TABLE OF CONTENTS

Unit IV

Unit III

Unit IV

Unit V

Unit VI

Unit IV

5 Basics of Serial Communication Protocols

5.1 : Introduction

Q.1 Give comparison between serial and parallel data transfer.

Ans. :

Sr. No.	Parallel data transfer	Serial data transfer
1.	Group of bits (usually 8-bits) of data is transferred at a time.	Only one bit of data is transferred at a time.
2.	Data transfer rate is high.	Data transfer rate is low.
3.	Preferred for short distance.	Preferred for long distance.
4.	Cabling cost is more.	Cabling cost is less.
5.	Less reliable.	More reliable
6.	Examples : GPIB, PCI, ISA etc.	Examples : RS 232C, USB, HDLC etc.

Table Q.1.1 Comparison between parallel and serial data transfer

5.2 : Basics of Serial Communication

Q.2 Give the classification of serial data transmission.

OR Explain full-duplex, half-duplex, and simplex serial data transfer.

Ans. : Serial data transmission can be classified on the basis of how transmission occurs.

1. Simplex 2. Half duplex 3. Full duplex

Simplex

- In simplex, the hardware exists such that data transfer takes place only in one direction. There is no possibility of data transfer in the other direction. A typical example is transmission from a computer to the printer.

Half Duplex

- The half duplex transmission allows the data transfer in both directions, but not simultaneously. A typical example is a walkie-talkie.

Full Duplex

- The full duplex transmission allows the data transfer in both direction simultaneously. The typical example is transmission through telephone lines.

Q.3 State the types of serial data communication.

Ans. : Serial data communication uses two types of communications

- Asynchronous serial data communication
- Synchronous serial data communication

Q.4 Write a note on asynchronous serial data communication.

Ans. : Fig. Q.4.1 shows the transmission format for asynchronous transmission. Asynchronous formats are character oriented. In this, the bits of a character or data word are sent at a constant rate, but characters can come at any rate (asynchronously) as long as they do not overlap. When no characters are being sent, a line stays high at logic 1 called **mark**, logic 0 is called **space**.

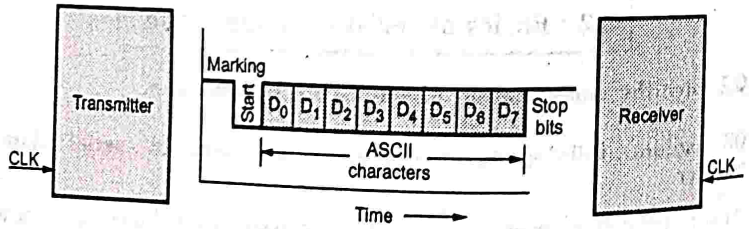


Fig. Q.4.1 Transmission format for asynchronous transmission

- The beginning of a character is indicated by a start bit which is always low. This is used to synchronize the transmitter and receiver. After the start bit, the data bits are sent with least significant bit first, followed by one or more stop bits (active high). The stop bits indicate the end of character. Different systems use 1, 1 1/2 or 2 stop bits. The combination of start bit, character and stop bits is known as **frame.**
- The start and stop bits carry no information, but are required because of the asynchronous nature of data. Fig. Q.4.2 illustrates how the data byte CAH would look when transmitted in the asynchronous serial format.

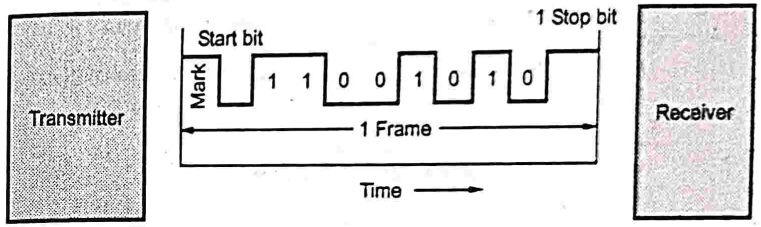


Fig. Q.4.2 Asynchronous format with data byte CAH

- The data rate can be expressed as bits/sec or characters/sec. The term bits/sec is also called the **baud rate.** The asynchronous format is generally used in low-speed transmission (less than 20 kbits/sec).

Q.5 Write a note on synchronous serial data communication.

Ans. : The start and stop bits in each frame of asynchronous format represents wasted overhead bytes that reduce the overall character rate. These start and stop bits can be eliminated by synchronizing receiver and transmitter.

- They can be synchronized by having a common clock signal. Such a communication is called **synchronous serial communication.** The Fig. Q.5.1 shows the transmission format of synchronous serial communication. In this transmission synchronous bits are inserted instead of start and stop bits.

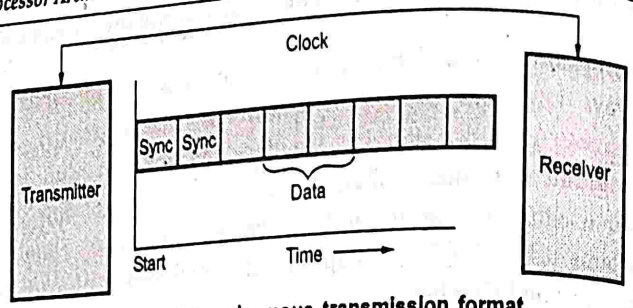


Fig. Q.5.1 Synchronous transmission format

Q.6 Give comparison between asynchronous and synchronous serial communication. ☞ [SPPU : June-22, Dec.-22, Marks 5]

Ans. :

Sr. No.	Asynchronous serial communication	Synchronous serial communication
1.	Transmitters and receivers are not synchronized by clock.	Transmitters and receivers are synchronized by clock.
2.	Bits of data are transmitted at constant rate.	Data bits are transmitted with synchronization of clock.
3.	Character may arrive at any rate at receiver.	Character is received at constant rate.
4.	Data transfer is character oriented.	Data transfer takes place in blocks.
5.	Start and stop bits are required to establish communication of each character.	Start and stop bits are not required to establish communication of each character; however, synchronization bits are required to transfer the data block.
6.	Used in low-speed transmissions at about speed less than 20 kbits/sec.	Used in high-speed transmissions.

Table Q.6.1 Comparison between asynchronous and synchronous serial data transfer

Q.7 What is a baud rate ?

Ans. : Baud rate is data transmission speed. The rate at which the bits are transmitted (bits/second) is called **baud** or **transfer rate.** The baud rate is the reciprocal of the time to send 1-bit. The common baud rates are multiples of 75 bits per second (bit/s) are typically : 75, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400 bit/s.

5.3 : Study of RS 232

Q.8 Write a note on RS232 C. ☞ [SPPU : June-22, Marks 6]

Ans. : • RS232 is widely accepted for single ended data transmission over short distances with low data rates.

• This standard describes the functions of 25 signal and handshake pins for serial data transfer. It also describes the voltage levels, impedance levels, rise and fall times, maximum bit rate, and maximum capacitance for these signal lines. RS 232C specifies 25 signal pins and it specifies that the DTE connector should be male, and the DCE connector should be a female. The most commonly used connector should be a female. The most commonly used connector, DB-25P is shown in the Fig. Q.8.1.

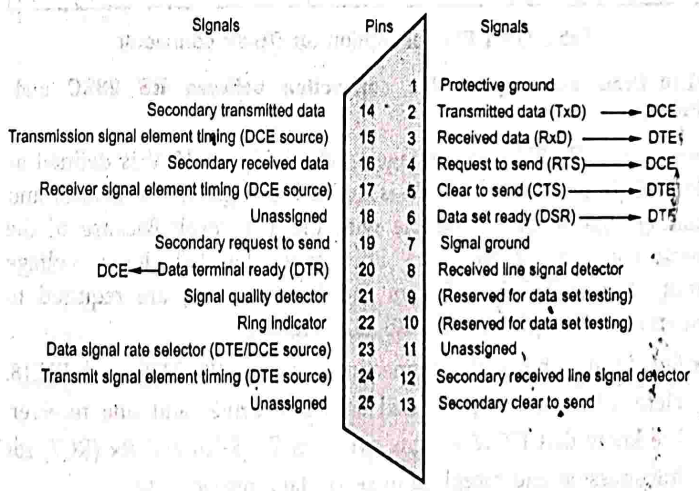


Fig. Q.8.1

Q.9 Explain the function of RS232C pins of DB-9 connector.

Ans. :

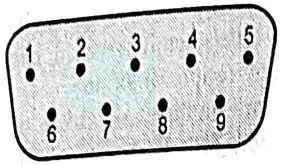


Fig. Q.9.1 DB-9P connector

The Table Q.9.1 describes the pins for DB-9P and Fig. Q.9.1 shows the DB-9P connector.

Pin no.	Pin	Description
1	$\overline{DCD}$	Data carrier detect
2	RxD	Received data
3	TxD	Transmitted data
4	DTR	Data terminal ready
5	GND	Signal ground
6	$\overline{DSR}$	Data set ready
7	RTS	Request to send
8	CTS	Clear to send
9	RI	Ring indicator

Table Q.9.1 Pin description for DB-9P connector

Q.10 Draw and explain the connection between RS 232C and PIC18

Ans. : • In RS 232C the voltage level + 3 V to + 15 V is defined as logic 0; from - 3 V to - 15 V is defined as logic 1. The control and timing signals are compatible with the TTL level. Because of the incompatibility of the data lines with the TTL logic, voltage translators, called line drivers and line receivers, are required to interface TTL logic with the RS 232C signals.

• Fig. Q.10.1 shows the connection between RS 232C and PIC18. Here, MAX 232 chip is used as a line driver and line receiver. We know that PIC18 assigns two pins Tx (RC6) and Rx (RC7) for transmission and reception of serial data, respectively.

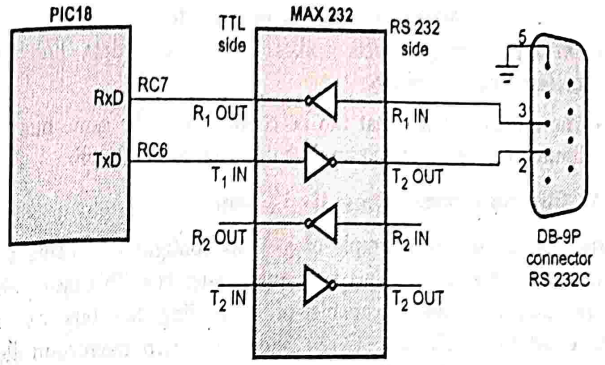


Fig. Q.10.1 Connection between RS 232C and PIC18

• These pins are TTL compatible; therefore, they require line driver and line receiver to make them RS 232C compatible. The MAX 232 has two sets of line drivers and line receivers for transmitting and receiving data.

• Only one set is required for one serial communication.

5.4 : Study of I^2C

Q.11 What are the features of the I^2C bus ?

☞ [SPPU : June-22, Marks 6]

Ans. : The important features of the I^2C-bus are :

- Only two bus lines are required; a serial data line (SDA) and a serial clock (SCL).
- Each device connected to the bus is software addressable by a unique address and simple master / slave relationships exist at all times; masters can operate as master - transmitters or as master - receivers.
- It's a true multi - master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.
- Serial, 8-bit oriented, bi-directional data transfers can be made at -
  - Up to 100 kbit/s in the Standard-mode,
  - Up to 400 kbit/s in the Fast-mode or

- Up to 3.4 Mbit/s in the High-speed mode
- On-chip filtering rejects spikes on the bus data line to preserve data integrity.
- The number of ICs that can be connected to the same bus is limited only by a maximum bus capacitance of 400 pF.

Q.12 Give bus configuration of the I^2C bus.

Ans. : Let us see the example of I^2C bus configuration using two microcontrollers. The I^2C-bus is a multi-master bus. This means that more than one device capable of controlling the bus can be connected to it. The Fig. Q.12.1 shows the two microcontrollers connected to I^2C-bus. Due to two microcontrollers, there exists master-slave and receiver-transmitter relationships on the I^2C-bus. It should be noted that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows :

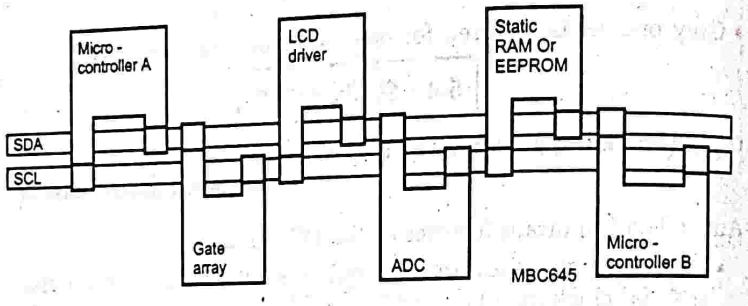


Fig. Q.12.1 Two microcontrollers connected to I^2C-bus

- **Case 1 :** Microcontroller A wants to send information to microcontroller B
  - Microcontroller A (master), addresses microcontroller B (slave).
  - Microcontroller A (master-transmitter), sends data to microcontroller B (slave- receiver).
  - Microcontroller A terminates the transfer.
- **Case 2 :** Microcontroller A wants to receive information from microcontroller B

  - Microcontroller A (master) addresses microcontroller B (slave).
  - Microcontroller A (master- receiver) receives data from microcontroller B (slave- transmitter).
  - Microcontroller A terminates the transfer.
- Even in this case, the master (microcontroller A) generates the timing and terminates the transfer.
- The possibility of connecting more than one microcontroller to the I^2C-bus means that more than one master could try to initiate a data transfer at the same time. To avoid the chaos an arbitration procedure has been developed.

Q.13 List I^2C signals.

Ans. :

Start - High-to-low transition of the SDA line while SCL line is high.

Stop - Low-to-high transition of the SDA line while SCL line is high.

Ack - Receiver pulls SDA low while transmitter allows it to float high.

Data - Transition takes place while SCL is low.

Q.14 Explain how to initiate and terminate data transfer on I^2C bus.

Ans. :

- During times of no data transfer (idle time), both the clock line (SCL) and the data line (SDA) are pulled high through the external pull-up resistors. The START and STOP conditions determine the start and stop of data transmission. The START condition is defined as a high to low transition of the SDA when the SCL is high. The STOP condition is defined as a low to high transition of the SDA when the SCL is high. Fig. Q.14.1 shows the START and STOP conditions. The master generates these conditions for starting and terminating data transfer. Due to the

definition of the START and STOP conditions, when data is being transmitted, the SDA line can only change state when the SCL line is low.

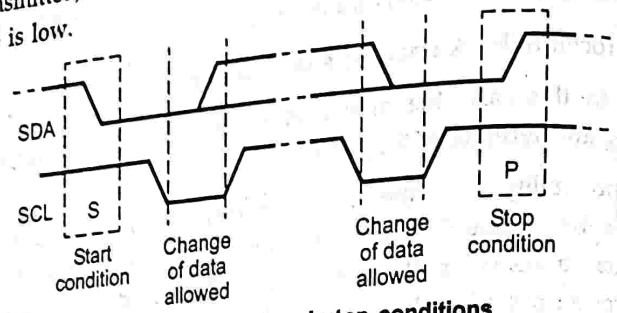


Fig. Q.14.1 Start and stop conditions

- START and STOP conditions are always generated by the master. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition.
- The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. In this respect, the START (S) and repeated START (Sr) conditions are functionally identical. Thus, hereafter, the S symbol will be used as a generic term to represent both the START and repeated START conditions, unless Sr is particularly relevant.
- Detection of START and STOP conditions by devices connected to the bus is easy if they incorporate the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period to sense the transition.

Q.15 Write a short note on data transfer on the I^2C bus.

[SPPU : Dec.-22, Marks 6]

Ans. : In the byte format, 8-bit data is put on the SDA line. The number of bytes that can be transmitted per transfer is unrestricted. Each byte has to be followed by an acknowledge bit. Data is transferred with the most significant bit (MSB) first as shown in Fig. Q.15.1. If a slave can't receive or transmit another complete byte of data until it has performed some other function, for

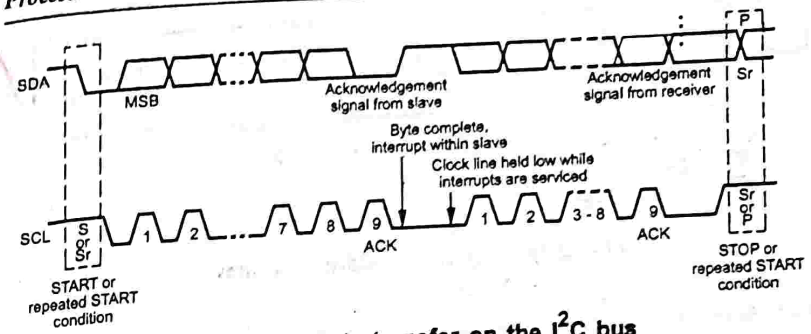


Fig. Q.15.1 Data transfer on the I^2C bus

example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer then continues when the slave is ready for another byte of data and releases clock line SCL.

- A message which starts with such an address can be terminated by generation of a STOP condition, even during the transmission of a byte. In this case, no acknowledge is generated.

Q.16 Write a note on addressing I^2C devices.

Ans. : There are two address formats. The simplest is the 7-bit address format with a R/W bit. The more complex is the 10-bit address with a R/W bit. For 10-bit address format, two bytes must be transmitted with the first five bits specifying this to be a 10-bit address. The 10-bit addressing allows the use of up to 1024 additional slave addresses. The 10-bit addressing does not affect the existing 7-bit addressing. Devices with 7-bit and 10-bit addresses can be connected to the same I^2C-bus.

- Fig. Q.16.1 shows the 7-bit address format and Fig. Q.16.1(a) shows the 10-bit address format.

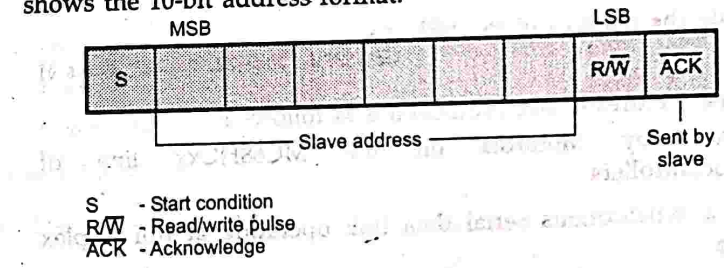


Fig. Q.16.1 7-bit address format

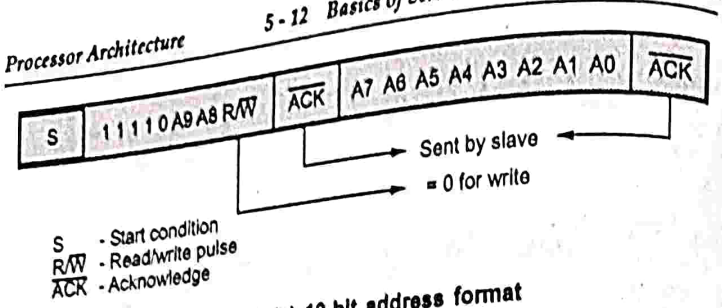


Fig. Q.16.1 (a) 10-bit address format

Q.17 State advantages, disadvantages and applications of the I^2C bus.

Ans. :

Advantages of I^2C

- Good for communication with on-board devices that are accessed occasionally.
- Easy to link multiple devices because of addressing scheme.
- Cost and complexity do not scale up with the number of devices.

Disadvantage of I^2C

- The complexity of supporting software components can be higher than that of competing schemes (for example, SPI).

Applications of I^2C

- Used as a control interface to signal processing devices that have separate data interfaces, e.g. RF tuners, video decoders and encoders, and audio processors.

5.5 : Study of SPI

Q.18 State the features of the SPI bus.

☞ **[SPPU : June-22, Dec.-22, Marks 6]**

Ans. : The features of SPI protocol are as follows :

- Defined by Motorola on the MC68HCxx line of microcontrollers.
- It is a synchronous serial data link operating at full duplex mode.

- Generally faster than I^2C, capable of 3 Mbps data transfer rate.
- The SPI dataword size is 8-bits.
- The SPI provides hold facility which allows the transmitter to suspend data transfer.
- In SPI data can be transferred in multiple bytes known as blocks or pages.

Q.19 State the standard naming convention used in SPI.

Ans. :

- The standard naming convention used for the SPI signals is as follows :

Data signals : MOSI : Master data output, slave data input

MISO : Master data input, slave data output

Control signals : SS : Slave Select signal (*our example uses two of those, SS_ADC and SS_MEM*)

SCK : Serial Clock (*always driven by the Master*)

Q.20 Explain clock polarity and clock phase in SPI.

Ans. : The SPI interface requires the master and slave devices to be agreed regarding the idle state of the clock signal as well as the way in which the data will be clocked during the SPI transfer.

Clock Polarity : The Clock Polarity (**CPOL**) parameter indicates the level of the clock signal when it is idle.

Clock Phase : The Clock Phase (**CPHA**) parameter indicates when the data should be presented and when it should be sampled.

Q.21 State the applications of the SPI bus.

Ans. : Applications of SPI bus are :

- Like I^2C, used in EEPROM, Flash, and real time clocks.
- Better suited for "data streams", i.e. ADC converters.

- Full duplex capability, hence can be used in communication between a codec and digital signal processor.

Q.22 Compare SPI and I^2C.

Ans. :

- For point-to-point, SPI is simple and efficient.
- Less overhead than I^2C due to lack of addressing, plus SPI is full duplex.
- For multiple slaves, each slave needs separate slave select signal hence more effort and more hardware required than I^2C.

5.6 : UART/USART

Q.23 Explain USART module in PIC18. List the registers associated with transmission and reception modes of USART.
☞ [SPPU : Dec.-22, Marks 6]

Ans. :

- The Universal Synchronous Asynchronous Receiver Transmitter (USART) module is one of the three serial I/O modules incorporated into PIC18FXX8 devices.
- USART is also known as a Serial Communications Interface or SCI.
- It can be configured in the following modes:
  - Asynchronous (full-duplex)
  - Synchronous - **Master** (half-duplex)
  - Synchronous - **Slave** (half-duplex)
- In full-duplex asynchronous mode, it is used to communicate with peripheral devices, such as CRT terminals and personal computers.
- In a half-duplex synchronous system, it is used to communicate with peripheral devices, such as A/D or D/A integrated circuits, serial EEPROMs, etc.

Registers associated with transmission and reception modes of USART are :

1. TXREG USART Transmit Register
2. RCREG : Receive Register
3. TXSTA : Transmit Status and Control Register
4. RCSTA : Receive Status and Control Register
5. SPBRG : Serial Port Baud Rate Generator
6. PIR1 : Peripheral Interrupt Request Register.

Q.24 Explain the use of TXREG, TSR and RCREG.

Ans. : Transmit Register and TSR (Transmit Shift Register)

- It is an 8-bit Transmit Buffer register. The data to be transmitted through Tx (RC6) pin is loaded into TxREG register.
- TSR (Transmit Shift Register) is loaded with new data from the TXREG register (if available). Once the TXREG register transfers the data to the TSR register, the TXREG register is empty and flag bit TXIF (PIR1 register) is set.
- TXIF will reset only when new data is loaded into the TXREG register.
- The data from TSR along with start and stop bits are transmitted serially through Tx pin.

RCREG : Receive Register

- It is an 8-bit Receive Buffer register. The received through Rx (RC7) pin is loaded into RCREG register.
- On completely receiving the word, the RSR (Receive Shift Register) will transfer the data to the RCREG register.

Q.25 Draw the format of TXSTA register and explain the significance of each bit.

Ans. : Fig. Q.25.1 shows the Transmit Status and Control register (TXSTA)

- The TXSTA register is used to select the clock source, mode of transmission, baud rate, and to enable / disable transmission.

CSRC	TX9	TXEN	SYNC	-	BRGH	TRMT	TX9D

bit 7 bit 0

bit 7 **CSRC :** Clock Source Select bit

Asynchronous mode :

Don't care

Synchronous mode :

1 = Master mode (clock generated internally from BRG)

0 = Slave mode (clock from external source)

bit 6 **TX9 :** 9 bit Transmit Enable bit

1 = Selects 9-bit transmission

0 = Selects 8 bit transmission

bit 5 **TXEN :** Transmit Enable bit

1 = Transmit enable

0 = Transmit disabled

Note : SREN/CREN overrides TXEN in Sync mode,

bit 4 **SYNC :** USART Mode select bit

1 = Synchronous mode

0 = Asynchronous mode

bit 3 **Unimplemented :** Read as '0'

bit 2 **BRGH :** High Baud Rate Select bit

Asynchronous mode :

1 = High speed

0 = Low Speed

Synchronous mode :

Unused in this mode.

bit 1 **TRMT :** Transmit Shift Register Status bit

1 = TSR empty

0 = TSR full

bit 0 **TX9D :** 9th bit of Transmit Data

Can be address / data bit or a parity bit.

Fig. Q.25.1 Transmit Status and Control register (TXSTA)

Q.26 Draw the format of RCSTA register and explain the significance of each bit.

Ans. : Fig. Q.26.1 shows the Receive Status and Control register (RCSTA)

- The RCSTA register is used to enable the single or continuous receive mode and to enable / disable reception.

SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D

bit 7 bit 0

bit 7 **SPEN :** Serial Port Enable bit

1= Serial port enabled (configures RX/DT and TX/CK pins as serial port pins)

0 = Serial port disabled

bit 6 **RX9 :** 9-bit Receive Enable bit

1 = Selects 9-bit reception

0 = Selects 8-bit reception

bit 5 **SREN :** Single Receive Enable bit

Asynchronous mode :

Don't care

Synchronous mode - Master :

1 = Enables single receive

0 = Disables singles receive (this bit is cleared after reception is complete)

Synchoronous mode : Slave :

Unused in this mode

bit 4 **CREN :** Continous Receive Enable bit

Asynchronous mode :

1 = Enables continous receive

0 = Disables continous receive

Synchronous mode :

1 = Enables continous receive until CREN is cleared (CREN overrides SREN)

0 = Disables continous receive

bit 3 ADDEN : Address detect Enable bit

Asynchronous mode 9-bit (RX9 = 1) :

1 = Enables address detection, enables interrupt and load of the receive buffer when RSR<8> is set

0 = Disables address detectionm all bytes are received and ninth bit can be used parity bit

bit 2 FERR : Framing Error bit

1 = Framing error (can be updated by reading RCREG register and receive next valid byte)

0 = No framing error

bit 1 OERR : Overrun Error bit

1 = Overrun error (can be cleared by clearing bit CREN)

0 = No overrun error

bit 0 RX9D : 9th bit of Received Data

Can be addresss/data bit or parity bit

Fig. Q.26.1 Receive Status and Control register (RCSTA)

Q.27 Explain the use of SPBRG register.

Ans. : The SPBRG register is a bit register used to hold the value which decides the baud rate, i.e., it is used to program the baud rate in both the Asynchronous and Synchronous modes of the USART.

Q.28 Explain the procedure to calculate the value to be loaded in the SPBRG register for a specific baud rate.

Ans. : Given the desired baud rate and FOSC, the nearest integer value for the SPBRG register can be calculated using the formula shown in Table Q.28.1. It shows the formula for the computation of the baud rate for different USART modes.

SYNC	BRGH = 0 (Low speed)	BRGH = 1 (High speed)
0	(Asynchronous) Baud Rate = Fosc/(64 (X+1))	Baud Rate = Fosc /(16 (X+1))
1	(Synchronous) Baud Rate = Fosc/(4 (X+1))	NA

Note : X = value in SPBRG (0 to 255)

Table Q.28.1 : Baud rate formula

- As shown in Table Q.28.1, bit BRGH (TXSTA register) also controls the baud rate in asynchronous mode. In Synchronous mode, bit BRGH is ignored.
- **For example,** when SYNC = 0 and BRGH = 0,

$$X = \frac{FOSC}{\text{Desired Baud Rate} \times 64} - 1$$

Q.29 Find the value to be loaded in SPBRG register to have the following baud rates. Assume asynchronous mode with Fosc = 10 MHz and low baud rate.
a. 9600 b. 4800

Ans. : For high speed asynchronous mode, BRGH = 0 and SYNC = 0. Therefore, baud rate is given by

$$\text{Desired Baud Rate} = \frac{FOSC}{64(X+1)} \text{ and}$$

$$X = \frac{FOSC}{\text{Desired Baud Rate} \times 64} - 1$$

a) For baud rate = 9600

$$X = \frac{100 \times 10^6}{9600 \times 64} - 1 = 15.28 \approx 15 \text{ in decimal} = 0 \text{ FH}$$

b) For baud rate = 4800

$$X = \frac{100 \times 10^6}{4800 \times 64} - 1 = 31.55 \approx 32 \text{ in decimal} = 20 \text{ H}$$

Note : Since we have to choose integer value to be loaded in SPBRG register, there may be an error in the baud rate value.

- It may be advantageous to use the high baud rate (BRGH = 1) even for slower baud clocks. This is because the FOSC /(16(X + 1)) equation can reduce the baud rate error in some cases.

Q.30 Find the value to be written into the SPBRG register to generate 9600 baud for asynchronous mode high-speed transmission assuming the frequency of the crystal oscillator is 20 MHz. Also show that high baud rate reduces the baud rate error.

Ans. : The value (for BRGH = 1) to be written into the SPBRG register is

$$X = \frac{20\times10^6}{9600\times16} - 1 = 130.2 - 1 = 129.2 \approx 129$$

The actual baud rate is

$$\text{Baud Rate} = \frac{20\times10^6}{16(129+1)} = 9615.38$$

$$\text{Error} = \frac{\text{Calculated Baud Rate} - \text{Desired Baud Rate}}{\text{Desired Baud Rate}} \times 100$$

$$= \frac{9615.38 - 9600}{9600} \times 100 = 0.16\ \%$$

The same baud rate can also be achieved by using low speed (BRGH = 0) approach in which

$$X = \frac{20\times10^6}{9600\times64} - 1 = 32.5 = 31.55 \approx 32$$

The actual baud rate is

$$\text{Baud Rate} = \frac{20\times10^6}{64(32+1)} = 9469.59$$

$$\text{Error} = \frac{9469.69 - 9600}{9600} \times 100 = 1.36\ \%$$

The results show that high baud rate reduces the baud rate error.

Q.31 Explain the interrupt flags bits used in transmission and reception modes of USART.

Ans. : The Peripheral Interrupt Request (PIR) registers contain the individual flag bits for the peripheral interrupts.

