LPC 2148 Microcontroller Interfacing

Unit 4 Material – Microcontroller & Embedded Systems, 4th Sem, Computer Science & Engineering, RV College Of Engineering

Covers: Interfacing & Programming with LPC 2148 GPIO – LEDs,Switches,SevenSegmentDisplay,LCD,Matrix Keyboard,StepperMotor. Analog Interfaces: ADC- Sensor Interfaces,DAC-Waveform Generation, High Power Device Interface, Opto-Isolator Interface: Proximity Sensor Interface.

2/23/2020

Unit 4

LPC 2148 - ARM7 core based Microcontroller from NXP

LPC 2148 is a Popular ARM microcontroller from NXP Semiconductors, used my many popular industry embedded products, is based on ARM7TDMI-S ARM 7 core. [It has features like Thumb instructions set (T), JTAG debugger (D), fast multiplier (M) and the embedded ICE (I) [ICE- in circuit emulator, useful while debugging hardware & software, S – synthesizable]. It is known for tiny size and low power cosumption

It finds many Applications like:

- Industrial Control
- Medical Systems
- Access Control
- Point-of-Sale
- Communication Gateway, protocol converters
- Embedded Soft Modem, voice recognition, low end imaging
- General Purpose Applications

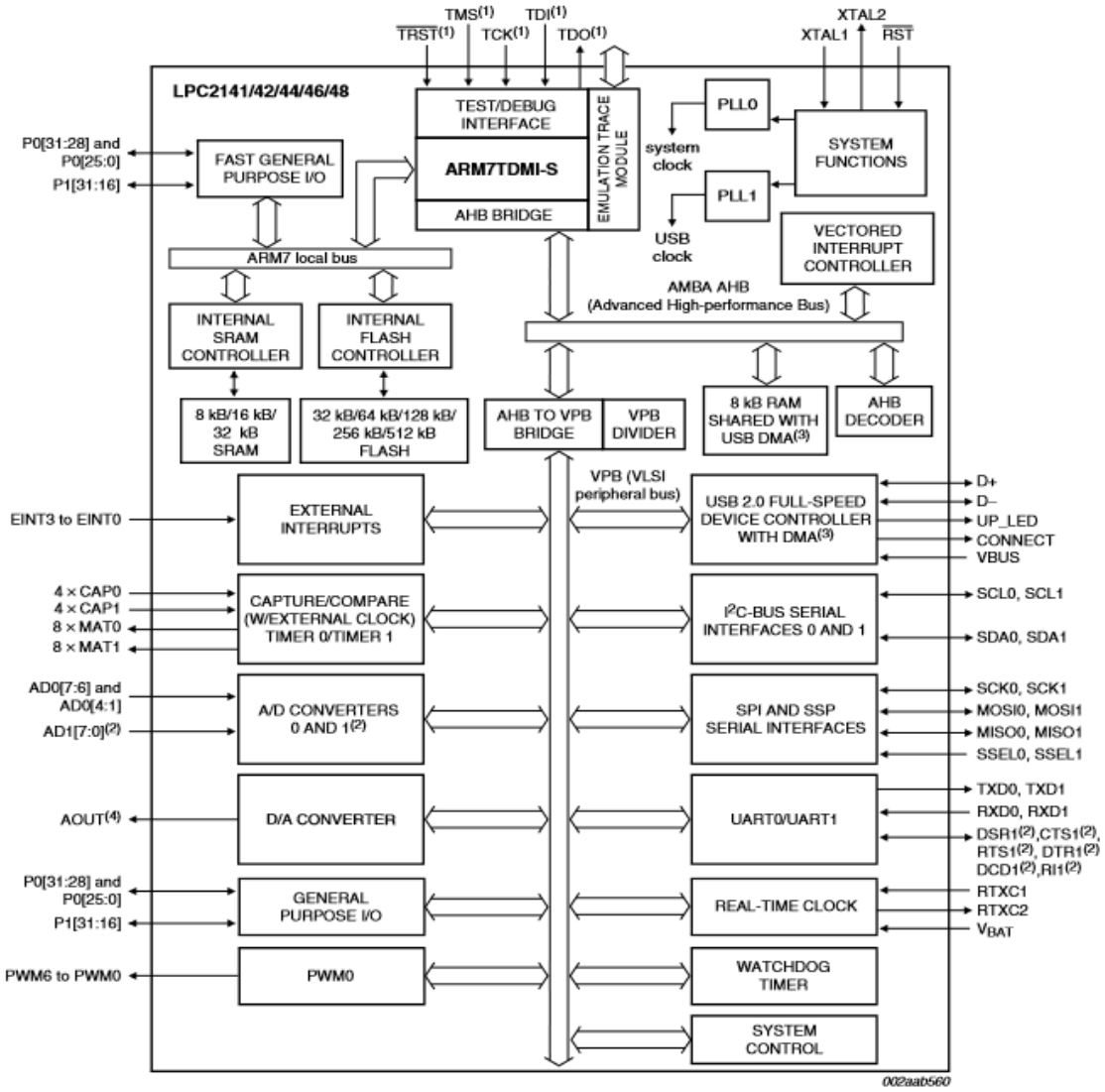
Features of the LPC 2148 Microcontroller

1. Uses the core ARM 7TDMI-S in a tiny LQFP64 package **(** Low Profile Quad Flat Package (LQFP)), operating at 60 MHz clock speed
2. 8kb-40kb of on-chip static RAM (40kb for LPC 2148)
3. 32kb to 512kb of on-chip flash memory(512kb for LPC 2148)
4. 128 bit memory accelerator enables high-speed 60MHz operation
5. USB 2.0 Full speed device controller with DMA (used as USB device, not as a host)
6. 10 bit ADCs provide a total of 6/14(2148 has 14) analog inputs
7. Single 10-bit DAC provides variable analog output
8. Two 32bit timers/event counters
9. PWM Unit
10. RTC (Real Time Clock) with battery backup facility
11. 2 UARTs
12. I2C interface (2 sets)
13. SPI & SSP interfaces (supports SD card)
14. Up to 45 GPIO pins
15. Single power supply with POR(Power On Reset) and BOD(Brown out Detect) circuits, support idle & power-down modes

Different serial protocols supported by ARM LPC 2148 microcontroller and application of each of the protocol.

UART – used to connect to PC/computer/GSM/GPS modules
I2C – used to interface RTC /EEPROM/Sensors
SPI - used to interface data flash memory/SD card
USB 2.0 Full speed device– used to connect other Host devices like computers

Internal Block Diagram of LPC 2148



Internal Buses :

AMBA - Advanced Microcontroller Bus Architecture, is a standard defined by ARM for on-chip buses in its SoC (System On Chip) designs.

It defines three buses with different protocols and speed,

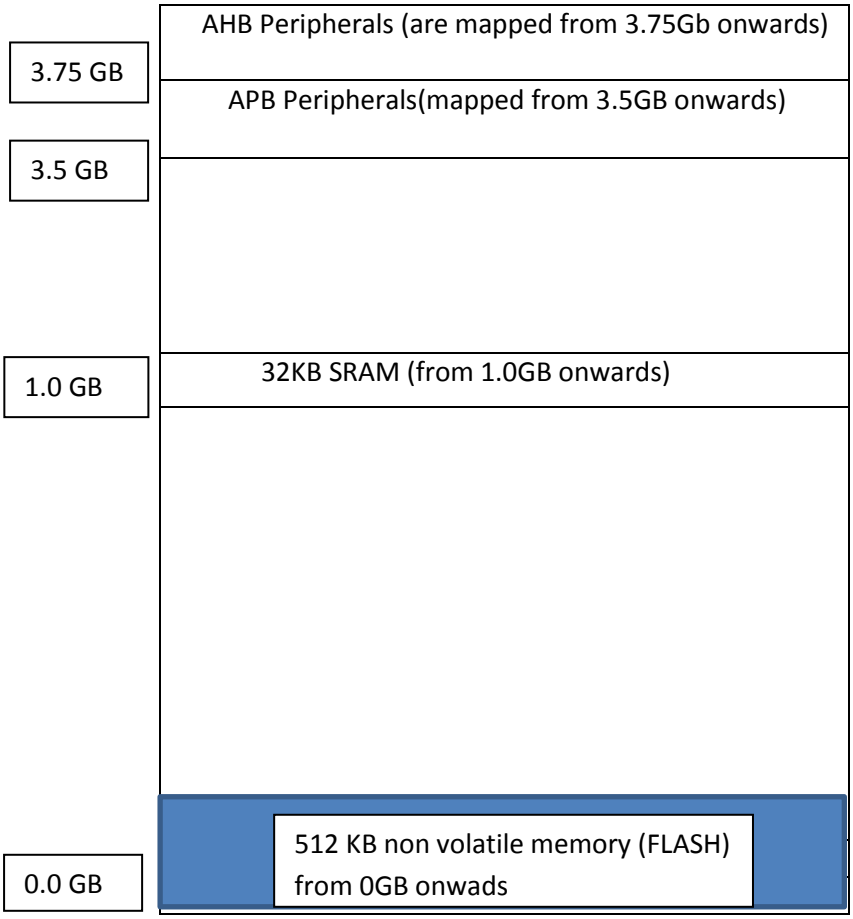
a) Local bus – the fastest bus, which connects the processor core with the memory, as the memory accesses have to be very fast. Connected to Flash memory & SRAM using Memory controllers. Fast GPIO block is also connected to this bus

b) AHB (Advanced High Performance) bus – AHB Bridge is used to interface local bus with AHB bus, VIC (Vectored Interrupt Controller) is connected to this bus.

c) VLSI Peripheral Bus (VPB bus) – there is a bridge which communicates between the low speed VPB bus and the higher speed AHB bus, this bus is used to connect to all the on chip peripherals.

VPB Bus and Divider – VPB divider has a register, programming of this register reduces the processor clock frequency (CCLK) to generate lower speed Peripheral clock (PCLK) used by the peripherals. On reset, PCLK is ¼ of CCLK.

Memory Map & Memory : The total memory space is 2^{32} = 4GB. It supports memory mapped I/O, that means, both memory and peripheral addresses are shared in 4 GB. LPC 2148 has so many inbuilt peripherals like GPIO, timers etc. All these peripherals have many programmable registers to configure the working of the peripherals. Each register has an address allocated from the total memory map of 4GB. AHB peripherals refer to peripherals connected to AHB bus. APB peripherals refer to peripherals connected to APB bus.



The LPC 2148 provides Flash memory (code memory) of 512 KB mapped from the memory address 0, and SRAM of 32KB (+8KB of USB DMA memory) mapped from the memory address 1GB.

Generation of CPU Clock (CCLK) and PCLK (Peripheral Clock):
CCLK is connected to ARM CPU core, decides the frequency at which the Core runs (execution of instructions). PCLK is derived from CCLK and used to drive the different on chip peripherals. CCLK and PCLK are generated from on chip PLL (Phase locked loop) modules based on the crystal connected to XTAL1 & XTAL2.

[*Extra Information*: There are two PLL modules in the LPC 2148 Microcontroller, PLL0 and PLL1. The PLL0 is used to generate the CCLK clock (system clock / CPU Clock), while the PLL1 has to supply the clock for the USB at the fixed rate of 48MHz.

The PLL0, PLL1 accept the clock (FOSC) obtained from the crystal connected at XTAL1, XTAL2 and multiplies this to enhance the frequency in the range of 10MHz to 60MHz. Generally 12MHz crystal is connected, PLL0, by default multiplies this to 60 MHz. The default value can be changed by suitably programming PLLCFG register.

Both the PLLs are turned off on Reset (so CCLK = FOSC, if crystal frequency is 12MHz, the CCLK=12 MHz). So on Reset PLLs to be enabled and configured; wait for the PLL to lock to obtain the multiplied higher speed CCLK.

The PLL signal must also be supplied to other on – chip peripherals. By default, the peripheral clock (PCLK) runs at a quarter speed of the CCLK (i.e 15MHz if CCLK is 60MHz). The relation between the CPU clock and peripheral clock can be configured by using a special register called "VPBDIV" (by default VPBDIV = 0x00).]

If PLLs are enabled, the default values are:
FOSC = 12MHz (if 12MHz Crystal is used),CCLK = 60MHz;PCLK = 15MHz

Interfacing & Programming GPIO

LPC 2148 provides 2 ports, of size 32 bits, used to interface LCD, relays, LEDs etc.

Port0 is a 32 bit I/O port, of which 28 pins can be used for digital i/o, P0.31 used only for o/p, pins P0.24,P0.26 & P0.27 are reserved and not available for use.

Port 1 is a 32 bit I/O port, but only P0.16 – P0.31 are available for I/O. Hence totally 45 I/O are possible, but all the pins have alternate functions also, as detailed below.

Pin Connect Block : The purpose of this is to configure the GPIO pins to the desired functions. This acts like a multiplexer. Each pin of the chip has a maximum of four functions. To select one specific function for a pin, a multiplexer with two select pins, is necessary. The select pins function is provided by the bits of the PINSEL registers.

Three PINSEL registers, (PINSEL0,PINSEL1,PINSEL2) are required, first two are for the Port0. Two bits are required to configure one pin, hence PINSEL0 is used to

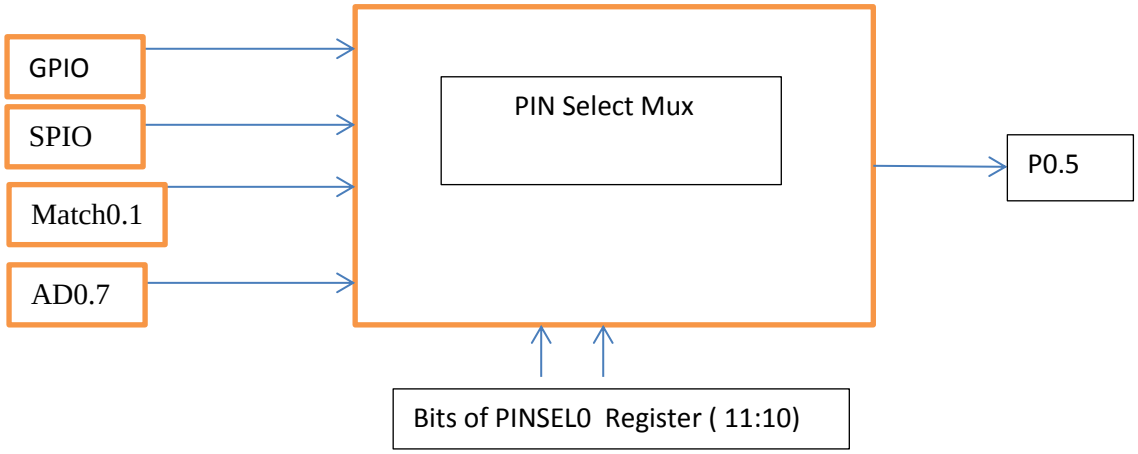
configure the pins P0.0 to P0.15, PINSEL1 is used to configure P0.16 to P0.31, PINSEL2 is used to configure P1.16 to P1.31.

