Microcontrollers and Application Notes for VI Sem ECE

Department of Electronics and Communication Engineering,

SYLLABUS

MALLA REDDY ENGINEERING COLLEGE (Autonomous)
MICROCONTROLLERS AND APPLICATIONS

Course Objective:
This course introduces the architecture of 8051 Microcontroller, the instruction set of 8051, real-time interrupts, real time Timers and interfacing with 8051 Microcontroller.

MODULE I: 8051 Microcontroller [8 Periods]
Introduction, Architecture ofa 8051 microcontroller: Internal and External memories – Counters and Timers – Synchronous serial communication, asynchronous serial communication – Interrupts, I/O Ports, signal description of 8051.
MODULE II: Instruction Set of 8051 [10 Periods]
Basic assembly language programming – Data transfer instructions – Data and Bit manipulation instructions – Arithmetic instructions – Logical operations, Internal RAM, and SFRs – Program flow control instructions – Interrupt control flow.
MODULE III: Real-Time Control - Interrupts [12 Periods]
A: Interrupt handling structure of an MCU – Interrupt Latency and Interrupt deadline – Multiple sources of the interrupts – Non-maskable interrupt sources
B: Enabling or disabling of the sources – Polling to determine the interrupt source and assignment of the priorities among them – Interrupt structure in Intel 8051.
MODULE IV: Real-Time Control – Timers [8 Periods]
Programmable Timers in the MCU's – Free running counter and real time control – Interrupt interval and density constraints, watch dog timer.
MODULEV: Interfacing [10 Periods]
Switch and Keypad - LED and Array of LEDs - Seven Segment, LCD and its interfaces Stepper motor and DC motor interfacing.
TEXT BOOKS:
1. Microcontrollers Architecture, Programming, Interfacing and System Design – Raj Kamal,Pearson Education, 2005.
2. The 8051 Microcontroller and Embedded Systems – Mazidi and Mazidi, PHI, 2000.

REFERENCES:
1. Kenneth. J. Ayala, "The 8051 Microcontroller", Cengage Learning, 3rd Edition, 2004.
2. Microcontrollers (Theory & Applications) – A.V. Deshmuk, WTMH, 2005.
3. Design with PIC Microcontrollers – John B. Peatman, Pearson Education, 2005.

E-RESOURCES:
1. https://www.edgefx.in/8051-microcontroller-architecture/
2. http://www.newagepublishers.com/samplechapter/002079.pdf
3. http://8051-microcontrollers.blogspot.in/2015/11/timers-and-counterstimers.html#.WYbVGLpuLIU
104

4. http://ymk.k-space.org/Lecture_Nov5.pdf
5. http://www.rtcmagazine.com/technologies/view/Microcontrollers
6. http://www.satishkashyap.com/2012/02/video-lectures-on-microprocessors-and.html

Course Outcomes:

At the end of the course, students will be able to:

1. Describe the basic architecture of 8051microcontroller
2. Write assembly language programs for 8051 microcontroller.
3. Know the interrupt handling techniques.
4. Know the usage of timers in real time applications.
5. Develop a microcontroller based system.

UNIT - 1

MICROPROCESSORS AND MICROCONTROLLERS

Microprocessor	*Microcontroller*
Arithmetic and logic unit Accumulator Working Registers Program Counter Stack Pointer Clock Circuit Interrupt circuit	ALU Timer/ Counter IO Ports Accumulator Registers Internal ROM Interrupt Circuits Internal RAM Stack Pointer Clock Program Counter
Block diagram of microprocessor	*Block diagram of microcontroller*
Microprocessor contains ALU, General purpose registers, stack pointer, program counter, clock timing circuit, interrupt circuit	Microcontroller contains the circuitry of microprocessor, and in addition it has built in ROM, RAM, I/O Devices, Timers/Counters etc.
It has many instructions to move data between memory and CPU	It has few instructions to move data between memory and CPU
Few bit handling instruction	It has many bit handling instructions
Less number of pins are multifunctional	More number of pins are multifunctional
Single memory map for data and code (program)	Separate memory map for data and code (program)
Access time for memory and IO are more	Less access time for built in memory and IO.
Microprocessor based system requires additional hardware	It requires less additional hardwares
More flexible in the design point of view	Less flexible since the additional circuits which is residing inside the microcontroller is fixed for a particular microcontroller
Large number of instructions with flexible addressing modes	Limited number of instructions with few addressing modes

RISC AND CISC CPU ARCHITECTURES

Microcontrollers with small instruction set are called reduced instruction set computer (RISC) machines and those with complex instruction set are called complex instruction set computer (CISC). Intel 8051 is an example of CISC machine whereas microchip PIC 18F87X is an example of RISC machine.

RISC	CISC
Instruction takes one or two cycles	Instruction takes multiple cycles
Only load/store instructions are used to access memory	In additions to load and store instructions, memory access is possible with other instructions also.
Instructions executed by hardware	Instructions executed by the micro program
Fixed format instruction	Variable format instructions
Few addressing modes	Many addressing modes
Few instructions	Complex instruction set
Most of the have multiple register banks	Single register bank
Highly pipelined	Less pipelined
Complexity is in the compiler	Complexity in the microprogram

HARVARD & VON- NEUMANN CPU ARCHITECTURE

Von-Neumann (Princeton architecture)	**Harvard architecture**
CPU Data Program Memory Data Memory Address Bus	CPU Data Data Memory Address Bus Data Program Memory Address Bus
Von-Neumann (Princeton architecture)	**Harvard architecture**
It uses single memory space for both instructions and data.	It has separate program memory and data memory
It is not possible to fetch instruction code and data	Instruction code and data can be Fetched simultaneously
Execution of instruction takes more machine cycle	Execution of instruction takes less machine cycle
Uses CISC architecture	Uses RISC architecture
Instruction pre-fetching is a main feature	Instruction parallelism is a main feature
Also known as control flow or control driven computers	Also known as data flow or data Driven computers
Simplifies the chip design because of single memory space	Chip design is complex due to separate memory space
Eg. 8085, 8086, MC6800	Eg. General purpose microcontrollers, special DSP chips etc.

COMPUTER SOFTWARE

A set of instructions written in a specific sequence for the computer to solve a specific task is called a program and software is a collection of such programs.

The program stored in the computer memory in the form of binary numbers is called machine instructions. The *machine language* program is called *object code*.

An *assembly language* is a mnemonic representation of machine language. Machine language and assembly language are low level languages and are processor specific.

The assembly language program the programmer enters is called *source code*. The source code (assembly language) is translated to object code (machine language) using *assembler.*

Programs can be written in *high level languages* such as C, C++ etc. High level language will be converted to machine language using *compiler or interpreter*. Compiler reads the entire program and translate into the object code and then it is executed by the processor. Interpreter takes one statement of the high level language as input and translate it into object code and then executes.

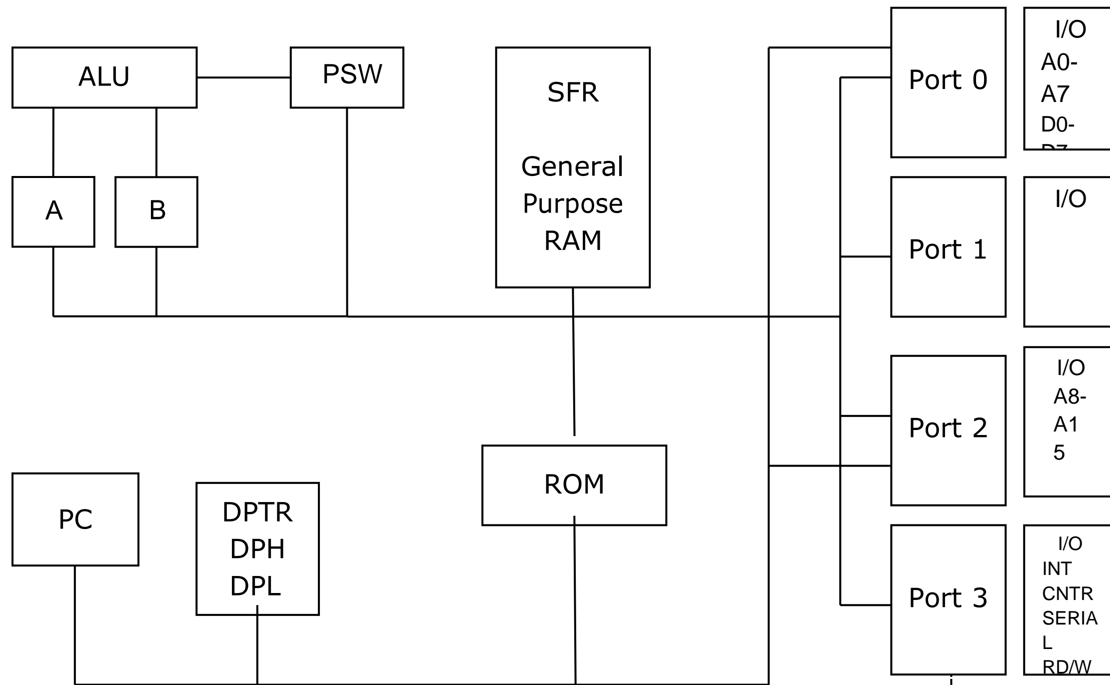
THE 8051 ARCHITECTURE

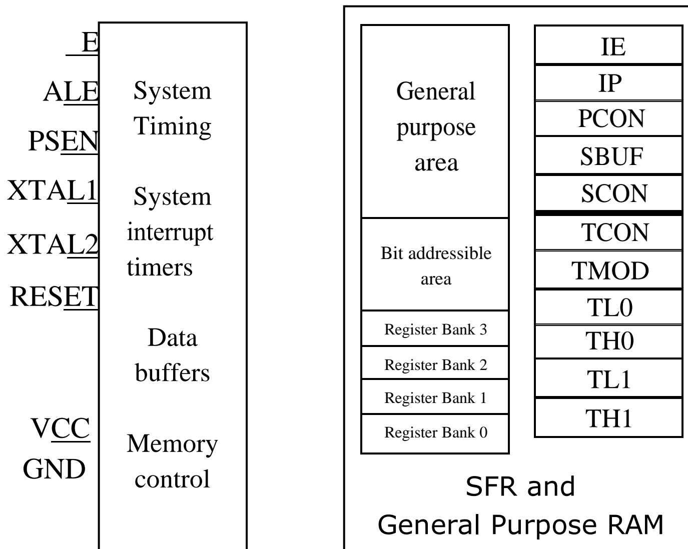
Introduction

Salient features of 8051 microcontroller are given below.

- Eight bit CPU
- On chip clock oscillator
- 4Kbytes of internal program memory (code memory) *[ROM]*
- 128 bytes of internal data memory *[RAM]*
- 64 Kbytes of external program memory address space.
- 64 Kbytes of external data memory address space.
- 32 bi directional I/O lines (can be used as four 8 bit ports or 32 individually addressable I/O lines)
- Two 16 Bit Timer/Counter :T0, T1
- Full Duplex serial data receiver/transmitter
- Four Register banks with 8 registers in each bank.
- Sixteen bit Program counter (PC) and a data pointer (DPTR)
- 8 Bit Program Status Word (PSW)
- 8 Bit Stack Pointer
- Five vector interrupt structure (RESET not considered as an interrupt.)
- 8051 CPU consists of 8 bit ALU with associated registers like accumulator ‘A’ , B register, PSW, SP, 16 bit program counter, stack pointer.
- ALU can perform arithmetic and logic functions on 8 bit variables.
- 8051 has 128 bytes of internal RAM which is divided into
    - Working registers [00 – 1F]
    - Bit addressable memory area [20 – 2F]
    - General purpose memory area (Scratch pad memory) [30-7F]

The 8051 architecture.





- 8051 has 4 K Bytes of internal ROM. The address space is from 0000 to 0FFFh. If the program size is more than 4 K Bytes 8051 will fetch the code automatically from external memory.
- Accumulator is an 8 bit register widely used for all arithmetic and logical operations. Accumulator is also used to transfer data between external memory. B register is used along with Accumulator for multiplication and division. A and B registers together is also called MATH registers.

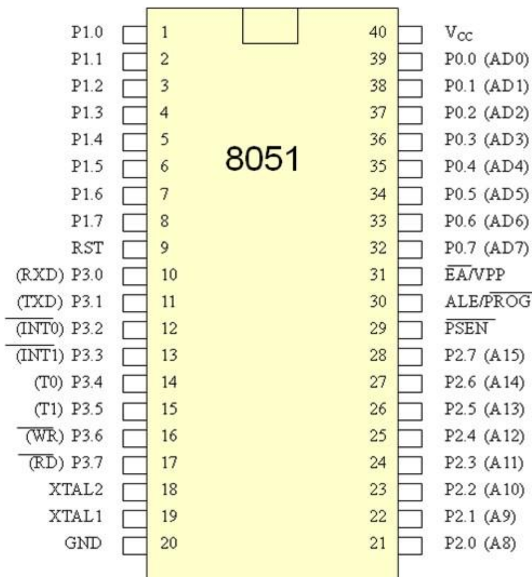
- PSW (Program Status Word). This is an 8 bit register which contains the arithmetic status of ALU and the bank select bits of register banks.

CY	AC	F0	RS1	RS0	OV	-	P

CY - carry flag
AC - auxiliary carry flag
F0 - available to the user for general purpose
RS1,RS0 - register bank select bits
OV - overflow
P - parity

- Stack Pointer (SP) – it contains the address of the data item on the top of the stack. Stack may reside anywhere on the internal RAM. On reset, SP is initialized to 07 so that the default stack will start from address 08 onwards.
- Data Pointer (DPTR) – DPH (Data pointer higher byte), DPL (Data pointer lower byte). This is a 16 bit register which is used to furnish address information for internal and external program memory and for external data memory.
- Program Counter (PC) – 16 bit PC contains the address of next instruction to be executed. On reset PC will set to 0000. After fetching every instruction PC will increment by one.

PIN DIAGRAM



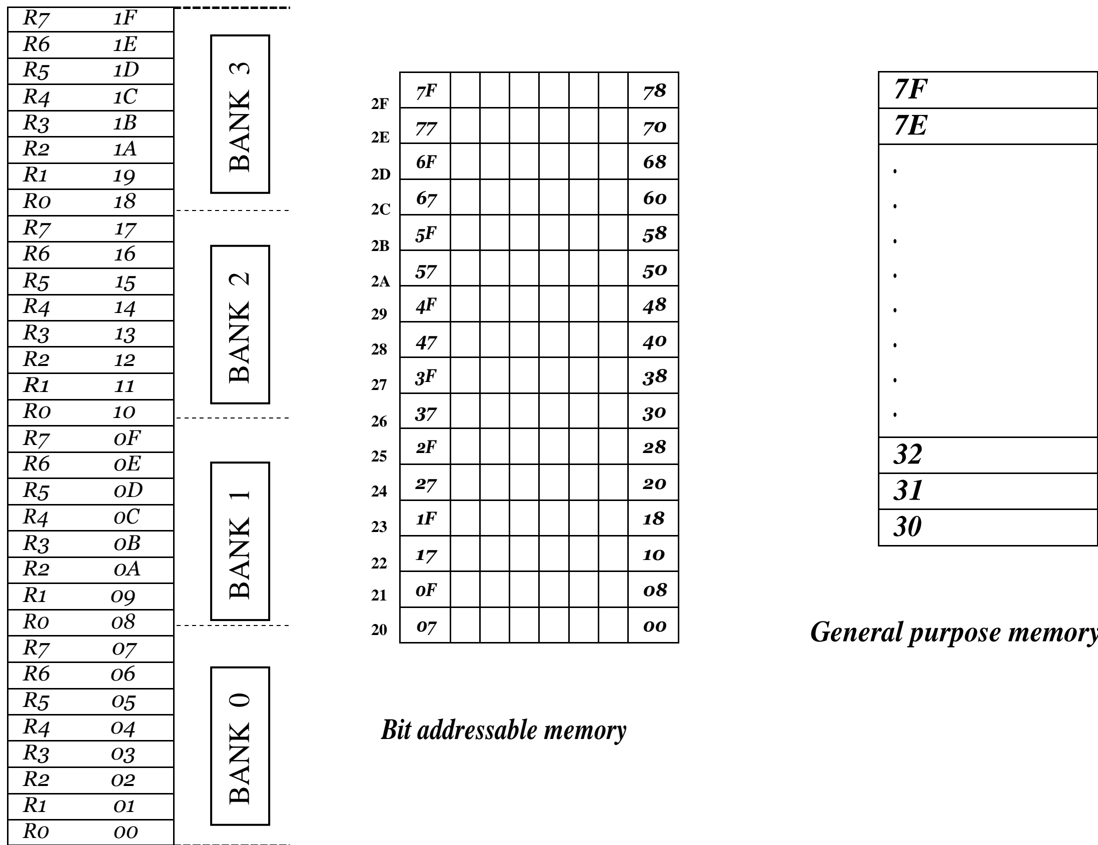
Pinout Description

Pins 1-8	**PORT 1**. Each of these pins can be configured as an input or an output.
Pin 9	**RESET**. A logic one on this pin disables the microcontroller and clears the contents of most registers. In other words, the positive voltage on this pin resets the microcontroller. By applying logic zero to this pin, the program starts execution from the beginning.
Pins10-17	**PORT 3**. Similar to port 1, each of these pins can serve as general input or output. Besides, all of them have alternative functions

Pin 10	**RXD.** Serial asynchronous communication input or Serial synchronous communication output.
Pin 11	**TXD.** Serial asynchronous communication output or Serial synchronous communication clock output.
Pin 12	**INT0.**External Interrupt 0 input
Pin 13	**INT1.** External Interrupt 1 input
Pin 14	**T0.** Counter 0 clock input
Pin 15	**T1.** Counter 1 clock input
Pin 16	**WR**. Write to external (additional) RAM
Pin 17	**RD.** Read from external RAM
Pin 18, 19	**XTAL2, XTAL1.** Internal oscillator input and output. A quartz crystal which specifies operating frequency is usually connected to these pins.
Pin 20	**GND**. Ground.
Pin 21-28	**Port 2**. If there is no intention to use external memory then these port pins are configured as general inputs/outputs. In case external memory is used, the higher address byte, i.e. addresses A8-A15 will appear on this port. Even though memory with capacity of 64Kb is not used, which means that not all eight port bits are used for its addressing, the rest of them are not available as inputs/outputs.
Pin 29	**PSEN.** If external ROM is used for storing program then a logic zero (0) appears on it every time the microcontroller reads a byte from memory.
Pin 30	**ALE.** Prior to reading from external memory, the microcontroller puts the lower address byte (A0-A7) on P0 and activates the ALE output. After receiving signal from the ALE pin, the external latch latches the state of P0 and uses it as a memory chip address. Immediately after that, the ALE pin is returned its previous logic state and P0 is now used as a Data Bus.
Pin 31	**EA**. By applying logic zero to this pin, P2 and P3 are used for data and address transmission with no regard to whether there is internal memory or not. It means that even there is a program written to the microcontroller, it will not be executed. Instead, the program written to external ROM will be executed. By applying logic one to the EA pin, the microcontroller will use both memories, first internal then external (if exists).
Pin 32-39	**PORT 0**. Similar to P2, if external memory is not used, these pins can be used as general inputs/outputs. Otherwise, P0 is configured as address output (A0-A7) when the ALE pin is driven high (1) or as data output (Data Bus) when the ALE pin is driven low (0).
Pin 40	**VCC**. +5V power supply.

MEMORY ORGANIZATION

Internal RAM organization



Bit addressable memory

General purpose memory

Working Registers

Register Banks: 00h to 1Fh. The 8051 uses 8 general-purpose registers R0 through R7 (R0, R1, R2, R3, R4, R5, R6, and R7). There are four such register banks. Selection of register bank can be done through RS1,RS0 bits of PSW. On reset, the default Register Bank 0 will be selected.

Bit Addressable RAM: 20h to 2Fh . The 8051 supports a special feature which allows access to bit variables. This is where individual memory bits in Internal RAM can be set or cleared. In all there are 128 bits numbered 00h to 7Fh. Being bit variables any one variable can have a value 0 or 1. A bit variable can be set with a command such as SETB and cleared with a command such as CLR. Example instructions are:

SETB 25h ; sets the bit 25h (becomes 1)

CLR 25h ; clears bit 25h (becomes 0)

Note, bit 25h is actually bit 5 of Internal RAM location 24h.

The Bit Addressable area of the RAM is just 16 bytes of Internal RAM located between 20h and 2Fh.

General Purpose RAM: 30h to 7Fh. Even if 80 bytes of Internal RAM memory are available for general-purpose data storage, user should take care while using the memory location from 00 -2Fh

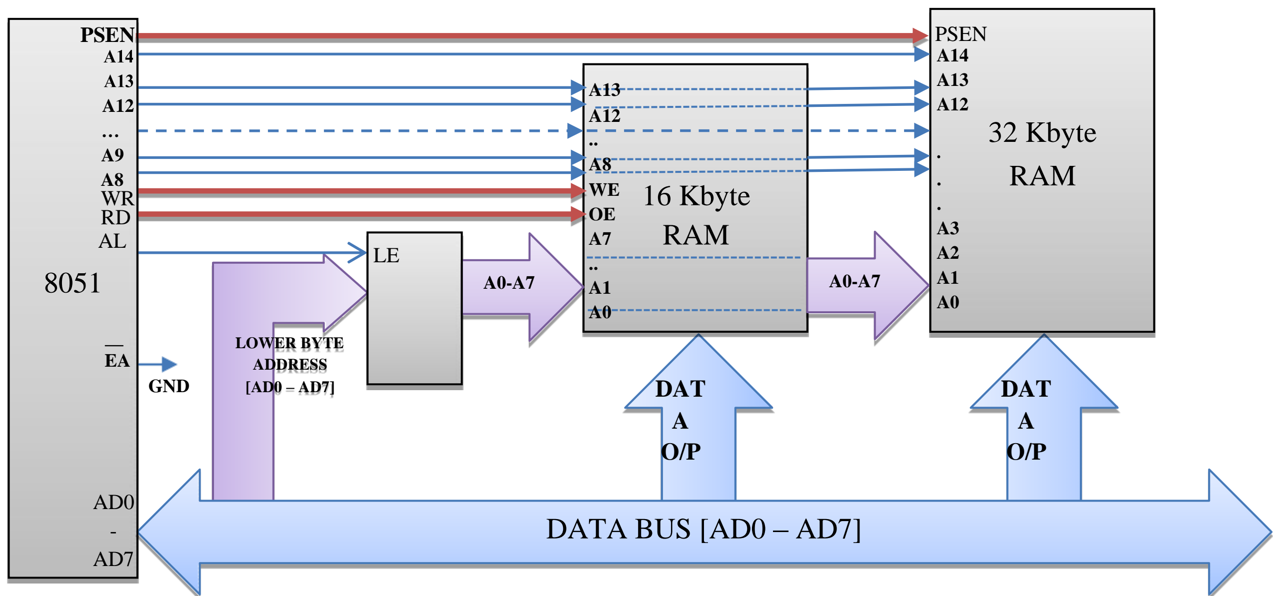
since these locations are also the default register space, stack space, and bit addressable space. It is a good practice to use general purpose memory from 30 – 7Fh. The general purpose RAM can be accessed using direct or indirect addressing modes.

