VECTOR'S 8051 DOCUMENTATION FOR AT89s52

Head Office: #502, Naga Suri Plaza, Beside Minerva coffee shop, Behind Maithrivanam, Ameerpet, Hyderabad 500016 Tel: 91-40-2373 6669, Cell: 98 66 66 66 99

info@vectorindia.org , www.vectorindia.org

Chapter 1

Microcontrollers

Introduction

Microcontrollers have only been with us for a few decades but their impact (direct or indirect) on our lives is profound. Usually these are supposed to be just data processors are performing exhaustive numeric operations but their presence is unnoticed at most of the places like

- At supermarkets in Cash Registers, Weighing Scales, etc.
- At home in Ovens, Washing Machines, Alarm Clocks, etc…
- At play in Toys, VCRs, Stereo Equipment, etc.
- At office in Typewriters, Photocopiers, Elevators, etc…
- In industry in Industrial Automation, safety systems, etc…
- On roads in Cars, Traffic Signals, etc.

Simply an embedded controller is a controller that is embedded in a greater system. One can define an embedded controller as a controller (or computer) that is embedded into some device for some purpose other than to provide general purpose computing. Some devices like 68000, 32032, x86, Z80, and so on that are used as embedded controllers but they are not microcontrollers.

Example

For suppose if we want to control a device such as a microwave oven, car braking system or a cruise missile. An embedded controller may also embed on the on-chip resources like a microcontroller. Microcontrollers and microprocessors are widely used in embedded systems. However, we preferred microcontrollers to microprocessors for embedded systems due to low power consumption.

If we want to know about microcontroller first of all we have to know General purpose Microprocessor and Microcontroller.

Microprocessor

This is a normal CPU (Central Processing Unit) as you can find in a PC. Communication with external devices is achieved via a data bus, hence the chip mainly features data and address pins as well as a couple of control pins.

All peripheral devices (memory, floppy controller, USB controller, timer . . .) are connected to the bus. A microprocessor cannot be operated stand-alone; at the very least it requires some memory and an output device to be useful.

Please note that a processor is no controller. Nevertheless, some manufacturers and vendors list their controllers under the term "microprocessor". In this text we use the term processor just for the processor core (the CPU) of a microcontroller.

- Must add RAM, ROM, I/O ports, and timers externally to make them functional
- Make the system bulkier and much more expensive
- Have the advantage of versatility on the amount of RAM, ROM, and I/O ports

Microcontroller

A microcontroller already contains all components which allow it to operate standalone, and it has been designed in particular for monitoring and/or control tasks. In consequence, in addition to the processor it includes memory, various interface controllers, one or more timers, an interrupt controller, and last but definitely not least general purpose I/O pins which allow it to directly interface to its environment. Microcontrollers also include bit operations which allow you to change one bit within a byte without touching the other bits.

- The fixed amount of on-chip ROM, RAM, and number of I/O ports makes them ideal for many applications in which cost and space are critical
- In many applications, the space it takes, the power it consumes, and the price per unit are much more critical considerations than the computing power
- An embedded product uses a microprocessor (or microcontroller) to do one task and one task only
    - There is only one application software that is typically burned into ROM
- A PC, in contrast with the embedded system, can be used for any number of applications
    - It has RAM memory and an operating system that loads a variety of applications into RAM and lets the CPU run them
    - A PC contains or is connected to various embedded products
        - ❖ Each one peripheral has a microcontroller inside it that performs only one task

A digital computer having microprocessor as the CPU along with I/O devices and memory is known as microcomputer.

A microcontroller is a highly integrated chip, which includes on single chip, all or most of the parts needed for a controller. The microcontroller typically includes: CPU (Central Processing Unit), RAM (Random Access Memory), EPROM/PROM/ROM (Erasable Programmable Read Only Memory), I/O (input/output) – serial and parallel, timers, interrupt controller. For example, Intel 8051 is 8-bit microcontroller and Intel 8096 is 16-bit microcontroller.

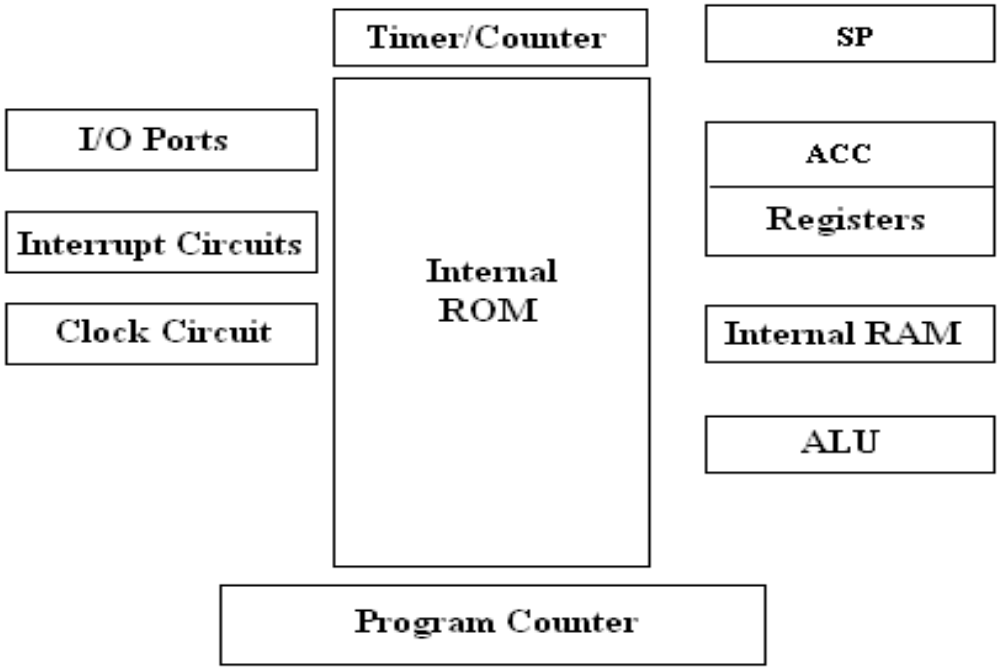
Microcontrollers and Microprocessors

A controller is used to control some process. At one time, controllers were built exclusively from logic components, and were usually large, heavy boxes. Later on, microprocessors were used and the entire controller could fit on a small circuit board. This is still common– one can find many controllers powered by one of the many common microprocessors (including Zilog Z80, Intel 8088, Motorola 6809, and others). As the process of miniaturization continued, all of the components needed for a controller were built right onto one chip. A one chip on a computer, or microcontroller was born.

By only including the features specific to the task (control), cost is relatively low. A typical microcontroller has bit manipulation instructions, easy and direct access to I/O (input/output), and quick and efficient interrupt processing. Figure 1.3 shows the block diagram of a typical microcontroller.

A CPU built into a single VLSI chip is called microprocessor. It contains arithmetic and logic unit (ALU), Instruction decodes and control unit, Instruction register, Program counter (PC), clock circuit (internal or external), reset circuit (internal or external) and registers. For example, Intel 8085 is 8-bit microprocessor and Intel 8086/8088 is 16-bit microprocessor. Microprocessor is general-purpose digital computer central processing unit (CPU).The microprocessor is general- Microprocessor is general-purpose digital computer central processing unit (CPU).

Block Diagram of Microcontroller



Central Processing Unit (CPU)

CPU is the brain of the computer system. It performs all operations on data. It continuously performs two operations: fetching and executing the instructions. It understands and executes the instructions based on a set of binary codes called the instruction set. The microprocessor is general-purpose device and additional external circuitry is added to make it microcomputer.

Another simple definition for CPU is it Executes information stored in memory

I/O (Input/output) devices

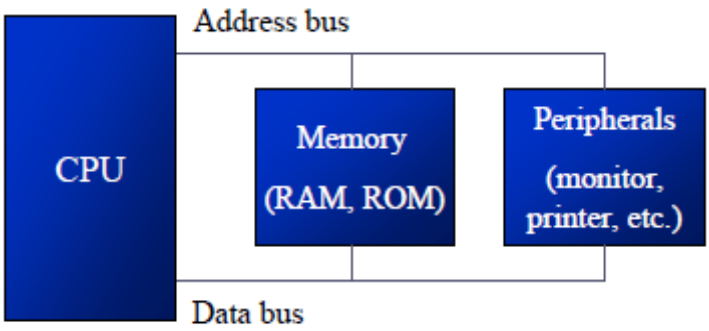
- Provide a means of communicating with CPU

Memory

RAM (Random Access Memory) - Temporary storage of programs that computer is running and the data is lost when computer is off

ROM (Read Only Memory) – contains programs and information essential to operation of the computer and the information cannot be changed by use, and is not lost when power is off

- **It is called nonvolatile memory**



EEPROM (also written **E^{2}PROM** and pronounced "e-e-prom," "double-e prom," "e-squared," or simply "e-prom") stands for **E**lectrically **E**rasable **P**rogrammable **R**ead-**O**nly

Memory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed,

Example

- Calibration tables
- Device configuration

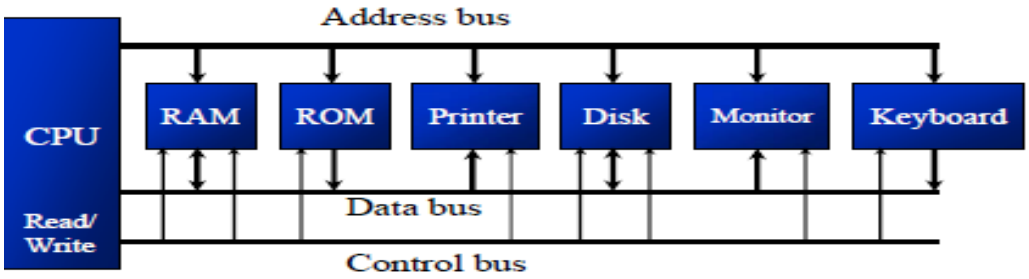
When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices. **EEPROMs** are realized as arrays of floating-gate transistors.

EEPROM is user-modifiable read-only memory **(ROM)** that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage generated externally or internally in the case of modern **EEPROMs**

Bus Unit

The CPU is connected to memory and I/O through strips of wire called a bus

- Carries information from place to place
  - Address bus
  - Data bus
  - Control bus



Address bus

For a device (memory or I/O) to be recognized by the CPU, it must be assigned an address

- The address assigned to a given device must be unique
- The CPU puts the address on the address bus, and the decoding circuitry finds the device
- The address bus is unidirectional

Data bus

- The CPU either gets data from the device or sends data to it
- Data buses are bidirectional

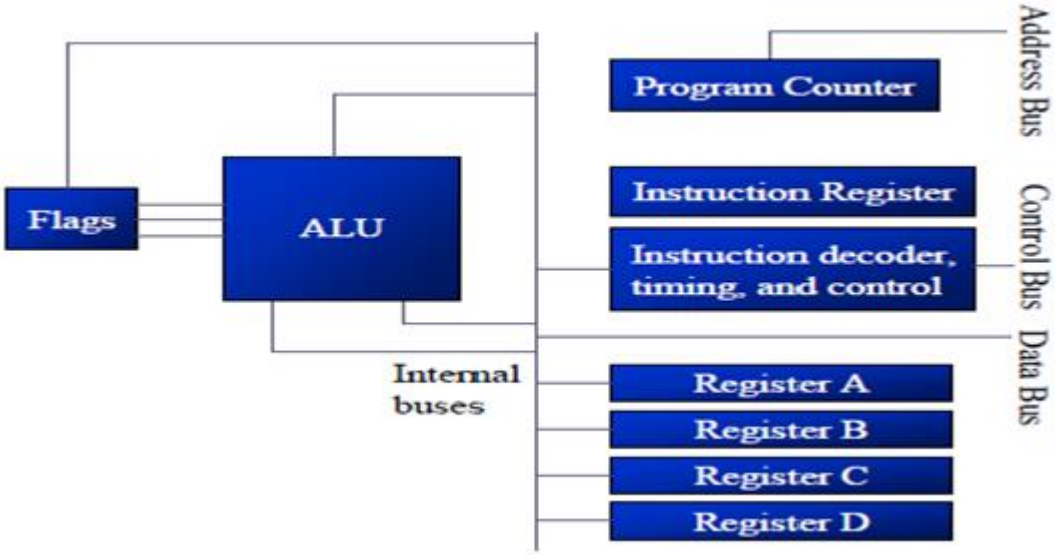
Control bus

- Provides read or write signals to the device to indicate if the CPU is asking for information or sending it information
- For the CPU to process information, the data must be stored in RAM or ROM, which are referred to as primary memory
- ROM provides information that is fixed and permanent
    - Tables or initialization program
- RAM stores information that is not permanent and can change with time
    - Various versions of OS and application packages
    - CPU gets information to be processed
        - first form RAM (or ROM)
        - if it is not there, then seeks it from a mass storage device, called secondary memory, and transfers the information to RAM

Registers

The CPU uses registers to store information temporarily

- Values to be processed
- Address of value to be fetched from memory
- Registers can be 8-, 16-, 32-, or 64-bit
- The disadvantage of more and bigger registers



ALU (arithmetic/logic unit)

Performs arithmetic functions such as add, subtract, multiply, and divide, and logic

Functions such as AND, OR, and NOT

Program counter

Points to the address of the next instruction is to be fetched

- As each instruction is executed, the program counter is incremented to point to the address of the next instruction is to be fetched

One more important register in the 8051 is the PC (Program Counter). The program counter points to the address of next instruction to be fetched. As the CPU fetches the opcode from program ROM, the program counter is incremented by 1 to point to the next instruction. The PC in the 8051 is 16 bit register. This means that the 8051 can access program addresses 0000 to FFFFH, a total of 64K bytes of code.

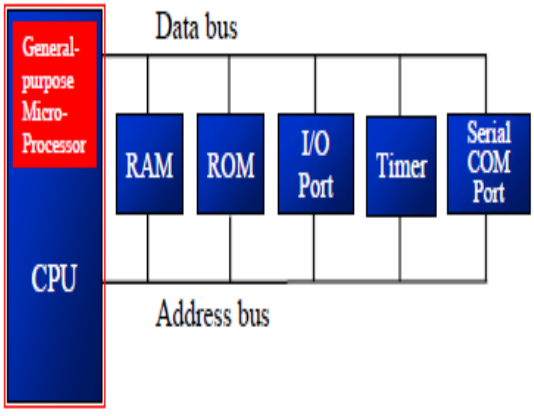
Instruction decoder

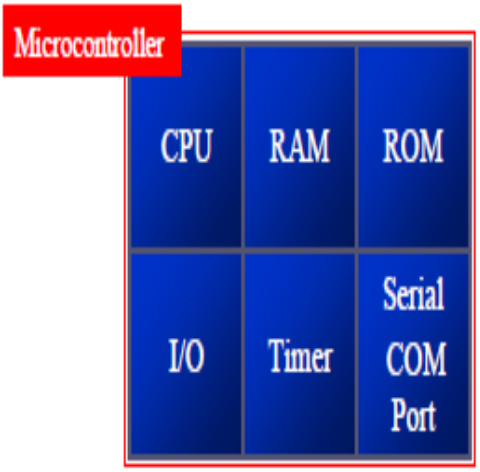
Interprets the instruction fetched into the CPU

- A CPU capable of understanding more instructions requires more transistors to design

Comparing Microprocessors and Microcontrollers

- Microprocessor is a single chip CPU, microcontroller contains, a CPU and much of the remaining circuitry of a complete microcomputer system in a single chip.
- Microcontroller includes RAM, ROM, serial and parallel interface, timer, interrupt schedule circuitry (in addition to CPU) in a single chip.
  - RAM is smaller than that of even an ordinary microcomputer, but enough for its applications.
  - Interrupt system is an important feature, for example opening of microwave oven's door cause an interrupt to stop the operation. This example is used in real-time by using microcontrollers
- Microprocessors are most commonly used as the CPU in microcomputer systems. Microcontrollers are used in small, minimum component designs performing control-oriented activities.
- Microprocessor instruction sets are _processing intensive_, implying powerful addressing modes with instructions catering to large volumes of data. Their instructions operate on nibbles, bytes, etc. Microcontrollers have instruction sets catering to the control of inputs and outputs. Their instructions operate also on a single bit. E.g., a motor may be turned ON and OFF by a 1-bit output port.





Types of Microcontrollers

Microcontrollers can be classified on the basis of internal bus width, architecture, memory and instruction set. Figure 1.4 shows the various types of microcontrollers.

The 8-Bit Microcontroller

When the ALU performs arithmetic and logical operations on a byte (8-bits) at an instruction, the microcontroller is an 8-bit microcontroller. The internal bus width of 8-bit microcontroller is of 8-bit.

Examples

The 8- Bit Microcontrollers are **Intel 8051 family and Motorola MC68HC11 family.**

The 16-Bit Microcontroller

When the ALU performs arithmetic and logical operations on a word (16-bits) at an instruction, the microcontroller is a 16-bit microcontroller. The internal bus width of 16-bit microcontroller is of 16-bit.

Examples

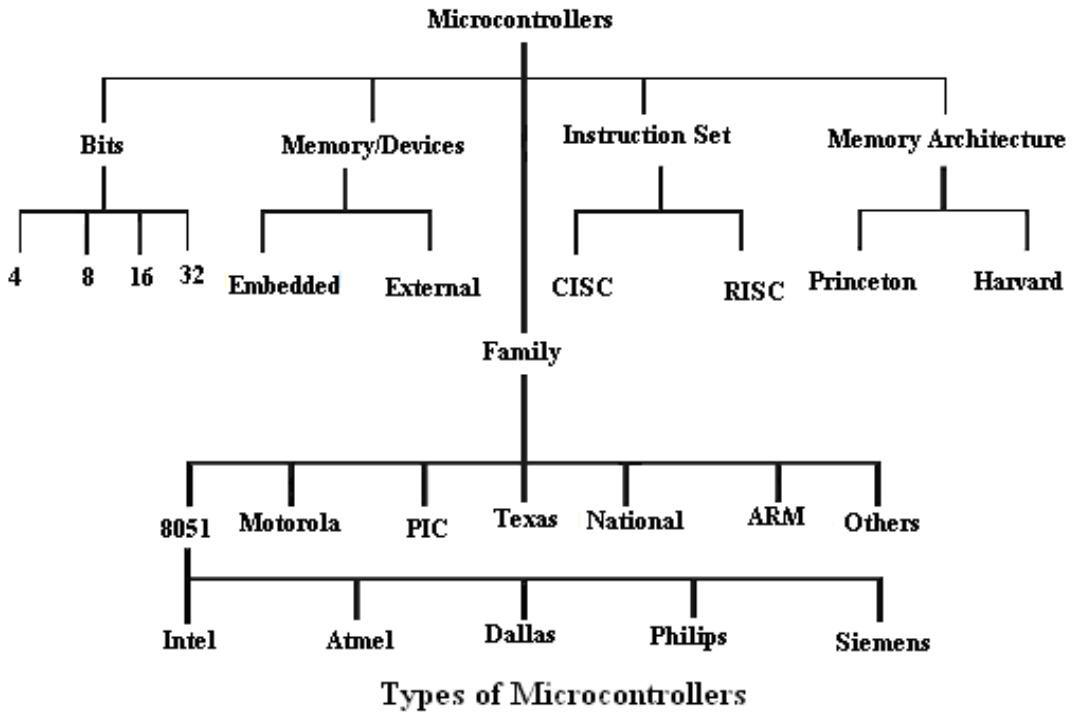
The 16-bit microcontrollers are Intel 8096 family and Motorola MC68HC12 and MC68332 families. The performance and computing capability of 16 bit microcontrollers are enhanced with greater precision as compared to the 8-bit microcontrollers.

The 32-Bit Microcontroller

When the ALU performs arithmetic and logical operations on a double word (32-bits) at an instruction, the microcontroller is a 32-bit microcontroller. The internal bus width of 32-bit microcontroller is of 32-bit.

Examples

The 32-bit microcontrollers are Intel 80960 family and Motorola M683xx and Intel/Atmel 251 family. The performance and computing capability of 32 bit microcontrollers are enhanced with greater precision as compared to the 16-bit microcontrollers.



Types of Microcontrollers

Microcontroller Architectural Features

There are mainly two categories of processors, namely, Von-Neuman (or) Princeton architecture and Harvard Architecture. These two architectures differ in the way data and programs are stored and accessed.

Von-Neuman Architecture

Microcontrollers based on the Von-Neuman architecture have a single data bus that is used to fetch both instructions and data. Program instructions and data are stored in a common main memory. When such a controller addresses main memory, it first fetches an instruction, and then it fetches the data to support the instruction. The two separate fetches slows up the controller's operation. The Von-Neuman architecture's main advantage is that it simplifies the microcontroller design because only one memory is accessed. In microcontrollers, the contents of RAM can be used for data storage and program instruction storage.

Example: Motorola 68HC11 microcontroller

Harvard Architecture

Microcontrollers based on the Harvard Architecture have separate data bus and an instruction bus. This allows execution to occur in parallel. As an instruction is being "pre-fetched", the current instruction is executing on the data bus. Once the current instruction is complete, the next instruction is ready to go. This pre-fetch theoretically allows for much faster execution than Von-Neuman architecture, on the expense of complexity. The Harvard Architecture executes the instructions in fewer instruction cycles than the Von-Neuman architecture.

Example: Intel MCS-51 family (8051), PIC microcontrollers

Microcontrollers based on instruction set it's divided in to two types

- CISC (COMPLEX INSTRUCTION SET COMPUTER)
- RISC (REDUCED INSTRUCTION SET COMPUTER)

CISC (Complex Instruction Set Computer) Architecture

Almost today's all microcontrollers are based on the CISC (Complex Instruction Set Computer) concept. When a microcontroller has an instruction set that supports many addressing modes for the arithmetic and logical instructions, data transfer and memory accesses instructions, the microcontroller is said to be of CISC architecture.

The typical CISC microcontroller has well over 80 instructions, many of them very powerful and very specialized for specific control tasks. It is quite common for the instructions to all behave quite differently. Some might only operate on certain address spaces or registers, and others might only recognize certain addressing modes.

The advantages of the CISC architecture are that many of the instructions are macro like, allowing the programmer to use one instruction in place of many simpler instructions.

Eg: Intel 8096 and 8051 family

RISC (Reduced Instruction Set Computer) Architecture

The industry trend for microprocessor design is for Reduced Instruction Set Computers (RISC) designs. When a microcontroller has an instruction set that supports fewer addressing modes for the arithmetic and logical instructions and for data transfer instructions, the microcontroller is said to be of RISC architecture. The benefits of RISC design simplicity are a smaller chip, smaller pin count, and very low power consumption.

Some of the typical features of RISC processor- Harvard architecture are

1. Allows simultaneous access of program and data.
2. Overlapping of some operations for increased processing performance.
3. Instruction pipelining increases execution speed.
4. Orthogonal (symmetrical) instruction set for programming simplicity.
5. Allows each instruction to operate on any register or use any addressing mode.

Applications

- Home monitoring System
- Automotive Appliances
  - Microwave Oven
  - Refrigerators
  - Television
  - VCRs
  - Stereos
- Automobiles
  - Engine Control
  - Diagnostics
  - Climate Control
- Environmental Control
  - Green House
  - Factory
  - Home

- Instrumentation
- Aerospace
- Robotics
- Data logging (from Sensors like Temperature, Humidity etc...)

Commercial Microcontroller Devices

Microcontrollers come in many varieties. Depending on the power and features that are needed, one might choose a 4 bit, 8 bit, 16 bit, or 32 bit microcontroller. In addition, some specialized versions are available which include features specific for communications, keyboard handling, signal processing, video processing, and other tasks. The examples of different types of commercial microcontroller devices are given in the following table

Model (Manufacturer)	I/O	Pins	RAM (bytes)	ROM (bytes)	Counters	Extra Features
8048 (Intel)	27	40	64	1K	1	8k External memory
8051 (Intel)	32	40	128	4K	2	128k External memory, Boolean processing, serial port
COP800 Family (National)	24	28	64	1K	1	Serial bit I/O, 8- channel A/D converter
6805 (Motorola)	20	28	64	1K	1	PLL frequency synthe-sizer,
68hc11(Motorola)	40	52	256	8K	2	A/D, PWM generator, pulse accumulator
TMS370 (Texas)	55	68	256	4K	2	watchdog timer, Instru-ments) Serial ports, A/D (8 bit, 8 channel)
PIC (Micro Chip)	12	18	25	1K	0	small pin count, very low power consumption

Micro-coded processor

It's a processor within a processor, or a state machine that executes each different instruction as the address to a subroutine of instructions.

Hardwired processor

A processor which uses the bit pattern of the instruction to access specific logic gates (unique to the instruction), which are executed as a combinatorial circuit to carry out the instruction

8051 Microcontroller

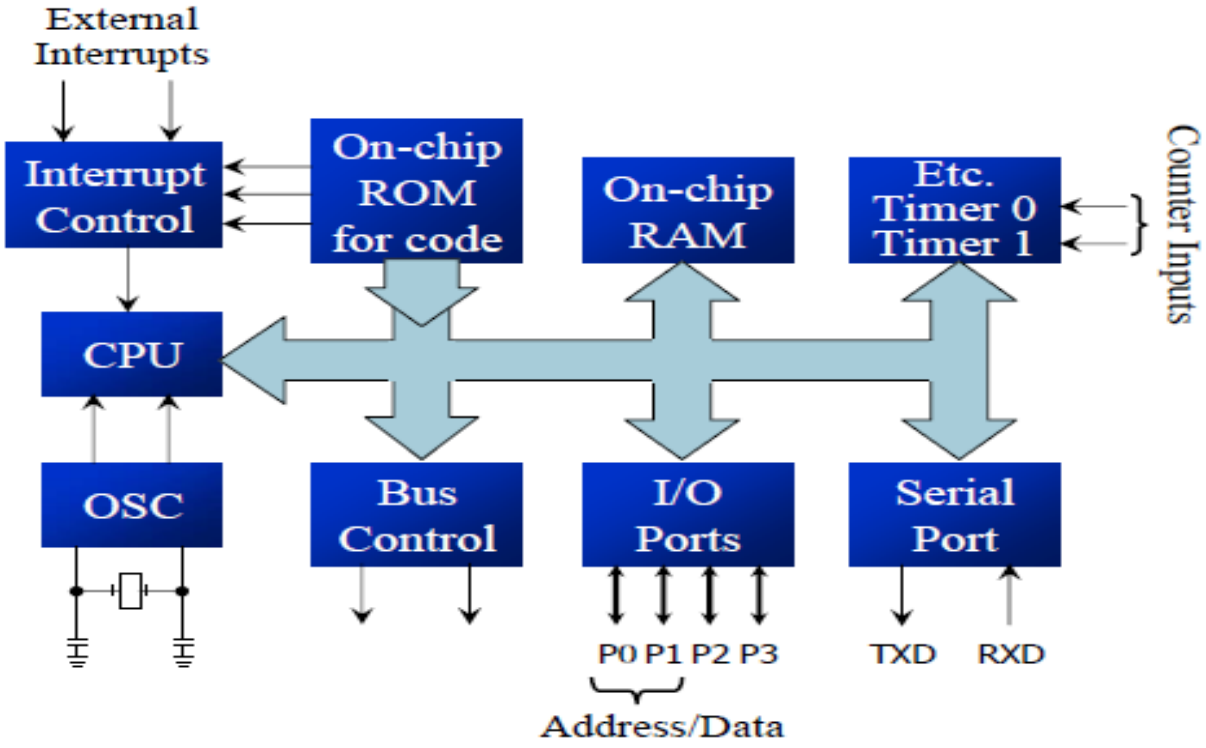
Features

The main features of 8051 microcontroller are

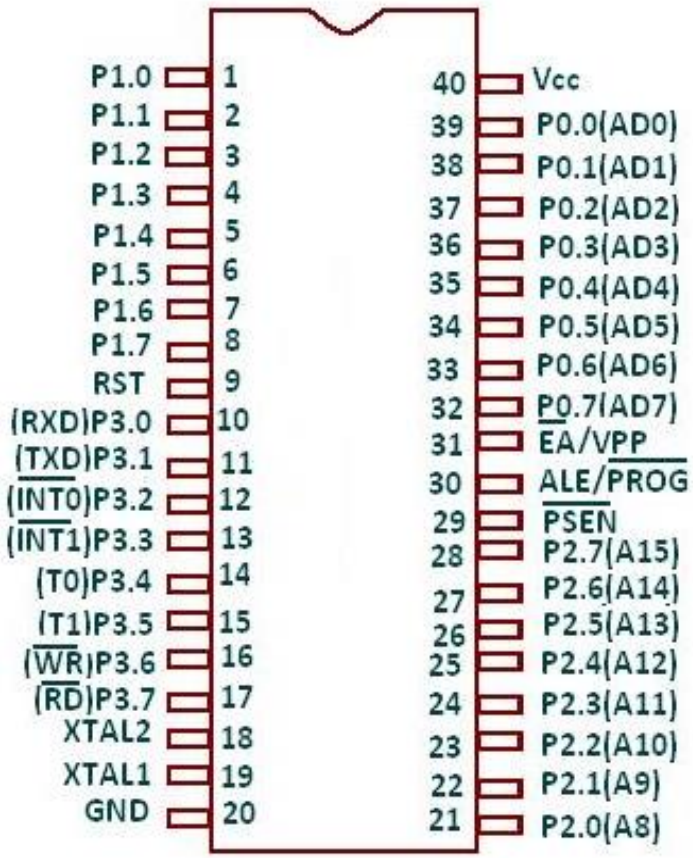
- RAM – 128 Bytes (Data memory)
- ROM – 4Kbytes (ROM signify the on – chip program space)
- 8-bit data bus – It can access 8 bits of data in one operation
- 16-bit address bus – It can access 2^{16} memory locations for dual purpose – 64 KB (65536 locations) each of RAM and ROM
- Serial Port – Using UART makes it simpler to interface for serial communication.
- Two 16 bit Counter/Timers
- Input/output Pins – 4 Ports of 8 bits each on a single chip.
- 6 Interrupt Sources
- 8 bit ALU (Arithmetic Logic Unit), Accumulator and 8-bit Registers; hence it is an 8-bit microcontroller
- Harvard Memory and CISC Architecture
- 8051 can execute 1 million one-cycle instructions per second with a clock frequency of 12MHz.
- 8051 consists of 16-bit program counter and data pointer
- 8051 also consists of 32 general purpose registers each of 8 bits

The 8051 contains a micro coded processor that is in contrast to other microcontroller that uses a hardwired one.

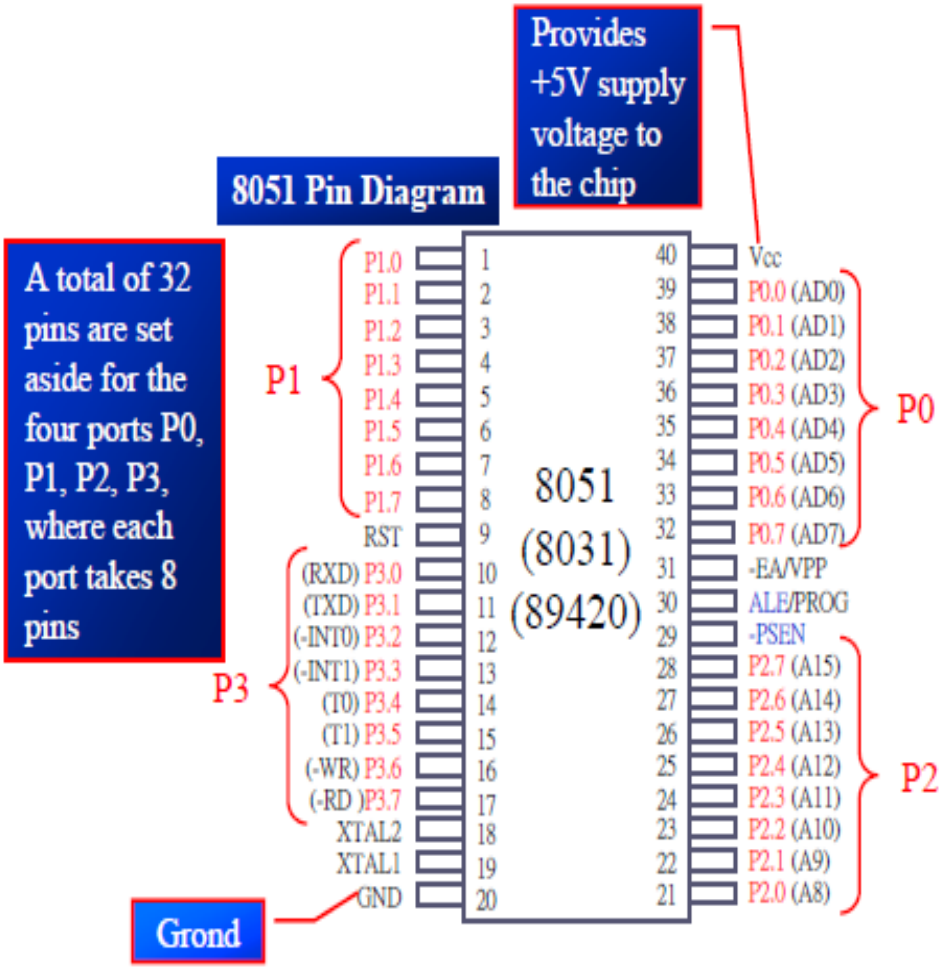
Block Diagram of 8051



8051 Pin Diagram



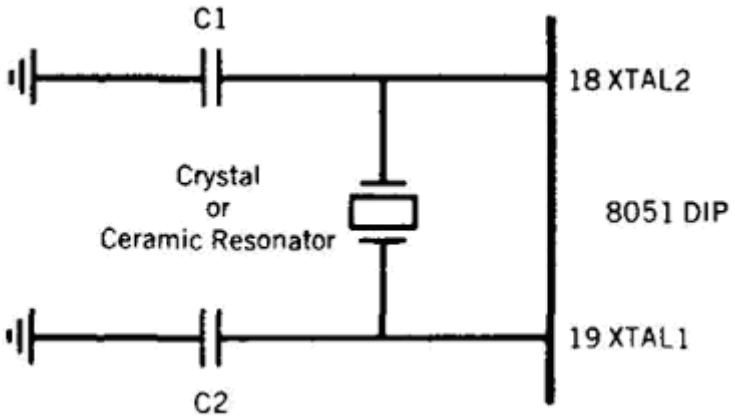
PIN Description of 8051



Crystal Circuit

The 8051 has an on-chip oscillator but requires an external clock to run it

- A quartz crystal oscillator is connected to inputs XTAL1 (pin19) and XTAL2 (pin18)
- The quartz crystal oscillator also needs two capacitors of 30 pF value



Note: 1. C1, C2 = 30 pF ± 10 pF for Crystals

= 40 pF ± 10 pF for Ceramic Resonators

RESET (RST)

RESET pin is an input and is active high (normally low)

- Upon applying a high pulse to this pin, the microcontroller will reset and terminate all activities
  - This is often referred to as a power-on reset
  - Activating a power-on reset will cause all values in the registers to be lost

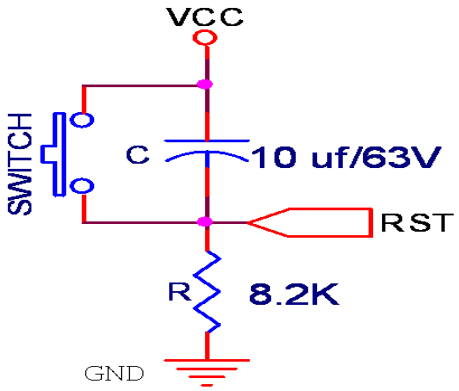
RESET value of some 8051 registers

we must place the first line of source code in ROM location 0

Register	Reset Value
PC	0000
DPTR	0000
ACC	00
PSW	00
SP	07
B	00
P0-P3	FF

In order for the RESET input to be effective, it must have a minimum duration of 2 machine cycles

- In other words, the high pulse must be high for a minimum of 2 machine cycles before it is allowed to go low



EA bar

EA, "external access'', is an input pin and must be connected to Vcc or GND

- The 8051 family members all come with on-chip ROM to store programs

  -EA pin is connected to Vcc

- The 8031 and 8032 family members do no have on-chip ROM, so code is stored on an external ROM and is fetched by 8031/32

  -EA pin must be connected to GND to indicate that the code is stored externally

The following two pins are used mainly in 8031-based systems

PSEN, "program store enable'', is an output pin

- This pin is connected to the OE pin of the ROM

ALE, "address latch enable", is an output pin and is active high

- Port 0 is also designated as AD0-AD7, allowing it to be used for both address and data
- When connecting an 8051/31 to an external memory, port 0 provides both address and data
- The 8051 multiplexes address and data through port 0 to save pins
- ALE indicates if P0 has address or data
  - When ALE=0, it provides data D0-D7
  - When ALE=1, it has address A0-A7

In 8051-based systems with no external memory connection

- Both P1 and P2 are used as simple I/O

In 8031/51-based systems with external memory connections

- Port 2 must be used along with P0 to provide the 16-bit address for the external memory
- P0 provides the lower 8 bits via A0 – A7
- P2 is used for the upper 8 bits of the 16-bit address, designated as A8 – A15, and it cannot be used for I/O

8051 Memory Organization

We will discuss about Memory Organization of Microcontroller 8051 Family. Most microprocessors implement Von Neuman architecture for memory but, the 8051 implements Harvard architecture for its memory. A separate memory space for program (code memory) and data, the architecture provides on-chip memory as well as off-chip memory expansion capabilities. The 8051 has three basic memory address spaces

- 64K-bytes of Program Memory
- 64K-bytes of External Data Program Memory
- 256-bytes of Internal Data Memory

Data Memory Address Space Memory Organization Of 8051

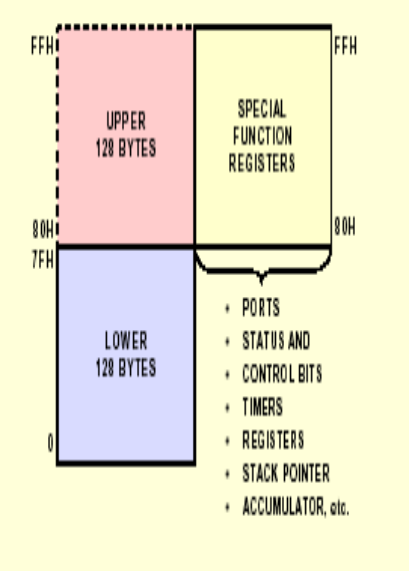
The data memory address space consists of an internal and an external memory space. Internal data memory is divided into the following three physically separate and distinct blocks

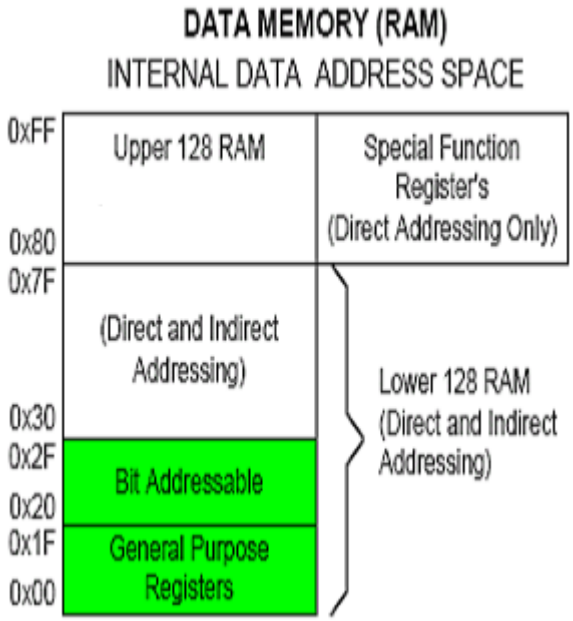
- Lower 128 bytes of RAM
- Upper 128 bytes of RAM (accessible in the 8032/8052 only)
- 128-bytes of Special Function Register (SFR) area

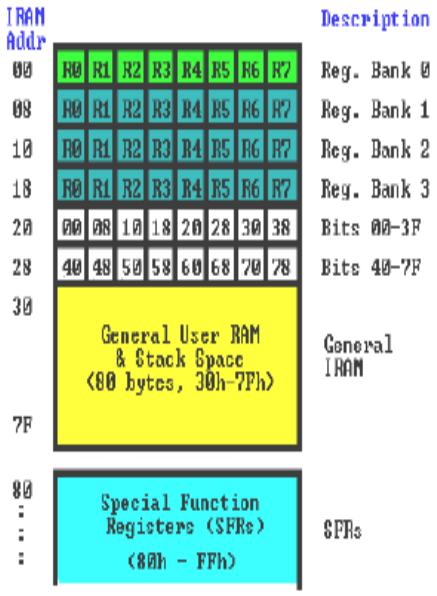
Lower 128 bytes of RAM address range is 00H to 7FH.How can manufacturer divided these 128 bytes as given in below table:

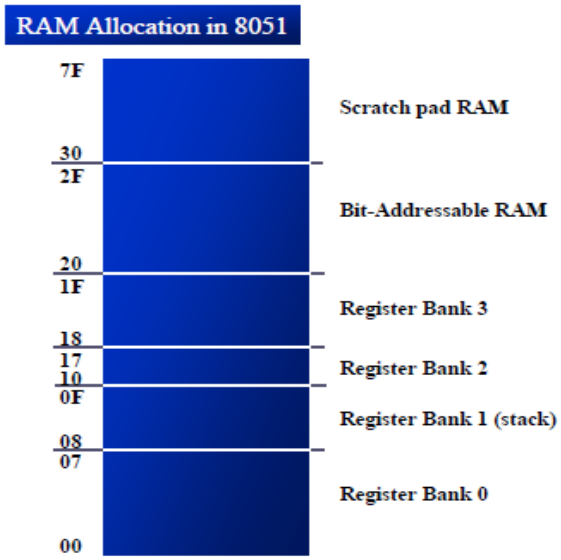
RAM AREA NAME	ADDRESS RANGE	TOTAL(in bytes)
Register banks and the Stack	00H – 1FH	32
Bit Addressable Area	20H – 2FH	16
Scratch Pad Area	30H – 7FH	80

The upper RAM area and the SFR (Special Function Register) area share the same address locations. They are accessed through different addressing modes. Any location in the general-purpose RAM can be accessed freely using the direct or indirect addressing modes.





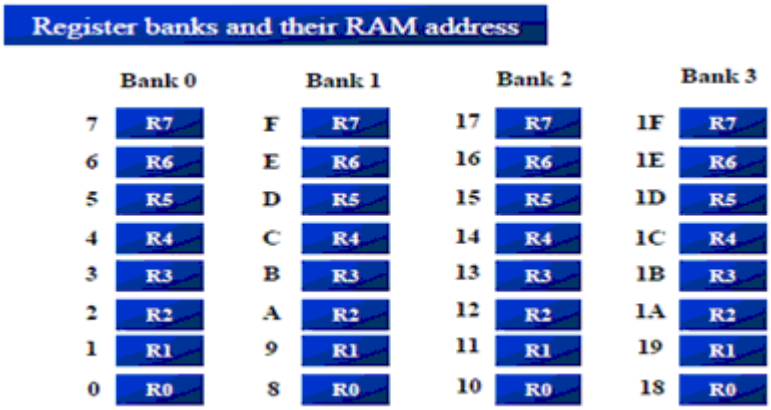




Register Banks in the 8051

Manufacturer allotted 32 bytes for Register Banks. These 32 bytes are divided into 4 banks of registers in which each bank has 8 registers, named as R0-R7.Each register it takes 1 byte. So, each bank occupies 8 bytes.

How can they give the address to that registers in each and every bank it shows in below diagram.



Bank 1 uses the same ram space as the stack. Register bank 0 is the default when the 8051 is powered up. We can switch to other banks by use of the PSW (Program status word) register. PSW is one of the SFR (Special Function Register) and also this one is bit addressable register. So, we can access bit addressable instructions SETB and CLR. In this register PSW.3 and PSW.4 bits are used to select the desired register bank as shown in below table.

Bank	RS1 (PSW.4)	RS0 (PSW.3)
BANK 0	0	0
BANK 1	0	1
BANK 2	1	0
BANK 3	1	1

If we are using CLR PSW.x instruction then it makes zero value in that particular bit, and if we are using SETB PSW.x instruction then one(1) value passes in that particular bit. Where ‘x’ represents either 3 (or) 4

Special Function Registers (SFRS) Memory Organization Of 8051

- Special Function Registers (SFRs) are areas of memory that control specific functionality of the 8051 processor. SFRs are accessed as if they were normal Internal RAM. The only difference is that Internal RAM is from address 00H through 7FH whereas SFR registers exist in the address range of 80H through FFH. Each SFR has an address (80H through FFH) and a name.
- SFRs related to the I/O ports. The 8051 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by these SFRs.
- Total SFR memory is 128 bytes in that manufacturer allotted 21 bytes for 21 registers. Each and every register is used for some specific application. That's why these registers called as a Special Function Registers.
- In total 21 SFRs only 11 SFRs are Bit – Addressable SFRs and these SFRs also Byte Addressable SFRs
- Total 21 SFRs are Byte Addressable Registers.
- SFRs which in some way control the operation or the configuration of some aspect of the 8051.For example, TCON controls the timers, SCON controls the serial port. The remaining SFRs are that they don't directly configure the 8051 but obviously the 8051 cannot operate without them. For example, once the serial port has been configured using SCON, the program may read or write to the serial port using the SBUF register. In that 21 registers some of the registers explanation as given in below.

F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8									CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2								A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH				PCON	87

Bit-addressable Registers

PORT 0, PORT1, PORT2, PORT3 (BIT-ADDRESSABLE Registers)

PORT0 is input/output port 0. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin P0.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level like PORT1(P1),PORT2(P2),PORT3(P3) also having same operation, addresses are different those are available in above table.

SP (STACK POINTER, ADDRESS 81H)

SP means Stack Pointer used to access the stack. This SFR indicates where the next value to be taken from the stack will be read from in Internal RAM. If we push a value onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the value 07H, a PUSH instruction will push the value onto the stack at address 08H. This SFR is modified by all instructions, which modify the stack, such as PUSH, POP, and LCALL, RET, RETI, and whenever interrupts are provoked by the micro controller.

DPL/DPH (DATA POINTER LOW/HIGH, ADDRESS 82H/83H)

The SFRs DPL and DPH work together to represent a 16-bit value called the Data Pointer. The data pointer is used in operations regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte integer value, it can represent values from 0000H to FFFFH (0 through 65,535 decimal).

PCON (POWER CONTROL, ADDRESS 87H)

The Power Control SFR is used to control the 8051's power control modes. Additionally, one of the bits in PCON is used to double the effective baud rate of the 8051's serial port.

TCON (TIMER CONTROL, ADDRESS 88H, BIT-ADDRESSABLE) of 8051

The Timer Control SFR is used to configure and modify the way in which the 8051's two timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags which are set when an external interrupt has occurred.

TMOD (TIMER MODE, ADDRESS 89H)

The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, an 8-bit auto reload timer, a 13-bit timer, or two separate timers. Additionally, we may configure the timers to only count when an external pin is activated or to count "events" that are indicated on an external pin.

SCON (SERIAL CONTROL, ADDRESS 98H, BIT-ADDRESSABLE)

The Serial Control SFR is used to configure the behavior of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

SBUF (SERIAL CONTROL, ADDRESS 99H)

The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Likewise, any value that the 8051 receive via the serial port's RXD pin, will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from.

IE (INTERRUPT ENABLE, ADDRESS A8H)

The Interrupt Enable SFR is used to enable and disable specific interrupts. The lower 7 bits of the SFR are used to enable/disable the specific interrupts, where as the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

IP (INTERRUPT PRIORITY, ADDRESSES B8H, BIT-ADDRESSABLE)

The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority.

PSW (PROGRAM STATUS WORD, ADDRESSES D0H, BIT-ADDRESSABLE)

The Program Status Word is used to store a number of important bits that are set and cleared by 8051 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags, which are used to select, which of the "R" register banks are currently selected.

ACC (ACCUMULATOR, ADDRESSES E0H, BIT- ADDRESSABLE)

The Accumulator is one of the most-used SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR at 0E0h, which means the instruction MOV A, #20h is really the same as MOV 0E0h, #20h. However, it is a good idea to use the first method since it only requires two bytes whereas the second option requires three bytes.

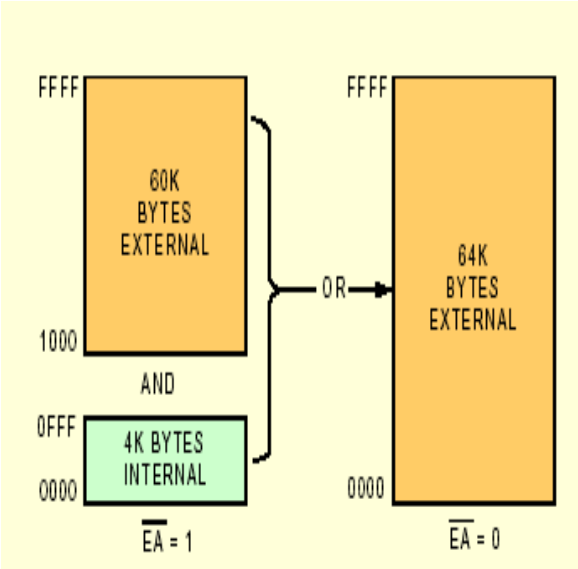
B (B REGISTER, ADDRESSES F0H, BIT-ADDRESSABLE)

The "B" register is used in two instructions: the multiply and divide operations. Programmers also commonly use the B register as an auxiliary register to temporarily store values.

Program Memory Address Space (ROM) Of 8051

The 64K-byte Program Memory space consists of an internal and an external memory portion. If the External Access (EA) pin is held high, the 8051 executes 4K bytes as internal memory address range is 0000H-0FFFH and locations 1000H through FFFFH (60 bytes) are fetched from external program memory (or) If the External Access (EA) pin is held low, the 8051 executes total 64K bytes as external memory address range is 0000H-FFFFH. Locations 00 through 23H in program memory are used by the interrupt service routines.

Program Memory



Assembly Language

An assembly language is a low-level programming language for a computer, microcontroller, or other programmable device, in which each statement corresponds to a single machine code instruction. Each assembly language is specific to particular computer architecture, in contrast to most high-level programming languages, which are generally portable across multiple systems.

An assembler creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities.

Assembly language uses a mnemonic to represent each low-level machine operation or opcode. Some opcodes require one or more operands as part of the instruction, and most assemblers can take labels and symbols as operands to represent addresses and constants, instead of hard coding them into the program.

A program written in assembly language consists of a series of (mnemonic) processor instructions and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by a list of data, arguments or parameters. These are translated by an assembler into machine language instructions that can be loaded into memory and executed.

Syntax

- **LABEL: MNEMONIC OPERANDS ; COMMENTS**

Example: MOV destination, source; copy source to dest.

- The instruction tells the CPU to move (in reality, COPY) the source operand to the destination operand

Register are used to store information temporarily, while the information could be

- a byte of data to be processed, or
- an address pointing to the data to be fetched

The vast majority of 8051 register are 8-bit registers

- There is only one data type, 8 bits

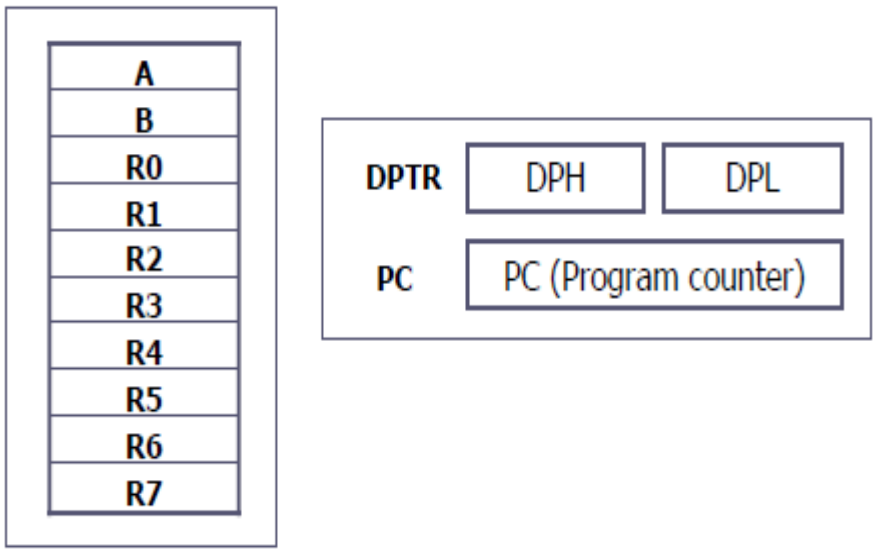
The 8 bits of a register are shown from MSB D7 to the LSB D0

- With an 8-bit data type, any data larger than 8 bits must be broken into 8-bit chunks before it is processed.

The most widely used registers

- A (Accumulator)

  - For all arithmetic and logic instructions
- B, R0, R1, R2, R3, R4, R5, R6, R7
- DPTR (data pointer), and PC (program counter)



The steps of Assembly language program are outlines as follows:

1) First we use an editor to type a program, many excellent editors or word processors are available that can be used to create and/or edit the program

- Notice that the editor must be able to produce an ASCII file
- For many assemblers, the file names follow the usual DOS conventions, but the source file has the extension "**asm**" or "**src**", depending on which assembly you are using

2) The “asm” source file containing the program code created in step 1 is fed to an 8051 assembler

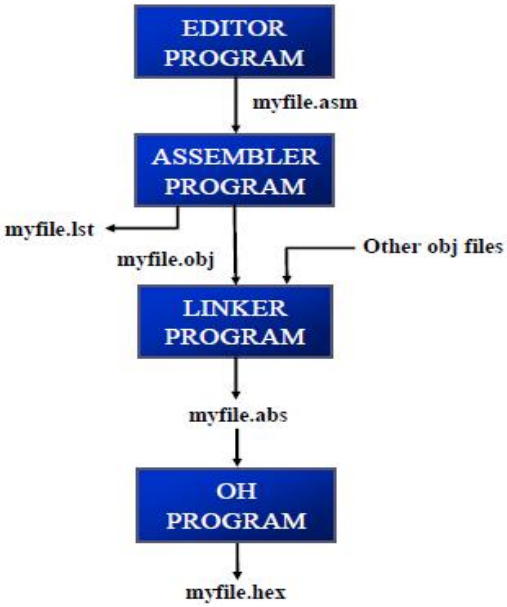
- The assembler converts the instructions into machine code
- The assembler will produce an object file and a list file
- The extension for the object file is “**obj**” while the extension for the list file is “**lst**”

3) Assembler require a third step called linking

- The linker program takes one or more object code files and produce an absolute object file with the extension “abs”
- This abs file is used by 8051 trainers that have a monitor program

4) Next the “abs” file is fed into a program called “OH” (object to hex converter) which creates a file with extension “hex” that is ready to burn into ROM

- This program comes with all 8051 assemblers
- Recent Windows-based assemblers combine step 2 through 4 into one step



Addressing Modes

It is a way to address an operand i.e., an "addressing mode" refers to how you are addressing a given memory location. Total 5 types of addressing modes are available in 8051 those are given in below.

The MCS-51 instruction set offers several addressing modes, including

- direct register, using ACC (the accumulator) and R0-R7
- direct memory, which access the internal RAM or the SFRs, depending on the address
- Indirect memory, using R0, R1, or DPTR to hold the memory address. The instruction used may vary to access internal RAM, external RAM, or program memory.
- individual bits of a range of IRAM and some of the SFR's

Types of addressing modes

- Immediate Addressing Mode
- Register Addressing Mode
- Direct Addressing Mode
- Indirect Addressing Mode
- Indexed Addressing Mode

To Understand the Addressing Modes we are using MOV instruction that instruction explained in Data Transfer Instruction Set in below

Examples for Addressing Modes using data transfer instruction set

Immediate Addressing	MOV A,#20h
Direct Addressing	MOV A,30h
Indirect Addressing	MOV A,@R0
External Direct	MOVX A,@DPTR
Code Indirect	MOVC A,@A+DPTR

Each of these addressing modes provides important flexibility.

Immediate Addressing Mode

- Specify data by its value
- The value of a constant can follow the opcode in program memory
- The # sign is the designator of the constant

Possibilities

- MOV A, #data(8-bit)
- MOV Rn, #data(8-bit) ; n=0 - 7
- MOV DPTR, #data (16-bit)
- MOV @Ri, #data(8-bit) ; i=0/1

Examples

MOV A, #0 ; put 0 in the accumulator

A = 00000000

MOV R0, #0x11 ; put 11hex in the R0 (BANK 0)

R0 = 00010001

MOV @R0, #11 ; put 11 decimal in to internal RAM with address contained in R0

11h= 00001011

MOV DPTR, #77h ; put 77 hex in DPTR

DPTR=77h

Direct Addressing mode

- Specify data by its 8-bit address
- At least one of the operand is specified by an 8 bit address field
- Internal data RAM and SFRs can be directly addressed

Possibilities

- MOV Addr, #data(8-bit)
- MOV A, Addr
- MOV Addr, A
- MOV Rn, Addr
- MOV Addr, Rn
- MOV Addr1, Addr2

Examples

MOV A, 0x70 ; copy contents of RAM at 70h to "A"

MOV 0xD0, A ; put contents of a into PSW I.E., PSW address is 0Xd0

Register Addressing Mode

Either source or destination is one of R0-R7 and another one is accumulator

Possibilities

- MOV R0, A
- MOV A, R0

Examples

MOV R0, #34h ; put 34h value in R0 register

MOV A, R0 ; put contents of R0 in to "A" register

Indirect Addressing Mode

- The instruction specifies a register that contains the address of the operand
- Both Internal and External RAM can be indirectly addressed
- Address register can be R0 or R1 of a selected register bank or the SP for 8 bit and DPTR for 16 bit address

Possibilities for internal RAM

- MOV @Ri, A ; i=0/1
- MOV A, @Ri ; i = 0/1
- MOV @Ri, ADDR
- MOV ADDR, @Ri

Examples

Uses register R0 or R1 for 8-bit address

MOV 0xD0, #0 ; use register bank 0

MOV R0, #0x3C ; put 3Ch value in R0 register

MOV A, #34h ; put 34h value accumulator

MOV @R0, #3 ; memory at 3C gets #3(it's a decimal value) i.e., 3

; M [3C] ← 3

MOV @R0, A ; memory at 3C gets contents of 'A' register i.e., M [3C] ← 34h

Indirect (External RAM)

- All external data moves must involve A register
- Rp, where p=0/1 can address 256 bytes; DPTR can address 64K bytes
- MOVX is normally used with external RAM or I/O address.

Possibilities

- MOVX @DPTR,A
- MOVX A,@DPTR

Examples

MOV DPTR, #4000h ; DPTR ← 4000h

MOV A, #5h ; A ← 5h

MOVX @DPTR, A ; M [4000] ← A

MOVX A, @DPTR ; A ← M [4000]

- Note that 9000 is an address in external memory and 'A' is an accumulator register.

