program Memory (ROM), Data Memory (RAM), External Memory.

Differences between microprocessor and microcontroller

The main difference can be stated as on-chip memory i.e., a Microcontroller has both Program Memory (ROM) and Data Memory (RAM) on the same chip (IC), whereas a Microprocessor has to be externally interfacing with the memory modules.

Hence, it is clear that the memory is an important part of the 8051 Microcontroller Architecture (for that matter, any Microcontroller). So, it is important for us to understand the 8051 Microcontroller Memory Organization i.e., how memory is organized, how the processor accesses each memory and how to interface external memory with 8051 Microcontroller.

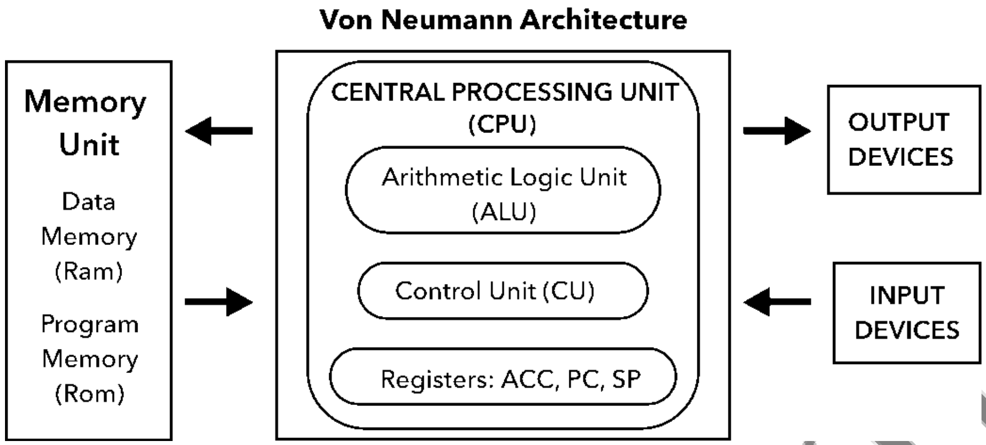
Before going in to the details of the 8051 Microcontroller Memory Organization, we will first see a little bit about the Computer Architecture and then proceed with memory organization of 8051 Microcontroller.

1.2.3. Types of Computer Architecture

Basically, Microprocessors or Microcontrollers are classified based on the two types of Computer Architecture: Von Neumann Architecture and Harvard Architecture.

Von Neumann Architecture

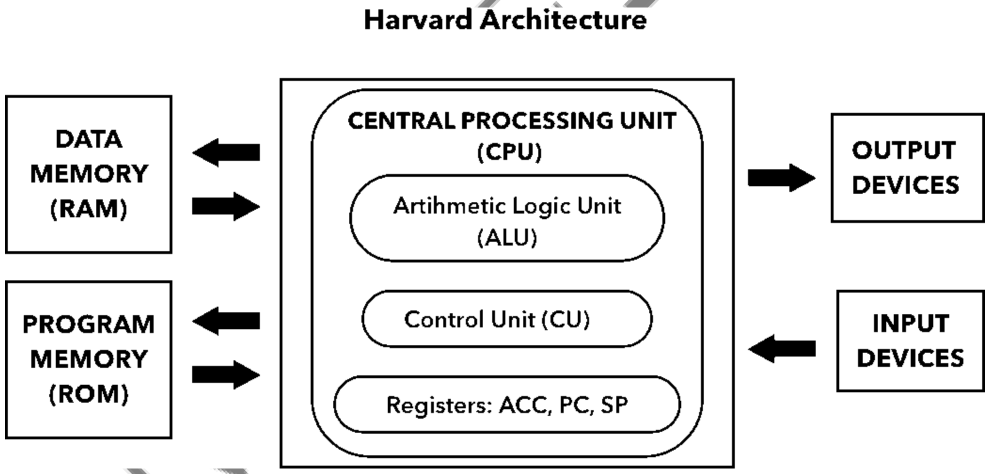
Von Neumann Architecture or Princeton Architecture is a Computer Architecture, where the Program i.e., the Instructions and the Data are stored in a single memory. Since the Instruction Memory and the Data Memory are the same, the Processor or CPU cannot access both Instructions and Data at the same time as they use a single bus. This type of architecture has severe limitations to the performance of the system as it creates a bottleneck while accessing the memory.



Harvard Architecture

Harvard Architecture, in contrast to Von Neumann Architecture, uses separate memory for Instruction (Program) and Data. Since the Instruction Memory and Data Memory are separate in a Harvard Architecture, their signal paths i.e., buses are also different and hence, the CPU can access both Instructions and Data at the same time.

Almost all Microcontrollers, including 8051 Microcontroller implement Harvard Architecture.



1.2.4. 8051 Microcontroller Memory Organization

The 8051 Microcontroller Memory is separated in Program Memory (ROM) and Data Memory (RAM). The Program Memory of the 8051 Microcontroller is used for storing the program to be executed i.e., instructions. The Data Memory on the other hand, is used for storing temporary variable data and intermediate results.

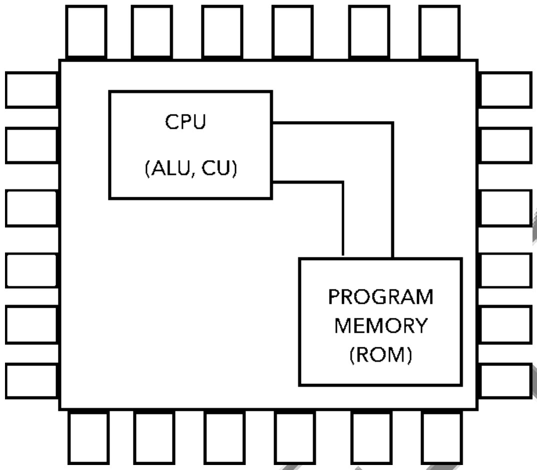
8051 Microcontroller has both Internal ROM and Internal RAM. If the internal memory is inadequate, you can add external memory using suitable circuits.

1.2.4.1. Program Memory (ROM) of 8051 Microcontroller

In 8051 Microcontroller, the code or instructions to be executed are stored in the Program Memory, which is also called as the ROM of the Microcontroller. The original 8051 Microcontroller by Intel has 4KB of internal ROM.

Some variants of 8051 like the 8031 and 8032 series doesn't have any internal ROM (Program Memory) and must be interfaced with external Program Memory with instructions loaded in it.

8051 Program Memory

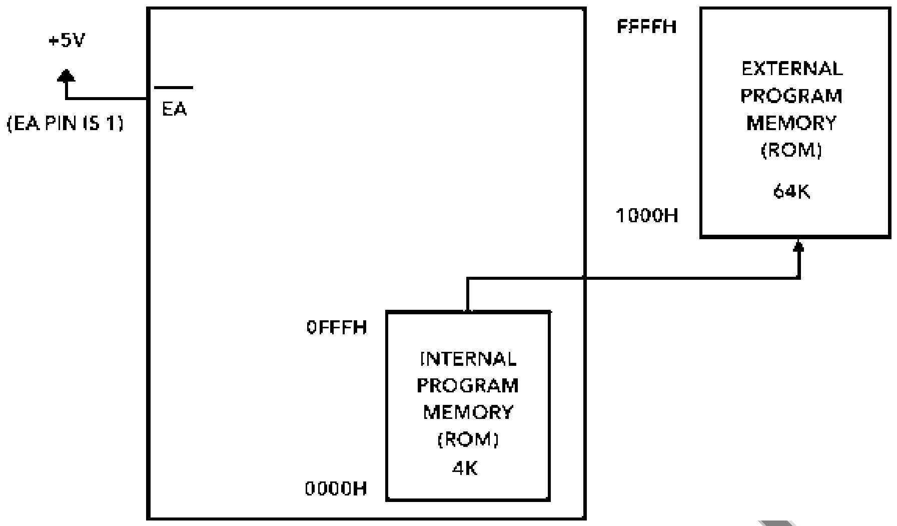


Almost all modern 8051 Microcontrollers, like 8052 Series, have 8KB of Internal Program Memory (ROM) in the form of Flash Memory (ROM) and provide the option of reprogramming the memory.

In case of 4KB of Internal ROM, the address space is 0000H to 0FFFH. If the address space i.e., the program addresses exceed this value, then the CPU will automatically fetch the code from the external Program Memory.

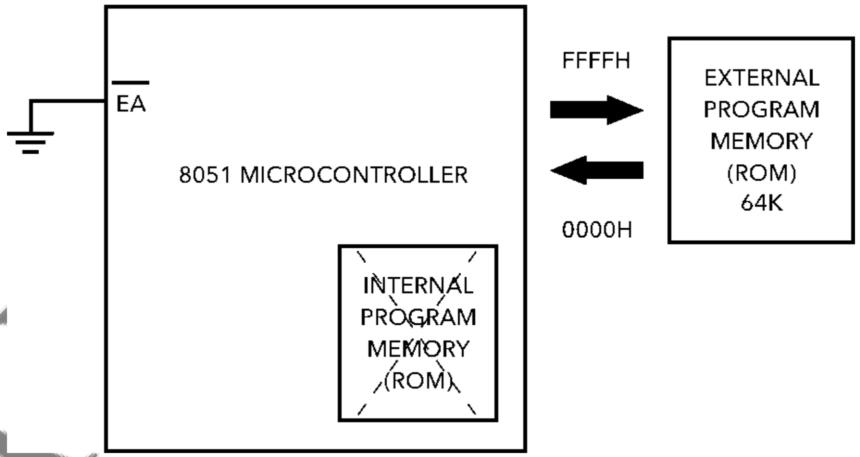
For this, the External Access Pin (EA Pin) must be pulled HIGH i.e., when the EA Pin is high, the CPU first fetches instructions from the Internal Program Memory in the address range of 0000H to 0FFFFH and if the memory addresses exceed the limit, then the instructions are fetched from the external ROM in the address range of 1000H to FFFFH.

Using Both Internal And External Program Memory With 8051


+5V
(EA PIN IS 1)
EA
FFFFH
EXTERNAL PROGRAM MEMORY (ROM) 64K
1000H
0FFFH
INTERNAL PROGRAM MEMORY (ROM) 4K
0000H

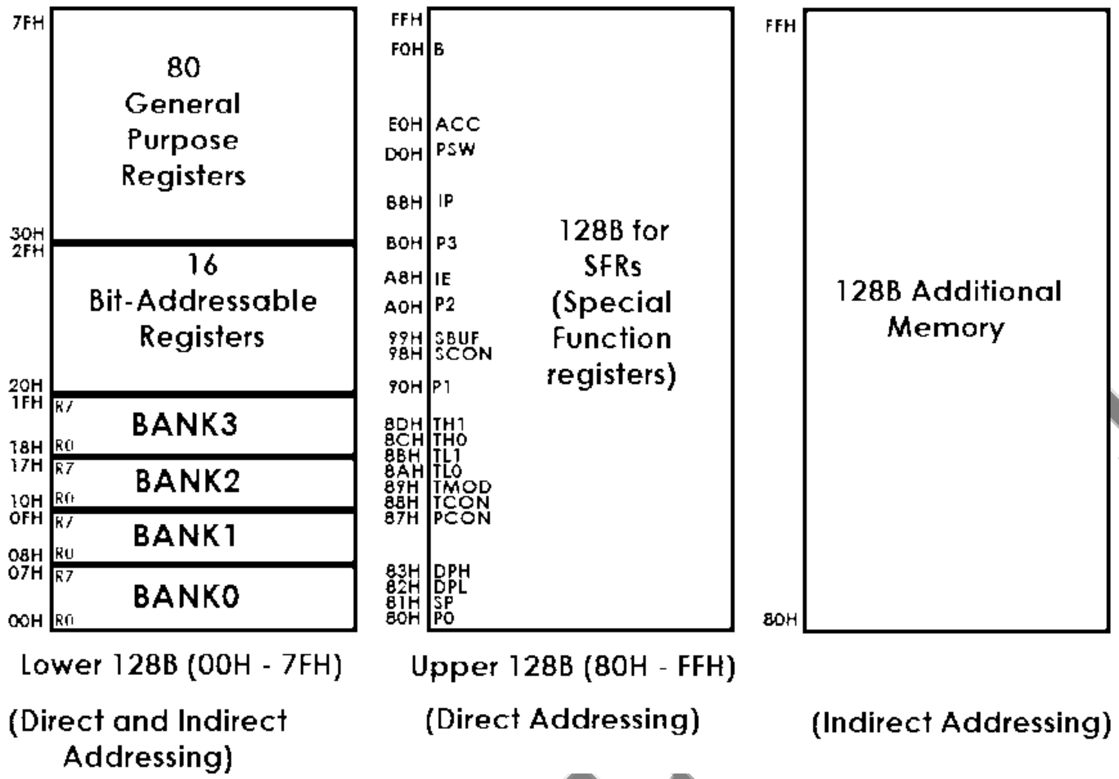
There is another way to fetch the instructions: ignore the Internal ROM and fetch all the instructions only from the External Program Memory (External ROM). For this scenario, the EA Pin must be connected to GND. In this case, the memory addresses of the external ROM will be from 0000H to FFFFH.

Using Only External Program Memory With 8051


EA
8051 MICROCONTROLLER
FFFFH
0000H
EXTERNAL PROGRAM MEMORY (ROM) 64K
INTERNAL PROGRAM MEMORY (ROM)

1.2.4.2. Data Memory (RAM) of 8051 Microcontroller

The Data Memory or RAM of the 8051 Microcontroller stores temporary data and intermediate results that are generated and used during the normal operation of the microcontroller. Original Intel's 8051 Microcontroller had 128B of internal RAM.



But almost all modern variants of 8051 Microcontroller have 256B of RAM. In this 256B, the first 128B i.e., memory addresses from 00H to 7FH is divided in to Working Registers (organized as Register Banks), Bit – Addressable Area and General Purpose RAM (also known as Scratchpad area).
In the first 128B of RAM (from 00H to 7FH), the first 32B i.e., memory from addresses 00H to 1FH consists of 32 Working Registers that are organized as four banks with 8 Registers in each Bank.

The 4 banks are named as Bank0, Bank1, Bank2 and Bank3. Each Bank consists of 8 registers named as R0 – R7. Each Register can be addressed in two ways: either by name or by address. To address the register by name, first the corresponding Bank must be selected. In order to select the bank, we have to use the RS0 and RS1 bits of the Program Status Word (PSW) Register (RS0 and RS1 are 3rd and 4th bits in the PSW Register).

When addressing the Register using its address i.e., 12H for example, the corresponding Bank may or may not be selected. (12H corresponds to R2 in Bank2).

The next 16B of the RAM i.e., from 20H to 2FH are Bit – Addressable memory locations. There are totally 128 bits that can be addressed individually using 00H to 7FH or the entire byte can be addressed as 20H to 2FH.