EXTERNAL MEMORY INTERFACING

Eg. Interfacing of 16 K Byte of RAM and 32 K Byte of EPROM to 8051

Number of address lines required for ***16 Kbyte memory is 14 lines*** and that ***of 32Kbytes of memory is 15 lines.***

The connections of external memory is shown below.



The lower order address and data bus are multiplexed. De-multiplexing is done by the latch. Initially the address will appear in the bus and this latched at the output of latch using ALE signal. The output of the latch is directly connected to the lower byte address lines of the memory. Later data will be available in this bus. Still the latch output is address it self. The higher byte of address bus is directly connected to the memory. The number of lines connected depends on the memory size.

The RD and WR (both active low) signals are connected to RAM for reading and writing the data.

PSEN of microcontroller is connected to the output enable of the ROM to read the data from the memory.

EA (active low) pin is always grounded if we use only external memory. Otherwise, once the program size exceeds internal memory the microcontroller will automatically switch to external memory.

STACK

A stack is a last in first out memory. In 8051 internal RAM space can be used as stack. The address of the stack is contained in a register called stack pointer. Instructions PUSH and POP are used for stack operations. When a data is to be placed on the stack, the stack pointer increments before storing the data on the stack so that the stack grows up as data is stored (pre-increment). As the data is retrieved from the stack the byte is read from the stack, and then SP decrements to point the next available byte of stored data (post decrement). The stack pointer is set to 07 when the 8051 resets. So that default stack memory starts from address location 08 onwards (to avoid overwriting the default register bank ie., bank 0).

Eg; Show the stack and SP for the following.

```
                     [SP]=07        //CONTENT OF SP IS 07 (DEFAULT VALUE)
MOV R6,              [R6]=25H       //CONTENT OF R6 IS 25H
#25H MOV             [R1]=12H       //CONTENT OF R1 IS 12H
R1, #12H             [R4]=F3H       //CONTENT OF R4 IS F3H
MOV R4, #0F3H

PUSH 6               [SP]=08        [08]=[06]=25H        //CONTENT OF 08 IS 25H
PUSH 1               [SP]=09        [09]=[01]=12H        //CONTENT OF 09 IS 12H
PUSH 4               [SP]=0A        [0A]=[04]=F3H        //CONTENT OF 0A IS F3H

POP 6                [06]=[0A]=F3H        [SP]=09 //CONTENT OF 06 IS F3H
POP 1                [01]=[09]=12H        [SP]=08 //CONTENT OF 01 IS 12H
POP 4                [04]=[08]=25H        [SP]=07 //CONTENT OF 04 IS 25H
```

Counters and Timers:

A **timer** is a specialized type of clock which is used to measure time intervals. A timer that counts from zero upwards for measuring time elapsed is often called a **stopwatch**. It is a device that counts down from a specified time interval and used to generate a time delay, for example, an hourglass is a timer.

A **counter** is a device that stores (and sometimes displays) the number of times a particular event or process occurred, with respect to a clock signal. It is used to count the events happening outside the microcontroller. In electronics, counters can be implemented quite easily using register-type circuits such as a flip-flop.

Difference between a Timer and a Counter

The points that differentiate a timer from a counter are as follows −

Timer	Counter
The register incremented for every machine cycle.	The register is incremented considering 1 to 0 transition at its corresponding to an external input pin (T0, T1).
Maximum count rate is 1/12 of	Maximum count rate is 1/24 of the oscillator

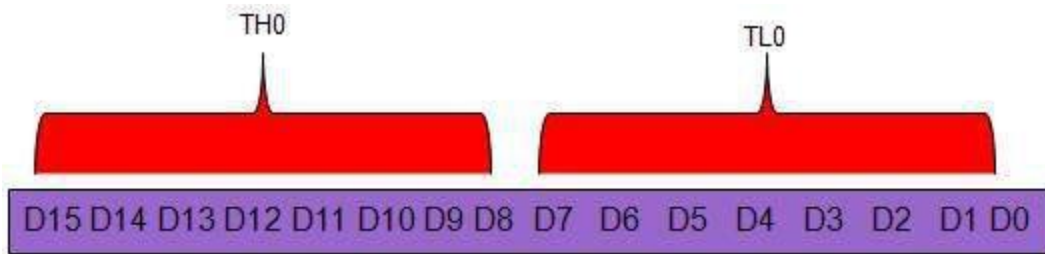
the oscillator frequency.	frequency.
A timer uses the frequency of the internal clock, and generates delay.	A counter uses an external signal to count pulses.

Timers of 8051 and their Associated Registers

The 8051 has two timers, Timer 0 and Timer 1. They can be used as timers or as event counters. Both Timer 0 and Timer 1 are 16-bit wide. Since the 8051 follows an 8-bit architecture, each 16 bit is accessed as two separate registers of low-byte and high-byte.

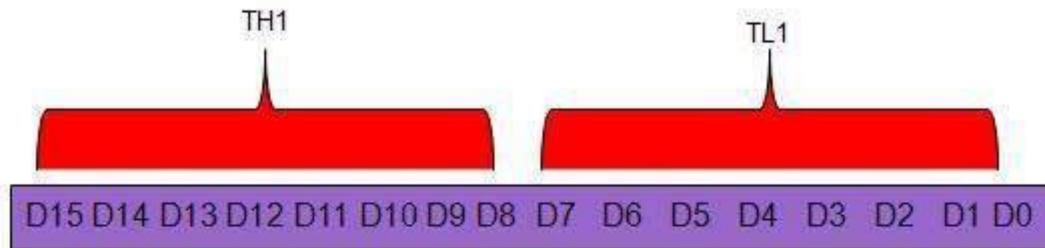
Timer 0 Register

The 16-bit register of Timer 0 is accessed as low- and high-byte. The low-byte register is called TL0 (Timer 0 low byte) and the high-byte register is called TH0 (Timer 0 high byte). These registers can be accessed like any other register. For example, the instruction **MOV TL0, #4H** moves the value into the low-byte of Timer #0.



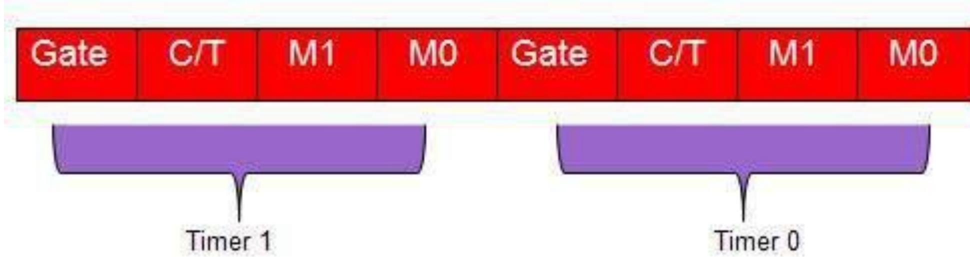
Timer 1 Register

The 16-bit register of Timer 1 is accessed as low- and high-byte. The low-byte register is called TL1 (Timer 1 low byte) and the high-byte register is called TH1 (Timer 1 high byte). These registers can be accessed like any other register. For example, the instruction **MOV TL1, #4H** moves the value into the low-byte of Timer 1.



TMOD (Timer Mode) Register

Both Timer 0 and Timer 1 use the same register to set the various timer operation modes. It is an 8-bit register in which the lower 4 bits are set aside for Timer 0 and the upper four bits for Timers. In each case, the lower 2 bits are used to set the timer mode in advance and the upper 2 bits are used to specify the location.



Gate − When set, the timer only runs while INT(0,1) is high.

C/T − Counter/Timer select bit.

M1 − Mode bit 1.

M0 − Mode bit 0.

GATE

Every timer has a means of starting and stopping. Some timers do this by software, some by hardware, and some have both software and hardware controls. 8051 timers have both software and hardware controls. The start and stop of a timer is controlled by software using the instruction **SETB TR1** and **CLR TR1** for timer 1, and **SETB TR0** and **CLR TR0** for timer 0.

The SETB instruction is used to start it and it is stopped by the CLR instruction. These instructions start and stop the timers as long as GATE = 0 in the TMOD register. Timers can be started and stopped by an external source by making GATE = 1 in the TMOD register.

C/T (CLOCK / TIMER)

This bit in the TMOD register is used to decide whether a timer is used as a **delay generator** or an **event manager**. If C/T = 0, it is used as a timer for timer delay generation. The clock source to create the time delay is the crystal frequency of the 8051. If C/T = 0, the crystal frequency attached to the 8051 also decides the speed at which the 8051 timer ticks at a regular interval.

Timer frequency is always 1/12th of the frequency of the crystal attached to the 8051. Although various 8051 based systems have an XTAL frequency of 10 MHz to 40 MHz, we normally work with the XTAL frequency of 11.0592 MHz. It is because the baud rate for serial communication of the 8051.XTAL = 11.0592 allows the 8051 system to communicate with the PC with no errors.

M1 / M2

M1	M2	Mode
0	0	13-bit timer mode.
0	1	16-bit timer mode.
1	0	8-bit auto reload mode.

1	1	Spilt mode.

Different Modes of Timers

Mode 0 (13-Bit Timer Mode)

Both Timer 1 and Timer 0 in Mode 0 operate as 8-bit counters (with a divide-by-32 prescaler). Timer register is configured as a 13-bit register consisting of all the 8 bits of TH1 and the lower 5 bits of TL1. The upper 3 bits of TL1 are indeterminate and should be ignored. Setting the run flag (TR1) does not clear the register. The timer interrupt flag TF1 is set when the count rolls over from all 1s to all 0s. Mode 0 operation is the same for Timer 0 as it is for Timer 1.

Mode 1 (16-Bit Timer Mode)

Timer mode "1" is a 16-bit timer and is a commonly used mode. It functions in the same way as 13-bit mode except that all 16 bits are used. TLx is incremented starting from 0 to a maximum 255. Once the value 255 is reached, TLx resets to 0 and then THx is incremented by 1. As being a full 16-bit timer, the timer may contain up to 65536 distinct values and it will overflow back to 0 after 65,536 machine cycles.

If a value, say YYXXH, is loaded into the Timer bytes, then the delay produced by the Timer will be equal to the product :
[(**FFFFH – YYXXH +1**) x (period of one timer clock)].
It can also be considered as follows: convert YYXXH into decimal, say NNNNN, then delay will be equal to the product :
[(**65536-NNNNN**) x (period of one timer clock)].
The period of one timer clock is 1.085 µs for a crystal of 11.0592 MHz frequency as discussed above.
Now to produce a desired delay, divide the required delay by the Timer clock period. Assume that the division yields a number NNNNN. This is the number of times Timer must be updated before it stops. Subtract this number from 65536 (binary equivalent of FFFFH) and convert the difference into hex. This will be the initial value to be loaded into the Timer to get the desired delay.

Mode 2 (8 Bit Auto Reload)

Both the timer registers are configured as 8-bit counters (TL1 and TL0) with automatic reload. Overflow from TL1 (TL0) sets TF1 (TF0) and also reloads TL1 (TL0) with the contents of Th1 (TH0), which is preset by software. The reload leaves TH1 (TH0) unchanged.

The benefit of auto-reload mode is that you can have the timer to always contain a value from 200 to 255. If you use mode 0 or 1, you would have to check in the code to see the overflow and, in that case, reset the timer to 200. In this case, precious instructions check the value and/or get reloaded. In mode 2, the microcontroller takes care of this. Once you have configured a timer in mode 2, you don't have to worry about checking to see if the timer has overflowed, nor do you have to worry about resetting the value because the microcontroller hardware will do it all for you. The auto-reload mode is used for establishing a common baud rate.

Mode 3 (Split Timer Mode)

Timer mode "3" is known as **split-timer mode**. When Timer 0 is placed in mode 3, it becomes two separate 8-bit timers. Timer 0 is TL0 and Timer 1 is TH0. Both the timers count from 0 to 255 and in case of overflow, reset back to 0. All the bits that are of Timer 1 will now be tied to TH0.

When Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be set in modes 0, 1 or 2, but it cannot be started/stopped as the bits that do that are now linked to TH0. The real timer 1 will be incremented with every machine cycle.

Initializing a Timer

Decide the timer mode. Consider a 16-bit timer that runs continuously, and is independent of any external pins.

Initialize the TMOD SFR. Use the lowest 4 bits of TMOD and consider Timer 0. Keep the two bits, GATE 0 and C/T 0, as 0, since we want the timer to be independent of the external pins. As 16-bit mode is timer mode 1, clear T0M1 and set T0M0. Effectively, the only bit to turn on is bit 0 of TMOD. Now execute the following instruction −

MOV TMOD,#01h

Now, Timer 0 is in 16-bit timer mode, but the timer is not running. To start the timer in running mode, set the TR0 bit by executing the following instruction −

SETB TR0

Now, Timer 0 will immediately start counting, being incremented once every machine cycle.

Reading a Timer

A 16-bit timer can be read in two ways. Either read the actual value of the timer as a 16-bit number, or you detect when the timer has overflowed.

Detecting Timer Overflow

When a timer overflows from its highest value to 0, the microcontroller automatically sets the TFx bit in the TCON register. So instead of checking the exact value of the timer, the TFx bit can be checked. If TF0 is set, then Timer 0 has overflowed; if TF1 is set, then Timer 1 has overflowed.

Synchronous and Asynchronous serial communication:

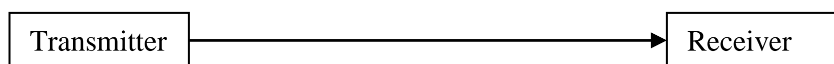
DATA COMMUNICATION

The 8051 microcontroller is parallel device that transfers eight bits of data simultaneously over eight data lines to parallel I/O devices. Parallel data transfer over a long is very expensive. Hence, a serial communication is widely used in long distance communication. In serial data communication, 8-bit data is converted to serial bits using a parallel in serial out shift register and then it is transmitted over a single data line. The data byte is always transmitted with least significant bit first.

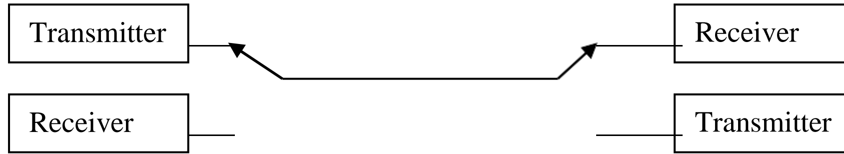
BASICS OF SERIAL DATA COMMUNICATION,

Communication Links

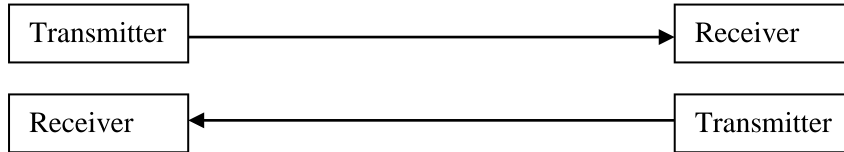
1. ***Simplex communication link:*** In simplex transmission, the line is dedicated for transmission. The transmitter sends and the receiver receives the data.



2. ***Half duplex communication link:*** In half duplex, the communication link can be used for either transmission or
reception. Data is transmitted in only one direction at a time.



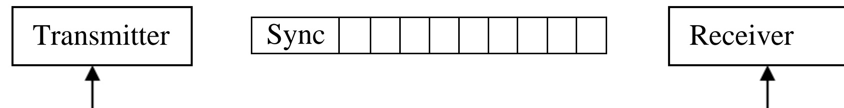
3. ***Full duplex communication link:*** If the data is transmitted in both ways at the same time, it is a full duplex i.e. transmission and reception can proceed simultaneously. This communication link requires two wires for data, one for transmission and one for reception.



Types of Serial communication:

Serial data communication uses two types of communication.

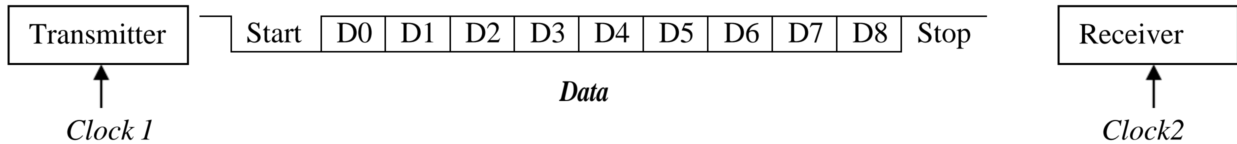
1. ***Synchronous serial data communication:*** In this transmitter and receiver are synchronized. It uses a common clock to synchronize the receiver and the transmitter. First the synch character is sent and then the data is transmitted. This format is generally used for high speed transmission. In Synchronous serial data communication a block of data is transmitted at a time.



Data

Clock

2. ***Asynchronous Serial data transmission:*** In this, different clock sources are used for transmitter and receiver. In this mode, data is transmitted with start and stop bits. A transmission begins with start bit, followed by data and then stop bit. For error checking purpose parity bit is included just prior to stop bit. In Asynchronous serial data communication a single byte is transmitted at a time.



Baud rate:

The rate at which the data is transmitted is called baud or transfer rate. The baud rate is the reciprocal of the time to send one bit. In asynchronous transmission, baud rate is not equal to number of bits per second. This is because; each byte is preceded by a start bit and followed by parity and stop bit. For example, in synchronous transmission, if data is transmitted with 9600 baud, it means that 9600 bits are transmitted in one second. For bit transmission time = 1 second/ 9600 = 0.104 ms.

8051 SERIAL COMMUNICATION

The 8051 supports a full duplex serial port.

Three special function registers support serial communication.

1. SBUF Register: Serial Buffer (SBUF) register is an 8-bit register. It has separate SBUF registers for data transmission and for data reception. For a byte of data to be transferred via the TXD line, it must be placed in SBUF register. Similarly, SBUF holds the 8-bit data received by the RXD pin and read to accept the received data.
2. SCON register: The contents of the Serial Control (SCON) register are shown below. This register contains mode selection bits, serial port interrupt bit (TI and RI) and also the ninth data bit for transmission and reception (TB8 and RB8).

Serial Port Control (SCON) Register							
D7	D6	D5	D4	D3	D2	D1	D0
SM0	**SM1**	**SM2**	**REN**	**TB8**	**RB8**	**TI**	**RI**

- SM0 (SCON.7) : Serial communication mode selection bit
- SM1 (SCON.6) : Serial communication mode selection bit

SM0	SM1	Mode	Description	Baud rate
0	0	Mode 0	8-bit shift register mode	Fosc / 12
0	1	Mode 1	8-bit UART	Variable (set by timer 1)
1	0	Mode 2	9-bit UART	Fosc/ 32 or Fosc/64
1	1	Mode 3	9-bit UART	Variable (set by timer 1)

- SM2 (SCON.5) : Multiprocessor communication bit. In modes 2 and 3, if set this will enable multiprocessor communication.
- REN (SCON.4) : Enable serial reception
- TB8 (SCON.3) : This is 9th bit that is transmitted in mode 2 & 3.
- RB8 (SCON.2) : 9th data bit is received in modes 2 & 3.
- TI (SCON.1) : Transmit interrupt flag, set by hardware must be cleared by software.
- RI (SCON.0) : Receive interrupt flag, set by hardware must be cleared by software.

3. PCON register: The SMOD bit (bit 7) of PCON register controls the baud rate in asynchronous mode transmission.

Power mode Control (PCON) Register							
D7	D6	D5	D4	D3	D2	D1	D0
SMOD	--	--	--	**GF1**	**GF0**	**PD**	**IDL**

- SMD (PCON.7): Serial rate modify bit. Set to 1 by program to double baud rate using timer 1 for modes 1, 2, and 3. cleared by program to use timer 1 baud rate.
- GF1 (PCON.3) : General Purpose user flag bit.
- GF0 (PCON.2) : General Purpose user flag bit.
- PD (PCON.1) : Power down bit. Set to 1 by program to enter power down configuration for CHMOS processors.
- IDL (PCON.0) : Idle mode bit. Set to 1 by program to enter idle mode configuration for CHMOS processors.

SERIAL COMMUNICATION MODES

1. ***Mode 0***

   In this mode serial port runs in synchronous mode. The data is transmitted and received through RXD pin and TXD is used for clock output. In this mode the baud rate is 1/12 of clock frequency.

2. ***Mode 1***

   In this mode SBUF becomes a 10 bit full duplex transceiver. The ten bits are 1 start bit, 8 data bit and 1 stop bit. The interrupt flag TI/RI will be set once transmission or reception is over. In this mode the baud rate is variable and is determined by the timer 1 overflow rate. Baud rate =

   $[2^{smod}/32]$ x Timer 1 overflow Rate

   = $[2^{smod}/32]$ x [Oscillator Clock Frequency] / [12 x [256 – [TH1]]]

3. ***Mode 2***

   This is similar to mode 1 except 11 bits are transmitted or received. The 11 bits are, 1 start bit, 8 data bit, a programmable 9th data bit, 1 stop bit.

   Baud rate = $[2^{smod}/64]$ x Oscillator Clock Frequency

4. ***Mode 3***

   This is similar to mode 2 except baud rate is calculated as in mode 1

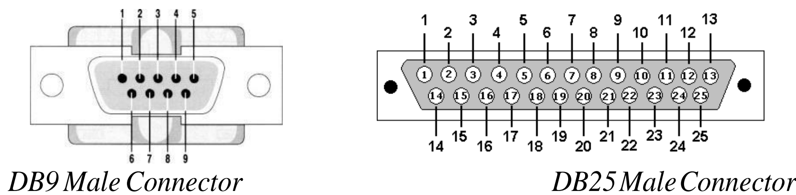
CONNECTIONS TO RS-232

RS-232 standards:

To allow compatibility among data communication equipment made by various manufactures, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. Since the standard was set long before the advent of logic family, its input and output voltage levels are not TTL compatible.