- Bits 4 and 5 of PIR1 register are used by the USART. They are :
  - RCIF : USART Receive Interrupt Flag bit
  - TXIF : USART Transmit Interrupt Flag bit

PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

bit 7 bit 0

bit 5 **RCIF** : USART Receive Interrupt Flag bit

1 = The USART receive buffer, RCREG is full (cleared when RCREG, is read)

0 = The USART receive buffer is empty

bit 4 **TXIF** : USART Receive Interrupt Flag bit

1 = The USART transmit buffer, TXREG is empty (cleared when TXREG is written)

0 = The USART transmit buffer is full

Fig. Q.31.1 Bits of Peripheral Interrupt Request (PIR1) register used for USART

5.7 : USART Asynchronous Mode

Q.32 Write a note on USART asynchronous mode.

Ans. :

- In this mode, the USART uses standard Non-Return-to-Zero (NRZ) format (one Start bit, eight or nine data bits, and one Stop bit). The most common data format is 8 bits.
- An on-chip dedicated 8-bit Baud Rate Generator can be used to derive standard baud rate frequencies from the oscillator.
- The USART transmits and receives the LSB first. The USART's transmitter and receiver are functionally independent but use the same data format and baud rate.
- The Baud Rate Generator produces a clock, either x16 or x64 of the bit shift rate, depending on the BRGH bit (TXSTA register).

- Parity is not supported by the hardware but can be implemented in software (and stored as the ninth data bit).
- Asynchronous mode is stopped during Sleep. It is selected by making the SYNC bit = 0 (TXSTA register).
- The USART Asynchronous module consists of the following important elements:
  - Baud Rate Generator
  - Sampling Circuit
  - Asynchronous Transmitter
  - Asynchronous Receiver

Q.33 Draw and explain the block diagram of USART transmitter of PIC 18.

Ans. : Fig. Q.33.1 shows the block diagram of USART transmitter.

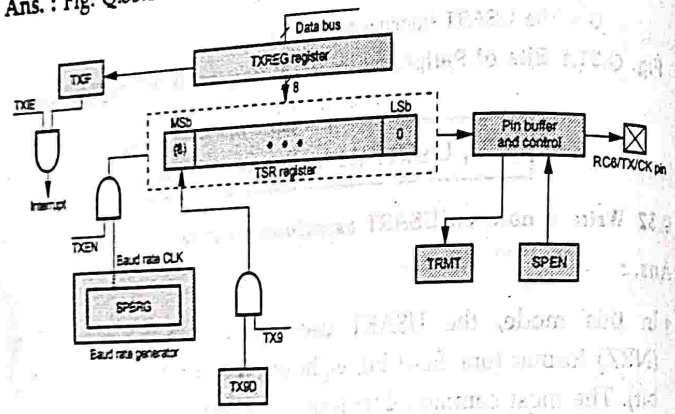


Fig. Q.33.1 Block diagram of USART transmitter

- The heart of the transmitter is the **Transmit (Serial) Shift Register (TSR)**. The TSR register obtains its data from the Read/Write Transmit Buffer register (TXREG).
- It is loaded with data in software. The TSR register is not loaded until the Stop bit has been transmitted from the previous load. As soon as the Stop bit is transmitted, the TSR is loaded with new data from the TXREG register (if available).

- Once the TXREG register transfers the data to the TSR register, the TXREG register is empty, and flag bit TXIF (PIR1 register) is set. This interrupt can be enabled/disabled by setting/clearing enable bit TXIE (PIE1 register). It will reset only when new data is loaded into the TXREG register.
- TRMT is a read-only bit that is set when the TSR register is empty. No interrupt logic is tied to this bit, so the user has to poll this bit to determine if the TSR register is empty.
- SPEN bit is set to enable the serial port.
- Tx9D holds the ninth bit to be used in 9-bit transmission.

Q.34 List the steps for setting up an asynchronous transmission in PIC 18.

Ans. :

1. Initialize the SPBRG register and BRGH bit for the appropriate baud rate.
2. Make SYNC bit = 0 and SPEN bit = 1 to enable the asynchronous serial port
3. If interrupts are desired, set enable bit TXIE.
4. If 9-bit transmission is desired, set transmit bit TX9.
5. Make TXEN = 1 to enable the transmission which will also set bit TXIF.
6. If 9-bit transmission is selected, the ninth bit should be loaded in bit TX9D.
7. Load data to the TXREG register (starts transmission).

Q.35 Draw the timing diagram for 8/9 bit asynchronous transmission in PIC 18.

Ans. : Fig. Q.35.1 shows the timing diagram for 8/9 bit asynchronous transmission.

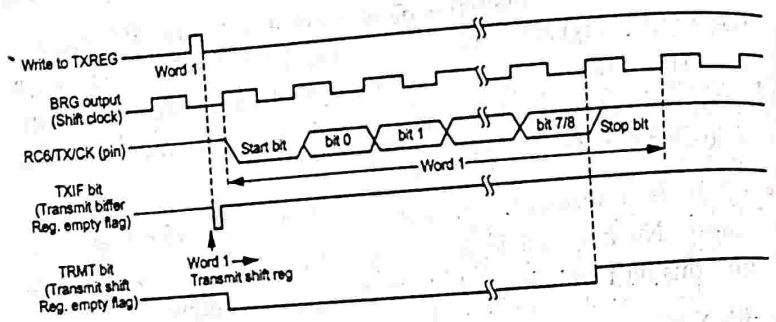


Fig. Q.35.1 Timing diagram for 8/9 bit asynchronous transmission

Q.36 Draw and explain the block diagram of USART receiver of PIC 18.

Ans. : Fig. Q.36.1 shows the block diagram of USART receiver.

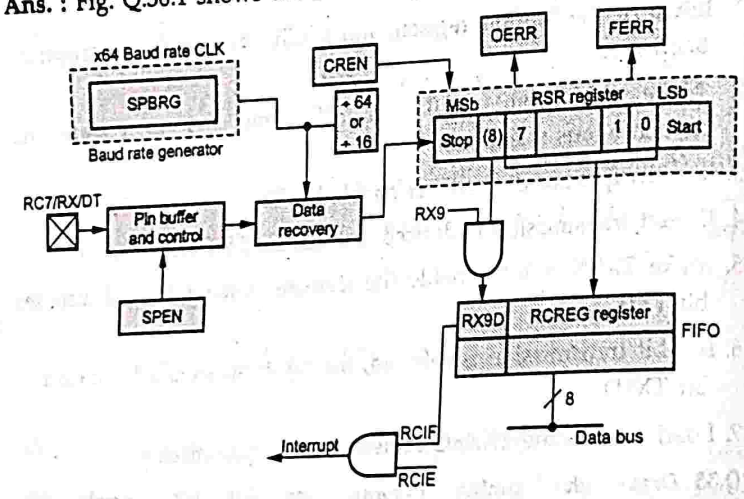


Fig. Q.36.1 Block diagram of USART receiver

- The data is received on the RC7/RX/DT pin and drives the data recovery block. The data recovery block is actually a high-speed shifter, operating at x16 times the baud rate, whereas the main receive serial shifter operates at the bit rate or at FOSC. This mode would typically be used in RS-232 systems.

- Once the Asynchronous mode is selected, reception is enabled by setting bit CREN.
- The heart of the receiver is the **Receive Serial Shift Register (RSR)**. After sampling the Stop bit, the received data frame in the RSR is transferred to the RCREG register (if it is empty).
- If the transfer is complete, the flag bit, RCIF, is set. The actual interrupt can be enabled/disabled by setting/clearing the enable bit, RCIE. Flag bit RCIF is a read-only bit which is cleared by the hardware. It is cleared when the RCREG register has been read and is empty.
- The RCREG is a double-buffered register, which means it's a two-level deep FIFO. With the help of it, we can receive two bytes and load them in FIFO.
- OERR bit is set if the third byte is received and RCREG is full.
- FERR bit is set if a low-level stop bit is detected.
- SPEN bit is set to enable the serial port.
- In 9-bit transmission, the ninth bit is stored in Rx9D.

Q.37 List the steps for setting up an asynchronous reception in PIC 18.

Ans. :

1. Initialize the SPBRG register and BRGH bit for the appropriate baud rate.
2. Make SYNC bit = 0 and SPEN bit = 1 to enable the asynchronous serial port
3. If interrupts are desired, set enable bit RCIE.
4. If 9-bit reception is desired, set bit RX9.
5. Enable the reception by setting bit CREN.
6. Flag bit RCIF will be set when reception is complete and an interrupt will be generated if enable bit RCIE was set.

7. Read the RCSTA register to get the ninth bit (if enabled) and determine if any error occurred during reception.
8. Read the 8-bit received data by reading the RCREG register.
9. If any error occurred, clear the error by clearing enable bit CREN.

Q.38 Draw the timing diagram for 8/9 bit asynchronous reception in PIC 18.

Ans. :

- Fig. Q.38.1 shows the timing diagram for 8/9 bit asynchronous reception.

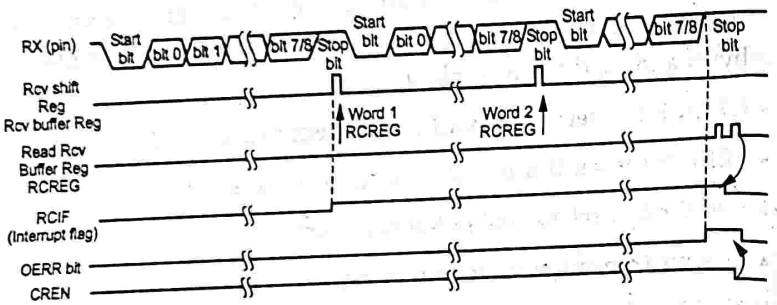


Fig. Q.38.1 Timing diagram for 8/9 bit asynchronous reception

- **Note :** This timing diagram shows three words appearing on the RX input. The RCREG (receive buffer) is read after the third word, causing the OERR (overrun) bit to be set.

Q.39 List the steps for setting up an asynchronous reception with address detect enable in PIC18.

Ans. : Steps to follow for setting up an Asynchronous Reception with Address Detect Enable

1. Initialize the SPBRG register and BRGH bit for the appropriate baud rate.
2. Make SYNC bit = 0 and SPEN bit = 1 to enable the asynchronous serial port

3. If interrupts are required, set the RCEN bit and select the desired priority level with the RCIP bit.
4. Set the RX9 bit to enable 9-bit reception.
5. Set the ADDEN bit to enable address detect.
6. Enable reception by setting the CREN bit.
7. The RCIF bit will be set when reception is complete. The interrupt will be acknowledged if the RCIE and GIE bits are set.
8. Read the RCSTA register to determine if any error occurred during reception, as well as read bit 9 of data (if applicable).
9. Read RCREG to determine if the device is being addressed.
10. If any error occurred, clear the CREN bit.
11. If the device has been addressed, clear the ADDEN bit to allow all received data into the receive buffer and interrupt the CPU.

5.8 : Serial Communication Programming using Embedded C

Q.40 List the steps for programming PIC 18 to transfer data serially.

Ans. : Steps for Programming PIC18 to Transfer Data Serially

1. Load the value in TXSTA register for the desired configuration as shown below

Value	Configuration
20H	Asynchronous mode with 8-bit data frame, low baud rate and transmit enable.
60H	Asynchronous mode with 9-bit data frame, low baud rate and transmit enable.
24H	Asynchronous mode with 8-bit data frame, high baud rate and transmit enable.
64H	Asynchronous mode with 9-bit data frame, high baud rate and transmit enable.

2. Make TX pin of PORTC (RC6) an output pin.
3. Initialize the SPBRG register and BRGH bit for the appropriate baud rate.
4. Set SPEN bit in the RCSTA register to enable the serial port.
5. Load the character byte required to transmit in TXREG register.
6. Monitor the TXIF bit of PIR1 register to make UART ready for the next byte.
7. Repeat steps 5 and 6 to transmit the next character.

Q.41 List the steps for programming PIC 18 to receive data serially.

Ans. : Steps for Programming PIC18 to Receive Data Serially

1. Load the value in RCSTA register for the desired configuration as shown below

Value	Configuration
90H	Asynchronous mode with 8-bit data frame and receive enable.
D0H	Asynchronous mode with 8-bit data frame and receive enable.

2. Initialize the SPBRG register and BRGH bit for the appropriate baud rate.
3. Make the RX Pin of PORT (RC7) as input for data to receive.
4. Monitor the RCIF bit = 1 of the PIR1 register to check for the entire character reception.
5. When RCIF is raised, the RCREG register has the byte. So save the contents of the RCREG register.
6. Repeat steps 5 and 6 to receive the next character.

Q.42 Write a C program for the PIC18 to transfer letter "A" serially at 9600 baud, continuously. Use 8-bit data and 1-stop bit. Assume XTAL = 10 MHz.

Ans. :

```
#include<P18F458.h>
void main(void)
{
   TXSTA = 0x20;              // Asynchronous mode with 8-bit
                              // data frame, low baud rate
                              // and transmit enable : TX9 = 0,
                              TXEN = 1, SYNC  = 0,
                              BRGH = 0
   SPBRG = 15;                // 9600 baud rate
   TRISBbits.TRISC6=0;        // configure pin 6 of Port C
                              (Transmitter pin)  as output
   RCSTAbits.SPEN=1;          // Enable serial port
   while(1)
{
   TXREG = 'A' ;              // Send character
   while(PIR1bits.TXIF==0);   // Wait until  TXREG register
                              // becomes empty
}
}
```

Q.43 Write a C program for the PIC18 to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously. Assume XTAL = 10 MHz.

Ans. :

```
#include<P18F458.h>
void main(void)
{
   TXSTA = 0x20;              // Asynchronous mode with 8-bit data frame,
                              // low baud rate
                              // and transmit enable : TX9 = 0, TXEN = 1,
                              // SYNC  = 0,
                              //  BRGH = 0
   SPBRG = 15;                // 9600 baud rate
   TRISBbits.TRISC6=0;        // configure pin 6 of Port C
                              //(Transmitter pin)  as output
```

```
RCSTAbits.SPEN=1;  // Enable serial port
while(1)
{
TXREG = 'Y' ;                 // Send character Y
while(PIR1bits.TXIF==0);      // Wait until TXREG register becomes
                                 empty
TXREG = 'E' ;                 // Send character E
while(PIR1bits.TXIF==0);      // Wait until TXREG register becomes
                                 empty
TXREG = 'S' ;                 // Send character S
while(PIR1bits.TXIF==0);      // Wait until TXREG register becomes
                                 empty
 }
}
```

Q.44 Write a C program for the PIC18 to receive bytes of data serially, and put them in PORT B, set the baud rate at 4800, 8-bit data, and 1 stop bit. Assume XTAL = 10 MHz.

Ans. :

```
#include<P18F458.h>
void main(void)
{
  RCSTA = 0x90; // Enable serial port and Continuous 8-bit reception
                // SPEN =1, RX9 = 0, CREN = 1
  SPBRG =  32;              // 4800 baud rate
  TRISBbits.TRISC7=0;       // configure pin 7 of Port C (receiver pin)
                            // as input
  TRISB = 0;                // configure Port B as output
  while(1)
  {
  while(PIR1bits.RCIF==0);// Wait until RCREG register becomes
                            //  full
  PORTB = RCREG;            // Send received data to PORT B
  }
}
```


Q.45 Write a C program for the PIC18 to send the two messages "Normal Speed" and "High Speed" to the serial port. Assuming that SW is connected to pin PORTB.0, monitor its status and set the baud rate as follows :
SW = 0, 9600 baud rate
SW = 1, 38400 baud rate
Assume that XTAL = 10 MHz for both cases.

Ans. :

```
#include<P18F458.h>
#define SWBit PORTBbits.RB0   // Switch input
void main(void)
{
  unsigned char z;
  unsigned char Mes1[ ]="Normal Speed";
  unsigned char Mes2[ ]="High Speed";
  TRISBbits.TRISB0=1;         // configure pin 0 of Port B as input
  TXSTA = 0x20;               // Asynchronous mode with 8-bit data
                              // frame, low baud rate
                              // and transmit enable : TX9 = 0,
                              // TXEN = 1, SYNC  = 0, BRGH = 0
  SPBRG = 15;                 // 9600 baud rate
  RCSTAbits.SPEN=1;           // Enable serial port
  If (SWBit == 0)
  {
    for (z =0; z<12; z++)
    {
      while(PIR1bits.TXIF==0);  // Wait until  TXREG register
                                //becomes empty
      TXREG = Mes1[z ] ;        // Send character
    }
  }
  else
  {
```

```
    TXSTAbits.BRGH = 1;              // Select high baud rate (Low
                                        baud rate x 4)
    for (z =0; z<10; z++)
    {
       while(PIR1bits.TXIF==0);      // Wait until TXREG register
                                     //becomes empty
       TXREG = Mes2[z ] ;            // Send character
    }
  }
while(1)
}
```

END...

Unit III

6 PIC Interrupts

6.1 : Interrupt Vs Polling

Q.1 What is ISR ? ☞ [SPPU : Dec.-16, 22, Mark 1]

Ans. : In the interrupt method, whenever any device needs microcontroller's service, it tells this to microcontroller by sending an interrupt signal. Upon receiving an interrupt signal, the microcontroller stops executing its current program and calls a subroutine called **Interrupt service routine** which services the interrupt.

Q.2 Compare polling and interrupt method.
☞ [SPPU : Dec.-22, Marks 4]

Ans. :

Sr. No.	Polling method	Interrupt method
1.	In polling, a lot of microcontroller's processing power is wasted in checking whether the I/O devices need service.	In interrupt method, microcontroller's time is not wasted in checking whether the I/O devices need service.
2.	Since all the devices are polled in a sequential manner, there is no explicit priority mechanism in the polling method.	Interrupt method can assign priorities to the interrupts so that they can be served according to priority basis.

3.	In polling, there is no possibility of ignoring a device request for service.	In interrupt method, the microcontroller can ignore (mask) a device request for service.
4.	Inefficient way of handling I/O devices.	Efficient way of handling I/O devices.
5.	No external hardware signal required.	External hardware signal is required to interrupt microcontroller.

Table Q.2.1 Comparison of interrupt method and polling

Q.3 Explain the steps in executing the interrupt.

Ans. : In response to the interrupt, the microcontroller goes through following steps :

1. It completes the execution of current instruction and save the address of the next instruction, i.e. the contents of PC on the stack.
2. It also saves the current status of all the interrupts internally i.e. not on the stack.
3. It then transfers the program control to a fixed location in memory called the **Interrupt Vector Table (IVT)**. IVT holds the address of the interrupt service routine.
4. The microcontroller gets the address of the Interrupt Service Routine (ISR) from IVT and transfers program control to the ISR. It executes instructions within the interrupt service routine in sequence till it reaches the RETI (Return From The Interrupt) instruction.
5. Upon executing the RETI instruction, it gets the Program Counter (PC) address from the stack and restores the interrupt logic. Since PC is loaded with the address of the next instruction from where the interrupt was initiated, it starts to execute from that address.

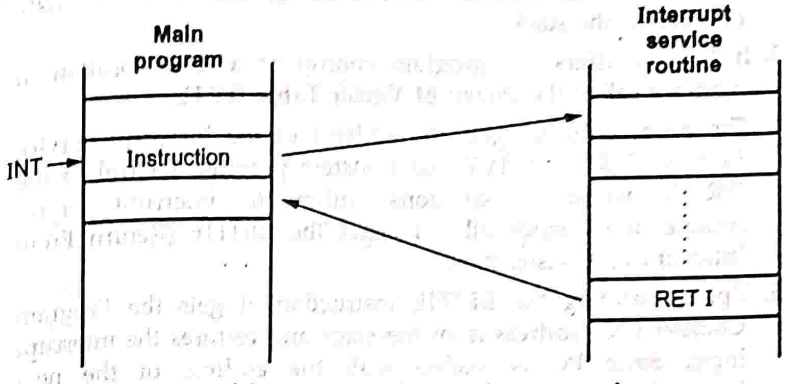


Fig. Q.3.1 Interrupt service routine processing

6.2 : IVT (Interrupt Vector Table)

Q.4 What is IVT ? ☞ [SPPU : Dec.-16,22, June-22, Marks 3]

Ans. : • IVT stands for **Interrupt Vector Table.** It holds the addresses of the interrupt service routines for various interrupts supported by the microcontroller.

• Table Q.4.1 shows the interrupt vector table for the PIC18.

Interrupt	Vector Address (RAM Location)
Power-on Reset	0000H
High Priority Interrupt	0008H
Low Priority Interrupt	0018H

Table Q.4.1 IVT for PIC18

6.3 : Steps in Executing Interrupt

Q.5 What are the steps the microcontroller does perform upon activation of an interrupt ? ☞ [SPPU : June-22, Marks 7]

Ans. : In response to the interrupt, the microcontroller goes through the following steps :

1. It completes the execution of the current instruction and saves the address of the next instruction, i.e., the contents of the PC on the stack.

2. It also saves the current status of all the interrupts internally i.e., not on the stack.
3. It then transfers the program control to a fixed location in memory called the **Interrupt Vector Table (IVT).**
4. The microcontroller gets the address of the Interrupt Service Routine (ISR) from IVT and transfers program control to the ISR. It executes instructions within the interrupt service routine in sequence till it reaches the RETFIE (Return From Interrupt Exit) instruction.
5. Upon executing the RETFIE instruction, it gets the Program Counter (PC) address from the stack and restores the interrupt logic. Since PC is loaded with the address of the next instruction from where the interrupt was initiated, it starts to execute from that address.

6.4 : Sources of Interrupts

Q.6 Draw and explain the interrupt structure of PIC18 ?
[SPPU : Dec.-15, 16, 22, June-22, Marks 7]

Ans. : The PIC18 devices have multiple interrupt sources depending on which peripherals are incorporated in the chip. Some of the most widely used interrupt sources of PIC18 are :

- External hardware interrupts : INT0, INT1 and INT2
- Timer interrupts : TMR0IF, TMR1IF, TMR2IF and TMR3IF
- Serial communication interrupts : TXIF and RCIF
- RB Port change interrupt : RBIF
- A/D converter interrupt : ADIF
- Compare, capture, PWM interrupt : CCP1IF

- Fig. Q.6.1 shows the simplified interrupt structure of PIC18.
- Each interrupt source has three bits to control its operation. The functions of these bits are :
  - Flag bit to indicate that an interrupt event occurred.
  - Enable bit that allows program execution to branch to the interrupt vector address when the flag bit is set.

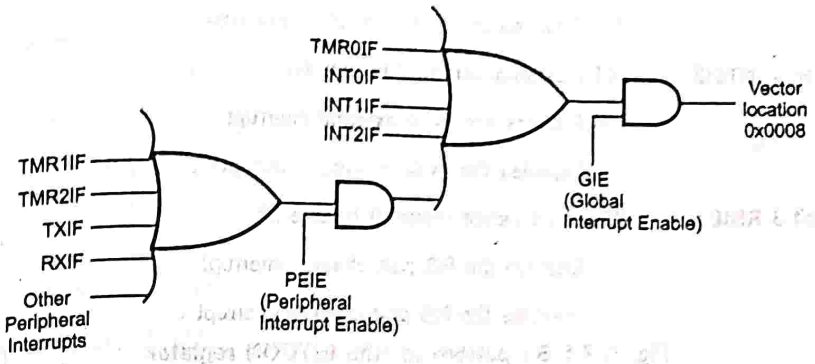


Fig. Q.6.1 Simplified interrupt structure of PIC18

- Priority bit to select high priority or low priority.

6.5 : Enabling and Disabling Interrupts

Q.7 Write a note on enabling and disabling interrupts.
[SPPU : Dec.-22, Marks 7]

Ans. : When PIC18 is reset, all interrupts are disabled. These are enabled by software. All of the bits that generate interrupts can be set or cleared by software.

- Fig. Q.7.1 shows the bit pattern for the INTCON register.

GIE	PEIE	TMR0IE	INT0IE	RBIE			
bit 7							bit 0

bit 7 **GIE :** Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 **PEIE :** Peripheral Interrupt Enable bit
1 = Enables all unmasked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **TMR0IE :** TMR0 Overflow Interrupt Enable bit

1 = Enables the TMR0 overflow interrupt

0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE :** INT0 External Interrupt Enable bit

1 = Enables the INT0 external interrupt

0 = Disables the INT0 external interrupt

bit 3 **RBIE :** RB Port Change Interrupt Enable bit

1 = Enables the RB port change interrupt

0 = Disables the RB port change interrupt

Fig. Q.7.1 Bit pattern for the INTCON register

- The PEIE bit (INTCON register) enables/disables all peripheral interrupt sources. The GIE bit (INTCON register) enables/disables all interrupt sources.

Steps in Enabling Interrupt

1. Make GIE bit (INTCON register : Bit 7) = 1 to enable all maskable interrupts.
2. Set the dedicated interrupt enable flag bit for the interrupt to be used.
3. If we want to use peripheral interrupts, enable them by making PEIE (Peripheral Interrupt Enable bit) of INTCON register high.

6.6 : Interrupt Registers

Q.8 List various registers used to control interrupt operation.

Ans. : For the PIC18F458 microcontroller, 13 registers are used to control interrupt operation. These registers are :

- RCON
- INTCON
- INTCON2
- INTCON3
- PIR1, PIR2, PIR3
- PIE1, PIE2, PIE3
- IPR1, IPR2, IPR3

Q.9 Explain INTCON registers of PIC18.

OR Illustrates the use of following bits of INTCON2 register :
I) INTEDG1 II) TMR0IP ☞ **[SPPU : June-22, Marks 4]**

Ans. : The INTCON registers are readable and writable registers that contain various enable, priority and flag bits. Because of the number of interrupts to be controlled, PIC18FXX8 devices have three INTCON registers.

INTCON Register

GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF

bit 7 bit 0

bit 7 **GIE/GIEH :** Global Interrupt Enable bit

When IPEN (RCON < 7) = 0;

1 = Enables all unmasked interrupts

0 = Disables all interrupts

When IPEN (RCON < 7 >) = 1;

1 = Enables all high priority interrupts

0 = Disables all priority interrupts

bit 6 **PEIE/GIEL :** Peripheral Interrupt Enable bit

When IPEN (RCON < 7 >) = 0;

1 = Enables all unmasked peripheral interrupts

0 = Disables all peripheral interrupts

When IPEN (RCON < 7 >) = 1;

1 = Enables all low priority peripheral interrupts

0 = Disables all low priority peripheral interrupts

bit 5 **TMR0IE :** TMR0 Overflow Interrupt Enable bit

1 = Enables the TMR0 overflow interrupt

0 = Disables the TMR0 overflow interrupt

bit 4 **INT0IE :** INT0 External Interrupt Enable bit

1 = Enables the INT0 external interrupt

0 = Disables the INT0 external interrupt

bit 3 **RBIE :** RB Port Change Interrupt Enable bit

1 = Enables the RB port change interrupt

0 = Disables the RB port change interrupt

bit 2 **TMR0IF :** TMR0 Overflow Interrupt Flag bit

1 = TMR0 register has overflowed (must be cleared in software)

0 = TMR0 register did not overflow

bit 1 **INT0IF :** INT0 External Interrupt Flag bit

1 = The INT0 external interrupt occurred (must be cleared in software by reading PORTB)

0 = The INT0 external interrupt did not occur

bit 0 **RBIF :** RB Port Change Interrupt Flag bit

1 = At least one of the RB7 : RB4 pins changed state (must be cleared in software)

0 = None of the RB7 : RB4 pins have changed state

Fig. Q.9.1 Bit pattern for INTCON register

INTCON2 Register

$\overline{\text{RBPU}}$	INTEDG0	INTEDG1	–	–	TMR0IP	–	RBIP

bit 7 bit 0

bit 7 $\overline{\text{RBPU}}$: PORTB Pull-up Enable bit

1 = All PORTB pull-ups are disabled

0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG0 :** External Interrupt 0 Edge Select bit

1 = Interrupt on rising edge

0 = Interrupt on falling edge

bit 5 **INTEDG1 :** External Interrupt 1 Edge Select bit

1 = Interrupt on rising edge

0 = Interrupt on falling edge

bit 4-3 **Unimplemented :** Read as '0'

bit 2 **TMR0IP :** TMR0 Overflow Interrupt Priority bit

1 = High priority

2 = Low priority

bit 1 **RBIP :** RB Port Change Interrupt Priority bit

1 = High priority

0 = Low priority

Fig. Q.9.2 Bit pattern for INTCON2 register

INTCON3 Register

INT2IP	INT1IP	–	INT2IE	INT1IE	–	INT2IF	INT1IF

bit 7 bit 0

bit 7 **INT2IP :** INT2 External Interrupt Priority bit

1 = High priority

0 = Low priority

bit 6 **INT1IP :** INT1 External Interrupt Priority bit

1 = High priority

0 = Low priority

bit 5 **Unimplemented :** Read as '0'

bit 4 **INT2IE :** INT2 External Interrupt Enable bit

1 = Enables the INT2 external interrupt

0 = Disables the INT2 external interrupt

bit 3 **INT1IE : INT1 External Interrupt Enable bit**

1 = Enables the INT1 external interrupt

0 = Disables the INT1 external interrupt

bit 2 **Unimplemented : Read as '0'**

bit 1 **INT2IF : INT2 External Interrupt Flag bit**

1 = The INT2 external interrupt occurred (must be cleared in software)

0 = The INT2 external interrupt did not occur

bit 0 **INT1IF : INT1 External Interrupt Flag bit**

1 = The INT1 external interrupt occurred (must be cleared in software)

0 = The INT1 external interrupt did not occur

Fig. Q.9.3 Bit pattern for INTCON3 register

Q.10 Explain PIR register of PIC18. ☞ [SPPU : June-22, Marks 8]

OR Explain RCIF and TXIF flags in programming serial communication interrupt. ☞ [SPPU : Dec.-22, Marks 4]

Ans. : The Peripheral Interrupt Request (PIR) registers contain the individual flag bits for the peripheral interrupts.

- Due to the number of peripheral interrupt sources, there are three Peripheral Interrupt Request (Flag) registers (PIR1, PIR2, PIR3).

