Unit V- PIC Interfacing-III

Interfacing of ADC 0808 with PIC

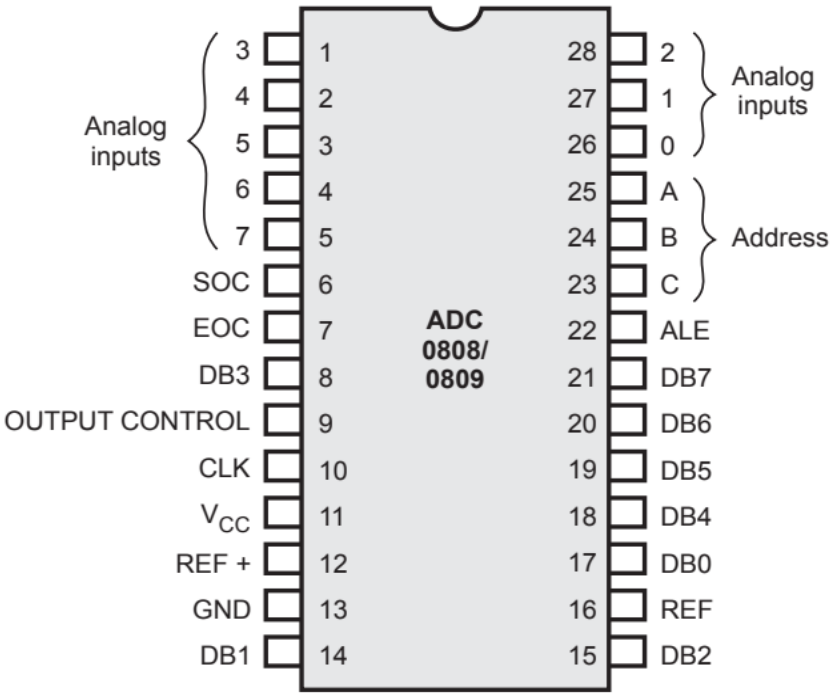
- The ADC 0808 and ADC 0809 are monolithic CMOS devices with an 8-channel multiplexer. These devices are also designed to operate from common microprocessor/ microcontroller control buses, with tri-state output latches driving the data bus. The main features of these devices are :

Features

- 8-bit successive approximation ADC.
- 8-channel multiplexer with address logic.
- Conversion time 100 μs.
- It eliminates the need for external zero and full-scale adjustments.
- Easy to interface to all microprocessors.
- It operates on single 5 V power supply.
- Output meet TTL voltage level specifications.

Pin Diagram

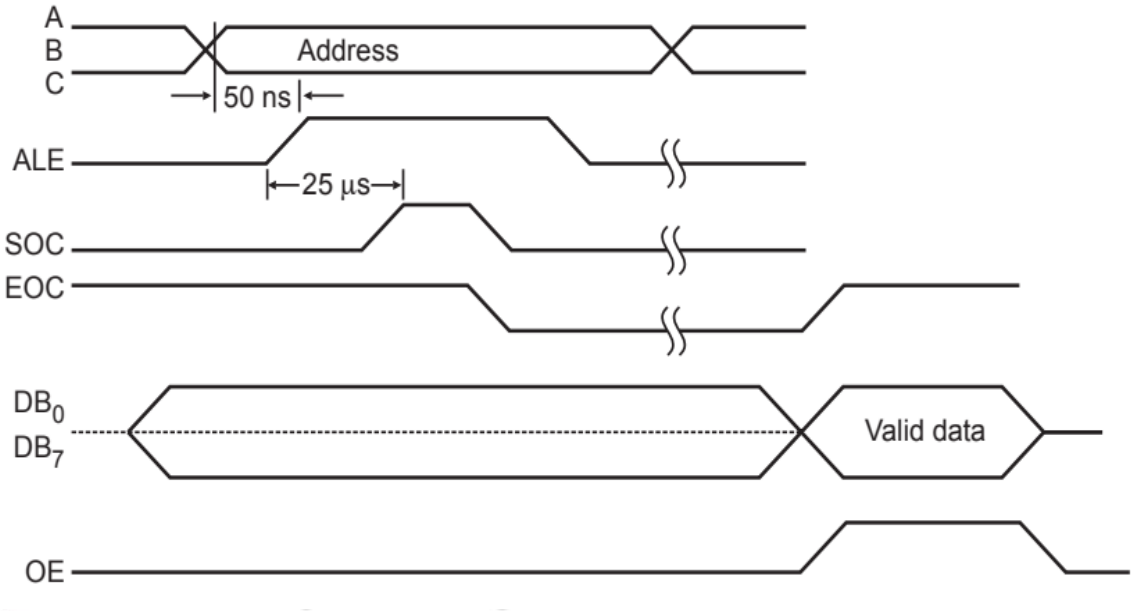
- Fig. shows pin diagram of 0808/0809 ADC.



Pin diagram of ADC 0808/0809

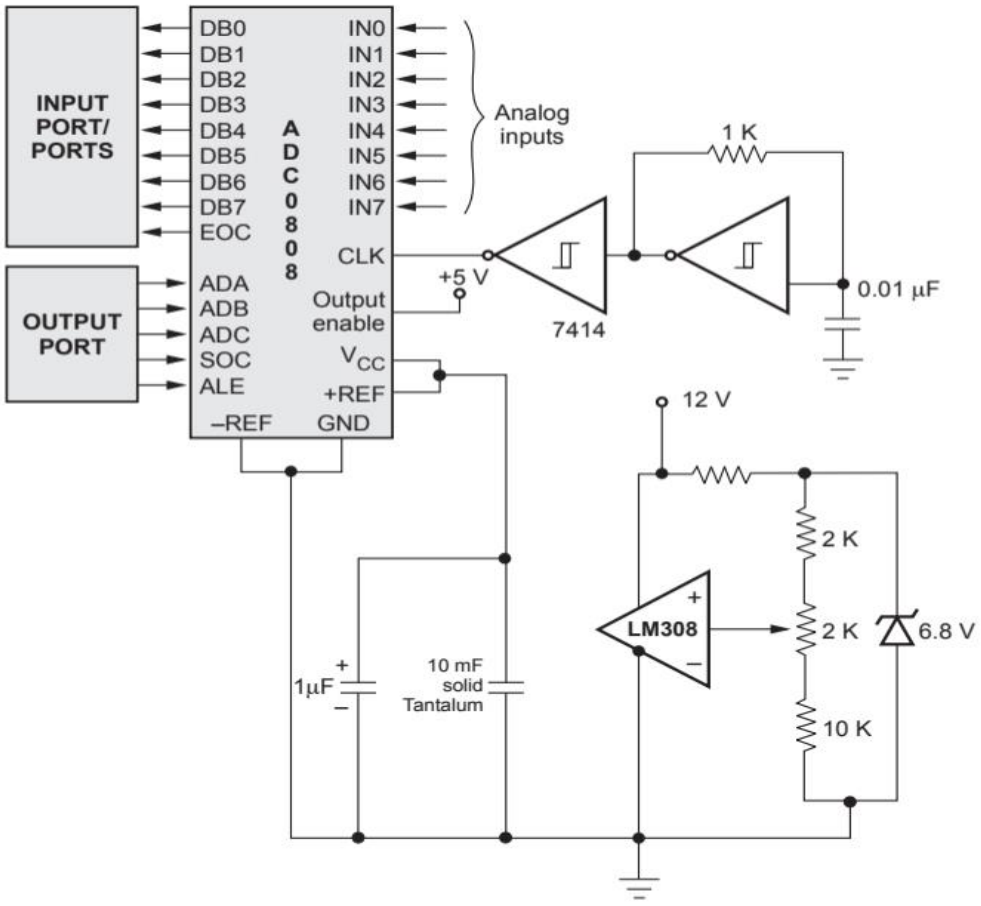
Operation

- ADC 0808/0809 has eight input channels, so to select desired input channel, it is necessary to send 3-bit address on A, B and C inputs. The address of the desired channel is sent to the multiplexer address inputs through port pins. After at least 50 ns, this address must be latched. This can be achieved by sending ALE signal. After another 2.5 μs, the start of conversion (SOC) signal must be sent high and then low to start the conversion process. To indicate end of conversion ADC 0808/0809 activates EOC signal. The microprocessor system can read converted digital word through data bus by enabling the output enable signal after EOC is activated.



Interfacing

- Fig. shows typical interfacing circuit for ADC 0808 with microprocessor/ microcontroller system.



Typical interface for 0808/0809

Analog to Digital Converter (A/D) Module

- In the PIC18 family, has on-chip 10-bit resolution ADC. The number of channels supported by a particular microcontroller varies depending on the family member.
- The PIC18F458 has ADC module with eight channel (AN0-AN7) 10-bit ADC.

ADC Registers

- The A/D module has four registers. These registers are :
  - A/D Result High Register (ADRESH)
  - A/D Result Low Register (ADRESL)
  - A/D Control Register 0 (ADCON0)
  - A/D Control Register 1 (ADCON1)
- The ADCON0 register, shown in Fig. controls the operation of the A/D module. It has conversion clock source selection bits, channel section bits, A/D conversion status bits and A/D on bit.
- The ADCON1 register, shown in Fig. configures the functions of the port pins. It has A/D result format selection bit, A/D clock section bit and A/D port configuration control bits.
- The ADRESH and ADRESL registers contain the result of the A/D conversion.

ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	–	ADON

bit 7 bit 0

bit 7-6 ADCS1 : ADCS0 : A/D Conversion Clock Select bits (ADCON0 bits in bold)

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	$F_{OSC}/2$
0	01	$F_{OSC}/8$
0	10	$F_{OSC}/32$
0	11	F_{RC} (Clock derived from the internal A/D RC oscillator)
1	00	$F_{OSC}/4$
1	01	$F_{OSC}/16$
1	10	$F_{OSC}/64$
1	11	F_{RC} (clock derived from the thermal A/D RC oscillator)

bit 5-3 CHS2:CHS0 : Analog Channel Select bits

000 = Channel 0 (AN0)

001 = Channel 1 (AN1)

010 = Channel 2 (AN2)

011 = Channel 3 (AN3)

100 = Channel 4 (AN4)

101 = Channel 5 (AN5)

110 = Channel 6 (AN6)

111 = Channel 7 (AN7)

bit 2 GO/$\overline{\text{DONE}}$: A/D Conversion Status bit

When ADON = 1 :

1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)

0 = A/D conversion not in progress

bit 1 **Unimplemented :** Read as '0'

bit 0 **ADON :** A/D On bit

1 = A/D converter module is powered up

0 = A/D converter module is shut-off and comsumes no operating current

Steps for Programming ADC

- The following steps should be followed for doing an A/D conversion :
  1. Configure the A/D module :
     - Configure analog pins, voltage reference and digital I/O (ADCON1)
     - Select A/D input channel (ADCON0)
     - Select A/D conversion clock (ADCON0)
     - Turn on A/D module (ADCON0)
  2. Configure A/D interrupt (if desired) :
     - Clear ADIF bit
     - Set ADIE bit
     - Set GIE bit
  3. Wait the required acquisition time.
  4. Start conversion :
     - Set GO/DONE bit (ADCON0)

5. Wait for A/D conversion to complete, by either :
   - Polling for the GO/DONE bit to be cleared
     OR
   - Waiting for the A/D interrupt
6. Read A/D Result registers (ADRESH/ADRESL); clear bit ADIF if required.
7. For next conversion, go to step 3

Write PIC18F C program to read data from channel 0 of ADC and display the result on Port C and Port D after every quarter of second.

Solution :

Assume Clock source = $F_{osc}/64$, ADC result = right justified

```
#include <p18f458.h>
void main(void)
{
    TRISC = 0;                      // Configure Port C as output
    TRISD = 0;                      // Configure Port D as output
    TRISAbits.TRISA0 = 1;           // Configure RA0 pin as input for analog input
    ADCON0 = 0X81;                  // Fosc/64, Channel 0, A/D ON
    ADCON1 = 0XCE;                  // Result right justified, AN0 = Analog input
    while(1)
    {
        Delay(1);                       // Wait for required acquisition time
        ADCON0bits.GO = 1;              // Start conversion
        while(ADCON0bits.DONE == 1);    // Wait for the conversion
        PORTC = ADRESL;                 // Send lower byte of result to Port C
        PORTD = ADRESH;                 // Send higher byte of result to Port D
        Delay(250);                     // Wait for 250 ms
    }
}
void Delay(unsigned int cnt)
{
    unsigned int i, j;
    for(i=0; i<1275; i++)
        for(j=0; j<cnt; j++);
}
```

Programming A/D Converter using Interrupt

- We can use interrupt method for checking the end of conversion instead of polling. Here, we have to enable A/D interrupt by making analog to digital interrupt enable bit, ADIE = 1.
- Upon completion of conversion, analog to digital interrupt flag, ADIF goes HIGH.

Write PIC18F C program to read data from channel 0 of ADC using interrupt and display the result on Port C and Port D after every quarter of second.

Solution :

Assume Clock source = $F_{osc}/64$, ADC result = right justified

```
#include <p18f458.h>
void Delay (unsigned int);
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
      If (INTCONbits.ADIF = 1)          // If ADC caused interrupt execute ADC_ISR
            ADC_ISR();
}
#pragma code HiPrio_Int = 0x08          // High priority interrupt
void HiPrio_Int (void)
{
 _asm
       GOTO CHK_ISR      ; High priority interrupt land at address 0008H. To avoid
                         ; using limited memory space allocated to IVT and to have
                         ; more space for ISR we use GOTO instruction to CALL ISR
                         ; written at different place
 _endasm
}
#pragma code

void main(void)
{
      TRISC = 0;                    // Configure Port C as output
      TRISD = 0;                    // Configure Port D as output
      TRISAbits.TRISA0 = 1;         // Configure RA0 pin as input for analog input
      ADCON0 = 0X81;                // Fosc/64, Channel 0, A/D ON
```

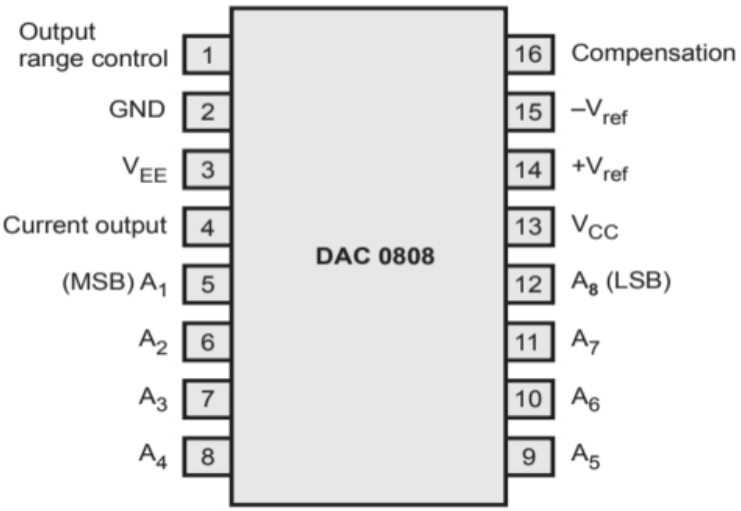
```
        ADCON1 = 0XCE;              // Result right justified, AN0 = Analog input
        PIR1bits.ADIE = 1;          // Enable AD interrupt
        INTCONbits.PEIE = 1;        // Enable peripheral interrupts
        INTCONbits.GIE = 1;         // Enable all interrupts globally
        while(1)
        {
            Delay(1);               // Wait for required acquisition time
            ADCON0bits.GO = 1;      // Start conversion
        }
}
void ADC_ISR (void)
{           PORTC = ADRESL;         // Send lower byte of result to Port C
            PORTD = ADRESH;         // Send higher byte of result to Port D
            Delay(250);             // Wait for 250 ms
            PIR1bits.ADIF = 0;      // Clear ADC interrupt flag
}
void Delay(unsigned int cnt)
{
    unsigned int i, j;
    for(i=0; i<1275; i++)
        for(j=0; j<cnt; j++);
}
```

Interfacing of DAC 0808 with PIC

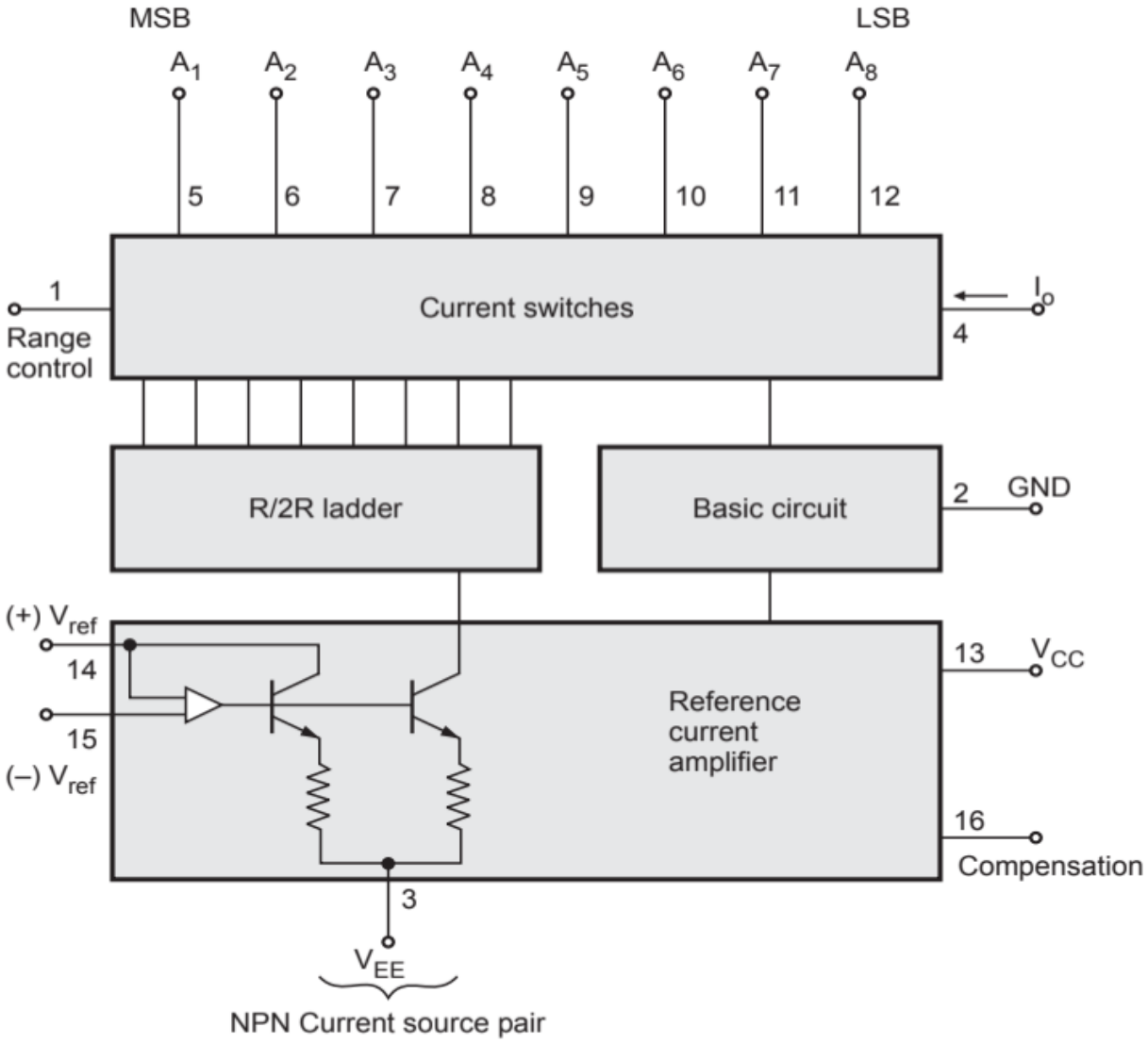
- The DAC 0808 is an 8-bit R/2R ladder type D/A converter compatible with TTL and CMOS logic. It is designed to use where the output current is linear product of an eight-bit digital word.
- Fig. shows the pin diagram and block diagram for DAC 0808. The DAC 0808 consists of a reference current amplifier, an R/2R ladder and eight high speed