In RS232, a logic one (1) is represented by -3 to -25V and referred as MARK while logic zero (0) is represented by +3 to +25V and referred as SPACE. For this reason to connect any RS232 to a microcontroller system we must use voltage converters such as MAX232 to convert the TTL logic level to RS232 voltage levels and vice-versa. MAX232 IC chips are commonly referred as line drivers.

In RS232 standard we use two types of connectors. DB9 connector or DB25 connector.



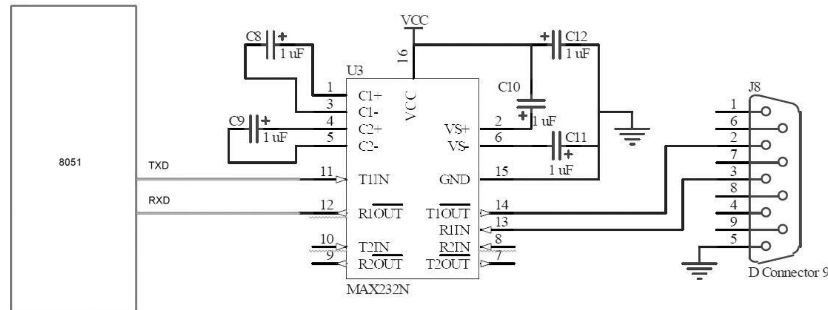
DB9 Male Connector *DB25 Male Connector*

The pin description of DB9 and DB25 Connectors are as follows

DB-25 Pin No.	DB-9 Pin No.	Abbreviation	Full Name
Pin 2	Pin 3	TD	Transmit Data
Pin 3	Pin 2	RD	Receive Data
Pin 4	Pin 7	RTS	Request To Send
Pin 5	Pin 8	CTS	Clear To Send
Pin 6	Pin 6	DSR	Data Set Ready
Pin 7	Pin 5	SG	Signal Ground
Pin 8	Pin 1	CD	Carrier Detect
Pin 20	Pin 4	DTR	Data Terminal Ready
Pin 22	Pin 9	RI	Ring Indicator

The 8051 connection to MAX232 is as follows.

The 8051 has two pins that are used specifically for transferring and receiving data serially. These two pins are called TXD, RXD. Pin 11 of the 8051 (P3.1) assigned to TXD and pin 10 (P3.0) is designated as RXD. These pins TTL compatible; therefore they require line driver (MAX 232) to make them RS232 compatible. MAX 232 converts RS232 voltage levels to TTL voltage levels and vice versa. One advantage of the MAX232 is that it uses a +5V power source which is the same as the source voltage for the 8051. The typical connection diagram between MAX 232 and 8051 is shown below.



SERIAL COMMUNICATION PROGRAMMING IN ASSEMBLY AND C.

Steps to programming the 8051 to transfer data serially

1. The TMOD register is loaded with the value 20H, indicating the use of the Timer 1 in mode 2 (8-bit auto reload) to set the baud rate.
2. The TH1 is loaded with one of the values in table 5.1 to set the baud rate for serial data transfer.
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits.
4. TR1 is set to 1 start timer 1.
5. TI is cleared by the “CLR TI” instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction JNB TI, target to see if the character has been transferred completely.
8. To transfer the next character, go to step 5.

Example 1. Write a program for the 8051 to transfer letter ‘A’ serially at 4800- baud rate, 8 bit data, 1 stop bit continuously.

```
ORG 0000H
LJMP
START
ORG 0030H
START: MOV TMOD, #20H  ; select timer 1 mode 2
MOV TH1, #0FAH         ; load count to get baud rate of 4800
MOV SCON, #50H         ; initialize UART in mode 2
                        ; 8 bit data and 1 stop bit
SETB TR1               ; start timer
AGAIN: MOV SBUF, #'A'  ; load char ‘A’ in SBUF
BACK: JNB TI, BACK ; Check for transmit interrupt flag
CLR TI             ; Clear transmit interrupt flag
SJMP AGAIN
END
```

Example 2. Write a program for the 8051 to transfer the message ‘EARTH’ serially at 9600 baud, 8 bit data, 1 stop bit continuously.

```
ORG 0000H
LJMP
```

START

```
ORG 0030H
        START: MOV TMOD, #20H      ; select timer 1 mode 2
        MOV TH1, #0FDH             ; load count to get reqd. baud rate of 9600
        MOV SCON, #50H             ; initialise uart in mode 2
                                   ; 8 bit data and 1 stop bit
        SETB TR1                   ; start timer
        LOOP: MOV A, #'E'     ; load 1st letter 'E' in a
        ACALL LOAD                 ; call load subroutine
        MOV A, #'A'           ; load 2nd letter 'A' in a
        ACALL LOAD                 ; call load subroutine
        MOV A, #'R'           ; load 3rd letter 'R' in a
        ACALL LOAD                 ; call load subroutine
        MOV A, #'T'           ; load 4th letter 'T' in a
        ACALL LOAD                 ; call load subroutine
        MOV A, #'H'           ; load 4th letter 'H' in a
        ACALL LOAD                 ; call load subroutine
        SJMP LOOP                  ; repeat steps

        LOAD: MOV SBUF, A
        HERE: JNB TI, HERE ; Check for transmit interrupt flag
                CLR TI        ; Clear transmit interrupt flag
                RET

        END
```

Interrupts:

An interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention. Whenever an interrupt occurs, the controller completes the execution of the current instruction and starts the execution of an **Interrupt Service Routine** (ISR) or **Interrupt Handler**. ISR tells the processor or controller what to do when the interrupt occurs. The interrupts can be either hardware interrupts or software interrupts.

Hardware Interrupt

A hardware interrupt is an electronic alerting signal sent to the processor from an external device, like a disk controller or an external peripheral. For example, when we press a key on the keyboard or move the mouse, they trigger hardware interrupts which cause the processor to read the keystroke or mouse position.

Software Interrupt

A software interrupt is caused either by an exceptional condition or a special instruction in the instruction set which causes an interrupt when it is executed by the processor. For example, if the processor's arithmetic logic unit runs a command to divide a number by zero, to cause a divide-by-zero exception, thus causing the computer to abandon the calculation or display an error message. Software interrupt instructions work similar to subroutine calls.

What is Polling?

The state of continuous monitoring is known as **polling**. The microcontroller keeps checking the status of other devices; and while doing so, it does no other operation and consumes all its processing time for monitoring. This problem can be addressed by using interrupts.

In the interrupt method, the controller responds only when an interruption occurs. Thus, the controller is not required to regularly monitor the status (flags, signals etc.) of interfaced and inbuilt devices.

Interrupts v/s Polling

Here is an analogy that differentiates an interrupt from polling −

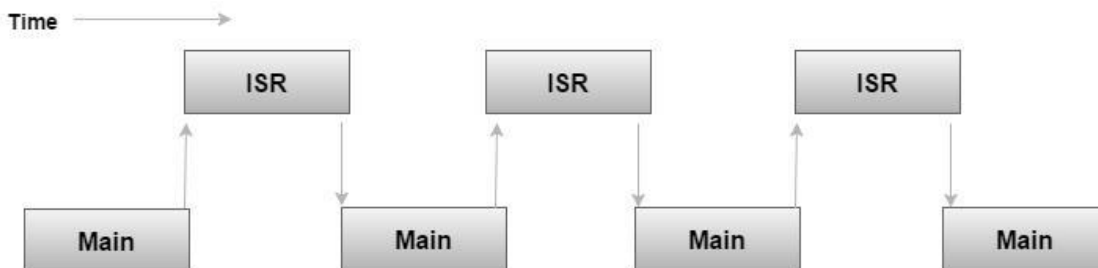
Interrupt	Polling
An interrupt is like a **shopkeeper**. If one needs a service or product, he goes to him and apprises him of his needs. In case of interrupts, when the flags or signals are received, they notify the controller that they need to be serviced.	The polling method is like a **salesperson**. The salesman goes from door to door while requesting to buy a product or service. Similarly, the controller keeps monitoring the flags or signals one by one for all devices and provides service to whichever component that needs its service.

Interrupt Service Routine

For every interrupt, there must be an interrupt service routine (ISR), or **interrupt handler**. When an interrupt occurs, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine, ISR. The table of memory locations set aside to hold the addresses of ISRs is called as the Interrupt Vector Table.

Program Execution without Interrupts

Program Execution with Interrupts



ISR : Interrupt Service Routine

Interrupt Vector Table

There are six interrupts including RESET in 8051.

Interrupts	ROM Location (Hex)	Pin
Serial COM (RI and TI)	0023	
Timer 1 interrupts(TF1)	001B	
External HW interrupt 1 (INT1)	0013	P3.3 (13)
External HW interrupt 0 (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
Reset	0000	9

- When the reset pin is activated, the 8051 jumps to the address location 0000. This is power-up reset.
- Two interrupts are set aside for the timers: one for timer 0 and one for timer 1. Memory locations are 000BH and 001BH respectively in the interrupt vector table.
- Two interrupts are set aside for hardware external interrupts. Pin no. 12 and Pin no. 13 in Port 3 are for the external hardware interrupts INT0 and INT1, respectively. Memory locations are 0003H and 0013H respectively in the interrupt vector table.
- Serial communication has a single interrupt that belongs to both receive and transmit. Memory location 0023H belongs to this interrupt.

Steps to Execute an Interrupt

When an interrupt gets active, the microcontroller goes through the following steps −

- The microcontroller closes the currently executing instruction and saves the address of the next instruction (PC) on the stack.
- It also saves the current status of all the interrupts internally (i.e., not on the stack).
- It jumps to the memory location of the interrupt vector table that holds the address of the interrupts service routine.
- The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine, which is RETI (return from interrupt).

- Upon executing the RETI instruction, the microcontroller returns to the location where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top bytes of the stack into the PC. Then, it start to execute from that address.

Edge Triggering vs. Level Triggering

Interrupt modules are of two types − level-triggered or edge-triggered.

Level Triggered	Edge Triggered
A level-triggered interrupt module always generates an interrupt whenever the level of the interrupt source is asserted.	An edge-triggered interrupt module generates an interrupt only when it detects an asserting edge of the interrupt source. The edge gets detected when the interrupt source level actually changes. It can also be detected by periodic sampling and detecting an asserted level when the previous sample was de-asserted.
If the interrupt source is still asserted when the firmware interrupt handler handles the interrupt, the interrupt module will regenerate the interrupt, causing the interrupt handler to be invoked again.	Edge-triggered interrupt modules can be acted immediately, no matter how the interrupt source behaves.
Level-triggered interrupts are cumbersome for firmware.	Edge-triggered interrupts keep the firmware's code complexity low, reduce the number of conditions for firmware, and provide more flexibility when interrupts are handled.

Enabling and Disabling an Interrupt

Upon Reset, all the interrupts are disabled even if they are activated. The interrupts must be enabled using software in order for the microcontroller to respond to those interrupts.

IE (interrupt enable) register is responsible for enabling and disabling the interrupt. IE is a bitaddressable register.

Interrupt Enable Register

EA	-	ET2	ES	ET1	EX1	ET0	EX0

- **EA** − Global enable/disable.
- **-** − Undefined.
- **ET2** − Enable Timer 2 interrupt.

- **ES** − Enable Serial port interrupt.
- **ET1** − Enable Timer 1 interrupt.
- **EX1** − Enable External 1 interrupt.
- **ET0** − Enable Timer 0 interrupt.
- **EX0** − Enable External 0 interrupt.

To enable an interrupt, we take the following steps −

- Bit D7 of the IE register (EA) must be high to allow the rest of register to take effect.
- If EA = 1, interrupts will be enabled and will be responded to, if their corresponding bits in IE are high. If EA = 0, no interrupts will respond, even if their associated pins in the IE register are high.

Interrupt Priority in 8051

We can alter the interrupt priority by assigning the higher priority to any one of the interrupts. This is accomplished by programming a register called **IP** (interrupt priority).

The following figure shows the bits of IP register. Upon reset, the IP register contains all 0's. To give a higher priority to any of the interrupts, we make the corresponding bit in the IP register high.

-	-	-	-	PT1	PX1	PT0	PX0

-	IP.7	Not Implemented.
-	IP.6	Not Implemented.
-	IP.5	Not Implemented.
-	IP.4	Not Implemented.
PT1	IP.3	Defines the Timer 1 interrupt priority level.
PX1	IP.2	Defines the External Interrupt 1 priority level.
PT0	IP.1	Defines the Timer 0 interrupt priority level.
PX0	IP.0	Defines the External Interrupt 0 priority level.

Interrupt inside Interrupt

What happens if the 8051 is executing an ISR that belongs to an interrupt and another one gets active? In such cases, a high-priority interrupt can interrupt a low-priority interrupt. This is known as **interrupt**

inside interrupt. In 8051, a low-priority interrupt can be interrupted by a high-priority interrupt, but not by any another low-priority interrupt.

Triggering an Interrupt by Software

There are times when we need to test an ISR by way of simulation. This can be done with the simple instructions to set the interrupt high and thereby cause the 8051 to jump to the interrupt vector table. For example, set the IE bit as 1 for timer 1. An instruction **SETB TF1** will interrupt the 8051 in whatever it is doing and force it to jump to the interrupt vector table.

I/O Ports:

8051 microcontroller have 4 I/O ports each of 8-bit, which can be configured as input or output. Hence, total 32 I/O pins allows the microcontroller to be connected with the peripheral devices.

1) PORT 0

P0 can be used as a bidirectional I/O port or it can be used for address/data connected for accessing external memory. When control is 1 the port is used for address or data interfacing. When the control is 0 then the port can be used as a bidirectional I/O port.

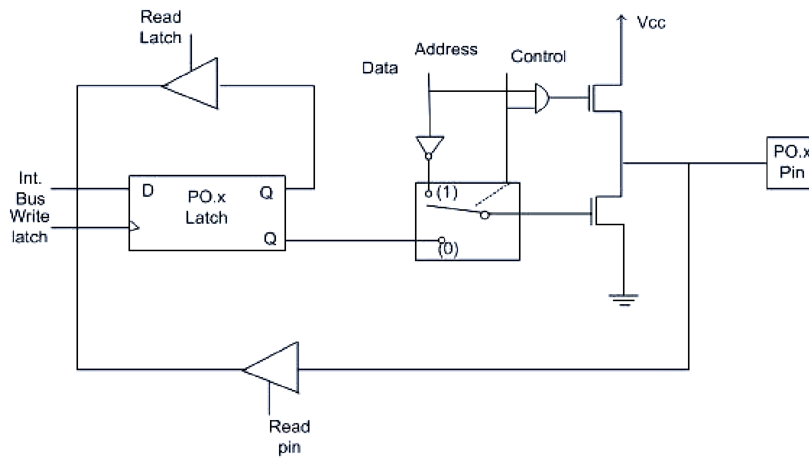


Fig: Structure of port 0 pin

PORT 0 as an Input Port

If the control is 0 then the port is used as an input port and 1 is written to the latch. In this type of situation both the output MOSFETs are off. Since the output pin has floats therefore, whatever data written on pin is directly read by read pin.

PORT 0 as an Output Port

If we want to write 1 on pin of P0, a '1' written to the latch which turns 'off' the lower FET while due to '0' control signal upper FET also turns off.

Suppose we want to write '0' on pin of port 0, when '0' is written to the latch, the pin is pulled down by the lower FET. Hence the output becomes zero.

2) PORT 1

PORT 1 is dedicated only for I/O interfacing. When used as an output port, not needed to connect additional pull-up resistor like port 0.

To use PORT 1 as an input port '1' has to be written to the latch. In this mode 1 is written to the pin by the external device then it read fine.

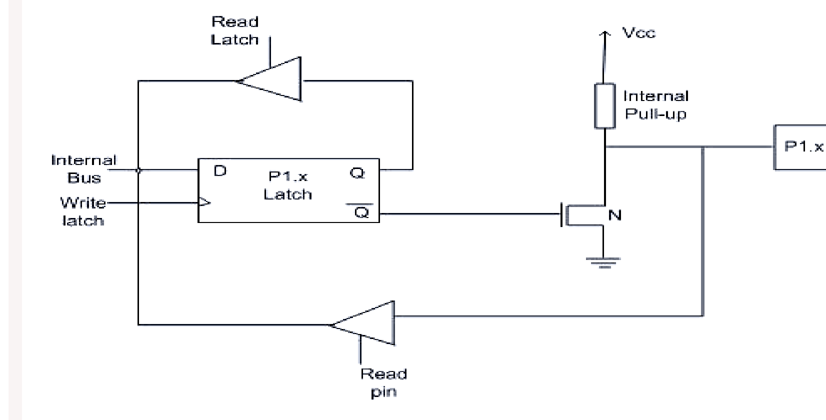


Fig: Structure of port 1 pin

3) PORT 2

PORT 2 is used for higher external address byte or a normal I/O port. Here, the I/O operation is similar to PORT 1. Latch of PORT 2 remains stable when Port 2 pin are used for external memory access.

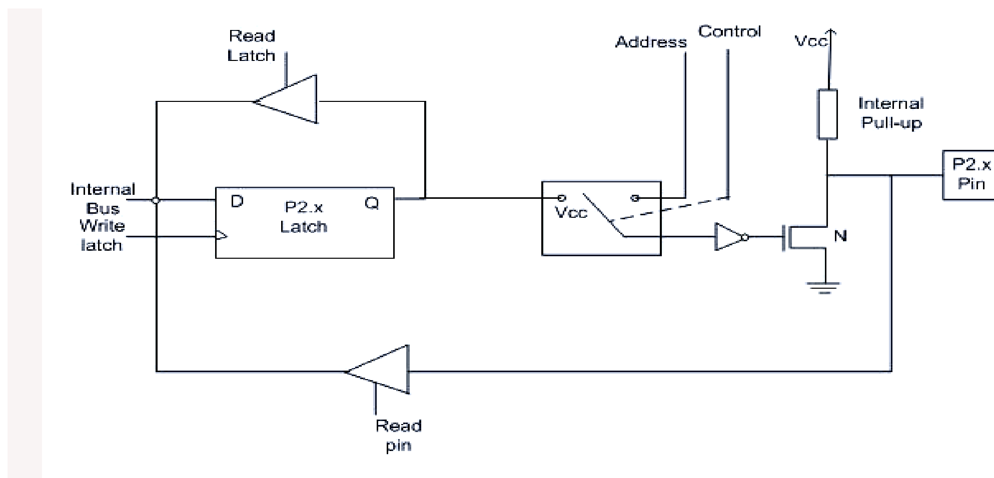


Fig: Structure of port 2 pin

4) PORT 3

Following are the alternate functions of PORT 3:

PORT 3 Pin	Function	Description

P3.0	RXD	Serial Input
P3.1	TXD	Serial Output
P3.2	INT0	External Interrupt 0
P3.3	INT1	External Interrupt 1
P3.4	T0	Timer 0
P3.5	T1	Timer 1
P3.6	WR	External Memory Write
P3.7	RD	External Memory Read

It works as an I/O port same like port 2. Alternate functions of port 3 makes its architecture different than other ports.

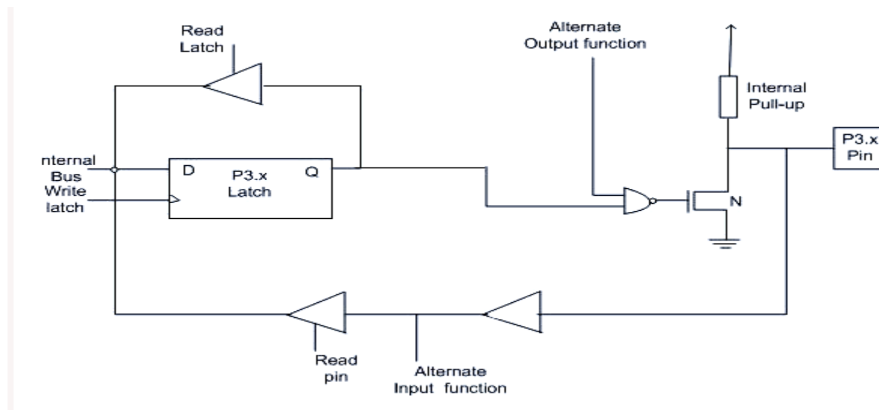


Fig: Structure of port 3 pin

UNIT 2

INSTRUCTION SYNTAX.

General syntax for 8051 assembly language is as follows.

LABEL: OPCODE OPERAND ;COMMENT

LABEL : (*THIS IS NOT NECESSARY UNLESS THAT SPECIFIC LINE HAS TO BE ADDRESSED*). The label is a symbolic address for the instruction. When the program is assembled, the label will be given specific address in which that instruction is stored. Unless that specific line of instruction is needed by a branching instruction in the program, it is not necessary to label that line.

OPCODE: Opcode is the symbolic representation of the operation. The assembler converts the opcode to a unique binary code (machine language).

OPERAND: While opcode specifies what operation to perform, operand specifies where to perform that action. The operand field generally contains the source and destination of the data. In some cases only source or destination will be available instead of both. The operand will be either address of the data, or data itself.

COMMENT: Always comment will begin with ***;*** *or //* symbol. To improve the program quality, programmer may always use comments in the program.

ADDRESSING MODES

Various methods of accessing the data are called addressing modes.

8051 addressing modes are classified as follows.

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.
5. Relative addressing.
6. Absolute addressing.
7. Long addressing.
8. Indexed addressing.
9. Bit inherent addressing.
10. Bit direct addressing.

1. *Immediate addressing.*

In this addressing mode the data is provided as a part of instruction itself. In other words data immediately follows the instruction.

Eg.
```
MOV A,#30H
ADD A, #83                    # Symbol indicates the data is immediate.
```

2. ***Register addressing.***

In this addressing mode the register will hold the data. One of the eight general registers (R0 to R7) can be used and specified as the operand.

Eg. MOV A,R0
ADD A,R6

R0 – R7 will be selected from the current selection of register bank. The default register bank will be bank 0.

3. ***Direct addressing***

There are two ways to access the internal memory. Using direct address and indirect address. Using direct addressing mode we can not only address the internal memory but SFRs also. In direct addressing, an 8 bit internal data memory address is specified as part of the instruction and hence, it can specify the address only in the range of 00H to FFH. In this addressing mode, data is obtained directly from the memory.