PIR1: Peripheral Interrupt Request (Flag) Register 1

PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

bit 7 bit 0

bit 7 **PSPIF : Parallel Slave Port Read/Write Interrupt Flag bit**

1 = A read or a write operation has taken place (must be cleared in software)

0 = No read or write has occurred

bit 6 **ADIF : A/D Converter Interrupt Flag bit**

1 = An A/D conversion completed (must be cleared in software)

0 = The A/D conversion is not complete

bit 5 **RCIF : USART Receive Interrupt Flag bit**

1 = The USART receive buffer, RCREG is full (cleared when RCREG is read)

0 = The USART receive buffer is empty

bit 4 **TXIF : USART Transmit Interrupt Flag bit**

1 = The USART transmit buffer, TXREG is empty (cleared when TXREG is written)

0 = The USART transmit buffer is full

bit 3 **SPPIF : Master Synchronous Serial Port Interrupt Flag bit**

1 = The transmission/reception is complete (must be cleared in software)

0 = Waiting to transmit/receive

bit 2 **CCP1IF : CCP1 Interrupt Flag bit**

Capture mode :

1 = A TMR1 register capture occurred (must be cleared in software)

0 = No TMR1 register capture occurred

Compare mode :

1 = A TMR1 register compare match occurred (must be cleared in software)

0 = No TMR1 register compare match occurred

PWM mode :

Unused in this mode.

bit 1 **TMR2IF : TMR2 to PR2 Match Interrupt Flag bit**

1 = TMR2 to PR2 match occurred (must be cleared in software)

0 = No TMR2 to PR2 match occurred

bit 0 **TMR1IF :** TMR1 Overflow Interrupt Flag bit

1 = TMR1 register overflowed (must be cleared in software)

0 = TMR1 register did not overflow

Fig. Q.10.1 Bit pattern for Peripheral Interrupt Request (Flag) Register 1

PIR2 : Peripheral Interrupt Request (Flag) Register 2

–	CMIF	–	EEIF	BCLIF	LVDIF	TMR3IF	ECCP1IF

bit 7 bit 0

bit 7 **Unimplemented :** Read as '0'

bit 6 **CMIF :** Comparator Interrupt Flag bit

1 = Comparator input has changed

0 = Comparator input has not changed

bit 5 **Unimplemented :** Read as '0'

bit 4 **EEIF :** EEPROM Write Operation Interrupt Flag bit

1 = Write operation is complete (must be cleared in software)

0 = Write operation is not complete

bit 3 **BCLIF :** Bus Collision Interrupt Flag bit

1 = A has collision occurred (must be cleared in software)

0 = No bus collision occurred

bit 2 **LVDIF :** Low-Voltage Detect Interrupt Flag bit

1 = A low-voltage condition occurred (must be cleared in software)

0 = The device voltage is above the Low-Voltage Detect trip point

bit 1 **TMR3IF :** TMR3 Overflow Interrupt Flat bit

1 = TMR3 register overflowed (must be cleared in software)

0 = TMR3 register did not overflow

bit 0 **ECCP1IF :** ECCP1 Interrupt Flag bit

Capture mode :

1 = A TMR1 (TMR3) register capture occurred
(Must be cleared in software)

0 = No TMR1 (TMR3) register capture occurred

PWM mode :

Unused in this mode.

Fig. Q.10.2 Bit pattern for Peripheral Interrupt Request (Flag) Register 2

PIR 3 : Peripheral Interrupt Request (Flag) Register 3

IRXIF	WAKIF	ERRIF	TXB2IF	TXB1IF	TXB0IF	RXB1IF	RXB0IF

bit 7 bit 0

bit 7 **IRXIF :** Invalid Message Received Interrupt Flag bit

1 = An invalid message has occurred on the CAN bus

0 = An invalid message has not occurred on the CAN bus

bit 6 **WAKIF :** Bus Activity Wake-up Interrupt Flag bit

1 = Activity on the CAN bus has occurred

0 = Activity on the CAN bus has not occurred

bit 5 **ERRIF :** CAN bus Error Interrupt Flag bit

1 = An error has occurred in the CAN module (multiple sources)

0 = An error has not occurred in the CAN module

bit 4 **TXB2IF :** Transmit Buffer 2 Interrupt Flag bit

1 = Transmit Buffer 2 has completed transmission of a message and may be reloaded

0 = Transmit Buffer 2 has not completed transmission of a message

bit 3 **TXB1IF :** Transmit Buffer 1 Interrupt Flag bit

1 = Transmit Buffer 1 has completed transmission of a message and may be reloaded

0 = Transmit Buffer 1 has not completed transmission of a message

bit 2 **TXB0IF :** Transmit Buffer 0 Interrupt Flag bit

1 = Transmit Buffer 0 has completed transmission of a message and may be reloaded

0 = Transmit Buffer 0 has not completed transmission of a message

bit 1 **RXB1IF :** Receiver Buffer 1 Interrupt Flag bit

1 = Transmit Buffer 1 has completed transmission of a message and may be reloaded

0 = Transmit Buffer 1 has not completed transmission of a message

bit 0 **TXB1IF :** Receiver Buffer 0 Interrupt Flag bit

1 = Transmit Buffer 0 has completed transmission of a message and may be reloaded

0 = Transmit Buffer 0 has not completed transmission of a message

Fig. Q.10.3 Bit pattern for Peripheral Interrupt Request (Flag) Register 3

Q.11 Explain PIE register of PIC18.

Ans. : The Peripheral Interrupt Enable (PIE) registers contain the individual enable bits for the peripheral interrupts.

- Due to the number of peripheral interrupt sources, there are three Peripheral Interrupt Enable registers (PIE1, PIE2, PIE3).
- When IPEN is clear, the PEIE bit must be set to enable any of these peripheral interrupts.

PIE1: Peripheral Interrupt Enable Register 1

PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

bit 7 bit 0

bit 7 **PSPIE :** Parallel Slave Port Read/Write Interrupt Enable bit

1 = Enables the PSP read/write interrupt

0 = Disables the PSP read/write interrupt

bit 6 **ADIE :** A/D Converter Interrupt Enable bit

1 = Enables the A/D interrupt

0 = Disables the A/D interrupt

bit 5 **RCIE :** USART Receive Interrupt Enable bit

1 = Enables the USART receiver interrupt

0 = Disables the USART receive interrupt

bit 4 **TXIE :** USART Transmit Interrupt Enable bit

1 = Enables the USART transmit interrupt

0 = Disables the USART transmit interrupt

bit 3 **SSPIE :** Master Synchronous Serial Port Interrupt Enable bit

1 = Enables the MSSP interrupt

0 = Disables the MSSP interrupt

bit 2 **CCP1IE :** CCP1 Interrupt Enable bit

1 = Enables the CCP1 interrupt

0 = Disables the CCP1 interrupt

bit 1 **TMR2IE :** TMR2 to PR2 Match Interrupt Enable bit

1 = Enables the TMR2 to PR2 match interrupt

0 = Disables the TMR2 to PR2 match interrupt

bit 0 **TMR1IE :** TMR1 Overflow Interrupt Enable bit

1 = Enables the TMR1 overflow interrupt

0 = Disables the TMR1 overflow interrupt

Fig. Q.11.1 Bit pattern for peripheral interrupt enable register 1

PIE2 : Peripheral Interrupt Enable Register 2

–	CMIE	–	EEIE	BCLIE	LVDIE	TMR2IE	ECCP1IE

bit 7 bit 0

bit 7 **Unimplemented : Read as '0'**

bit 6 **CMIE : Comparator Interrupt Enable bit**

1 = Enables the comparator interrupt

0 = Disables the comparator interrupt

bit 5 **Unimplemented : Read as '0'**

bit 4 **EEIE : EEPROM Write Interrupt Enable bit**

1 = Enabled

0 = Disabled

bit 3 **BCLIE : Bus Colliion Interrupt Enable bit**

1 = Enabled

0 = Disabled

bit 2 **LVDIE : Low-Voltage Detect Interrupt Enable bit**

1 = Enabled

0 = Disabled

bit 1 **TMR3IE : TMR3 Overflow Interrupt Enable bit**

1 = Enables the TMR3 overflow interrupt

0 = Disables the TMR3 overflow interrupt

bit 0 **ECCP1IE : ECCP1 Interrupt Enable bit**

1 = Enables the ECCP1 interrupt

0 = Disables the ECCP1 interrupt

Fig. Q.11.2 Bit pattern for Peripheral Interrupt Enable Register 2

PIE3: Peripheral Interrupt Enable Register 3

IRXIE	WAKIE	ERRIE	TXB2IE	TXB1IE	TXB0IE	RXB1IE	RXB0IE

bit 7 bit 0

bit 7 **IRXIE : Invalid CAN Message Received Interrupt Enable bit**

1 = Enables the invalid CAN message received interrupt

0 = Disables the invalid CAN message received interrupt

bit 6 **WAKIE : Bus Activity Wake-up Interrupt Enable bit**

1 = Enables the bus activity wake-up interrupt

0 = Disables the bus activity wake-up interrupt

bit 5 **ERRIE : CAN bus Error Interrupt Enable bit**

1 = Enables the CAN bus error interrupt

0 = Disables the CAN bus error interrupt

bit 4 **TXB2IE : Transmit Buffer 2 Interrupt Enable bit**

1 = Enables the Transmit Buffer 2 interrupt

0 = Disables the Transmit Buffer 2 interrupt

bit 3 **TXB1IE : Transmit Buffer 1 Interrupt Enable bit**

1 = Enables the Transmit Buffer 1 interrupt

0 = Disables the Transmit Buffer 1 interrupt

bit 2 **TXB0IE : Transmit Buffer 0 Interrupt Enable bit**

1 = Enables the Transmit Buffer 0 interrupt

0 = Disables the Transmit Buffer 0 interrupt

bit 1 **RXB1IE : Receive Buffer 1 Interrupt Enable bit**

1 = Enables the Receive Buffer 1 interrupt

0 = Disables the Receive Buffer 1 Interrupt

bit 0 **RXB0IE : Receive Buffer 0 Interrupt Enable bit**

1 = Enables the Receive Buffer 0 Interrupt

0 = Disables the Receive Buffer 0 Interrupt

Fig. Q.11.3 Bit pattern for Peripheral Interrupt Enable Register 3

Q.12 Explain IPR register of PIC18.

Ans. : The Interrupt Priority (IPR) registers contain the individual priority bits for the peripheral interrupts. Due to the number of peripheral interrupt sources, there are three Peripheral Interrupt Priority registers (IPR1, IPR2 and IPR3).

- The operation of the priority bits requires that the Interrupt Priority Enable bit (IPEN) be set.

IPR1: Peripheral Interrupt Priority Register 1

PSPIP	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP

bit 7 bit 0

bit 7 **PSPIP :** Parallel Slave Port Read/Write Interrupt Priority bit
1 = High priority
0 = Low priority

bit 6 **ADIP :** A/D Converter Interrupt Priority bit
1 = High priority
0 = Low priority

bit 5 **RCIP :** USART Transmit Interrupt Priority bit
1 = High priority
0 = Low priority

bit 4 **TXIP :** USART Transmit Interrupt Priority bit
1 = High priority
0 = Low priority

bit 3 **SSPIP :** Master Synchronous Serial Port Interrupt Priority bit
1 = High priority
0 = Low priority

bit 2 **CCP1IP :** CCP1 Interrupt Priority bit
1 = High priority
0 = Low priority

bit 1 **TMR2IP :** TMR2 to PR2 Match Interrupt bit

1 = High priority
0 = Low priority

bit 0 **TMR1IP :** TMR1 Overflow Interrupt Priority bit
1 = High priority
0 = Low priority

Fig. Q.12.1 Bit pattern for Peripheral Interrupt Priority Register 1

IPR2 : Peripheral Interrupt Priority Register 2

–	CMIP	–	EEIP	BCLIP	LVDIP	TMR3IP	ECCP1IP

bit 7 bit 0

bit 7 **Unimplemented :** Read as '0'

bit 6 **CMIP :** Comparator Interrupt Priority bit
1 = High priority
0 = Low priority

bit 5 **Unimplemented :** Read as '0'

bit 4 **EEIP :** EEPROM Write Interrupt Priority bit
1 = High priority
0 = Low priority

bit 3 **BCLIP :** Bus Collision Interrupt Priority bit
1 = High priority
0 = Low priority

bit 2 **LVDIP :** Low-Voltage Detect Interrupt Priority bit
1 = High priority
0 = Low priority

bit 1 **TMR3IP :** TMR3 Overflow Interrupt Priority bit
1 = High priority
0 = Low priority

bit 0 **ECCP1IP :** ECCP1 Interrupt Priority bit

1 = High priority

0 = Low priority

Fig. Q.12.2 Bit pattern for Peripheral Interrupt Priority Register 2

IPR3 : Peripheral Interrupt Priority Register 3

IRXIP	WAKIP	ERRIP	TXB2IP	TXB1IP	TXB0IP	RXB1IP	RXB0IP
bit 7							bit 0

bit 7 **IRXIP** : Invalid Message Received Interrupt Priority bit

1 = High priority

0 = Low priority

bit 6 **WAKIP** : Bus Activity Wake-up Interrupt Priority bit

1 = High priority

0 = Low priority

bit 5 **ERRIP** : CAN bus Error Interrupt Priority bit

1 = High priority

0 = Low priority

bit 4 **TXB2IP** : Transmit Buffer 2 Interrupt Priority bit

1 = High priority

0 = Low priority

bit 3 **TXB1IP** : Transmit Buffer 1 Interrupt Priority bit

1 = High priority

0 = Low priority

bit 2 **TXB0IP** : Transmit Buffer 0 Interrupt Priority bit

1 = High priority

0 = Low priority

bit 1 **RXB1IP** : Receive Buffer 1 Interrupt Priority bit

1 = High priority

0 = Low priority

bit 0 **RXB0IP** : Receive Buffer 0 Interrupt Priority bit

1 = High priority

0 = Low priority

Fig. Q.12.3 Bit pattern for Peripheral Interrupt Priority Register 3

Q.13 Explain RCON register of PIC18.

Ans. : The Reset Control (RCON) register contains the IPEN bit which is used to enable prioritized interrupts.

IPEN	–	–	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

bit 7 **IPEN** : Interrupt Priority Enable bit

1 = Enable priority levels on interrupts

0 = Disable priority levels on interrupts

(PIC 16CXXX Compatibility mode)

bit 6-5 **Unimplemented** : Read as '0'

bit 4 $\overline{RI}$: RESET Instruction Flag bit

bit 3 $\overline{TO}$: Watchdog Time-out Flag bit

bit 2 $\overline{PD}$: Power-down Detection Flag bit

bit 1 $\overline{POR}$: Power-on Reset Status bit

bit 0 $\overline{BOR}$: Brown-out Reset bit

Fig. Q.13.1 Bit pattern for Reset Control (RCON) register

6.7 : Priority of Interrupts

Q.14 Write a short note on interrupt priority in PIC18.

Ans. : • There are three Peripheral Interrupt Priority registers (IPR1, IPR2 and IPR3).

- The Interrupt Priority (IPR) registers contain the individual priority bits for the peripheral interrupts. They specify the priority of a particular request either High or Low.
  - Priority Bit = 1 High Priority
  - Priority Bit = 0 Low Priority
- The high priority signals are located at program memory location 0008h and the low priority signals at 0018h.
- If the priority is disabled, by default, all the interrupts branch to 0008h memory location.
- Once the execution reaches the ISR, the interrupt is determined by polling the interrupt flags and associated code sequence is executed. After execution of code, the flag bit must be cleared in software to avoid recursion.

6.8 : Programming of Timer using Interrupts

Q.15 List the steps for programming PIC18 timer using interrupt.

Ans. : Steps for programming PIC18 timer using interrupt

1. Enable GIE, PEIE, TMRxIE.
2. Configure the TxCON register.
3. Clear TMRxIF Timer interrupt flag.
4. Load the count in Timer register TMRx
5. Set TMRxON to start the Timer operation.
6. When TMRxIF = 1, code will jump to ISR to execute it, and after execution, control returns to the main program.

Q.16 Write a C program for the PIC18 to generate a square wave of 25 Hz frequency on RB4 using Timer0 ISR. Assume XTAL = 10 MHz.

Ans. : Given : F_{OSC} = 10 MHz, No Prescaler

$$F_{TIMER} = \frac{F_{OSC}}{4} = \frac{10}{4} = 2.5 \text{ MHz}$$

$$\text{Period} = \frac{1}{F_{TIMER}} = \frac{1}{2.5 \text{ MHz}} = 0.4 \ \mu s$$

$$T = \frac{1}{25 \text{ Hz}} = 0.04 \ s$$

$$T_{ON} = T_{OFF} = \frac{T}{2} = 0.02 \ s$$

$$\text{Count} = \frac{\text{Desired Delay}}{\text{Timer Priod}} = \frac{0.02 \ s}{0.4 \ \mu s} = 50000$$

The value to be loaded in 16-bit Timer register is :

65536 – Count = 65536 – 50000 = 15536 = 3CB0H

```
#include <P18F458.h>
void Timer0_ISR(void);
void CHK_ISR (void);
#define  PORTBit  PORTBbits.RB4
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
   If (INTCONbits.TMR0IF = 1) // If Timer0 caused interrupt execute
                                 Timer0_ISR
      Timer0_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)
```

```
{
  TRISBbits.TRISB4=0;          // configure pin 4 of Port B as output
  INTCONbits.GIE = 1 ;         // Enable all interrupts globally
  INTCONbits.PEIE = 1 ;        // Enable all peripheral interrupts
  T0CON=0x08;                  // Timer0, 16 bit mode, no prescaler
  INTCONbits.TMR0IF = 0 ;      // Clear Timer0 interrupt flag bit
  TMR0H=0x3C;                  // Load Higher byte in TMR0H
  TMR0L= 0xB0;                 // Load Lower byte in TMR0L
  INTCONbits.TMR0IE = 1 ;      // Enable Timer0 interrupt
  T0CONbits.TMR0ON = 1;        // Start the Timer0
  while(1);                    // Wait for the interrupt
}
void Timer0_ISR(void)
 {
   PORTBit = ~PORTBit;         // Toggle the bit RB4
   TMR0H=0x3C;                 // Load Higher byte in TMR0H
   TMR0L= 0xB0;                // Load Lower byte in TMR0L
   INTCONbits.TMR0IF = 0 ;     // Clear Timer0 interrupt flag bit
 }
```

Q.17 Write a C program for the PIC18 to generate a square wave of 2 kHz frequency on RB4 using Timer0 ISR and a square wave of 5 kHz frequency on RB5 using Timer1 ISR. Assume XTAL = 10 MHz.

Ans. : Given : F_{OSC} = 10 MHz, No Prescaler

$$F_{TIMER} = \frac{F_{OSC}}{4} = \frac{10}{4} = 2.5 \text{ MHz}$$

$$\text{Period} = \frac{1}{F_{TIMER}} = \frac{1}{2.5 \text{ MHz}} = 0.4 \ \mu\text{s}$$

For 2 kHz

$$T = \frac{1}{2 \text{ kHz}} = 0.5 \text{ ms}$$

$$T_{ON} = T_{OFF} = \frac{T}{2} = 0.25 \text{ ms}$$

$$\text{Count} = \frac{\text{Desired Delay}}{\text{Timer Priod}} = \frac{0.25 \text{ ms}}{0.4 \ \mu\text{s}} = 625$$

The value to be loaded in 16-bit Timer0 register is :

65536 – Count = 65536 – 625 = 65911 = **FD8FH**

For 5 kHz

$$T = \frac{1}{5 \text{ kHz}} = 0.2 \text{ ms}$$

$$T_{ON} = T_{OFF} = \frac{T}{2} = 0.1 \text{ ms}$$

$$\text{Count} = \frac{\text{Desired Delay}}{\text{Timer Priod}} = \frac{0.1 \text{ ms}}{0.4 \ \mu\text{s}} = 250$$

The value to be loaded in 16-bit Timer1 register is :

65536 – Count = 65536 – 250 = 65286 = **FF06H**

```
#include <P18F458.h>
void Timer0_ISR(void);
void CHK_ISR (void);
#define  PORTBit1    PORTBbits.RB4
#define  PORTBit2    PORTBbits.RB5
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
  If (INTCONbits.TMR0IF = 1) // If Timer0 caused interrupt execute
                             Timer0_ISR
    Timer0_ISR();
  If (PIR1bits.TMR1IF = 1)   // If Timer1 caused interrupt execute
                             Timer1_ISR
    Timer1_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)
{
   TRISBbits.TRISB4=0;          // configure pin 4 of Port B as output
   TRISBbits.TRISB5=0;          // configure pin 5 of Port B as output
   INTCONbits.GIE = 1 ;         // Enable all interrupts globally
   INTCONbits.PEIE = 1 ;        // Enable all peripheral interrupts

   T0CON=0x08;                  // Timer0, 16 bit mode, no prescaler
   INTCONbits.TMR0IF = 0 ;      // Clear Timer0 interrupt flag bit
   TMR0H=0xFD;                  // Load Higher byte in TMR0H
   TMR0L= 0x8F;                 // Load Lower byte in TMR0L
   INTCONbits.TMR0IE = 1 ;      // Enable Timer0 interrupt

   T1CON=0x88;                  // Timer1, 16 bit mode, no prescaler
   PIR1bits.TMR1IF = 0 ;        // Clear Timer1 interrupt flag bit
   TMR1H=0xFF;                  // Load Higher byte in TMR1H
   TMR1L= 0x06;                 // Load Lower byte in TMR1L
   PIE1bits.TMR1IE = 1 ;        // Enable Timer1 interrupt

   T0CONbits.TMR0ON=1;          // Start the Timer0
   T1CONbits.TMR1ON=1;          // Start the Timer1
   while(1);                    // Wait for the interrupts
}
void Timer0_ISR(void)
 {
    PORTBit1 = ~PORTBit1;       // Toggle the bit RB4
    TMR0H=0xFD;                 // Load Higher byte in TMR0H
    TMR0L= 0x8F;                // Load Lower byte in TMR0L
    INTCONbits.TMR0IF = 0 ;     // Clear Timer0 interrupt flag bit
 }
void Timer1_ISR(void)
```



```
{
   PORTBit2 = ~PORTBit2;        // Toggle the bit RB5
   TMR1H=0xFF;                  // Load Higher byte in TMR1H
   TMR1L= 0x06;                 // Load Lower byte in TMR1L
   PIR1bits.TMR0IF = 0 ;        // Clear Timer0 interrupt flag bit
}
```

6.9 : Programming of External Hardware Interrupts

Q.18 List the steps for programming PIC18 external interrupts.

Ans. : Steps for programming PIC18 external interrupts

1. Set PORTB External interrupt pin as an input.
2. Also, make PORTB pins as digital input as it is multiplexed with ADC channels. This is done by disabling the ADON bit in the ADCON1 register or by making the PBADEN configuration a bit low.
3. Configure INTCON2 register for edge trigger i.e. positive (rising) or negative (falling) edge.
4. Enable external interrupt (INT0, INT1, INT2) by setting a respective interrupt to enable bit in the INTCON register.
5. Enable Global Interrupt (GIE).

Q.19 Write a C program for the PIC18 toggle the LED connected to pin 0 of the PORT C every time INT0 is activated by a switch connected at INT0(RB0). Assume XTAL = 10 MHz.

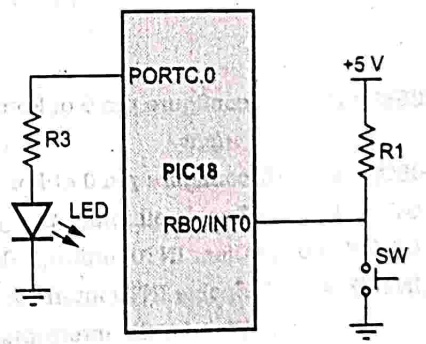


Fig. Q.19.1

Ans. :

```
#include <P18F458.h>
void INT0_ISR(void);
void CHK_ISR (void);
#define  PORTBit  PORTCbits.RC0
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
   If (INTCONbits.INT0IF = 1)  // If INT 0 caused interrupt execute
                               INT0_ISR

      INT0_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)
{
   TRISBbits.TRISB0=1;      // configure pin 0 of Port B (INT0) as
                               input
   TRISBbits.TRISC0=0;      // configure pin 0 of Port C as output
   INTCONbits.GIE = 1 ;     // Enable all interrupts globally
   INTCONbits.INT0IF = 0 ; // Clear INT0 interrupt flag bit
   INTCONbits.INT0IE = 1 ; // Enable INT0 interrupt
   while(1);                // Wait for the interrupts
}
```



```
void INT0_ISR(void)
{
    PORTBit = ~PORTBit;     // Toggle the bit 0 of PORT C
    INTCONbits.INT0IF = 0 ; // Clear Timer0 interrupt flag bit
}
```

6.10 : Programming of Serial Communication Interrupt

Q.20 List the steps for programming PIC18 serial communication using interrupt.

Ans. : Steps for programming PIC18 USART using interrupt

1. Initialize the baud rate by loading value to the SPBRG register.
2. Set bit SPEN in the RCSTA for serial port enable.
3. Set bit BRGH in the TXSTA for low or high speed.
4. Clear bit SYNC in the TXSTA registers for asynchronous communication.
5. Set bit TXEN in the TXSTA register to enable transmission.
6. Set bit CREN in the RCSTA register to enable reception.
7. Clear RCIF and TXIF
8. Enable GIE, PEIE, RCIE and TXIE for ISR.

Q.21 Write a C program for the PIC18 to read data from PORT D and transfer it serially at 9600 baud, continuously. Use 8-bit data and 1-stop bit.
Assume XTAL = 10 MHz.

Ans. :

```
#include <P18F458.h>
void Transmit_ISR(void);
void CHK_ISR (void);
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
   If (PIR1bits.TXIF = 1)      // If transmit interrupt occurs execute
                               // Transmit_ISR

      Transmit_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)
{
      TRISD = 0xFFH          // Configure PORT D as input
      TXSTA = 0x20;          // Asynchronous mode with 8-bit data
                             // frame, low baud rate and transmit
                             // enable : TX9 = 0,
                             // TXEN = 1, SYNC  = 0,
                             // BRGH = 0
    SPBRG = 15;              // 9600 baud rate
  TRISBbits.TRISC6=0;        // configure pin 6 of Port C
                             //  (Transmitter pin) as output
  PIR1bits. TXIF = 0 ;       // Clear transmit interrupt flag bit
  RCSTAbits.SPEN=1;          // Enable serial port
  PIE1bits.TXIE = 1          // Enable transmit interrupt
  INTCONbits.GIE = 1 ;       // Enable all interrupts globally
  INTCONbits.PEIE = 1 ;      // Enable all peripheral interrupts
  while(1);                  // Wait for the interrupts
}
void Transmit_ISR(void);
```



```
{
  TXREG = PORTD ;          // Transfer the character read from
                           // PORTD to TXREG register

}
```

Q.22 Write a C program for the PIC18 to read data from PORT D and transfer it serially at 9600 baud, continuously and simultaneously send incoming data from the serial port to PORT B. Use 8-bit data and 1-stop bit. Assume XTAL = 10 MHz.

Ans. :

```
#include <P18F458.h>
void Transmit_ISR(void);
void Receive_ISR(void);
void CHK_ISR (void);
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
  If (PIR1bits.TXIF = 1)          // If transmit interrupt occurs
                                  // execute Transmit_ISR
    Transmit_ISR();
  If (PIR1bits.RCIF = 1)          // If recieve interrupt occurs
                                  // execute Receive_ISR
    Receive_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)
```

```
{
    TRISD = 0xFFH     // Configure PORT D as input
    TRISB = 0x00H     // Configure PORT B as output

TRISBbits.TRISC6=0; // configure pin 6 of Port C (Transmitter pin)
                      as output
TRISBbits.TRISC7=1; // configure pin 7 of Port C (receiver pin) as
                      input
    TXSTA = 0x20;     // Asynchronous mode with 8-bit data frame,
                      // low baud rate
                      // and transmit enable : TX9 = 0, TXEN = 1,
                        SYNC  = 0, BRGH = 0
    SPBRG = 15;       // 9600 baud rate
    RCSTA = 0x90;     // Enable serial port and Continuous
                      // 8-bit reception
                      //  SPEN =1, RX9 = 0, CREN = 1
PIR1bits. TXIF = 0 ;  // Clear transmit interrupt flag bit
PIR1bits. RCIF = 0 ;  // Clear receive interrupt flag bit

PIE1bits.TXIE = 1     // Enable transmit interrupt
PIE1bits.RCIE = 1     // Enable receive interrupt
INTCONbits.GIE = 1 ; // Enable all interrupts globally
INTCONbits.PEIE = 1 ;// Enable all peripheral interrupts
while(1);             // Wait for the interrupts
}
  void Transmit_ISR(void);
{
  TXREG = PORTD ;     // Transfer the character read
                        from PORTD to TXREG register
}
void Receive_ISR(void);
{
  PORTD = RCREG ;       // Send received data to PORT B
}
```

END...

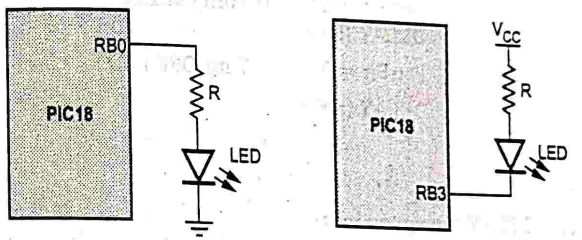
Unit III

7 Interfacing of LED, LCD, Keyboard, Relay and Buffer

7.1 : Interfacing of LED

Q.1 With a diagram explain how LED can be interfaced to PIC 18 microcontroller.

Ans. : Since the current sinking and sourcing capacities in the PIC18 family is about 25 mA, we can drive LED using PORT pins in two ways, as shown in Fig. Q.1.1.



(a) LED connected in current source mode (b) LED connected in current sink mode

Fig. Q.1.1

- Current sourcing mode
- Current sinking mode

For current source mode $R = \frac{V_{out} - V_{LED}}{I_{LED}}$

For current sink mode $R = \frac{V_{CC} - V_{LED}}{I_{LED}}$

where V_{LED} is the voltage across LED

I_{LED} is the current through LED

V_{out} is the voltage at output pin

• Since I_{LED} is the range of 10-15 mA, V_{LED} = 1.5 V, V_{CC} = 5 V, V_{out} = 5 V, suitable value for R is in the range of 270 Ω - 330 Ω.

