AVIT
AARUPADAI VEEDU INSTITUTE OF TECHNOLOGY
VINAYAKA MISSION'S RESEARCH FOUNDATION
(Deemed to be University under section 3 of the UGC Act 1956)
Accredited by NAAC
Approved by AICTE

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

EMBEDDED SYSTEMS LAB I – LAB MANUAL

PROG, BRANCH,	**M.E., E.S.T**
YEAR/ SEMESTER, SECTION	**I/ I, -**
SUBJECT	**EMBEDDED SYSTEMS LAB - I**
ACADEMIC YEAR	**2021-2022 (ODD SEMESTER)**

HOD/ ECE

EX.NO 1. Design with 8 bit Microcontrollers 8051/PIC Microcontrollers

I) I/O Programming, Timers, Interrupts, Serial port programming

ii) PWM Generation,

Motor Control, ADC/DAC, LCD and RTC Interfacing, Sensor Interfacing

iii) Both Assembly and C programming

AIM:

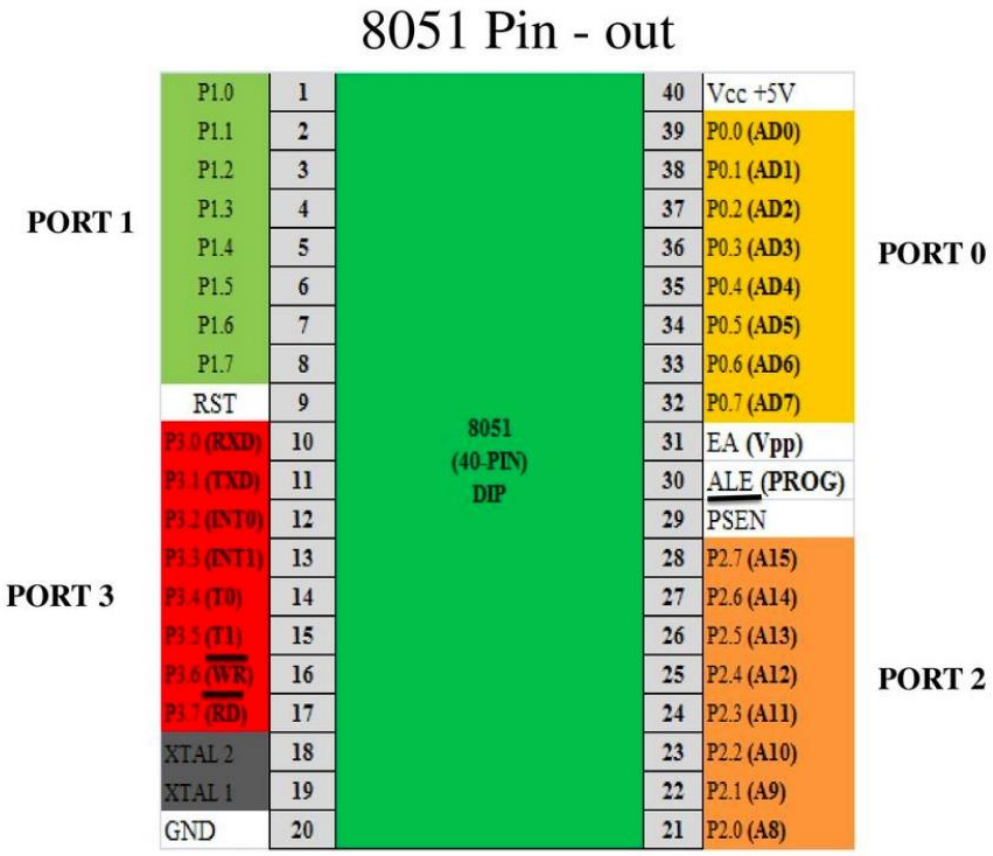
To design with 8 bit microcontrollers 8051/PIC microcontrollers

i) I/O Programming, Timers, Interrupts, Serial port programming ii) PWM Generation, Motor Control, ADC/DAC, LCD and RTC Interfacing, Sensor Interfacing iii) Both Assembly and C programming

Components:

- 8051 Microcontroller (AT89S52)
- ADC0808/0809
- 16x2 LCD
- Resistor (1k,10k)
- POT(10k x4)
- Capacitor(10uf,1000uf)
- Red led
- Bread board or PCB
- 7805
- 11.0592 MHz Crystal
- Power
- Connecting wires

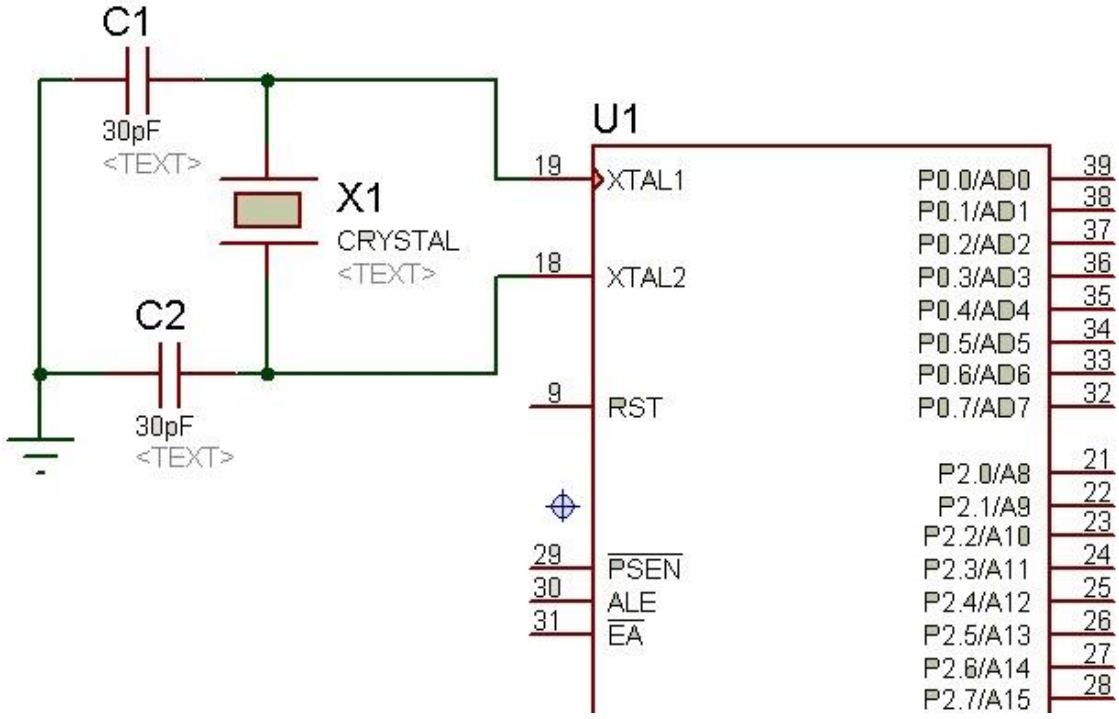
1. **I/O PROGRAMMING**
   PIN DIAGRAM

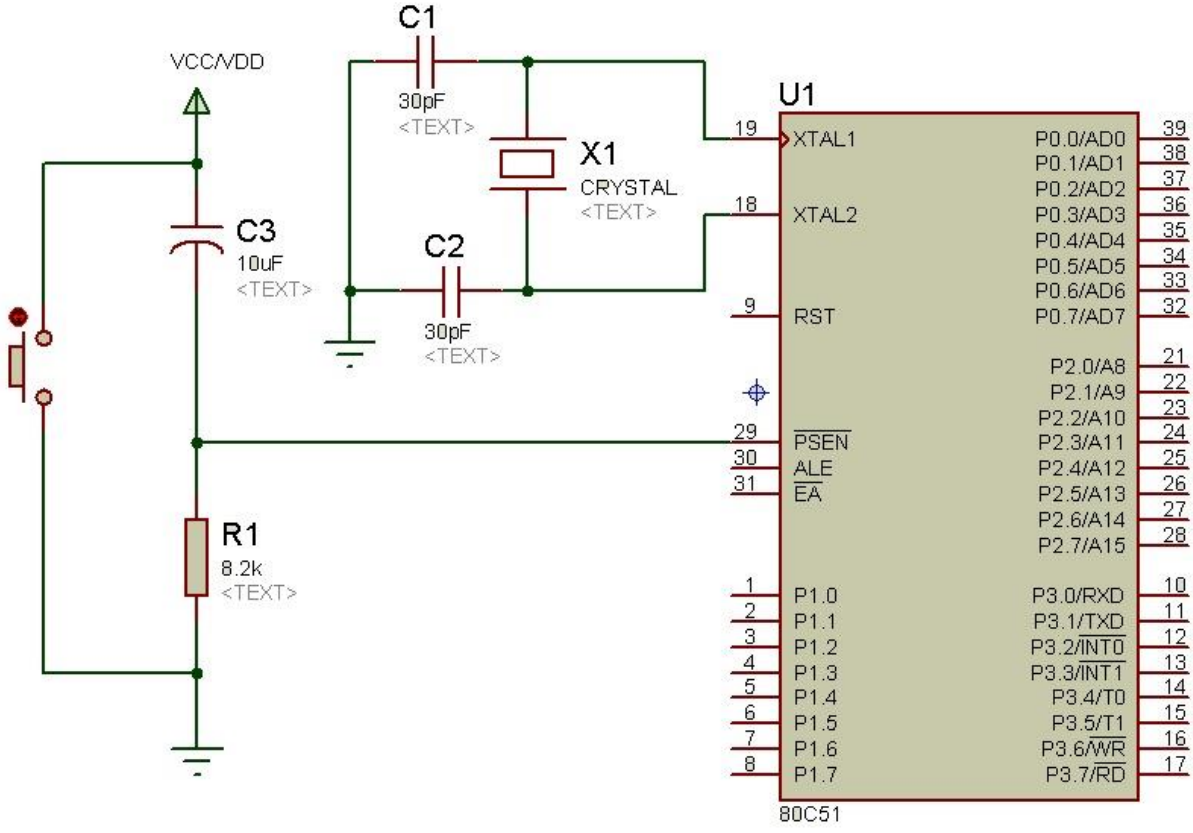


Pin Configuration Details

A brief explanation of all pins is given below

- Vcc : It provides supply voltage to chip. The operating voltage is 5 volt.
- GND : It connect with ground of voltage supply
- XTAL1 and XTAL2 pin: Although 8051 have on chip crystal oscillator. But still it requires an external clock oscillator. External crystal oscillator is connected to XTAL1 and XTAL2 pins. It also requires two capacitors of 30pF as shown in figure below. Capacitors one terminal is connected with crystal oscillator and other terminal with ground. Processing speed of 8051 microcontroller depends on crystal oscillator frequency. But each microcontroller have maximum limit of operating frequency. We cannot connect crystal oscillator more than maximum operating limit frequency.


C1
30pF
<TEXT>
X1
CRYSTAL
<TEXT>
C2
30pF
<TEXT>
U1
19
XTAL1
18
XTAL2
9
RST
29
PSEN
30
ALE
31
EA
P0.0/AD0
P0.1/AD1
P0.2/AD2
P0.3/AD3
P0.4/AD4
P0.5/AD5
P0.6/AD6
P0.7/AD7
P2.0/A8
P2.1/A9
P2.2/A10
P2.3/A11
P2.4/A12
P2.5/A13
P2.6/A14
P2.7/A15


VCC/VDD
C1
30pF
<TEXT>
X1
CRYSTAL
<TEXT>
C2
30pF
<TEXT>
C3
10uF
<TEXT>
R1
8.2k
<TEXT>
U1
XTAL1
XTAL2
RST
PSEN
ALE
EA
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
P1.6
P1.7
P0.0/AD0
P0.1/AD1
P0.2/AD2
P0.3/AD3
P0.4/AD4
P0.5/AD5
P0.6/AD6
P0.7/AD7
P2.0/A8
P2.1/A9
P2.2/A10
P2.3/A11
P2.4/A12
P2.5/A13
P2.6/A14
P2.7/A15
P3.0/RXD
P3.1/TXD
P3.2/INT0
P3.3/INT1
P3.4/T0
P3.5/T1
P3.6/WR
P3.7/RD
80C51

- EA Pin number 33 is used to store program. All family of 8051 microcontrollers comes with on chip ROM to store programs. For such purpose EA pin is connected with Vcc. EA stands for external access.
- PSEN Pin number 29 is an output pin. It is stands for “Program store enable”. It is also used for programming.

Input/output ports P0, P1, P2 and P3 use to interface 8051 microcontroller with external devices. I have already explained it above.

8051 TIMERS:

```
 // Use of Timer mode 1 for blinking LED using polling method
// XTAL frequency 11.0592MHz
#include<reg51.h>
sbit led = P1^0;                          // LED connected to 1st pin o
f port P1
void delay();


main()
{
       unsigned int i;
       while(1)
       {
       led=~led;                          // Toggle LED
       for(i=0;i<1000;i++)
       delay();                           // Call delay
       }
}


void delay()                              // Delay generation using Tim
er 0 mode 1
{
       TMOD = 0x01;                       // Mode1 of Timer0
```

```
	TH0= 0xFC;				// FC66 evaluated hex value f
or 1millisecond delay
	TL0 = 0x66;
	TR0 = 1;				// Start Timer
	while(TF0 == 0);			// Using polling method
	TR0 = 0;				// Stop Timer
	TF0 = 0;				// Clear flag
}
```

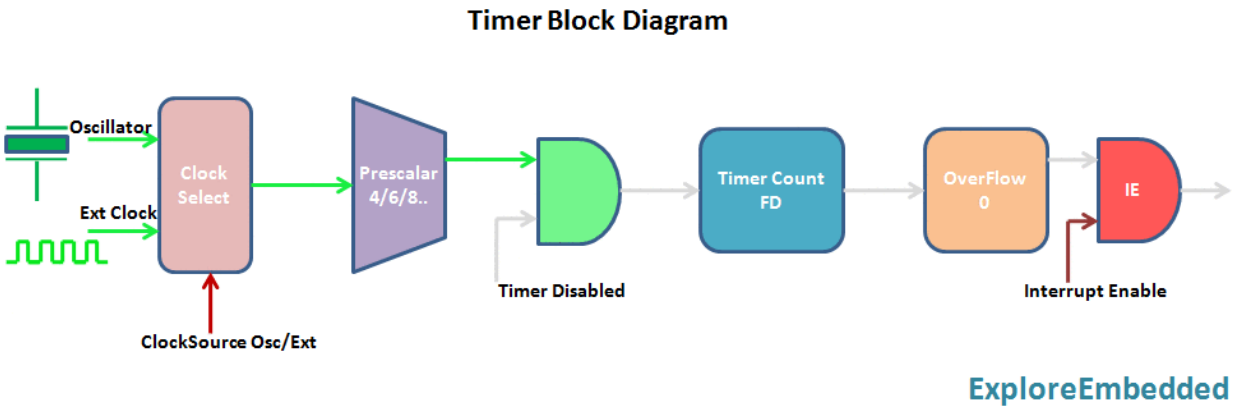
Example code

Time delay in Mode1 using interrupt method

```
// Use of Timer mode 1 for blinking LED with interrupt method
// XTAL frequency 11.0592MHz
#include<reg51.h>
sbit LED = P1^0;			// LED connected to 1st pin o
f port P1
void Timer(void) interrupt 1			// Interrupt No.1 for
Timer 0
{
	led=~led;			// Toggle LED on interrupt
}

main()
{
	TMOD = 0x01;			// Mode1 of Timer0
	TH0=0x00;			// Initial values loaded to T
imer
	TL0=0x00;
	IE = 0x82;			// Enable interrupt
	TR0=1;				// Start Timer
```

```
        while(1);                               // Do nothing
}
```

TIMER BLOCK DIAGRAM:



Timer Registers

TMOD							
Timer1				**Timer 0**			
7	6	5	4	3	2	1	0
Gate	C/T	M1	M0	Gate	C/T	M1	M0

INTRUPPTS METHOD:

Interrupt Method

```
#include<reg51.h>
sbit LED = P0^0;

void timer0_isr() interrupt 1
{
    TH0 = 0X4B;        //ReLoad the timer value
```

```
    TL0 = 0XFD;
    LED =! LED;         // Toggle the LED pin
}

void main()
{
    TMOD = 0x01;        //Timer0 mode 1
    TH0 = 0X4B;         //Load the timer value
    TL0 = 0XFD;
    TR0 = 1;            //turn ON Timer zero
    ET0 = 1;            //Enable TImer0 Interrupt
    EA = 1;             //Enable Global Interrupt bit
    while(1)
    {
        // Do nothing
    }
}
```

Write a program that continuously gets 8-bit data from PO and sends it to PI while simultaneously creating a square wave of 200 (as period on pin P2.1. Use Timer 0 to create the square wave. Assume that XTAL =11.0592 MHz.

Solution:

We will use Timer 0 in mode 2 (auto-reload). TH0 = 100/1.085 μs = 92.

```
;--Upon wake-up go to main, avoid using memory space ;allocat-
ed to Interrupt Vector Table
          ORG   0000H
          LJMP  MAIN        ;bypass interrupt vector table
;
;--ISR for Timer 0 to generate square wave
          ORG   000BH       ;Timer 0 interrupt vector table
          CPL   P2.1        ;toggle P2.1 pin
          RETI              ;return from ISR
;
;--The main program for initialization
          ORG   0030H       ;after vector table space
MAIN:     MOV   TMOD,#02H   ;Timer 0, mode 2(auto-reload)
          MOV   P0,#0FFH    ;make P0 an input port
          MOV   TH0,#-92    ;TH0=A4H for -92
          MOV   IE,#82H     ;IE=10000010(bin) enable Timer 0
          SETB  TR0         ;Start Timer 0
BACK:     MOV   A,P0        ;get data from P0
          MOV   P1,A        ;issue it to P1
          SJMP  BACK        ;keep doing it
                            ;loop unless interrupted by TF0
          END
```

SERIAL PORT PROGRAMMING:

If you do not want to do complicated programming using serial communications registers, you can simply use Mikro c compiler which has built in libraries for the UART communication. Now I will explain about UART programming through mikro c and after that I will provide an example of UART in keil compiler. We can use built-in UART library for handling 8051 serial communication, there are several functions. In this library.

- UART_init() is used for initialization of 8051 UART module
- UART_init() is used for initialization of 8051 UART module
- UART_data_read() function checks if some data received on serial Port
- UART_data_read() function checks if some data received on serial Port, UART_read() reads one byte from serial port
- whereas UART_Read_Text() can read multiple bytes
- UART_Write() writes one byte to serial port. UART_Write() writes one byte to serial port
- while UART_Write_Text() can write multiple bytes.
- UART_Set_Active() function is only used in micro-controllers that support more than one UART ports.

```
sbit LCD_RS at P2_0_bit;

sbit LCD_EN at P2_1_bit;
```

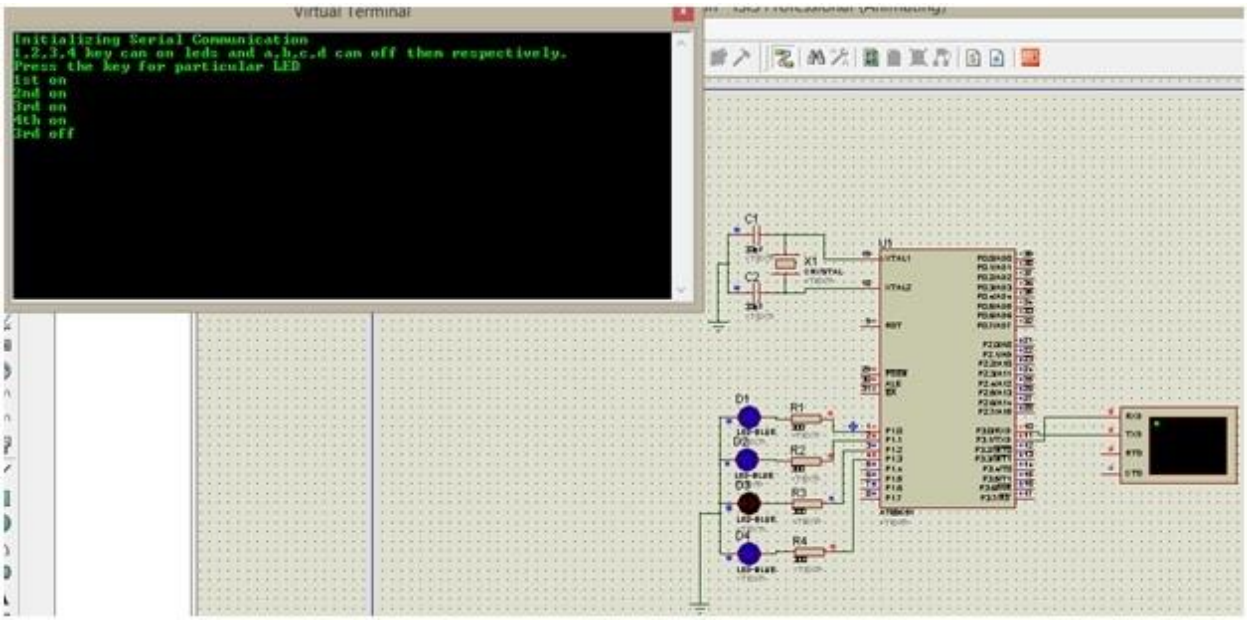
```
sbit LCD_D7 at P2_5_bit;

sbit LCD_D6 at P2_4_bit;

sbit LCD_D5 at P2_3_bit;

sbit LCD_D4 at P2_2_bit;


void main() {

unsigned int sum;

char txt[6];

P0=0xFF;

P1=0xFF;


uart1_init(9600);

PCON.SMOD=0;

Lcd_init();

lcd_cmd(_LCD_CURSOR_OFF);

lcd_out(1,1,"****************");

uart1_write_text("Uart OK");

while(1){

sum=P0+P1;

wordToStr(sum,txt);

lcd_out(2,1,"P0+P1=");

lcd_out(2,7,txt);
```

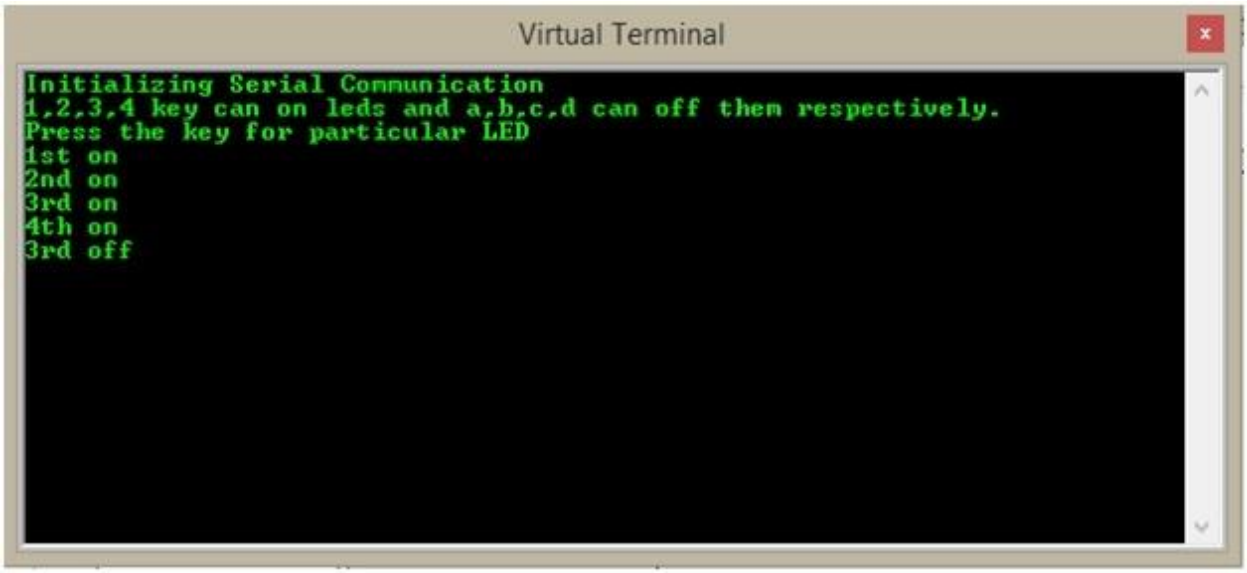
```
uart1_write_text("P0+P1=");

uart1_write_text(txt);

uart1_write_text("\r\n");

delay_ms(500);

}

}
```

In above we are transmitting data through UART port and also displaying same data on LCD. If you don't know how to display data on LCD, you can check tutorial here. Now lets see the explanation of code.