Eg. MOV A,60h
ADD A,30h

4. ***Indirect addressing***

The indirect addressing mode uses a register to hold the actual address that will be used in data movement. Registers R0 and R1 and DPTR are the only registers that can be used as data pointers. Indirect addressing cannot be used to refer to SFR registers. Both R0 and R1 can hold 8 bit address and DPTR can hold 16 bit address.

Eg. MOV A,@R0
ADD A,@R1
MOVX A,@DPTR

5. ***Indexed addressing.***

In indexed addressing, either the program counter (PC), or the data pointer (DTPR)—is used to hold the base address, and the A is used to hold the offset address. Adding the value of the base address to the value of the offset address forms the effective address. Indexed addressing is used with JMP or MOVC instructions. Look up tables are easily implemented with the help of index addressing.

Eg. MOVC A, @A+DPTR // *copies the contents of memory location pointed by the sum of the accumulator A and the DPTR into accumulator A.*
MOVC A, @A+PC // *copies the contents of memory location pointed by the sum of the accumulator A and the program counter into accumulator A.*

6. ***Relative Addressing.***

Relative addressing is used only with conditional jump instructions. The relative address, (offset), is an 8 bit signed number, which is automatically added to the PC to make the address of the next instruction. The 8 bit signed offset value gives an address range of +127 to —128 locations. The jump destination is usually specified using a label and the assembler calculates the jump offset accordingly. The advantage of relative addressing is that the program code is easy to relocate and the address is relative to position in the memory.

Eg: SJMP LOOP1
JC BACK

7. ***Absolute addressing***

Absolute addressing is used only by the AJMP (Absolute Jump) and ACALL (Absolute Call) instructions. These are 2 bytes instructions. The absolute addressing mode specifies the lowest 11 bit of the memory. The upper 5 bit of the destination address are the upper 5 bit of the current program counter. Hence, absolute addressing allows branching only within the current 2 Kbyte page of the program memory.

Eg. AJMP LOOP1

ACALL LOOP2

8. *Long Addressing*

The long addressing mode is used with the instructions LJMP and LCALL. These are 3 byte instructions. The address specifies a full 16 bit destination address so that a jump or a call can be made to a location within a 64 Kbyte code memory space.

Eg. LJMP FINISH
LCALL DELAY

9. *Bit Inherent Addressing*

In this addressing, the address of the flag which contains the operand, is implied in the opcode of the instruction.

Eg. CLR C ; *Clears the carry flag to 0*

10. *Bit Direct Addressing*

In this addressing mode the direct address of the bit is specified in the instruction. The RAM space 20H to 2FH and most of the special function registers are bit addressable. Bit address values are between 00H to 7FH.

Eg. CLR 07h ; *Clears the bit 7 of 20h RAM space*
SETB 07H ; *Sets the bit 7 of 20H RAM space.*

INSTRUCTION SET

1. *Instruction Timings*

The 8051 internal operations and external read/write operations are controlled by the oscillator clock. T-state, Machine cycle and Instruction cycle are terms used in instruction timings.

T-state is defined as one subdivision of the operation performed in one clock period. The terms 'T- state' and 'clock period' are often used synonymously.

Machine cycle is defined as 12 oscillator periods. A machine cycle consists of six states and each state lasts for two oscillator periods. An instruction takes one to four machine cycles to execute an instruction.

Instruction cycle is defined as the time required for completing the execution of an instruction. The 8051 instruction cycle consists of one to four machine cycles.

Eg. If 8051 microcontroller is operated with 12 MHz oscillator, find the execution time for the following four instructions.

1. *ADD A, 45H*
2. *SUBB A, #55H*
3. *MOV DPTR, #2000H*
4. *MUL AB*

Since the oscillator frequency is 12 MHz, the clock period is, Clock period = 1/12 MHz = ***0.08333 µS****. Time for 1 machine cycle = 0.08333 µS x 12 =* ***1 µS****.*

	Instruction	*No. of machine cycles*	*Execution time*
1.	*ADDA, 45H*	*1*	*1 µs*
2.	*SUBBA, #55H*	*2*	*2 µs*
3.	*MOV DPTR, #2000H*	*2*	*2 µs*
4.	*MULAB*	*4*	*4 µs*

2. *8051 Instructions*

The instructions of 8051 can be broadly classified under the following headings.

1. Data transfer instructions
2. Arithmetic instructions
3. Logical instructions
4. Branch instructions
5. Subroutine instructions
6. Bit manipulation instructions

Data transfer instructions.

In this group, the instructions perform data transfer operations of the following types.

a. Move the contents of a register Rn to A
  - *i.* MOV A,R2
  - *ii.* MOV A,R7

b. Move the contents of a register A to Rn
  - *i.* MOV R4,A
  - *ii.* MOV R1,A

c. Move an immediate 8 bit data to register A or to Rn or to a memory location(direct or indirect)
  - *i.* MOV A, #45H
  - *ii.* MOV R6, #51H
  - *iii.* MOV 30H, #44H
  - *iv.* MOV @R0, #0E8H
  - *v.* MOV DPTR, #0F5A2H
  - *vi.* MOV DPTR, #5467H

d. Move the contents of a memory location to A or A to a memory location using direct and indirect addressing
  - *i.* MOV A, 65H
  - *ii.* MOV A, @R0
  - *iii.* MOV 45H, A
  - *iv.* MOV @R1, A

e. Move the contents of a memory location to Rn or Rn to a memory location using direct addressing
  - *i.* MOV R3, 65H
  - *ii.* MOV 45H, R2

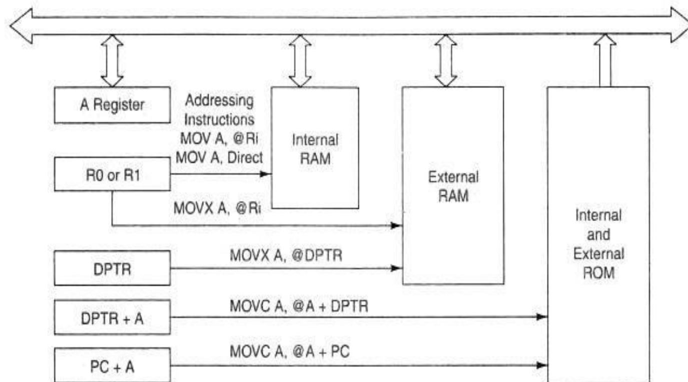
f. Move the contents of memory location to another memory location using direct and indirect addressing
  - *i.* MOV 47H, 65H
  - *ii.* MOV 45H, @R0

g. Move the contents of an external memory to A or A to an external memory

i. MOVX A,@R1
ii. MOVX @R0,A

h. Move the contents of program memory to A

i. MOVC A, @A+PC
ii. MOVC A, @A+DPTR



MOVX A,@DPTR

iii. MOVX@DPTR,A

Arithmetic instructions.

The 8051 can perform addition, subtraction. Multiplication and division operations on 8 bit numbers.

Addition

In this group, we have instructions to

i. Add the contents of A with immediate data with or without carry.
   i. ADD A, #45H
   ii. ADDC A, #OB4H

ii. Add the contents of A with register Rn with or without carry.
   i. ADD A, R5
   ii. ADDC A, R2

iii. Add the contents of A with contents of memory with or without carry using direct and indirect addressing
   i. ADD A, 51H
   ii. ADDC A, 75H
   iii. ADD A, @R1
   iv. ADDC A, @R0

CY AC and OV flags will be affected by this operation. Subtraction

In this group, we have instructions to

i. Subtract the contents of A with immediate data with or without carry.
   i. SUBB A, #45H
   ii. SUBB A, #OB4H

ii. Subtract the contents of A with register Rn with or without carry.

   i. SUBB A, R5
   ii. SUBB A, R2

iii. Subtract the contents of A with contents of memory with or without carry using direct and indirect addressing
   i. SUBB A, 51H
   ii. SUBB A, 75H
   iii. SUBB A, @R1
   iv. SUBB A, @R0

CY AC and OV flags will be affected by this operation. Multiplication

MUL AB. This instruction multiplies two 8 bit unsigned numbers which are stored in A and B register. After multiplication the lower byte of the result will be stored in accumulator and higher byte of result will be stored in B register.

```
Eg.     MOV A,#45H          ;[A]=45H
        MOV B,#0F5H         ;[B]=F5H
        MUL AB              ;[A] x [B] = 45 x F5 = 4209
                            ;[A]=09H, [B]=42H
```

Division

DIV AB. This instruction divides the 8 bit unsigned number which is stored in A by the 8 bit unsigned number which is stored in B register. After division the result will be stored in accumulator and remainder will be stored in B register.

```
Eg.   MOV A,#45H        ;[A]=0E8H
      MOV B,#0F5H       ;[B]=1BH
      DIV AB            ;[A] / [B] = E8 /1B = 08 H with remainder 10H
                        ;[A] = 08H, [B]=10H
```

DA A (Decimal Adjust After Addition).

When two BCD numbers are added, the answer is a non-BCD number. To get the result in BCD, we use DA A instruction after the addition. DA A works as follows.

- If lower nibble is greater than 9 or auxiliary carry is 1, 6 is added to lower nibble.
- If upper nibble is greater than 9 or carry is 1, 6 is added to upper nibble.

```
Eg 1:  MOV A,#23H
       MOV R1,#55H
       ADD A,R1      // [A]=78
       DA A          // [A]=78        no changes in the accumulator after da a

Eg 2:  MOV A,#53H
       MOV R1,#58H
       ADD A,R1      // [A]=ABh
       DA A          // [A]=11, C=1 . ANSWER IS 111. Accumulator data is changed after DA A
```

Increment: *increments the operand by one.*

INC A INC Rn INC DIRECT INC @RiINC DPTR

INC increments the value of source by 1. If the initial value of register is FFh, incrementing the value will cause it to reset to 0. The Carry Flag is not set when the value "rolls over" from 255 to 0.

In the case of "INC DPTR", the value two-byte unsigned integer value of DPTR is incremented. If the initial value of DPTR is FFFFh, incrementing the value will cause it to reset to 0.

Decrement: *decrements the operand by one.*

DEC A DEC Rn DEC DIRECT DEC @Ri

DEC decrements the value of *source* by 1. If the initial value of is 0, decrementing the value will cause it to reset to FFh. The Carry Flag is not set when the value "rolls over" from 0 to FFh.

Logical Instructions

Logical AND

ANL destination, source:ANL does a bitwise "AND" operation between *source* and *destination*, leaving the resulting value in *destination*. The value in source is not affected. "AND" instruction logically AND the bits of source and destination.
ANL A,#DATA ANL A, Rn
ANL A,DIRECT ANL
A,@Ri
ANL DIRECT,A ANL DIRECT, #DATA

Logical OR

ORL destination, source:ORL does a bitwise "OR" operation between *source* and *destination*,

leaving the resulting value in *destination*. The value in source is not affected. " OR " instruction logically OR the bits of source and destination.
ORL A,#DATA ORL A, Rn
ORL A,DIRECT ORL
A,@Ri
ORL DIRECT,A ORL DIRECT, #DATA

Logical Ex-OR

XRL destination, source: XRL does a bitwise "EX-OR" operation between *source* and *destination*, leaving the resulting value in *destination*. The value in source is not affected. " XRL " instruction logically EX-OR the bits of source and destination.
XRL A,#DATA XRL A,Rn
XRL A,DIRECT XRL
A,@Ri
XRL DIRECT,A XRL DIRECT, #DATA

Logical NOT

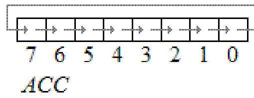
CPL complements *operand*, leaving the result in *operand*. If *operand* is a single bit then the state of the bit will be reversed. If *operand* is the Accumulator then all the bits in the Accumulator will be reversed.

CPL A, CPL C, CPL bit address

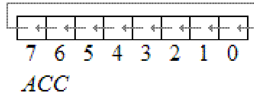
SWAP A – Swap the upper nibble and lower nibble of A.

Rotate Instructions

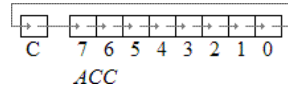
RR A
This instruction is rotate right the accumulator. Its operation is illustrated below. Each bit is shifted one location to the right, with bit 0 going to bit 7.



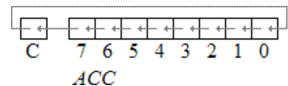
RL A
Rotate left the accumulator. Each bit is shifted one location to the left, with bit 7 going to bit 0



RRC A
Rotate right through the carry. Each bit is shifted one location to the right, with bit 0 going into the carry bit in the PSW, while the carry was at goes into bit 7



RLC A
Rotate left through the carry. Each bit is shifted one location to the left, with bit 7 going into the carry bit in the PSW, while the carry goes into bit 0.



Branch (JUMP) Instructions

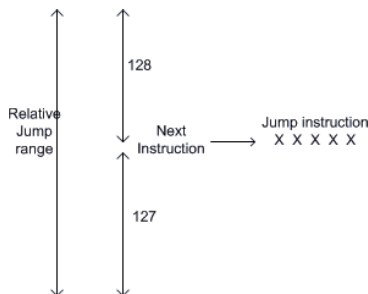
Jump and Call Program Range

There are 3 types of jump instructions. They are:-

1. Relative Jump
2. Short Absolute Jump
3. Long Absolute Jump

Relative Jump

Jump that replaces the PC (program counter) content with a new address that is greater than (the address following the jump instruction by 127 or less) or less than (the address following the jump by 128 or less) is called a relative jump. Schematically, the relative jump can be shown as follows: -



The advantages of the relative jump are as follows:-

1. Only 1 byte of jump address needs to be specified in the 2's complement form, ie. For jumping ahead, the range is 0 to 127 and for jumping back, the range is -1 to -128.
2. Specifying only one byte reduces the size of the instruction and speeds up program execution.
3. The program with relative jumps can be relocated without reassembling to generate absolute jump addresses.

Disadvantages of the absolute jump: -

1. Short jump range (-128 to 127 from the instruction following the jump instruction)

Instructions that use Relative Jump

SJMP <relative address>; *this is unconditional jump*

The remaining relative jumps are conditional jumps

JC <relative address>
JNC <relative address>
JB bit, <relative address>
JNB bit, <relative address>
JBC bit, <relative address>
CJNE <destination byte>, <source byte>, <relative address>
DJNZ <byte>, <relative address>
JZ <relative address>
JNZ <relative address>

Short Absolute Jump

In this case only 11bits of the absolute jump address are needed. The absolute jump address is calculated in the following manner.

In 8051, 64 kbyte of program memory space is divided into 32 pages of 2 kbyte each. The hexadecimal addresses of the pages are given as follows:-

Page (Hex)	*Address (Hex)*
00	*0000 - 07FF*
01	*0800 - 0FFF*
02	*1000 - 17FF*
03	*1800 - 1FFF*
.	
.	
1E	*F000 - F7FF*
1F	*F800 - FFFF*

It can be seen that the upper 5bits of the program counter (PC) hold the page number and the lower 11bits of the PC hold the address within that page. Thus, an absolute address is formed by taking page numbers of the instruction (from the program counter) following the jump and attaching the specified 11bits to it to form the 16-bit address.

Advantage: The instruction length becomes 2 bytes.

Example of short absolute jump: -

```
ACALL <address 11>
AJMP   <address 11>
```

Long Absolute Jump/Call

Applications that need to access the entire program memory from 0000H to FFFFH use long absolute jump. Since the absolute address has to be specified in the op-code, the instruction length is 3 bytes (except for JMP @ A+DPTR). This jump is not re-locatable.

Example: -

```
LCALL <address 16>
LJMP <address 16>
JMP @A+DPTR
```

Another classification of jump instructions is

1. Unconditional Jump
2. Conditional Jump

1. **The unconditional jump** is a jump in which control is transferred unconditionally to the target location.
    a. **LJMP** (long jump). This is a 3-byte instruction. First byte is the op-code and second and third bytes represent the 16-bit target address which is any memory location from 0000 to FFFFH
    eg: LJMP 3000H
    b. **AJMP:** this causes unconditional branch to the indicated address, by loading the 11 bit address to 0 -10 bits of the program counter. The destination must be therefore within the same 2K blocks.
    c. **SJMP** (short jump). This is a 2-byte instruction. First byte is the op-code and second byte is the relative target address, 00 to FFH (forward +127 and backward -128 bytes from the current PC value). To calculate the target address of a short jump, the second byte is added to the PC value which is address of the instruction immediately below the jump.

2. Conditional Jump instructions.

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

All conditional jumps are short jumps.

Bit level jump instructions:

Bit level JUMP instructions will check the conditions of the bit and if condition is true, it jumps to the address specified in the instruction. All the bit jumps are relative jumps.

JB bit, rel ; jump if the direct bit is set to the relative address specified. JNB
bit, rel ; jump if the direct bit is clear to the relative address specified.
JBC bit, rel ; jump if the direct bit is set to the relative address specified and then clear the bit.

Subroutine CALL And RETURN Instructions

Subroutines are handled by CALL and RET instructions There

are two types of CALL instructions

1. **LCALL address(16 bit)**
   This is long call instruction which unconditionally calls the subroutine located at the indicated 16 bit address. This is a 3 byte instruction. The LCALL instruction works as follows.
   a. During execution of LCALL, [PC] = [PC]+3; (if address where LCALL resides is say, 0x3254; during execution of this instruction [PC] = 3254h + 3h = 3257h
   b. [SP]=[SP]+1; (if SP contains default value 07, then SP increments and [SP]=08
   c. [[SP]] = [PC_{7-0}]; (lower byte of PC content ie., 57 will be stored in memory location 08.
   d. [SP]=[SP]+1; (SP increments again and [SP]=09)
   e. [[SP]] = [PC_{15-8}]; (higher byte of PC content ie., 32 will be stored in memory location 09.
   With these the address (0x3254) which was in PC is stored in stack.
   f. [PC]= address (16 bit); the new address of subroutine is loaded to PC. No flags are affected.

2. **ACALL address(11 bit)**
   This is absolute call instruction which unconditionally calls the subroutine located at the indicated 11 bit address. This is a 2 byte instruction. The SCALL instruction works as follows.
   a. During execution of SCALL, [PC] = [PC]+2; (if address where LCALL resides is say, 0x8549; during execution of this instruction [PC] = 8549h + 2h = 854Bh
   b. [SP]=[SP]+1; (if SP contains default value 07, then SP increments and [SP]=08
   c. [[SP]] = [PC_{7-0}]; (lower byte of PC content ie., 4B will be stored in memory location 08.
   d. [SP]=[SP]+1; (SP increments again and [SP]=09)
   e. [[SP]] = [PC_{15-8}]; (higher byte of PC content ie., 85 will be stored in memory location 09.

   With these the address (0x854B) which was in PC is stored in stack.

f. $[PC_{10-0}]$= address (11 bit); the new address of subroutine is loaded to PC. No flags are affected.

RET instruction

RET instruction pops top two contents from the stack and load it to PC.

g. $[PC_{15-8}]$ = [[SP]];content of current top of the stack will be moved to higher byte of PC.
h. [SP]=[SP]-1; (SP decrements)
i. $[PC_{7-0}]$ = [[SP]] ;content of bottom of the stack will be moved to lower byte of PC.
j. [SP]=[SP]-1; (SP decrements again)

Bit manipulation instructions.

8051 has 128 bit addressable memory. Bit addressable SFRs and bit addressable PORT pins. It is possible to perform following bit wise operations for these bit addressable locations.

1. LOGICAL AND
   a. ANL C,BIT(BIT ADDRESS) ; 'LOGICALLY AND' CARRY AND CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY
   b. ANL C, /BIT; ; 'LOGICALLY AND' CARRY AND COMPLEMENT OF CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY

2. LOGICAL OR
   a. ORL C,BIT(BIT ADDRESS) ; 'LOGICALLY OR' CARRY AND CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY
   b. ORL C, /BIT; ; 'LOGICALLY OR' CARRY AND COMPLEMENT OF CONTENT OF BIT ADDRESS, STORE RESULT IN CARRY
3. CLR bit
   a. CLR bit ; CONTENT OF BIT ADDRESS SPECIFIED WILL BE CLEARED.
   b. CLR C ; CONTENT OF CARRY WILL BE CLEARED.
4. CPL bit
   a. CPL bit ; CONTENT OF BIT ADDRESS SPECIFIED WILL BE COMPLEMENTED.
   b. CPL C ; CONTENT OF CARRY WILL BE COMPLEMENTEd

ASSEMBLER DIRECTIVES

Assembler directives tell the assembler to do something other than creating the machine code for an instruction. In assembly language programming, the assembler directives instruct the assembler to

1. Process subsequent assembly language instructions
2. Define program constants
3. Reserve space for variables

The following are the widely used 8051 assembler directives.

ORG (origin)

The ORG directive is used to indicate the starting address. It can be used only when the program counter needs to be changed. The number that comes after ORG can be either in hex or in decimal.

Eg: ORG 0000H ;Set PC to 0000

.EQU and SET

EQU and SET directives assign numerical value or register name to the specified symbol name.

EQU is used to define a constant without storing information in the memory. The symbol defined with EQU should not be redefined.

SET directive allows redefinition of symbols at a later stage.

DB (DEFINE BYTE)

The DB directive is used to define an 8 bit data. DB directive initializes memory with 8 bit values. The numbers can be in decimal, binary, hex or in ASCII formats. For decimal, the 'D' after the decimal number is optional, but for binary and hexadecimal, 'B' and ‘H’ are required. For ASCII, the number is written in quotation marks (‘LIKE This).

```
DATA1:  DB   40H            ; hex
DATA2:  DB   01011100B      ; b i n a r y
DATA3:  DB   48             ; decimal
DATA4:  DB   ' HELLO W’     ; ASCII
```

END

The END directive signals the end of the assembly module. It indicates the end of the program to the assembler. Any text in the assembly file that appears after the END directive is ignored. If the END statement is missing, the assembler will generate an error message.

