KTU NOTES

MODULE III
PROGRAMMING AND INTERFACING OF 8051

3.1 SIMPLE PROGRAMMING EXAMPLES IN ASSEMBLY LANGUAGE

ASSEMBLER DIRECTIVES.

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

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

The following are the widely used 8051 assembler directives.

ORG (origin)

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

```
Eg: ORG 0000H        ; Set PC to 0000.
```

EQU and SET

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

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

SET directive allows redefinition of symbols at a later stage.

DB (DEFINE BYTE)

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

```
DATA1:  DB  40H          ; hex
DATA2:  DB  01011100B    ; binary
DATA3:  DB  48           ; decimal
DATA4:  DB  'HELLO W'    ; ASCII
```

END

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

3.2 ASSEMBLY LANGUAGE PROGRAMS.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Assume that ten 8 bit numbers are stored in internal data memory from location 50H, henceR0 or R1 must be used as a pointer.

The program is as follows.

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

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

```
ORG 0000H
MOV 40H,#05H
MOV 41H,#55H
MOV 42H,#06H
MOV 43H,#1AH
MOV 44H,#09H
MOV R0,#40H
MOV R5,#05H
MOV B,R5CLR A
Loop: ADD A,@RO
INC RO
DJNZ R5,Loop
DIV AB
MOV 55H,A
END
```

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

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

13. Write a program to exchange the lower nibble of data present in external memory 6000H and6001H

```
ORG 0000H                 ; Set program counter 00h
MOV DPTR, #6000H ; Copy address 6000H to DPTR
MOVX A, @DPTR         ; Copy contents of 6000H to A
MOV R0, #45H          ; Load pointer, R0=45H
MOV @RO, A            ; Copy cont of A to RAM pointed by R0
INC DPL               ; Increment pointer
MOVX A, @DPTR         ; Copy contents of 6001H to A
XCHD A, @R0           ; Exchange lower nibble of A with RAM pointe
d by RO
MOVX @DPTR, A         ; Copy contents of A to 6001H
DEC DPL               ; Decrement pointer
MOV A, @R0            ; Copy cont of  RAM pointed by R0 to A
MOVX @DPTR, A         ; Copy cont of A to RAM pointed by DPTR
END
```

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

```
          ORG 00008              ; Set program counter 00008
          MOV DPTR, #6000h       ; Copy address 6000H to DPTR
          MOVX A, @DPTR          ; Copy number to A
          MOV R0,#08             ; Copy 08 in RO
          MOV R2,#00             ; Copy 00 in R2
          MOV R3,#00             ; Copy 00 in R3
          CLR C                  ; Clear carry flag
          BACK:  RLC A           ; Rotate A through carry flag
          JC NEXT                ; If CF = 1, branch to next
          INC R2                 ; If CF = 0, increment R2
          AJMP NEXT2
NEXT:     INC R3                 ; If CF = 1, increment R3
NEXT2:   DJNZ RO,BACK            ; Repeat until RO is zero
         END
```

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

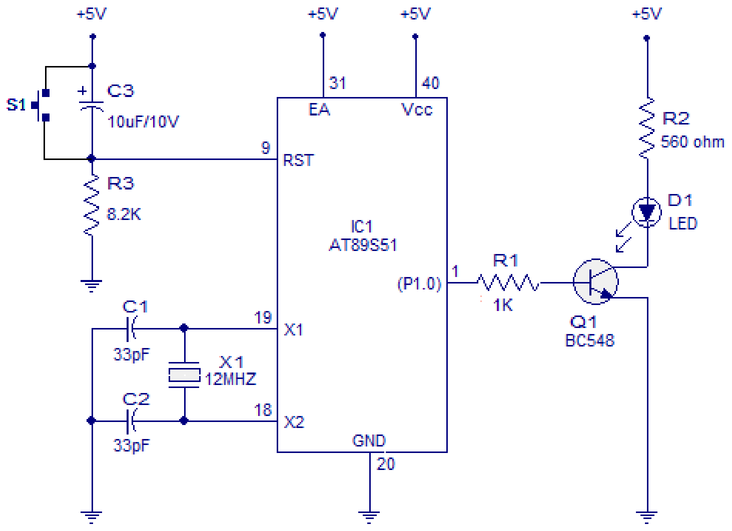
```
              ORG 0000H       ; Set program counter 0000h
              MOV R1,#04      ; Set up loop count to 4
     again:   MOV A,55H       ; Place the least significant byte of data in A
              CLR C           ; Clear tne carry flag
              RLC A           ; Rotate contents of A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

3.3 INTERFACING WITH 8051 USING ASSEMBLY LANGUAGE PROGRAMMING:

LED INTERFACING TO 8051

Blinking 1 LED using 8051

This is the first project regarding 8051 and of course one of the simplest, blinking LED using 8051. The microcontroller used here is AT89S51 In the circuit, push button switch S1, capacitor C3 and resistor R3 forms the reset circuitry. When S1 is pressed, voltage at the reset pin (pin9) goes high and this resets the chip. C1, C2 and X1 are related to the on chip oscillator which produces the required clock frequency. P1.0 (pin1) is selected as the output pin. When P1.o goes high the transistor Q1 is forward biased and LED goes ON. When P1.0 goes low the transistor goes to cut off and the LED extinguishes. The transistor driver circuit for the LED can be avoided and the LED can be connected directly to the P1.0 pin with a series current limiting resistor(~1K). The time for which P1.o goes high and low (time period of the LED) is determined by the program. The circuit diagram for blinking 1 LED is shown below.



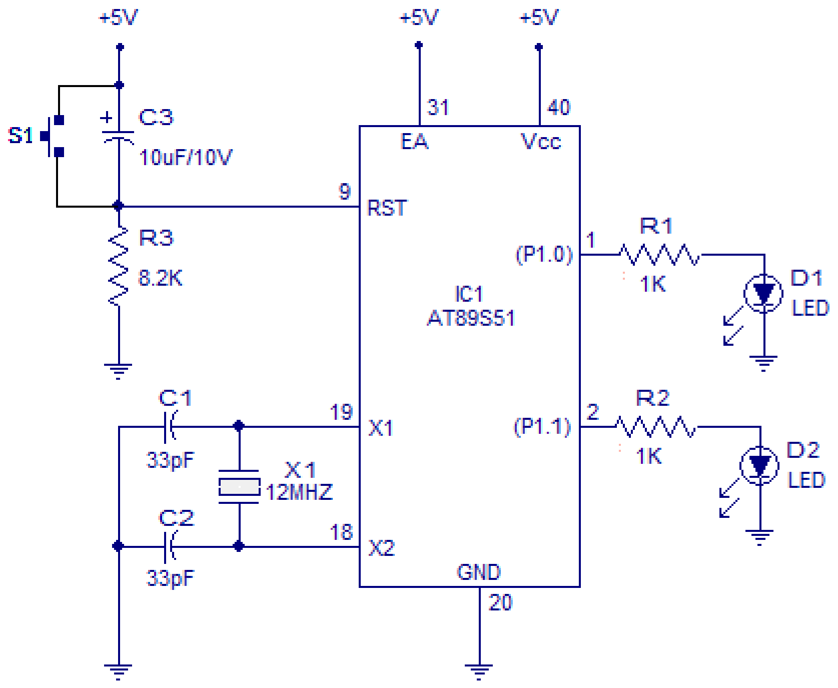
Blinking LED using 8051

Program