- First we initialized serial port at 9600 baud rate using UART_Init() function.
- Next we need to Clear SMOD bit of PCON register
- Actually there is some bug in uart_init() function of UART library, it writes HIGH to SMOD bit that causes erroneous baud rate.
- we can send some string to serial port with uart_write_text() function.
- Now in while(1) loop, we are sending "P0+P1" string to serial port with uart_write_text function.
- now we can send same text array to serial port that is being display on character LCD.

At the end, we are sending new line character, OK code is now ready we can compile and test it.





CODE of serial communication 8051 microcontroller

```
#include <REGX51.H>


void cct_init(void);



void SerialInitialize(void);


void uart_msg(unsigned char *c);


void uart_tx(unsigned char);
```

```
sbit led1 = P1^0;

sbit led2 = P1^1;

sbit led3 = P1^2;

sbit led4 = P1^3;

void main()

{

    cct_init();

    SerialInitialize();

    EA = 1;

    ES = 1;
```

```
    uart_msg("Initializing Serial Communication");

            uart_tx(0x0d);

            uart_msg("1,2,3,4 key can on leds and a,b,c,d can off them respectively.");

            uart_tx(0x0d);                                  //next line

            uart_msg("Press the key for particular LED");

            uart_tx(0x0d);

    while(1);

}
```

```
void cct_init(void)          //initialize cct

{

   P0 = 0x00;                          //not used

   P1 = 0x00;                          //output port used for leds

   P2 = 0x00;                          //not used

   P3 = 0x03;                          //used for serial communication

}


void SerialInitialize(void)      //Initialize Serial Port

{
```

```
   TMOD = 0x20;               //Timer 1 In Mode 2 -Auto Reload to Generate Baud
Rate

   SCON = 0x50;               //Serial Mode 1, 8-Data Bit, REN Enabled

   TH1 = 0xFD;                //Load Baud Rate 9600 To Timer Register

   TR1 = 1;                   //Start Timer

}

void uart_msg(unsigned char *c)

{

        while(*c != 0)

        {
```

```
                uart_tx(*c++);

        }

}


void uart_tx(unsigned char serialdata)

{

   SBUF = serialdata;              //Load Data to Serial Buffer Register

   while(TI == 0);                 //Wait Until Transmission To Complete

   TI = 0;                         //Clear Transmission Interrupt Flag
```

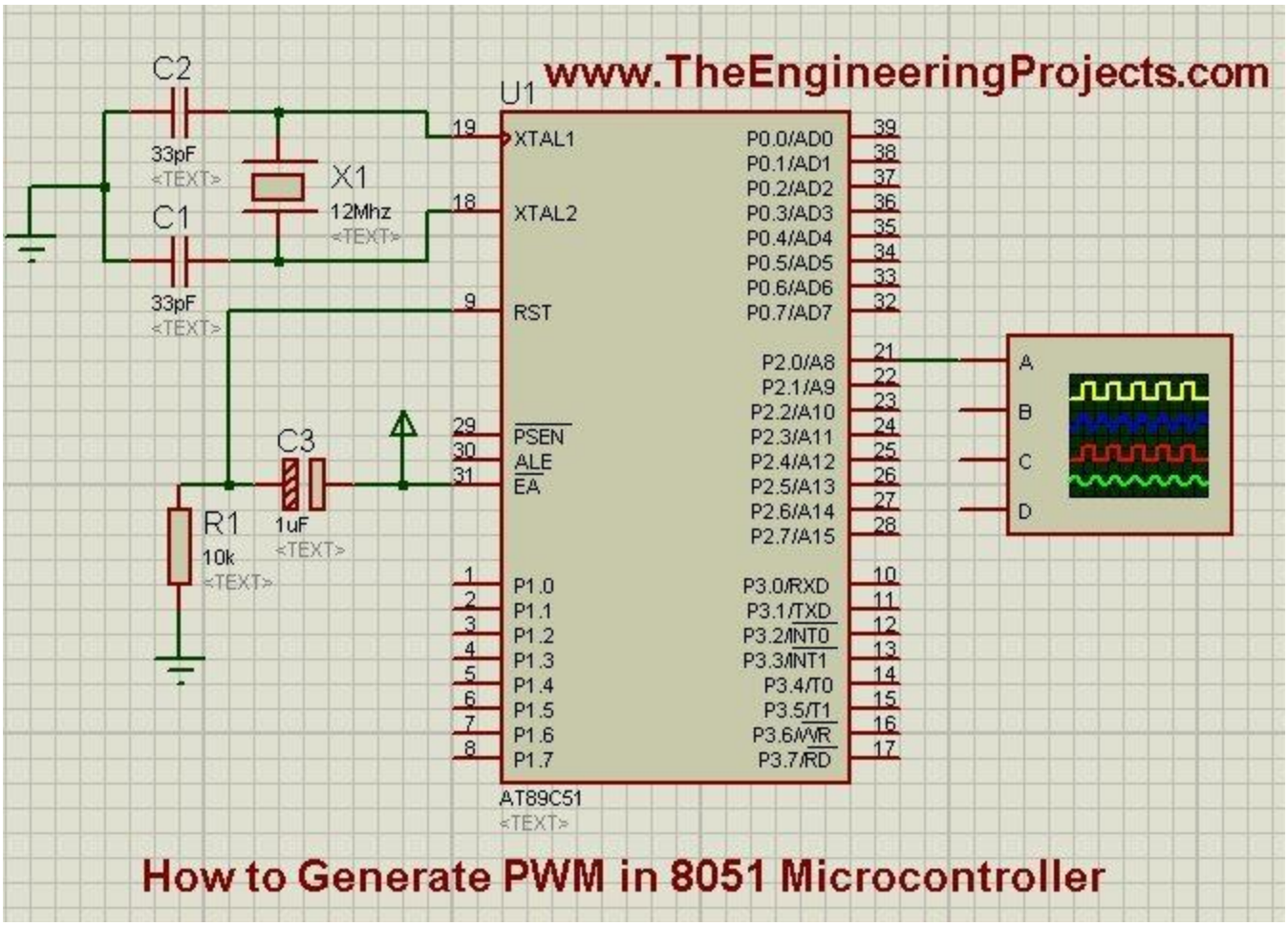
```
}




void serial_ISR (void) interrupt 4

{

   char chr;                              //receive character

   if(RI==1)

   {

      chr = SBUF;

      RI = 0;
```

```
}

P0 = ~P0;                              //Show the data has been updated

switch(chr)

{

 case '1':  led1 = 1; uart_msg("1st on"); uart_tx(0x0d); break;

 case '2':  led2 = 1; uart_msg("2nd on"); uart_tx(0x0d); break;

 case '3':  led3 = 1; uart_msg("3rd on"); uart_tx(0x0d); break;

 case '4':  led4 = 1; uart_msg("4th on"); uart_tx(0x0d); break;
```

```
    case 'a':  led1 = 0; uart_msg("1st off"); uart_tx(0x0d);  break;

    case 'b':  led2 = 0; uart_msg("2nd off"); uart_tx(0x0d); break;

    case 'c':  led3 = 0; uart_msg("3rd off"); uart_tx(0x0d); break;

    case 'd':  led4 = 0; uart_msg("4th off"); uart_tx(0x0d); break;

    default: ;    break;                    //do nothing

   }
   RI = 0;

}
```

8051 PWM GENERATION:

- First of all, design a simple circuit as shown in the below figure:



- Now what we are gonna do is we are gonna generate a PWM pulse using timer0 interrupt and then we are gonna send it to P2.0.
- I have attached an oscilloscope on which we can easily monitor this PWM pulse and can check whether it's correct or not.

- Now, copy the below code and paste it into your **Keil uvision** software. I have used Keil uvision 3 for this code compiling.

```
#include<reg51.h>

// PWM_Pin
sbit PWM_Pin = P2^0;            //  Pin  P2.0  is  named  as
PWM_Pin

// Function declarations
void cct_init(void);
```

```
void InitTimer0(void);
void InitPWM(void);

// Global variables
unsigned char PWM = 0;    // It can have a value from 0 (0%
duty cycle) to 255 (100% duty cycle)
unsigned int temp = 0;    // Used inside Timer0 ISR

// PWM frequency selector
/* PWM_Freq_Num can have values in between 1 to 257  only
 * When PWM_Freq_Num is equal to 1, then it means highest PWM
frequency
 * which is approximately 1000000/(1*255) = 3.9kHz
 * When PWM_Freq_Num is equal to 257, then it means lowest
PWM frequency
 * which is approximately 1000000/(257*255) = 15Hz
 *
 * So, in general you can calculate PWM frequency by using
the formula
 *     PWM Frequency = 1000000/(PWM_Freq_Num*255)
 */
#define PWM_Freq_Num   1      //   Highest   possible   PWM
Frequency


// Main Function
int main(void)
{
   cct_init();           // Make all ports zero
   InitPWM();              // Start PWM

   PWM = 127;              // Make 50% duty cycle of PWM

   while(1)               // Rest is done in Timer0 interrupt
   {}
}

// Init CCT function
void cct_init(void)
{
      P0 = 0x00;
      P1 = 0x00;
      P2 = 0x00;
      P3 = 0x00;
```

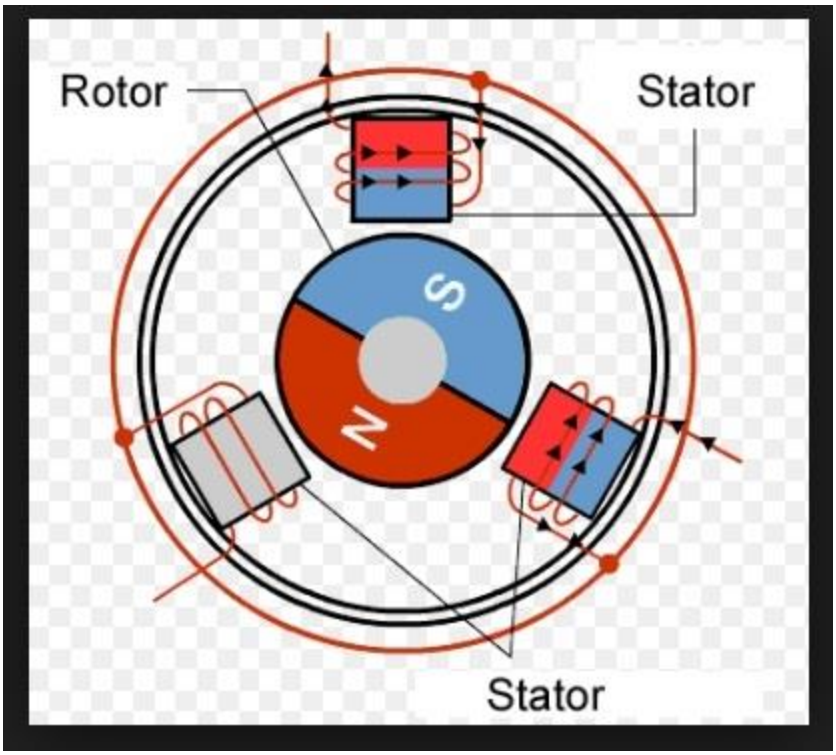
```
}

// Timer0 initialize
void InitTimer0(void)
{
      TMOD &= 0xF0;    // Clear 4bit field for timer0
      TMOD |= 0x01;    // Set timer0 in mode 1 = 16bit mode

      TH0 = 0x00;      // First time value
      TL0 = 0x00;      // Set arbitrarily zero

      ET0 = 1;         // Enable Timer0 interrupts
      EA  = 1;         // Global interrupt enable

      TR0 = 1;         // Start Timer 0
}

// PWM initialize
void InitPWM(void)
{
      PWM = 0;         // Initialize with 0% duty cycle
      InitTimer0();        //  Initialize  timer0  to  start
generating interrupts
                              //  PWM  generation  code  is
written inside the Timer0 ISR
}

// Timer0 ISR
void Timer0_ISR (void) interrupt 1
{
      TR0 = 0;    // Stop Timer 0

      if(PWM_Pin) // if PWM_Pin is high
      {
            PWM_Pin = 0;
            temp = (255-PWM)*PWM_Freq_Num;
            TH0  = 0xFF - (temp>>8)&0xFF;
            TL0  = 0xFF - temp&0xFF;
      }
      else       // if PWM_Pin is low
      {
            PWM_Pin = 1;
            temp = PWM*PWM_Freq_Num;
            TH0  = 0xFF - (temp>>8)&0xFF;
```

```
            TL0  = 0xFF - temp&0xFF;
        }

        TF0 = 0;      // Clear the interrupt flag
        TR0 = 1;      // Start Timer 0
}
```

- I have added the comments in the above codes so it won't be much difficult to understand. If you have a problem then ask in the comments and I will resolve them.
- Now in this code, I have used a PWM variable and I have given 127 to it as a starting value.
- PWM pulse varies from 0 to 255 as it's an 8-bit value so 127 is the mid-value which means the duty cycle will be 50%.
- You can change its value as you want it to be.

Stepper Motor

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.

The relation between RPM, steps per revolution, and steps per second is like below:

```
Steps per Second = rpm x steps per revolution / 60
```

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
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

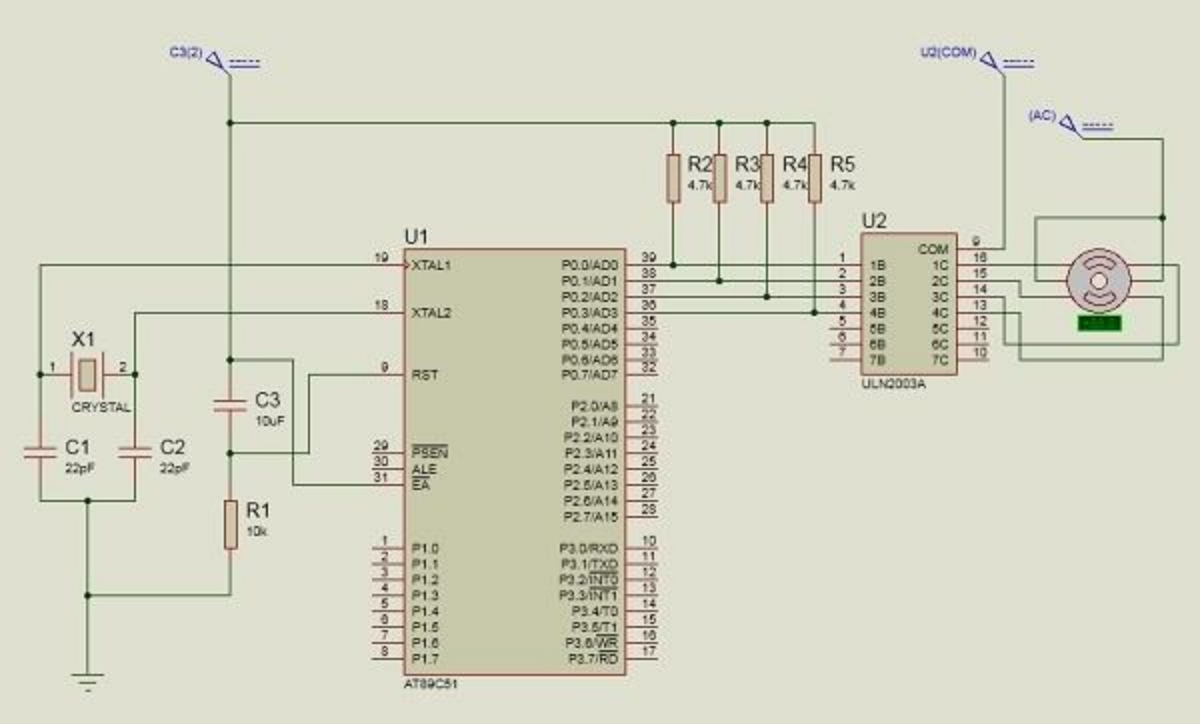
- **Half Drive Mode** − In this mode, one and two coils are energized alternately. At first, one coil is energized then two coils are energized. This is basically a

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

The circuit diagram is like below: We are using the full drive mode.



Example

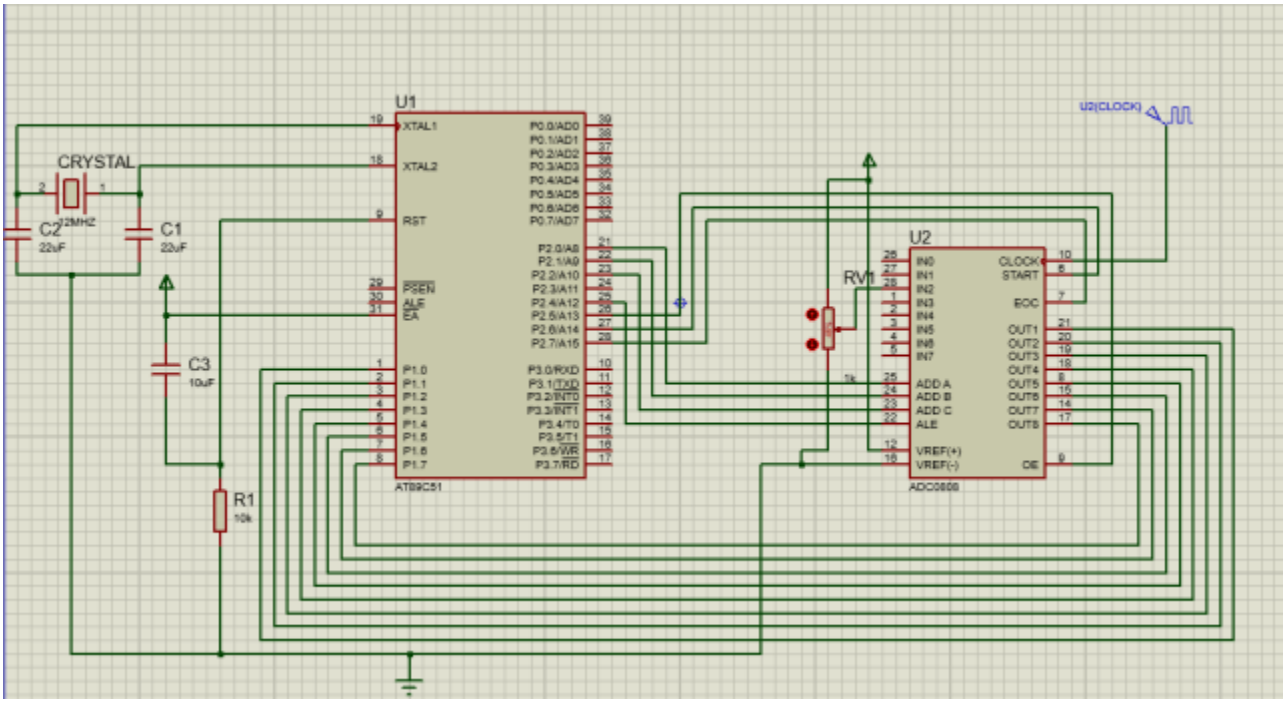
```
#include<reg51.h>
sbit LED_pin = P2^0; //set the LED pin as P2.0
void delay(int ms){
   unsigned int i, j;
   for(i = 0; i<ms; i++){ // Outer for loop for given milliseconds
value
      for(j = 0; j< 1275; j++){
         //execute in each milliseconds;
       }
   }
}
void main(){
   int rot_angle[] = {0x0C,0x06,0x03,0x09};
   int i;
   while(1){
      //infinite loop for LED blinking
      for(i = 0; i<4; i++){
         P0 = rot_angle[i];
```

```
            delay(100);
        }
    }
}
```

ADC/DAC INTERFACING:

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. In our case, we are using Proteus as the simulation software and the AT89C51 microcontroller.

- 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.

With this, you have successfully interfaced the 8051 to the ADC. Now let us look at the logic to use the ADC with the microcontroller.

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.