1.2.5. Interfacing External Memory with 8051 Microcontroller

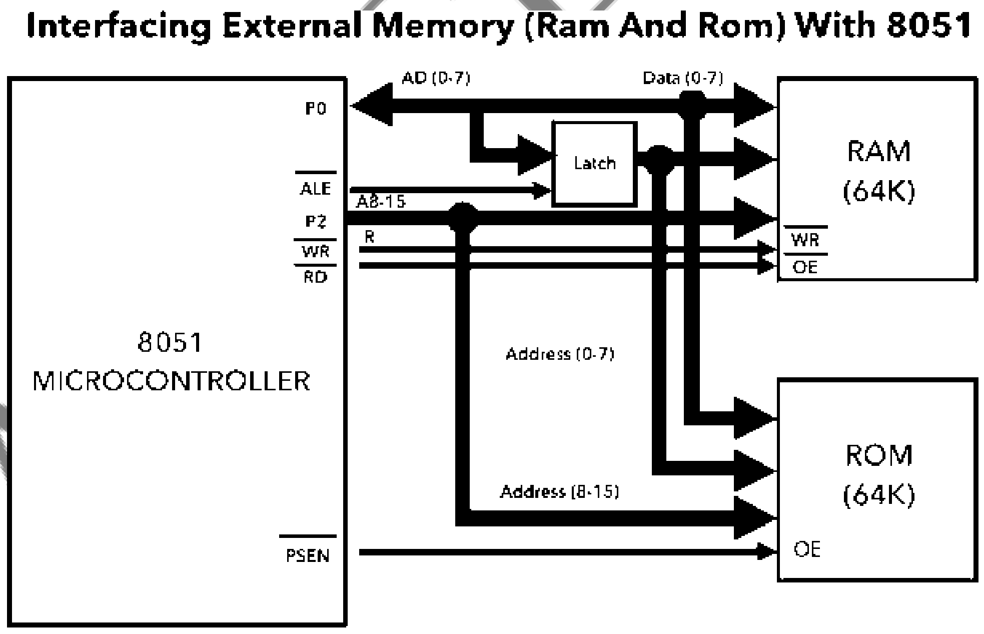
It is always good to have an option to expand the capabilities of a Microcontroller, whether it is in terms of Memory or IO or anything else. Such expansion will be useful to avoid design throttling. We have seen that a typical 8051 Microcontroller has 4KB of ROM and 128B of RAM (most modern 8051 Microcontroller variants have 8K ROM and 256B of RAM).

The designer of an 8051 Microcontroller based system is not limited to the internal RAM and ROM present in the 8051 Microcontroller. There is a provision of connecting both external RAM and ROM i.e., Data Memory and Program.

The reason for interfacing external Program Memory or ROM is that complex programs written in high – level languages often tend to be larger and occupy more memory.
Another important reason is that chips like 8031 or 8032, which doesn't have any internal ROM, have to be interfaced with external ROM.

A maximum of 64KB of Program Memory (ROM) and Data Memory (RAM) each can be interface with the 8051 Microcontroller.

The following image shows the block diagram of interfacing 64KB of External RAM and 64KB of External ROM with the 8051 Microcontroller.



An important point to remember when interfacing external memory with 8051 Microcontroller is that Port 0 (P0) cannot be used as an IO Port as it will be used for multiplexed address and data bus (A0 – A7 and D0 – D7). Not always, but Port 2 may be used as higher byte of the address bus.

2.2. Embedded C Programming with Keil Language

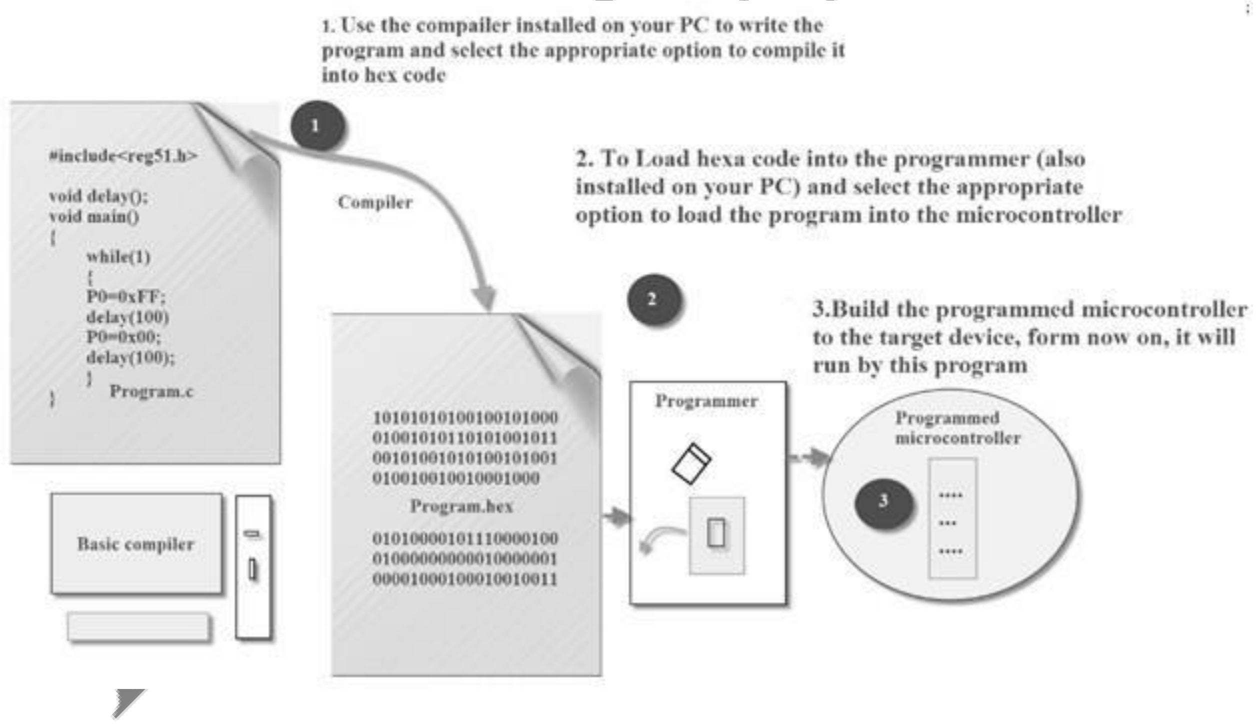
Embedded C is most popular programming language in software field for developing electronic gadgets. Each processor used in electronic system is associated with embedded software.

Embedded C programming plays a key role in performing specific function by the processor. In day-to-day life we used many electronic devices such as mobile phone, washing machine, digital camera, etc. These all device working is based on microcontroller that are programmed by embedded C.

In embedded system programming C code is preferred over other language. Due to the following reasons:

- Easy to understand
- High Reliability
- Portability
- Scalability

Let's see the block diagram representation of embedded system programming:


1. Use the compailer installed on your PC to write the program and select the appropriate option to compile it into hex code

2. To Load hexa code into the programmer (also installed on your PC) and select the appropriate option to load the program into the microcontroller

3.Build the programmed microcontroller to the target device, form now on, it will run by this program

```
#include<reg51.h>

void delay();
void main()
{
      while(1)
      {
      P0=0xFF;
      delay(100)
      P0=0x00;
      delay(100);
      }
}
```

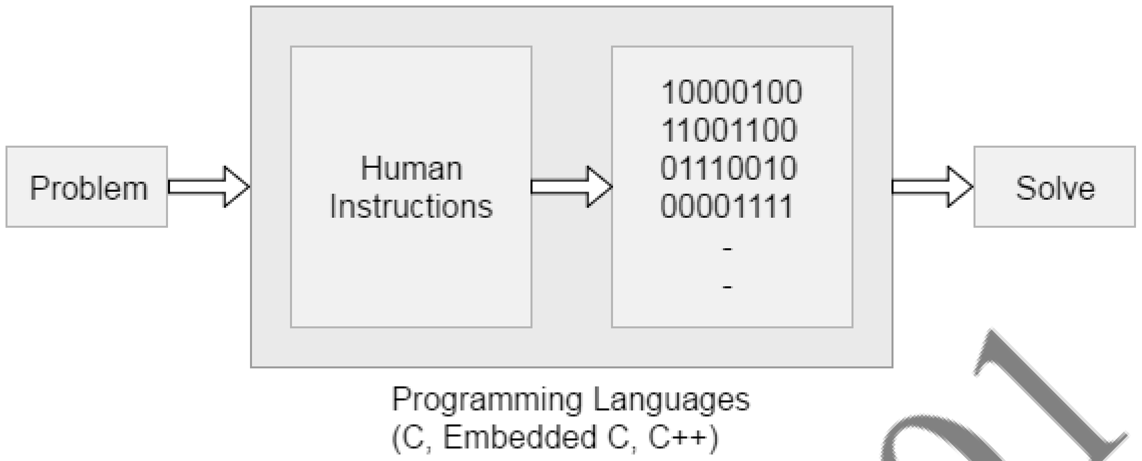
```
1010101010010010100O
01001010110101001011
00101001010100101001
010010010010001000
```

```
0101000010111000010O
01000000000010000001
00001000100010010011
```

2.2.1. Embedded System Programming:

Basic Declaration

Let's see the block diagram of Embedded C Programming development:

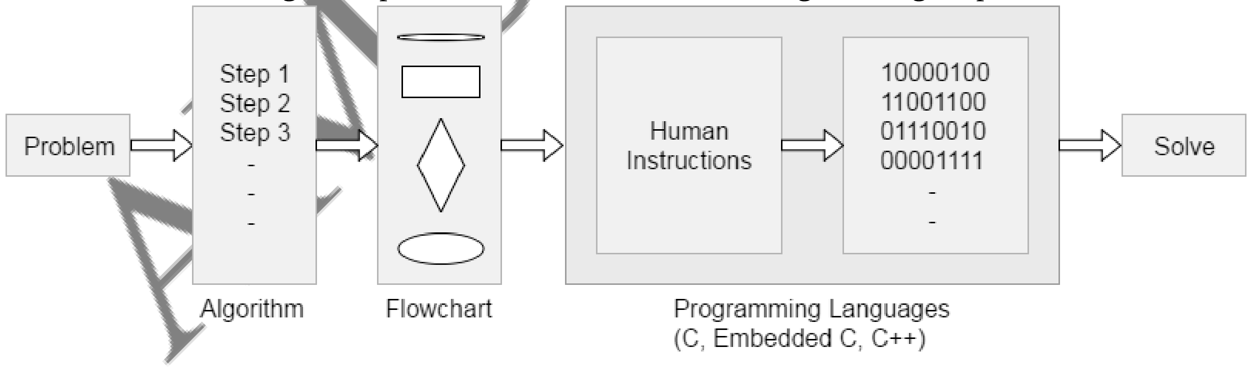


Function is a collection of statements that is used for performing a specific task and a collection of one or more functions is called a programming language. Every language is consisting of basic elements and grammatical rules. The C language programming is designed for function with variables, character set, data types, keywords, expression and so on are used for writing a C program.

The extension in C language is known as embedded C programming language. As compared to above the embedded programming in C is also have some additional features like data types, keywords and header file etc is represented by

#include<**microcontroller** name.h>

Basic Embedded C Programming Steps
Let's see the block diagram representation of Embedded C Programming Steps:



The microcontroller programming is different for each type of operating system. Even though there are many operating system are exist such as Windows, Linux, RTOS, etc but RTOS has several advantage for embedded system development.

2.2.2. Basics of Embedded C Program

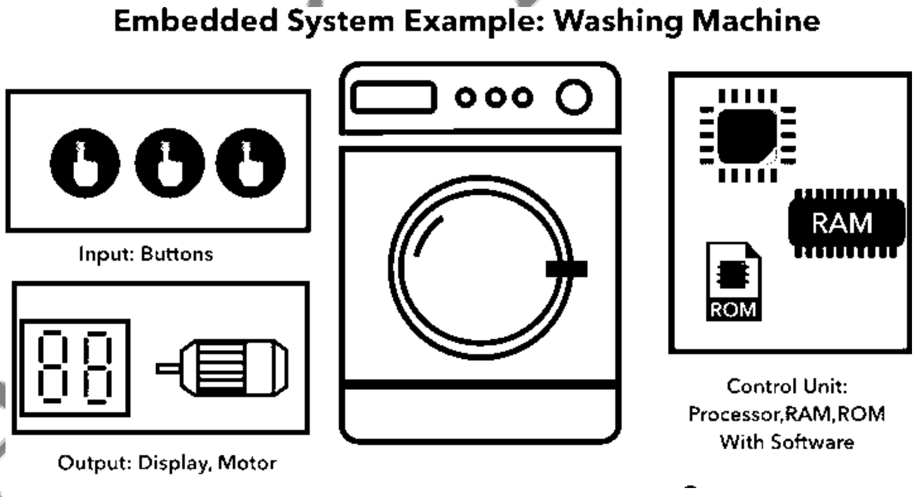
Embedded C is one of the most popular and most commonly used Programming Languages in the development of Embedded Systems. So, we will see some of the Basics of Embedded C Program and the Programming Structure of Embedded C.

2.2.2.1. What is an Embedded System?

An Embedded System is a combination of Hardware and Software. My desktop computer also has hardware and software. Does that mean a desktop computer is also an Embedded System? NO. A desktop computer is considered a general purpose system as it can do many different tasks that too simultaneously. Some common tasks are playing videos, working on office suites, editing images (or videos), browsing the web, etc.

An Embedded System is more of an application oriented system i.e. it is dedicated to perform a single task (or a limited number of tasks, but all working for a single main aim).

An example for embedded system, which we use daily, is a Wireless Router. In order to get wireless internet connectivity on our mobile phones and laptops, we often use routers. The task of a wireless router is to take the signal from a cable and transmit it wirelessly. And take wireless data from the device (like a mobile) and send it through the cable.
We use washing machines almost daily but wouldn't get the idea that it is an embedded system consisting of a Processor (and other hardware as well) and software.


Embedded System Example: Washing Machine
Input: Buttons
Output: Display, Motor
RAM
ROM
Control Unit: Processor,RAM,ROM With Software

It takes some inputs from the user like wash cycle, type of clothes, extra soaking and rinsing, spin rpm, etc., performs the necessary actions as per the instructions and finishes washing and drying the clothes. If no new instructions are given for the next wash, then the washing machines repeats the same set of tasks as the previous wash.

Embedded Systems can not only be stand-alone devices like Washing Machines but also be a part of a much larger system. An example for this is a Car. A modern day Car has several individual embedded systems that perform their specific tasks with the aim of making a smooth and safe journey.

Some of the embedded systems in a Car are Anti-lock Braking System (ABS), Temperature Monitoring System, Automatic Climate Control, Tire Pressure Monitoring System, Engine Oil Level Monitor, etc.

2.2.3. Programming Embedded Systems

As mentioned earlier, Embedded Systems consists of both Hardware and Software. If we consider a simple Embedded System, the main Hardware Module is the Processor. The Processor is the heart of the Embedded System and it can be anything like a Microprocessor, Microcontroller, DSP, CPLD (Complex Programmable Logic Device) or an FPGA (Field Programmable Gated Array).

All these devices have one thing in common: they are programmable i.e., we can write a program (which is the software part of the Embedded System) to define how the device actually works.

Embedded Software or Program allow Hardware to monitor external events (Inputs / Sensors) and control external devices (Outputs) accordingly. During this process, the program for an Embedded System may have to directly manipulate the internal architecture of the Embedded Hardware (usually the processor) such as Timers, Serial Communications Interface, Interrupt Handling, and I/O Ports etc.

From the above statement, it is clear that the Software part of an Embedded System is equally important as the Hardware part. There is no point in having advanced Hardware Components with poorly written programs (Software).

There are many programming languages that are used for Embedded Systems like Assembly (low-level Programming Language), C, C++, JAVA (high-level programming languages), Visual Basic, JAVA Script (Application level Programming Languages), etc.

In the process of making a better embedded system, the programming of the system plays a vital role and hence, the selection of the Programming Language is very important.

2.2.4. Factors for Selecting the Programming Language

The following are few factors that are to be considered while selecting the Programming Language for the development of Embedded Systems.

- **Size:** The memory that the program occupies is very important as Embedded Processors like Microcontrollers have a very limited amount of ROM (Program Memory).

- **Speed**: The programs must be very fast i.e., they must run as fast as possible. The hardware should not be slowed down due to a slow running software.
- **Portability:** The same program can be compiled for different processors.
- Ease of Implementation
- Ease of Maintenance
- Readability

Earlier Embedded Systems were developed mainly using Assembly Language. Even though Assembly Language is closest to the actual machine code instructions and produces small size hex files, the lack of portability and high amount of resources (time and man power) spent on developing the code, made the Assembly Language difficult to work with.

