Microcontrollers

Semester V – Electronics and Telecommunication Engineering / Electronics Engineering
(Savitribai Phule Pune University)

Strictly as per the New Credit System Syllabus (2015 Course)
Savitribai Phule Pune University w.e.f. academic year 2017-2018

U. S. Shah
Formerly, Lecturer
Department of Electronics Engineering
Vishwakarma Institute of Technology (V.I.T.), Pune.
Maharashtra, India.

Tech-Max Publications, Pune
Innovation Throughout
Engineering Division

PO272A

Microcontrollers

(Semester V - Electronics and Telecommunication Engineering / Electronics Engineering, (Savitribai Phule Pune University))

U. S. Shah

First Printed in India : July 2014 (Pune University)
First Edition : June 2017

This edition is for sale in India, Bangladesh, Bhutan, Maldives, Nepal, Pakistan, Sri Lanka and designated countries in South-East Asia. Sale and purchase of this book outside of these countries is unauthorized by the publisher.

Printed at : Image Offset, Dugane Ind. Area, Survey No. 28/25, Dhayari, Near Pari Company, Pune - 41, Maharashtra State, India. E-mail : rahulshahimage@gmail.com

ISBN 978-93-5224-566-6

Published by

Tech-Max Publications

Head Office : B/5, First floor, Maniratna Complex, Taware Colony, Aranyeshwar Corner,
Pune - 411 009. Maharashtra State, India
Ph : 91-20-24225065, 91-20-24217965. Fax 020-24228978.
Email : info@techmaxbooks.com,
Website : www.techmaxbooks.com

[304184] (FID : TP477) (Book Code : PO272A)

Shruthika. J. Bhatemao.

Preface

My dear students,

I am extremely happy to come out with this book on **"Microcontrollers"** for the students. This book has been strictly written as per the syllabus. I have divided the syllabus into small chapters so that the topics can be arranged and understood properly. The topics within the chapters have been arranged in a proper sequence to ensure smooth flow of the subject.

I am thankful to Shri. Pradeep Lunawat and Shri. Sachin Shah for the encouragement and support that they have extended. I am also thankful to the staff members of Tech-Max Publications and others for their efforts to make this book as good as it is. We have jointly made every possible efforts to eliminate all the errors in this book. However if you find any, please let me know, because that will help me to improve further.

I am also thankful to my family members and friends for patience and encouragement.

- U. S. Shah

For suggestions and queries you can mail at

urvashimshah@gmail.com

□□□

Syllabus

304184 : Microcontrollers (E&TC)

Teaching Scheme	Credits : TH-03	Examination Scheme
Lecture : 3 Hrs/Week		In Sem. : 30 Marks
		End Sem. : 70 Marks

Course Objectives :

- To understand architecture and features of typical Microcontroller.
- To understand need of microcontrollers in real life applications.
- To learn interfacing of real world peripheral devices
- To study various hardware and software tools for developing applications.

Course Outcomes :

- On completion of the course, student will be able to

1) Learn importance of microcontroller in designing embedded application.
2) Learn use of hardware and software tools.
3) Develop interfacing to real world devices.

Course Contents

Unit I : Introduction to Microcontroller Architecture **(06 Hrs.)**

Overview of MCS-51 architecture, Block diagram and explanation of 8051, Port structure , memory organization, Interrupt structure, timers and its modes, serial communication modes. Overview of Instruction set, Sample programs (assembly): Delay using Timer and interrupt, Programming Timer 0&1, Data transmission and reception using Serial port. **(Refer Chapter 1)**

Unit II : IO Port Interfacing-I **(06 Hrs.)**

Interfacing of : LEDS, Keypad, 7-segment multiplexed display, LCD, ADC 0809(All programs in assembly). Programming environment : Study of software development tool chain (IDE), hardware debugging tools (timing analysis using logic analyser) **(Refer Chapter 2)**

Unit III : Parallel Port Interfacing-II **(06 Hrs.)**

Interfacing of: DAC, Temperature sensors, Stepper motor, Motion detectors, Relay, Buzzer, Optoisolaters, Design of DAS and Frequency counter : All programs in assembly **(Refer Chapter 3)**

Unit IV : PIC Microcontroller Architecture **(06 Hrs.)**

Features, comparison & selection of PIC series as per application. PIC18FXX architecture- MCU, Program and Data memory organization, Pin out diagram, Reset operations, Oscillator options (CONFIG), BOD, power down modes & configuration bit settings, timer and its programming, Brief summary of Peripheral support, Overview of instruction set. **(Refer Chapters 4, 5, 6 and 7)**

Unit V : Real World Interfacing Part I **(06 Hrs.)**

Port structure with programming, Interrupt Structure (Legacy and priority mode) of PIC18F with SFRs. Interfacing of LED, LCD (4&8 bits), and Key board, use of timers with interrupts, CCP modes: Capture, Compare and PWM generation, DC Motor speed control with CCP: All programs in embedded C.

(Refer Chapters 8, 9, 10 and 11)

Unit VI : Real World Interfacing Part II **(06 Hrs.)**

Basics of Serial Communication Protocol: Study of RS232,RS 485, I2C,SPI, MSSP structure(SPI &I2C),UART, Sensor interfacing using ADC, RTC(DS1306) with I2C and EEPROM with SPI. Design of PIC test Board, Home protection System: All programs in embedded C. **(Refer Chapters 12, 13 and 14)**

304192 : Microcontrollers Lab (E&TC)

Teaching Scheme	Credits : PR-02	Examination Scheme
Practical : 4 Hrs/Week		Practical : 50 Marks
		Term Work : 50 Marks

List of Practical's: Minimum 10 experiments

(Experiment number 2,3, 5,6, 7, 9,10, 12 are compulsory; Any one from 1 and 4 , 8, 11 and 13)

1. Simple programmes on Memory transfer.
2. Parallel port interacting of LEDs-different programs(flashing, Counter, BCD, HEX, Display of Characteristic)
3. Waveform Generation using DAC
4. Interfacing of Multiplexed 7-segment display (counting application)
5. Interfacing of LCD to 8051 (4 and 8 bit modes)
6. Interfacing of Stepper motor to 8051- software delay using Timer
7. Write a program for interfacing button, LED, relay & buzzer as follows
   A. On pressing button1 relay and buzzer is turned ON and LED's start chasing from left to right
   B. On pressing button2 relay and buzzer is turned OFF and LED start chasing from right to left .
8. Interfacing 4×4 keypad and displaying key pressed on LCD.
9. Generate square wave using timer with interrupt
10. Interfacing serial port with PC both side communication.
11. Interfacing EEPROM 24C128 using I2C to store and retrieve data
12. Interface analog voltage 0-5V to internal ADC and display value on LCD
13. Generation of PWM signal for DC Motor control.

304204 : Microcontrollers and Applications (Electronics Engg.)

Teaching Scheme	Credits : TH-04	Examination Scheme
Lectures : 4 Hrs/Week		In Semester Assessment : 30 Marks
		End Semester Examination : 70 Marks

Course Objectives :

- To understand the applications of Microprocessors & Microcontrollers.
- To understand need of microcontrollers in embedded system.
- To understand architecture and features of typical Microcontroller.
- To learn interfacing of real world input and output devices
- To study various hardware & software tools for developing applications

Course Outcomes :

After successfully completing the course students will be able to

- Learn importance of microcontroller in designing embedded application
- Describe the 8051 & PIC18FXX microcontroller architectures and its feature.
- Develop interfacing to real world devices
- Learn use of hardware & software tools

Unit I : Introduction to microcontroller Architecture **(08 L)**

Microprocessor and microcontroller comparison, advantages & applications. Harvard & Von Neumann architecture, RISC & CISC processors. Role of microcontroller in embedded system. Selection criteria of microcontroller. Overview of MCS-51 architecture, Block diagram and explanation of 8051, Port structure, memory organization, Interrupt structure, timers and its modes, serial communication modes. Overview of Instruction set, Sample programs (assembly): Delay using Timer and interrupt, Programming Timer 0&1, Data transmission and reception using Serial port. **(Refer Chapter 1)**

Unit II : Interfacing-I **(06 L)**

Software and Hardware tools for development of microcontroller based systems such as assemblers, compilers, IDE, Emulators, debuggers, programmers, development board, DSO, Logic Analyzer. Interfacing LED with and without interrupt, Keypads, Seven Segment multiplexed Display, LCD, ADC Interfacing. All Programs in assembly language. **(Refer Chapter 2)**

Unit III : Interfacing-II **(06 L)**

Interfacing of DAC, Temperature sensors, Stepper motor, Motion detectors, Relay, Buzzer, Opto-isolators, Design of DAS and Frequency counter. All programs in assembly **(Refer Chapter 3)**

Unit IV : PIC Microcontroller Architecture **(06 L)**

PIC 10, PIC12, PIC16, PIC18 series comparison, features and selection as per application. PIC18FXX architecture, registers, memory Organization and types, stack, oscillator options, BOD, power down modes and configuration bit settings, timer and its programming. Brief summary of Peripheral support, Overview of instruction set, MPLAB IDE & C18 Compiler. **(Refer Chapters 4, 5, 6 and 7)**

Unit V : Real World Interfacing Part I **(06 L)**

Port structure with programming, Interrupt Structure (Legacy and priority mode) of PIC18F with SFRS. Interfacing of switch, LED, LCD (4&8 bits), and Key board. Use of timers with interrupts, CCP modes: Capture, Compare and PWM generation, DC Motor speed control with CCP: All programs in embedded C.