Q.2 Write an embedded PIC 18 C program to flash the LED connected to pin 0 of PORTB, as shown in Fig. Q.1.1.

Ans. : Assume that T_{ON} and T_{OFF} of LED = 500 ms.

```
# include < PIC18F458.h >
#define PortBit PORTBbits.RB0      // Declare single bit
void DELAY (unsigned int);
void main (void)
{
        TRISBbits.TRISB0=0;  // Configure pin 0 of Port B as output

        while(1)
        {
            PortBit = 1;      // Turn ON LED
            DELAY (500);
            PortBit = 0;      // Turn OFF LED
            DELAY (500);

        }
}
void DELAY (unsigned int t);
{
   unsigned int i, j;
   for (i = 0; i<t; i++)
      for (j = 0; j<165; j++);
}
```

Q.3 Write PIC18 C program to blink LEDs connected to Port C of PIC18. ☞ [SPPU : Dec.-12, May-13, Marks 8]

Ans. : Assume that T_{ON} and T_{OFF} of LED = 500 ms.

```
# include < PIC18F458.h >
void DELAY (unsigned int);
void main (void)
{
        TRISC= 0    ;   // Make Port C as an output
    while(1)
    {
        PORTC = 0x00;
        DELAY (500);
        PORTC = 0xFF;
        DELAY (500);
    }
}
void DELAY (unsigned int t);
{
   unsigned int i, j;
   for (i = 0; i<t; i++)
      for (j = 0; j<165; j++);
}
```

Q.4 An LED is connected to each pin of PORT D. Write a PIC18 C program that will turn each LED from pin D0 to D7. Call a delay module before turning on the next LED. ☞ [SPPU : Dec.-14, Marks 8]

Ans. :

```
# include < PIC18F458.h >
#define PD0 PORTDbits.RD0
#define PD1 PORTDbits.RD1
#define PD2 PORTDbits.RD2
#define PD3 PORTDbits.RD3
#define PD4 PORTDbits.RD4
#define PD5 PORTDbits.RD5
#define PD6 PORTDbits.RD6
#define PD7 PORTDbits.RD7
void DELAY (unsigned int);
void main (void)
```

```
{
    TRISD  = 0;          // configure Port D as output
    while(1)
    {
       PD0 = 1;          // Turn ON LED connected to RD0
       DELAY (500);
       PD1 = 1;          // Turn ON LED connected to RD1
       DELAY (500);
       PD2 = 1;          // Turn ON LED connected to RD2
       DELAY (500);
       PD3 = 1;          // Turn ON LED connected to RD3
       DELAY (500);
       PD4 = 1;          // Turn ON LED connected to RD4
       DELAY (500);
       PD5 = 1;          // Turn ON LED connected to RD5
       DELAY (500);
       PD6 = 1;          // Turn ON LED connected to RD6
       DELAY (500);
       PD7 = 1;          // Turn ON LED connected to RD7
       DELAY (500);
       PORTD = 0;        // Turn OFF all LEDs
    }
}
void DELAY (unsigned int t);
{
   unsigned int i, j;
   for (i = 0; i<t; i++)
      for (j = 0; j<165; j++);
}
```


7.2 : Interfacing of LCD

Q.5 Explain the pin description of LCD module.

OR Explain the function of following LCD pins :
i) RS ii) RW iii) EN

☞ [SPPU : June-22, Marks 3]

Ans. :

Pin	Symbol	I/O	Description
1	V_{SS}	-	Ground
2	V_{CC}	-	+ 5 V power supply
3	V_{EE}	-	Used for controlling LCD contrast
4	RS	I	Register select input is used to select either of the two available registers in the module : Data register or command register. When RS = 0 Data register is selected. When RS = 1 Command register is selected.
5	R/$\overline{W}$	I	Allows the user to write information to the LCD or read information from it. For reading R/$\overline{W}$ = 1 and for writing R/$\overline{W}$ = 0
6	E	I	This pin is used by LCD to latch information available at its data pins.
7-14	DB_0-DB_7	I/O	This 8-bit data bus is used to send information to the LCD or read the contents of the internal registers of the LCD.

Table Q.5.1 Pin description for LCD module

Q.6 Draw and explain the interfacing of LCD module with PIC18.

☞ [SPPU : June-22, Marks 4, Dec.-22, Marks 8]

Ans. : Fig. Q.6.1 shows the interfacing of a 20 character × 2 line LCD module with the PIC 18. As shown in the Fig. Q.6.1, the data lines are connected to the PORTC of PIC 18 and control lines RS, R/$\overline{W}$ and E are driven by RB0, RB1 and RB2 lines of PORTB, respectively. The voltage at V_{EE} pin is adjusted by a potentiometer to adjust the contrast of the LCD.

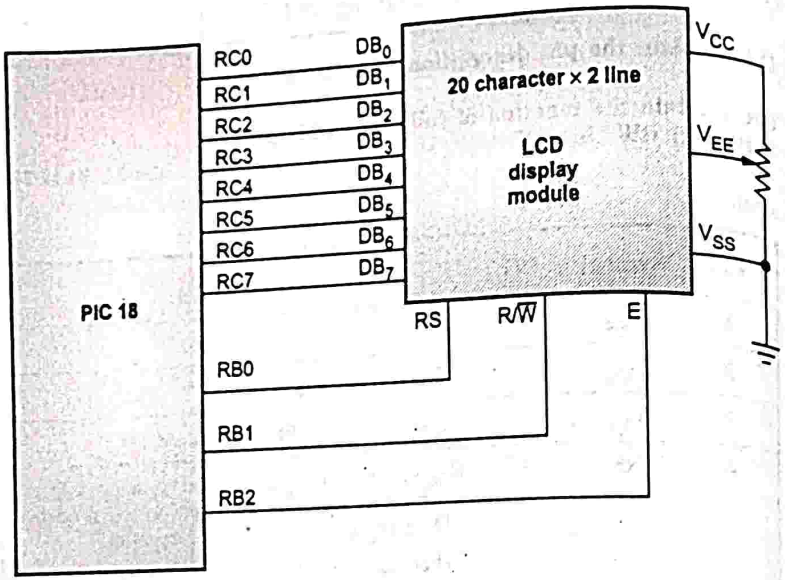


Fig. Q.6.1 Interfacing LCD module with PIC 18

Q.7 List the commands available for LCD module.

Ans. :

• Table Q.7.1 lists the command available for LCD module.

Command	Command code										Description
	RS	R/W	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	
Clear display	0	0	0	0	0	0	0	0	0	1	Sets DD RAM address to 0 and clears entire display.
Return home	0	0	0	0	0	0	0	0	1	-	Sets DD RAM address to 0 and returns the cursor to the home position. DD RAM contents remain unchanged.

Entry mode set	0	0	0	0	0	0	0	1	1/D	S	Sets cursor move direction and specifies or not to shift the display. These operations are performed during data write and read.
Display ON/OFF control	0	0	0	0	0	0	1	D	C	B	Sets ON/OFF of entire display (D), cursor ON/OFF (C), and blink of cursor position character (B).
Cursor and display shift	0	0	0	0	0	1	S/C	R/L	-	-	Moves the cursor and shifts the display without changing DD RAM contents.
Function set	0	0	0	0	1	DL	N	F	-	-	Sets interface data length (DL), number of display lines (N), and character font (F).
Set CG RAM address	0	0	0	1	ACG						Sets CG RAM address. CG RAM is accessed after this setting.
Set DD RAM address	0	0	1	ADD							Sets DD RAM address. DD RAM is accessed after this setting
Read busy flag and address	0	1	BF	AC							Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents
Write data to CG or DD RAM	1	0	Write data								Writes data into DD or CG RAM.
Read data from CG or DD RAM	1	1	Read data								Reads data from DD or CG RAM.

Table Q.7.1 List of LCD module commands

Q.8 Draw and explain the interfacing of LCD with PORT D and PORT E of PIC 18xxx microcontroller. Write C code to display 'WELCOME'. ☞ [SPPU : May-15, June-22, Dec.-22, Marks 8]

Ans. : Fig. Q.8.1 shows the interfacing of a 16 character × 2 line LCD module with the PIC 18. As shown in Fig. Q.8.1, the data lines are connected to the PORT D of PIC18 microcontroller and control lines RS, R/$\overline{W}$ and E are driven by RE0, RE1 and RE2 lines of PORT E, respectively. The voltage at V_{EE} pin is adjusted by a potentiometer to adjust the contrast of the LCD.

Program

```
# include < PIC18F458.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(unsigned char, unsigned char);
void DELAY (unsigned int);
void main (void)
{
   TRISD= 0;                   // Make Port D as an output
   TRISE= 0;                   // Make Port E as an output
   EN = 0;                     // Make enable low
   LCD(0x38, 0);               // Initialize LCD 2 lines, 5 × 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);
   LCD (0x86, 0);              // Line 1, position 6
   DELAY (15);
   LCD ('W', 1);               // Display character W
   DELAY (15);
```

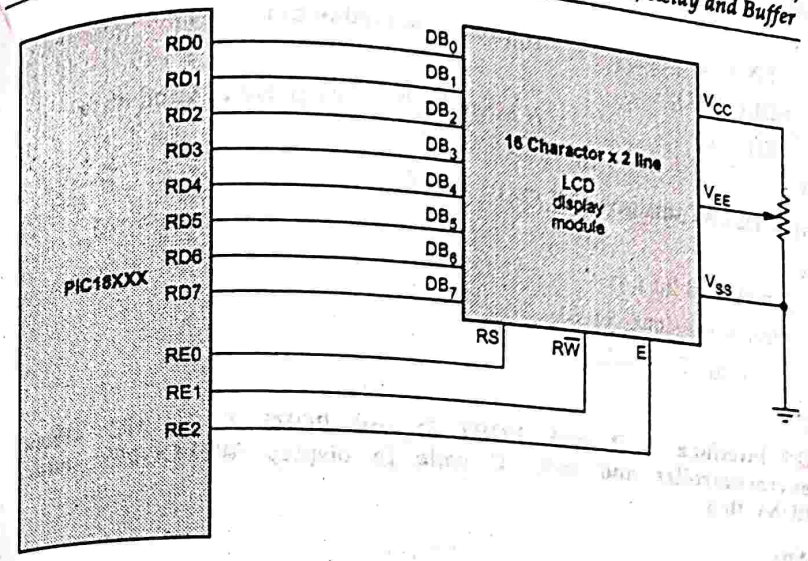



Fig. Q.8.1

```
   LCD ('E', 1);               // Display character E
   DELAY(15);
   LCD ('L', 1);               // Display character L
   DELAY (15);
   LCD ('C', 1);               // Display character C
   DELAY (15);
   LCD ('O', 1);               // Display character O
   DELAY(15);
   LCD ('M', 1);               // Display character M
   DELAY(15);
   LCD ('E', 1);               // Display character E
}
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 DELAY(unsigned int cnt)
{
    unsigned int i, j;
    for(i=0; i<cnt; i++)
        for(j=0; j<165; j++);
}
```

Q.9 Interface LCD with PORT D and PORT E of PIC 18xxx microcontroller and write C code to display 'WELCOME' using BUSY flag.

Ans. :

Program

```
# include < PIC18F458.h >
#define LCDData PORTD          // LCD data pins
#define RS PORTEbits.RE0       // RS pin of LCD
#define RW PORTEbits.RE1       // RW pin of LCD
#define EN PORTEbits.RE2       // EN pin of LCD
#define BUSY PORTDbits.RD7     // BUSY pin of LCD
void LCD(unsigned char, unsigned char);
void DELAY (unsigned int);
void READY ();

void main (void)
{
    TRISD= 0;               // Make port D as an output
    TRISE= 0;               // Make port E as an output
    EN = 0;                 // Make enable low
    DELAY (250);
    LCD(0x38, 0);           // Initialize LCD 2 lines, 5 × 7 matrix
    DELAY (250);
    LCD (0x0E, 0);          // Display on, cursor on
```



```
    READY (15);
    LCD (0x01, 0);          // Clear LCD
    READY (15);
    LCD (0x06, 0);          // Shift cursor right
    READY (15);
    LCD (0x86, 0);          // Line 1, position 6
    READY (15);
    LCD ('W', 1);           // Display character W
    READY (15);
    LCD ('E', 1);           // Display character E
    READY (15);
    LCD ('L', 1);           // Display character L
    READY (15);
    LCD ('C', 1);           // Display character C
    READY (15);
    LCD ('O', 1);           // Display character O
    READY (15);
    LCD ('M', 1);           // Display character M
    READY (15);
    LCD ('E', 1);           // Display character E
}
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 READY ()
{
    TRISD= 0xFF;            // Make port D as an input
```

```
    RS = 0;                    // RS = 0 for LCD command
    RW = 1;
    do
    {
      EN = 1;                  // Strobe the enable pin
      DELAY(1);
      EN = 0;                  // Make high to low pulse to latch data
    } while (BUSY == 1);
    TRISD= 0;                  // Make port D as an output
}

void DELAY(unsigned int cnt)
{
    unsigned int i, j;
    for(i=0; i<cnt; i++)
      for(j=0; j<165; j++);
}
```

7.3 : Interfacing Key Board (4 × 4 Matrix)

Q.10 For 4 × 4 matrix keyboard, write an embedded PIC18 C program to read keypad and send the keycode to PORT D.

OR Draw the interfacing diagram for 4×4 matrix keyboard with PIC18 and explain. ☞ **[SPPU : June-22, Marks 6]**

Ans. :

- Fig. Q.10.1 shows the 4 × 4 matrix keyboard connected to the PORT B of PIC18. 4 lines of port B (RB0 - RB3) are used as a scan lines and remaining 4 lines (RB7 - RB4) are used as return lines.

The steps in algorithm are as follows :

1. Initialise RB4 - RB7 as inputs i.e. write '1' to these pins.
2. Check if all the keys are released by writing '0' to RB3 - RB0 and check if all return lines are in state '1'. If No then wait.

   If Yes then go to step 3.

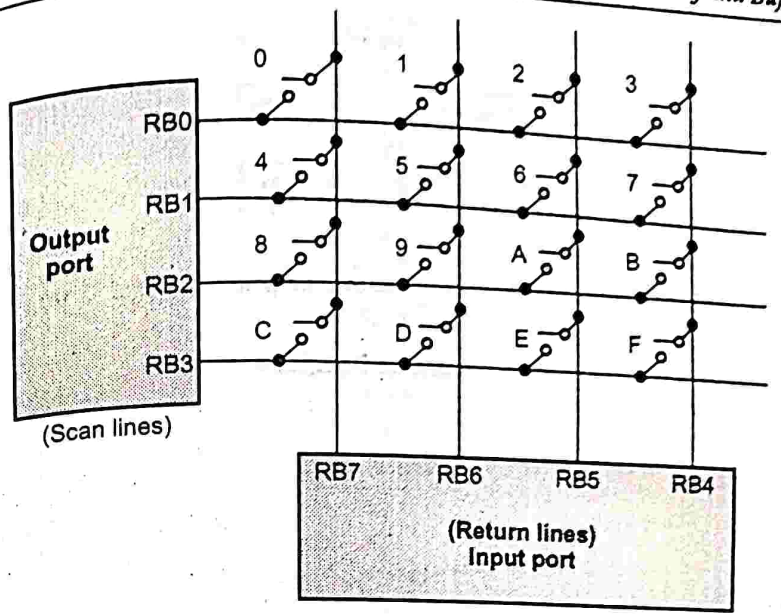


Fig. Q.10.1 4 × 4 matrix keyboard connected to PORT B of PIC 18

3. Call debounce.
4. Wait for key closure. Ground all scan lines by writing '0' and then check if at least one of return lines shows '0' level.

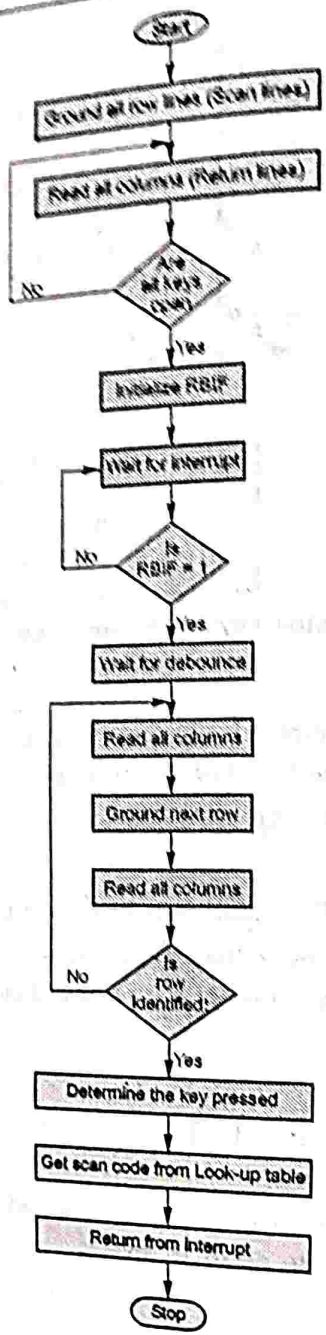
   Key pressed ? No step 4

   Yes step 5
5. Call debounce. (allow sufficient time for debounce)
6. Is key really pressed ? (Ground all scan lines by writing '0' and then check if at least one of the return lines shows '0' level.)

   No step 4

   Yes step 7
7. Find key code and display the key pressed on 7-segment display.

   (By grounding one scan line at a time and checking return lines for any one line to go to '0' level.)
8. Go to step 1.



Program

```
# include < PIC18F458.h >
void RBIF_ISR(void);
void DELAY(unsigned int);
unsigned char KeyCode [4] [4] = { '0', '1', '2', '3',
                                  '4', '5', '6', '7',
                                  '8', '9', 'A', 'B',
                                  'C', 'D', 'E', 'F',};
#pragma interrupt CHK_ISR
void CHK_ISR (void)
{
   If (INTCONbits.RBIF = 1)     // If RBIF caused interrupt execute
                                // RBIF_ISR
      RBIF_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)
{     TRISD = 0;              // Make PORT D an output Port
      INTCONbits.RBPU         // Enable PORT B pull-up resistors
      TRISB= 0XF0;            // Make RB7-RB4 as inputs pins and
                              // RB3-RB0 as output pins
```

```
    PORTB = 0xF0;           // Make all row (scan) lines low
    while (PORTB != 0xF0);  // Check if all Keys are open,
                            // otherwise wait
    INTCONbits.RBIE = 1;    // Enable PORT B change interrupt
    INTCONbits.GIE = 1;     // Enable interrupts globally
    while(1);               // Wait for interrupt
}
void RBIF_ISR(void)
{  unsigned char temp, COL = 0, ROW = 4;
   DELAY (10);              // wait for 10 ms (debounce delay)
   temp = PORT B;           // Read PORT B - column lines
                            // If we assume key1 is pressed, we get
                            // temp = 1011 0000
   temp ^= 0xF0;            // Invert higher nibble, we get temp
                            // = 0100 0000
   while (temp <<=1)        // Contents of temp are shifted left by 1,
                            // thus
                            // temp = 1000 0000
   {
      COL++;                // Since while is true COL = COL + 1
   }                        // When temp is shifted left again is
                            // becomes zero and while loop
                            // is terminated
                            // Thus when key1 is pressed we get
                            // COL = 1
   PORTB = 0XFE;            // Make  Row0 = 0
   if (PORTB  != 0XFE)      // In our case(key 1)
                            // we get PORTB = 1011 1110 = 0xBE
      ROW = 0;              // Row0 detected
   else
   {  PORTB = 0XFD;                 // Make  Row1 = 0
      if (PORTB  != 0XFD)
         ROW = 1;                   // Row1 detected
      else
      {     PORTB = 0XFB;           // Make  Row2 = 0
            if (PORTB  != 0XFB)
               ROW = 2;             // Row2 detected
```



```
            else
            {     PORTB = 0XF7; // Make  Row3 = 0
                  if (PORTB  != 0XF7)
                     ROW = 3;   // Row3 detected
            }
      }
}
If (ROW < 4)
   PORTD = KeyCode [ROW] [COL];  // Send detected key
                                 // to PORT D
                                 // In our case(key1)
                                 // PORTD = KeyCode[0][1] = '1'
while (PORTB != 0xF0)
      PORTB = 0XF0;              // Wait for key release
INTCONbits.RBIF = 0;             // Clear RBIF interrupt flag
}
void DELAY(unsigned int cnt)
{
   unsigned int i, j;
   for(i=0;  i<cnt;  i++)
      for(j=0; j<165; j++);
}
```

7.4 : Interfacing Relay

Q.11 Explain the interfacing of relay to PIC18.

Ans. : To control ON/OFF operation of ac devices we can use electromagnetic relay. This relay has both normally open and closed contacts. When a current is passed through the coil of the relay, the switch arm is pulled down, opening the top contact and closing the bottom contact, as shown in Fig. Q.11.1.

- The relay contacts are rated for maximum current of about 20 A - 25 A. These relays are also called mechanical relays because mechanical contact makes the circuit to ON or OFF. The mechanical relays having higher current ratings are sometimes called **contactors.**

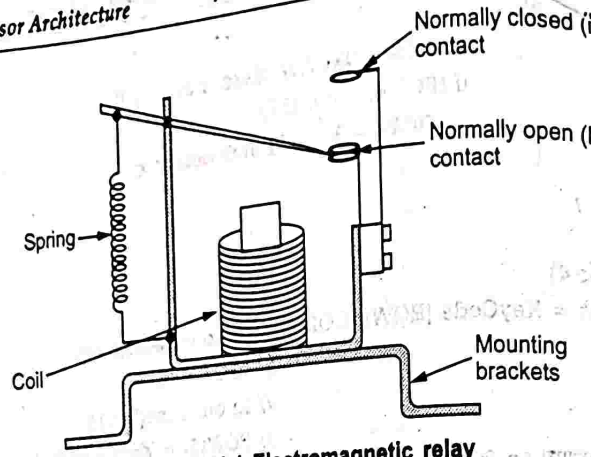
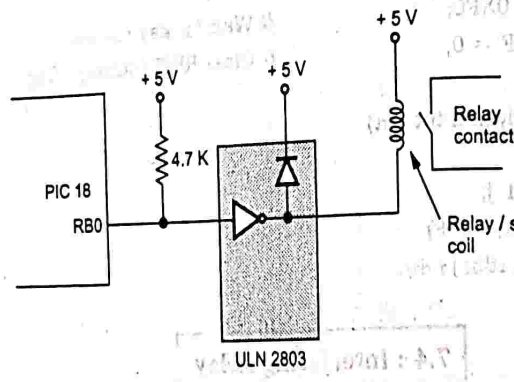
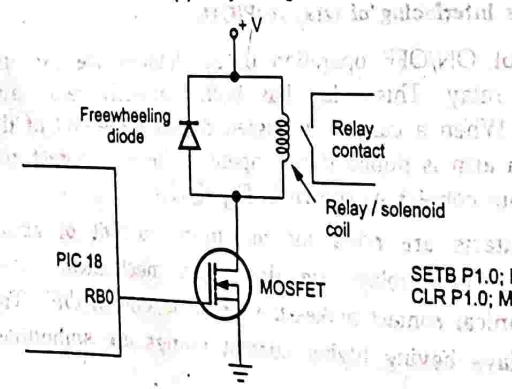


Fig. Q.11.1 Electromagnetic relay



(a) Relay driving circuit



(b) Relay driving circuit using MOSEFT

Fig. Q.11.2

- The output current of PIC18 microcontroller pins is not sufficient to drive the relay, or solenoid. The relay/solenoid coil needs around 50 mA - 100 mA to be energized and microcontroller's pin can provide this current. For this reason, we have to use driver such as the ULN2803 or a power transistor or a power MOSFET between the microcontroller and the relay/solenoid as shown in the Fig. Q.11.2.

Q.12 Write PIC18 C program to make relay shown in Fig. Q.11.2 (a) ON and make it OFF after 2 seconds.

Ans. :

```
#include < PIC18F458.h >
#define PORTBIT PORTBbits.RB0   // Single bit declaration
void main(void)
{
   unsigned int i;
   TRISBbits.TRISB0= 0;        // Make Port RB0 as an output
   PORTBIT = 1;                // Make RB0 = 1 and hence Relay ON
   DELAY (2000);               // Wait for 2 seconds
   PORTBIT = 0;                // Make RB0 = 0 and hence Relay OFF
   }
}
void DELAY(unsigned int cnt)
{
   unsigned int i, j;
   for(i=0; i<cnt; i++)
      for(j=0; j<165; j++);
}
```

7.5 : Interfacing Buzzer

Q.13 Explain the interfacing of buzzer to PIC18.

Ans. : Buzzers are mainly divided into two types -

- Active buzzers and
- Passive buzzers

- Active buzzer produces a single type of sound or tone. It requires a dc power source for generating the sound or beep.
- On the other hand, passive buzzer can generate different sounds or beep. It depends upon the frequency which is provided to the buzzer. It required an ac power source for generating the sound or beep.

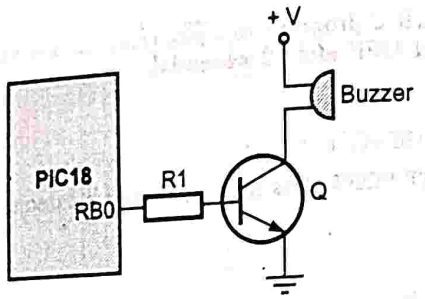


Fig. Q.13.1 Interfacing of a simple active buzzer with PIC18 microcontroller

- Fig. Q.13.1 shows the interfacing of a simple active buzzer with PIC18 microcontroller.
- Usually, buzzer takes more current and hence it could not possibly directly driven by microcontroller pin. In this condition, an NPN or PNP transistor is added in the circuit for safely turning ON or OFF the buzzer. This is illustrated in Fig. Q.13.1.
- The embedded C program given in Q.12 can be used to turn ON and turn OFF the buzzer shown in Fig. Q.13.1

END...

Unit IV

8 CCP Modules

8.1 : CCP Modules

Q.1 State various CCP modes and associated timers.

Ans. : Different timers are associated with the CCP module depending on the mode used. Table Q.1.1 shows the PIC18 timers for different CCP modes.

CCP Modes	Timer
Capture	1 or 3
Compare	1 or 3
PWM	2

Table Q.1.1

8.2 : CCP Registers

Q.2 List CCP registers available in CCP module.

Ans. : Each CCP module has three registers :

1. CCPxCON (8-bit) : CCP control register
2. CCPRxH (8-bit) : CCPR high register
3. CCPRxL (8-bit) : CCPR low register.

Q.3 Draw and explain the bit pattern of CCP control register for CCP1 module.

Ans. : Fig. Q.3.1 shows the CCP control register for CCP1 module.

-	-	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0

bit 7 bit 0

bit 7 - 6 Unimplemented : Read as '0'

bit 5 - 4 DCxB1 : DCxB0 : PWM Cycle bit 1 and bit 0

Capture mode :

Unused.

Compare mode :

Unused

PWM code :

These bits are the two LSbs (bit 1 and bit 0) of the 10-bit PWM duty cycle.
The upper eight bits (DCxp : DCx2) of the duty cycle are found in CCPRxL

bit 3 - 0 **CCPxM3 : CCPxM0 : CCPx Mode Select bits**

0000 = Capture /Compare /PWM off (resets CCPx module)

0001 = Reversed

0010 = Compare mode, toggle output on match (CCPxIF) bit is set)

0011 = Compare mode, CAN message received (CCP1 only)

0100 = Compare mode, every falling edge

0101 = Compare mode, every rising edge

0110 = Capture mode, every 4th rising edge

0111 = Capture mode, every 16th rising edge

1000 = Compare mode, initialize CCP.pin low, on compare match force CCP pin high (CCPIF bit is set)

1001 = Compare mode, initialize CCP pin high, on compare match force CCP pin low(CCPIF bit is set)

1010 = Compare mode, CCP pin is unaffected (CCPIF bit is set)

1011 = Compare mode, trigger event (CCP1IF bit is set ; CCP resetsTMR1 or TMR3 and starts an A/D conversion
if the A/D module is enabled)
11XX = PWM mode

Fig. Q.3.1 CCP1CON : CCP1 control register

Q.4 What do you mean by duty cycle register.

Ans. : • In PWM mode, 10-bit register is used to decide the duty cycle. This 10-bit is formed using :

- **(Bit 1 and Bit 0)** lower 2-bits : **DCxB1 : DCxB0** : from CCPx Control Register
- **(Bit 9 - Bit 2)** upper 8-bits : **CCPRxL register** : Bit 7 - Bit 0

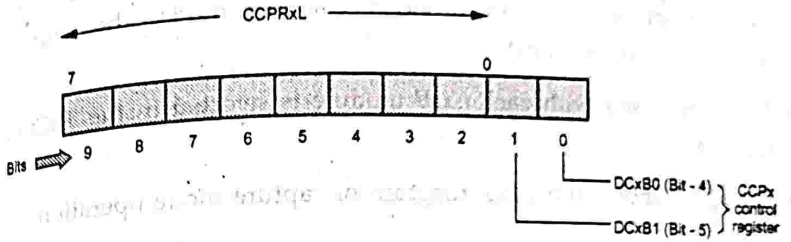


Fig. Q.4.1 PWM 10-bit duty cycle register

8.3 : Capture Mode

Q.5 Explain the capture mode of CCP module.

Ans. : This mode provides access to the current state of that timer TMR1 or TMR3 register.

- When capture mode with the bit selection in the CCP1CON register is selected, CCPR1H : CCPR1L captures the 16-bit value of the TMR1 or TMR3 register when an event occurs on pin RC2/CCP1.
- The combination of the four bits (CCP1M3 - CCP1M0) of the control register determines the event :
  - CCP1M3-CCP1M0 : 0100 : Every falling edge
  - CCP1M3-CCP1M0 : 0101 : Every rising edge
  - CCP1M3-CCP1M0 : 0110 : Every 4th rising edge
  - CCP1M3-CCP1M0 : 0111 : Every 16th rising edge
- When a capture is made, the interrupt request flag bit, CCP1IF (PIR registers), is set. It must be cleared in software.