There are other high-level programming languages that offered the above mentioned features but none were close to C Programming Language. Some of the benefits of using Embedded C as the main Programming Language:

- Significantly easy to write code in C
- Consumes less time when compared to Assembly
- Maintenance of code (modifications and updates) is very simple
- Make use of library functions to reduce the complexity of the main code
- You can easily port the code to other architecture with very little modifications

2.2.5. Embedded System and Its Real Time Applications

- Embedded System and its Real Time Applications. Focuses on different topics like What is an Embedded System, What are the Real Time Applications of Embedded Systems, What is the Future of Embedded Systems, etc.
- The World is filled with Embedded Systems. The development of Microcontroller has paved path for several Embedded System application and they play a significant role (and will continue to play in the future as well) in our modern day life in one way or the other.
- Starting from consumer electronics like Digital Cameras, DVD Players to high end and advanced systems like Flight Controllers and Missile Guidance Systems, embedded systems are omnipresent and became an important part of our life.
- The way we live our life has been significantly improved with the utilization of Embedded Systems and they will continue to be an integral part of our lives even tomorrow.
- Another important concept we are hearing these days is Real – Time Systems. In a real time system, Real Time Computing takes place, where a computer (an Embedded System) must generate response to events within certain time limits.
- Before going in to the details of Real Time Applications of Embedded Systems, we will first see what an Embedded System is, what is a real time system and what is real time operating system.

2.2.6. *Components of Embedded System*

An Embedded System consists of four main components. They are the Processor (Microprocessor or Microcontroller), Memory (RAM and ROM), Peripherals (Input and Output) and Software (main program).

Processor: The heart of an Embedded System is the Processor. Based on the functionality of the system, the processor can be anything like a General Purpose Processor, a single purpose processor, an Application Specific Processor, a microcontroller or an FPGA.
Memory: Memory is another important part of an embedded system. It is divided in to RAM and ROM. Memory in an Embedded System (ROM to be specific) stores the main program and RAM stores the program variables and temporary data.

Peripherals: In order to communicate with the outside world or control the external devices, an Embedded System must have Input and Output Peripherals. Some of these peripherals include Input / Output Ports, Communication Interfaces, Timers and Counters, etc.

Software: All the hardware work according to the software (main program) written. Software part of an Embedded System includes initialization of the system, controlling inputs and outputs, error handling etc.

NOTE: Many Embedded Systems, usually small to medium scaled systems, generally consists of a Microcontroller as the main processor. With the help of a Microcontroller, the processor, memory and few peripherals will be integrated in to a single device.

2.2.7. Introduction to Embedded C Programming Language

Before going in to the details of Embedded C Programming Language and basics of Embedded C Program, we will first talk about the C Programming Language.

The C Programming Language became so popular that it is used in a wide range of applications ranging from Embedded Systems to Super Computers.

Embedded C Programming Language, which is widely used in the development of Embedded Systems, is an extension of C Program Language. The Embedded C Programming Language uses the same syntax and semantics of the C Programming Language like main function, declaration of datatypes, defining variables, loops, functions, statements, etc.

The extension in Embedded C from standard C Programming Language include I/O Hardware Addressing, fixed point arithmetic operations, accessing address spaces, etc.

2.2.7.1. Difference between C and Embedded C

There is actually not much difference between C and Embedded C apart from few extensions and the operating environment. Both C and Embedded C programming are ISO Standards that have almost same syntax, datatypes, functions, etc.

Embedded C is basically an extension to the Standard C Programming Language with additional features like Addressing I/O, multiple memory addressing and fixed-point arithmetic, etc.

C Programming Language is generally used for developing desktop applications, whereas Embedded C is used in the development of Microcontroller based applications.

2.2.7.2. Keywords in Embedded C

A Keyword is a special word with a special meaning to the compiler (a C Compiler for example, is a software that is used to convert program written in C to Machine Code). For example, if we take the Keil's Cx51 Compiler (a popular C Compiler for 8051 based Microcontrollers) the following are some of the keywords:

- bit
- sbit
- sfr
- small
- large

The following table lists out all the keywords associated with the Cx51 C Compiler.

_at_	alien	bdata
bit	code	compact
data	far	idata
interrupt	large	pdata
priority	reentrant	sbit
sfr	sfr16	small
task	using	xdata

Data Types in Embedded C

Data Types in C Programming Language (or any programming language for that matter) help us declaring variables in the program. There are many data types in C Programming Language like signed int, unsigned int, signed char, unsigned char, float, double, etc. In addition to these there few more data types in Embedded C.

The following are the extra data types in Embedded C associated with the Keil's Cx51 Compiler.

- bit
- sbit
- sfr
- sfr16

The following table shows some of the data types in Cx51 Compiler along with their ranges.

Data Type	**Bits (Bytes)**	**Range**
bit	1	0 or 1 (bit addressable part of RAM)
signed int	16 (2)	-32768 to +32767
unsigned int	16 (2)	0 to 65535
signed char	8 (1)	-128 to +127
unsigned	8 (1)	0 to 255
float	32 (4)	±1.175494E-38 to ±3.402823E+38
double	32 (4)	±1.175494E-38 to ±3.402823E+38
sbit	1	0 or 1 (bit addressable part of RAM)
sfr	8 (1)	RAM Addresses (80h to FFh)
sfr16	16 (2)	0 to 65535

Basic Structure of an Embedded C Program (Template for Embedded C Program)

The next thing to understand in the Basics of Embedded C Program is the basic structure or Template of Embedded C Program. This will help us in understanding how an Embedded C Program is written.

The following part shows the basic structure of an Embedded C Program.

- Multiline Comments.......... Denoted using /*……*/
- Single Line Comments ....Denoted using //
- Preprocessor Directives ....#include<…> or #define
- Global Variables.......... Accessible anywhere in the program
- Function Declarations.......... Declaring Function
- Main Function..........Main Function, execution begins here

  {

  Local Variables ....Variables confined to main function

  Function Calls ....Calling other Functions

```
    Infinite Loop ..........Like while(1) or for(;;)
    Statements . . . . .
    ….
    ….
    }
o   Function Definitions.......... Defining the Functions
    {
    Local Variables ..........Local Variables confined to this Function
    Statements . . . . .
    ….
    ….
    }
```

Before seeing an example with respect to 8051 Microcontroller, we will first see the different components in the above structure.

2.2.7.3. Different Components of an Embedded C Program

Comments: Comments are readable text that are written to help us (the reader) understand the code easily. They are ignored by the compiler and do not take up any memory in the final code (after compilation).

There are two ways you can write comments: one is the single line comments denoted by // and the other is multiline comments denoted by /*….*/.

Preprocessor Directive: A Preprocessor Directive in Embedded C is an indication to the compiler that it must look in to this file for symbols that are not defined in the program.

Global Variables: Global Variables, as the name suggests, are Global to the program i.e., they can be accessed anywhere in the program.

Local Variables: Local Variables, in contrast to Global Variables, are confined to their respective function.

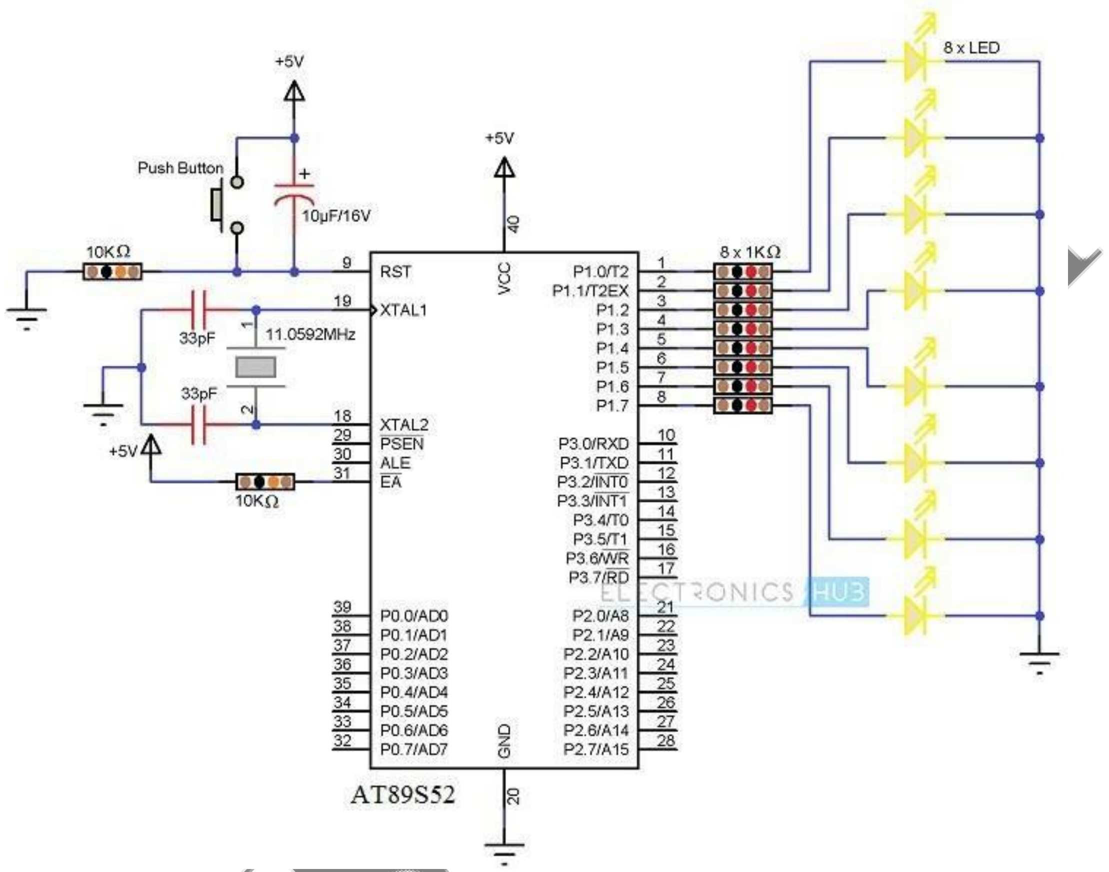
Main Function: Every C or Embedded C Program has one main function, from where the execution of the program begins.

Basic Embedded C Program

Till now, we have seen a few Basics of Embedded C Program like difference between C and Embedded C, basic structure or template of an Embedded C Program and different components of the Embedded C Program.

Example of Embedded C Program

The following image shows the circuit diagram for the example circuit. It contains an 8051 based Microcontroller (AT89S52) along with its basic components (like RESET Circuit, Oscillator Circuit, etc.) and components for blinking LEDs (LEDs and Resistors).



In order to write the Embedded C Program for the above circuit, we will use the Keil C Compiler. This compiler is a part of the Keil µVision IDE. The program is shown below.

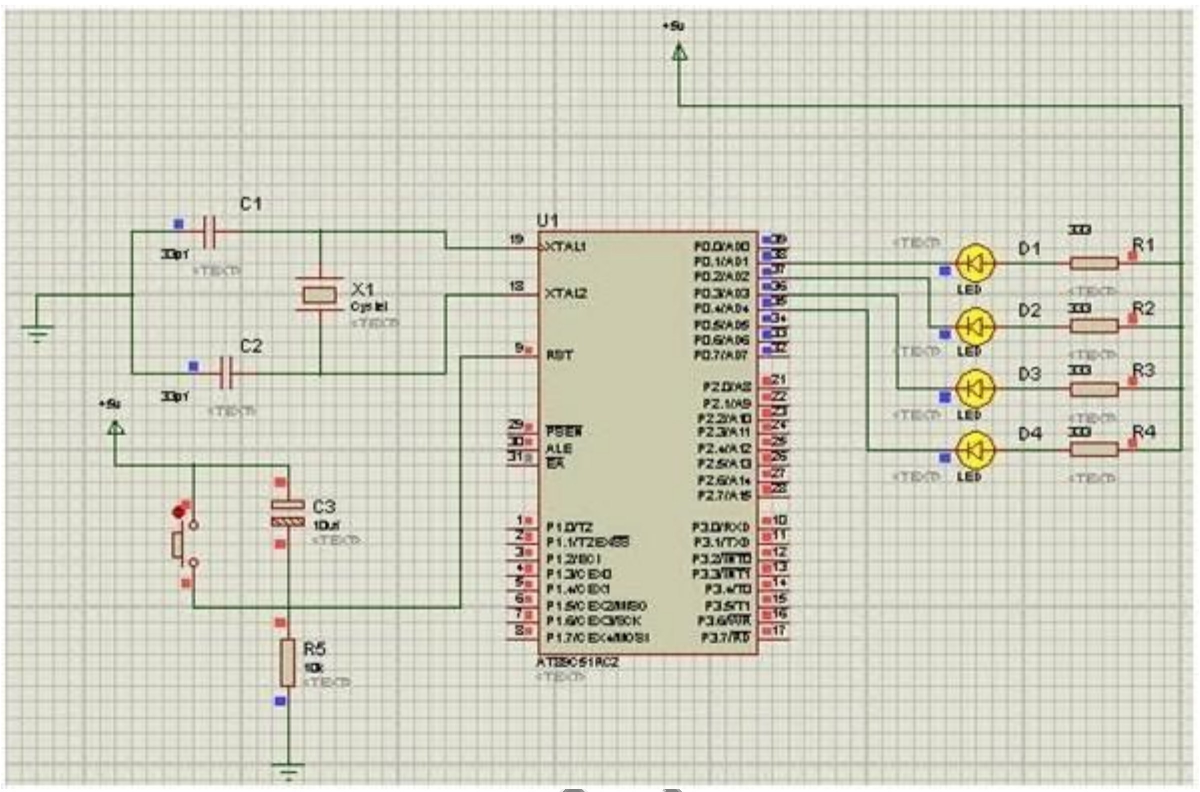
```
#include<reg51.h> // Preprocessor Directive
void delay (int); // Delay Function Declaration
void main(void) // Main Function
{
P1 = 0x00;
/* Making PORT1 pins LOW. All the LEDs are OFF.
 * (P1 is PORT1, as defined in reg51.h) */
        while(1) // infinite loop
        {

                P1 = 0xFF; // Making PORT1 Pins HIGH i.e. LEDs are ON.
                delay(1000);
                /* Calling Delay function with Function parameter as 1000.
                 * This will cause a delay of 1000mS i.e. 1 second */
                P1 = 0x00; // Making PORT1 Pins LOW i.e. LEDs are OFF.
                delay(1000);
        }
}
        void delay (int d) // Delay Function Definition
        {

                unsigned int i=0; // Local Variable. Accessible only in this function.
                 /* This following step is responsible for causing delay of 1000mS
                 * (or as per the value entered while calling the delay function) */
                for(; d>0; d–)
                {
                for(i=250; i>0; i – -);
                for(i=248; i>0; i – -);
                }
        }
```

2.2.7.4. LED Blinking using 8051 Microcontroller

LED is a semiconductor device used in many electronic devices, mostly used for indication purposes. It is used widely as indicator during test for checking the validity of results at different stages.

It is very cheap and easily available in variety of shape, color and size. The LEDs are also used in designing of message display boards and traffic control signal lights etc.

Consider the Proteus Software based simulation of LED blinking using 8051 Microcontroller is shown below:-



In above Proteus based simulation the LEDs are interfaced to the PORT0 of the 8051 microcontroller.

Let's see the Embedded C Program for generating the LED output sequence as shown below:

00000001
00000010
00000100.....
.... And so on up to 10000000.

```
#include<reg51.h>
void main()
{
unsigned int k;
 unsigned char l,b;
 while(1)
 {
        P0=0x01;
        b=P0;
        for(l-0;l<3000;l++);
        for(k=0;k<8;k++)
        {
                b=b<<1;
```

```
                        P0=b;
                }
        }
       }
```

Consider the Embedded C Program for generating the LED output sequence as shown below is:-

00000001
00000011
00000111.....
.... And so on up to 11111111.