(Refer Chapters 8, 9, 10 and 11)

Unit VI : Real World Interfacing Part II **(06 L)**

Basics of Serial Communication Protocol: Study of RS232, RS 485, I2C, SPI, MSSP structure (SPI &I2C), UART, Sensor interfacing using ADC, RTC (DS1306) with I2C and EEPROM with SPI. Design of PIC test Board, Home protection System: All programs in embedded C. **(Refer Chapters 12, 13 and 14)**

304212 : Microcontrollers Lab (Electronics Engg.)

Teaching Scheme	Credits : PR-02	Examination Scheme
Practical : 4 Hrs/Week		Practical : 50 Marks
		Term work : 50 Marks

List of Experiments :

Experiments 1 and 2 are compulsory. Perform any 8 experiments from 3 to 12.

1) Interfacing LED bank to 8051 microcontroller using timer with interrupt.
2) Interfacing Seven Segment Display to 8051 microcontroller
3) Write a program for interfacing button, LED, relay & buzzer to PIC18FXX as follows:
   a) when button 1 is pressed, relay and buzzer is turned ON and LED's start chasing from left to right
   b) when button 2 is pressed, relay and buzzer is turned OFF and LED start chasing from right to left
4) Display message on LCD without using any standard library function for PIC18Fxx.
5) Interfacing 4 × 4 keypad and displaying key pressed on LCD OR on HyperTerminal for PIC18Fxx.
6) Generate square wave using timer with interrupt for PIC18Fxx.
7) Serially transfer the data on PC using serial port of PIC18Fxx.
8) Generation of PWM signal from PIC18Fxx for DC Motor control.
9) Interface analog voltage 0-5V to internal ADC and display value on LCD.
10) Using DAC generate various waveforms.
11) Interfacing DS1307 RTC chip using I2C and display date and time on LCD.
12) Interfacing EEPROM 24C128 using SPI to store and retrieve data.

❑❑❑

UNIT I

Syllabus : Overview of MCS-51 architecture, Block diagram and explanation of 8051, Port structure , memory organization, Interrupt structure, timers and its modes, serial communication modes. Overview of Instruction set, Sample programs (assembly): Delay using Timer and interrupt, Programming Timer 0&1, Data transmission and reception using Serial port.

Chapter 1 : Introduction to Microcontroller Architecture 1-1 to 1-102

UNIT II

Syllabus : Interfacing of: LEDS, Keypad, 7-segment multiplexed display, LCD, ADC 0809(All programs in assembly). Programming environment: Study of software development tool chain (IDE), hardware debugging tools (timing analysis using logic analyser)

UNIT III

Syllabus : Interfacing of: DAC, Temperature sensors, Stepper motor, Motion detectors, Relay, Buzzer, Optoisolaters, Design of DAS and Frequency counter: All programs in assembly

UNIT IV

Syllabus : Features, comparison & selection of PIC series as per application. PIC18FXX architecture- MCU, Program and Data memory organization, Pin out diagram, Reset operations, Oscillator options (CONFIG), BOD, power down modes & configuration bit settings, timer and its programming, Brief summary of Peripheral support, Overview of instruction set.

Chapter 5 : PIC18F458 Instruction Set 5-1 to 5-19

Chapter 6 : PIC Programming in C 6-1 to 6-12

UNIT V

Syllabus : Port structure with programming, Interrupt Structure (Legacy and priority mode) of PIC18F With SFRS. Interfacing of LED, LCD (4&8 bits), and Key board, use of timers with interrupts, CCP modes: Capture, Compare and PWM generation, DC Motor speed control with CCP: All programs in embedded C.

UNIT VI

Syllabus : Basics of Serial Communication Protocol: Study of RS232,RS 485, I2C,SPI, MSSP structure(SPI & I2C),UART, Sensor interfacing using ADC, RTC(DS1306) with I2C and EEPROM with SPI. Design of PIC test Board, Home protection System: All programs in embedded C.

❑❑❑

8051 Programs

PIC18 Programs

RC1	RC0	
0	0	Send '0' to Port B
0	1	Send '1' to Port B
1	0	Send '2' to Port B
1	1	Send '3' to Port B

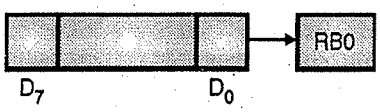


Fig. P. 27 : LSB going out first

❑❑❑

UNIT - I

CHAPTER 1

Introduction to Microcontroller Architecture

1.1 8 Bit Microprocessor and Microcontroller Architecture

- The microprocessor is a central processing unit of a general purpose digital computer. They can address megabytes of memory and operate on 8, 16, or 32 bit data.
- It consists of an ALU, accumulator, working registers, program counter, stack pointer, clock and interrupt circuits. Fig. 1.1.1 shows general architecture of a microprocessor.
- The microprocessor alone is not a complete digital computer. In order to make it a complete computers one should add memory devices (ROM, RAM, EEPROM, EPROM, PROM), memory decoders, I/O devices (keyboard and display controller i.e. IC 8279, Timer/Counter i.e. IC 8253/8254, programmable peripheral interface i.e. IC 8255, programmable interrupt controller i.e. IC 8259 etc).

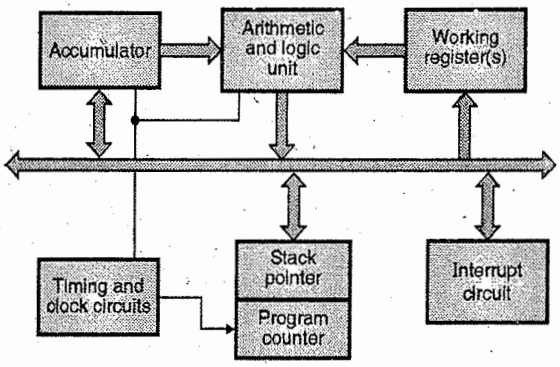


Fig. 1.1.1 : General architecture of a microprocessor

- Due to varieties of memory and I/O devices, the hardware design of a microprocessor is arranged so that a very small or very large system can be configured around the CPU depending on the application of the user. For a small application the minimum size of memory and I/O (s) must be interfaced to the CPU. This increases the hardware. In turn the PCB size and cost of the system also increases.
- In order to avoid these drawbacks **Microcontrollers** were developed. The Microcontroller is an on-chip true computer. It is optimized for specific applications.
- It consists all the features of a microprocessor as well as the features required to build a complete computer.

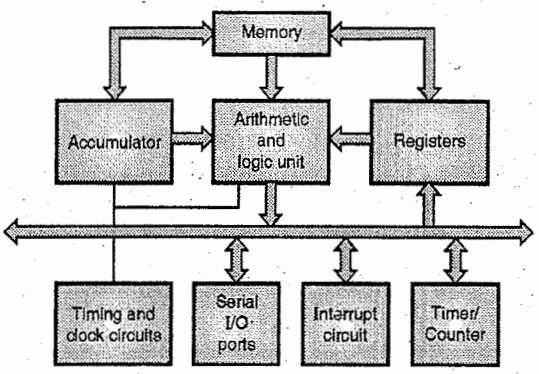


Fig. 1.1.2 : General architecture of a microcontroller

The microcontroller has **on-chip** (built in) peripheral devices. They are ROM, RAM, Serial I/O, Parallel I/O, timer/counters, interrupts and clock. These on chip peripherals make if possible to house single chip microcomputer system.

The advantages of built-in peripherals are

(i) The hardware is less because of single chip microcomputer system.

(ii) Because of less hardware, the PCB size is small.

(iii) It increases system reliability.

(iv) They have smaller access time. Hence, speed is high.

- The microcontroller consists of RAM, ROM, I/O devices, counters and clock circuit etc.
- The first 8 bit microcontroller, 8048 was introduced by Intel in 1976 with a view to control general tasks.
- Later on, high performance microcontroller families like MCS51, MCS96 were developed. The MCS51 family includes a whole family of microcontrollers that have numbers ranging from 8031 to 8751. They are available in MOS or CMOS technology with different packages.
- The 8051 microcontroller in the MCS51 family was designed for 8 bit mathematical and single bit boolean operations. These families provide separate program and data memory. The program or data memory may be internal or external. They provide high speed and have low system cost.

1.1.1 History of Microprocessors

- The microprocessors are classified depending on the number of bits on which the ALU can operate at a time. e.g. a 8085 microprocessor that has 8 bit ALU is called as 8 bit microprocessor irrespective of the size of the data and address bus of the microprocessor.
- The first microprocessor was introduced by Intel in 1971. It was called as Intel 4004 and is a 4 bit PMOS PROCESSOR and comprised 2300 transistors. The other companies like Toshiba, Rockwell also developed 4 bit microprocessors. These microprocessors are first generation microprocessors.
- In 1972 Intel introduced the first 8 bit microprocessor 8008. It also used the PMOS technology. These processors were slow and not compatible with the TTL logic. So, in 1974 Intel introduced a faster NMOS microprocessor Intel 8080. 8080 is a second generation microprocessor.
- But the drawback of Intel 8080 is that it needed three power supplies (+5V, – 5V and +12 V). Hence, Motorola introduced a new 8 bit processor MC6800 that operates on single +5V supply.
- In 1975 Intel developed an improved version of 8080 called 8085 microprocessor. The other manufacturers of 8 bit microprocessors are Zilog, National semiconductors, Motorola etc.