Indexed Addressing Mode

- Source or destination address is the sum of the base address and the accumulator.
- This Addressing Mode is only for code memory read.
- Base address can be or PC

Possibilities

- MOVC A,@A + DPTR
- MOVC A,@A + PC

Examples

MOV DPTR, #4000h

MOV A, #5h

MOVC A, @A+ DPTR; A ← M [4005]

Instruction Set

Instruction set is a collection of Instructions are used in 8051 to perform specific applications. Total 4 types of Instruction Sets are available in 8051 those are discussed in below.

Types of Instruction Sets

- Data Transfer Instruction Set
- Arithmetic Instruction Set
- Logical Instruction Set
- Branch Instruction Set

Data Transfer Instruction Set

Is used for transfer the data from one place to another place this instruction set having one mnemonic and two operands.

In these two operands one is source operand and another one is destination operand. In this instruction set mnemonic is "MOV", operands are either constant data, registers, addresses or indirect registers those examples are given below.

Syntax: MOV Destination, Source ; destination ← source

According to above syntax whatever the data in source operand that is moved on to the destination operand

Examples

MOV a, #data ; move data to accumulator

MOV addr, a ; move accumulator to addr

MOV Rn, addr ; move data of that address to register of current bank (Where n=0-7)

MOV addr1, addr2; move data in that addr to internal RAM

MOV @Ri,#data; move data to internal RAM with address contained in Ri (where i=0/1)

MOV DPTR, #data (16-bit); move 16-bit data into data pointer

Above examples are transfers the data from source to destination and those operands are related to internal ram section only. Above examples and remaining examples of data transfer instruction set explanation as given in below (In Addressing modes Examples).

A and B are Registers

A, B registers are used for arithmetic instructions. They can be accessed by direct mode as special function registers

B – Address 0F0h

A – Address 0E0h - use "ACC" for direct mode

Stack-oriented data transfer – another form of register indirect addressing, but using SP

Examples

MOV SP, #0x40 ; Initialize SP

PUSH 0x55 ; SP ← SP+1, M [SP] ← M [55] i.e., M [41] ← M [55]

POP B ; B ← M [55], SP ← SP-1

The stack is a section of RAM used by the CPU to store information temporarily

- This information could be data or an address

The register used to access the stack is called the SP (stack pointer) register

- The stack pointer in the 8051 is only 8 bit wide, which means that it can take value of 00 to FFH
- When the 8051 is powered up, the SP register contains value 07
    - RAM location 08 is the first location begin used for the stack by the 8051

The storing of a CPU register in the stack is called a PUSH

- SP is pointing to the last used location of the stack
- As we push data onto the stack, the SP is incremented by one
    - This is different from many microprocessors

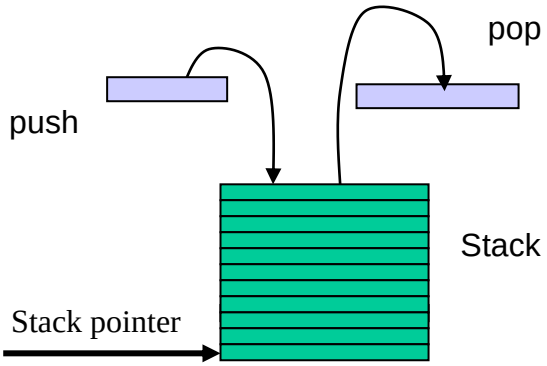
Loading the contents of the stack back into a CPU register is called a POP

- With every pop, the top byte of the stack is copied to the register specified by the instruction and the stack pointer is decremented once

Note: can only specify RAM or SFRs (direct mode) to push or pop. Therefore, to push/pop the accumulator, must use acc, not a.

Possibilities

PUSH acc



Exchange Instructions

- two way data transfer i.e., move data in two directions
- All exchanges are internal to 8051
- All exchange instructions use “Accumulator”
- When using XCHD, the upper nibbles are not exchanged

Possibilities

- XCH A, Rn --> **n=0-7**
- XCH A, addr
- XCH A, @Ri --> **i=0(or)1**
- XCHD A, @Ri

Examples

XCH A, 0x30 ; A →← M [30]

XCH A, R0 ; A →← R0

XCH A, @R0 ; A →← M [R0]

XCHD A, @R0 ; exchange lower nibbles of ‘A' register and M [R0] register.

Data Processing Instructions

- Arithmetic Instruction Set
- Logical Instruction Set

Arithmetic Instruction Set

- Add
- Subtract
- Increment
- Decrement
- Multiply
- Division
- Decimal adjust

Mnemonic	Description
ADD A, #data	add A to byte, put result in A
ADDC A, addr	add A, data in address with carry
SUBB A, addr	subtract A, data in address with borrow
INC A	increments the data by 1 in “A” register
INC addr	increment data by 1 in memory(address)
INC DPTR	increment data by 1 in data pointer
DEC A	decrement the data by 1 in accumulator
DEC addr	decrement the data by 1 in addr
MUL AB	multiply accumulator with b register
DIV AB	divide accumulator by b register
DA A	decimal adjust the accumulator

Options

ADD	A, #data (8bit)
ADDC	A, Rn
SUBB	A, direct_addr
	A, @Ri

Examples

ADD A, byte ; A ← A + byte

ADDC A, byte ; A ← A + byte + C

These instructions affect 3 bits in PSW:

C = 1 if result of add is greater than FF

AC = 1 if there is a carry out of bit 3

OV = 1 if there is a carry out of bit 7, but not from bit 6, or vice versa.

Program Status Word (PSW)

Bit	7	6	5	4	3	2	1	0
Flag	CY	AC	F0	RS1	RS0	OV		P
Name	Carry Flag	Auxiliary Carry Flag	User Flag 0	Register Bank Select 1	Register Bank Select 0	Overflow flag		Parity Bit

Q) What is the value of the C, AC and an OV flags after the second Instruction is executed for below example?

MOV A, #0x3F

ADD A, #0xD3

Ans:

	0011 1111	C = 1
	1101 0011	AC = 1
	0001 0010	OV = 0

Subtract

SUBB A, #data	subtract with borrow

Example:

SUBB A, #0x4F ; A ← A – 4F – C

- Notice that there is no subtraction WITHOUT borrow. Therefore, if a subtraction without borrow is desired, it is necessary to clear the C flag.

Increment and Decrement

Options

INC A

DEC Rn

direct_addr

@Ri

INC DPTR

- The increment and decrement instructions do not affect the C flag.
- Notice we can only INCREMENT the data pointer, not decrement.

Multiply

When multiplying two 8-bit numbers, the size of the maximum product is 16-bits

FF x FF = FE01 i.e., in decimal (255 x 255 = 65025)

MUL AB ; A * B (Result is stored like below format)

A ← Lower byte, B ← Higher byte

Note: When we are getting more than 8 bits (1 byte), then B gets the HIGH byte, A gets the LOW byte

Division

DIV AB ; divide A by B

A ← Quotient (A/B), B ← Remainder (A/B)

OV - used to indicate a divide by zero condition.

C – Set to zero

Decimal Adjust

DA adjusts the contents of the Accumulator to correspond to a BCD (Binary Coded Decimal) number after two BCD numbers have been added by the ADD or ADDC instruction. If the carry bit is set or if the value of bits 0-3 exceeds 9, 0x06 is added to the accumulator. If the carry bit was set when the instruction begins (or) if 0x06 was added to the accumulator in the first step, 0x60 is added to the accumulator. The **Carry bit (C)** is set if the resulting value is greater than 0x99, otherwise it is cleared.

Syntax: DA A ; decimal adjust A

Examples:

```
MOV A, #0x23
MOV B, #0x29
ADD A, B    ; A ← 23 + 29 = 4C
DA A        ; A ← A + 6 = 52
```

Note: This instruction does not convert binary to BCD. In the above example Accumulator gets 4c but BCD values are 0-9. So, in lower nibble we got “C” hexadecimal that’s why we have to add 0x06 to the number. For suppose if we will get “A-F” hexa decimal values in Higher nibble we have to add 6 to higher nibble means add with 0x60. Both are possible by using “DA” instruction only. When we are u8sing DA instruction automatically it added “6” value to that hexadecimal number (A-F).

Logical Instruction set

- Bitwise logic operations (AND, OR, XOR, NOT)
- Clear
- Rotate
- Swap
- Logic instructions do NOT affect the flags in PSW

ANL – AND
ORL – OR
XRL – exclusive OR
CPL – Complement

Examples: ANL 00001111
10101100
00001100

ORL	00001111	**XRL**	00001111	**CPL**	10101100
	10101100		10101100		01010011
	10101111		10100011		

Q) Write a program to save the accumulator in R7 of bank 2.

Solution:

```
CLR PSW.3
SETB PSW.4
MOV R7, A
```

Byte Level Operations

Options

ANL	A, #n & A, @Rp
ORL	A, add & addr, A
XRL	A, Rn & Addr, #n

- **ANL** -Both inputs are high then only output is high
- **ORL** -Any one of the input is high then only output is high
- **XRL** -Both inputs are same output is low vice versa.

CPL

- It complements the value in a register i.e., when this instruction is executed it makes zeros to ones and ones to zeros.
- We are using only one operand in this instruction i.e., Accumulator. This register only acts as both source and destination operand final result is stored in 'a' register itself.

Syntax: CPL A

Other Logical Instructions

- CLR - clear
- RL – rotate left
- RLC – rotate left through Carry
- RR – rotate right
- RRC – rotate right through Carry
- SWAP – swap accumulator nibbles

CLR

Set all bits to 0

Syntax: CLR A

CLR bit_addr

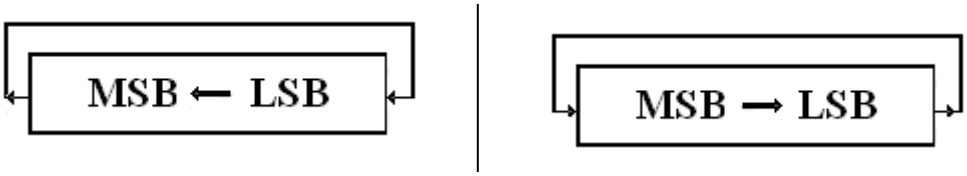
Rotate

Rotate instructions operate only on a. We are using only one operand in below instructions i.e., Accumulator. This register only acts as both source and destination operand final result is stored in 'a' register itself.

Syntax: RL A (rotate the value in A register to left side)

RR A (rotate the value in A register to right side)

RL **RR**



Example

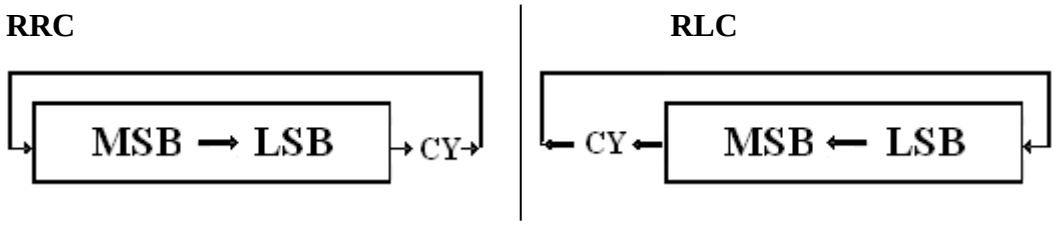
MOV A, #0xF0 ; A ← 11110000

RL A ; A ← 11100001

Rotate through Carry

Syntax: RLC A (rotate the value in A register to left side through carry flag)

RRC A (rotate the value in A register to right side through carry flag)



Example

MOV A, #0A9h ; A ← A9

ADD A, #14h ; A ← BD (10111101), C←0

RRC A ; A ← 01011110, C←1

Swap

This instruction swaps the data in A register i.e., it changes lower nibbles to higher nibbles and higher nibbles to lower nibbles

Syntax: SWAP A

Example:

MOV A, #72h ; A ← 72h

SWAP A ; A ← 27h

Bit Logic Operations

Some logic operations can be used with single bit operands

Options

SETB C

CLR C

CPL C

CLR bit_address

SETB bit_address

CPL bit_address

ANL C, bit_address

ORL C, bit_address

Note: "bit_address" can be any of the bit-addressable RAM locations or SFRs.

Rotate and Multiplication/Division

- Note that a shift left is the same as multiplying by 2, shift right is divide by 2

```
MOV A, #3       ; A← 00000011 (3H)
CLR C           ; C← 0
RLC A           ; A← 00000110 (6H)
RLC A           ; A← 00001100 (12H)
RRC A           ; A← 00000110 (6H)
```

Shift/Multiply Example

- Program segment to multiply by 2 and add 1

```
        CLR C
        RL A        ; multiply by 2
        INC A       ; and add one
```

Branch Instruction Set

- Unconditional jumps
- Conditional jumps
- Call and return

Unconditional Jumps

- SJMP <rel addr> - Sort jump, relative address is 8-bit 2's complement number, so jump can be up to 127 locations forward, or 128 locations back.
- LJMP <address 16> - Long jump
- AJMP <address 11> - Absolute jump to anywhere within 2K block of program memory
- JMP @A + DPTR - Long indexed jump

Infinite Loops

Example

Start: MOV C, P3.7

MOV P1.6, C

SJMP **Start**

Microcontroller application programs are almost always infinite loops!

Re-locatable Code

Start: MOV C, P1.6

MOV P3.7, C

SJMP **Start**

Conditional Jump

These instructions cause a jump to occur only if a condition is true. Otherwise, program execution continues with the next instruction.

loop: MOV A, P1

JZ **loop** ; if a=0, go to loop, else go to next instruction

MOV B, A

Syntax

Mnemonic	**Description**
JZ <rel addr>	Jump if a = 0
JNZ <rel addr>	Jump if a != 0
JC <rel addr>	Jump if C = 1

JNC <rel addr>	Jump if C != 1
JB <bit>, <rel addr>	Jump if bit = 1
JNB <bit>,<rel addr>	Jump if bit != 1
JBC <bit>, <rel addr>	Jump if bit =1, clear bit
CJNE A, direct, <rel addr>	Compare A and memory, jump if not equal

Conditional Jumps for Branching

If condition is true go to label else go to next instruction

Example: If C is 1 led should be ON else led should be off i.e., when C is zero by using conditional jump instruction

JZ **led_off**

SETB C

MOV P1.6, C

SJMP **skipover**

led_off: CLR C

MOV P1.6, C

skipover: MOV A, P0

More Conditional Jumps

Mnemonic	**Description**
CJNE A, #data <rel addr>	Compare A and data, jump if not equal
CJNE Rn, #data <rel addr>	Compare Rn and data, jump if not equal
CJNE @Rn, #data <rel addr>	Compare Rn and memory, jump if not equal
DJNZ Rn, <rel addr>	Decrement Rn and then jump if not zero
DJNZ direct, <rel addr>	Decrement memory and then jump if not zero

Examples

1) Increment a value up to 4 2) Decrement R0 value from 4 to 0 and increment A value

```
CLR A                    MOV R0, #4
loop:                    loop:  INC A
CJNE A, #4, loop         DJNZ R0, loop
```

Call and Return

Call is similar to a jump, but

- Call instruction pushes PC on stack before branching
- Allows RETURN back to main program

Absolute call: (2 BYTE INSTRUCTIUON) ACALL <address ll> ; stack ← PC
; PC ← address 11

Long call: (3 BYTE INSTRUCTIUON) LCALL<address 16> ; stack ← PC
; PC ← address 16

Return (RET)

- Return instruction pops PC from stack to get address to jump to

Syntax

```
RET        ; PC ← stack
```

Initializing Stack Pointer

- The Stack Pointer (SP) is initialized to 0x07. (Same address as R7)
- When using subroutines, the stack will be used to store the PC, so it is very important to initialize the stack pointer. Location 2F is often used.

```
MOV SP, #0x2F
```

Sub Routines

- Subroutines allow us to have "structured" assembly language programs.
- This is useful for breaking a large design into manageable parts.
- It saves code space when subroutines can be called many times in the same program.

Assembler Directive (or) pseudo–instructions

- Assembler specific Keywords which instruct the assembler how to process subsequent assembly language instructions.

- **Assembler Directives**
  - Instructions for the ASSEMBLER
  - NOT 8051 instructions
  - Give direction to the assembler

Categories of Directives in A51 assembler from KEIL IDE

- **Segment Control**

  Generic Segments: **SEGMENT, RSEG**

  Absolute Segments: **CSEG, DSEG, BSEG, ISEG, XSEG**
- **Symbol Definition**

  Generic Symbols: **EQU, SET**

  Address Symbols: **BIT, CODE, DATA, IDATA, XDATA**

  SFR Symbols: **sfr, sfr16, sbit**
- **Memory Initialization**

  DB, DW, DD
- **Memory Reservation**

  DBIT, DS, DSB, DSW, DSD
- **Procedure Declaration**

  PROC/ENDP, LABEL
- **Program Linkage**

  PUBLIC, EXTRN/EXTRN, NAME
- **Address Control**

  ORG, EVEN, USING

Note: From above assembler directives few are discussed in below with examples

Segment Control Directives

A segment is a block of code or data memory the assembler creates from code or data in an 8051 assembly source file. They allow user to locate program code, constant data, and variables in specific memory areas of 8051 CPU.

- CSEG – Stands for code segment
- Segment, rseg - it combines two ASM files

Syntax

- **CSEG** AT Address
- user_seg_name **segment** class_symbol

  rseg user_seg_name

Example

```
CSEG AT 1000h          ; address of next instruction is 1000h
delay segment code     ; this one we have to written in sub header file then only
rseg delay             ; this file added to main file
```

Symbol definition directives

EQU

Used to create symbols that can be used to represent registers, numbers, and addresses

Syntax

Symbol **EQU** constant

Example:

```
GREEN_LED EQU P1.6      ; symbol for Port 1, bit 6
```

DATA

Used to define a name for memory locations

Syntax

Symbol **DATA** data_address

Example

```
SP      DATA   0x81      ; special function registers
MY_VAL DATA   0x44      ; RAM location
```

data_address

- is a data memory address in the range 0 to 127
- Or a special function registers (SFRs) address in the range 128 - 255.

BIT

Used to define a name for memory locations of particular bits

Syntax

Symbol **BIT** bit_address

Example: B0 **BIT** 20H.0

bit_address:

- Is the address of a bit in internal data memory in the area 20H-2FH
- Or a bit address of an 8051 bit-addressable SFR.

Memory initialization directives

DB

These are used to initialize code or constant space in byte units.

Syntax **usr_symbol:** DB 8-bit_constant

DB 8_bit_constants_with_comma

DB 'string'

Example

char: **DB** 'a'

STR: **DB** "VECTOR"

Num: **DB** 25, 36, 45

Other

- The END directive signals the end of the assembly module.
- Any text in the assembly file that appears after the END directive is ignored while assembling.

MACRO

Macro's in A51:

- A macro is a name that you assign to one or more assembly statements.
- The Ax51 assembler will replace the macro name with the text specified in the macro definition.

```
Usage
Symbol MACRO no_arg     ; begin macro definition

; User code goes here

ENDM                    ; end of macro definition

Usage
Symbol MACRO args_up_to_16    ; begin macro definition

; User code goes here

ENDM                          ; end of macro definition
Example: (MACRO.ASM)

REGBANK0 MACRO     ; SIMPLE macro definition
CLR RS0
CLR RS1
ENDM               ; end MACRO

REGBANK1 MACRO     ; SIMPLE macro definition
SETB RS0
CLR RS1
ENDM               ; end MACRO

SUM MACRO X, Y     ; ARGUMENTED macro definition
MOV A, #X
MOV B, #Y
ADD A, B
ENDM               ; end MACRO
```

```
CSEG AT 0
REGBANK0            ; USING MACRO
MOV R0, #1
REGBANK1            ; USING MACRO
MOV R0, #2
SUM 1, 2            ; USING MACR0
END
```

Tools & Development

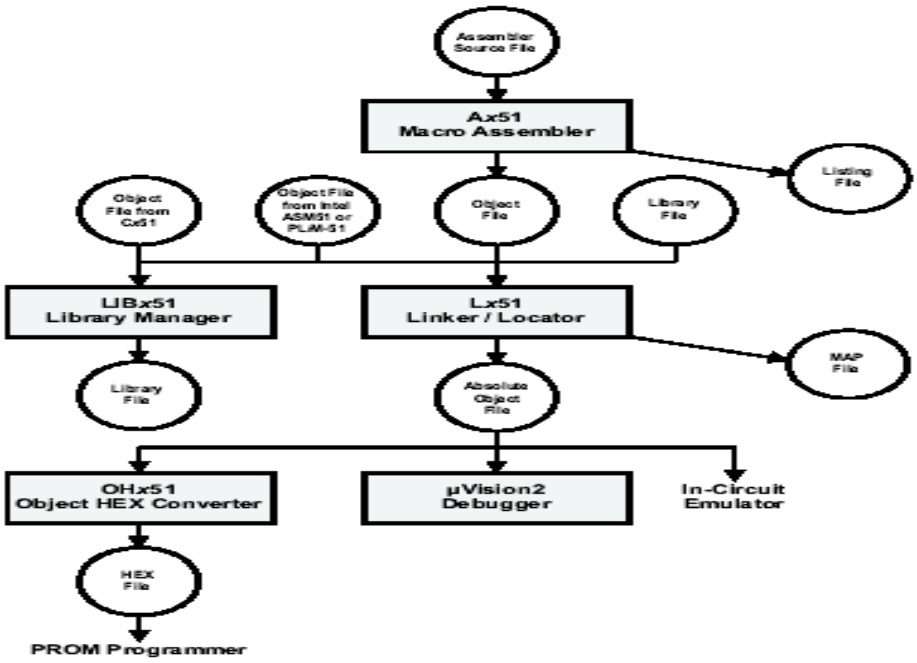
IDE (Integrated Development Environment)

A Development Tool chain i.e. a package of tools

Tools for 8051 development are in IDE

- Editor,
- Cross Assembler,
- Cross Compiler,
- linker,
- simulated Debugger,
- simulated peripherals at least

Flow Chart for KEIL IDE for 8051



File Extensions

Extension	Content and Description
.A51 .ASM .SRC	Source code file: contains ASCII text that is the input for the **Ax51** assembler.
.C .C51	C source code file: contains ASCII text that is the input for the **Cx51** ANSI C compiler.
.INC .H	Include file: contains ASCII text that is merged into an source code file with the include directive. Also these files are input files for **Ax51** or **Cx51**.
.OBJ	Relocatable object file: is the output of the **Ax51** or **Cx51** that contains the program code and control information. Several relocatable object files are typically input files for the **Lx51** Linker/Locater.
.LST	Listing object file: is generated by **Ax51** or **Cx51** to document the translation process. A listing file typically contains the ASCII program text and diagnostic information about the source module. Appendix F describes the format of the **Ax51** listing file.
. *(none)* .ABS	Absolute object file: is the output of the **Lx51**. Typically it is a complete program that can be executed on the *x*51 CPU.
.M51 .MAP	Linker map file: is the listing file generated from **Lx51**. A map file contains information about the memory usage and other statistic information.
.HEX .H86	Hex file: is the output file of the **OHx51** object hex converter in Intel HEX file format. HEX files are used as input file for PROM programmers or other utility programs.

Machine cycle

To execute an instruction–the processor must these four steps refer to Machine Cycle.

1. Fetch the instruction from memory
2. Decode the instruction
3. Execute the instruction
4. Store the result back in the memory.

- Generally one machine cycle = X clock cycles (_X_ depends on the particular instruction being executed). For short clock cycle less time it takes to complete one machine cycle, so instructions are executed faster.
- Fetching and executing an instruction

Fetching involves the following steps

- Contents of PC are placed on address bus.
- READ signal is activated.
- Data (instruction opcode) are read from RAM and placed on data bus.
- Opcode is latched into the CPU's internal instruction register.
- PC is incremented to prepare for the next fetch from memory.

While execution involves decoding the opcode and generating control signals to gate internal registers in and out of the ALU and to signal the ALU to perform the specified operation.

Clock cycle - the basic unit of time for processor activity.

Instruction/machine cycle - is the time period (measured by the number of oscillator clock cycles) during which a CPU processes a machine language instruction from its memory.

8051 Machine Cycle

- 8051 being mostly CISC processor, has 255 instructions of varying lengths and machine cycles.
- Instruction sizes may be 1,2,3 bytes
- Instructions may require 1, 2 or 4 machine cycles time for execution.
- Two oscillator pulses define one state
- 6 states (i.e., 6x2=12 clock cycles) define a machine cycle in this original core and compatible variants.
- 1/f gives the pulse time P
- Hence Time taken for executing an 8051 instruction is

T (inst) = (no of machines cycles x 12) / freq

1) By using above formula calculate time for 1machine cycle by using 12 MHz frequency

T= (1x12)/12MHz =>1microsecond

So, if you are taking 12 MHz frequency time period is 1 microsecond for 1 machine cycle.

2) By using above formula calculate time for 1machine cycle by using 11.0592 MHz frequency

T= (1x12)/11.0592MHz =>1.085microsecond

So, if you are taking 12 MHz frequency time period is 1.085 microsecond for 1 machine cycle.

8051 instructions having varying machine cycles. So, apply these values based on instructions and calculate Time Period i.e., delay. This delay we are using in peripheral programs like LED, 7-Segment etc… for knowing the status in between two operations Examples are given below for 1 ms time delay.

Assembly Language Programming

Delay

/*Write an ALP to generate 1ms time delay by using 12 MHz crystal frequency*/

```
CSEG AT 0       ; starts from 0th memory location (Assembler Directive)

MOV R7, #250    ; for MOV instruction it takes 1 machine cycle i.e., ;

                ; 1m=1microsecond

l1:
DJNZ R7, l1      ; for branch instruction it takes 2 machine cycles i.e.,
                 ; 2m.c=2microseconds it checks R7 value it rotates there
                 ; up to R7 gets zero value it rotates total 250 times
                 ; i.e., 250*2=500machine cycles=500 Microseconds
MOV R6, #249     ; for MOV instruction it takes 1 machine cycle i.e.,
                 ; 1m.c=1microsecond
l2:
DJNZ R6, l2       ; for branch instruction it takes 2 machine cycles
                  ; i.e., 2m.c=2microseconds it checks R7 value it rotates
                  ; There up to R7 gets zero value it rotates total 249
                  ; Times i.e., 249*2=498machine cycles=500 Microseconds
END               ; at the end calculate total machine cycles
```

Output: *1+500+1+498=1000*

*/*Write an ALP to generate 1ms time delay by using 11.0592 MHz crystal frequency*/*

```
CSEG AT 0                       ; cycles

MOV R7, #225                    ; 1.085
l1: DJNZ R7, l1                 ; 2*1.085=2.17 =>2.17*225=488.25
MOV R6, #225                    ; 1.085
l2: DJNZ R6, l2                 ; 2*1.085=2.17 =>2.17*225=488.25
MOV R5, #09                     ; 1.085
l3: DJNZ R5, l3                 ; 2*1.085=2.17 =>2.17*9=19.53
END

Output:
1.085+ (225*2.17) +1.085+ (225*2.17) +1.085+ (9*2.17) =999.285 approx=1millisec
```

*/*Write an ALP to generate 10ms time delay by using 12 MHz crystal frequency*/*

```
CSEG AT 0
ACALL DELAY10MS                 ; including ACALL
SJMP $                          ; It rotates there it self

DELAY1MS:                       ; cycles
MOV R7, #250                    ; 1
L1: DJNZ R7, L1                 ; 2*250=500
MOV R6, #247                    ; 1
L2: DJNZ R6, L2                 ; 2*247=494
RET                             ; 2

DELAY10MS:                      ; Cycles
MOV R5, #09                     ; 1
L3: ACALL DELAY1MS              ; 9*1ms=9ms
DJNZ R5, L3                     ; 9*2us=18us
MOV R7, #250                    ; 1
L4: DJNZ R7, L4                 ; 2*250=500
MOV R5, #237                    ; 1
L5: DJNZ R5, L5                 ; 2*237=474
```

```
NOP                             ; 1
RET                             ; 2
END
/*OUTPUT:
TOTAL                           ; 10MILLI SECONDS*/
```

/*Write an ALP to generate 100ms time delay by using 1ms & 10ms delays frequency is 12 MHz*/

```
CSEG AT 0
ACALL DELAY100MS                ; Including ACALL
SJMP $                          ; It rotates there it self

DELAY1MS:                       ; cycles
MOV R7, #250                        ; 1
L1: DJNZ R7, L1                 ; 2*250=500
MOV R6, #247                        ; 1
L2: DJNZ R6, L2                 ; 2*247=494
RET                     ; 2

DELAY10MS:                      ; Cycles
MOV R5, #09                     ; 1
L3: ACALL DELAY1MS              ; 9*1ms=9ms
DJNZ R5, L3                     ; 9*2us=18us
MOV R7, #250                        ; 1
L4: DJNZ R7, L4                 ; 2*250=500
MOV R5, #237                        ; 1
L5: DJNZ R5, L5                 ; 2*237=474
NOP                             ; 1
RET                     ; 2

DELAY100MS:                     ; Cycles
MOV R0, #09                     ; 1
L6: ACALL DELAY10MS             ; 9*10ms=90ms
DJNZ R0, L6                     ; 9*2us=18us
MOV R5, #09                     ; 1
L7: ACALL DELAY1MS              ; 9*1ms=9ms
```

```
DJNZ R5, L7                    ; 9*2=18us
MOV R7, #250                   ; 1
L8: DJNZ R7, L8                ; 2*250=500
MOV R5, #228                   ; 1
L9: DJNZ R5, L9                ; 2*228=456
RET
END
/*OUTPUT:
TOTAL                          ; 100MILLI SECONDS*/
```

Note: Similar instruction comments are given in first program

I/O Programming for I/O Port Pins

The four 8-bit I/O ports P0, P1, P2 and P3 each port having 8 pins

- All the ports upon RESET are configured as input, ready to be used as input ports
- To reconfigure it as an input, "1" value must be sent to the port
- To use any of these ports as an input port, it must be programmed
- When the first 0 is written to a port, it becomes an output

Port 0

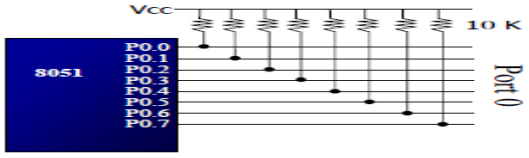
It can be used for input or output; each pin must be connected externally to a 10K ohm pull-up resistor

- This is due to the fact that P0 is an open drain, unlike P1, P2, and P3
- Open drain is a term used for MOS chips in the same way that open collector is used for TTL chips

Dual Role of Port 0

Port 0 is also designated as AD0-AD7, allowing it to be used for both address and data

- When connecting an 8051/31 to an external memory, port 0 provides both address and data



File Name: DELAY.ASM (Sub Header File)

```
DELAY SEGMENT CODE
RSEG DELAY

DELAY1MS:                       ; cycles
MOV R7, #250                         ; 1
L1: DJNZ R7, L1                 ; 2*250=500
MOV R6, #247                         ; 1
L2: DJNZ R6, L2                 ; 2*247=494
RET                       ; 2

DELAY10MS:                      ; Cycles
MOV R5, #09                     ; 1
L3: ACALL DELAY1MS              ; 9*1ms=9ms
DJNZ R5, L3                     ; 9*2us=18us
MOV R7, #250                         ; 1
L4: DJNZ R7, L4                 ; 2*250=500
MOV R5, #237                    ; 1
L5: DJNZ R5, L5                 ; 2*237=474
NOP                             ; 1
RET                             ; 2

DELAY100MS:                     ; Cycles
MOV R0, #09                     ; 1
L6: ACALL DELAY10MS             ; 9*10ms=90ms
DJNZ R0, L6                     ; 9*2us=18us
MOV R5, #09                     ; 1
L7: ACALL DELAY1MS              ; 9*1ms=9ms
DJNZ R5, L7                     ; 9*2=18us
MOV R7, #250                    ; 1
L8: DJNZ R7, L8                 ; 2*250=500
MOV R5, #228                    ; 1
L9: DJNZ R5, L9                 ; 2*228=456
RET
END
```

/* The following code will continuously send out to port 0 the alternating value 55H and AAH */

```
$include(delay.asm)
CSEG AT 0                ; it starts from 0th memory location
BACK: MOV A, #55H
MOV P0, A
ACALL DELAY100MS
MOV A, #0AAH
MOV P0, A
ACALL DELAY100MS
SJMP BACK
END                      ; it ends the program
```

- ***In order to make port 0 an input, the port must be programmed by writing 1 to all the bits***

/* Port 0 is configured first as an input port by writing 1s to it, and then data is received from that port and sent to P1 */

```
CSEG AT 0                ; it starts from 0th memory location
MOV A, #0FFH             ; A=FF hex
MOV P0, A                ; make P0 an i/p port by writing it all 1s
BACK: MOV A, P0          ; get data from P0
MOV P1, A                ; send it to port 1
SJMP BACK                ; keep doing it
END                      ; it ends the program
```

Port 1

Port 1 can be used as input or output

- In contrast to port 0, this port does not need any pull-up resistors since it already has pull-up resistors internally
- Upon reset, port 1 is configured as an input port

/* The following code will continuously send out to port 0 the alternating value 55H and AAH */

```
CSEG AT 0                ; it starts from 0th memory location
MOV A, #55H
BACK: MOV P1, A
ACALL DELAY100MS
CPL A
SJMP BACK
END                      ; it ends the program
```

To make port 1 an input port, it must be programmed as such by writing 1 to all its bits

/* Port 1 is configured first as an input port by writing 1s to it, then data is received from that port and saved in R7 and R5 */

```
$include(delay.asm)
CSEG AT 0                ; it starts from 0th memory location
MOV A, #0FFH             ; A=FF hex
MOV P1, A                ; make P1 an input port by writing it all 1s
MOV A, P1                ; get data from P1
MOV R7, A                ; save it to in register R7
ACALL DELAY100MS         ; wait
MOV A, P1                ; another data from P1
MOV R5, A                ; save it to in register R5
END                      ; it ends the program
```

Port 2

Port 2 can be used as input or output

- Just like P1, port 2 does not need any pull up resistors since it already has pull-up resistors internally
- Upon reset, port 2 is configured as an input port

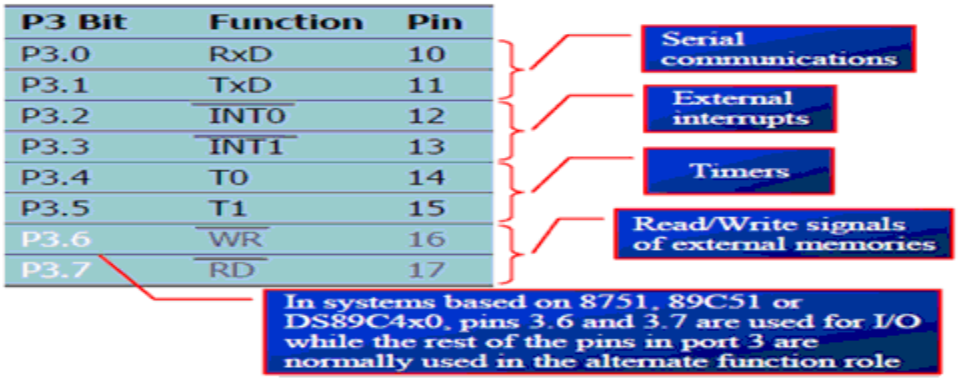
Dual Role of Port 2 and as input

- To make port 2 an input port, it must be programmed as such by writing 1 to all its bits
- In many 8051-based system, P2 is used as simple I/O
- In 8031-based systems, port 2 must be used along with P0 to provide the 16- bit address for the external memory
    - Port 2 is also designated as A8 – A15, indicating its dual function
    - Port 0 provides the lower 8 bits via A0 – A7

Port 3

Port 3 can be used as input or output

- Port 3 does not need any pull-up resistors
- Port 3 is configured as an input port upon reset, this is not the way it is most commonly used
- Port 3 has the additional function of providing some extremely important signals



I/O Ports and Bit Addressability

P0	P1	P2	P3	Port Bit
P0.0	P1.0	P2.0	P3.0	D0
P0.1	P1.1	P2.1	P3.1	D1
P0.2	P1.2	P2.2	P3.2	D2
P0.3	P1.3	P2.3	P3.3	D3
P0.4	P1.4	P2.4	P3.4	D4
P0.5	P1.5	P2.5	P3.5	D5
P0.6	P1.6	P2.6	P3.6	D6
P0.7	P1.7	P2.7	P3.7	D7

/* Sometimes we need to access only 1 or 2 bits of the port */

```
$include(delay.asm)
CSEG AT 0                ; it starts from 0th memory location
BACK: CPL P1.2           ; complement P1.2
ACALL DELAY10MS          ; wait up to 10ms
SJMP BACK                ; go to back label
/*another variation of the above program*/
AGAIN: SETB P1.2         ; set only P1.2
ACALL DELAY10MS          ; wait up to 10ms
CLR P1.2                 ; clear only P1.2
ACALL DELAY10MS          ; wait up to 10ms
SJMP AGAIN               ; go to AGAIN label
END                      ; it ends the program
```

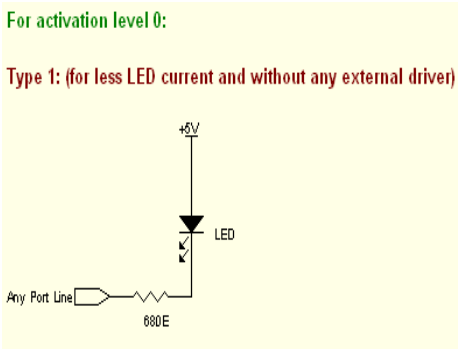
Peripheral Programs in Assembly language by using Delay

LED:

If any LED wants to glow by using microcontroller we have to know about Active Low LED and Active High LED. Those are discussed in below

First you can connect LED with the Port line. The activation level is selectable as either 1 or 0. If you want to use activation level 1, then we have to pass high value from microcontroller to the led then the connected LED glows at the port line for the active high level condition. Otherwise, LED glows at the low level signal when you are connected LED as an Active Low condition.

The actual real time circuits that represent the above choice of point LED interfacing is given here for your reference.



Note: For driving LEDs, 10K resistor is not required. But when you drive relays, the pull up 10K resistor is required because the output current sourcing capacity of any port line varies between 60 and 100 micro A. This small current is insufficient to drive the base of high current transistor.

Note: If you want to know about LED refer Diodes topic in previous chapter.

/*Flash an LED connected to any of the port pin using logical instruction set at the rate of 10ms*/

FILE NAME: LED.ASM

Solution 1: *Here Delay definitions (Subroutines) in Main File only*

```
LED EQU P1.0        ; Port 1.0 pin is assigned to variable name of LED
CSEG AT 0           ; starts from 0th memory location
AGAIN:
CLR LED             ; LED gets zero value P1.0 ← 0
ACALL DELAY100MS    ; wait up to 100milliseconds
SETB LED            ; LED gets one value P1.0 ← 1
ACALL DELAY100MS    ; wait up to 100milliseconds
SJMP AGAIN          ; it jumps to label name of AGAIN

/* below steps it generates 1ms time delay*/
DELAY1MS:                       ; cycles
MOV R7, #250                        ; 1
L1: DJNZ R7, L1                 ; 2*250=500
MOV R6, #247                        ; 1
L2: DJNZ R6, L2                 ; 2*247=494
RET                     ; 2

DELAY10MS:                      ; Cycles
MOV R5, #09                     ; 1
L3: ACALL DELAY1MS              ; 9*1ms=9ms
DJNZ R5, L3                     ; 9*2us=18us
MOV R7, #250                    ; 1
```

```
L4: DJNZ R7, L4                 ; 2*250=500
MOV R5, #237                    ; 1
L5: DJNZ R5, L5                 ; 2*237=474
NOP                             ; 1
RET                             ; 2

DELAY100MS:                     ; Cycles
MOV R0, #09                     ; 1
L6: ACALL DELAY10MS             ; 9*10ms=90ms
DJNZ R0, L6                     ; 9*2us=18us
MOV R5, #09                     ; 1
L7: ACALL DELAY1MS              ; 9*1ms=9ms
DJNZ R5, L7                     ; 9*2=18us
MOV R7, #250                    ; 1
L8: DJNZ R7, L8                 ; 2*250=500
MOV R5, #228                    ; 1
L9: DJNZ R5, L9                 ; 2*228=456
RET
END
```

Solution 2: Here Delay definitions (Subroutines) in DELAY.ASM file, that file you can include in this main file with the help of $include

```
/*Note: If we want to maintain any file as a sub header file don't include CSEG
and END assembler directives*/

$include (DELAY.ASM)
LED EQU P1.0     ; Port 1.0 pin is assigned to variable name of LED
CSEG AT 0        ; starts from 0th memory location (Assembler Directive)
AGAIN:
CLR LED            ; LED gets zero value P1.0 ← 0
ACALL DELAY100MS   ; wait up to 100milliseconds
SETB LED           ; LED gets one value P1.0 ← 1
ACALL DELAY100MS   ; wait up to 100milliseconds
SJMP AGAIN         ; it jumps to label name of AGAIN

END                ; It ends the program (Assembler Directive)
```

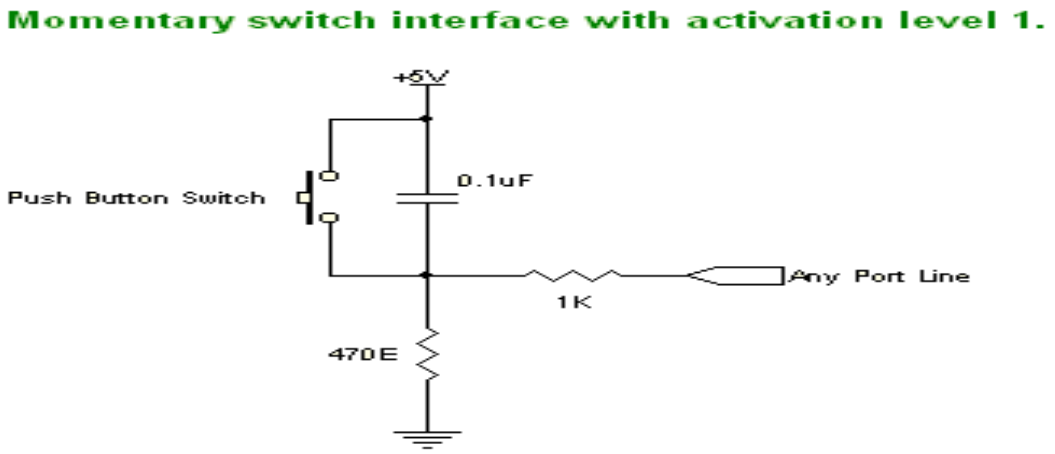
Switch

Momentary Switch Interface

When you select the Momentary switch interface, a momentary switch can be connected to the I/O lines of the controller. The activating level for the individual switches can be set either as 1 or 0. Activating level means the level when the switch is in the pressed condition.

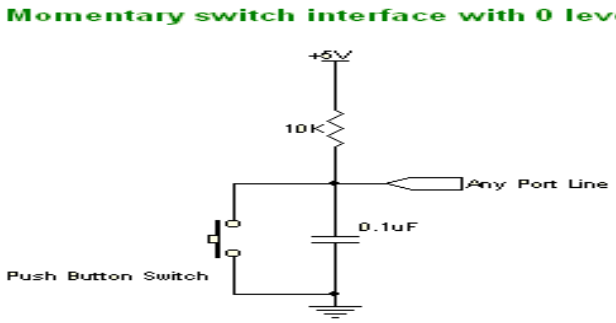
Example

Some real time circuit examples are given here



If level 1 is selected as activation level for a particular switch, then during normal (unpressed) condition the port line (connected to that switch) assumes a 0 level.

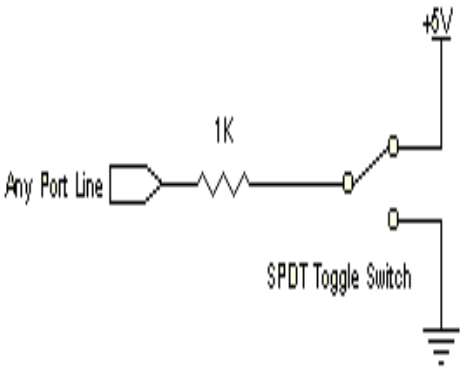
When the same switch is in the pressed condition, level 1 will be fed into the port line.



If level 0 is selected as activation level for a particular switch, then during normal (unpressed) condition the port line (connected to that switch) assumes a 1 level.

When the same switch is in the pressed condition, level 0 will be fed into the port line.

Toggle switch interface



In the Toggle switch interface, a toggle switch can be connected with any on of the port lines of the controller.

/* A momentary switch (SW) is connected to P2.0 and 8 LEDs are connected to P1 now flash the LEDs when we pressed the switch using complement logic*/

```
CSEG AT 0          ; Start from 0th memory location

LED EQU 90H        ; 90H is the address of P1 is assigned to LED
                   ; Variable name
SW EQU P2.0        ; Port pin P2.0 is assigned to SW variable name
MOV A, LED         ; A ← LED ← 0XFF (Default value of P1 is 0xFF)
BACK:
JNB SW, DONE       ; if switch is zero it jumps to the label name of
                   ; DONE else Next Instruction
SJMP BACK          ; It jumps to the label name of BACK
```

```
DONE: CPL A          ; it complements the Accumulator value
MOV LED, A           ; LED←P1←A
SJMP BACK            ; It jumps to the label name of BACK

END                  ; It ends the program
```

7 Seven Segment Display

Introduction

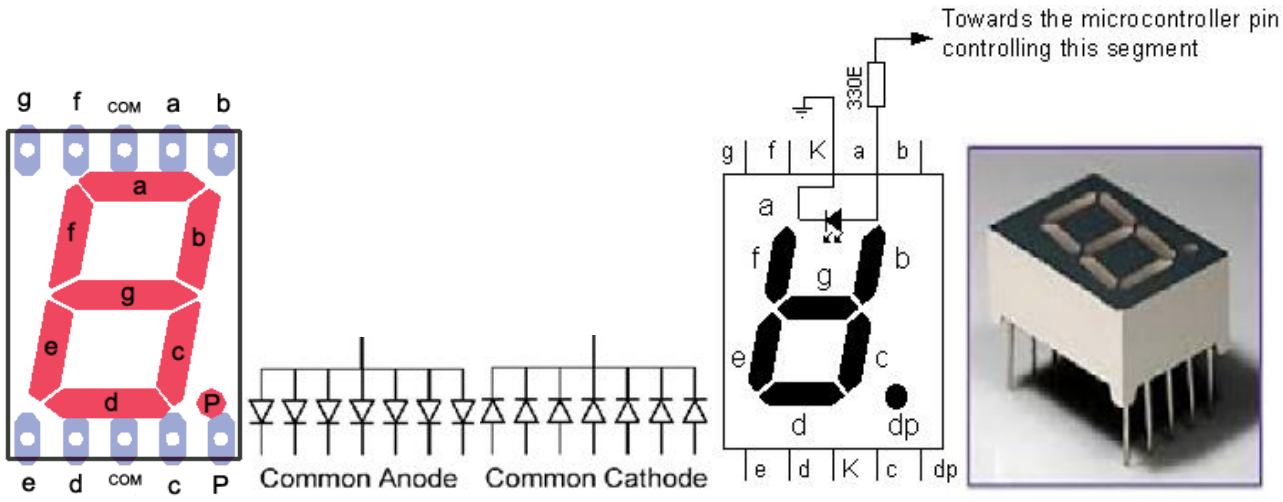
For the seven segment display you can use the LT-541 or LSD5061-11 chip. Each of the segments of the display is connected to a pin on the 8051 (the schematic shows how to do this). In order to light up a segment on the pin must be set to 0V. To turn a segment off the corresponding pin must be set to 5V. This is simply done by setting the pins on the 8051 to '1' or '0'.

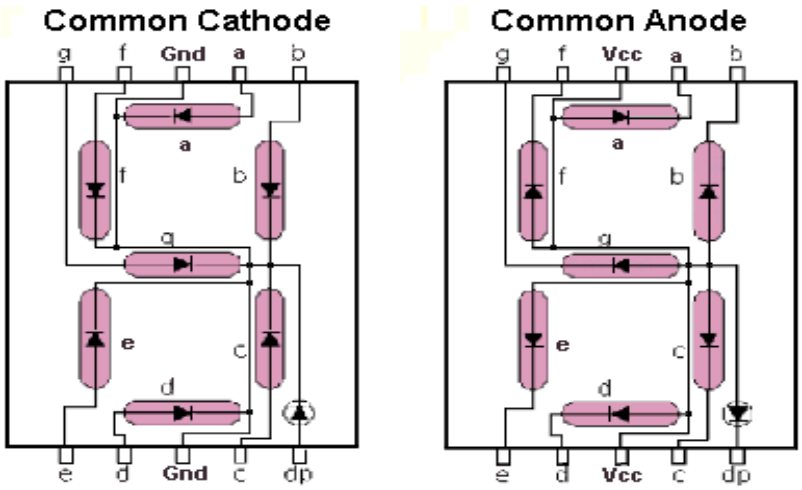
LED displays are

- Power-hungry (10ma per LED)
- Pin-hungry (8 pins per 7-seg display)

But they are cheaper than LCD display

7-SEG Display is available in two types – one is Common anode and second one is Common cathode, but common anode display are most suitable for interfacing with 8051 since 8051 port pins can sink current better than sourcing it.



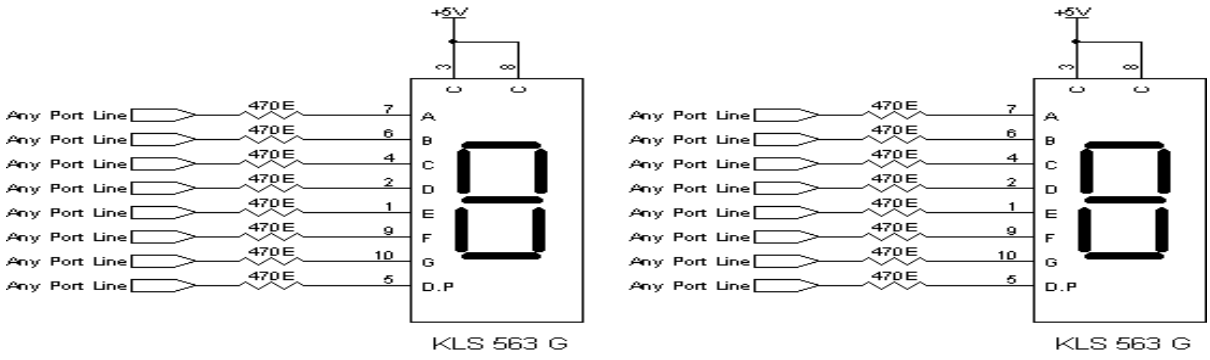


Possible configurations are:

- Non Multiplexed displays with 7 segment inputs.
- Multiplexed 7 segment input displays with internal multiplexer.
- Multiplexed 7 segment input displays with external multiplexer.

In the seven segment displays, the port lines are directly connected to the segments of the display and segments glow according to the display data available at the port lines and also the display type, common anode or common cathode. In the external multiplexer configuration, a multiplexing device, like 74LS139 (2 to 4), 74LS138 (3 to 8) or 74LS154 (4 to 16) is connected between the port lines and the digit select control lines of the display. Some of the real time circuit examples to connect the seven segment displays are given here for your reference.

Non-multiplexed displays with 7-segment input. (For a two digit display)

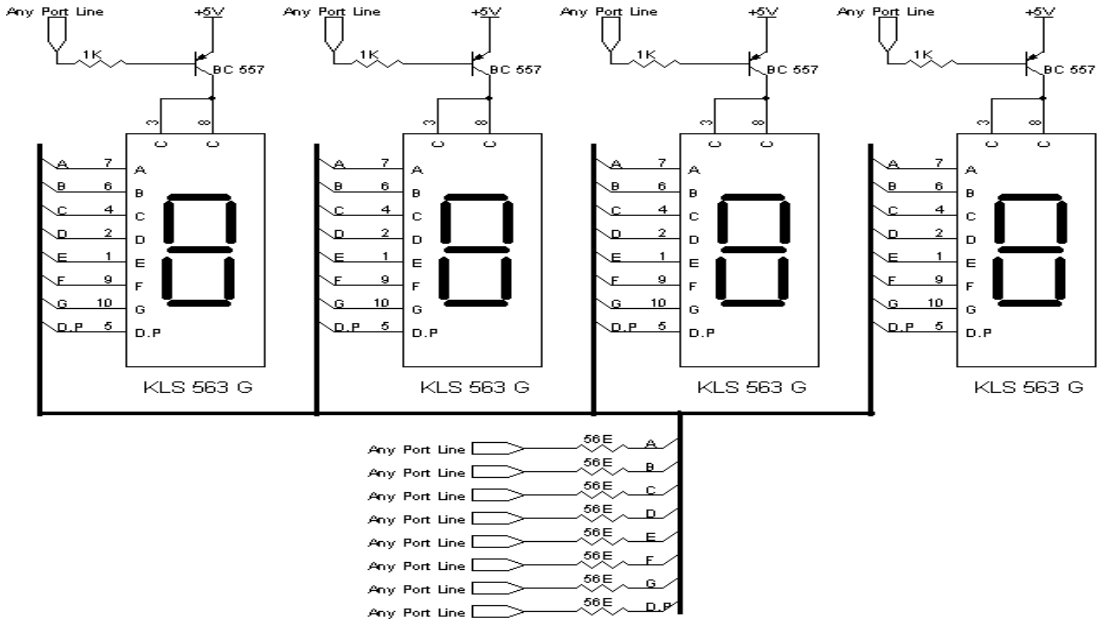


The circuit shown here is meant for the common anode display with the seven segment inputs. In this configuration, the port lines are connected to the segments of the display directly or through buffers. When using this configuration, you can connect a maximum of 4 digits of displays with the controllers having 32 I/O lines.

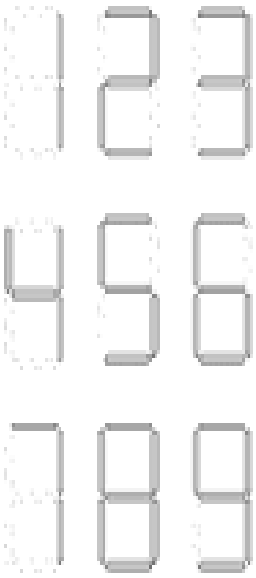
For the common cathode type displays, you can't connect the port lines directly to the segment lines as you have seen above because the source current of any port line is about 60microA.

You can use a buffer between the port lines and the segment lines of the display as shown here.

Multiplexed 7 segment input displays with internal multiplexer



- Seven Segment Inputs.
- Internal Multiplexer with activation level 0.
- Common anode displays.
- 4 digits of display.



Digit Shown	Illuminated Segment (1 = illumination)						
	a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	0	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

/* Write an ALP to display 0-9 numbers on 7-Segment Display (Common Cathode) by using delay header file*/

FILE NAME: SEG. ASM

```
$include (DELAY.ASM)

/* in 100H code memory location place common Anode values*/

CSEG AT 0100H
LUT: DB 0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90

CSEG AT 0          ; starts from 0th memory location
MOV DPTR, #LUT     ; It moves 100h (code memory location) to DPTR
AGAIN:
CLR A              ; A ← 0
MOVC A,@A+DPTR     ; it moves the data in M [DPTR+0] to "A" register
MOV P1, A          ; P1←A
MOV A, DPL         ; A←DPL
INC DPTR           ; DPTR=DPTR+1
ACALL DELAY100MS   ; wait up to 100 milliseconds
```

```
CJNE A, #09, AGAIN; if "A" register value is not equal to "09" it jumps
                  ; to label Else it executes next instruction
SJMP $            ; It rotates here it self

END               ; It ends the program (Assembler Directive)

Note: "DELAY.ASM" sub header file is mentioned in above refer that file
```

Embedded C

Introduction

It is not part of the C language. But, it is a C language extension that is the subject of a technical report by the ISO working group named **"Extensions for the Programming Language C to Support Embedded Processors"**. It aims to provide portability and access to common performance-increasing features of processors used in the domain of DSP and embedded processing. The Embedded C specification for fixed-point, named address spaces, and registers gives the programmer direct access to features in the target processor, thereby significantly improving the performance of applications. The hardware I/O extension is a portability feature of Embedded C. Its goal is to allow easy porting of device-driver code between systems.

KEIL software is an IDE EMBEDDED C

(Using C51 cross compiler)

Language Extensions

C51 provides a number of extensions of ANSI standard C. Most of these provide direct support for elements of the 8051 architecture.

C51 includes extensions for

- Memory types and memory areas on the 8051
- Memory models
- Memory type specifies
- Bit variables and bit addressable data

- Special function registers
- Pointers
- Function Attributes

Key Words

To facilitate many of the features of the 8051, C51 adds a number of new keywords to the scope of the language. The following is the list of keywords

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

Memory Areas

The 8051 architecture supports several physically separate memory areas or memory spaces for program and data. Each memory area offers certain advantages and disadvantages. Those are discussed in below.

Program Memory

Program memory may be accessed using the **code** memory type specifier in the **C51** compiler.

Internal Data Memory

Internal data can be broken into three distinct memory types those are **data**, **idata**, and **bdata**.

- The **data** memory specifier always refers to the first 128 bytes of internal data memory. These variables are accessed using direct addressing.
- The **idata** memory specifier refers to all 256 bytes for (89s52) and 128 bytes for (8051) of internal data memory; however, this memory type specifier code is generated by indirect addressing which is slower than direct addressing.

- The **bdata** memory specifier refers to the 16 bytes of bit-addressable memory in the internal data memory i.e., RAM area (20h to 2Fh). This memory type specifier allows you to declare data types that can also be accessed at the bit level.

External Data Memory

The **C51** Compiler offers two different memory types that access external data: **xdata** and **pdata**.

- The **xdata** memory specifier refers to any location in the 64 Kbytes address space of external data memory.
- The **pdata** memory type specifier refers to only one (1) page or 256 bytes of external data memory.

Memory Models

The memory models determine which default memory type to use for function arguments, automatic variables and declarations with no explicit memory type specifier.

Small Model

- In this model, by default all variables reside in the internal data memory.
- Variable access is very efficient.
- However the stack should fit into the internal RAM

Compact Model

- In this case, all variables reside in one page of external data memory.
- This memory model can accommodate a maximum of 256 bytes of variables.
- In this model, C51 accesses the external memory with instructions that utilize the @R0 and @R1 operands.

Large Model

- In this model, all variables reside in the external data memory up to 64k bytes
- The data pointer is used for addressing.