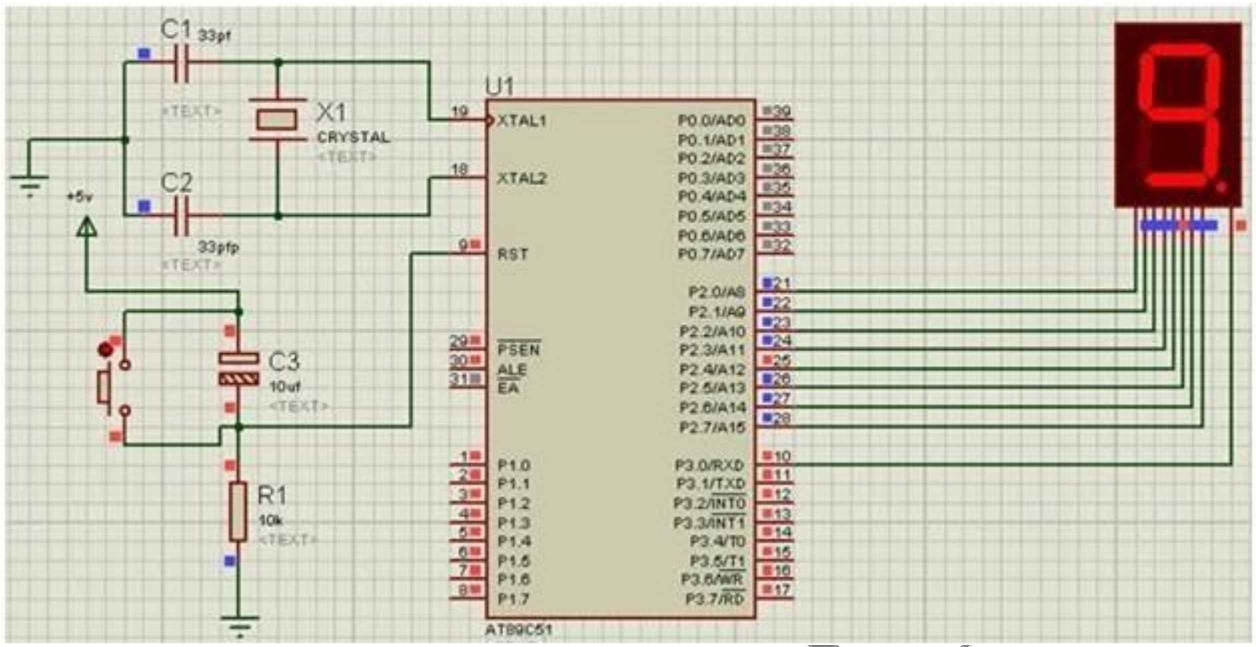
```
#include<reg51.h>
void main()
{
        unsigned int i;
        unsigned char j,b;
        while(1)
        {
                P0=0x01;
                b=P0;
                for(j-0;j<3000;j++);
                for(j=0;j<8;j++)
                {
                        bb=b<<1;
                        b=0x01;
                        P0=b;
                }
        }
}
```

Displaying Number on 7-Segment Display using 8051 Microcontroller

Electronic display used for displaying alphanumeric character is known as 7-Segment display it is used in many systems for displaying the information.

It is constructed using eight LEDs which are connected in sequential way so as to display digits from 0 to 9, when certain combinations of LEDs are switched on. It displays only one digit at a time.

Consider the Proteus software based simulation of displaying number on 7-segment display using 8051 microcontroller is:-



Consider the program for displaying the number from '0 to F' on 7-segment display is:- 10s

```
#include<reg51.h>
sbit a= P3^0;
sbit x= P3^1;
sbit y= P3^2;
sbit z= P3^3;
void main()
{
        unsigned char m[10]={0?40,0xF9,0?24,0?30,0?19,0?12,0?02,0xF8,0xE00,0?10};
        unsigned int i,j;
        a=x=y=z=1;
        while(1)
        {
                for(i=0;i<10;i++)
                {
                        P2=m[i];
                        for(j=0;j<60000;j++);
                }
        }
}
```

Consider the program for displaying numbers from '00 to 10' on a 7segment display is:-

```
#include<reg51.h>
sbit x= P3^0;
sbit y= P3^1;
void display1();
void display2();
void delay();
void main()
{
        unsigned char m[10]={0?40,0xF9,0?24,0?30,0?19,0?12,0?02,0xF8,0xE00,0?10};
        unsigned int i,j;
        ds1=ds2=0;
        while(1)
        {
                for(i=0,i<20;i++)
                display1();
                display2();
        }
}
void display1()
{
        x=1;
        y=0;
        P2=m[ds1];
        delay();
        x=1;
        y=0;
        P2=m[ds1];
        delay();
}
void display2()
{
        ds1++;
        if(ds1>=10)
        {
                ds1=0;
                ds2++;
                if(ds2>=10)
                {
                        ds1=ds2=0;
                }
        }
}
void delay()
{
        unsigned int k;
        for(k=0;k<30000;k++);
}
```

2.3. RTOS (Real-Time Operating Systems for Embedded Developers)

Embedded developers are often accustomed to bare metal programming or have reservations towards using an RTOS. Here’s what they are, and why you should consider using one. Today’s product development cycles are becoming increasingly complex. With available development time shrinking yet the required feature set expanding, busy developers need to find ways of doing more in less time. It can often make sense to use a real-time operating system (RTOS) to gain efficiencies in task management and resource sharing.

2.3.1. What is an RTOS?

Simply put, an RTOS is a piece of software designed to efficiently manage the time of a central processing unit (CPU). This is especially relevant for embedded systems when time is critical.

The key difference between an operating system such as Windows and an RTOS often found in embedded systems is the response time to external events. An ordinary OS provides a non-deterministic response to events with no guarantee with respect to when they will be processed, albeit while trying to stay responsive. The user perceiving the OS to be responsive is more important than handling underlying tasks. On the other hand, an RTOS' goal is fast and more deterministic reaction.

Developers used to OS’s such as Windows or Linux will be quite familiar with the characteristics of an embedded RTOS. They are designed to run in systems with limited memory, and to operate indefinitely without the need to be reset.

Because an RTOS is designed to respond to events quickly and perform under heavy loads, it can be slower at big tasks when compared to another OS.

2.3.2. RTOS scheduling

An RTOS is valued for how quickly it can respond and in that, the advanced scheduling algorithm is the key component.

The time-criticality of embedded systems vary from soft-real time washing machine control systems through hard-real time aircraft safety systems. In situations like the latter, the fundamental demand to meet real-time requirements can only be made if the OS scheduler’s behavior can be accurately predicted.

Many operating systems give the impression of executing multiple programs at once, but this multi-tasking is something of an illusion. A single processor core can only run a single thread of execution at any one time. An operating system’s scheduler decides which program, or thread, to run when. By rapidly switching between threads, it provides the illusion of simultaneous multitasking.

The flexibility of an RTOS scheduler enables a broad approach to process priorities, although an RTOS is more commonly focused on a very narrow set of applications. An RTOS scheduler should give minimal interrupt latency and minimal thread switching overhead. This is what makes an RTOS so relevant for time-critical embedded systems.

2.3.3. The use of RTOS in embedded designs

Many embedded programmers shy away from using an RTOS because they suspect that it adds too much complexity to their application, or it is simply unknown territory. An RTOS typically requires anything up to 5% of the CPU's resources to perform its duties. While there will always be some resource penalties, an RTOS can make up for it in areas such as simplified determinism, ease of use though HW abstraction, reduced development time and easier debugging.

Using an RTOS means you can run multiple tasks concurrently, bringing in the basic connectivity, privacy, security, and so on as and when you need them. An RTOS allows you to create an optimized solution for the specific requirements of your project.

2.3.4. Introducing the Zephyr RTOS

There are numerous RTOS solutions out there. Many developers in the Nordic world are focused on low power embedded systems. If that's you, we'd suggest you check out Zephyr, which is well-suited for connectivity solutions in which ultra-low power is a requirement.
The modular Zephyr RTOS supports multiple architectures, so developers are able to easily tailor a solution to meet their needs.

2.3.5. Real-time operating system (RTOS): Components, Types, Examples

Real-time operating system (RTOS) is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay. The full form of RTOS is Real time operating system.

In a RTOS, Processing time requirement are calculated in tenths of seconds increments of time. It is time-bound system that can be defined as fixed time constraints. In this type of system, processing must be done inside the specified constraints. Otherwise, the system will fail.

2.3.5. Why use an RTOS?

Here are important reasons for using RTOS:

- It offers priority-based scheduling, which allows you to separate analytical processing from non-critical processing.
- The Real time OS provides API functions that allow cleaner and smaller application code.

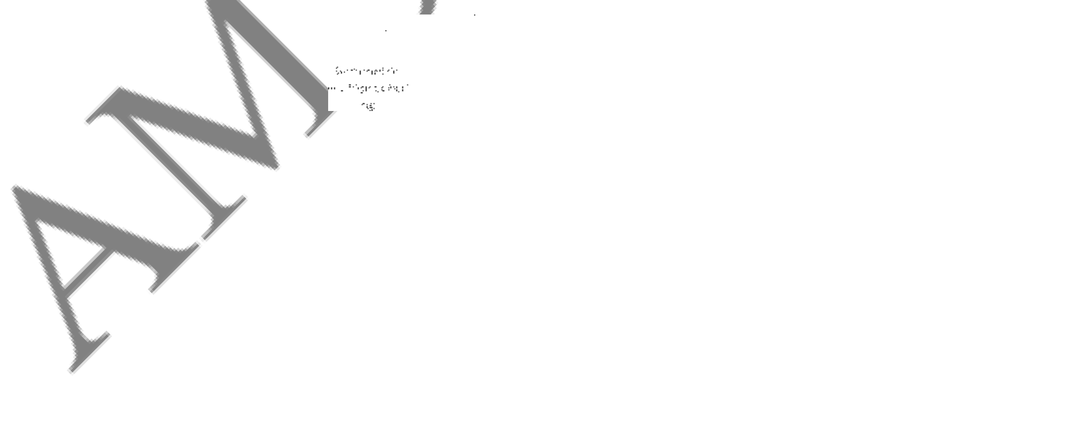
- Abstracting timing dependencies and the task-based design results in fewer interdependencies between modules.
- RTOS offers modular task-based development, which allows modular task-based testing.
- The task-based API encourages modular development as a task, will typically have a clearly defined role. It allows designers/teams to work independently on their parts of the project.
- An RTOS is event-driven with no time wastage on processing time for the event which is not occur

2.3.6. Terms used in RTOS

Here, are essential terms used in RTOS:

- **Task –** A set of related tasks that are jointly able to provide some system functionality.
- **Job –** A job is a small piece of work that can be assigned to a processor, and that may or may not require resources.
- **Release time of a job –** It's a time of a job at which job becomes ready for execution.
- **Execution time of a job:** It is time taken by job to finish its execution.
- **Deadline of a job:** It's time by which a job should finish its execution.
- **Processors:** They are also known as active resources. They are important for the execution of a job.
- **Maximum It is the** allowable response time of a job is called its relative deadline.
- **Response time of a job:** It is a length of time from the release time of a job when the instant finishes.
- **Absolute deadline:** This is the relative deadline, which also includes its release time.

2.3.7. Components of RTOS



Components of Real Time Operating System

Here, are important Component of RTOS

The Scheduler: This component of RTOS tells that in which order, the tasks can be executed which is generally based on the priority.

Symmetric Multiprocessing (SMP): It is a number of multiple different tasks that can be handled by the RTOS so that parallel processing can be done.

Function Library: It is an important element of RTOS that acts as an interface that helps you to connect kernel and application code. This application allows you to send the requests to the Kernel using a function library so that the application can give the desired results.

Memory Management: this element is needed in the system to allocate memory to every program, which is the most important element of the RTOS.

Fast dispatch latency: It is an interval between the termination of the task that can be identified by the OS and the actual time taken by the thread, which is in the ready queue, that has started processing.
User-defined data objects and classes: RTOS system makes use of programming languages like C or C++, which should be organized according to their operation.

2.3.8. Features of RTOS

Here are important features of RTOS:

- Occupy very less memory
- Consume fewer resources
- Response times are highly predictable
- Unpredictable environment
- The Kernel saves the state of the interrupted task ad then determines which task it should run next.
- The Kernel restores the state of the task and passes control of the CPU for that task

Factors for selecting an RTOS

Here, are essential factors that you need to consider for selecting RTOS:

- **Performance**: Performance is the most important factor required to be considered while selecting for a RTOS**.**
- **Middleware**: if there is no middleware support in Real time operating system, then the issue of time-taken integration of processes occurs.
- **Error-free**: RTOS systems are error-free. Therefore, there is no chance of getting an error while performing the task.
- **Embedded system usage**: Programs of RTOS are of small size. So we widely use RTOS for embedded systems.
- **Maximum Consumption**: we can achieve maximum Consumption with the help of RTOS.

- **Task shifting**: Shifting time of the tasks is very less.
- **Unique features**: A good RTS should be capable, and it has some extra features like how it operates to execute a command, efficient protection of the memory of the system, etc.
- **24/7 performance**: RTOS is ideal for those applications which require to run 24/7.

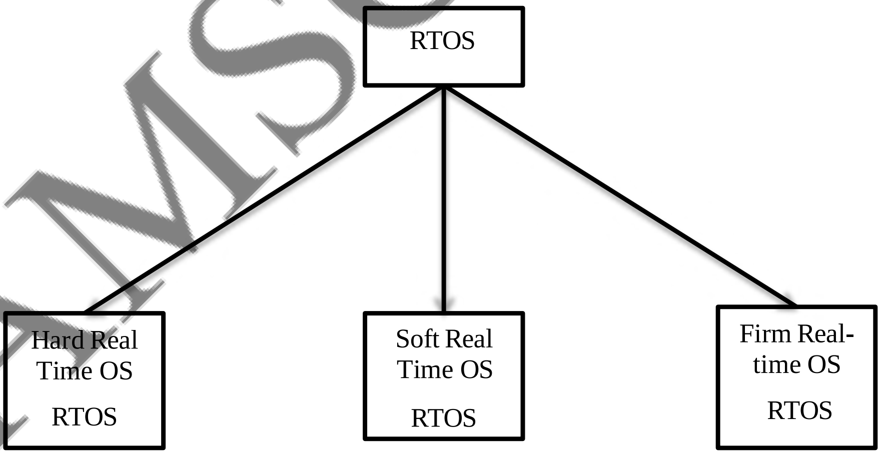
2.3.9. Difference between in GPOS and RTOS

Here are important differences between GPOS and RTOS:

General-Purpose Operating System (GPOS)	**Real-Time Operating System (RTOS)**
It used for desktop PC and laptop.	It is only applied to the embedded application.
Process-based Scheduling.	Time-based scheduling used like round-robin scheduling.
Interrupt latency is not considered as important as in RTOS.	Interrupt lag is minimal, which is measured in a few microseconds.
No priority inversion mechanism is present in the system.	The priority inversion mechanism is current. So it can not modify by the system.
Kernel's operation may or may not be preempted.	Kernel's operation can be preempted.
Priority inversion remain unnoticed	No predictability guarantees

2.3.10. Types of Real Time Operating System (RTOS)

The real-time operating systems can be of 3 types –



1. **Hard Real-Time operating system:**
   These operating systems guarantee that critical tasks be completed within a range of time. For example, a robot is hired to weld a car body. If the robot welds too early or too late, the car cannot be sold, so it is a hard real-time system that requires complete car welding by

robot hardly on the time., scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

2. **Soft real-time operating system:**
   This operating system provides some relaxation in the time limit.
   For example – Multimedia systems, digital audio systems etc. Explicit, programmer-defined and controlled processes are encountered in real-time systems. A separate process is changed with handling a single external event. The process is activated upon occurrence of the related event signalled by an interrupt.