1.1.2 Comparison of Different 8 Bit Microprocessors

Processor	Number of Transistors	Year of Introduction	Features				
			CLK speed	Bus width		Number of Instructions	Some special features
				Data	Address		
Intel 4004	2300	1971	108 KHz	4 bits	4 bits	46	World's first microprocessor
Intel 8008	3500	1972	500 KHz , 800 KHz	8 bits	8 bits	66	Can Handle interrupts.
Intel 8080	4500	1973	2 MHz	8 bits	16 bits	111	Needs three power supplies.
MC 6800	4000	1974	1 MHz	8 bits	16 bits	72 basic instructions with total 197 instructions	It has 2 accumulators and no on chip general purpose data register.
Intel 8085	6500	1976	3 MHz, 5 MHz, 6 MHz	8 bits	16 bits	80 basic instructions with total 246 instructions	Serial Communication was introduced
Zilog Z80	8500	1976	2.5 – 8 MHz	8 bits	16 bits	158 basic instructions	Powerful instruction set
MC 6802	8500	1977	1 MHz	8 bits	16 bits	197	It has on – chip 128 bytes of RAM and on chip clock oscillator.
MC6809	9000	1979	2 MHz	8 bits	16 bits	1464	It supports 8 bit external bus.

1.2 Comparison between Microprocessors and Microcontrollers

SPPU - May 12, Dec. 12, May 13, May 14, Oct. 16

University Questions

Q. Differentiate between microprocessor & microcontroller with general architecture and features. **(May 2012, Dec. 2012, May 2013, May 2014, 8 Marks)**

Q. Differentiate between microprocessor and microcontroller **(Oct. 2016 (In Sem.), 5 Marks)**

Sr. No.	Microprocessor	Microcontroller
1.	A microprocessor is a chip that is dependent on other chips for many functions.	A microcontroller is a single chip microcomputer that has everything in-built.
2.	A microprocessor contains ALU, general purpose registers, stack pointer, program counter, timing and control circuit and interrupt circuit.	A microcontroller contains the circuitry of a microprocessor and has built in RAM, ROM, I/O devices, timers and counters.
3.	It is suited to processing information in computer systems.	It is suited to control of I/O devices requiring a minimum component count.
4.	It has one or two bit manipulation instructions.	It has many bit manipulation instructions.
5.	It has less number of multifunctioned pins.	It has more number of multifunctioned pins.
6.	They have large memory address space and more data.	They have relatively small address space and less data.
7.	It has a single memory map for data and code.	It has a separate memory map for data and code.
8.	Design is very flexible.	Design is less flexible.
9.	Microprocessor based system requires more hardware.	Microcontroller based system requires less hardware reducing PCB size and increasing reliability.
10.	Access times for memory and I/O devices are more.	Less access times for built-in memory and I/O devices.
11.	The clock rates are very fast.	The clock rates are relatively slow.
12.	It has many instructions to move data between memory and CPU.	It has one or two instructions to move data between memory and CPU.
13.	They are expensive.	They are cheap.

1.3 Advantages of Microcontrollers

The **advantages** of built-in peripherals are

(i) The hardware is less because of single chip microcomputer system. This reduces the cost.
(ii) Because of less hardware, the PCB size is small.
(iii) It increases system reliability.
(iv) They have smaller access time. Hence, speed of operation is high.
(v) It is easy to use, troubleshoot and maintain.
(vi) Most of the pins can be programmed by the user inorder to perform different functions.
(vii) Microcontrollers have inbuilt serial ports, timers and counters, RAM, ROM, A/D converters, flash memory.
(viii) Microcontrollers consume less power.
(ix) A microcontroller can repeatedly do many tasks, this saves the human efforts i.e. it is labor saving.

1.4 Disadvantages/Limitations of 8 bit Microcontroller

SPPU - Aug. 14

University Question

Q. What are limitations of 8 bit microcontroller ?

(Aug. 2014 (In Sem.), 5 Marks)

The drawbacks/limitations of 8 bit microcontrollers are as follows :

(i) The microcontrollers have a complex architecture than that of a microprocessor. Hence, it is difficult to understand their functionality.
(ii) As microcontrollers are RISC microcontrollers, the length of programs is big.
(iii) Only a single WREG/A (accumulator) is present.
(iv) They cannot access the program memory.
(v) The microcontrollers cannot be directly interfaced with high power devices.

1.5 Applications of Microcontrollers

- Microcontrollers are widely used in embedded system products to obtain a particular task. e.g. printer is an embedded system as the processor inside it does only one task i.e. acquiring the data and printing it.
- Some applications of embedded products using microcontrollers are given below :

Home :		
1. Appliances	9. TVs	17. Sewing machines
2. Telephones	10. Cable TV tuner	18. Lighting control
3. Security systems	11. VCR	19. Paging
4. Intercom	12. Camcorder	20. Camera
5. Garage door openers	13. Remote control	21. Toys
6. Answering machines	14. Video games	22. Exercise equipment
7. Fax machines	15. Cellular phones.	
8. Home computers	16. Musical instruments	

Office :		
1. Telephones	4. Fax machines	7. Laser printer
2. Computers	5. Microwave	8. Colour printer
3. Security systems	6. Copier	9. Paging

Automation :		
1. Trip computer	5. Instrumentation	9. Climate control
2. Engine control	6. Security system	10. Cellular phones
3. Air bag	7. Transmission control	11. Keyless entry
4. ABS	8. Entertainment	

- Other applications are :
  (i) Square wave generation
  (ii) Pulse generation
  (iii) Pulse width modulation
  (iv) Sine wave generation
  (v) Staircase ramp generation
  (vi) Pulse width measurement
  (vii) Frequency counter
  (viii) Driving an electromechanical relay for switching on/off an ac motor.
- The industrial applications include:
  (i) Sensing Robot arm position
  (ii) Measurement of angular speed
  (iii) Control the speed and direction of DC motor
  (iv) Stepper motor control.

1.6 Harvard and Von Neumann Architectures

SPPU - Dec. 12, May 13, Aug. 14, Aug. 15

University Questions

Q. Compare Von Neumann and Harvard architecture. **(Dec. 2012, Aug. 2014 (In Sem.), 5 Marks)**

Q. Differentiate Harvard and Von Neumann Architecture by giving one example of each. **(May 2013, Aug. 2015 (In Sem.), 4 Marks)**

Sr. No.	Von-Neumann architecture	Harvard architecture
1.	Please refer Fig. 1.6.2	Please refer Fig. 1.6.1
2.	In Von-Neumann's architecture, data bus and address bus are not separate. Thus a greater flow of data is not possible through the central processing unit, and of course, a greater speed of work. It allows storing or modifying programs easily.	In Harvard architecture, data bus and address bus are separate. Thus a greater flow of data is possible through the central processing unit, and of course, a greater speed of work.
3.	Microcontrollers with von-Neumann's architecture are called 'CISC microcontrollers'. CISC stands for **Complex Instruction Set Computer.**	Microcontrollers with Harvard architecture are also called "RISC microcontrollers". RISC stands for **Reduced Instruction Set Computer.**
4.	It is also typical for Von Neumann's architecture to have more instructions than Harvard architecture.	It is also typical for Harvard architecture to have fewer instructions than von-Neumann's, and to have instructions usually executed in one cycle.

Sr. No.	Von-Neumann architecture	Harvard architecture
5.	Time division multiplexing is used for fetching the program and data.	The address and data buses are separate. Hence, there is no need to have time division multiplexing.
6.	It needs multiple fetches for processing an instruction.	Its internal organization is such that an instruction can be prefetched and decoded while multiple data are being fetched and processed.
7.	**Example :** MC68HC11 supports Von Neumann's architecture.	**Example :** MCS-51 family of microcontrollers, PIC microcontrollers use Harvard architecture.

- There are two main classes of computer architectures. There are :

(i) Harvard architecture
(ii) Von Neumann (or Princeton) architecture.

1.6.1 Harvard Architecture

Fig. 1.6.1 shows Harvard Architecture.

This System architecture was designed and recommended by Harvard University. As per this architecture, the Processor is having two different Memory spaces with separate Memory maps. They are Data Memory and Program Memory respectively. Therefore, the distinction between Data Memory (Memory space used to store Data) and Program Memory (Memory space used to store Programs) is Physical. It is connected to the processor through separate sets of Address, Data and Control buses. This architecture is normally used by RISC Processors.

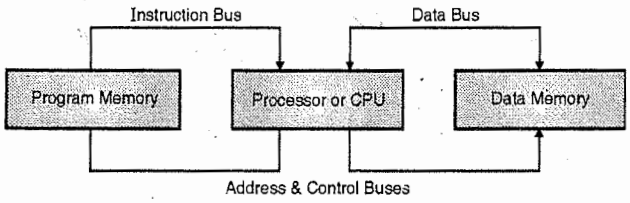


Fig. 1.6.1 : Harvard architecture

1.6.1.1 Advantages

- Being accessed by separate sets of buses, both Program and Data can easily be simultaneously accessed, therefore improving the performance.
- The implementation of Pipelining is streamlined and decreases CPU stalls and so effectively increasing the efficiency of the pipeline.