Explicitly declared memory types

Memory Type Description

code Program memory (64 Kbytes); accessed by opcode **MOVC A, @A+DPTR.**

data Direct addressable internal data memory; fastest access to variables(128 bytes).

idata Indirectly addressable internal data memory; accessed across the full internal address space (256 bytes) for 89s52 and (128 bytes) for 8051.

bdata Bit-addressable internal data memory; supports mixed bit and byte access (16 bytes).

xdata External data memory (64 Kbytes); accessed by opcode **MOVX A, @DPTR**

pdata Paged (256 bytes) external data memory; accessed by opcode **MOVX A, @Rn.**

Examples

```
char data var1;
char code text[] = "ENTER PARAMETER:";
unsigned long xdata array [100];
float idata x, y, z;
unsigned int pdata dimension;
unsigned char xdata vector [10][4][4];
char bdata flags;
```

Data Types

Data type	bits	bytes	range
bit	1		0 or 1
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
enum	8 / 16	1 or 2	-128 to +127 or –32768 to +32767
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to +2147483647
unsigned long	32	4	0 to 4294967295

float	32	4	±1.175494E-38 to ±3.402823E+38
sbit	1		0 or 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

Example 1

```
int bdata ibase;              /* Bit-addressable int */
char bdata bary [4];          /* Bit-addressable array */
sbit mybit0 = ibase ^ 0;      /* bit 0 of ibase */
sbit mybit15 = ibase ^ 15;    /* bit 15 of ibase */
sbit Ary07 = bary[0] ^ 7;     /* bit 7 of bary[0] */
sbit Ary37 = bary[3] ^ 7;     /* bit 7 of bary[3] */

main()
{
Ary37 = 1;                    /* clear bit 7 in bary[3] */
bary[3] = 'a';                /* Byte addressing */
ibase = -1;                   /* Word addressing */
mybit15 = 1;                  /* set bit 15 in ibase */
}
```

NOTE: You may not specify **bit** variables for the bit positions of a **float**.

Special Function Registers

- The **C51** compiler provides you an include file "**reg51.h**" for 8051, which contains declarations for the SFRs available on it.
- The **C51** compiler provides access to SFRs with the **sfr**, **sfr16**, and **sbit** data types.

Examples

```
sfr P0 = 0x80;      /* Port-0, address 80h */
sfr P1 = 0x90;      /* Port-1, address 90h */
sfr P2 = 0xA0;      /* Port-2, address 0A0h */
sfr P3 = 0xB0;      /* Port-3, address 0B0h */
```

Access to 16-bit SFRs is possible only when the low byte immediately precedes the high byte. The low byte is used as the address in the **sfr16** declaration.

```
sfr16 DPTR = 0x82;          /* DPTR : DPL 82h, DPH 83h */
```

For 8052

```
sfr16 T2 = 0xCC;            /* Timer 2: T2L 0CCh, T2H 0CDh */
sfr16 RCAP2 = 0xCA;         /*  RCAP2L 0CAh, RCAP2H 0CBh */
```

With typical 8051 applications, it is often necessary to access individual bits within an SFR. The **C51** compiler makes this possible with the **sbit** data type which provides access to bit-addressable SFRs and other bit-addressable objects.

How can we access in our program we mentioned in below.

```
sfr PSW = 0xD0;
sfr IE = 0xA8;
sbit OV = PSW ^ 2;
sbit CY = PSW ^ 7;
sbit EA = IE ^ 7;

sbit OV = 0xD0 ^ 2;
sbit CY = 0xD0 ^ 7;
sbit EA = 0xA8 ^ 7;

sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

Absolute Variable Location

- Variables may be located at absolute memory locations in your C program source modules using the **_at_** keyword.

- The usage for this feature is:

  type [memory_space] variable_name **_at_** constant t;

Where

- **memory_space** is the memory space for the variable. If it is missing from the declaration, the default memory space is used.
- **type** is a variable type.
- **Variable_name** is the variable name.(user defined)
- **Constant** is the address at which to locate the variable.
- The following restrictions apply to absolute variable location:

1. Absolute variables cannot be initialized.
2. Functions and variables of type **bit** cannot be located at an absolute address.

The following example demonstrates how to locate several different variable types using the **_at_** keyword.

Example

```
char xdata text[256] _at_ 0xE000;   /* array at xdata 0xE000 */
int xdata i1 _at_ 0x8000;           /* int at xdata 0x8000 */
main()
{
i1 = 0x1234;
text[0] = 'a';
}
```

Pointers

The **C51** compiler provides two different types of pointers:

- **generic pointers**
- **Memory-specific pointers**

Generic Pointers

- Generic pointers are declared in the same fashion as standard C pointers.

Examples

```
char *s;          /* string ptr */
int *numptr;      /* int ptr */
long *state;      /* Texas */
```

- Generic pointers are always stored using three bytes**.** The first byte is the memory type, the second is the high-order byte of the offset, and the third is the low-order byte of the offset.
- Generic pointers may be used to access any variable regardless of its location in 8051 memory space.

Memory-specific Pointers

- Memory-specific pointers always include a memory type specification in the pointer declaration and always refer to a specific memory area.

Examples

```
char data *str;          /* ptr to string in data */
int xdata *numtab;       /* ptr to int(s) in xdata */
long code *powtab;       /* ptr to long(s) in code */
```

- Memory-specific pointers can be stored using only one byte (**idata**, **data**, **bdata**, and **pdata** pointers) or two bytes (**code** and **xdata** pointers).
- The code generated for a memory-specific pointer executes more quickly than the equivalent code generated for a generic pointer. This is because the memory area is known at compile-time rather than at run-time. The compiler can use this information to optimize memory accesses. If execution speed is a priority, you should use memory-specific pointers instead of generic pointers wherever possible.
- Like generic pointers, you may specify the memory area in which a memory-specific pointer is stored.

Examples

char data * xdata str; ***/* ptr in xdata to data char */***

int xdata * data numtab; ***/* ptr in data to xdata int */***

long code * idata powtab; ***/* ptr in idata to code long */***

- Memory-specific pointers may be used to access variables in the declared 8051 memory area only.
- Memory-specific pointers provide the most efficient method of accessing data objects, but at the cost of reduced flexibility.

Embedded C Programming

Time delay

There are two way s to create a time delay in 8051 C

- Using the 8051 timer (we will discussed in timers topic)
- Using a simple for loop

be mindful of three factors that can affect the accuracy of the delay

- The 8051 design
  - The number of machine cycle
  - The number of clock periods per machine cycle
- The crystal frequency connected to the X1 – X2 input pins Compiler choice
  -- C compiler converts the C statements and functions to Assembly Language instructions
  -- Different compilers produce different code

/*Write an ECP (Embedded C Programming) to generate 1ms time delay by using 1ms delay frequency is 12 MHz*/

```
void delay(unsigned int i);
main()
{
delay(1);// function name is "delay" call this function with 1 ms
while(1);// By using this loop it rotates there it self it never
            // fails
}
```

```
/*DELAY function definition*/
void delay(unsigned int i)
{
unsigned char j;
for(;i>0;i--) // in above delay calling function having one
            // value "i" variable assigned to that value
{
for(j=255;j>0;j--); //it rotates 255 times there it self
for(j=232;j>0;j--); //it rotates 232 times there it self

}
}
```

/*Write an ECP (Embedded C Programming) to generate 10ms time delay by using 1ms delay frequency is 12 MHz*/

```
void delay(unsigned int);     // Function declaration of "delay"
main()
{
      delay(10);  // function name is "delay" call this function 10
                  // times for 10 ms
      while(1);   // By using this loop it rotates there itself it
                  // never fails
}
DELAY function definition
void delay(unsigned int i)
{
unsigned char j;
for(;i>0;i--) // in above delay calling function having one
            // value "i" variable assigned to that value
{
for(j=255;j>0;j--); //it rotates 255 times there it self
for(j=232;j>0;j--); //it rotates 232 times there it self

}
}
```

/*Write an ECP (Embedded C Programming) to generate 100ms time delay by using 1ms & 10ms delays frequency is 12 MHz*/

```
void delay(unsigned int);      // Function declaration of "delay"

main()
{
      delay(100); // function name is "delay" call this function 100
                  // times for 100 ms
      while(1);   // By using this loop it rotates there it self it
                  // never fails
}

/*DELAY function definition*/
void delay(unsigned int i)
{
unsigned char j;
for(;i>0;i--) // in above delay calling function having one
            // value "i" variable assigned to that value
{
for(j=255;j>0;j--); //it rotates 255 times there it self
for(j=232;j>0;j--); //it rotates 232 times there it self

}
}
```

***Note**: Like above programs you can generate any delay with the help of same function definition how much you want to generate that value you can place in delay function calling. But, by using for loops we are not getting exact delay it's an approximate delay. If you want exact delay then you can use timers.*

LED

/*Write an ECP to Flash an LED connected to any of the port pin at the rate of 500ms (check at Active Low LED circuit)

FILE NAME: LED.C

```
Solution 1: Here Delay definitions (Subroutines) in Main File only

#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)
#include"delay.h" // Include delay header file as predefined

sbit led=P1^0;    // single bit declaration data type to declare single
                  // port pin to user defined variable named as "led"
void delay(unsigned int);     // Function declaration of "delay"

main()
{
while(1)      // infinite loop never fails why because 1 is constant
              // non-zero value it rotates infinitely
{
led=0;       // led← P1^0 ← 0, i.e., led gets ON position
delay(500); // it is delay function calling i.e., it waits up to 500ms
led=1;       // led← P1^0 ← 1, i.e., led gets OFF position
delay(500); // it is delay function calling i.e., it waits up to 500ms
}
}
/*DELAY function definition*/
void delay(unsigned int i)
{
unsigned char j;
for(;i>0;i--) // in above delay calling function having one
              // value "i" variable assigned to that value
{
for(j=255;j>0;j--); //it rotates 255 times there it self
```

```
for(j=232;j>0;j--); //it rotates 232 times there it self
}
}
```

Solution 2:

Here Delay definitions (Subroutines) in DELAY.H file, that file you can include in this main file with the help of "#include"

File Name: delay.h (Sub Header File)

```
#ifndef _delay_h_       // if not define delay header file
#define _delay_h_       // define header file
/*DELAY function definition*/
void delay(unsigned int i)
{
unsigned char j;
for(;i>0;i--) // in above delay calling function having one
            // value "i" variable assigned to that value
{
for(j=255;j>0;j--); //it rotates 255 times there it self
for(j=232;j>0;j--); //it rotates 232 times there it self
}
}

#endif                  // it ends if statement
```

MAIN FILE NAME: LED.C

```
#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)
#include "delay.h" // It includes delay sub header file
sbit led=P1^0;    // single bit declaration data type to declare single
                  // port pin to user defined variable named as "led"
void delay(unsigned char);    // Function declaration of "delay"
```

```
main()
{
while(1)      // infinite loop never fails why because 1 is constant
              // non-zero value it rotates infinitely
{
led=0;       // led← P1^0 ← 0, i.e., led gets ON position
delay(500); // it is delay function calling i.e., it waits up to 500ms
led=1;       // led← P1^0 ← 1, i.e., led gets OFF position
delay(500); // it is delay function calling i.e., it waits up to 500ms
}
}
```

***Note**: Open one file save that file with one name in that just we have to write function definition then that one will be considering as a sub header file. Include that header file in your main file with as it is what we saved with that file name. In sub header file we have to write function definitions only. Active low, active high led circuits are explained in above.*

SWITCH

/* Write an ECP a momentary switch (SW) is connected to P2.0 and 8 LEDs are connected to P1 now "ON" the LEDs when we pressed the switch and vice versa*/

```
#include <reg51.h> // It includes all SFRs addresses in this header
                   // file (Predefined)
#define led P1     //define port to 8leds
sbit sw=P2^0;      // single bit declaration data type to declare single
                   // port pin to user
main()
{
      if(sw==0)    // it checks the condition switch is pressed or not
            led=0x00;    // if switch is pressed led gets "ON" position
      else               //if switch is not pressed
            led=0xff;    //led gets "OFF" position
}
/*NOTE: for above example switch and leds are connected in active low */
```

7-SEGMENT

/* Write an ECP to display 0-9 numbers on Common Cathode 7-Segment Display by using delay header file*/

FILE NAME: SEG. C

```
#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)
#include"delay.h" // include delay sub header file
#define seg P1    //define port 1 for segment

/*Look UP table for 7 segment common Anode values*/
code char a[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90};

main()
{
      unsigned char i;        //declare "i" variable
      while(1)                // infinite loop
      {
      for(i=0;i<=9;i++)       //for getting 0-9 values use for loop up
                              // to 9 times
      {
            seg =a[i];        //assign array values to segment
            delay (1000);     //wait second
      }
      }
}
```

Note: "delay.h" is defined in LED's topic use that file.

/* Write an ECP to display 0-99 numbers on multiplexed Common Cathode 7-Segment Display by using delay header file*/

FILE NAME: SEGMUX. C

```
#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)
#include"delay.h" // include delay sub header file


#define seg P2    //define port 2 for multiplexed (2) 7-segment
sbit sel1=P3^0;   //for 1st segment selection assign port pin
sbit sel2=P3^1;   //for 2nd segment selection assign port pin

/*Look UP table for 7 segment common Anode values*/
code char a[]={0xc0,0xf9,0xa4,0xb0,0x99,0x92,0x82,0xf8,0x80,0x90};
main()
{
      unsigned char i, j; //declare "i","j" variables
      sel1=sel2=0;        //first "OFF" both segments
      for(i=0;i<=9;i++)   //for getting 0-9 values by using look up
      {                   // table on 1st segment
      for(j=0;j<=9;j++)   //for getting 0-9 values by using look up
      {                   //table on 2nd segment

            sel1=1;       // 1st segment gets "ON" position
            seg=a[i];     //assign array values to 1st segment
            delay(5);     //wait 5 milli seconds
            sel1=0;       //1st segment gets "OFF" position
            sel2=1;       // 2nd segment gets "ON" position
            seg=a[j];     //assign array values to 2nd segment
            delay(5);     //wait 5 milli seconds
            sel2=0;       // 2nd segment gets "OFF" position
      }
      }
}
Note: "delay.h" is defined in LED's topic use that file.
```

Key Board

Before learning "Interfacing Matrix Keypad to Microcontroller" you should know how to interface a switch to microcontroller.

Revise the article for **interfacing switches to 8051.**

Introduction

Keypads are a part of HMI or Human Machine Interface and play really important role in a small embedded system where human interaction or human input is needed. Matrix keypads are well known for their simple architecture and ease of interfacing with any microcontroller.

►Constructing a Matrix Keypad

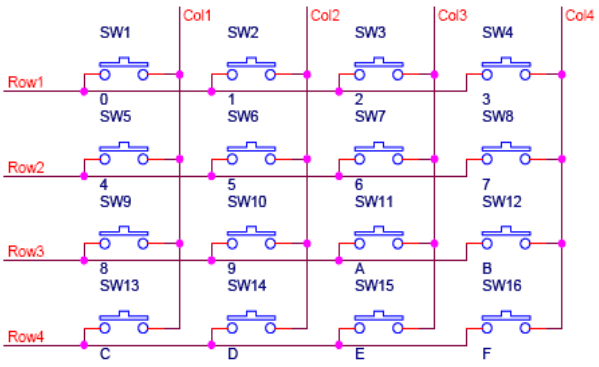
Keyboards are organized in a matrix of rows and columns

- The CPU accesses both rows and columns through ports
- Therefore, with two 8-bit ports, an 8 x 8 matrix of keys can be connected to a microprocessor
- When a key is pressed, a row and a column make a contact
- Otherwise, there is no connection between rows and columns
- In IBM PC keyboards, a single microcontroller takes care of hardware and software interfacing

  A 4x4 matrix connected to two ports
- The rows are connected to an output port and the columns are connected to an input port

Construction of a keypad is really simple. As per the outline shown in the figure below we have four rows and four columns. In between each overlapping row and column line there is a key

So keeping this outline we can construct a keypad using simple SPST Switches as shown below:



Now our keypad is ready, all we have to do is connect the rows and columns to a port of microcontroller and program the controller to read the input.

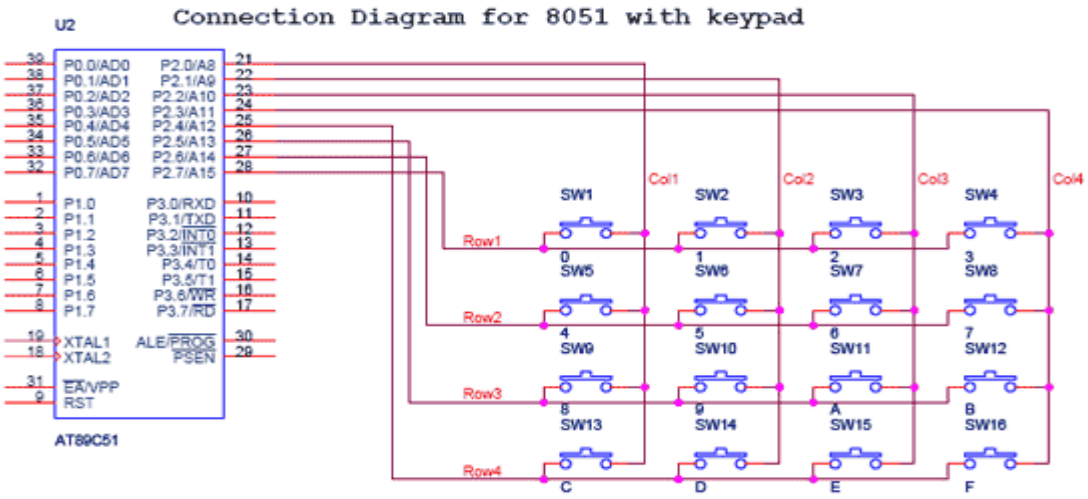
►Scanning a Matrix Keypad

There are many methods depending on how you connect your keypad with your controller, but the basic logic is same. We make the columns as i/p and we drive the rows making them o/p, this whole procedure of reading the keyboard is called scanning.

In order to detect which key is pressed from the matrix, we make row lines low one by one and read the columns. Let's say we first make Row1 low, and then read the columns. If any of the key in row1 is pressed, will make the corresponding column as low i.e. if second key is pressed in Row1, then column2 will give low. So we come to know that key 2 of Row1 is pressed. This is how scanning is done.

So to scan the keypad completely, we need to make rows low one by one and read the columns. If any of the buttons is pressed in a row, it will take the corresponding column to a low state which tells us that a key is pressed in that row. If button 1 of a row is pressed then Column 1 will become low, if button 2 then column2 and so on...

Keypad Connections with 8051 Microcontroller



As you can see no pin is connected to ground, over here the controller pin itself provides the ground.

/ Write an ECP to display keypad values on led*/*

```
#include<reg51.h>
#include"delay.h"

#define led P1                              // define 8 leds to Port1

/* Declare rows and columns to Port2*/
sbit C0=P2^0;
sbit C1=P2^1;
sbit C2=P2^2;
sbit C3=P2^3;
sbit R0=P2^4;
sbit R1=P2^5;
sbit R2=P2^6;
sbit R3=P2^7;

/* Define 2 dimensional array for displaying corresponding data based on key
pressing*/
/* Total 4 rows and 4 columns i.e., 16 keys each key having one data*/
char keypad[4][4]={{'1','2','3','4'},
                   {'5','6','7','8'},
                   {'9','A','B','C'},
                   {'D','E','F's,'0'}};

/*Main Function*/
main()
{
      int row=0,col=0;// initialize variables for identify row and
                      // column
      R0=R1=R2=R3=0;  // 1st make all rows as zeros
      C0=C1=C2=C3=1;  // make all columns are ones

      while(C0&C1&C2&C3);     // check switch pressing or not
```

```
while(1)                    // infinite loop
{
/* Row Checking*/
      R0=0;R1=R2=R3=1;// check in row0 any one of key pressed or
                      //  not
      if(!(C0&C1&C2&C3))// if condition is true zero value is
                        // assigned to row
      {
            row=0;
            break;
      }
      R1=0;R0=R2=R3=1;// check in row1 any one of key pressed or
                       // not
      if(!(C0&C1&C2&C3))// if condition is true one value is
                        // assigned to row
      {
            row=1;
            break;
      }
      R2=0;R1=R0=R3=1;// check in row2 any one of key pressed or
                      // not

      if(!(C0&C1&C2&C3))// if condition is true two value is
                        // assigned to row
      {
            row=2;
            break;
      }
      R3=0;R1=R2=R0=1;// check in row3 any one of key pressed or
                      // not
      if(!(C0&C1&C2&C3))// if condition is true three value is
                        // assigned to row
      {
            row=3;
            break;
      }
}
```

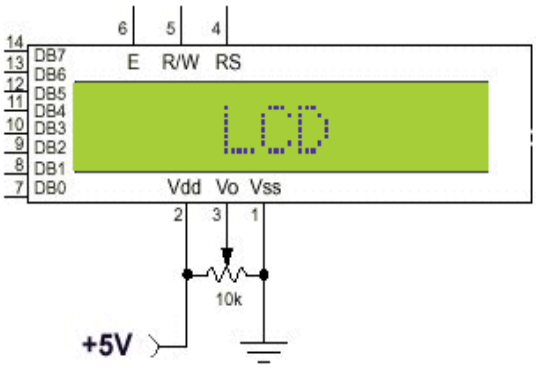
```
/* Column Check*/
      if(C0==0)
            col=0;
      else if(C1==0)
            col=1;
      else if(C2==0)
            col=2;
      else if(C3==0)
            col=3;

      while (!(C0&C1&C2&C3));       // for key releasing

      led = keypad[row][col];       // assign key value to led to see
                                    // the output
      delay (500);                  // wait 500 milli seconds
}
Note: "delay.h" is defined in LED's topic use that file.
```

LCD (Liquid Crystal Display)

- LCD is finding widespread use replacing LEDs
- The declining prices of LCD
- The ability to display numbers, characters, and graphics
- Incorporation of a refreshing controller into the LCD, thereby relieving the CPU of the task of refreshing the LCD
- Ease of programming for characters and graphics

LCD also called as Liquid Crystal Display is very helpful in providing user interface as well as for debugging purpose. The most common type of LCD controller is HITACHI 44780 which provides a simple interface between the controller & an LCD. These LCD's are very simple to interface with the controller as well as cost effective.

The LCD's provide an easy way to get text display for an embedded system.The most commonly used **ALPHANUMERIC** displays are 1x16 (Single Line & 16 characters), 2x16 (Double Line & 16 character per line) & 4x20 (four lines & Twenty characters per line).



UNDERSTANDING LCD

Pin Descriptions for LCD

Pin	Symbol	I/O	Descriptions
1	VSS	--	Ground
2	VCC	--	+5V power supply
3	VEE	--	Power supply to control contrast
4	RS	I	RS=0 to select command register, RS=1 to select data register
5	R/W	I	R/W=0 for write, R/W=1 for read
6	E	I/O	Enable
7	DB0	I/O	The 8-bit data bus
8	DB1	I/O	The 8-bit data bus
9	DB2	I/O	The 8-bit data bus
10	DB3	I/O	The 8-bit data bus
11	DB4	I/O	The 8-bit data bus
12	DB5	I/O	The 8-bit data bus
13	DB6	I/O	The 8-bit data bus
14	DB7	I/O	The 8-bit data bus

used by the LCD to latch information presented to its data bus

Pin-out

8 data pins D7:D0

- Bi-directional data/command pins. Alphanumeric characters are sent in ASCII format.

RS: Register Select

- RS = 0 -> Command Register is selected
- RS = 1 -> Data Register is selected

R/W: Read or Write

- 0 -> Write, 1 -> Read

E: Enable (Latch data)

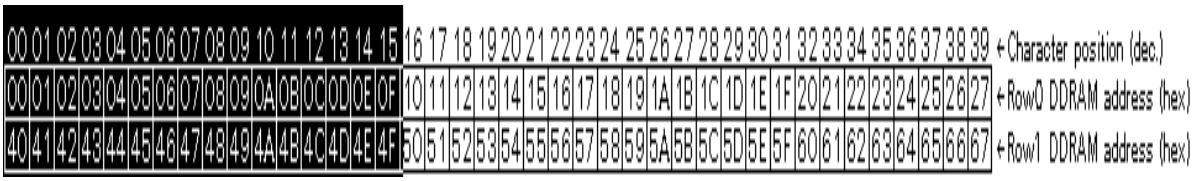
- This pin is used to latch the data present on the data pins. A high-to-low edge is needed to latch the data.

VEE: contrast control

- When writing to the display, data is transferred only on the high to low transition of this signal. However, when reading from the display, data will become available shortly after the low to high transition and remain available until the signal falls low again.

Display Data RAM (DDRAM)

Display data RAM (DDRAM) is where you send the characters (ASCII code) you want to see on the LCD screen. It stores display data represented in 8-bit character codes. Its capacity is 80 characters (bytes). Below you see DD RAM address layout of a 2*16 LCD.



In the above memory map, the area shaded in black is the visible display (For 16x2 displays). For first line addresses for first 15 characters is from 00h to 0Fh. But for second line address of first character is 40h and so on up to 4Fh for the 16th character. So if you want to display the text at specific positions of LCD, we require to manipulate address and then to set cursor position accordingly.

Character Generator RAM (CGRAM)-User defined character RAM

In the character generator RAM, we can define our own character patterns by program. CG RAM is 64 bytes, allowing for eight 5*8 pixel, character patterns to be defined.

Registers

The HD44780 has two 8-bit registers, an instruction register (IR) and a data register (DR). The IR stores instruction codes. The DR temporarily stores data to be written into DDRAM or CGRAM and temporarily stores data to be read from DDRAM or CGRAM. Data written into the DR is automatically written into DDRAM or CGRAM by an internal operation. . These two registers can be selected by the register selector (RS) signal. See the table below:

		Register Selection
RS	**R/W**	**Operation**
0	0	IR write as an internal operation (display clear, etc.)
0	1	Read busy flag (DB7) and address counter (DB0 to DB6)
1	0	DR write as an internal operation (DR to DDRAM or CGRAM)
1	1	DR read as an internal operation (DDRAM or CGRAM to DR)

Busy Flag (BF)

When the busy flag is 1, the LCD is in the internal operation mode, and the next instruction will not be accepted. When RS = 0 and R/W = 1 (see the table above), the busy flag is output to DB7 (MSB of LCD data bus). The next instruction must be written after ensuring that the busy flag is 0.

LCD Commands

The LCD's internal controllers accept several commands and modify the display accordingly. These commands would be things like:

- Clear screen
- Return home
- Shift display right/left

Set Cursor Move Direction

- ID - Increment the Cursor after Each Byte Written to Display if Set
- S - Shift Display when Byte Written to Display

Enable Display/Cursor

- D - Turn Display on (1)/Off (0)
- C - Turn Cursor on (1)/Off (0)
- B - Cursor Blink On (1)/Off(0)

Move Cursor/Shift Display

- SC - Display Shift on (1)/Off (0)
- RL - Direction of Shift Right (1)/Left (0)

The HD44780 instruction set is shown below:

R/S	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Instruction/Description
4	5	14	13	12	11	10	9	8	7	Pins
0	0	0	0	0	0	0	0	0	1	Clear Display
0	0	0	0	0	0	0	0	1	*	Return Cursor and LCD to Home Position
0	0	0	0	0	0	0	1	ID	S	Set Cursor Move Direction
0	0	0	0	0	0	1	D	C	B	Enable Display/Cursor
0	0	0	0	0	1	SC	RL	*	*	Move Cursor/Shift Display
0	0	0	0	1	DL	N	F	*	*	Set Interface Length
0	0	0	1	A	A	A	A	A	A	Move Cursor into CGRAM
0	0	1	A	A	A	A	A	A	A	Move Cursor to Display
0	1	BF	*	*	*	*	*	*	*	Poll the "Busy Flag"
1	0	D	D	D	D	D	D	D	D	Write a Character to the Display at the Current Cursor Position
1	1	D	D	D	D	D	D	D	D	Read the Character on the Display at the Current Cursor Position

The bit descriptions for the different commands are:

"*" - Not Used/Ignored. This bit can be either "1" or "0"

Set Interface Length

- DL - Set Data Interface Length 8(1)/4(0)
- N - Number of Display Lines 1(0)/2(1)
- F - Character Font 5x10(1)/5x7(0)

Poll the "Busy Flag"

- BF - This bit is set while the LCD is processing

Move Cursor to CGRAM/Display

- A – Address

Read/Write ASCII to the Display

- D – Data

Set cursor position (DDRAM address)

As said earlier if we want to display the text at specific positions of LCD, we require to manipulate address and then to set cursor position accordingly.

I want to display "VECTOR" in message "Hi VECTOR" at the right corner of first line then I should start from 10th character. So, referring to table 80h+0Ah= 8Ah.

Interfacing LCD to 8051

The 44780 standard requires 3 control lines as well as either 4 or 8 I/O lines for the data bus. The user may select whether the LCD is to operate with a 4-bit data bus or an 8-bit data bus.

If a 4-bit data bus is used, the LCD will require a total of 7 data lines.

If an 8-bit data bus is used, the LCD will require a total of 11 data lines.

The three control lines are EN, **RS**, and **RW**.

Note that the EN line must be raised/lowered before/after each instruction sent to the LCD regardless of whether that instruction is read or write, text or instruction. In short, you must always manipulate EN when communicating with the LCD. EN is the LCD's way of knowing that you are talking to it. If you don't raise/lower EN, the LCD doesn't know you're talking to it on the other lines.

Checking the Busy Flag

You can use subroutine for checking busy flag or just a big (and safe) delay.

1. Set R/W Pin of the LCD HIGH(read from the LCD)
2. Select the instruction register by setting RS pin LOW
3. Enable the LCD by Setting the enable pin HIGH
4. The most significant bit of the LCD data bus is the state of the busy flag (1=Busy,0=ready to accept instructions/data). The other bits hold the current value of the address counter.

If the LCD never come out from "busy" status because of some problems, the program will"hang," waiting for DB7 to go low. So in real applications it would be wise to put some kind of time limit on the delay--for example, a maximum of 100 attempts to wait for the busy signal to go low. This would guarantee that even if the LCD hardware fails, the program would not lock up.

/* Write an ECP to display "V" character on LCD by using 8-bit mode*/

```
#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)
#include"delay.h" // It includes delay sub header file
#define lcd P0    // define one port to lcd data lines
sbit RS=P1^0;     // define port pin for register select
sbit RW=P1^1;     // define port pin for read/write operations
sbit EN=P1^2;     // define port pin for enable

/* Function Declarations*/
void init_lcd(void);                // initialization of lcd
void cmd_lcd(unsigned char);        // command mode of lcd
void data_lcd(unsigned char);       // data mode of lcd
void write_lcd(unsigned char);      // write command or data
void delay(unsigned int );          // for delay

/* Main Function */
main()
{
      init_lcd();                   // it calls initialize function
      cmd_lcd(0x80);                // call command to get cursor from
                                    // starting position
```

```
      data_lcd('V');                // display "v" character on LCD
      while(1);                     // stop here
}

/* Function Definitions of LCD*/
void init_lcd()
{
      cmd_lcd(0x38);          // 8-bit mode 5*7 pixels
      cmd_lcd(0x01);          // clear the screen
      cmd_lcd(0x0C);          // display on with cursor off
      cmd_lcd(0x06);          // shift cursor right side
      cmd_lcd(0x80);          // cursor is in 1st position
}

void cmd_lcd(unsigned char ch)
{
      RS=0;                   // LCD switches in to command mode
      Write_lcd(ch);          // write command by using write function
                              // calling
}

void data_lcd(unsigned char ch)
{
      RS=1;                   // LCD switches in to data mode
      Write_lcd(ch);          // write data by using write function
                              // calling
}

void write_lcd(unsigned char ch)
{
      lcd=ch;                 // lcd <-- ch
      RW=0;                   // LCD switches to write operation
      EN=1;                   // to latch the information
      delay(2);               // wait 2 milliseconds
      EN=0;                   // make enable low
}
Note: "delay.h" header file available in "LEDs program"
```

/ Write an ECP to display "V" character on LCD by using 4-bit mode*/*

```
#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)
#include"delay.h" // It includes delay sub header file

#define lcd P0    // define one port to lcd data lines
sbit RS=P1^0;     // define port pin for register select
sbit RW=P1^1;     // define port pin for read or write
sbit EN=P1^2;     // define port pin for enable

/* Function Declarations*/
void init_lcd(void);                // initialization of lcd
void cmd_lcd(unsigned char);        // command mode of lcd
void data_lcd(unsigned char);       // data mode of lcd
void write_lcd(unsigned char);      // write data  or command
void delay(unsigned int );          // for delay

/* Main Function*/
main()
{
      init_lcd();             // it calls initialize function
      cmd_lcd(0x80);          // call command to get cursor from
                              // starting position
      data_lcd('V');          // display "v" character on LCD
      while(1);               // stop here.
}
/* Function Definitions of LCD*/
void init_lcd()
{
      cmd_lcd(0x02);    // return cursor to home position
      cmd_lcd(0x28);    // 4-bit mode(it accepts only 4 data lines)
      cmd_lcd(0x01);    // clear the screen
      cmd_lcd(0x0C);    // display on with cursor off
      cmd_lcd(0x06);    // shift cursor right side
      cmd_lcd(0x80);    // cursor is in 1st position
}
```

```
void cmd_lcd(unsigned char ch)
{
      RS=0;              //LCD switches in to command mode
      Write_lcd(ch);     // write command by using write function
                         // calling
}
void data_lcd(unsigned char ch)
{
      RS=1;                    // LCD switches in to data mode
      Write_lcd(ch);           // write data by using write function
                               // calling
}
void write_lcd(unsigned char ch)
{
      lcd=(ch & 0xf0);         // higher nibble value passes to lcd
      RW=0;                    // LCD switches to write operation
      EN=1;                    // to latch the information
      delay(2);                // wait 2 milliseconds
      EN=0;                    // make enable low
      lcd=(ch<<4);             //lower nibble value gets higher position
                               // and passes to lcd
      RW=0;                    // LCD switches to write operation
      EN=1;                    // to latch the information
      delay(2);                // wait 2 milliseconds
      EN=0;                    // make enable low
}
```

Note: "delay.h" header file available in "LEDs program"

Timers

The 8051 has two timers/counters, they can be used either as

- Timers to generate a time delay or as
- Event counters to count events happening outside the microcontroller

Both Timer 0 and Timer 1 are 16 bits wide

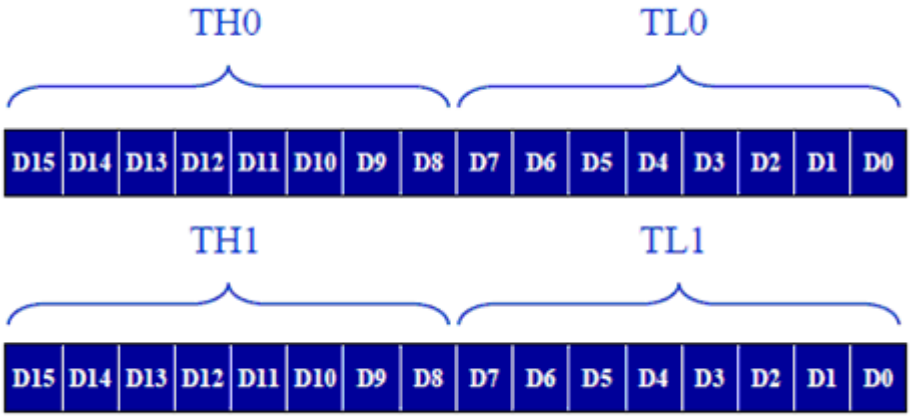
- Since 8051 has an 8-bit architecture, each 16-bits timer is accessed as two separate registers of low byte and high byte

Accessed as low byte and high byte

- The low byte register is called TL0/TL1 and
- The high byte register is called TH0/TH1
- Accessed like any other register

MOV TL0, #4FH

MOV R5, TH0



Both timers 0 and 1 use the same register, called TMOD (timer mode), to set the various timer operation modes

TMOD is an 8-bit register

- The lower 4 bits are for Timer 0
- The upper 4 bits are for Timer 1
- In each case,
  - The lower 2 bits are used to set the timer mode
  - The upper 2 bits to specify the operation

GATE: Gating control when set

- Timer/counter is enabled only while the INTx pin is high and the TRx control pin is set
- When cleared, the timer is enabled whenever the TRx control bit is set

C/T: Timer or counter selected

- Cleared for timer operation (input from internal system clock)
- Set for counter operation (input from Tx input pin)

Mode1 and Mode2

M1	M0	Mode	Operating Mode
0	0	0	**13-bit timer mode** 8-bit timer/counter THx with TLx as 5-bit prescaler
0	1	1	**16-bit timer mode** 16-bit timer/counter THx and TLx are cascaded; there is no prescaler
1	0	2	**8-bit auto reload** 8-bit auto reload timer/counter; THx holds a value which is to be reloaded TLx each time it overfolws
1	1	3	**Split timer mode**

Timer Mode 0 and Mode 1

The timer/counters can be operated in one of four modes, under software control. In mode 0, the timer/counter will behave like a 13 bit counter. When the counter overflows, the TF0 or TF1 (timer flag) bit in the TCON (timer control) SFR is set. This will cause the appropriate timer interrupt (assuming it is enabled). Both timer 0 and timer 1 operate in the same way for mode 0. The operation of the timers in mode 1 is the same as it is for mode 0 with the exception that all sixteen bits of the timer are used instead of only thirteen.

Timer Mode 2

In mode 2, the timer is set up as an eight bit counter which automatically reloads whenever an overflow condition is detected. The low byte of the timer (TL0 or TL1) is used as the counter and the high byte of the timer (TH0 or TH1) holds the reload value for the counter.

When the timer/counter overflows, the value in THx is loaded into TLx and the timer continues counting from the reload value. Both timer 0 and timer 1 function identically in mode 2. Timer 1 is often used in this mode to generate baud rates for the UART.

Timer Mode 3

In mode 3, timer 0 becomes two eight bit counters which are implemented in TH0 and TL0. The counter implemented in TL0 maintains control of all the timer 0 flags, but the counter in TH0 takes over the control flags in TCON from timer 1. This implies that timer 1 can no longer force interrupts, however, it can be used for any purpose which will not require the overflow interrupt such as a baud rate generator for the UART, or as a timer/counter which is polled by the software.

This is useful when an application must use a UART mode which requires baud rate generation from timer 1 and also requires two timer/counters. When timer 1 is placed in mode 3 it simply freezes.

If C/T = 0, it is used as a timer for time delay generation. The clock source for the time delay is the crystal frequency of the 8051

If desired, the timer/counters can force a software interrupt when they overflow. The TCON (Timer Control) SFR is used to start or stop the timers as well as hold the overflow flags of the timers. The TCON SFR is detailed below in Table A7. The timer/counters are started or stopped by changing the timer run bits (TR0 and TR1) in TCON. The software can freeze the operation of either timer as well as restart the timers simply by changing the TRx bit in the TCON register. The TCON register also contains the overflow flags for the timers. When the timers overflow, they set their respective flag (TF0 or TF1) in this register. When the processor detects a 0 to 1 transition in the flag, an interrupt occurs if it is enabled. It should be noted that the software can set or clear this flag at any time. Therefore, an interrupt can be prevented as well as forced by the software.

Timer Control Register (TCON) - Bit Addressable

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	Timer 1 overflow flag. Set when timer 1 overflows. Cleared by processor upon vectoring to the interrupt service routine.
TR1	Timer 1 control bit. If TR1=1, timer 1 runs. If TR1=0, timer 1 stops.
TF0	Timer 0 overflow flag. Set when timer 0 overflows. Cleared by processor upon vectoring to the interrupt service routine.
TR0	Timer 0 control bit. If TR0=1, timer 1 runs. If TR0=0, timer 1 stops.
IE1	External interrupt 1 edge flag. Set when a valid falling edge is detected at pin P3.3. Cleared by hardware when the interrupt is serviced.
IT1	Interrupt 1 type control bit. If IT1=1, interrupt 1 is triggered by a falling edge on P3.3. If IT1=0, interrupt 1 is triggered by a low logic level on P3.3
IE0	External interrupt 0 edge flag. Set when a valid falling edge is detected at pin P3.2. Cleared by hardware when the interrupt is serviced.
IT0	Interrupt 0 type control bit. If IT0=1, interrupt 1 is triggered by a falling edge on P3.2. If IT0=0, interrupt 0 is triggered by a low logic level on P3.2

Table A - 7

Example 1

Indicate which mode and which timer is selected for each of the following.

(a) MOV TMOD, #01H (b) MOV TMOD, #20H (c) MOV TMOD, #12H

Solution

We convert the value from hex to binary. From Figure 9-3 we have:

(a) TMOD = 00000001, mode 1 of timer 0 is selected.

(b) TMOD = 00100000, mode 2 of timer 1 is selected.

(c) TMOD = 00010010, mode 2 of timer 0, and mode 1 of timer 1 are selected.

Example 2

Find the timer's clock frequency and its period for various 8051-based systems, with the crystal frequency 11.0592 MHz when C/T bit of TMOD is 0(it acts as a timer according to TMOD register).

Solution

f=1/12 × 11.0529 MHz = 921.6 MHz;

T = 1/921.6 kHz = 1.085 us

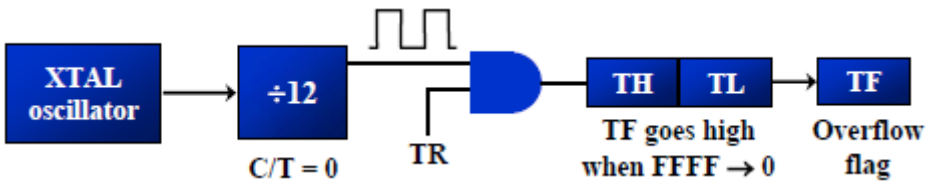
The following are the characteristics and operations of mode1:

1. It is a 16-bit timer; therefore, it allows value of 0000 to FFFFH to be loaded into the Timer's register TL and TH

2. After TH and TL are loaded with a 16-bit initial value, the timer must be started
   - This is done by SETB TR0 for timer 0 and SETB TR1 for timer 1

3. After the timer is started, it starts to count up
   - It counts up until it reaches its limit of FFFFH
   - When it rolls over from FFFFH to 0000, it sets high a flag bit called TF (timer flag)
     – Each timer has its own timer flag: TF0 for timer 0, and TF1 for timer 1
     – This timer flag can be monitored
   - When this timer flag is raised, one option would be to stop the timer with the instructions CLR TR0 or CLR TR1, for timer 0 and timer 1, respectively

4. After the timer reaches its limit and rolls over, in order to repeat the process
   - TH and TL must be reloaded with the original value, and
   - TF must be reloaded to 0



To generate a time delay

1. Load the TMOD value register indicating which timer (timer 0 or timer 1) is to be used and which timer mode (0 or 1) is selected.
2. Load registers TL and TH with initial count value
3. Start the timer
4. Keep monitoring the timer flag (TF) with the JNB TFx, target instruction to see if it is raised
   - Get out of the loop when TF becomes high
5. Stop the timer
6. Clear the TF flag for the next round
7. Go back to Step 2 to load TH and TL again

Example 3

In the following program, we create a square wave of 50% duty cycle (with equal portions high and low) on the P1.5 bit. Timer 0 is used to generate the time delay.(with 12 MHz frequency)

```
CSEG AT 0
MOV TMOD, #01           ; Timer 0, mode 1(16-bit mode)
HERE: MOV TL0, #0CH     ; TL0=0CH, the low byte
MOV TH0, #0FEH          ; TH0=FEH, the high byte
CPL P1.5                ; toggle P1.5
ACALL DELAY             ; It calls delay function (subroutine)
SJMP HERE               ; It jumps to "HERE" label

DELAY:
SETB TR0                ; start the timer 0
AGAIN: JNB TF0, AGAIN   ; monitor timer flag 0 until it rolls over
CLR TR0                 ; stop timer 0
CLR TF0                 ; clear timer 0 flag
RET                     ; It returns from the delay subroutine
END                     ; It ends the program
```

In the above program notice the following steps:

1. TMOD is loaded.
2. FFF2H is loaded into TH0-TL0.
3. P1.5 is toggled for the high and low portions of the pulse.
4. The DELAY subroutine using the timer is called.
5. In the DELAY subroutine, timer 0 is started by the SETB TR0 instruction.
6. Timer 0 counts up with the passing of each clock, which is provided by the crystal oscillator. As the timer counts up, it goes through the states of FE0B, FE0C, FE0D, FE0E, FE0F, FF10, FF11, and so on until it reaches FFFFH. One more clock rolls it to 0, raising the timer flag (TF0=1). At that point, the JNB instruction falls through.

7. Timer 0 is stopped by the instruction CLR TR0. The DELAY subroutine ends and the process is repeated.

Notice that to repeat the process, we must reload the TL and TH registers, and start the process is repeated

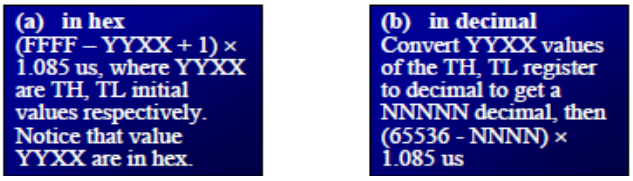
Example 4

In Example 9-4, calculate the amount of time delay in the DELAY subroutine generated by the timer. Assume XTAL = 12 MHz

Solution

The timer works with a clock frequency of 1/12 of the XTAL frequency; therefore, we have 12 MHz / 12 = 1 MHz as the timer frequency. As a result, each clock has a period of T = 1/1 MHz = 1us. In other words, Timer 0 counts up each 1us resulting in delay = number of counts × 1us. The number of counts for the roll over is FFFFH – FE0CH = 1F3H (499 decimal). However, we add one to 499 because of the extra clock needed when it rolls over from FFFF to 0 and raise the TF flag.

This gives 500 × 1us = 500us for half the pulse. For the entire period it is T = 2 × 500us = 1000us i.e., 1ms as the time delay generated by the timer.



Example 5

In Example 4, calculate the frequency of the square wave generated on pin P1.5.

Solution

In the timer delay calculation of Example 4, we did not include the overhead due to instruction in the loop. To get a more accurate timing, we need to add clock cycles due to these instructions in the loop. To do that, we use the machine cycle from Table A-1 in Appendix A, as shown below.

```
CSEG AT 0
                                Cycles
HERE: MOV TL0, #0F2H            2
MOV TH0, #0FFH                  2
CPL P1.5                        1
ACALL DELAY                     2
SJMP HERE                       2

DELAY:
SETB TR0                        1
AGAIN: JNB TF0, AGAIN           14
CLR TR0                         1
CLR TF0                         1
RET                             2
END
```

Total-28

T = 2 × 28 × 1us = 56us and F = 17857.1 Hz

Example 6

In the following program, we create a square wave of 50% duty cycle (with equal portions high and low) on the P1.5 bit. Timer 0 is used to generate the time delay (with 12 MHz frequency) by using timer0 mode 0.

```
CSEG AT 0

MOV TMOD, #00          ; Timer 0, mode 0(13-bit mode)
HERE: MOV TL0, #0BH    ; TL0=0CH, the low byte
MOV TH0, #0F0H         ; TH0=FEH, the high byte
CPL P1.5               ; toggle P1.5
ACALL DELAY            ; It calls delay function (subroutine)
SJMP HERE              ; It jumps to "HERE" label


DELAY:
SETB TR0               ; start the timer 0
AGAIN: JNB TF0, AGAIN ; monitor timer flag 0 until it rolls over
CLR TR0                ; stop timer 0
CLR TF0                ; clear timer 0 flag
RET                    ; it returns from the delay subroutine

END                    ; It ends the program
```

Maximum value is 8191-500=7691 (decimal) = 1111000001011 (Binary) =THX (8 bits) and TLX (Lower 5 bits prescaler) now, hex value is F00BH.

Timer 0 mode 0 maximum value in hexadecimal TL0= 1FH and TH0=FFH

SERIAL COMMUNICATION

Basics

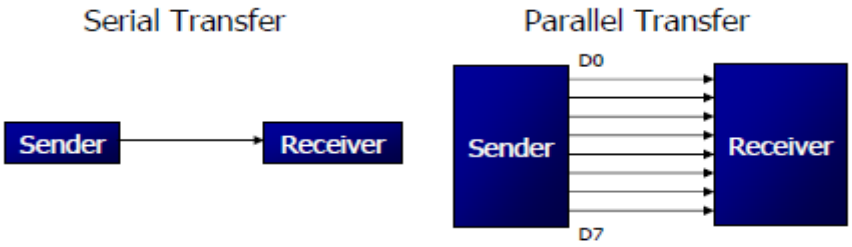
Computers transfer data in two ways:

Parallel

- Often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away

Serial

- To transfer to a device located many meters away, the serial method is used
- The data is sent one bit at a time



- At the transmitting end, the byte of data must be converted to serial bits using parallel-in-serial-out shift register
- At the receiving end, there is a serial-in- parallel-out shift register to receive the serial data and pack them into byte
- When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation
- If data is to be transferred on the telephone line, it must be converted from 0s and 1s to audio tones
- This conversion is performed by a device called a modem, **"Modulator/demodulator"**

Serial data communication uses two methods

- Synchronous method transfers a block of data at a time
- Asynchronous method transfers a single byte at a time

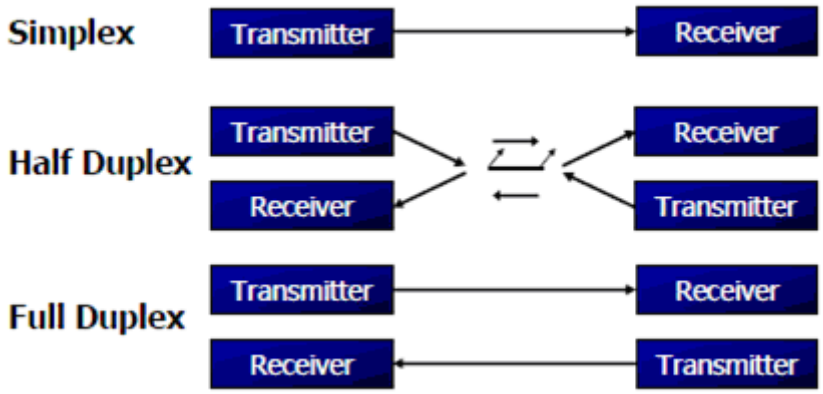
It is possible to write software to use either of these methods, but the programs can be tedious and long

- There are special IC chips made by many manufacturers for serial communications
  - UART (universal asynchronous Receiver transmitter)
  - USART (universal synchronous-asynchronous Receiver-transmitter)

If data can be transmitted and received, it is a duplex transmission

- If data transmitted one way a time, it is referred to as half duplex
- If data can go both ways at a time, it is full duplex

This is contrast to simplex transmission



A protocol is a set of rules agreed by both the sender and receiver on

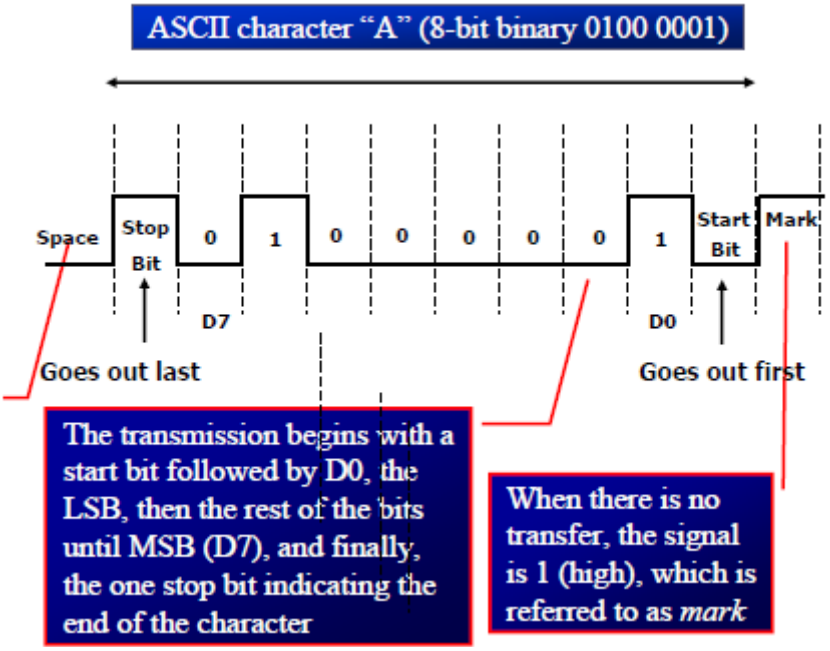
- How the data is packed
- How many bits constitute a character
- When the data begins and ends
- Asynchronous serial data

Communication is widely used for character-oriented transmissions

- Each character is placed in between start and stop bits, this is called framing
- Block-oriented data transfers use the synchronous method

The start bit is always one bit, but the stop bit can be one or two bits

The start bit is always a 0 (low) and the stop bit(s) is 1 (high)



Due to the extended ASCII characters, 8-bit ASCII data is common

- In older systems, ASCII characters were 7- bit

In modern PCs the use of one stop bit is standard

- In older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte

The rate of data transfer in serial data communication is stated in **bps (bits per second)**

Another widely used terminology for bps is **baud rate**

- It is modem terminology and is defined as the number of signal changes per second
- In modems, there are occasions when a single change of signal transfers several bits of data

As far as the conductor wire is concerned, the baud rate and bps are the same, and we use the terms interchangeably

The data transfer rate of given computer system depends on communication ports incorporated into that system

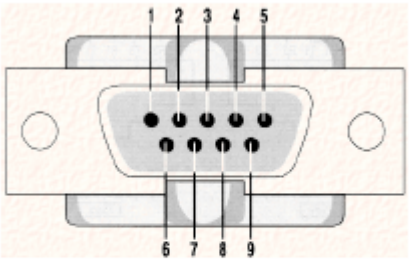
- IBM PC/XT could transfer data at the rate of 100 to 9600 bps
- Pentium-based PCs transfer data at rates as high as 56K bps
- In asynchronous serial data communication, the baud rate is limited to 100K bps

An interfacing standard RS232 was set by the Electronics Industries Association (EIA) in 1960

- The standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible
- In RS232, a 1 is represented by -3 ~ -25 V, while a 0 bit is +3 ~ +25 V, making -3 to +3 undefined

IBM introduced the DB-9 version of the serial I/O standard

RS232 Connector DB-9



RS232 DB-9 Pins

Pin	Description
1	Data carrier detect (-DCD)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready (-DSR)
7	Request to send (-RTS)
8	Clear to send (-CTS)
9	Ring indicator (RI)

Current terminology classifies data communication equipment as

DTE (data terminal equipment)

- refers to terminal and computers that send and receive data

DCE (data communication equipment)

- refers to communication equipment, such as modems
- The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground

DTR (data terminal ready)

- When terminal is turned on, it sends out signal DTR to indicate that it is ready for communication

DSR (data set ready)

- When DCE is turned on and has gone through the self-test, it assert DSR to indicate that it is ready to communicate

RTS (request to send)

- When the DTE device has byte to transmit, it assert RTS to signal the modem that it has a byte of data to transmit

CTS (clear to send)

- When the modem has room for storing the data it is to receive, it sends out signal CTS to DTE to indicate that it can receive the data now

DCD (data carrier detect)

- The modem asserts signal DCD to inform the DTE that a valid carrier has been detected and that contact between it and the other modem is established

RI (ring indicator)

- An output from the modem and an input to a PC indicates that the telephone is ringing
- It goes on and off in synchronous with the ringing sound

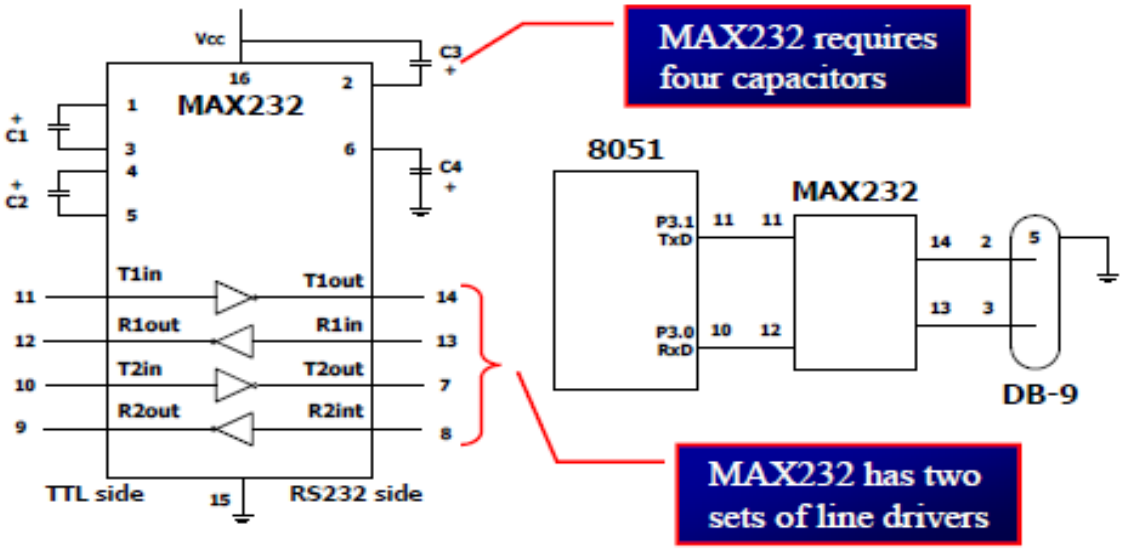
MAX 232

- A line driver such as the MAX232 chip is required to convert RS232 voltage levels to TTL levels, and vice versa
- 8051 has two pins that are used specifically for transferring and receiving data serially

  - These two pins are called TxD and RxD and are part of the port 3 group (P3.0 and P3.1)

  - These pins are TTL compatible; therefore, they require a line driver to make them RS232 compatible

- We need a line driver (voltage converter) to convert the R232's signals to TTL voltage levels that will be acceptable to 8051's TxD and RxD pins



- To allow data transfer between the PC and an 8051 system without any error, we must make sure that the baud rate of 8051 system matches the baud rate of the PC's COM port
- HyperTerminal function supports baud rates much higher than listed below

PC Baud Rates
110
150
300
600
1200
2400
4800
9600
19200

Baud Rates supported by 486/Pentium IBM PC BIOS

With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates.

(a) 9600 (b) 2400 (c) 1200

Solution

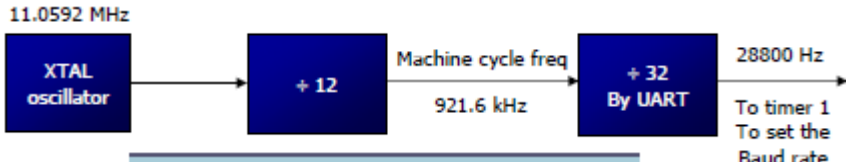
The machine cycle frequency of 8051 = 11.0592 / 12 = 921.6 kHz, and 921.6 kHz / 32 = 28,800 Hz is frequency by UART to timer 1 to set baud rate.

(a) 28,800 / 3 = 9600 where -3 = FD (hex) is loaded into TH1

(b) 28,800 / 12 = 2400 where -12 = F4 (hex) is loaded into TH1

(c) 28,800 / 24 = 1200 where -24 = E8 (hex) is loaded into TH1

Notice that dividing 1/12 of the crystal frequency by 32 is the default value upon activation of the 8051 RESET pin.



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

TF is set to 1 every 12 ticks, so it functions as a frequency divider

SBUF Register

SBUF is an 8-bit register used solely for serial communication

- For a byte data to be transferred via the TxD line, it must be placed in the SBUF register
  - The moment a byte is written into SBUF, it is framed with the start and stop Bits and transferred serially via the TxD line

- SBUF holds the byte of data when it is received by 8051 RxD line
  - When the bits are received serially via RxD, the 8051 deframes it by eliminating the stop and start bits, making a byte out of the data received, and then placing it in SBUF