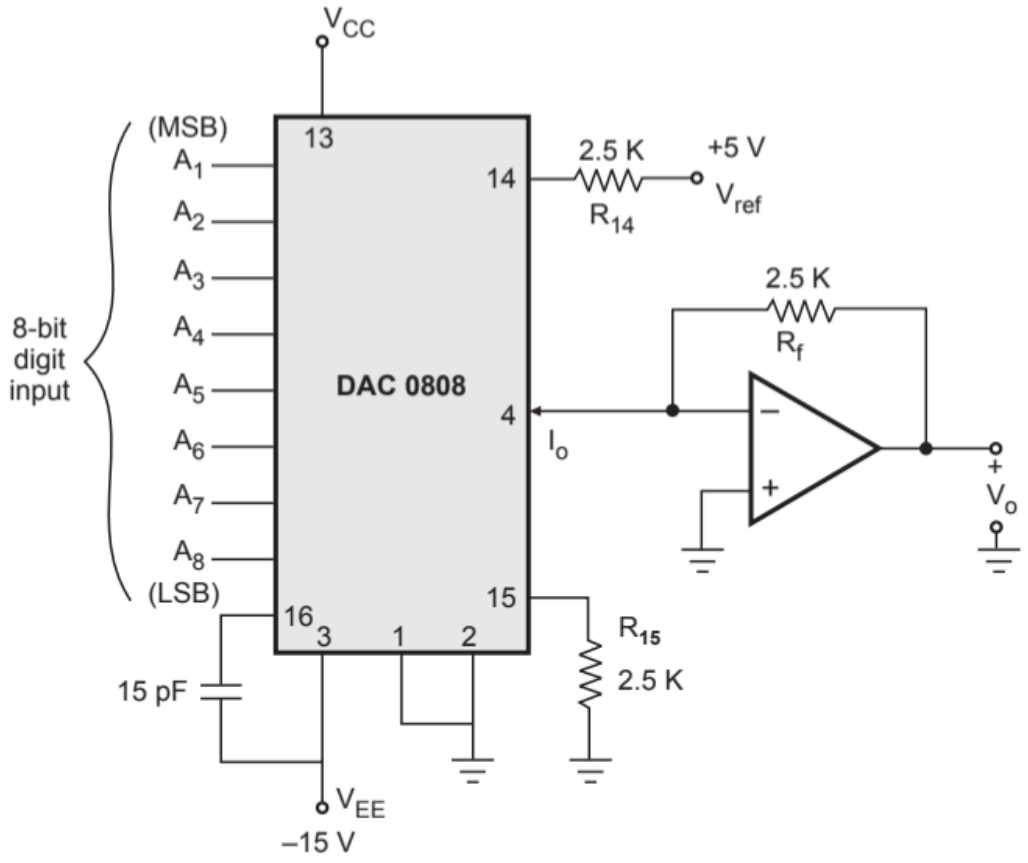
current switches. It has eight input data lines A_1 (MSB) through A_8 (LSB) which control the positions of current switches.

- It requires 2 mA reference current for full scale input and two power supplies $V_{CC} = +5$ V and $V_{EE} = -15$ V (V_{EE} can range from – 5 V to – 15 V).
- The voltage V_{ref} and resistor R_{14} determines the total reference current source and R_{15} is generally equal to R_{14} to match the input impedance of the reference current amplifier.



Pin diagram





The output current I_o can be given a

$$I_o = \frac{V_{ref}}{R_{14}}\left(\frac{A_1}{2}+\frac{A_2}{4}+\frac{A_3}{8}+\frac{A_4}{16}+\frac{A_5}{32}+\frac{A_6}{64}+\frac{A_7}{128}+\frac{A_8}{256}\right)$$

Note : Input A_1 through A_8 can be either 0 or 1. Therefore, for typical circuit full scale current can be given as,

$$I_o = \frac{5}{2.5\,K}\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\frac{1}{256}\right)$$

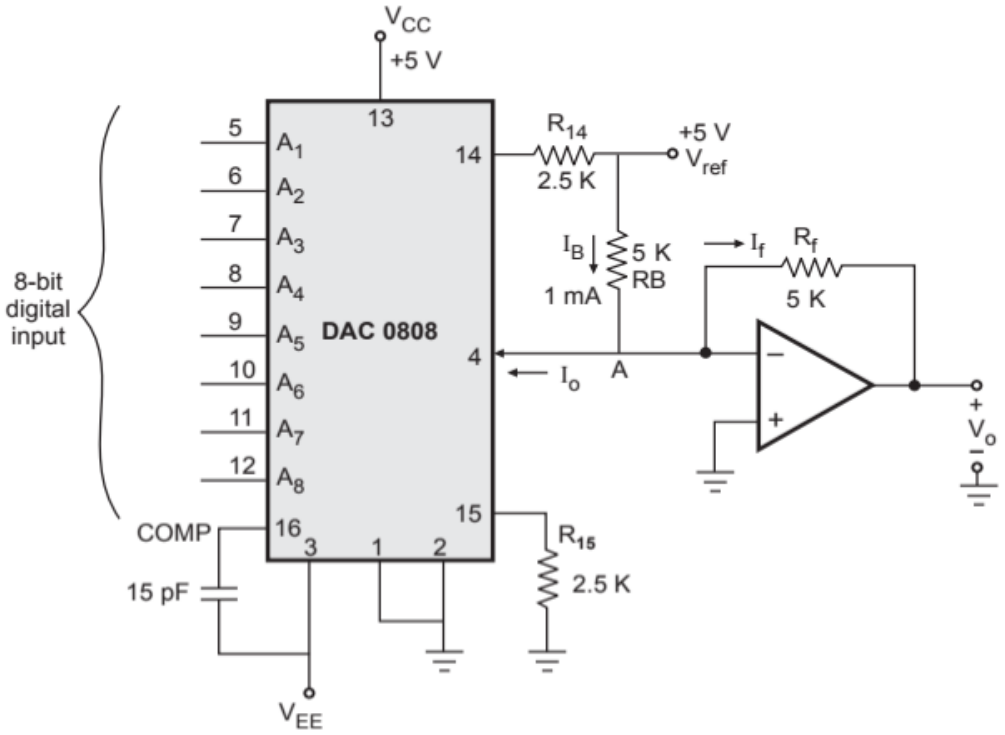
$$= \frac{2\,mA \times 255}{256} = 1.992\ mA$$

It shows that the full scale output current is always 1 LSB less than the reference current source of 2 mA. This output current is converted into voltage by I to V converter. The output voltage for full scale input can be given as,

$$V_o = 1.992 \times 2.5\,K = 4.98\ V$$

The arrow on the pin 4 shows the output current direction. It is inward. This means that DAC 0808 sinks current. At $(0000\,0000)_2$ binary input it sinks zero current and at $(1111\,1111)_2$ binary input it sinks 1.992 mA.

- The circuit shown in the Fig. gives output in the unipolar range. When digital input is 00H, the output voltage is 0 V and when digital input is FFH $(1111\,1111)_2$, the output voltage is + 5 V. This circuit can be modified to give bipolar output.
- Fig. shows the circuit for giving output in the bipolar range. Here, resistor R_B (5 K) is connected between V_{ref} and the output terminal of DAC 0808. This gives a constant current source of 1 mA.



Interfacing DAC in the bipolar range

The circuit operation can be observed for three conditions :

Condition 1 : For binary input (00H)

- When binary input is 00H, the output current I_o at pin 4 is zero. Due to this current flowing through R_B (1 mA) flows through R_f giving $V_o = -5$ V.

Condition 2 : For binary input 80H

When binary input is 80H, the output current I_o at pin 4 is 1 mA. By applying KCL at node A we get,

$$-I_B + I_o + I_f = 0$$

Substituting values of I_B and I_o we get,

$$-(1\text{ mA}) + (1\text{ mA}) + I_f = 0$$

$$\therefore \quad I_f = 0$$

and therefore the output voltage is zero.

Condition 3 : For binary input FFH

When binary input is FFH, the output current I_o at pin 4 is 2 mA. By applying KCL at node A we get,