ASSEMBLY LANGUAGE PROGRAMS

1. **Write a program to add the values of locations 50H and 51H and store the result in locations in 52h and 53H.**

```
ORG 0000H          ; Set program counter 0000H
MOV A,50H          ; Load the contents of Memory location 50H into A ADD ADD A,51H
        ; Add the contents of memory 51H with CONTENTS A
MOV 52H,A          ; Save the LS byte of the result in 52H
MOV A, #00         ; Load 00H into A
ADDC A, #00        ; Add the immediate data and carry to A
MOV 53H,A          ; Save the MS byte of the result in location 53h
END
```

2. **Write a program to store data FFH into RAM memory locations 50H to 58H using direct addressing mode**

```
ORG 0000H          ; Set program counter 0000H
MOV A, #0FFH ; Load FFH into A
MOV 50H, A         ; Store contents of A in location 50H
MOV 51H, A         ; Store contents of A in location 5IH
MOV 52H, A         ; Store contents of A in location 52H
MOV 53H, A         ; Store contents of A in location 53H
MOV 54H, A         ; Store contents of A in location 54H
MOV 55H, A         ; Store contents of A in location 55H
MOV 56H, A         ; Store contents of A in location 56H
MOV 57H, A         ; Store contents of A in location 57H
MOV 58H, A         ; Store contents of A in location 58H
END
```

3. **Write a program to subtract a 16 bit number stored at locations 51H-52H from 55H-56H and store the result in locations 40H and 41H. Assume that the least significant byte of data or the result is stored in low address. If the result is positive, then store 00H, else store 01H in 42H.** ORG 0000H

```
; Set program counter 0000H
MOV A, 55H     ; Load the contents of memory location 55 into A
CLR C          ; Clear the borrow flag
SUBB A,51H     ; Sub the contents of memory 51H from contents of A
MOV 40H, A     ; Save the LSByte of the result in location 40H
MOV A, 56H     ; Load the contents of memory location 56H into A
SUBB A, 52H    ; Subtract the content of memory 52H from the content A
MOV 41H,       ; Save the MSbyte of the result in location 415.
MOV A, #00     ; Load 005 into A
ADDC A, #00    ; Add the immediate data and the carry flag to A MOV
42H, A         ; If result is positive, store00H, else store 0lH in 42H
END
```

4. Write a program to add two 16 bit numbers stored at locations 51H-52H and 55H-56H and store the result in locations 40H, 41H and 42H. Assume that the least significant byte of data and the result is stored in low address and the most significant byte of data or the result is stored in high address.

```
ORG 0000H          ; Set program counter 0000H
MOV A,51H          ; Load the contents of memory location 51H into A
ADD A,55H          ; Add the contents of 55H with contents of A
MOV 40H,A          ; Save the LS byte of the result in location 40H
MOV A,52H          ; Load the contents of 52H into A
ADDC A,56H         ; Add the contents of 56H and CY flag with A
MOV 41H,A          ; Save the second byte of the result in 41H
MOV A,#00          ; Load 00H into A
ADDC A,#00 ; Add the immediate data 00H and CY to A
MOV 42H,A          ; Save the MS byte of the result in location 42H
END
```

5. Write a program to store data FFH into RAM memory locations 50H to 58H using indirect addressing mode.

```
      ORG 0000H          ; Set program counter 0000H
      MOV A, #0FFH       ; Load FFH into A
      MOV RO, #50H       ; Load pointer, R0-50H
      MOV R5, #08H       ; Load counter, R5-08H
Start:MOV @RO, A         ; Copy contents of A to RAM pointed by R0
      INC RO             ; Increment pointer
      DJNZ R5, start ; Repeat until R5 is zero
      END
```

6. Write a program to add two Binary Coded Decimal (BCD) numbers stored at locations 60H and 61H and store the result in BCD at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H     ; Set program counter 00004
MOV A,60H     ; Load the contents of memory location 60H into A
ADD A,61H     ; Add the contents of memory location 61H with contents of A
DA A          ; Decimal adjustment of the sum in A
MOV 52H, A    ; Save the least significant byte of the result in location 52H
MOV A,#00     ; Load 00H into .A
ADDC A,#00H   ; Add the immediate data and the contents of carry flag to A
MOV 53H,A     ; Save the most significant byte of the result in location 53:,
END
```

7. Write a program to clear 10 RAM locations starting at RAM address 1000H.

```
       ORG 0000H           ;Set program counter 0000H
       MOV DPTR, #1000H    ;Copy address 1000H to DPTR
       CLR A        ;Clear A
       MOV R6, #0AH        ;Load 0AH to R6
again: MOVX @DPTR,A        ;Clear RAM location pointed by DPTR
```

```
INC DPTR            ;Increment DPTR
DJNZ R6, again      ;Loop until counter R6=0
END
```

8. Write a program to compute 1 + 2 + 3 + N (say N=15) and save the sum at70H

```
        ORG 0000H                ; Set program counter 0000H
        N EQU 15
        MOV R0,#00               ; Clear R0
        CLR A                    ; Clear A
again:  INC R0                   ; Increment R0
        ADD A, R0                ; Add the contents of R0 with A
        CJNE R0,# N, again ; Loop until counter, R0, N
        MOV 70H,A                ; Save the result in location 70H END
```

9. Write a program to multiply two 8 bit numbers stored at locations 70H and 71H and store the result at memory locations 52H and 53H. Assume that the least significant byte of the result is stored in low address.

```
ORG 0000H ; Set program counter 00 OH
MOV A, 70H ; Load the contents of memory location 70h into A
MOV B, 71H ; Load the contents of memory location 71H into B
MUL AB     ; Perform multiplication
MOV 52H,A ; Save the least significant byte of the result in location 52H MOV 53H,B ; Save the most significant byte of the result in location 53
END
```

10. Ten 8 bit numbers are stored in internal data memory from location 5oH. Write a program to increment the data.

Assume that ten 8 bit numbers are stored in internal data memory from location 50H, hence R0 or R1 must be used as a pointer.

The program is as follows.

```
OPT 0000H
MOV R0,#50H
MOV R3,#0AH
Loopl: INC @R0
INC RO
DJNZ R3, loopl END
END
```

11. Write a program to find the average of five 8 bit numbers. Store the result in H. (Assume that after adding five 8 bit numbers, the result is 8 bit only).

```
ORG 0000H
MOV 40H,#05H
MOV 41H,#55H
MOV 42H,#06H
MOV 43H,#1AH
MOV 44H,#09H
MOV R0,#40H
MOV R5,#05H
MOV B,R5
CLR A
Loop: ADD A,@RO
INC RO
```

```
DJNZ R5,Loop
DIV AB
MOV 55H,A
END
```

12. Write a program to find the cube of an 8 bit number program is as follows

```
ORG 0000H
MOV
R1,#N
MOV A,R1
MOV B,R1
MUL AB                  //SQUARE IS
COMPUTED MOV R2, B
MOV B, R1
MUL AB
MOV 50,A
MOV 51,B
MOV A,R2
MOV B, R1
MUL AB
ADD A, 51H
MOV 51H,
A MOV 52H,
B MOV A, #
00H ADDC
A, 52H
MOV 52H, A              //CUBE IS STORED IN
52H,51H,50H END
```

13. Write a program to exchange the lower nibble of data present in external memory 6000H and 6001H

```
ORG 0000H               ; Set program counter 00h
MOV DPTR, # 6000 H ; Copy address 6000 H to DPTR
MOVX A, @DPTR       ; Copy contents of 60008 to A
MOV R0, #45H        ; Load pointer, R0=45H
MOV @RO, A          ; Copy cont of A to RAM pointed by 80
INC DPL             ; Increment pointer
MOVX A, @DPTR       ; Copy contents of 60018 to A
XCHD A, @R0         ; Exchange lower nibble of A with RAM pointed by RO
MOVX @DPTR, A       ; Copy contents of A to 60018
DEC DPL             ; Decrement pointer
MOV A, @R0          ; Copy cont of  RAM pointed by R0 to A
MOVX @DPTR, A       ; Copy cont of A to RAM pointed by DPTR
END
```

14. Write a program to count the number of and o's of 8 bit data stored in location 6000H.

```
ORG 00008               ; Set program counter 00008
MOV DPTR, #6000h    ; Copy address 6000H to DPTR
MOVX A, @DPTR           ; Copy num be r to A
MOV R0,#08              ; Copy  08 in RO
MOV R2,#00              ; Copy 00 in R2
MOV R3,#00              ; Copy 00 in R3
CLR C                   ; Clear carry flag
BACK: RLC A         ; Rotate A through carry flag
```

```
JC NEXT                 ;IfCF=1,branchtonext
INC R2          ;IfCF=0,incrementR2 AJMP
NEXT2 NEXT:  INC R3         ;IfCF=1,incrementR3
NEXT2: DJNZ RO,BACK     ;RepeatuntilROiszero
END
```

15. Write a program to shift a 24 bit number stored at 57H-55H to the left logically four places. Assume that the least significant byte of data is stored in lower address.

```
        ORG 0000H      ; Set program counter 0000h
        MOV R1,#04     ; Set up loop count to 4
again:  MOV A,55H      ; Place the least significant byte of data in A
        CLR C          ; Clear tne carry flag
        RLC A          ; Rotate contents of A (55h) left through carry
        MOV 55H,A
        MOV A,56H
        RLC A          ; Rotate contents of A (56H) left through carry
        MOV 56H,A
        MOV A,57H
        RLC A          ; Rotate contents of A    (57H) left through carry
        MOV 57H,A
        DJNZ R1,again ; Repeat until R1 is zero
        END
```

16. Two 8 bit numbers are stored in location 1000h and 1001h of external data memory. Write a program to find the GCD of the numbers and store the result in 2000h. ALGORITHM

- *Step 1 :Initialize external data memory with data and DPTR with address*
- *Step 2 :Load A and TEMP with the operands*
- *Step 3 :Are the two operands equal? If yes, go to step 9*
- *Step 4 :Is (A) greater than (TEMP) ? If yes, go to step 6*
- *Step 5 :Exchange (A) with (TEMP) such that A contains the bigger number*
- *Step 6 :Perform division operation (contents of A with contents of TEMP)*
- *Step 7 :If the remainder is zero, go to step 9*
- *Step 8 :Move the remainder into A and go to step 4*
- *Step 9 :Save the contents 'of TEMP in memory and terminate the program*