Example for **Pin P0.5 :**

Bits of PINSEL0 Register	Port Pin	value of PINSEL bits	Function Selected	Reset Value
11:10	P0.5	00	GPIO	0
		01	MISO0(SPI-0)	
		10	Match 0.1 (timer 0)	
		11	AD0.7	

To use P0.5 as GPIO, we have to make it 00,

PINSEL0 = 0x0000 0000; By default, on reset, all port pins act as GPIO pins



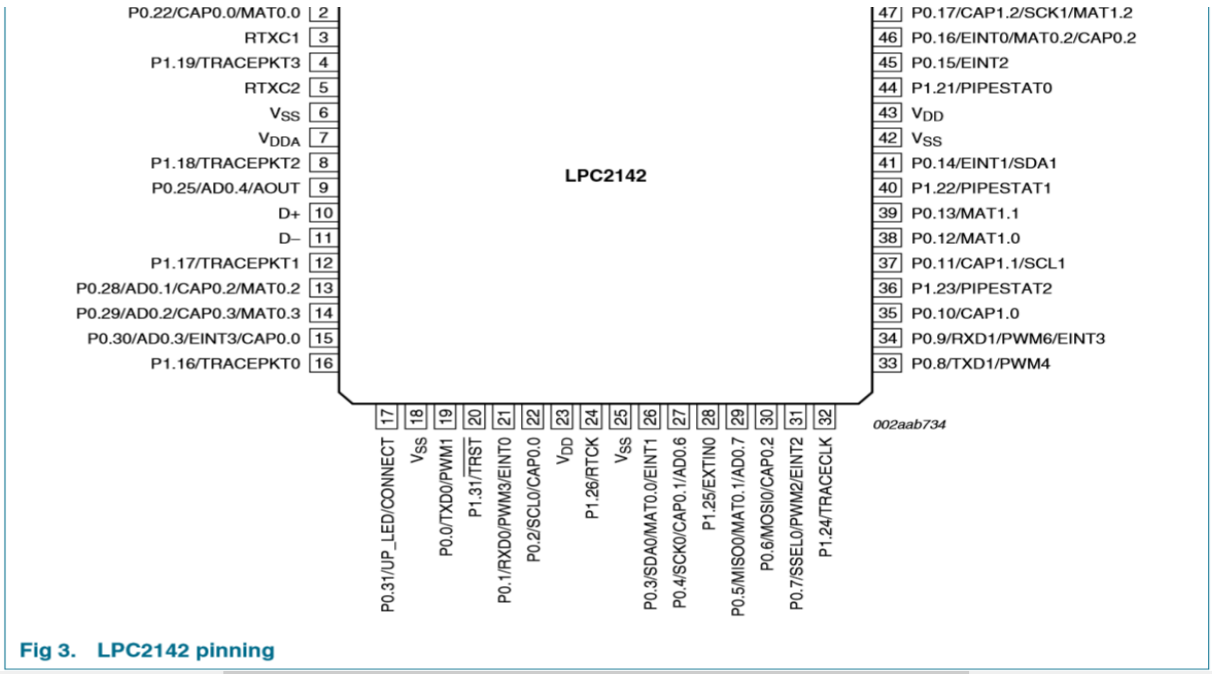


Fig 3. LPC2142 pinning

Using GPIO Pins :- These pins are used for connecting LEDs, Switches, LCD, Relays etc. Following registers are provided to use them,

1. IODIR (IO direction register, IODIR0 for P0 & IODIR1 for P1) – The bit setting of this register configures the pin as input or output, 1 for output, 0 for input

2. IOSET(IO set register, IOSET0 & IOSET1) – This register is used to set the output pins of the chip. To make a pin to be '1', the corresponding bit in the register is to be '1'. Writing zeros have no effect

3. IOCLR(IO Clear register, IOCLR0 & IOCLR1) – To make an output pin to have a '0' value, i.e to clear it. The corresponding bit in this register has to be a '1'. Writing zeros have no effect.

4. IOPIN (IO Pin register, IOPIN0 & IOPIN1): From this register the value of the corresponding pin can be read, irrespective of whether the pin is an input or output pin.

Example1: Set the lower sixteen GPIO pins of P0 to 1.

First set the direction, IODIR0 = 0x0000FFFF
Set the output to 1, IOSET0 = 0x0000FFFF

Example2: Generate the **asymmetric square wave** at the lowest four pins of Port0

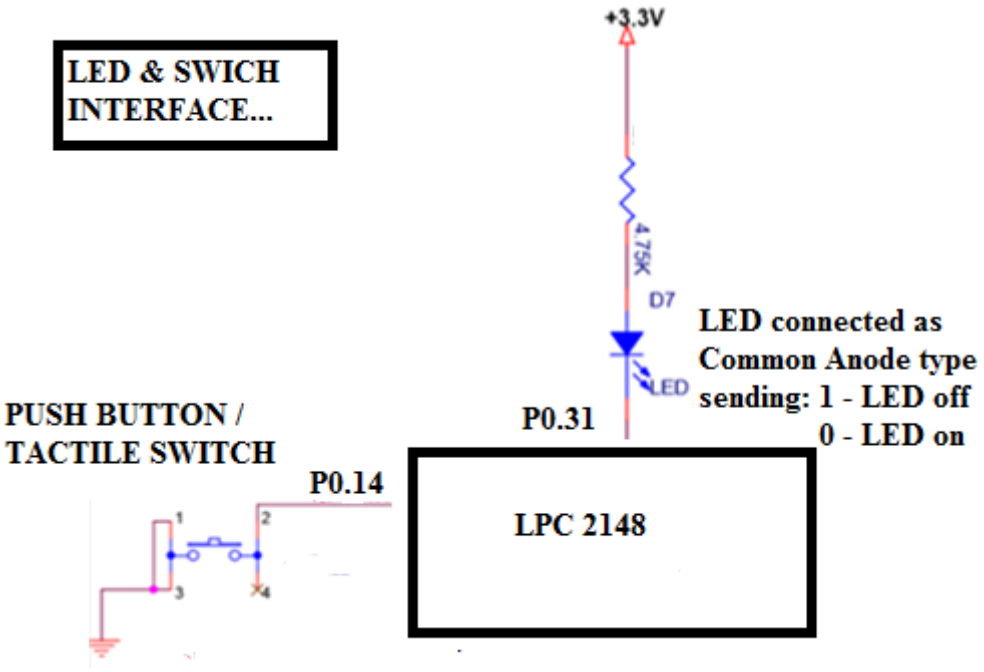
```
#include <LPC214X.H>
int main(void)
{
  unsigned int x;
  IODIR0 = 0xFFFFFFFF;    //Make all the pins as outputs
 for(;;)
 {
   IOSET0 = 0x0000000F;  //Set the port pins P0.0 to P0.3
   for(x=0;x<10000;x++)
   IOCLR0 = 0x0000000F;   //Clear the port pins
   for(x=0;x<20000;x++);
 }
}
```

Example3: Interface an LED to P0.10 and generate the asymmetric square wave.

```
#include <LPC214X.H>
int main(void)
{
  unsigned int x;
  IODIR0 = 0xFFFFFFFF;    //Make all the pins as outputs
 for(;;)
 {
   IOSET0 = 1 << 10;  //Set the port pin P0.10
   for(x=0;x<30000;x++);  //delay for ON time
   IOCLR0 = 1 << 10; //Clear the port pin P0.10
   for(x=0;x<40000;x++);  // delay for OFF time
 }
}
```

Question 1: Interface an LED and Push button (press to on) Switch , and write a program to blink the led, when the switch is keep pressed.

Interfacing



Embedded C Program / Driver Program to LED,Switch

Assume P0.31 connected to LED, Assume P0.14 connected to Switch
Configure P0.31 as GPIO output, as LED is output device and P0.14 as GPIO input, as Switch is input device.

Working of LED - (common anode). 1 – Led off, 0 – Led on
Working of switch: when switch is not pressed, P0.14 receives = 1, when switch is pressed P0.14 receives = 0

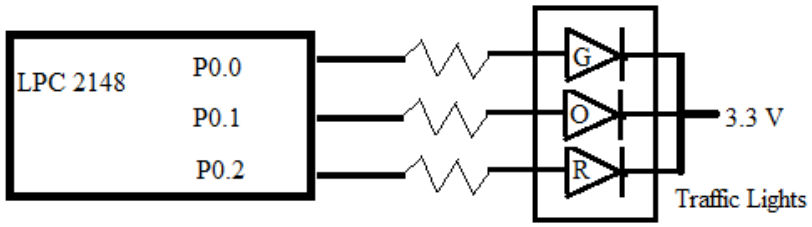
```
#include <lpc214x.h>
#define LED_OFF (IO0SET = 1U << 31)
#define LED_ON (IO0CLR = 1U << 31)
#define SW2 (IO0PIN & (1 << 14))


void delay_ms(unsigned int j);

int main( )
{
    IO0DIR = 1U << 31;
    IO0SET = 1U << 31;
    while(1)
     {
          if (!(IO0PIN & (1 << 14)))       //  (if(!SW2 )
           {
                    IO0CLR = 1U << 31;  //LED_ON
                    delay_ms(250);
                    IO0SET = 1U << 31;  //LED_OFF
                    delay_ms(250);
           }
     }
}
void delay_ms(unsigned int j)  ; assume delay_ms(1) produces 1ms
{
  unsigned int x, i;
  for(i=0; i<j; i++)
   {
     for(x=0; x<1000; x++);
   }
}
```

Question2: Interface 3LEDs (Red,Yellow,Green) to LPC 2148 and Write Embedded C program to simulate traffic light system.

Interfacing :

Choose any three port lines (say P0.0,P0.1,P0.2) to connect to 3 Leds to represent Traffic Lights, one end of the junction. (Relays / Driver also can be used to connect higher voltage bulbs).

Here, LEDs are connected as Command Cathode type, i.e Microcontroller port lines are connected to Anode of the LEDs. Sending '1' – LED is on, sending '0' – LED is off



Embedded C Program / Driver Program for Traffic Leds

Configure P0.0,P0.1,P0.2 as outputs, as LEDs are output devices.

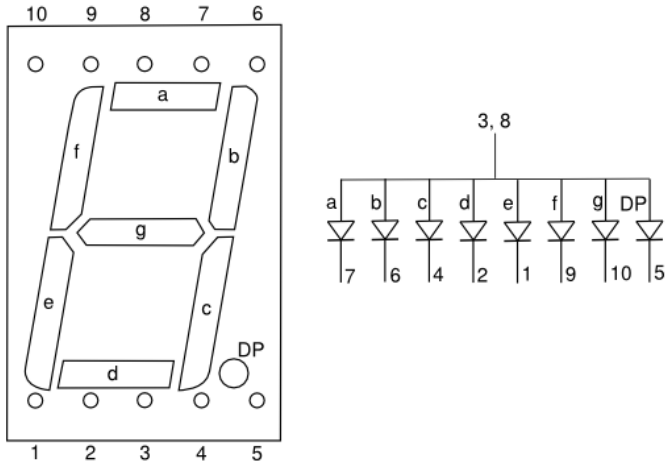
```
#include  <LPC214x.h>
#define val1  30000
#define val2  5000
#define val3  50000
void delay(unsigned long int);
int main(void)
{
     I0DIR0 = 0X00000007; //make P0.0,P0.1,P0.2 as outputs
     while(1)
        {
          // set GREEN for 30seconds
          I0SET0  =  1 << 0;  //set P0.0 to 1
          delay(val1);
          I0CLR0  =  1 <<  0;  //clear P0.0 to 0

         // set ORANGE/Yellow for 5 seconds
          I0SET0  =  1 << 1;  //set P0.1 to 1
          delay(val2);
          I0CLR0  =  1 << 1;  //clear P0.1 to 0

           // set RED for 105 seconds (3 x 35 = 105), assuming it
           //is part of junction with four roads
          I0SET0  =  1 << 2;  //set P0.2 to 1
          delay(val3); delay(val3); delay(val2);
          I0CLR0  =  1 << 2;  //clear P0.2 to 0
        }
}
  void delay_ms(unsigned int j) ; assume delay_ms(1) produces 1ms
  {
   unsigned int x, i;
   for(i=0; i<j; i++)
    {
      for(x=0; x<1000; x++);
    }
  }
```

Seven Segment Display Interface: Interface multi-digit seven segment displays and write an Embedded C program to display strings.

Seven Segment displays, are more preferred compared to normal LEDs, used in the previous examples. It is part of almost in embedded system project. These displays are visible during all the times of a day, can be seen from long distance and very cost effective.

Interfacing :-

Connecting to the microcontroller can be achieved by many methods. Here we are employing set of shift registers (IC 4094), cascaded together, to drive multiple displays. One 8 bit shift register is required to drive one seven segment display, as each seven segment display requires 8 bits of information, to drive 8 segments a to dp, as shown below.

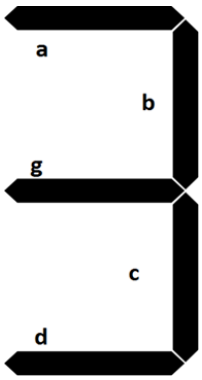


Common Anode Seven Segment Display with Pin Numbering

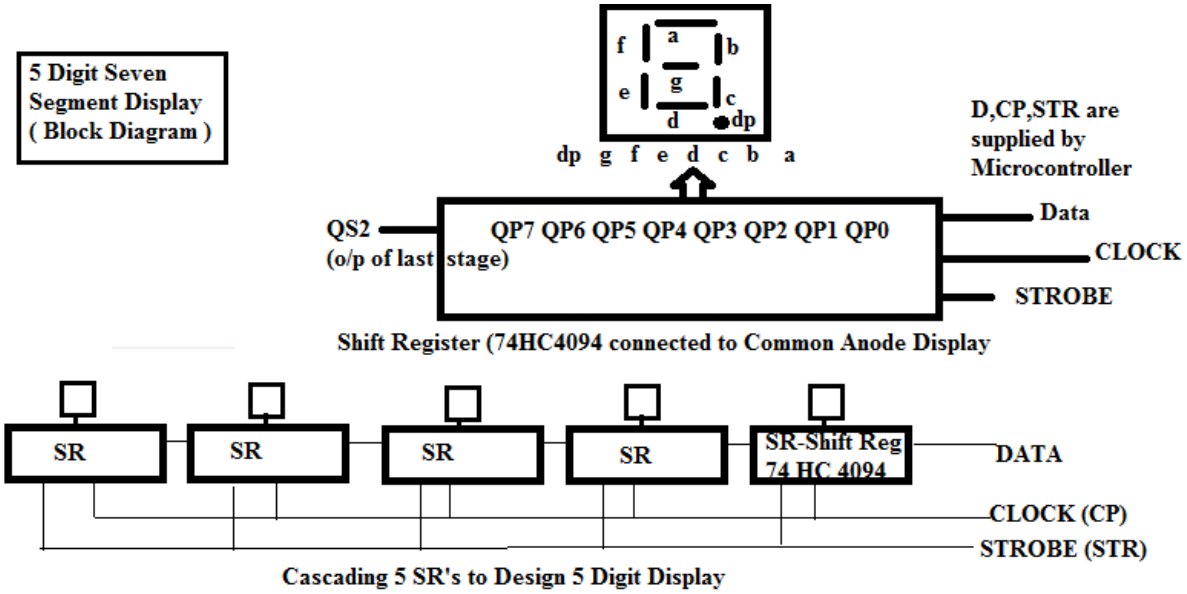
The seven segment displays are available in two types, common anode display and common cathode display. In common anode display, the anodes of all the segments/LEDs are shorted together and connected to power supply +5V. Hence, applying '0' voltage to the cathodes of the segments, will enable the segment to glow. To display '3', we have to send following bit pattern,

DP	G	f	e	d	c	b	a
1	0	1	1	0	0	0	0

This is 'B0' in hexadecimal.

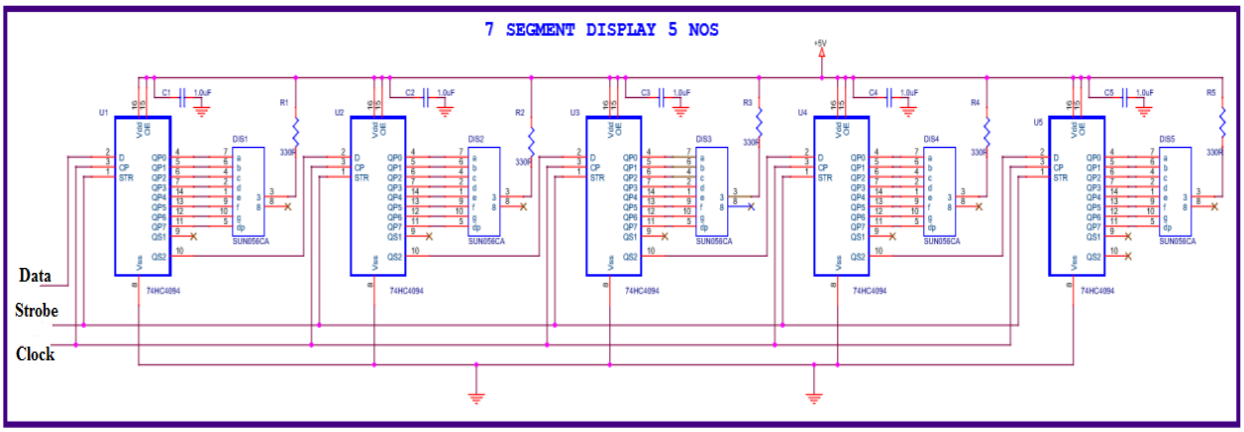


By using multiple seven segment digits, we can design multi digit display system. Shift register can be used to supply 8 bit data to every seven segment digit. Following block diagram describes the interfacing details of building 5 digit display system.