- If another capture occurs before the value in register CCPR1 is read, the old captured value will be lost.
- In capture mode, the RC2/CCP1 pin should be configured as an input.
- The timers used with the capture feature (either Timer1 and/or Timer3) must be running in Timer mode or Synchronized Counter mode. In Asynchronous Counter mode, the capture operation may not work.
- The timer used with each CCP module is selected in the T3CON register.
- Fig. Q.5.1 shows the block diagram of capture mode operation.

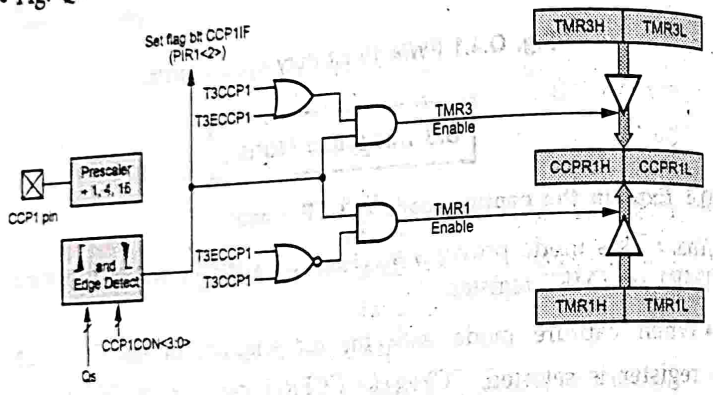


Fig. Q.5.1 Block diagram of capture mode operation

Q.6 List the steps in programming PIC18F in capture mode.
☞ [SPPU : June-22, Dec.-22, Marks 6]

Ans. : Initialization of PIC18F458 Capture Mode

1. Initialize the CCPCON1 register for capturing the rising or falling edge.
2. Configure the CCP1 pin as an input pin.
3. Configure T3CON register to select either Timer0 or Timer3.
4. Initialize TxCON register to either configure Timer1 or Timer3 depending on step 3.

5. Set CCPR1 and TMR3/TMR1 register to 0.
6. Enable CCP1 interrupt enable flag : CCP1IE = 1.
7. Clear CCP1 interrupt flag : CCP1IF = 0.
8. Wait for interrupt flag. If it is set, read captured value.

Q.7 The input signal is connected to the CCP1 (RC2) pin. Write PIC18F C program to measure the period of the input signal and send it on PORTB and PORTD.

Ans. : This program starts timer1 from value 0 after detecting 1st rising edge and send captured count after detecting 2nd rising edge on Port B and Port D.

```
#include <p18f458.h>
void main (void)
{
  CCP1CON=0x05;          // Capture mode is selected for detecting
                         //  rising edge
  T3CON = 0x0;           // Select Timer1 for capture
  TRISB = 0;             // Configure Port B as an output
  TRISC = 0;             // Configure Port C as an output
  TRISCbits.TRISC2=1;    // Configure the CCP1 pin as an input pin
  CCPR1=0x00;            // Clear capture count register
  T1CON=0x00;            // Enable TMR1 Register, No pre-scale,
                         //  use internal clock, Timer OFF
  PIE1bits.CCP1IE=1;     // Enable CCP1 interrupt Enable flag
  while(1)
  {
    TMR1H = 0;                        // Clear Timer1 register
    TMR1L = 0;
    PIR1bits.CCP1IF=0;                // Clear CCP1 interrupt flag
    while(PIR1bits.CCP1IF == 0);      // Wait for 1st rising edge
    T1CONBits.TMR1ON = 1;             // Turn-On Timer1
    PIR1bits.CCP1IF=0;                // Clear CCP1 interrupt flag
                                      // for the next edge
    while(!(PIR1bits.CCP1IF));        // Wait for 2nd rising edge
    T1CONBits.TMR1ON = 0;             // Stop Timer1
```

```
    PORTB = CCPR1L ;        // Send lower byte of period to Port B
    PORTD = CCPR1H ;        // Send higher byte of period to Port D
}
```

8.4 : Compare Mode

Q.8 Explain the compare mode of CCP module. ☞ [SPPU : Dec.-22, Marks 6]

Ans. : CCP in compare mode is used to generate a waveform of various duty cycles like PWM and also used to trigger an event when the pre-determined time expires. It is also used to generate specific time delay.

- In compare mode, the 16-bit CCPR1 and ECCPR1 register value is constantly compared against either the TMR1 register pair value or the TMR3 register pair value. When a match occurs, the CCP1 pin can have one of the following actions :
  - Driven high
  - Driven low
  - Toggle output (High-to-low or low-to-high)
  - Remains unchanged.
- **CCP1M3:CCP1M0 :** CCP1 module mode select bit. These bits decide which action takes on compare match. At the same time, interrupt flag bit CCP1IF is set.
  - 0010 = Toggle output on match
  - 1000 = Initialize CCP1 pin low, on compare match this pin is set to high.
  - 1001 = Initialize CCP1 pin high, on compare match this pin is set low.
  - 1010 = On compare match generates software interrupt.
  - 1011 = On compare match trigger special event, reset the timer, start ADC conversion.
- Fig. Q.8.1 shows the compare mode block diagram.

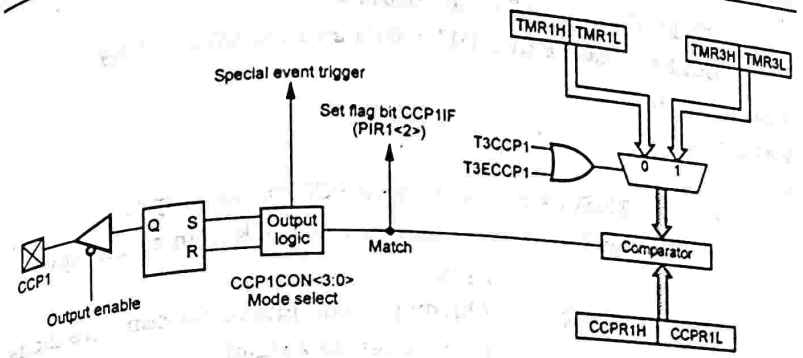


Fig. Q.8.1 Compare mode block diagram

Q.9 List the steps in programming PIC18F in compare mode. ☞ [SPPU : June-22, Marks 6]

Ans. : Steps for Programming

1. Set the CCP1 pin as an output.
2. Configure T3CON Register for Timer1 or Timer 3
3. If Timer1 is used then configure T1CON Register also.
4. Configure CCP1CON Register for compare mode.
5. Load desired count in CCPR1 (CCPR1H: CCPR1L) Register.
6. Initialize TMR1 or TMR3 register value.
7. Start Timer
8. Wait for CCP1IF (PIR1<2>) interrupt flag to set.

Q.10 Write PIC18F C program to generate a square wave of 2.5 kHz having a 50 % duty cycle using timer 3 compare mode. Assume oscillator frequency = 10 MHz. ☞ [SPPU : Dec.-15, Marks 8]

Ans. :

- **For 50 % duty cycle**
- Period of waveform = 1/ 2.5 kHz = 400 µs
- Timer clock period = 1/ (F_{OSC}/4) = 1/ (10 MHz/4) = 0.4 µs
- For 50 % duty cycle : ON time = OFF time = 200 µs

200 µs /0.4 µs = 500 i.e. 0x01F4

CCPR1 = 0x03E8 CCPR1H = 01H and CCPR1L = F4H

```
#include <p18f458.h>
void main(void)
{
    TRISCbits.TRISC2=0; // Configure RC2 pin as output
    TRISCbits.TRISC0=1; // Configure the T1CK1 pin as an input
                        // pin
    CCP1CON=0x02;       // Module is configured for compare mode
                        // and is set up so that
                        // upon a compare match of CCPR1
                        // and TMR3, RC2 is driven low
    T3CON=0xC0;         // Timer3 used for compare mode with
                        // TMR3 register in 16-bit format
    CCPR1=500;          // Load a count for generating 2.5 kHz
    PIE1bits.CCP1IE=1;  // Enable CCP1 interrupt Enable flag
    while (1)
    {
     TMR3=0;                // Clear Timer3 Register
     T3CONbits.TMR3ON=1;    // Turn On Timer3
     PIR1bits.CCP1IF=0;     // Clear interrupt flag
     // Wait for CCP Interrupt flag to set, it is set on compare match
       between CCPR1 and TMR3
        while(PIR1bits.CCP1IF==0);
        T3CONbits.TMR3ON=0;  // Turn OFF Timer3
    }
}
```

Q.11 A 1 Hz pulse is given to Timer3 (T1CK1) and LED is connected to the CCP1 pin. Write PIC18F C program to toggle LED after every 20 pulses.

Ans. :

```
#include <p18f458.h>
void main(void)
{
    TRISCbits.TRISC2=0; // Configure RC2 pin as output
    TRISCbits.TRISC0=1; // Configure the T1CK1 pin as an input
                        // pin
    CCP1CON=0x02;       // Module is configured for compare mode
                        // and is set up so that
                        // upon a compare match of CCPR1 and
                        // TMR3, RC2 is driven low
    T3CON=0xC2;         // Timer3 used for compare mode with
                        // TMR3 register in 16-bit format
    CCPR1=20;           // Load a count 20
    PIE1bits.CCP1IE=1;  // Enable CCP1 interrupt Enable flag
    while (1)
    {
     TMR3=0;                // Clear Timer3 Register
     T3CONbits.TMR3ON=1;    // Turn On Timer3
     PIR1bits.CCP1IF=0;     // Clear interrupt flag
      // Wait for CCP Interrupt flag to set, it is set on compare
        match between CCPR1
      //and TMR3
      while(PIR1bits.CCP1IF==0);
   T3CONbits.TMR3ON=0;      // Turn OFF Timer3
    }
}
```

8.5 : PWM Mode

Q.12 Explain the PWM mode of CCP module.

Ans. : Pulse Width Modulation (PWM) is a digital signal which is most commonly used in control circuitry. This signal is set high and low) in a predefined time and speed. The time during which the signal stays high is called the "**on time**" and the time during which the signal stays low is called the "**off time**".

- **Duty cycle of the PWM :** The percentage of time in which the PWM signal remains HIGH (on time) is called as **duty cycle.**
- **Frequency of a PWM :** The frequency of a PWM signal determines how fast a PWM completes one period. One Period is complete ON and OFF of a PWM signal.
- The PIC18F when configured in Pulse-Width Modulation (PWM) mode, the CCP1 pin produces up to a 10-bit resolution PWM output. This means that for a value of 0 there will be a duty cycle of 0 % and for a value of 1024 (2^{10}) there be a duty cycle of 100 %.
- Since the CCP1 pin is multiplexed with the PORTC data latch, the TRISC<2> bit must be cleared to make the CCP1 pin an output.
- Fig. Q.12.1 shows a simplified block diagram of the CCP module in PWM mode.

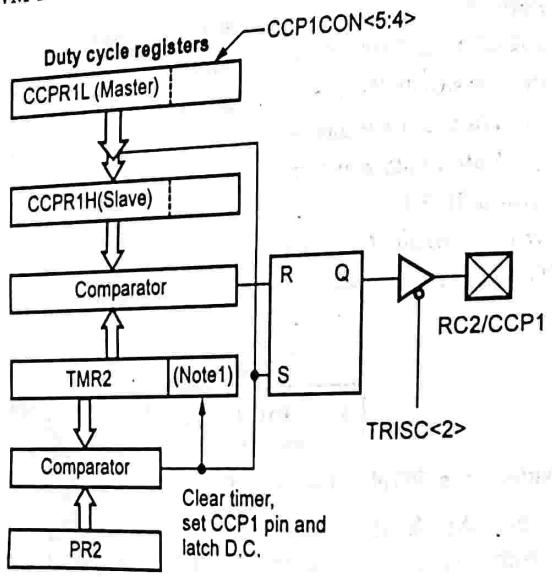


Fig. Q.12.1 Simplified block diagram of the CCP module in PWM mode

Q.13 List the steps in programming PIC18F in PWM mode.

Ans. : PWM Programming steps

- Following steps are needed to setup the PWM module in PIC18F458.

1. Set the PWM period by writing to the PR2 register.
2. Set the PWM duty cycle by writing to the CCPR1L register and CCP1CON<5:4> bits.
3. Configure the CCP1CON module for PWM operation.
4. Make the CCP1 pin an output by clearing the TRISC<2> bit.
5. Clear Timer2 register
6. Set the TMR2 prescale value
7. Enable Timer2 by writing to T2CON.

Q.14 Explain the calculations for PWM period and duty cycle with the help of example.

Ans. : 1. Set the PWM Period

- The PWM period is set by writing the PR2 register. So the value can be calculated by using this formula.i.e,

$$\text{PWM period} = [(PR2) + 1] \times 4 \times T_{osc} \times N$$

where T_{osc} is the $1/F_{osc}$

N : TMR2 prescale value. Can be set to 1, 4 or 16 by programming Timer2 control register (T2CON)

$$\therefore \quad PR2 = \frac{\text{PWM period}}{T_{osc} \times 4 \times N} - 1 = \frac{F_{osc}}{\text{PWM frequency} \times 4 \times N} - 1$$

- For example, if we use a 20 MHz clock, the O/P frequency is 2 kHz and N = 4 then

$$PR2 = \frac{20\ \text{MHz}}{2\ \text{kHz} \times 4 \times 4} - 1 = 624$$

2. Set PWM Duty Cycle

- The PWM duty cycle is specified by writing to the CCPRxL register and to the CCPxCON<5:4> bits. Up to 10-bit resolution is available. The CCPRxL contains the eight MSbs and the CCPxCON<5:4> bits contain the two LSbs. This 10-bit value is

represented by CCPRxL: CCPxCON<5:4>. The duty cycle will be in the range of 0 to 1024.

- Now, let us see how to set a value for the CCPR1L which decides the duty cycle of a pulse. We know that a duty cycle is some % of the PR2 (period) register. For example, if PR2 is 201 then a 25 % duty cycle of the 201 is given by,

$$(CCPR1L : CCP1CON < 5 : 4 >) = PR2 \times \left(\frac{\text{Duty Cycle}}{100}\right)$$

$$(CCPR1L: CCP1CON < 5 : 4 >) = (201) \times (25/100)$$

$$(CCPR1L: CCP1CON < 5 : 4 >) = 50.25$$

- Here, hexademical equivalent of the integer part of the result i.e., 50 is loaded as 8 most significant bits in CCPR1L and 2 LSB bits in CCP1CON <5:4> are used to represent fraction portion of the result, i.e. 0.25.
- The decimal point portion of the duty cycle count is set as shown in Table Q.14.1.

CCP1CON <5:4>		Decimal Point
DC1B2	DC1B1	
0	0	0
0	1	0.25
1	0	0.5
1	1	0.75

Table Q.14.1 Representation decimal point portion

CCPR1L = 50 = 0b00110010 = 0x32

CCP1CON <5:4> = DC1B2 : DC1B1 = 0b01

Q.15 Write PIC18F C program to generate 10 % duty cycle PWM waveform with frequency 1 kHz. Assume XTAL = 10 MHz.

☞ [SPPU : May-15, 16, Marks 8]

Ans. : Assume N = 16

$$PR2 = \frac{Fosc}{PWM\ frequency \times 4 \times N} - 1 = \frac{10\,MHz}{1\,KHz \times 4 \times 16} - 1 = 155$$

$$(CCPR1L : CCP1CON<5 : 4>) = PR \times \frac{10}{100} = 155 \times \frac{10}{100} = 15.50$$

CCPR1L = 15 = 0b00001111 = 0 × 0F

CCP1CON <5:4> = DC1B2: DC1B1 = 0b10

```
#include <p18f458.h>
void main(void)
{
  PR2 = 155;                  // Configure Timer2 period
  CCPR1L = 15                 // Set up PWM duty cycle
  CCP1CON = 0x2C;             // DC1B1:DC1B0 = 10, PWM mode
                              // (CCP1M3:CCP1M0 = 1100)
  TRISCbits.TRISC2=0;         // Configure RC2 pin as output
  T2CON = 0x02;               // Set prescalar 16, no postscale
  TMR2 = 0                    // Clear Timer2
  T2CONbits.TMR2ON = 1;       // Timer2 on
  while(1)
  {
    PIR1bits.TMR2IF = 0;      // Clear Timer2 flag
    while (PIR1bits.TMR2IF==0); // Wait for end of period
  }
}
```

Q.16 Create a 2 kHz frequency with 25 % duty cycle on the CCP1 pin. Assume XTAL = 10 MHz.

☞ [SPPU : Dec.-14, Marks 8]

Ans. : Assume N = 16

$$PR2 = \frac{Fosc}{PWM\ frequency \times 4 \times N} - 1 = \frac{10\,MHz}{2\,KHz \times 4 \times 16} - 1 = 77$$

$$(CCPR1L:CCP1CON<5:4>) = PR \times \frac{25}{100} = 77 \times \frac{25}{100} = 19.25$$

CCPR1L = 19 = 0b00011001 = 0 × 13

CCP1CON <5:4> = DC1B2: DC1B1 = 0b01.

```
#include <p18f458.h>
void main(void)
{
   PR2 = 77;                    // Configure Timer2 period
   CCPR1L = 19                  // Set up PWM duty cycle
   CCP1CON = 0x1C;              // DC1B1:DC1B0 = 01, PWM mode
                                // (CCP1M3:CCP1M0 = 1100)
   TRISCbits.TRISC2=0;          // Configure RC2 pin as output
   T2CON = 0x02;                // Set prescalar 16, no postscale
   TMR2 = 0                     // Clear Timer2
   T2CONbits.TMR2ON = 1;        // Timer2 on
   while(1)
   {
      PIR1bits.TMR2IF = 0;      // Clear Timer2 flag
      while (PIR1bits.TMR2IF==0); // Wait for end of period

   }
}
```

8.6 : DC Motor Speed Control with CCP

Q.17 How the speed of DC motor is controlled by PWM, explain in brief. ☞ **[SPPU : Dec.-16, Marks 6]**

Ans. : We know that the speed of the dc motor depends on the applied voltage. The average applied DC voltage and hence the power can be varied using technique called **pulse width modulation.** In this technique, the dc power supply is not a voltage of fixed amplitude; however it is a pulsating DC voltage. By changing pulse width we can change the applied power. This is illustrated in the Fig. Q.17.1.

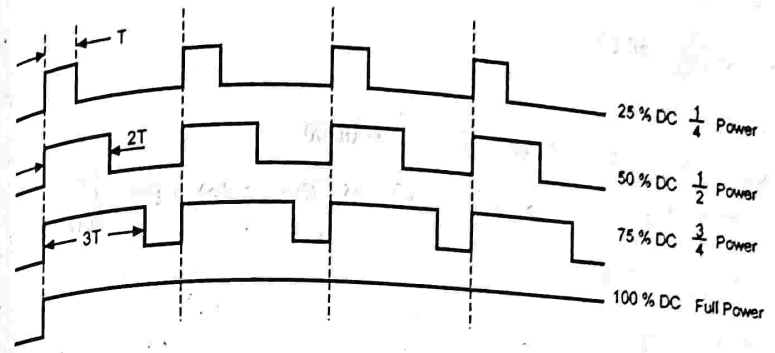


Fig. Q.17.1 Applied power variation using pulse width modulation

Q.18 Draw an interfacing diagram and write an algorithm for DC motor speed controller using PIC18xxx

☞ **[SPPU : May-15, 16, Dec.-15, Marks 10]**

Ans. :

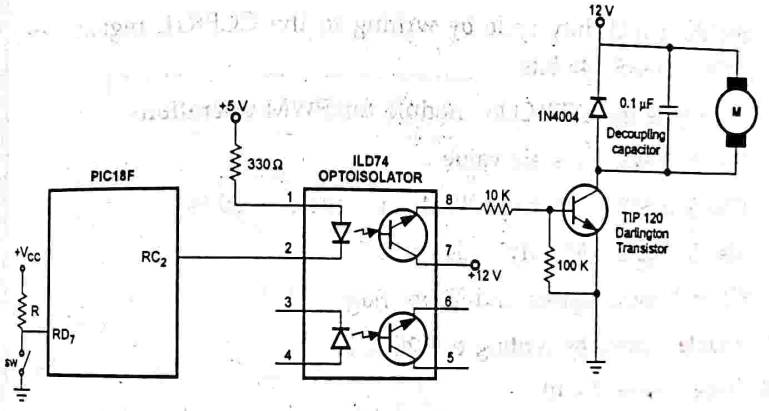


Fig. Q.18.1 DC motor connection using a bipolar transistor

Q.19 Let us write an PIC18 C program to read the state of ON-OFF switch connected to RD7. If it is low apply 50 % of power; otherwise apply 75 % of power using PWM technique.

Ans. : Let us take period PR2 = 100

For 50 % Duty Cycle : (CCPR1L : CCP1CON<5 : 4>) = $PR \times \frac{50}{100}$

$= 100 \times \frac{50}{100} = 50.00$

CCPR1L = 50

CCP1CON <5 : 4> = DC1B2 : DC1B1 = 0b00

For 75 % Duty Cycle : (CCPR1L : CCP1CON<5 : 4>) = $PR \times \frac{75}{100}$

$= 100 \times \frac{75}{100} = 75.00$

CCPR1L = 75

CCP1CON <5 : 4> = DC1B2 : DC1B1 = 0b00

Algorithm

1. Make the CCP1 pin an output by clearing the TRISC<2> bit to give PWM output
2. Configure RD7 pin as input pin to read switch input
3. Set the PWM period by writing to the PR2 register.
4. Set the PWM duty cycle by writing to the CCPR1L register and CCP1CON<5 : 4> bits.
5. Configure the CCP1CON module for PWM operation.
6. Set the TMR2 prescale value
7. Check if SW = 0, Set up PWM duty cycle = 50 %
   else Set up PWM duty cycle = 75 %
8. Clear Timer2 register and Timer flag
9. Enable Timer2 by writing to T2CON.
10. Repeat steps 7 - 10.

Program

```
#include <p18f458.h>
void main(void)
{
   TRISCbits.TRISC2 = 0;    // Configure RC2 pin as output to
                            // produce PWM output
```



```
   TRISDbits.TRISD7 = 1;    // Configure RD7 pin as input
   PR2 = 100;               // Configure Timer2 period
   CCP1CON = 0x0C;          // DC1B1 : DC1B0 = 00, PWM mode
                            // (CCP1M3 : CCP1M0 = 1100)
   T2CON = 0x02;            // Set prescalar 16, no postscale
   while(1)
   {
     If (PORTDbits.RD7 == 0)
     CCPR1L = 50                    // Set up PWM duty cycle 50 %
     else
      CCPR1L = 75                   // Set up PWM duty cycle 75 %
     TMR2 = 0                       // Clear Timer2
     PIR1bits.TMR2IF = 0;           // Clear Timer2 flag
     T2CONbits.TMR2ON = 1;          // Timer2 on

     while (PIR1bits.TMR2IF==0);    // Wait for end of period
   }
}
```

8.7 : Stepper Motor Interfacing with PIC

Q.20 Draw and explain the stepper motor interface.

Ans. : A stepper motor is a digital motor. It can be driven by digital signal. Fig. Q.20.1 shows the typical 2 phase motor interfaced using PIC. Motor shown in the circuit has two phases, with center-tap winding.

- The center taps of these windings are connected to the 12 V supply. Due to this, motor can be excited by grounding four terminals of the two windings. Motor can be rotated in steps by giving proper excitation sequence to these windings.
- The lower nibble of Port B of the PIC18F is used to generate excitation signals in the proper sequence.

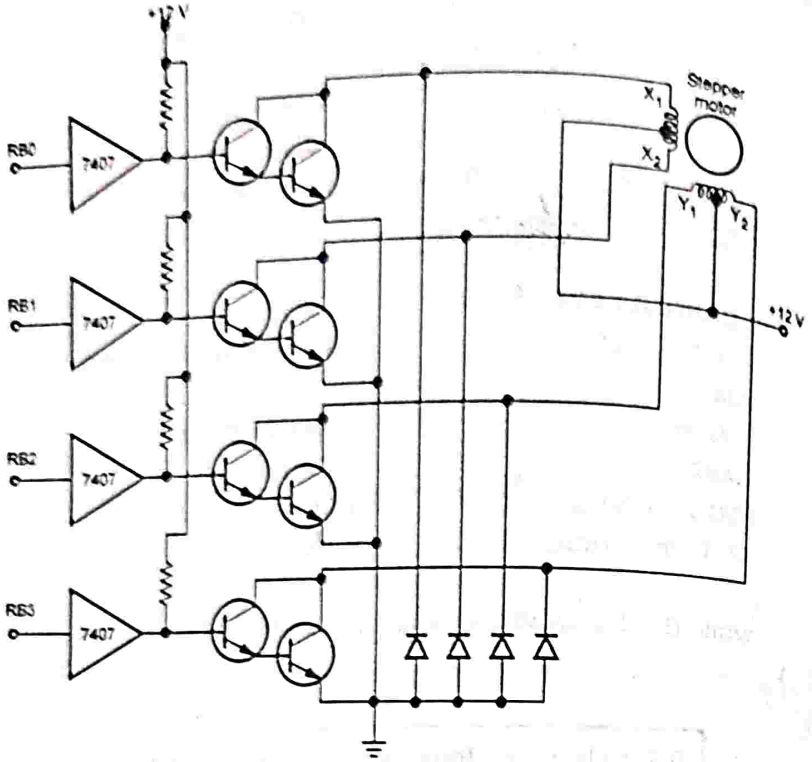


Fig. Q.20.1 Stepper motor interface

- Particular winding is excited by making corresponding Port pin low. For example, winding X1 can be excited by making RB0 low.
- Table Q.20.1 shows typical excitation sequence. The given excitation sequence rotates the motor in clockwise direction. To rotate motor in anticlockwise direction we have to excite motor in a reverse sequence. The excitation sequence for stepper motor may change due to change in winding connections. However, it is not desirable to excite both the ends of the same winding simultaneously.

Step	Y_2	Y_1	X_2	X_1	RB3	RB2	RB1	RB0	CODE
1	1	0	1	0	0	1	0	1	05H
2	1	0	0	1	0	1	1	0	06H
3	0	1	0	1	1	0	1	0	0AH
4	0	1	1	0	1	0	0	1	09H
5	1	0	1	0	0	1	0	1	05H

Table Q.20.1 Full step excitation sequence

- The excitation sequence given in Table Q.20.1 is called **full step sequence**. In which excitation ends of the phase are changed in one step. The excitation sequence given in Table Q.20.2 takes two steps to change the excitation ends of the phase. Such a sequence is called **half step sequence** and in each step the motor is rotated by 0.9°.

Step	Y_2	Y_1	X_2	X_1	RB3	RB2	RB1	RB0	CODE
1	1	0	1	0	0	1	0	1	05H
2	1	0	0	0	0	1	1	1	07H
3	1	0	0	1	0	1	1	0	06H
4	0	0	0	1	1	1	1	0	0EH
5	0	1	0	1	1	0	1	0	0AH
6	0	1	0	0	1	0	1	1	0BH
7	0	1	1	0	1	0	0	1	09H
8	0	0	1	0	1	1	0	1	0DH
1	1	0	1	0	0	1	0	1	05H

Fig. Q.20.2 Half step excitation sequence

Q.21 Write an PIC18 C program to read the state of ON-OFF switch connected to RD7. If it is rotate stepper motor clockwise, otherwise rotate motor anti-clockwise.

Ans. :

```
#include <p18f458.h>
void main(void)
{
  TRISB = 0;                // Configure Port B (RB0 - RB3) as output
                            // to drive stepper motor
  TRISDbits.TRISD7 = 1;     // Configure RD7 pin as input
  while(1)
  {
    If (PORTDbits.RD7 == 0)
    {
      PORTB = 0x05;
      Delay(100);
      PORTB = 0x06;
      Delay(100);
      PORTB = 0x0A;
      Delay(100);
      PORTB = 0x09;
      Delay(100);
    }
    else
    {
      PORTB = 0x09;
      Delay(100);
      PORTB = 0x0A;
      Delay(100);
      PORTB = 0x06;
      Delay(100);
      PORTB = 0x05;
      Delay(100);
    }
  }
}
void Delay(unsigned int cnt)
{
  unsigned int i, j;
  for(i=0; i<1275; i++)
    for(j=0; j<cnt; j++);
}
```

END...

Unit V

9 PIC Interfacing - III

9.1 : Interfacing of ADC 0808 with PIC

Q.1 Draw interfacing circuit for ADC0808 with PIC18.

Ans. : Fig. Q.1.1 shows typical interfacing circuit for ADC 0808 with microcontroller system.

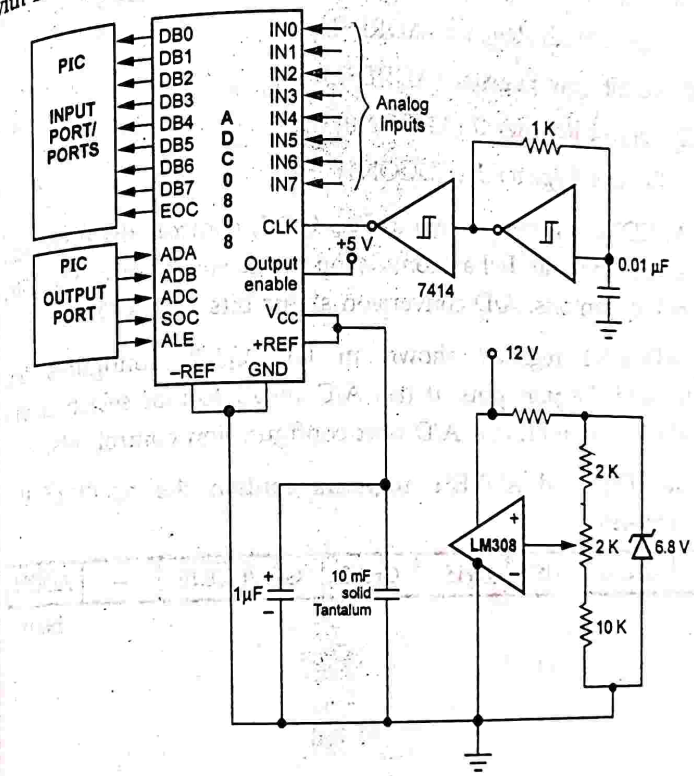


Fig. Q.1.1 Typical Interface for 0808/0809

9.2 : Analog to Digital Converter (A/D) Module

Q.2 Write a note on ADC registers available A/D module of PIC18.