```
        ORG 0000H              ; Set program counter 0000H
        TEMP EQU 70H
        TEMPI EQU 71H
        MOV DPTR, #1000H       ; Copy address 100011 to DPTR
        MOVX A, @DPTR          ; Copy First number to A
        MOV TEMP, A            ; Copy First number to temp INC DPTR
        MOVX A, @DPTR          ; Copy Second number to A
LOOPS: CJNE A, TEMP, LOOP1 ; (A) /= (TEMP) branch to LOOP1
        AJMP LOOP2                   ; (A) = (TEMP) branch to L00P2
LOOP1:  JNC LOOP3                    ; (A) > (TEMP) branch to LOOP3
        NOV TEMPI, A           ; (A) < (TEMP) exchange (A) with (TEMP)
        MOV A, TEMP
        MOV TEMP, TEMPI
LOOP3:  MOV B, TEMP
        DIV AB                 ; Divide (A) by (TEMP)
        MOV A, B                     ; Move remainder to A
        CJNE A,#00, LOOPS      ; (A)/=00 branch to LOOPS
LOOP2:  MOV A, TEMP
        MOV DPTR, #2000H
        MOVX @DPTR, A                ; Store the result in 2000H
        END
```

UNIT- 3

REAL TIME CONTROL –INTERRUPTS

3.1 INTERRUPT HANDLING STRUCTURE

- IDENTIFICATION OF INTERRUPT SOURCE ON INTERRUPT
- ENABLE/MASK
  - Primary Level
  - Secondary Level
- FINDING VECTOR ADDRESS FOR CPU TO VECTOR TO ISR
  - Vector to ISR –ADDR
  - Vector to pointer bytes of ISR
- PRIORITY ASSIGNMENT
  - Default priority on reset
  - User assignment to override default
    - 1 or 0
    - Between O & P

3.2 ROUTINE

- A SUBPROGRAM OR PROCEDURE OR SUB ROUTINE IMPLEMENTED BY CALL AND RETURN

3.3 INTERRUPT SOURCE

- EXTERNAL/INTERNAL EVENT FROM DEVICE OR CIRCUIT CAUSING AN INTERRUPT
- UNPLANNED W.R.T INSTRUCTION EXECUTION

3.4 INTERRUPT SERVICE ROUTINE

- EXECUTED IN RESPONSE TO AN INTERRUPT
- ONE ISR FOR EACH SOURCE OF INTERRUPT
- ISR, SAVES PARAMETERS (PSW, REGS, SFR's etc) BEFORE EXECUTING ACTUAL PROGRAM
- AT THE END OF THE ISR RETURN FROM INTERRUPT INSTRUCTIONS IS EXECUTED (RETI FOR 8051)
- BEFORE RETI PARAMETERS SAVED EARLIER SHOULD BE RETRIEVED
- RETI DOES FOLLOWING:
  - RESTORES PC
  - RESTORE INTERRUPT STRUCTURE

- CHECK PENDING INTERRUPTS. IF YES INITIATE ANOTHER ISR. IF NO, GO BACK TO MAIN PROGRAM

3.5.1 SERVICING 8051 INTERRUPTS

- COMPLETE CURRENT INSTRUCTION. CHECK IF EA IS 1 (PRIMARY LEVEL ENABLE)
- CHECK SOURCE FOR UNMASKED INTERRUPTS AND DO THE FOLLOWING
  - SAVE PC
  - RESET EA MOMENTARILY (UNTIL EXECUTION OF FIRST INSTRUCTIONS) AND SETS AGAIN, AT ISR AUTOMATICALLY
  - PC IS SET AS PER VECTOR ADDRESS OF INTERRUPT
  - GO TO ISR
  - DURING ISR, HIGHER PRIORITY INTERRUPT CAN INTERRUPT ISR, IF EA IS NOT MADE 0 BY PROGRAMMER
  - LOWER PRIORITY INTERRUPTS CHECKED AFTER RETI

3.5.2 IDENTIFICATION OF INTERRUPT SOURCE

- INTERRUPT SOURCE IS IDENTIFIED BY CPU BY LOOKING AT RESPECTIVE FLAGS IN INTERRUPT STATUS REGISTER OR INTERRUPT PENDING

REGISTER

- IN 8051 TCON AND SCON HAVE FLAGS
- IN SCON TI or RI WILL INDICATE INTERRUPT FLAG
- FLAGS SHOULD BE RESET AS A PART OF ISR, WHEREVER REQUIRED
- IN 8051 TI/RI SHOULD BE RESET IN ISR
- SOME FLAGS GET RESET AUTOMATICALLY ON TRANSFER OF CONTROL TO ISR LIKE TIMER FLAGS AND EXTERNAL/INTERNAL FLAGS OF 8051

3.5.3 ADDRESS OF ISR

THREE METHODS

- DIRECT ADDRESS ACCESS
  - Interrupt source provides address. It is done during interrupt acknowledge cycle. Example 8085

- VECTOR ADDRESS AS ISR ADDRESS
  - Interrupt source auto generates vector address.
    - Example –
      - 8051 interrupts
      - 8085 (Trap, RST 5.5, 6.5 & 7.5)
  - ISR can be at vector address itself or you can have a jump instructions

  to a memory where ISR is available

- ISR address separated by 8 bytes

- VECTOR ADDRESS AS POINTER TO ISR ADDRESS

  - Interrupt source provides to CPU a vector address pointer. From this pointed memory, ISR address is picked up
  - Interrupting device (external) can put vector address pointer on data bus during INTACK cycle
    - Example:
      - 68HCII/12

      80X86 (Here interrupt type number is put and address pointer is calculated by multiplying it by 4)
      - 8259 putting vector address pointer on data bus (In case of 8086 type number is sent as mentioned above)
  - ISR address pointers separated by four bytes
  - Example of programming for serial port:
    - Main Program
      - SETB ES
      - MOV SCON,#0C0H
      - MOV R0,#60H
    - ISR
      - ORG 23H
      - DJNZ R0,SEND
      - RETI
      - SEND: MOV SBUF,@R0
      - CLR Ti
      - RETI

3.6 INTERRUPT LATENCY AND INTERRUPT DEAD LINE

- INTERRUPT LATENCY (T_d) IS THE INTERVAL BETWEEN INTERRUPT EVENT AND START OF ISR FOR THAT EVENT
- INTERRUPT DEAD LINE (T_{lat})IS THE TIME BY WHICH INTERRUPT SHOULD BE SERVICED ELSE INFORMATION IS LOST \

  - Example:
    - Latency of serial port interrupt should be less than 11/9600 second of interrupts are occurring every 11/9600 second

$T_d < T_{lat}$

3.7 MULTIPLE SOURCES OF INTERRUPTS

- INTERNAL (TO overflow in 8051)
- EXTERNAL (INTO in 8051)
- HARDWARE
- SOFTWARE

- Hardware interrupts related to internal devices
  - INTERNAL TIMER OVERFLOWS (ALL MCs)
  - INPUT CAPTURE INTERRUPT (68HC11/12)
  - OUTPUT COMPARE INTERRUPT (68HC11/12)
  - INTERRUPT AT END OF SERIAL TRANSMISSION (8051,68HC11,80196)
  - INTERRUPT AT END OF SERIAL RECEPTION (8051,68HC11,80196)
  - INTERRUPT WHEN SERIAL RECEIVE BUFFER (FIFO) IS HALF FULL OR FULL (80196)
  - INTERRUPT ON START OF ANALOG TO DIGITAL CONVERSION OR AT END OF CONVERSION

- Hardware interrupts related to external sources or peripheral
  - INTERRUPTS OCCURRING AT PINS OF MICRO CONTROLLER OR MICRO PROCESSOR
    - INTR or NMI in 8086
    - INT0 or INT1 in 8051
    - RST 5.5, RST 6.5, RST 7.5 in 8085
  - IN SUCH INTERRUPTS ADDRESSES OF ISRs WILL BE ANY ONE OF THE 3 METHODS DESCRIBED IN 3.5.3

- Software related interrupts
  - SW Instructions
    - SWI INSTRUCTION IN 68HC11
    - RST1, RST2 etc in 8085
    - INT N in 8086
    - INT 3 in 8086 (Breakpoint interrupt)
    - INT O (Overflow interrupt which is Type 4)
  - ERRORS
    - Division by 0
    - Illegal opcode
    - Overflow

3.8 NON MASKABLE INTERRUPT SOURCE (NMI)

- INTERRUPTS CAN BE MASKED, BUT THEY ARE CERTAIN INTERRUPTS WHICH CANNOT BE MASKED. THEY HAVE TO BE SERVICED IMMEDIATELY
- NORMALLY NMIs WILL BE EMERGENCY CONDITIONS
- 8051 DOES NOT HAVE ANY NMIs
- 8086 HAS ONE NMI. IT COULD BE CONNECTED TO POWER FAILURE CONDITION OR DRAM PARITY ERROR DETECTION

- 68HC11 HAS CLOCK MONITOR FAILURE AN ILLEGAL (UNIMPLEMENTED) INSTRUCTIONS AS NMIs

3.9 ENABLING (Unmasking) /DISABLING (Masking) OF INTERRUPT SOURCE

- INTERRUPTS CAN BE DISABLED OR ENABLED
- TWO LEVELS OF INTERRUPT ENABLING
- PRIMARY LEVEL/GLOBAL LEVEL. THIS SHOULD BE SAID TO ENABLE ANY INTERRUPT. IF RESET ALL INTERRUPTS WILL BE DISABLED – For Ex: DURING CRITICAL REGION OF PROGRAM ONE COULD LIKE TO DISABLE

ALL INTERRUPTS

- ENABLE INTERRUPTS STRUCTURE (GLOBAL LEVEL) BUT DISABLE INDIVIDUAL INTERRUPT SOURCES OR GROUP OF SOURCES
- EXAMPLES OF 8051
    - IE.7 BIT is global enable
    - IE.6 – IE.0 are used for selective enabling of interrupt sources
    - INSTRUCTIONS
        - SETB IE.7: Enable Global Level. Enable all interrupt sources
        - CLR ES : Disable or Mask serial port interrupt
        - MOV IE,#83H : Enable global level, Timer 0 and external interrupt 0

- EXAMPLES OF ISR FOR SERIAL PORT TRANSMIT INTERRUPT
    - ORG 0023H: Vector address for serial port interrupt
    - LJMP 400H: ISR is at 400H
    - ORG 400H:
    - ISR: CLR EA: Disable global level interrupt
    - MOV SBUF,#41H Put character in SBUF
    - CLR TI Resets TI Flag
    - CLR ES Disable serial port interrupt
    - SETB EA Enable global level interrupt
    - RETI Return from Interrupt

3.10 POLLING TO DETERMINE THE INTERRUPT SOURCES AND ASSIGNMENTS OF THE PRIORITIES AMONG THEM

- SEVERAL INTERRUPTS MAY OCCUR IN QUICK SUCCESSIONS
    - Several interrupts occurring at the same time

    (during execution time of a set of instructions)
    - Interrupts occurring during an ISR
- INTERRUPT STRUCTURE SHOULD HAVE THE FOLLOWING FEATURES TO TAKE CARE OF ABOVE SITUATION
    - Multiple sources should have default priorities (8051,80960 etc)

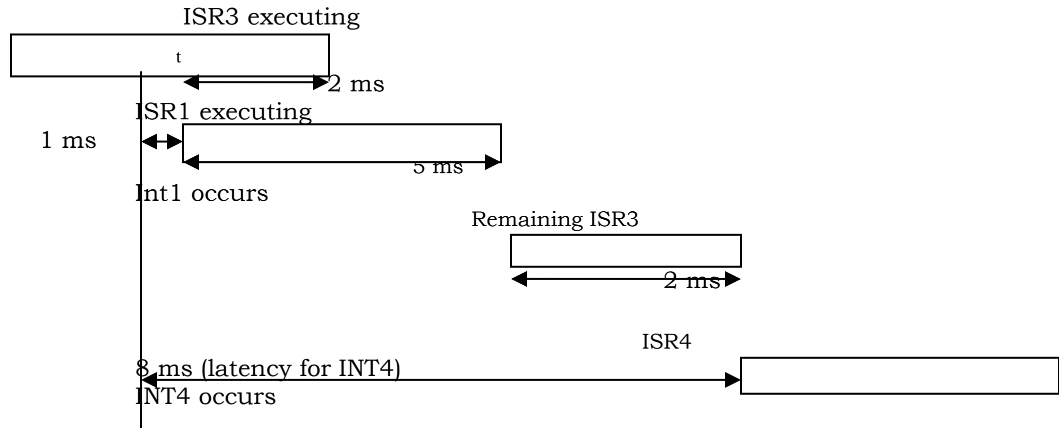
- Each of the multiple interrupt sources should have user assigned priorities (8051 – 0 OR 1) (80196 – 0-31)
- User assigned priorities over ride default priorities
- User can change priorities depending on need
- EXAMPLE
  - Let serial data is being received by 8051 serial port at 9600 baud. In this case interrupts will occur every 10 or 11/9600 second. Hence interrupt should be every 11/9600 else data will be lost. Compare this with A/D conversion interrupt. This can be serviced every 1 sec. Hence between the two, serial port will be assigned higher priority
- POLLING PROCEDURES
  - Two Polling Procedures
    - In 8051 family polling for higher priority takes place during ISR
    - IN 68HC11 family polling is at end of ISR
  - 8051
    - Polling during ISR, at end of each instruction (except during initial instruction and critical region)
    - User assigned priority
    - Default assigned priority if multiple interrupts have same user priority
    - Vectored priority method, with diversion to higher ISR, during an ISR
  - 68HC11
    - No polling during ISR
    - Polling at end of ISR
    - Polling of user assigned priorities is done first
    - Polling of default priority next
    - Vectored priority method with no diversion during ISR

3.11 ADVANTAGE OF POLLING FOR A PENDING HIGHER PRIORITY INTERRUPT DURING AN ISR

- EXAMPLE
  - Let us take four interrupt sources INT1, INT2, INT3, INT4. Services are ISR1, ISR2, ISR3, ISR4. ISR1 has highest priority.
    - ISR3 under service. Execution Time of 2ms left at t
    - At t4 ISR4 is pending since 1ms
    - INT1 occurs at t – Its execution time is 5 ms

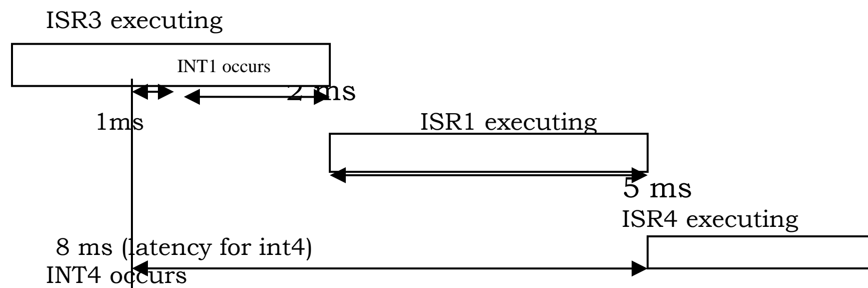
    Therefore latency for INT4 = 1+2+5 = 8 ms
    Latency for INT1 = 0
    (Neglecting context switching)
    - INT3 restarts after 5ms takes 7ms to complete

- CONCLUSION
  - ISR1 done immediately
  - ISR4 is affected badly. It can be serviced earlier by user assigned higher priority
  - DIAGRAM TO FOLLOW



3.12 ADVANTAGE OF POLLING A PENDING HIGHER PRIORITY INTERNAL SOURCE AT END OF AN ISR

- EXAMPLE
  - Same as in 3.11 – but latency times will be different as shown



  - Here ISR1is executed only after ISR3 is completed. Hence latency time is 2ms. This is due to pre-emption allowed at end of ISR
  - Latency of INT4 is still 8ms. To reduce it ISR has to be disabled. Then latency for INT4 becomes 3 ms.
  - Latency is predictable as an ISR is not disturbed. Hence

completion of ISRs are not known

- Disabling interrupt during ISR in earlier example makes it same
  as this example

3.13 INTERRUPT STRUCTURE IN 8051

- Seven interrupt source groups
- EA BIT Can disable interrupts from all source groups
- If EA has enabled the interrupt structure then that source group is recognized and serviced whose enable bit is set in IE register
- Every source group has a unique vector address
- Each source group may have one or two interrupt sources. For example: GROUP 1 has only one i.e. INTO (external interrupt) where as GROUP 7 has
  two sources i.e. TI (Transmitter interrupt and receiver interrupt)
- Once CPU recognizes interrupt, EA is temporarily disabled till the first instructions of ISR, so that PC is properly saved on the stack before any other interrupt is given to CPU
- There are totally seven source groups as there are seven vector addresses
- At the end of each instruction polling is done for pending, identified and enabled interrupt sources of priority greater than the one presently serviced by ISR
- Each source group can be assigned a low priority or high priority (only two priority levels)
- Polling is for user assigned priority first and then default priority
- Priorities are as below:
  - Group 1 INT0 Highest (0003h)
  - Group 2 T2 (8052 only) (002bh)
  - Group 3 SI, synchronous port interface (select members of 8051 family)
  - Group 4 TF0 (000BH) (0053H)
  - Group 5 INT1 (0013H)
  - Group 6 TF1 (001BH)
  - Group 7 SI Serial port (0023H)

UNIT 4

REAL TIME CONTROL TIMERS

4.1 PROGRAMMABLE FEATURES OF TIMERS

- Start/Stop
  - Ex 1: In 8051 set TRO=1 for starting timer 0
  - Ex 2: In 68HC11/12 it is not programmable as timer is free running
- Programming Pre-Scaler
  - Ex 1: In 8051 it is not programmable, It is fixed as 12
  - Ex 2: In 8096 it is not programmable, it is fixed at 8
  - In 68HC11/68HC12 it is programmable by two bits PR0 & PR1.

  So four factors – they are 1,4,8 & 16.
- Programming inputs of timer at start (Pre loading timers)
  - Ex 1: 8051 has the facility. TL0 & TH0 to be loaded with initial

  non zero value. Let value be X after (2^n-1-x) pulses , outputs will be 1s. After (2^n-x) pulses all outputs will become 0 (TL0 and TH0 will also become 0). It is called overflow.
  - Ex 2 : In 68HC11 this feature is not available as it is free running
- Auto Loading inputs
  - After one cycle of counts i.e. when over flow occurs, inputs will be loaded again automatically and cycle repeats

4.2 Overflow events in 8051

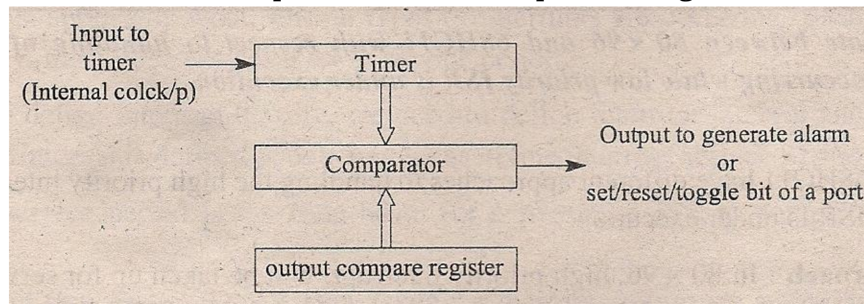
- Overflow sets a flag (TF0 in 8051 for timer 0)
- TF0 interrupts CPU
- TF0 reset automatically on branching to ISR
- Hence TF0 can be used for next interrupt.
- ISR can start or stop an event. For example –Robot arm movement
- Without interrupt TF0 can be polled and on being set a port bit can be set/reset/toggled
- ISR can be used to maintain a real time clock in memory locations

4.3 Free Running counter and real time control

- Programmable timers not ideal for real time control as time is lost between start/stop. (latency of ISR introduces delay in case of interrupt driven timer
- Free running counter is ideal for real time control as no time is lost
- It does not have start/stop and preloading the input facility
- Once it starts on power up, it keeps running
- It is like needle of a clock
- A watch is like three free running counter one for second, one for minute and one for hour

4.4 Output compare with free running counter

- Output of counter is compared with a register which is preset to a pre defined value
- On successful comparison an event can start like raising an alarm
- On successful comparison an interrupt can be generated



Sequence of steps to move a robotic arm every 16 seconds using output compare Register and 16 bit free running counter receiving pulses every 2ms.

- Assumptions
  - ISR1 for timer over flow interrupt. It updates a memory location (M1) to keep track of number of overflows
  - ISR2 is for output compare interrupt
  - Overflow and OC interrupt masked to begin with
  - M1 is FFH and indicates initial start condition
- Count inputs to timer should be 16s/24s = 8000000
- To start with we start robot arm and after 8000000pulses we stop robot arm
- Timer should overflow N times where N = 8000000/65536 = 122 times = 7AH (122 decimal)

- After 16 seconds the timer count should be overflow 122 times and should

have output count of 8000000 -122*65536 =4608=1200H

- During first ISR1 when M1=FF disable interrupts and load OC register. Now enable interrupts

- Now robot arm is started during first ISR1. Since timer is free running it would have advanced to a value x. Hence for 16 second calculation x should be added to 1200H I.E. 16 seconds will be completed after timer reaches the value x+1200H. Put the value x+1200 in OC register

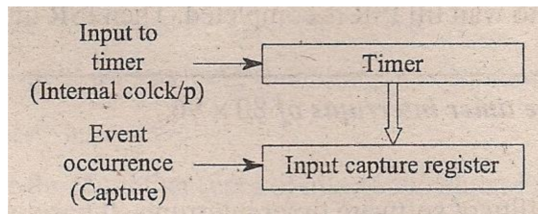
- In subsequent ISR1 increment M1 and check if it 7AH, if not enable further overflow interrupts and return. If Yes, disable further interrupts and stop the timer

- In ISR2 check M1 if it is 7AH stop robot arm and disable OC interrupt, else,

just return after enabling OC interrupt

4.5 Input capture with free running counter

- Input event occurring at a particular time instant is noted by reading output of free running counter and putting the value in a separate register called input capture register
- Can be used for noting time of occurrence of any event. You can also find time interval between two events
- Time can be noted either at positive transition of event or negative transition or both



4.6 Sequence of steps to find time taken by weight lifter in lifting the weight

- Assumptions
  - ISR3 executed on timer overflow. It updates M1 to keep track of overflows
  - ISR4 executed on input capture interrupt. It captures counts into a register

- Assume M1 is an 8 bit location. M2 & M3 are two separate 16 bit locations. M2 & M3 are reset to 0
- M1 = FFH indicates initial condition
- Enabled timer at input capture interrupts
- ISR3, increments M1 and enables overflow interrupt if M1 is not equal to FFH. Else, mask further interrupts (IC interrupts)
- ISR4, if M2 = OOH store IC 1, 16 bits at M2. Enable further interrupts

- ISR4, on next ICI interrupt, if M3 = OOH store IC2 as 16 bits of M3.

Mask IC and overflow interrupts

- At M2 & M3 we have 16 bit values X&Y
- M1+1 (because initial value of M1 was FF) will give number of over flows. M1 is 8 bit value
- Now time interval = (M3-M2)*2ms. If number of overflows is 0. If it is not 0 then time interval = [{FFFFH+1-M2}+Number of overflows*2^{16}+M3]

4.7 Real time clock interrupts (RTCs)

- Real time clock interrupts are generated by using free running timer or counter with a known clock signal
- The time never stops and cannot be reset or preloaded
- The input clock can be pre scaled by a factor, 1,4,8,16 etc . Pre scaling means division of the clock
- RTCs will initiate an ISR which can be used to maintain system time continuously. After updating timing information it enables next RTC interrupt. ISR Can also be used to perform a specific task
- Operating System can use these interrupts to initiate number of the tasks in a sequence

4.8 RTC interrupt in 68HC11 micro controller

- 68HC11 has got a 13 bit real time counter
- Input to this is E-CLOCK which is normally 2mhz
- A pair of bits RTR0-RTR1 in PACTL decides pre-scaling factor R. R can be 1,2,4 or 8. Hence the time period can be set as R*4096microsecond so it will be

4.096 or 8.192 or 16.384 or 32.768ms

- The bit 6 of the masked register TMASK 2.6 is used to enable real time interrupt
- Real time interrupt flag (RTIF) is bit 6 of TFLAG register (TFLAG2.6)
- RTC interrupt vector address if FFF0H-FFF1H

4.9 REAL TIME CLOCK INTERRUPTS USING OUTPUT COMPARE (IF RTC FEATURE IS NOT THERE THEN OC FEATURE CAN BE USED AS IN 80X96)

- Output compare register is compared with output of free running counter
- Interrupts are generated on successful comparison. These are called OC interrupts
- ISR maintains system time in memory location which is updated on every OC interrupt. It also enables next OC interrupt
- After successive intervals RTC OC interrupts will be generated
- 80X96 microcontroller has this feature. It has timer 1 which is 16 bit free running counter clocked internally at 1.66microsecond for a Fosc of 16MHZ

4.10 DIFFERENCE BETWEEN REAL TIME CLOCK INTERRUPT USING TIMERS AND OC WITH FREE RUNNING COUNTER

- Timer based RTC interrupts depend on 2^n*R*T and hence some specific value
  of R. Here R is a pre-scalar and width of the timer and T is input clock period
- In 68HC11 N=13, R=1,2,4 or 8
- In OC based RTC interrupt, interrupts can be programmed with any value i.e.
  X and intervals at which interrupts occur will be X*T where T is input clock period to free running counter and X is between 0 and 65536

4.11 SOFTWARE TIMERS

- Software timers are similar to real time clock
- Software timer is used to generate interrupt at certain instance. The instance is when a specific count value (software) stored in CAM is compared with a specific free running timer
- The interrupt flag is a bit of an addressable register
- A vector address is assigned for SWT interrupt
- ISR maintains memory location for time updates
- Specific task can be run after memory location reaches a certain value
- These memory locations are CAM locations. Each location of CAM contains 16 bit values for comparison with timer and few other N bits for commands. In 80X96 23 bit entries are used

4.12 DIFFERENCE BETWEEN SOFTWARE TIMER INTERRUPT AND OTHER RTC INTERRUPTS

- RTC interrupt occurs after R*2^{13}*P*T. Where P is fixed or program within specific clock periods and R is programmed at any instant

- SWT interrupt is where timer shows the count value = X. Where X can be defined

at any instant and it can have any value between 0 and 65536

- Similarly software timer defers from over flow interrupt in the sense that it has a distinct vector address when an ISR is initiated after equality of specific count

value and out of timer. Whereas overflow interrupt occurs only after $2^{16}*P*T$ where P is fixed in 80X96 and T is the duration of clock pulse. P is programmable in 68 HC11. Software timer can be defined on quick successions by defining different count values.

- Software defers from output compare interrupt in the sense that OC interrupt initiates an ISR where an output bit is set or reset for external control. Where as

in software timer interrupt ISR executes or initiates a specific user task

4.13 INTERRUPT INTERVAL AND DENSITY CONSTRAINTS

- Let T_{intv} is the interrupt interval between successive interrupt events
- Let T_I be the interrupt service period for I^{th} interrupt
- $T_I = T_{initial} + T_{execution} + T_{end}$
- Now T_I / T_{intv} is the fraction of time spent in a interrupt service
- $T_{initial}$ = Initial actions like disabling interrupts, save CPU registers on the stack
- $T_{end} = T_{end}$ actions like reassigning priorities, re-enabling interrupts and retrieving CPU registers
- $T_{execution}$ or T_{exec} is the time taken by the current ISR. If critical region is there then it should be added to it
- For N interrupts, Interrupt density = $^{i=n} E_{i=1} T_I / T_{intv}$. So interrupt density

depends on sum of ratios of all ISR execution times and their interval of occurrences

- Interrupt constraint means that interrupt density should be less than 1. Hence T_I should be as low as possible
- T_{inst} is the time taken by the current instruction during which interrupt occurred. DIV instruction normally takes the longest time
- T_{enwait} is the time during which interrupt remains disabled. It is actually waiting

for enabling the interrupts. For Eg waiting for critical region to be over. It may also include the execution time of current ISR.

- Interrupt latency when no other higher priority than the present one is pending is given below:
    - $T_{lat} = T_{inst} + T_{enwait} + T_{end} + T_{init}$
    - T_{end} is relevant when you are waiting for current ISR to be over
- Interrupt latency when other higher interrupt than present one is pending is given below:
    - $T_{lat} = T_{inst} + T_{enwait} + T_{init} + T_{end} + T_h$

- o T_h is the time for initializing, execution time and ending of each of higher interrupts than the present priority interrupts

- As an example assume that a service routine is executing an instruction of 3 micro second when the interrupt event occurs. The initial terminating actions before the execution of task related instruction starts take 1 micro second. Initial actions in ISR takes 10 micro second. Now the worst case latency period when there are other 3 higher priority interrupts pending which can take 80 micro second, 40 micro second, 100 micro second respectively
  - o T_{inst} = 3 micro second
  - o T_{end} = 1 micro second
  - o T_{init} = 10 microsecond
  - o T_{lat} (worst case) = T_{inst} + (3 + 10+ 1 + 80) + (10+ 1 + 40) + (10 + 1 + 100)
  - o T_{lat} for highest priority routine = 3 + 10
  - o T_{lat} for second highest priority routine = 3+10+1+80
  - o T_{lat} for third highest priority routine = (3 +10 + 1 + 80) + (10 + 1 + 40)

4.1 4 EXAMPLES ON TIMERS RELATED ACTIVITIES

- 8BIT timer receiving input pulses at 4 Microsec interval. Find preload value so that it times out after 100microsecond. 100Microsecond/4Microsecond = 25, it should overflow after 25 pulses. So preload value should be (256-25) = 231=E7H
- In above example if input is 1MicroscaledSec then it should be prescaled by a factor of 4 so that P*T = 4Microsec therefore P=4 since T=1. This is assuming that preload value is (256-25) i.e. overflow after 25 pulses
- In 8051 prescale factor is not there interrupt has to be generated after 100Microsecond , the following program has to be written
  - ▪ Assume F_{OSC} = 12MHZ

  Clock to timer is $f_{OSC}/12$ = 1MHZ
  Hence pulse width = 1Microsec
  Now timer zero to overflow after 100 pulse. So load (256-100) as preload value i.e.156 or 9CH.
  - o Program timer 0 in mode 3 so that TL0 and TH0 can be used as 2 independent 8 bit timers use TL0 here
  - o Load TL0 with 9C
  - o Enable timers 0 for overflow interrupt
  - o Run timer by setting TR0
  - o Program