74HC4094 is a 8bit Serial-Input Shift Register with Latched 3 state outputs. It consists of an 8 bit shift register and 8bit D-type latch with 3 state parallel outputs. Data is shifted serially through the shift register on the positive going transition of the clock input signal. Shift register require Clock pulses to clock in the data serially through 'Data' pin, 8 clock pulses require for one byte of data. The data of each stage of the shift register is provided with a latch, which latches the data on negative going transition of the strobe input signal. Strobe signal is generated once all the 8 bits of data representing one digit is serially shifted. The outputs of the latches are connected to three state buffers, which is enabled by making output enable pin high. (Here, in the circuit we have connected Output Enable permanently to high, tied to Vcc.). The output of the last stage QS1/QS2' is used to cascade multiple shift registers, so that they appear to work as one unit.

Microcontroller generates Shift register 3 inputs: CLOCK (CP), DATA (D) and STROBE(STR). This 8 bit data has to be transferred to the shift register serially. Hence to send B0H (to display 3), we have to start sending the bits from MSB onwards i.e D7 (1)first, D6 (0) next and so on with D0(0) being the last. As shift registers are cascaded, 8*5=40 clocks are required to clock in 5 bytes of data. To send "12345", first we have to send '1', then '2','3' ,'4' and lastly '5'. All the shift registers are cascaded; the data is fed to the shift registers using serial in parallel out method, from the first shift register. Strobe is used to copy the shifted data to the output pins. STB is generated after shifting is completed.



Embedded Software: Write a C program to display messages "FIRE" & "HELP" on 5 digit seven segment display alternately with a suitable delay.

```
//Seven Segment Display Program:
//P0.19 Data pin of 1st shift register
//P0.20 Clock pin of shift registers, make 1 to 0
//P0.30 Strobe pin of shift registers: 1 to 0

#include <lpc214x.h>
#define LED_OFF (IO0SET = 1U << 31)
#define LED_ON (IO0CLR = 1U << 31)
#define PLOCK 0x00000400
void delay_ms(unsigned int j);
void SystemInit(void);
unsigned char getAlphaCode(unsigned char alphachar);
void alphadisp7SEG(char *buf);
int main()
{
      IO0DIR |= 1U << 31 | 1U << 19 | 1U << 20 | 1U << 30 ; // to set as o/ps
       LED_ON; // make D7 Led on .. just indicate the program is running
       SystemInit();
       while(1)
          {
            alphadisp7SEG("fire ");
            delay_ms(500);
            alphadisp7SEG("help ");
            delay_ms(500);
          }

}
```

```
unsigned char getAlphaCode(unsigned char alphachar)
{
  switch (alphachar)
            {
            // dp g f e d c b a - common anode: 0 segment on, 1 segment off
            case 'f': return 0x8e;
            case 'i': return 0xf9;
            case 'r': return 0xce;
            case 'e':return 0x86;  // 1000 0110
            case 'h':return 0x89;
            case 'l': return 0xc7;
            case 'p':return 0x8c;
            case ' ': return 0xff;
            //simmilarly add for other digit/characters
            default : break;
            }
  return 0xff;
}
void alphadisp7SEG(char *buf)
{
  unsigned char  i,j;
  unsigned char seg7_data,temp=0;
  for(i=0;i<5;i++) // because only 5 seven segment digits are present
    {
       seg7_data = getAlphaCode(*(buf+i));
     // instead of this look up table can be used
    // to shift the segment data(8bits)to the hardware (shift registers) using Data,Clock,Strobe
            for (j=0 ; j<8; j++)
            {
                    //get one bit of data for serial sending
                    temp = seg7_data & 0x80;  // shift data from Most significan bit (D7)
                    if(temp == 0x80)
                            IOSET0 |= 1 << 19; //IOSET0 | 0x00080000;
                    else
                            IOCLR0 |= 1 << 19; //IOCLR0 | 0x00080000;
                    //send one clock pulse
                    IOSET0 |= 1 << 20;  //IOSET0 | 0x00100000;
                    delay_ms(1);
                    IOCLR0 |= 1 << 20;  //IOCLR0 | 0x00100000;
                    seg7_data = seg7_data << 1; // get next bit into D7 position
            }
    }
```

```
   // send the strobe signal
   IOSET0 |= 1 << 30; //IOSET0 | 0x40000000;
   delay_ms(1);      //nop();
   IOCLR0 |= 1 << 30;  //IOCLR0 | 0x40000000;
   return;
}
 void SystemInit(void)
 {
   PLL0CON = 0x01;
   PLL0CFG = 0x24;
   PLL0FEED = 0xAA;
   PLL0FEED = 0x55;
   while( !( PLL0STAT & PLOCK ))
   { ; }
   PLL0CON = 0x03;
   PLL0FEED = 0xAA;  // lock the PLL registers after setting the required PLL
   PLL0FEED = 0x55;
   VPBDIV = 0x01;      // PCLK is same as CCLK i.e 60Mhz
 }

 void delay_ms(unsigned int j)
 {
  unsigned int x,i;
  for(i=0;i<j;i++)
   {
     for(x=0; x<10000; x++);
   }
 }
```

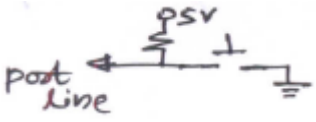
Extra Information:

```
// CODE to display an integer number/long integer number

// long int dig_value;

// unsigned char buf[5];

// sprintf(buf,"%05lu",dig_value);

// alphadisp7SEG(&buf[0]);
```

Matrix Keyboard Interface: Write an embedded C program to interface 4 X 4 matrix keyboard using lookup table and display the key pressed on the Terminal.

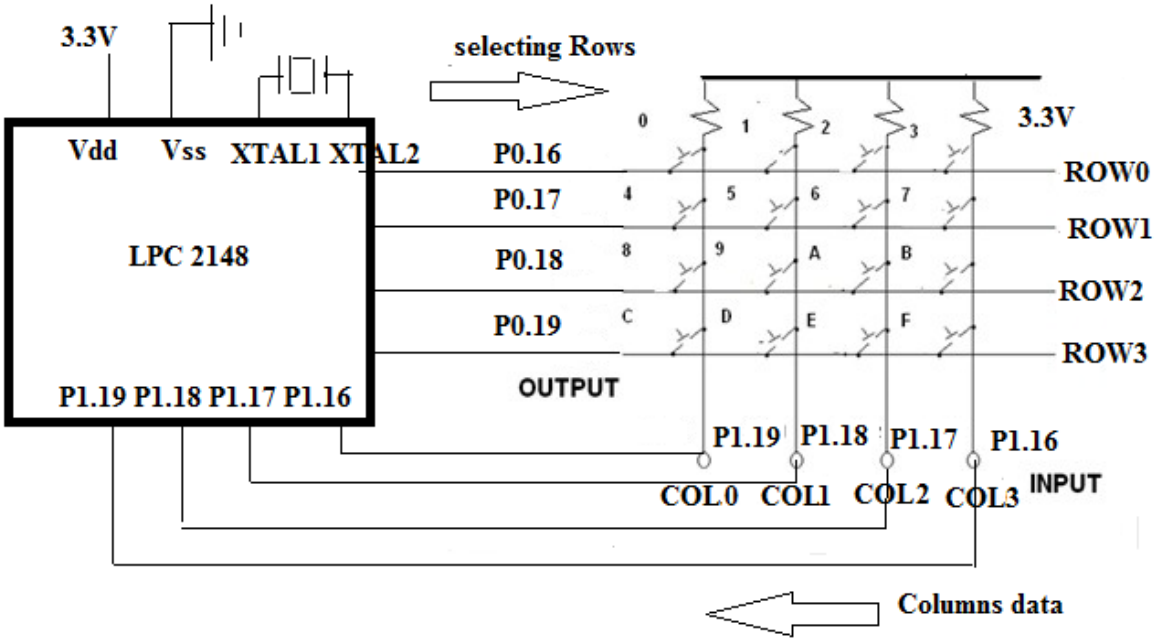
Every embedded product available in the market, do provide matrix keyboards to feed numbers and text. This is the common user interface (UI) to achieve size and cost optimization.

- One port line is required to interface one key



[when key is not pressed, port line carries logic 1, ∵it is pulled upto 5V, when key is pressed, port line gets grounded, hence logic ∅ is applied]. When number of keys increases, requirement of no. of port lines increases, hence concept of matrix arrangement of keys (matrix keyboard) are used. The layout is shown below, to build 4x4 matrix keyboard (16Keys).

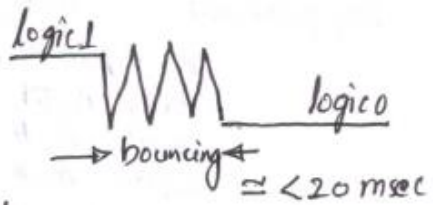
Interfacing



- Here keys are placed/connected at the intersection of rows and columns. 16 keys, numbered 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F are arranged in 4 rows x 4 columns. Example: key'0' is connected at the intersection of row0 and col3, key'F' is connected at the intersection of row3 and col0.
- Here 4 rows (row0,row1,row2,row3) are connected to P0.16, P0.17, P0.18, P0,19 and 4 columns are connected to P1.16, P1.17, P1.18, P1.19. The rows are output to the microcontroller and the columns are the inputs to the microcontroller. All columns are pulled high, i.e connected to 3.3V. When no key is pressed, columns, give '1111' at their pins to the microcontroller.

- when any of the keys is pressed – corresponding row and column gets shorted and the voltage (logic 0 or 1) of row is transferred to corresponding column. Since all columns are pulled high, when no key is pressed, columns, give '1111' at their pins. To check for a key press, output the logic '0' to the selected row, then check for the data on columns, if any key is pressed in the selected row, we will get data other than '1111', the pressed key makes the corresponding column data to '0' from '1'. Example: if selected row is row0, key0 is pressed, the column data "0111".

- Key debouncing : In majority of cases, keys used are of mechanical type and it generates / produces bouncing at its contacts,
  So by calling delay 0f 20 msec and rechecking the key value is referred as debouncing.



Debouncing (≃ 20 ms delay) overcomes two things – false / wrong key press identification and single key press treated as multiple key presses.

Embedded Software / Driver for reading matrix keyboard

Algorithm for reading matrix keyboard / Working method:

- If no key is pressed, we will have on columns 0-3, '1111' on P1.16 to P1.19, as all the inputs are pulled up by pull up resistors.
- If we press any key, let '0' key be pressed, it will short row0 and col0 lines (P0.16 & P1.19), so whatever data (0 or 1) available at row0 (P0.16) is available at col0 (P1.19). Since already columns are pulled high, it is required to apply logic '0' to see change in col0 when the key is pressed.
- To identify which key is pressed, perform the following steps in the loop,
  - Check for a key press in first row by out putting – '0111'on row's, check which column data is changed, if no key press go for next row. If key press found, exit loop with row no and column no.
  - Check for a key press in second row by out putting – '1011'on row's, check which column data is changed, if no key press go for next row. If key press found, exit loop with row no and column no.
  - Check for a key press in third row by out putting – '1101'on row's, check which column data is changed, if no key press go for next row. If key press found, exit loop with row no and column no.
  - Check for a key press in last row by out putting – '1110'on row's, if no key is pressed go for the first row again. If key press found, exit loop with row no and column no.

- Once the key press is found, use the row number and column number and use the "look up table" to convert the key position(row no : column no) to corresponding ascii code/key code related to that key. Use appropriate delay for debouncing.
- Output the key code on the serial port or it can be used for further processing
- Wait for the pressed key release; be in the loop until all the columns data become "1111'.

Program :

```
//Matrix 4 x 4 Keyboard
//Columns & Rows are pulled to +5v,if dont press key, we receive '1' on columns
//Method: Sending '0' to a selected row, checking for '0' on each column
//ROWS - ROW0-ROW3 -> P0.16,P0.17,P0.18,P0.19
//COLS - COL0-COL3 -> P1.19,P1.18,P1.17,P1.16
#include <lpc214x.h>
#define PLOCK 0x00000400
#define LED_OFF (IO0SET = 1U << 31)
#define LED_ON  (IO0CLR = 1U << 31)
#define COL0 (IO1PIN & 1 <<19)
#define COL1 (IO1PIN & 1 <<18)
#define COL2 (IO1PIN & 1 <<17)
#define COL3 (IO1PIN & 1 <<16)
void SystemInit(void);
void delay_ms(unsigned int j);
void uart_init(void);
unsigned char lookup_table[4][4]={  {'0', '1', '2','3'},
                                    {'4', '5', '6','7'},
                                    {'8', '9', 'a','b'},
                                    {'c', 'd', 'e','f'}};
unsigned char rowsel=0,colsel=0;
int main( )
{
 SystemInit();
 uart_init();//initialize UART0 port
 IO0DIR |= 1U << 31 | 0x00FF0000; // to set P0.16 to P0.23 as o/ps
do
  {
    while(1)
     {
      //check for keypress in row0,make row0 '0',row1=row2=row3='1'
      rowsel=0;IO0SET = 0X000F0000;IO0CLR = 1 << 16;
      if(COL0==0){colsel=0;break;};if(COL1==0){colsel=1;break;};
      if(COL2==0){colsel=2;break;};if(COL3==0){colsel=3;break;};
      //check for keypress in row1,make row1 '0'
```

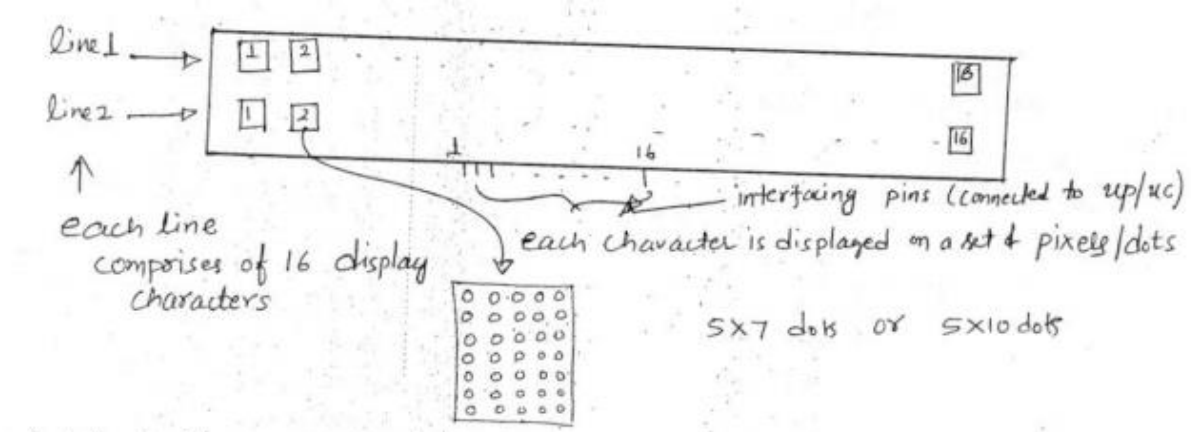
```
        rowsel=1;IO0SET = 0X000F0000;IO0CLR = 1 << 17;
        if(COL0==0){colsel=0;break;};if(COL1==0){colsel=1;break;};
        if(COL2==0){colsel=2;break;};if(COL3==0){colsel=3;break;};
        //check for keypress in row2,make row2 '0'
        rowsel=2;IO0SET = 0X000F0000;IO0CLR = 1 << 18;//make row2 '0'
        if(COL0==0){colsel=0;break;};if(COL1==0){colsel=1;break;};
        if(COL2==0){colsel=2;break;};if(COL3==0){colsel=3;break;};
        //check for keypress in row3,make row3 '0'
        rowsel=3;IO0SET = 0X000F0000;IO0CLR = 1 << 19;//make row3 '0'
        if(COL0==0){colsel=0;break;};if(COL1==0){colsel=1;break;};
        if(COL2==0){colsel=2;break;};if(COL3==0){colsel=3;break;};
      };
  delay_ms(50);  //allow for key debouncing
  while(COL0==0 || COL1==0 || COL2==0 || COL3==0);//wait for key release
  delay_ms(50);   //allow for key debouncing
  IO0SET = 0X000F0000; //disable all the rows
  U0THR = lookup_table[rowsel][colsel];  //send to serial port(check on the terminal)
 }
 while(1);
}
void uart_init(void)
{
  //configurations to use serial port
 PINSEL0 |= 0x00000005;  // P0.0 & P0.1 ARE CONFIGURED AS TXD0 & RXD0
 U0LCR = 0x83;   /* 8 bits, no Parity, 1 Stop bit    */
 U0DLM = 0; U0DLL = 8; // 115200 baud rate
 U0LCR = 0x03;  /* DLAB = 0                         */
 U0FCR = 0x07;  /* Enable and reset TX and RX FIFO. */
}
void SystemInit(void)
{
  PLL0CON = 0x01;
  PLL0CFG = 0x24;
  PLL0FEED = 0xAA;
  PLL0FEED = 0x55;
  while( !( PLL0STAT & PLOCK ))
  { ; }
  PLL0CON = 0x03;
  PLL0FEED = 0xAA;  // lock the PLL registers after setting the required PLL
  PLL0FEED = 0x55;
  VPBDIV = 0x01;      // PCLK is same as CCLK i.e 60Mhz
}
void delay_ms(unsigned int j)
{
  unsigned int x,i;
 for(i=0;i<j;i++)
  {
    for(x=0; x<10000; x++);
  }
}
```

Alpha Numeric LCD Interface : Interface LCD and Write an Embedded C program to display text messages on the multiple lines of the display.