Assembly language program to interface ADC 0808 with 8051
Here is how the assembly code for the same looks like

```
ORG 0000H; Starting address

MOV P1,#0FFH; Makes port 1 input port

SETB P2.7; Makes EOC pin high

CLR P2.4; Clears ALE pin

CLR P2.6; Clears Start pin

CLR P2.5; Clears OE pin

BACK: CLR P2.2; Clears ADD C

SETB P2.1; Sets ADD B

CLR P2.0; Clears ADD A (this selects the second address
line)

ACALL DELAY

SETB P2.4; Sets ALE high

ACALL DELAY

SETB P2.6; sends a command to start conversion

ACALL DELAY

CLR P2.4; makes ALE low

CLR P2.6; makes Start pin low

HERE: JB P2.7,HERE; waits for low pulse at EOC
```

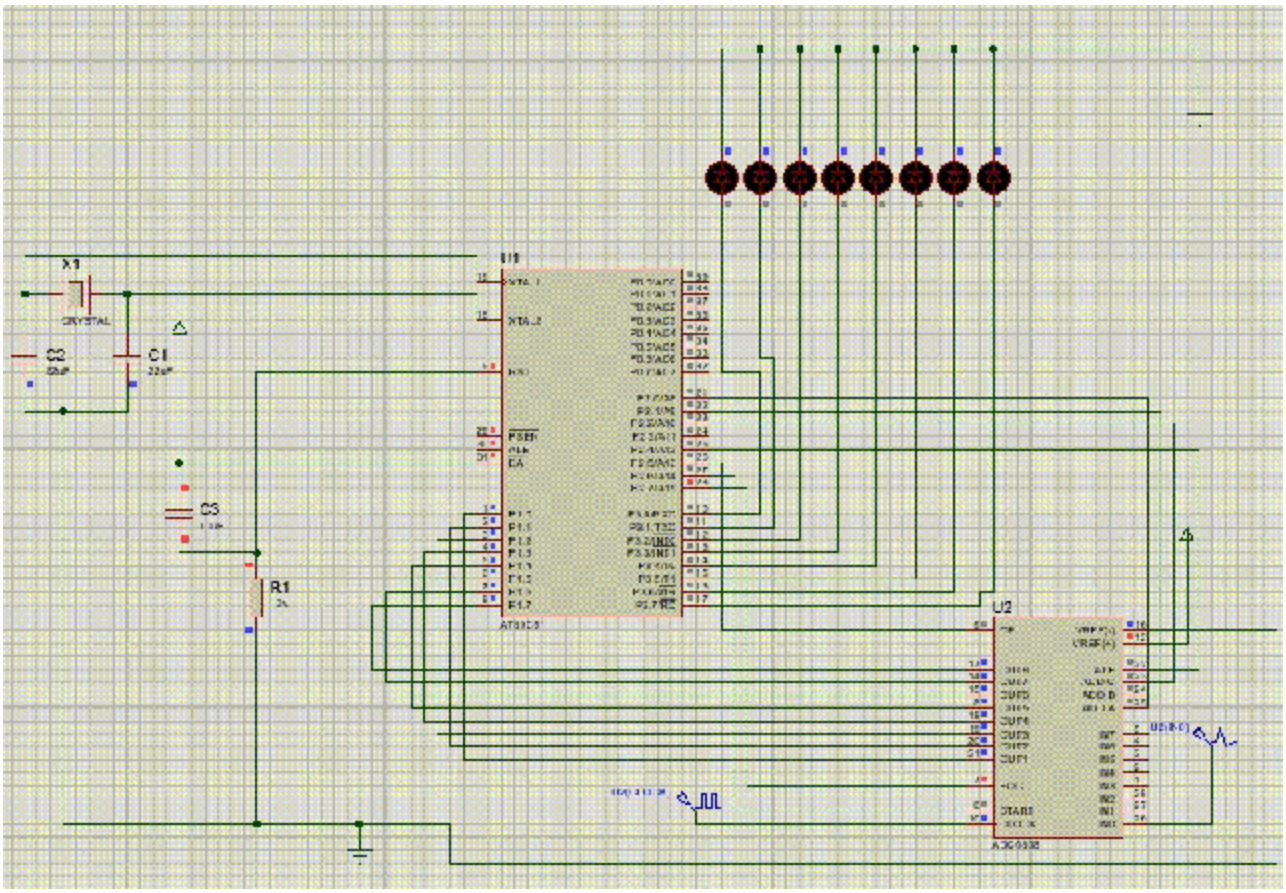
```
HERE1: JNB P2.7,HERE1; waits for low pulse to finish

SETB P2.5; enables OE pin to extract data from ADC

ACALL DELAY

MOV A,P1; moves acquired data to accumulator

CLR P2.5; clears OE

SJMP BACK; repeatedly gets data from ADC

DELAY: MOV R3,#50

HERE2: MOV R4,#255

HERE3: DJNZ R4,HERE3

DJNZ R3,HERE2

RET

END
```

Now that we have a basic understanding of how to interface an ADC with the 8051, let us look at an example in which we connect LEDs to 8051 to see the data conversion.

C program to interface ADC 0808 with 8051

Seeing data conversion using LEDs

To see the data conversion of an ADC, we will extract the data using the code shown above. Then we will transfer the binary data to port 3 to see the data.



The C code for the same is given below:

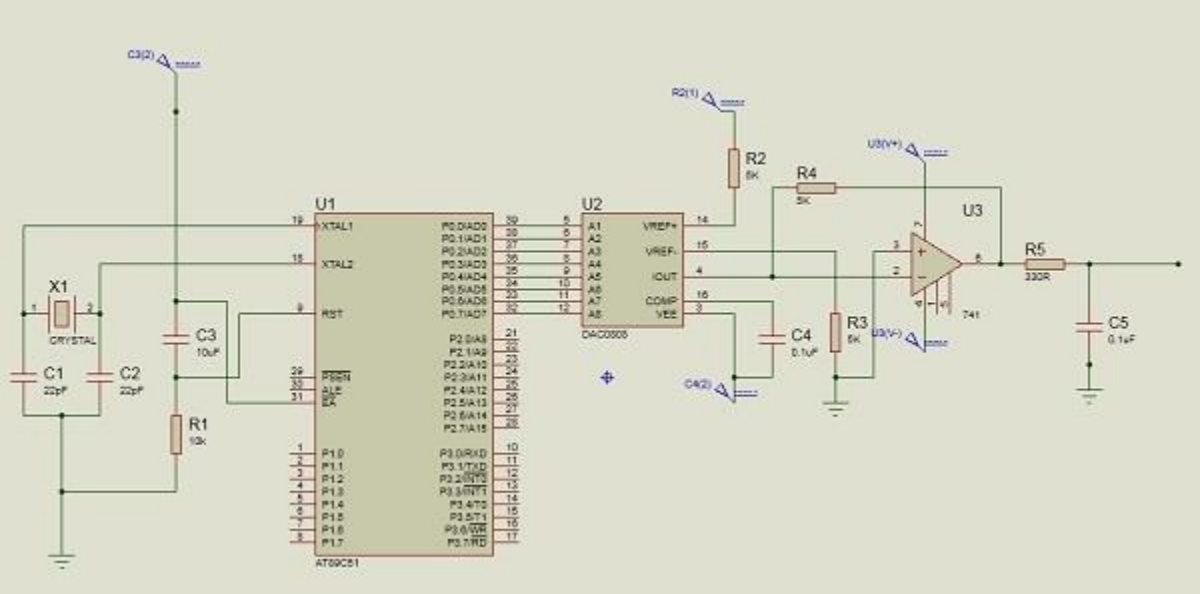
```
#include <reg51.h>

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 ;

}

}
```

DAC INTERFACING:

Circuit Diagram −

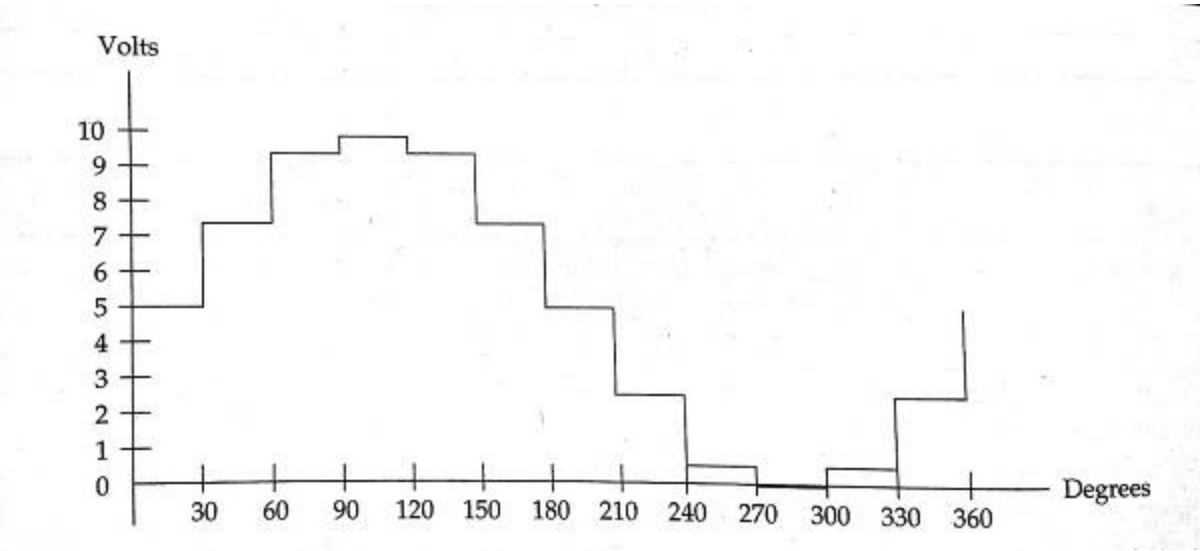


Source Code

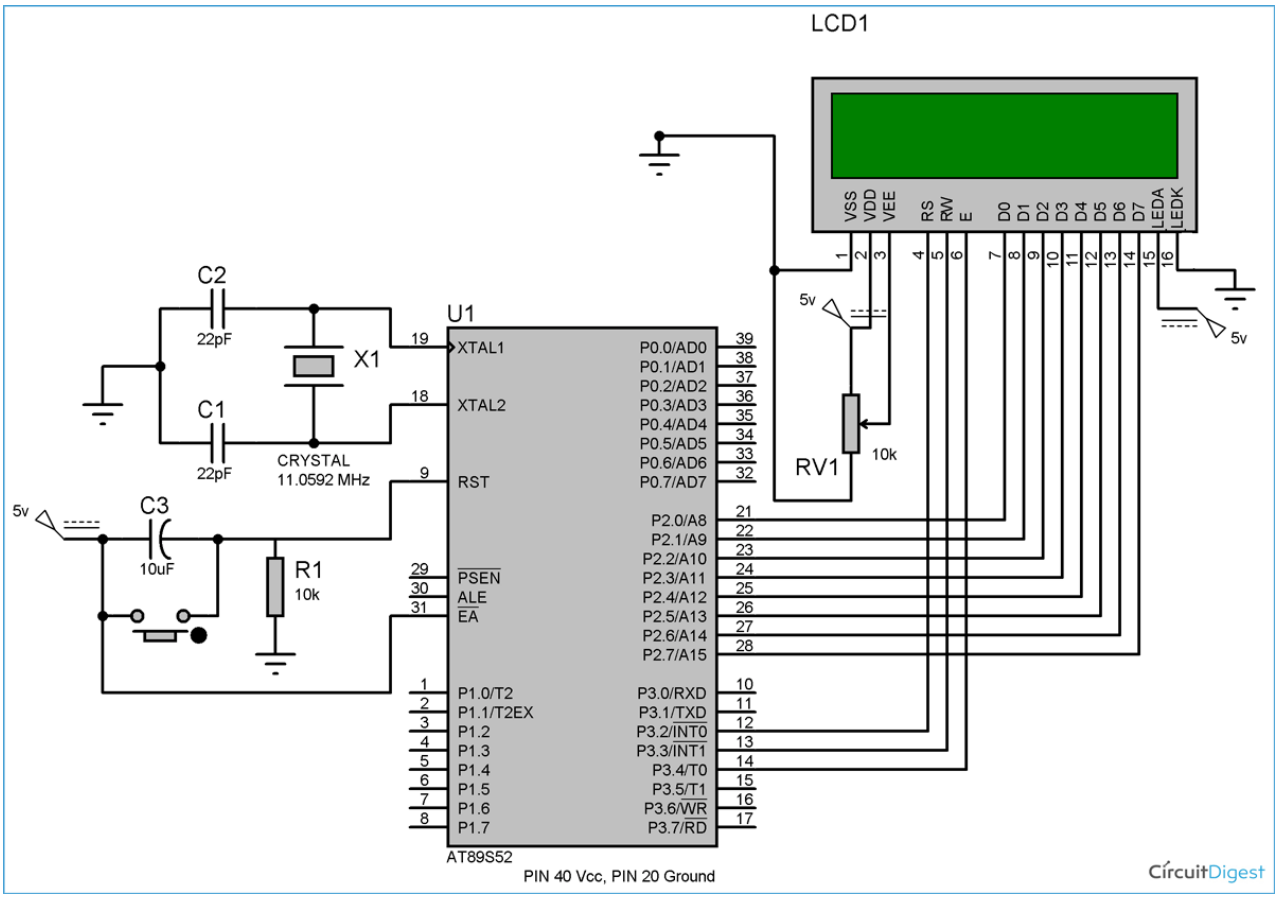
```
#include<reg51.h>
sfr DAC = 0x80; //Port P0 address
void main(){
   int sin_value[12] = {128,192,238,255,238,192,128,64,17,0,17,64};
   int i;
   while(1){
      //infinite loop for LED blinking
      for(i = 0; i<12; i++){
         DAC = sin_value[i];
      }
   }
}
```

Output

The output will look like this −



Circuit Diagram and Explanation



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. If you don't have any idea about that then I recommend you to read this Article LED Interfacing with 8051 Microcontroller before going through LCD interfacing.

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.

Code Explanation

I have tried to explain the code through comments (in code itself).

As I have explained earlier about command mode and data mode, you can see that while sending command (function lcd_cmd) we have set RS=0, RW=0 and a HIGH to LOW pulse is given to E by making it 1, then 0. Also when sending data (function lcd_data) to LCD we have set RS=1, RW=0 and a HIGH to LOW pulse is given to E by making it 1 to 0. Function msdelay() has been created to create delay in milliseconds and called frequently in the program, it is called so that LCD module can have sufficient time to execute the internal operation and commands.

A while loop has been created to print the string, which is calling the lcd_data function each time to print a character until the last character (null terminator- '\0').

We have used the lcd_init() function to get the LCD ready by using the preset command instructions (explained above).

Code

```
// 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
{
```

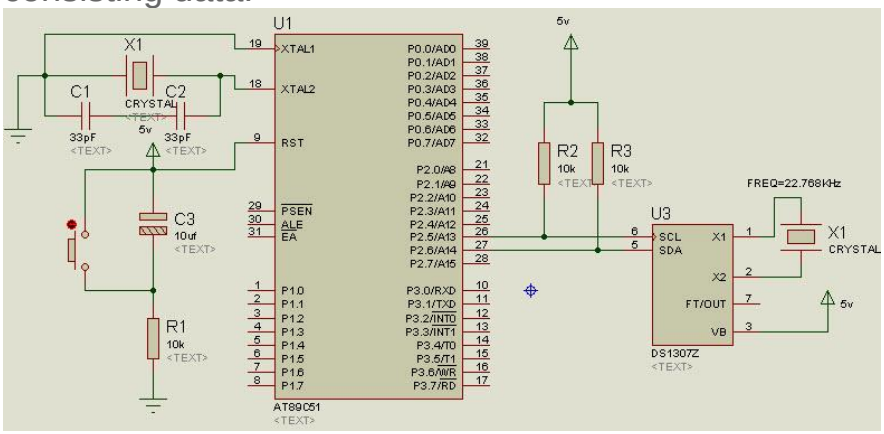
```
    display_port = command;
    rs= 0;
    rw=0;
    e=1;
    msdelay(1);
    e=0;
}

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);
    }
}
```

RTC INTERFACING:

Interfacing RTC ds1307 with 8051

RTC can be interfaced to microcontroller by using different serial bus protocols such as I2C and SPI protocols that provide communication link between them. The figure shows, real time clock interfacing with 8051 microcontroller using I2C bus protocol. I2C is a bi-directional serial protocol, which consist of two

wires such as SCL and SDA to transfer data between devices connected to bus. 8051 microcontroller has no inbuilt RTC device therefore we have connected externally through a serial communication for ensuring the consisting data.



RTC Interfacing with 8051 Microcontroller

I2C devices have open drain outputs therefore, a pull-up resistors must be connected to the I2C bus line with a voltage source. If the resistors are not connected to the SCL and SDL lines, the bus will not work.

Step4: RTC Data Framing Format

Since RTC interfacing with8051 microcontroller uses I2C bus therefore the data transfer is in the form of bytes or packets and each byte is followed by an acknowledgement.

Transmitting Data Frame:

In transmitting mode, the master release the start condition after selecting slave device by address bit. The address bit contains 7-bit, which indicate the slave devices as ds1307 address. Serial data and serial clock are transmitted on SCL and SDL lines. START and STOP conditions are recognized as beginning and ending of a serial transfer. Receive and transmit operations are followed by the R/W bit.

Start	slave address	R/W	Control register Address	Control Status register1	Control Status register2	8-bit Data	ACK	Stop

Transmitting Data Frame

Start: Primarily, the data transfer sequence initiated by the master generating the start condition.

7-bit Address: After that the master sends the slave address in two 8-bit formats instead of a single 16-bit address.
Control/Status Register Address: The control/status register address is to allow the control status registers.
Control/Status Register1: The control status register1 used to enable the RTC device
Control/Status Register2: It is used to enable and disable interrupts.
R/W: If read and write bit is low, then the write operation is performed.
ACK: If write operation is performed in the slave device, then the receiver sends 1-bit ACK to microcontroller.
Stop: After completion of write operation in the slave device, microcontroller sends stop condition to the slave device.

Receiving Data Frame:

Start	slave address	R/W =1	Control register Address	Control Status register1	Control Status register2	8-bit Data	ACK	Stop

Receiving Data Frame

Start: Primarily, the data transfer sequence initiated by the master generating the start condition.
7-bit Address: After that the master sends slave address in two 8-bit formats instead of a single 16-bit address.
Control/Status Register Address: The control/status register address is to allow control status registers.
Control/Status Register1: The control status register1 used to enable the RTC device

Control/Status Register2: It is used to enable and disable interrupts.
R/W: If read and write bit is high, then the read operation is performed.
ACK: If write operation is performed in the slave device, then the receiver sends 1-bit ACK to microcontroller.
Stop: After completion of write operation in the slave device, microcontroller sends stop condition to the slave device.

Step5: RTC Programming

Write Operation from Master to Slave:

1. Issue the start condition from master to slave
2. Transfer the slave address in write mode on SDL line
3. Send the control register address
4. Send the control/status register1value
5. Send the control/status register2 value
6. Send the date of the like minutes, seconds and hours

7. Send the stop bit

```
#include<reg51.h>

sbit SCL=P2^5;
sbit SDA=P2^6;
void start();
void wirte(unsigned char);
delay(unsigned char);

void main()
{

start();
write(0xA2); //slave address//
write(0x00); //control register address//
write(0x00); //control register 1 value//
write(0x00); //control regiter2 vlaue//
write (0x28); //sec value//
write(0x50) ;//minute value//
write(0x02);//hours value//
}

void start()
{

SDA=1; //processing the data//
SCL=1; //clock is high//
delay(100);
SDA=0; //sent the data//
delay(100);
SCL=0; //clock signal is low//
}
void write(unsigned char d)
{

unsigned char k, j=0×80;
for(k=0;k<8;k++)
{
SDA=(d&j);
J=j>>1;
```

```
SCL=1;
delay(4);
SCL=0;
}
SDA=1;
SCL=1;
delay(2);
c=SDA;
delay(2);
SCL=0;
}
void delay(int p)
{
unsignedinta,b;
For(a=0;a<255;a++); //delay function//
For(b=0;b<p;b++);
}
```

Read Operation from Slave to Master:

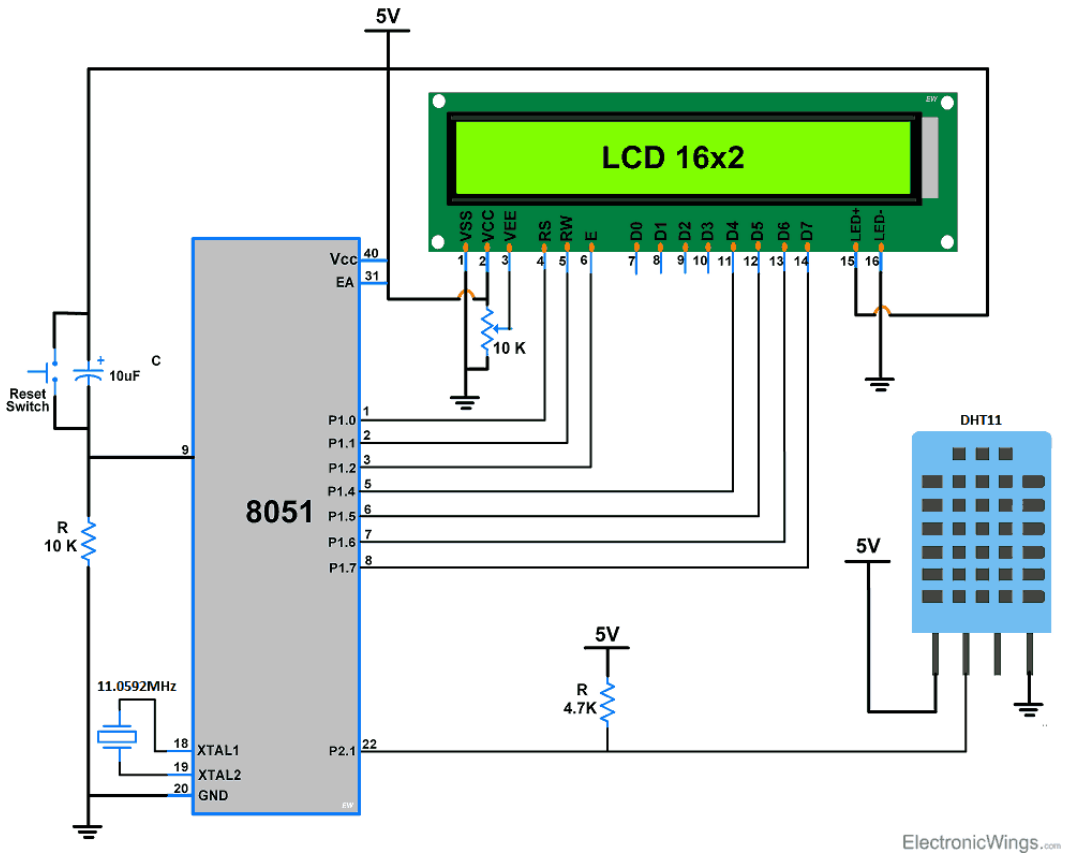
```
#include<reg51.h>
sbit SCL=P2^5;
sbit SDA=P2^6;
void start();
void write(usigned char );
void read();
void ack();
void delay(unsigned char);
void main()
{
start();
write(0xA3);// slave address in read mode//
read();
ack();
sec=value;
}
void start()
{

SDA=1; //processing the data//
SCL=1; //clock is high//
delay(100);
```

```
SDA=0; //sent the data//
delay(100);
SCL=0; //clock signal is low//
}
void write(unsigned char d)
{

unsigned char k, j=0×80;
for(k=0;k<8;k++)
{
SDA=(d&j);
J=j>>1;
SCL=1;
delay(4);
SCL=0;
}
SDA=1;
SCL=1;
delay(2);
c=SDA;
delay(2);
SCL=0;
}
void delay(int p)
{
unsignedinta,b;
For(a=0;a<255;a++); //delay function//
For(b=0;b<p;b++);
}
Void read ()
{
Unsigned char j, z=0×00, q=0×80;
SDA=1;
for(j=0;j<8;j++)
{
SCL=1;
delay(100);
flag=SDA;
if(flag==1)
{
z=(z|q);
```

```
q=q>>1;
delay (100);
SCL=0;
}
void ack()
{
SDA=0; //SDA line goes to low//
SCL=1; //clock is high to low//
delay(100);
SCL=0;
}
```

SENSOR INTERFACING:

Circuit diagram



DHT11 Sensor Interfacing with 8051.