1.6.1.2 Disadvantages

- 2 Sets of buses are needed and therefore relatively complex bus structure.
- Does not allow sizes of Data and Program to be adjusted flexibly and dynamically, as they are stored in different Memory spaces physically.
- **Examples :**
  (i) The Intel MCS-51 family of microcontrollers has Harvard architecture. This is because there are different memory spaces for program and data and separate (internal) buses for the address and data.
  (ii) The PIC Microcontrollers by Microchip also use Harvard architecture.

1.6.2 Von-Neumann Architecture

This System architecture was designed and recommended by Von-Neumann at the Princeton University. As per this architecture, the Processor is having a single memory space addressed through a single Memory map. The memory may contain both Data as well as Programs in it. The distinction between Data Memory (Memory space used to store Data) and Program Memory (Memory space used to store Programs) is Logical. Therefore same memory space shares Data as well as Program Code. It is connected to the processor through a single set of Address, Data and Control buses. This architecture is normally used by CISC Processors.

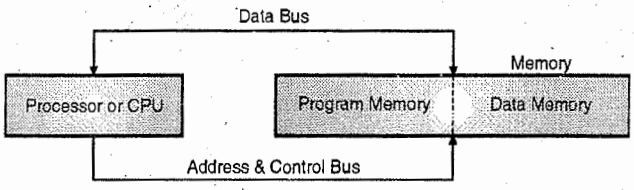


Fig. 1.6.2 : Von-Neumann architecture

1.6.2.1 Advantages

- Simple construction, less complex bus structure.
- Allows sizes of Data and Program within the memory space to be adjusted flexibly and dynamically.

1.6.2.2 Disadvantages

- Being accessed by same set of buses, both Program and Data cannot be simultaneously accessed.

- The implementation of Pipelining becomes difficult and increases CPU stalls decreasing the efficiency of the pipeline.

1.7 RISC and CISC

SPPU - May 15

University Question

Q. What is RISC microcontroller ? **(May 2015, 1 Mark)**

- **Complex Instruction Set Computers (CISC) and Reduced Instruction Set Computers (RISC)** are two terms commonly used while discussing about the different microprocessors and microcontrollers.
- CISC processor have large number of instructions. A large instruction set helps the assembly language programmers by providing flexibility of writing short and effective programs.
- The purpose of the CISC architecture is to write a program in as few lines of assembly language code as possible.
- eg. : In 8051 the MUL(multiply) instruction is a complex instruction for which only the operands need to be specified in the instruction. The multiplication operation is to be done by hardware.
- The building of complex instructions in hardware helps the user in two ways. Firstly the implementation becomes faster and secondly the program space required is less. Thus the programmer operates at a higher level.
- The programmers require few, simple and fast instructions in comparison to large complex and slow CISC instructions. However it is obtained at the cost of writing more instructions to accomplish a task with the help of **RISC processors**.
- RISC (Reduced Instruction Set Computer) machines are widely used nowadays. The designer designed the RISC architectures considering the following points :
  (i) a simple and limited instruction set.
  (ii) large number of general purpose registers.
  (iii) emphasis on optimizing the instruction pipeline.

The **characteristics of RISC processors** are :

(i) **One instruction per clock cycle :** In RISC processors there is one instruction per clock cycle. A machine cycle is defined as the time required for fetching the operands, decoding and executing instruction and storing the result in register. Because of this feature the RISC instructions are not complicated. They can be executed very fast.

(ii) **Hardwired instructions :** As the RISC instructions are simple, microinstructions are not required. The machine instructions are hardwired, and can be executed faster than the instructions implemented with microinstructions.

(iii) **Reduced number of instructions :** It provides a limited instruction set that simplifies the design of control unit.

(iv) **Simple addressing modes :** It has simple addressing modes. Almost all the instructions use simple register addressing mode. RISC processors do not support complex addressing modes. Other addressing modes like displacement, PC relative can be used.

(v) **Instruction Pipelining :** Pipelining means that the CPU comprises several independent units that operate in parallel. One of the unit fetches the instruction and others decode and execute the instruction. At an instant many instructions are in different processing stages. All the RISC processors support pipelining.

(vi) **Simple Instruction Format :** RISC processors uses simple instruction formats with fixed instruction length. The instruction length is aligned on the word boundaries. The opcodes are fixed for an instruction. As word length units are fetched, the fetching operation is optimized.

(vii) **Register to Register Operations :** It is an important characteristic of a RISC processor that encourages the optimization of register use. The operands that are frequently used are stored in high-speed storage to perform the register-to-register operations.

The RISC processors have multiple sets of registers.

- The registers are organized into overlapped windows. They act as small, fast buffer for holding a subset of all variables that are likely to be used.
- Fig. 1.7.1 shows overlapping register window.

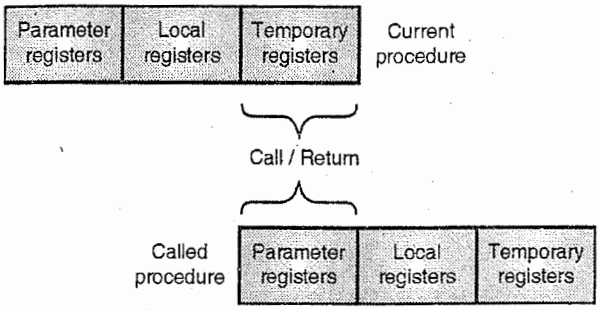


Fig. 1.7.1 : Overlapping register window

- The windows are divided into three fixed size areas. The parameter passing registers hold parameters to be passed down from the procedure referred as the called procedure. The results are to be passed back. Local registers are used for local variables assigned by the compiler.
- The temporary registers are used to exchange parameters and the results with the procedure called by the current procedure. The temporary registers of current procedure are same like the parameter registers of the called procedure.
- The overlap allows parameters to be passed with actually moving the data.
- **An example of RISC microcontroller is the PIC family of microcontrollers.**
- **8051 microcontroller combines the features of RISC and CISC processor.** It supports register to register operation as it uses a large register set. However, it comprises of CISC feature having a large instruction set.

1.7.1 CISC and RISC Comparison

SPPU - Dec. 13, May 15, Dec. 15, May 16, Oct. 16

University Questions

Q. How is RISC microcontroller different than CISC microcontroller ? **(May 2015, 5 Marks)**

Q. Compare RISC and CISC microcontroller with example. **(Dec. 2013, Dec. 2015, May 2016, 6 Marks)**

Q. Differentiate between RISC and CISC. **(Oct. 2016 (In Sem.), 6 Marks)**

In Table 1.7.1, we compare the main features of RISC and CISC processors. The comparison involves five areas : instruction sets, addressing modes, register file and cache design, clock rate and expected CPI, and control mechanisms.

Table 1.7.1 : Comparison of CISC and RISC Architectures

Sr. No.	Architectural Characteristic	Complex Instruction Set Computer (CISC)	Reduced Instruction Set Computer (RISC)
1.	**Instruction formats**	Instructions with variable formats (16-64 bits per instruction)	Instructions with fixed (32-bit) format and most register-based instructions.
2.	**Number of instructions**	120 to 350	Less than 100
3.	**Addressing modes.**	12-24.	Limited to 3-5.
4.	**General-purpose registers and cache design.**	8-24 GPRs, mostly with a unified cache for instructions and data, recent designs also use split caches.	Large numbers (32-192) of GPRs with mostly split data cache and instruction cache.
5.	**Clock rate and CPI**	33-50 MHz in 1992 with a CPI between 2 and 15	50-150 MHz in 1993 with one cycle for almost all instructions and an average CPI < 1.5.
6.	**CPU Control**	Mostly microcoded using control memory (ROM), but modern CISC also uses hardwired control.	Mostly hardwired without control memory.
7.	**HLL Support**	Many HLL statement are directly implemented.	Very few.
8.	**Pipelining.**	Not pipelined or less pipelined.	Highly pipelined.
9.	**Register sets.**	Single register set.	Multiple register sets.
10.	**Instruction execution.**	Instructions are executed by micro-program.	Instructions are executed by hardware.

Sr. No.	Architectural Characteristic	Complex Instruction Set Computer (CISC)	Reduced Instruction Set Computer (RISC)
11.	Complexity.	Complexity is in the micro-program.	Complexity is the compiler.
12.	Instructions and cycles.	There are complex instructions requiring multiple cycles for execution.	There are simple instructions and require single clock cycle for execution.
13.	Memory reference.	Most of the instructions refer to the memory.	Very few instructions refer to the memory.
14.	Complex addressing modes.	Supported.	Complex addressing modes are synthesized in software.
15.	Instruction size	Variable Instruction Size – many variations	Fixed Instruction Size (Same as Data Bus)
16.	ALU Instruction	ALU Instructions operate on all types – Register and Memory operands	ALU Instructions operate only on Register Operands
17.	Pipeline operation	Stalled, less streamlined Pipeline operation	Deep and Streamlined Pipeline operation
18.	Regularity	Low Chip Regularity – Higher chip turnaround times	High Chip Regularity – Less chip turnaround time

1.8 Criteria for Selecting a Microcontroller

SPPU - May 12, Dec. 12, May 13, Dec. 13, May 14, Aug. 14, Dec. 15

University Questions

Q. Explain criteria for choosing a microcontroller.
(May 2012, Dec. 2012, May 2013, Dec. 2015, 6 Marks)

Q. Explain selection criteria of microcontrollers for particular application.
(Dec. 2013, May 2014, Aug. 2014 (In Sem.), 5 Marks)

(1) The first and important criterion i.e. **primary criteria** in choosing a microcontroller is that it should meet the requirements and cost effectively. While analyzing the requirements of a microcontroller based project we must decide whether an 8 bit, 16 bit, or 32 bit microcontroller is capable of handling the project efficiently.