```
START: CPL P1.0
            ACALL WAIT
            SJMP START
WAIT:  MOV R4,#05H
WAIT1: MOV R3,#00H
WAIT2: MOV R2,#00H
WAIT3: DJNZ R2,WAIT3
          DJNZ R3,WAIT2
          DJNZ R4,WAIT1
          RET
```

Blinking 2 LED alternatively using 8051.

This circuit can blink two LEDs alternatively. P1.0 and P1.1 are assigned as the outputs. When P1.0 goes high P1.0 goes low and vice versa and the LEDs follow the state of the corresponding port to which it is connected. Here there is no driver stage for the LEDs and they are connected directly to the corresponding ports through series current limiting resistors (R1 & R2). Circuit diagram is shown below.



Blinking 2 LED alternatively using 8051

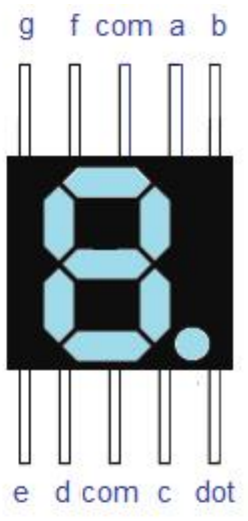
Program

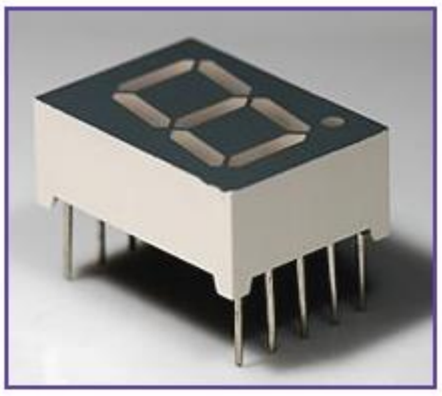
```
START: CPL P1.0
       ACALL WAIT
       CPL P1.0
       CPL P1.1
       ACALL WAIT
       CPL P1.1
       SJMP START
WAIT:  MOV R4,#05H
WAIT1: MOV R3,#FFH
WAIT2: MOV R2,#FFH
WAIT3: DJNZ R2,WAIT3
       DJNZ R3,WAIT2
       DJNZ R4,WAIT1
       RET
```

INTERFACING 7 SEGMENT DISPLAY TO 8051.

A NOTE ABOUT 7 SEGMENT LED DISPLAY.

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





Digit drive pattern.

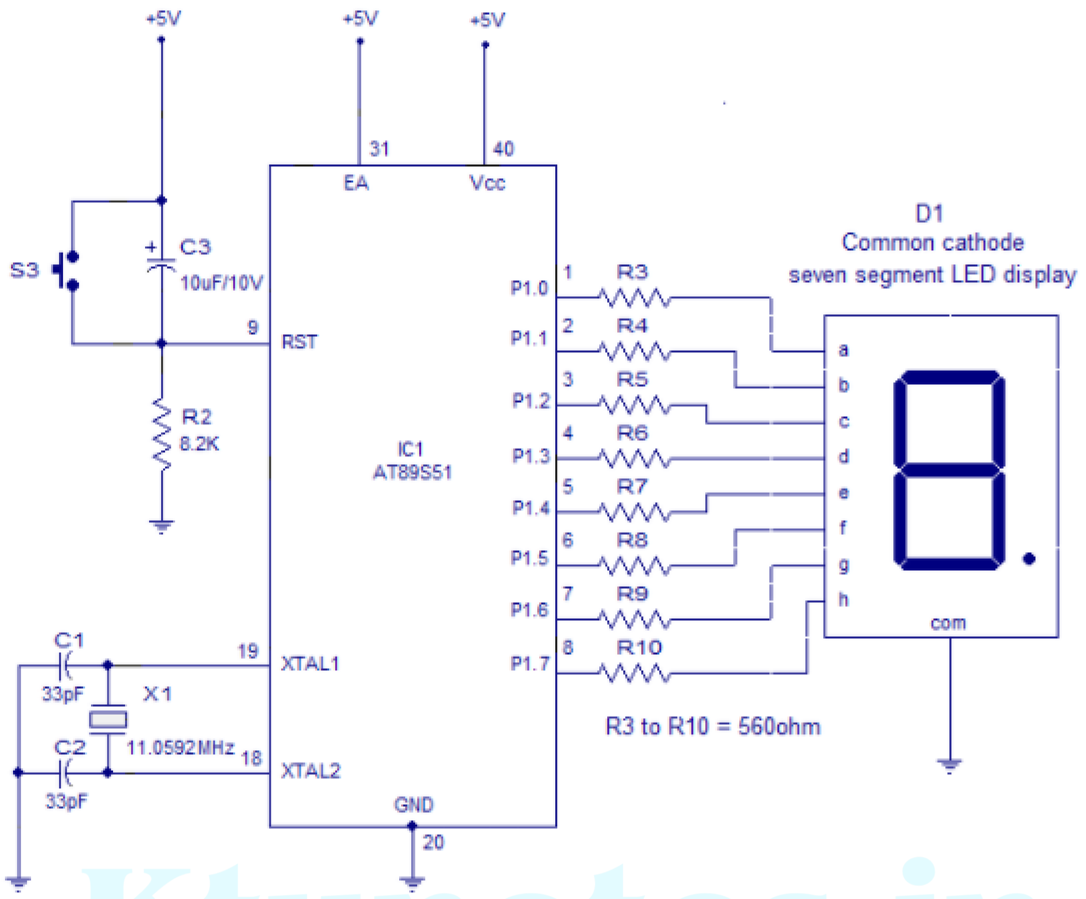
Digit drive pattern of a seven segment LED display is simply the different logic combinations of its terminals **'a' to 'h'** in order to display different digits and characters. The common digit drive patterns (0 to 9) of a seven segment display are shown in the table below.

*	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0	*
Character	**h**	**g**	**f**	**e**	**d**	**c**	**b**	**a**	**HEX**
0	0	0	1	1	1	1	1	1	0x3F
1	0	0	0	0	0	1	1	0	0x06
2	0	1	0	1	1	0	1	1	0x5B
3	0	1	0	0	1	1	1	1	0x4F
4	0	1	1	0	0	1	1	0	0x66
5	0	1	1	0	1	1	0	1	0x6D
6	0	1	1	1	1	1	0	1	0x7D
7	0	0	0	0	0	1	1	1	0x07
8	0	1	1	1	1	1	1	1	0x7F
9	0	1	1	0	1	1	1	1	0x6F

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

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

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



Interfacing 7 segment display to 8051

PROGRAM.

```
ORG 000H                        //initial starting address
START: MOV A,#00001001H         // initial value of accumulator
MOV B,A
MOV R0,#0AH                     //Register R0 initialized as counter which counts from 10 to
0
LABEL: MOV A,B
INC A
MOV B,A
MOVC A,@A+PC                    // adds the byte in A to the program counters address
MOV P1,A
ACALL DELAY                     // calls the delay of the timer
DEC R0                          //Counter R0 decremented by 1
MOV A,R0                        // R0 moved to accumulator to check if it is zero in next
instruction.
JZ START                        //Checks accumulator for zero and jumps to START.
                                  Done to check if counting has been finished.
SJMP LABEL
DB 3FH                          // digit drive pattern for 0
DB 06H                        // digit drive pattern for 1
DB 5BH                        // digit drive pattern for 2
DB 4FH                        // digit drive pattern for 3
DB 66H                        // digit drive pattern for 4
DB 6DH                        // digit drive pattern for 5
DB 7DH                        // digit drive pattern for 6
DB 07H                        // digit drive pattern for 7
DB 7FH                        // digit drive pattern for 8
DB 6FH                        // digit drive pattern for 9
DELAY: MOV R4,#05H            // subroutine for delay
WAIT1: MOV R3,#FFH
WAIT2: MOV R2,#FFH
WAIT3: DJNZ R2,WAIT3
DJNZ R3,WAIT2
DJNZ R4,WAIT1
RET
END
```

ABOUT THE PROGRAM.

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

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

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

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

3.4 PROGRAMMING IN C

Embedded C

For programming the embedded hardware devices, we need to use Embedded C language instead of our conventional C language.

The key differences between conventional C and Embedded C are

- Embedded C has certain predefined variables for registers, ports etc. which are in 8051 e.g. ACC, P1, P2, TMOD etc.
- We can run super loop (infinite loop) in embedded C language.

We know that the programming in C language is solely done by dealing with different variables.

In case of Embedded C, these variables are nothing else but the memory locations of different memories of the microcontroller like code memory (ROM), data memory (RAM), external memory etc. To use these memory locations as variables, we need to use data types.

Data types

There are 7 different data types in embedded C for 8051…

1) unsigned char
   This data type is used to define an unsigned 8-bit variable. All 8-bits of this variable are used to specify data. Hence the range of this data type is $(0)_{10}\,to\,(255)_{10}$.
   e.g. unsigned char count;
2) signed char
   This data type is used to define a signed 8-bit variable. Here MSB of variable is used to show sign (+/-) while rest 7 bits are used to specify the magnitude of the variable. Hence the range of this data type is $(-128)_{10}\,to\,(127)_{10}$.
   e.g. signed char temp;
3) unsigned int
   This data type is used to define a 16-bit variable. Hence from this we can comment that this data types combines any 2 memory locations of the data memory as one variable. Here all 16 bits are used to specify data. So the range of this data type is $(0)_{10}\,to\,(65535)_{10}$.
4) signed int
   This data type is used to define a signed variable like *signed char* but of 16-bit size. Hence its range is $(-32768)_{10}\,to\,(32767)_{10}$
5) sfr
   This is an 8-bit data type used for defining names of Special Function Registers (SFR's) that are located in RAM memory locations 80 H to FF H only.
   e.g. sfr P0 = 0x80;

6) bit
   This data type is used to access single bits from the bit-addressable area of RAM.
   e.g. bit MYBIT = 0x32;
7) sbit
   The sbit data type is the one which is used to define or rather access single bits of the bit addressable SFR's of 8051 microcontroller.
   e.g. sbit En = P2^0;

With these data types in mind, let's take a look at the structure of a program in Embedded C.

DOCUMENTATION/COMMENTARY

Effective coding requires the use of documentation or commentary to indicate any important details of what the code is doing. An Embedded C program typically begins with some documentation information like the name of the file, the author, the date that the code was created, and any specific details about the functioning of the code. Embedded C supports single-line comments that begin with the characters "//" or multi-line comments that begin with "/*" and end with "*/" on a subsequent line.

Pre-processor Directives

Pre-processor directives are not normal code statements. They are lines of code that begin with the character "#" and appear in Embedded C programming before the main function. At runtime, the compiler looks for pre-processor directives within the code and resolves them completely before resolving any of the functions within the code itself. Some pre-processor directives can skip part of the main code based on certain conditions, while others may ask the pre-processor to replace the directive with information from a separate file before executing the main code or to behave differently based on the hardware resources available. Many types of pre-processor directives are available in the Embedded C language.

Global Variable Declaration

Global declarations happen before the main function of the source code. Engineers can declare global variables that may be called by the main program or any additional functions or sub-programs within the code. Engineers may also define functions here that will be accessible anywhere in the code.

Main Program

The main part of the program begins with **main( )**. If the main function is expected to return an integer value, we would write **int main( ).** If no return is expected, convention dictates that we should write **void main(void)**.

- **Declaration of local variables** - Unlike global variables, these ones can only be called by the function in which they are declared.
- **Initializing variables/devices** - A portion of code that includes instructions for initializing variables, I/O ports, devices, function registers, and anything else needed for the program to execute
- **Program body** - Includes the functions, structures, and operations needed to do something useful with our embedded system

Subprograms

An embedded C program file is not limited to a single function. Beyond the main( ) function, programmers can define additional functions that will execute following the main function when the code is compiled.

Decision control structures

The decision control structures are used to decide whether to execute a particular block of code depending on the condition specified. Following are some decision control structures:

- *if* statement
- *if…else* statement

if statement

```
if(condition)
{
statement-1;  statement-2;
……..
}
```

if…else statement

```
if(condition)
{
statement-1;  statement-2;
………
}
else
{
statement n;
}
```

Loop statements

The loop statements are the one which are used when we want to execute a certain block of code for more than one times either depending on situation or by a predefined number of times.

Embedded C is basically having two loop statements:

- ***for*** loop
- ***while*** loop

1) for loop

   for loops are used to repeat any particular piece of code a predefined number of times.