```
MOV SBUF, #'D'        ; load SBUF=44h, ASCII for 'D'
MOV SBUF, A           ; copy accumulator into SBUF
MOV A, SBUF           ; copy SBUF into accumulator
```

SCON Register

SCON is an 8-bit register used to program the start bit, stop bit, and data bits of data framing, among other things

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0	SCON.7	Serial port mode specifier
SM1	SCON.6	Serial port mode specifier
SM2	SCON.5	Used for multiprocessor communication
REN	SCON.4	Set/cleared by software to enable/disable reception
TB8	SCON.3	Not widely used
RB8	SCON.2	Not widely used
TI	SCON.1	Transmit interrupt flag. Set by HW at the begin of the stop bit mode 1. And cleared by SW
RI	SCON.0	Receive interrupt flag. Set by HW at the begin of the stop bit mode 1. And cleared by SW

Note: Make SM2, TB8, and RB8 =0

SM0, SM1

They determine the framing of data by specifying the number of bits per character, and the start and stop bits

SM0	SM1	
0	0	Serial Mode 0
0	1	Serial Mode 1, 8-bit data, 1 stop bit, 1 start bit
1	0	Serial Mode 2
1	1	Serial Mode 3

Only mode 1 is of interest to us

SM2

This enables the multiprocessing capability of the 8051.

REN (receive enable)

It is a bit-addressable register

- When it is high, it allows 8051 to receive data on RxD pin
- If low, the receiver is disable

TI (transmit interrupt)

When 8051 finishes the transfer of 8-bit character

- It raises TI flag to indicate that it is ready to transfer another byte
- TI bit is raised at the beginning of the stop bit

RI (receive interrupt)

When 8051 receives data serially via RxD, it gets rid of the start and stop bits and places the byte in SBUF register

- It raises the RI flag bit to indicate that a byte has been received and should be picked up before it is lost
- RI is raised halfway through the stop bit

Serial Communication Programming

Programming for Serial Data Transmitting

In programming the 8051 to transfer character bytes serially

1. TMOD register is loaded with the value 20H, indicating the use of timer 1 in mode 2 (8-bit auto-reload) to set baud rate
2. The TH1 is loaded with one of the values to set baud rate for serial data transfer
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-Bit data is framed with start and stop bits
4. TR1 is set to 1 to start timer 1
5. TI is cleared by CLR TI instruction
6. The character byte to be transferred serially is written into SBUF register

7. The TI flag bit is monitored with the use of instruction JNB TI, xx to see if the Character has been transferred completely
8. To transfer the next byte, go to step 5

/* Write a program for the 8051 to transfer letter "A" serially at 9600 baud, continuously. (11.0592 MHz frequency) */

```
CSEG AT 0

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

END                         ; it ends the program
```

Programming Serial Data Receiving

In programming the 8051 to receive character bytes serially

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

/* Write a program for the 8051 to receive bytes of data serially, and put them in P1, set the baud rate at 9600, 8-bit data, and 1 stop bit */

```
CSEG AT 0
MOV TMOD, #20H              ; timer 1, mode 2(auto reload)
MOV TH1, #-3                ; 9600 baud rate
MOV SCON, #50H              ; 8-bit, 1 stop, REN enabled
SETB TR1                    ; start timer 1
HERE: JNB RI, HERE          ; wait for char to come in
MOV A, SBUF                 ; saving incoming byte in A
MOV P1, A                   ; send to port 1
CLR RI                      ; get ready to receive next byte
SJMP HERE                   ; keep getting data
END                         ; it ends the program
```

/* Write a program for the 8051 to receive bytes of data serially, transfer what we receive at 9600 baud, continuously. (11.0592 MHz frequency) */

```
CSEG AT 0
MOV TMOD, #20H              ; timer 1, mode 2(auto reload)
MOV TH1, #-3                ; 9600 baud rate
MOV SCON, #50H              ; 8-bit, 1 stop, REN enabled
SETB TR1                    ; start timer 1
HEREE: JNB RI, HEREE        ; wait for char to come in
MOV A, SBUF                 ; saving incoming byte in A
MOV SBUF, A                 ; Accumulator value to transfer
HERE: JNB TI, HERE          ; wait for the last bit
CLR TI                      ; clear TI for next char
CLR RI                      ; get ready to receive next byte
SJMP HEREE                  ; keep getting data
END                         ; it ends the program
```

/* Write a program for the 8051 to transfer letter "A" continuously as well as parallel to receive bytes of data serially at 9600 baud rate, transfer what we receive on P1 (f=11.0592 MHz frequency) */

```
CSEG AT 0
MOV TMOD, #20H              ; timer 1, mode 2(auto reload)
MOV TH1, #-3                ; 9600 baud rate
MOV SCON, #50H              ; 8-bit, 1 stop, REN enabled
SETB TR1                    ; start timer 1
AGAIN: MOV SBUF, #"A"       ; Accumulator value to transfer
HERE: JNB TI, HERE          ; wait for the last bit

JB RI, HEREE                ; last bit is completed
HEREE: MOV A, SBUF          ; saving incoming byte in A
MOV P1, A                   ; send Accumulator value to P1
CLR TI                      ; clear TI for next char
CLR RI                      ; get ready to receive next byte
SJMP AGAIN                  ; keep sending A
END                         ; it ends the program
```

Doubling the baud rate

There are two ways to increase the baud rate of data transfer

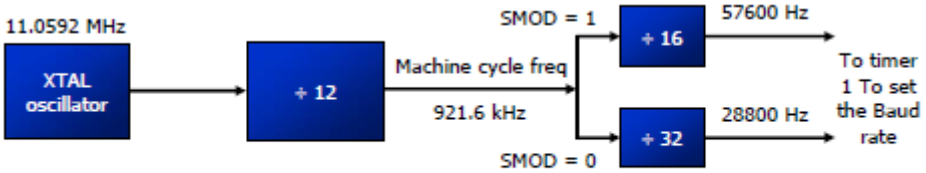
- To use a higher frequency crystal
- To change a bit in the PCON register

PCON register is an 8-bit register

- When 8051 is powered up, SMOD is zero
- We can set it to high by software and thereby double the baud rate

SMOD	--	--	--	GF1	GF0	PD	IDL

```
MOV A, PCON         ; place a copy of PCON in ACC
SETB ACC.7          ; make D7=1
MOV PCON, A         ; changing any other bits
```



Baud Rate comparison for SMOD=0 and SMOD=1

TH1	(Decimal)	(Hex)	SMOD=0	SMOD=1
	-3	FD	9600	19200
	-6	FA	4800	9600
	-12	F4	2400	4800
	-24	E8	1200	2400

Interrupts

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service

A single microcontroller can serve several devices by two ways

- Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal
- Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device
- The program which is associated with the interrupt is called the interrupt service routine (ISR) or interrupt handler

Polling

- The microcontroller continuously monitors the status of a given device
- When the conditions met, it performs the service
- After that, it moves on to monitor the next device until every one is serviced

Polling can monitor the status of several devices and serve each of them as certain conditions are met

- The polling method is not efficient, since it wastes much of the microcontroller's time by polling devices that do not need service

Example

JNB TF, target

The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time)

- Each devices can get the attention of the microcontroller based on the assigned priority
- For the polling method, it is not possible to assign priority since it checks all devices in a round-robin fashion

The microcontroller can also ignore (mask) a device request for service

- This is not possible for the polling method

Interrupt Service Routine

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler

- When an interrupt is invoked, the microcontroller runs the interrupt service routine
- For every interrupt, there is a fixed location in memory that holds the address of its ISR
- The group of memory locations set aside to hold the addresses of ISRs is called interrupt vector table

Steps for executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps

1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack
2. It also saves the current status of all the interrupts internally (i.e: not on the stack)
3. It jumps to a fixed location in memory, called the interrupt vector table that holds the address of the ISR

4. 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 until it reaches the last instruction of the subroutine which is RETI (return from interrupt)

5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted

- First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC
- Then it starts to execute from that address

Interrupts in 8051

Six Interrupts in 8051 those are mentioned in below

- Reset – power-up reset
- Two interrupts are set aside for the timers: one for timer 0 and one for timer 1
- Two interrupts are set aside for hardware external interrupts
- - P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2)
- Serial communication has a single interrupt that belongs to both receive and transfer

Interrupt vector table

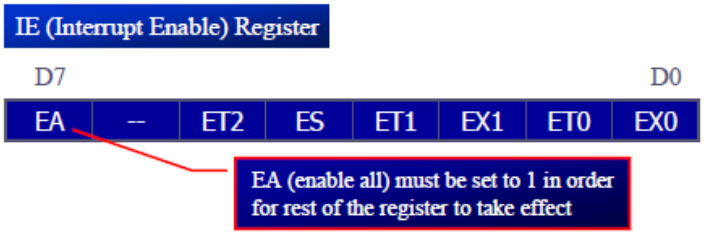
Interrupt	ROM Location (hex)	Pin
Reset	0000	9
External HW (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
External HW (INT1)	0013	P3.3 (13)
Timer 1 (TF1)	001B	
Serial COM (RI and TI)	0023	

Enabling and Disabling an Interrupt

Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated

The interrupts must be enabled by software in order for the microcontroller to respond to them

- There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts



EA	IE.7	Disables all interrupts
--	IE.6	Not implemented, reserved for future use
ET2	IE.5	Enables or disables timer 2 overflow or capture interrupt (8952)
ES	IE.4	Enables or disables the serial port interrupt
ET1	IE.3	Enables or disables timer 1 overflow interrupt
EX1	IE.2	Enables or disables external interrupt 1
ET0	IE.1	Enables or disables timer 0 overflow interrupt
EX0	IE.0	Enables or disables external interrupt 0

To enable an interrupt, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect

2. The value of EA

- If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high
- If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high

Timer Interrupts

The timer flag (TF) is raised when the timer rolls over

- In polling TF, we have to wait until the TF is raised

  - The problem with this method is that the microcontroller is tied down while waiting for TF to be raised, and can not do anything else
- Using interrupts solves this problem and, avoids tying down the controller

  - If the timer interrupts in the IE register is enabled, whenever the timer rolls over, TF is raised, and the microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR

  - In this way, the microcontroller can do other until it is notified that the timer has rolled over 1

/* Write a program that continuously get 8-bit data from P3 and sends it to P1 while simultaneously creating a square wave of 200 µs period on pin P2.1. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz. */

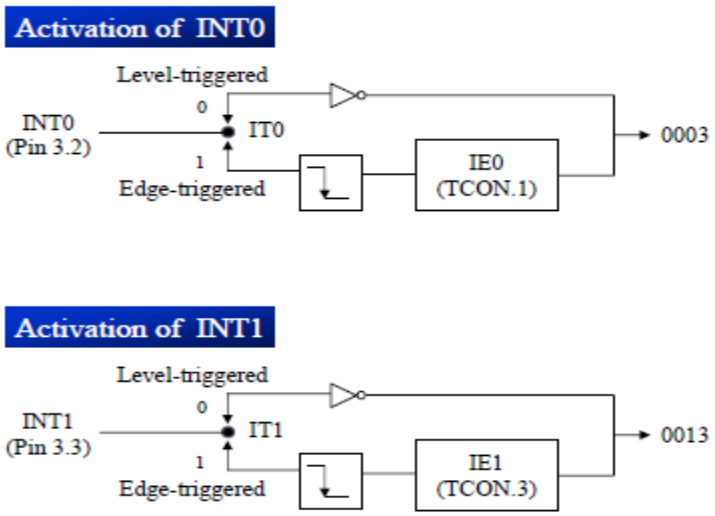
```
/*We will use timer 0 in mode 2 (auto reload). TH0 = 100/1.085 us = 92*/
/*upon wake-up go to main, avoid using memory allocated to Interrupt Vector
Table*/
CSEG AT 0
LJMP MAIN                    ; by-pass interrupt vector table

/*ISR for timer 0 to generate square wave*/
CSEG AT 000BH                ; Timer 0 interrupt vector table
CPL P2.1                     ; toggle P2.1 pin
RETI                         ; return from ISR

/* The main program for initialization */
CSEG AT 0030H            ; after vector table space
MAIN: MOV TMOD, #02H     ; Timer 0, mode 2
MOV P3, #0FFH            ; make P0 an input port
MOV TH0, #0A4H           ; TH0=A4H for -92
MOV IE, #82H             ; IE=10000010 (bin) enable Timer 0
SETB TR0                 ; Start Timer 0
BACK: MOV A, P3          ; get data from P3
CPL A                    ; compliment "A" value
MOV P1, A                ; issue it to P1
MOV P3, A                ; issue to P3
SJMP BACK                ; keep doing it loop unless interrupted by TF0
END                      ; it ends the program
```

Note: Same process to Timer1 we have change timer mode TH and TL values after check TF1 flag that will be discussed in class.

External Hardware Interrupts

The 8051 has two external hardware interrupts

- Pin 12 (P3.2) and pin 13 (P3.3) of the 8051, designated as INT0 and INT1, are used as external hardware interrupts
  - The interrupt vector table locations 0003H and 0013H are set aside for INT0 and INT1
- There are two activation levels for the external hardware interrupts
  - Level triggered
  - Edge triggered



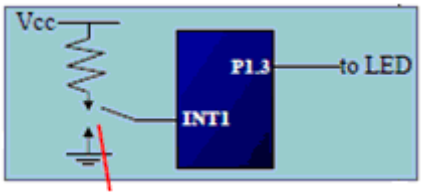
Level Triggered Interrupt

In the level-triggered mode, INT0 and INT1 pins are normally high

- If a low-level signal is applied to them, it triggers the interrupt
- Then the microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt
- The low-level signal at the INT pin must be removed before the execution of the last instruction of the ISR, RETI; otherwise, another interrupt will be generated

This is called a level-triggered or level activated interrupt and is the default mode upon reset of the 8051

/* assume that the INT1 pin is connected to a switch that is normally high. Whenever it goes low, it should turn on an LED. The LED is connected to P1.3 and is normally off. When it is turned on it should stay on for a fraction of a second. As long as the switch is pressed low, the LED should stay on. */



Solution:

```
CSEG AT 0000H
LJMP MAIN                    ; by-pass interrupt vector table

/*ISR for INT1 to turn on LED*/

CSEG AT 0013H                ; INT1 ISR
SETB P1.3                    ; turn on LED
MOV R3, #255
BACK: DJNZ R3, BACK          ; keep LED on for a while
CLR P1.3                     ; turn off the LED
RETI                         ; return from ISR

/*MAIN program for initialization*/

CSEG AT 30H
MAIN: MOV IE, #10000100B; enable external INT 1(IE= 84H)
HERE: SJMP HERE          ; stay here until get interrupted

END                      ; it ends the program
```

Note: Same process to Interrupt0 we have to change IE register values after check INT0 (P3.2) pin that will be discussed in class.
On reset, IT0 (TCON.0) and IT1 (TCON.2) are both low, making external interrupt level-triggered

Edge-Triggered Interrupt

To make INT0 and INT1 edge triggered interrupts, we must program the bits of the TCON register

- The TCON register holds, among other bits, the IT0 and IT1 flag bits that determine level- or edge-triggered mode of the hardware interrupt
  - IT0 and IT1 are bits D0 and D2 of the TCON register
  - They are also referred to as TCON.0 and TCON.2 since the TCON register is Bit addressable

TCON (Timer/Counter) Register (Bit- Addressable)

D7							D0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 on/off
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter 0 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn timer/counter 0 on/off
IE1	TCON.3	External interrupt 1 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt
IE0	TCON.1	External interrupt 0 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt

/* Assume that pin 3.2 (INT0) is connected to a pulse generator, write a program in which the falling edge of the pulse will send a high to P1.3, which is connected to an LED (or buzzer). In other words, the LED is turned on and off at the same rate as the pulses are applied to the INT0 pin. */

```
CSEG AT 0000H
LJMP MAIN                       ; by-pass interrupt vector table

/*ISR for INT0 to turn on LED*/

CSEG AT 0003H                   ; INT0 ISR
SETB P1.3                       ; turn on LED
MOV R3, #255
BACK: DJNZ R3, BACK             ; keep LED on for a while
CLR P1.3                        ; turn off the LED
RETI                            ; return from ISR

/*MAIN program for initialization*/

CSEG AT 30H
MAIN: SETB TCON.0               ; make INT0 edge-triggered int.
MOV IE, #10000001B              ; enable external INT 0(IE= 81H)
HERE: SJMP HERE                 ; stay here until get interrupted

END                             ; it ends the program
```

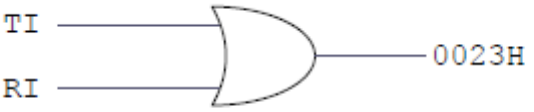
Note: Same process to External Interrupt 1 just we have to change IE register value and set TCON.3 value for edge triggered according to requirement

Serial Communication Interrupt

- TI (transfer interrupt) is raised when the last bit of the framed data, the stop bit, is transferred, indicating that the SBUF register is ready to transfer the next byte
- RI (received interrupt) is raised when the entire frame of data, including the stop bit, is received

In the 8051 there is only one interrupt set aside for serial communication

- This interrupt is used to both send and receive data
- If the interrupt bit in the IE register (IE.4) is enabled, when RI or TI is raised the 8051 gets interrupted and jumps to memory location 0023H to execute the ISR
- In that ISR we must examine the TI and RI flags to see which one caused the interrupt and respond accordingly



Serial interrupt is invoked by TI or RI flags

/* Write a program in which the 8051 gets data from P1 and sends it to P2 continuously while incoming data from the serial port is sent to P0. Assume that XTAL=11.0592. Set the baud rata at 9600 */

```
CSEG AT 0000H
LJMP MAIN

CSEG AT 23H
LJMP SERIAL             ; jump to serial interrupt ISR

CSEG AT 30H
MAIN: MOV P1, #0FFH     ; make P1 an input port
MOV IE, 10010000B       ; enable serial int.
MOV TMOD, #20H          ; timer 1, auto reload
MOV TH1, #0FDH          ; 9600 baud rate
MOV SCON, #50H          ; 8-bit, 1 stop, "REN" enabled
SETB TR1                ; start timer 1
BACK: MOV P1, A         ; read data from A send it to P1
MOV P2, A               ; also send it to P2
SJMP BACK               ; stay in loop indefinitely
```

```
CSEG AT 100H
SERIAL: JB TI, TRANS     ; jump if TI is high
MOV A, SBUF              ; otherwise due to receive

MOV P0, A                ; send incoming data to P0
CLR RI                   ; clear RI since CPU doesn't
RETI                     ; return from ISR
TRANS: CLR TI            ; clear TI since CPU doesn't
RETI                     ; return from ISR
END
```

/* Write an ECP to generate EXT Int0, Timer 0 and Serial Interrupt */

```
#include<reg51.h> // It includes all SFRs addresses in this header file
                  // (Predefined)

#define LEDS P1   // declare port for 8 leds
sbit LED1=P0^0;   // declare port pin for led
sbit LED2=P0^1;   // declare port pin for led
sbit LED3=P0^2;   // declare port pin for led

/*Interrupt service routine function for external interrupt 0*/

void ExtInt0(void) interrupt 0
{
  LED1=~LED1;                 // compliment led1
}

/*Interrupt service routine function for timer 0*/

void T0_Int(void) interrupt 1
{
  LED2=~LED2;                 // compliment led2
}
```

```
/*Interrupt service routine function for serial interrupt*/

void SerTxRxInt(void) interrupt 4
{
  if(RI==1)                          // if mc receives the data
  {
   LED3=~LED3;                       // compliment led3
   RI=0;                             // RI flag make it zero
  }
  if(TI==0)                          // if mc not transmits the data
  {
     TI=0;                           // TI flag make it zero
  }

}

main()
{
   unsigned char x=0x01;// declare and assign some value to "x"
   EA=EX0=ET0=ES=IT0=1; // Enable 3 interrupts (Ext INT0, Timer0,
                        // Serial Interrupt)
   SCON=0x50;           // select standard UART mode and Enable
                        // reception and transmission
   TMOD=0x21;           // Select timer1 mode2 for UART and
                        // timer0 mode1 for timer0 interrupt
   TH1=TL1=-3;          // pass "-3" value on TH1 for setting the
                        // baud rate (9600)
   TR1=1;               // set timer1 run bit then it starts the timer1
   TR0=1;               // set timer0 run bit then it starts the timer0
```

```
/* Normal Task continuation task */

   while(1)                    // infinite loop
   {
              LEDS=x;          // Assign "x" value to LEDS
              delay(1000);     // wait some time by using delay