The other features that are important for selecting a microcontroller are :

(i) **Data Size :** 8, 16, 32 Bit μC, as per the needed Processing power.

(ii) **Clock Speed :** As per the needed speed to complete operations in specified time.

(iii) **Memory Size :** As per the required size of program and estimate of space needed for live data.

(iv) **Power Consumption :** As per the Availability of Power and type of power available. It is critical for products that are battery operated.

(v) **Parallel and Serial I/O Ports :** As per the number and type of devices to be interfaced in the system.

(vi) **Interrupts and Timers :** As per the need of the application and device interfaced and controlled.

(vii) **One time Development Cost :** Meeting the budget and financial constraints.

(viii) **Packaging -** the type of packaging used by microcontroller eg. 40 pin DIP (Dual in Line Package) or QFP (Quad Flat Package) etc. packaging is important in terms of space, assembling and prototyping the end product.

(ix) **Cost per unit -** the final cost of the product in which a microcontroller is used e.g. some microcontrollers cost 50 cents per unit when 1,00,000 units are purchased at a time.

(2) The second criterion in selecting a microcontroller is how easy it is to develop products around it. The important factors to be considered are availability of an assembler, debugger, a code efficient C/C++ language compiler, emulator, technical support and both in-house and outside expertise.

- **Availability :** of μC and other system components – current and over a period of time.
- **Upgradability :** of μC and other system components – ease, incremental cost and value addition.

- **Maintainability :** of μC and other system components – on-site and component level repairs.

(3) The third criterion in selecting a microcontroller is its ready availability in required quantities presently and in future. This criterion is important for some designers than the first two criteria. Presently for the 8 bit microcontrollers, the 8051 family has the largest number of suppliers. The 8051 microcontroller was originated by intel. But today several companies produce 8051. They include intel, Atmel, AMD, Philips / signetics, Infineon, sil labs, Matra and Dallas semiconductor.

1.9 Performance of Microcontroller

SPPU - Dec. 16

University Question

Q. How performance of any microcontroller is evaluated ? **(Dec. 2016, 8 Marks)**

The different parameters that are used for evaluating the performance of any microcontroller are :

(i) Processing capability : It depends on the number of instructions and flexibility of each instruction.

(ii) Clock speed : The processing speed of a microcontroller depends on the clock frequency of the microcontroller. The operations should be completed in the time specified.

(iii) Word length : The word length of the microcontroller depends on the width of internal data bus, registers, ALU. Depending on the processing power the microcontroller may be 8 bit, 16 bit, 32 bit, 64 bit. A microcontroller with longer word length is more powerful and can process data at a faster speed.

(iv) Address bus width : The address bus width of a microcontroller decides the memory addressing capability. The maximum size of memory unit is decided by this parameter.

(v) Parallel and serial I/O ports : Depending on the need of application and devices interfaced and controlled.

(vi) Ease to upgrade to higher performance versions of microcontroller or compatibility with low performance versions of microcontroller.

(vii) The number of on-chip timers and interrupts.

(viii) Packaging : The type of packaging used by microcontroller e.g. 40 pin DIP (dual in line package) or QFP (quad flat package) etc. This parameter is important with regards to assembling and prototyping end product.

(ix) Power consumption : This parameter will specify the microcontroller power consumption in its normal, idle, power down modes. The power consumption of a microcontroller should be low. This allows the microcontroller to implement more functionality. Also use of fans and blowers for cooling can be avoided. This parameters is critical for battery operated products.

1.10 Embedded System and its Characteristics

SPPU - Aug. 14, Aug. 15

University Question

Q. Define embedded system. Explain its characteristics. **(Aug. 2014(In Sem.), Aug. 2015(In Sem.), 5 Marks)**

Definition : An **embedded system** is an electronic device that incorporates microprocessors within its implementation. It is a combination of hardware and software for doing a particular job.

Function : The main task of the microprocessor is to simplify the system design and provide flexibility. The microprocessor adds new features to the system.

- However unlike the PCs embedded systems may not have a disk drive. The main task of an embedded system is to perform one and only one task. Hence, the software is stored in a read-only memory (ROM) chip. The software of an embedded system can be thus modified by replacing or "reprogramming" the ROM.
- Embedded systems are found in a wide range of application areas. Initially they were used for expensive industrial control applications. But as the technology advanced, it brought down the cost of dedicated processors. The dedicated processors were then used in moderately expensive applications like automobiles, communications and office equipment, televisions. However today the embedded systems have become so inexpensive that they are used in almost every electronic product in our life.

- The examples of the applications of embedded systems are listed in the Table 1.10.1.

Table 1.10.1

Applications	Examples
Aerospace	Navigation systems, automatic landing systems, flight attitude controls, engine controls, space exploration.
Automotive	Fuel injection control, passenger environmental control, antilock breaking systems, air bag controls, GPS mapping.
Children's Toys	Video games, Aeroplanes, Robots.
Communications	Satellites, Network Routers, Switches, Hubs
Computer Peripherals	Printers, Scanners, Keyboards, Displays, Modems, Hard disk drives, CD- ROM drives.
Home	Dishwashers, microwave ovens, VCR's, televisions, stereos, fire / security alarm systems, cameras, answering machines, sewing machines, lighting control, security systems, fax machines, paging, musical instruments, cell phones, exercise equipments, intercom.
Industrial	Elevator controls, surveillance systems, robots
Medical	Patient monitors, heart pacers, imaging systems (e.g. X-RAY, MRI, ultra sound)
Instrumentation	Oscilloscopes, Signal generators, Power supplies, Signal analyzers.
Office	Telephones, Computers, Copier, Laser printer, Colour printer, cell phones, fax machines.
Personal	Wrist watches, Video games, MP3 players, GPS, Cell phones, Camera, Computers, TVs, Cable TV Tuner, Remote Control.

1.10.1 Characteristics of Embedded Systems

- An embedded system designer should be able to design an embedded system that satisfies the required functionality of the embedded system. At the same time the designer has to provide an optimized design. The characteristics of an embedded system play a significant role in the implementation of an embedded system.
- The different **characteristics** of an Embedded system are as follows :

(i) **Power :** The amount of power consumed by the system. By knowing the amount of power required by a system we may determine the lifetime of a battery, cooling requirements for an integrated circuit etc.

(ii) **Nonrecurring engineering cost (NRE cost):** It is the one time momentary cost that is required at the time of designing the system. A number of units can be designed. Once a system is designed. Additional design cost is not required for the other units.

(iii) **Reliability :** The system should be able to work properly even if anything happens to the system. The system software must cope up with all kinds of situations without human intervention. The amount by which the system software can cope with the situations without any interventions decides the reliability of the embedded system.

(iv) **Response time :** The response time of an embedded system is the time required for the embedded system to react to certain situations or events quickly. The response time must be as small as possible.

(v) **Flexibility :** Flexibility of an embedded system is the amount by which the designer can change the functionality of the embedded system without bearing a heavy nonrecurring engineering cost. Generally the software of an embedded system is more flexible.

(vi) **Maintainability :** The maintainability of an embedded system is the ability by which the designer can change the embedded system after the system has been released. The system may be designed by a new team of engineers who were not involved in the earlier design process.

(vii) **Safety :** This parameter indicates the probability by which the system is safe to the user i.e. it indicates the probability that the system will not be harmful to the user.

(viii) **Correctness :** This parameter is of importance because it gives the user a confidence that he has been successful in implementing the embedded systems functionality correctly. While the entire process is being designed the system's functionality can be tested by the test circuitry.

(ix) **Memory size :** This parameter is important while designing an embedded system. An embedded system must be designed such that the design meets with all the system requirements and acquires a finite amount of memory space. The memory space is normally restricted.

The physical space that is required for an embedded system is specified in bytes for the system software and it is specified in terms of gates or transistors for the system hardware.

(x) **Unit cost :** The unit cost includes the cost that is required for producing each unit of the embedded system. The unit cost is found out excluding the NRE cost.

(xi) **Time to prototype :** This parameter specifies the time that is required for an embedded system to construct a working version of the system. The working version of the system may be such that it may be bigger or expensive than the final system that is to be implemented. It can be used to verify how useful and correct the system is inorder to refine the functionality of the embedded system.

(xii) **Performance :** This parameter specifies the execution time of the embedded system. For good system performance it is recommended that the execution time should be as small as possible.

(xiii) **Time to market :** This parameter specifies the time that is required to develop the embedded system, so that the embedded system can he released and sold to the customers. This time is dependent on the design time, manufacturing time and the testing time.

(xiv) **Testability :** It is the ability of the embedded system to prepare a setup to test the embedded system Software.