OR Explain in detail the functions of ADCON1 SFR of PIC18 microcontroller. ☞ [SPPU : Dec.-22, Marks 7]

OR Explain in detail the functions of following flags related to onboard ADC of PIC18 microcontroller. ☞ [SPPU : Dec.-22, Marks 6]
i) GO/DONE ii) ADON

OR Explain in detail the functions of ADCON0 SFR of PIC18 microcontroller. ☞ [SPPU : June-22, Marks 7]

Ans. : 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. Q.2.1, 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. Q.2.2, 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 /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 iscomplete)

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

Fig. Q.2.1 ADCON0 register

ADFM	ADCS2	-	-	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM** : A/D Result Format Select bit

1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'

0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'

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 5-4 **Unimplemented** : Read as '0'

bit 3-0 **PCFG3:PCFG0** : A/D Port Configuration Control bits

PCF6	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	V_{REF+}	V_{REF-}	C/R
0000	A	A	A	A	A	A	A	A	V_{DD}	V_{SS}	8/0
0001	A	A	A	A	V_{REF+}	A	A	A	AN3	V_{SS}	7/1

0010	D	D	D	A	A	A	A	A	V_{DD}	V_{SS}	5/0
0011	D	D	D	A	V_{REF+}	A	A	A	AN3	V_{SS}	4/1
0100	D	D	D	D	A	D	A	A	V_{DD}	V_{SS}	3/0
0101	D	D	D	D	V_{REF+}	D	A	A	AN3	V_{SS}	2/1
011x	D	D	D	D	D	D	D	D	-	-	0/0
1000	A	A	A	A	V_{REF+}	V_{REF-}	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	V_{DD}	V_{SS}	6/0
1010	D	D	A	A	V_{REF+}	A	A	A	AN3	V_{SS}	5/1
1011	D	D	A	A	V_{REF+}	V_{REF-}	A	A	AN3	AN2	4/2
1100	D	D	D	A	V_{REF+}	V_{REF-}	A	A	AN3	AN2	3/2
1101	D	D	D	D	V_{REF+}	V_{REF-}	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	V_{DD}	V_{SS}	1/0
1111	D	D	D	D	V_{REF+}	V_{REF-}	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O

C/R = # of analog input channels/# of A/D voltage references

Fig. Q.2.2 ADCON1 register

- Fig. Q.2.3 shows the A/D result formats.

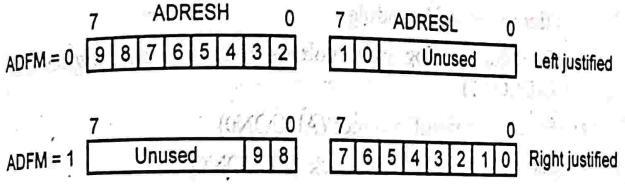


Fig. Q.2.3 A/D result formats

Q.3 Draw the block diagram of ADC in PIC18.

Ans. : Fig. Q.3.1 shows the block diagram of ADC.

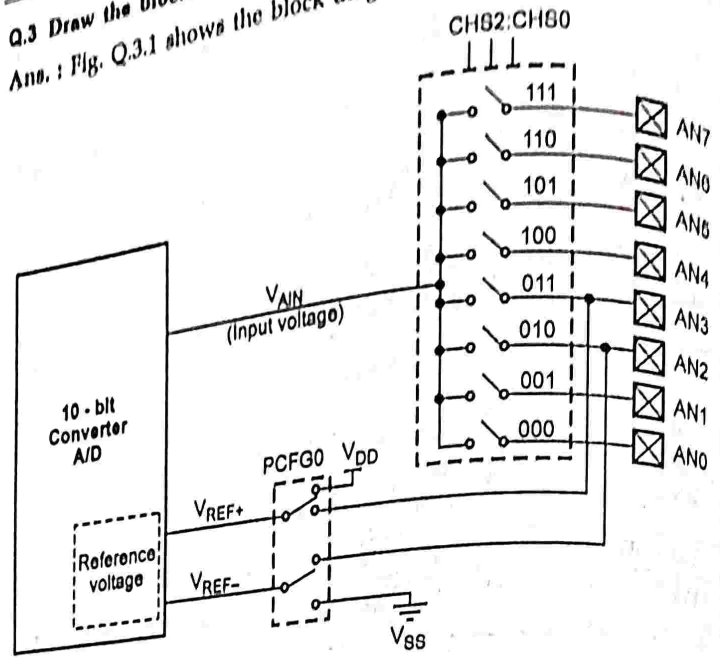


Fig. Q.3.1 Block diagram of ADC

Q.4 List the steps for programming ADC.

Ans. : 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.

Q.5 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.

Ans. :

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.

Q.6 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.

Ans. :

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++);
}
```

Q.7 Draw and explain interfacing diagram of DAC 0808 to PIC18. Write PIC18 C program to generate triangular waveform.

☞ [SPPU : June-22, Dec.-22, Marks 8]

Ans. : Fig. Q.7.1 shows the interfacing of DAC 0808 with PIC18 microcontroller.

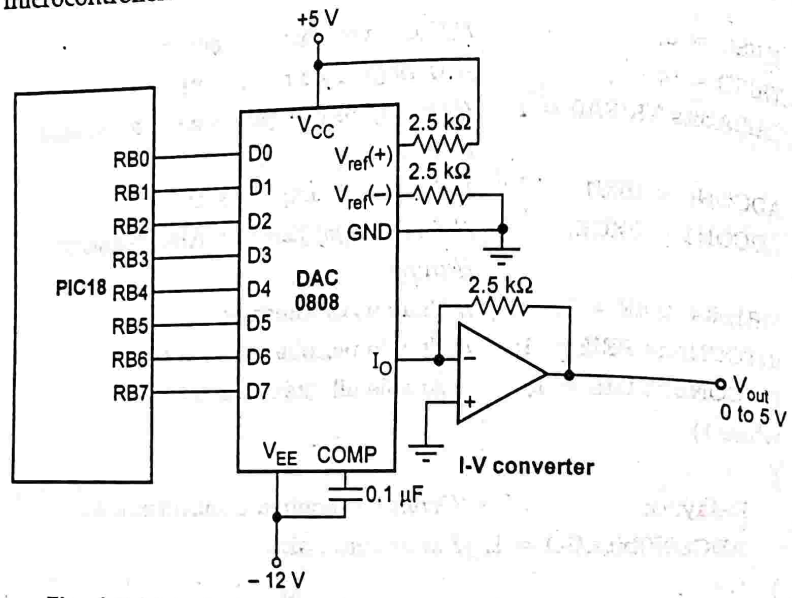


Fig. Q.7.1 Interfacing of DAC 0808 with PIC18 microcontroller

- 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,

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

Note : 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 $(11111111)_2$ binary input it sinks 1.992 mA.

- The circuit shown in the Fig. Q.7.1 gives output in the unipolar range. When digital input is 00H, the output voltage is 0 V and when digital input is FFH $(11111111)_2$, the output voltage is + 5 V.
- 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;
   }
}
```

Q.8 Write a PIC18 C program to generate staircase waveform.

Ans. : Let us see the staircase waveform

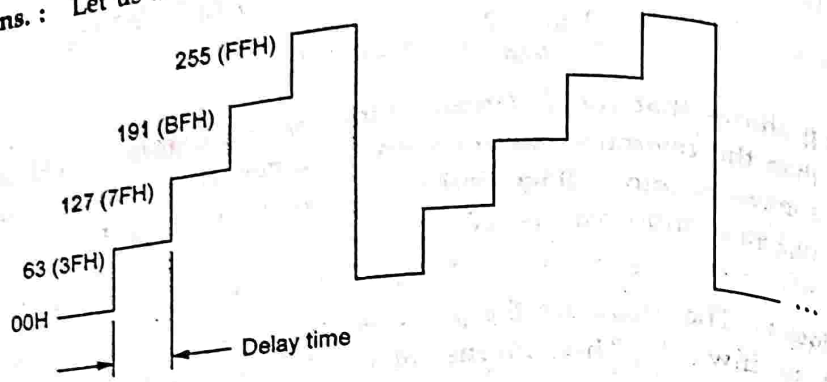


Fig. Q.8.1

```
#include <p18f458.h>
void main ( )
{
   unsigned char wavevalues [5] = {0, 63, 127, 191, 255};
   unsigned char x;
   TRISB = 0;  // Configure Port B as output
   while (1)
   {
      for (x = 0; x < 5; x++ )
      {
         DACDATA = wavevalues [x] ;
         Delay (200) ;
      }
   }
}
void Delay(unsigned int cnt)
{
   unsigned int i, j;
   for(i=0; i<1275; i++)
      for(j=0; j<cnt; j++);
}
```


9.3 : Temperature Sensor Interfacing using ADC

Q.9 Draw and explain the interfacing of temperature sensor with PIC18. [SPPU : June-22, Marks 8]

Ans. : 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.
- The output analog voltage can be converted to digital form using ADC so that a microcontroller can process it.
- Fig. Q.9.1 shows the circuit connection for temperature sensor LM35 with PIC18 microcontroller.

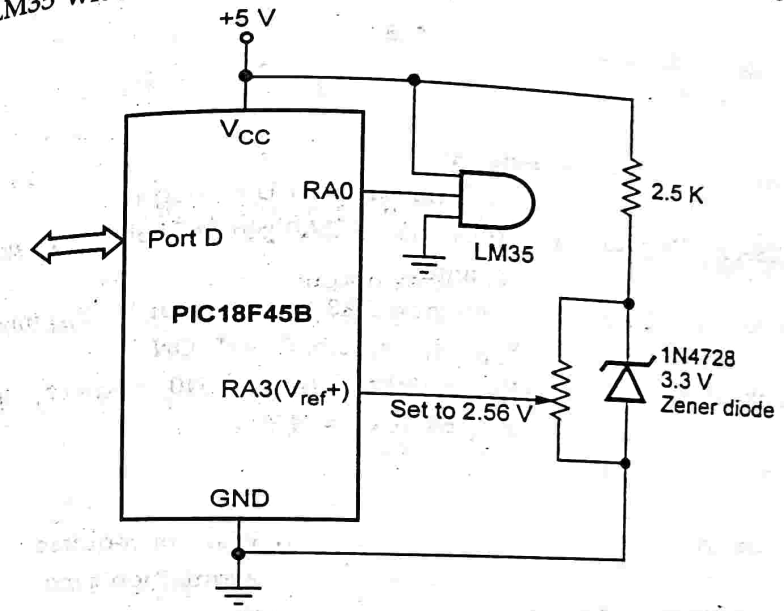


Fig. Q.9.1 Sensor Interface with PIC18F458

- 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.

Q.10 Write PIC18 C code for reading the temperature for the circuit shown in Fig. Q.9.1 and displaying output on Port D.

Ans. :

```
#include <p18f458.h>
void main(void)
{
  Unsigned char Byte_L, Byte_H;
  TRISD = 0;                  // Configure Port D as output
  TRISAbits.TRISA0 = 1;       // Configure RA0 pin as input for analog
                              // input
  TRISAbits.TRISA3 = 1;// Configure RA3 pin as input for Vref input
  ADCON0 = 0X81;      // Fosc/64, Channel 0, A/D ON
  ADCON1 = 0XC1;      // Result right justified, AN0 = Analog input,
                         Fosc/64, AN3 = +Vref
  while(1)
  {
    Delay(1);                          // Wait for required
                                       // acquisition time
    ADCON0bits.GO = 1;                 // Start conversion
      while(ADCON0bits.DONE == 1);     // Wait for the conversion
      Byte_L = ADRESL;                 // Save lower byte
                                       // of result
      Byte_H = ADRESH;                 // Save higher byte
                                       // of result
```



```
      Byte_L >>=2;                     // Shift right by 2-bits
      Byte_L &= 0x3F;                  // mask the upper 2 bits
      Byte_H <<=6;                     // Shift left by 6-bits
      Byte_H &= 0xC0;                  // mask the lower 6 bits
      PORTD = Byte_L | Byte_H
  }
}
void Delay(unsigned int cnt)
{
  unsigned int i, j;
  for(i=0; i<1275; i++)
    for(j=0; j<cnt; j++);
}
```

Q.11 What are the features of DS1306 RTC ?

☞ [SPPU : May-16, Dec.-22, Marks 4]

Ans. : The features of DS1306 RTC are :

- The DS1306 Serial Alarm Real Time Clock provides a full BCD clock calendar which is accessed via a simple serial interface.
- The clock/calendar provides seconds, minutes, hours, day, date, month and year information. The end of the month date is automatically adjusted for months with less than 31 days, including corrections for leap year.
- The clock operates in either the 24-hour or 12-hour format with AM/PM indicator.
- It has 96 byte nonvolatile RAM for data storage.
- It has two Time of Day Alarms - programmable on combination of seconds, minutes, hours and day of the week.
- It has 1 Hz and 32.768 KHz clock outputs.
- Its serial interface supports Motorola Serial Peripheral Interface (SPI) serial data ports or standard 3-wire interface.
- It has burst Mode for reading/writing successive addresses in clock/RAM.
- It offers dual power supplies as well as a battery input pin.

- The dual power supplies support a programmable trickle charge circuit which allows a rechargeable energy source to be used for a backup supply.
- The V_{BAT} pin allows the device to be backed up by a non-rechargeable battery.

- It has two operating voltage ranges : 2.5 - 5.5 volt operation or optional 2.0 - 5.5 volt full operation.
- It has wide operating temperature range – 40 °C to + 85 °C.
- It is available in space efficient 16-pin TSSOP package.

Q.12 Explain the pin diagram of DS1306 RTC ? ☞ [SPPU : Dec.-22, Marks 3]

Ans. : Fig. Q.12.1 shows the pin diagram of DS1306.

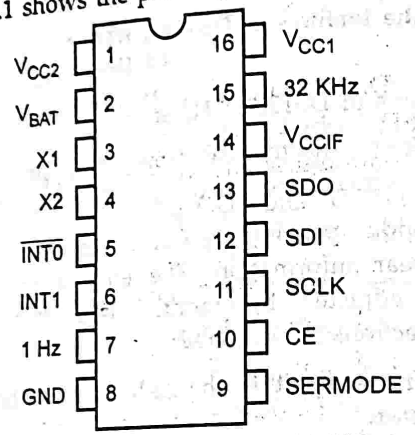


Fig. Q12.1 Pin diagram of DS1306

- V_{CC1} - DC power is provided to the device on this pin. VCC1 is the primary power supply.
- V_{CC2} - This is the secondary power supply pin. In systems using the trickle charger, the rechargeable energy source is connected to this pin.
- V_{BAT} - Battery input for any standard 3 volt lithium cell or other energy source.
- V_{CCIF} **(Interface Logic Power Supply Input)** - The VCCIF pin allows the DS1306 to drive SDO.

- **SERMODE** - pin offers the flexibility to choose between two serial interface modes.
  - When connected to GND, standard 3-wire communication is selected.
  - When connected to VCC, Motorola SPI communication is selected.
- **SCLK (Serial Clock Input)** - SCLK is used to synchronize data movement on the serial interface for either the SPI or 3-wire interface.
- **SDI (Serial Data Input)** - When SPI communication is selected, the SDI pin is the serial data input for the SPI bus. When 3-wire communication is selected, this pin must be tied to the SDO pin (the SDI and SDO pins function as a single I/O pin when tied together).
- **SDO (Serial Data Output)** - When SPI communication is selected, the SDO pin is the serial data output for the SPI bus. When 3-wire communication is selected, this pin must be tied to the SDI pin (the SDI and SDO pins function as a single I/O pin when tied together).
- **CE (Chip Enable)** - The Chip Enable signal must be asserted high during a read or a write for both 3-wire and SPI communication.
- $\overline{INT0}$ **(Interrupt 0 Output)** - This pin is an active low output of the DS1306 that can be used as an interrupt input to a processor. It can be programmed to be asserted by Alarm 0. The $\overline{INT0}$ pin is an open drain output and requires an external pull-up resistor.
- **1 Hz (1 Hz Clock Output)** - The 1 Hz pin provides a 1 Hz squarewave output.
- **INT1 (Interrupt 1 Output)** - The INT1 pin is an active high output of the DS1306 that can be used as an interrupt input to a processor. The INT1 pin can be programmed to be asserted by alarm 1.
- X_1, X_2 - Connections for a standard 32.768 KHz quartz crystal.

Q.13 Draw the block diagram of DS1306 RTC ?

Ans. : Fig. Q13.1 shows the simplified block diagram of DC1306 RTC.

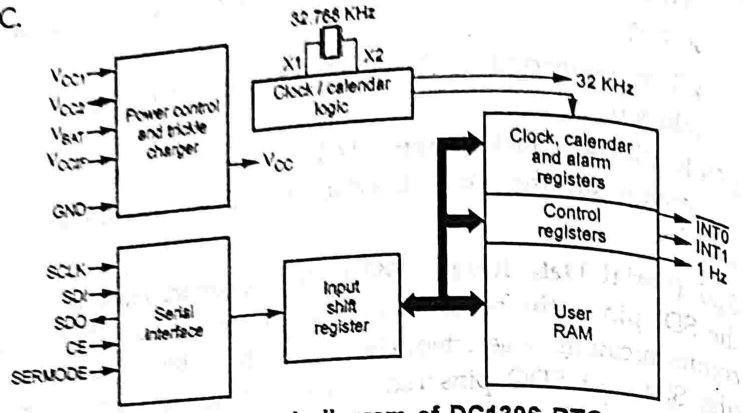


Fig. Q13.1 Block diagram of DC1306 RTC

Q.14 Draw and explain the RAM address map of DS1306.

Ans. : Fig. Q.14.1 shows the address map for the RTC DS1306.

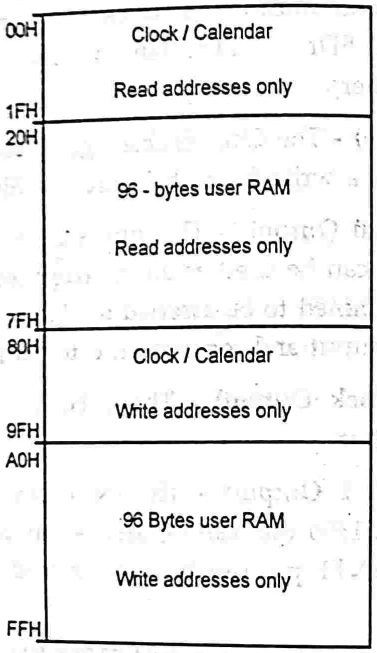


Fig. Q.14.1 RAM address map of DS1306

- Data is written to the RTC by writing to address locations 80h to 9Fh and is written to the RAM by writing to address locations A0h to FFh. RTC data is read by reading address locations 00h to 1Fh and RAM data is read by reading address locations 20h to 7Fh.

Q.15 Write a note on real time registers.

Ans. : • The real time clock registers are illustrated in Fig. Q.15.1.

Hex address Read	Hex address Write		Bit 7					Bit 0	Range
00H	80H		0	10 Sec				Sec	00 - 59
01H	81H		0	10 Min				Min	00 - 59
02H	82H		0	12/24	10 A/P	10 Hr		Hours	01 - 12 + A/P 00 - 23
03H	83H		0	0	0	0		Day	01 - 07
04H	84H		0	0	10 Date			Date	01 - 31
05H	85H		0	0	10 Month			Month	01 - 12
06H	86H		10 Year					Year	00 - 99
07H	87H		M	10 Sec alarm				Sec alarm	00 - 59
08H	88H		M	10 Min alarm				Min alarm	00 - 59
09H	89H	Alarm 0	M	12/24	10 A/P	10 Hr		Hour alarm	01 - 12 + A/P 00 - 23
0AH	8AH		M	0	0	0		Day alarm	01 - 07
0BH	8BH		M	10 Sec alarm				Sec alarm	00 - 59
0CH	8CH		M	10 Min alarm				Min alarm	00 - 59
0DH	8DH	Alarm 1	M	12/24	10 A/P	10 Hr		Hour alarm	01 - 12 + A/P 00 - 23
0EH	8EH		M	0	0	0		Day alarm	01 - 07
0FH	8FH		Control register						
10H	90H		Status register						
11H	91H		Trickle charger register						
12H - 1FH	92H - 9FH		Reserved						

Range for alarm registers does not include mask 'M' bit.

Fig. Q.15.1 Real time clock registers

- The time, calendar and alarm are set or initialized by writing the appropriate register bytes.
- Note that some bits are set to zero. These bits will always read 0 regardless of how they are written. Also note that registers 12h to 1Fh (read) and registers 92h to 9Fh are reserved. These registers will always read 0 regardless of how they are written.
- The contents of the time, calendar, and alarm registers are in the Binary-Coded Decimal (BCD) format.
- The DS1306 can be run in either 12-hour or 24-hour mode. Bit 6 of the hours register is defined as the 12 or 24 hour mode select bit. When high, the 12-hour mode is selected.
- The DS1306 contains two time of day alarms. Time of Day Alarm 0 can be set by writing to registers 87h to 8Ah. Time of Day Alarm 1 can be set by writing to registers 8Bh to 8Eh. Bit 7 of each of the time of day alarm registers are mask bits. When all of the mask bits are logic 0, a time of day alarm will only occur once per week when the values stored in timekeeping registers 00h to 03h match the values stored in the time of day alarm registers. An alarm will be generated every day when bit 7 of the day alarm register is set to a logic 1. An alarm will be generated every hour when bit 7 of the day and hour alarm registers is set to a logic 1. Similarly, an alarm will be generated every minute when bit 7 of the day, hour, and minute alarm registers is set to a logic 1. When bit 7 of the day, hour, minute, and seconds alarm registers is set to a logic 1, alarm will occur every second.

Q.16 List the special purpose registers of DS 1306.

Ans. : The DS1306 has three special purpose registers. Control register, Status register and Trickle charger register that control the real time clock, interrupts and trickle charger.

Q.17 Explain the control register of DS1306.

Ans. : Fig. Q17.1 shows the bit pattern for the control register. Control register has an address 0FH for read and 8FH for write.

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0	WP	0	0	0	1 Hz	AIE1	AIE0

Fig. Q17.1 Bit pattern for the control register

- **WP (Write Protect)** - Up on power-up this bit is undefined. Therefore, before any write operation to the clock or RAM, this bit must be logic 0. When high, the write protect bit prevents a write operation to any other register and the WP bit is the only bit in the control register that can be written.
- **1 Hz (1 Hz output enable)** - This bit controls the 1 Hz output. When this bit is a logic 1, the 1 Hz output is enabled. When this bit is a logic 0, the 1 Hz output is high Z.
- **AIE0 (Alarm Interrupt Enable 0)** - When set to a logic 1, this bit permits the Interrupt 0 Request Flag (IRQF0) bit in the status register to assert $\overline{INT0}$. When the AIE0 bit is set to logic 0, alarm 0 interrupt is disabled.
- **AIE1 (Alarm Interrupt Enable 1)** - When set to a logic 1, this bit permits the Interrupt 1 Request Flag (IRQF1) bit in the status register to assert INT1. When the AIE1 bit is set to logic 0, alarm 1 interrupt is disabled.

Q.18 Explain the status register of DS1306.

Ans. : Fig. Q18.1 shows the bit pattern for the status register. Is a read only register. Status register has an address 10H for read.

BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0
0	0	0	0	0	0	IRQF1	IRQF0

Fig. Q.18.1 Bit pattern for the status register

- **IRQF0 (Interrupt 0 Request Flag)** - A logic 1 in the Interrupt Request Flag bit indicates that the current time has matched the Alarm 0 registers. If the AIE0 bit is also a logic 1, the $\overline{INT0}$ pin

will go low. IRQF0 is cleared when any of the Alarm 0 registers are read or written.

- **IRQF1 (Interrupt 1 Request Flag)** - A logic 1 in the Interrupt Request Flag bit indicates that the current time has matched the Alarm 1 registers. If the AIEI bit is also a logic 1, the INT1 pin will generate a 62.5 ms active high pulse. IRQF1 is cleared when any of the Alarm 1 registers are read or written.

Q.19 Explain the PIC18 interfacing to DS1306 using MSSP module (SPI mode).

Ans. :

- The SPI mode of serial communication is selected by connecting the SERMODE pin to VCC. Four pins are used for the SPI. The four pins are the SDO (Serial Data Out), SDI (Serial Data In), CE (Chip Enable), and SCLK (Serial Clock). The DS1306 is the slave device in an SPI application, with the microcontroller being the master.
- The master Synchronous Serial Port (MSSP) module of PIC18 supports SPI bus protocol. Fig. Q19.1 shows the interfacing of DS1306 with PIC18 microcontroller. As shown in Fig. Q19.1, pins RC2, RC3, RC4 and RC5 are designated to CE, SCLK, SDI and SDO, respectively.

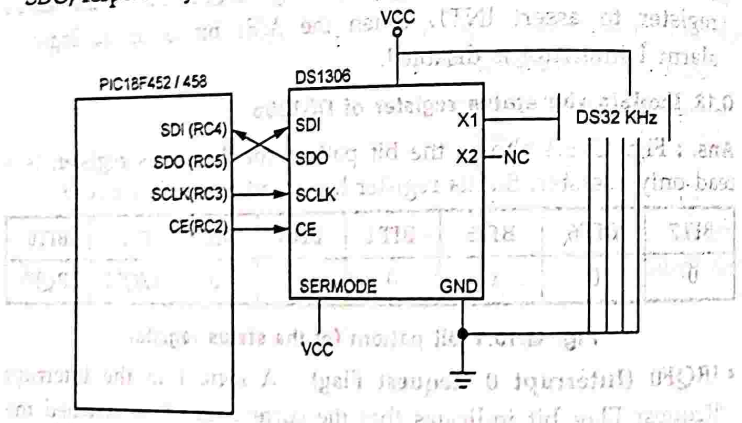
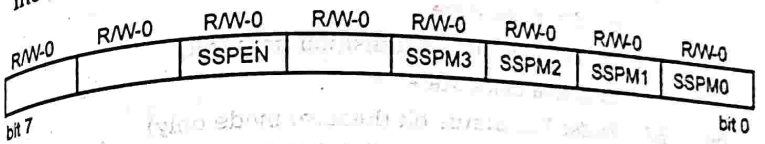


Fig. Q.19.1 Interfacing of DS1306 with PIC18 microcontroller

- Three registers are associated with SPI of MSSP module. These are SPPBUF, SSPCON1 and SSPSTAT.
- During transfer operation a byte of data is placed in SSPBUF. The SSPBUF also holds the byte received via SPI bus. SSPCON1 register is used to select SPI mode operation of the PIC18. The SSPEN bit in the SSPCON1 register is set to logic 1 to allow the use of the PIC18 pins for SPI data bus protocol. The bits SSPM3:SSPM0 of SSPCON1 register are used to select SPI master mode. See Fig. Q.19.2



Bit 5 **SSPEN** : Synchronous Serial Port Enable bit

1 = Enables serial port and configures SCK, SDO, SDI and SS as serial port pins

0 = Disables serial port and configures these pins as I/O port pins

Bit 3-0 **SSPM3:SSPM0** : Synchronous Serial Port Mode Select bits

0101 = SPI Slave mode, clock = SCK pin, SS pin control disabled, SS can be used as I/O pin

0100 = SPI Slave mode, clock = SCK pin, SS pin control enabled

0011 = SPI Master mode, clock = TMR2 output/2

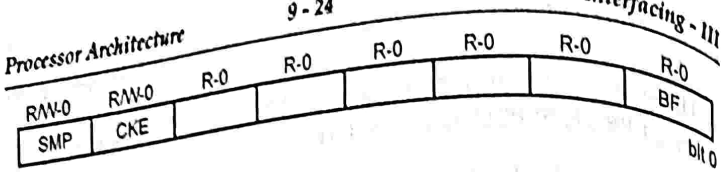
0010 = SPI Master mode, clock = FOSC/64

0001 = SPI Master mode, clock = FOSC/16

0000 = SPI Master mode, clock = FOSC/4

Fig. Q.19.2

After the selection of SSPCON1, it is necessary to select the proper bits for timing in the SSPSTAT register in Fig. Q.19.3.



bit 7 **SMP** : Sample bit
SPI Master mode :
1 = Input data sampled at end of data output time
0 = Input data sampled at middle of data output time
SPI Slave mode :
SMP must be cleared when SPI is used in Slave mode.

bit 6 **CKE** : SPI Clock Edge Select bit
1 = Transmit occurs on transition from active to Idle clock state
0 = Transmit occurs on transition from Idle to active clock state

bit 0 **BF** : Buffer Full Status bit (Receive mode only)
1 = Receive complete, SSPBUF is full
0 = Receive not complete, SSPBUF is empty

Note : Other bits of SSPSTAT register are used for I^2C module

Fig. Q.19.3

Q.20 Write a program in PIC18 C to set the time 18.50.32 in 24 hour clock mode and date for the DS1306 interface shown in Fig. Q.19.1.

Ans. :

```
#include<p18f458.h>
unsigned char SPI (unsigned char);
void DELAY (int ms);
void main()
{  SSPSTAT = 0;              // Read at middle of data output time
                             // and send on transition from Idle
                             // to active clock state
   SSPCON1 = 0x22;           // Enable Synchronous Serial Port
                             // Set SPI Master mode, clock = FOSC/64
   TRISC = 0;                // Make Port C output
   TRISCBITS.TRISC4 = 1;     // except SDI
   TRISCBITS.TRISC7 = 1;     // and RX
   PORTCbits.RC2 = 1;        // enable the RTC
   DELAY(1);                 // wait for 1 ms
```



```
   SPI (0x8f);               // load control register address
   SPI (0x00);               // clear WP bit to enable write operation
   PORTCbits.RC2 = 0;        // end of single byte write
   DELAY(1);                 // wait for 1 ms
   PORTCbits.RC2 = 1;        // begin multi-byte write
   SPI (0x80);               // load seconds register address
   SPI (0x32);               // 32 seconds
   SPI (0x50);               // 50 minutes
   SPI (0x18);               // 24-hour clock at 18 hours
   SPI (0x00);               // Monday
   SPI (0x20);               // 20th day of month
   SPI (0x03);               // March
   SPI (0x23);               // 2023
   PORTCbits.RC2 = 0;        // end of multi-byte write
   DELAY(1);                 // wait for 1 ms
}
//-- SPI Write/Read subroutine
unsigned char SPI (unsigned char myByte)
  {
    SSPBUF = myByte;         // load SSPBUF for transfer
    while (!SSPSTATbits.BF); // wait for all bits
    return SSPBUF;           // return with received byte
    }
```

SPI Read / Write subroutine

```
  unsigned char SPI (unsigned char D_Byte)
{
    SSPBUF = D_byte;         // load SSPBUF with data to be
                             // transferred
    while(!SSPSTATbits.BF);  // wait for transfer
    return SSPBUF;           // return with received byte
}
```

Delay subroutine

```
void DELAY (unsigned int)
{
  unsigned int i, j;
  for(i=0; i<t; i++)
    for(j=0; j<165; j++);

}
```

Q.21 What are the features of DS1307 RTC ? ☞ [SPPU : May-16, Marks 4]

Ans. : 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

Q.22 Explain the pin diagram of DS1307 RTC.