```
MOV   TL0,#9CH
MOV   TMOD,#03H
MOV   IE,#82H
SETB  TR0
```

Notes

1. If timer has to be reloaded. Stop timer and reload and start again

2. If timer is used in auto reload mode. No need to reload, stop or start

- Spindle rotates as per windspeed. After each rotation a pulse is generated. 8 bit timer overflows in 20 seconds – find spindle speed in RPM
    - 20 seconds = 256rotations as timer overflows after 256 pulse and
      each pulse is after one rotation.

  Therefore 1 rotation = 20/256 =10/128 seconds
  OR
  In one second 128/10 rotations takes place
  Hence in one minute 128/10 * 60 or 768 rotations take place
  Hence RPM = 768
  NOTE: Timer is used as counter for above purpose.
  Let us assume that spindle makes 25 rotations/minute if wind speed is 1kmph (25RPM =1kmph i.e. if wind speed is 1kmph spindle will make 25 rotation per minute)
  Then what is wind speed if spindle makes 768 rotations per minute. Now observed wind speed = 768/25=30.7kmph
  i.e. if spindle speed is 25 rpm wind speed = 1kmph
  hence if spindle speed is 768 wind speed = ?
  768/25 = 20.7kmph

  - **EXAMPLE**

Move a robotic arm for 1024.P.T.second

1. Load timer 1 with (2^{16}-1024). First start robotic arm by sending an out bit=1. Start timer 1 by setting TR1=1. On interrupt from T1 after overflow that is after 1024.P.T seconds send out bit = 0 to stop robotic arm

2. Now if we have to move robotic arm after ($4*2^{16}$ + 1024).P.T seconds.

Now after first overflow don't send a 0 bit to stop robotic arm. It should stop after 5th overflow. So timer is programmed without auto reload capacity. If P.T. = 8Microsec i.e. P=8, T=1, then total time elapsed will be (4*65536 + 1024) * 8 = 2.1seconds

PROGRAM

- Reset Timer
- Initialize TMOD,TL0,TH0
- Enable interrupt
- Run timer
- CLR P1.0 (Take action)
- Self Loop (Wait for interrupt)
    - ISR

- INC M1
- CJNE M1,#N,EXIT
- SETB P1.1
- RETI
- EXIT: SETB TR0
- RETI

3. Using 2 timers find revolutions per second in a motor

    - Use 16-bit timer T0 as a counter with an initial value=0=X
    - Use 16-bit timer T1 as timer with initial value = Y
    - T1 will overflow after (2^{16}-Y).P.T seconds
    - Read output of T0. It gives a count of N

    1 count = 1 revolution so N counts = N revolutions

    - N revolution = (2^{16}-Y).P.T seconds therefore 1 second = N/(2^{16}--Y).P.T revolution

    PROGRAM

    - Timer 0
        - Reset timer 0
        - Use timer 0 in mode 1 as counter
        - Initialize timer 0 (TL0,TH0)
        - Run timer 0
    - Timer 1
        - Reset timer 1
        - Use timer 1 in mode 1 as timer
        - Initialize time 1 with y (TL0,TH0)
        - Enable interrupt
        - Run timer 1
        - Self loop
    - ISR for timer 1
        - sStop timer 1
        - Increment M, check if M = 0, if yes read timer 0
        - Store TL0,TH0 in memory (This gives you revolution/second)

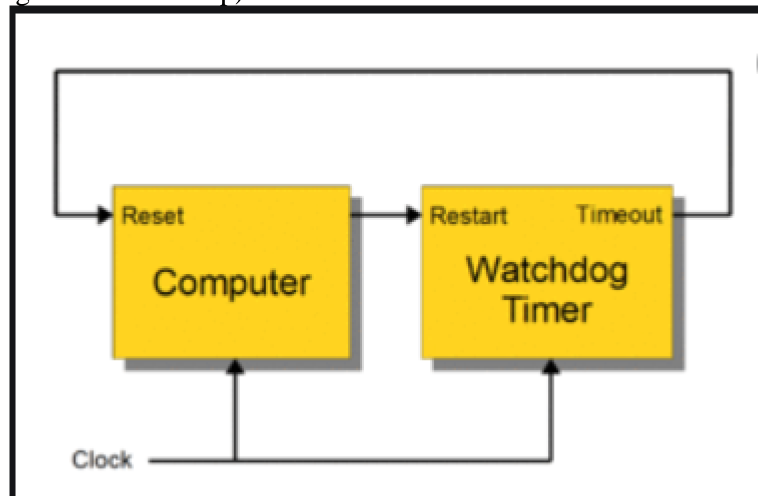
4. Switch OFF and ON a heater after 15 seconds using timer which is not free running assume 8051
    a. Let F_{OSC} = 12 MHZ

    Therefore input to timer 0=1Microsec
    Therefore it overflows after 65536Microsec
    65536 Microsec = 1 overflow
    $15*10^6$ Microsec = ?

Total number of overflows = $15*10^6/65536$

Here quotient = overflows
$15*10^6/65536$ = quotient * 2^{16}+remainder
Therefore remainder to be preloaded in the beginning and no. of overflows will be = the quotient

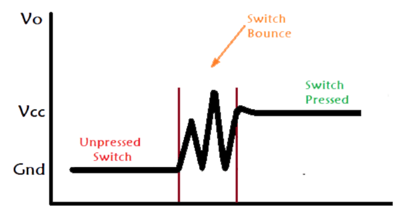
Watchdog timer:
A Watchdog Timer is a circuit that automatically invokes a reset unless the system being watched sends regular hold-off signals to the Watchdog.
Watchdog Circuit To make sure that a particular program is executing properly the Watchdog circuit is used. For instance the program may reset a particular flip-flop periodically. And the flip-flop is set by an external circuit. Suppose the flip-flop is not reset for long time it can be known by using external hardware. This will indicate that the program is not executed properly and hence an exception or interrupt can be generated. Watch Dog Timer(WDT) provides a unique clock, which is independent of any external clock. When the WDT is enabled, a counter starts at 00 and increments by 1 until it reaches FF. When it goes from FF to 00 (which is FF + 1) then the processor will be reset or an exception will be generated. The only way to stop the WDT from resetting the processor or generating an exception or interrupt is to periodically reset the WDT back to 00 throughout the program. If the program gets stuck for some reason, then the WDT will not be set. The WDT will then reset or interrupt the processor. An interrupt service routine will be invoked to take into account the erroneous operation of the program. (getting stuck or going into infinite loop).



UNIT 5

What is Switch Bouncing?

When we press a pushbutton or toggle switch or a micro switch, two metal parts come into contact to short the supply. But they don't connect instantly but the metal parts connect and disconnect several times before the actual stable connection is made. The same thing happens while releasing the button. This results the **false triggering or multiple triggering** like the button is pressed multiple times. Its like falling a bouncing ball from a height and it keeps bouncing on the surface, until it comes at rest.



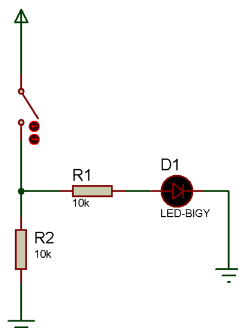
Simply, we can say that the **switch bouncing** is the non-ideal behavior of any switch which generates **multiple transitions of a single input**. Switch bouncing is not a major problem when we deal with the power circuits, but it cause problems while we are dealing with the logic or digital circuits. Hence, to remove the bouncing from the circuit **Switch Debouncing Circuit** is used.

What is Software Debouncing?

Debouncing occurs in software also, while programming programmers add delays to get rid of software debouncing. Adding a delay force the controller to stop for a particular time period, but adding delays is not a good option into the program, as it pause the program and increase the processing time. The best way is to use interrupts in the code for software bouncing. Arduino have code to prevent the software bouncing.

Switch Debouncing Methods

First, we will demonstrate the **circuit without the switch debounce**.



You can also see the waveform in oscilloscope while push button in bouncing. It shows that how much bouncing has occurred during the switching of the pushbutton.
There are three commonly used **methods to prevent the circuit from switch bouncing**.

- Hardware Debouncing
- RC Debouncing

- Switch Debouncing IC

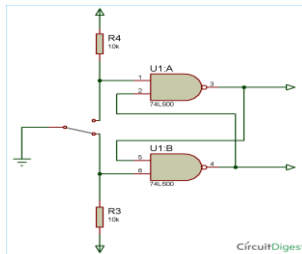
1. Hardware Debouncing

In the hardware debouncing technique we use an S-R flip flop to prevent the circuit from switch bounces. This is the best debouncing method among all.

Components Required

- Nand Gate IC 74HC00
- Toggle Switch
- Resistor (10k -2nos.)
- Capacitor (0.1uf)
- LED
- Breadboard

Circuit Diagram



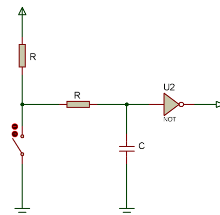
Working of the Hardware Debounce Circuit

The circuit is consists of two Nand gates (74HC00 IC) forming a SR flip flop. As you can see in the circuit diagram whenever the toggle switches to the A side the output logic gets 'HIGH'. Here, we have used an oscilloscope to detect the bouncing. And, as you can see in the waveform given below, the logic is shifting with a slight curve rather than bouncing. The resistors used in the circuit are pull-up resistors.

Whenever, the switch is moving between the contacts to create the bounce, the flip flop maintains the output because the '0' is fed back from the output of the Nand gates.

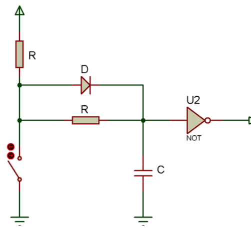
2. R-C Debouncing

The R-C is defined by its name only, the circuit used a RC network for the protection from switch bounce. The capacitor in the circuit filter the instant changes in the switching signal. When the switch is in open state the voltage across the capacitor remain zero. Initially, when the switch is open the capacitor charge through the R1 and R2 resistor.



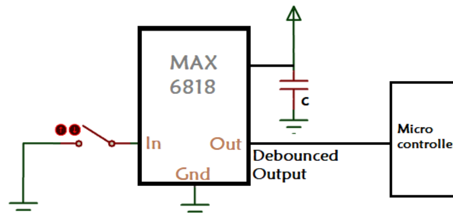
When the switch is closed the capacitor starts discharging to zero hence the voltage at input terminal of the inverting Schmitt trigger is zero, so the output becomes HIGH.

In the bouncing condition, the capacitor stops the voltage at Vin until it reaches to Vcc or Ground. To increase the speed of RC debouncing we can connect a diode as shown in the below image. Thus, it reduces the charging time of the capacitor.



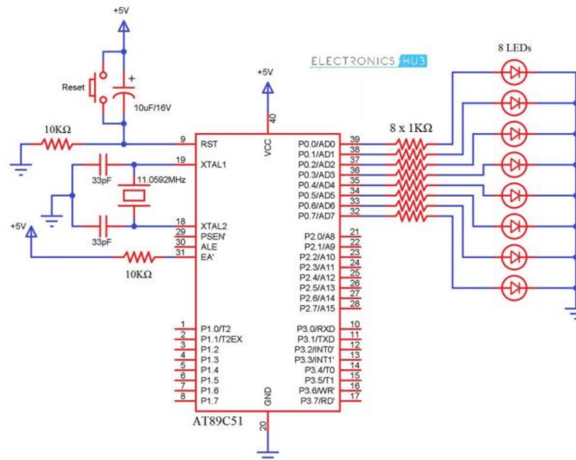
3. Switch Debouncing IC

There are ICs available in market for switch debouncing. Some of the **debouncing ICs are MAX6816, MC14490, and LS118**.Below is the circuit diagram for switch debouncing using MAX6818.



Interfacing LED with 8051 Microcontroller

Light Emitting Diodes or LEDs are the mostly commonly used components in many applications. They are made of semiconducting material. In this, It describe about basics of Interfacing LED with 8051 Microcontroller. Principle behind **Interfacing LED** with **8051.**The main principle of this circuit is to **interface LEDs** to the **8051** family micro controller. Commonly, used **LEDs** will have voltage drop of 1.7v and current of 10mA to glow at full intensity. This is applied through the output pin of the micro controller.

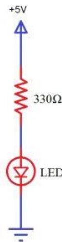


In this circuit, LEDs are connected to the port P0. The controller is connected with external crystal oscillator to pin 18 and 19 pins. Crystal pins are connected to the ground through capacitors of 33pf.

How to Control LEDs?

Light Emitting Diodes are the semi conductor light sources. Commonly used LEDs will have a cut-off voltage of 1.7V and current of 10mA. When an LED is applied with its required voltage and current it glows with full intensity.

The Light Emitting Diode is similar to the normal PN diode but it emits energy in the form of light. The color of light depends on the band gap of the semiconductor. The following figure shows “how an LED glows?”



Thus, LED is connected to the AT89C51 microcontroller with the help of a current limiting resistor. The value of this resistor is calculated using the following formula.

R= (V-1.7)/10mA, where V is the input voltage.

Generally, microcontrollers output a maximum voltage of 5V. Thus, the value of resistor calculated for this is 330 Ohms. This resistor can be connected to either the cathode or the anode of the LED.

Interfacing 16×2 LCD with 8051

We use LCD display for the messages for more interactive way to operate the system or displaying error messages etc. interfacing LCD to microcontroller is very easy if you understanding the working of LCD

LCD display is an inevitable part in almost all embedded projects and this article is about interfacing a 16×2 LCD with **8051 microcontroller**. Many guys find it hard to interface LCD module with the 8051 but the fact is that if you learn it properly, its a very easy job and by knowing it you can easily design embedded projects like digital voltmeter / ammeter, digital clock, home automation displays, status indicator display, **digital code locks**, **digital speedometer/ odometer**, display for music players etc etc. Thoroughly going through this article will make you able to display any text (including the extended characters) on any part of the 16×2 display screen. In order to understand the interfacing first you have to know about the 16×2 LCD module.

16×2 LCD module.

16×2 LCD module is a very common type of LCD module that is used in 8051 based embedded projects. It consists of 16 rows and 2 columns of 5×7 or 5×8 LCD dot matrices. The module were are talking about here is type number JHD162A which is a very popular one . It is available in a 16 pin package with back light ,contrast adjustment function and each dot matrix has 5×8 dot resolution. The pin numbers, their name and corresponding functions are shown in the table below.

Pin No:	Name	Function

1	**VSS**	This pin must be connected to the ground
2	**VCC**	Positive supply voltage pin (5V DC)
3	**VEE**	Contrast adjustment
4	**RS**	Register selection
5	**R/W**	Read or write
6	**E**	Enable
7	**DB0**	Data
8	**DB1**	Data
9	**DB2**	Data
10	**DB3**	Data
11	**DB4**	Data
12	**DB5**	Data
13	**DB6**	Data
14	**DB7**	Data

15	LED+	Back light LED+
16	LED-	Back light LED-

VEE pin is meant for adjusting the contrast of the LCD display and the contrast can be adjusted by varying the voltage at this pin. This is done by connecting one end of a POT to the Vcc (5V), other end to the Ground and connecting the center terminal (wiper) of of the POT to the VEE pin. See the circuit diagram for better understanding.

The JHD162A has two built in registers namely data register and command register. Data register is for placing the data to be displayed , and the command register is to place the commands. The 16×2 LCD module has a set of commands each meant for doing a particular job with the display. We will discuss in detail about the commands later. High logic at the RS pin will select the data register and Low logic at the RS pin will select the command register. If we make the RS pin high and the put a data in the 8 bit data line (DB0 to DB7) , the LCD module will recognize it as a data to be displayed . If we make RS pin low and put a data on the data line, the module will recognize it as a command.

R/W pin is meant for selecting between read and write modes. High level at this pin enables read mode and low level at this pin enables write mode.

E pin is for enabling the module. A high to low transition at this pin will enable the module.

DB0 to DB7 are the data pins. The data to be displayed and the command instructions are placed on these pins.

LED+ is the anode of the back light LED and this pin must be connected to Vcc through a suitable series current limiting resistor. LED- is the cathode of the back light LED and this pin must be connected to ground.

16×2 LCD module commands.

16×2 LCD module has a set of preset command instructions. Each command will make the module to do a particular task. The commonly used commands and their function are given in the table below.

Command	Function
0F	LCD ON, Cursor ON, Cursor blinking ON
01	Clear screen
02	Return home

04	Decrement cursor
06	Increment cursor
0E	Display ON ,Cursor blinking OFF
80	Force cursor to the beginning of 1st line
C0	Force cursor to the beginning of 2nd line
38	Use 2 lines and 5×7 matrix
83	Cursor line 1 position 3
3C	Activate second line
08	Display OFF, Cursor OFF
C1	Jump to second line, position1
OC	Display ON, Cursor OFF
C1	Jump to second line, position1
C2	Jump to second line, position2

LCD initialization.

The steps that has to be done for initializing the LCD display is given below and these steps are common for almost all applications.

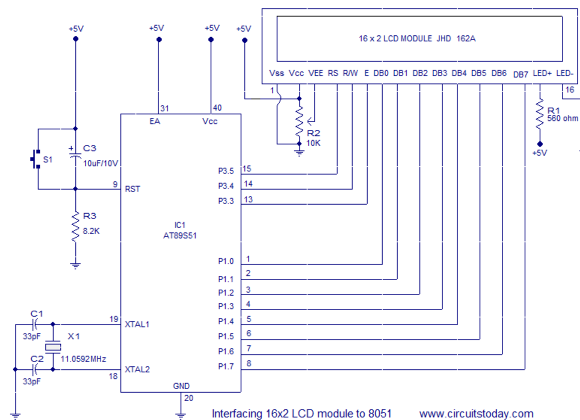
- Send 38H to the 8 bit data line for initialization
- Send 0FH for making LCD ON, cursor ON and cursor blinking ON.
- Send 06H for incrementing cursor position.
- Send 01H for clearing the display and return the cursor.

Sending data to the LCD.

The steps for sending data to the LCD module is given below. I have already said that the LCD module has pins namely RS, R/W and E. It is the logic state of these pins that make the module to determine whether a given data input is a command or data to be displayed.

- Make R/W low.
- Make RS=0 if data byte is a command and make RS=1 if the data byte is a data to be displayed.
- Place data byte on the data register.
- Pulse E from high to low.
- Repeat above steps for sending another data.

Circuit diagram.



The circuit diagram given above shows how to interface a 16×2 LCD module with AT89S1 microcontroller. Capacitor C3, resistor R3 and push button switch S1 forms the reset circuitry. Ceramic capacitors C1,C2 and crystal X1 is related to the clock circuitry which produces the system clock frequency. P1.0 to P1.7 pins of the microcontroller is connected to the DB0 to DB7 pins of the module respectively and through this route the data goes to the LCD module. P3.3, P3.4 and P3.5 are connected to the E, R/W, RS pins of the microcontroller and through this route the control signals are transffered to the LCD module. Resistor R1 limits the current through the back light LED and so do the back light intensity. POT R2 is used for adjusting the contrast of the display. Program for interfacing LCD to 8051 microcontroller is shown below.

Program.

```
MOV A,#38H // Use 2 lines and 5x7 matrix
ACALL CMND
MOV A,#0FH // LCD ON, cursor ON, cursor blinking ON
ACALL CMND
MOV A,#01H //Clear screen
ACALL CMND
```

```
MOV A,#06H //Increment cursor
ACALL CMND
MOV A,#82H //Cursor line one , position 2
ACALL CMND
MOV A,#3CH //Activate second line
ACALL CMND
MOV A,#49D
ACALL DISP
MOV A,#54D
ACALL DISP
MOV A,#88D
ACALL DISP
MOV A,#50D
ACALL DISP
MOV A,#32D
ACALL DISP
MOV A,#76D
ACALL DISP
MOV A,#67D
ACALL DISP
MOV A,#68D
ACALL DISP

MOV A,#0C1H //Jump to second line, position 1
ACALL CMND

MOV A,#67D
ACALL DISP
MOV A,#73D
ACALL DISP
MOV A,#82D
ACALL DISP
MOV A,#67D
ACALL DISP
MOV A,#85D
ACALL DISP
MOV A,#73D
ACALL DISP
MOV A,#84D
ACALL DISP
MOV A,#83D
ACALL DISP
MOV A,#84D
ACALL DISP
MOV A,#79D
ACALL DISP
MOV A,#68D
ACALL DISP
MOV A,#65D
ACALL DISP
MOV A,#89D
ACALL DISP
```

```
HERE: SJMP HERE

CMND: MOV P1,A
CLR P3.5
CLR P3.4
SETB P3.3
CLR P3.3
ACALL DELY
RET

DISP:MOV P1,A
SETB P3.5
CLR P3.4
SETB P3.3
CLR P3.3
ACALL DELY
RET

DELY: CLR P3.3
CLR P3.5
SETB P3.4
MOV P1,#0FFh
SETB P3.3
MOV A,P1
JB ACC.7,DELY

CLR P3.3
CLR P3.4
RET

END
```

Subroutine CMND sets the logic of the RS, R/W, E pins of the LCD module so that the module recognizes the input data (given to DB0 to DB7) as a command.Subroutine DISP sets the logic of the RS, R/W, E pins of the module so that the module recognizes the input data as a data to be displayed .

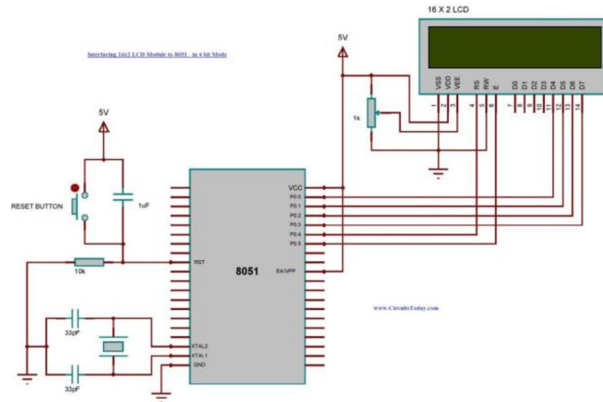
Interfacing LCD Module to 8051 in 4 Bit Mode (using only 4 pins of a port)

The microcontroller like 8051 has only limited number of GPIO pins (GPIO – general purpose input output). So to design complex projects we need sufficient number of I/O pins . An LCD module can be interfaced with a microcontroller either in **8 bit mode** (as seen above) or in 4 bit mode. 8 bit mode is the conventional mode which uses 8 data lines and RS, R/W, E pins for functioning. However 4 bit mode uses only 4 data lines along with the control pins. This will saves the number of GPIO pins needed for other purpose.

Objectives

- Interface an LCD with 8051 in 4 bit mode

- Use a single port of the microcontroller for both data and control lines of the LCD.



LCD Module to 8051 – 4 Bit Mode

As shown in the circuit diagram, port 0 of the controller is used for interfacing it with LCD module. In 4 bit mode only 4 lines D4-D7, along with RS, R/W and E pins are used. This will save us 4 pins of our controller which we might employ it for other purpose. Here we only need to write to the LCD module. So the R/W pin can be ground it as shown in the schematic diagram. In this way the total number of pins can be reduced to 6. In 4 Bit mode the data bytes are split into two four bits and are transferred in the form of a nibble. The data transmission to a LCD is performed by assigning logic states to the control pins RS and E. The reset circuit, oscillator circuit and power supply need to be provided for the proper working of the circuit.

Program – Interface LCD Module to 8051 – 4 Bit Mode

```
RS EQU P0.4
EN EQU P0.5
PORT EQU P0
U EQU 30H
L EQU 31H
ORG 000H

MOV DPTR,#INIT_COMMANDS
ACALL LCD_CMD
MOV DPTR,#LINE1
ACALL LCD_CMD
MOV DPTR,#TEXT1
ACALL LCD_DISP
MOV DPTR,#LINE2
ACALL LCD_CMD
MOV DPTR,#TEXT2
ACALL LCD_DISP
SJMP $


SPLITER: MOV L,A
ANL L,#00FH
SWAP A
ANL A,#00FH
MOV U,A
```

```
RET

MOVE: ANL PORT,#0F0H
ORL PORT,A
SETB EN
ACALL DELAY
CLR EN
ACALL DELAY
RET


LCD_CMD: CLR A
MOVC A,@A+DPTR
JZ EXIT2
INC DPTR
CLR RS
ACALL SPLITER
MOV A,U
ACALL MOVE
MOV A,L
ACALL MOVE
SJMP LCD_CMD
EXIT2: RET

LCD_DATA: SETB RS
ACALL SPLITER
MOV A,U
ACALL MOVE
MOV A,L
ACALL MOVE
RET

LCD_DISP: CLR A
MOVC A,@A+DPTR
JZ EXIT1
INC DPTR
ACALL LCD_DATA
SJMP LCD_DISP
EXIT1: RET

DELAY: MOV R7, #10H
L2: MOV R6,#0FH
L1: DJNZ R6, L1
DJNZ R7, L2
RET

INIT_COMMANDS: DB 20H,28H,0CH,01H,06H,80H,0
LINE1: DB 01H,06H,06H,80H,0
LINE2: DB 0C0H,0
CLEAR: DB 01H,0

TEXT1: DB " CircuitsToday ",0
```

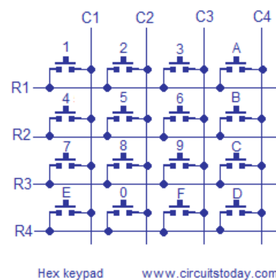
```
TEXT2: DB "4bit Using 1Port",0

END
```

Interfacing hex keypad to 8051

interfacing a hex key pad to 8051 microcontroller. A clear knowledge on interfacing hex key pad to 8051 is very essential while designing embedded system projects which requires character or numeric input or both. For example projects like digital code lock, numeric calculator etc. Before going to the interfacing in detail, let's have a look at the hex keypad.

Hex keypad.

Hex key pad is essentially a collection of 16 keys arranged in the form of a 4×4 matrix. Hex key pad usually have keys representing numerics 0 to 9 and characters A to F. The simplified diagram of a typical hex key pad is shown in the figure below.

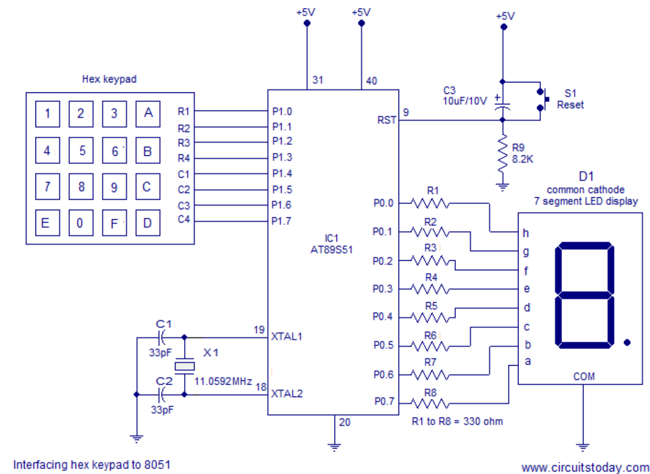


Hex keypad

The hex keypad has 8 communication lines namely R1, R2, R3, R4, C1, C2, C3 and C4. R1 to R4 represents the four rows and C1 to C4 represents the four columns. When a particular key is pressed the corresponding row and column to which the terminals of the key are connected gets shorted. For example if key 1 is pressed row R1 and column C1 gets shorted and so on. The program identifies which key is pressed by a method known as column scanning. In this method a particular row is kept low (other rows are kept high) and the columns are checked for low. If a particular column is found low then that means that the key connected between that column and the corresponding row (the row that is kept low) is been pressed. For example if row R1 is initially kept low and column C1 is found low during scanning, that means key 1 is pressed.

Interfacing hex keypad to 8051.

The circuit diagram for demonstrating interfacing hex keypad to 8051 is shown below.Like previous 8051 projects, AT89S51 is the microcontroller used here. The circuit will display the character/numeric pressed on a seven segment LED display. The circuit is very simple and it uses only two ports of the microcontroller, one for the hex keypad and the other for the seven segment LED display.



Interfacing hex keypad to 8051

The hex keypad is interfaced to port 1 and seven segment LED display is interfaced to port 0 of the microcontroller. Resistors R1 to R8 limits the current through the corresponding segments of the LED display. Capacitors C1, C2 and crystal X1 completes the clock circuitry for the microcontroller. Capacitor C3, resistor R9 and push button switch S1 forms a debouncing reset mechanism.

Program.