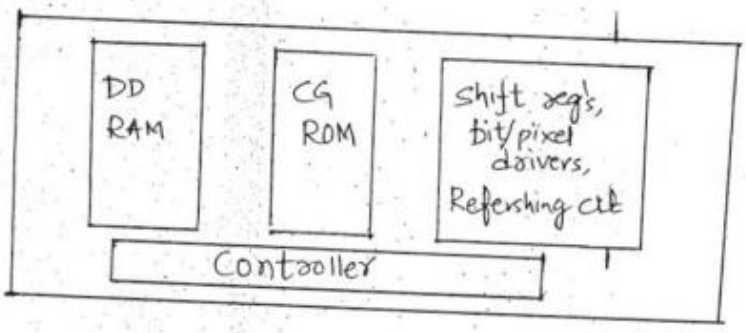
Features..

LCD's are preferred to seven segment displays because of their versatility and capability to house more information.2 line (2 x 16) and 4 line (4x20) is the most popular, low cost character oriented LCD, suitable for understanding the working and programming of LCD. You have seen LCD modules used in many of the electronics devices like coin phone, billing machine and weighing machines. It is a powerful display options for stand-alone systems. Because of low power dissipation, high readability, flexibility for programmers, LCD modules are becoming popular.

Working principle..

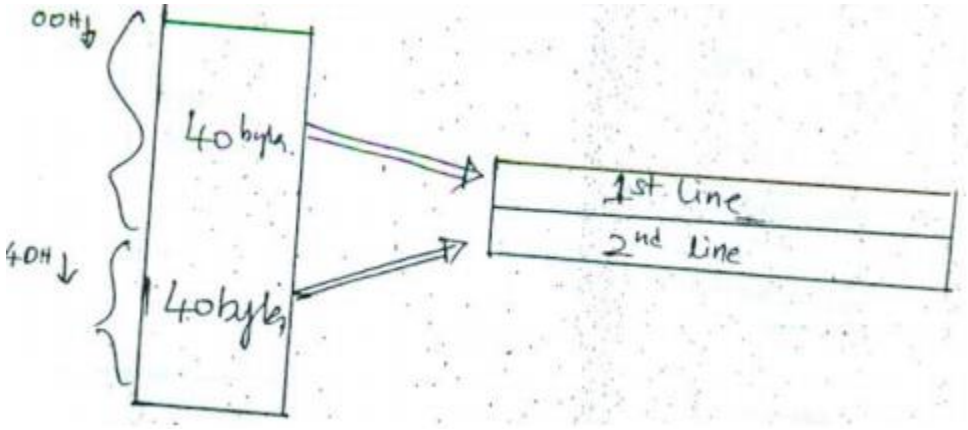


What is inside LCD...



LCD consists of DDRAM, CGROM, Shift registers, bit/pixel drivers, refreshing logics and lcd controller. The data to be displayed on lcd, is to be written on to the DDRAM-display data Ram using the ascii format. CGROM-Character generator rom, contains dot/pixel patterns for every character to be displayed (pre programmed). Shift registers are used to convert CGROM parallel data to serial data(serializing), drivers are required to drive (ON/OFF) the bits, refreshing logics are required to hold the display data, as the dots are displayed row by row basis continuously, like in CRT.

DDRAM **:** The data to be displayed on LCD, is to be written on to the display data RAM using ASCII format, i.e. if 'A' is to be displayed, ASCII code and A. i.e. 65 to be written. The first 40 bytes (00 – 39 decimal) of RAM allocated for 1st Line, another 40 bytes (40H – 40Hto 79 decimal) of RAM to 2nd line.



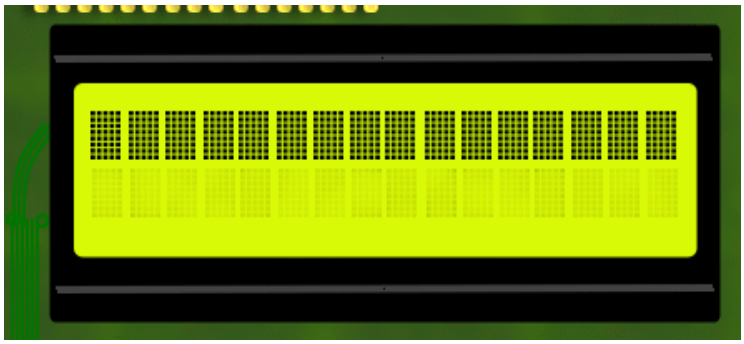
Also first 16 locations are allocated to corresponding 16 display digits of 1st line. If we want to write to the 4th digit, we have to write to the 4th location in DDRAM. Even though display shows 16 digits at a time, memory holds 40 characters for each line, so using shift command we can rotate the display to make all the 40 characters visible.

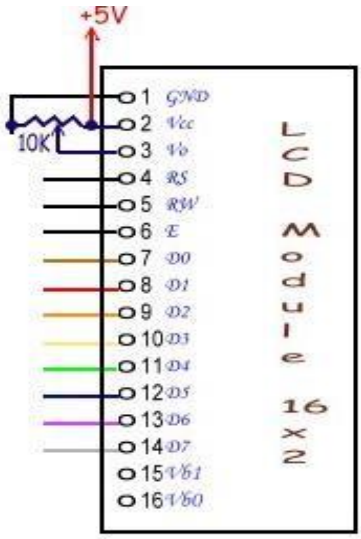
CGROM : character generator ROM : it contains dot/pixel patterns for every character to be displayed (pre prog rammed), in 5 x 7 mode, 7 bytes of CGROM required to produce the character on the display so CGROM is used to convert asci code to dot patterns on display.

Shift – reg's, bit/pixel drives and refreshing logic: Shift registers used to convert CGROM parallel data to serial data (serialising), drives required to drive (on/off) the bits, refreshing logics are required to hold the display data, as the dots are displayed row by row basis continuously, like in CRT.

Interfacing :

Pin Details : LCD provides many control pins, to enable the microcontroller or





microprocessor to communicate, whatever the data we write to LCD is of two types, either it is a command to the LCD(to configure) or ASCII code of character to be displayed on LCD (to DDRAM). RS signal is used for this,

RS (Register Select): 0 or 1

0 - writing command byte into command register of LCD

1 - writing data (ASCII code) into Data register of LCD

R/W : (Read/Write) (Note: Many a times, R/W is grounded)

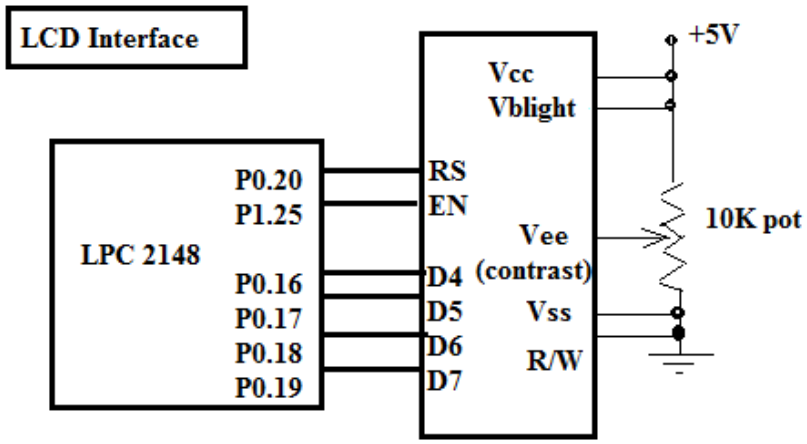
0- Write to LCD (Data/Command)

1- Read from the LCD

E (Enable) : 1 to 0 pulse

- Enable is required to perform the writing/reading to LCD, E – '1' (for 450nsec) & then '0' (High to Low Pulse)

D0-D7 - It is a bidirectional data bus, used to write data/command to LCD or reading status. In 4bit nibble mode, only lines D4 – D7 are used for communication.



Embedded software / Driver Program for 4 x 20 alphanumeric LCD

LCD Command Table

Instruction	D7	D6	D5	D4	D3	D2	D1	D0	Description
Clear display	0	0	0	0	0	0	0	1	Clears Display and returns cursor to home position.
Cursor home	0	0	0	0	0	0	1	X	Returns cursor to home position. Also returns display being shifted to the original position.
Entry mode set	0	0	0	0	0	1	I/D	S	I/D = 0 - cursor is in decrement position. I/D = 1 - cursor is in increment position. S = 0 - Shift is invisible. S = 1 - Shift is visible
Display ON- OFF Control	0	0	0	0	1	D	C	B	D- Display, C- Cursor, B-Blinking cursor 0 - OFF 1 - ON
Cursor/ Display Shift	0	0	0	1	S/C	R/L	X	X	S/C = 0 - Move cursor. S/C = 1 - Shift display. R/L = 0 - Shift left. R/L = 1- Shift right.
Function Set	0	0	1	DL	N	F	X	X	DL = 0 - 4 bit interface. DL = 1 - 8 bit interface. N = 0 - 1/8 or 1/11 Duty (1 line). N = 1 - 1/16 Duty (2 lines). F = 0 - 5x7 dots. F = 1 - 5x10 dots.

Two steps are involved,

1. Configure the LCD for different parameters/settings, by writing series of commands (command bytes) like
    - Function set command(0x28)
    - Display On command(0x0C)
    - Clear display (0x01)

2. Writing actual string data to LCD, character by character, (by default characters are displayed from line1 first column position, we can issue DDRAM address command - 0x80 + char pos, for first line, 0xc0 + char pos, for second line,0x94+char pos, for third line, 0xD4+char pos, for fourth line).

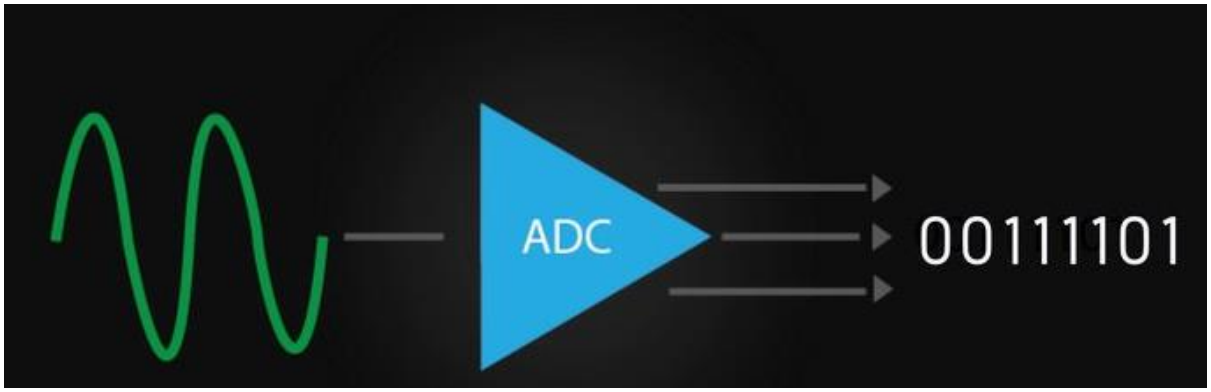
```
//Alpha-numeric LCD Interface (4Lines,20characters)
//Connected in 4bit nibble mode
//LCD handshaking:RS->P0.20,EN->P0.25 ,R/W -Gnd
//LCD data:D4,D5,D6,D7 -> P0.16,P0.17,P0.18,P0.19

#include <lpc214x.h>
#define PLOCK 0x00000400
#define LED_OFF (IO0SET = 1U << 31)
#define LED_ON  (IO0CLR = 1U << 31)
#define RS_ON  (IO0SET =  1U << 20)
#define RS_OFF (IO0CLR = 1U << 20)
#define EN_ON  (IO1SET = 1U << 25)
#define EN_OFF (IO1CLR = 1U << 25)

void SystemInit(void);
static void delay_ms(unsigned int j);//millisecond delay
static void delay_us(unsigned int count);//microsecond delay
static void LCD_SendCmdSignals(void);
static void LCD_SendDataSignals(void);
static void LCD_SendHigherNibble(unsigned char dataByte);
static void LCD_CmdWrite( unsigned char cmdByte);
static void LCD_DataWrite( unsigned char dataByte);
static void LCD_Reset(void);
static void LCD_Init(void);
void LCD_DisplayString(const char *ptr_stringPointer_u8);
int main()
{
    SystemInit();
    IO0DIR |= 1U << 31 | 0x00FF0000 ; // to set P0.16 to P0.23 as o/ps
    IO1DIR |= 1U << 25;                    // to set P1.25 as o/p used for EN
                                           // make D7 Led on off for testing
    LED_ON; delay_ms(500);LED_OFF;delay_ms(500);
    LCD_Reset();
    LCD_Init();
    delay_ms(100);
    LCD_CmdWrite(0x80); LCD_DisplayString("RV College Of Engrng");
    LCD_CmdWrite(0xc0); LCD_DisplayString("   Computer Sciene");
    LCD_CmdWrite(0x94); LCD_DisplayString("    4th Semester");
    LCD_CmdWrite(0xD4); LCD_DisplayString("      B Section");
    while(1);
}
static void LCD_CmdWrite( unsigned char cmdByte)
{
   LCD_SendHigherNibble(cmdByte);
    LCD_SendCmdSignals();
    cmdByte = cmdByte << 4;
    LCD_SendHigherNibble(cmdByte);
    LCD_SendCmdSignals();
}
```

```
static void LCD_DataWrite( unsigned char dataByte)
{
    LCD_SendHigherNibble(dataByte);
    LCD_SendDataSignals();
    dataByte = dataByte  <<  4;
    LCD_SendHigherNibble(dataByte);
    LCD_SendDataSignals();
}
static void LCD_Reset(void)
{
  /* LCD reset sequence for 4-bit mode*/
    LCD_SendHigherNibble(0x30);
    LCD_SendCmdSignals();
    delay_ms(100);
    LCD_SendHigherNibble(0x30);
    LCD_SendCmdSignals();
    delay_us(200);
    LCD_SendHigherNibble(0x30);
    LCD_SendCmdSignals();
    delay_us(200);
    LCD_SendHigherNibble(0x20);
    LCD_SendCmdSignals();
    delay_us(200);
}
static void LCD_SendHigherNibble(unsigned char dataByte)
{
   //send the D7,6,5,D4(uppernibble) to P0.16 to P0.19
    IO0CLR = 0X000F0000;IO0SET = ((dataByte >>4) & 0x0f) << 16;
}
static void LCD_SendCmdSignals(void)
{
    RS_OFF;  // RS - 1
    EN_ON;delay_us(100);EN_OFF; // EN - 1 then 0
}
static void LCD_SendDataSignals(void)
{
    RS_ON;// RS - 1
    EN_ON;delay_us(100);EN_OFF; // EN - 1 then 0
}
static void LCD_Init(void)
{
          delay_ms(100);
          LCD_Reset();
          LCD_CmdWrite(0x28u);   //Initialize the LCD for 4-bit 5x7 matrix type
          LCD_CmdWrite(0x0Eu);  // Display ON cursor ON
          LCD_CmdWrite(0x01u);  //Clear the LCD
          LCD_CmdWrite(0x80u);  //go to First line First Position
}
```