                  SBUF='A';          // "A" character is placed in SBUF
                                     // register for transmitting
                  while(TI==0);      // check TI flag
                  x=x<<1;            // x value is left shifted by 1
   }
}
```

Snap Shots for Keil uvision2

How to use Keil uvision2?

1) First we have to open Keiluvision2.So, you can follow below steps

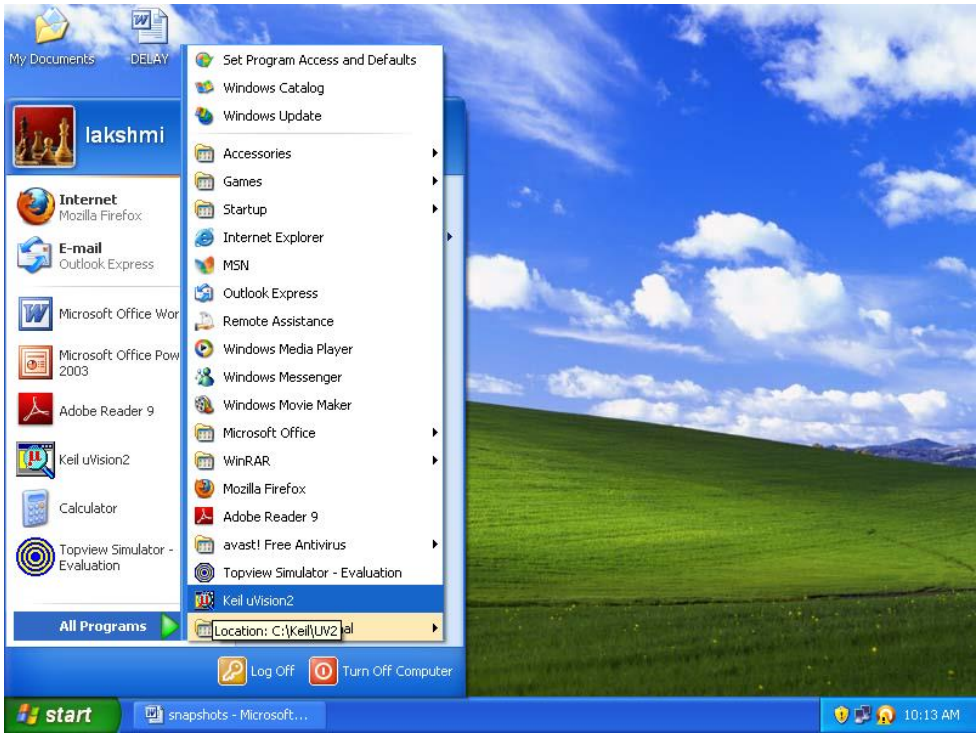
- First double click on Keil uvision2 icon on desktop

Another way is there to open Keil uvision2 i.e.,

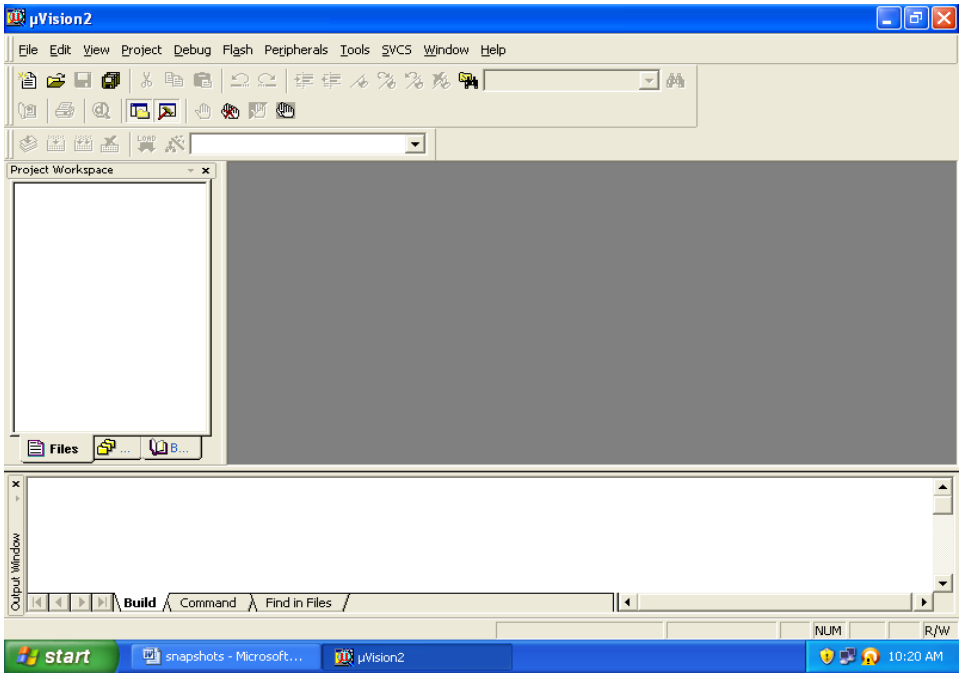
- Click on Start button

- Click on All Programs

- Click on Keil uvision2

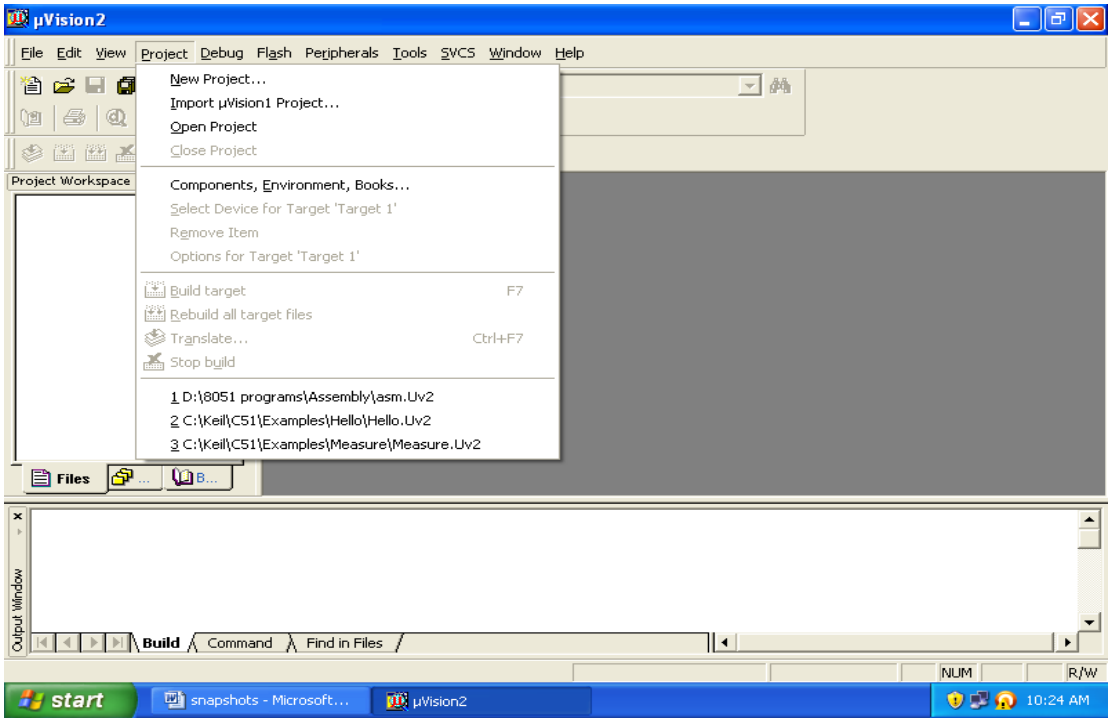


- Now it opens Keil uvision2

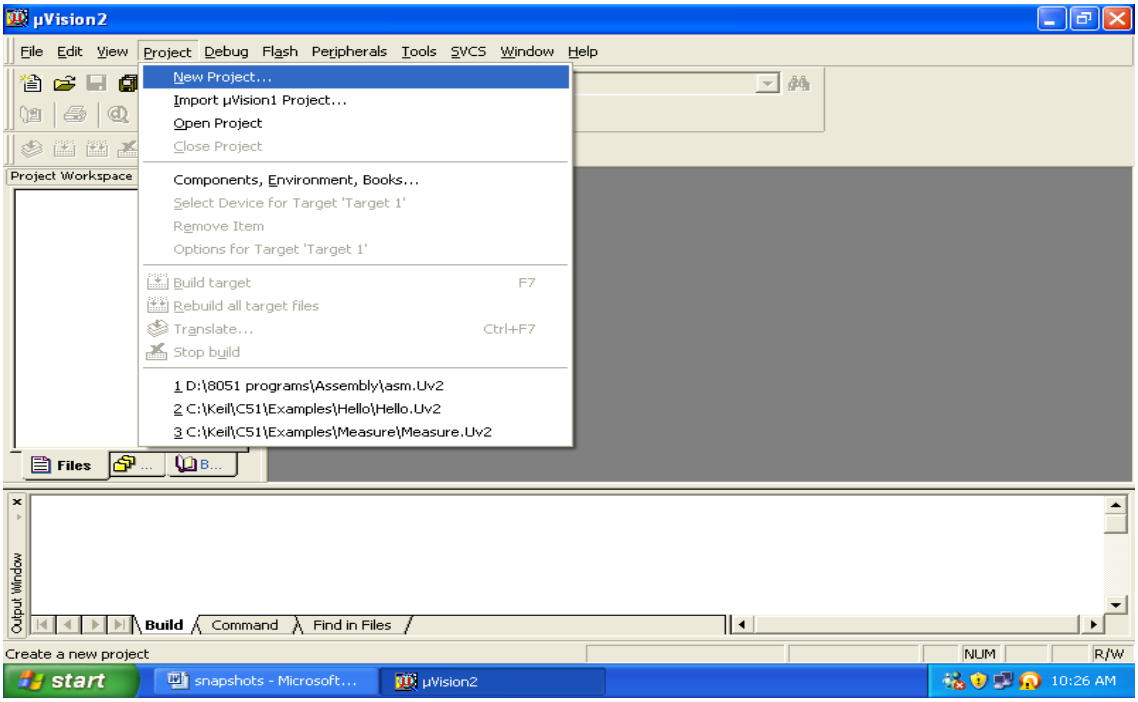


2) Create a Project for that follows these steps

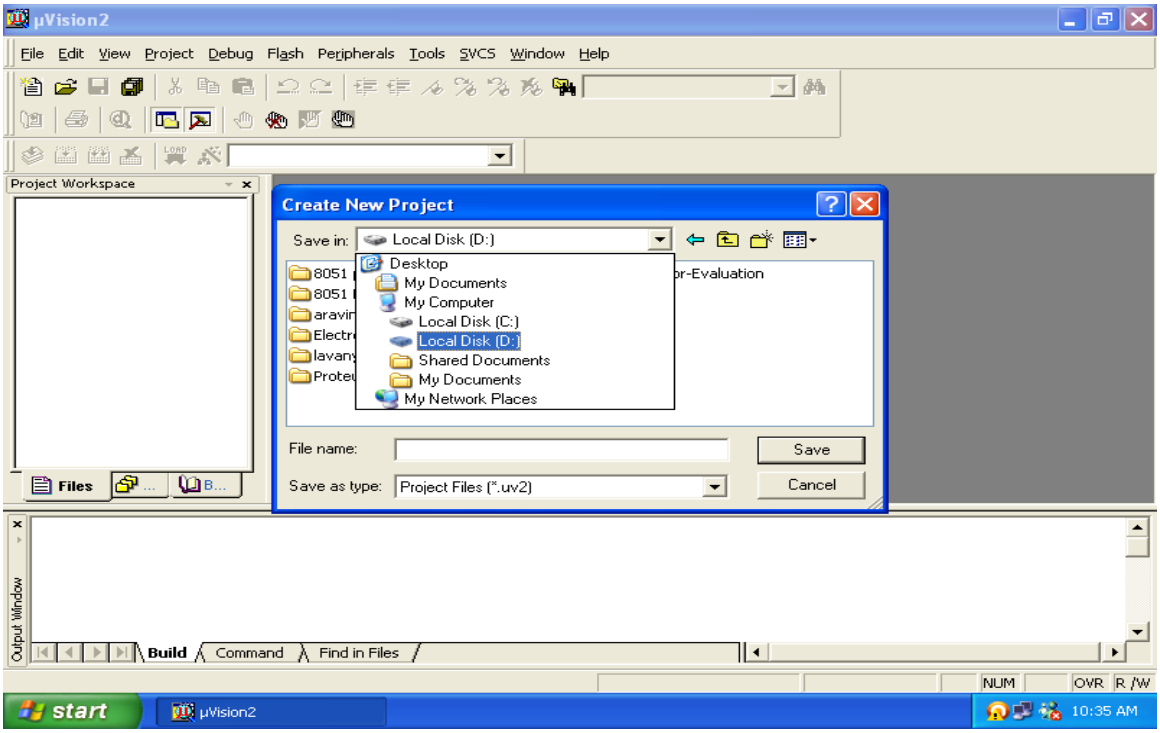
- Click on Project on Menu bar



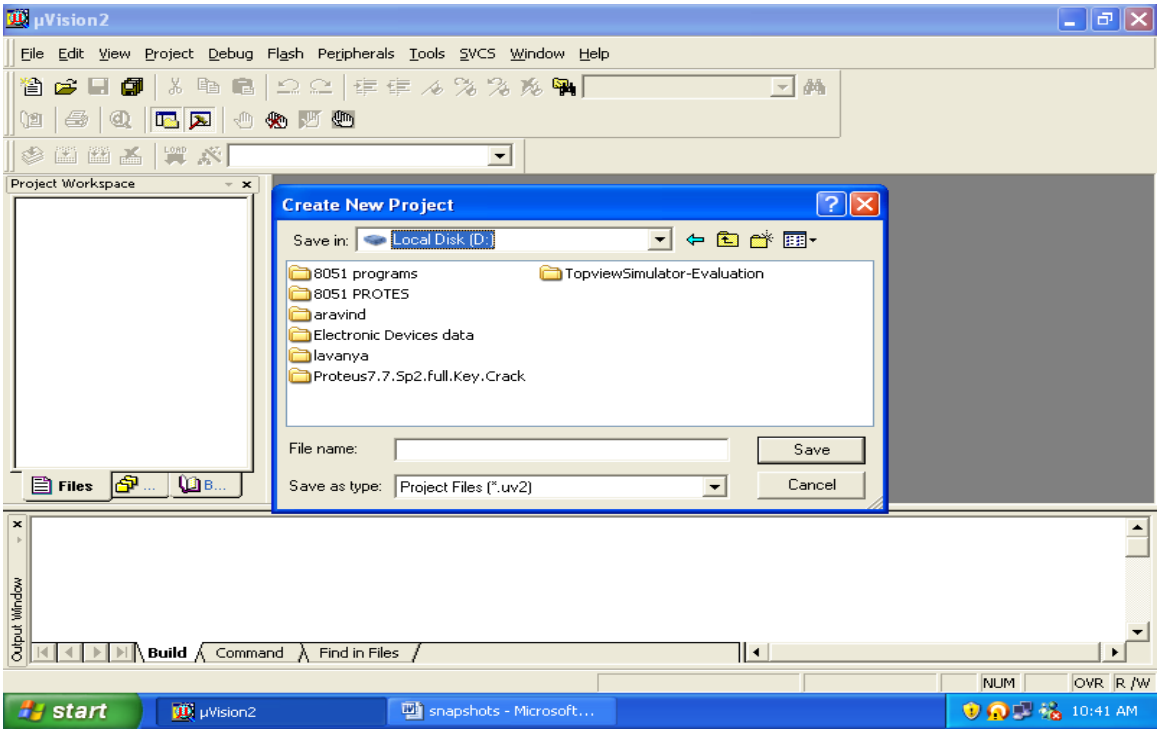
- Click on New Project



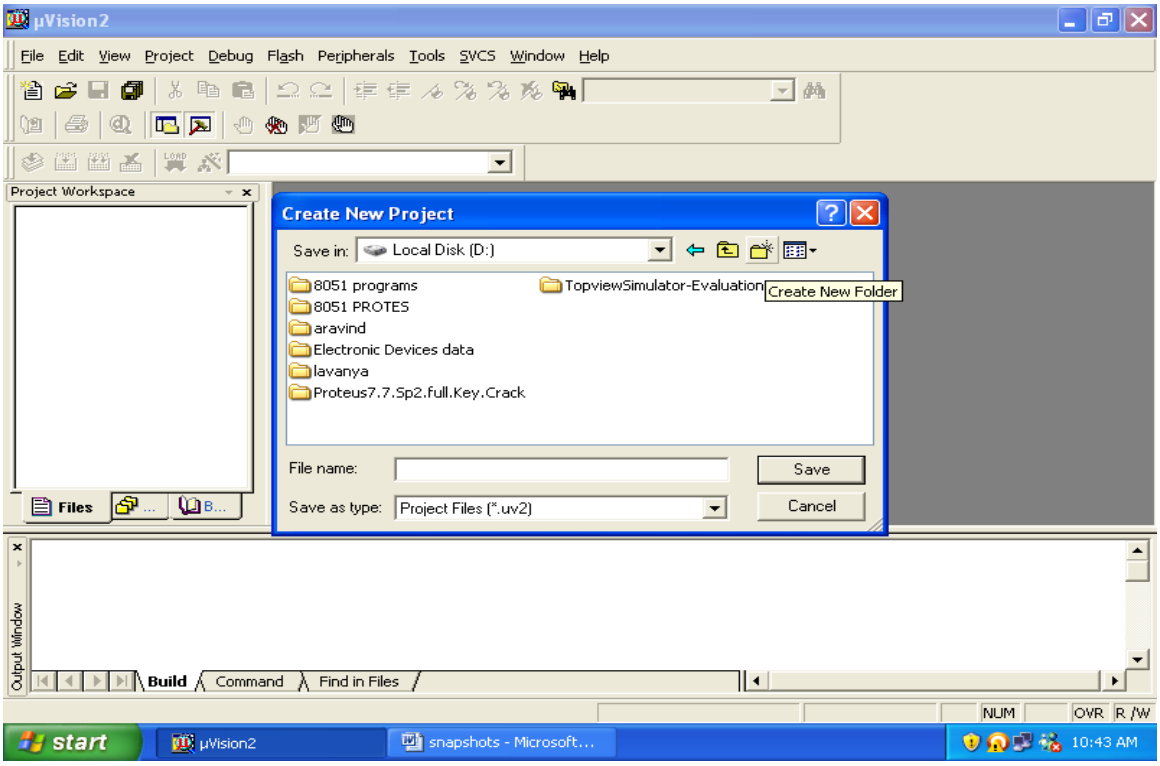
- Now it opens one dialogue box, select path where you want to save your project. So, click on scroll bar click on one drive where you want to save.



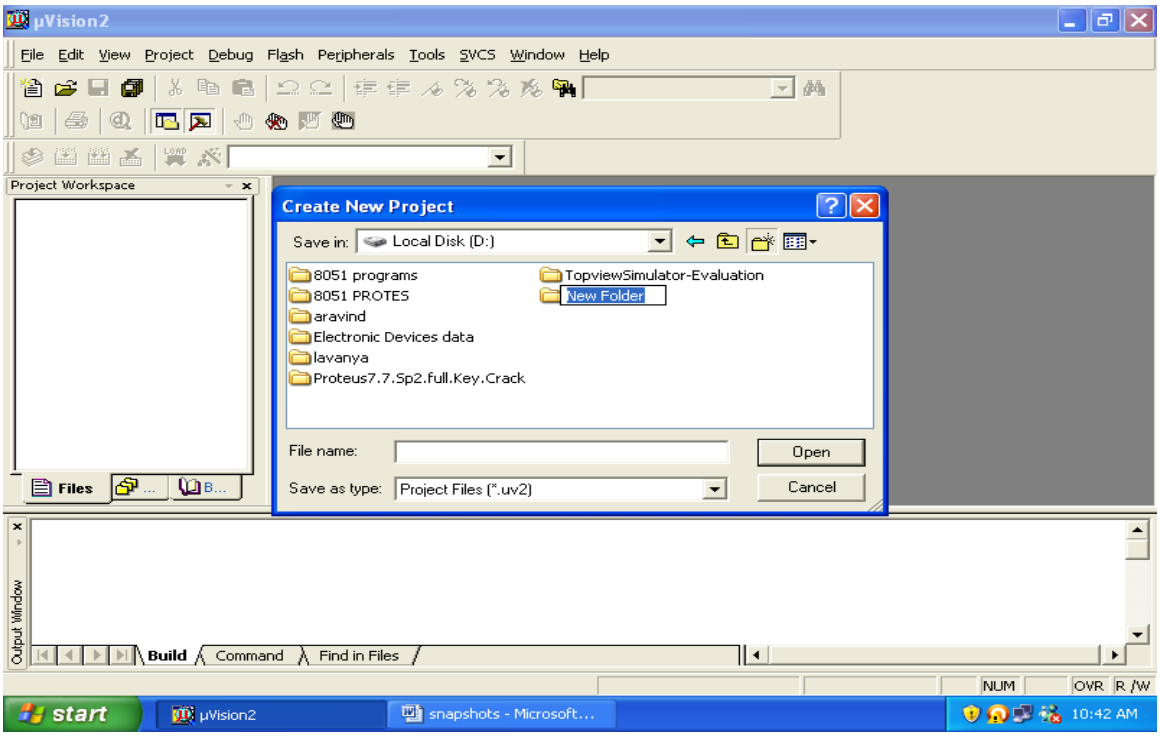
- Now it opens that drive



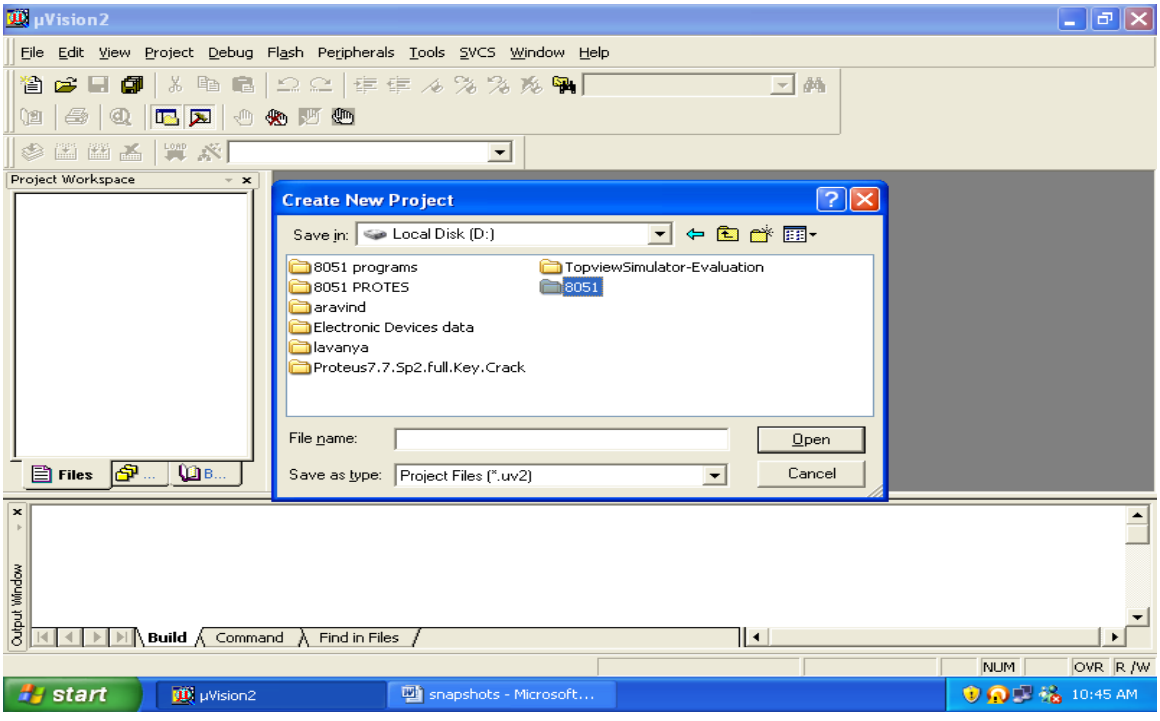
- Click on create new folder icon to save your project in that folder



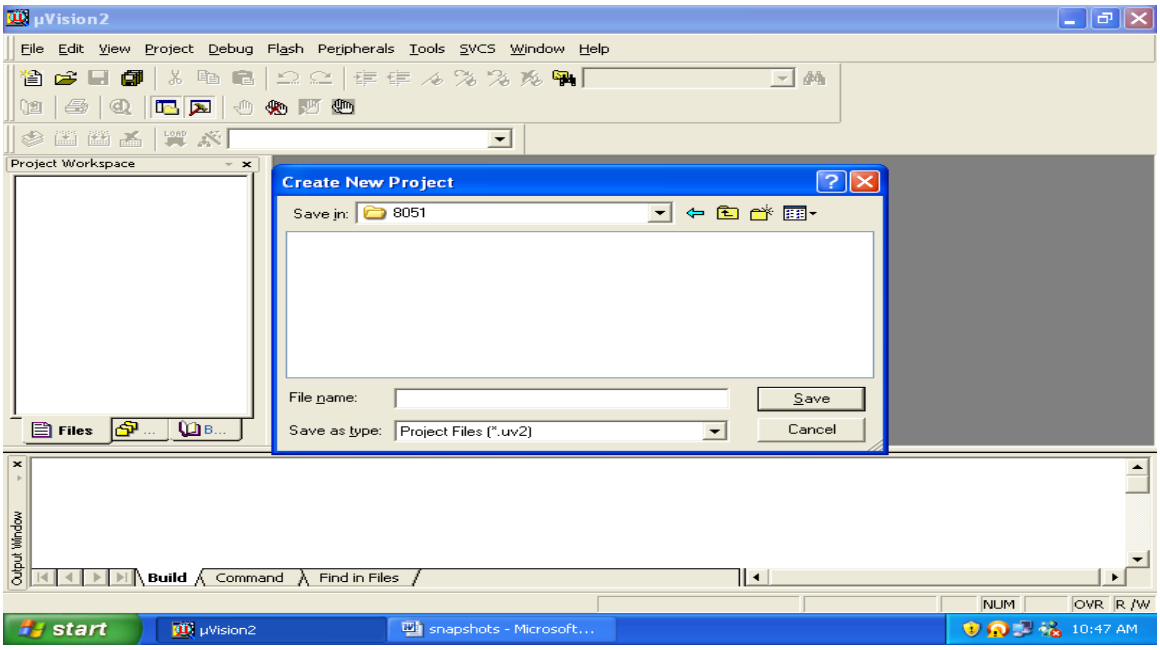
- Now it opens with “New Folder”



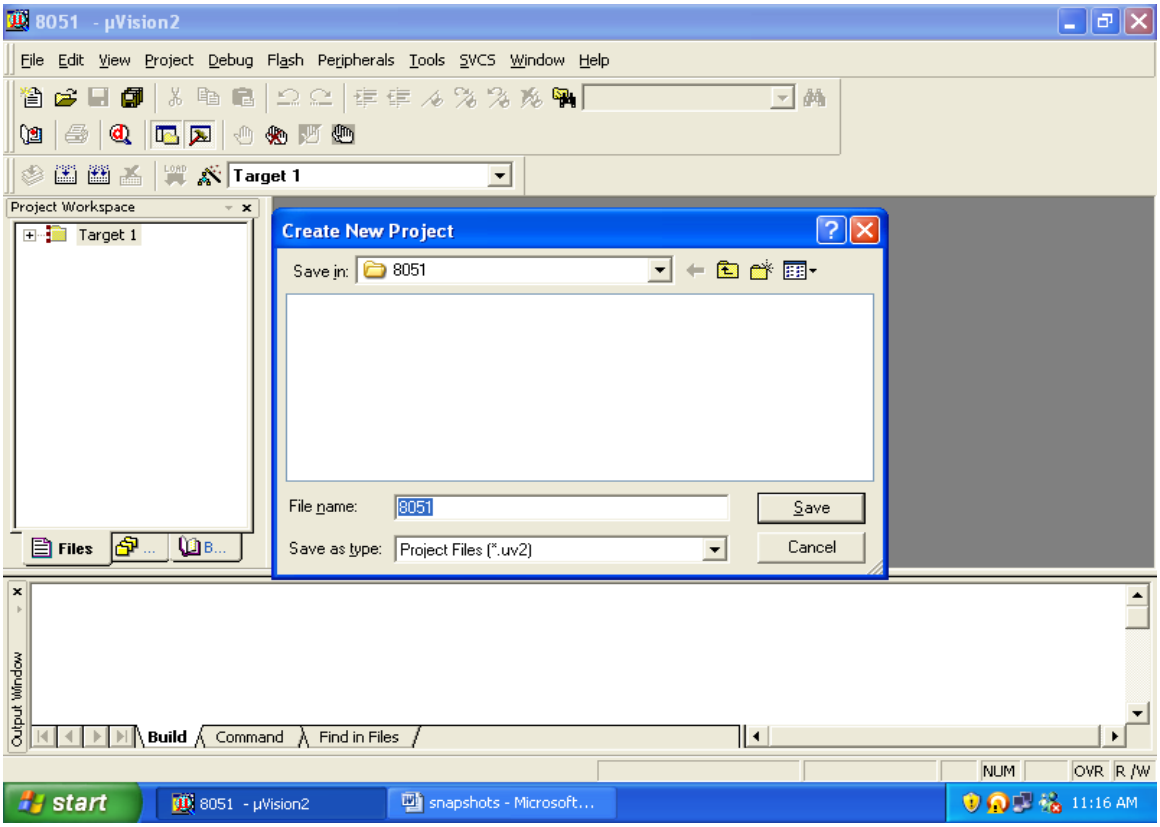
- Give one name to that folder



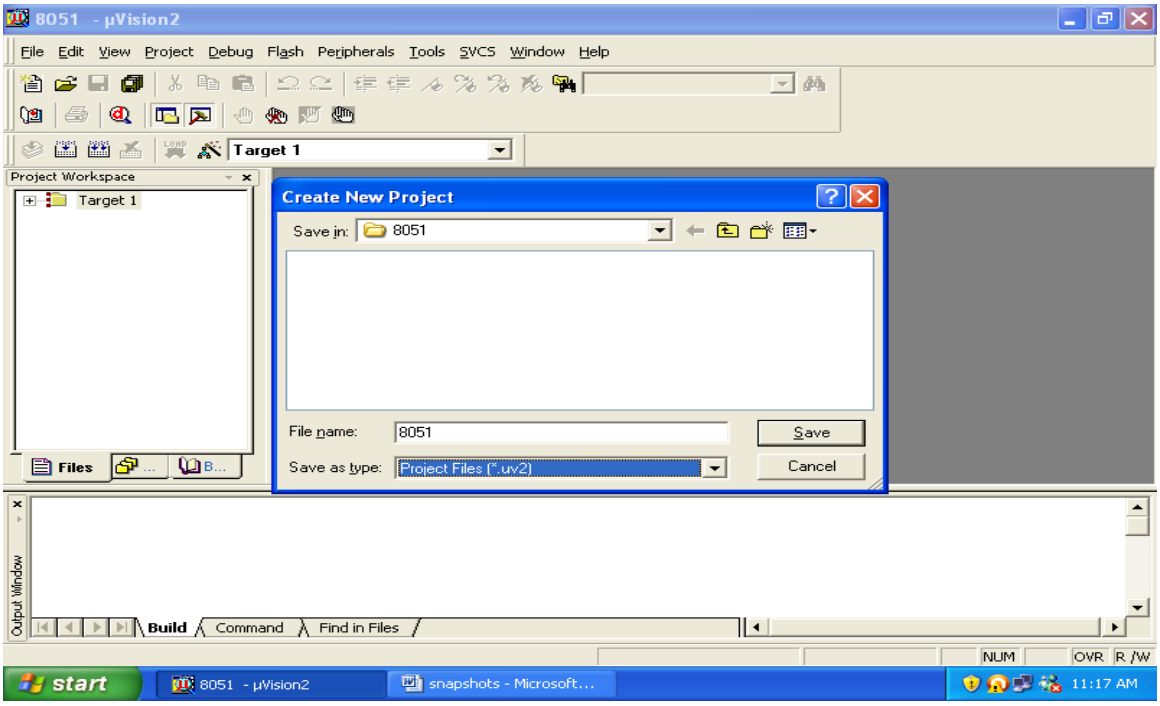
- Double click on that folder now it opens another dialogue box to save your project



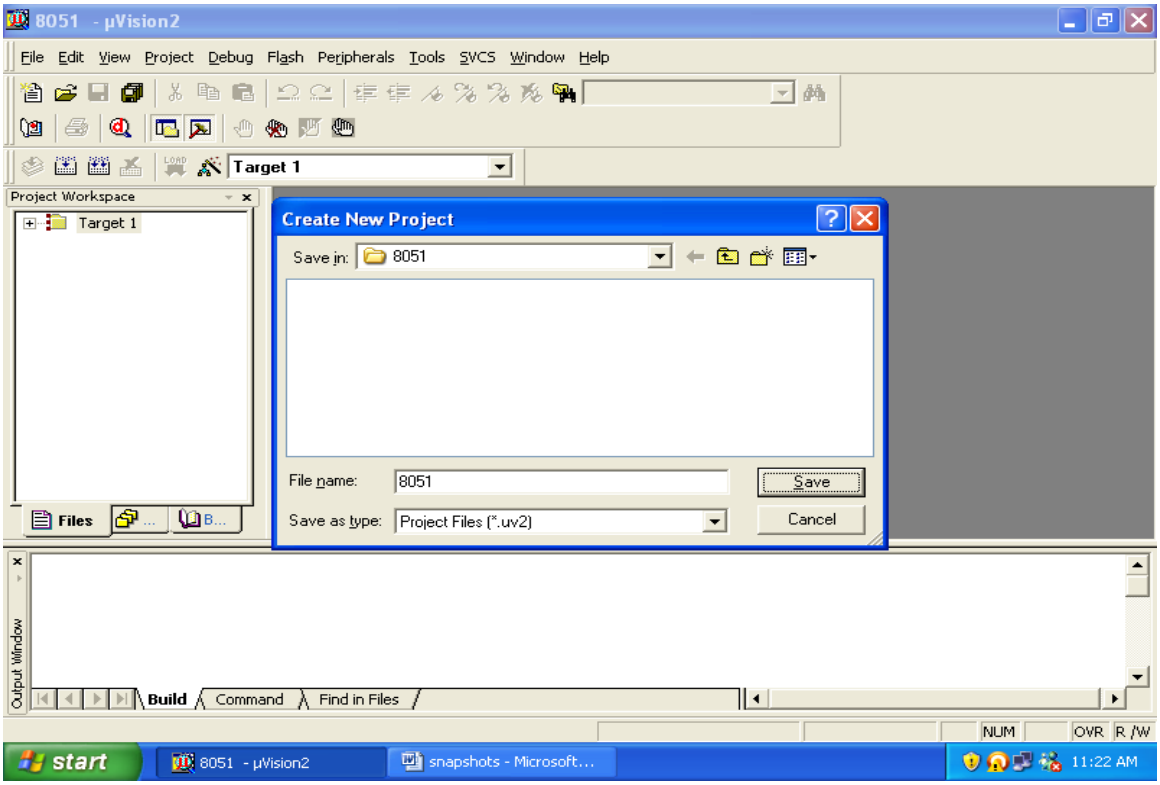
- Give project name in "File name" blank with out any extension



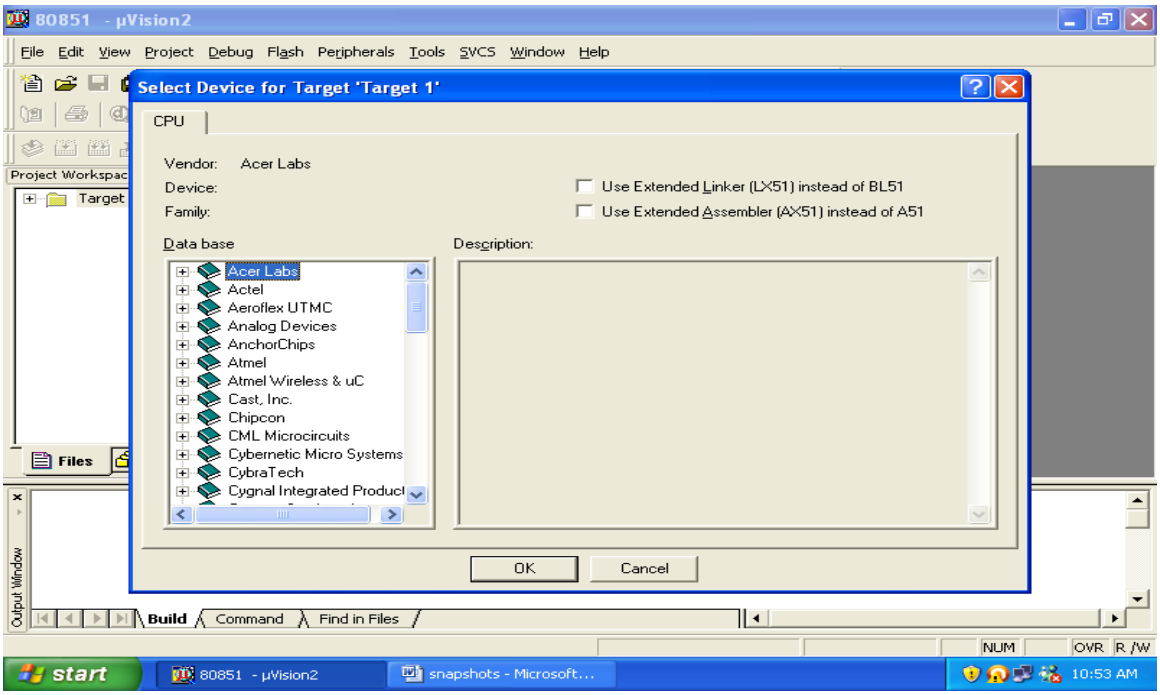
- Select “project files (*.uv2)” in “save as type scroll bar”



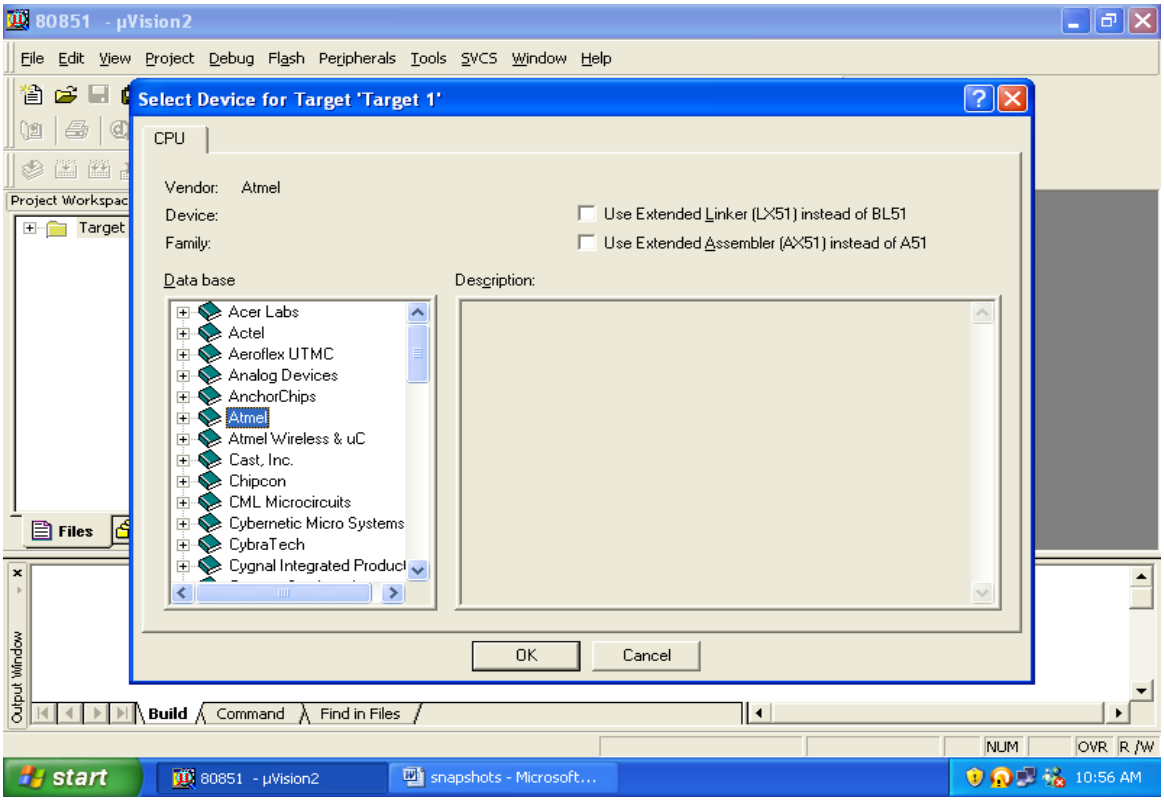
- Click on “Save” button now it opens one dialogue box.



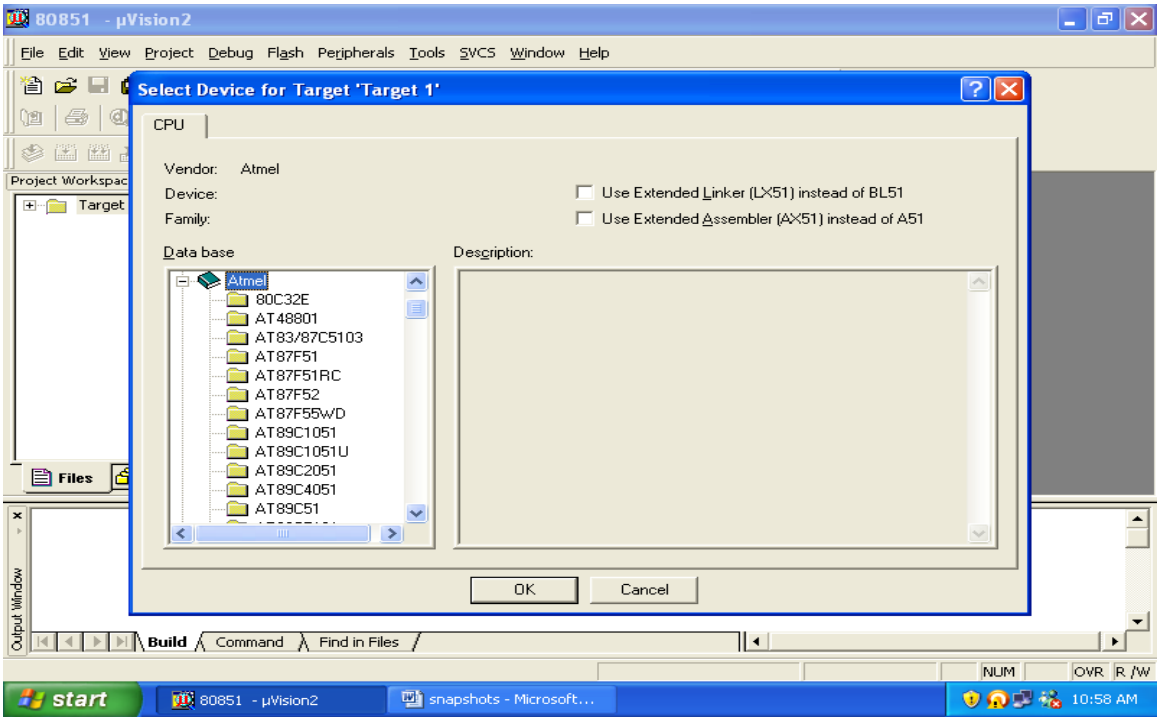
- In that dialogue box they given one list it's related to manufacturer name of IC's. Select one manufacturer name related to your target board.



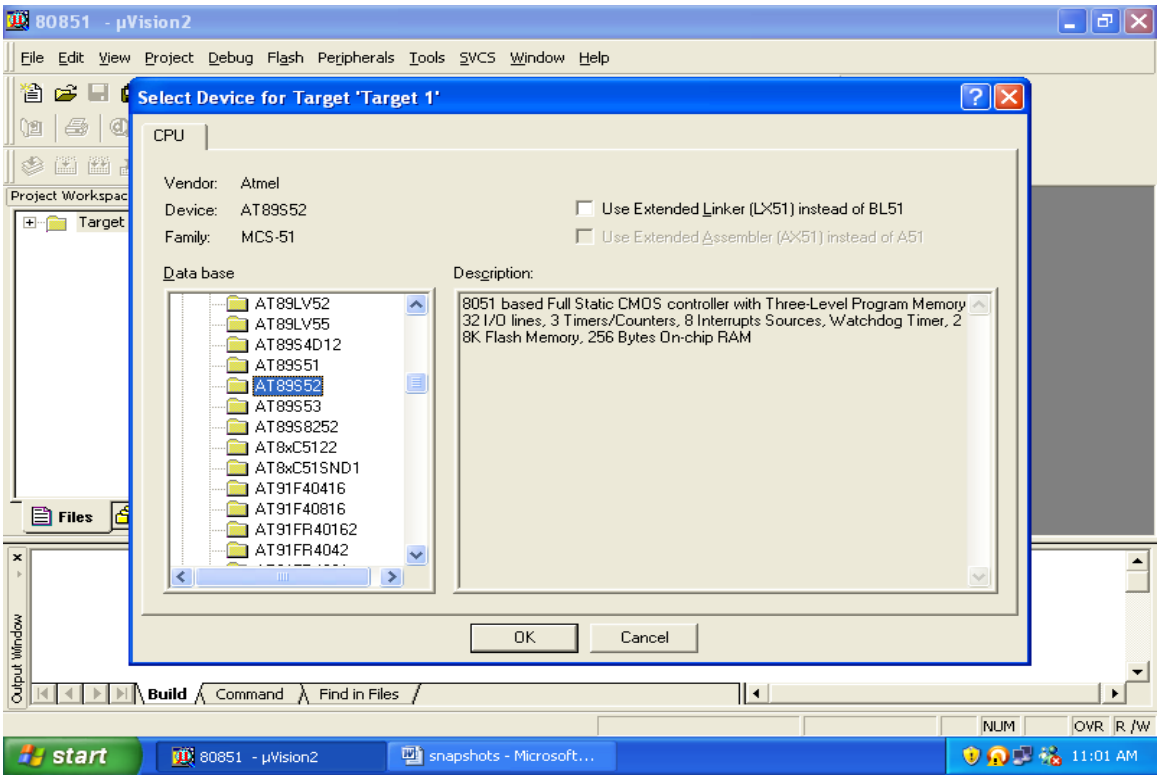
- Select "ATMEL" and click on "+" button



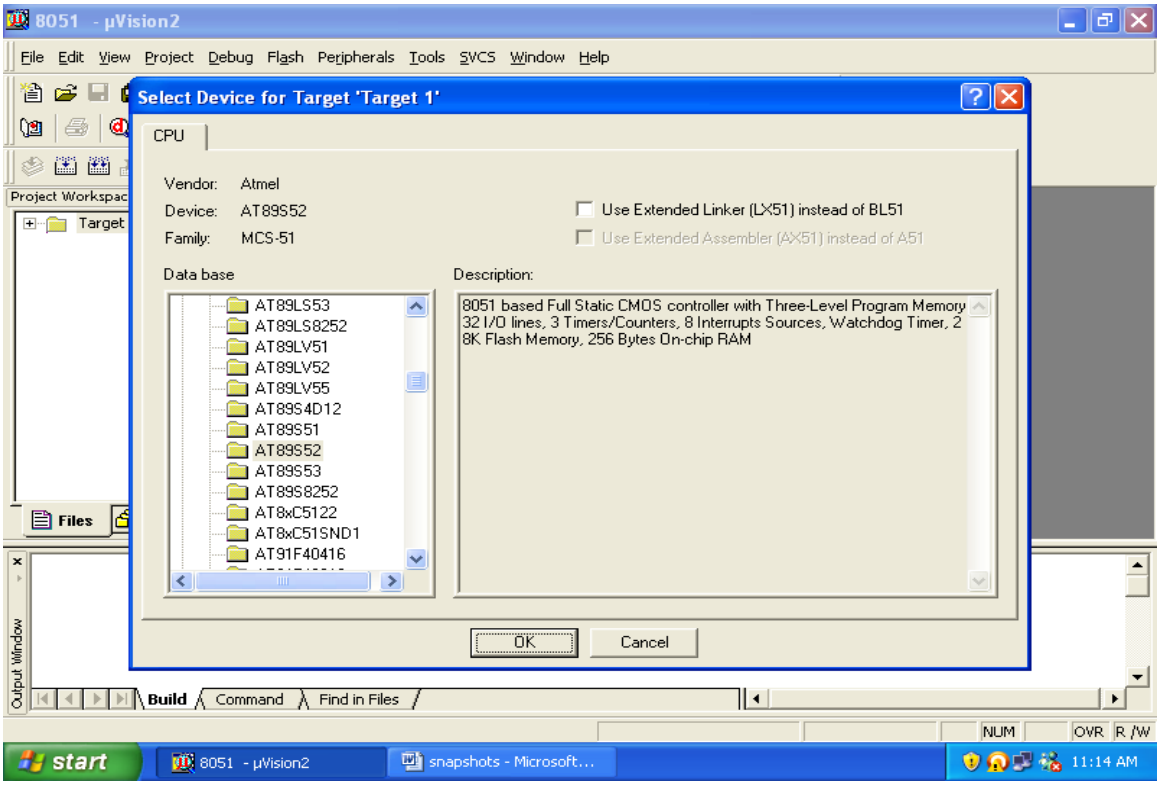
- Now it opens ATMEL related ICs



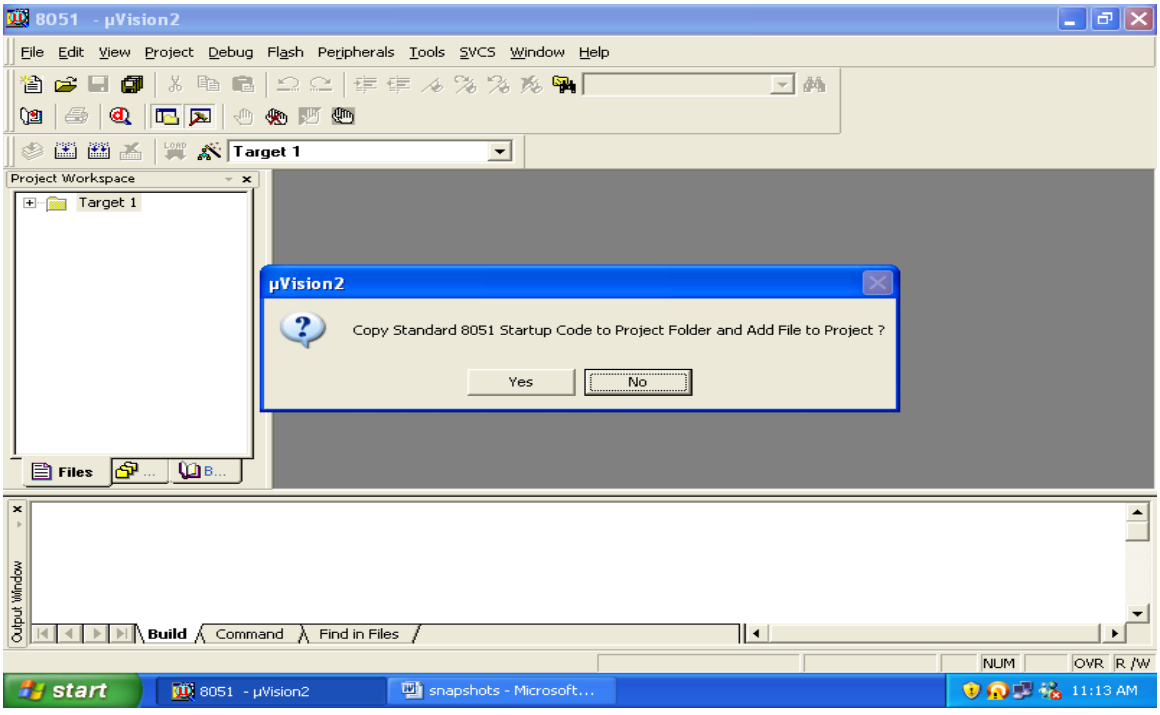
- Search for your required IC with the help of Scroll bar select that IC click on that IC name for example select “89s52”



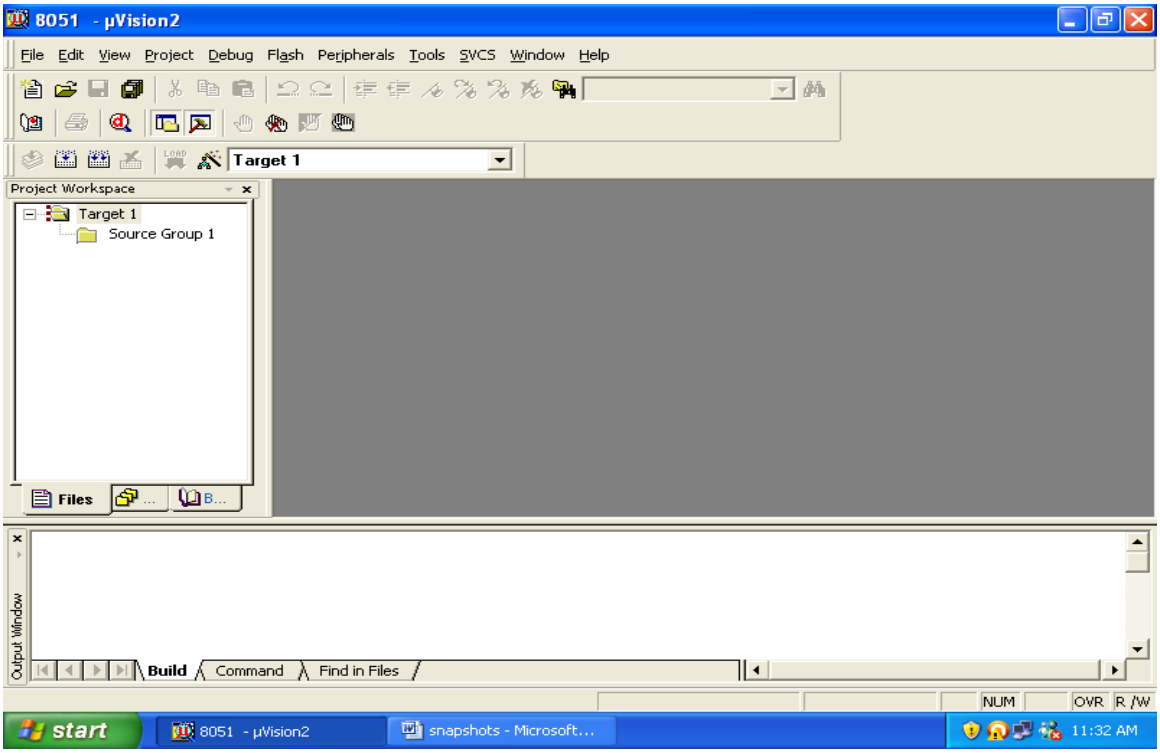
- Then click “OK” button



- It opens one dialogue box it’s related to start up code for 8051 there is no need of start up code. So, click on “NO” button

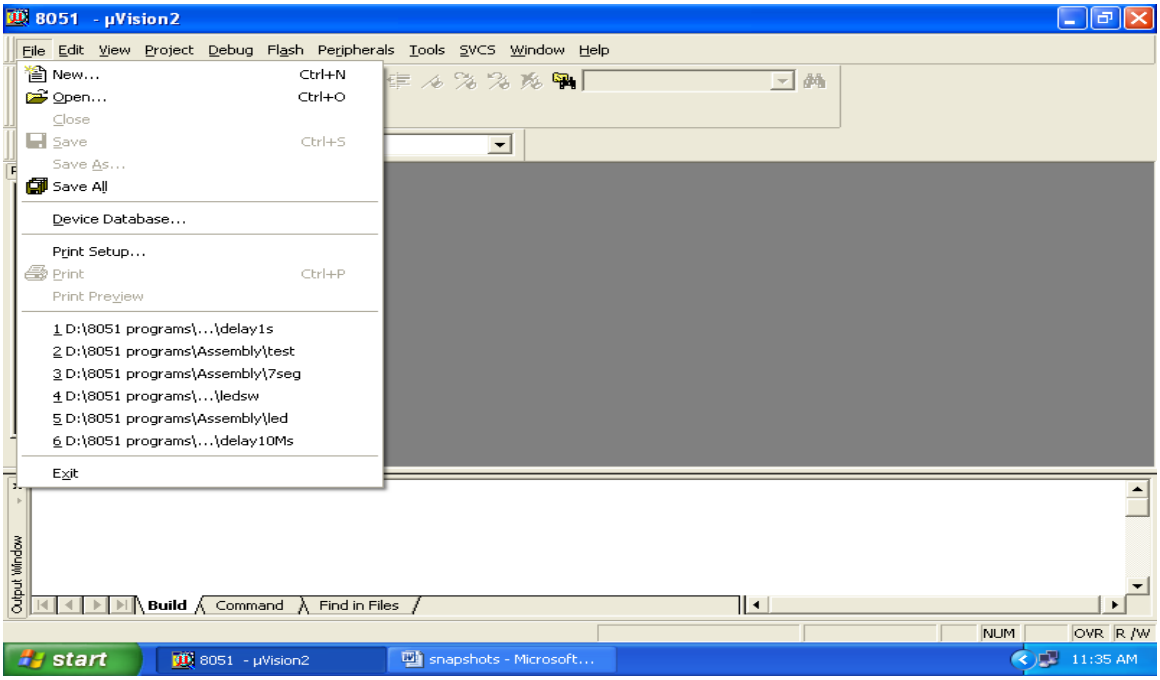


- Now Target is created with the project name of (8051). Project name is your wish

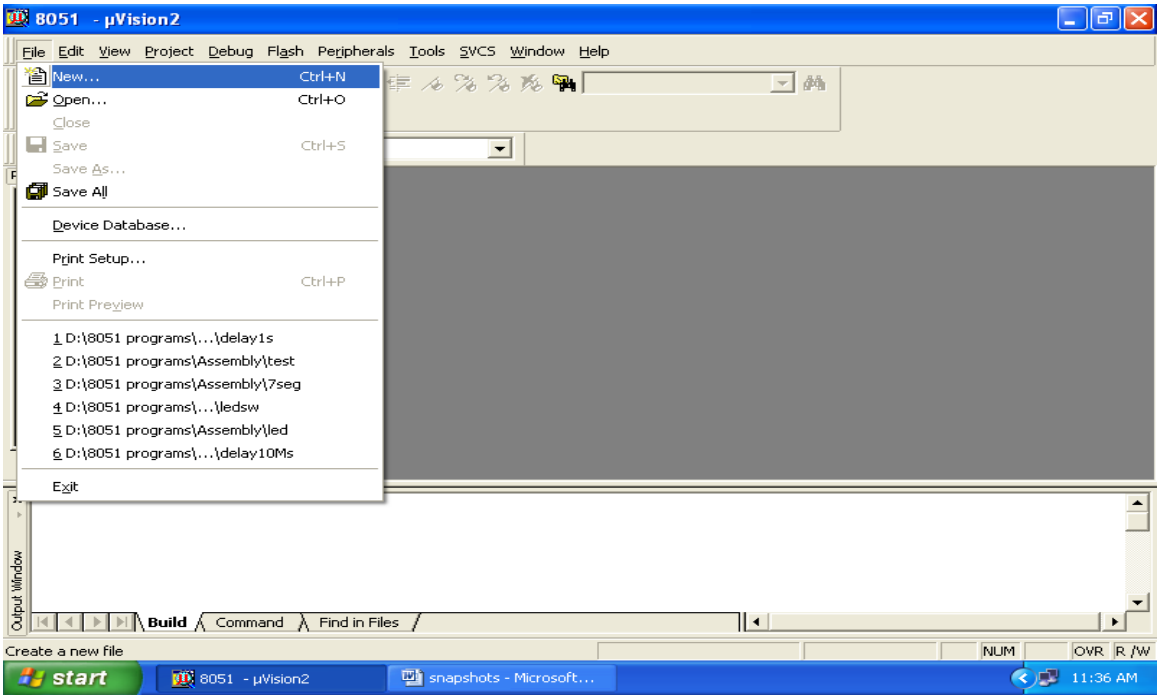


3) Now create a file for that follows these steps

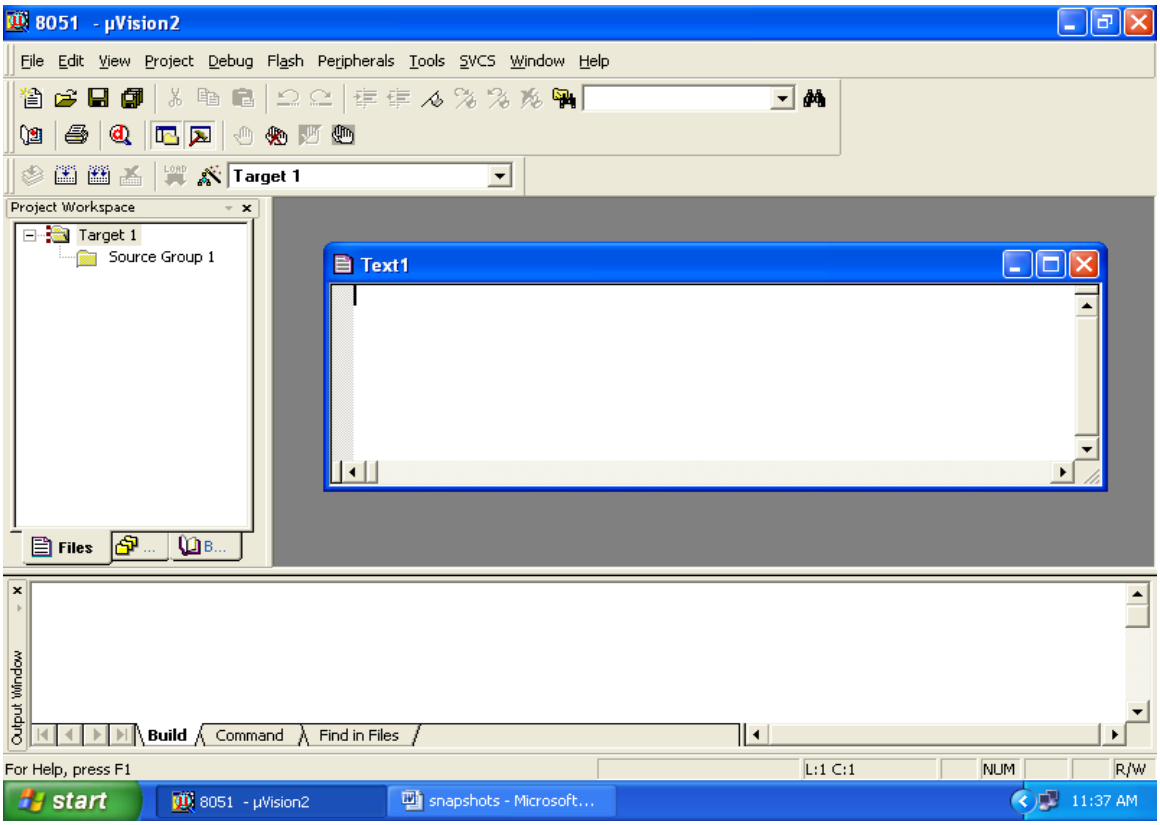
- Click on file on Menu bar



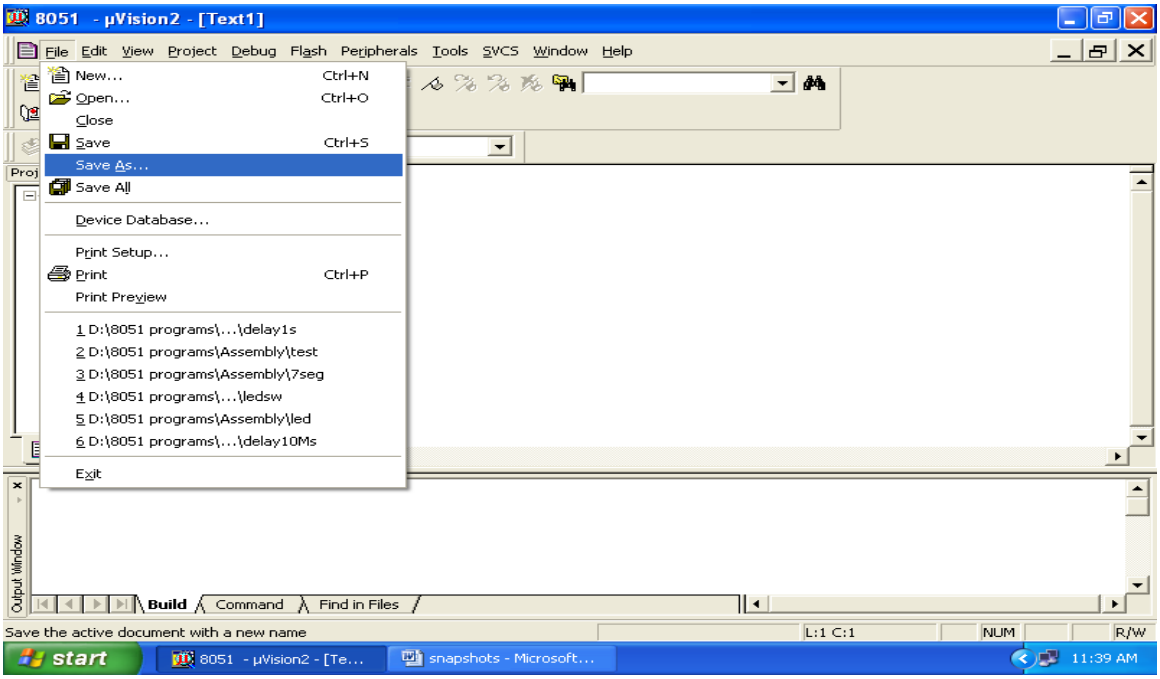
- Click on “New” for creating new file



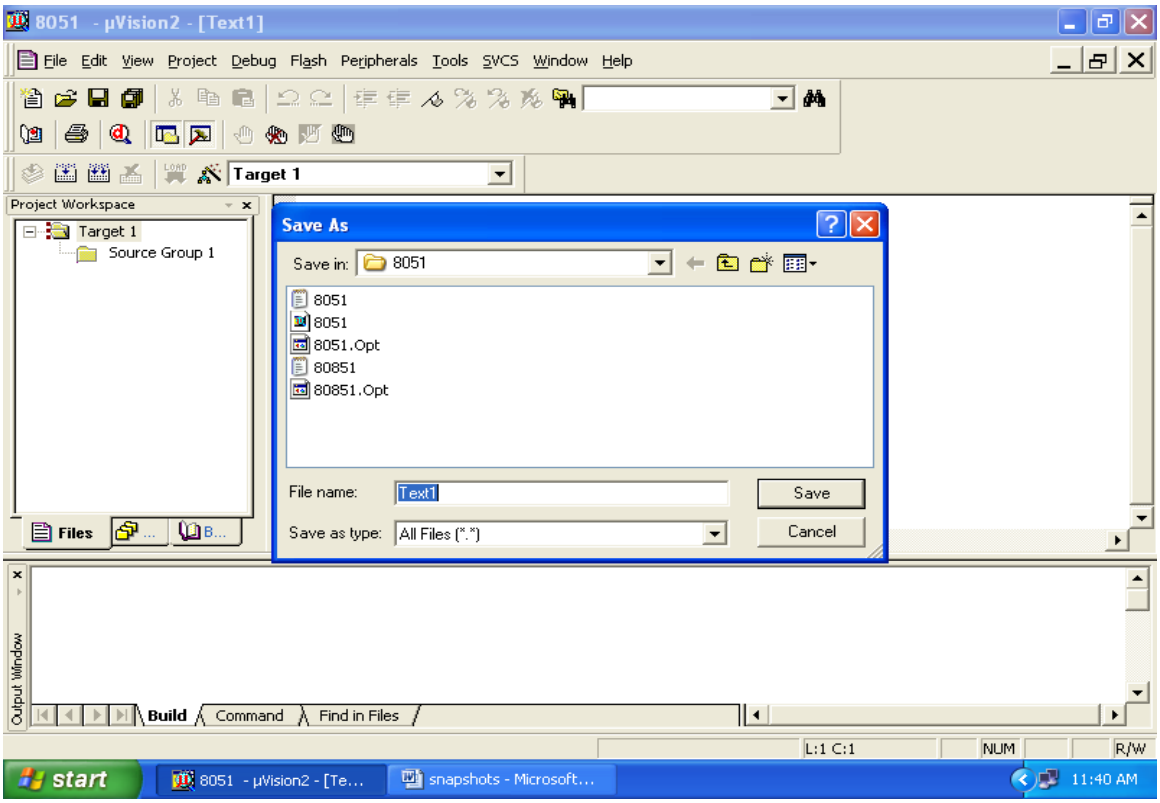
- It opens the file



- Save that file for that again click on “File” menu bar click on save as (short cut key (ctrl+s))

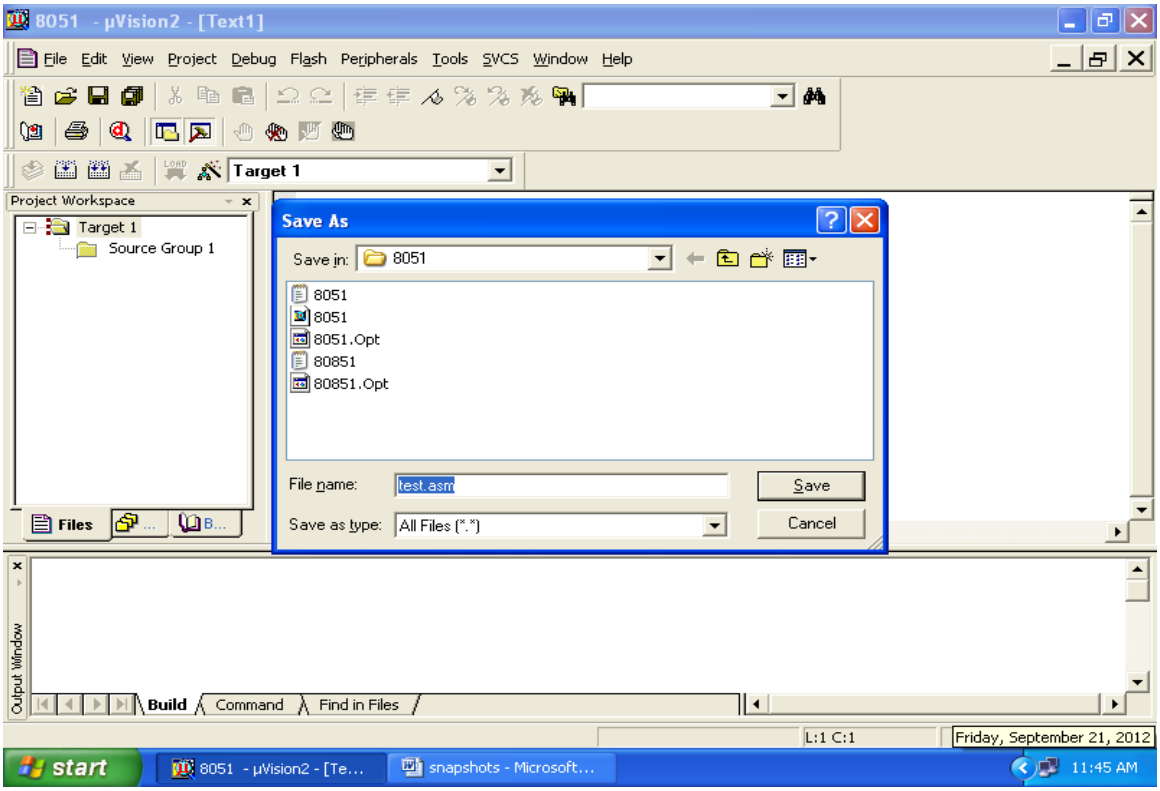


- It opens one dialogue box for saving the file



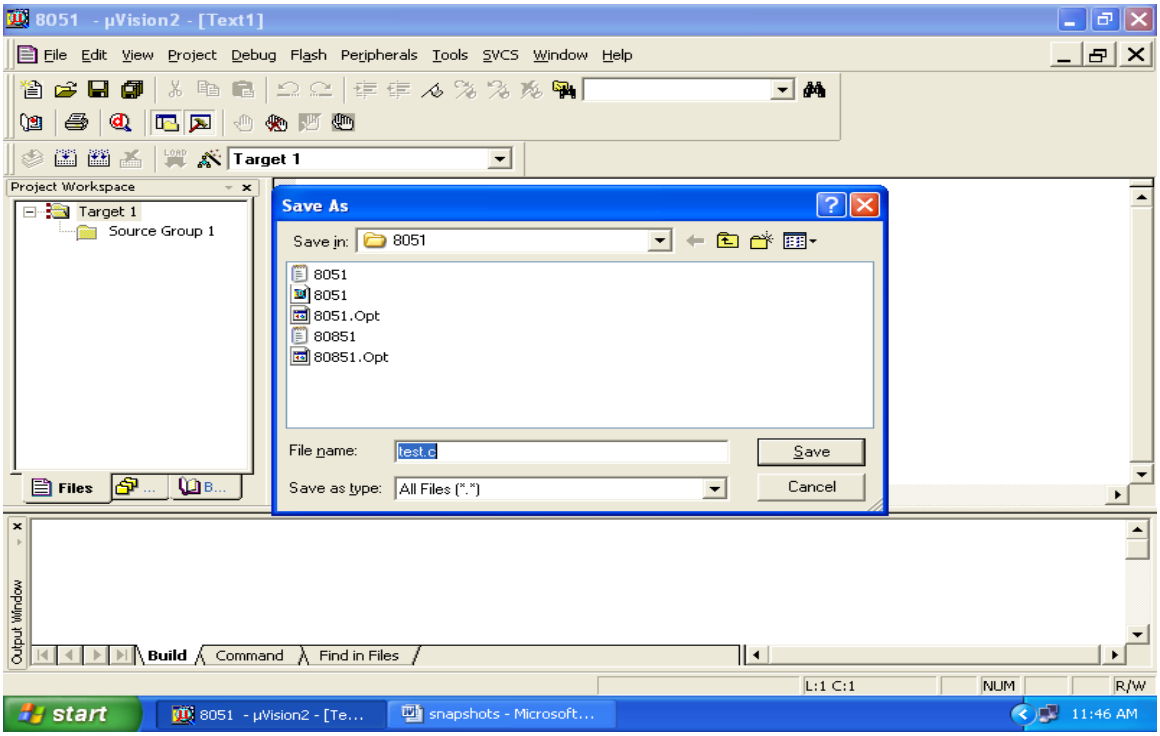
- Give any name in “File name” blank with extension of “.asm” or “.c” based on program

  - If it is Assembly language program save with “.asm” extension, if it is Embedded C programming save with “.c” extension.
  - **For example File Name: “ test.asm”(**asm soured file)

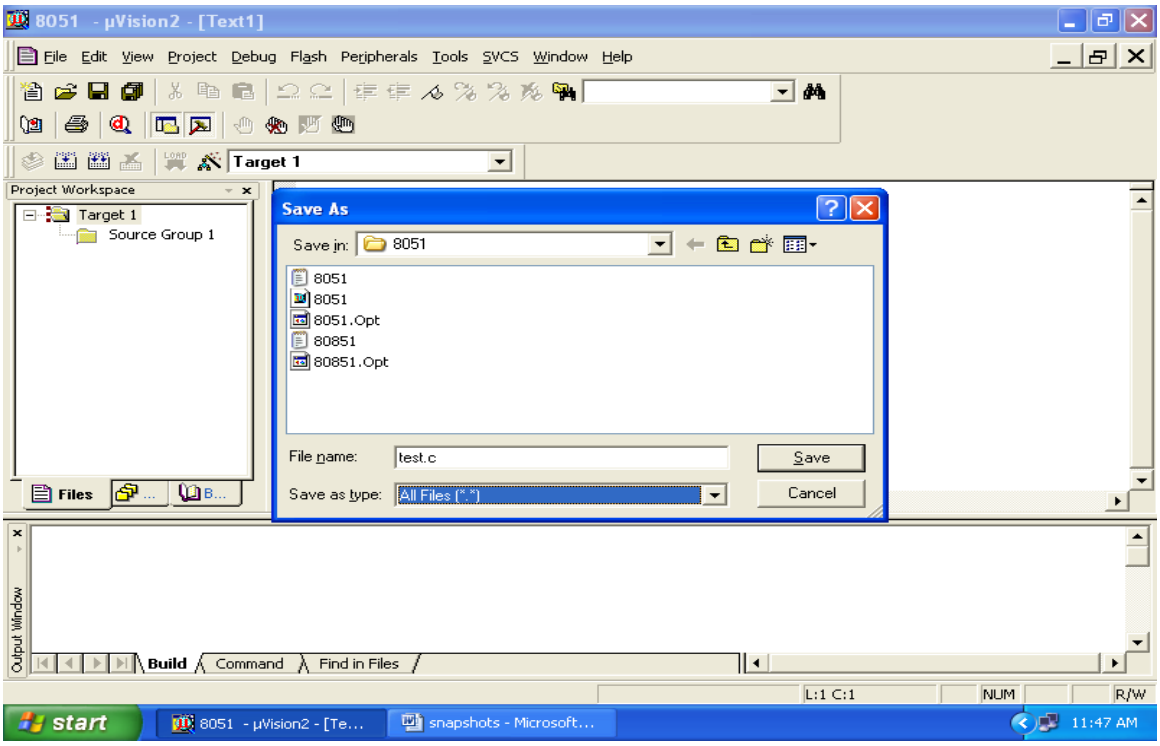


  - **For example File Name: “ test.c”** (c source file)

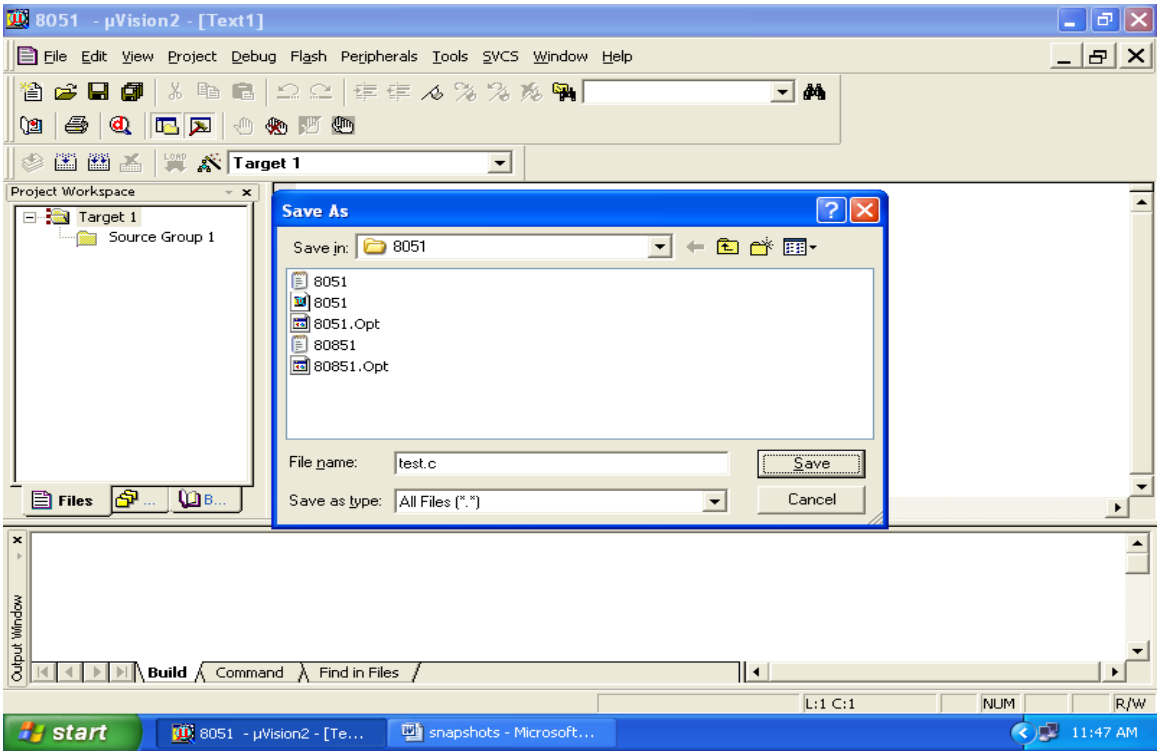
Note: If you want to practice in “VECTOR Lab” save your projects and files in your “server ID”. Otherwise you will loss your data. If any doubts in this concept plesae be contact your concern Lab Co-ordinators”.



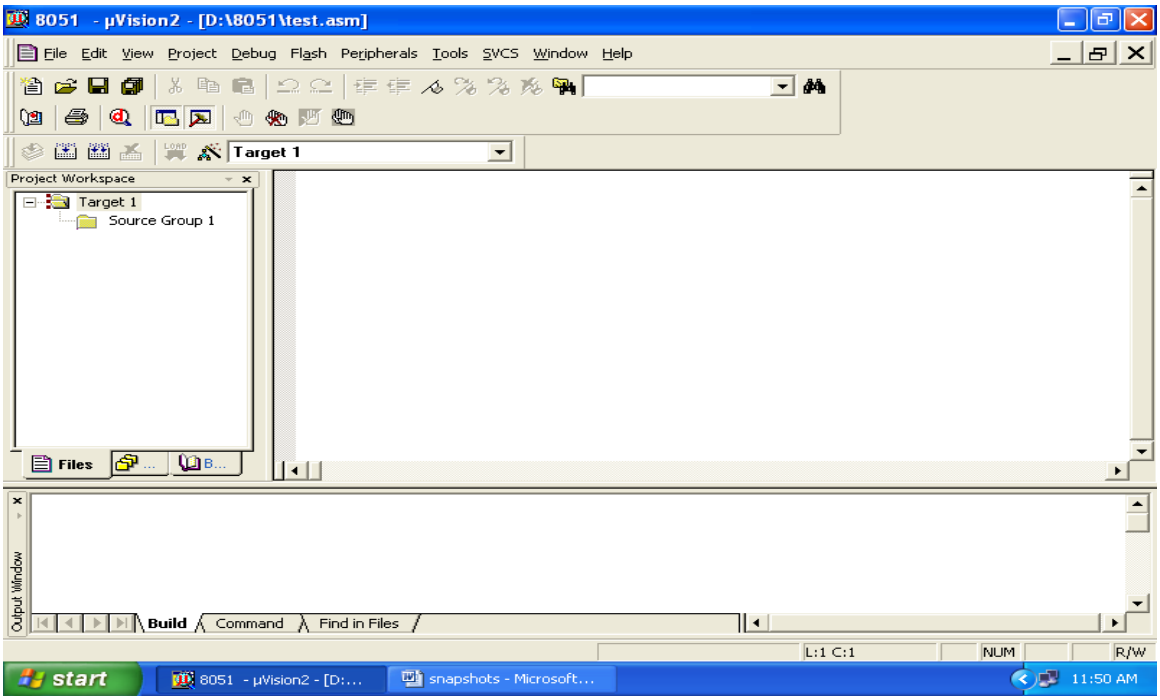
- Select “save as type” as All Files (*.*)



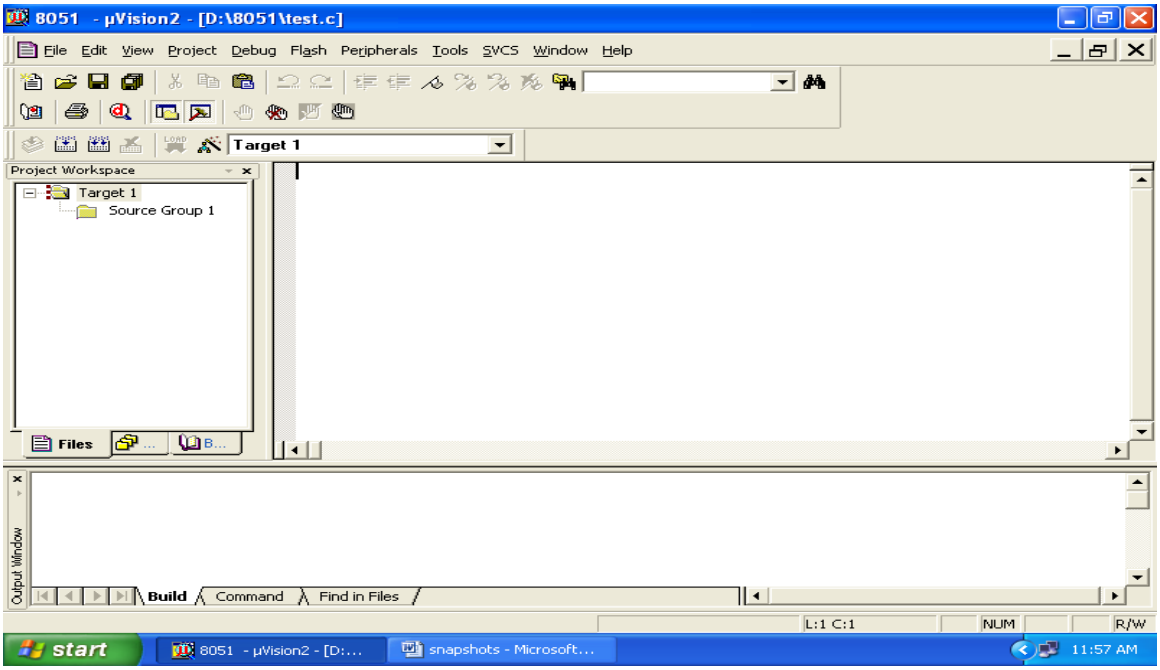
- Click on “Save” button



- Now file also created with name (Observe above menu bar it shows path where we saved and name of the project and file like for example[D:\8051\test.asm]

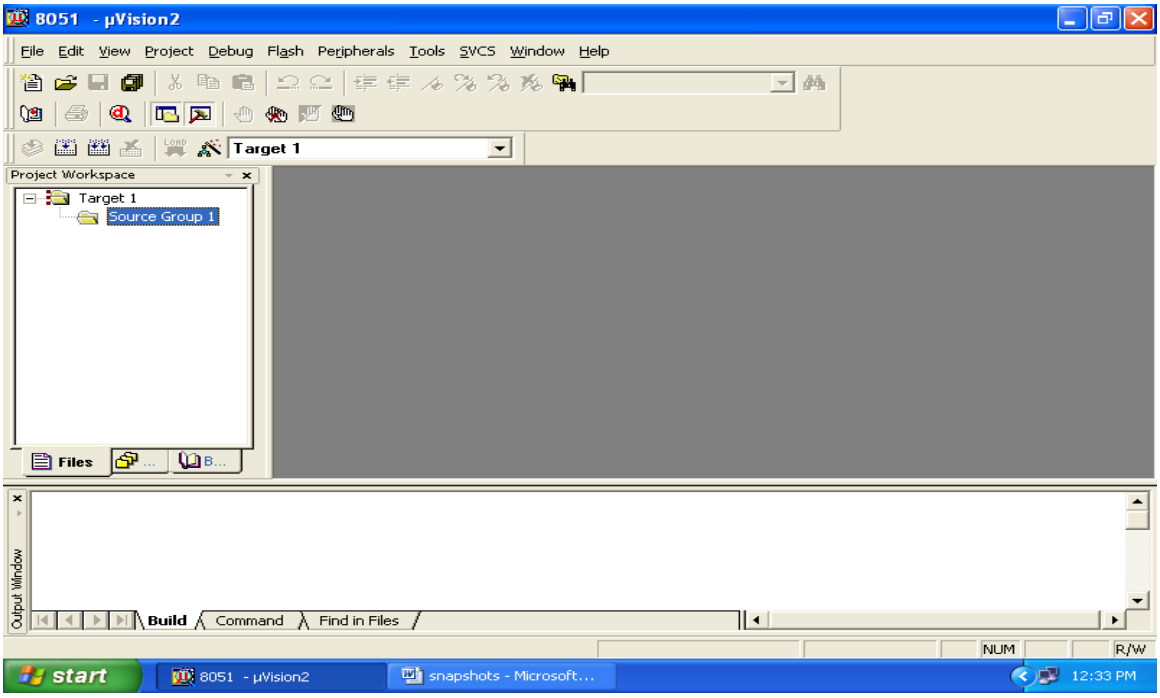


- If it is C source file [D:\8051\test.c]. Selection of path project and file names are based on user requirement it's not fixed. What you selected that one only it shows above menu bar.

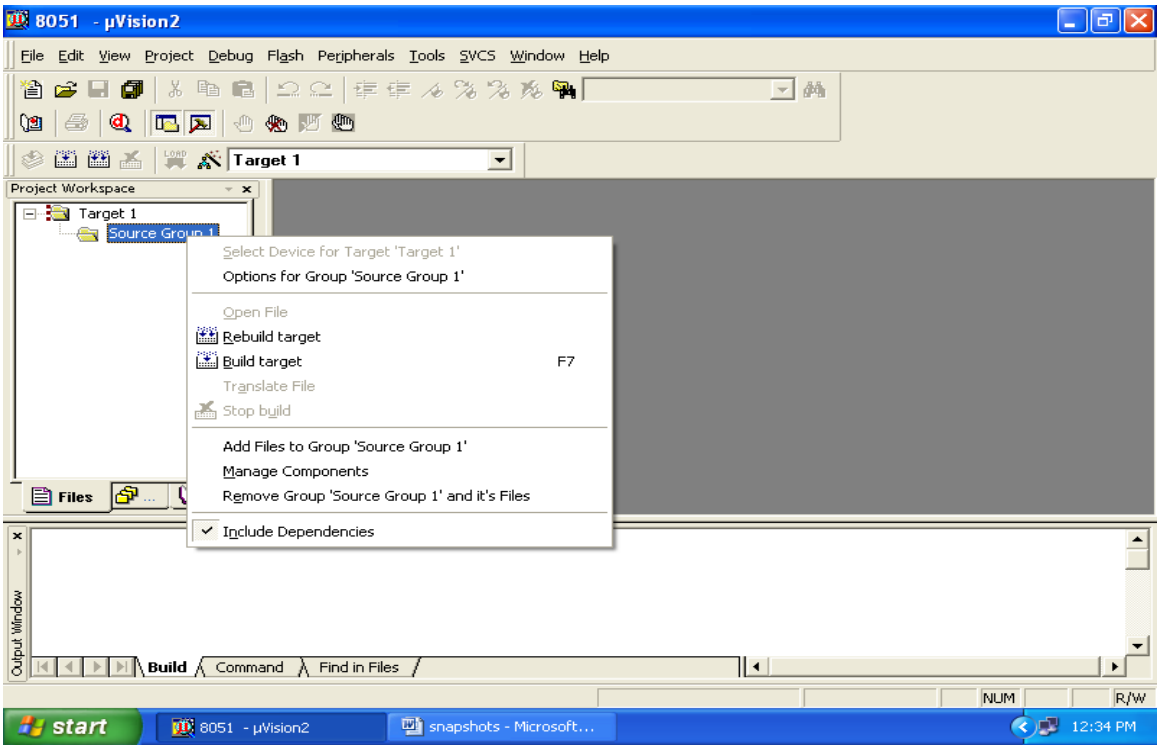


4) Which file you want to compile add that file to source group for that we have to follow these steps

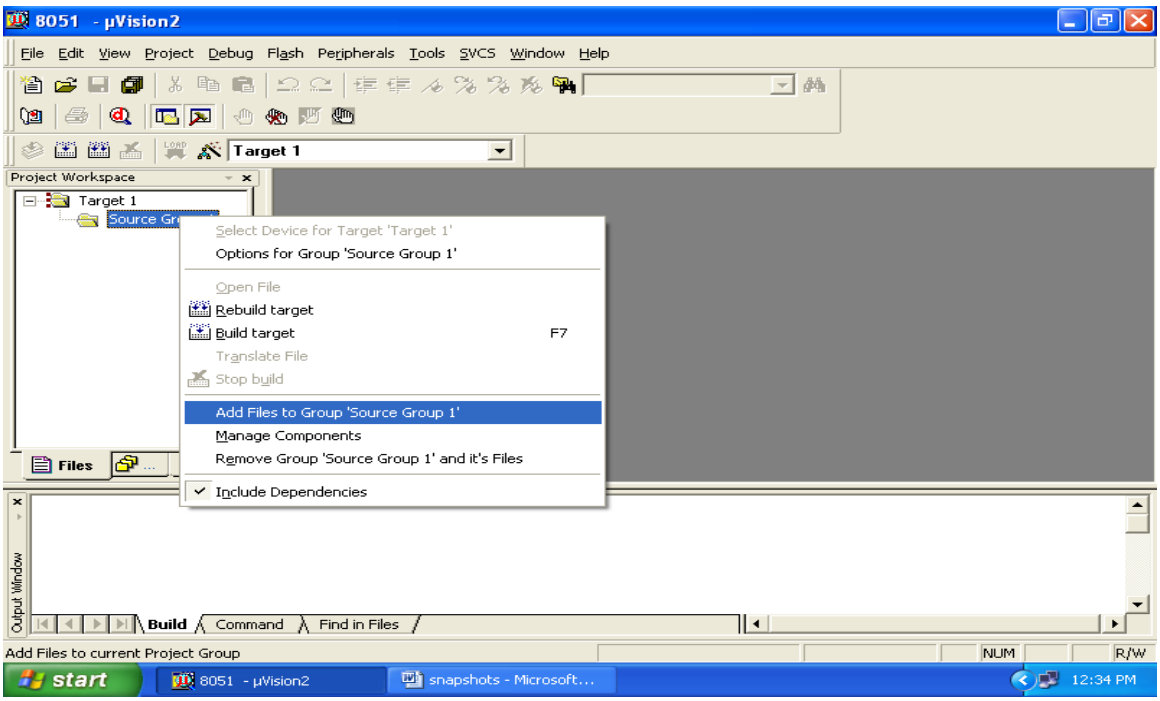
- Select source group on project window



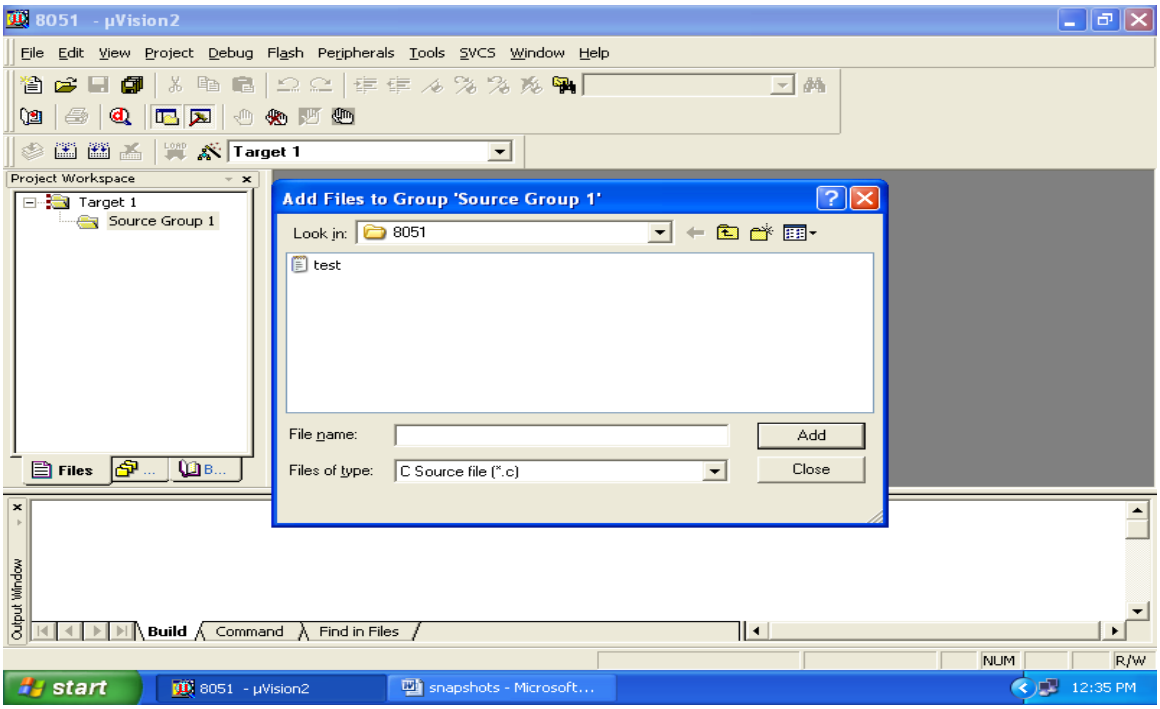
- Right click on that source group in this it's named as "Source Group 1"



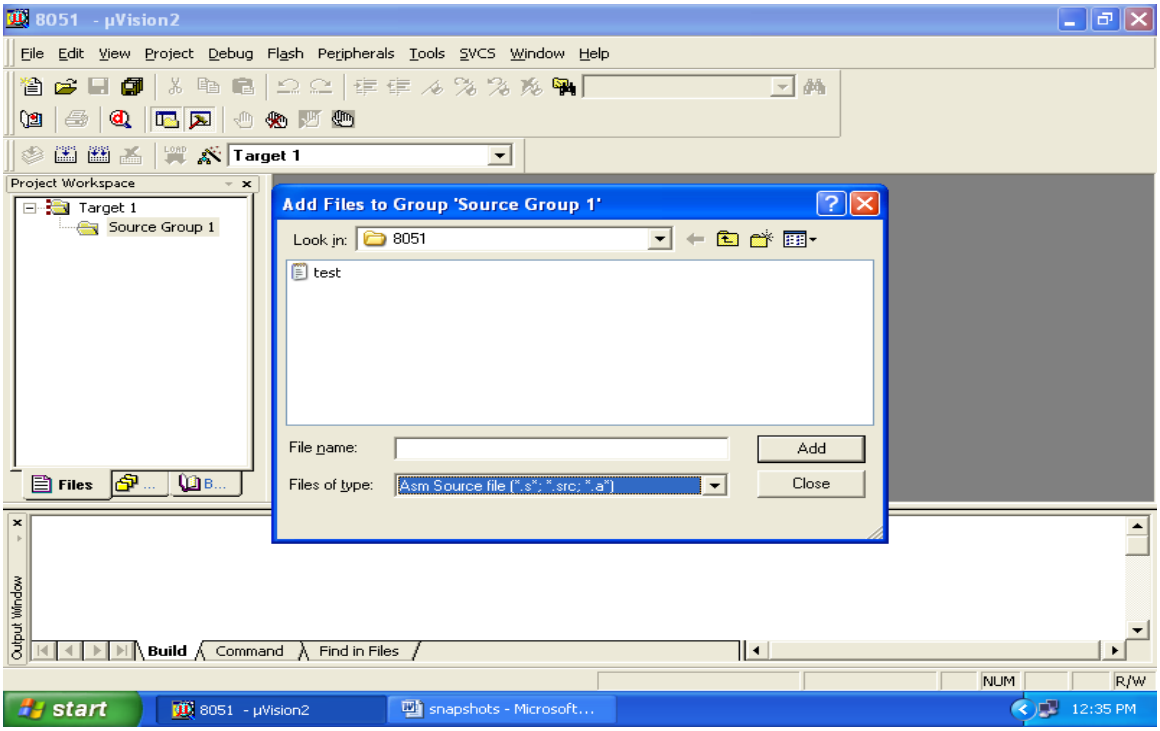
- Select Add Files to Group 'Source Group 1'



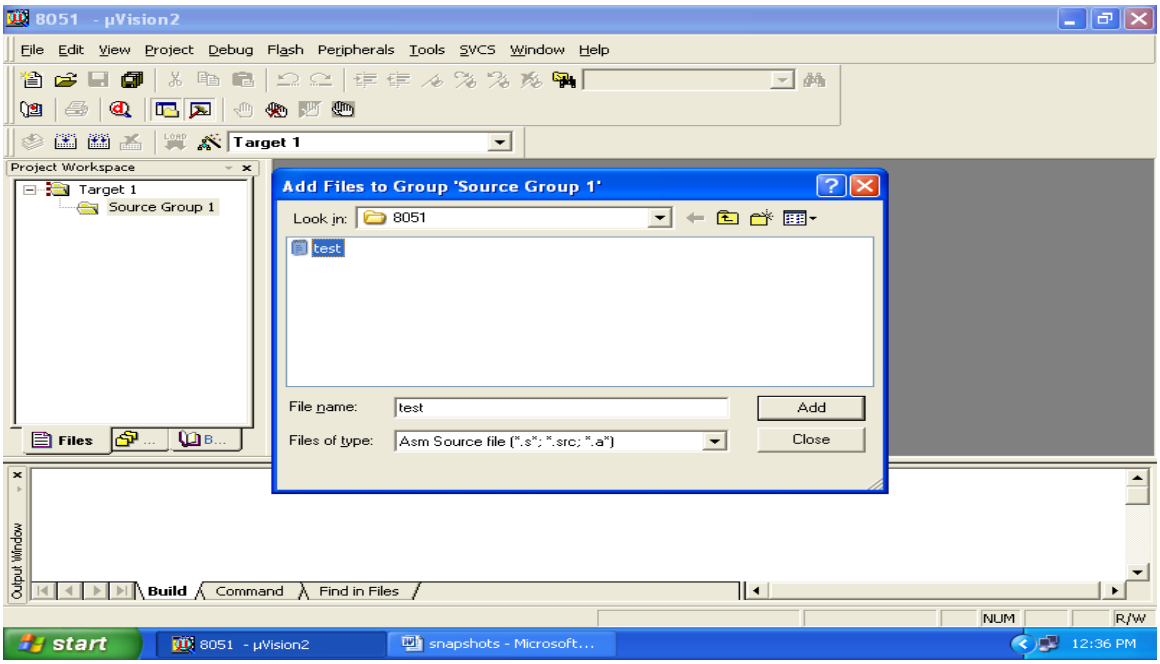
- Now it opens one dialogue box in this it shows which file you want to add and which file you want to compile (by default it shows c source files)



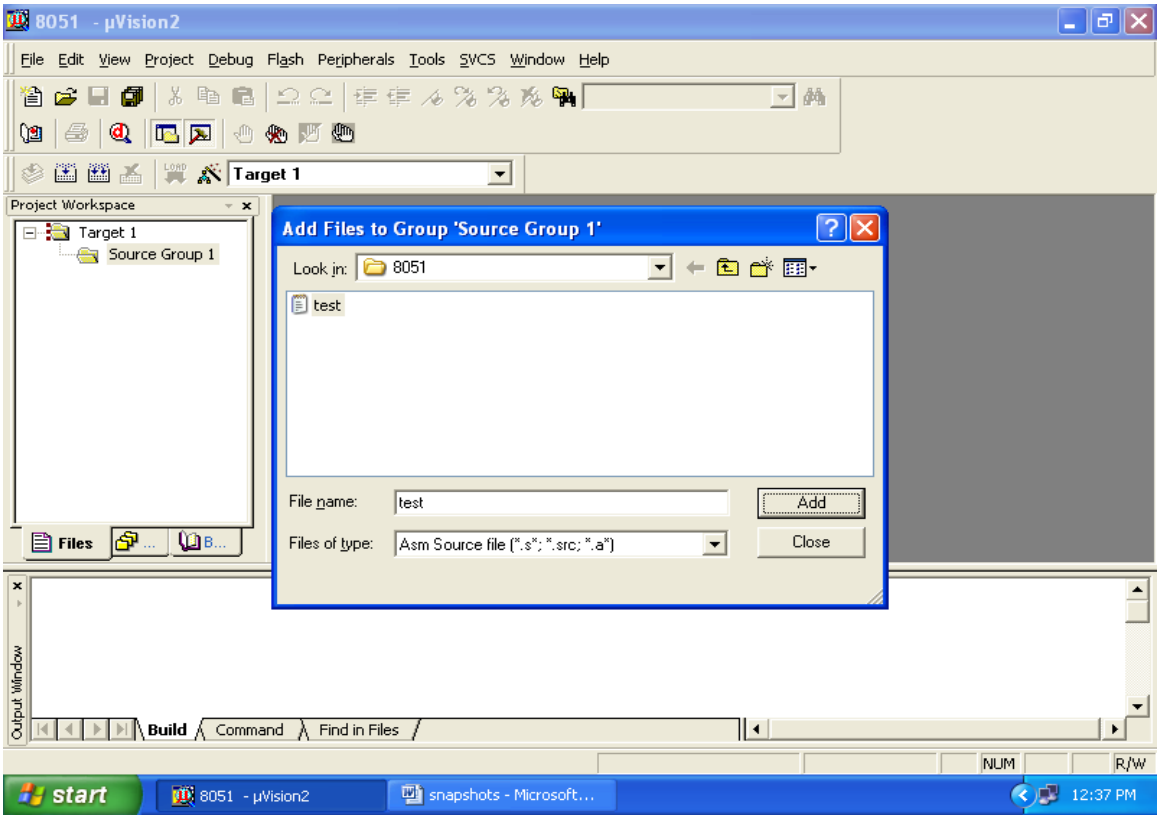
- If you want to compile asm files select file type as "Asm Source File". Then it listed total overall asm files



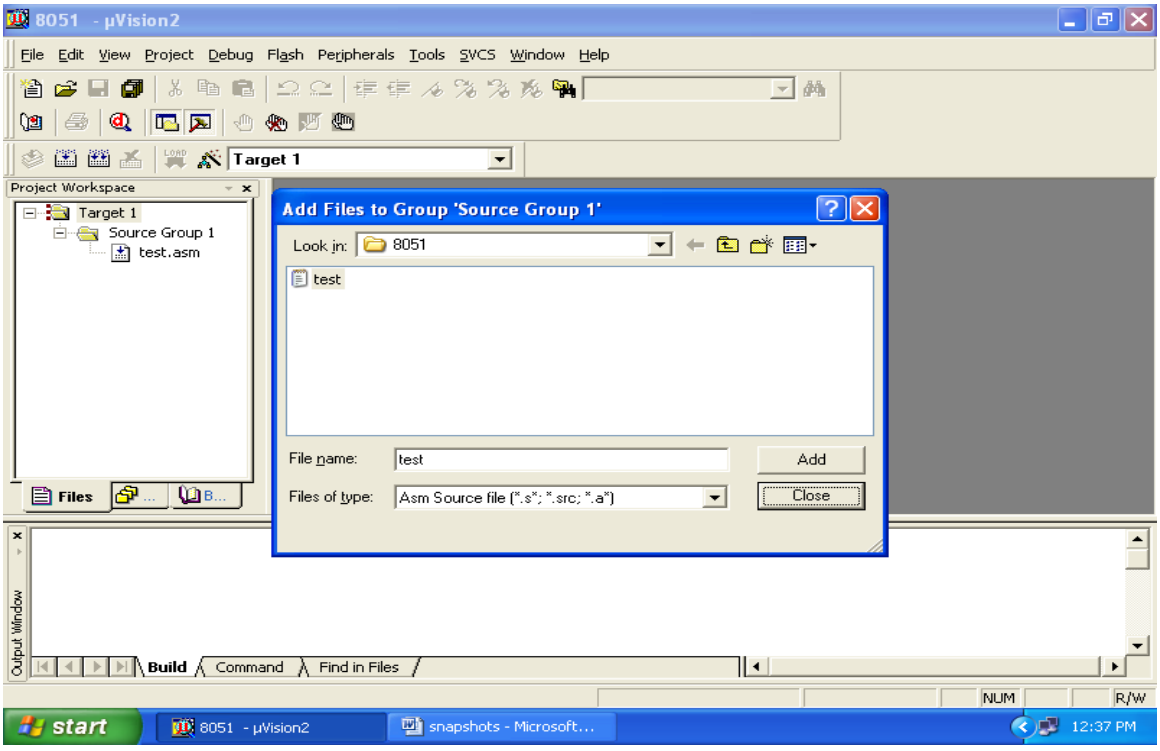
- Now “asm files” are opened then you can select required file name for writing program and for compilation



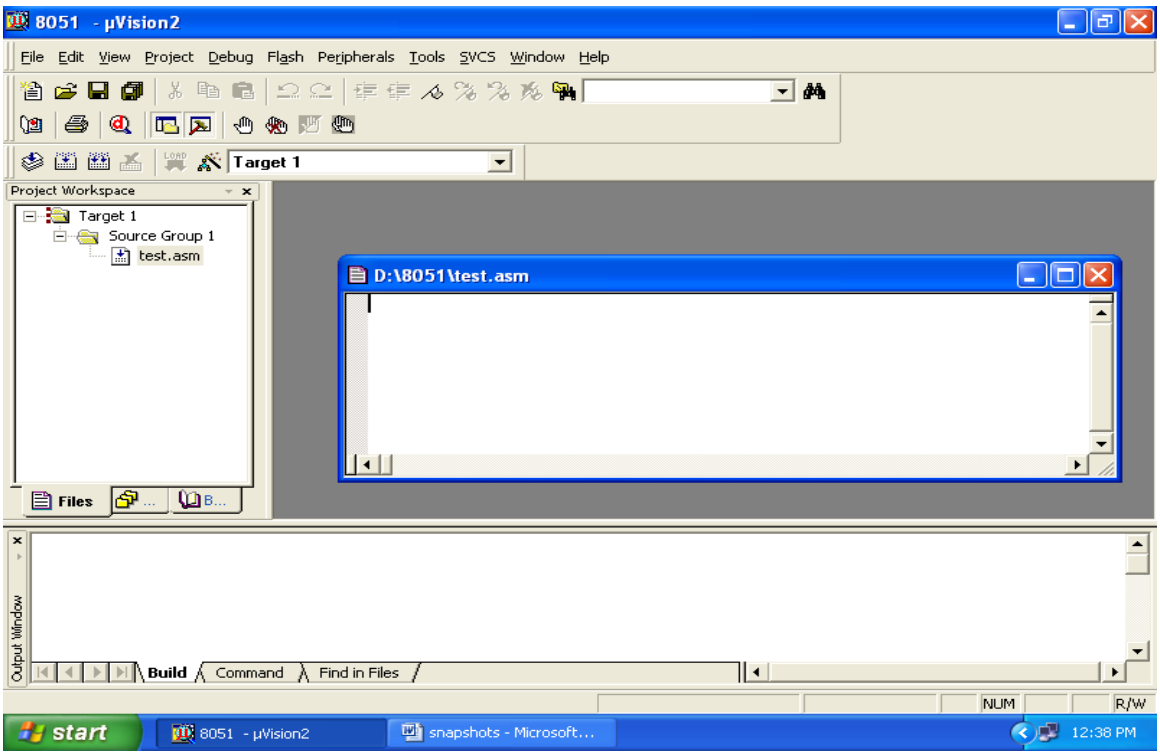
- Click on “Add” button



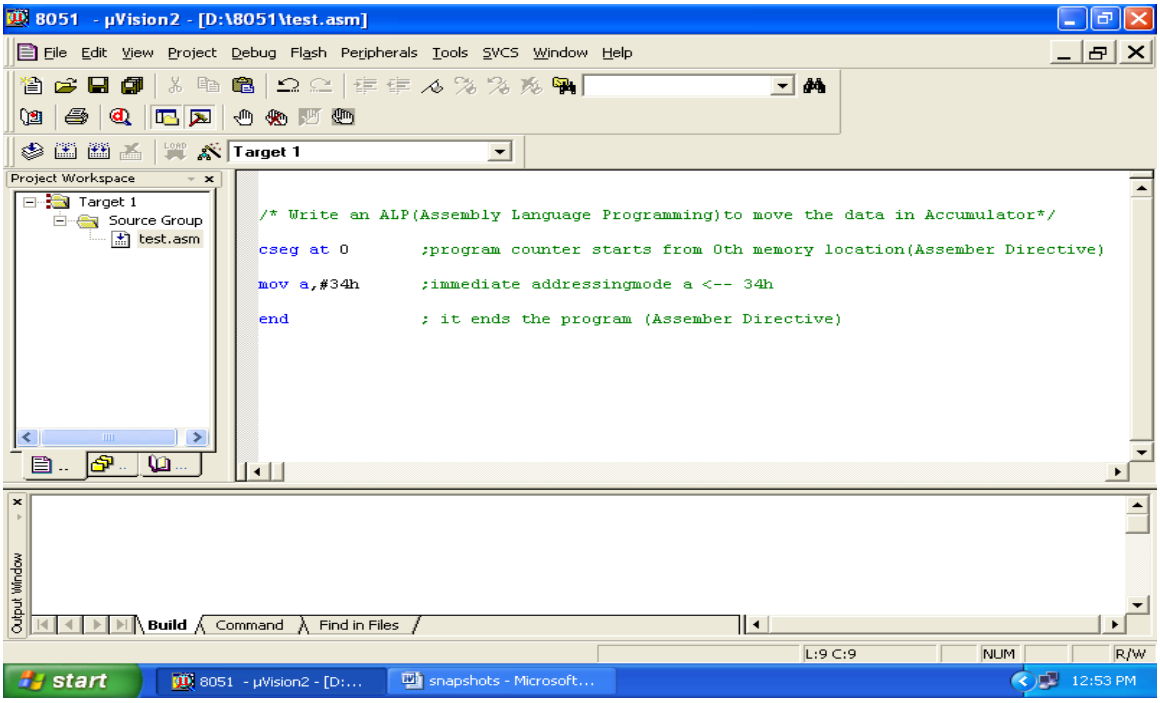
- Next click on “Close” button



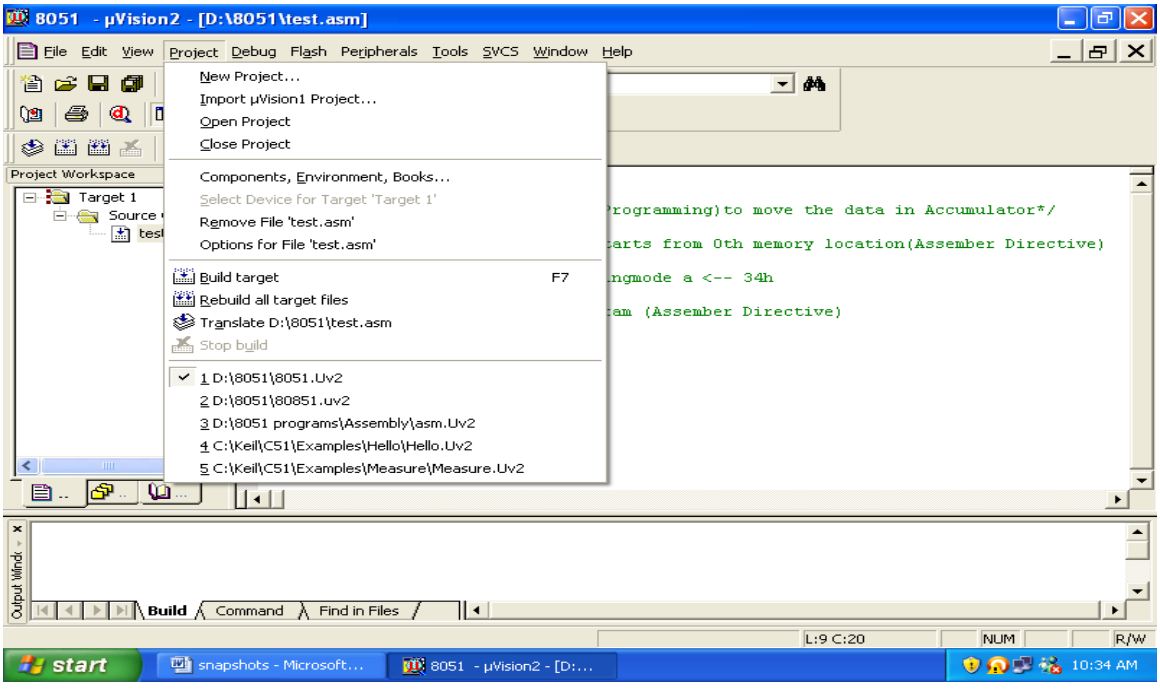
- Now file is added to source group (File name is not visible to you then you can click on left side “+” symbol of source group. Double click on file name.



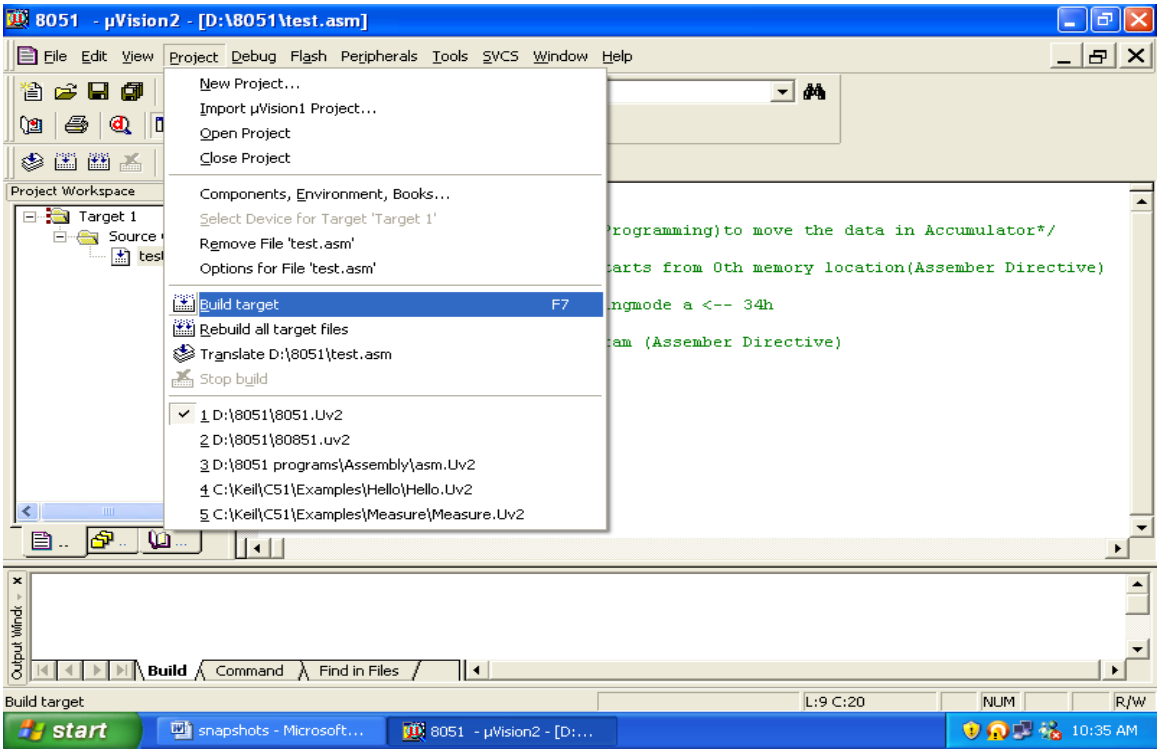
- Now you can write program if program is not available in that file