   Multitasking operation is accomplished by scheduling processes for execution independently of each other. Each process is assigned a certain level of priority that corresponds to the relative importance of the event that it services. The processor is allocated to the highest priority processes. This type of schedule, called, priority-based preemptive scheduling is used by real-time systems.

3. **Firm Real-time Operating System**:
   RTOS of this type have to follow deadlines as well. In spite of its small impact, missing a deadline can have unintended consequences, including a reduction in the quality of the product. Example: Multimedia applications.

2.3.11. Advantages of Real Time Operating System (RTOS)

The advantages of real-time operating systems are as follows-

1. **Maximum consumption**
   Maximum utilization of devices and systems. Thus more output from all the resources.

2. **Task Shifting**
   Time assigned for shifting tasks in these systems is very less. For example, in older systems, it takes about 10 microseconds. Shifting one task to another and in the latest systems, it takes 3 microseconds.

3. **Focus On Application –**
   Focus on running applications and less importance to applications that are in the queue.

4. **Real-Time Operating System In Embedded System –**
   Since the size of programs is small, RTOS can also be embedded systems like in transport and others.

5. **Error Free –**
   These types of systems are error-free.

6. **Memory Allocation –**
   Memory allocation is best managed in these types of systems.

2.3.12. Disadvantages of Real Time Operating System (RTOS)

The disadvantages of real-time operating systems are as follows-

1. **Limited Tasks –**
   Very few tasks run simultaneously, and their concentration is very less on few applications to avoid errors.

2. **Use Heavy System Resources –**
   Sometimes the system resources are not so good and they are expensive as well.

3. **Complex Algorithms –**
   The algorithms are very complex and difficult for the designer to write on.

4. **Device Driver And Interrupt signals –**
   It needs specific device drivers and interrupts signals to respond earliest to interrupts.

5. **Thread Priority –**
   It is not good to set thread priority as these systems are very less prone to switching tasks.

6. **Minimum Switching –** RTOS performs minimal task switching.

2.3.13. Comparison of Regular and Real-Time operating systems:

Regular OS	**Real-Time OS (RTOS)**
Complex	Simple
Best effort	Guaranteed response
Fairness	Strict Timing constraints
Average Bandwidth	Minimum and maximum limits
Unknown components	Components are known
Unpredictable behavior	Predictable behavior
Plug and play	RTOS is upgradeable

2.4. Multiple Tasks And Multiple Processes

2.4.1. Tasks and Processes

Many (if not most) embedded computing systems do more than one thing that is, the environment can cause mode changes that in turn cause the embedded system to behave quite differently. For example, when designing a telephone answering machine,

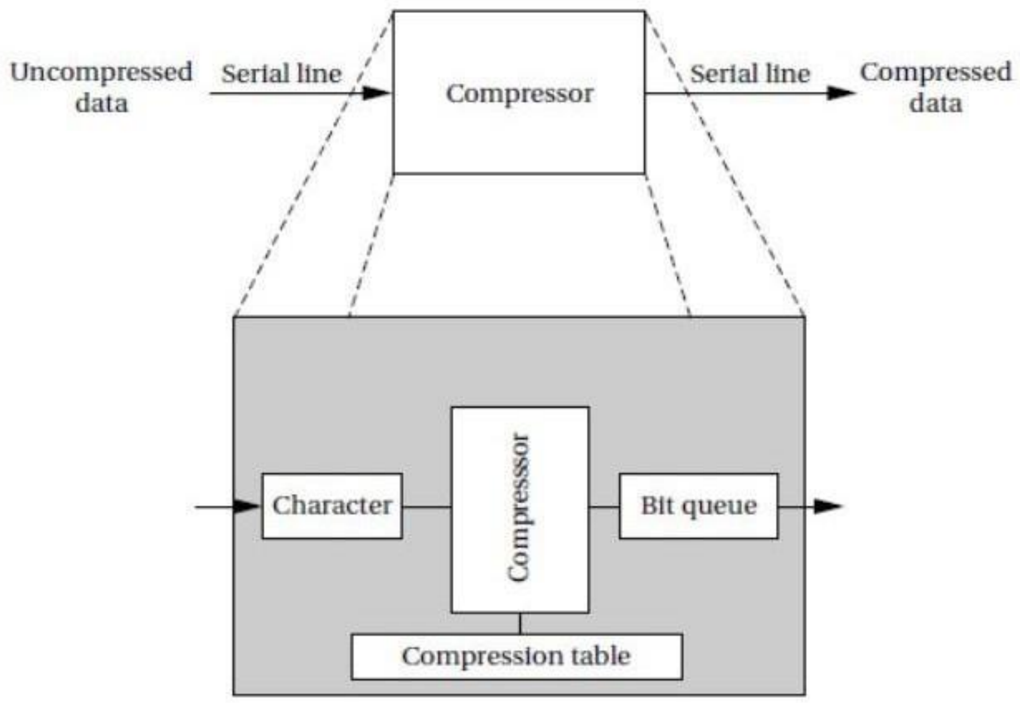
We can define recording a phone call and operating the user's control panel as distinct tasks, because they perform logically distinct operations and they must be performed at very different rates. These different **tasks** are part of the system's functionality, but that application-level organization of functionality is often reflected in the structure of the program as well.

A ***process*** is a single execution of a program. If we run the same program two different times, we have created two different processes. Each process has its own state that includes not only its registers but all of its memory. In some OSs, the memory management unit is used to keep each process in a separate address space. In others, particularly lightweight RTOSs, the processes run in the same address space. Processes that share the same address space are often called ***threads***. As shown in Figure, this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a predefined compression table. Such a box may be used, for example, to compress data being sent to a modem.

The program's need to receive and send data at different rates for example, the program may emit 2 bits for the first byte and then 7 bits for the second byte will obviously find itself reflected in the structure of the code. It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of output bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets.

But beyond the need to create a clean data structure that simplifies the control structure of the code, we must also ensure that we process the inputs and outputs at the proper rates. For example, if we spend too much time in packaging and emitting output characters, we may drop an input character. Solving timing problems is a more challenging problem.

The text compression box provides a simple example of rate control problems. A control panel on a machine provides an example of a different type of rate control problem, the ***asynchronous input***.



The control panel of the compression box may, for example, include a compression mode button that disables or enables compression, so that the input text is passed through unchanged when compression is disabled. We certainly do not know when the user will push the compression mode button the button may be depressed asynchronously relative to the arrival of characters for compression.

2.4.2. Multirate Systems

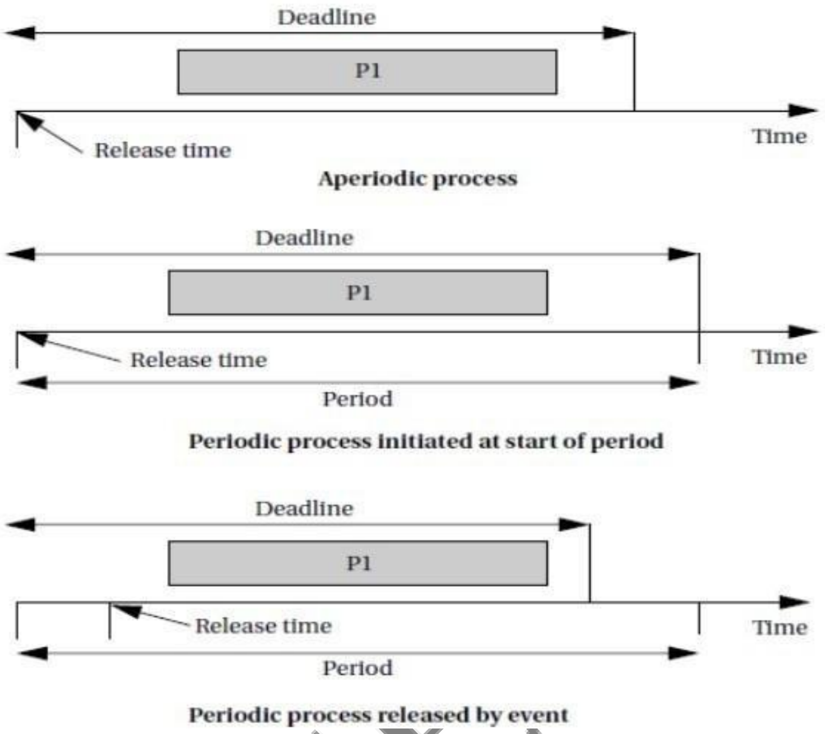
Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled. ***Multirate*** embedded computing systems are very common, including automobile engines, printers, and cell phones. In all these systems, certain operations must be executed periodically, and each operation is executed at its own rate.

2.4.3. Timing Requirements on Processes

Processes can have several different types of timing requirements imposed on them by the application. The timing requirements on a set of processes strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling proper, we outline the types of process timing requirements that are useful in embedded system design.

Figure illustrates different ways in which we can define two important requirements on processes: ***release time*** and ***deadline***.

The release time is the time at which the process becomes ready to execute; this is not necessarily the time at which it actually takes control of the CPU and starts to run. An aperiodic process is by definition initiated by an event, such as external data arriving or data computed by another process.



The release time is generally measured from that event, although the system may want to make the process ready at some interval after the event itself. For a periodically executed process, there are two common possibilities.

In simpler systems, the process may become ready at the beginning of the period. More sophisticated systems, such as those with data dependencies between processes, may set the release time at the arrival time of certain data, at a time after the start of the period.

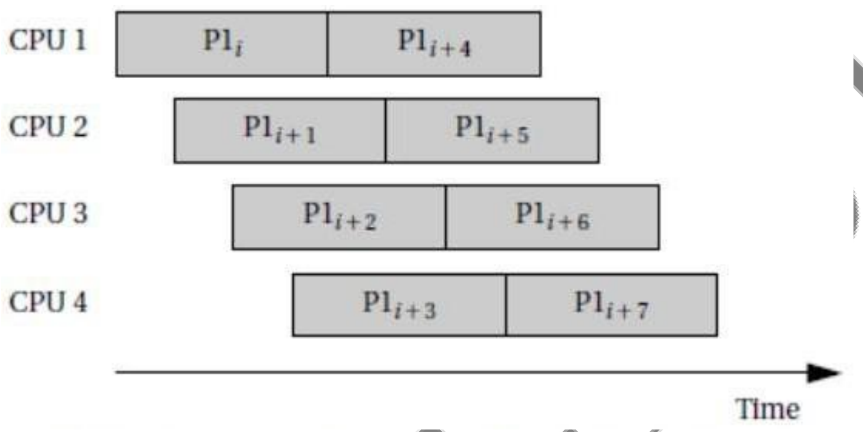
A deadline specifies when a computation must be finished. The deadline for an aperiodic process is generally measured from the release time, since that is the only reasonable time reference. The deadline for a periodic process may in general occur at some time other than the end of the period.

Rate requirements are also fairly common. A rate requirement specifies how quickly processes must be initiated.

The ***period*** of a process is the time between successive executions. For example, the period of a digital filter is defined by the time interval between successive input samples.

The process's ***rate*** is the inverse of its period. In a multirate system, each process executes at its own distinct rate.

The most common case for periodic processes is for the initiation interval to be equal to the period. However, pipelined execution of processes allows the initiation interval to be less than the period. Figure illustrates process execution in a system with four CPUs.



CPU Metrics

We also need some terminology to describe how the process actually executes. The ***initiation time*** is the time at which a process actually starts executing on the CPU. The ***completion time*** is the time at which the process finishes its work.

The most basic measure of work is the amount of ***CPU time*** expended by a process. The CPU time of process *i* is called *Ci* . Note that the CPU time is not equal to the completion time minus initiation time; several other processes may interrupt execution. The total CPU time consumed by a set of processes is

$$T = \sum Ti$$

We need a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is ***utilization***:

$$U = \text{CPU time for useful work/total available CPU time}$$

Utilization is the ratio of the CPU time that is being used for useful computations to the total available CPU time. This ratio ranges between 0 and 1, with 1 meaning that all of the available CPU time is being used for system purposes. The utilization is often expressed as a percentage. If we measure the total execution time of all processes over an interval of time *t*, then the CPU utilization is

$$U = T/t.$$

2.4.4. Multiple Tasks and Multiple Processes

`Most embedded systems require functionality and timing that is too complex to body in a single program. We break the system in to multiple tasks in order to manage when things happen. In this section we will develop the basic abstractions that will be manipulated by the RTOS to build multirate systems.

To understand why these parathion of an application in to tasks may be reflected in the program structure, consider how we would build a stand-alone compression unit based on the compression algorithm,this device is connected to serial ports on both ends. The input to the box is an uncompressed stream of bytes. The box emits a compressed string of bits on the output serial line, based on a predefined compression table. Such a box may be used, for example, to compress data being sent to a modem. The program's need to receive and send data at different rates—for example, the program may emit 2 bits for the first byte and then 7 bits for the second byte— will obviously find itself reflected in the structure of the code.

It is easy to create irregular, ungainly code to solve this problem; a more elegant solution is to create a queue of out put bits, with those bits being removed from the queue and sent to the serial port in 8-bit sets. But beyond the need to create a clean data structure that simplifies the control structure of the code, we must also ensure that we process the inputs and outputs at the proper rates. For example, if we spend too much time in packaging and emitting output characters, we may drop an input character. Solving timing problems is a more challenging problem.

The text compression box provides a simple example of rate control problems. A control panel on a machine provides an example of a different type of rate con- troll problem, the asynchronous input. The control panel of the compression box may, for example, include a compression mode button that disables or enables com- pression, so that the input text is passed through unchanged when compression is disabled. We certainly do not know when the user will push the compression modebutton— the button may be depressed a synchronously relative to the arrival of characters for compression. We do know, however, that the button will be depressed at a much lower rate than characters will be received, since it is not physically possible for a person to repeatedly depress a button at even slow serial line rates. Keeping up with the input and output data while checking on the button can introduce some very complex control code in to the program. Sampling the button's state too slowly can cause the machine to miss a button depression entirely, but sampling it too frequently and duplicating a data value can cause the machine to in correctly compress data.

One solution is to introduce a counter in to the main compression loop, so that a subroutine to check the input button is called once every times the compression loop is executed. But this solution does not work when either the compression loop or the button-handling routine has highly variable execution times—if the execution time of either varies significantly, it will cause the other to execute later than expected, possibly causing data to be lost. We need to be able to keep track of these two different tasks separately, applying different timing requirements to each.

This is the sort of control that processes allow. The above two examples illustrate how requirements on timing and execution rate can create major problems in programming. When code is written to satisfy several different timing requirements at once, the control structures necessary to get any sort of solution become very complex very quickly. Worse, such complex control is usually quite difficult to verify for either functional or timing properties.

2.4.5. Multi rate Systems

Implementing code that satisfies timing requirements is even more complex when multiple rates of computation must be handled. Multi rate embedded computing systems are very common, including auto mobile engines, printers, and cell phones.In all these systems, certain operations must be executed periodically, and each oper-ation is executed at its own rate. Application Example6.1 describes why auto mobile engines require multi rate control.