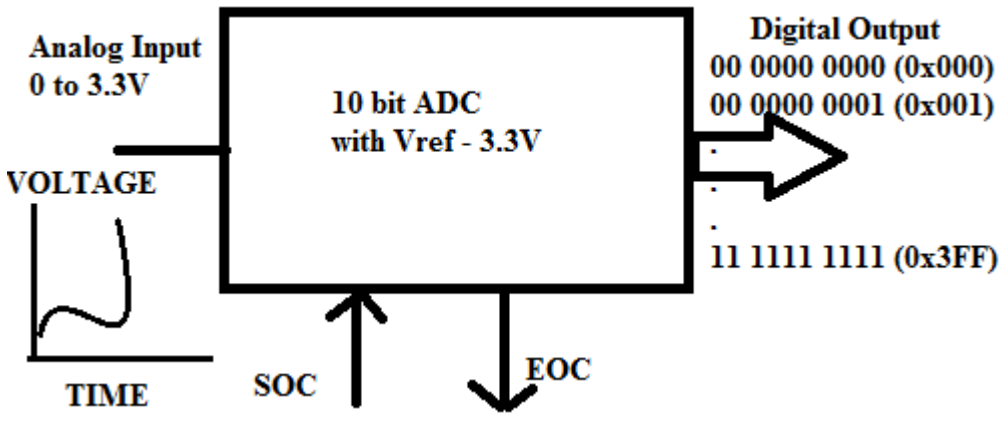
```
void LCD_DisplayString(const char *ptr_string)
{
          // Loop through the string and display char by char
    while((*ptr_string)!=0)
       LCD_DataWrite(*ptr_string++);
}
static void delay_us(unsigned int count)
{
        unsigned int j=0,i=0;
        for(j=0;j<count;j++)
        {
          for(i=0;i<10;i++);
        }
}
void SystemInit(void)
{
  PLL0CON = 0x01;
  PLL0CFG = 0x24;
  PLL0FEED = 0xAA;
  PLL0FEED = 0x55;
  while( !( PLL0STAT & PLOCK ))
  { ; }
  PLL0CON = 0x03;
  PLL0FEED = 0xAA;  // lock the PLL registers after setting the required PLL
  PLL0FEED = 0x55;
  VPBDIV = 0x01;      // PCLK is same as CCLK i.e 60Mhz
}
void delay_ms(unsigned int j)
{
 unsigned int x,i;
 for(i=0;i<j;i++)
  {
    for(x=0; x<10000; x++);
  }
}
```

ADC interfacing

ADCs finds applications in weight measurement, temperature measurement, speed measurement etc. In general, different physical quantities like temperature, humidity are converted by sensors in to electrical domain i.e voltage or current, which in turn fed to ADC for conversion to digital domain, for doing different types of processing. ADCs are built using techniques like dual slope conversion, successive approximation type etc.



A typical ADC has, has analog input (one which is varying with time), and corresponding digital output, as shown below.



SOC – Start of conversion, indicates initiation of conversion process.

EOC – End of Conversion signal, indicates conversion of analog input to digital data is completed. Some older embedded systems, external ADCs

like ADC 0808/9, ADC0804 were used with the microcontrollers for Analog input interfacing. Some ADC's like ADC0808/9 has multiple analog inputs (they have inbuilt analog multiplexer), so that simultaneously we can connect multiple sensors to one ADC. Now most of the latest microcontrollers, like LPC 2148 provide built in ADC's with multiple channels.

Resolution :

For 8 bit ADC , digital input - 00 to FF, analog output – 0 to 5V

Resolution = 5 V / 2^8 = 20 mv (appr), that means minimum of 20 mv is required to produce 1 bit change at the output.

For a given analog input,

digital_value = (input / 5) x 256 [= analog_input / resolution]

For 10 bit ADC like LPC 2148, digital input – 000 to 3FF, analog output – 0 to 3.3v

Resolution = 3.3 V / 2^{10} = 3.2 mv (appr), that means minimum of 3.2 mv is required to produce 1 bit change at the output.

For a given analog input,

digital_value = (input / 3.3) x 1024 [= analog_input / resolution]

$$\text{Step Size} = \frac{Input\ Range}{Resolution} = \frac{3.3V}{2^{10}}$$

$$= \frac{3.3}{1023} = 0.0032258064\,V$$

$$= 3.23\,\text{mV (Approx.)}$$

ADC Module of LPC 2148:

Analog Signal Interfacing using LPC 2148

Features

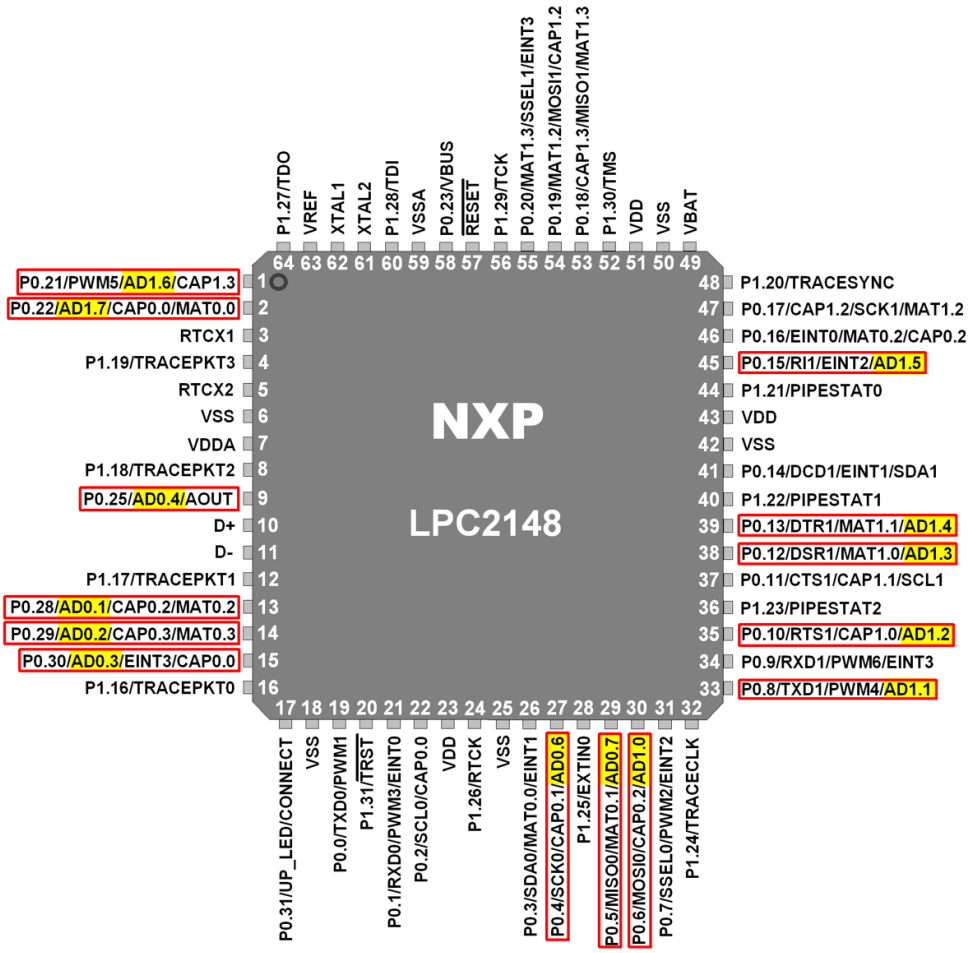
LPC 2148 provides two ADC's / Analog channels: ADC0 & ADC1. ADC0 provides 6 analog inputs, ADC1 provides 8 analog inputs. The features of ADC are:

- It is of 10 bit successive Approximation type analog to digital converter (2nos)

- Input multiplexing among 6 or 8 pins (ADC0 & ADC1)
- POWER-DOWN mode
- Measurement range 0 to Vref (typically 3V, not to exceed VDDA voltage level)
- 10 bit conversion time >= 2.44µs
- Burst conversion mode or single or multiple inputs
- Optional conversion on transition on input pin or Timer Match signal.
- Global start command for both converters
- it can generate 400000 samples/sec

Basic clocking for the ADC is provided by the APB block . A programmable divider I included in each converter, to scale this clock to the 4.5MHz(max) clock needed by the successive approximation process.

ADC Interfacing:



Pins provided by ADC:

ADC0 CHANNEL (6 analog inputs)

AD0.7 (P0.5),

AD0.6 (P0.4),

AD0.4 (P0.25),

AD0.3 (P0.30),

AD0.2 (P0.29),

AD0.1 (P0.28),

ADC1 CHANNEL (8 analog inputs),

AD1.0 to AD1.7 are the input analog pins and the corresponding port numbers are

[P0.6, P0.8 ,P0.10, P0.12, P0.13,P0.15, P0.21, P0.22]

The ADC cell can measure the voltage on any of these input signals. Make sure, not to apply any voltage more than 3.3V on these pins, to avoid error readings and possible damage to the pins.

ADC 0		ADC 1	
ADC Channel 0	LPC2148 Pins	ADC Channel 1	LPC2148 Pins
AD0.1	P0.28	AD1.0	P0.6
AD0.2	P0.29	AD1.1	P0.8
AD0.3	P0.30	AD1.2	P0.10
AD0.4	P0.25	AD1.3	P0.12
AD0.6	P0.4	AD1.4	P0.13
AD0.7	P0.5	AD1.5	P0.15
		AD1.6	P0.21
		AD1.7	P0.22

VREF: Voltage Reference, this provides voltage reference for the ADC circuit.

VDDA,VSSA : Analog Power and Ground. These should be nominally the same voltages as VDD and VSS, but should be isolated to minimize noise and error.

ADC Programming

ADC block provides many registers to the programmer, the important of them are:

Register Name	Register Function
ADCR	A/D Control Register. The ADCR register must be written to select the operating mode before A/D conversion can occur.
ADGDR	A/D Global Data Register. This register contains the ADC's DONE bit and the result of the most recent A/D conversion.
ADSTAT	A/D Status Register. This register contains DONE and OVERRUN flags for all of the A/D channels, as well as the A/D interrupt flag.
ADGSR	A/D Status Register. This register contains DONE and OVERRUN flags for all of the A/D channels, as well as the A/D interrupt flag.
ADINTEN	A/D Interrupt Enable Register. This register contains enable bits that allow the DONE flag of each A/D channel to be included or excluded from contributing to the generation of an A/D interrupt.
ADDRX	A/D Channel X Data Register. This register contains the result of the most recent conversion completed on channel X.

ADCR [AD0CR, AD1CR] – ADC Control Register

A/D control register, this register must be written to select the operating mode A/D conversion can occur.

ADGDR [AD0GDR, AD1GDR] – ADC Global Data Register

This register contains the ADCs DONE bit and result of the most recent conversion

ADSTAT [AD0STAT, AD1STAT] – ADC Status Register

This register contains the DONE and OVERRUN flags for all of the ADC channels, as well as the ADC interrupt flag

ADGSR –ADC Global start register

This address can be written to start conversion simultaneously in both the ADC converters.

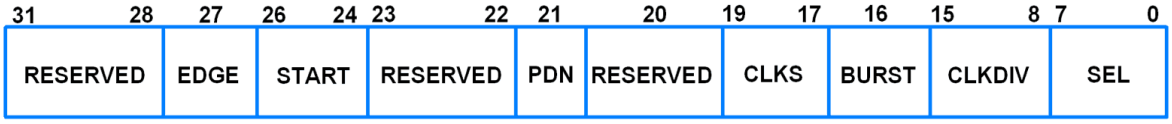
ADINTEN [AD0INTEN ,AD1INTEN] – ADC Interrupt Enable Register

This register contains enable bits that allow the DONE flag of each ADC channel to be included or excluded from contributing to the generation of an ADC interrupt.

ADDR0 to ADDR7 [AD0R0 – AD0R7 , AD1R0 – AD1R7]:Data Registers

These channel data registers contains the result of the most recent conversion completed on that particular channels.

ADC Control Register (32 bit Register) :



Bits - 7 to 0: (SEL) Selects which of the AD0.7 : AD0.0 / AD1.7 :AD1.0 to be sampled and converted. Example: for ADC0, bit 0 selects Pin AD0.0, and bit 7 selects AD0.7. In software-controlled mode, any one of these bits should be set to 1.

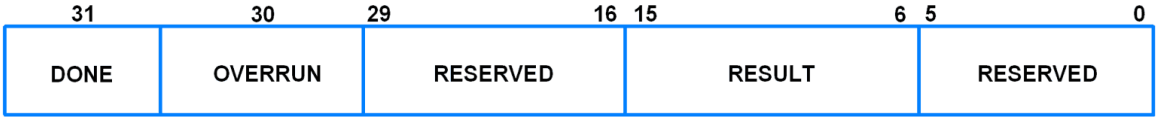
Bits - 15 to 8: (CLKDIV)- PCLK is divided by (this value + 1) to get ADC clock, which should be less than or equal to 4.5MHz

Bits – 21 : (PDN) : 1 – The ADC is operational , 0 – the ADC is in Power Down mode

Bits – 26 to 24 : 001 Start Conversion Now

(Note: Unused/reserved bits should not be written with 1)

ADC Global Data Register (32 bit Register)

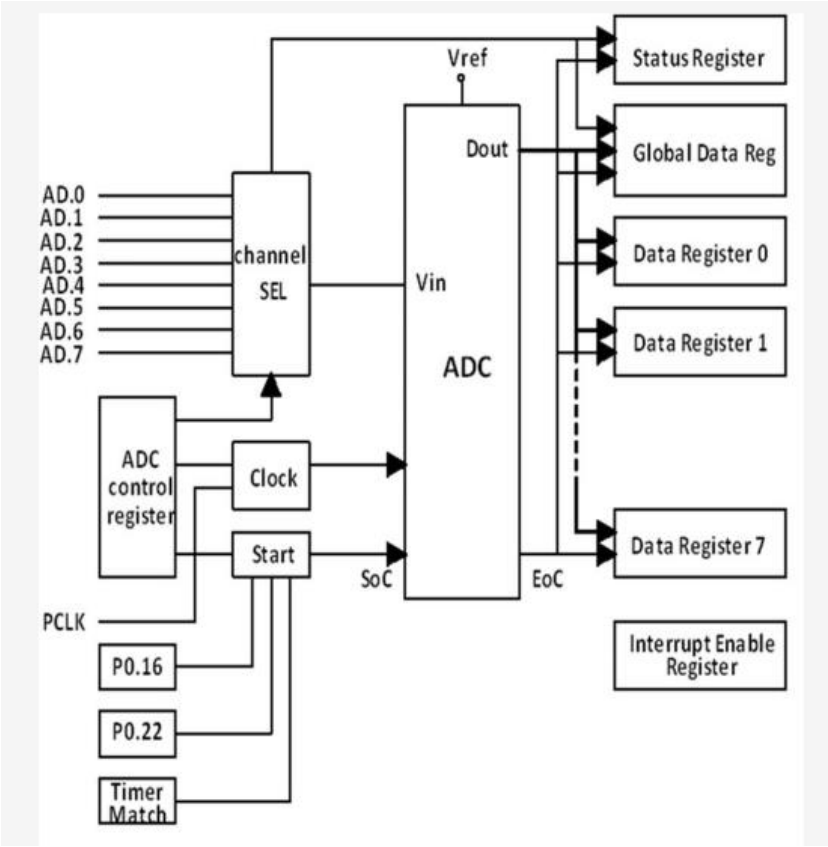


This register contains the ADC's DONE bit and the result of the most recent A/D conversion.