Ans. : The RTC DS1307 is 8 pin IC.

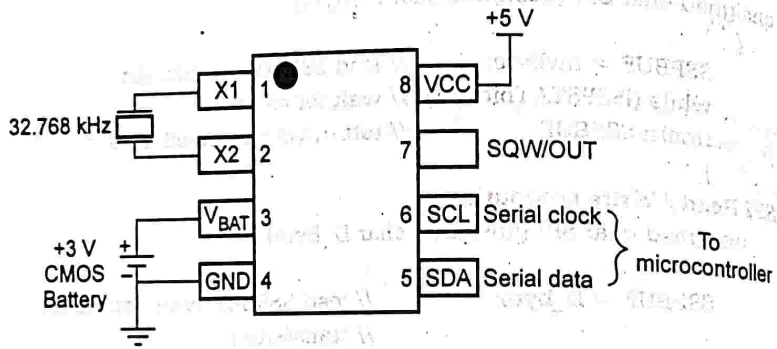


Fig. Q.22.1 DS1307 RTC pin diagram

- 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.

Q.23 What is timekeeper register ?

Ans. : 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.

Q.24 Draw the interfacing diagram of DS1307 (RTC) with PIC18. ☞ [SPPU : May-15, 16, Dec. 15, 16 Marks 4]

Ans. :

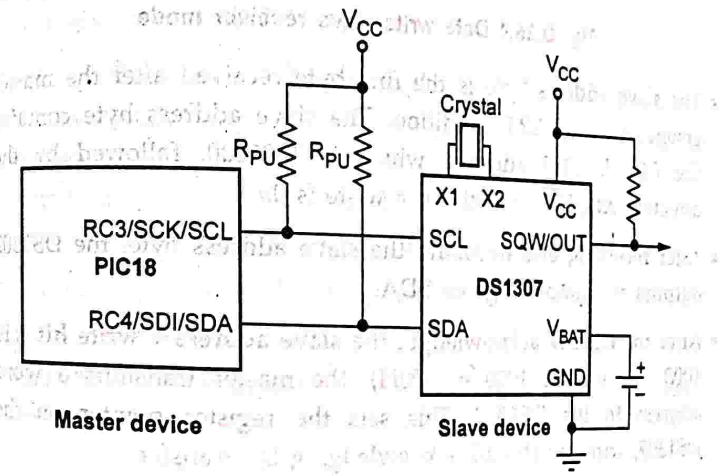


Fig. Q.24.1 Interfacing DS1307 (RTC) with PIC18

Q.25 Explain the operating modes of DS1307.

Ans. : 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. Q.25.1.

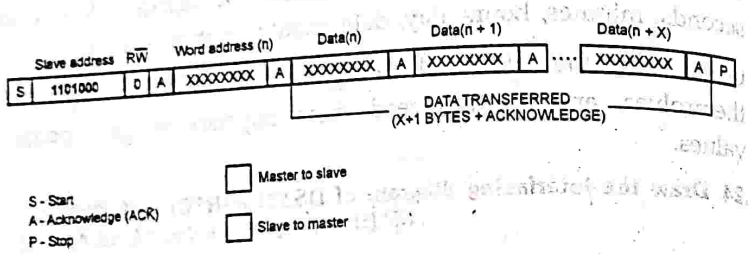


Fig. Q.25.1 Data write-slave receiver mode

- 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. Q.25.2.

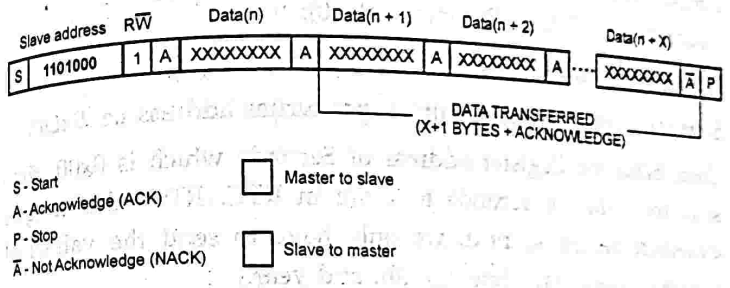


Fig. Q.25.2 Data read-slave transmitter mode

- 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.

Q.26 Explain the programming steps for RTCDS1307.

Ans. : 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.

Q.27 Interface DS1307 RTC chip with PIC18F458 using I^2C and display date and time on LCD. ☞ [SPPU : Dec.-14, 16 Marks 10]

Ans. : Fig. Q.27.1 shows the interfacing of DS1307 RTC and LCD with PIC18F458 microcontroller to display date and time on LCD.

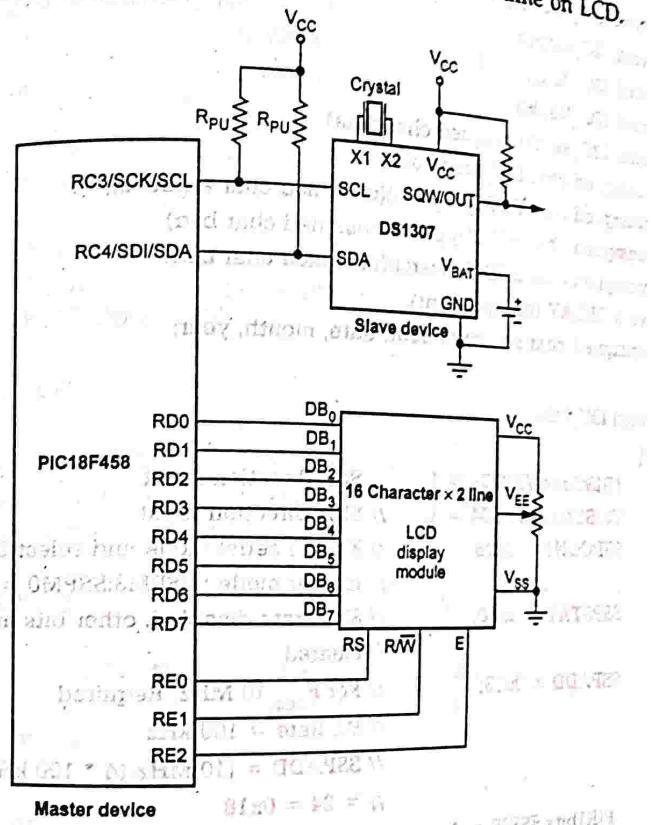


Fig. Q.27.1

```
#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 F_osc 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_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 will
                             //  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);
    }
}
```

9.4 : Interfacing of EEPROM using SPI with PIC

Q.28 Write a note on EEPROM interfacing with PIC.

Ans. : 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).
- Fig. Q.28.1 shows pin diagram for 25XX040A and Table Q.28.1 shows the pin functions.

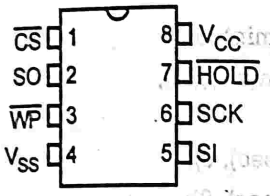


Fig. Q.28.1 Pin diagram

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

Table Q.28.1 Pin functions

- The 25XX040A contains an 8-bit instruction register. The device is accessed via the SI pin, with data being clocked in on the rising edge of SCK. The (CS) pin must be low and the (HOLD) pin must be high for the entire operation.
- Data (SI) is sampled on the first rising edge of SCK after (CS) goes low. If the clock line is shared with other peripheral devices on the SPI bus, the user can assert the (HOLD) input and place the 25XX040A in 'HOLD' mode. After releasing the (HOLD) pin, the operation will resume from the point when the (HOLD) was asserted.

Q.29 List the instruction set of 25XX040A EEPROM.

Ans. :

Instruction Name	Instruction Format	Description
READ	0000A_8 011	Read data from memory array beginning at selected address
WRITE	0000A_8 010	Write data to memory array beginning at selected address
WRDI	0000 x100	Reset the write enable latch (disable write operations)
WREN	0000 x110	Set the write enable latch (disable write operations)
RDSR	0000 x101	Read STATUS register
WRSR	0000 x001	Write STATUS register

Note : A_8 is the 9th address bit, which is used to address the entire 512 byte array.

x = don't care

Table Q.29.1 Instruction set

Q.30 Explain the read sequence of 25XX040A EEPROM.

☞ [SPPU : June-22, Marks 4]

Ans. : 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. Q.30.1.

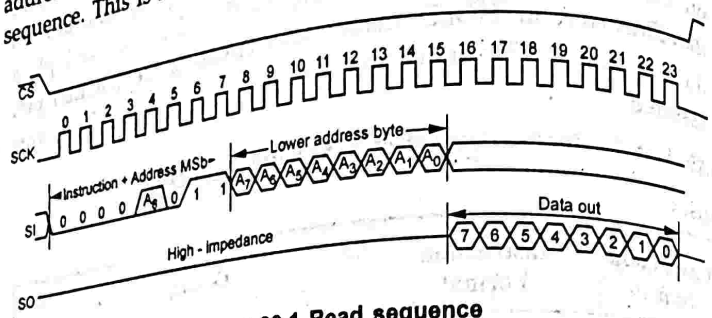


Fig. Q.30.1 Read sequence

- 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.
- 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.

Q.31 Explain the write sequence of 25XX040A EEPROM.

Ans. : 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. This is illustrated in Fig. Q.31.1.

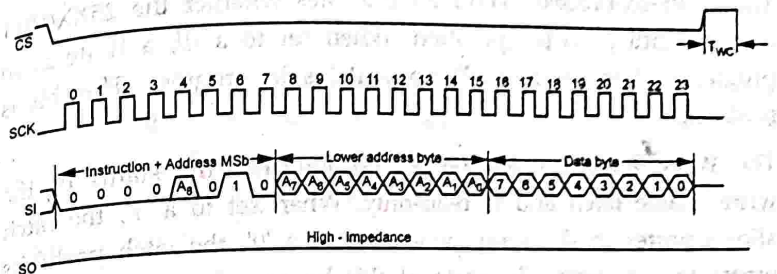


Fig. Q.31.1 Write sequence

- 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.
- Suppose a page write command attempts to write across a physical page boundary. In that case, the result is that the data wraps around to the beginning of the current page (overwriting data previously stored there).

Q.32 Write a note on RDSR.

Ans. : The Read Status Register instruction (RDSR) provides access to the STATUS register. The STATUS register may be read at any time, even during a write cycle. The STATUS register is formatted as follows :

7	6	5	4	3	2	1	0
–	–	–	–	W/R	W/R	R	R
X	X	X	X	BP1	BP0	WEL	WIP

W/R = writeable / readable. R = Read only

Fig. Q.32.1 Status register

- The **Write-In-Process (WIP)** bit indicates whether the 25XX040A is busy with a write operation. When set to a '1', a write is in progress, when set to a '0', no write is in progress. This bit is read-only.
- The **Write Enable Latch (WEL)** bit indicates the status of the write enable latch and is read-only. When set to a '1', the latch allows writes to the array, when set to a '0', the latch prohibits writes to the array. The state of this bit can always be updated via the WREN or WRDI commands regardless of the state of write protection on the STATUS register.
- The Block Protection **(BP0 and BP1)** bits indicate which blocks are currently write-protected.

Q.33 Draw and explain the interfacing of SPI EEPROM with PIC18xxx microcontroller.

Ans. : SPI allows 8-bit of data to be synchronously transmitted and received simultaneously.

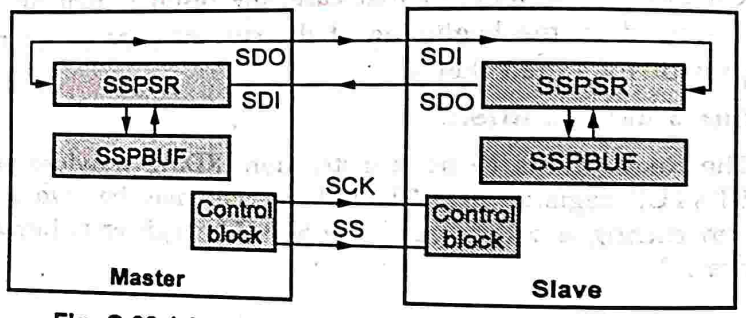


Fig. Q.33.1 Interfacing between master and slave

- 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.
- The PIC18Fxxx is connected to the SPI memory, as shown in Fig. Q.33.2. 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.

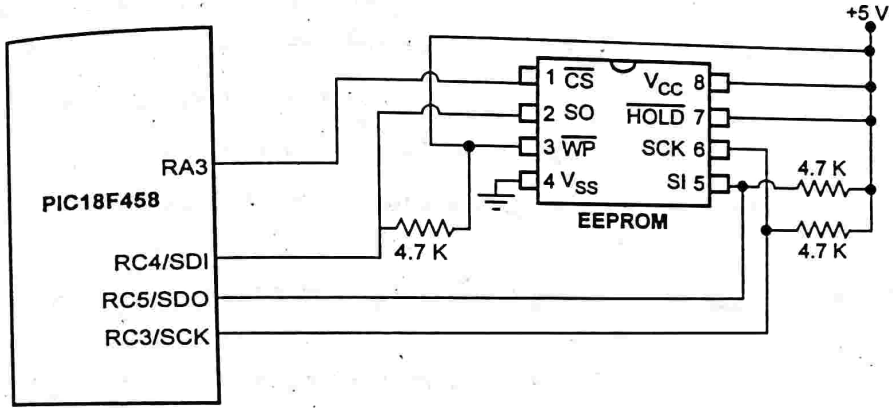


Fig. Q.33.2 PIC18Fxxx connection to the SPI memory

Q.34 Explain the purpose of registers associated with SPI module of PIC18F458.

Ans. : MSSP (Master synchronous serial port) module inside the Pic MCU supports the SPI protocol. Four registers are associated with SPI of the MSSP module of PIC18F458, which are :

- SSPBUF (Synchronous Serial Port Buffer Register)
- SSPCON1 (Synchronous Serial Port Control Register)

- SSPSTAT(Synchronous Serial Port Status Register)
- SSPSR (Shift Register But Use Cannot Access It)

SSPSR : It is the shift register used for shifting data in or out. It is not user-accessible.

SSPBUF : It is the buffer register to which data bytes are written to or read from.

END...

Unit VI

10 Current Trends in Processor Architecture

10.1 : RISC Design Philosophy

Q.1 List the features of RISC processors.

Ans. : RISC is an acronym for **Reduced Instruction Set Computer.**

- It is intended as a contrast with CISC machines which are **Complex Instruction Set Computers.**
- Most RISC processors **use hardwired control.**
- The most of the RISC processors **use 32-bit instructions.**
- They have **very few instructions.**
- The instructions are predominantly register-based.
- The **limited** (3 to 5) **addressing modes** are used by these processors.
- The memory access cycle is broken into pipelined access operations.
- This involves the use of caches and working registers.
- A large register file and separate instruction and data caches are used. This benefits internal data forwarding and eliminates unnecessary storage of intermediate results.
- Using hardwired control, the clock Cycles Per Instruction (CPI) are reduced to 1 for most RISC instructions. This is the advantage of having all instructions of equal size.
- RISC architecture is used in ARM cores.

Q.2 Explain major design rules of RISC processor. [SPPU : June-22, Marks 3]

Ans. : RISC processors follow the four major design rules.

1. Instructions

Reduced number of instructions : RISC processor provides limited number of instructions, which simplifies the design of control unit.

Simple instruction format : RISC processor uses simple instruction formats with fixed instruction length. The instruction length is aligned on word boundaries. Field locations, especially opcodes are fixed.

This architecture provides following benefits :

- With fixed fields, opcode decoding and register operand addressing can occur simultaneously.
- It simplifies the design of control unit.
- Instruction fetching is optimized since word-length units are fetched.

Hardwired instructions : With simple, one cycle instruction, there is no need for microinstructions. The machine instructions can be hardwired. These instructions are executed faster than the instructions implemented with microinstructions, since it is not necessary to access a microprogram control memory during instruction execution.

One instruction per cycle : RISC processor execute one instruction in a single cycle. In RISC processors, there is an one instruction per machine cycle. A machine cycle is defined to be the time it takes to fetch two operands from registers, performs an ALU operation, and stores the result in a register. Due to this, RISC machine instructions are not complicated and can execute about as fast as, microinstructions on CISC machines.

2. Pipelines

- Recall the concept of pipelining. The CPU contains several independent units that work in parallel. One of them fetches the instructions, and other ones decode and execute them. At any

instant, several instructions are in various stages of processing. All RISC processors provide this pipelining feature.

- In RISC processor most instructions are register to register. These instructions have the following two phases :
  - Instruction Fetch (I)
  - Execute (E)
- The instruction fetch phase fetches the instruction to be executed from memory and then execute phase performs an ALU operation with register input and output to execute the instruction.
- In case of load and store instructions, three phases are required :
  - Instruction fetch (I)
  - Execute (E)
  - Data transfer (D)
- Here, also the instruction fetch phase fetches the instruction to be executed. In execution phase address of memory is calculated and in data transfer phase actual data is transferred from register-to-memory or from memory-to-register depending upon the instruction.
- The Fig. Q.2.1 shows that how these phases can be overlapped in RISC processors to achieve the pipelining. The Fig. Q.2.1 (a) shows the two-way pipelining and the Fig. Q.2.1 (b) shows the three way pipelining. [See Fig. Q.2.1 on next page]
- In two way instruction pipelining, I and E phases of two different instructions are performed simultaneously. In three way instruction pipelining, three instructions can be overlapped. Therefore, maximum execution rate in two way instruction pipelining is twice the normal execution rate and it is three times the normal execution rate in case of three way instruction pipelining.

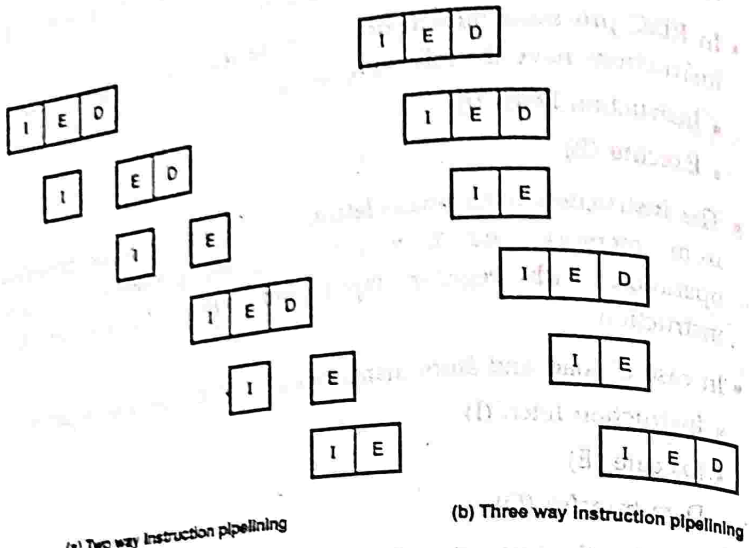


Fig. Q.2.1

3. Registers

Register to Register Operations : Risc processors have a large number of general purpose registers and they use efficient compiler technology to optimize register usage. It is the most important characteristics of RISC processor. This architecture encourages the optimization of register use, so that frequently accessed operands remain in high-speed storage.

4. Load-store architecture

RISC processors operate on data held in registers. Separate load and store instructions transfer data between the register bank and external memory.

The advantage is that the use of data items which are held in the register does not need memory access.

Q.3 Give comparison between RISC and CISC.

Ans. :

Sr. No.	Characteristics	RISC	CISC
1.	Instruction size	Fixed	Varies
2.	Instruction length	4 bytes	1, 2, 3 or 4 bytes
3.	Number of instruction	Less	More
4.	Instruction decoding	Easy (quick) to decode	Serial (slow) to decode
5.	Instruction sementics	Almost always one simple operation	Varies from simple to complex ; possibly many dependent operations per instruction
6.	Addressing modes	Complex addressing modes are synthesized in software.	Supports complex addressing modes.
7.	Instruction execution speed	Medium	Slow (depend on complexity of instruction)
8.	Instruction execution	By hardware. Simple instructions taking one cycle.	By microprogram. Complex instructions taking multiple cycles.
9.	Registers	Many, general purpose	Few, may be special purpose
10.	Memory references	Not combined with operations, i.e., load/store architecture	Combined with operations in many different types of instructions

11.	Hardware	Simple	Complicated
12.	Hardware design focus	Take the advantage of implementations with one pipeline and no microcode	Take the advantage of microcoded implementations
13.	Memory access	Rarely	Frequently
14.	Instruction format	Regular, consistent placement of fields	Field placement varies
15.	Pipelined	Highly pipelined.	Not pipelined or less pipelined.
16.	Conditional jump	Can be based on a bit anywhere in memory.	Conditional jump is usually based on status register bit.
17.	Compiler	Complicated	Simple
18.	Examples	ARM, 8051, ATMEL, AVR, etc.	Intel X86, Motorola 68000 series.

10.2 : ARM Design Philosophy

Q.4 List the features of RISC processors accepted by ARM processors. ☞ [SPPU : June-22, Marks 2]

Ans. : The features of RISC which are accepted as well as rejected by ARM processors are :

- A large uniform register file
- A load-store architecture
- **Simple addressing modes,** with all load/store addresses being determined from register contents and instruction fields only.
- **Uniform and fixed-length (32-bit) instruction fields.**
- Three-address instruction formats.

Q.5 List the features of RISC processors rejected by ARM processors. ☞ [SPPU : June-22, Marks 3]

Ans. : The features of RISC which are rejected by ARM processors are :

- **Register windows :** The main problem with register windows is the large chip area occupied by the large number of registers. This feature is therefore rejected by ARM processors to reduce cost.
- **Delayed branches :** Original ARM does not use delayed branch since they make exception handling more complex.
- **Single cycle execution of all instructions**
  ARM processor executes most of the data processing instructions in a single cycle. However, many other instructions need multiple clock cycles for their execution.

10.3 : Introduction to ARM Processor

Q.6 Write a note on ARM nomenclature.

Ans. : ARM Nomenclature identifies individual processors and provides basic information about the feature set.

- The letters or words after "ARM" are used to indicate the features of a processor.

ARMxyzTDMIEJF-S

- x - Family or series
- y - Memory Management/Protection Unit
- z - Cache
- T - 16 bit Thumb decoder
- D - JTAG Debugger
- M - Fast Multiplier
- I - Embedded In-circuit Emulator (ICE) Macrocell
- E - Enhanced Instructions for DSP (assumes TDMI)
- J - Jazelle (for accelerated JAVA execution)
- F - Vector Floating-point Unit

- S - Synthesizable Version
- **T - Thumb Instruction Set :** ARM Processors support both the 32-bit ARM Instruction Set and 16-bit Thumb Instruction Set. The original 32-bit ARM Instructions consist of 32-bit opcodes which turns out to be a 4-byte binary pattern. The 16-bit Thumb Instructions consist of 16-bit opcodes or 2-byte binary pattern to improve the code density.
- **D - JTAG Debug :** JTAG is a serial protocol used by ARM to transfer the debug information between the processor and the test equipment.
- **M - Fast Multiplier :** Older ARM Processors used a small and simple multiplier unit. This multiplier unit required more clock cycles to complete a single multiplication. With the introduction of Fast Multiplier unit, the clock cycles required for multiplication are significantly reduced and modern ARM Processors are capable of calculating a 32-bit product in a single cycle.
- **I - Embedded ICE Macrocell :** ARM Processors have on-chip debug hardware that allows the processor to set breakpoints and watchpoints.
- **E - Enhanced Instructions :** ARM Processors with this mode will support the extended DSP Instruction Set for high performance DSP applications. With these extended DSP instructions, the DSP performance of the ARM Processors can be increased without high clock frequencies.
- **J - Jazelle :** ARM Processors with Jazelle Technology can be used in accelerated execution of Java bytecodes. Jazelle DBX or Direct Bytecode eXecution is used in mobile phones and other consumer devices for high performance Java execution without affecting memory or battery.
- **F - Vector Floating-point Unit :** The Floating Point Architecture in ARM Processors provide execution of floating point arithmetic operations. The Dynamic Range and Precision offered by the

Floating Point Architecture in ARM Processors are used in many real time applications in the industrial and automotive areas.

- **S - Synthesizable :** The ARM Processor Core is available as source code. This software core can be compiled into a format that can be easily understood by the EDA Tools. Using the processor source code, it is possible to modify the architecture of the ARM Processor.

Q.7 Write a note on ARM architecture enhancements.

Ans. : Table Q.7.1 shows the significant architecture enhancements from the original architecture version 1 to the current version 6 architecture.

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor 26-bit addressing
ARMv2	ARM2	32-bit multiplier 32-bit coprocessor support
ARMv2a	ARM3	On-chip cache Atomic swap instruction Coprocessor 15 for cache management
ARMv3	ARM6 and ARM7DI	32-bit addressing Separate cpsr and spsr New modes-undefined instruction and abort MMU support-virtual memory
ARMv3M	ARM7M	Signed and unsigned long multiply instructions

ARMv4	StrongARM	Load-store instructions for signed and unsigned halfwords/bytes New mode-system Reserve SWI space for architecturally defined operations 26-bit addressing mode no longer supported
ARMv4T	ARM7TDMI and ARM9T	Thumb
ARMv5TE	ARM9E and ARM10E	Superset of the ARMv4T Extra instructions added for changing state between ARM and Thumb Enhanced multiply instructions Extra DSP-type instructions Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions Unaligned and mixed endian data handling New multimedia instructions

Table Q.7.1 : ARM architecture enhancements

10.4 : ARM Processor Families - ARM 7, ARM 9 and ARM 11

Q.8 List the features of ARM7 family processors.

☞ **[SPPU : June-22, Marks 4]**

Ans. : Features of ARM7 are :

1. **Pipeline depth :** 3 stage (Fetch, Decode, Execute)
2. **Operating frequency :** 80 MHz
3. **Power consumption :** 0.06 mW/MHz.

4. **MIPS/MHz :** 0.97
5. **Architecture used :** Von-Neumann
6. **MMU/MPU :** Not present
7. **Cache memory :** Not present
8. **Jazelle instruction :** Not present
9. **Thumb instruction :** Yes (16 bit instruction set)
10. **ARM instruction set :** Yes (32 bit)
11. **ISA (Instruction set architecture) :** V4T (4 TH Version)
12. **Interrupt Controller :** Not Present
13. **ISR entry :** Non Deterministic ISR entry
14. **Power management :** No in built Power Management
15. **Instruction Set performance v/s code size :** Optimal performance code size balance requires interworking between ARM & Thumb code
16. **Ease of application porting from one device to another :** Lack of standardization inhibits application porting.

Q.9 List the features of ARM9 family processors.

Ans. : Features of ARM9 are :

- **Pipeline Depth :** 5 stage (Fetch, Decode, Execute, Decode, Write)
- **Operating frequency :** 150 MHz
- **Power Consumption :** 0.19 mW/MHz
- **MIPS/MHz :** 1.1
- **Architecture used :** Harvard. It separates the data D and instruction I buses.
- **MMU/MPU :** Present
- **Cache Memory :** Present (separate 16 K/8 K)
- **ARM/ Thumb Instruction :** Support both
- **ARM920T**

- The first processor in the ARM9 family
- It includes a separate D + I cache and an MMU.
- Provides virtual memory support to operating systems.
- It executes the architecture v4T instructions
- ARM922T is a variation on the ARM920T with half the D + I cache size.

- **ARM940T**
  - It includes a smaller D + I cache and an MPU.
  - It is designed for applications that do not require a platform operating system.
  - It executes the architecture v4T instructions
- ARM946E-S and ARM966E-S
  - Both execute architecture v5TE instructions.
  - They support the optional Embedded Trace Macrocell (ETM), which allows a developer to trace instruction and data execution in real time on the processor.
  - The ARM946E-S includes TCM (Tightly Coupled Memory), cache, and an MPU. The sizes of the TCM and caches are configurable. It is designed for use in embedded control applications that require deterministic real-time response.
  - On the other hand, the ARM966E does not have the MPU and cache extensions but does have configurable TCMs.
- **ARM926EJ-S**
  - It is synthesizable processor core, announced in 2000.
  - It is designed for use in small portable Java-enabled devices such as 3G phones and Personal Digital Assistants (PDAs).
  - Supports the Jazelle technology, which accelerates Java bytecode execution.
  - It features an MMU, configurable TCMs, and D + I caches with zero or nonzero wait state memories.

Q.10 List the features of ARM11 family processors.

Ans. : Features of ARM11

- **Pipeline depth :** 8 stage pipeline with separate loadstore and arithmetic pipelines.
- **Operating frequency :** 335 MHz.
- **Power Consumption :** 0.4 mW/MHz.
- **MIPS/MHz :** 1.2
- **Architecture used :** Harvard
- **MMU/MPU :** Present
- **Multiplier unit :** 16 × 32 (16 bits of 32-bit size register)
- **Cache Memory :** Present (4 - 64 K size)
- **ARM1136J-S**
  - It was announced in 2003.
  - Designed for high performance and power efficient applications.
  - It executes architecture ARMv6 instructions.
  - Supports Single Instruction Multiple Data (SIMD) extensions for media processing, specifically designed to increase video processing performance.
  - The ARM1136JF-S is an ARM1136J-S with the addition of the vector floating-point unit for fast floating-point operations.
- Supports the thumb instruction set-memory BW & Size requirements reduces by up to 35 %
- Supports Jazelle Technology for efficient embedded JAVA execution
- Supports the DSP extensions
- Supports ARM Trust-Zone Technology for on chip security
- Physically tagged caches to improve OS context switch performance.

- Tightly coupled memories for real-time applications.