- The above circuit diagram shows the interfacing of 8051 with the DHT11 sensor.
- In that, a DHT11 sensor is connected to P2.1(Pin No 22) of the microcontroller.

Programming Steps

- First, initialize the LCD16x2_4bit.h library.
- Define pin no. to interface DHT11 sensor, in our program we define P2.1 (Pin no.22)
- Send the start pulse to the DHT11 sensor by making low to high on the data pin.
- Receive the response pulse from the DHT11 sensor.
- After receiving the response, receive 40-bit data serially from the DHT11 sensor.
- Display this received data on LCD16x2 along with error indication.

Program

```
/*
 * DHT11 Interfacing with 8051
 * http://www.electronicwings.com
 */



#include<reg51.h>
#include<stdio.h>
#include<string.h>
#include  <stdlib.h>
#include  "LCD16x2_4bit.h"

sbit DHT11=P2^1;         /* Connect DHT11 output Pin to P2.1 Pin */
int I_RH,D_RH,I_Temp,D_Temp,CheckSum;
void  timer_delay20ms()          /* Timer0 delay function */
{
        TMOD = 0x01;
        TH0 = 0xB8;              /* Load higher 8-bit in TH0 */
        TL0 = 0x0C;              /* Load lower 8-bit in TL0 */
```

```
        TR0 = 1;         /* Start timer0 */
        while(TF0 == 0);/* Wait until timer0 flag set */
        TR0 = 0;         /* Stop timer0 */
        TF0 = 0;         /* Clear timer0 flag */
}

void  timer_delay30us()          /* Timer0 delay function */
{
        TMOD = 0x01;             /* Timer0 mode1 (16-bit timer mode) */
        TH0 = 0xFF;              /* Load higher 8-bit in TH0 */
        TL0 = 0xF1;              /* Load lower 8-bit in TL0 */
        TR0 = 1;         /* Start timer0 */
        while(TF0 == 0);/* Wait until timer0 flag set */
        TR0 = 0;         /* Stop timer0 */
        TF0 = 0;         /* Clear timer0 flag */
}

void  Request()                  /* Microcontroller send request */
{
        DHT11 = 0;               /* set to low pin */
        timer_delay20ms();       /* wait for 20ms */
        DHT11 = 1;               /* set to high pin */
}

void  Response()                 /* Receive response from DHT11 */
{
        while(DHT11==1);
        while(DHT11==0);
        while(DHT11==1);
}

int  Receive_data()              /* Receive data */
{
        int q,c=0;
        for (q=0; q<8; q++)
```

```
        {
                while(DHT11==0);/* check received bit 0 or 1 */
                timer_delay30us();
                if(DHT11 == 1)   /* If high pulse is greater than 30ms */
                c = (c<<1)|(0x01);/* Then its logic HIGH */
                else            /* otherwise its logic LOW */
                c = (c<<1);
                while(DHT11==1);
        }
        return c;
}
void main()
{
        unsigned char dat[20];
        LCD_Init();             /* initialize LCD */

        while(1)
        {
                Request();      /* send start pulse */
                Response();     /* receive response */

                I_RH=Receive_data();     /* store first eight bit in I_RH */
                D_RH=Receive_data();     /* store next eight bit in D_RH */
                I_Temp=Receive_data();   /* store next eight bit in I_Temp */
                D_Temp=Receive_data();   /* store next eight bit in D_Temp */
                CheckSum=Receive_data();/* store next eight bit in CheckSum */

                if ((I_RH + D_RH + I_Temp + D_Temp) != CheckSum)
                {
                        LCD_String_xy(0,0,"Error");
                }

                else
```

```
                {
                        sprintf(dat,"Hum = %d.%d",I_RH,D_RH);
                        LCD_String_xy(0,0,dat);
                        sprintf(dat,"Tem = %d.%d",I_Temp,D_Temp);
                        LCD_String_xy(1,0,dat);
                        LCD_Char(0xDF);
                        LCD_String("C");
                        memset(dat,0,20);
                        sprintf(dat,"%d   ",CheckSum);
                        LCD_String_xy(1,13,dat);
                }
                delay(100);
        }
}
```

RESULT:

DESIGNED ALL PARAMETERS OF 8051 8 BIT PROCESSORS.

EX.NO:2

DESIGN WITH 16 BIT PROCESSORS

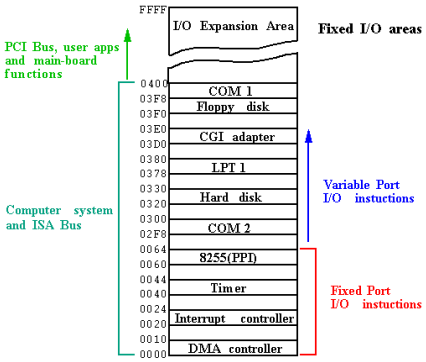
I/O programming, Timers, Interrupts, Serial Communication,

AIM:

TO DESIGN 16 BIT PROCESSORS

I/O programming,timers,interrupts,serial communication

I/O Map



TIMERS:

Procedure to Calculate the Delay Program

1. First we have to load the TMOD register value for ‘Timer0’ and ‘Timer1’in different modes. For example, if we want to operate timer1 in mode1 it must be configured as “TMOD=0x10”.

2. Whenever we operate the timer in mode 1, timer takes the maximum pulses of 65535. Then the calculated time-delay pulses must be subtracted from the maximum pulses, and afterwards converted to hexadecimal value. This value has to be loaded in timer1 higher bit and lower bits. This timer operation is programmed using embedded C in a microcontroller.

Example: 500us time delay

500us/1.080806us

461pulses

P=65535-461

P=65074

65074 conveted by hexa decimal =FE32

TH1=0xFE;

TL1=0x32;

3. Start the timer1 “TR1=1;”

4. Monitor the flag bit “while(TF1==1)”

5. Clear the flag bit “TF1=0”

6. Cleat the timer “TR1=0”

Example Programs:

WAP to generate 100 ms time delay using timer 0 mode 0

```
#include<reg51.h>
void main()
{
 unsigned char i;

 TMOD=0x00;
for(=0;i<20;i++)
{
  TL0=0x03;
TH0=0x0E;
TR0=1;
while(TF==0);
TF=0;
}
TR0=0;
}
```

100ms/1.08592us
=920878/20=4604 (20 for loop)
8191-4604=3587
3587 convetted by hex =0E03

Program- 1

WAP to generate 500ms using T1 M1

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

    unsigned char i;
    TMOD=0x10;
    for(i=0;i<10;i++)
    {
       TL1=0x24;
        TH1=0x4C;
       TR0=1;
       while(TF1==0);
        TF1=0;
     }
  TR1=0;
}
```

500ms/1.08592us
=460439
460439/10=46044
65535-46044=19492
19492 hexa= 4C24

Program- 2

WAP to generate 1sec time delay using T0 M1

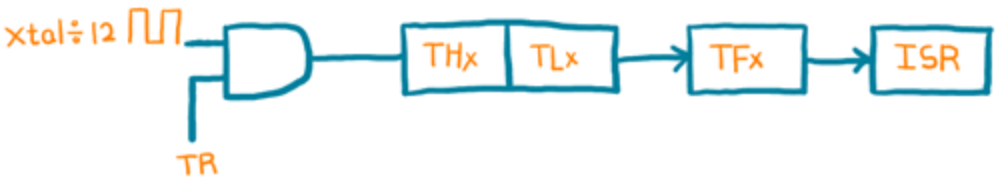
```
#include<reg51.h>
void main()
{

  unsinged char i;
  TMOD=0x01;
  For(i=0;i<20;i++)
  {
     TL0=0xFC;
     TH0=0x4B;
     TR0=1;
     while(TF0==1);
     TF0=0;
   }
TR0=0;
}
```

Timer interrupt in 8051

8051 has two timer interrupts assigned with different vector address. When Timer count rolls over from its max value to 0, it sets the timer flag TFx. This will interrupt the 8051 microcontroller to serve ISR (interrupt service routine) if global and timer interrupt is enabled.

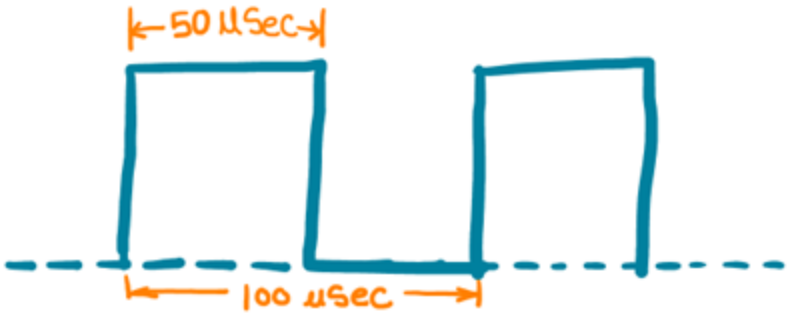
Interrupt source	Vector address
Timer 0 overflow (TF0)	000BH
Timer 1 overflow (TF1)	001BH

The timer overflow interrupt assigned with the vector address shown in the table. 8051 microcontroller jumps directly to the vector address on the occurrence of a corresponding interrupt.



Example

Here we will generate a square wave of 10Hz on PORT1.0 using Timer0 interrupt. We will use Timer0 in mode1 with 11.0592 MHz oscillator frequency.



As Xtal is 11.0592 MHz we have a machine cycle of 1.085uSec. Hence, the required count to generate a delay of 50mSec. is,

$$Count = (50 \times 10^{(-3)}) / (1.085 \times 10^{(-6)}) \approx 46080$$

And mode1 has a max count is 2^16 (0 - 65535) and it increment from 0 – 65535 so we need to load value which is 46080 less from its max. count i.e. 65535. Hence value need to be load is,

$$Value = (65535 - Count) + 1 = 19456 = (4C00)Hex$$

So we need to load 4C00 Hex value in a higher byte and lower byte as,

TH0 = 0x4C & TL0 = 0x00

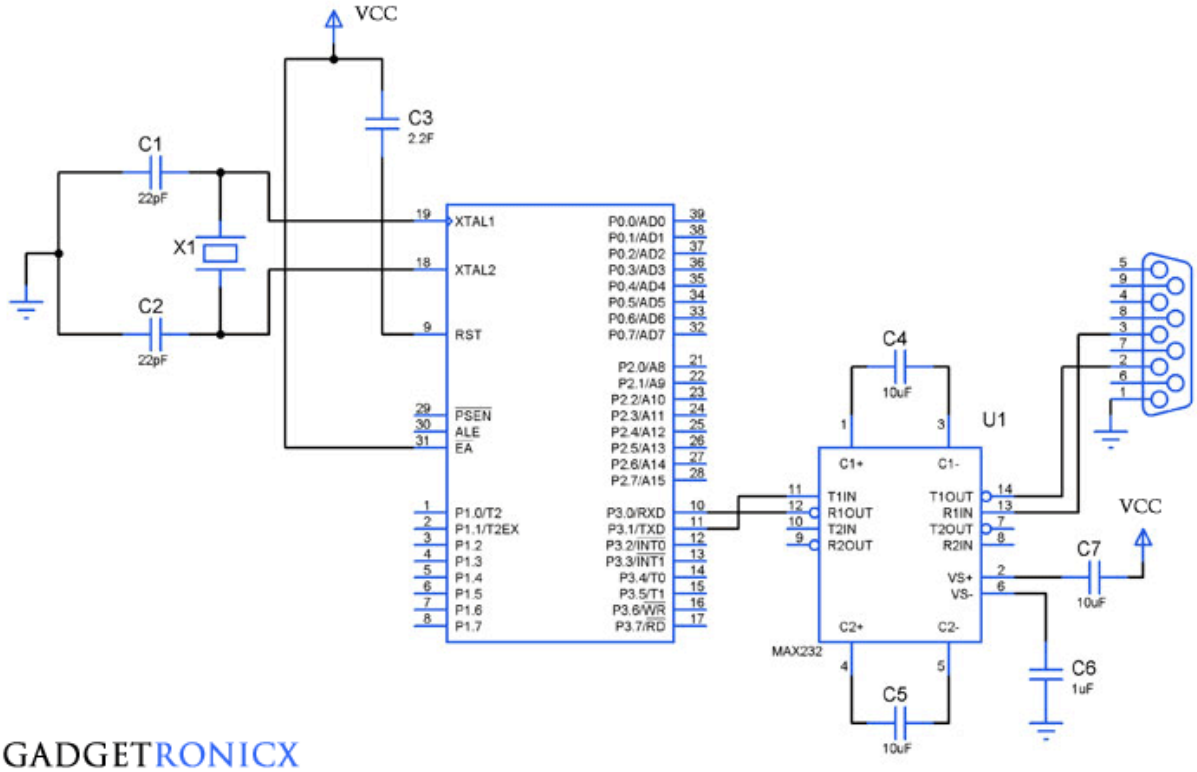
Note that the TF0 flag no need to clear by software as a microcontroller clears it after completing the ISR routine.

Program for the timer interrupt

```
/*
 * 8051_Timer_Interrupt
 * http://www.electronicwings.com
 */
#include<reg51.h>                   /* Include x51 header file */
sbit test = P1^0;                   /* set test pin0 of port1 */

void Timer_init()
{
        TMOD = 0x01;                /* Timer0 mode1 */
        TH0 = 0x4C;                 /* 50ms timer value */
        TL0 = 0x00;
        TR0 = 1;                    /* Start timer0 */
}
```

```
void Timer0_ISR() interrupt 1      /* Timer0 interrupt service routine (ISR) */
{
        test = ~test;              /* Toggle port pin */
        TH0 = 0x4C;                /* 50ms timer value */
        TL0 = 0x00;
}

int main(void)
{
        EA  = 1;                   /* Enable global interrupt */
        ET0 = 1;                   /* Enable timer0 interrupt */
        Timer_init();
        while(1);
}
```

Serial Communication in 8051 Microcontroller



GADGETRONICX

CODE:

```
#include<reg51.h>

void initialize()     // Initialize Timer 1 for serial communication

    {

     TMOD=0x20;    //Timer1, mode 2, baud rate 9600 bps

     TH1=0XFD;      //Baud rate 9600 bps

     SCON=0x50;

     TR1=1;              //Start timer 1

    }

void receive()         //Function to receive serial data

   {

    unsigned char value;

    while(RI==0);       //wait till RI flag is set

    value=SBUF;

    P1=value;

    RI=0;                     //Clear the RI flag

   }

void transmit()       // Funtion to transmit serial data

   {

    SBUF='o';            //Load  'o' in SBUF to transmit

    while(TI==0);      //Wait till TI flag is set or data transmission ends

    TI=0;                   //Clear TI flag

    SBUF='k';

    while(TI==0);

    TI=0;
```

```
    }
void main()
{
 while(1)
   {
    initialize();
    receive();
    transmit();
   }
}
```

RESULT:

DESIGNED ALL PARAMETERS OF 8051 16 BIT PROCESSORS.

EX.NO:3 Design with ARM Processors.

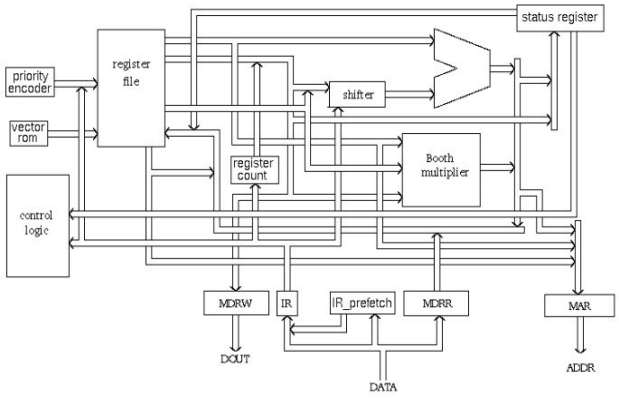
I/O programming, ADC/DAC, Timers, Interrupts,

AIM:

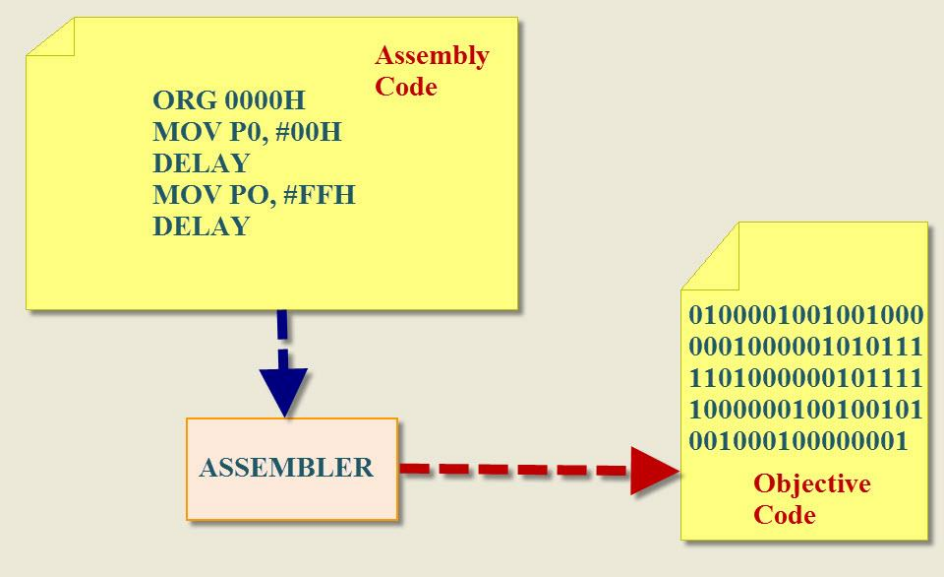
TO DESIGN with ARM Processors.

I/O programming, ADC/DAC, Timers, Interrupts,

ARM BLOCK DIAGRAM



ARM-Cortex Microcontroller Programming



The ARM Microcontroller Assembly Level Programming:

The ARM cortex microcontroller is a 32-bit microcontroller therefore all instructions are 32-bit long which is executed in a single cycle. It consists of an instruction set to perform the arithmetic, logical and boolean operations. The ARM is a load-store architecture, then instructions are executed conditionally.

Syntax: Load a // a obtained the value from the place called a // ADD12 // 12 is added to the load value a // Store a // final value is stored in the variable a//

The assembly language programming is developed by the mnemonics such as ADD, SUB, MUL so on but for ARM programming, some extra instructions added such as ADCNES and SWINE, etc.

EX: 1. ORG 0000h
MOV r1, #10
MOV r2, #15
ADD r3, r2, r1 // r3=r2+r1 and the final value stored in r3 register//

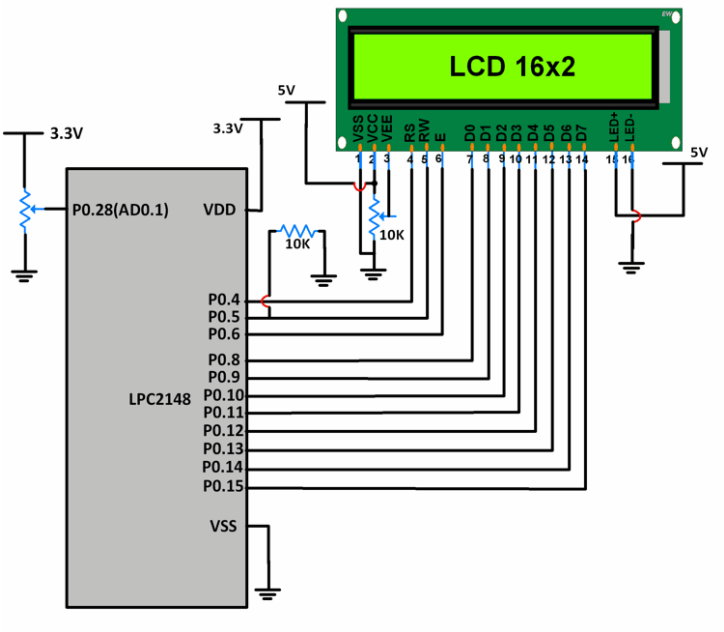
2.ORG 0000h
MOV r1, #10
MOV r2, #15
SUB r3, r2, r1 // r3=r2-r1 and the final value stored in r3 register//

The ARM Coretex-M3 Microcontroller Embedded C Level Programming:

WAP to toggle the single LED through Embedded C language using ARM cortex microcontroller.