Bit 5:0 - Reserved

Bits 15:6 - Result ,when DONE bit is set to 1, this field contains 10 bit ADC result 0 to 1023, correspond to 0V to 3.3V (Vref)

Bit 31 - DONE, This bit set to 1, when conversion completes, It is cleared when this register is read and when the AD0CR is written



Program Steps to Read Analog Input :

1. Select the analog input pin (1 to 6 of ADC0 or 0 to 7 of ADC1) and configure the pin as Analog input using PINSEL register
2. Write to ADC control Register:
   - Make ADC operational (set the Bit 21)
   - Select the channel(set any one bit of Bit0 to Bit7 – for AD0 to AD7)
   - Issue SOC signal (set 001 on bits : 26 25 24)

   Example: select AD0.1

   AD0CR = (1 << 1) | (1 << 21) | (1<<24) ; // assuming other bits are zero
3. Check for the conversion to complete, by reading Bit31 of GDR

   Example: for ADC0

   while((AD0GDR & (unsigned long) 1 << 31) == 0);
4. Read the Digital output from GDR, after aligning the result to LSB (Bit0) and masking other bits to Zero

   Example:

   i = (AD0GDR >> 6) & 0x3FF ;

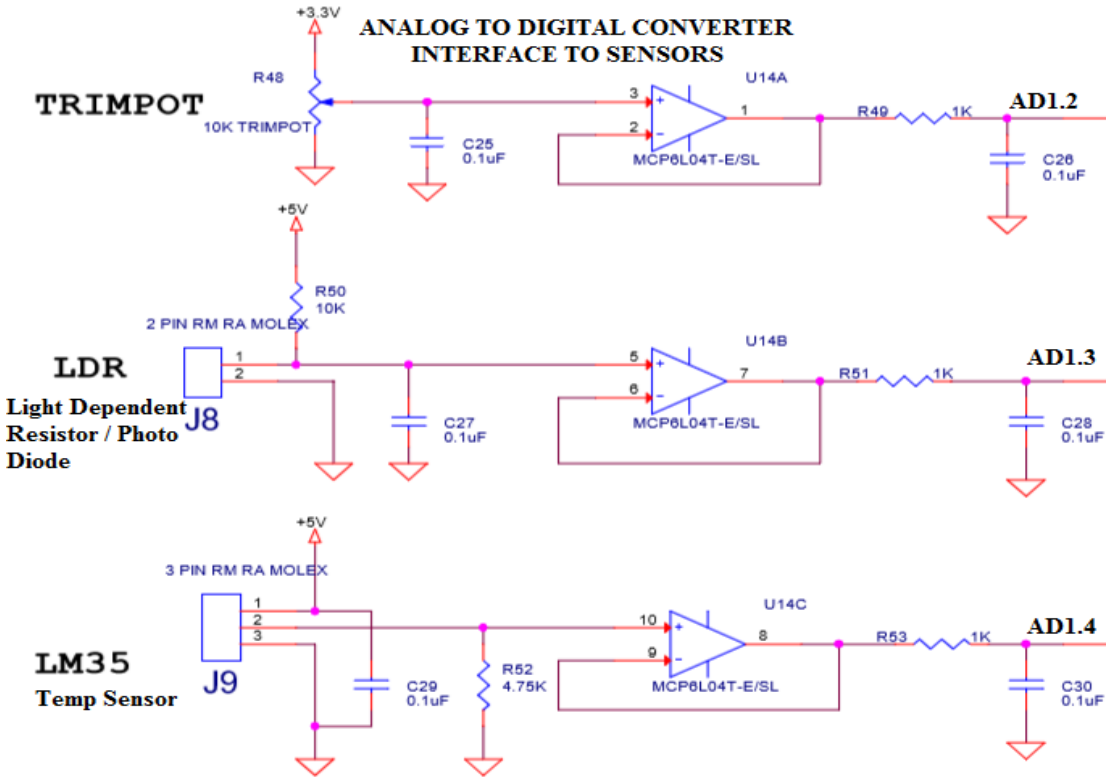
Example Program :

LDR (Light Dependant Resistor) / PhotoDiode is connected to P0.28 and LED is connected to P0.21, Read the Light intensity. Make the LED on whenever there is less/no light.

```
#include <lpc214x.h>
#define LED_ON  IO0SET  = 1 << 21
#define LED_OFF IO0CLR = 1 << 21
int main()
{
        unsigned int i;

        IO0DIR = (1 << 21) ;  // P0.21-o/p
        PINSEL1 = 1 << 24;  // P0.28 AS AD0.1(01)
        LED_ON;
        do
        {
                AD0CR= (1 << 1 | 1 << 21 | 1 << 24) ;
                While ( (AD0GDR & (unsigned long) 1 << 31) == 0);
                i = (AD0GDR >> 6 ) & 0x3FF ;
                if (i > 100)
                        LED_OFF;
                else
                        LED_ON;

        }
        while(1);
}
```

Interfacing of Potentiometer , LDR and Temperature Sensor (LM35)

Here ADC channel 1 is selected, and three ADC inputs AD1.2,AD1.3 and AD1.4 are used to read the the analog voltage from trimpot,LDR and LM35. All the inputs are connected to LPC 2148 through voltage followers for protection and noise reduction.



Software function to read the ADC

This function is a generic function used to read the analog input from any of the two channels. It takes two arguments, first argument indicates the channel number (0 or 1), second argument indicates the input number for the selected channel. Ex: adc(1,4) – means reading from AD1.4

```
unsigned int adc(int no,int ch)
{
        // adc(1,4) for temp sensor LM35, digital value will increase as temp increases
        // adc(1,3) for LDR - digival value will reduce as the light increases
        // adc(1,2) for trimpot - digital value changes as the pot rotation
        unsigned int val;
        PINSEL0 |=  0x0F300000;   /* Select the P0_13 AD1.4 for ADC function */
                                  /* Select the P0_12 AD1.3 for ADC function */
                                 /* Select the P0_10 AD1.2 for ADC function */
       switch (no)       //select adc
       {
          case 0: AD0CR = 0x00200600 | (1<<ch);       //select channel
                AD0CR | = (1<<24) ;                   //start conversion
                while ( ( AD0GDR &  ( 1U << 31 ) ) == 0);
```

```
                val = AD0GDR;
                break;

            case 1: AD1CR = 0x00200600  | ( 1 << ch );       //select channel
                AD1CR |=  ( 1 << 24 ) ;                     //start conversion
                while ( ( AD1GDR & (1U << 31) ) == 0);
                val = AD1GDR;
                break;
        }
   val = (val  >>  6) & 0x03FF;        // bit 6:15 is 10 bit AD value
   return  val;
}
```

Program to Control the DC Motor speed, using Potentiometer / Temperature sensor.

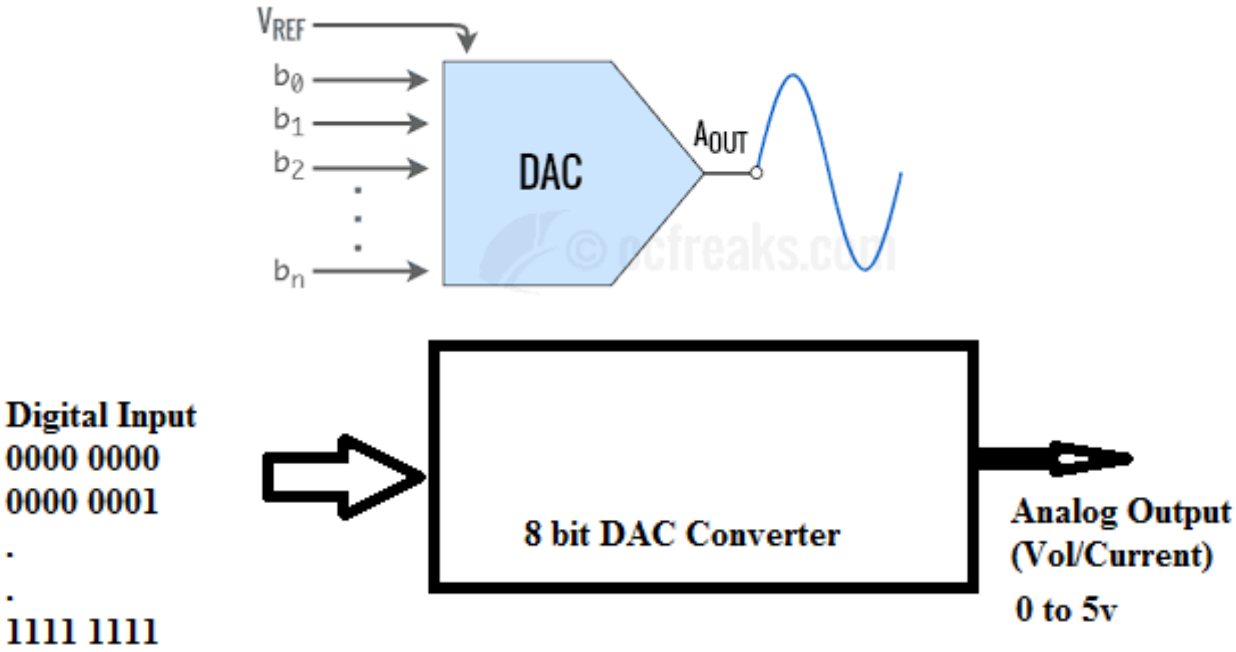
```
//DC Motor Speed Control
//P0.28 - used for direction control
//P0.9  - used for speed,generated by PWM6
//duty cycle - 0 to 100 controlled by PWM, fed from Potentiameter/Temp Sensor connected
to ADC

#include <lpc214x.h>
void delay_ms(unsigned int j);
void SystemInit(void);
void runDCMotor(int direction,int dutycycle);
unsigned int adc(int no,int ch);

int main()
{
      int dig_val;
      IO0DIR |= 1U << 31 | 0x00FF0000 | 1U << 30; // to set P0.16 to P0.23 as o/ps
      SystemInit( );
      do{
         dig_val = adc(1,2) / 10;  // reading from potentiameter
         if(dig_val > 100) dig_val =100;
         runDCMotor(2,dig_val);
      }
      while(1);
}
```

Note: SystemInit( ) and runDCMotor( ) are explained in the previous sections.

DAC Interface

D to A converter is used to convert digital data (binary word) to a proportional analog voltage or current.

Example: DAC-0800, an 8 bit DAC from National Semiconductors, with settling time around 100 nsec is very commonly used with 8 bit microcontrollers.



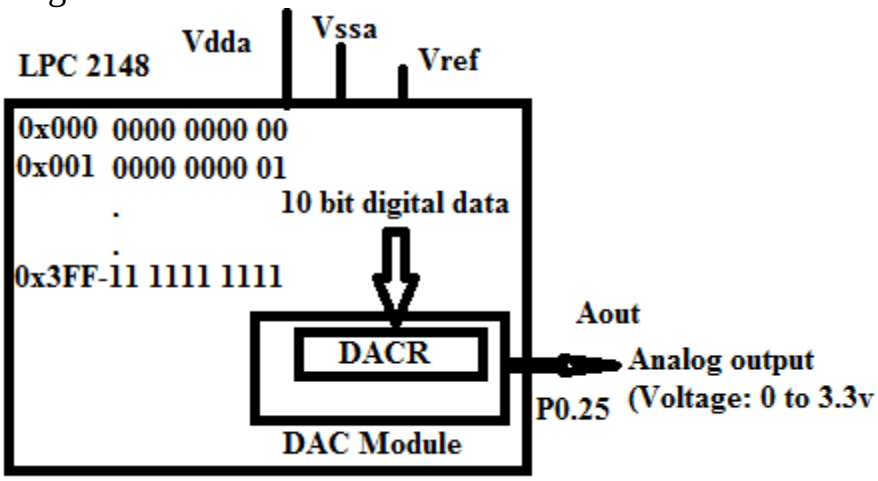
An 8 bit DAC has 2^8 =256 possible output voltage/current levels, so its resolution is 1 part in 256. The digital input : 00 – FF is mapped to 0 – 5V o/p, then

(1/256) * 5V = 20mv (approx.) is the resolution.

More the number of input DAC bits, better is the resolution. Most of the advanced and latest microcontrollers (like LPC 2148), has built in DAC module. So, we don't require external ICs like DAC-080.

DAC Module of LPC 2148:

LPC 2148, provides in-built 10-bit Digital to Analog Converter, as shown in the figure below.

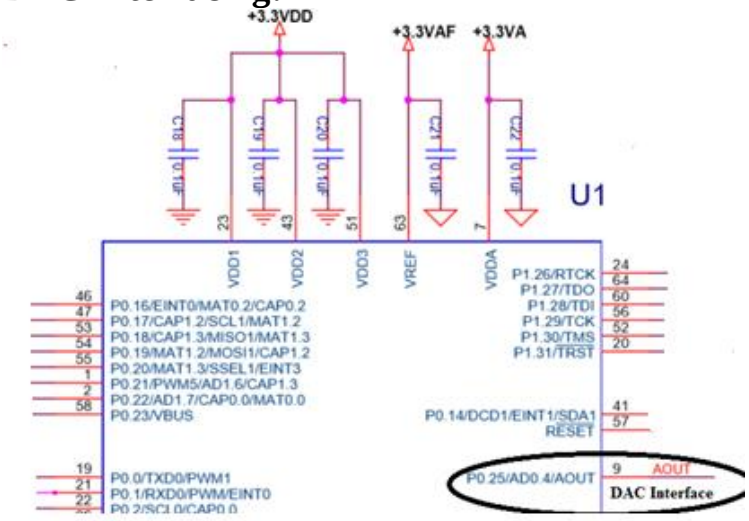


- DAC module of LPC 2148 is a 10 bit Digital to Analog converter used to convert 10 bit Digital data to corresponding Analog voltage.
- Digital I/P : **000 to 3FF (0 to 1023)**, corresponding Analog O/P : **0V to 3.3V**
- Resolution = (3.3/1024) ≈ **3.2mili volts**

Some of the features of DAC module are:

- Resistor string architecture,
- Buffered output,
- power-down mode,
- selectable speed Vs power.

DAC Interfacing:



Following are the DAC related pins, provided by LPC 2148, which is of use, while using the DAC module.

Aout - the DAC output pin, which produces analog voltage in the range of 0 to 3.3V (assuming Vref is 3.3V)
V_{DDA},V_{SSA} : are Voltage supply (3.3V) & Ground pins required for DAC Module.
V_{REF} – is the reference voltage required for DAC (refer internet, related to the construction of DAC and use of Ref. Voltage).

Pin	Type	Description
AOUT	Output	**Analog Output.** After the selected settling time after the DACR is written with a new value, the voltage on this pin (with respect to V_{SSA}) is VALUE/1024 * V_{REF}.
V_{REF}	Reference	**Voltage Reference.** This pin provides a voltage reference level for the D/A converter.
V_{DDA}, V_{SSA}	Power	**Analog Power and Ground.** These should be nominally the same voltages as V_3 and V_{SSD}, but should be isolated to minimize noise and error.

V_{DDA} – Analog 3.3V Power supply: This should be nominally the same voltage as V_{DD}, but should be isolated to minimize noise and error. This voltage is used to power the ADC & DAC. This pin must be tied to V_{DD}, when ADC & DAC are not used.

V_{SSA} – Analog Ground: 0V reference. This should be nominally the same voltage as V_{SS}, but should be isolated to minimize noise and error. This pin must be grounded if ADC/DAC is not used.

V_{REF} – A/D Converter Reference: This should be nominally the same voltage as V_{DD}, but should be isolated to minimize noise and error. Level on this pin is used as a reference for ADC & DAC. This pin must be tied to V_{DD} when the ADC/DAC is not used.

DAC Programming: Programming DAC, meaning generating required analog output voltage, involves following steps.

1. **Configuring P0.25 as A_{OUT}**: Bits 19:18 of the PINSEL1 register control whether the DAC is enabled and controlling the state of the pin P0.25/AD0.4/AOUT. When these bits are "10" , the DAC is powered on and active.

```
PINSEL1 |= 0x00080000;     /* P0.25 as DAC output : option 3  ->  10 (bits18,19)*/
```

2. **Writing 10bit digital data to DACR** (DAC register) : DACR is a 32 bit register, where the bits D15:6 (10bits) representing VALUE is used to write digital data to the DAC module.

DAC Register (DACR - 0xE006 C000)

This read/write register includes the digital value to be converted to analog, and a bit that trades off performance vs. power. Bits 5:0 are reserved for future, higher-resolution D/A converters.

Table 287: DAC Register (DACR - address 0xE006 C000) bit description

Bit	Symbol	Value	Description	Reset value
5:0	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
15:6	VALUE		After the selected settling time after this field is written with a new VALUE, the voltage on the A_{OUT} pin (with respect to V_{SSA}) is VALUE/1024 × V_{REF}.	0
16	BIAS	0	The settling time of the DAC is 1 µs max, and the maximum current is 700 µA.	0
		1	The settling time of the DAC is 2.5 µs and the maximum current is 350 µA.	
31:17	-		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

Waveform Generation Using DAC Module:

Different waveforms like square,triangular,sawtooth,sine can be generated using DAC module, by producing different analog voltages with reference to time. This is achieve using two methods.