$$-I_B + I_o + I_f = 0$$

Substituting values of I_B and I_o we get,

$$-(1\text{ mA}) + (2\text{ mA}) + I_f = 0$$

$$\therefore \quad I_f = -1\text{ mA}$$

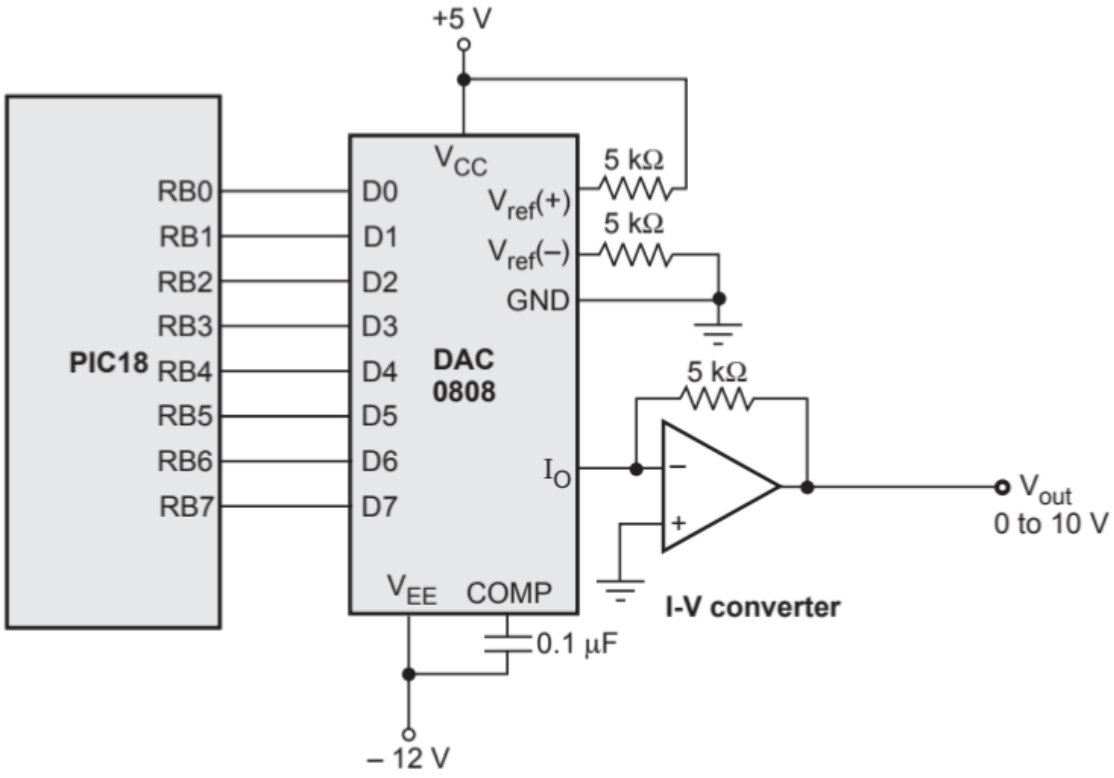
Therefore, the output voltage is + 5 V. In this way, circuit shown in the Fig. gives output in the bipolar range.

Important Electrical Characteristics for IC DAC 0808

- The important electrical characteristics for IC DAC0808 are :
  - Reference current : 2 mA
  - Supply voltage : + 5 V V_{CC} and - 15 V V_{EE}
  - Setting time : 300 ns
  - Full scale output current : 1.992 mA
  - Accuracy : 0.19 %
- Fig. shows the interfacing of DAC 0808 with PIC18 microcontroller.
- We now see how different waveforms can be generated using this circuit.

Square Wave

- To generate a square wave, first we have to output FF and then 00 on Port B of PIC18. Port B is connected as an input to the DAC 0808. According to frequency requirement, delay is provided between the two outputs.




```
#include <p18f458.h>
void main(void)
{
      TRISB = 0;      // Configure Port B as output
      while(1)
      {
            PORTB = 0;          // Output 00 on Port B
            Delay(50);          // wait for 50 ms
            PORTB = 0xFF;       // Output FF on Port B
            Delay(50);          // wait for 50 ms
      }
}
void Delay(unsigned int cnt)
{
      unsigned int i, j;
      for(i=0; i<1275; i++)
            for(j=0; j<cnt; j++);
}
```

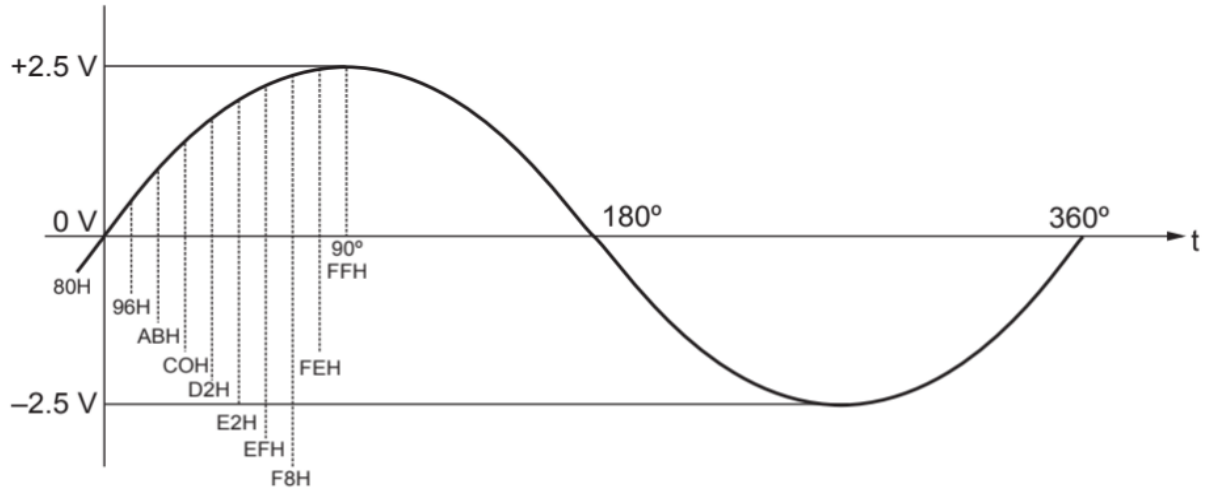
Triangular Wave

- To generate a triangular wave, we have to output data from 00 initially, and it should be incremented up to FFH (Decimal 255).
- When it reaches FFH (Decimal 256) it should be decremented up to 00.

```
#include <p18f458.h>
void main(void)
{
    Unsigned char i;
    TRISB = 0;      // Configure Port B as output
    while(1)
    {
        for(i=0; i<=255; i++)
                PORTB = i;
        for(i=255; i>=0; i--)
                PORTB = i;
    }
}
```

Sine wave

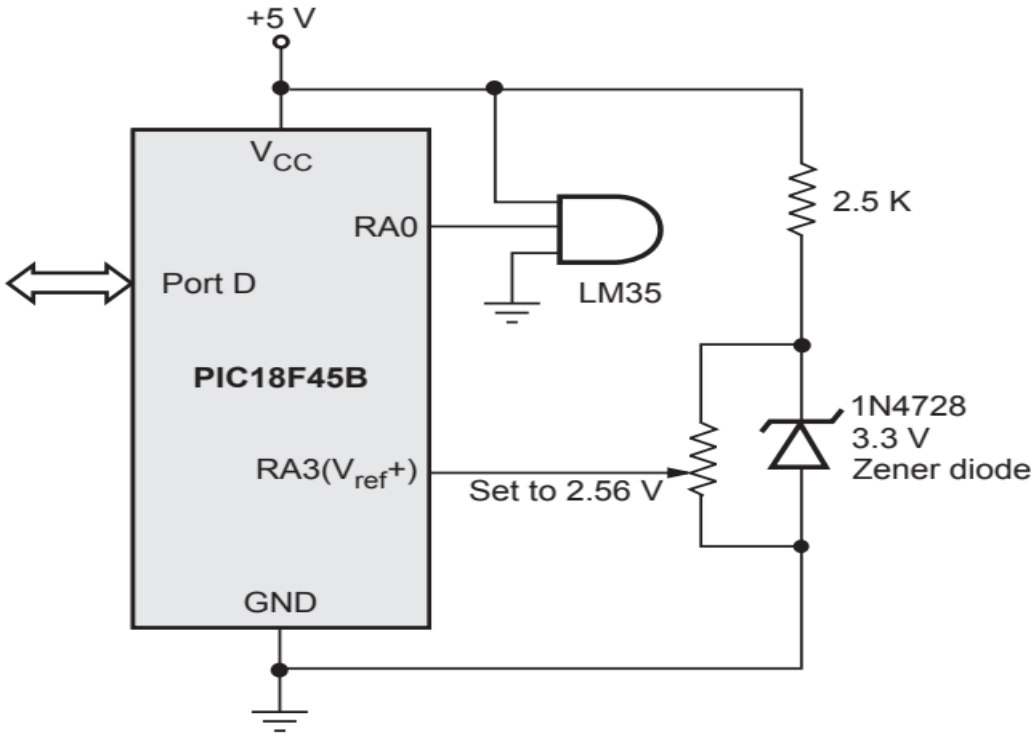
- To generate sine wave, we have to output digital equivalent values representing sine wave, as shown in Fig. Digital data 00H represents 0 V. 7FH represents 5 V and FFH means + 10 V. The digital equivalent for sine wave can be calculated as follows.



- We know that sin 0° = 0 and sin 90° = 1. The range sin 0° to sin 90° is distributed over digital range of 7FH to FFH i.e. (FFH – 7FH) 128 decimal steps. Therefore, taking 128 as an offset, we can write, Digital equivalent value (DEV) for $\sin\theta = (128 + 128 \times \sin\theta)$

Temperature Sensor Interfacing using ADC

- LM35 is a temperature sensor that can measure temperature in the range of – 55°C to 150 °C.
- It is a 3-terminal device that provides an analog voltage proportional to the temperature. The higher the temperature, the higher is the output voltage.



Sensor interface with PIC18F458

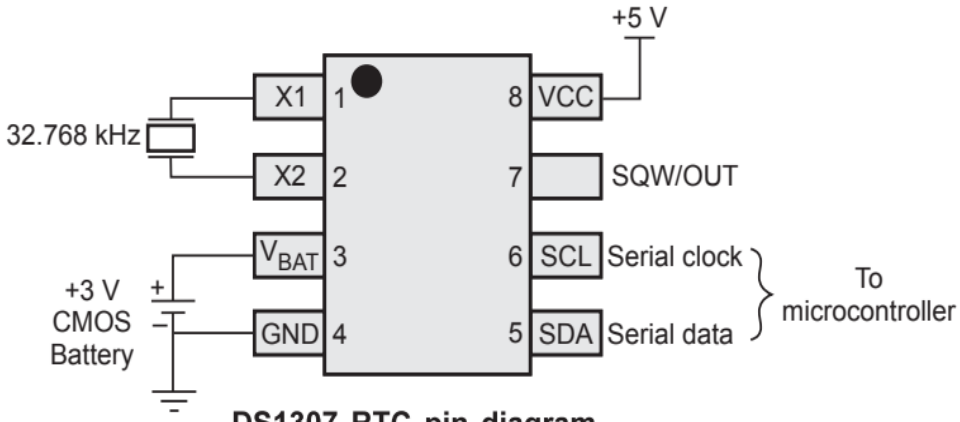
- The output analog voltage can be converted to digital form using ADC so that a microcontroller can process it.
- Fig. shows the circuit connection for temperature sensor LM35 with PIC18 microcontroller.
- The A/D has 10-bit resolution with 1024 steps. The LM35 output voltage increases by 10 mV for each °C increase in its temperature. Now if we use step size of 10 mV the output voltage will be 10.24 V for full scale output. This output voltage is not acceptable and hence we have to reduce the step size. If we use step size of 2.5 mV, the output voltage will be 1024 × 2.5 mV = 2.56 V for full scale output. This output voltage is acceptable and hence we have to set V_{ref} = 2.56 V to set step size = 2.5 mV. Due to this adjustment, the binary output number for the A/D is 4 times the real temperature (10 mV/4 mV = 4). We can scale down the binary output by dividing it by 4 to get real number for temperature.
- IN4728 zener diode is used to fixed the voltage across the 10 K pot at 2.56 V.