```
ORG 00H
MOV DPTR,#LUT // moves starting address of LUT to DPTR
MOV A,#11111111B // loads A with all 1's
MOV P0,#00000000B // initializes P0 as output port

BACK:MOV P1,#11111111B // loads P1 with all 1's
    CLR P1.0  // makes row 1 low
    JB P1.4,NEXT1  // checks whether column 1 is low and jumps to NEXT1 if not low
    MOV A,#0D   // loads a with 0D if column is low (that means key 1 is pressed)
    ACALL DISPLAY  // calls DISPLAY subroutine
NEXT1:JB P1.5,NEXT2 // checks whether column 2 is low and so on...
     MOV A,#1D
     ACALL DISPLAY
NEXT2:JB P1.6,NEXT3
     MOV A,#2D
     ACALL DISPLAY
NEXT3:JB P1.7,NEXT4
     MOV A,#3D
     ACALL DISPLAY
NEXT4:SETB P1.0
     CLR P1.1
     JB P1.4,NEXT5
     MOV A,#4D
     ACALL DISPLAY
NEXT5:JB P1.5,NEXT6
     MOV A,#5D
     ACALL DISPLAY
NEXT6:JB P1.6,NEXT7
```

```
      MOV A,#6D
      ACALL DISPLAY
NEXT7:JB P1.7,NEXT8
      MOV A,#7D
      ACALL DISPLAY
NEXT8:SETB P1.1
      CLR P1.2
      JB P1.4,NEXT9
      MOV A,#8D
      ACALL DISPLAY
NEXT9:JB P1.5,NEXT10
      MOV A,#9D
      ACALL DISPLAY
NEXT10:JB P1.6,NEXT11
       MOV A,#10D
       ACALL DISPLAY
NEXT11:JB P1.7,NEXT12
       MOV A,#11D
       ACALL DISPLAY
NEXT12:SETB P1.2
       CLR P1.3
       JB P1.4,NEXT13
       MOV A,#12D
       ACALL DISPLAY
NEXT13:JB P1.5,NEXT14
       MOV A,#13D
       ACALL DISPLAY
NEXT14:JB P1.6,NEXT15
       MOV A,#14D
       ACALL DISPLAY
NEXT15:JB P1.7,BACK
       MOV A,#15D
       ACALL DISPLAY
       LJMP BACK

DISPLAY:MOVC A,@A+DPTR // gets digit drive pattern for the current key from LUT
        MOV P0,A       // puts corresponding digit drive pattern into P0
        RET

LUT: DB 01100000B // Look up table starts here
     DB 11011010B
     DB 11110010B
     DB 11101110B
     DB 01100110B
     DB 10110110B
     DB 10111110B
     DB 00111110B
     DB 11100000B
     DB 11111110B
     DB 11110110B
     DB 10011100B
     DB 10011110B
```

```
DB 11111100B
DB 10001110B
DB 01111010B
END
```

About the program.

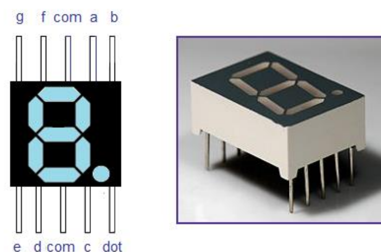
Firstly the program initializes port 0 as an output port by writing all 0's to it and port 1 as an input port by writing all 1's to it. Then the program makes row 1 low by clearing P1.0 and scans the columns one by one for low using JB instruction.If column C1 is found low, that means 1 is pressed and accumulator is loaded by zero and DISPLAY subroutine is called. The display subroutine adds the content in A with the starting address of LUT stored in DPTR and loads A with the data to which the resultant address points (using instruction MOVC A,@A+DPTR). The present data in A will be the digit drive pattern for the current key press and this pattern is put to Port 0 for display. This way the program scans for each key one by one and puts it on the display if it is found to be pressed.

Notes.

- The 5V DC power supply must be well regulated and filtered.
- Column scanning is not the only method to identify the key press. You can use row scanning also. In row scanning a particular column is kept low (other columns are kept high) and the rows are tested for low using a suitable branching instruction. If a particular row is observed low then that means that the key connected between that row and the corresponding column (the column that is kept low) is been pressed. For example if column C1 is initially kept low and row R1 is observed low during scanning, that means key 1 is pressed.
- A membrane type hex keypad was used during the testing. Push button switch type and dome switch type will also work. I haven't checked other types.
- The display used was a common cathode seven segment LED display with type number ELK5613A. This is just for information and any general purpose common cathode 7 segment LED display will work here.

Interfacing Seven segment display to 8051

interface a seven segment LED display to an 8051 microcontroller. 7 segment LED display is very popular and it can display digits from 0 to 9 and quite a few characters like A, b, C, ., H, E, e, F, n, o,t,u,y, etc. Knowledge about how to interface a seven segment display to a micro controller is very essential in designing embedded systems. A seven segment display consists of seven LEDs arranged in the form of a squarish **'8'** slightly inclined to the right and a single LED as the dot character. Different characters can be displayed by selectively glowing the required LED segments. Seven segment displays are of two types, ***common cathode and common anode.*** In common cathode type , the cathode of all LEDs are tied together to a single terminal which is usually labeled as '**com**' and the anode of all LEDs are left alone as individual pins labeled as a, b, c, d, e, f, g & h (or dot) . In common anode type, the anode of all LEDs are tied together as a single terminal and cathodes are left alone as individual pins. The pin out scheme and picture of a typical 7 segment LED display is shown in the image below.



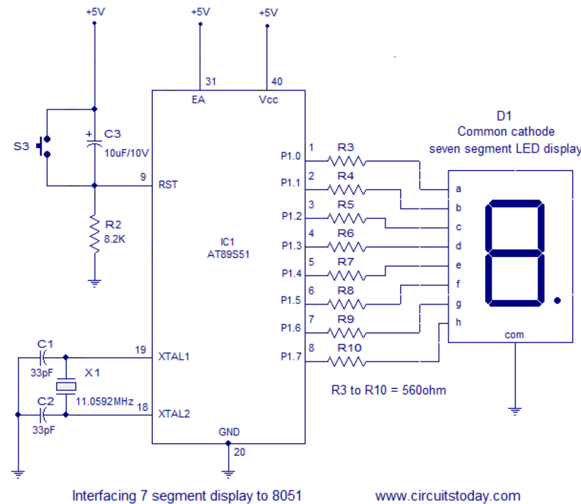
7 segment LED display

Digit drive pattern.

Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals **'a' to 'h'** in order to display different digits and characters. The common digit drive patterns (0 to 9) of a seven segment display are shown in the table below.

Digit	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

Interfacing seven segment display to 8051.



Interfacing 7 segment display to 8051

The circuit diagram shown above is of an AT89S51 microcontroller based 0 to 9 counter which has a 7 segment LED display interfaced to it in order to display the count. This simple circuit illustrates two things. How to setup simple 0 to 9 up counter using 8051 and more importantly how to interface a seven segment LED display to 8051 in order to display a particular result. The common cathode seven segment display D1 is connected to the Port 1 of the microcontroller (AT89S51) as shown in the circuit diagram. R3 to R10 are current limiting resistors. S3 is the reset switch and R2,C3 forms a debouncing circuitry. C1, C2 and X1 are related to the clock circuit. The software part of the project has to do the following tasks.

- Form a 0 to 9 counter with a predetermined delay (around 1/2 second here).
- Convert the current count into digit drive pattern.
- Put the current digit drive pattern into a port for displaying.

All the above said tasks are accomplished by the program given below.

Program.

```
ORG 000H //initial starting address
START: MOV A,#00001001B // initial value of accumulator
MOV B,A
MOV R0,#0AH //Register R0 initialized as counter which counts from 10 to 0
LABEL: MOV A,B
INC A
MOV B,A
MOVC A,@A+PC // adds the byte in A to the program counters address
MOV P1,A
ACALL DELAY // calls the delay of the timer
DEC R0//Counter R0 decremented by 1
MOV A,R0 // R0 moved to accumulator to check if it is zero in next instruction.
JZ START //Checks accumulator for zero and jumps to START. Done to check if counting has been finished.
SJMP LABEL
DB 3FH // digit drive pattern for 0
DB 06H // digit drive pattern for 1
DB 5BH // digit drive pattern for 2
```

```
DB 4FH // digit drive pattern for 3
DB 66H // digit drive pattern for 4
DB 6DH // digit drive pattern for 5
DB 7DH // digit drive pattern for 6
DB 07H // digit drive pattern for 7
DB 7FH // digit drive pattern for 8
DB 6FH // digit drive pattern for 9
DELAY: MOV R4,#05H // subroutine for delay
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
DJNZ R3,WAIT2
DJNZ R4,WAIT1
RET
END
```

About the program.

Instruction MOVC A,@A+PC is the instruction that produces the required digit drive pattern for the display. Execution of this instruction will add the value in the accumulator A with the content of the program counter(address of the next instruction) and will move the data present in the resultant address to A. After this the program resumes from the line after MOVC A,@A+PC.

In the program, initial value in A is 00001001B. Execution of MOVC A,@A+PC will add oooo1001B to the content in PC (address of next instruction). The result will be the address of label DB 3FH (line15) and the data present in this address ie 3FH (digit drive pattern for 0) gets moved into the accumulator. Moving this pattern in the accumulator to Port 1 will display 0 which is the first count.

At the next count, value in A will advance to 00001010 and after the execution of MOVC A,@+PC ,the value in A will be 06H which is the digit drive pattern for 1 and this will display 1 which is the next count and this cycle gets repeated for subsequent counts.

The reason why accumulator is loaded with 00001001B (9 in decimal) initially is that the instructions from line 9 to line 15 consumes 9 bytes in total.

The lines 15 to 24 in the program which starts with label DB can be called as a **Look Up Table (LUT)**. label DB is known as Define Byte – which defines a byte. This table defines the digit drive patterns for 7 segment display as bytes (in hex format). MOVC operator fetches the byte from this table based on the result of adding PC and contents in the accumulator.

Register B is used as a temporary storage of the initial value of the accumulator and the subsequent increments made to accumulator to fetch each digit drive pattern one by one from the look up table(LUT).

Note:- In line 6, Accumulator is incremented by 1 each time (each loop iteration) to select the next digit drive pattern. Since MOVC operator uses the value in A to fetch the digit drive pattern from LUT, value in ACC has to be incremented/manipulated accordingly. The digit drive patterns are arranged consecutively in LUT.

Register R0 is used as a counter which counts from 10 down to 0. This ensures that digits from o to 9 are continuously displayed in the 7 segment LED. You may note lines 4, 11, 12, and 13 in the above program. Line 4 initializes R0 to 10 (OAh). When the program counter reaches line 11 for the first time, 7 segment LED has already

displayed 0. So we can reduce one count and that is why we have written DEC Ro. We need to continuously check if R0 has reached full count (that is 0). In order to do that lines 12 and 13 are used. We move R0 to accumulator and then use the Jump if Zero (JZ) instruction to check if accumulator has reached zero. If Acc=0, then we makes the program to jump to START (initial state) and hence we restart the 7 segment LED to display from 0 to 9 again. If Acc not equal to zero, we continue the program to display the next digit (check line 14).

Interfacing Stepper Motor with 8051

- The stepper motors coil A,B,C,D is connected to the port 1 i.e. to P1.0, P1.2, P1.2 and P1.3.

- The Microcontroller does not provide sufficient current to drive motor and to safeguard 8081 from loading effect and burn out condition, a motor driver IC ULN 2003 between 8051 and stepper motor. ULN 2003 is a stepper motor driver.

- The stepper motor is user for controlling:

1. Position control
2. Direction control &
3. Speed control

- Calculations:

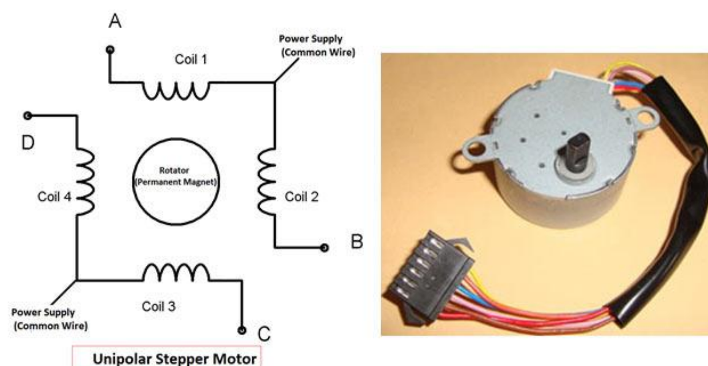
1. Total no. of steps= $\frac{Total\ rotational\ angle}{Step\ Angle}$

Ex: $\frac{360}{1.8}$ = 200 steps are required to complete one rotation

2. Total no. of repeated steps= $\frac{Total\ no.of\ steps}{Step\ sequence}$

Ex: $\frac{200}{4}$ = 50 repetition of sequence = (32) in Hexadecimal.

Stepper motors are basically two types: Unipolar and Bipolar. **Unipolar stepper** motor generally has five or six wire, in which four wires are one end of four stator coils, and other end of the all four coils is tied together which represents fifth wire, this is called common wire (common point). Generally there are two common wire, formed by connecting one end of the two-two coils as shown in below figure. Unipolar stepper motor is very common and popular because of its ease of use.



Unipolar Stepper Motor

In **Bipolar stepper** motor there is just four wires coming out from two sets of coils, means there are no common wire.

Stepper motor is made up of a stator and a rotator. Stator represents the four electromagnet coils which remain stationary around the rotator, and rotator represents permanent magnet which rotates. Whenever the coils energised by applying the current, the electromagnetic field is created, resulting the rotation of rotator (permanent magnet). Coils should be energised in a particular sequence to make the rotator rotate. On the basis of this "sequence" we can divide the working method of **Unipolar stepper motor** in three modes: Wave drive mode, full step drive mode and half step drive mode.

4-Step sequence (Full Drive mode)**:**

- In this type of functioning, the following 4 binary sequence/code are used for rotation: (Considering step angle= 1.8 degrees)

4- Step sequence binary pattern				HEX code	Comments
A	B	C	D		
1	0	0	1	09	Sequence for Clock wise rotation
1	1	0	0	0C	
0	1	1	0	06	
0	0	1	1	03	
0	0	1	1	03	Sequence for anti-clockwise rotation
0	1	1	0	06	
1	1	0	0	0C	
1	0	0	1	09	

8-Step Sequence(Half Drive mode:)**:**

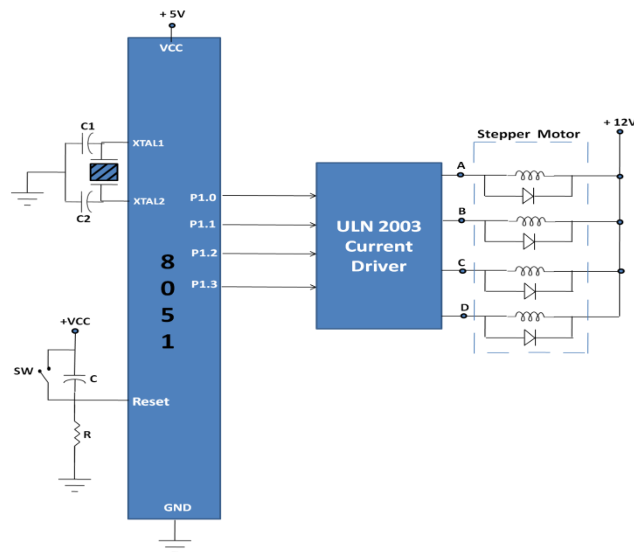
- In this type of functioning, the following 8 binary sequence/code are used for rotation: (Considering step angle= 0.9degrees)

4- Step sequence binary pattern				HEX code	Comments
A	B	C	D		
1	0	0	1		Sequence for clockwise rotation
1	0	0	0		
1	1	0	0		
0	1	0	0		
0	1	1	0		
0	0	1	0		
0	0	1	1		
0	0	0	1		
0	0	0	1		Sequence for anti-clockwise rotation
0	0	1	1		
0	0	1	0		
0	1	1	0		
0	1	0	0		
1	1	0	0		
1	0	0	0		
1	0	0	1		

Program

LABEL	OPCODE	OPERAND	COMMENT
	ORG	0000H	
	MOV	A,99#	
	MOV	R0, #200	
BACK:	MOV	P1,A	
	ACALL	DELAY	
	RR	A	
	DJNZ	R0,BACK	
HERE:	SJMP	HERE	
DELAY:	MOV	R2, #225	
L2:	MOV	R3, #225	
L1:	DJNZ	R3,L1	
	DJNZ	R2,L2	
	RET		
	END		

Interfacing Diagram

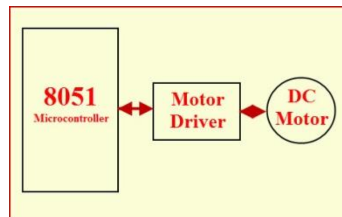


Troubleshooting

If your motor is not rotating OR vibrating but not rotating, then you must check the following checklist:

1. First check the circuit connections and code.
2. If the circuit and code is ok, then check that the stepper motor gets proper supply voltage (generally 12v), otherwise it just vibrate but not rotate.
3. If supply is fine, then check the four coil end points which in connected to ULN2003A. First find the two common end points and connect them to 12v, then connect the remaining four wires to ULN2003A and try every possible combination until motor get started. If you wouldn't connect them in proper order then the motor just vibrate instead of rotating.

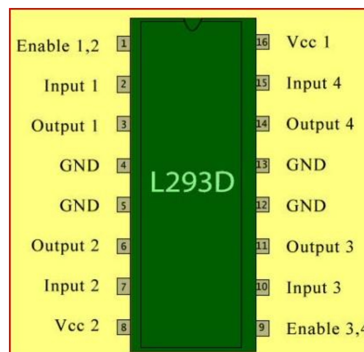
Interfacing DC Motor with 8051 Microcontroller



The main purpose of DC interfacing with 8051 microcontroller is for controlling the speed of the motor. The DC motor is an electrical machine with a rotating part termed as a rotor which has to be controlled. For example, consider the DC motor whose speed or direction of rotation of DC motor can be controlled using programming techniques which can be achieved by interfacing with 8051 microcontroller. So, in this article let us discuss about interfacing DC motor with 8051 microcontroller.

Motor Driver IC used for Interfacing DC motor with 8051

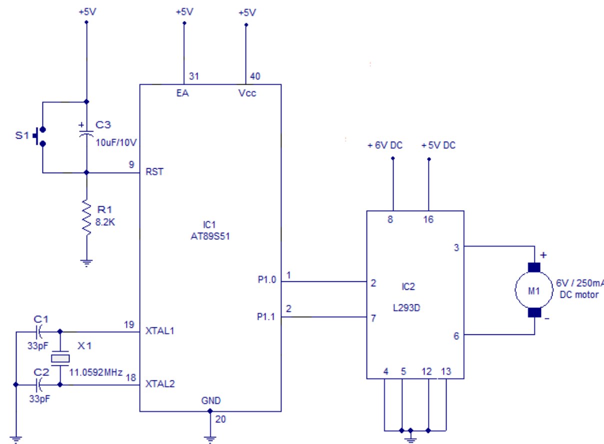
Here, interfacing 8051 with DC motor requires a motor driver. There are various types of driver ICs among which L293D is typically used for interfacing DC motor with 8051. L293 is an IC with 16 pins which are represented in the figure below.



Motor Driver IC L293D used for Interfacing DC

This L293 IC is having ratings of 600mA per channel and DC supply voltage in the range of 4.5V to 36V. These ICs can be protected from inductive spikes by connecting higher speed clamp diodes internally. This 16 pin L293D IC can be used for controlling the direction of two DC motors. The IC L293D works based on the H-bridge concept. The voltage can be made to flow in either direction using this circuit (H-bridge) such that

by changing the voltage direction the motor direction can be changed.



Bi directional DC motor using 8051.

This describes a bidirectional DC motor that changes its direction automatically after a preset amount of time (around 1S). AT89S51 is the microcontroller used here and L293 forms the motor driver. Circuit diagram is shown above

In the circuit components R1, S1 and C3 forms a debouncing reset circuitry. C1, C2 and X1 are related to the oscillator. Port pins P1.0 and P1.1 are connected to the corresponding input pins of the L293 motor driver. The motor is connected across output pins 3 and 6 of the L293. The software is so written that the logic combinations of P1.0 and P1.1 controls the direction of the motor. Initially when power is switched ON, P1.0 will be high and P1.1 will be low. This condition is maintained for a preset amount of time (around 1S) and for this time the motor will be running in the clockwise direction (refer the function table of L293). Then the logic of P1.0 and P1.1 are swapped and this condition is also maintained for the same duration . This makes the motor to run in the anti clockwise direction for the same duration and the entire cycle is repeated.

Progr am.

```
ORG 00H // initial starting address
MAIN: MOV P1,#00000001B // motor runs clockwise
ACALL DELAY // calls the 1S DELAY
MOV P1,#00000010B // motor runs anti clockwise
ACALL DELAY // calls the 1S DELAY
SJMP MAIN // jumps to label MAIN for repaeting the cycle
DELAY: MOV R4,#0FH
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
DJNZ R3,WAIT2
DJNZ R4,WAIT1
RET
END
```

Notes.

The maximum current capacity of L293 is 600mA/channel. So do not use a motor that consumes more than that. The supply voltage range of L293 is between 4.5 and 36V DC. So you can use a motor falling in that range.