1. Using the program to compute analog voltage at run time. Ex: by executing the following code, we will be outputting 000 (Ov) and 3FF (3.3V) continuously and generating the square waveform

```
while(1)
 {
    value = 0x000;
     DACR =  (value << 6) ; // bits D15-D6 refers to 10 bit data
     value = 0x3FF;
     DACR =  (value << 6) ;
 }
```

2. When we have to produce complex waveforms like sine, we require functions like sin( ) to compute the values, at run time. Since embedded system are not compute capable and requirement of real time data, the pre-computed values for different sine angles are stored in program/code memory are used to generate waveforms. This technique is called as look-up table method of generating the waveforms. Ex:

```
unsigned short int table[ ] = {0x010,0x028,0x0x45,0x67,..};
count = 50;
while(count--)
 {
   DACR = table[count] << 6;
 }
```

Write an Embedded C program to generate sine, full rectified sine, Triangular, Sawtooth and Square waveforms using DAC module.

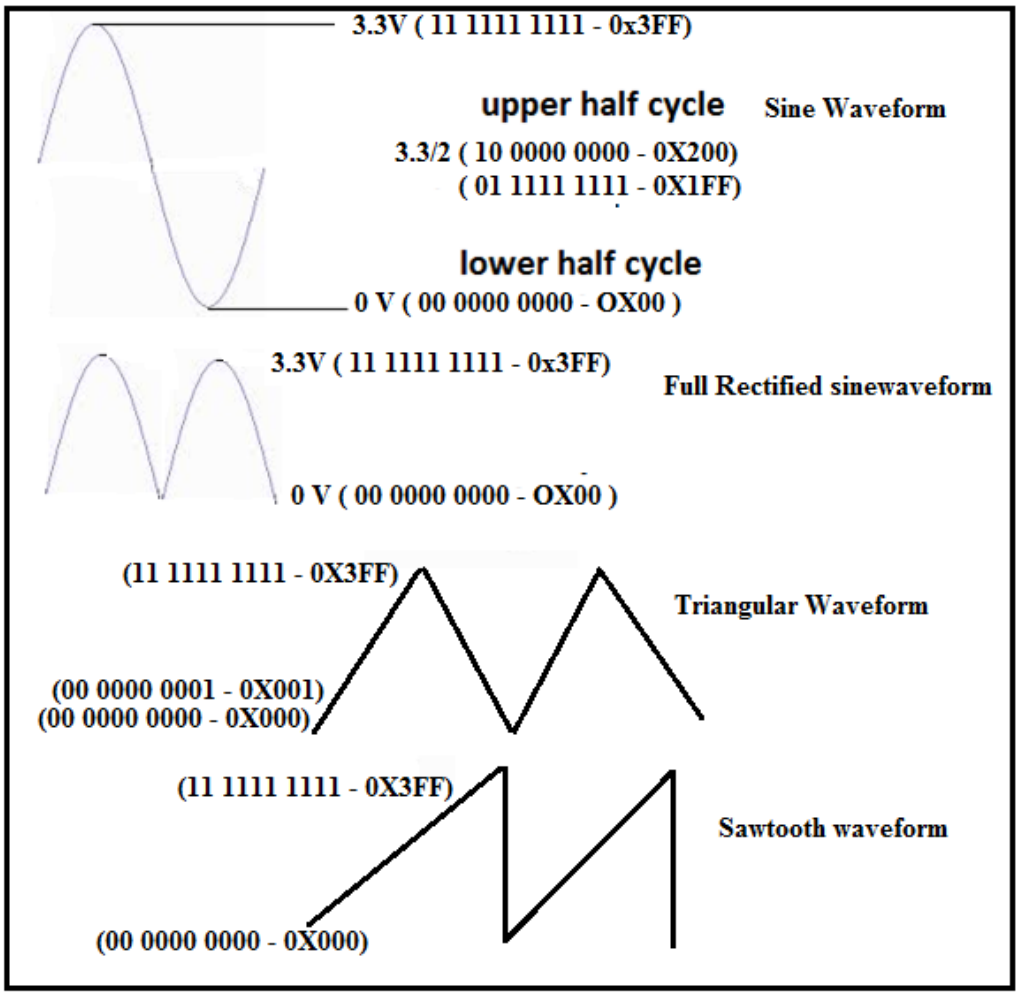
Look up Table Creation:

Look up tables are used extensively in embedded systems, to store precomputed digital values, corresponding to analog voltages and used to generate different waveforms using DAC Module. Here the explanation about creating sine table is given.

Formula for calculation of the sine table entries: **512 + 511 x Sin Θ**
(512 Corresponds to 1FFh, i.e. 3.3/2 V, 511 x SIN 90 gives 511, so 512 + 511 = 1023 (for 3.3V). Calculate the digital values to be outputted to DAC for angles in the steps of 6^o,

511 x sin 0 = 0	511 x sin 48 = 380
511 x sin 6 = 53	511 x sin 54 = 413
511 x sin 12 = 106	511 x sin 60 = 442
511 x sin 18 = 158	511 x sin 66 = 467
511 x sin 24 = 208	511 x sin 72 = 486
511 x sin 30 = 256	511 x sin 80 = 503
511 x sin 36 = 300	511 x sin 86 = 510
511 x sin 42 = 342	511 x sin 90 = 511

Output the above values in the reverse order to get other portion of the top half cycle, (add 512 for top half cycle, and subtract from 512 for the lower half cycle, refer the table declaration).


3.3V (11 1111 1111 - 0x3FF)
upper half cycle
Sine Waveform
3.3/2 (10 0000 0000 - 0X200)
(01 1111 1111 - 0X1FF)
lower half cycle
0 V (00 0000 0000 - OX00)
3.3V (11 1111 1111 - 0x3FF)
Full Rectified sinewaveform
0 V (00 0000 0000 - OX00)
(11 1111 1111 - 0X3FF)
Triangular Waveform
(00 0000 0001 - 0X001)
(00 0000 0000 - 0X000)
(11 1111 1111 - 0X3FF)
Sawtooth waveform
(00 0000 0000 - 0X000)

```
#include <lpc214x.h>
#include <stdio.h>
#define SW2 (IO0PIN & (1 << 14))
#define SW3 (IO0PIN & (1 << 15))
#define SW4 (IO1PIN & (1 << 18))
#define SW5 (IO1PIN & (1 << 19))
#define SW6 (IO1PIN & (1 << 20))
static void delay_ms(unsigned int j); //millisecond delay
short int sine_table[ ] =
{512+0,512+53,512+106,512+158,512+208,512+256,512+300,512+342,512+380,512+413,
512+442,512+467,512+486,512+503,512+510,512+511,
512+510,512+503,512+486,512+467,512+442,512+413,512+380,512+342,512+300,512+25
6,512+208,512+158,512+106,512+53,512+0,
512-53,512-106,512-158,512-208,512-256,512-300,512-342,512-380,512-413,512-442,512-
467,512-486,512-503,512-510,512-511,
512-510,512-503,512-486,512-467,512-442,512-413,512-380,512-342,512-300,512-
256,512-208,512-158,512-106,512-53};
short int sine_rect_table[ ] =
{512+0,512+53,512+106,512+158,512+208,512+256,512+300,512+342,512+380,512+413,
512+442,512+467,512+486,512+503,512+510,512+511,
512+510,512+503,512+486,512+467,512+442,512+413,512+380,512+342,512+300,512+25
6,512+208,512+158,512+106,512+53,512+0};

int main()
{
 short int value,i=0;
 PINSEL1 |= 0x00080000;     /* P0.25 as DAC output :option 3 - 10 (bits18,19)*/
 while(1)
          {
             if (!SW2)    /* If switch for sine wave is pressed */
               {
                     while (i!=60 )
                      {
                         value = sine_table[i++];
                         DACR = ( (1<<16) | (value<<6) );
                         delay_ms(1);
                      }
                     i=0;
               }
          else if (!SW3)
              {
                      while ( i!=30 )
                       {
                          value = sine_rect_table[i++];
                          DACR = ( (1<<16) | (value<<6) );
                          delay_ms(1);
                       }
```

```
                i=0;
            }
        else if ( !SW4)          /* 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 ( !SW5 )         /* If switch for sawtooth wave is pressed */
           {
                value = 0;
                while ( value != 1023 )
                 {
                   DACR = ( (1<<16) | (value<<6) );
                   value++;
                 }
           }
        else if ( !SW6 )         /* If switch for square wave is pressed */
          {
                value = 1023;
                DACR = ( (1<<16) | (value<<6) );
                delay_ms(1);
                value = 0;
                DACR = ( (1<<16) | (value<<6) );
                delay_ms(1);
          }
        else     /* If no switch is pressed, 3.3V DC */
         {
                value = 1023;
                DACR = ( (1<<16) | (value<<6) );
        }
    }
}
void delay_ms(unsigned int j)
{
  unsigned int x,i;
  for(i=0;i<j;i++)
   {
     for(x=0; x<10000; x++);
   }
}
```

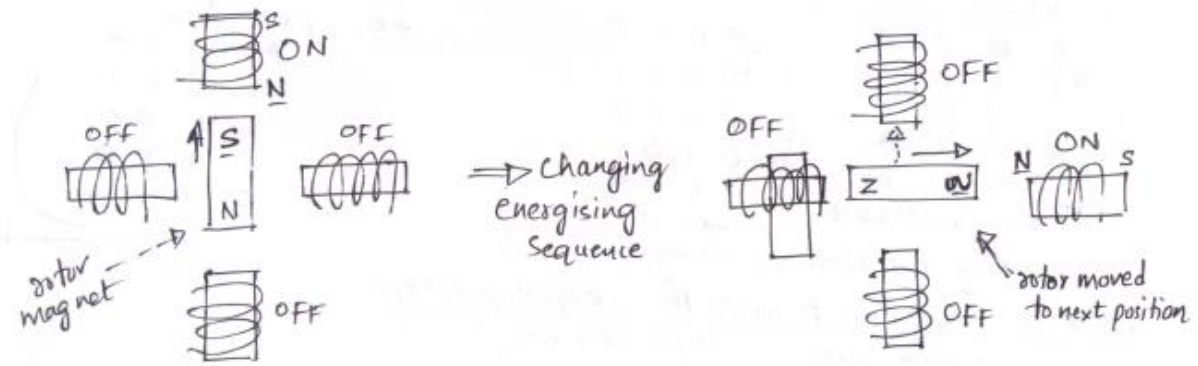
Stepper Motor Interfacing

Stepper motors move in steps unlike dc motors, under the digital control. The number of steps, RPM (no. of revolutions per minute), direction of movement (clockwise /anticlockwise) can be digitally controlled. Stepper motors finds application – in disk drives, printers, plotters, CNC machines, robotics etc.

Resolution: The term stepper motor resolution refers the angular movement, in degree per step.

Ex. 200 steps motor (per revolution) has step angle of = $\frac{360^0}{200} = 1.8^0$

Working principle: Stepper motors – has two parts a) rotor made up permanent magnet or soft iron b) stator – surrounding the rotor. Stator is constructed with the number of coils / windings, which on energising (supplying current) acts like a electromagnet.



As shown in the above diagram rotor (magnet) aligns to a particular position based on the energised coil (acting as electromagnet) as S & N poles gets attracted when we change energisation sequence, rotor aligns to new position, that is how the stepper motors movement is realised. In actual stepper motors – the stator and rotor poles are teethed to achieve higher resolution (more steps per revolution).

Stepper motors are of three types, based on the principle of construction

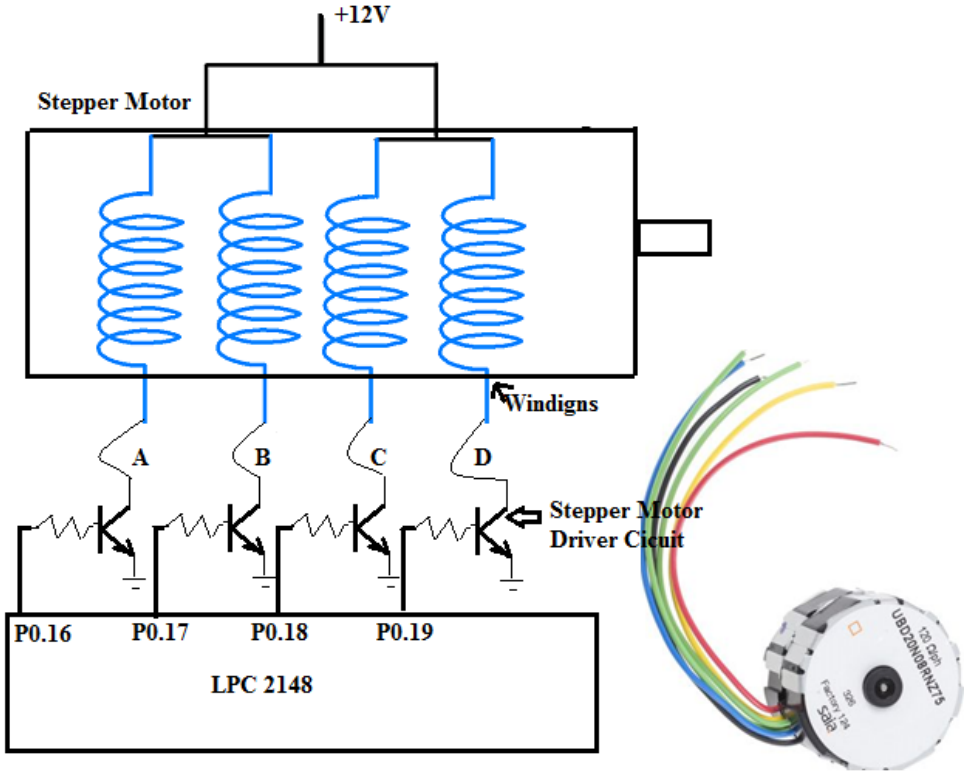
a) Permanent magnet SM (rotors are built using permanent magnets)
b) Variable Reluctance SM (rotors are built using soft iron and are teethed, each time coils are energised rotor moves to less reluctance position)
c) Hybrid motors (combination of permanent magnet and variable reluctance types)

Stepping Modes : Stepper motors movement is achieved by employing one of the following coil energising sequences.

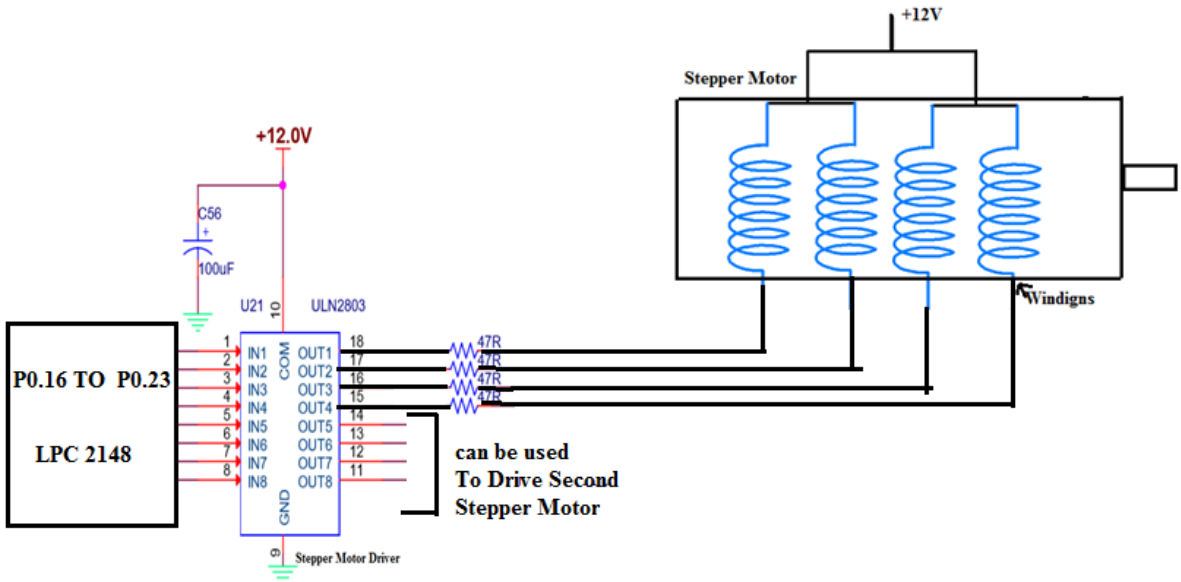
a) Full drive (2 coils are energised at a time, more torque) – 4 step sequence
b) Wave drive (one coil only energised at a time, less torque) – 4 step
c) Half drive (here step angle reduces by half, if 1.8" is step angle, here it becomes 0.9". it is a combination of the above two types, hence 8 step sequence and torque is in between above types)

Interfacing

Commonly available (2Phase, 4winding stepper motors) comes with 5/6 wire connector, 4 for windings, namely A,B,C,D and one/two connections for Power as shown in the figure below. As Microcontroller ports do not drive the current required for windings, transistors are used to derive the required current.