Interfacing of RTC with PIC18 using I^2C

- Real Time Clock (RTC) is used to track the current time and date. It is generally used in computers, laptops, mobiles, embedded system applications devices etc.
- In many embedded system, we need to put time stamp while logging data i.e. sensor values, GPS coordinates etc. For getting timestamp, we need to use RTC (Real Time Clock).
- Some microcontrollers like LPC2148, LPC1768 etc., have on-chip RTC. But in other microcontrollers like PIC, ATmega16/32, they do not have on-chip RTC. So, we should use external RTC chip.
- There are different types of ICs used for RTC like DS1307, DS12C887 etc. In this section we will see DS1307.

Features of DS1307

- The features of DS1307 RTC are :
  1. It counts seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation valid up to 2100.
  2. It has 56-byte, battery-backed, nonvolatile (NV) RAM for data storage
  3. It supports I^2C, two-wire serial interface.
  4. It has a programmable square wave output signal
  5. It has automatic power-fail detect and switch circuitry
  6. It consumes less than 500 nA in battery backup mode with oscillator running
  7. It has wide operating temperature range: – 40 °C to + 85 °C
  8. It is available in 8-pin DIP or SOIC



DS1307 RTC pin diagram

- The RTC DS1307 uses external crystal of frequency 32.768 kHz, so we need to connect crystal with 32.768 kHz to X1 and X2 pin.

- Connect 3 volt CMOS battery to V_{BAT} pin. RTC DS1307 has inbuilt mechanism to detect 5 volt VCC, if external 5 volt VCC is not there, then it takes the supply from 3 volt CMOS battery.
- The SDA (Serial Data) and SCL (Serial Clock) pins are I^2C serial communication pins which are used to connect with microcontroller's I^2C pins.
- SQW/OUT pin is square wave output driver. The SQW/OUT pin outputs one of four square-wave frequencies 1 Hz, 4 kHz, 8 kHz, 32 kHz by setting internal register bits.

Timekeeper Registers

- Generally, while using RTC (Real Time Clock) first time, we need to set current time and date in RTC. Then RTC keeps updating these values in seconds. In RTC DS1307, we can set this time and date in the **Timekeeper Register.** After setting time and date value, RTC DS1307 keeps updating it in seconds so we will get updated time later.
- The content of Timekeeper registers is in BCD (Binary Coded Decimal value) format.
- There are total eight registers in timekeeper register for setting seconds, minutes, hours, day, date, month, year and control.
- Once we set the value of these registers, they will keep updating themselves, and we can read these registers to get updated values.

ADDRESS	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	FUNCTION	RANGE
00H	CH	10 Seconds			Seconds				Seconds	00 - 59
01H	0	10 Minutes			Minutes				Minutes	00 - 59
02H	0	12 24	10 Hour PM/AM	10 Hour	Hours				Hours	1 - 12 + AM/PM 00 - 23
03H	0	0	0	0	0	DAY			Day	01 - 07
04H	0	0	10 Date		Date				Date	01 - 37
05H	0	0	0	10 Month	Month				Month	01 - 12
06H	10 Year				Year				Year	00 - 99
07H	OUT	0	0	SQWE	0	0	RS1	RS0	Control	–
08H - 3FH									RAM 56 × 8	00H - FFH

Address - 00H : 02H : Clock Registers

Address - 00H

- In this register bit - 7 is CH bit, which is crystal oscillator enable / disable bit, when it is zero, the crystal oscillator is enabled otherwise oscillator is not enabled, so we always make this bit zero while using RTC.
- Other bits are used for read / write the second. As timekeeper register stores the value in BCD format, here Bit - 4 to Bit - 6 stores the upper BCD digit of the seconds (value from 0 to 5), and Bit - 0 to Bit - 3 stores lower BCD digit of the seconds (value from 0 to 9). As seconds' value starts from 00 and ends at 59.

Address - 01H

- This address is used to read / write minutes' value.
- Upper BCD digit of minutes is stored in Bit - 4 to Bit - 6 and lower BCD digit is stored in Bit - 0 to Bit - 3

Address - 02H

- This address is used to read / write Hour.
- Clock can run in either 12 Hr or 24 Hr format.
- **12-hour format :** To set 12-hour clock format, we need to set Bit - 6 to logic 1. In 12 - hour clock format Bit - 5 will indicate AM / PM, Logic 1 is for PM and Logic 0 is for AM. Bit - 4 is indicated as 10 Hour, which is to store higher digit of hour value, which is 0 or 1 in case of 12-hour system. Bit-0 to Bit- 3 stores the value of lower digit of hour (value from 0 to 9).
- **24-hour format :** To set 24-hour clock format, we need to reset Bit - 6 to logic 0. Bit - 4 and Bit - 5 are indicated as 10 Hour, which is to store higher digit of hour value, which is 0 to 2 in case of 24-hour system. Bit - 0 to Bit - 3 stores the value of lower digit of hour (value from 0 to 9).

Address - 03H : 06H : Calendar Register

Address - 03H

- This address is used to read /write day value from 1 to 7. Bit - 0 to Bit - 2 are used to read /write day value.

Address - 04H

- This address is used to read / write the date value. Bit - 4 and Bit - 5 are used to read / write upper digit value of date (value from 0 to 3). Bit - 0 to Bit - 3 are used for lower digit of date value (value from 0 to 9).

Address - 05H

- This address is used to read / write the month. Bit - 4 used for upper digit value of month that is 0 or 1. And Bit - 0 to Bit - 3 are used to store the lower digit value of the month (value from 0 to 9).

Address-06H

- This address isused to read / write the year value. It provides only last two digits of year value. Bit - 0 to Bit - 3 stores lower digit, and Bit - 4 to Bit - 7 stores higher digit of the year.

Address - 07H : Control Register

Bit 7 - OUT : Output

7	6	5	4	3	2	1	0
OUT	0	0	SQWE	0	0	RS1	RS0

- This controls the output level of pin SQW/OUT. When square wave output is disabled, SQWE bit is zero. So, logic level on SQW/OUT is high when this OUT bit is high and zero when this OUT bit is zero.

Bit 4 - SQWE

To Enable / Disable square wave output on SQW / OUT pin

1 = Enable oscillator output

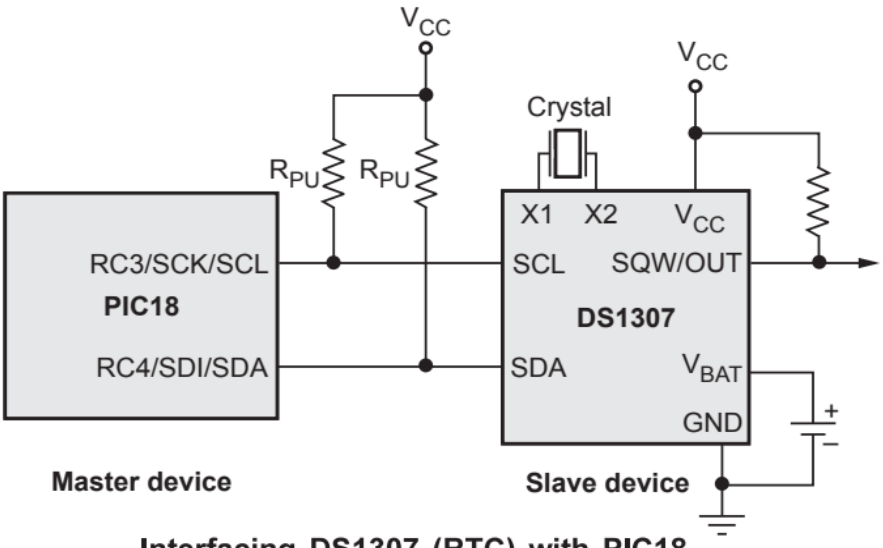
0 = Disable oscillator output

The frequency of square wave is depending on RS0 and RS1 bit.

Bit 0 : 1 - RS0 & RS1

RS1	RS0	Square-Wave Output Frequency
0	0	1 Hz
0	1	4.096 kHz
1	0	8.192 kHz
1	1	32.768 kHz

- The DS1307 supports the **I^2C protocol.** A device that sends data onto the bus is defined as a **transmitter** and a device receiving data as a **receiver.** The device that controls the message is called a **master.** The devices that are controlled by the master are referred to as **slaves.**



Interfacing DS1307 (RTC) with PIC18

- The bus must be controlled by a master device that generates the serial clock (SCL), controls the bus access, and generates the START and STOP conditions. The DS1307 operates as a slave on the I^2C bus.

Data Transfer on The I^2C Bus

- Data transfer can be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is HIGH.
- Changes in the data line while the clock line is high will be interpreted as control signals.
- Accordingly, the following bus conditions have been defined :
  - **Bus not busy :** Both data and clock lines remain HIGH.
  - **START data transfer :** A change in the state of the data line, from HIGH to LOW, while the clock is HIGH, defines a START condition.
  - **STOP data transfer :** A change in the state of the data line, from LOW to HIGH, while the clock line is HIGH, defines the STOP condition.
  - **Data valid :** The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal.
  - **Acknowledge :** Each receiving device, when addressed, is obliged to generate an acknowledge after the reception of each byte. The master device must generate an extra clock pulse which is associated with this acknowledge bit.