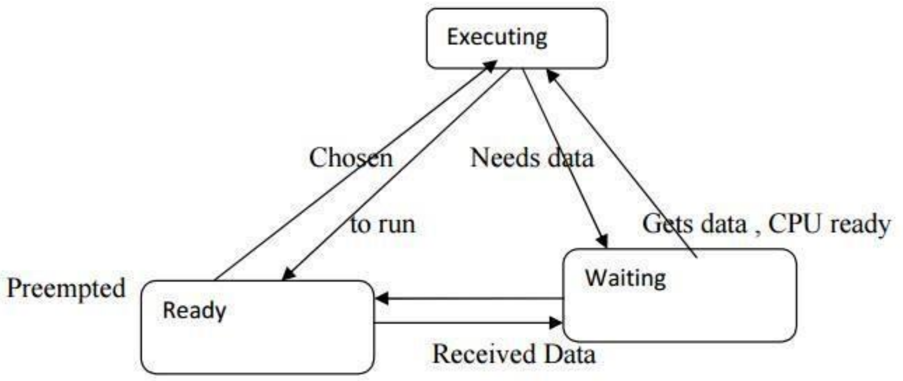
2.4.6. Timing Requirements on Processes

Processes can have several different types of timing requirements imposed on them by the application. The timing requirements on a set of process strongly influence the type of scheduling that is appropriate. A scheduling policy must define the timing requirements that it uses to determine whether a schedule is valid. Before studying scheduling proper, we outline the types of process timing requirements that are useful in embedded system design. Figure illustrates different ways in which we can define two important requirements on processes: eg. What happens when a process misses a deadline? The practical effects of a timing violation depend on the application—the results can be catastrophic in an auto mo- tive control system, where as a missed deadline in a multimedia system may cause an audio or video glitch. The system can be designed to take a variety of actions when a deadline is missed. Safety-critical systems may try to take compensatory measures such as approximating data or switching in to a special safety mode. Systems for which safety is not as important may take simple measures to avoid propagating bad data, such as inserting silence in a phone line, or may completely ignore the failure.Even if the modules are functionally correct, their timing improper behavior can introduce major execution errors. Application Example6.2 describes a timing problem in space shuttle software that caused the delay of the first launch of the shuttle. We need a basic measure of the efficiency with which we use the CPU. The simplest and most direct measure is utilization:

Utilization is the ratio of the CPU time that is being used for useful computations to the total available CPUtime. This ratio ranges between 0 and 1, with 1 meaning that all of the available CPU time is being used for system purposes. The utilization is often expressed as a percentage. If we measure the total execution time of all processes over an interval of time t, then the CPU utilization is U/T

2.4.7. Process State and Scheduling

The first job of the OS is to determine that process runs next. The work of choosing the order of running processes is known as scheduling. The OS considers a process to be in one of three basic scheduling states



Schedulability means whether there exists a schedule of execution for the processes in a system that satisfies all their timing requirements. In general, we must construct a schedule to show schedulability, but in some cases we can eliminate some sets of processes as unschedulable using some very simple tests. Utilization is one of the key metrics in evaluating a scheduling policy. Our most basic require- ment is that CPU utilization be no more than 100% since we can't use the CPU more than100% of the time.

When we evaluate the utilization of the CPU, we generally do so over a finite period that covers all possible combinations of process executions. For periodic processes, the length of time that must be considered is the hyper period, which is the least-common multiple of the periods of all the processes. If we evaluate the hyper period, we are sure to have considered all possible combinations of the periodic processes.

2.4.8. Running Periodic Processes

We need to find a programming technique that allows us to run periodic processes, ideally at different rates. For the moment, let's think of a process as a subroutine; we will call the mp1(), p2(), etc. for simplicity. Our goal is to run these subroutines at rates determined by the system

designer. Here is a very simple program that runs our process subroutines repeatedly: A timer is a much more reliable way to control execution of the loop. We would probably use the timer to generate periodic interrupts. Let's assume for the moment that the pall() function is called by the timer's interrupt handler. Then this code will execute each process once after a timer interrupt:

```
voidpall()
{
          p1();
          p2();
 }
```

But what happens when a process runs too long? The timer's interrupt will cause the CPU's interrupt system to mask its interrupts, so the interrupt will not occur until after the pall() routine returns. As a result, the next iteration will start late. This is a serious problem, but we will have to wait for further refinements before we can fix it.

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

```
voidpA()
{
         /*processesthatrunatrateA*/
          p1();
          p3();
}
void pB()
{
 /*processesthatrunatrateB*/
 }
```

This solution allows us to execute processes at rates that are simple multiples of each other. However, when the rates are n't related by a simple ratio, the counting process becomes more complex and more likely to contain bugs. We have developed somewhat more reliable code, but this programming style is still limited in capability and prone to bugs. To improve both the capabilities and reliability of our systems, we need to invent the RTOS.

2.5. Context Switch in Operating System

An operating system is a program loaded into a system or computer. and manage all the other program which is running on that OS Program, it manages the all other application programs. or in other words, we can say that the OS is an interface between the user and computer hardware.

we will learn about what is Context switching in an Operating System and see how it works also understands the triggers of context switching and an overview of the Operating System.

Context Switch
Context switching in an operating system involves saving the context or state of a running process so that it can be restored later, and then loading the context or state of another. process and run it.

Context Switching refers to the process/method used by the system to change the process from one state to another using the CPUs present in the system to perform its job.

Example – Suppose in the OS there (N) numbers of processes are stored in a Process Control Block(PCB). like The process is running using the CPU to do its job. While a process is running, other processes with the highest priority queue up to use the CPU to complete their job.

2.5.1. The Need for Context Switching
Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.

The operating system's need for context switching is explained by the reasons listed below.
1. One process does not directly switch to another within the system. Context switching makes it easier for the operating system to use the CPU's resources to carry out its tasks and store its context while switching between multiple processes.
2. Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.
3. Context switching enables all processes to share a single CPU to finish their execution and store the status of the system's tasks. The execution of the process begins at the same place where there is a conflict when the process is reloaded into the system.
4. Context switching only allows a single CPU to handle multiple processes requests parallelly without the need for any additional processors.

2.5.2. Context Changes as a Trigger
The three different categories of context-switching triggers are as follows.
1. Interrupts
2. Multitasking
3. User/Kernel switch

Interrupts: When a CPU requests that data be read from a disc, if any interruptions occur, context switching automatically switches to a component of the hardware that can handle the interruptions more quickly.

Multitasking: The ability for a process to be switched from the CPU so that another process can run is known as context switching. When a process is switched, the previous state is retained so that the process can continue running at the same spot in the system.

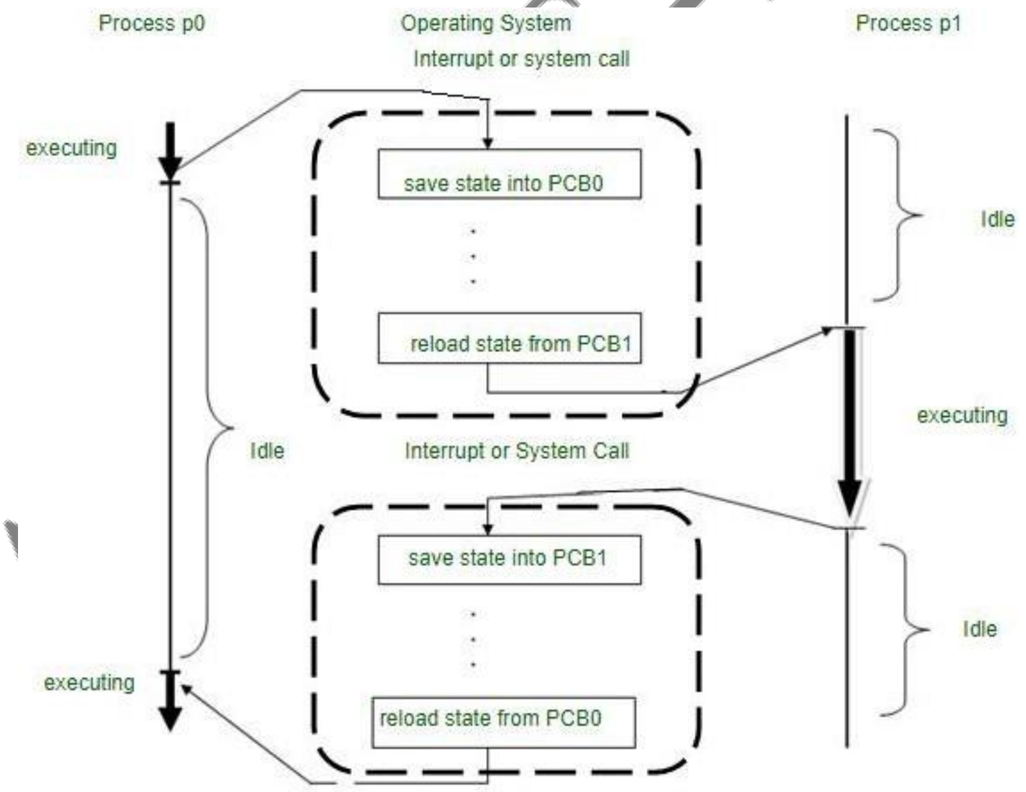
Kernel/User Switch: This trigger is used when the OS needed to switch between the user mode and kernel mode.

When switching between user mode and kernel/user mode is necessary, operating systems use the kernel/user switch.

2.5.3. Process Control Block

The Process Control block(PCB) is also known as a Task Control Block. it represents a process in the Operating System. A process control block (PCB) is a data structure used by a computer to store all information about a process. It is also called the descriptive process. When a process is created (started or installed), the operating system creates a process manager.

2.5.3.1. State Diagram of Context Switching



2.5.4. Working Process Context Switching

So the context switching of two processes, the priority-based process occurs in the ready queue of the process control block. These are the following steps.

- The state of the current process must be saved for rescheduling.
- The process state contains records, credentials, and operating system-specific information stored on the PCB or switch.
- The PCB can be stored in a single layer in kernel memory or in a custom OS file.
- A handle has been added to the PCB to have the system ready to run.
- The operating system aborts the execution of the current process and selects a process from the waiting list by tuning its PCB.
- Load the PCB's program counter and continue execution in the selected process.
- Process/thread values can affect which processes are selected from the queue, this can be important.

2.6. Priority Scheduling Algorithm: Preemptive, Non-Preemptive

What is Priority Scheduling?

Priority Scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.

The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority depends upon memory requirements, time requirements, etc.

2.6.1. Types of Priority Scheduling

Priority scheduling divided into two main types:

2.6.1.1. Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

2.6.1.2. Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy, will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

2.6.2. Characteristics of Priority Scheduling

- A CPU algorithm that schedules processes based on priority.
- It used in Operating systems for performing batch processes.
- If two jobs having the same priority are READY, it works on a FIRST COME, FIRST SERVED basis.

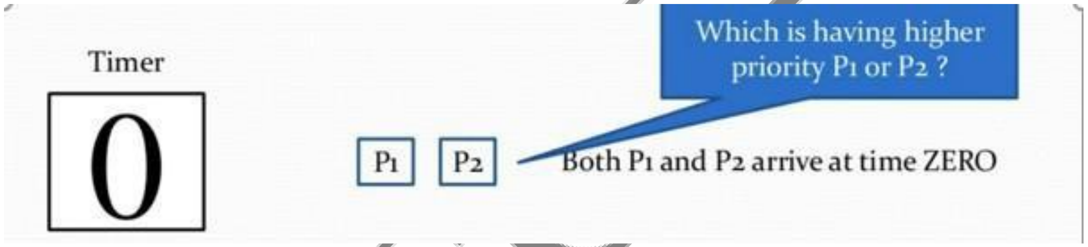
- In priority scheduling, a number is assigned to each process that indicates its priority level.
- Lower the number, higher is the priority.
- In this type of scheduling algorithm, if a newer process arrives, that is having a higher priority than the currently running process, then the currently running process is preempted.

2.6.3. Example of Priority Scheduling

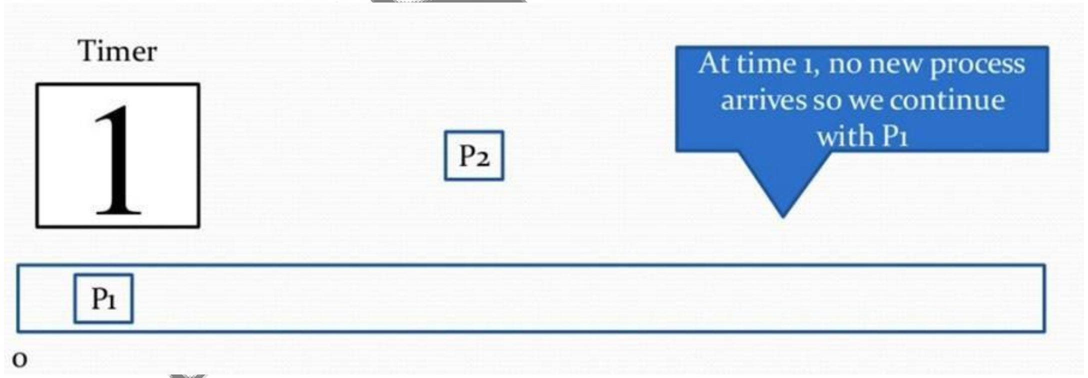
Consider following five processes P1 to P5. Each process has its unique priority, burst time, and arrival time.

Process	**Priority**	**Burst time**	**Arrival time**
P1	1	4	0
P2	2	3	0
P3	1	7	6
P4	3	4	11
P5	2	2	12

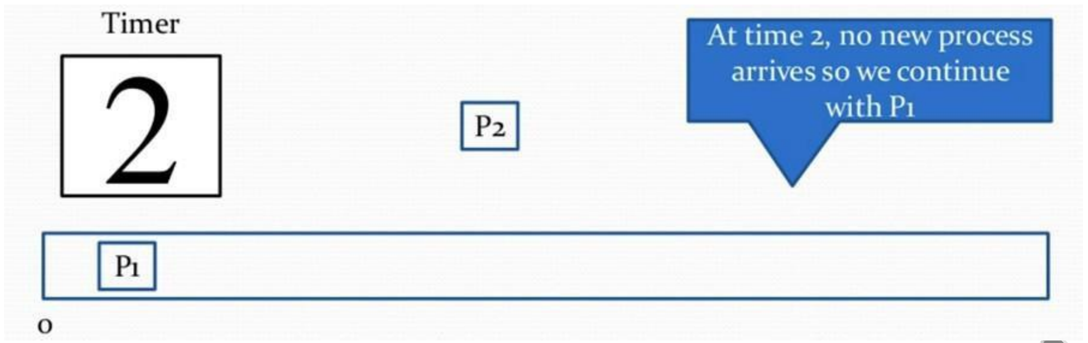
Step 0) At time=0, Process P1 and P2 arrive. P1 has higher priority than P2. The execution begins with process P1, which has burst time 4.


Timer
0
Which is having higher priority P1 or P2 ?
P1
P2
Both P1 and P2 arrive at time ZERO

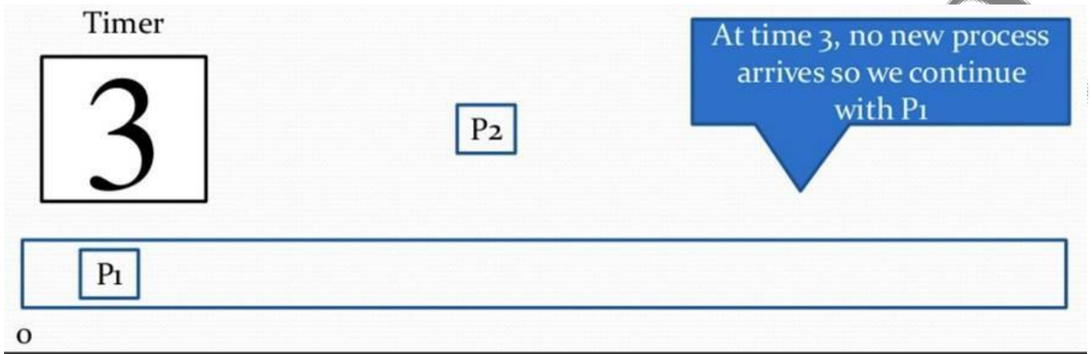
Step 1) At time=1, no new process arrive. Execution continues with P1.