Now stepper motor driving circuit IC's are commonly available. ULN2803 has 8 inbuilt transistor's, which is used in the following circuit to interface two stepper motors.



Excitation sequence : Following table indicates the sequence in which, windings are to be energised to achieve clock wise and anticlockwise direction. Each sequence drives the motor one step (i.e 1.8degree), after 4 steps repeat the sequence to continue rotation in the same direction.

Energising or Driving a winding, means making the current flow in the winding. Applying logic '1' to the input of ULN2803 driver, switches on the transistor present inside the driver and the current flows through the corresponding winding, from the +12V supply.

$A\ B\ C\ D$	$A\ B\ C\ D$		$A\ B\ C\ D$
1 0 0 0	0 0 0 1		1 0 0 1
0 1 0 0	0 0 1 0	Note : Full drive sequence	1 1 0 0
0 0 0 1	0 1 0 0		0 1 1 0
	1 0 0 0		0 0 1 1
→ clockwise	← anticlock wise		

(energising the windings one by one; 1 – energise the winding, 0 – winding not energised)

Embedded C Program / Driver Program for Stepper Motor: Program to rotate stepper motor in clockwise direction for "M" steps, anti-clock wise direction for "N" steps.

- Total number of steps for one revolution = 200 steps (200 teeth shaft)

  Step angle = 360°/200 = 1.8°
- Use appropriate delay in between consequent steps, to achieve required RPM
- 2Phase, 4winding stepper motor is used, along with driver circuit built using the ULN 2803, 12v power is used to drive the stepper motor. Digital input generated by the microcontroller, is used to drive and control the direction and rotation of stepper motors. If it is required to drive bigger/higher torque stepper motors only change is- use MOSFETS or higher power stepper driver ICs to drive motors.

//Stepper Motor Program:
//P0.16 to P0.19 are connected to Windings of SMotor

```
#include <lpc214x.h>
void delay_ms(unsigned int j);
int main()
{
    unsigned int  no_of_steps_clk = 100, no_of_steps_aclk = 100;
    IO0DIR |=  0x00FF0000; // to set P0.16 to P0.23 as o/ps
   do{
   IO0CLR = 0X000F0000;IO0SET = 0X00010000;delay_ms(10);if(--no_of_steps_clk == 0) break;
   IO0CLR = 0X000F0000;IO0SET = 0X00020000;delay_ms(10);if(--no_of_steps_clk == 0) break;
   IO0CLR = 0X000F0000;IO0SET = 0X00040000;delay_ms(10);if(--no_of_steps_clk == 0) break;
   IO0CLR = 0X000F0000;IO0SET = 0X00080000;delay_ms(10);if(--no_of_steps_clk == 0) break;
   }while(1);
   do{
   IO0CLR = 0X000F0000;IO0SET = 0X00080000;delay_ms(10);if(--no_of_steps_aclk == 0) break;
   IO0CLR = 0X000F0000;IO0SET = 0X00040000;delay_ms(10);if(--no_of_steps_aclk == 0) break;
   IO0CLR = 0X000F0000;IO0SET = 0X00020000;delay_ms(10);if(--no_of_steps_aclk == 0) break;
   IO0CLR = 0X000F0000;IO0SET = 0X00010000;delay_ms(10);if(--no_of_steps_aclk == 0) break;
   }while(1);
   IO0CLR = 0X00FF0000;
   while(1);
}
void delay_ms(unsigned int j)
{
 unsigned int x,i;
 for(i=0;i<j;i++)
  {
    for(x=0; x<10000; x++);
  }
}
```

[Extra Information: Write a program to move the stepper for given angle:

If angle is given,

$$steps = \frac{angle}{step\ angle}$$

ex: Step angle for 200 step motor

$$step\ angle = \frac{360^0}{200} = 1.8^0$$

(No. of rotor teeth x 4 = No. of steps per revolution)

50 X 4 = 200

↑ rotor teeth)

Write a program to move the stepper for given angle

$$steps\ per\ second = \frac{rpm\ x\ steps\ per\ rev}{60}$$

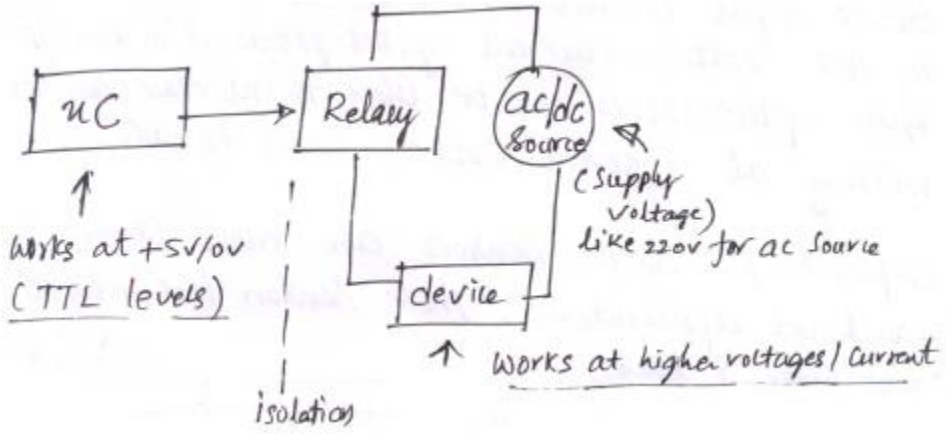
rpm – revolutions per minute

Interfacing High power devices using relays

Microcontroller port pin cannot be directly used to switch ON/OFF (control) high power ac/dc devices like bulb, fan, dc motor, solenoids etc (unlike LED's which can be directly controlled / connected by port pins) as they don't drive required high current or voltages by high power devices.

Relays are used to interface high power devices to microcontrollers. Relays are devices that can turn on or off the power supplied to other device like bulb, like a switch. Relays are used to control high-power circuits by low-power devices like microcontrollers.

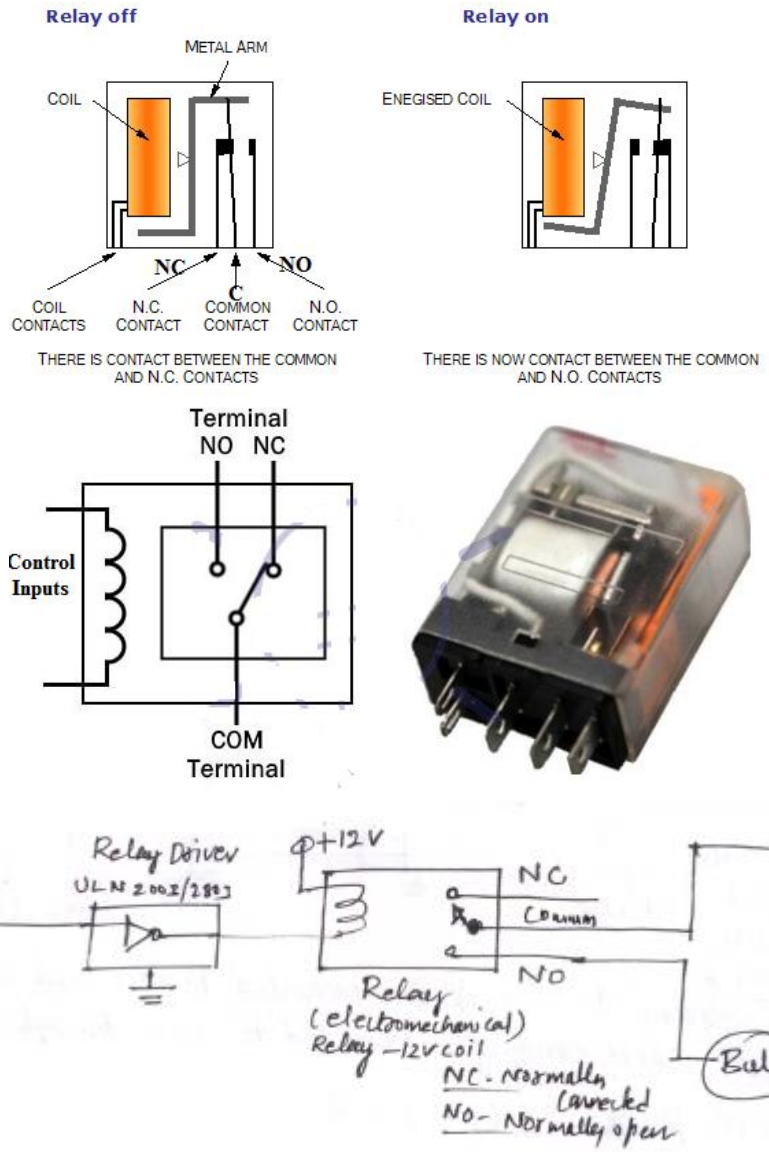
Following diagram shows, the working diagram of relay, high-power device(bulb), high power source (like AC-220V, or DC-24 V etc) and microcontroller. Relays provide isolation between microcontroller and high power source.



Relays are of two types

a) Electromechanical Relay – cheap, less reliable, because of movable ports used in the construction of these relays. Relay driver is required to drive these relays from the microcontroller ports.
b) Solid State Relay – costly, reliable (durable), no moving parts, no relay driver required.

Interfacing: Interface AC device to LPC 2148 using mechanical relay:
Assume P0.0 of microcontroller is used to drive the relay through the relay driver ULN 2803. The following figures shows the working of mechanical relay and the Microcontroller interfacing of AC device-bulb.



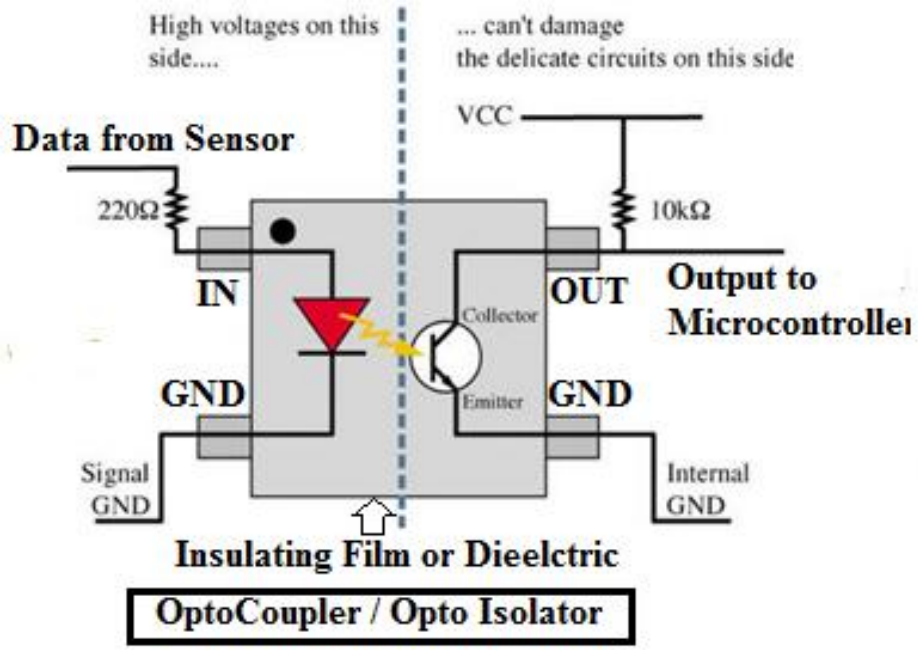
When P0.0 is logic ∅, Coil not energised, do-not conduct and hence no contact between Common and NO, bulb is off.
When P0.0 is logic1, Coil energised, connection establishes between common and NO, and bulb is ON.

Embedded C Program / Driver Program to control Relay

```
#include <lpc214x.h>
int main()
{
      IO0DIR = 1 << 0;
      IO0SET = 1 << 0; // switches on the relay
      while(1);
// ( IO0CLR = 1 << 0; // switches off  the relay )
}
```

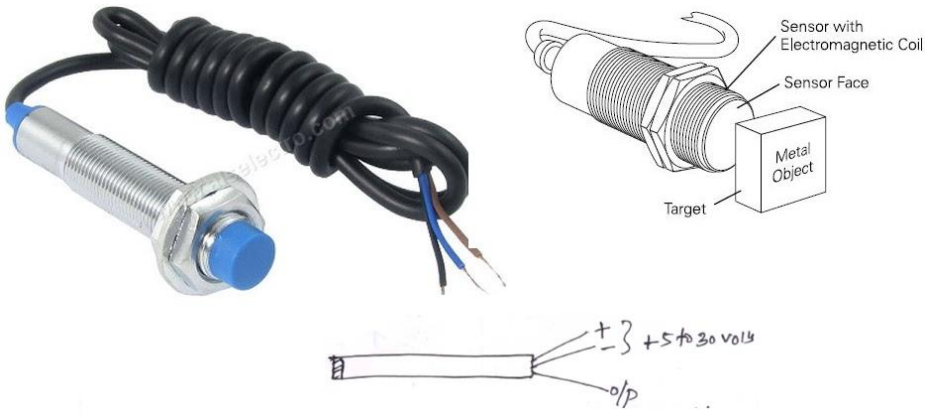
Interfacing Industrial Proximity sensor using Opto-Isolator

Opto-isolator, as shown in the fig, has photodiode and photo transistor,



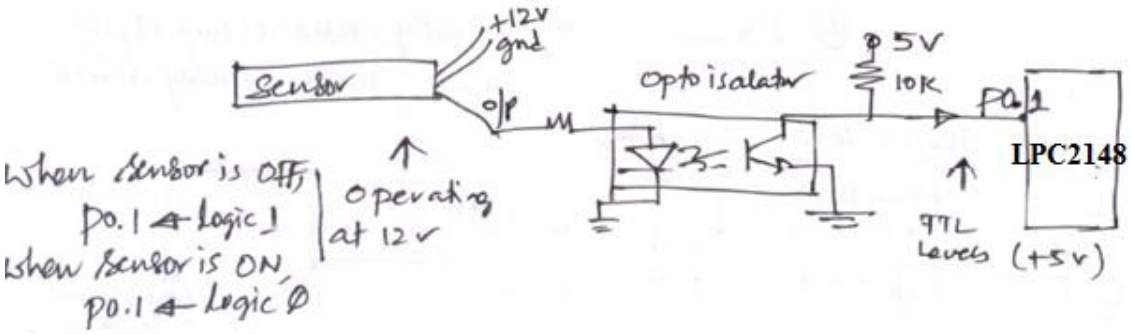
When diode conducts, it emits light and the transistor conducts. It provides complete isolation between diode and transistor, hence, input circuit which drives the diode do not get affected by the voltage/ current spikes produced in the o/p circuit. Also opto-isolators can be sued to interface devices (like sensors) working at different voltage levels to Microcontrollers.

Proximity Sensors (Capacitive / Inductive sensors) are most commonly used in the industrial applications. Used, extensively in robotics, CNC and other industrial automation applications. These sensors has inbuilt electronics and comes with 3 leads/pins, as shown in the figure.



Among the three pins, 2 pins are for power (say, +12V,GND) and one pin/lead is output. When the sensor comes near to the metal object, it detects and generates logic 1 (+12V) / 0(GND) output at the output pin. This change is output voltage levels is used as input to the microcontroller.

Since the sensor is producing +12V or 0V, which can't be connected to LPC 2148 Pins, which operate at 3.3V or 0V. Opto-isolator can be used to connect proximity sensor to LPC 2148 as shown in the figure.



Write a program to read sensor connected to p0.1 and make ac device ON/OFF correspondingly, connected to P0.0 through relay.

```
int main( )
{
        int result=0;
        IO0DIR | = 1U << 0; // configure P0.0 as output, connected to Relay
        while( 1 )
        {
                result = IO0PIN & (1U << 1); //P0.1 connected to sensor1
                if(result)
                        IO0CLR = 1U << 0;
                else
                        IO0SET =  1U << 0;
        }
}
```