Q.11 Give the comparison between ARM7, ARM9 and ARM11 family processors. ☞ [SPPU : Dec.-22, Marks 6]

Ans. : Table Q.11.1 shows a comparison between the ARM7, ARM9, and ARM11 cores.

Processor attribute	ARM7	ARM9	ARM11
Pipeline depth	3-stage	5-stage	8-stage
Typical MHz	80	150	335
mW/MHz	0.06	0.19 (+ cache)	0.4 (+ cache)
MIPS/MHz	0.97	1.1	1.2
Architecture	Von neumann	Harvard	Harvard
Multiplier	8 × 32	8 × 32	16 × 32
MMU/MPU	Absent	Present	Present
Cache Memory	Absent	Present	Present
Configurable TCM	Absent	Present	Present
Jazelle Technology	Absent	Present	Present

Table Q.11.1

10.5 : Features and Advantages of ARM Processor

Q.12 Give the features and advantages of ARM processor.

Ans. : The features of ARM microcontroller are as follows :

- Consists lof a large uniform register file.
- Supports load-store architecture.
- Uses simple addressing modes.
- Contains a reduced number of instructions.
- Instructions perform simple but powerful operations which can be executed in a single cycle. If the complicated

operations, such as division, are to be performed, the compiler or programmer synthesizes them by combining several simple instructions.

- It has **uniform and fixed-length (32-bit) instruction fields.**
- **Each instruction is a fixed length.** This **allows the pipeline** to fetch future instructions before decoding the current instruction. In contrast, the CISC processor contains the instructions of variable size, complex and take more cycles to execute. So in CISC complexity is in processor hardware whereas in RISC, complexity is in compiler. The uniform and fixed-length instruction fields simplify the instruction decoding.
- Most instructions have **three-address instruction format.**
- **Does not use delayed branch** since they make exception handling more complex.
- **Does not use register windows** to keep chip area small and hence the cost.
- It has **control over both the Arithmetic Logic Unit (ALU) and shifter** in every data-processing instruction to maximize the use of an ALU and a shifter.
- Supports **auto-increment and auto-decrement addressing modes** to optimize program loops.
- Supports **Load and Store Multiple instructions** to maximize data throughput.
- Supports conditiona execution of all instructions to maximize execution throughput.

10.6 : Suitability of ARM Processor in Embedded Applications

Q.13 List the features of ARM instructions suitable for embedded applications. ☞ [SPPU : Dec.-13, Marks 5]

Ans. : **Features :**

- **Variable cycle execution for certain instructions :** Some ARM instructions like load-store-multiple instructions vary in the number of execution cycles depending upon the number of

registers being transferred. Load-store-multiple instructions transfer data on sequential memory addresses, which increases performance since sequential memory accesses are often faster than random accesses. Multiple register transfers also improve the code density.

- **Inline barrel shifter to improve core performance and code density** : The ARM arithmetic logic unit has a barrel shifter that is capable of shift and rotate operations. This inline barrel shifter preprocesses one of the input registers before it is used by an instruction. This expands the capability of many instructions to improve core performance and code density.
- **Thumb 16-bit instruction set** : The Thumb instruction set consists of 16-bit instructions that act as a compact shorthand for a subset of the 32-bit instructions of the standard ARM. These instructions permit the ARM core to execute either 16- or 32-bit instructions. The 16-bit instructions improve code density by about 30 % over 32-bit fixed-length instructions.
- **Conditional execution** : These instructions are executed when a specific condition has been satisfied. This feature improves performance and code density by reducing branch instructions.
- **Enhanced instructions** : ARM instruction set also supports the enhanced Digital Signal Processor (DSP) instructions (fast 16×16-bit multiplier operations and saturation). These instructions allow ARM processor to serve as a combination of a processor plus a DSP.

10.7 : ARM7 Dataflow Model

Q.14 With a neat block diagram, explain data flow model of ARM7 core. **[SPPU : June-22, Marks 6]**

Ans. : Fig. Q.14.1 shows the basic structure of ARM7 core and how data moves between its different parts.

- The ARM7 core dataflow model shown in Fig. Q.14.1 is Von Neumann implementation of the ARM.
- Since ARM7 processor is basically a RISC processor, it uses a **load-store architecture**. This means, it has two instruction types,

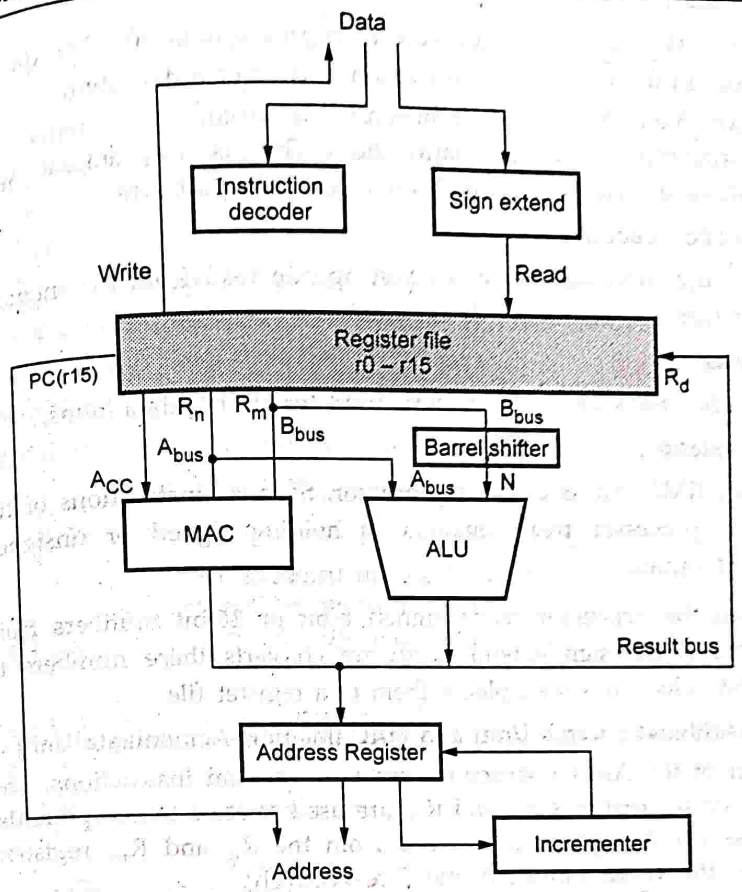


Fig. Q.14.1 ARM7 core dataflow model

load and store, for transferring data in and out of the processor respectively.

- **LOAD** : This instruction copies data from memory to registers in the processor core.
- **STORE** : This instruction copies data from registers in the processor core to memory.

- The ARM processor instruction set does not include the instructions that directly manipulate data in memory. The data processing is carried out only in registers.

Data bus :

- The data enters the ARM7 core through the data bus. The data is either in the form of an instruction opcode or a data item.
- Since Von Neumann architecture is used, data items and instructions share the same bus. This is in contrast with Hardvard architecture which uses two different buses.

Instruction decoder :

- This unit decodes the instruction opcode read from the memory and then the instruction is executed.

Register file :

- This is a bank of 32-bit registers used for storing data items.

Sign extend :

- The ARM7 core is a 32-bit processor. So most instructions of the ARM processor treat registers as holding signed or unsigned 32-bit values.
- When the processor reads signed 8-bit or 16-bit numbers from memory, the sign extend hardware converts these numbers to 32-bit values and then places them in a register file.

ALU (Arithmetic Logic Unit) and MAC (Multiply-Accumulate Unit) :

- Most of the ARM instructions are two operand instructions. The two source registers R_n and R_m are used to store these operands. These source operands are read from the R_n and R_m registers using the internal buses A and B respectively.
- The ALU or MAC reads the operand values from R_n and R_m registers via A and B buses respectively, performs the operation and stores the computed result via internal C bus in destination register, R_d and then to the register file.
- The load and store instructions generate address using ALU and stores it in the address register.

Address register :

- This holds the address generated by the load and store instructions and places it on the address bus.

Barrel shifter :

- The contents of the R_m register alternatively can be preprocessed in the barrel shifter before applying as an input to the ALU.
- A wide range of expressions and addresses can be calculated using the barrel shifter and ALU.

Incrementer

- For load and store instructions, the incrementer updates the contents of the address register before the processor core reads or writes the next register value from or to the consecutive memory location.
- The processor core continues the execution of instruction. Only when an exception or interrupt occurs, the normal execution flow is changed.

10.8 : Programmers Model

Q.15 Draw and explain the ARM programmer's model.

☞ [SPPU : Dec.-22, Marks 5]

Ans. : Fig. Q.15.1 shows the programming model of ARM processor. (Refer Fig. Q.15.1 on next page).

- The ARM processor has a total of 37 registers. All registers are 32-bits wide. They can be classified into two groups as,
  - General purpose registers
  - Special purpose registers.

General Purpose Registers

- Registers r0 to r12 are used as general purpose registers. Depending upon the context, registers r13 to r15 can also be used as general purpose registers.
- The general purpose registers hold either data or an address.

Special Purpose Registers

- Registers **r13 to r15, CPSR** (Current Program Status Register) and **SPSR** (Saved Program Status Register) are the special purpose registers.

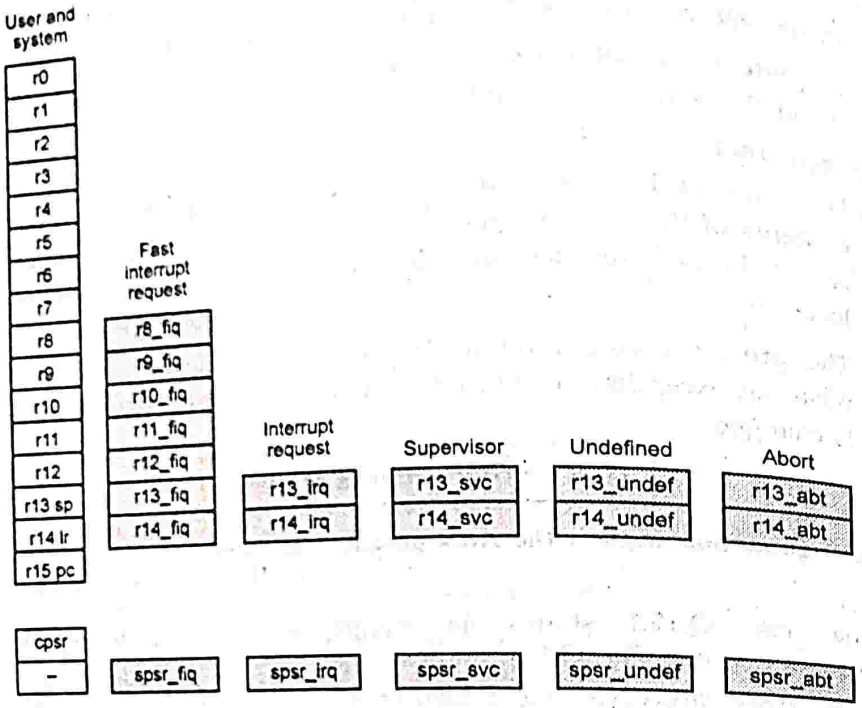


Fig. Q.15.1 Programming model of ARM processor

Registers r13 to r15

- In user mode, registers r13 to r15 are labeled as r13 sp, r14 lr and r15 pc respectively to differentiate them from other registers. The functions of these registers are given below.
  - **Stack pointer (r13 sp) :** Register r13 is the stack pointer. It stores the top of the stack in the current processor mode.
  - **Link register (r14 lr) :** Register r14 is the link register. The processor stores the return address in this register when a subroutine is called.

- **Program counter (r15 pc) :** Register r15 is the program counter and stores the address of the next instruction to be fetched from the memory by the processor.
  - It is used in most instructions as a pointer to the instruction which is two instructions after the instruction being executed.
  - All ARM instructions are four bytes long (one 32-bit word) and are always aligned on a word boundary. This means that the bottom two bits of the PC are always zero, and therefore the PC contains only 30 non-constant bits.
  - It can often be used in place of one of the general-purpose registers r0 to r14, and is therefore considered one of the general-purpose registers. However, there are also many instruction-specific restrictions or special cases about its use. Usually, the instruction is unpredictable if r15 is used in a manner that breaks these restrictions.

The Unbanked Registers r0 - r7

- Registers r0 to r7 are **unbanked registers.** This means that each of them refers to the same 32-bit physical register in all processor modes.
- They are completely general-purpose registers, with no special uses implied by the architecture, and can be used wherever an instruction allows a general-purpose register to be specified.

The Banked Registers, r8 - r14

- Registers r8 to r14 are **banked registers.**
- The physical register referred to by each of them depends on the current processor mode. Where a particular physical register is intended, without depending on the current processor mode, a more specific name (as described below) is used. Almost all instructions allow the banked registers to be used wherever a general-purpose register is allowed.

- Out of 37 registers, 20 registers which are shown shaded in Fig. 10.8.1 are the banked registers. Fig. Q.15.1 also shows which banked registers are used in which mode. Banked registers of a particular mode are denoted by, r number_mode.
- For example, supervisor, mode has banked registers r13_svc, r14_svc and spsr_svc.
- Registers r8 to r12 have two banked physical registers each. The first group of physical registers are referred to as r8_usr to r12_usr and the second group as r8_fiq to r12_fiq. The r8_usr to r12_usr group is used in all processor modes other than FIQ mode, and the other is used in FIQ mode.
- Registers r13 and r14 have six banked physical registers each. One is used in User and System modes, while each of the remaining five is used in one of the five exception modes.
- The registers r0 to r13 are orthogonal. This means, any instruction which you can apply to r0, you can equally well apply to any of the r1 to r13 registers. This is not the case with r14 and r15 registers.

10.9 : CPSR and SPSR Registers

Q.16 What are CPSR and SPSR registers ? ☞ **[SPPU : Dec.-22, Marks 6]**

Ans. : The **current program status register (cpsr)** is accessible in all processor modes. It contains condition code flags, interrupt disable bits, the current processor mode, and other status and control information.

- Each exception mode also has a **saved program status register (spsr)**, that is used to preserve the value of the cpsr when the associated exception occurs.

Note User mode and System mode do not have an SPSR, because they are not exception modes. All instructions which read or write the SPSR are UNPREDICTABLE when executed in User mode or System mode.

- Fig. Q.16.1 shows the format of the cpsr and spsr.

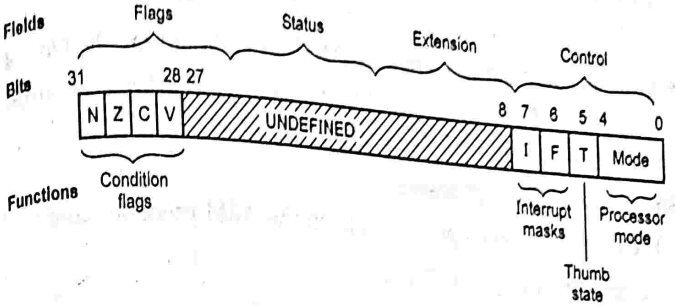


Fig. Q.16.1 Format of cpsr and spsr

Q.17 Explain the function of control flags of CPSR register.

Ans. : Control flags (Bits 0-7)

- The control bits change when an exception arises and can be altered by software only when the processor is in a privileged mode.

Bits 0 - 4 (Mode Select Bits) : Processor modes

- These bits determine the processor mode as shown in Table Q.17.1.

Processor mode	Mode Select Bits [4 : 0]
Abort	10111
Fast interrupt request	10001
Interrupt request	10010
Supervisor	10011
System	11111
Undefined	11011
User	10000

Table Q.17.1 Processor mode

Bit 5 (Thumb State Bit) :

- This bit gives the state of the core. The state of the core determines which instruction set is being executed.
- There are three instruction sets, ARM, Thumb and Jazelle. One of the three instruction set is active when the processor is in ARM state, Thumb state and Jazelle state respectively.

Bits 6 and 7 (Interrupt Masks)

- There are two interrupts available on the ARM processor core :
  - Interrupt Request (IRQ) and
  - Fast Interrupt Request (FIQ).
- These are maskable interrupts and their masking is controlled by bits 6 and 7 of cpsr. Bit 6(F) controls FIQ and bit 7(I) controls IRQ.
- When the bit is set to binary 1, the corresponding interrupt request is masked and when bit is 0, the interrupt is available.

Q.18 Explain the condition code flags of CPSR register.

Ans. : These flags in the cpsr can be tested by most instructions to determine whether the instruction is to be executed.

- The condition code flags are usually modified by : Execution of a comparison instruction (CMN, CMP, TEQ or TST).

Bit 28 (Overflow flag, V)

- It is set in one of two ways :
  - For an addition or subtraction, V is set to 1 if signed overflow occurred, regarding the operands and result as two's complement signed integers.
  - For non-addition/subtractions, V is normally left unchanged

Bit 29 (Carry flag, C)

- It is set in one of four ways :
  - For an addition, including the comparison instruction CMN, C is set to 1 if the addition produced a carry (that is, an unsigned overflow), and to 0 otherwise.

- For a subtraction, including the comparison instruction CMP, C is set to 0 if the subtraction produced a borrow (that is, an unsigned underflow), and to 1 otherwise.
- For non-addition/subtractions that incorporate a shift operation, C is set to the last bit shifted out of the value by the shifter.
- For other non-addition/subtractions, C is normally left unchanged.

Bit 30 (Zero flag, Z)

- It is set to 1 if the result of the instruction is zero (which often indicates an *equal* result from a comparison), and to 0 otherwise.

Bit 31 (Negative flag, N)

- It is set to bit 31 of the result of the instruction. If this result is regarded as a two's complement signed integer, then N = 1 if the result is negative and N = 0 if it is positive or zero.

10.10 : Modes of Operation

Q.19 Explain different processor modes of ARM controller.

☞ [SPPU : Dec.-14, 15, 22, Marks 7]

Ans. : In the ARM7, there are seven operating modes. These modes are protected or exception modes which have associated interrupt sources and their own register set.

1. **Supervisor mode (Default) :** This is protected mode for running system level code to access hardware or run OS calls. The ARM7 enters this mode after reset.
2. **FIQ (Fast Interrupt reQuest) :** This mode supports high speed interrupt handling.
3. **IRQ (Interrupt ReQuest) :** This mode supports all other interrupt sources in a system.
4. **Abort :** If an instruction or data is fetched from an invalid memory location, an abort exception will be generated.
5. **Undefined :** If a fetched opcode is not an ARM instruction, an undefined instruction exception will be generated.

6. **User :** This mode is used to run the application code. In the user mode we cannot change the contents of CPSR (Current Program Status Register) and modes can only be changed when an exception is generated. This mode is also known as **Unprivileged mode.**
7. **System :** This mode is used for running operating system tasks. It uses the same registers as user mode.

- All the above modes, except user mode, are privilege modes.

10.11 : Difference between PIC and ARM

Q.20 Give comparison between PIC and ARM.

Ans. : Table Q.20.1 gives the difference between PIC and ARM.

Parameter	PIC	ARM
Bus width	8/16/32-bit	32-bit mostly also available in 64-bit
Communication protocols	PIC, UART, USART, LIN, CAN, Ethernet, SPI, I2S	UART, USART, LIN, I2C, SPI, CAN, USB, Ethernet, I2S, DSP, SAI (serial audio interface), IrDA
Speed	4 Clock/instruction cycle	1 clock/ instruction cycle
Memory	SRAM, FLASH	Flash, SDRAM, EEPROM
ISA	Some feature of RISC	RISC
Memory architecture	Harvard architecture	Modified Harvard architecture
Power consumption	Low	Low
Families	PIC16,PIC17, PIC18, PIC24, PIC32	ARMv4,5,6,7 and series

Community	Very good	Vast
Manufacturer	Microchip average	Apple, Nvidia, Qualcomm, Samsung Electronics, and TI etc.
Cost	Average	Low
Popular microcontrollers	PIC18fXX8, PIC16f88X, PIC32MXX	LPC2148, ARM Cortex-M0 to ARM Cortex-M7, etc.

Table Q.20.1 Difference between PIC and ARM

END...

JUNE - 2022 [5869]-287 | Solved Paper

Course 2019

Time : $2\frac{1}{2}$ Hours] [Maximum Marks : 70

Instructions to the candidates :

1) *Answer Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8.*
2) *Neat diagrams must be drawn wherever necessary.*
3) *Figures to the right indicate full marks.*
4) *Assume suitable data, if necessary.*

Q.1 a) *Discuss the steps in executing interrupts in PIC 18 microcontroller.* **(Refer Q.5 of Chapter - 6)** [7]

b) *Explain PIR (Peripheral Interrupt Request Register) IPR (Peripheral Interrupt Priority Register).* **(Refer Q.10 of Chapter - 6)** [8]

c) *Explain function of following LCD pins :*
i) RS ii) RW iii) EN **(Refer Q.5 of Chapter - 7)** [3]

OR

Q.2 a) *Explain the interrupt structure of PIC18 along with IVT.* **(Refer Q.6 and Q.4 of Chapter - 6)** [8]

b) *Draw an interfacing diagram for 4 × 4 matrix keyboard with PICI8F microcontroller and explain it.* **(Refer Q.10 of Chapter - 7)** [6]

c) *Illustrate the use of following bits of INTCON2 register :* [4]
i) INTEDG1 ii) TMR0IP **(Refer Q.9 of Chapter - 6)**

Q.3 a) *List the steps involved in programming PIC microcontroller in capture mode.* **(Refer Q.6 of Chapter - 8)** [6]

b) *Explain RS232 standard with suitable diagram.* **(Refer Q.8 of Chapter - 5)** [6]

c) *Write short note on SPI protocol.* **(Refer Q.18 of Chapter - 5)**

OR [5]

Q.4 a) *Write the steps involved in programming compare mode of CCP1 module in PIC18F458.* **(Refer Q.9 of Chapter - 8)** [6]

b) *Write short note on I^2C bus.* **(Refer Q.11 of Chapter - 5)** [6]

c) *Distinguish between synchronous and asynchronous serial communication.* **(Refer Q.6 of Chapter - 5)** [5]

Q.5 a) *Explain in detail the functions of ADCON0 SFR of PIC18 microcontroller.* **(Refer Q.2 of Chapter - 9)** [7]

b) *Draw and explain the interfacing diagram of DAC0808 with PIC18FXXX.* **(Refer Q.7 of Chapter - 9)** [7]

c) *Explain the significance of ADC's EOC and SOC signals.* [4]

Ans. : The SOC (Start of Conversion) signal is an input signal for ADC. It initiates A to D conversion. After completion of conversion ADC activates EOC (End of Conversion) signal. This signal is used by the processor to read the converted value.

OR

Q.6 a) *Draw and explain the interfacing of LM34/LM35 with PIC18FXX for temperature measurement using on - chip ADC.* **(Refer Q.9 of Chapter - 9)** [8]

b) *A PIC 18 is connected to the 4 MHz crystal oscillator. Calculate the conversion time if we want to use only ADCS bits of the ADCON0 register.* [6]

Ans. :

AD Clock Source (TAD)		
Operation	**ADCS2: ADCS0**	**4 MHz**
2 TOSC	000	500 ns

4 TOSC	100	1.0 µs
8 TOSC	001	2.0 µs
16 TOSC	101	4.0 µs
32 TOSC	010	8.0 µs
64 TOSC	110	16.0 µs
RC	011	3 - 9 µs

c) *List out the steps necessary for reading from EEPROM of PIC18.* (Refer Q.30 of Chapter - 9) [4]

Q.7 a) *Draw and explain ARM core dataflow model.* (Refer Q.14 of Chapter - 10) [6]

b) *What are the main features of ARM7 architecture ? How it is different from pure RISC processor ?* (Refer Q.3 and Q.5 of Chapter - 10) [6]

c) *Describe the major design rules of risc philosophy ? list the features of risc processor accepted by ARM processor.* (Refer Q.2 and Q.4 of Chapter - 10) [5]

OR

Q.8 a) *Draw and explain the ARM family core architecture.* (Refer Q.14 of Chapter - 10) [6]

b) *Why does ARM use CPSR ? Explain the program status register ?* (Refer Q.16 of Chapter - 10) [7]

c) *Draw and explain programmers model of ARM processor.* (Refer Q.15 of Chapter - 10) [4]

DECEMBER - 2022 [5925]-270 Solved Paper

Course 2019

Time : $2\frac{1}{2}$ Hours] [Maximum Marks : 70

Q.1 a) *Write a short note on interrupt structure of PIC18 microcontroller.* (Refer Q.6 of Chapter - 6) [7]

b) *Justify the importance of Interrupt Control Register (INTCON) in PIC18F.* (Refer Q.7 of Chapter - 6) [7]

c) *Explain RCIF and TXIF flag in programming serial communication interrupt.* (Refer Q.10 of Chapter - 6) [4]

Ans. : In programming serial communication RCIF flag is used to check whether reception is complete or not. TXIF flag is used to check whether transmit buffer is full or empty.

OR

Q.2 a) *Draw an interfacing diagram for 16X2 LCD with PIC18F microcontroller and explain its working.* (Refer Q.6 and Q.8 of Chapter - 7) [8]

b) *Write the short note on :* [6]

i) ISR (Refer Q.1 of Chapter - 6)

ii) IVT (Refer Q.4 of Chapter - 6)

c) *Differentiate between interrupt and polling.* (Refer Q.2 of Chapter - 6) [4]

Q.3 a) *Explain the working of compare mode of CCP module in PIC18F with block diagram.* (Refer Q.8 of Chapter - 8) [6]

b) *Write short note on SPI protocol.* (Refer Q.18 of Chapter - 5) [6]

c) *Distinguish between synchronous and asynchronous serial communication.* **(Refer Q.6 of Chapter - 5)** [5]

OR

Q.4 a) *List the steps involved in programming PIC microcontroller in capture mode.* **(Refer Q.6 of Chapter - 8)** [6]

b) *Write short note on I2C bus.* **(Refer Q.15 of Chapter - 5)** [6]

c) *Explain USART module in PIC18F.* **(Refer Q.23 of Chapter - 5)** [5]

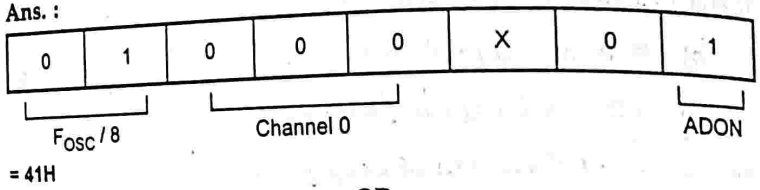
Q.5 a) *Explain in detail the functions of ADCON1 SFR of PIC18 microcontroller.* **(Refer Q.2 of Chapter - 9)** [7]

b) *State the features of RTC. Explain function of following pins of DS1306.* **(Refer Q.11 and Q.12 of Chapter - 9)** [7]

i) SERMODE ii) SDI iii) SDO

c) *Find the value for the ADCON0 register if we want $F_{OSC}/8$, channel 0 and ADON on.* [4]

Ans. :



= 41H

OR

Q.6 a) *Draw and explain the interfacing diagram of DAC0808 with PIC18FXXX.* **(Refer Q.7 of Chapter - 9)** [8]

b) *Assuming that R = 5 Ω and I_{ref} = 2 mA for DAC0808, calculate V_{out} forthe following binary inputs :* [6]

i) 10011001 (99H)

ii) 11001000 (C8H)

iii) 10001000 (88H)

Ans. :

i) $$I_0 = \frac{2\text{mA}\times(99)_H}{256} = \frac{2\text{ mA}\times153}{256} = 1.1953125\text{ mA}$$

$$V_0 = 5\text{K}\times1.1953125\text{ mA} = \mathbf{5.9765625\ V}$$

ii) $$I_0 = \frac{2\text{mA}\times(C8)_H}{256} = \frac{2\text{mA}\times200}{256} = 1.5625\text{ mA}$$

$$V_0 = 5\text{K}\times1.5625\text{ mA} = \mathbf{7.8125\ V}$$

iii) $$I_0 = \frac{2\text{mA}\times(88)_H}{256} = \frac{2\text{ mA}\times136}{256} = 1.0625\text{ mA}$$

$$V_0 = 5\text{ K}\times1.0625\text{ mA} = \mathbf{5.3125\ V}$$

c) *Explain in detail the functions of following flags related to onboard ADC of PIC18 microcontroller.* **(Refer Q.2 of Chapter - 9)** [4]

i) GO/DONE

ii) ADON

Q.7 a) *Describe the ARM bus technology.* [6]

Ans. : Embedded systems make use of different bus technologies than x86 PCs. The Peripheral Component Interconnect (PCI) bus, the most common PC bus technology, connects devices like video cards and hard disc controllers to the x86 processor bus. This technology is external or off-chip and is built into a PC's motherboard.

- Embedded devices, on the other hand, use an on-chip bus that is internal to the chip and allows different peripheral devices to communicate with an ARM core. There are two types of devices connected to the bus. The ARM processor core is a bus master, which means it can initiate a data transfer with another device on the same bus. Peripherals are typically bus slaves-logical devices that can only respond to a transfer request from a bus master device.

- A bus has two levels of architecture. The first is a physical level that addresses electrical properties and bus width (16, 32, or 64 bits). The second level is concerned with protocols, which are the logical rules that govern communication between the processor and a peripheral.
- ARM implements the electrical characteristics of the bus, but it also specifies the bus protocol on a regular basis.

b) *Compare the ARM7, ARM9 and ARM11 processors.*
(Refer Q.11 of Chapter - 10) [6]

c) *Discuss the different exceptions in ARM processor.* [5]

Ans. : Different exceptions in ARM processor are as follows :

1. **Reset :** Occurs when power is applied to the processor.
2. **Undefined instruction :** Occurs when the processor cannot decode an instruction.
3. **Software interrupt :** Occurs when you execute a SWI instruction.
4. **Prefetch abort :** Occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.
5. **Data abort :** It is similar to prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.
6. **Interrupt request :** Used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the CPSR.
7. **Fast interrupt request :** It is similar to the interrupt request but is reserved for hardware requiring faster response time. It can only be raised if FIQs are not masked in the CPSR.

OR

Q.8 a) *Explain CPSR register of ARM.*
(Refer Q.16 of Chapter - 10) [6]

b) *Write significance of special registers R13, R14 and R15 in ARM7.* **(Refer Q.15 of Chapter - 10)** [6]

c) *Write short note on ARM7 processor modes.*
(Refer Q.19 of Chapter - 10) [5]

END...