```
#include “stm32f10x_gpio.h”
#include “stm32f10x_rcc.h”

GPIO_InitTypeDef GPIO_InitStructure;
int i;
#define LED_PORT GPIOB
Void binky();
Void main()
{
Void binky();
}

void binky(void)
{
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //enable the PORTB pins//
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //set the port frequency//
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP; //set the PORTB in output//
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All ; //enabled all PORTB pins//
GPIO_Init(GPIOB, &GPIO_InitStructure); //initialize the PORTB pins//

while(1){
GPIO_WriteBit(LED_PORT,GPIO_Pin_8,Bit_SET);
for (i=0;i<1000000;i++);
GPIO_WriteBit(LED_PORT,GPIO_Pin_8,Bit_RESET);
for (i=0;i<1000000;i++);

GPIO_WriteBit(LED_PORT,GPIO_Pin_9,Bit_SET);
for (i=0;i<1000000;i++);
GPIO_WriteBit(LED_PORT,GPIO_Pin_9,Bit_RESET);
for (i=0;i<1000000;i++);

GPIO_WriteBit(LED_PORT,GPIO_Pin_10,Bit_SET);
for (i=0;i<1000000;i++);
GPIO_WriteBit(LED_PORT,GPIO_Pin_10,Bit_RESET);
for (i=0;i<1000000;i++);
}

}
```

ADC/DAC USING ARM:

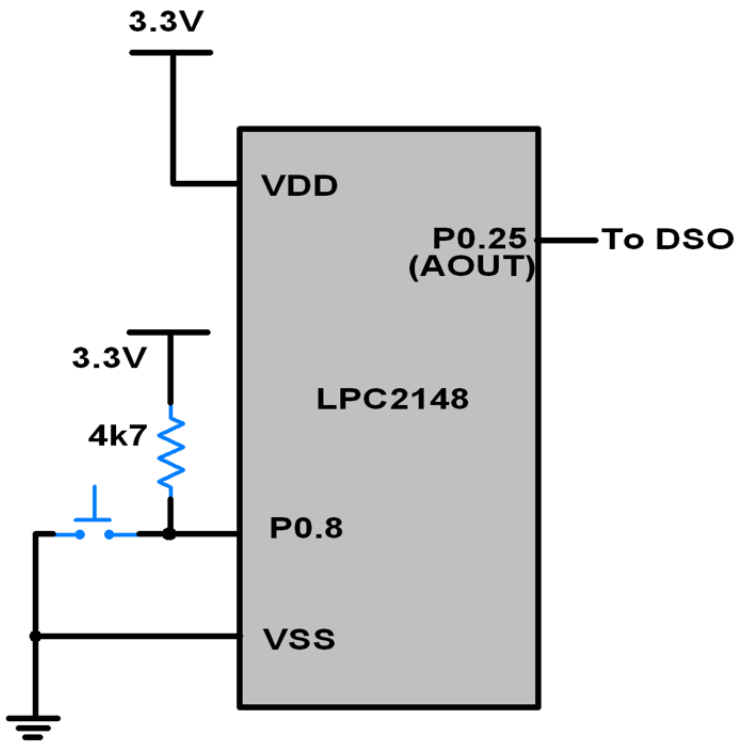


Program

```
/*
  ADC in LPC2148(ARM7)
  http://www.electronicwings.com/arm7/lpc2148-adc-analog-to-digital-converter
*/


#include <lpc214x.h>
#include <stdint.h>
#include "LCD-16x2-8bit.h"
#include <stdio.h>
#include <string.h>



int main(void)
{
        uint32_t result;
        float voltage;
        char volt[18];
        LCD_Init();
        PINSEL1 = 0x01000000; /* P0.28 as AD0.1 */
```

```
        AD0CR = 0x00200402; /* ADC operational, 10-bits, 11 clocks for conversion
*/
        while(1)
        {
                AD0CR = AD0CR | (1<<24); /* Start Conversion */
                while ( !(AD0DR1 & 0x80000000) ); /* Wait till DONE */
                result = AD0DR1;
                result = (result>>6);
                result = (result & 0x000003FF);
                voltage = ( (result/1023.0) * 3.3 ); /* Convert ADC value to equiv
alent voltage */
                LCD_Command(0x80);
                sprintf(volt, "Voltage=%.2f V  ", voltage);
                LCD_String(volt);
                memset(volt, 0, 18);
        }
}
```

DAC:



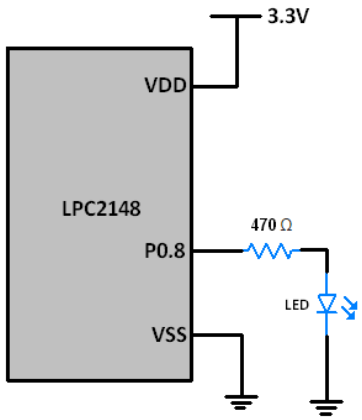
Program

```
/*
  DAC in LPC2148(ARM7)
  http://www.electronicwings.com/arm7/lpc2148-dac-digital-to-analog-converter
*/


#include <lpc214x.h>
#include <stdint.h>



void delay_ms(uint16_t j)
{
    uint16_t x,i;
        for(i=0;i<j;i++)
        {
    for(x=0; x<6000; x++);    /* loop to generate 1 milisecond delay with Cclk = 60MHz */
        }
}


int main (void)
{
  uint16_t value;
        uint8_t i;
        i = 0;
        PINSEL1 = 0x00080000;    /* P0.25 as DAC output */
        IO0DIR = ( IO0DIR & 0xFFFFF0FF ); /* Input pins for switch. P0.8 sine, P0.9 triangular, P0.10 sawtooth, P0.11 square */
        uint16_t sin_wave[42] = { 512,591,665,742,808,873,926,968,998,1017,1023,1017,998,968,926,873,808,742,665,591,512,

                                  436,359,282,216,211,151,97,55,25,6,0,6,25,55,97,151,211,216,282,359,436 };
        while(1)
        {
```

```
                if ( !(IO0PIN & (1<<8)) )        /* If switch for sine wave is pre
ssed */
                {
                                while(i !=42)
                                {
                                        value = sin_wave[i];
                                        DACR = ( (1<<16) | (value<<6) );
                                        delay_ms(1);
                                        i++;
                                }
                                i = 0;
                }
                else if ( !(IO0PIN & (1<<9)) )   /* If switch for triangular wave
is pressed */
                {
                                value = 0;
                                while ( value != 1023 )
                                {
                                        DACR = ( (1<<16) | (value<<6) );
                                        value++;
                                }
                                while ( value != 0 )
                                {
                                        DACR = ( (1<<16) | (value<<6) );
                                        value--;
                                }
                }
                else if ( !(IO0PIN & (1<<10)) )  /* If switch for sawtooth wave is
pressed */
                {

                                value = 0;
                                while ( value != 1023 )
                                {
```

```
                                        DACR = ( (1<<16) | (value<<6) );
                                        value++;
                                }
                }
                else if ( !(IO0PIN & (1<<11)) )  /* If switch for square wave is p
ressed */
                {
                                value = 1023;
                                DACR = ( (1<<16) | (value<<6) );
                                delay_ms(100);
                                value = 0;
                                DACR = ( (1<<16) | (value<<6) );
                                delay_ms(100);
                }
                else    /* If no switch is pressed, 3.3V DC */
                {
                        value = 1023;
                        DACR = ( (1<<16) | (value<<6) );
                }
        }
}
```

TIMERS:

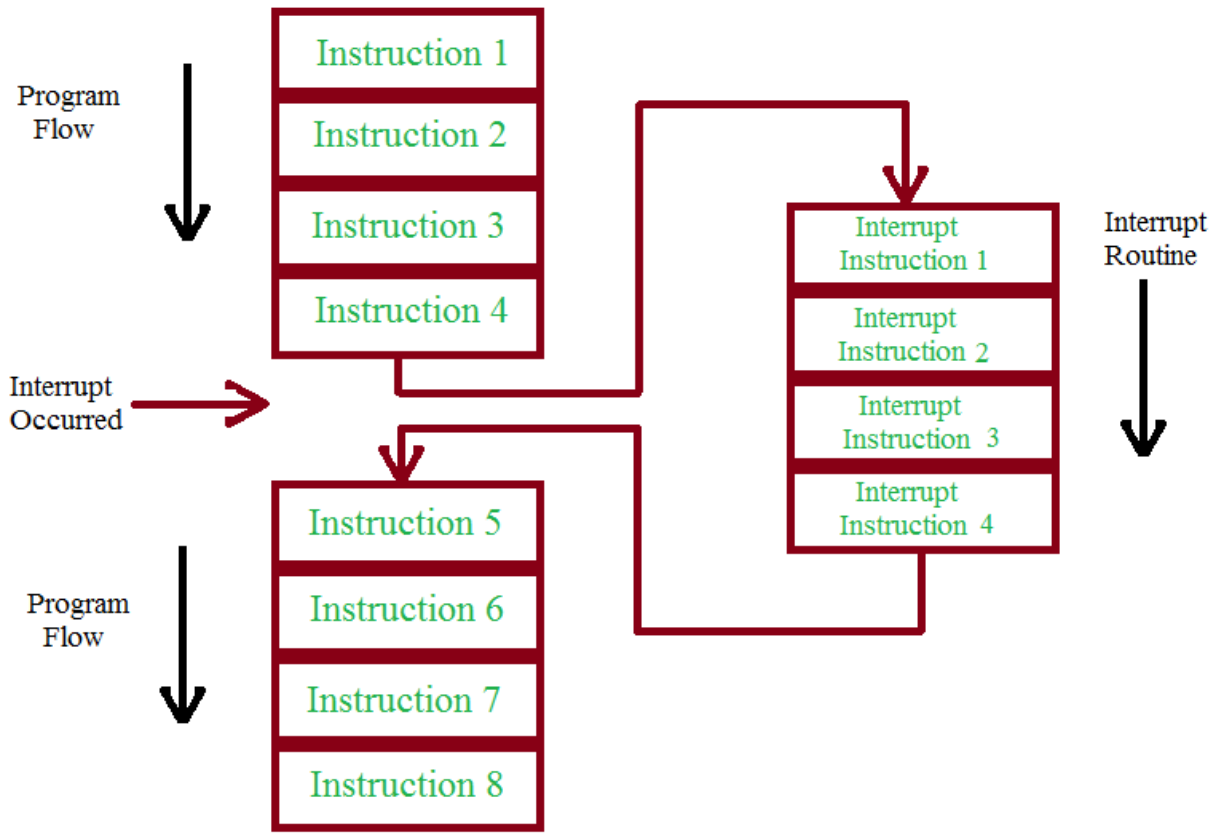


Program

```
/*
  Delay using Timer in LPC2148(ARM7)
  http://www.electronicwings.com/arm7/lpc2148-timercounter
*/


#include  <lpc214x.h>


__irq void T0_ISR (void)
{
        IO0PIN = ( IO0PIN ^ (0x00000100) );      /* Toggle P0.8 pin */
        T0IR = ( T0IR | (0x01) );
        VICVectAddr = 0x00;
}



int main (void)
{
        VPBDIV = 0x00000002; /* For Pclk = 30MHz */
        /* We have configured Cclk=60MHz. Above instruction makes Pclk = Cclk/2 =
30MHz */
        PINSEL0 = PINSEL0 | 0x00000020; /* Configure P0.2 as Capture 0.0 */
        IO0DIR = ( IO0DIR | (0x00000100) ); /* 8 P0.8-P0.15 as output pins for LED
port */
        IO0PIN = IO0PIN | 0x00000100; /* Writing 1 to LED pin P0.8 */
        VICVectAddr0 = (unsigned) T0_ISR; /* T0 ISR Address */
        VICVectCntl0 = 0x00000024; /* Enable T0 IRQ slot */
        VICIntEnable = 0x00000010; /* Enable T0 interrupt */
        VICIntSelect = 0x00000000; /* T0 configured as IRQ */
        T0TCR = 0x02; /* Reset TC and PR */
        T0CTCR = 0x00; /* Timer mode, increment on every rising edge */
        T0PR = 0x1D; /* Load Pre-Scalar counter with 29 (0 to 29 = 30), so that t
imer counts every 1usec */
        T0MR0 = 100000; /* Load timer counter for 100msec delay, 1usec*1000*100 *
/
        T0MCR = 0x0003; /* Interrupt generate on match and reset timer */
```

```
        T0TCR = 0x01; /* Enable timer */


        while (1);
}
```

INTERUPTS:



PROGRAM:

```
#include<lpc214x.h>

#define led 0xFFFFFFFF;

int z=0;

/* Function definition for the timer function */
```

```
void timer (void)

{

T0CTCR = 0x00;

T0PR = 60000-1;

T0TCR = 0x02;

T0MR0 = 1000-1;

T0MCR = (1<<0) | (1<<1);

}

/* Function definition for the delay function */

void delay (int d)

{

T0TCR=0x02;

T0TCR=0x01;

while (T0TC < d);

T0TCR=0x00;

}
```

```
/* ISR for the Interrupt */

__irq void timerISR (void)

{

long int val;

val = T0IR;

if (z==0)

{

IOSET1 = led;

}

else

{

IOCLR1 = led;

}

z=~z;

T0IR = val;

VICVectAddr = 0x00;
```

```
}

/* Function definition for Interrupt Function */

void interrupt (void)

{

VICIntSelect = 0x00;

VICIntEnable |= (1<<4);

VICVectCntl0 = 0x20 | 4;

VICVectAddr0 = (unsigned) timerISR; // Pointer for the ISR

}



int main()

{

PINSEL2 = 0x00;

IODIR1 = led;

/* Enable PLL block and set the CCLK and PCLK to 60 MHz */

PLL0CON = 0x01;
```

```
PLL0CFG = 0x24;

PLL0FEED = 0xaa;

PLL0FEED = 0x55;

while (!(PLL0STAT & 0x00000400));

PLL0CON = 0x03;

PLL0FEED = 0xaa;

PLL0FEED = 0x55;

VPBDIV = 0x01;

/* Call the Timer Function */

timer();

/* Call the Interrupt Function */

interrupt();

/* Enable Timer */

T0TCR = 0x01;

while(1);