```
for(initializations ; conditions ; updates)
      {
        statement-1;
        statement-2;
        .........
      }
```

2) while loop

while loop also has the provision to repeat a certain block of code but here the block is repeated depending on the condition specified. The loop keeps on repeating until the condition becomes false.

Format of while loop is:

```
 while(condition)
{
statement-1;
statement- 2;
.........
}
```

Break & Continue Statements

1) break

The break statement, whenever is encountered in the loop, it forces the control to terminate the loop in which it is written.

2) **continue**

Whenever this statement is encountered in any loop, the statements in the loop after it won't be executed i.e. will be skipped and again control will be transferred to check the condition of the loop.

Format of any C Program

#include <reg51.h> ← ***Header File***

sbit *<name>=<bit address>;* ← ***sfr bit definitions***

sfr *<name>=<sfr address>;* ← ***sfr definition***

Data-type udf1(*data-type var_name);* ← ***User defined function***

Data-type udf2(*data-type var_name);*

void main(void) ← ***main function***

{

statement-1;

statement-2;

…………………;

Functions

Sometimes, there comes a situation in which in a program a group of statements is used frequently. Writing these statements again & again makes our program clumsy to write as well as it consumes more memory space. To overcome this problem there is a facility in C language to define a function. In function we can write the particular group of statements which is getting repeated continuously. Now anytime when we want to use that code group, we just have to call the function and it's done.

Types of functions

- No arguments, no return values
- With no arguments and a return value
- With arguments but no return value
- With arguments and return value

There are 3 ways to deal with a function:

- ❖ Define first, then use
- ❖ Do prototyping (i.e. Define first, use after main( ))
- ❖ Do prototyping in header file

a) **Define first, then use**
In this case, before writing the main function, we define the user-defined function and then use it in main( ) function whenever required.

b) **Do prototyping and define after main function**
In this case the function name, data type and argument data type are specified before writing main function to declare that we'll later implement this function.

c) **Do prototyping in header file**
In this case, define the function in a (user defined) header file and then just include that header file in your program.

Data Types in Embedded C

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
(signed) char	8-bit	-128 to +127
unsigned int	16-bit	0 to 65535
(signed) int	16-bit	-32768 to +32767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 – FFH only

Delay generation in 8051

The delay length in 8051 microcontroller depends on three factors:

- ❖ The crystal frequency
- ❖ the number of clock per machine
- ❖ the C compiler.

The original 8051 used 1/12 of the crystal oscillator frequency as one machine cycle. In other words, each machine cycle is equal to 12 clocks period of the crystal frequency connected to X1-X2 pins of 8051. To speed up the 8051, many recent versions of the 8051 have reduced the number of clocks per machine cycle from 12 to four, or even one. The frequency for the timer is always 1/12th the frequency of the crystal attached to the 8051, regardless of the 8051 version. In other words, AT89C51, DS5000, and DS89C4x0 the duration of the time to execute an instruction varies, but they all use 1/12th of the crystal's oscillator frequency for the clock source.

8051 has two different ways to generate time delay using C programming, regardless of 8051 version.

The **first method** is simply using ***Loop*** program function in which *Delay( )* function is made or by providing *for();* delay loop in Embedded C programming. You can define your own value of delay and how long you want to display. For example- *for(i=0;i<"any decimal value";i++);* this is the delay for loop used in embedded C.

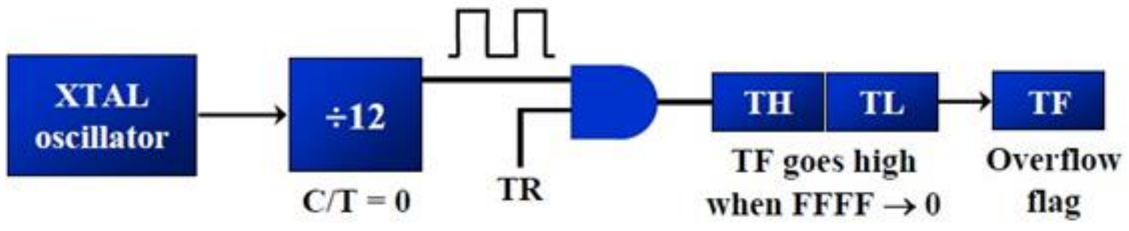
<u>Code to generate 250 ms delay on Port P1 of 8051:</u>

```
#include "REG52.h"

void MSDelay(unsigned int);

void main( )
{
  while (1) //repeat forever
{
   P1=0x55;
    MSDelay(250);
   P1=0xAA;
   MSDelay(250);
}
}
void MSDelay(unsigned int itime)
{
   unsigned int i,j;
   for (i=0;i<itime;i++)      // this is For( ); loop delay used to define delay value in
                                Embedded C
{
   for (j=0;j<1275;j++);
}
}
```

The **second method** is using Timer registers TH, TL and TMOD that are accessible in embedded C by defining header file ***reg52.h*** Both timers 0 and 1 use the same register, called TMOD (timer mode), to set the various timer operation modes in 8051 C programming. There are four operating modes of timer 0 and 1.

<u>To generate Time delay using timer registers:</u>

- 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)
- Load registers TL and TH with initial count value
- Start the timer
- Keep monitoring the timer flag (TF) until it rolls over from FFFFH to 0000.
- 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 TR is turned off by setting value to 0 and TF must be reloaded to 0.



Code generating delay using timer register:

```
#include <REG52.h>
void T0Delay(void);
void main(void){

while (1)
{
P1=0x55;
T0Delay( );
P1=0xAA;
T0Delay ( );
}
}

void T0Delay( )
{
TMOD=0x01;    // timer 0, mode 1
TL0=0x66;     // load TL0
TH0=0xFC;     // load TH0
TR0=1;        // turn on Timer0
while (TF0==0);        // wait for TF0 to roll over
TR0=0;        // turn off timer
TF0=0;        // clear TF0
}
```

Steps for generating precise Delay using 8051 Timers

In order to produce time delay accurately,

1. Divide the time delay with timer clock period.

   NNNN=time delay/1.085μs

2. Subtract the resultant value from 65536.

   MMMM=65536-NNNN

3. Convert the difference value to the hexa decimal form.

   MMMMd = XXYYh

4. Load this value to the timer register.

TH=XXh

TL=YYh

Delay Function to Generate 1 ms Delay
In order to generate a delay of 1ms, the calculations using above steps are as follows.

1. NNNN = 1ms/1.085µs ≈ 922.
2. MMMM = 65536-922 = 64614
3. 64614 in Hexadecimal = FC66h
4. Load TH with 0xFC and TL with 0x66

The following function will generate a delay of 1 ms using 8051 Timer 0.

```
Void delay ( )
 {
   TMOD = 0x01;    // Timer 0 Mode 1
   TH0= 0xFC;     //initial value for 1ms
   TL0 = 0x66;
   TR0 = 1;    // timer start
   while (TF0 == 0); // check overflow condition
   TR0 = 0;    // Stop Timer
   TF0 = 0;   // Clear flag
 }
```

Port programming

1. Write an 8051 C program to send values 00 – FF to port P1.

```
#include <reg51.h>
 void main(void)
{
unsigned char i;  for (i=0;i<=255;i++)
P1=i;
}
```

2. Write an 8051 C program to send the ASCII characters of 0, 1, 2, 3, 4, 5, A, B, C, and D to port P1

```
#include <reg51.h>
 void main(void)
{
```

```
unsigned char mynum( )="012345ABCD";
unsigned char i;
for (i=0;i<=10;i++)
 P1=mynum(i);
}
```

3. Write an 8051 C program to toggle all the bits of P1 continuously.

```
#include <reg51.h>
void main(void)
{
While (1)
{
p1=0x55;
p1=0xAA;
}
}
```

4. Write an 8051 C program to send values of –4 to +4 to port P1.

```
//Singed numbers
 #include <reg51.h>
void main(void)
{
char mynum[ ]={+1,-1,+2,-2,+3,-3,+4,-4};
unsigned char i;  for (i=0;i<=8;i++)
P1=mynum[i];
}
```

5. Write an 8051 C program to send values of –4 to +4 to port P1

```
//Singed numbers
 #include <reg51.h>
 void main(void)
{
char mynum[ ];
signed char i;
for (i=-4;i<=4;i++)
P1=mynum[i];
}
```

6. Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 50,000 times.

```
#include <reg51.h>
sbit MYBIT=P1^0;
void main(void)
```

```
{
unsigned int z;
for (z=0;z<=50000;z++)
{
MYBIT=0;
MYBIT=1;
}
}
```

Note: sbit keyword allows access to the single bits of the SFR registers

7. LEDs are connected to bits P1 and P2. Write an 8051 C program that shows the count from 0 to FFH (0000 0000 to 1111 1111 in binary) on the LEDs.

```
#include <reg51.h>
#define LED P2;
void main(void)
{
P1=00; //clear P1
 LED=0; //clear P2
while(1)
{
P1++; //increment P1
 LED++; //increment P2
}
}
```

Note: Ports P0 – P3 are byte-accessable and we can use the P0 – P3 labels as defined in the 8051 header file <reg51.h>

8. Write an 8051 C program to get a byte of data form P1, wait 1/2 second, and then send it to P2.

```
#include <reg51.h>
void MSDelay(unsigned int);
void main(void)
{
unsigned char mybyte;
P1=0xFF; //make P1 input port
while (1)
{
mybyte=P1; //get a byte from P1
MSDelay(500);
P2=mybyte; //send it to P2
}
}
```

```
void MSDelay(unsigned int itime)
        {
        unsigned int i,j;
        for (i=0;i<itime;i++)
        for (j=0;j<1275;j++);
        }
```

9. Write an 8051 C program to get a byte of data form P0. If it is less than 100, send it to P1; otherwise, send it to P2.