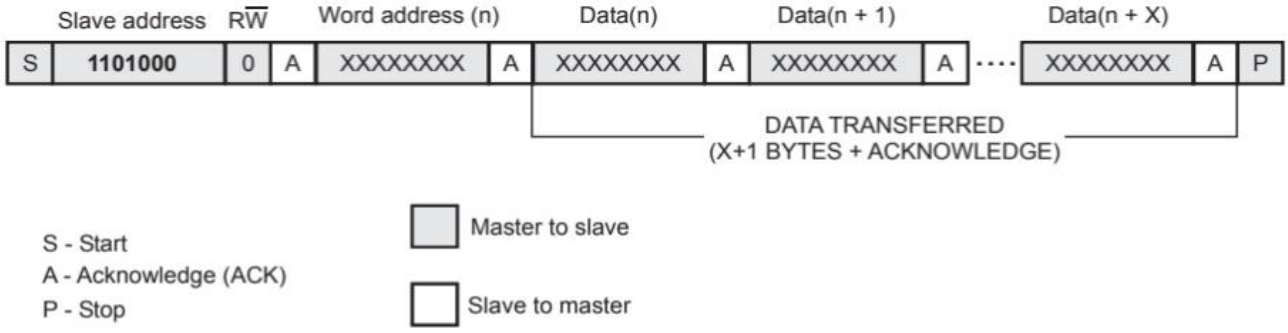
- There is one clock pulse per bit of data. Each data transfer is initiated with a START condition and terminated with a STOP condition.
- The number of data bytes transferred between START and STOP conditions is not limited, and is determined by the master device.
- The information is transferred byte-wise and each receiver acknowledges with a ninth bit.
- Within the I^2C bus specifications a standard mode (100 kHz clock rate) and a fast mode (400 kHz clock rate) are defined. The **DS1307 operates in the standard mode (100 kHz) only.**

Operating Modes of DS1307

- The DS1307 can operate in the following two modes :
  1. Slave Receiver Mode (Write Mode)
  2. Slave Transmitter Mode (Read Mode)

Slave Receiver Mode (Write Mode)

- Serial data and clock are received through SDA and SCL.
- After each byte is received an acknowledge bit is transmitted.
- START and STOP conditions are recognized as the beginning and end of a serial transfer.
- Hardware performs address recognition after reception of the slave address and direction bit as shown in Fig. 9.5.3.



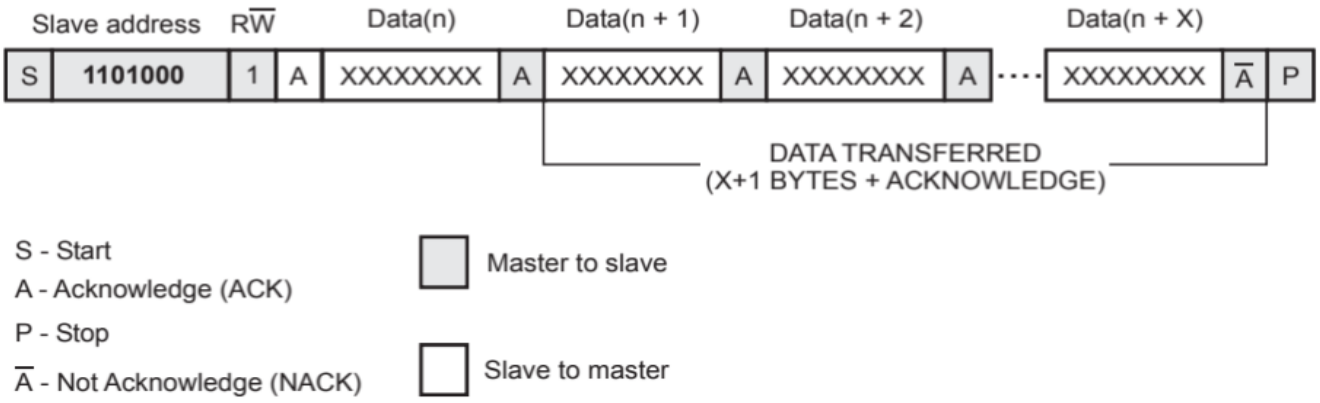
Data write-slave receiver mode

- The slave address byte is the first byte received after the master generates the START condition. The slave address byte contains the 7-bit DS1307 address, which is 1101000, followed by the direction bit ($R/\overline{W}$), which for a write is 0.
- After receiving and decoding the slave address byte, the DS1307 outputs an acknowledge on SDA.

- After the DS1307 acknowledges the **slave address + write bit** (110 1000 + 0 = 1101 1000 = D0H), the master transmits a word address to the DS1307. This sets the register pointer on the DS1307, with the DS1307 acknowledging the transfer.
- The master can then transmit zero or more bytes of data with the DS1307 acknowledging each byte received. The register pointer automatically increments after each data byte are written.
- The master will generate a STOP condition to terminate the data write.

2. Slave Transmitter Mode (Read Mode)

- The first byte is received and handled as in the slave receiver mode. However, in this mode, the direction bit will indicate that the transfer direction is reversed.
- The DS1307 transmits serial data on SDA while the serial clock is input on SCL.
- START and STOP conditions are recognized as the beginning and end of a serial transfer as shown in Fig.



Data read-slave transmitter mode

- The slave address byte is the first byte received after the START condition is generated by the master. The slave address byte contains the 7 - bit DS1307 address, which is 1101000, followed by the direction bit ($R/\overline{W}$), which is 1 for a read.
- After receiving and decoding the slave address the DS1307 outputs an acknowledge on SDA.
- The DS1307 then begins to transmit data starting with the register address pointed to by the register pointer. If the register pointer is not written to before the initiation of a read mode the first address that is read is the last one stored in the register pointer.
- The register pointer automatically increments after each byte are read. The DS1307 must receive a Not Acknowledge to end a read.

Programming Steps for RTC DS1307

- Initially, while using RTC first time, we have to set the clock and calendar values, then RTC always keeps updating this clock and calendar values.
- We will set the RTC clock and calendar values in 1st step and in 2nd step, we will read these values.

Step 1 : Setting Clock and Calendar to RTC DS1307

1. In RTC coding, we require the first RTC device address (slave address) through which the microcontroller wants to communicate with the DS1307.
2. DS1307 RTC device address is 0xD0 (Address : 110 1000 + Direction bit : 0).
3. Initialize I^2C in PIC18F458.
4. Start I^2C communication with device writes address i.e. 0xD0.
5. Then, Send the Register address of Seconds which is 0x00, then send the value of seconds to write in RTC. RTC address gets auto-incremented so next, we only have to send the values of minutes, hours, day, date, month, and year.
6. And stop the I^2C communication.

Step 2 : Reading Time and Date value from RTC DS1307

1. In the second step, we will read the data from the RTC, i.e. second, minute, hours, etc.
2. Start the I^2C communication with device writes address i.e. 0xD0.
3. Then write the register value from where we have to read the data (we read from location 00 i.e. read the second).
4. Then send repeated start with device read address i.e. 0xD1 (Address : 110 1000 + Direction bit : 1).
5. Now read the data with acknowledgment from location 00.
6. For reading the last location always read with the negative acknowledgment, then the device will understand this is the last data to read from the device.
7. For read next byte, the location of the register address will get auto-incremented.

I^2C Registers

- To establish I^2C communication between devices, we have to configure PIC18F458. To do this, PIC18F458 has different registers which are as follows :

SSPBUF : Serial Transmit/Receive Buffer

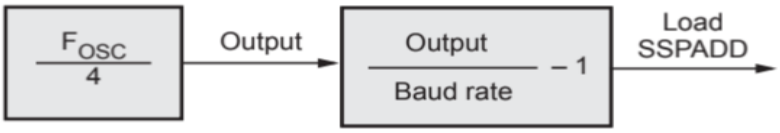
- In I^2C protocol, SSPBUF is a data register which is used for transmission and reception of data.

SSPSR : MSSP Shift Register

- It is a shift register that is used to shift data in and out on the SDA line. It is not directly accessible.

SSPADD Register : MSSP Address Register

- **In Master mode :** SSPADD register is used to decide or generate a baud/bit rate. Now how to calculate the value for baud/bit rate.

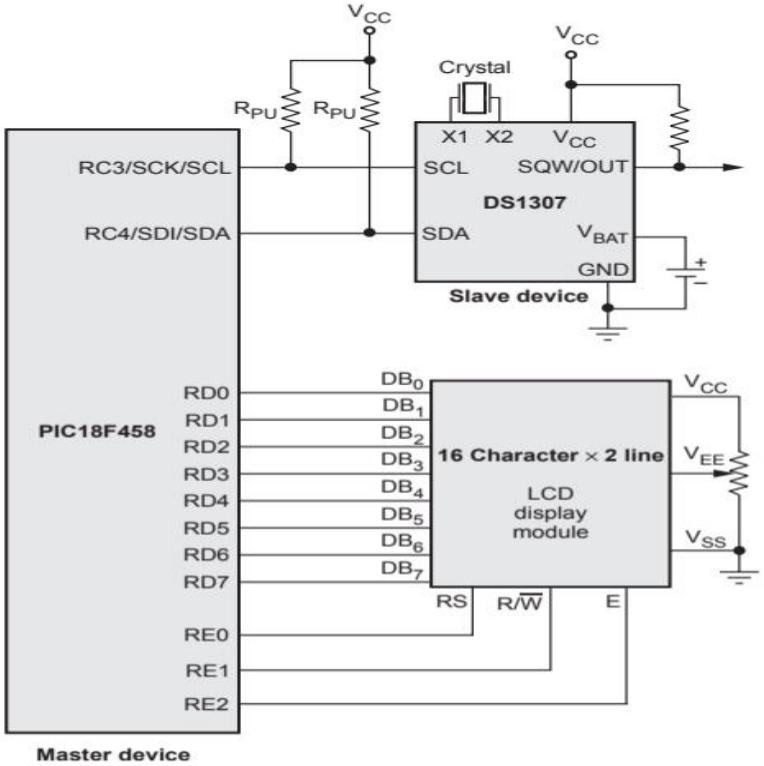


I^2C Baud generation

- Load the generated value in the SSPADD register.
- The above formula is used to decide I^2C clock frequency.