}
```

RESULT:

DESIGNED ALL PARAMETERS OF ARM PROCESSORS

EX.NO:4

Study of one type of Real Time Operating Systems (RTOS)

AIM:

TO STUDY OF REAL TIME OPERATING SYSTEM

What is a Real-Time Operating System (RTOS)?

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

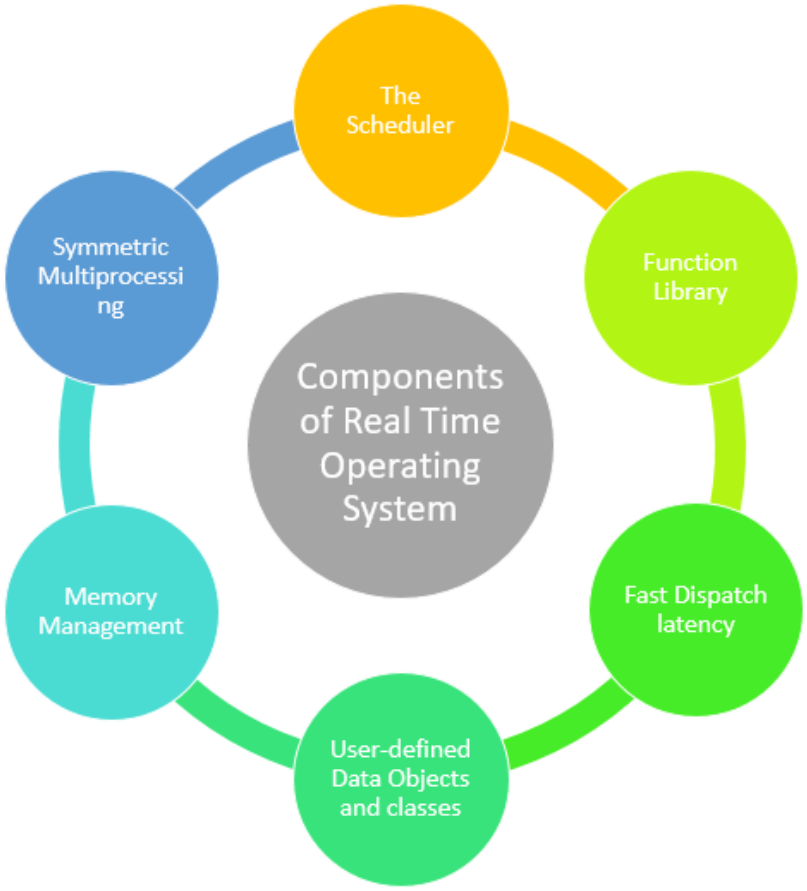
In a RTOS, Processing time requirement are calculated in tenths of seconds increments of time. It is time-bound system that can be defined as fixed time constraints. In this type of system, processing must be done inside the specified constraints. Otherwise, the system will fail.

Why use an RTOS?

Here are important reasons for using RTOS:

- It offers priority-based scheduling, which allows you to separate analytical processing from non-critical processing.
- The Real time OS provides API functions that allow cleaner and smaller application code.
- Abstracting timing dependencies and the task-based design results in fewer interdependencies between modules.
- RTOS offers modular task-based development, which allows modular task-based testing.
- The task-based API encourages modular development as a task, will typically have a clearly defined role. It allows designers/teams to work independently on their parts of the project.
- An RTOS is event-driven with no time wastage on processing time for the event which is not occur

Components of RTOS



Components of Real Time Operating System

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

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

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

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

Fast dispatch latency: It is an interval between the termination of the task that can be identified by the OS and the actual time taken by the thread, which is in the ready queue, that has started processing.

User-defined data objects and classes: RTOS system makes use of programming languages like C or C++, which should be organized according to their operation.

Types of RTOS

Three types of RTOS systems are:

Hard Real Time :

In Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration.

Example: Medical critical care system, Aircraft systems, etc.

Firm Real time:

These type of RTOS also need to follow the deadlines. However, missing a deadline may not have big impact but could cause undesired affects, like a huge reduction in quality of a product.

Example: Various types of Multimedia applications.

Soft Real Time:

Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of time is acceptable. So, deadlines are handled softly by this type of RTOS.

Example: Online Transaction system and Livestock price quotation System.

Terms used in RTOS

Here, are essential terms used in RTOS:

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

Features of RTOS

Here are important features of RTOS:

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

Factors for selecting an RTOS

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

- **Performance**: Performance is the most important factor required to be considered while selecting for a RTOS**.**
- **Middleware**: if there is no middleware support in Real time operating system, then the issue of time-taken integration of processes occurs.

- **Error-free**: RTOS systems are error-free. Therefore, there is no chance of getting an error while performing the task.
- **Embedded system usage**: Programs of RTOS are of small size. So we widely use RTOS for embedded systems.
- **Maximum Consumption**: we can achieve maximum Consumption with the help of RTOS.
- **Task shifting**: Shifting time of the tasks is very less.
- **Unique features**: A good RTS should be capable, and it has some extra features like how it operates to execute a command, efficient protection of the memory of the system, etc.
- **24/7 performance**: RTOS is ideal for those applications which require to run 24/7.

Difference between in GPOS and RTOS

Here are important differences between GPOS and RTOS:

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

Priority inversion remain unnoticed	No predictability guarantees

Applications of Real Time Operating System

Real-time systems are used in:

- Airlines reservation system.
- Air traffic control system.
- Systems that provide immediate updating.
- Used in any system that provides up to date and minute information on stock prices.
- Defense application systems like RADAR.
- Networked Multimedia Systems
- Command Control Systems
- Internet Telephony
- Anti-lock Brake Systems
- Heart Pacemaker

Disadvantages of RTOS

Here, are drawbacks/cons of using RTOS system:

- RTOS system can run minimal tasks together, and it concentrates only on those applications which contain an error so that it can avoid them.
- RTOS is the system that concentrates on a few tasks. Therefore, it is really hard for these systems to do multi-tasking.
- Specific drivers are required for the RTOS so that it can offer fast response time to interrupt signals, which helps to maintain its speed.
- Plenty of resources are used by RTOS, which makes this system expensive.
- The tasks which have a low priority need to wait for a long time as the RTOS maintains the accuracy of the program, which are under execution.
- Minimum switching of tasks is done in Real time operating systems.
- It uses complex algorithms which is difficult to understand.
- RTOS uses lot of resources, which sometimes not suitable for the system.

Summary:

- RTOS is an operating system intended to serve real time application that process data as it comes in, mostly without buffer delay.
- It offers priority-based scheduling, which allows you to separate analytical processing from non-critical processing.
- Important components of RTOS system are: 1)The Scheduler, 2) Symmetric Multiprocessing, 3) Function Library, 4) Memory Management, 5) Fast dispatch latency, and 6) User-defined data objects and classes
- Three types of RTOS are 1) Hard time 2) Soft time ,and 3) Firm time
- RTOS system occupy very less memory and consume fewer resources
- Performance is the most important factor required to be considered while selecting for a RTOS.
- General-Purpose Operating System (GPOS) is used for desktop PC and laptop while Real-Time Operating System (RTOS) only applied to the embedded application.
- Real-time systems are used in Airlines reservation system, Air traffic control system,etc.
- The biggest drawback of RTOS is that the system only concentrates on a few tasks.

RESULT:

STUDIED ONE TYPE OF REAL TIME OPERATING SYSTEMS.

EX.NO:5

Electronic Circuit Design of sequential, combinational digital circuits using CAD Tools

AIM:

TO DESIGN Electronic Circuit Design of sequential, combinational digital circuits using CAD Tools

COMPONENTS REQUIRED:

CADD SIMULATOR

COMBINATIONAL CIRCUITS:

1. Combinational Circuits**:**
These circuits developed using AND, OR, NOT, NAND and NOR logic gates. These logic gates are building blocks of combinational circuits. A combinational circuit consists of input variables and output variables. Since these circuits are not dependent upon previous input to generate any output, so are combinational logic circuits. A combinational circuit can have an n number of inputs and m number of outputs. In combinational circuits, the output at any time is a direct function of the applied external inputs.

Figure - Block diagram of Combinational circuit

2. Sequential circuits**:**
A sequential circuit is specified by a time sequence of inputs, outputs, and internal states. The output of a sequential circuit depends not only the combination of present inputs but also on the previous outputs. Unlike combinational circuits, sequential circuits include memory elements with combinational circuits.

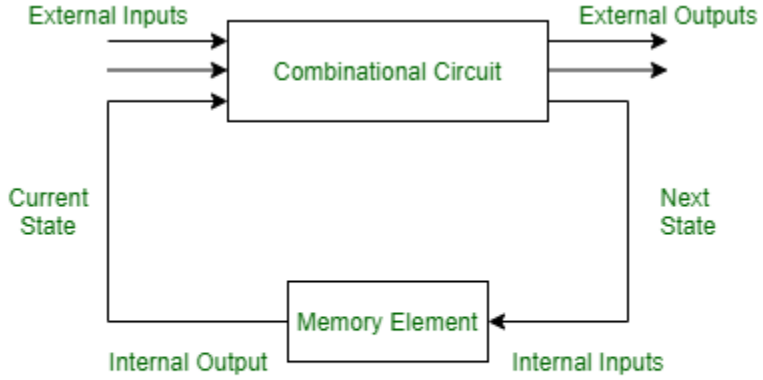
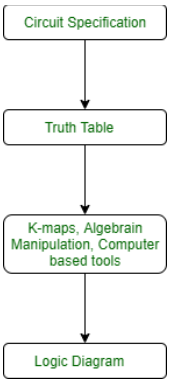
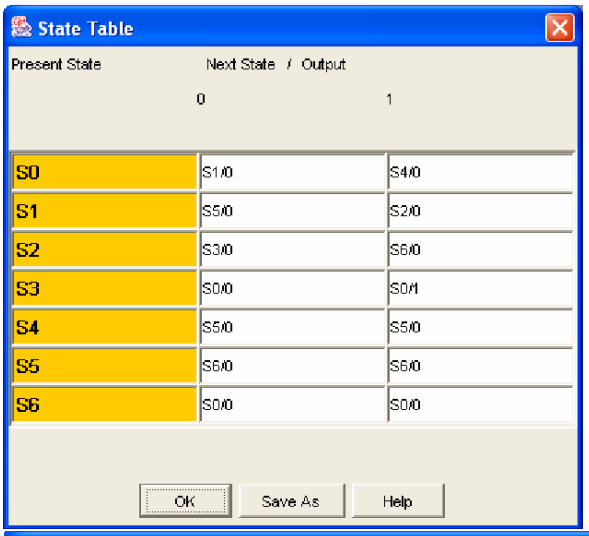


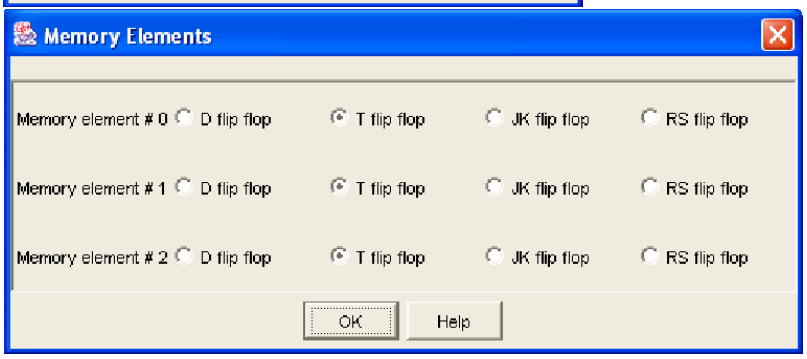
Figure - Sequential Circuit

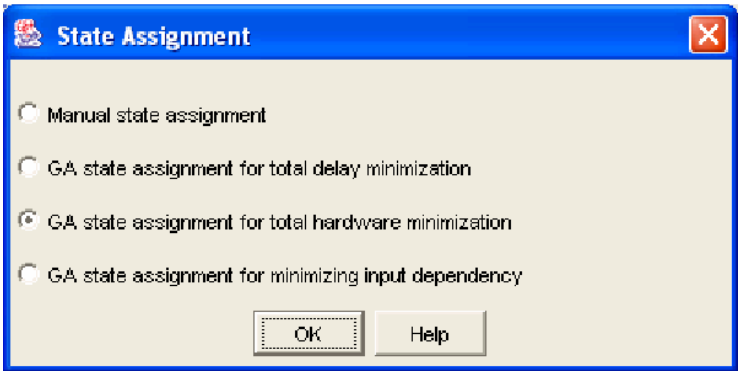
1. Analysis and Design of Combinational circuits**:**

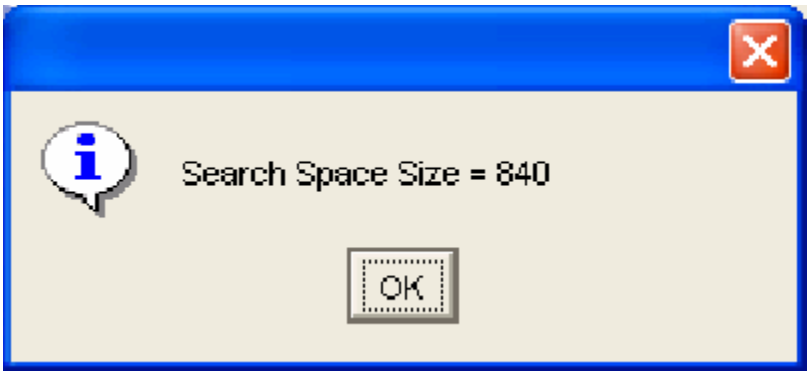

Circuit Specification
Truth Table
K-maps, Algebrain Manipulation, Computer based tools
Logic Diagram

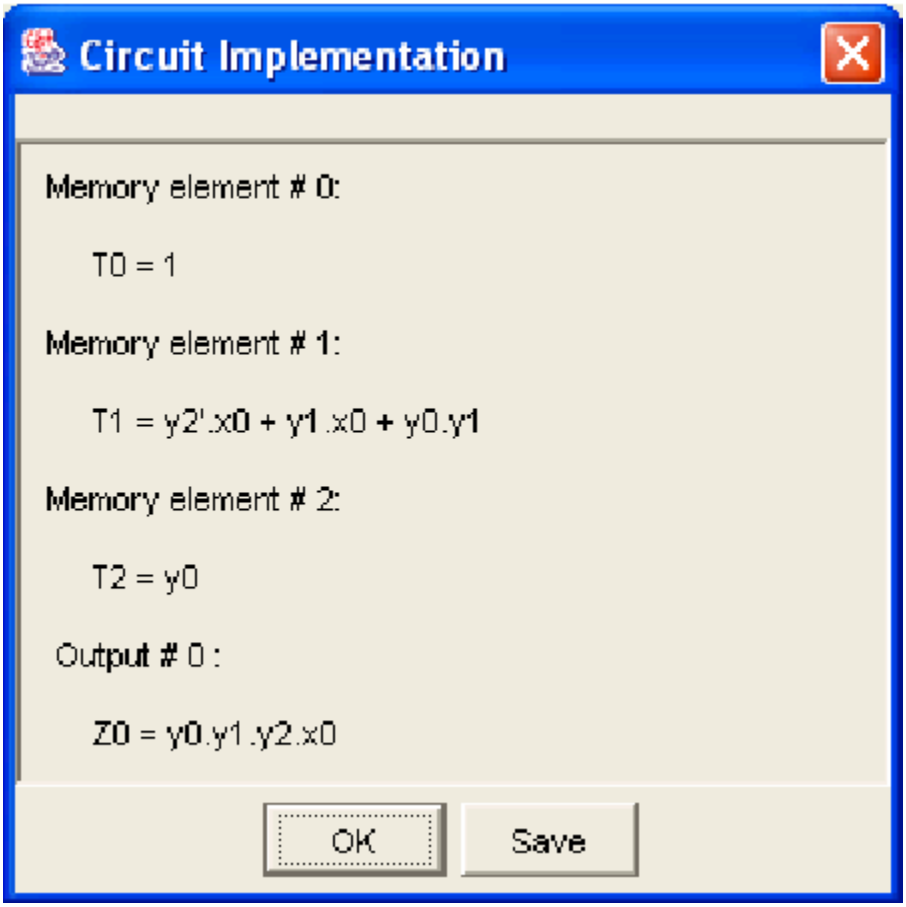
CAD TOOLS:

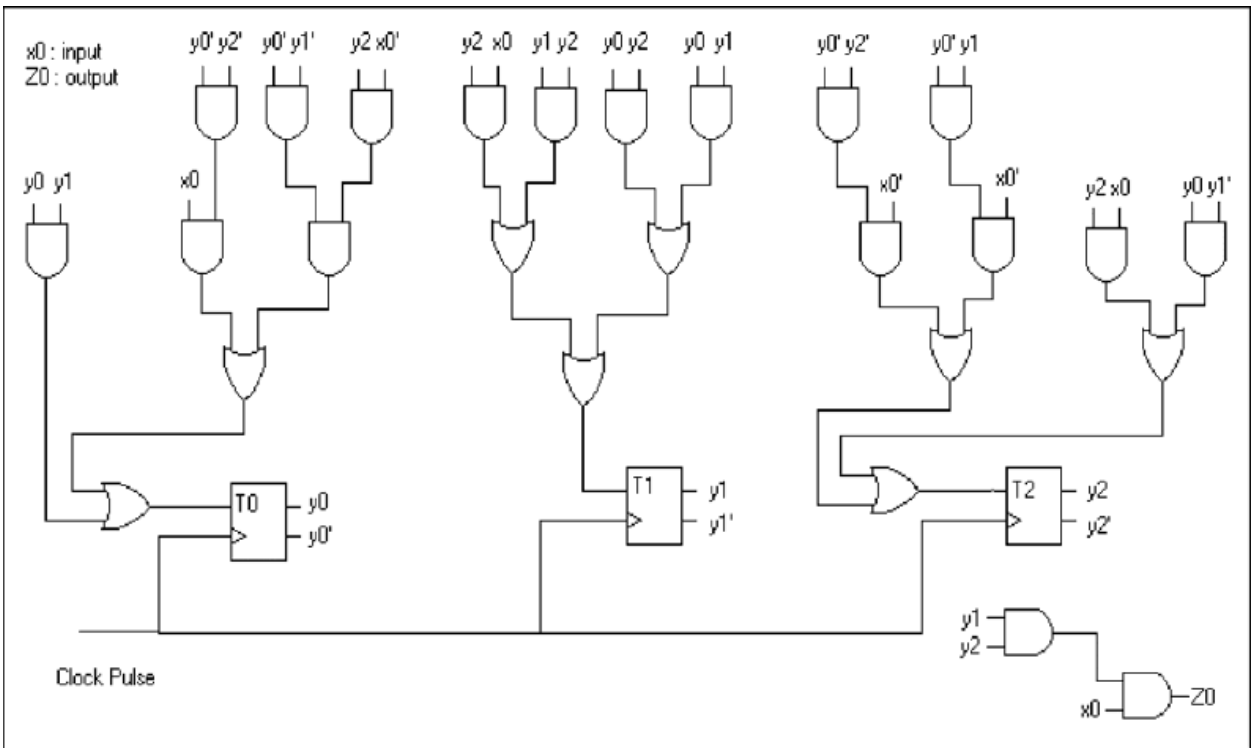

State Table
Present State
Next State / Output
0
1
S0
S1/0
S4/0
S1
S5/0
S2/0
S2
S3/0
S6/0
S3
S0/0
S0/1
S4
S5/0
S5/0
S5
S6/0
S6/0
S6
S0/0
S0/0
OK
Save As
Help


Memory Elements
Memory element # 0
D flip flop
T flip flop
JK flip flop
RS flip flop
Memory element # 1
D flip flop
T flip flop
JK flip flop
RS flip flop
Memory element # 2
D flip flop
T flip flop
JK flip flop
RS flip flop
OK
Help


State Assignment
Manual state assignment
GA state assignment for total delay minimization
GA state assignment for total hardware minimization
GA state assignment for minimizing input dependency
OK
Help


Search Space Size = 840
OK


Circuit Implementation
Memory element # 0:
T0 = 1
Memory element # 1:
T1 = y2'.x0 + y1.x0 + y0.y1
Memory element # 2:
T2 = y0
Output # 0 :
Z0 = y0.y1.y2.x0
OK
Save



RESULT:

Electronic Circuit Design of sequential, combinational digital circuits using CAD Tools was designed

Successfully.

EX.NO 6:

Simulation of digital controllers using MATLAB/LabVIEW

Aim:

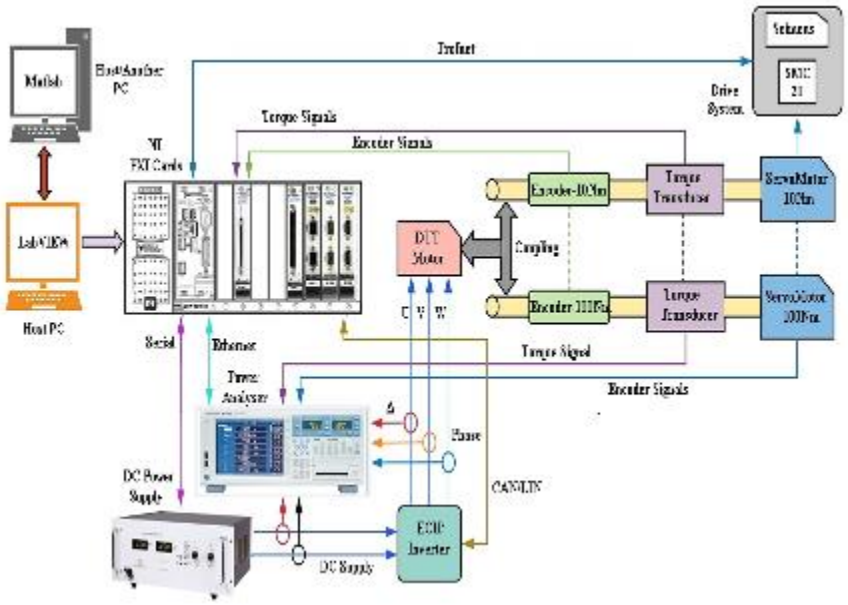
To Simulation of digital controllers using MATLAB/LabVIEW

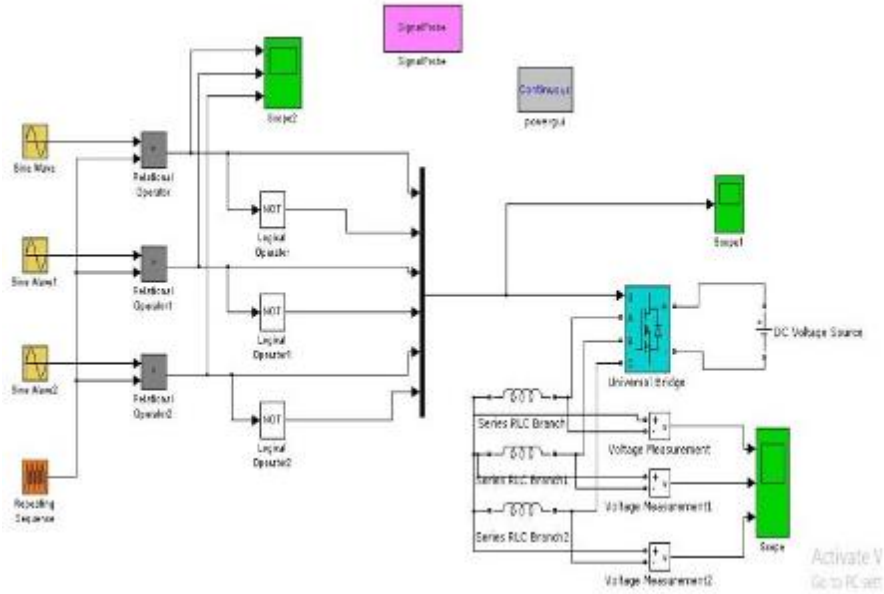
Introduction

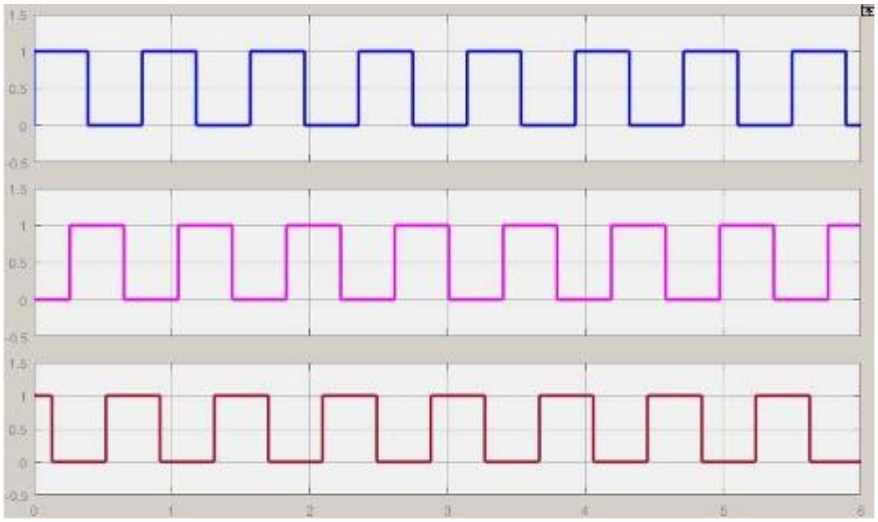
Traditionally offline simulation has been used extensively to investigate the performance of an electrical system because of its minimal effort and low cost. But, due to the computational resources and run time restrictions, the emulation precision and reliability suffer from various levels of model reductions [1]. So offline simulation does not replicate the real behavior of the electrical system.

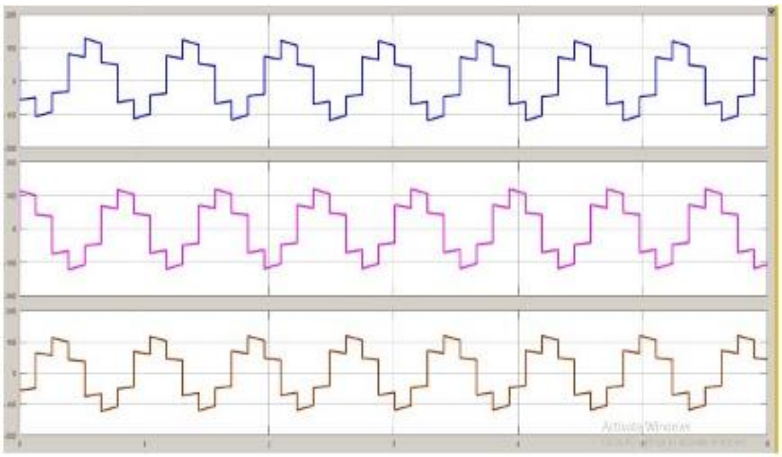
HIL simulation is acknowledged as a commercial and competent industrial prototyping system for modeling power system controllers [2]. Digital real-

time simulation (DRTS) of the electrical system is the replication of output (voltage/currents), with the required precision, which symbolizes the response of the real system being modeled [3]. The technology of power electronics is developing complex and multi-disciplinary field of electrical engineering. The primary reason behind this is an advancement in power semiconductor devices. This development trend in power electronics makes new challenges to the conventional power system and power electronic engineers [4]. Power electronic engineers are progressively interested in the modeling control system for power electronics designs. A real-time simulation platform allows engineers to investigate their control strategies by importing the controller model into a real-time platform. PHIL simulations enable researchers to test plants under dangerous conditions, such as faults, that could otherwise damage expensive equipment [5]. Moreover, HIL investigation permits the model of a novel device to be examined under a broad scope of realistic conditions repeatedly, securely, and economically.




Sine Wave
Sine Wave1
Sine Wave2
Repeating Sequence
Relational Operator
Relational Operator1
Relational Operator2
NOT
Logical Operator
Logical Operator1
Logical Operator2
Demux
Demux1
Universal Bridge
DC Voltage Source
Series RLC Branch
Series RLC Branch1
Series RLC Branch2
Voltage Measurement
Voltage Measurement1
Voltage Measurement2
Scope





RESULT:

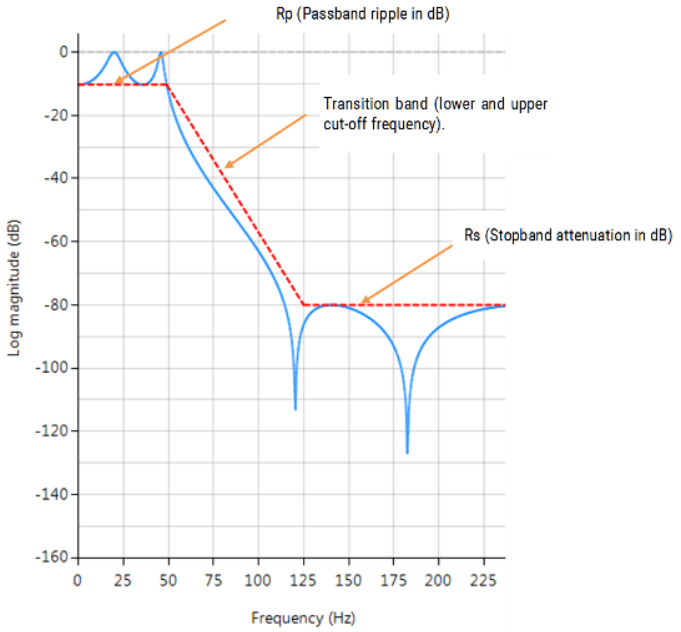
Simulation of digital controllers using MATLAB/LabVIEW is generated successfully.

EX.NO:7

Programming with DSP processors for Correlation, Convolution, Arithmetic adder, Multiplier, Design of Filters - FIR based , IIR based

AIM:

To develop a program for DSP processors for Correlation, Convolution, Arithmetic adder, Multiplier, Design of Filters - FIR based , IIR based

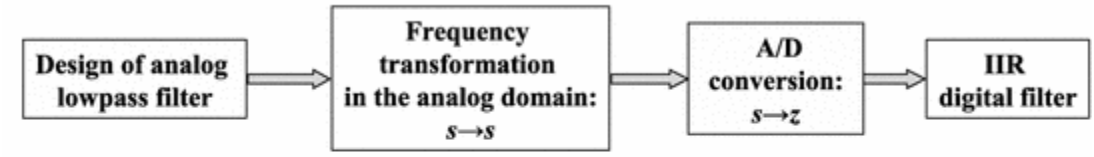


If the application requires real-time processing, causal filters are the only choice for implementation. Following consequences must be considered if causality is desired.

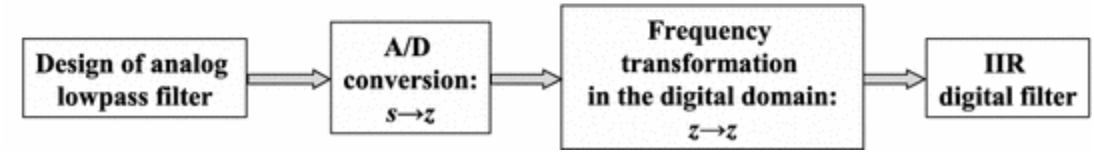
Ideal filters with finite bands of zero response (example: brick-wall filters), cannot be implemented using causal filter structure. A direct consequence of causal filter is that the response cannot be ideal. Hence, we must design the filter that provides a close approximation to the desired response . If tolerance specification is given, it has to be met.
For example, in the case of designing a low pass filter with given passband frequency (ω_P) and stopband frequencies (ω_S), additional tolerance specifications like allowable passband ripple factor (δ_P), stopband ripple factor (δ_S) need to be considered for the

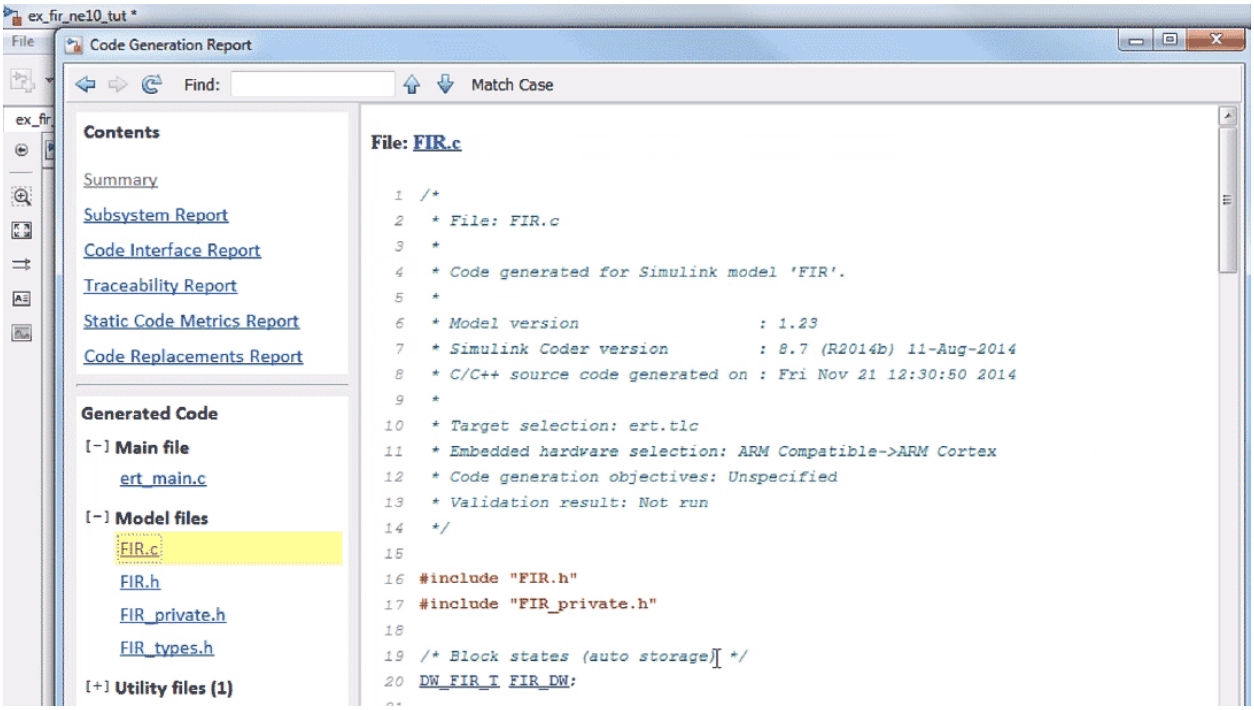
design,. Therefore, the practical filter design involves choosing $\omega_P, \omega_S, \delta_P$ and δ_S and then designing the filter with $N, M, \{a_k\}$ and $\{b_k\}$ that satisfies all the given requirements/responses. Often, iterative procedures may be required to satisfy all the above (example: Parks and McClellan algorithm used for designing optimal causal FIR filters ***[1]***).

Approach I



Approach II





RESULT:

Programming with DSP processors for Correlation, Convolution, Arithmetic adder, Multiplier, Design of Filters - FIR based , IIR based are developed.

EX.NO:8

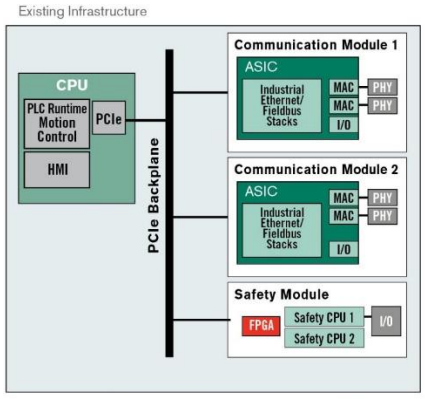
Design with Programmable Logic Devices using Xilinx/Altera FPGA and CPLD Design and Implementation of simple Combinational/Sequential Circuits

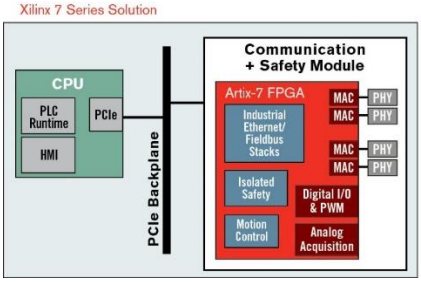
AIM;

Design with Programmable Logic Devices using Xilinx/Altera FPGA and CPLD Design and Implementation of simple Combinational/Sequential Circuits

Programmable Logic Controller

Employing the Artix®-7 FPGA and Xilinx IP solutions enables a smaller form factor programmable logic controller (PLC) with greater flexibility, lower BOM cost and lower total power consumption compared to traditional architectures. Serving as a companion device to the main processor, the FPGA replaces communication expansion modules.





Programmable Logic Controllers (PLC) grew from their classical role as a dedicated layer in the Automation Pyramid to flexible Intelligent Edge nodes with versatile and customizable functionality. While SW-runtimes for IEC 61131 guarantee traditional PLC functions, network offloading, image processing and Digital Twins inside the same component increase the value of this class of industrial products.

That's where intelligent and adaptive SOCs like Xilinx Zynq UltraScale+ MPSoC and Xilinx Versal come into the play. The applications processors inside the SOC communicate with the Programmable Logic through a high-performance architecture. Logic can be used for custom logic as well as for Deep Learning using Neural Networks which sit in the FPGA portion and run at optimal power to performance ratio. Typical applications to enrich a PLC, a PAC and an IPC are:

- Real time networking with scalable number of ports (TSN, UDP IP, Industrial Ethernet)
- Predictive Maintenance
- Comprehensive data acquisition with pre-processing of data from the field using chip-internal RAM and DSP resources
- Fast and deterministic control loops
- Interfacing to optical sensors and optimized Image Signal Processing including Sensor Fusion
- Hypervisor support for isolation of SW applications
- Cyber and physical security related functions tied to FIPS140 and IEC62443 including hardware and software field updates for a solid security lifecycle management

Because Xilinx's development tools are safety certified and because Xilinx provides in-depth information including reports from assessors about Zynq UltraScale+ MPSoC on the Safety Lounge, this architecture can be used for Safe PLC designs.

The PLC (Programmable Logic Controller) has been widely used to implement real-time controllers in nuclear RPSs (Reactor Protection Systems). Increasing complexity and maintenance cost, however, are now demanding more powerful and cost-effective implementation such as FPGA (Field-Programmable Gate Array). Abandoning all experience and knowledge accumulated over the decades and starting an all-new development approach is too risky for such safety-critical systems. This paper proposes an RPS software development process with a platform change from PLC to FPGA, while retaining all outputs from the established development. This paper transforms FBD designs of the PLC-based software development into a behaviorally-equivalent Verilog program, which is a starting point of a typical FPGA-based hardware development. We expect that the proposed software development process can bridge the gap between two software developing approaches with different platforms, such as PLC and FPGA. This paper also demonstrates its effectiveness using an example of a prototype version of a real-world RPS in Korea.

- **Previous** article in issue
- **Next** article in issue

KEYWORDS

Embedded Software Development

PLC

FPGA

FBD

Verilog

Program Transformation

1. INTRODUCTION

A safety grade PLC is an industrial digital computer used to develop safety-critical systems such as RPS (Reactor Protection System) for nuclear power plants. The software loaded into a PLC is designed using specific PLC programming languages [1] such as FBD (Function Block Diagram) and LD (Ladder Diagram), which are then translated and compiled into a C program and executable machine code of a specific target PLC.

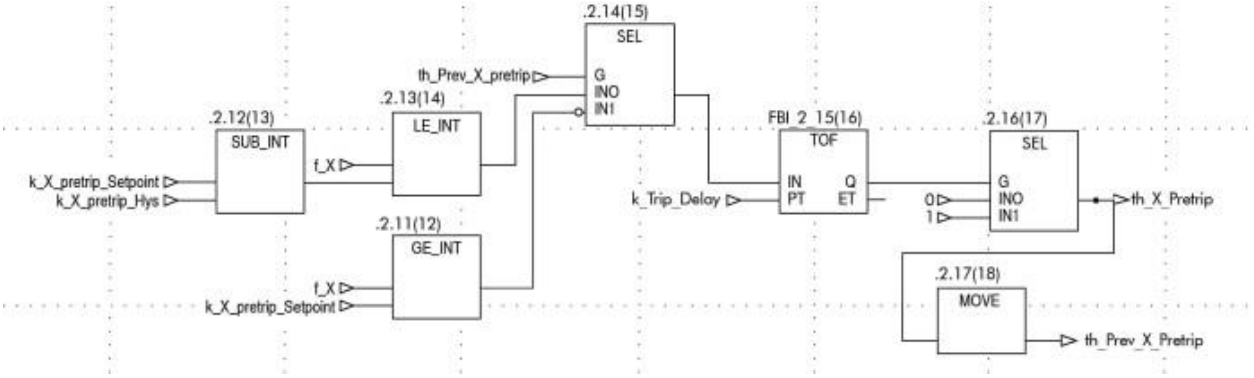
Since the complexity of new RPSs and the maintenance cost of old RPSs have increased rapidly, we need to find an efficient alternative for the PLC-based RPS implementation. One solution [2, 3] proposed is to replace PLC with FPGA, which can provide a powerful computation with lower hardware cost. However, it is a challenge for software engineers in the nuclear domain to abandoning all experience, knowledge and practice, based on PLC and start an FPGA-based development from scratch. Such change is also too risky from the viewpoint of safety. We need to transit to the new development approach safely and seamlessly, allowing all software engineers to become familiar with the processes and procedures required for a proper set up.

This paper proposes an RPS software development process with a change in the hardware platform from PLC to FPGA.

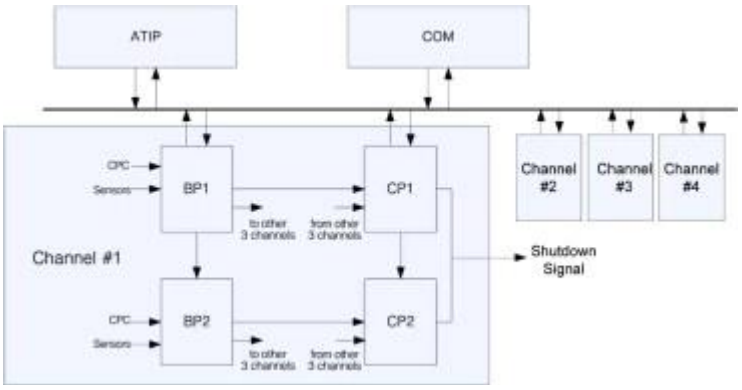
Itprovides the fundamentals for a seamless transition from PLC-based to FPGA-based development. We propose the use of FBD programs in the design phase of the existing PLC-based software development to produce the Verilog program, which is a starting point of typical FPGA developments. The '*FBDtoVerilog*' mechanically transforms FBDs into behaviorally-equivalent [38] Verilog programs, and all V&V activities and safety analyses applied beforehand are still valid in the new hardware platform – FPGA. In order to demonstrate the effectiveness of the proposed approach, we performed a case study with an example of a preliminary version of RPS in a Korean nuclear power plant, from software requirements of PLC to netlists of FPGA.

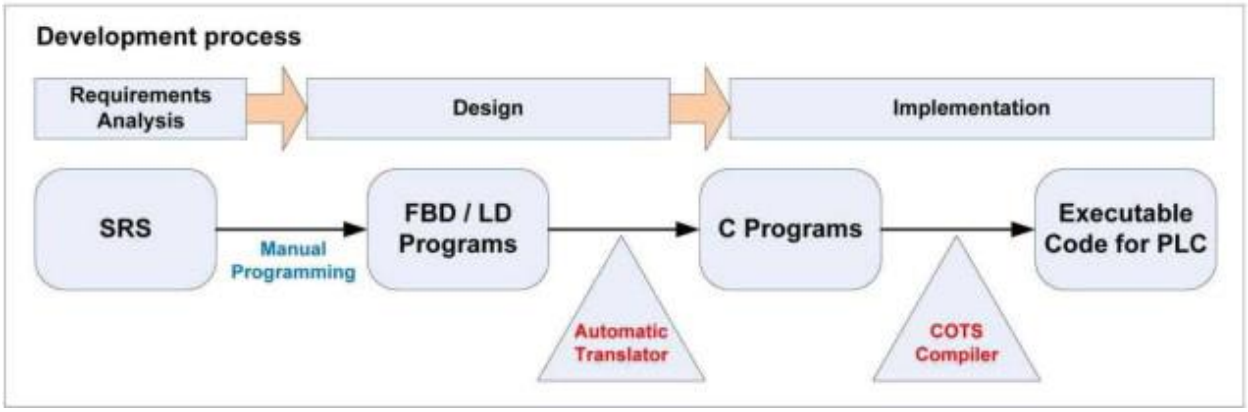
The paper is organized as follows: Section 2 introduces the FBD and Verilog programming languages, which are pertinent to our discussion. It also includes a brief introduction to PLC and FPGA. Section 3 explains the PLC-based RPS development in comparison with the FPGA-based development. It also introduces a typical RPS

architecture to aid understanding. Section 4 proposes an RPS development process with changed platform from PLC to FPGA. Section 5 shows a case study, pointing out how the requirements and FBD designs of the existing PLC-based RPS software development can be effectively transformed into a starting point of the FPGA-based development. Related researches are surveyed in Section 6, and Section 7 concludes the paper and gives remarks on future research.