```
#include <reg51.h>
void main(void)
{
unsigned char mybyte;
P0=0xFF; //make P0 input port
while (1)
{
mybyte=P0; //get a byte from P0
if (mybyte<100)
P1=mybyte; //send it to P1
else
P2=mybyte; //send it to P2
}
}
```

```
void MSDelay(unsigned int itime)
        {
        unsigned int i,j;
        for (i=0;i<itime;i++)  for
        (j=0;j<1275;j++);
        }
```

10. Write an 8051 C program to toggle only bit P2.4 continuously without disturbing the rest of the bits of P2

```
//Toggling an individual bit
#include <reg51.h>
sbit mybit=P2^4;
void main(void)
{
while (1)
{
mybit=1; //turn on P2.4
mybit=0; //turn off P2.4
}
}
```

Note:

- ❖ Ports P0 – P3 are bit-addressable and we use sbit data type to access a single bit of P0 - P3
- ❖ Use the Px^y format, where x is the port 0, 1, 2, or 3 and y is the bit 0 – 7 of that port

11. Write an 8051 C program to monitor bit P1.5. If it is high, send 55H to P0; otherwise, send AAH to P2

```
#include <reg51.h>
sbit mybit=P1^5;
 void main(void)
{
mybit=1; //make mybit an input
while (1)
{
        if (mybit==1)
```

```
                P0=0x55;
        else
                P2=0xAA;
        }
        }
```

12. A door sensor is connected to the P1.1 pin, and a buzzer is connected to P1.7. Write an 8051 C program to monitor the door sensor, and when it opens, sound the buzzer. You can sound the buzzer by sending a square wave of a few hundred Hz.

```
#include <reg51.h>
void MSDelay(unsigned int);
sbit Dsensor=P1^1;
sbit Buzzer=P1^7;
void main(void)
{
Dsensor=1; //make P1.1 an input
while (1)
{
while (Dsensor==1)//while it opens
{
Buzzer=0;  MSDelay(200);
 Buzzer=1;  MSDelay(200);
}
}
}
```

```
void MSDelay(unsigned int itime)

{

unsigned int i,j;

for (i=0;i<itime;i++)  for
(j=0;j<1275;j++);

}
```

13. Write an 8051 C program to toggle all the bits of P0, P1, and P2 continuously with a 250 ms delay. Use the sfr keyword to declare the port addresses

```
sfr P0=0x80;
sfr P1=0x90;
sfr P2=0xA0;
void MSDelay(unsigned int);
void main(void)
{
        while (1)
        {
                P0=0x55;
                P1=0x55;
                P2=0x55;
                MSDelay(250);
                P0=0xAA;
```

```
void MSDelay(unsigned int itime)
        {
        unsigned int i,j;
        for (i=0;i<itime;i++)  for
        (j=0;j<1275;j++);
        }
```

```
                P1=0xAA;
                P2=0xAA;
                MSDelay(250);
        }
}
```

14. The data pins of an LCD are connected to P1. The information is latched into the LCD whenever its Enable pin goes from high to low. Write an 8051 C program to send "ECED-JCET" to this LCD

```
#include <reg51.h>
#define LCDData P1 //LCDData declaration
sbit En=P2^0; //the enable pin  void main(void)
{
        unsigned char message[ ] ="ECED-JCET";
        unsigned char z;
        for (z=0;z<9;z++)      //send 9 characters
        {
        LCDData=message[z];
        En=1; //a high-
        En=0; //-to-low pulse to latch data
        }
}
```

15. Write an 8051 C program to turn bit P1.5 on and off 50,000 times.

```
#include <reg51.h>
sbit MYBIT=0x95;
void main(void)
{
        unsigned int z;
        for (z=0;z<50000;z++)
        {  MYBIT=1;  MYBIT=0;
        }
}
```

Note

- We can access a single bit of any SFR if we specify the bit address

16. Generate a square wave with ON time 3ms and OFF time 5ms at port 0.Assume crystal frequency 11.059MHz .

```
#include <reg51.h>
sbit wave =P0^0;
void MSdelay (unsigned int );
void main(void)
{
wave =1;
MSdelay(3);
wave =0;
MSdelay (5);
}
```

```
void MSDelay(unsigned int itime)
        {
        unsigned int i,j;
        for (i=0;i<itime;i++)  for
        (j=0;j<1275;j++);
        }
```

17. Generate a square wave with ON time 3ms and OFF time 5ms at port 0.Assume crystal frequency 22MHz , Timer 0 in mode 1.

```
#include <reg51.h>
sbit wave =P0^0;
void delay3 ( );
void delay5 ( );
void main(void)
{
wave =1;
delay(3);
wave =0;
delay (5);
}
```

```
Void delay3 ( )
 {
  TMOD = 0x01;   // Timer 0 Mode 1
  TH0= 0xEA;   //initial value for 1ms
  TL0 = 0x8A;
  TR0 = 1;    // timer start
  while (TF0 == 0); // check overflow condition
  TR0 = 0;   // Stop Timer
  TF0 = 0;  // Clear flag
 }
Void delay5 ( )
 {
  TMOD = 0x01;   // Timer 0 Mode 1
  TH0= 0xDC;   //initial value for 1ms
  TL0 = 0x3B;
  TR0 = 1;    // timer start
  while (TF0 == 0); // check overflow condition
  TR0 = 0;   // Stop Timer
  TF0 = 0;  // Clear flag
 }
```

Code Conversion Programs

1. **Write an 8051 C program to convert packed BCD 0x29 to ASCII and display the bytes on P1 and P2.**

```
#include <reg51.h>
void main(void)
{
unsigned char x,y,z;
unsigned char mybyte=0x29;
x=mybyte&0x0F;
P1=x|0x30;
y=mybyte&0xF0;
y=y>>4;
P2=y|0x30;
}
```

2. Write an 8051 C program to convert ASCII digits of '4' and '7' to packed BCD and display them on P1.

```
#include <reg51.h>
void main(void)
{
unsigned char bcdbyte;
 unsigned char w='4';
 unsigned char z='7';
w=w&0x0F;
w=w<<4;
z=z&0x0F;
bcdbyte=w|z;
P1=bcdbyte;
}
```

3. Write an 8051 C program to calculate the checksum byte for the data 25H, 62H, 3FH, and 52H.

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[ ]={0x25,0x62,0x3F,0x52};
 unsigned char sum=0;
unsigned char x;
unsigned char chksumbyte;  for (x=0;x<4;x++)
{
P2=mydata[x];
sum=sum+mydata[x];
}
chksumbyte=~sum+1;
P2=chksumbyte;
}
```

4. Write an 8051 C program to perform the checksum operation to ensure data integrity. If data is good, send ASCII character 'G' to P0. Otherwise send 'B' to P0.

```
#include <reg51.h>
void main(void)
{
unsigned char mydata[ ]={0x25,0x62,0x3F,0x52,0xE8};
unsigned char chksum=0;
unsigned char x;
for (x=0;x<5;x++)  chksum=chksum + mydata[x];
if (chksum==0)
P0='G';
else
P0='B';
}
```

5. **Write an 8051 C program to convert 11111101 (FD hex) to decimal and display the digits on P0, P1 and P2.**

```
#include <reg51.h>
void main(void)
{
unsigned char x,binbyte,d1,d2,d3;
binbyte=0xFD;
 x=binbyte/10;
d1=binbyte%10;
d2=x%10;
d3=x/10;
P0=d1;
P1=d2;
P2=d3;
}
```

INTERFACING THE KEYBOARD TO 8051 MICROCONTROLLER

The key board here we are interfacing is a matrix keyboard. This key board is designed with a particular rows and columns. These rows and columns are connected to the microcontroller through its ports of the micro controller 8051. We normally use 8*8 matrix key board. So only two ports of 8051 can be easily connected to the rows and columns of the key board.

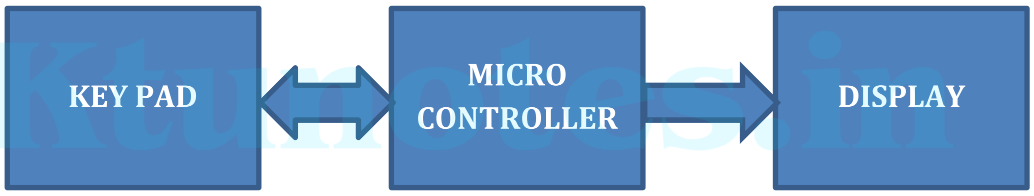
When ever a key is pressed, a row and a column gets shorted through that pressed key and all the other keys are left open. When a key is pressed only a bit in the port goes high. Which indicates microcontroller that the key is pressed. By this high on the bit key in the corresponding column is identified.

Once we are sure that one of key in the key board is pressed next our aim is to identify that key. To do this we firstly check for particular row and then we check the corresponding column the key board.

To check the row of the pressed key in the keyboard, one of the row is made high by making one of bit in the output port of 8051 high . This is done until the row is found out. Once we get the row next out job is to find out the column of the pressed key. The column is detected by contents in the input ports with the help of a counter. The content of the input port is rotated with carry until the carry bit is set.