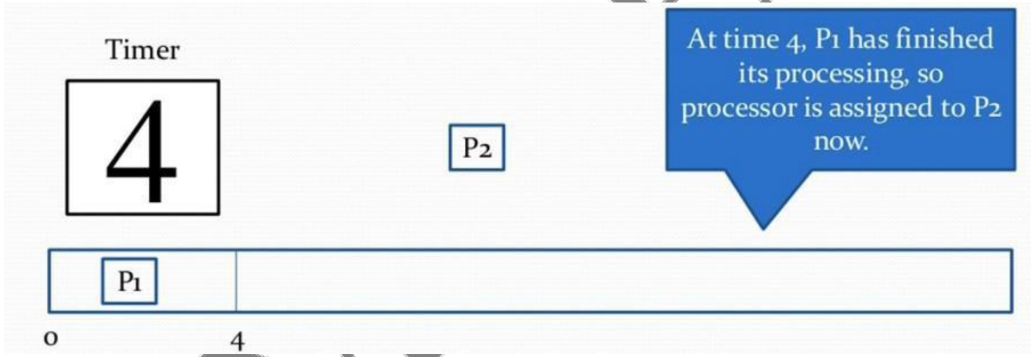
Step 2) At time 2, no new process arrives, so you can continue with P1. P2 is in the waiting queue.



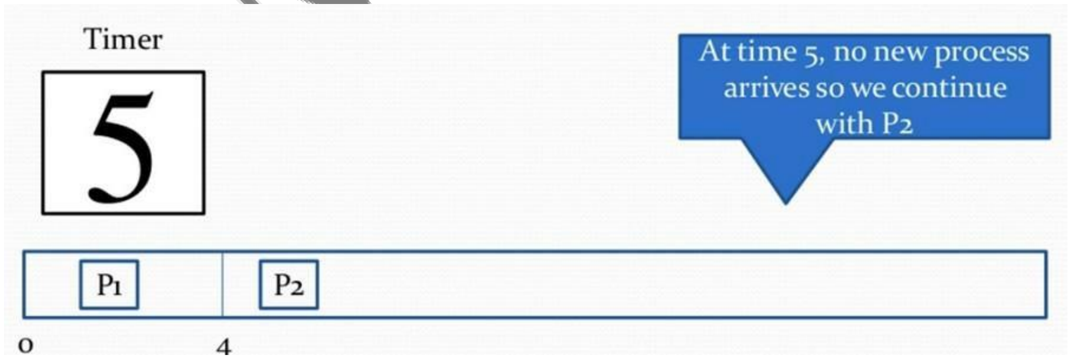
Step 3) At time 3, no new process arrives so you can continue with P1. P2 process still in the waiting queue.



Step 4) At time 4, P1 has finished its execution. P2 starts execution.

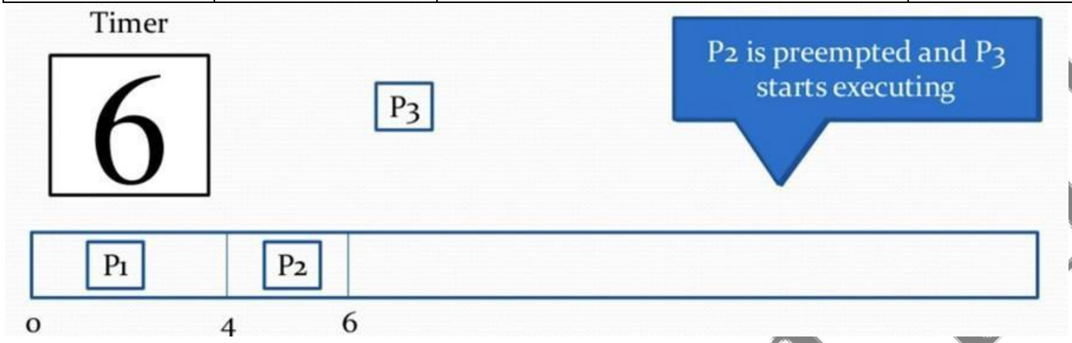

Timer
4
P2
At time 4, P1 has finished its processing, so processor is assigned to P2 now.
P1
0
4

Step 5) At time= 5, no new process arrives, so we continue with P2.

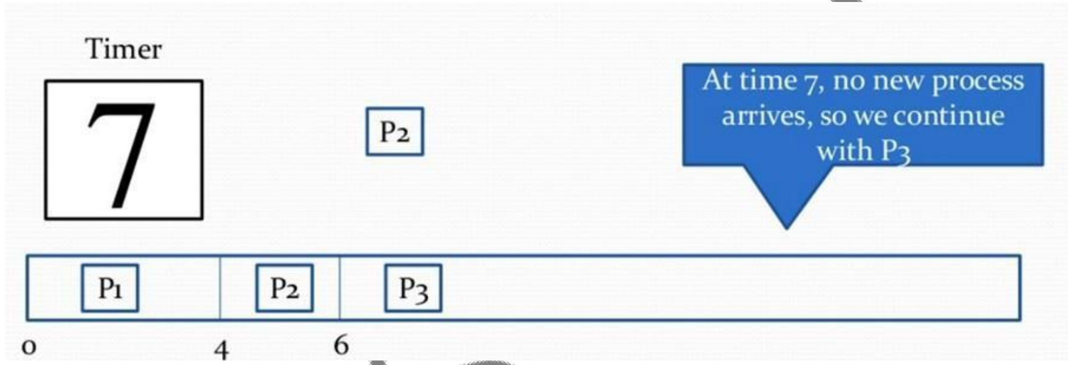


Step 6) At time=6, P3 arrives. P3 is at higher priority (1) compared to P2 having priority (2). P2 is preempted, and P3 begins its execution.

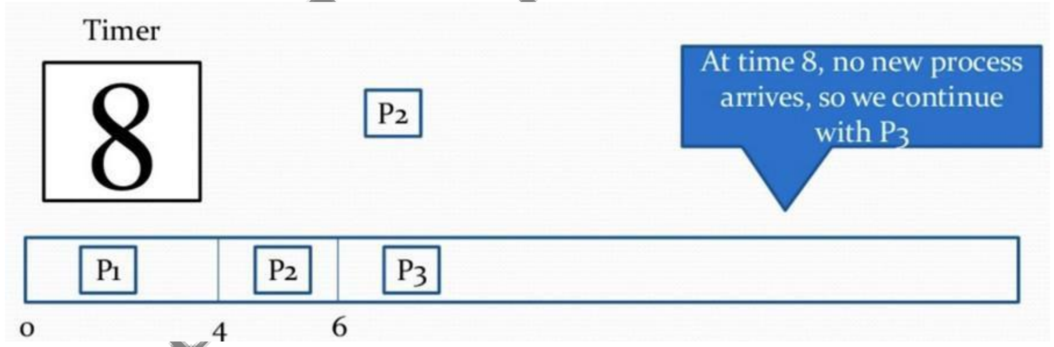
Process	**Priority**	**Burst time**	**Arrival time**
P1	1	4	0
P2	2	1 out of 3 pending	0
P3	1	7	6
P4	3	4	11
P5	2	2	12



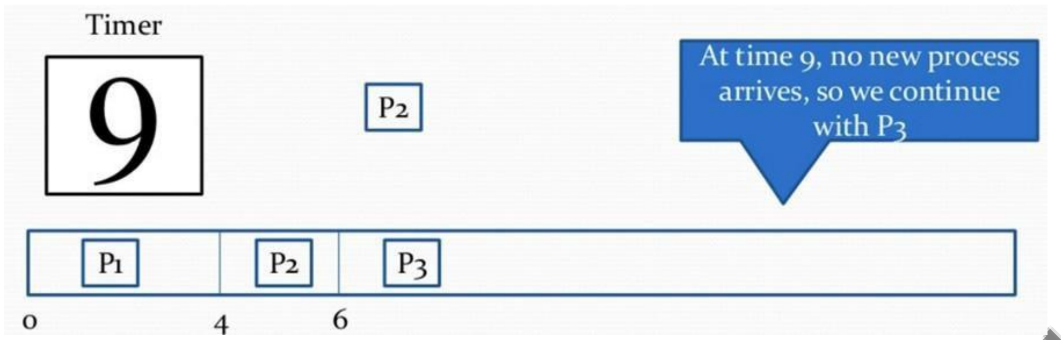
Step 7) At time 7, no-new process arrives, so we continue with P3. P2 is in the waiting queue.



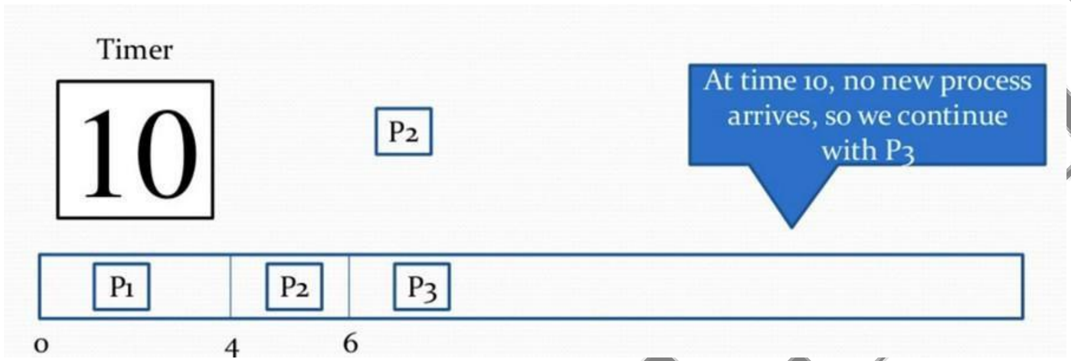
Step 8) At time= 8, no new process arrives, so we can continue with P3.



Step 9) At time= 9, no new process comes so we can continue with P3.

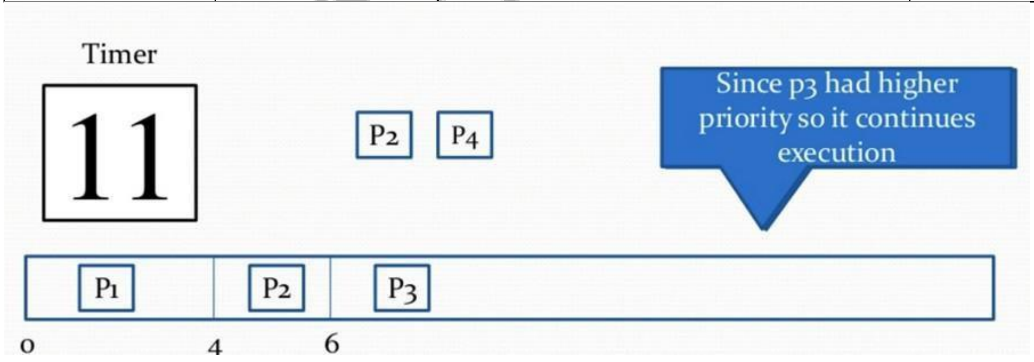


Step 10) At time interval 10, no new process comes, so we continue with P3

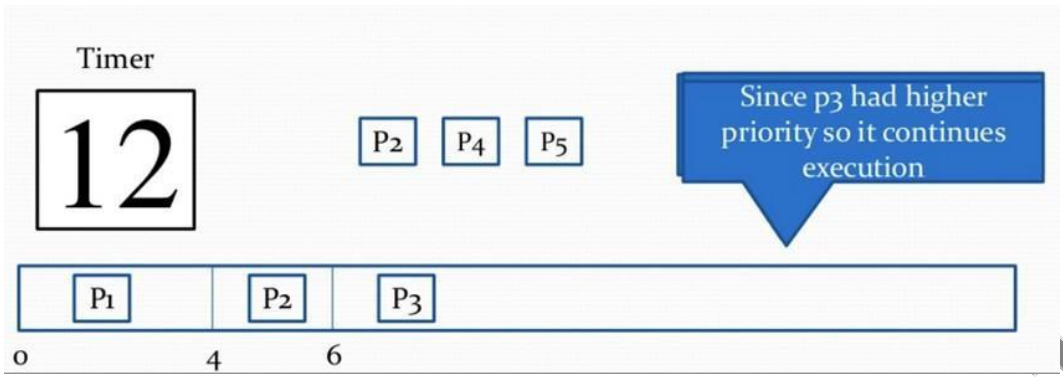


Step 11) At time=11, P4 arrives with priority 4. P3 has higher priority, so it continues its execution.

Process	**Priority**	**Burst time**	**Arrival time**
P1	1	4	0
P2	2	1 out of 3 pending	0
P3	1	2 out of 7 pending	6
P4	3	4	11
P5	2	2	12

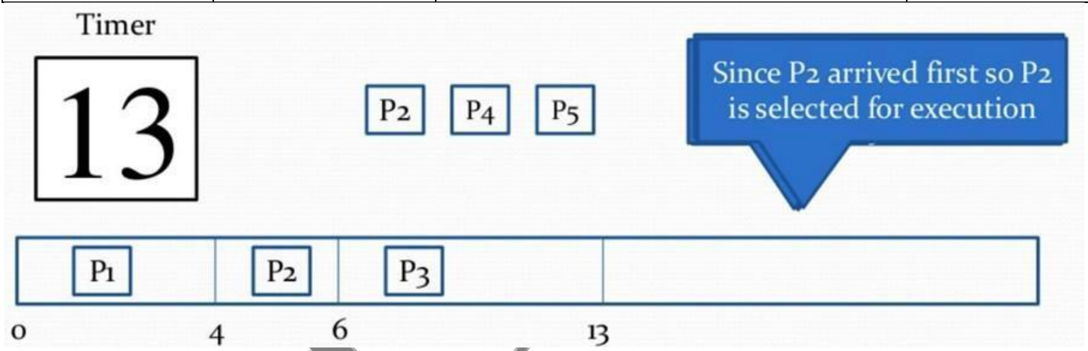


Step 12) At time=12, P5 arrives. P3 has higher priority, so it continues execution.

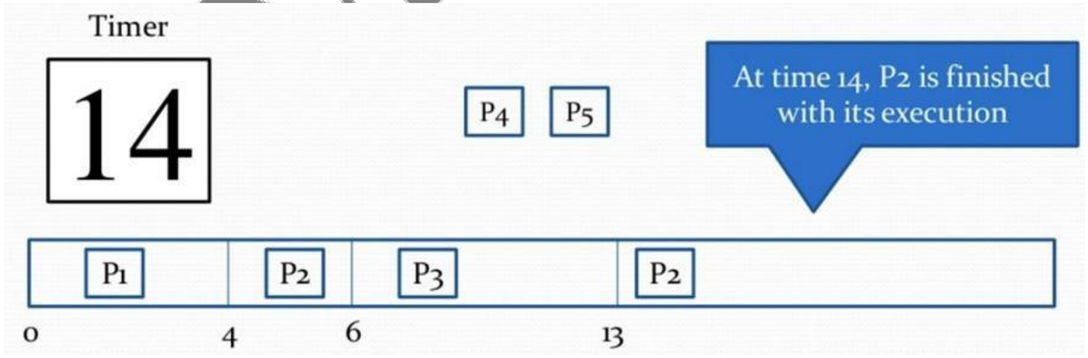


Step 13) At time=13, P3 completes execution. We have P2,P4,P5 in ready queue. P2 and P5 have equal priority. Arrival time of P2 is before P5. So P2 starts execution.