1.11 Role of Microcontroller in Embedded System

SPPU - May 15, Dec. 16

University Question

Q. Explain the role of microcontroller in embedded system. **(May 2015, Dec. 2016, 5 Marks)**

- Generally it is seen that microprocessors and microcontrollers are used for embedded systems. In an embedded system the microprocessor/microcontroller is assigned the duty of completing only one job at a time. The examples of some of embedded systems designed using microcontrollers are intercom, cable TV tuner, camera, printers, telephones, fax machines, cell phones, video games, remote controls etc.
- Unlike an embedded system a PC can perform a variety of applications because it consists of RAM memory and an operating system that loads the application software into the RAM, so that the CPU can execute it. In an embedded system only one application software is burned onto the ROM.
- Though the microcontrollers are preferred to be used for embedded systems, there are times when the microcontroller is inadequate to complete a task. To find a solution to this problem various companies have targeted their microprocessor for embedded systems. Hence these processors are called **high-end embedded processors.**
- The terms embedded processor and microcontroller interchangeable.
- The requirement of an embedded system is to reduce power consumption and space. This is done by integrating two or more functions on the CPU chip.
- The embedded processors based on x86 and 680x0 have low power consumption
- An x86 PC is connected to embedded products like keyboard, printer, sound card, CD-ROM driver etc. Each of these peripherals has a microcontroller inside it that performs one task e.g. a printer is an example of embedded system as the processor inside it performs only one task i.e. getting the data and printing it.

1.12 Introduction to 8051

- The Intel 8051 is a Harvard architecture, single chip microcontroller that was developed by Intel in 1980 for use in embedded systems.
- It is one of the most popular 8 bit microcontrollers. It can address 128 kbytes of external memory and has a basic instruction time of 1 μs (at 12 MHz).
- Intel's original 8051 family was developed using NMOS technology. The later versions were developed using CMOS technology.

- The other manufactures of 8051 are Atmel, Infineon Technologies, Maxim Integrated Products, NXP, ST Microelectronics, Texas Instruments, Silicon laboratories and Cypress semiconductor.
- 8051 includes a **Boolean processing engine**. The function of Boolean processing engine is to perform logical operations on the data. This feature is very useful for different industrial control applications.
- It has four separate register banks that can be used to reduce interrupt latency compared to common method of storing interrupt context on stack.
- The 8051 UARTs make it simple to use the chip as a serial communications interface.
- The 8051 microcontroller runs at 12 clock cycles per machine cycle. Most of the instructions are executed in one or two machine cycles. With a clock frequency of 12 MHz the 8051 can execute one million one-cycle instructions per second or 500,000 two-cycle instructions per second.

1.13 8051 Family Devices and Derivatives

SPPU - May 12, May 13

University Question

Q. State family members and resources of 8051 microcontroller series.

(May 2012, May 2013, 8 Marks)

- A microcontroller family member has the same architecture, the difference is in the number of pins and the packaging mode.
- Fig. 1.13.1 shows the families of 8051 series of microcontrollers.

Applications : They are **used in applications** like mobile phones, MP3 audio systems, MPEG processing, image processing, aerospace systems, automated automobiles and other high end embedded computing systems.

1.13.1 Overview of the 8051 Family

- Intel introduced 8 bit microcontroller 8051 in 1981. It has 128 bytes of RAM and 4 KB on-chip ROM, two timers, one serial port, four I/O ports on-chip.
- 8051 is an 8-bit Boolean processor. It indicates that it can operate on 8 bit data at a time.

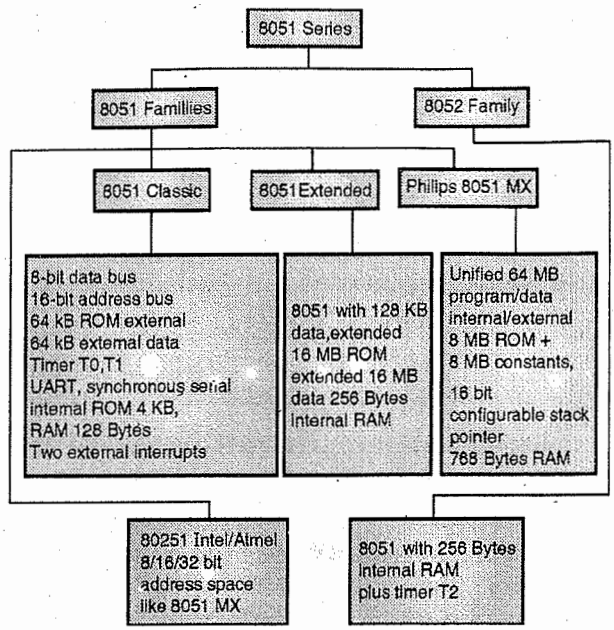


Fig. 1.13.1 : Families of 8051 series

- 8051 has become popular after intel allowed other manufacturers to make 8051 with the condition that they remain code-compatible with 8051. This has led to different versions of 8051 with different speeds and on-chip ROM.
- All the versions are compatible with original 8051.
- 8052 and 8031 are 8051 family members.

8052 Microcontroller

- It supports all the standard features of 8051. It has an extra 128 bytes of RAM and an extra timer T_2.
- 8052 has three timers and 256 bytes of internal RAM. It also has 8 KB on-chip ROM.

8031 Microcontroller

- It is called as **ROM less 8051**. This is because it has 0 KB of on-chip ROM. Hence, inorder to use 8031 we need to add extra ROM to it.
- The extra ROM requires program that 8031 will fetch and execute.
- ROM that can be attached to 8031 can be as large as 64 Kbytes.
- 8031 looses two ports, if external memory is added.

Different 8051 Microcontrollers

- 8051 is generally not seen in the port number as it is available in different memory types like UV-EPROM, flash, NVRAM. The UV-EPROM version of 8051 is 8751.
- The flash ROM version is marketed by Atmel corporation and Dallas semiconductor. The Atmel flash 8051 is called as AT89C51 and Dalls semiconductor calls them DS89 C4x0.
- The NVRAM version is manufactured by Dallas semiconductor. It is called DS5000.
- There is also an OTP (One time programmable) version of 8051 made by various manufacturers.

8751 Microcontroller

- It has 4 KB of on-chip UV-EPROM.
- If this chip is to be used for development purpose, then a PROM burner and UV-EPROM eraser is required.
- As it supports on-chip ROM, it requires approximately 20 minutes to erase the 8751 microcontroller before it can be reprogrammed.

DS89C4x0 from Dallas semiconductor (Maxim)

- To eliminate the need of PROM burner, Dallas semiconductor has introduced DS89C4x0 that can be programmed through the serial COM port of the IBM PC.
- The on-chip ROM is flash ROM.
- DS89C4x0 (420/430/440/450) supports an onchip loader. It supports programs to be loaded into the on-chip flash ROM with the help of serial COM port of a IBM PC.
- The in-system program loading of DS89C4x0 through serial port makes it ideal home development system.
- The NV-RAM version of 8051 is called as DS5000. One byte can be modified at a time. It supports a loader.

Table 1.13.1 : List versions of 8051 microcontroller from Dallas semiconductor

Number	ROM	RAM	I/O Pins	Timers	Interrupts	V_{cc}
DS89C420/ DS89C430	16 KB (Flash)	256	32	3	6	5V
DS89C440	32 KB (Flash)	256	32	3	6	5V
DS89C450	64 KB (Flash)	256	32	3	6	5V
DS5000	8 KB NVRAM	256	32	3	6	5V
DS80C320	0 KB	256	32	3	6	5V
DS87520	16 KB (UVRAM)	256	32	3	6	5V

Table 1.13.1 list version of 8051 microcontrollers from Dallas semiconductor (Maxim).

AT89C51 from Atmel corporation

Table 1.13.2 : List versions of 8051 from Atmel corporation

Port number	ROM	RAM	I/O Pins	Timer	Interrupt	V_{cc}	Packaging
AT89C51	4 KB	128 bytes	32	2	6	5V	40
AT89LV51	4 KB	128 bytes	32	2	6	3V	40
AT89C1051	1 KB	64 bytes	15	1	3	3V	20
AT89C2051	2 KB	128 bytes	15	2	6	3V	20
AT89C52	8 KB	128 bytes	32	3	8	5V	40
AT89LV52	8 KB	128 bytes	32	3	8	3V	40

C : indicates CMOS

OTP version of 8051

- Some of the versions of 8051 are one time programmable (OTP). The versions that are used for product development are the flash and the NV-RAM versions.
- The OTP version of 8051 is used for bulk production in cases where a product is completely designed and finalized. The price per unit is less.

8051 Family from philips

Philips is another major producer of 8051. Their products include A/D converters, extended I/O and both OTP and flash.

1.14 Features of 8051 SPPU - May 13

University Question

Q. State salient features of 8051 microcontroller.

(May 2013, 3 Marks)

The features of 8051 microcontroller are as follows :

(1) 8 bit CPU optimized for control applications.
(2) 4 KB of on chip program memory.
(3) 128 bytes of on chip data memory.
(4) 64 KB program external ROM and 64 KB external RAM addressability.
(5) 32 bidirectional and individually addressable I/O lines arranged as four 8 bit ports P0-P3.
(6) Two 16 bit timer/counters.
(7) Full duplex serial data transmitter/receiver.
(8) Four register banks.
(9) 8 bit program status word and stack pointer.
(10) Interrupt structure with two priority levels.
(11) On chip oscillator and clock circuits.
(12) Direct bit and byte addressability.
(13) Binary or decimal arithmetic.
(14) Signed-overflow detection and parity computation.
(15) Integrated Boolean processor for control applications.
(16) Full depth stack for subroutine return linkage and data storage.