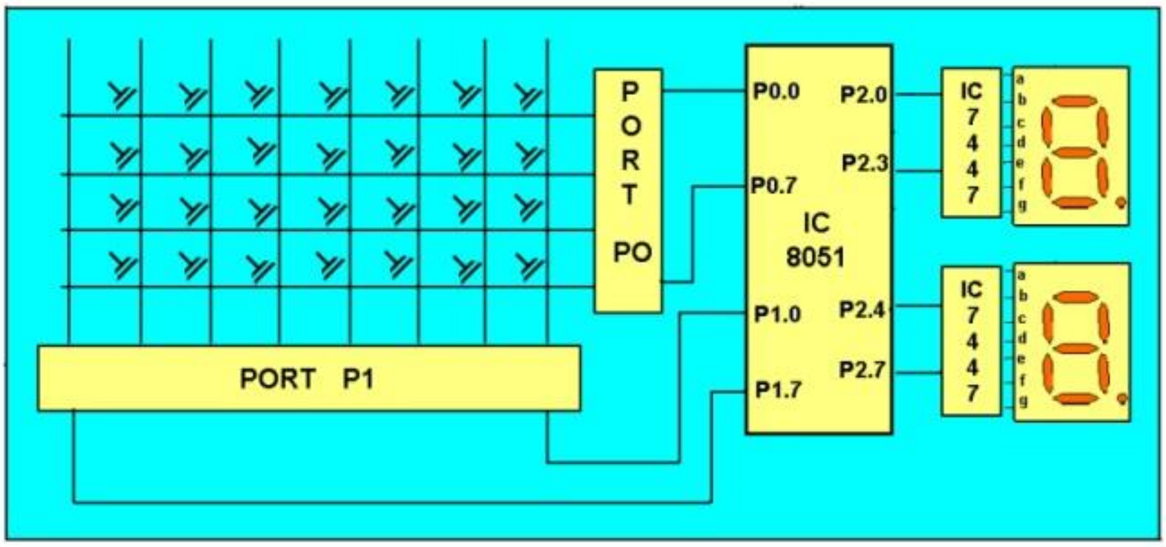
The contents of the counter is then compared and displayed in the display. This display is designed using a seven segment display and a BCD to seven segment decoder IC 7447.

The BCD equivalent number of counter is sent through output part of 8051 displays the number of pressed key.



Circuit diagram of INTERFACING KEY BOARD TO 8051.

The programming algorithm, program and the circuit diagram is as follows. Here program is explained with comments.



- The 8051 has 4 I/O ports P0 to P3 each with 8 I/O pins, P0.0 to P0.7,P1.0 to P1.7, P2.0 to P2.7, P3.0 to P3.7. The one of the port P1 (it understood that P1 means P1.0 to P1.7) as an I/P port for microcontroller 8051, port P0 as an O/P port of microcontroller 8051 and port P2 is used for displaying the number of pressed key.
- Make all rows of port P0 high so that it gives high signal when key is pressed.
- See if any key is pressed by scanning the port P1 by checking all columns for non zero condition.
- If any key is pressed, to identify which key is pressed make one row high at a time.
- Initiate a counter to hold the count so that each key is counted.
- Check port P1 for nonzero condition. If any nonzero number is there in [accumulator], start column scanning by following step 9.
- Otherwise make next row high in port P1.
- Add a count of 08h to the counter to move to the next row by repeating steps from step 6.
- If any key pressed is found, the [accumulator] content is rotated right through the carry until carry bit sets, while doing this increment the count in the counter till carry is found.
- Move the content in the counter to display in data field or to memory location
- To repeat the procedures go to step 2.

Start of main program:

to check that whether any key is pressed

```
start:  mov a,#00h
        mov p1,a          ;making all rows of port p1 zero
        mov a,#0fh
        mov p1,a          ;making all rows of port p1 high
press:  mov a,p2
        jz press          ;check until any key is pressed
```

after making sure that any key is pressed

```
            mov a,#01h      ;make one row high at a time
            mov r4,a
            mov r3,#00h     ;initiating counter
      next:  mov a,r4
            mov p1,a        ;making one row high at a time
            mov a,p2        ;taking input from port A
            jnz colscan     ;after getting the row jump to check
                            column
            mov a,r4
            rl a            ;rotate left to check next row
            mov r4,a
            mov a,r3
            add a,#08h      ;increment counter by 08 count
            mov r3,a
            sjmp next       ;jump to check next row
```

after identifying the row to check the column following steps are followed

```
   colscan:  mov r5,#00h
       in:  rrc a          ;rotate right with carry until get the carry
            jc out         ;jump on getting carry
            inc r3         ;increment one count
            jmp in
      out:  mov a,r3
            da a           ;decimal adjust the contents of counter
                             before display
            mov p2,a
            jmp start      ;repeat for check next key.
```

INTERFACING DAC TO 8051

The Digital to Analog converter (DAC) is a device, that is widely used for converting digital pulses to analog signals. There are two methods of converting digital signals to analog signals. These two methods are binary weighted method and R/2R ladder method. In this article we will use the MC1408 (DAC0808) Digital to Analog Converter. This chip uses R/2R ladder method. This method can achieve a much higher degree of precision. DACs are judged by its resolution. The resolution is a function of the number of binary inputs. The most common input counts are 8, 10, 12 etc. Number of data inputs decides the resolution of DAC. So if there are n digital input pin, there are 2^n analog levels. So 8 input DAC has 256 discrete voltage levels.

The MC1408 DAC (or DAC0808)

In this chip the digital inputs are converted to current. The output current is known as I_{out} by connecting a resistor to the output to convert into voltage. The total current provided by the I_{out} pin is basically a function of the binary numbers at the input pins D_0 - D_7 (D_0 is the LSB and D_7 is the MSB) of DAC0808 and the reference current I_{ref}. The following formula is showing the function of I_{out}

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

The I_{ref} is the input current. This must be provided into the pin 14. Generally 2.0mA is used as I_{ref}

We connect the I_{out} pin to the resistor to convert the current to voltage. But in real life it may cause inaccuracy since the input resistance of the load will also affect the output voltage. So practically I_{ref} current input is isolated by connecting it to an Op-Amp with $R_f = 5K\Omega$ as feedback resistor. The feedback resistor value can be changed as per requirement.

Generating Sinewave using DAC and 8051 Microcontroller

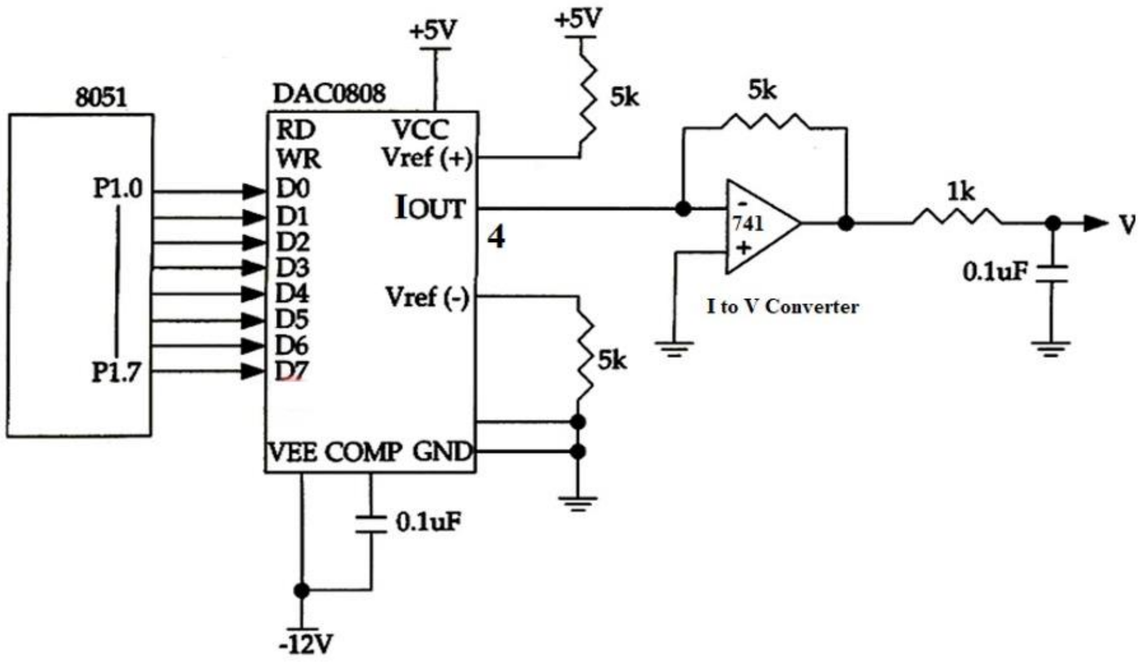
For generating sinewave, at first we need a look-up table to represent the magnitude of the sine value of angles between 0° to 360°. The sine function varies from -1 to +1. In the table only integer values are applicable for DAC input. In this example we will consider 30° increments and calculate the values from degree to DAC input. We are assuming full-scale voltage of 10V for DAC output. We can follow this formula to get the voltage ranges.

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

Let us see the lookup table according to the angle and other parameters for DAC.

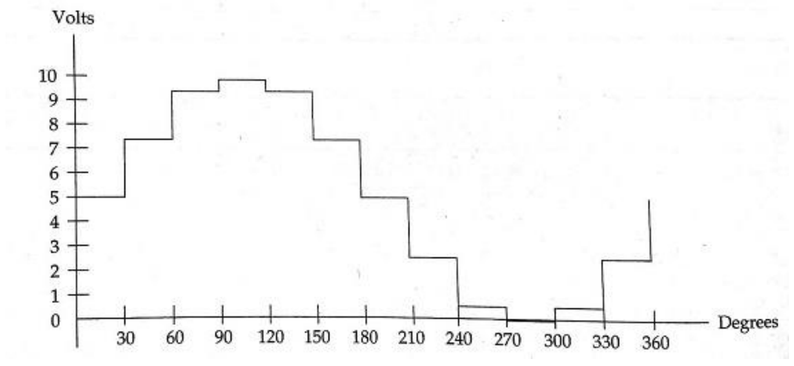
Angle(in θ)	**sinθ**	**V_{out} (Voltage Magnitude)**	**Values sent to DAC (Vout* 25.6)**
0	0	5	128
30	0.5	7.5	192
60	0.866	9.33	238
90	1.0	10	255
120	0.866	9.33	238
150	0.5	7.5	192
180	0	5	128
210	-0.5	2.5	64
240	-0.866	0.669	17
270	-1.0	0	0
300	-0.866	0.669	17
330	-0.5	2.5	64
360	0	5	128

Circuit Diagram –



Program

```
#include<reg51.h>
sfr DAC = 0x80;  //Port P0 address
void main(){
  int sin_value[12] = {128,192,238,255,238,192,128,64,17,0,17,64};
  int i;
  while(1){
    //infinite loop for LED blinking
    for(i = 0; i<12; i++){
      DAC = sin_value[i];
    }
  }
}
```



INTERFACING ADC TO 8051

An analog to digital converter or ADC, as the name suggests, converts an analog signal to a digital signal. An analog signal has a continuously changing amplitude with respect to time. A digital signal, on the contrary, is a stream of 0s and 1s. An ADC maps analog signals to their binary equivalents. To do this, ADCs use various methods like Flash conversion, slope integration, or successive approximation.
To understand the ADC in a better way, let us look at an example. Let us say we have an input signal which varies from 0 to 8 volt, and we use a 3-bit ADC to convert this signal to binary data. A 3-bit ADC can represent 2^3 or 8 different voltage levels using 3 bits of data. How convenient! In this case, the ADC maps the data in the following manner.