```
(1)  typedef enum {T0, T1, T2, T3, T4, T5} timer_state;
(2)  'define k_Pretrip_Setpoint 30;
(3)  'define k_X_Pretrip_Hys 10;

(4)  module th_X_Pretrip(clk, f_X, th_X_Pretrip);
(5)    input clk;
(6)    input [0:6] f_X;
(7)    output th_X_Pretrip;

(8)    reg th_Prev_X_Pretrip;
(9)    timer_state reg timer;

(10)   initial th_Prev_X_Pretrip = 1;
(11)   initial timer = T0;

(12)   assign th_X_Pretrip =
(13)     (th_Prev_X_Pretrip == 0 && f_X <= 'k_Pretrip_Setpoint - 'k_X_Pretrip_Hys)?1:
(14)     (th_Prev_X_Pretrip == 0 && f_X > 'k_Pretrip_Setpoint - 'k_X_Pretrip_Hys && timer == T5)?0:
(15)     (th_Prev_X_Pretrip == 0 && f_X > 'k_Pretrip_Setpoint - 'k_X_Pretrip_Hys && timer != T5)?1:
(16)     (th_Prev_X_Pretrip == 1 && f_X < 'k_Pretrip_Setpoint)?1:
(17)     (th_Prev_X_Pretrip == 1 && f_X >= 'k_Pretrip_Setpoint && timer == T5)?0:
(18)     (th_Prev_X_Pretrip == 1 && f_X >= 'k_Pretrip_Setpoint && timer != T5)?1:0;

(19)   always @(posedge clk) begin
(20)       if(f_X >= 'k_Pretrip_Setpoint) begin
(21)           case (timer)
(22)                T0: timer = T1;
(23)                T1: timer = T2;
(24)                T2: timer = T3;
(25)                T3: timer = T4;
(26)                T4: timer = T5;
(27)                T5: timer = T5;
(28)           endcase
(29)       else
(30)           timer = T0;
(31)       end

(32)       th_Prev_X_Pretrip = th_X_Pretrip;