Interface DS1307 RTC chip with PIC18F458 using I^2C and display date and time on LCD.

Solution : Fig. shows the interfacing of DS1307 RTC and LCD with PIC18F458 microcontroller to display date and time on LCD.



```
#include<P18F458.h>
#define LCDData PORTD            // LCD data pins
#define RS PORTBbits.RE0         // RS pin of LCD
#define RW PORTBbits.RE1         // RW pin of LCD
#define EN PORTBbits.RE2         // EN pin of LCD
void LCD_Init();
void LCD(unsigned char, unsigned char);
void LCD_str (unsigned char *str)
void I2C_Init();
char I2C_Start(char slave_write_address)
void I2C_Ready()
void I2C_stop();
void I2C_Ack()
void I2C_Nack()
char I2C_send(unsigned char data)
unsigned char I2C_read(void);
unsigned char RTC1307_read(unsigned char address);
unsigned char BCD2Upperch(unsigned char bcd)
unsigned char BCD2Lowerch(unsigned char bcd)
void DELAY (unsigned int);
unsigned char sec, min, hour, date, month, year;

void I2C_Init()
{
   TRISCbits.TRISC3 = 1;          // SCL direction input
   TRISCbits.TRISC4 = 1;          // SDA direction input
   SSPCON1 = 0x28;                // Enable serial mode and select I2C master
                                     mode : SSPM3:SSPM0 = 1000
   SSPSTAT = 0x80;                // Slew rate disabled, other bits are cleared
   SSPADD = 0x18;                 // For Fosc 10 MHz, Required Bit Rate = 100 kHz
                                  // SSPADD = (10 MHz/(4 * 100 kHz) - 1 = 24 = 0x18
     PIR1bits.SSPIE = 1;          //  Enable SSPIF interrupt
  PIR1bits.SSPIF = 0;             // Clear SSP interrupt flag
   SSPCON2 = 0;                   // Clear to reset state
 }
```

```
char I2C_Start(char slave_write_address)
{
   SSPCON2bits.SEN=1;        //  Send start pulse to the slave device
   while(SSPCON2bits.SEN);  // Wait for completion of start pulse
   SSPIF=0;
   if(!SSPSTATbits.S)            // Check whether START start bit is detected or not.
          return 0;              // if not Return 0 to indicate start failed
   return (I2C_send(slave_write_address));   // Write slave device address with write to
                                             // communicate
}

void I2C_Stop()
{
   I2C_Ready();
   SSPCON2bits.PEN = 1;           // Stop communication
   while(SSPCON2bits.PEN);        // Wait for end of stop pulse
}
```

```
void I2C_Stop()
{
   I2C_Ready();
   SSPCON2bits.PEN = 1;          // Stop communication
   while(SSPCON2bits.PEN);       // Wait for end of stop pulse
}
void I2C_Ack()
{
   SSPCON2bits.ACKDT=0;          // Acknowledge data 1:NACK, 0:ACK
   SSPCON2bits.ACKEN=1;          // Enable ACK to send
   while(SSPCON2bits.ACKEN);
 }
void I2C_Nack()
{
   SSPCON2bits.ACKDT=1;          // Acknowledge data 1:NACK, 0:ACK
   SSPCON2bits.ACKEN=1;          // Enable ACK to send
   while(SSPCON2bits.ACKEN);
}
void I2C_Ready()
{
   while(!SSPIF);                // Wait for operation complete
   SSPIF=0;                      // Clear SSPIF interrupt flag
}
char I2C_send(unsigned char data)
{
      SSPBUF=data;                 // Write data to SSPBUF
      I2C_Ready();
      if ( SSPCON2bits.ACKSTAT)   // Check for acknowledge bit
            return 1;
      else
            return 2;
  }
  char I2C_Read(char flag)
  {
```

```
        unsigned char buffer;
        SSPCON2bits.RCEN=1;         // Enable receive
           while(!SSPSTATbits.BF);  // Wait for buffer full flag which wiil be set when byte is
                                    // received
        buffer=SSPBUF;              // Copy SSPBUF to buffer
        if(flag==0)
           I2C_Ack();               // Send acknowledgment to continue reading
        else
           I2C_Nack();              // Send negative acknowledgment after read to stop
                                    // reading
        I2C_Ready();
        return(buffer);
 }

LCD_Init()
{               LCD(0x38, 0);       // Initialize LCD 2 lines, 5 x 7 matrix
                DELAY (250);
                LCD (0x0E, 0);      // Display On, cursor On
                DELAY (15);
                LCD (0x01, 0);      // Clear LCD
                DELAY (15);
                LCD (0x06, 0);      // Shift cursor right
                DELAY (15);
}
void LCD (unsigned char value, unsigned char flag)
{
     LCDdata = value;               // put the value on the pins
     RS = flag;                     // RS = 0 for LCD command and RS =1 for LCD data

        RW = 0;
        EN = 1;                         // strobe the enable pin
        DELAY(1);
        EN = 0;                         // Make high to low Pulse to latch data
  }
  void LCD_str (unsigned char *str)
  {
        while (*str)
        LCD (*str++, 1);
  }
```

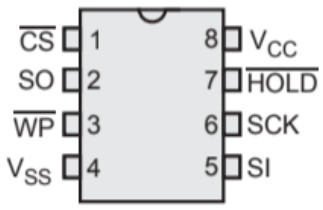
```
void DELAY(unsigned int cnt)
{
	unsigned int i, j;
	for(i=0;  i<cnt;  i++)
		for(j=0; j<165; j++);
}
unsigned char RTC1307_read(unsigned char address)
{
   unsigned char temp;
   I2C_start();
   I2C_send(0xD0);
   I2C_send(address);
   I2C_restart();
   I2C_send(0xD1);
   temp = I2C_read();
   return temp;
}
unsigned char BCD2Upperch(unsigned char bcd)
{
   unsigned char temp;
   temp = bcd >> 4;
   temp = temp | 0x30;
   return (temp);
}
unsigned char BCD2Lowerch(unsigned char bcd)
{

   unsigned char temp;
   temp = bcd & 0x0F;
   temp = temp | 0x30;
   return (temp);
}
```

```
void main()
{
   TRISD = 0;                 // Configure Port D as output
   LCD _Init();               // Initialize LCD
   I2C_Init();                // Initialize I2C Protocol
   I2C_start();               // Start the I2C Protocol
   I2C_send(0xD0);            // Slave Address with Write operation
   I2C_send(0x00);            // Registor Address to be pointed i.e. seconds
   I2C_send(0x80);            // Write to seconds register disabling  CH bit, i.e Stop
                              // Oscilator
   I2C_send(0x00);            // Reset Minutes register to zero
   I2C_send(0x11);            // Hour
   I2C_send(0x01);            // Sunday
   I2C_send(0x11);            //11 Date
   I2C_send(0x07);             //7 July
   I2C_send(0x15);            //2015
   I2C_stop();                //Stop the I2C protocol

   I2C_start();
   I2C_send(0xD0);
   I2C_send(0x00);
   I2C_send(0x00);             //Start the clock and set seconds hand to zero
   I2C_stop();

   while(1)
   {
      sec = RTC1307_read(0x00);
      min = RTC1307_read(0x01);
      hour = RTC1307_read(0x02);

      date = RTC1307_read(0x04);
      month = RTC1307_read(0x05);
      year = RTC1307_read(0x06);
      LCD (0x80, 0);                    //Display time on  line 1, position 1
      LCD_str("Time : ");
      LCD (BCD2Upperch(hour), 0);
      LCD (BCD2Lowerch(hour), 0);
```

```
        LCD (':', 0);
        LCD (BCD2Upperch(min), 0);
        LCD ta(BCD2Lowerch(min), 0);
        LCD (':', 0);
        LCD (BCD2Upperch(sec), 0);
        LCD (BCD2Lowerch(sec), 0);
        LCD (0xC0, 0);                    // Display date on  line 2, position 1
        LCD_str ("Date: ");
        LCD (BCD2Upperch(date), 0);
        LCD (BCD2Lowerch(date) , 0);
        LCD ('/', 0);
        LCD (BCD2Upperch(month), 0);
        LCD (BCD2Lowerch(month), 0);
        LCD ('/', 0);
        LCD (BCD2Upperch(year), 0);
        LCD (BCD2Lowerch(year), 0);
        DELAY(1000);
    }
}
```

Interfacing of EEPROM using SPI with PIC

- Sometimes, it can be useful to interface the PIC with an external memory for data logging or other reasons, since limited memory is available on the PIC itself.
- There are various types of memory which a PIC can communicate with. One common type of memory is EEPROM memory, which is non-volatile, keeping its data when power is turned off.
- One type of EEPROM memory uses SPI to communicate with the PIC. In this section, we will discuss interfacing of 25XX040A: SPI based Serial EEPROM to a PIC18Fxxx.
- The 25XX040A is a 512-byte (4 kbit) Serial EEPROM designed to interface directly with the Serial Peripheral Interface (SPI) port of many of today's popular microcontroller families, including PIC microcontrollers.
- The memory is accessed via a simple Serial Peripheral Interface (SPI) compatible serial bus.
- The bus signals required are a **clock input (SCK)** plus **separate data in (SI)** and **data out (SO)** lines.
- Access to the device is controlled through a Chip Select (CS) input.
- Communication to the device can be paused via the hold pin (HOLD).