Input voltage	Binary equivalent
0-1 volt	000B
1-2 volt	001B
2-3volt	010B
3-4 volt	011B
4-5 volt	100B
5-6 volt	101B
6-7 volt	110B
7-8 volt	111B

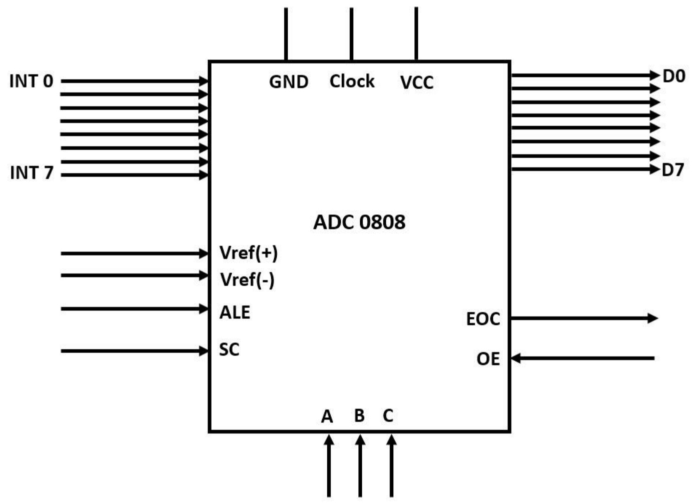
If you look at the table above, you will understand how the ADC maps analog data to digital values. In the case mentioned above, we can see that the tiniest change we can detect is that of 1 volt. If the change is smaller than 1 volt, the ADC can't detect it. This minimum change that an ADC can detect is known as the **step size of the ADC**. To calculate it, we can use the formula:

Step size=(Vmax-Vmin)/2^n (where n is the number of bits(resolution) of an ADC)
The step size of an ADC is inversely proportional to the number of bits of an ADC. So using an ADC with higher bits can detect smaller changes, but this increases the cost of production. Due to this reason, most on-chip ADCs' have an 8-bit/10-bit resolution. Given below is the resolution vs. step size for various configurations with a range of 0-5v input signal.

Number of bits	Number of steps	step size(mV)
8	256	5/256=19.53
10	1024	5/1024=4.88
12	4096	5/4096=1.2
16	65536	0.076 (precise conversion)

ADC 0808

The ADC 0808 is a popular 8-bit ADC with a step size of 19.53 millivolts. It does not have an internal clock. Therefore, it requires a clock signal from an external source. It has eight input pins, but only one of them can be selected at a time because it has eight digital output pins. It uses the principle of successive approximation for calculating digital values, which is very accurate for performing 8-bit analog to digital conversions. Let us look at the pin description to get more insights into ADC 0808.



Input pins (INT0-INT7)

The ADC 0808 has eight input analog pins. These pins are multiplexed together, and only one of them can be selected using three select lines.

Select lines and ALE

It has three select lines, namely A, B, and C, that are used to select the desired input lines. The ALE pin also needs to be activated by a low to high pulse to select a particular input. The input lines are selected as follows:

A	B	C	Selected analog channel	ALE pin
0	0	0	INT0	Low to High pulse
0	0	1	INT1	Low to High pulse
0	1	0	INT2	Low to High pulse
0	1	1	INT3	Low to High pulse
1	0	0	INT4	Low to High pulse
1	0	1	INT5	Low to High pulse
1	1	0	INT6	Low to High pulse
1	1	1	INT7	Low to High pulse

Output pins (D0-D7)

The ADC has eight output pins that give the binary equivalent of a given analog value.

VCC and Ground

These two pins are used to provide the required voltage to power the microcontroller. In most cases, the ADC uses 5V DC to power up.

Clock

As mentioned earlier, the 0808 does not have an internal clock and needs an external clock signal to operate. It uses a clock frequency of 20Mhz, and using this clock frequency it can perform one conversion in 100 microseconds.

VREF (+) and VREF (-)

These two pins are used to provide the upper and the lower limit of voltages which determine the step size for the conversion. Here Vref(+) has a higher voltage, and Vref(-) has the lower voltage. If Vref(+) has an input voltage 5v and Vref(-) has a voltage of 0v then the step size will be 5v-0v/2^8= 15.53 mv.

Start conversion

This pin is used to tell the ADC to start the conversion. When the ADC receives a low to high pulse on this pin, it starts converting the analog voltage on the selected pin to its 8-bit digital equivalent.

End of conversion

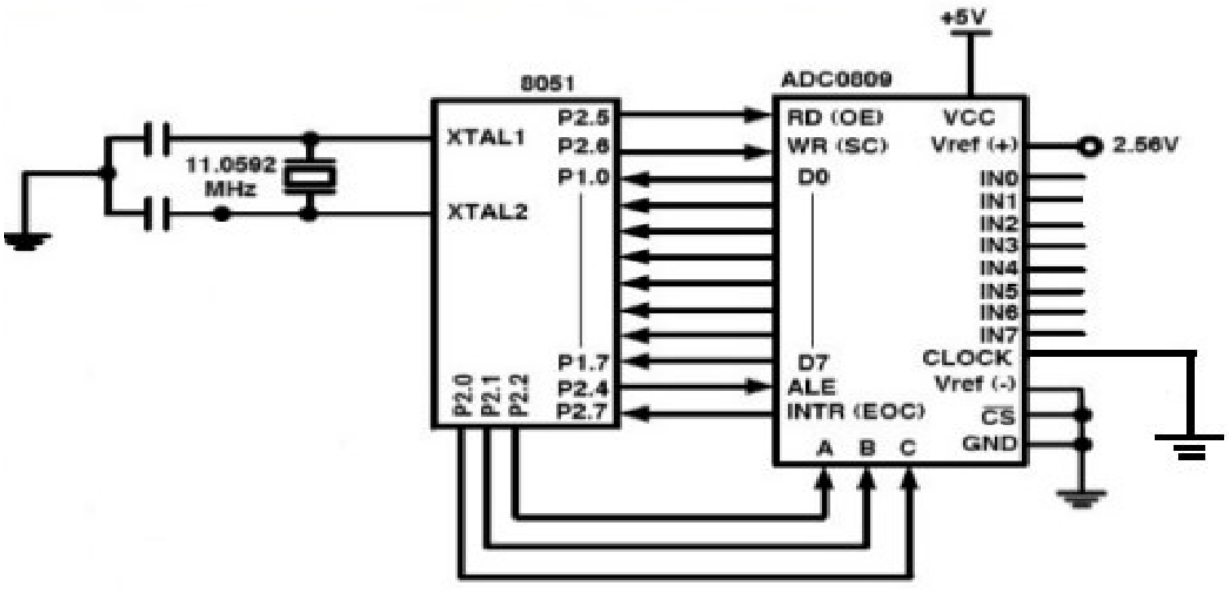
Once the conversion is complete, the ADC sends low to high signal to tell a microcontroller that the conversion is complete and that it can extract the data from the 8 data pins.

Output enable

This pin is used to extract the data from the ADC. A microcontroller sends a low to high pulse to the ADC to extract the data from its data buffers

Interfacing 8051 with 0808

Most modern microcontrollers with 8051 IP cores have an inbuilt ADC. Older versions of 8051 like the MCS-51 and A789C51 do not have an on-chip ADC. Therefore to connect these microcontrollers to analog sensors like temperature sensors, the microcontroller needs to be hooked to an ADC. It converts the analog values to digital values, which the microcontroller can process and understand. Here is how we can interface the 8051 with 0808.



To interface the ADC to 8051, follow these steps.

- Connect the oscillator circuit to pins 19 and 20. This includes a crystal oscillator and two capacitors of 22uF each. Connect them to the pins, as shown in the diagram.
- Connect one end of the capacitor to the EA' pin and the other to the resister. Connect this resistor to the RST pin, as shown in the diagram.
- We are using port 1 as the input port, so we have connected the output ports of the ADC to port 1.
- As mentioned earlier, the 0808 does not have an internal clock; therefore, we have to connect an external clock. Connect the external clock to pin 10.
- Connect Vref (+) to a voltage source according to the step size you need.
- Ground Vref (-) and connect the analog sensor to any one of the analog input pins on the ADC. We have connected a variable resistor to INT2 for getting a variable voltage at the pin.
- Connect ADD A, ADD B, ADD C, and ALE pins to the microcontroller for selecting the input analog port. We have connected ADD A- P2.0; ADD B- P2.1; ADD C- P2.2 and the ALE pin to port 2.4.
- Connect the control pins Start, OE, and Start to the microcontroller. These pins are connected as follows in our case Start-Port-2.6; OE-Port-2.5 and EOC-Port-2.7.