(33)   end

(44) endmodule
```





Functional Safety of *FBDtoVerilog* : Functional safety and correctness of the '*FBDtoVerilog*' translator should be demonstrated thoroughly in various ways. Whereas it was a supporting CASE tool for formal verification using SMV, VIS and HW-CBMC, it is now a development tool bridging PLC-based and FPGA-based development. More rigorous demonstration of functional safety and correctness is highly required.

- ***Functional Safety of FPGA Synthesis Tools*** : Functional safety and correctness of FPGA synthesis tools (e.g., '*Xilinx ISE Design Suite*' and '*Altera Quartus II*') is also one of key issues in overcoming the wide-spread commercialization of the FPGA-based RPS development

RESULT:

Programmable Logic Devices using Xilinx/Altera FPGA and CPLD Design and Implementation of simple Combinational/Sequential Circuits are designed suceessfully.

EX.NO:9

Network Simulators Simple wired/ wireless network simulation using NS2

Aim :

To study Network Simulators Simple wired/ wireless network simulation using NS2

Theory

The wireless simulation described in section IX, supports multi-hop ad-hoc networks or wireless LANs. But we may need to simulate a topology of multiple LANs connected through wired nodes, or in other words we need to create a wired-cum-wireless topology.

In this section we are going to extend the simple wireless topology created in section IX to create a mixed scenario consisting of a wireless and a wired domain, where data is exchanged between the mobile and non-mobile nodes. We are going to make modifications to the tcl script called wireless1.tcl created in section IX.2 and name the resulting wired-cum-wireless scenario file wireless2.tcl.

For the mixed scenario, we are going to have 2 wired nodes, W(0) and W(1), connected to our wireless domain consisting of 3 mobilenodes (nodes 0, 1 & 2) via a base-station node, BS. Base station nodes are like gateways between wireless and wired domains and allow packets to be exchanged between the two types of nodes. For details on base-station node please see section 2 (wired-cum-wireless networking) of chapter 15 of ns notes&doc (now renamed as ns Manual). Fig1. shows the topology for this example described above.

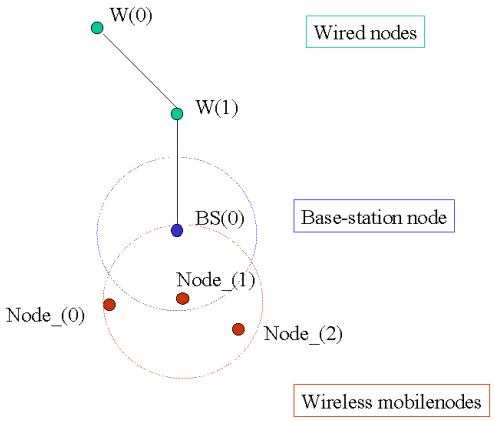


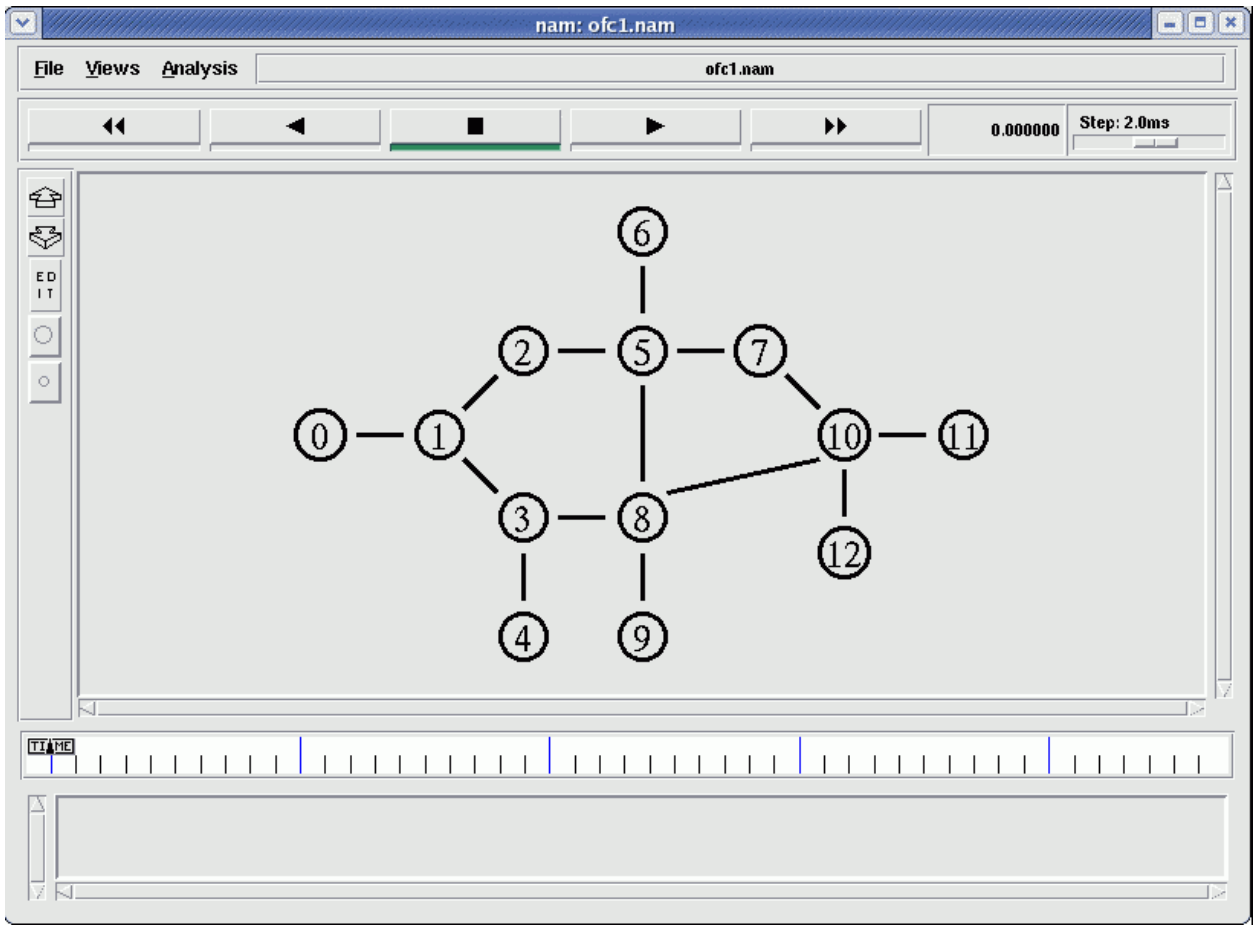
Fig1. Topology for wired-cum-wireless simulation example.

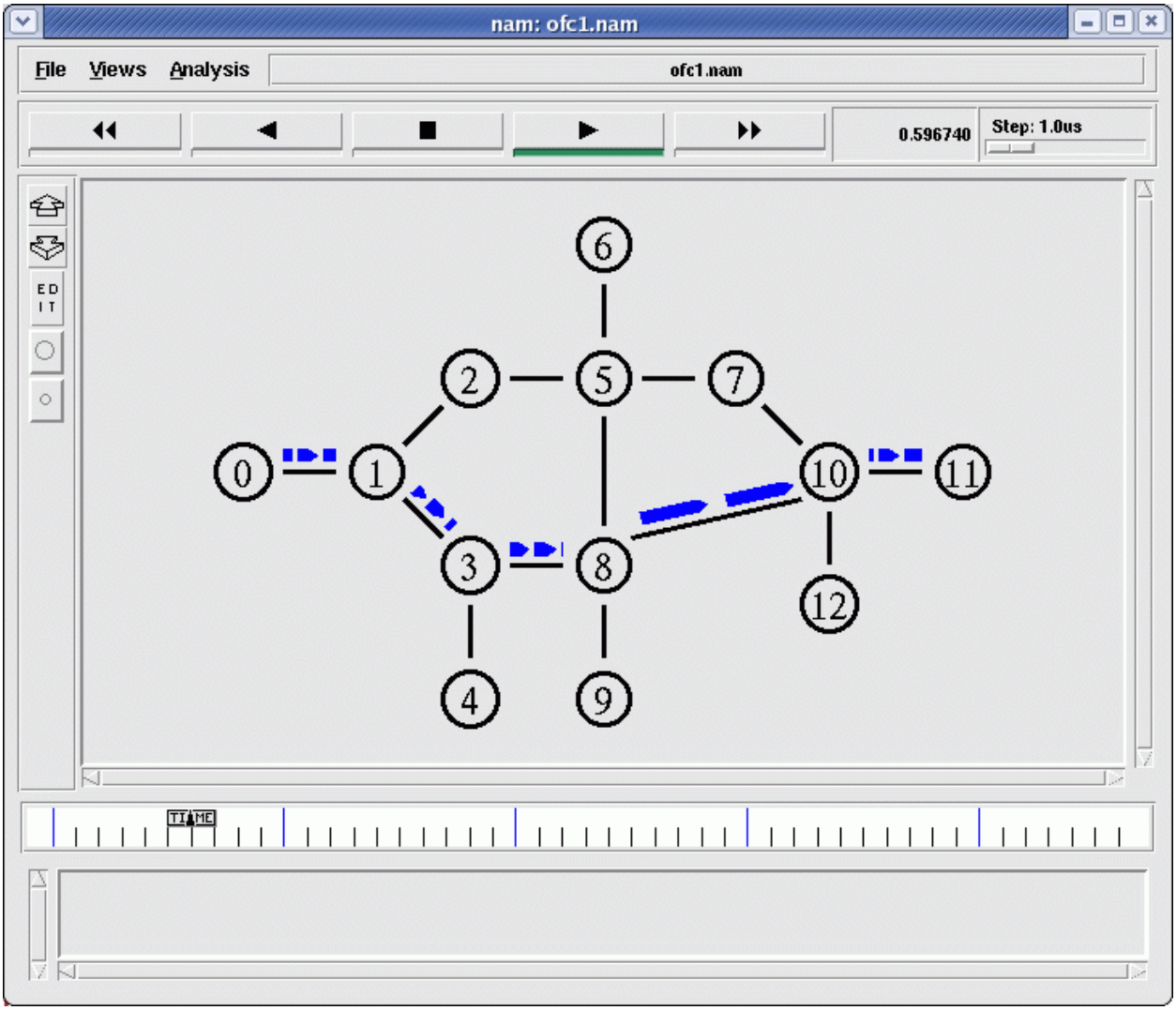
```
# create four nodes
set node1 [$ns node]
set node2 [$ns node]
set node3 [$ns node]
set node4 [$ns node]

# create links between the nodes
$ns duplex-link $node1 $node3 2Mb 20ms DropTail
$ns duplex-link $node2 $node3 2Mb 20ms DropTail
$ns duplex-link $node3 $node4 1Mb 20ms DropTail
$ns queue-limit $node3 $node4 4

# set the display layout of nodes and links for nam
$ns duplex-link-op $node1 $node3 orient right-down
$ns duplex-link-op $node2 $node3 orient right-up
$ns duplex-link-op $node3 $node4 orient right

# define different colors for nam data flows
$ns color 0 Green
$ns color 1 Blue
$ns color 2 Red
$ns color 3 Yellow

# monitor the queue for the link between node 2 and node 3
$ns duplex-link-op $node3 $node4 queuePos 0.5
```

- Define traffic patterns by creating agents, applications and flows. In NS2, packets are always sent from one agent to another agent or a group of agents. In addition, we need to associate these agents with nodes.

```
# TCP traffic source
# create a TCP agent and attach it to node node1
set tcp [new Agent/TCP]
$ns attach-agent $node1 $tcp
$tcp set fid_ 1
$tcp set class_ 1

# window_ * (packetsize_ + 40) / RTT
$tcp set window_ 30
$tcp set packetSize_ $packetSize

# create a TCP sink agent and attach it to node node4
set sink [new Agent/TCPSink]
$ns attach-agent $node4 $sink

# connect both agents
$ns connect $tcp $sink

# create an FTP source "application";
set ftp [new Application/FTP]
$ftp attach-agent $tcp
```

```
# UDP traffic source
# create a UDP agent and attach it to node 2
set udp [new Agent/UDP]
$udp set fid_ 2          # red color
$ns attach-agent $node2 $udp

# create a CBR traffic source and attach it to udp
set cbr [new Application/Traffic/CBR]
$cbr set packetSize_ $packetSize
$cbr set rate_ 0.25Mb
$cbr set random_ false
$cbr attach-agent $udp

# creat a Null agent (a traffic sink) and attach it to node 4
set null [new Agent/Null]
$ns attach-agent $node4 $null
$ns connect $udp $null
```






```
set ns_   [new Simulator]     ;# Create a NS simulator object

  $ns_ node-config \
                   -llType          LL
                   -ifqType         "Queue/DropTail/PriQueue"
                   -ifqLen          50
                   -macType         Mac/802_11
                   -phyType         "Phy/WirelessPhy"

                  -addressingType flat or hierarchical or expanded
                   -adhocRouting    DSDV or DSR or TORA
                   -propType        "Propagation/TwoRayGround"
                   -antType         "Antenna/OmniAntenna"
                   -channelType     "Channel/WirelessChannel"
                   -topoInstance    $topo
                   -energyModel     "EnergyModel"
                   -initialEnergy   (in Joules)
                   -rxPower         (in W)
                   -txPower         (in W)

                   -agentTrace      ON or OFF
```

```
                    -routerTrace    ON or OFF
                    -macTrace       ON or OFF
                    -movementTrace  ON or OFF
```

RESULT:

Simple wired/ wireless network simulation using NS2 are studied successfully.

EX.NO 10:

Programming of TCP/IP protocol stack.

AIM:

To study Programming of TCP/IP protocol stack.

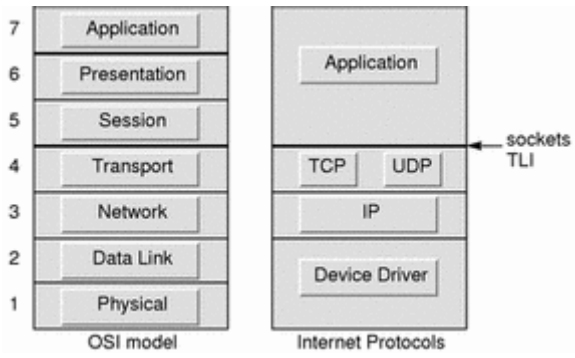
TCP/IP Protocol Stack

The TCP/IP protocol suite can be described using a reference model similar to the OSI reference model. Figure 1-4 shows the corresponding OSI layers and some example services at each layer. TCP/IP does not delineate the presentation and session layers as the OSI model does; application code provides the necessary presentation or session functionality.

The TCP/IP protocols are defined in documents called *Requests for Comments* (RFCs). RFCs are maintained by the Network Information Center (NIC), the organization that handles address registration for the Internet.

RFCs define a number of applications, the most widely used being `telnet`, a terminal emulation service on remote hosts, and `ftp`, which allows files to be transferred between systems.

Figure 1-4 TCP/IP Protocol Stack



TCP/IP Protocol Stack Description

The following sections describes the parts of the TCP/IP protocol stack.

Device Drivers

The device driver layer (also called the Network Interface) is the lowest TCP/IP layer and is responsible for accepting packets and transmitting them over a specific network. A network interface might consist of a device driver or a complex subsystem that uses its own data link protocol.

Internet Protocol (IP) Layer

The Internet Protocol layer handles communication from one machine to another. It accepts requests to send data from the transport layer along with an identification of the machine to which the data is to be sent. It encapsulates the data into an IP datagram, fills in the datagram header, uses the routing algorithm to determine how to deliver the datagram, and passes the datagram to the appropriate device driver for transmission.

The IP layer corresponds to the network layer in the OSI reference model. IP provides a connectionless, "unreliable" packet-forwarding service that routes packets from one system to another.

Transport Layer

The primary purpose of the transport layer is to provide communication from one application program to another. The transport software divides the stream of data being transmitted into smaller pieces called packets in the ISO terminology and passes each packet along with the destination information to the next layer for transmission.

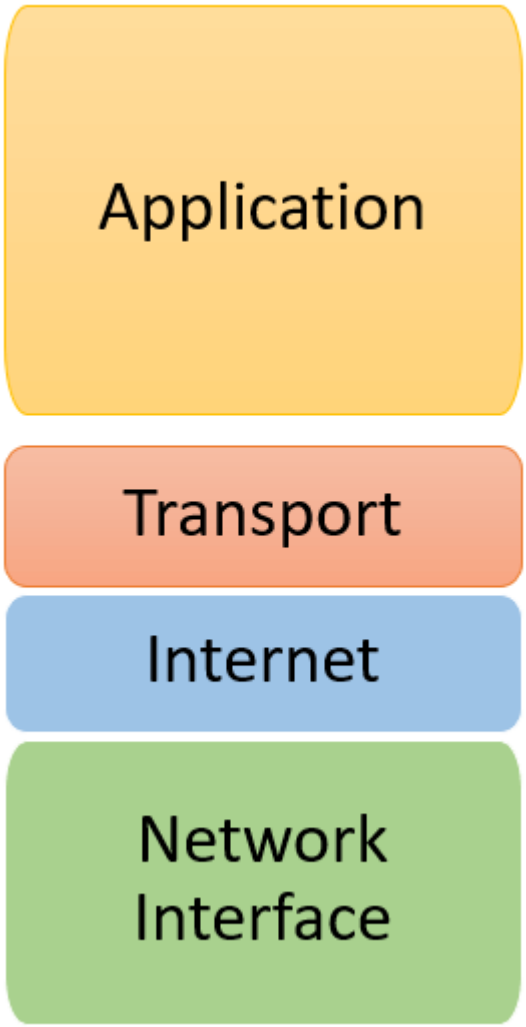
This layer consists of Transport Control Protocol (TCP), a connection-oriented transport service (COTS), and the user datagram protocol (UDP), a connectionless transport service (CLTS).

Application Layer

The application layer consists of user-invoked application programs that access services available across a TCP/IP Internet. The application program passes data in the required form to the transport layer for delivery.

Four Layers of TCP/IP model

In this TCP/IP tutorial, we will explain different layers and their functionalities in TCP/IP model:



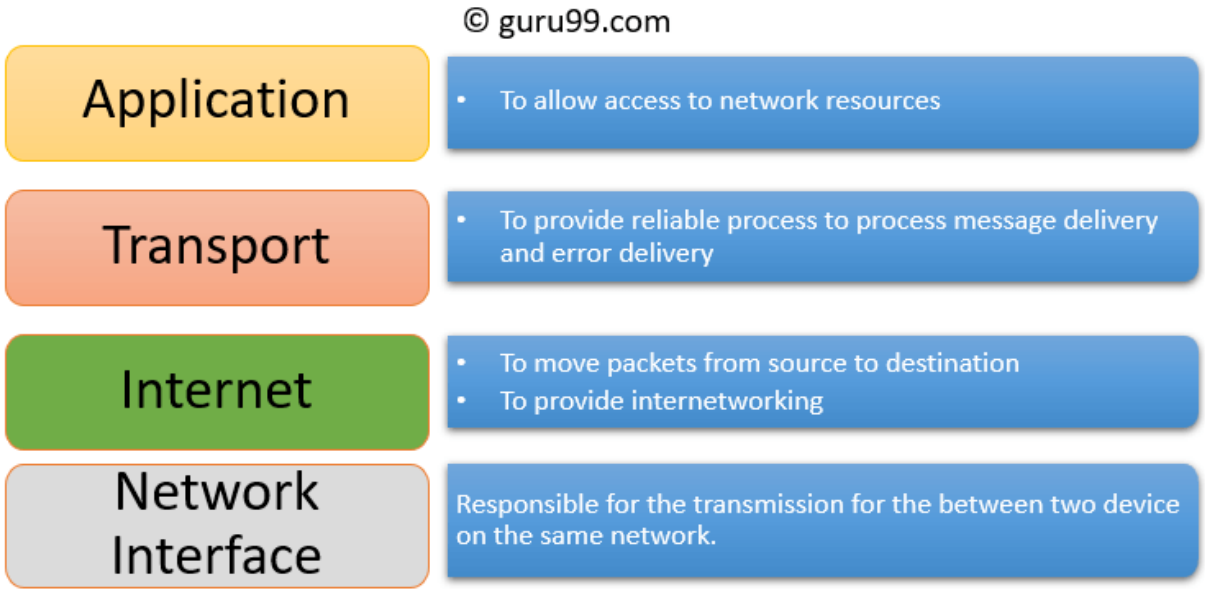
TCP/IP Conceptual Layers

The functionality of the TCP IP model is divided into four layers, and each includes specific protocols.

TCP/IP is a layered server architecture system in which each layer is defined according to a specific function to perform. All these four TCP IP layers work collaboratively to transmit the data from one layer to another.

- Application Layer

- Transport Layer
- Internet Layer
- Network Interface



Four Layers of TCP/IP model

Application Layer

Application layer interacts with an application program, which is the highest level of OSI model. The application layer is the OSI layer, which is closest to the end-user. It means the OSI application layer allows users to interact with other software application.

Application layer interacts with software applications to implement a communicating component. The interpretation of data by the application program is always outside the scope of the OSI model.

Example of the application layer is an application such as file transfer, email, remote login, etc.

The function of the Application Layers are:

- Application-layer helps you to identify communication partners, determining resource availability, and synchronizing communication.
- It allows users to log on to a remote host
- This layer provides various e-mail services
- This application offers distributed database sources and access for global information about various objects and services.

Transport Layer

Transport layer builds on the network layer in order to provide data transport from a process on a source system machine to a process on a destination system. It is hosted using single or multiple networks, and also maintains the quality of service functions.

It determines how much data should be sent where and at what rate. This layer builds on the message which are received from the application layer. It helps ensure that data units are delivered error-free and in sequence.

Transport layer helps you to control the reliability of a link through flow control, error control, and segmentation or de-segmentation.

The transport layer also offers an acknowledgment of the successful data transmission and sends the next data in case no errors occurred. TCP is the best-known example of the transport layer.

Important functions of Transport Layers:

- It divides the message received from the session layer into segments and numbers them to make a sequence.
- Transport layer makes sure that the message is delivered to the correct process on the destination machine.
- It also makes sure that the entire message arrives without any error else it should be retransmitted.

Internet Layer

An internet layer is a second layer of TCP/IP layes of the TCP/IP model. It is also known as a network layer. The main work of this layer is to send the packets from

any network, and any computer still they reach the destination irrespective of the route they take.

The Internet layer offers the functional and procedural method for transferring variable length data sequences from one node to another with the help of various networks.

Message delivery at the network layer does not give any guaranteed to be reliable network layer protocol.

Layer-management protocols that belong to the network layer are:

1. Routing protocols
2. Multicast group management
3. Network-layer address assignment.

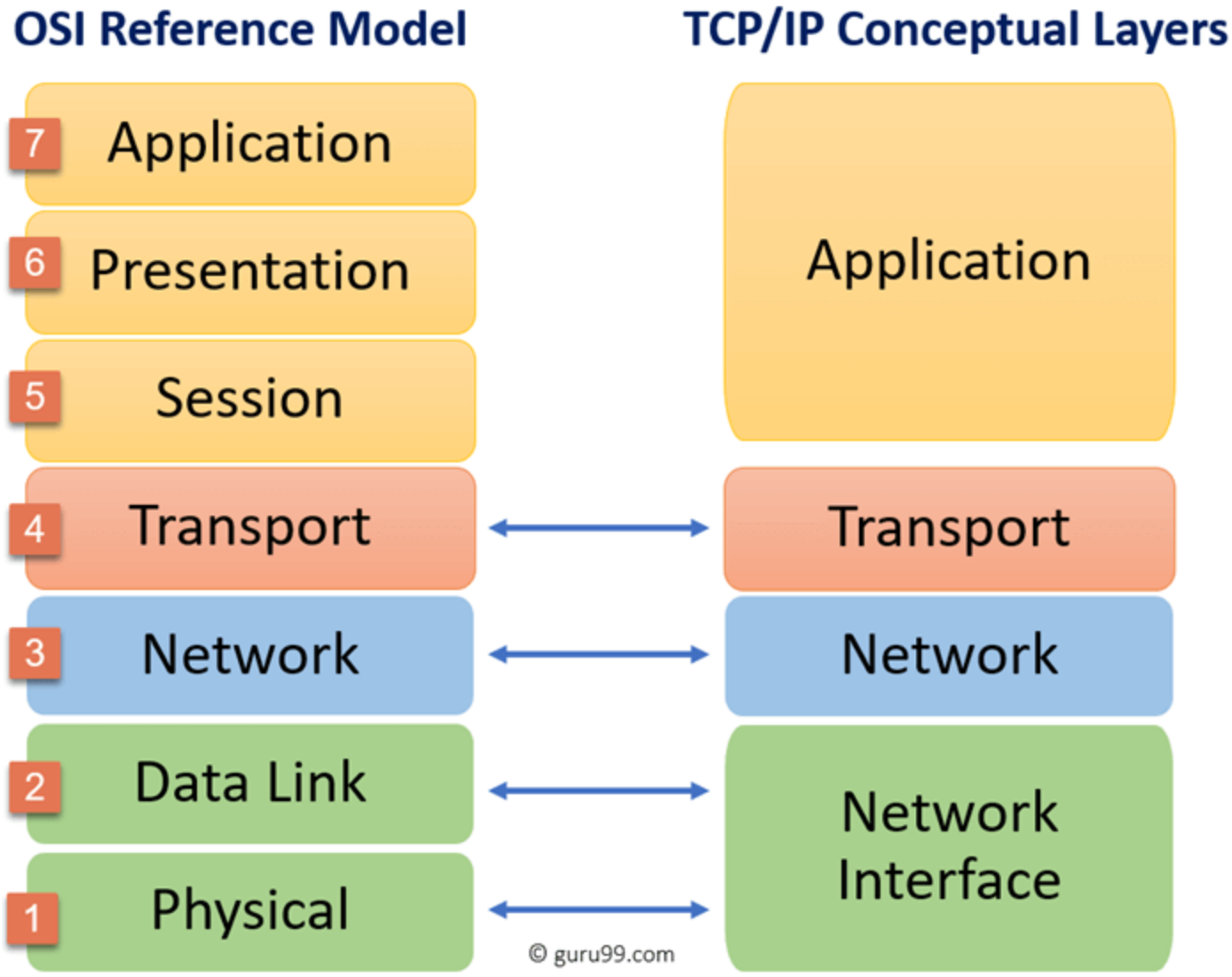
The Network Interface Layer

Network Interface Layer is this layer of the four-layer TCP/IP model. This layer is also called a network access layer. It helps you to defines details of how data should be sent using the network.

It also includes how bits should optically be signaled by hardware devices which directly interfaces with a network medium, like coaxial, optical, coaxial, fiber, or twisted-pair cables.

A network layer is a combination of the data line and defined in the article of OSI reference model. This layer defines how the data should be sent physically through the network. This layer is responsible for the transmission of the data between two devices on the same network.

Differences between OSI and TCP/IP models



RESULT:

Programming of TCP/IP protocol stack are studied.