Pin diagram

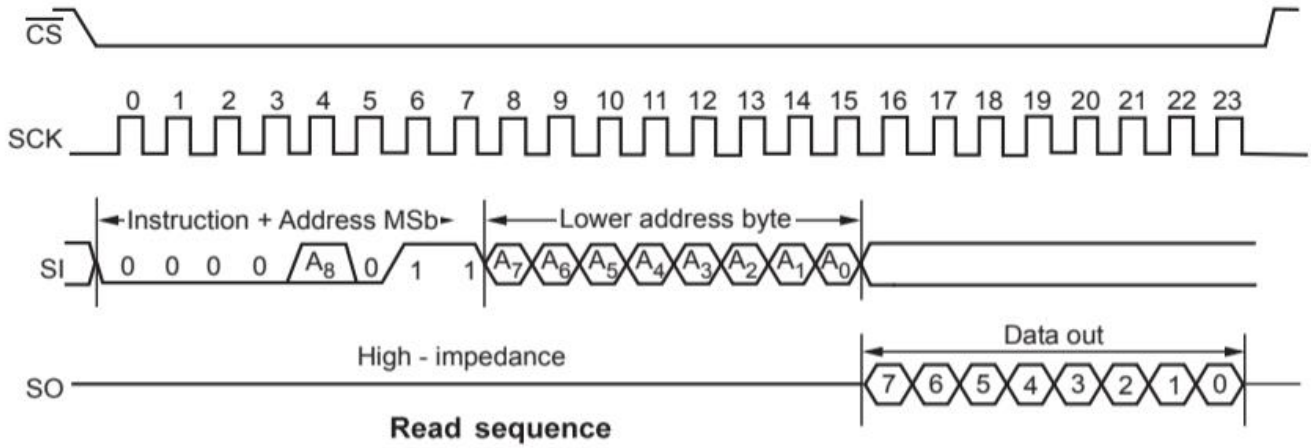
Name	Function
$\overline{CS}$	Chip Select Input
SO	Serial Data Output
$\overline{WP}$	Write-Protect
VSS	Ground
SI	Serial Data Input
SCK	Serial Clock Input
$\overline{HOLD}$	Hold Input
VCC	Supply Voltage

Pin functions

Instruction Name	Instruction Format	Description
READ	0 0 0 0 A_8 011	Read data from memory array beginning at selected address
WRITE	0 0 0 0 A_8 010	Write data to memory array beginning at selected address
WRDI	0 0 0 0 x 100	Reset the write enable latch (disable write operations)
WREN	0 0 0 0 x 110	Set the write enable latch (disable write operations)
RDSR	0 0 0 0 x 101	Read STATUS register
WRSR	0 0 0 0 x 001	Write STATUS register

Read Sequence

- The device is selected by pulling $\overline{CS}$ low. The 8-bit READ instruction is transmitted to the 25XX040A followed by a 9-bit address. The MSb (A8) is sent to the slave during the instruction sequence. This is illustrated in Fig. 9.6.2.
- After the correct READ instruction and address are sent, the data stored in the memory at the selected address is shifted out on the SO pin.
- Data stored in the memory at the next address can be read sequentially by continuing to provide clock pulses to the slave. The internal address pointer is automatically incremented to the next higher address after each byte of data is shifted out.

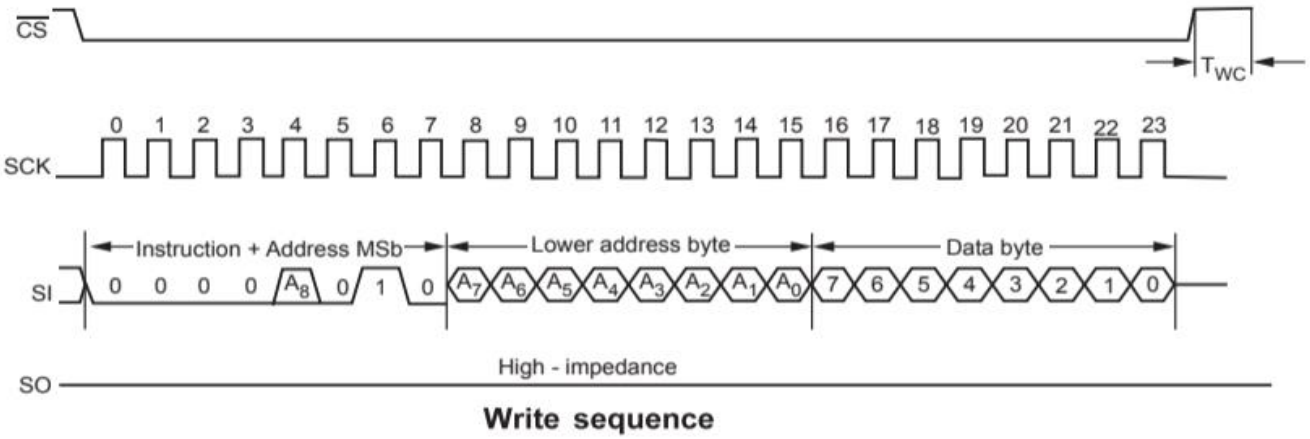


Read sequence

- When the highest address is reached (1FFh), the address counter rolls over to address 000h allowing the read cycle to be continued indefinitely. The read operation is terminated by raising the ($\overline{CS}$) pin.

Write Sequence

- Before any attempt to write data to the 25XX040A, the write enable latch must be set by issuing the WREN instruction. This is done by setting ($\overline{CS}$) low and then clocking out the proper instruction into the 25XX040A. After all eight bits of the instruction are transmitted, ($\overline{CS}$) must be driven high to set the write enable latch.
- After setting the write enable latch, the user may proceed by driving ($\overline{CS}$) low, issuing a write instruction, followed by the remainder of the address, and then the data to be written. Keep in mind that the Most Significant address bit (A8) is included in the instruction byte for the 25XX040A.

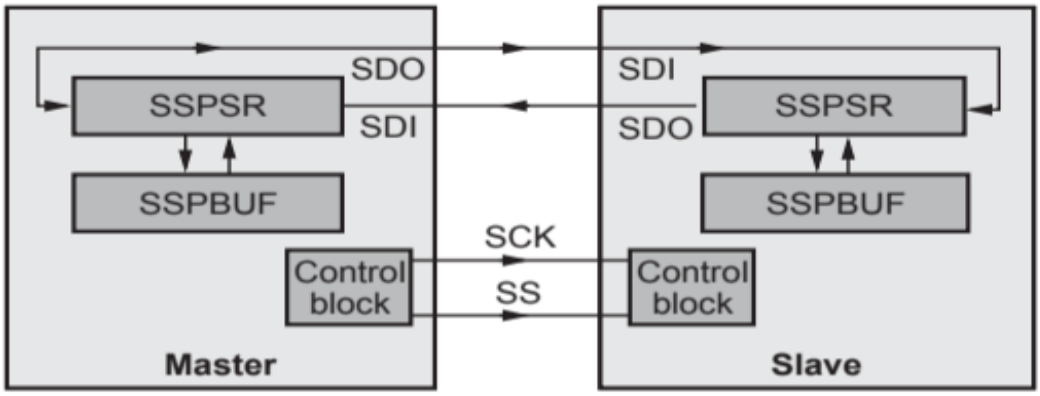


Write sequence

- EEPROM is divided into pages. Each page is of 16 bytes. Physical page boundaries start at addresses that are integer multiples of the page buffer size (16 bytes). So if starting write address is integer multiple of the page buffer size, we can write 16 bytes in one write cycle.

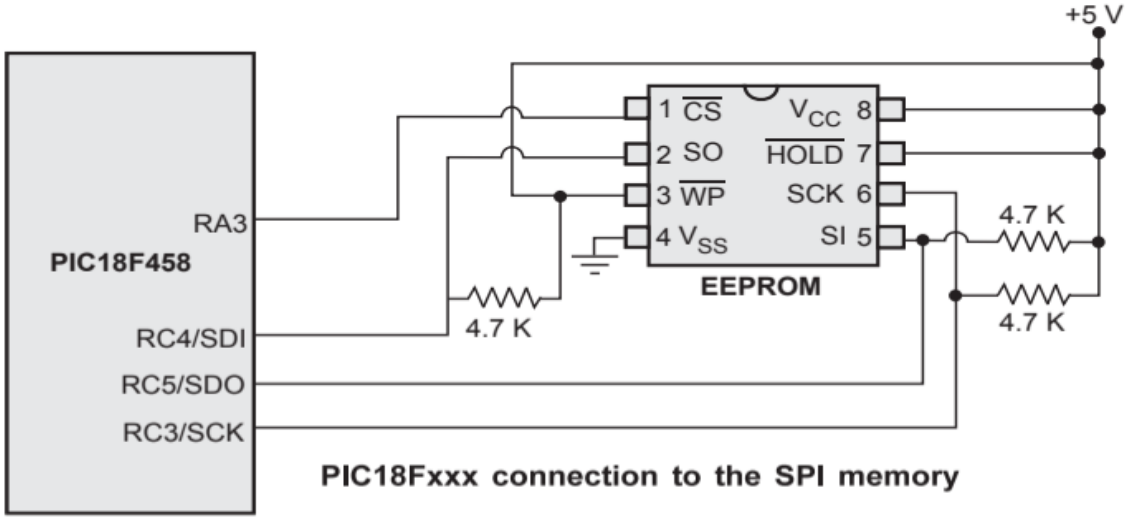
Interfacing

- SPI allows 8-bit of data to be synchronously transmitted and received simultaneously.
- Data leaving the master is put on the SDO (Serial Data Output) line and data entering the master enters via the SDI (Serial Data Input) line.
- A clock (SCK), is generated by the master device. It controls when and how quickly data is exchanged between the two devices.
- Master select devices using the SS (Slave Select) line.



Interfacing between master and slave

- The PIC18Fxxx is connected to the SPI memory, as shown in Fig. The Chip Select signal (($\overline{CS}$)) is pulled low during communication. It cannot be permanently tied low, even if only one memory is used, because the low-high transition of ($\overline{CS}$) latches in data. ($\overline{HOLD}$) and ($\overline{WP}$) (Write Protect) can be useful in some instances.



PIC18Fxxx connection to the SPI memory

****