Logic to communicate between 8051 and ADC 0808

Several control signals need to be sent to the ADC to extract the required data from it.

- **Step 1:** Set the port you connected to the output lines of the ADC as an input port. You can learn more about the Ports in 8051 here.
- **Step 2:** Make the Port connected to EOC pin high. The reason for doing this is that the ADC sends a high to low signal when the conversion of data is complete. So this line needs to be high so that the microcontroller can detect the change.
- **Step 3:** Clear the data lines which are connected to pins ALE, START, and OE as all these pins require a Low to High pulse to get activated.
- **Step 4:** Select the data lines according to the input port you want to select. To do this, select the data lines and send a High to Low pulse at the ALE pin to select the address.
- **Step 5:** Now that we have selected the analog input pin, we can tell the ADC to start the conversion by sending a pulse to the START pin.
- **Step 6:** Wait for the High to low signal by polling the EOC pin.
- **Step 7:** Wait for the signal to get high again.
- **Step 8:** Extract the converted data by sending a High to low signal to the OE pin.

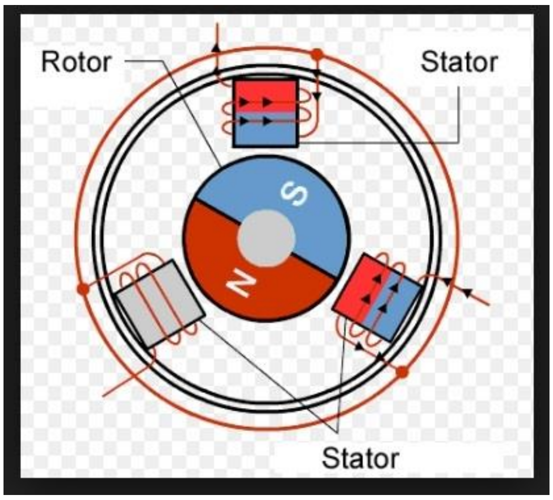
Program

```
#include <reg51.h>
sbit ALE = P2^4;
sbit OE = P2^5;
sbit SC = P2^6;
sbit EOC = P2^7;
sbit ADDR_A = P2^0;
sbit ADDR_B = P2^1;
sbit ADDR_C = P2^2;
sfr MYDATA =P1;
sfr SENDDATA =P3;
```

```
void MSDelay(unsighned int) // Function to generate time delay
{
unsighned int i,j;
for(i=0;i<delay;i++)
for(j=0;j<1275;j++);
}
void main()
{
unsigned char value;
MYDATA = 0xFF;
EOC = 1;
ALE = 0;
OE = 0;
SC = 0;
while(1)
{
ADDR_C = 0;
ADDR_B = 0;
ADDR_A = 0;
MSDelay(1);
ALE = 1;
MSDelay(1);
SC = 1;
MSDelay(1);
ALE = 0;
SC = 0;
while(EOC==1);
while(EOC==0);
OE=1;
MSDelay(1);
value = MYDATA;
SENDDATA = value;
OE = 0 ;
}
}
```

STEPPER MOTOR INTERFACING WITH 8051

Stepper motors are used to translate electrical pulses into mechanical movements. In some disk drives, dot matrix printers, and some other different places the stepper motors are used. The main advantage of using the stepper motor is the position control. Stepper motors generally have a permanent magnet shaft (rotor), and it is surrounded by a stator.



Normal motor shafts can move freely but the stepper motor shafts move in fixed repeatable increments.

Some parameters of stepper motors –

- **Step Angle** – The step angle is the angle in which the rotor moves when one pulse is applied as an input of the stator. This parameter is used to determine the positioning of a stepper motor.
- **Steps per Revolution** – This is the number of step angles required for a complete revolution. So the formula is 360° /Step Angle.
- **Steps per Second** – This parameter is used to measure a number of steps covered in each second.
- **RPM** – The RPM is the Revolution Per Minute. It measures the frequency of rotation. By this parameter, we can measure the number of rotations in one minute.

Interfacing Stepper Motor with 8051 Microcontroller

Weare using Port P0 of 8051 for connecting the stepper motor. HereULN2003 is used. This is basically a high voltage, high current Darlington transistor array. Each ULN2003 has seven NPN Darlington pairs. It can provide high voltage output with common cathode clamp diodes for switching inductive loads.

The Unipolar stepper motor works in three modes.

- **Wave Drive Mode** – In this mode, one coil is energized at a time. So all four coils are energized one after another. This mode produces less torque than full step drive mode.

The following table is showing the sequence of input states in different windings.

Steps	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0

Steps	Winding A	Winding B	Winding C	Winding D
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

- ❖ **Full Drive Mode –** In this mode, two coils are energized at the same time. This mode produces more torque. Here the power consumption is also high

The following table is showing the sequence of input states in different windings.

Steps	Winding A	Winding B	Winding C	Winding D
1	1	1	0	0
2	0	1	1	0
3	0	0	1	1
4	1	0	0	1

- ❖ **Half Drive Mode –** In this mode, one and two coils are energized alternately. At first, one coil is energized then two coils are energized. This is basically a combination of wave and full drive mode. It increases the angular rotation of the motor

The following table is showing the sequence of input states in different windings.

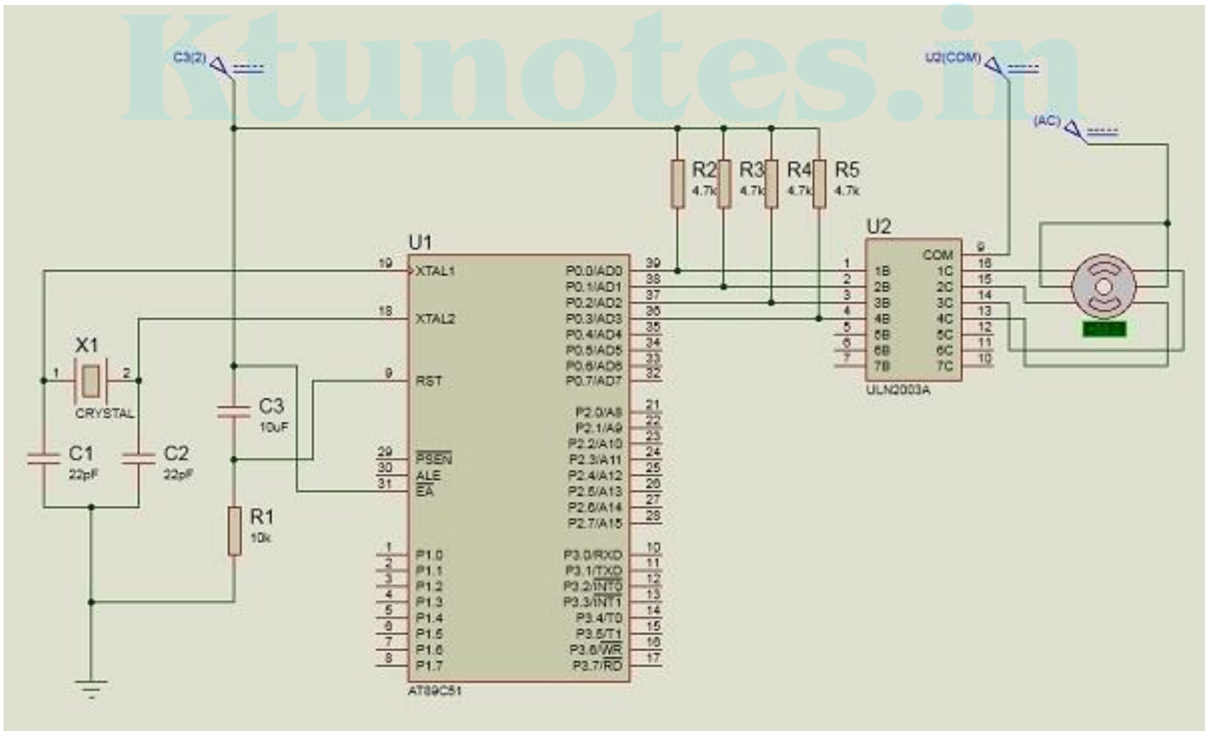
Steps	Winding A	Winding B	Winding C	Winding D
1	1	0	0	0
2	1	1	0	0
3	0	1	0	0
4	0	1	1	0
5	0	0	1	0
6	0	0	1	1
7	0	0	0	1
8	1	0	0	1

Program

```
// Wave drive Mode
#include<reg51.h>
void ms_delay(unsigned int t)      //To create a delay of 200 ms = 200 x 1ms
   {
   unsigned i,j ;
   for(i=0;i<t;i++)      //200 times 1 ms delay
   for(j=0;j<1275;j++);   //1ms delay
   }
void main()
{
   while(1) // To repeat infinitely
   {
      P2=0x08;         //P2 = 0000 1000 First Step
      ms_delay(200);
      P2=0x04;         //P2 = 0000 0100 Second Step
      ms_delay(200);
      P2=0x02;         //P2 = 0000 0010 Third Step
      ms_delay(200);
      P2=0x01;         //P2 = 0000 0001 Fourth Step
      ms_delay(200);
   }
}
// Full drive Mode
#include<reg51.h>
void ms_delay(unsigned int t) //To create a delay of 200 ms = 200 x 1ms
   {
   unsigned i,j ;
   for(i=0;i<t;i++)      //200 times 1 ms delay
   for(j=0;j<1275;j++);   //1ms delay
   }
void main()
{
   while(1) // To repeat infinitely
   {
      P2=0x0C;         //P2 = 0000 1000 First Step
      ms_delay(200);
      P2=0x06;         //P2 = 0000 0100 Second Step
      ms_delay(200);
      P2=0x03;         //P2 = 0000 0010 Third Step
      ms_delay(200);
      P2=0x09;         //P2 = 0000 0001 Fourth Step
      ms_delay(200);
   }
}

// Half Drive Mode
#include<reg51.h>
void ms_delay(unsigned int t)  //To create a delay of 200 ms = 200 x 1ms
   {
   unsigned i,j ;
   for(i=0;i<t;i++)
   for(j=0;j<1275;j++);
```

```
  }
void main()
{
  while (1)
  {
    P2 = 0x08;      //P2 = 0000 1000 First Step
    ms_delay(200)
    P2 = 0x0C;      //P2 = 0000 1100 Second Step
    ms_delay(200)
    P2 = 0x04;      //P2 = 0000 0100 Third Step
    ms_delay(200)
    P2 = 0x06;      //P2 = 0000 0110 Fourth Step
    ms_delay(200)
    P2 = 0x02;      //P2 = 0000 0010 Fifth Step
    ms_delay(200);
    P2 = 0x03;      //P2 = 0000 0011 Sixth Step
    ms_delay(200);
    P2 = 0x01;      //P2 = 0000 0001 Seventh Step
    ms_delay(200);
    P2 = 0x09;      //P2 = 0000 1001 Eight Step
    ms_delay(200);
  }
}
```

The circuit diagram is shown below: It uses the full drive mode.