  - Before writing the program first of all you have to know about Addressing modes, instruction sets and Assembler directives. So, first you can read after you can write program



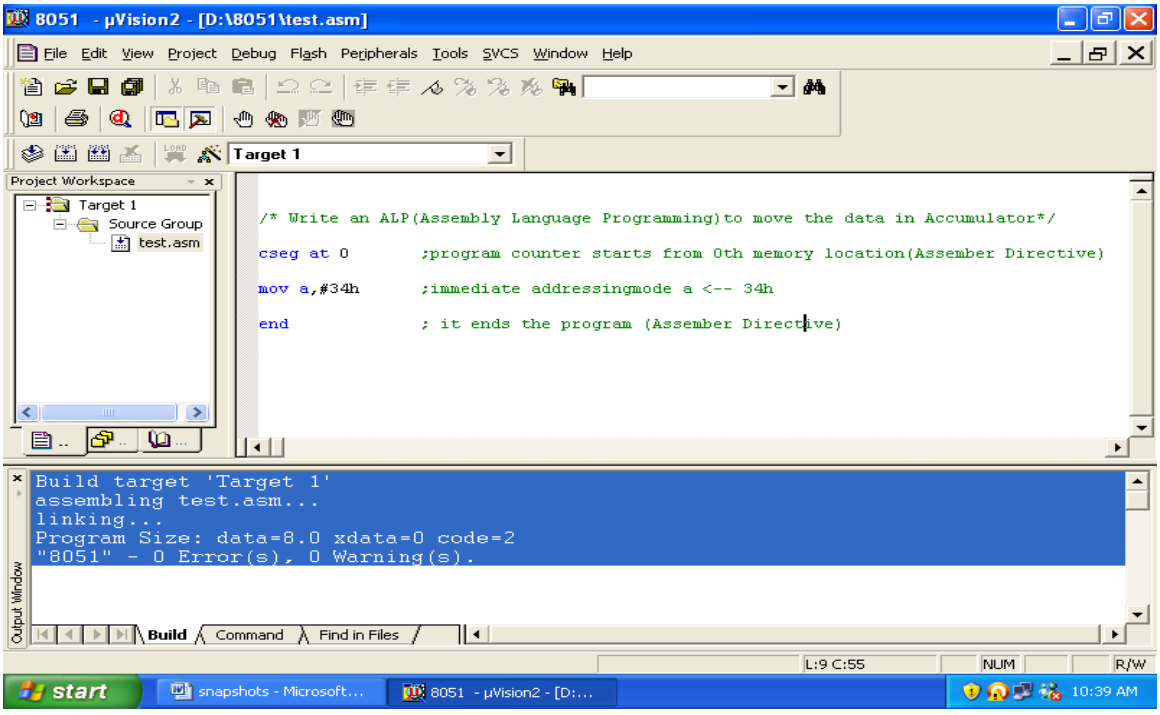
- Every time you can save that file with the short cut key (CTRL+S)
- After you can compile for that click on "project" on menu bar



- Next click on Build Target (or) press F7 short cut key

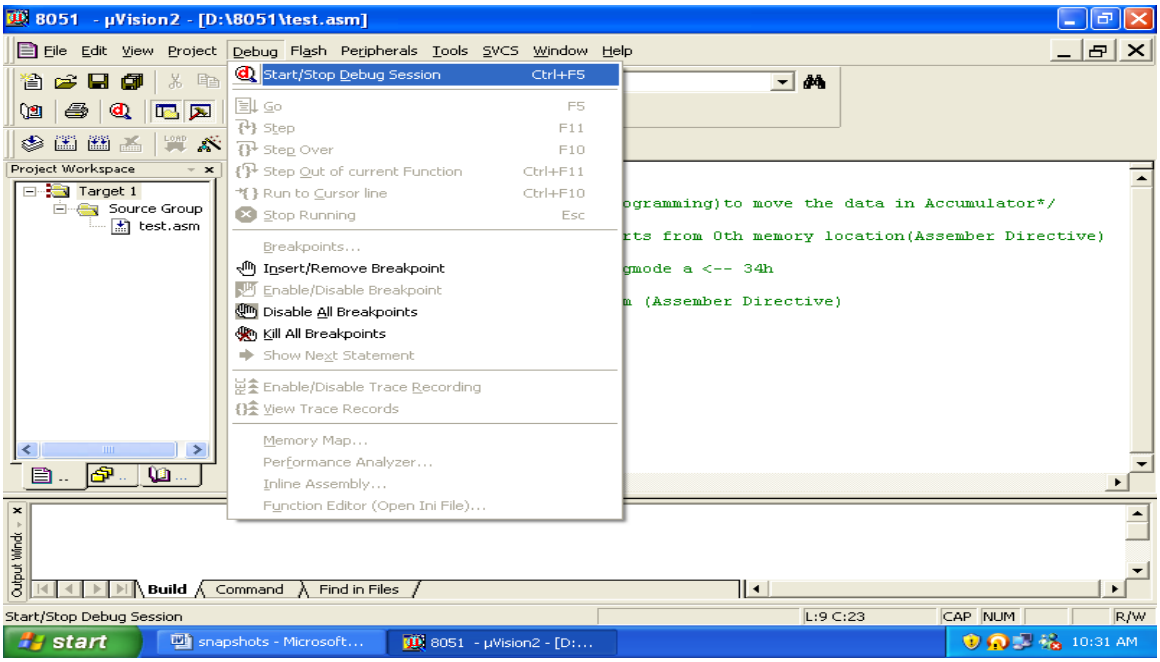


- Now check the output on output window. In that it shows errors, warnings and how much size your program (blue color marking is output window)

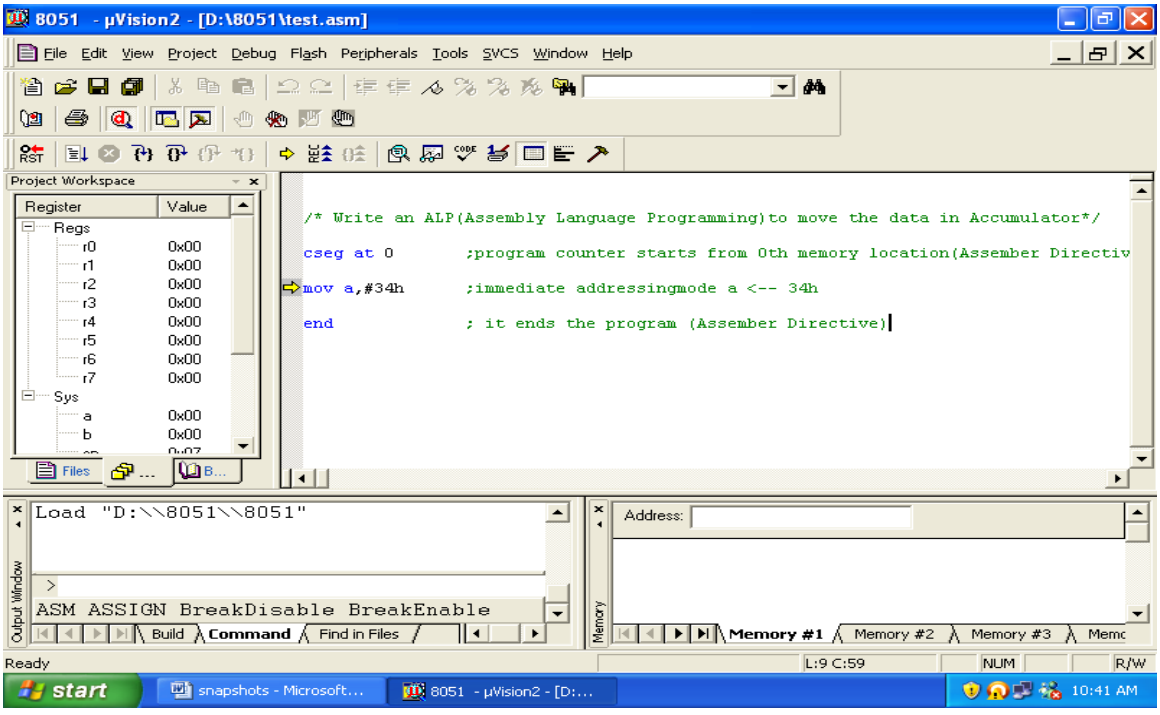


5) Next if you want to see the output on this keil you have to know debugging process for that follows these steps

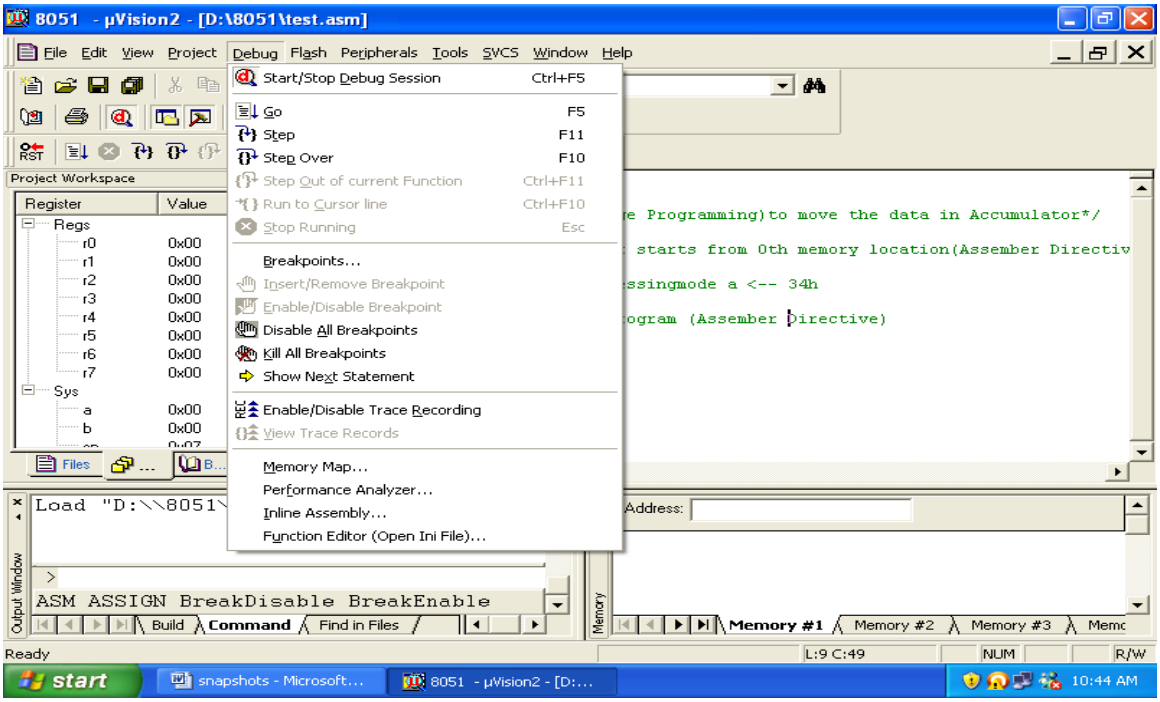
- Click on “debugging” symbol or short cut key (CTRL+F5)
- Or click on “Debug” on menu bar and click on start/stop Debug Session



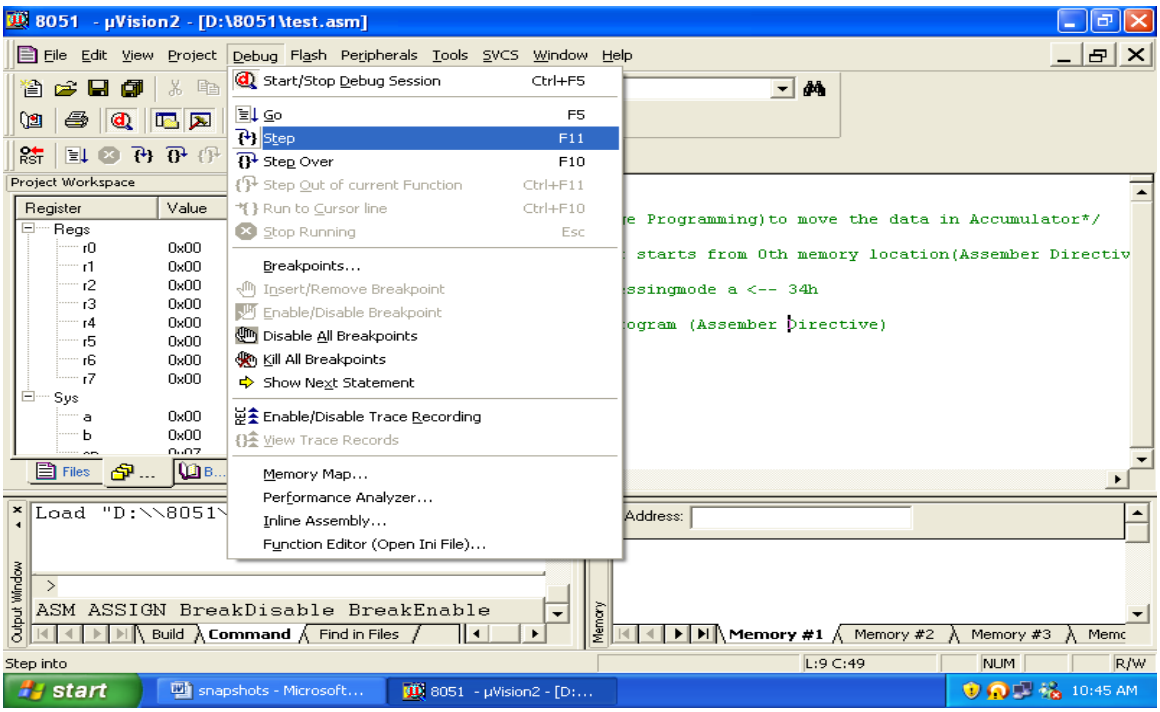
- then it opens on window



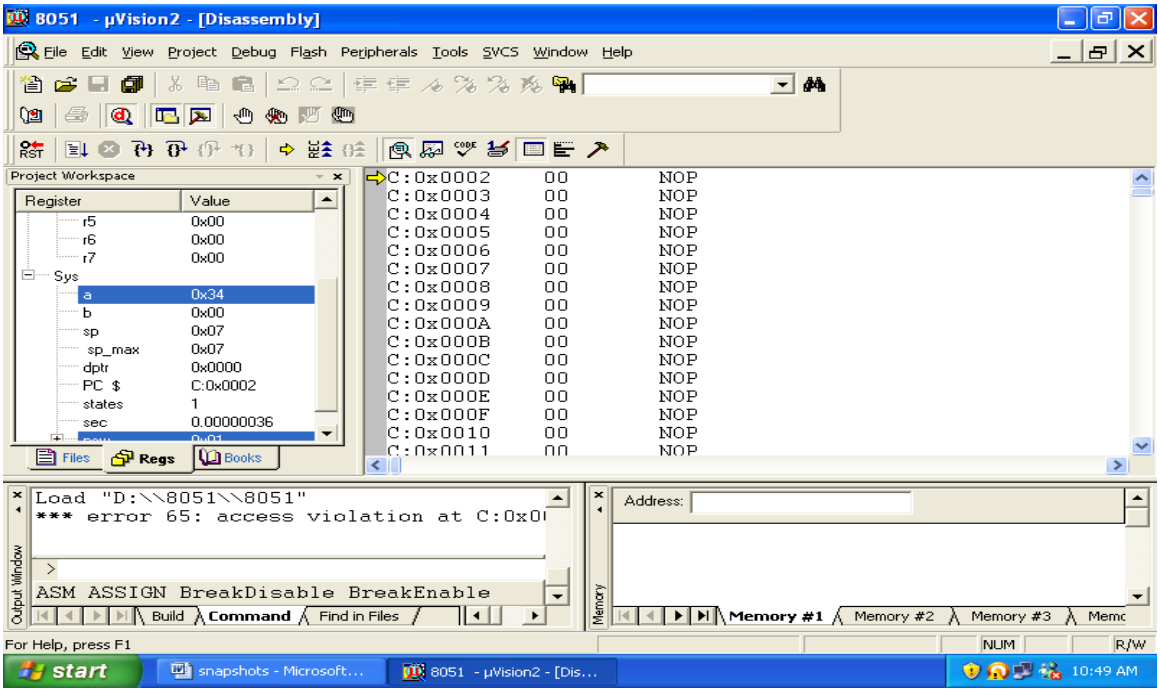
- Now you can do step by step execution for that click on “Debug” on menu bar



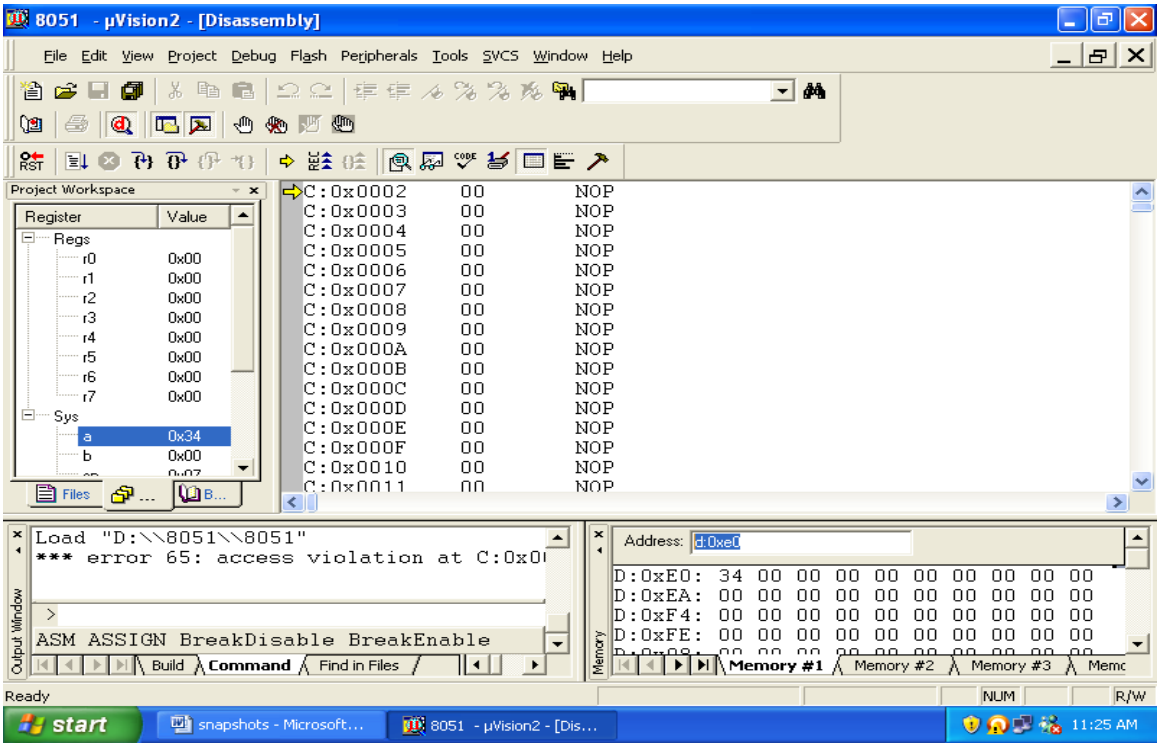
- Click on “Step” short cut key (F11) (or) same symbol is available in debug tool bar



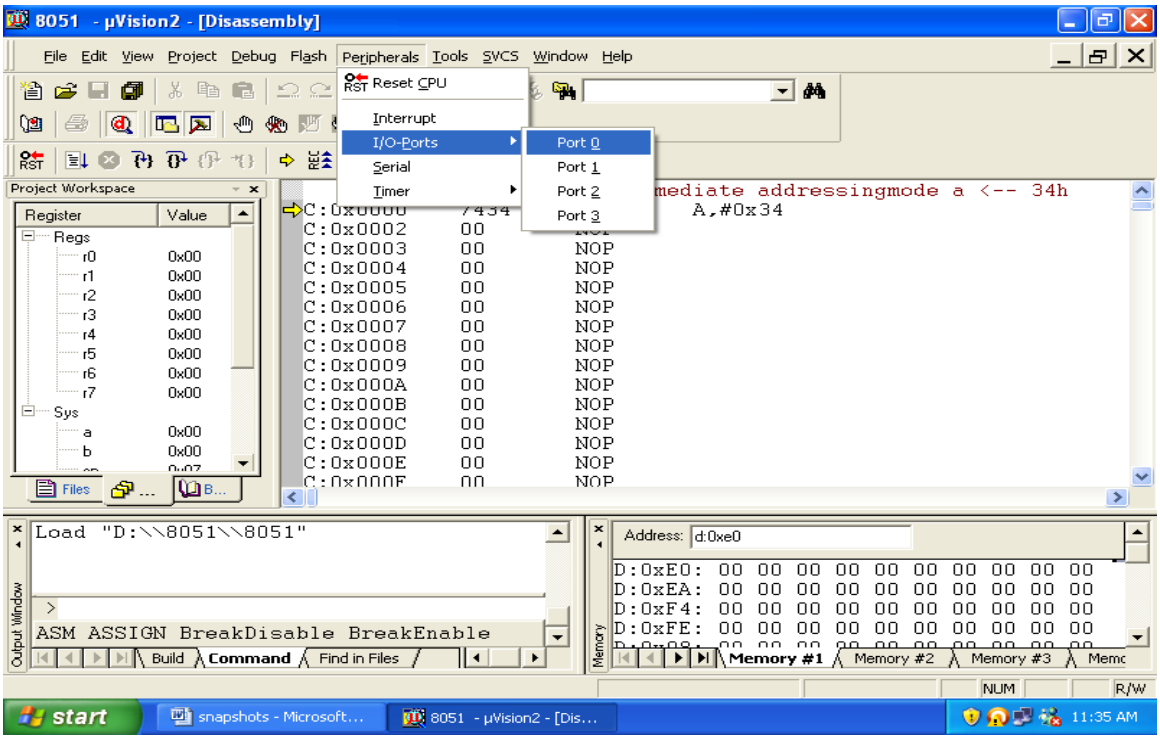
- Output shows on register window observe blue color marking



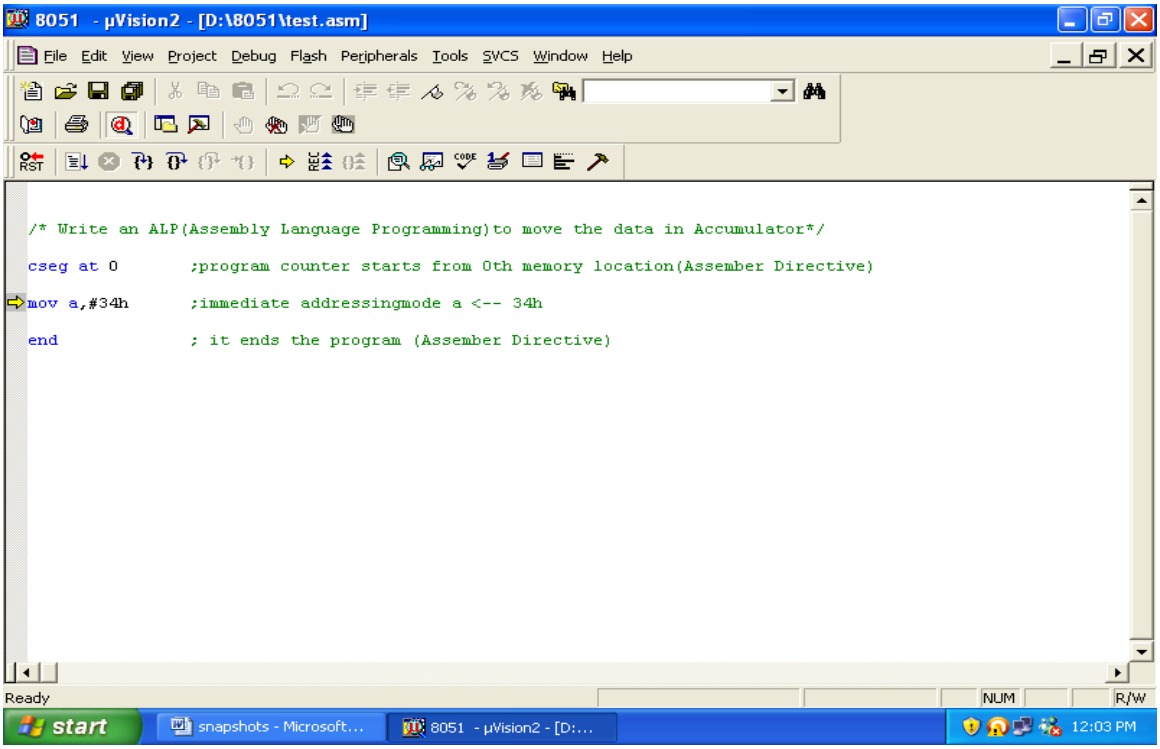
- Another way to see the output your program output is a← 34h that one you can see on memory window example "A" register is internal RAM. So, you can type D: address on memory window. Now "A" register address is 0xE0 (or) 0E0H. D:0E0h



- If you want to see the output on ports then you can select peripherals and click on desired port.



- With out all windows it visible likes this.



- If you want to see project window to see output on general purpose registers, A and B and some SFRs, another one is memory window to see data on particular memory, another one is Disassembly window to see the program in opcodes format, another one is watch & call stack window to see user defined variables output those all are available in “View” on menu bar.

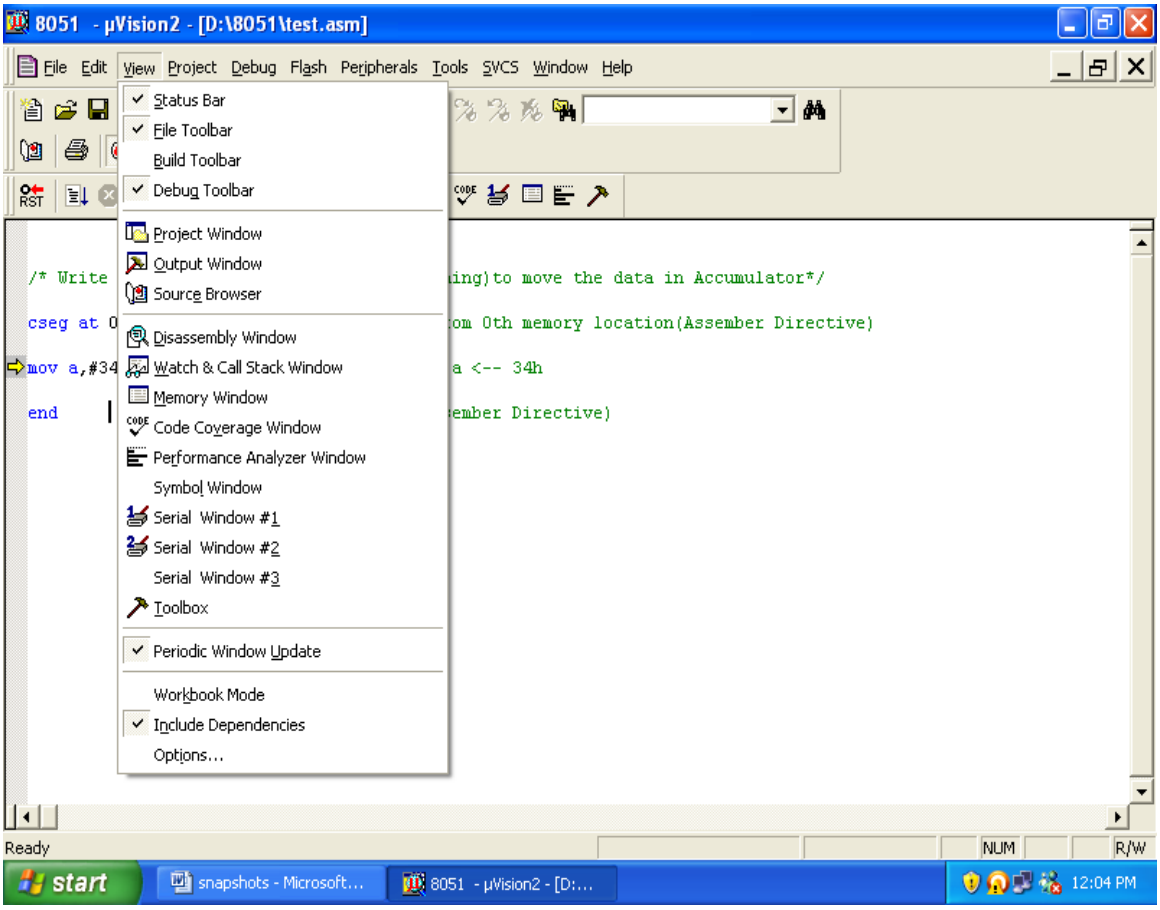
- Which one wants you can click on that window it’s opened if you don’t want again you can click on same “View” button on menu bar and which window you don’t want click on that it disabled.

- On memory window if you want to see the data on code memory type C: address, and for external memory type X: address and for internal data type D: address

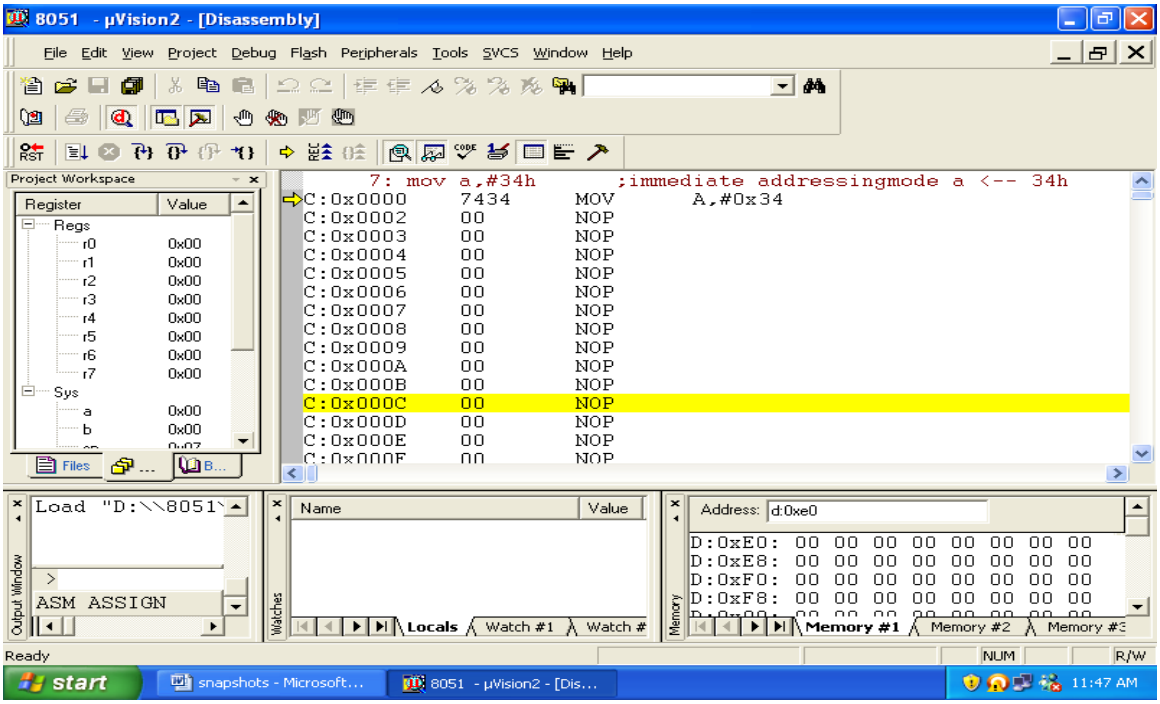
- If you want to come out from the debugging process again click on “Debug” on menu bar and click on start/stop debugging (or) press CTRL+F5 (or) click on “d” symbol on debug tool bar.

- You can see these all windows at a time in below.

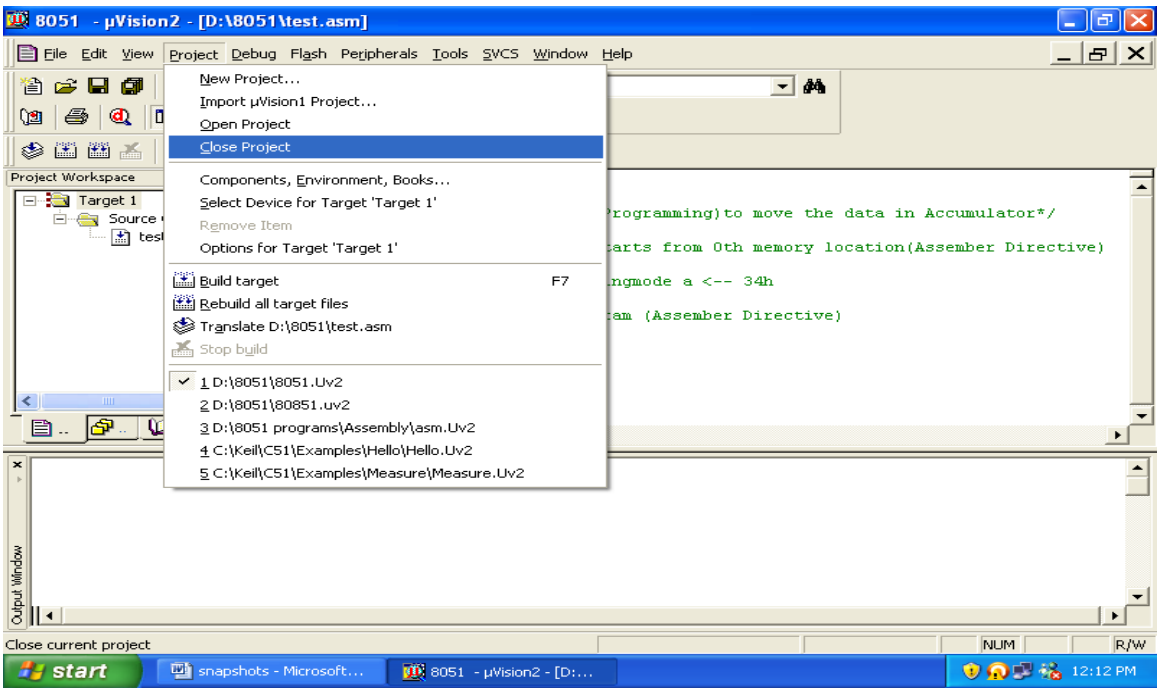
- Now you can follow above process to open all windows



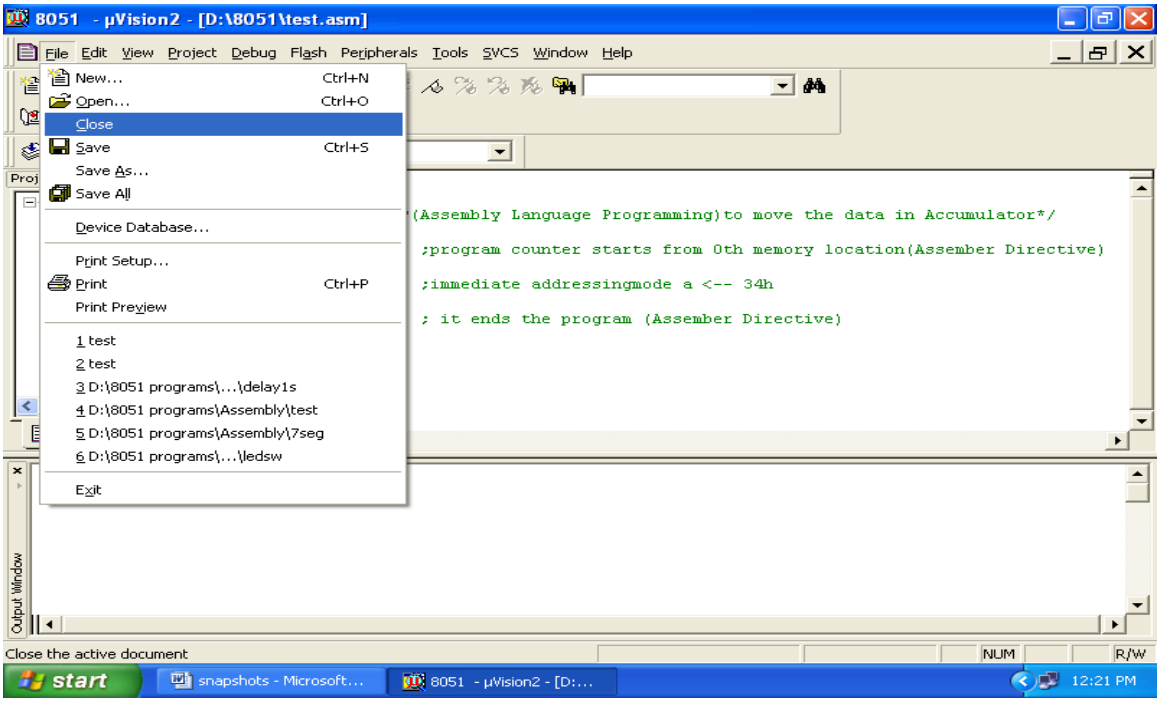
- Now you can see all windows which are explained in previously.



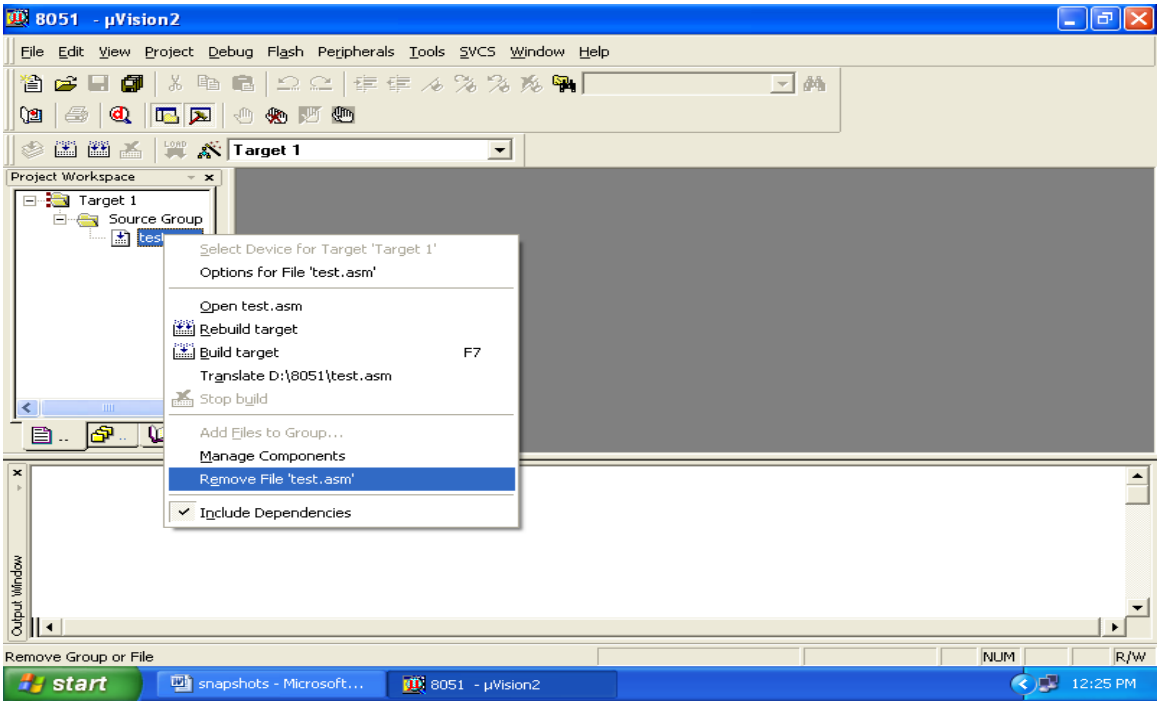
- If you want to create another project follow same process how to create project before that one first close previous project.
- First come out from the debugging and after click on “Project” on Menu bar after you can click on “close project” option.



- If you want to create new file follow same process “How to create file” we discussed previously. Before that one close the file
- Click on “File” on menu bar and click on “Close” option.



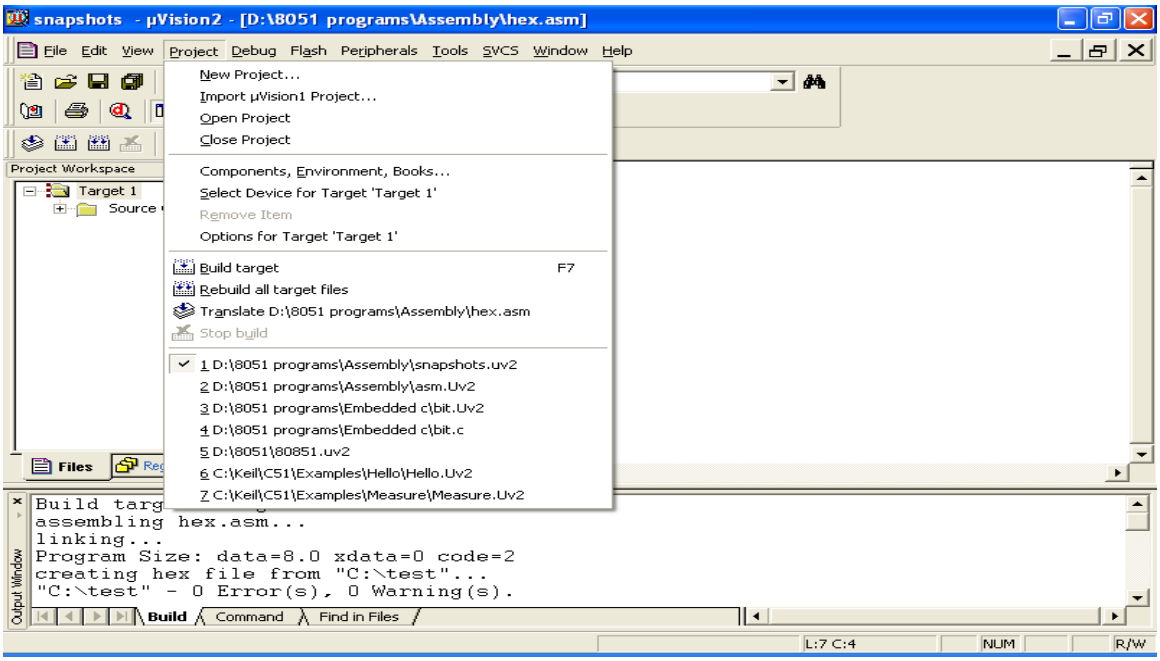
- Next remove file from source group on project window for that which file you want to remove on that you can give right click after click on “remove file”.



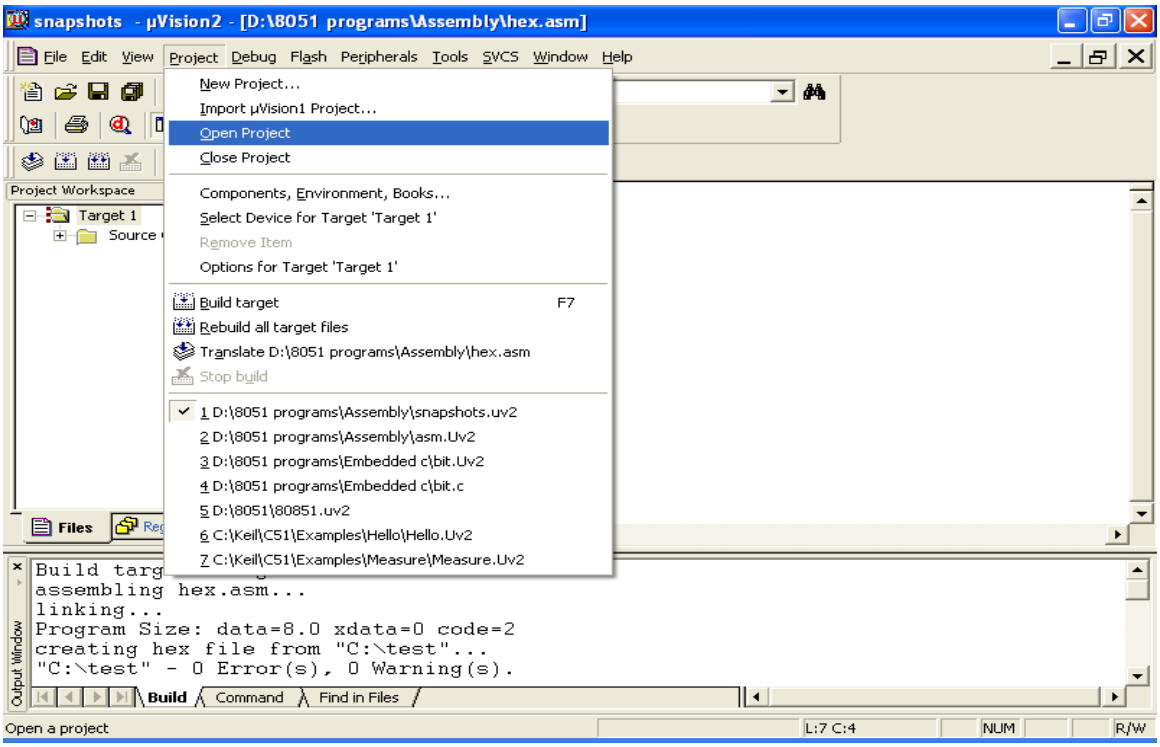
Again if you want to create and save the project and file we have to follow same above process.

6) How to open Project

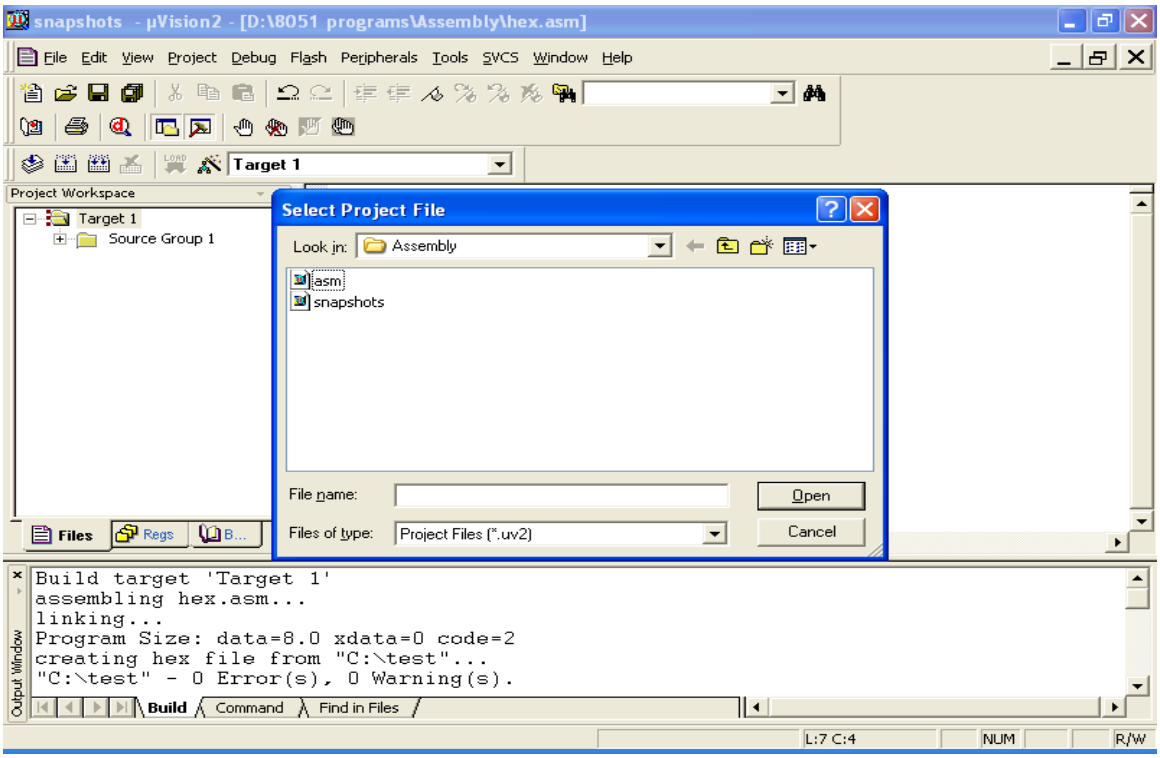
- Click on Project on menu bar



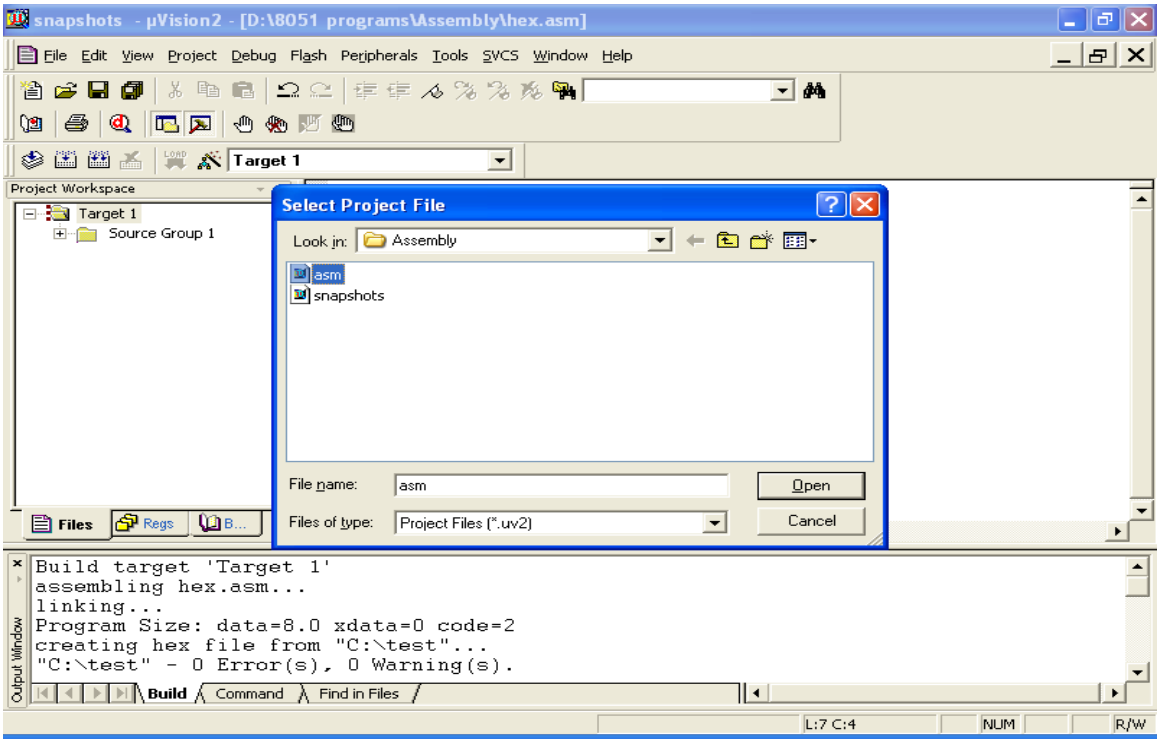
- Click on "Open Project" option



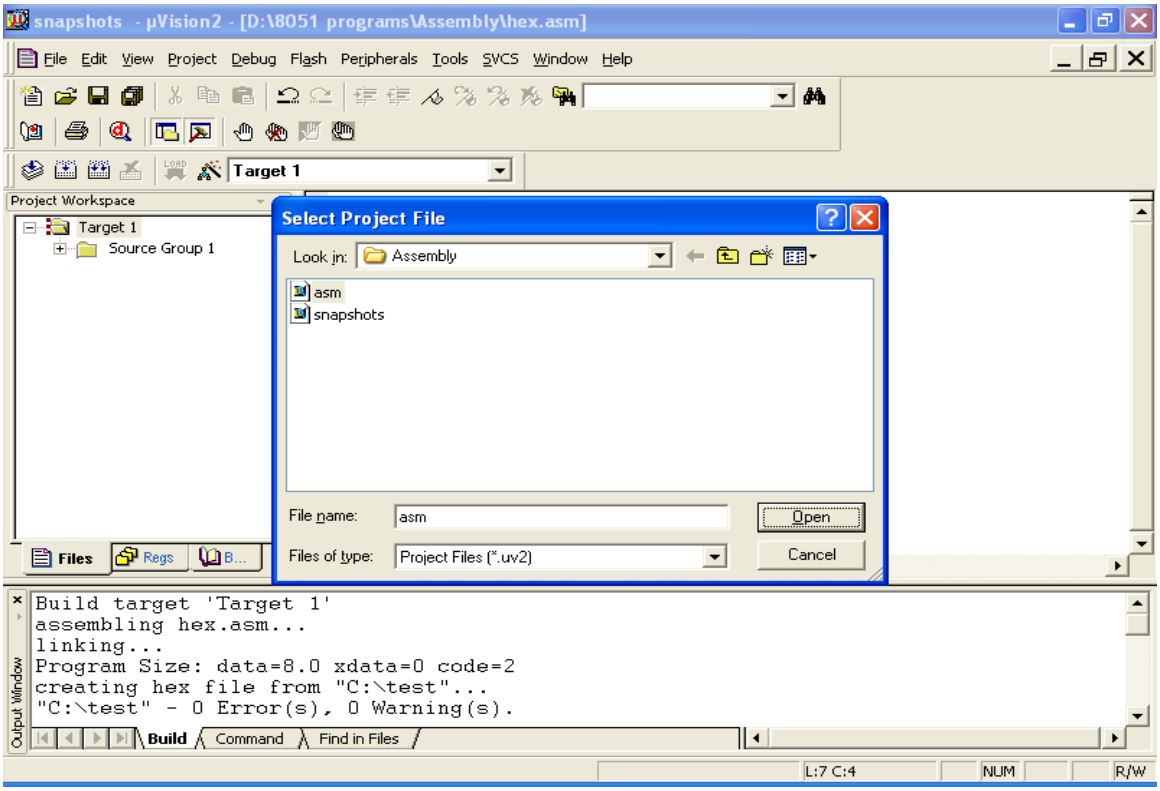
- It opens one dialogue box like this



- Select your project name which one you want to open

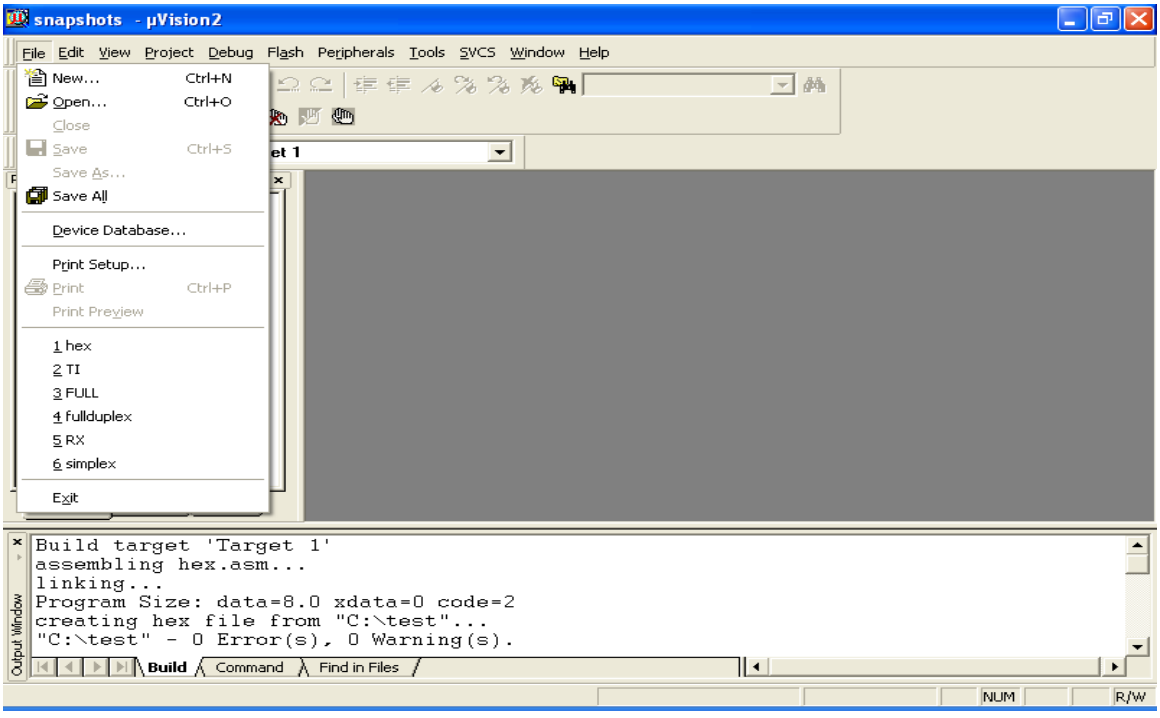


- Click on “open”. It opens that project.

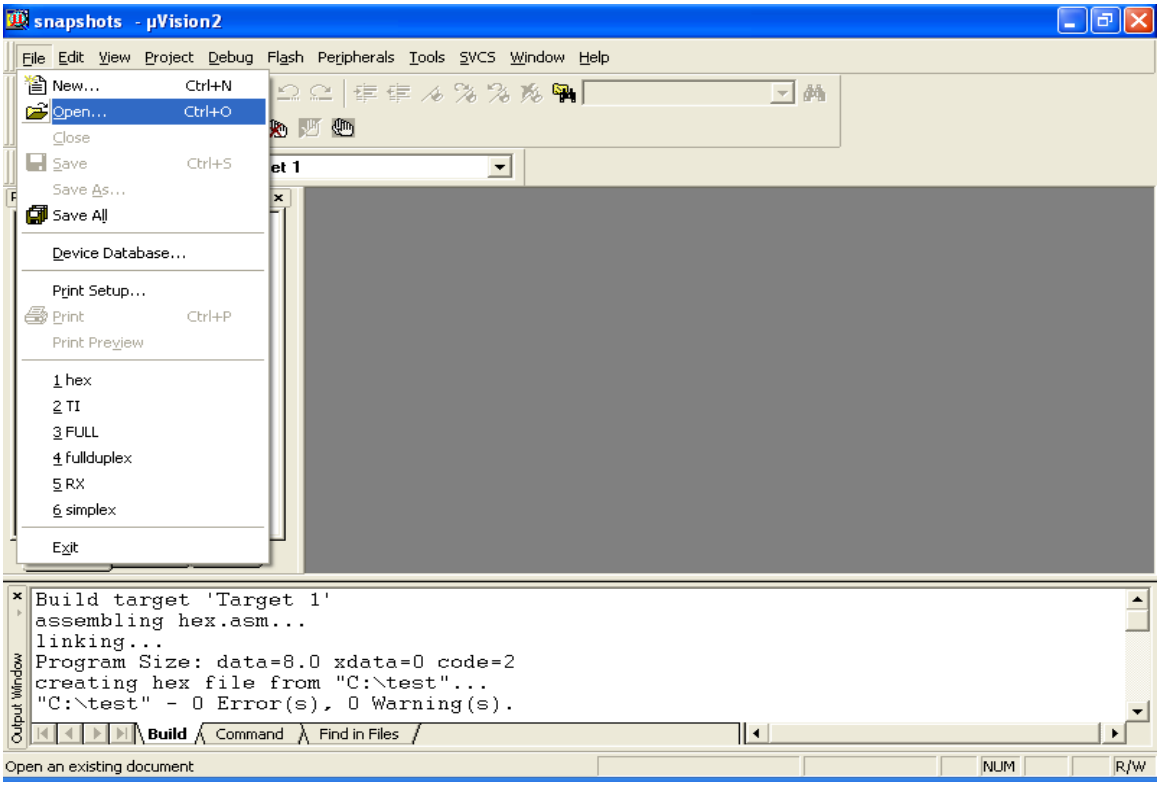


7) How to open file

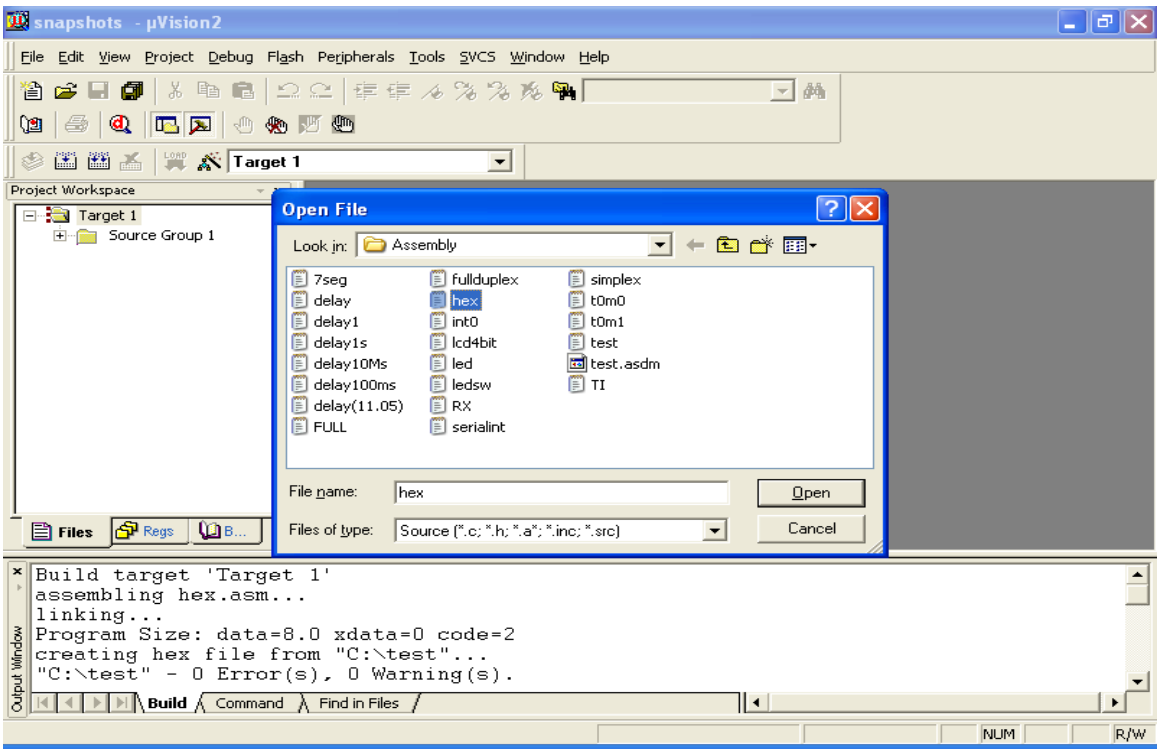
- First click on File on menu bar



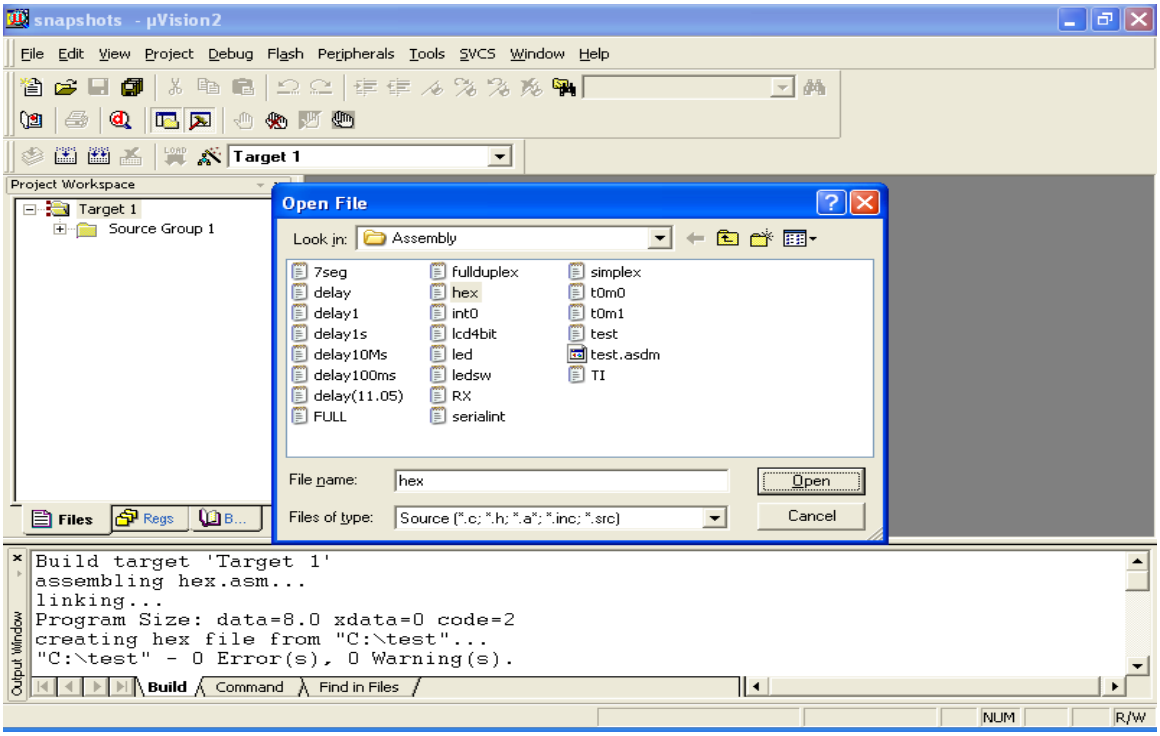
- Click on “open”



- It opens one dialogue box in that select path and your filename

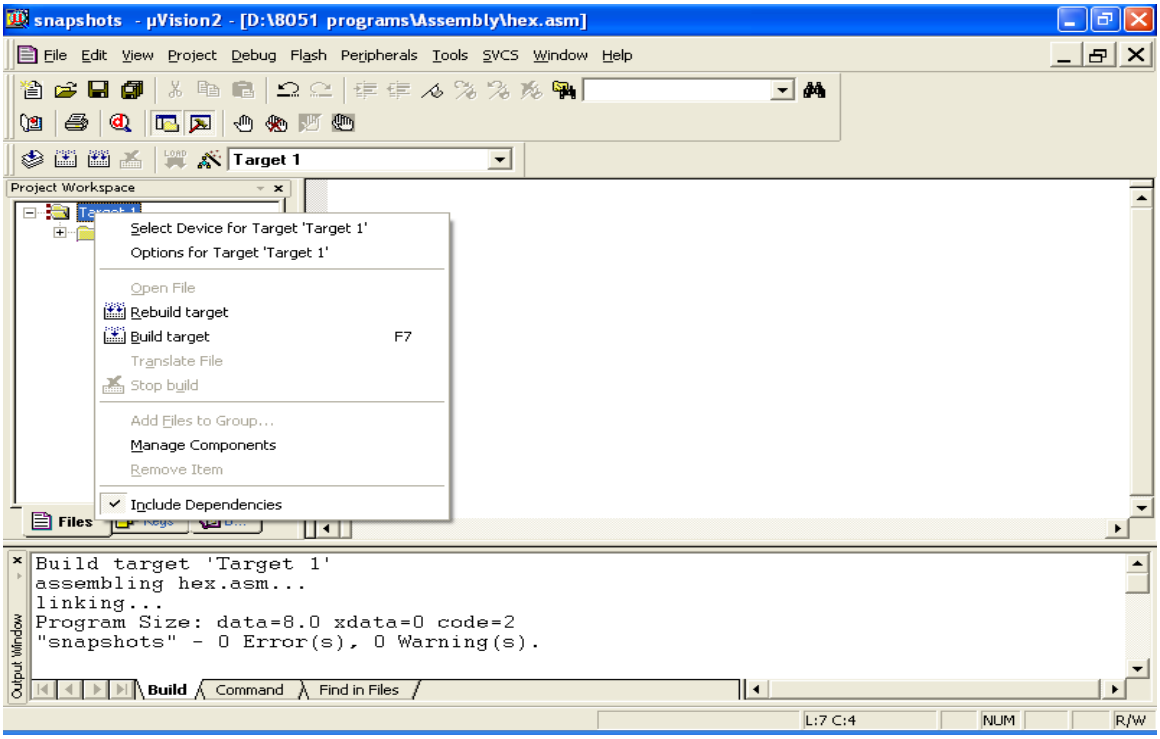


- Click on open option then it opens the file.

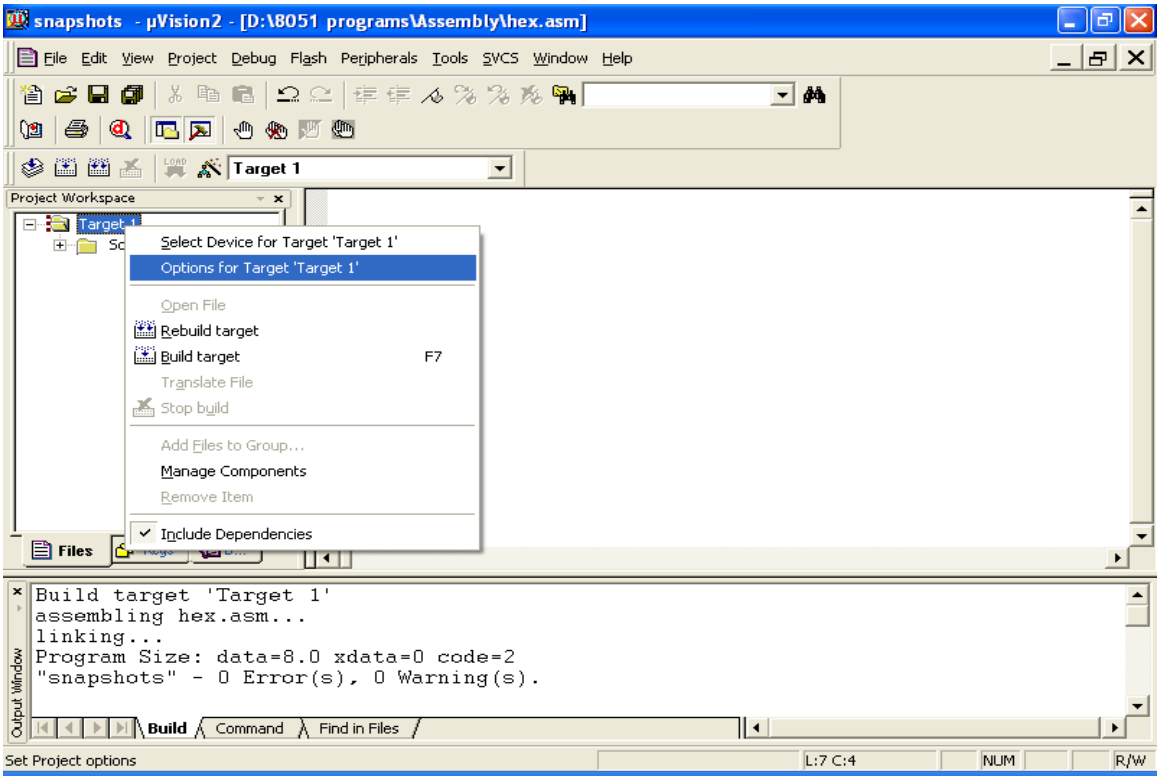


8) How to Create Hex File

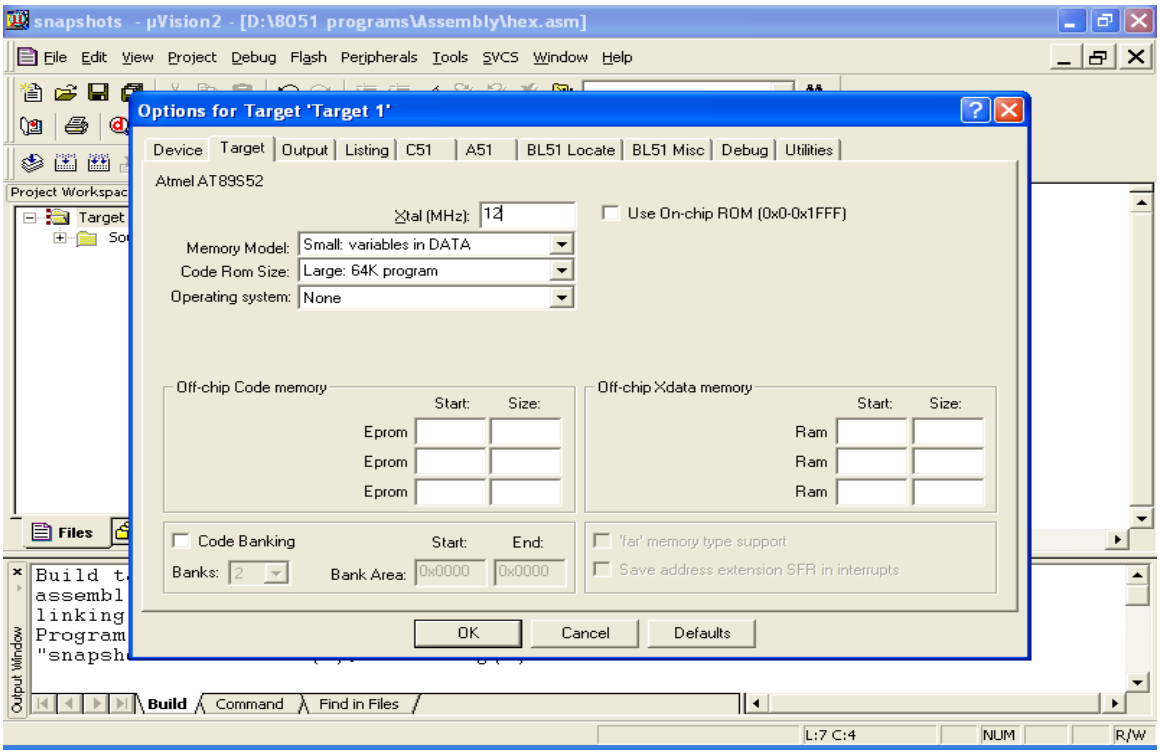
- First select target after right click on target



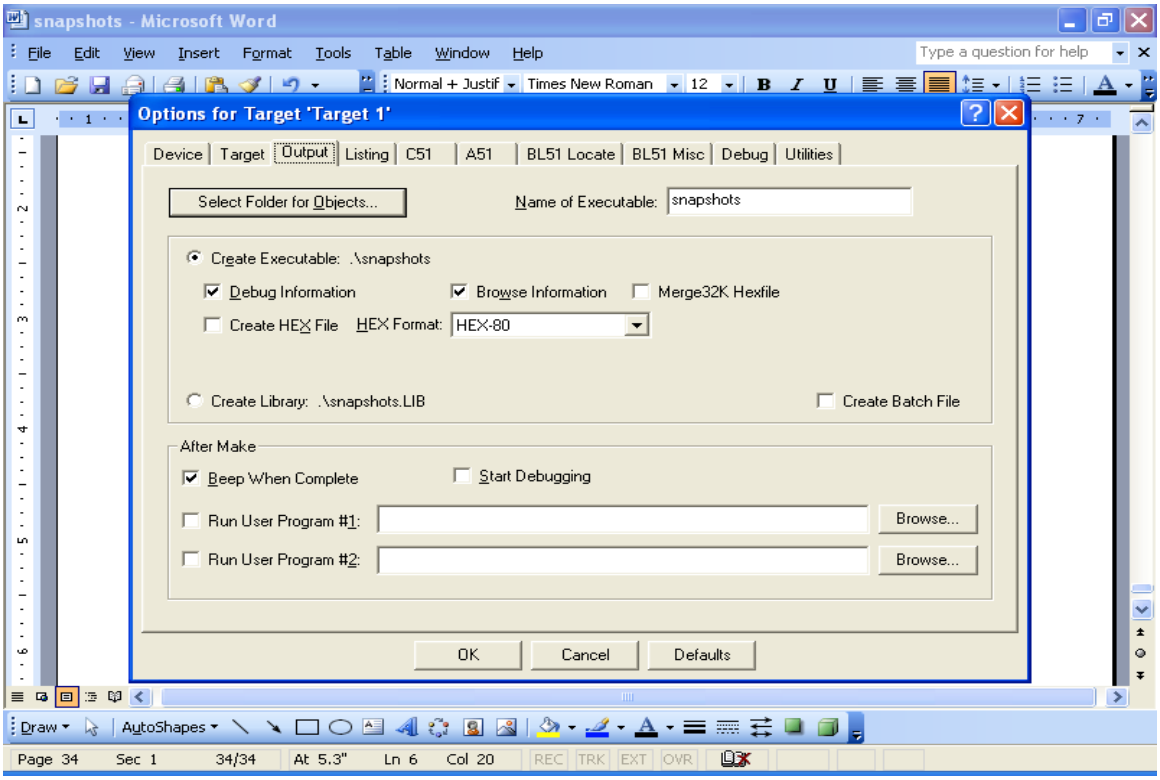
- Click on “Options for Target”



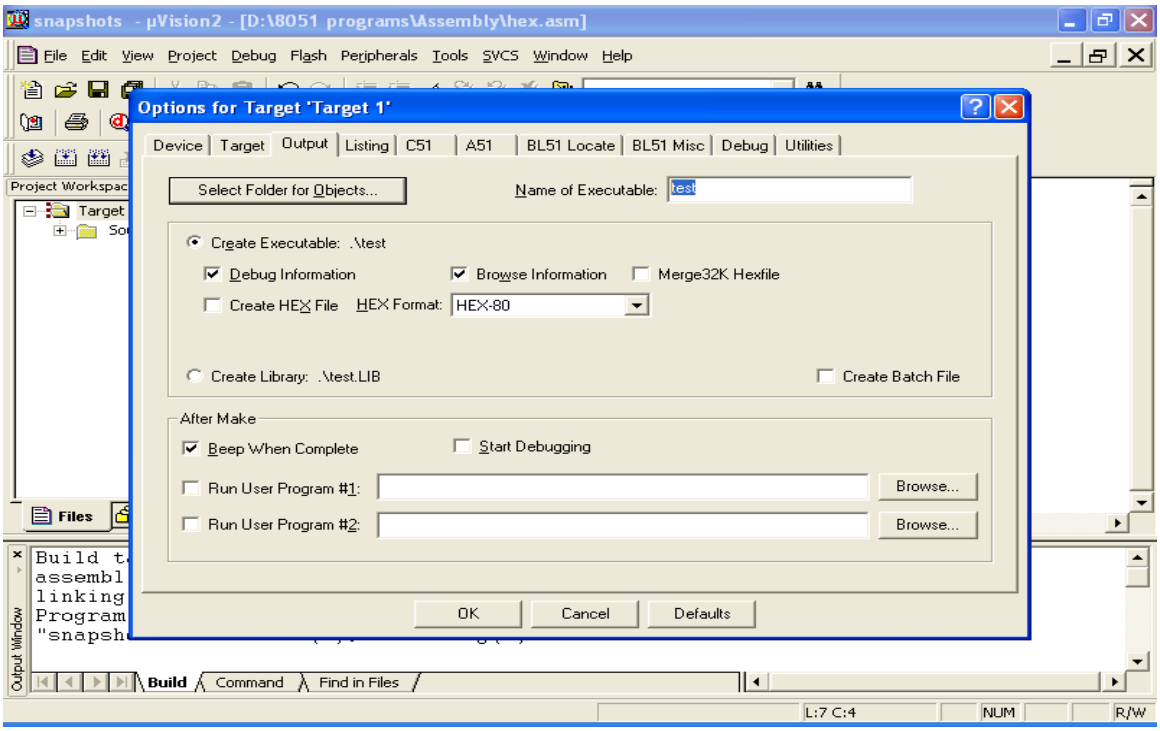
- It opens one dialogue box like below set your desired frequency



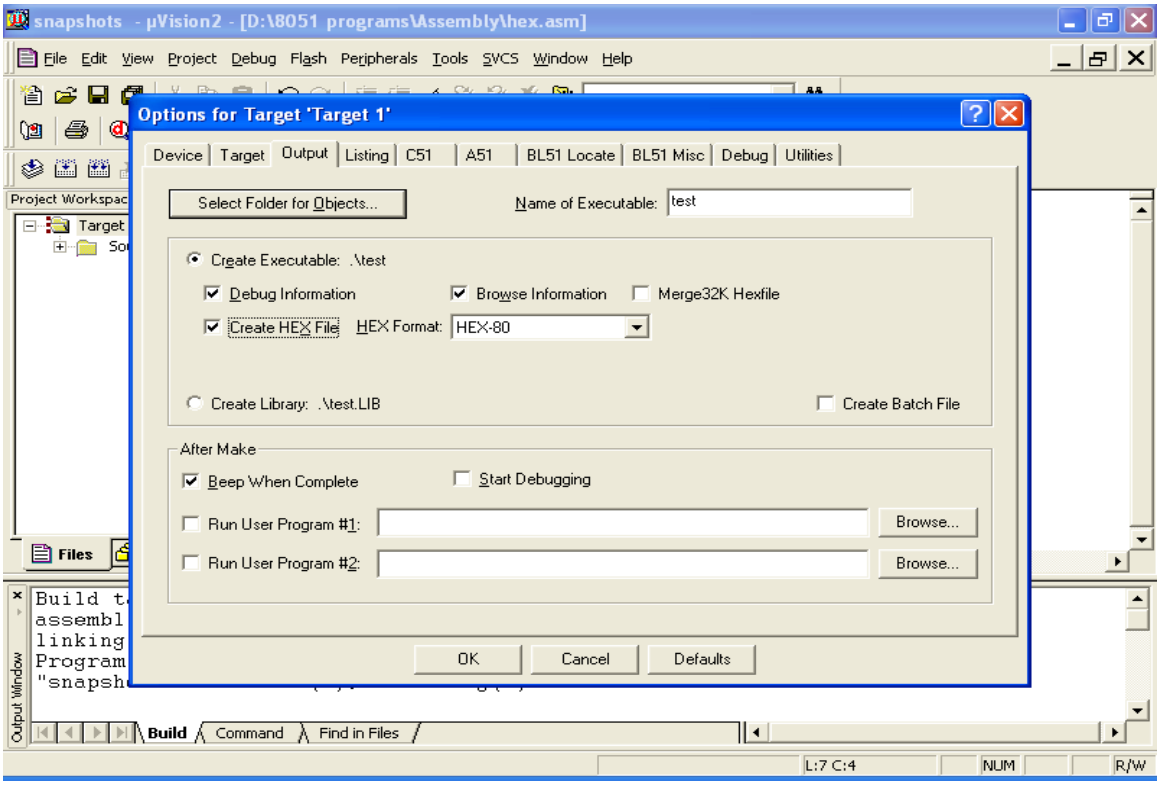
- Click on output tab



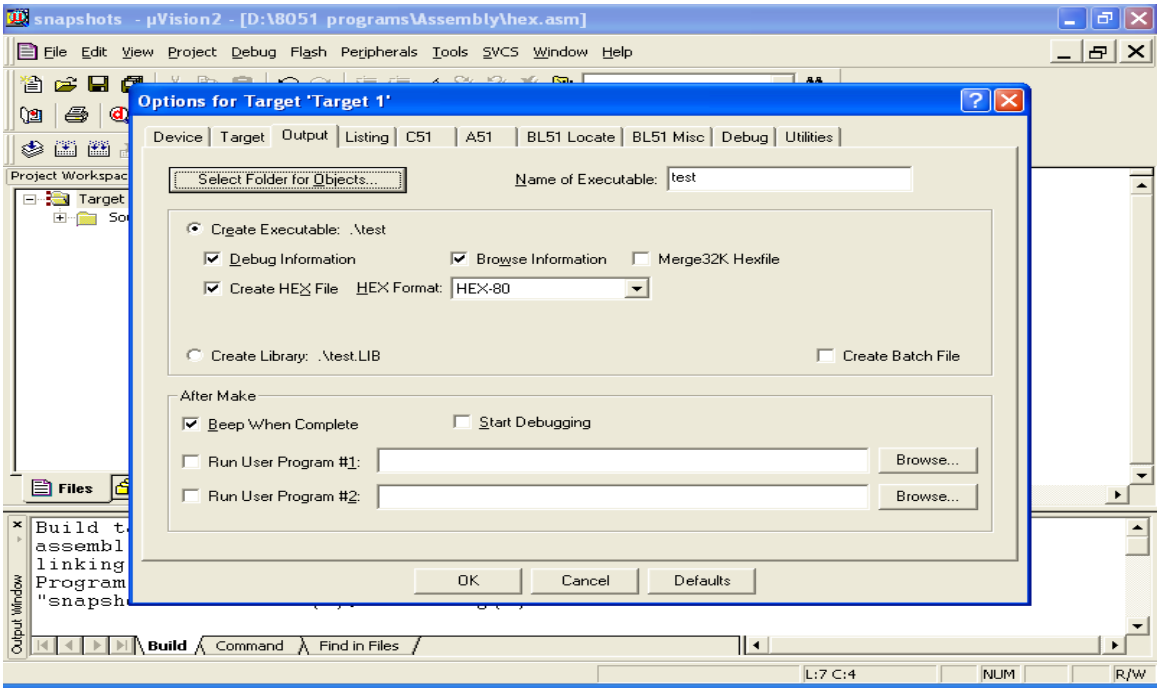
- Give desired hex file name in Name of Executable blank



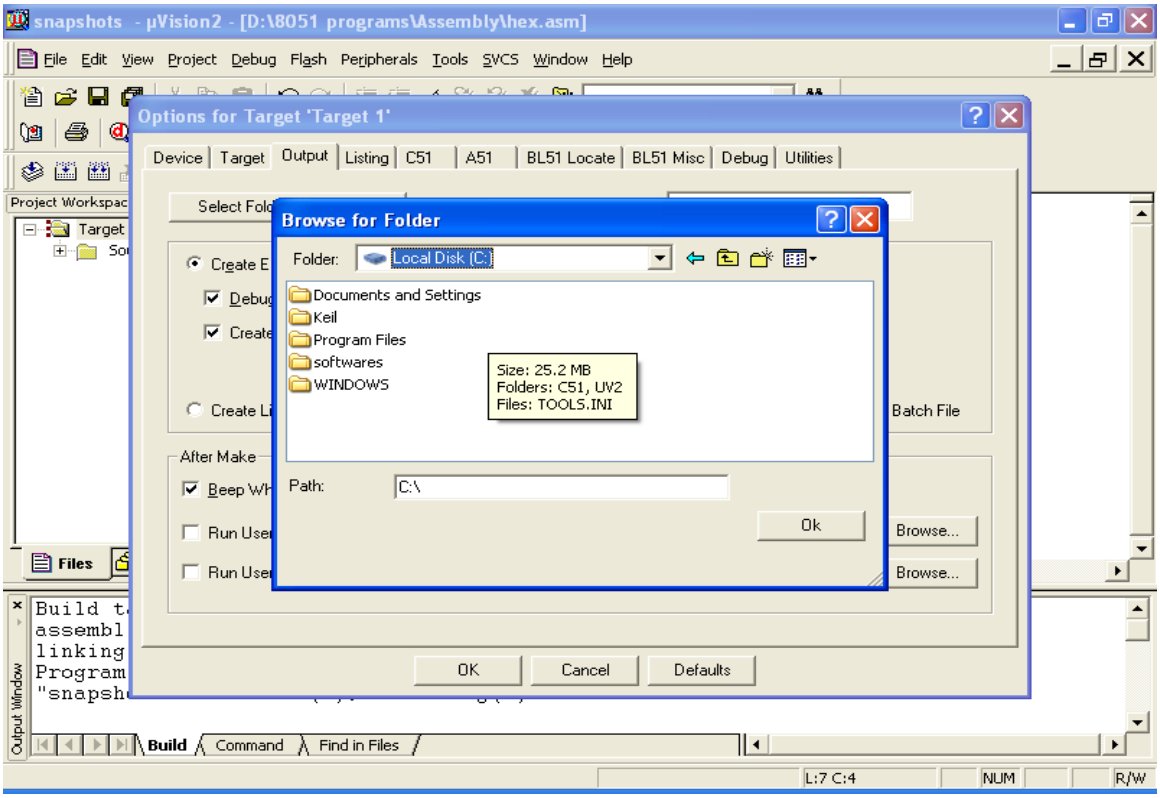
- Enable create HEX File i.e., click on empty square box



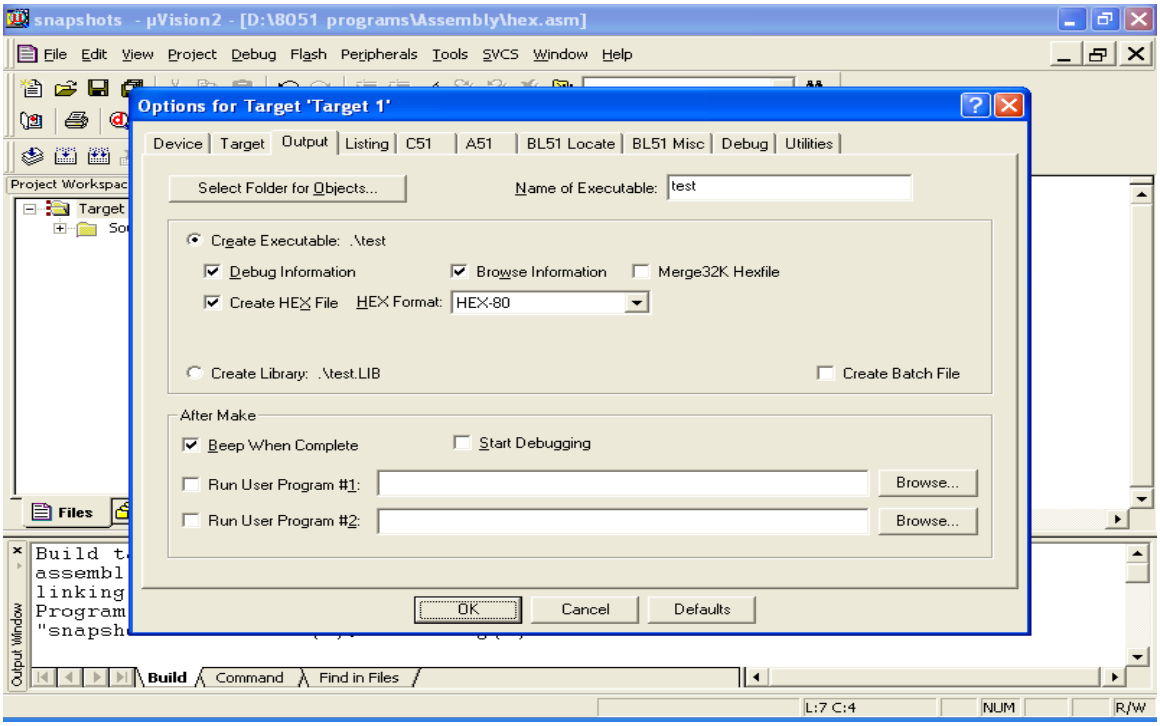
- If you want to save your hex file in particular place click on “select folder for objects” select path where you want to save



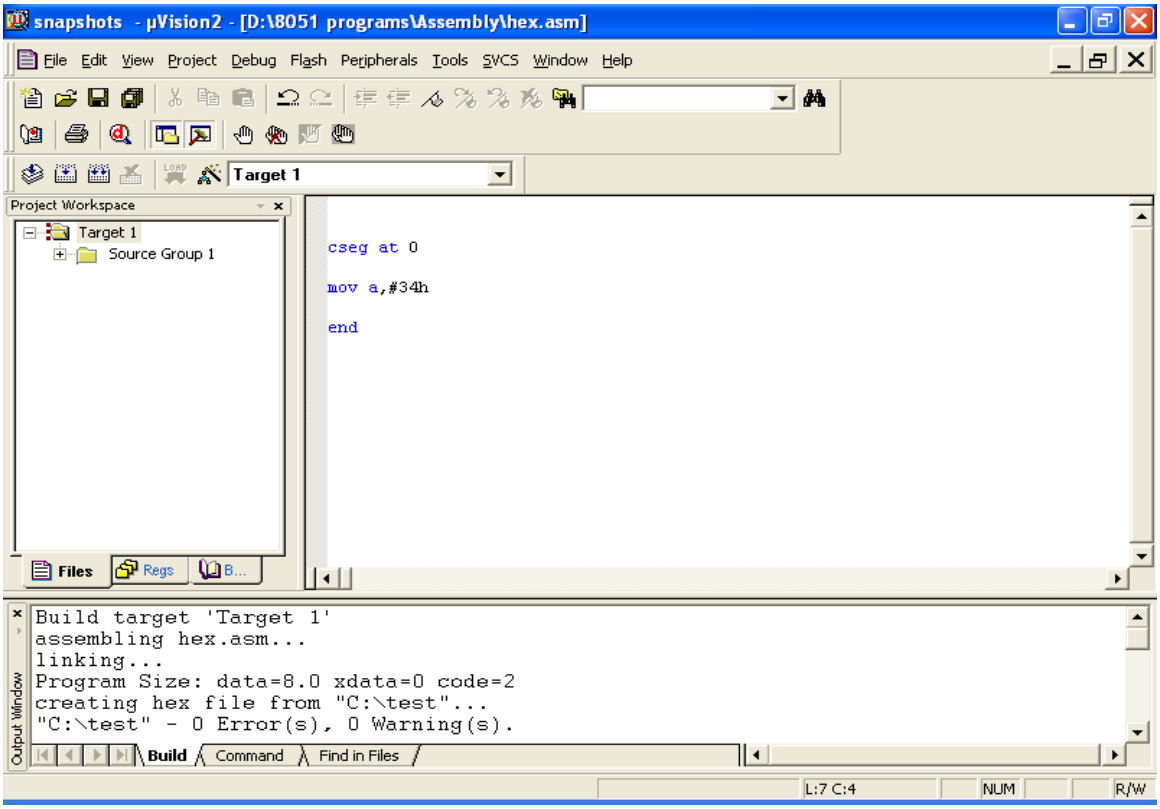
- It opens one dialogue box from there you can set your path



- Click “Ok” button



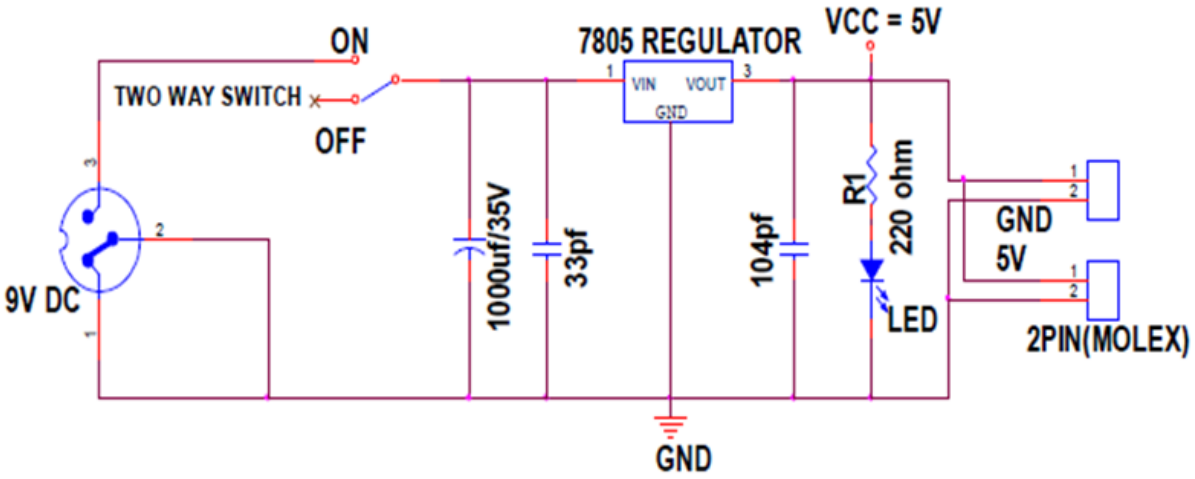
- After you can save press CTRL+S and build your target press F7 now hex file is created

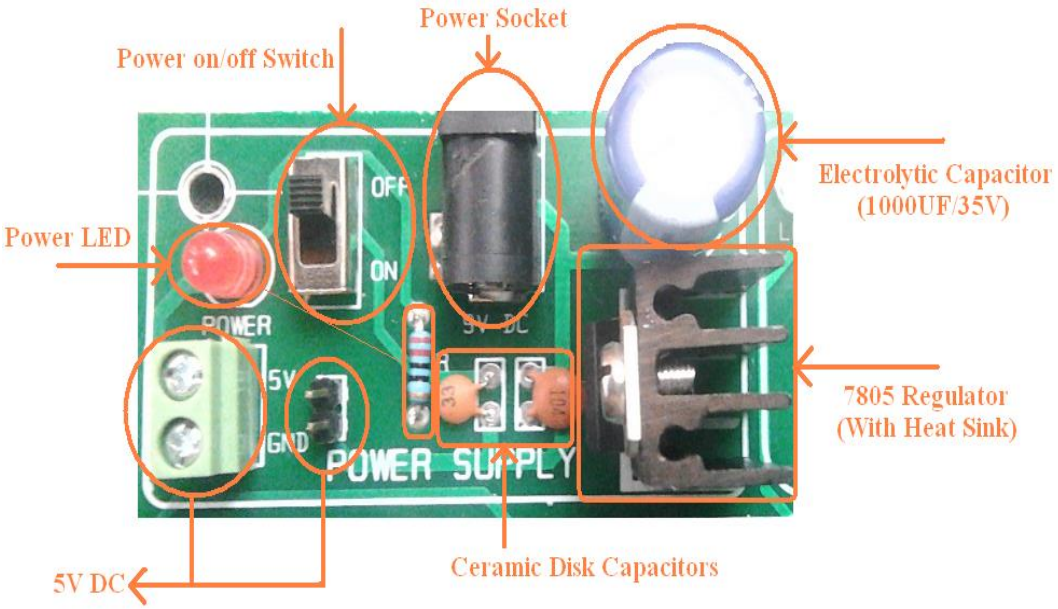


See output window it shows hex file name including with path.

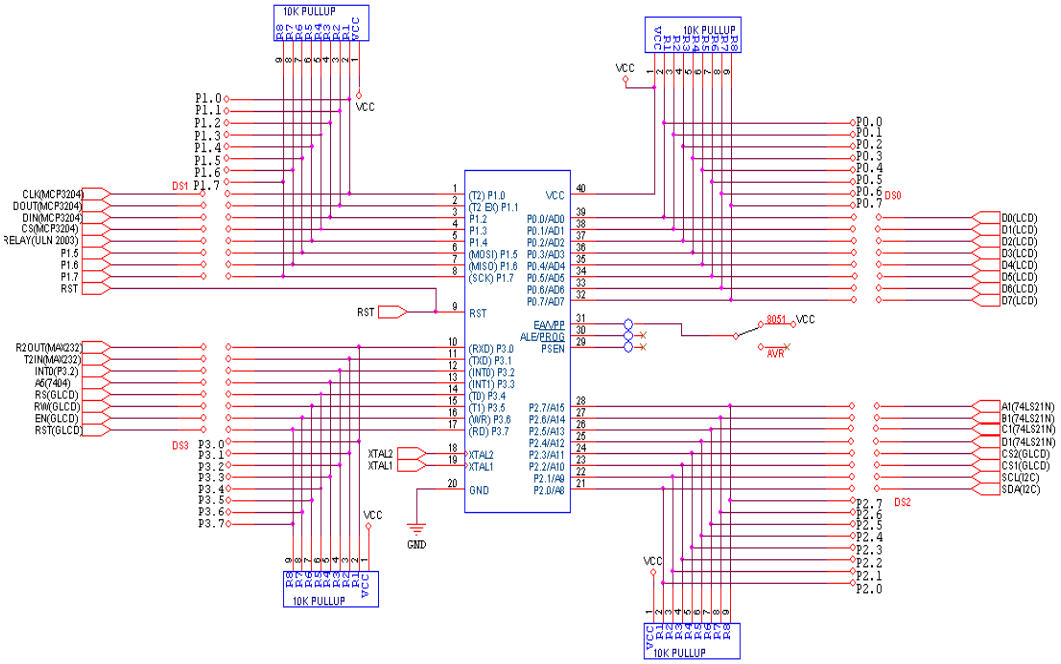
SCHEMATICS