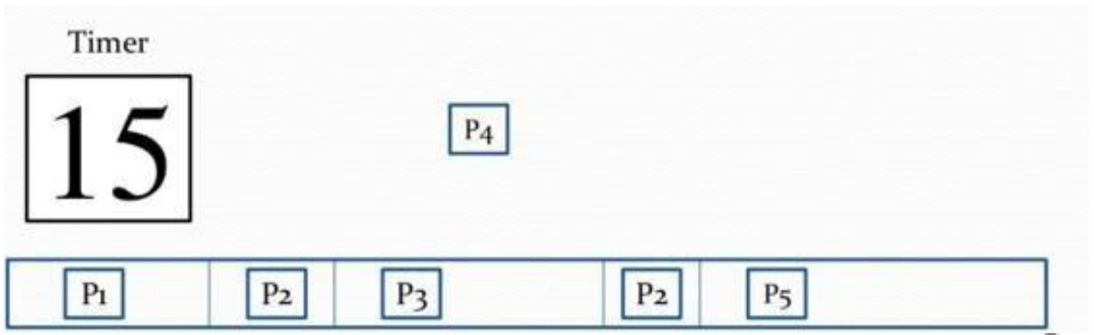
Process	Priority	Burst time	Arrival time
P1	1	4	0
P2	2	1 out of 3 pending	0
P3	1	7	6
P4	3	4	11
P5	2	2	12



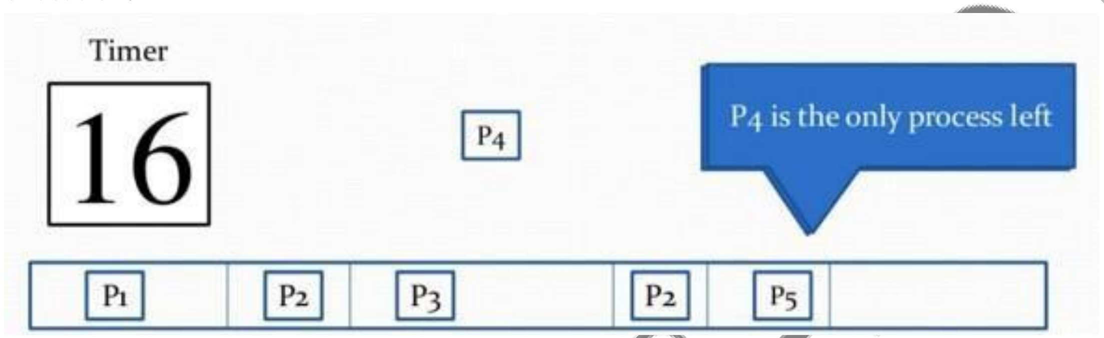
Step 14) At time =14, the P2 process has finished its execution. P4 and P5 are in the waiting state. P5 has the highest priority and starts execution.



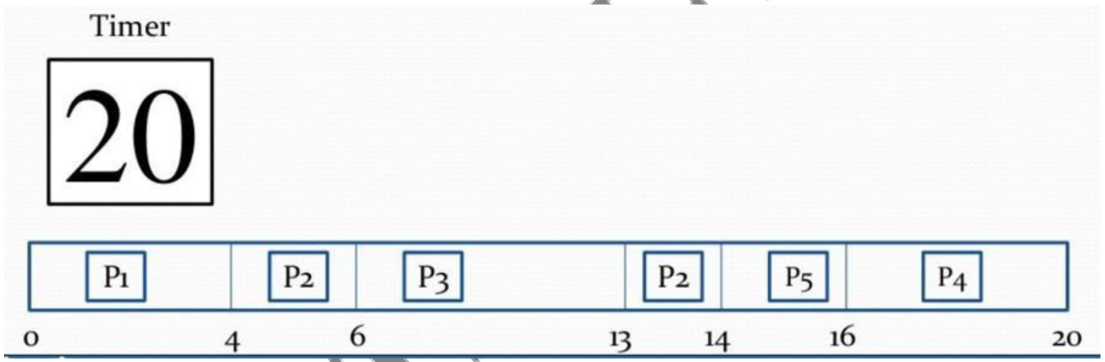
Step 15) At time =15, P5 continues execution.



Step 16) At time= 16, P5 is finished with its execution. P4 is the only process left. It starts execution.



Step 17) At time =20, P5 has completed execution and no process is left.



Step 18) Let’s calculate the average waiting time for the above example.

Waiting Time = start time – arrival time + wait time for next burst

P1 = o - o = o

P2 =4 - o + 7 =11

P3= 6-6=0

P4= 16-11=5

Average Waiting time = (0+11+0+5+2)/5 = 18/5= 3.6

2.6.4. Advantages of priority scheduling

Here, are benefits/pros of using priority scheduling method:

- Easy to use scheduling method
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time

- This method provides a good mechanism where the relative important of each process may be precisely defined.
- Suitable for applications with fluctuating time and resource requirements.

2.6.5. Disadvantages of priority scheduling

Here, are cons/drawbacks of priority scheduling

- If the system eventually crashes, all low priority processes get lost.
- If high priority processes take lots of CPU time, then the lower priority processes may starve and will be postponed for an indefinite time.
- This scheduling algorithm may leave some low priority processes waiting indefinitely.
- A process will be blocked when it is ready to run but has to wait for the CPU because some other process is running currently.
- If a new higher priority process keeps on coming in the ready queue, then the process which is in the waiting state may need to wait for a long duration of time.

2.6.6. Scheduling policies

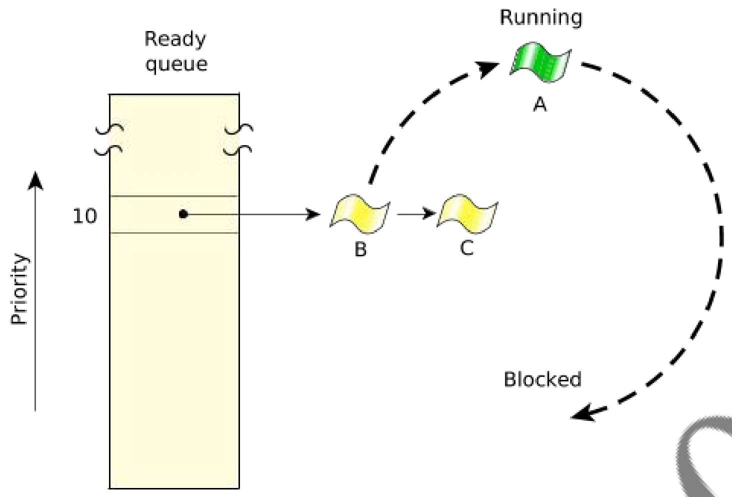
To meet the needs of various applications, the QNX Neutrino RTOS provides these scheduling algorithms:

- FIFO scheduling
- round-robin scheduling
- sporadic scheduling

Each thread in the system may run using any method. The methods are effective on a per-thread basis, not on a global basis for all threads and processes on a node.

Remember that the FIFO and round-robin scheduling policies apply only when two or more threads that share the *same priority* are READY (i.e., the threads are directly competing with each other). The sporadic method, however, employs a “budget” for a thread's execution. In all cases, if a higher-priority thread becomes READY, it immediately preempts all lower-priority threads.

In the following diagram, three threads of equal priority are READY. If Thread A blocks, Thread B will run. Although a thread inherits its scheduling policy from its parent process, the thread can request to change the algorithm applied by the kernel.



Thread A blocks; Thread B runs.

Summary:

- Priority scheduling is a method of scheduling processes that is based on priority. In this algorithm, the scheduler selects the tasks to work as per the priority.
- In Priority Preemptive Scheduling, the tasks are mostly assigned with their priorities.
- In Priority Non-preemptive scheduling method, the CPU has been allocated to a specific process.
- Processes are executed on the basis of priority so high priority does not need to wait for long which saves time
- If high priority processes take lots of CPU time, then the lower priority processes may starve and will be postponed for an indefinite time.

UNIT III

IOT AND ARDUINO PROGRAMMING

- Introduction to the Concept of IoT Devices
- IoT Devices Versus Computers
- IoT Configurations
- Basic Components
- Introduction to Arduino
- Types of Arduino
- Arduino Toolchain
- Arduino Programming Structure
- Sketches
- Pins
- Input/Output From Pins Using Sketches
- Introduction to Arduino Shields
- Integration of Sensors and Actuators with Arduino

3.1. IoT (Internet of Things)

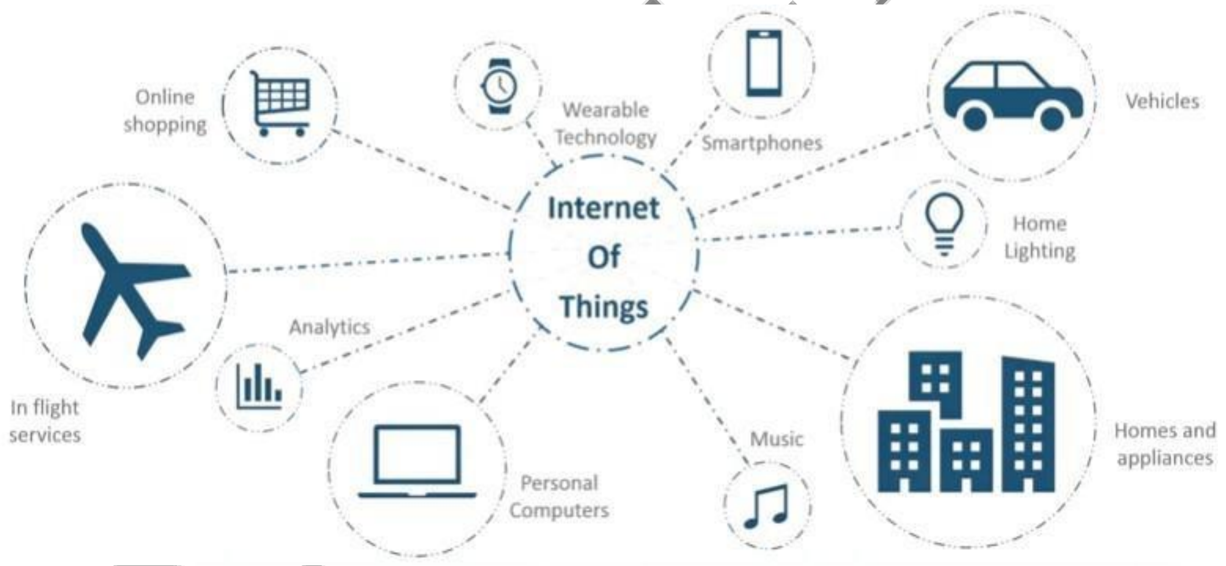
IoT stands for Internet of Things, which means accessing and controlling daily usable equipments and devices using Internet.

3.1.1 What is an Internet of Things (IoT)

Our mobile device which contains GPS Tracking, Mobile Gyroscope, Adaptive brightness, Voice detection, Face detection etc. These components have their own individual features, but what about if these all communicate with each other to provide a better environment? For example, the phone brightness is adjusted based on my GPS location or my direction.

Connecting everyday things embedded with electronics, software, and sensors to internet enabling to collect and exchange data without human interaction called as the Internet of Things (IoT).

The term "Things" in the Internet of Things refers to anything and everything in day to day life which is accessed or connected through the internet.

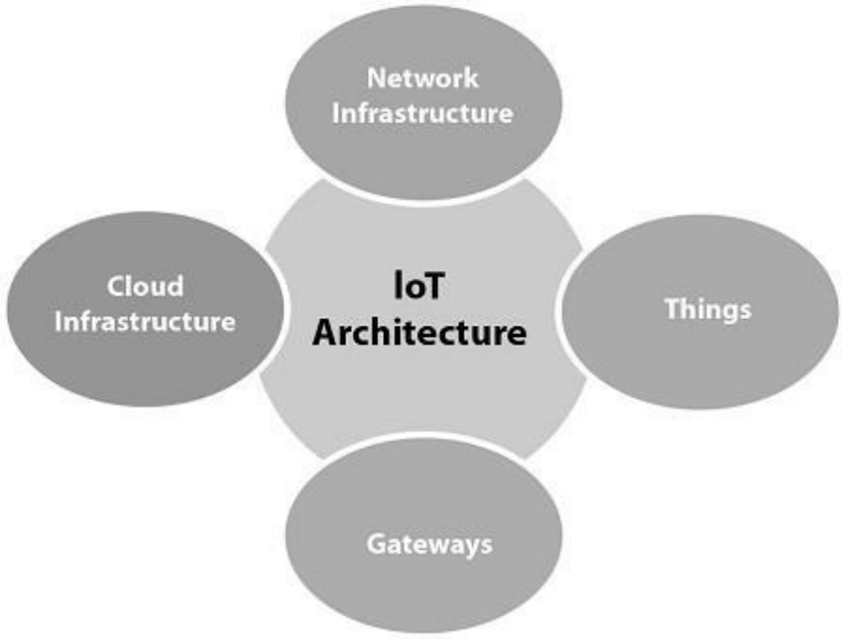


IoT is an advanced automation and analytics system which deals with artificial intelligence, sensor, networking, electronic, cloud messaging etc. to deliver complete systems for the product or services. The system created by IoT has greater transparency, control, and performance.
As we have a platform such as a cloud that contains all the data through which we connect all the things around us. For example, a house, where we can connect our home appliances such as air conditioner, light, etc. through each other and all these things are managed at the same platform. Since we have a platform, we can connect our car, track its fuel meter, speed level, and also track the location of the car.

If there is a common platform where all these things can connect to each other would be great because based on my preference, I can set the room temperature. For example, if I love the room temperature to to be set at 25 or 26-degree Celsius when I reach back home from my office, then according to my car location, my AC would start before 10 minutes I arrive at home. This can be done through the Internet of Things (IoT).

3.1.2. How does Internet of Thing (IoT) Work?

The working of IoT is different for different IoT echo system (architecture). However, the key concept of there working are similar. The entire working process of IoT starts with the device themselves, such as smartphones, digital watches, electronic appliances, which securely communicate with the IoT platform. The platforms collect and analyze the data from all multiple devices and platforms and transfer the most valuable data with applications to devices.



3.1.3. Features of IOT

The most important features of IoT on which it works are connectivity, analyzing, integrating, active engagement, and many more. Some of them are listed below:

Connectivity: Connectivity refers to establish a proper connection between all the things of IoT to IoT platform it may be server or cloud. After connecting the IoT devices, it needs a high speed messaging between the devices and cloud to enable reliable, secure and bi-directional communication.

Analyzing: After connecting all the relevant things, it comes to real-time analyzing the data collected and use them to build effective business intelligence. If we have a good insight into data gathered from all these things, then we call our system has a smart system.

Integrating: IoT integrating the various models to improve the user experience as well.
Artificial Intelligence: IoT makes things smart and enhances life through the use of data. For example, if we have a coffee machine whose beans have going to end, then the coffee machine itself order the coffee beans of your choice from the retailer.

Sensing: The sensor devices used in IoT technologies detect and measure any change in the environment and report on their status. IoT technology brings passive networks to active networks. Without sensors, there could not hold an effective or true IoT environment.

Active Engagement: IoT makes the connected technology, product, or services to active engagement between each other.

Endpoint Management: It is important to be the endpoint management of all the IoT system otherwise, it makes the complete failure of the system. For example, if a coffee machine itself order the coffee beans when it goes to end but what happens when it orders the beans from a retailer and we are not present at home for a few days, it leads to the failure of the IoT system. So, there must be a need for endpoint management.

Advantages and Disadvantages of (IoT)
Any technology available today has not reached to its 100 % capability. It always has a gap to go. So, we can say that **Internet of Things** has a significant technology in a world that can help other technologies to reach its accurate and complete 100 % capability as well.

3.1.3.1. Advantages of IoT
Internet of things facilitates the several advantages in day-to-day life in the business sector. Some of its benefits are given below:

- **Efficient resource utilization:** If we know the functionality and the way that how each device work we definitely increase the efficient resource utilization as well as monitor natural resources.
- **Minimize human effort:** As the devices of IoT interact and communicate with each other and do lot of task for us, then they minimize the human effort.
- **Save time:** As it reduces the human effort then it definitely saves out time. Time is the primary factor which can save through IoT platform.