LCD INTERFACING WITH 8051 MICROCONTROLLER

Display units are the most important output devices in embedded projects and electronics products. 16x2 LCD is one of the most used display unit. 16x2 LCD means that there are two rows in which 16 characters can be displayed per line, and each character takes 5X7 matrix space on LCD. In this tutorial we are going to connect 16X2 LCD module to the 8051 microcontroller (AT89S52). **Interfacing LCD with 8051 microcontroller** might look quite complex to newbies, but after understanding the concept it would look very simple and easy. Although it may be time taking because you need to understand and connect 16 pins of LCD to the microcontroller. So first let's understand the 16 pins of LCD module.

We can divide it in five categories, Power Pins, contrast pin, Control Pins, Data pins and Backlight pins.

Category	**Pin NO.**	**Pin Name**	**Function**
Power Pins	1	VSS	Ground Pin, connected to Ground
	2	VDD or Vcc	Voltage Pin +5V
Contrast Pin	3	V0 or VEE	Contrast Setting, connected to Vcc thorough a variable resistor.
Control Pins	4	RS	Register Select Pin, RS=0 Command mode, RS=1 Data mode
	5	RW	Read/ Write pin, RW=0 Write mode, RW=1 Read mode
	6	E	Enable, a high to low pulse need to enable the LCD
Data Pins	7-14	D0-D7	Data Pins, Stores the Data to be displayed on LCD or the command instructions

Backlight Pins	15	LED+ or A	To power the Backlight +5V
	16	LED- or K	Backlight Ground

All the pins are clearly understandable by their name and functions, except the control pins, so they are explained below:

RS: RS is the register select pin. We need to set it to 1, if we are sending some data to be displayed on LCD. And we will set it to 0 if we are sending some command instruction like clear the screen (hex code 01).

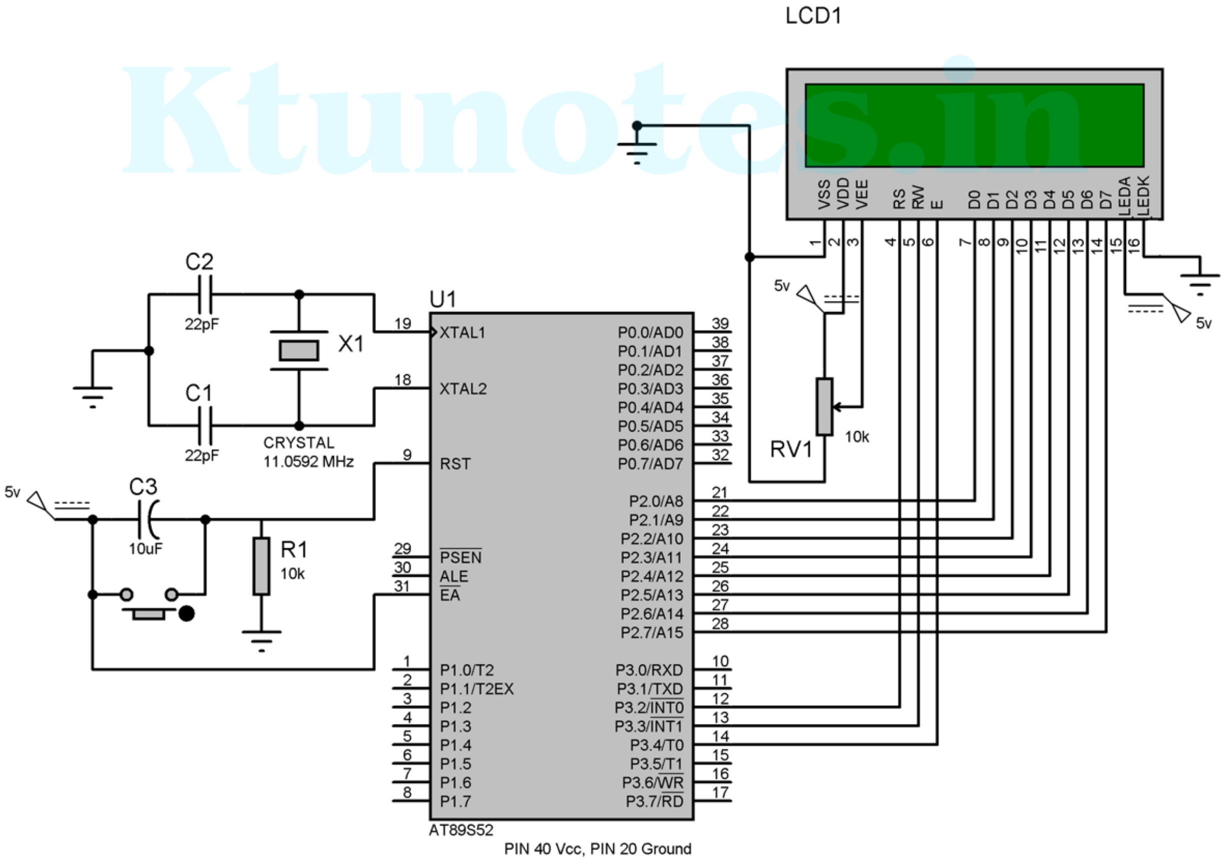
RW: This is Read/write pin, we will set it to 0, if we are going to write some data on LCD. And set it to 1, if we are reading from LCD module. Generally this is set to 0, because we do not have need to read data from LCD. Only one instruction “Get LCD status”, need to be read some times.

E: This pin is used to enable the module when a high to low pulse is given to it. A pulse of 450 ns should be given. That transition from HIGH to LOW makes the module ENABLE.

There are some preset command instructions in LCD, we have used them in our program below to prepare the LCD (in lcd_init() function). Some important command instructions are given below:

Hex Code	Command to LCD Instruction Register
0F	LCD ON, cursor ON
01	Clear display screen
02	Return home
04	Decrement cursor (shift cursor to left)
06	Increment cursor (shift cursor to right)
05	Shift display right
07	Shift display left
0E	Display ON, cursor blinking

80	Force cursor to beginning of first line
C0	Force cursor to beginning of second line
38	2 lines and 5×7 matrix
83	Cursor line 1 position 3
3C	Activate second line
08	Display OFF, cursor OFF
C1	Jump to second line, position 1
OC	Display ON, cursor OFF
C1	Jump to second line, position 1
C2	Jump to second line, position 2



Circuit diagram for **LCD interfacing with 8051 microcontroller** is shown in the above figure. If you have basic understanding of 8051 then you must know about EA(PIN 31), XTAL1 & XTAL2, RST pin(PIN 9), Vcc and Ground Pin of 8051 microcontroller. I have used these Pins in above circuit.

So besides these above pins we have connected the data pins (D0-D7) of LCD to the Port 2 (P2_0 – P2_7) microcontroller. And control pins RS, RW and E to the pin 12,13,14 (pin 2,3,4 of port 3) of microcontroller respectively.

PIN 2(VDD) and PIN 15(Backlight supply) of LCD are connected to voltage (5v), and PIN 1 (VSS) and PIN 16(Backlight ground) are connected to ground.

Pin 3(V0) is connected to voltage (Vcc) through a variable resistor of 10k to adjust the contrast of LCD. Middle leg of the variable resistor is connected to PIN 3 and other two legs are connected to voltage supply and Ground.

Program

```
// Program for LCD Interfacing with 8051 Microcontroller (AT89S52)

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

void msdelay(unsigned int time)  // Function for creating delay in milliseconds.
{
   unsigned i,j ;
   for(i=0;i<time;i++)
   for(j=0;j<1275;j++);
}
void lcd_cmd(unsigned char command)  //Function to send command instruction to LCD
{
   display_port = command;
   rs= 0;
   rw=0;
   e=1;
   msdelay(1);
   e=0;
}

void lcd_data(unsigned char disp_data)  //Function to send display data to LCD
{
   display_port = disp_data;
   rs= 1;
```

```
  rw=0;
  e=1;
  msdelay(1);
  e=0;
}

 void lcd_init()    //Function to prepare the LCD  and get it ready
{
  lcd_cmd(0x38);  // for using 2 lines and 5X7 matrix of LCD
  msdelay(10);
  lcd_cmd(0x0F);  // turn display ON, cursor blinking
  msdelay(10);
  lcd_cmd(0x01);  //clear screen
  msdelay(10);
  lcd_cmd(0x81);  // bring cursor to position 1 of line 1
  msdelay(10);
}
void main()
{
  unsigned char a[15]="CIRCUIT DIGEST";    //string of 14 characters with a null terminator.
  int l=0;
  lcd_init();
  while(a[l] != '\0') // searching the null terminator in the sentence
  {
    lcd_data(a[l]);
    l++;
    msdelay(50);
  }
}
```