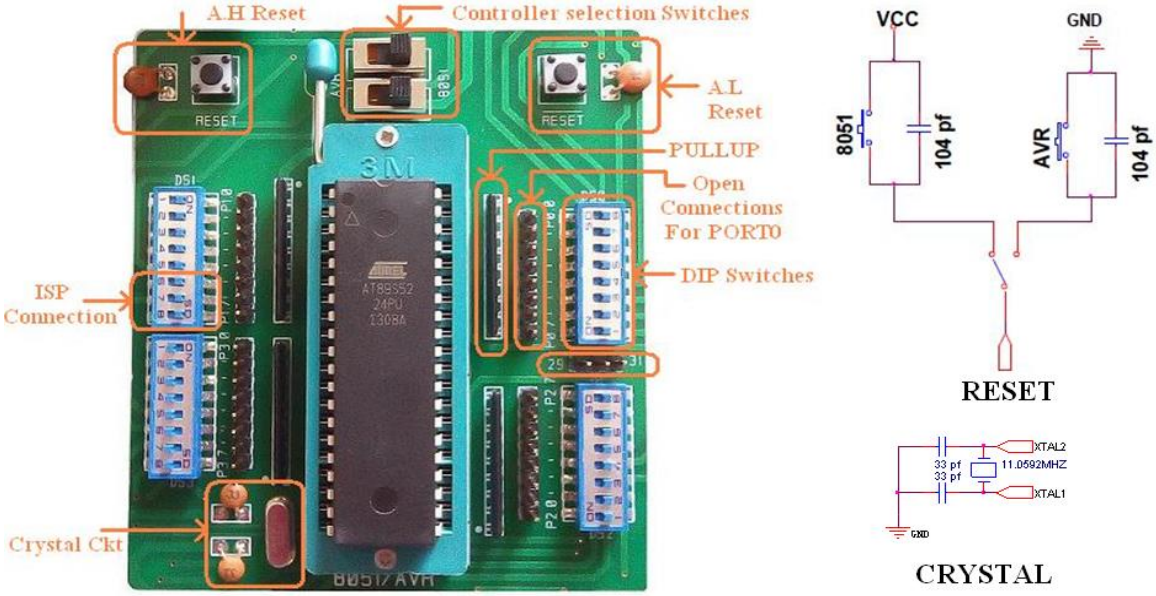
POWER SUPPLY:



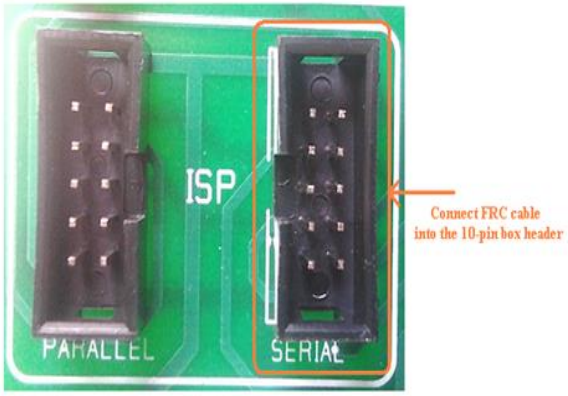


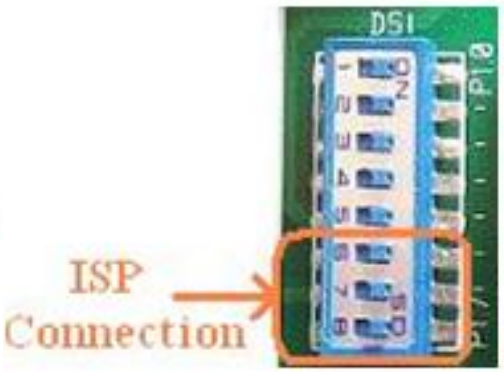
AT89S52:

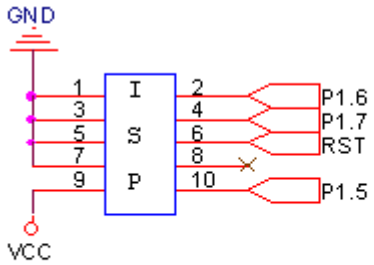

10K PULLUP
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
P1.6
P1.7
RST
VCC
GND
XTAL2
XTAL1
P0.0
P0.1
P0.2
P0.3
P0.4
P0.5
P0.6
P0.7
P2.0
P2.1
P2.2
P2.3
P2.4
P2.5
P2.6
P2.7
P3.0
P3.1
P3.2
P3.3
P3.4
P3.5
P3.6
P3.7
8051
AVR
DS0
DS1
DS2
DS3


A.H Reset
Controller selection Switches
A.L Reset
PULLUP
Open Connections For PORT0
DIP Switches
ISP Connection
Crystal Ckt
RESET
8051
AVR
VCC
GND
104 pf
RESET
XTAL2
XTAL1
33 pf
33 pf
11.0592MHZ
CRYSTAL

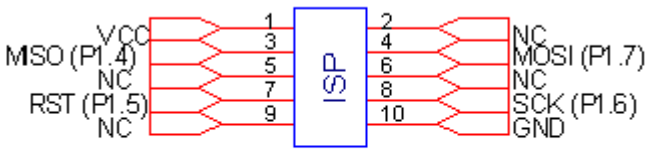
IN SYSTEM PROGRAMMERS:





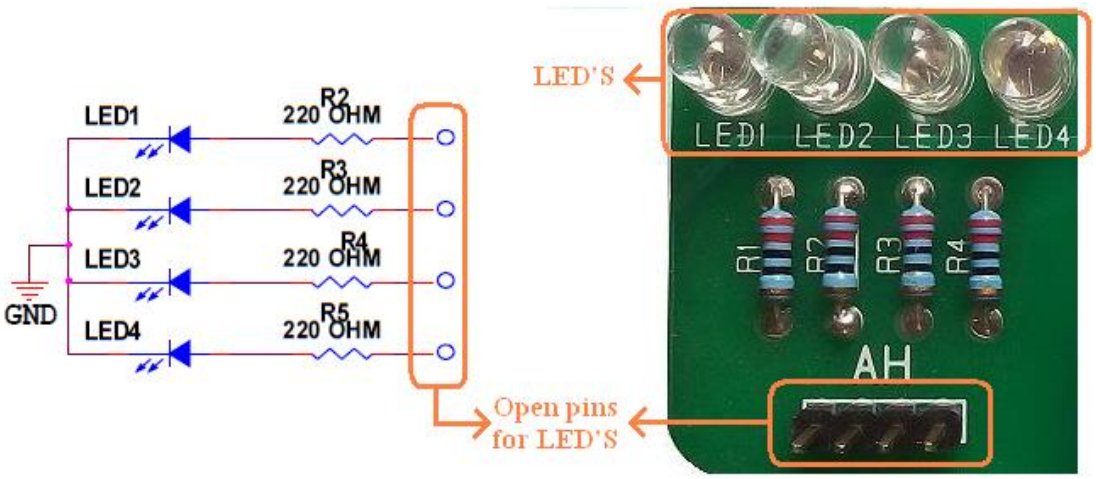


PARALLEL ISP

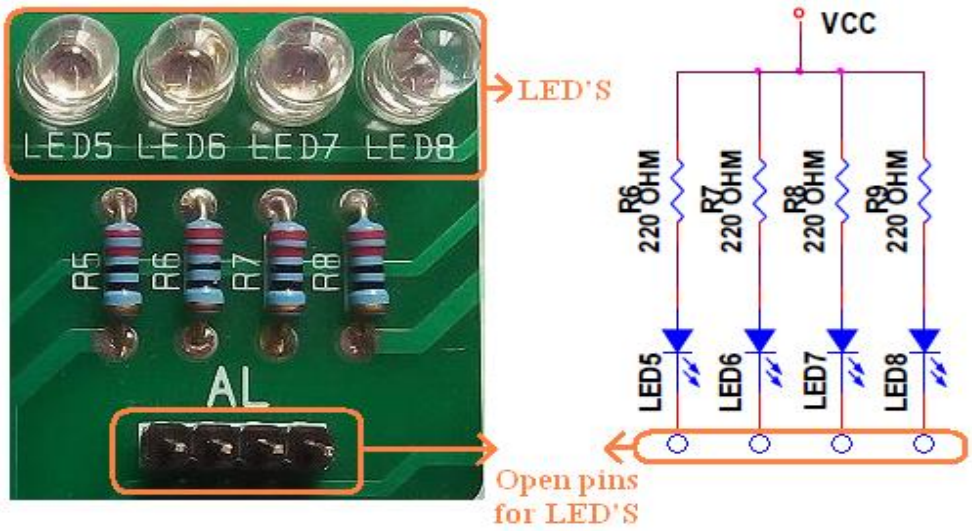


SERIAL ISP

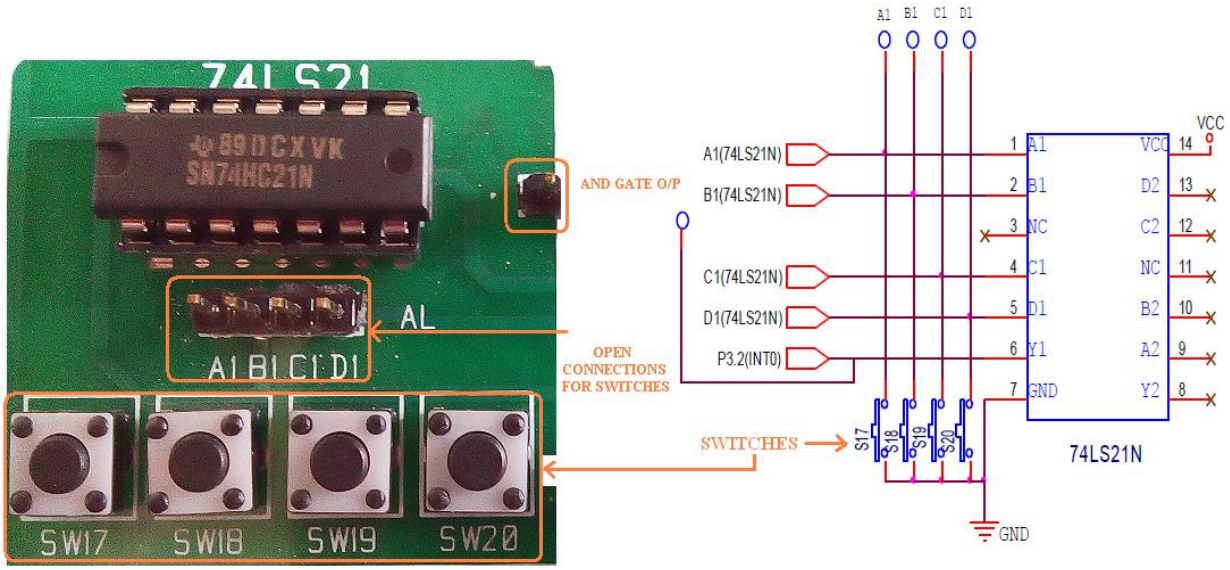
ACTIVE HIGH LED'S



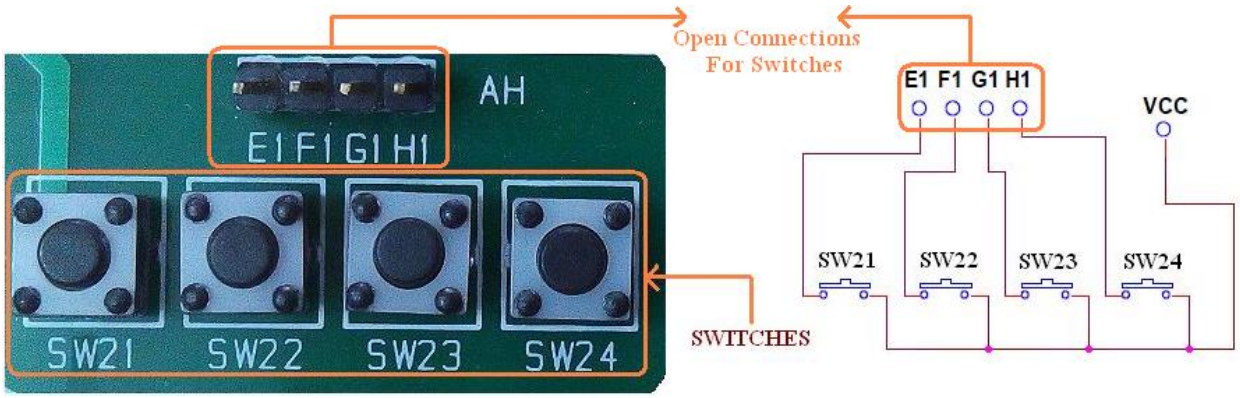
ACTIVE LOW LED'S



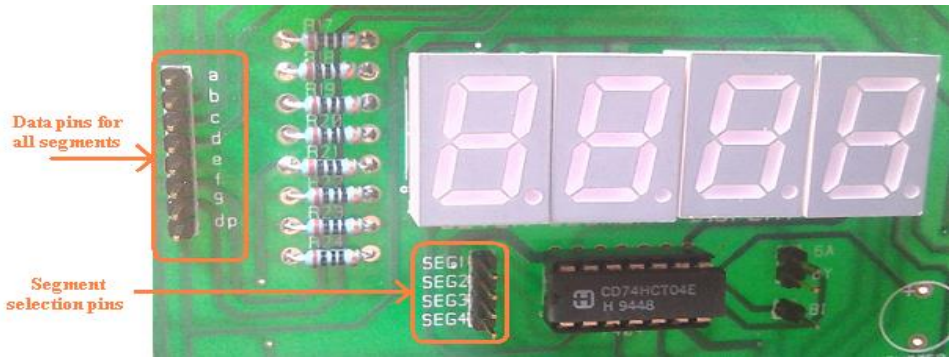
AND GATE & ACTIVE LOW SWITCHES:

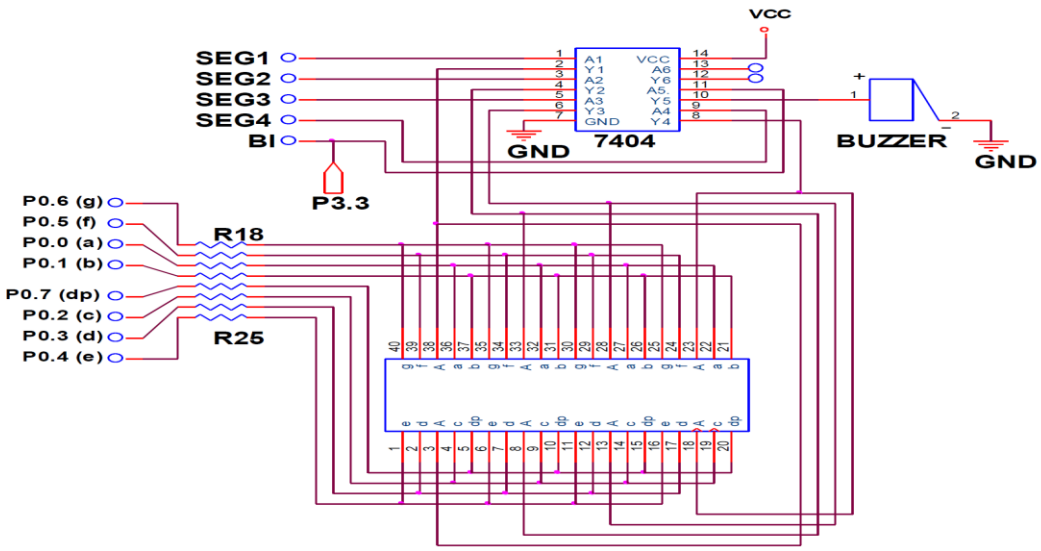


ACTIVE HIGH SWITCHES:

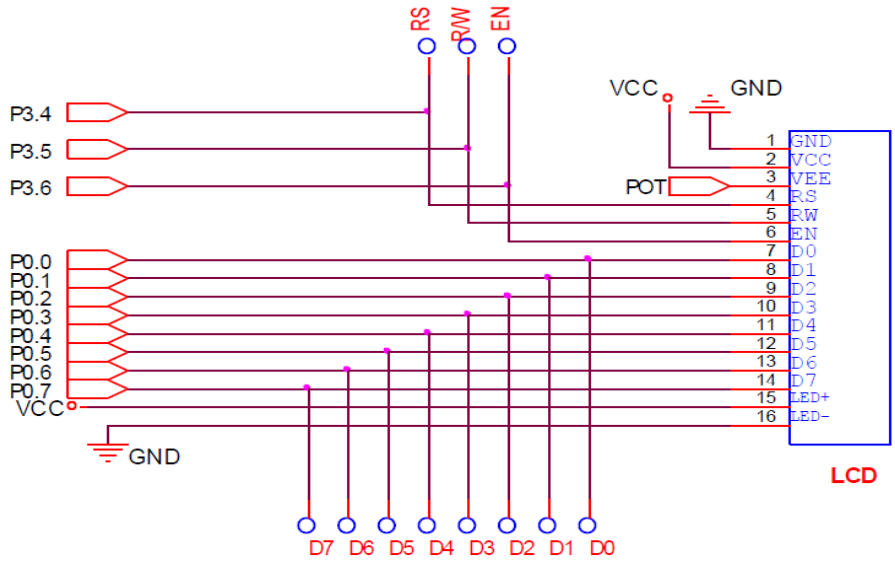


7-SEGMENT DISPLAY'S

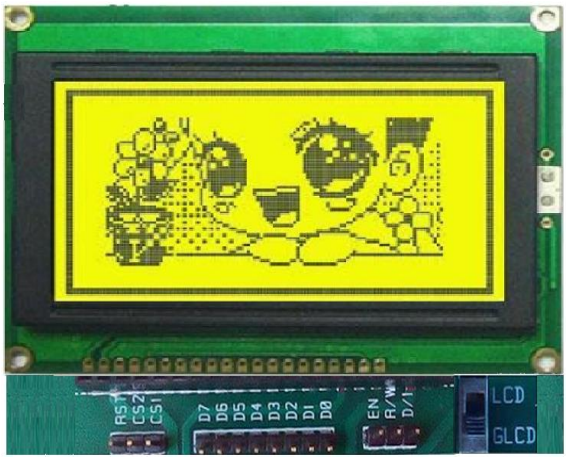


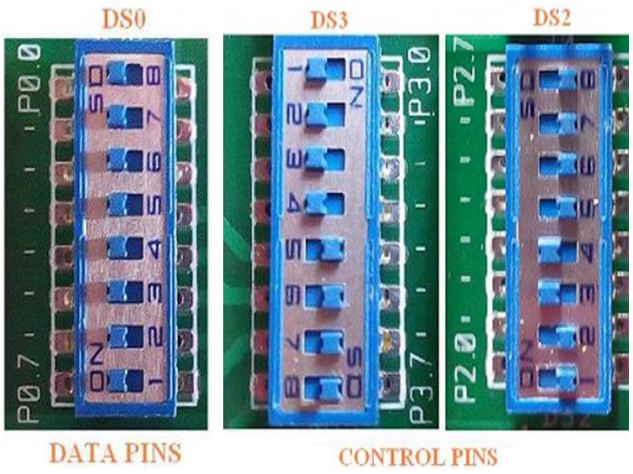


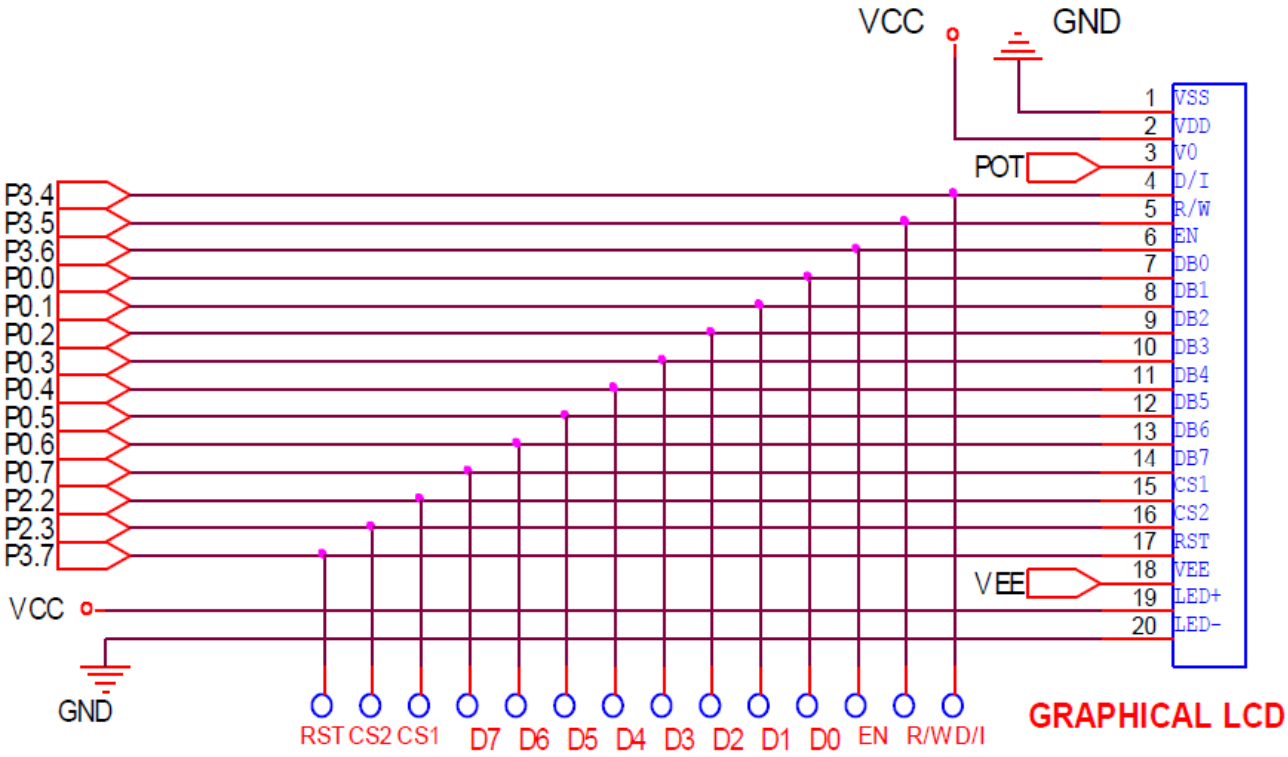
ALPHANUMERIC LCD:



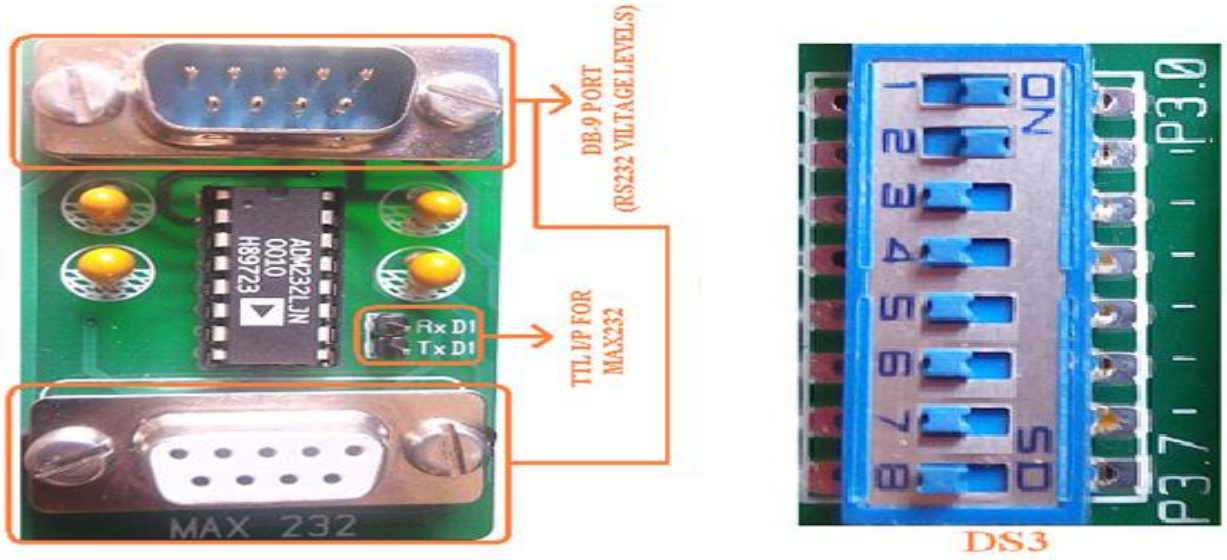
GRAPHICAL LCD:

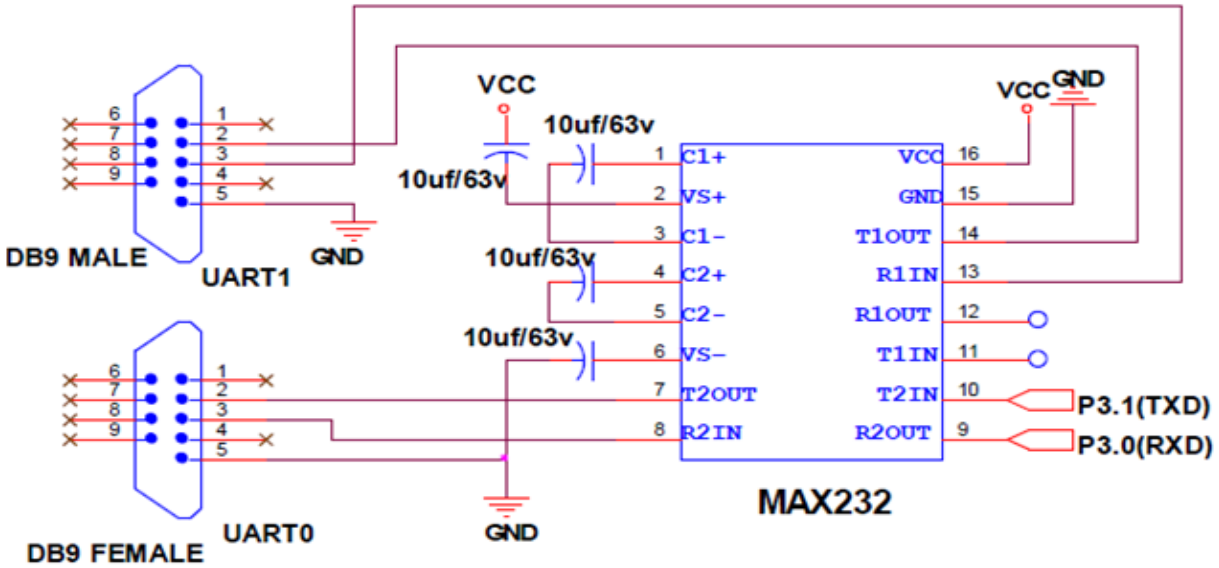




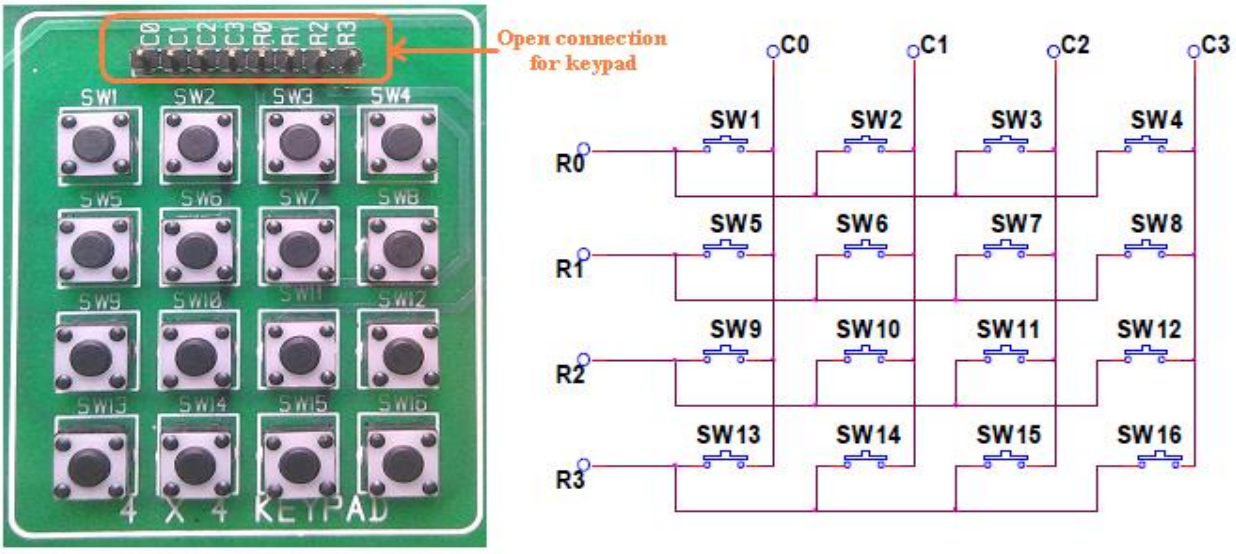


MAX 232:

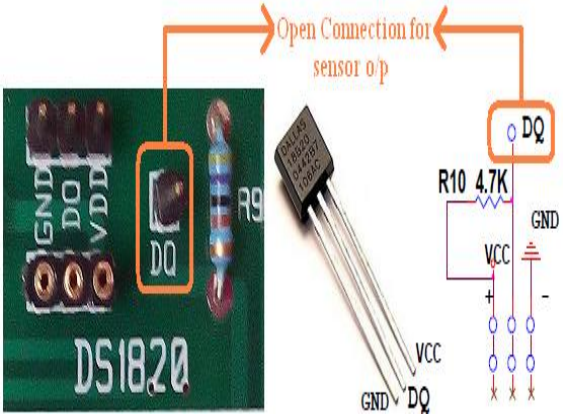

DB-9 PORT
(RS232 VILTAGE LEVELS)
TTL I/P FOR
MAX232
Rx D1
Tx D1
MAX 232
DS3
P3.0
P3.1


VCC
10uf/63v
10uf/63v
10uf/63v
10uf/63v
GND
VCC
GND
C1+
VS+
C1-
C2+
C2-
VS-
T2OUT
R2IN
VCC
GND
T1OUT
R1IN
R1OUT
T1IN
T2IN
R2OUT
P3.1(TXD)
P3.0(RXD)
MAX232
DB9 MALE
UART1
DB9 FEMALE
UART0
GND

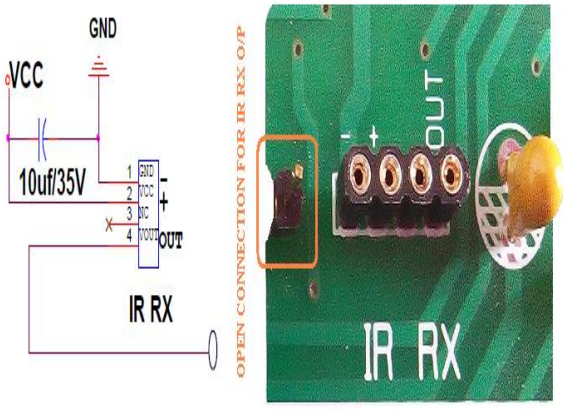
4*4 MATRIX KEYPAD:



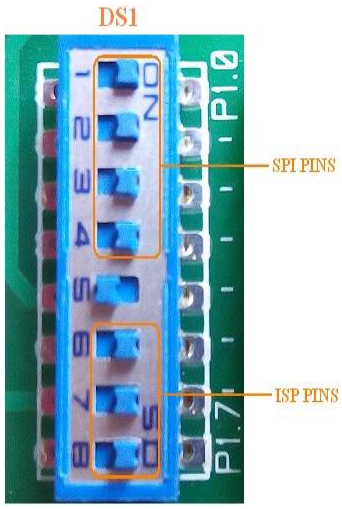
DS18B20 :

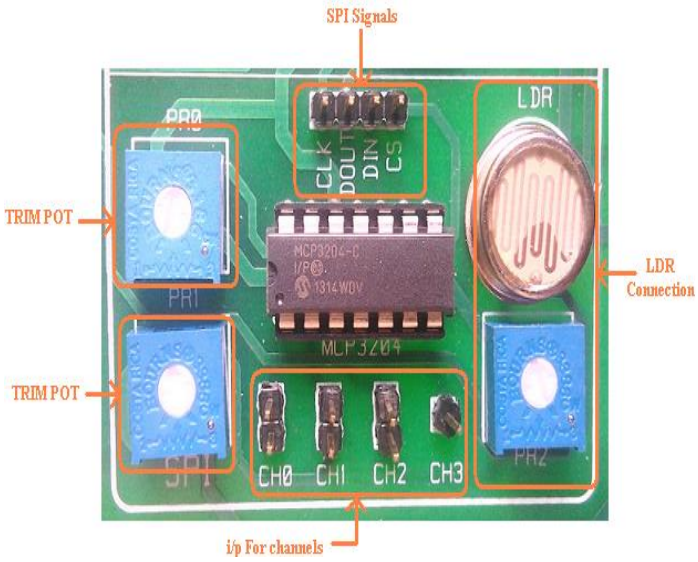


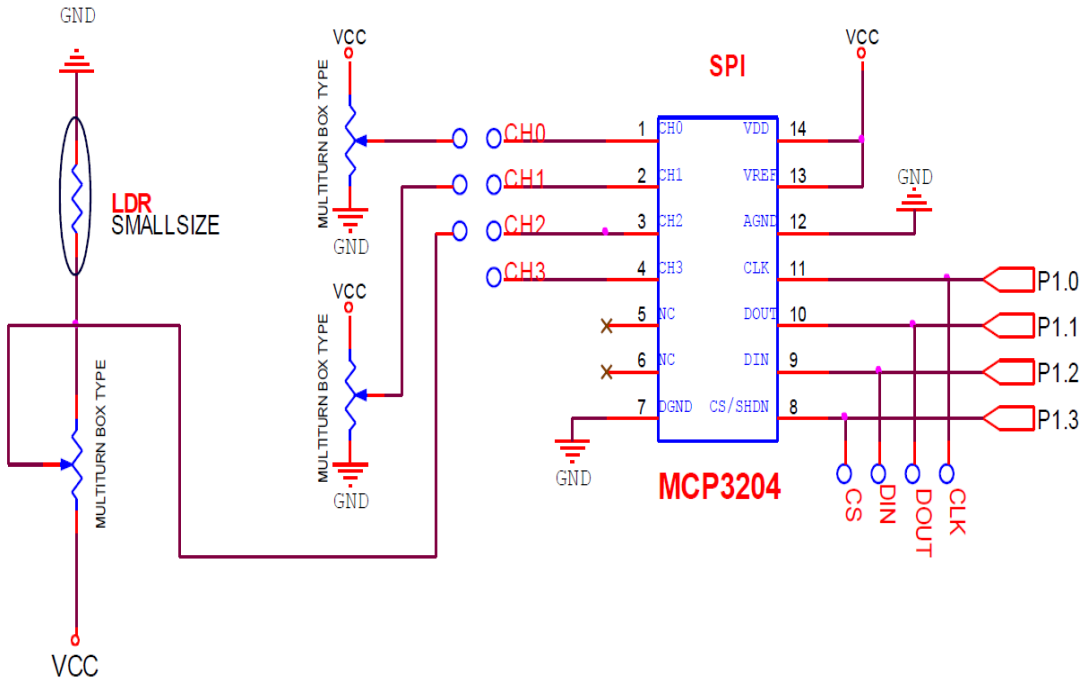
IR RX :



SPI(MCP3204):

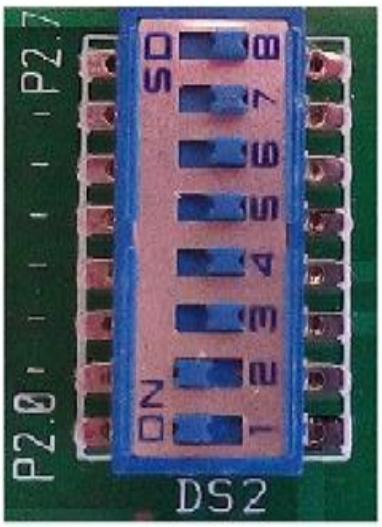

DS1
ON
1 2 3 4 5 6 7 8
P1.0
P1.7
SPI PINS
ISP PINS

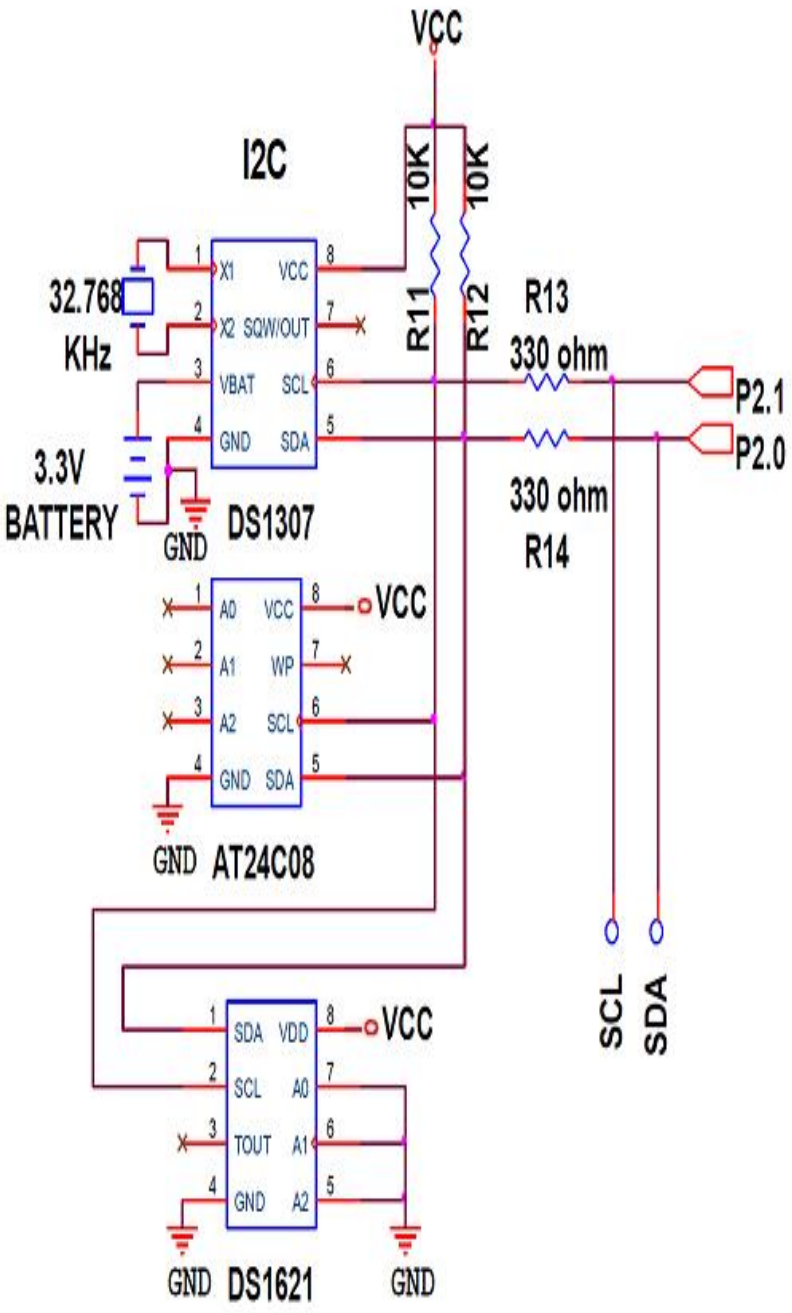

SPI Signals
LDR
CLK
DOUT
DIN
CS
TRIM POT
TRIM POT
LDR
Connection
MCP3204
CH0
CH1
CH2
CH3
i/p For channels


GND
LDR
SMALLSIZE
MULTITURN BOX TYPE
VCC
VCC
MULTITURN BOX TYPE
GND
VCC
MULTITURN BOX TYPE
GND
SPI
CH0
CH1
CH2
CH3
1 CH0
2 CH1
3 CH2
4 CH3
5 NC
6 NC
7 DGND
VDD 14
VREF 13
AGND 12
CLK 11
DOUT 10
DIN 9
CS/SHDN 8
VCC
GND
GND
MCP3204
P1.0
P1.1
P1.2
P1.3
CS
DIN
DOUT
CLK

I2C:

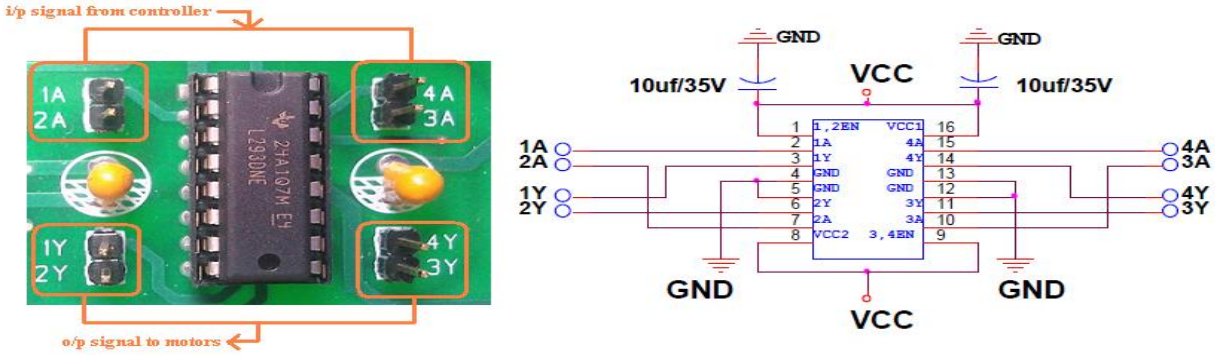

SCL
SDA
I2C


P2.7
P2.0
SD
DN
DS2

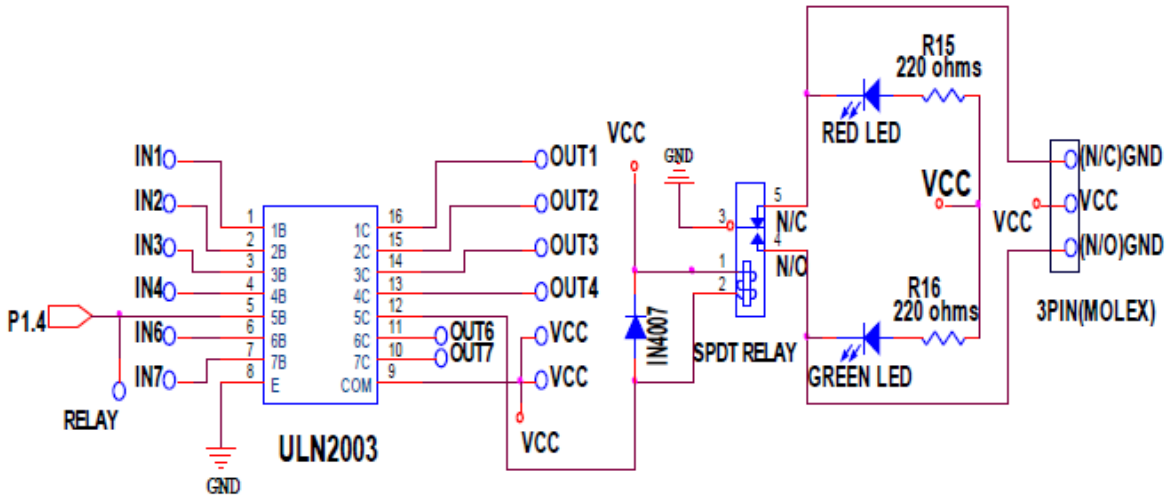

VCC
I2C
10K
10K
R11
R12
R13
330 ohm
32.768
KHz
X1
X2
VBAT
GND
VCC
SQW/OUT
SCL
SDA
P2.1
P2.0
3.3V
BATTERY
GND
DS1307
330 ohm
R14
A0
A1
A2
GND
VCC
WP
SCL
SDA
VCC
GND
AT24C08
SDA
SCL
TOUT
GND
VDD
A0
A1
A2
VCC
GND
DS1621
GND
SCL
SDA

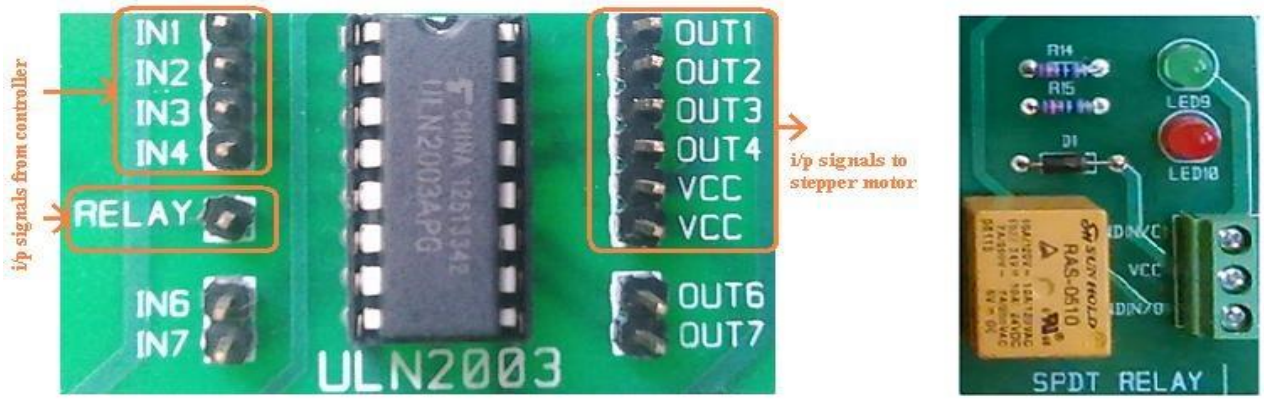
6

L293D:



ULN2003:





ASSIGNMENTS

8051 Micro controller (mandatory) Programs On

1. Data Transfer Operations……
2. Logical & Bit Operations…….
3. Arithmetic Operations……
4. Branching Operations…….
5. Interfacing of LED's ……..
6. Interfacing of Switches, LED's/Buzzer…….
7. Interfacing Seven Segment Display & switches……
8. Interfacing of LCD……..
9. Interfacing 4x4 Matrix Keypad & LCD/Seven Segment……
10. Timers/Counters……
11. Serial Communication using UART peripheral's Standard UART Mode ……..
12. Interrupts……..

Project on ***any ONE*** of the below

13. SMS Matrix Keypad…….
14. Interfacing of I2C EEPROM AT24C01 & LCD/UART……..
15. Interfacing of I2C RTC DS1307 & LCD/UART……..
16. Interfacing of SPI ADC MCP3201 & LCD/UART……..
17. Interfacing of 128*64 Graphical LCD……..
18. IR decoding, Philips Remote Control RC5 protocol implementation ……
19. Smart Card Authentication……
20. Memory Mapped I/0 using 8255 PPI……
21. Temperature Sensing using 1-wire protocol……..
22. Implementing a Round Robin Scheduler……..
23. Multitasking Application using Real Time OS…….
24. RF Based Automation……
25. Distance measurement with ultrasonic sensor.....

Data Transfer Operations (In Assembly Language) Exercises

1. Place the number 3Bh in internal RAM locations 30h to 32h.
2. Copy the byte at internal RAM address 27h to external RAM address 27h.
3. Set timer 1 to A23Dh.
4. Copy the data in external RAM location 0123h to TL0 and the data in external RAM location 0234h to TH0.
5. Exchange the contents of the B register and external RAM address 02CFh.
6. Copy the internal code byte at address 0300h to external RAM address 0300h.
7. Exchange both low nibbles of registers R0 and R1: put the low nibble of R0 in R1, and the low nibble of R1 in R0.
8. PUSH the contents of the B register to TMOD.

Logical & Bit Operations (In Assembly Language) Exercises

1. Set port 0, bits 1, 3, 5 and7, to 1; set the rest to 0.
2. Clear bit 3 of RAM location 22h without affecting any other bit.
3. Invert the data of the port 0 pins and write the data to port 1.
4. Swap the nibbles of R0 and R1 so that the low nibble of R0 swaps with the high nibble of R1 and the high nibble of R0 swaps with the low nibble of R1.
5. Complement the lower nibble of RAM location 2Ah.
6. Make the high nibble of R5 the complement of the low nibble of R6.
7. Move bit 4 of RAM location 30h to bit 2 of A.
8. Store the least significant nibble of A in both nibbles of RAM address 3Ch; for example, if A=36h, then 3Ch == 66h.
9. Set the Carry flag to 1 if the number in A is even; set the Carry flag to 0 if the number in A is odd.
10. Treat registers R0 and R1 as 16-bit registers, and rotate them on place to the left; bit 7 of R0 becomes bit 0 of R1, bit 7 of R1 becomes bit 0 of R0, and so on.
11. Reverse the bits of R0 registers.
12. Rotate the DPTR one place to the right.

Arithematic Operations (In Assembly Language) Exercises

1. Add 55h and 66h and put the result in r4 and r5 of bank 2