1.15 Pin Functions of 8051

The pin diagram of 8051 is shown in Fig. 1.15.1. It is available in a standard 40 pin dual in line package.

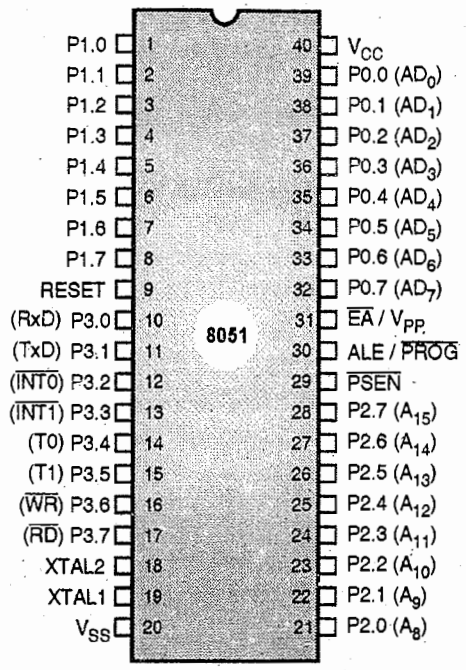
- The devices 8031, 8751 have the same pin-out, same timing and same electrical characteristics. The difference lies in the on chip program memory, that is different for different user requirements.

V_{CC} : Supply voltage

GND : Ground

Port 0

- Port 0 is an 8-bit open drain bidirectional I / O port.
- As an output port each pin can sink eight TTL inputs.
- When 1s are written to port 0 pins, the pins can used as high-impedance inputs.



m(19.3)**Fig. 1.15.1 : Pin diagram of 8051**

- Port 0 may also be configured to be the multiplexed low-order address / data bus during accesses to external program and data memory.
- Port 0 also receives the code bytes during Flash Programming, and outputs the code bytes during program verification.

Port 1

- Port 1 is an 8-bit bidirectional I / O port with internal pull-ups.
- The Port 1 output buffers can sink / source four TTL inputs.
- When 1s are written to Port 1 pins are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 1 pins that are externally being pulled low will source current because of the internal pull ups.

Port 2

- Port 2 is a 8-bit bidirectional I / O port with internal pull ups.
- The port 2 output buffers can sink / source four TTL inputs.
- When 1s are written to port 2 pins they are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 2 pins that are externally being pulled low will source current due to internal pull-ups.

Port 3

- Port 3 is an 8-bit bidirectional I / O port with internal pull-ups.
- The port 3 output buffers can sink / source four TTL input.
- When 1s are written to port 3 pins they are pulled high by the internal pull-ups and can be used as inputs. As inputs, port 3 pins that are externally pulled low will source current due to internal pull-ups.
- Port 3 also serves various other functions as listed in Table 1.15.1.

Table 1.15.1

P3.0	RxD (serial input port)
P3.1	TxD (serial output port)
P3.2	$\overline{INT0}$ (external interrupt)
P3.3	$\overline{INT1}$ (external interrupt)
P3.4	T0 (Timer / Counter 0 external input)
P3.5	T1 (Timer / Counter 1 external input)
P3.6	$\overline{WR}$ (external data memory write strobe)
P3.7	$\overline{RD}$ (external data memory read strobe)

- Port 3 also receives some control signals for flash programming and program verification.

RST (Reset input)

A high on this input pin for two machine cycle, while the oscillator is running resets the device.

ALE / $\overline{PROG}$

SPPU - Dec. 13

University Question

Q. Explain the function of ALE / $\overline{PROG}$ pin of 8051.

(Dec. 2013, 2 Marks)

Address latch enable output pulse for latching the low byte of address during the access to external memory.

- This pin is also program pulse input ($\overline{PROG}$) during Flash programming. When 8051 is switched ON (or reset), it checks this pin (i.e. ALE/$\overline{PROG}$). If the pin is given logic '0' externally, it enters into flash programming mode; else it enters into normal execution mode.
- In normal operation ALE is emitted at constant rate of 1/6 of the oscillator frequency, and may be used for external clocking or timing purposes.
- If desired ALE operation can be disabled by setting bit 0 of SFR location 8EH. When the bit set, ALE is active only during some instructions like MOVX and MOVC.
- Setting the ALE-disable bit has no effect if the microcontroller is in external execution mode.

$\overline{PSEN}$

SPPU - Dec. 13

University Question

Q. Explain function of $\overline{PSEN}$ pin of 8051.

(Dec. 2013, 2 Marks)

- Program store enable is the read strobe to external program memory.
- When 8051 is executing code from external program memory, $\overline{PSEN}$ is activated twice each machine cycle, except that two $\overline{PSEN}$ activations are skipped during each access to external data memory.

$\overline{EA}$/V_{PP}

SPPU - Dec. 13

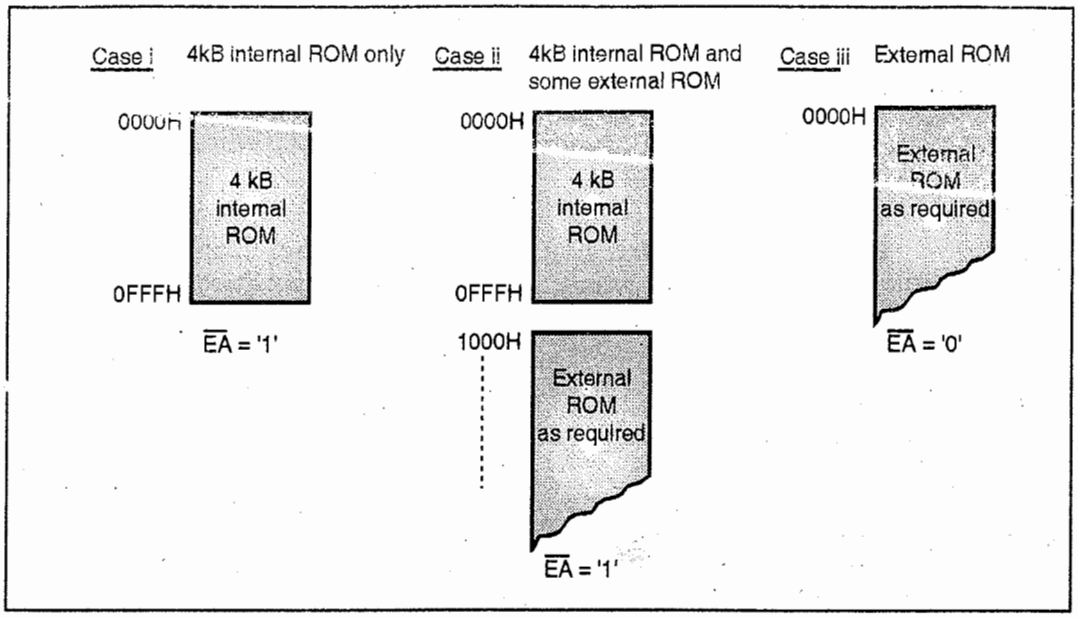
University Question

Q. Explain function of $\overline{EA}$/V_{PP} pin of 8051.

(Dec. 2013, 2 Marks)

External access enable. $\overline{EA}$ must be pulled down to ground so that the microcontroller can fetch the code from external program memory locations beginning from 0000H to 0FFFH.

- $\overline{EA}$ should be pulled to V_{CC} for fetching from internal program memory.
- This pin also receives 12V, programming enable voltage (V_{PP}), during flash programming.
- The effect of $\overline{EA}$ pin is as shown in Fig. 1.15.2.



m(19.4)**Fig. 1.15.2 : Effect of $\overline{EA}$ pin**

TXD

SPPU - Dec. 13

University Question

Q. Explain function of TXD pin of 8051.

(Dec. 2013, 2 Marks)

It is the transmit data pin for serial port in UART mode. Its function is to transmit the data serially.

XTAL1

Input to the inverting oscillator amplifier and input to the internal clock operating circuit.

XTAL2

Output from the inverting oscillator amplifier. A crystal may be connected between XTAL1 and XTAL2 pins.

Syllabus Topic : Block Diagram and Explanation of 8051 (Architecture)

1.16 Block Diagram and Explanation of 8051 (Architecture)

SPPU - May 14

University Question

Q. Draw and explain architecture of microcontroller.

(May 2014, 8 Marks)

Fig. 1.16.1 shows the architectural block diagram of 8051. It consists of following elements :

(i) Eight bit register A (Accumulator) and register B.

(ii) Arithmetic and Logic Unit (ALU).

(iii) 16 bit Program counter (PC).

(iv) 16 bit Data pointer (DPTR).

(v) 8 bit Program Status Word (PSW).

(vi) 8 bit stack pointer (SP).