2. Add the content in 55h,56h and 57h and put the result a(msb) and b(lsb)
3. Add the byte in intram loc 45h with extram loc 45h
4. Sub the contents of r0 and r1 and put the res in r0 of bank1
5. Multiply FFh and FFH and store the result in r2 and r3
6. Div 58 by 5 and store the result in r0 and r1
7. Add one to every external ram location from 10h to 13h
8. Square the content of acc and store the result in ext ram loc 00 and 01
9. Store 0fdh in tl0 and increment it by 2 and put the res in th0
10. Store 0ech in r5 and convert it to bcd and store res in r6
11. Add the content of tl0 and tl1 and store the res in th0
12. Add the content of p0 with p1 and store the res in p2

Branching Operations (In Assembly Language) Exercises

1. Put a random number in R3 and increment it until it equals E1h.
2. Put a random number in address 20h and increment it until it equals a random number put in R5.
3. Detect both the OV flag & CY flag being set in 8051.If set put 1 in R7,else put 0 in R7.
4. Count the number of 1s in any number in register B and put the count in R5.
5. Transfer the data in internal RAM locations 10h to 20h to internal RAM locations s30h to 40h.
6. Write a program to copy a block of 10 bytes of data from RAM locations starting at 35H to RAM locations starting at 60H.
7. Assuming that in ROM space at 250H contains ‘Vector’, write a program to transfer the bytes into RAM locations starting at 40H.
8. Let the assembler locate (initialize) the string ‘Welcome’ in ROM space .Write an ALP to bring in the string into the RAM space.

9. Write a program to add the following numbers and save the result in R2, R3. The data is stored in on-chip ROM.
   MYDATA: DB 53, 94, 56, 92, 74, 65, 43, 23, 83
10. Write a sub-routine that adds to 8-bit numbers and stores the result in r6 (MSB) and r7(LSB) and call it.
11. Write a sub-routine to create a delay of about 1 ms and call it.
12. Write a sub-routine to create any approximate delay within of 1 ms up to 100ms.

Interfacing of LEDs

1. Flash an LED connected to port line using complement logic at the rate of 1 second for 10 times and stop.
2. Flash an LED connected to port line using the set and clear logic each at the rate of 1second for 10 times and stop.
3. Display on the LEDS connected to port the pattern as explained:

   First all the LEDS should be off and remaining in this state for about 1 sec.
   After 1 second laps switch on the only LED0 (bit 0)
   After 2 seconds laps switch on the only LED1 (bit 1)
   ……
   ……
   After 8 seconds laps switch on the only LED7 (bit 7)
   Switch of all the LEDS for about 3 secs.
   Repeat this pattern for 5 times ONLY.

Interfacing of Switches and LEDs & Buzzer

1. Read the status of momentary switch (SW) connected to any port pin and display it on the LED & make the buzzer beep connected to another port pins.

2. A momentary switch (SW) is connected P2.5 and four LEDs are connected to lower nibble of port 0.Your program should behave as follows :
   Depending on the number of times the SW pressed -
   The 1st time, LED 0 should be ON only,
   The 2nd time, LED 1 should be ON only,
   The 3rd time, LED 2 should be ON only,
   The 4th time, LED 3 should be ON only,
   If pressed here after none should be ON until system reset.
3. Read the status of all momentary switches SW's connected to PORT 0 and display their status of being pressed on corresponding LEDS connected to PORT 2 infinitely.
4. Read the status of 4 momentary switches SW's connected to lower nibble of PORT 0 and display their status on an LED connected to P2.0 only.

Interfacing of Seven Segments & Switches

1. Show up counting from 00 to 99 on seg1, seg2 @ of 1s.
2. Display "HELP" on segments and Make if flash 5 times
3. Implement a 2 digit up and down counter. Use two switches as mentioned_sw1 for incrementing count & sw2 for decrement count.
4. Display on 4 digits @ 1s per display the consecutively values 1.234, 12.34, 123.4 infinitely.

Interfacing of LCD

1. Display and Scroll the string "VECTOR", towards right, on Line 1 of LCD until it disappears from the screen.
2. Display and circularly Scroll the string "VECTOR", towards right, on Line 1 of LCD infinitely on the screen.
3. Display 4 new customized characters.

Understanding Timer/Counter peripherals usage

Write8051 assembly programs to create generate a square wave of duty cycles 25%, 50%, 33%, and 75%.

What value do we need to load into the timer's registers if we want to have time delay of 10ms.Write the program for infinitely creating a pulse width of 10ms P0.7?

Assuming that XTAL=12MHz, write a program to generate a square wave of 2 KHz frequency on P0.6.

Solution:

T = 1/f = 1/2 kHz = 500us the period of square wave

½ of it for high and low portions of the pulse is 250us

250us/1us=250 and 65536 – 250 = 65286(convert to hex).

Load TLX and THX

Write an embedded C program to count pulse inputs at the pin T1.

Serial Communication:

1) Write an ALP to serially transmit the message "Welcome to VECTOR" on serial window/pc hyper terminal window.

2) Write an ALP to infinitely receive any character then re-transmit its ASCII value in decimal and hex format. Output should be seen as for example below

Output: Received 'A' 65 41

Received 'a' 97 61

……..

3) Using the concept of multiple file assembly, assuming three files, say mainfile.asm, serial.asm and delay.asm are edited as mentioned below

Mainfile.asm – contains the main logic of the application which invokes routines from other files

Serial.asm - contains UART input and output routines

Delay.asm – contains loop delay and timer delay routine

Write an ALP to show 00 – 99 secs counters on the serial window.

4) Write an Embedded-C program for demonstrating full duplex feature of UART in mode 1.

Interrupts

1.Test with some form of visual outputs the response of the microcontroller to various interrupt sources.