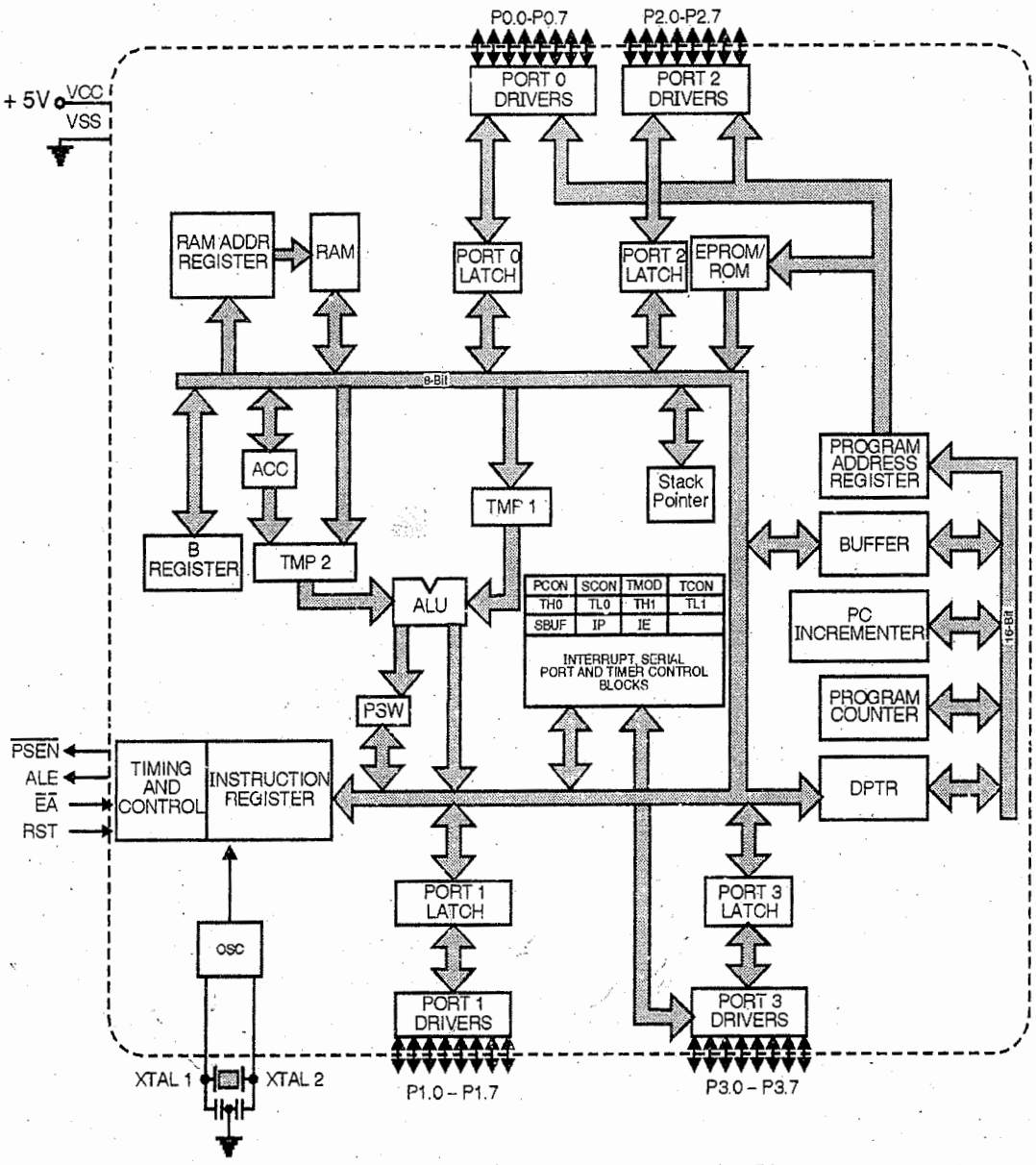
(vii) 128 byte Internal RAM and 4 KB Internal ROM.

(viii) Four 8 bit ports : Port 0, Port 1, Port 2, Port 3.

(ix) Two 16 bit Timer/counters, serial port and interrupt control.

(x) Control Registers.

(xi) On chip oscillator.



m(19.5)**Fig. 1.16.1 : Architectural block diagram of 8051**

Syllabus Topic : Overview of MCS-51 Architecture

1.16.1 Overview of MCS-51 Architecture

- **Harvard Architecture :** Separate Program and Data Memory.
- **High Integration :** Built-in Program (ROM) and Data (RAM) Memory, Timers, Interrupt Control, Serial and Parallel Ports.
- The Important components into the 8051 internal architecture can be expressed as follows :
  - Internal Clock Oscillator driving the system clock generated from the external Crystal Oscillator connected across XTAL1, XTAL2 pins.
  - Core MCS-51 CPU : The Processor logic that executes the MCS-51 instruction set Instructions and drive the processor engine.

- Interrupt control logic : Manages Interrupts from 5 Interrupt sources (2 External : #INT0 and #INT1 and 3 Internal – Timer 0, Timer 1 overflow and Serial Interrupt) based on the IE and IP - SFR configured by user.
- 4KB internal ROM used as Internal program memory.
- 128 Byte RAM : Used as internal data memory – Constitutes Register Banks(00-1Fh), Bit Addressable Memory(20-2Fh), User Memory (30-7Fh)
- Two 8/16 Bit Timers : Timer 0 and Timer 1 – Operating in 4 possible modes (Mode 0 to Mode 3) as per the TCON, TMOD, TL0, TH0, TL1, TH1 – SFR configured by the user.
- Asynchronous full duplex Serial Port compliant to UART (Universal Asynchronous Receiver/ Transmitter) operating 1 of 4 Modes based on SCON and SBUF – SFR registers configured by the user.
- 4 Parallel Ports of 8 Bits each – A total of 32 Digital I/O lines – Quasi – bi-directional, General Purpose – some of them with Alternate functions – Accessed by SFR – Port 0, Port 1, Port 3, Port 4.
- Bus Control Logic that controls the operations on the buses and helps 8051 access the external memory.

1.16.2 Accumulator (ACC)

SPPU - May 14

University Question

Q. Explain the use of the following register : Accumulator. **(May 2014, 2 Marks)**

Size : ACC is the accumulator register. It is an 8 bit register. It address is E0H. It is a bit addressable register.

Function : It is most versatile and holds source operand and receives the result of arithmetic operations including addition, subtraction, integer multiplication, division and Boolean bit manipulations.

Uses : (i) It is also used for data transfer between 8051 and any external memory.

(ii) Several functions like rotate, swap etc. apply specifically on the accumulator.

(iii) It is used along with register B for multiplication and division operations.

1.16.3 B Register

SPPU - May 14

University Question

Q. Explain the use of the following register : Register B **(May 2014, 2 Marks)**

Function and use :

The B register is used with the A register for multiplication and division operations. For other instructions it is treated as a scratch pad register. (i.e. it has no other function other than as a location where data can be stored). Its address is F0H. It is a bit addressable register.

1.16.4 Arithmetic and Logic Unit (ALU)

Function : The ALU can perform arithmetic and logic operations on eight bit data. It can perform arithmetic operations like addition, subtraction, multiplication, division and logical operations like AND, OR, EX-OR, complement, rotate etc.

- The ALU also takes care of branching instructions.

1.16.5 Program Status Word (PSW) and Flags

SPPU - Dec. 12, May 13, May 14, May 16

University Questions

Q. Explain the significance of PSW register with the help of example. **(Dec. 2012, 4 Marks)**

Q. Explain PSW register of 8051. **(May 2013, May 2014, 4 Marks)**

Q. Draw and explain the flag structure of 8051 with bank 2 selection. **(May 2016, 6 Marks)**

Many instructions affect the status flags. In order to address these flags conveniently they are grouped to form the **program status word**.

What is a flag ? Flags are 1 bit registers provided to store the status of the result of some instructions. A Flag is a flip flop that indicates some condition produced by the execution of an instruction. e.g. : The carry flag (CY) will be set if there is a carry or borrow out of the MSB of the result.

Size of PSW :

Fig. 1.16.2 shows the 8051 PSW. It is of **8 bits.**

- It contains (i) math flags (ii) user program flag F0 (iii) register select bits that identify the register bank that is currently in use. Its reset value is 00H.
- The 8051 has **four math flags** that include carry (CY), Auxiliary carry (AC), overflow (OV) and Parity (P).

CY flag : The carry flag will be set when there will be a carry or borrow out of the MSB (D7 bit) of result. It is used for detecting errors from unsigned arithmetic operations.

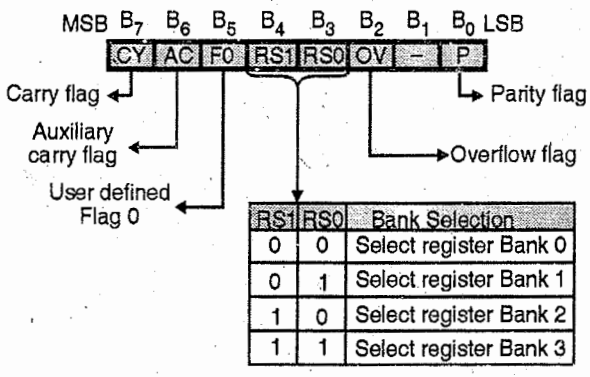
AC flag : The auxiliary carry flag is set, whenever there is a carry out of the lower nibble into the higher nibble or whenever there is a borrow from higher nibble into the lower nibble.

OV flag : The overflow flag will be set, if an arithmetic overflow has occurred i.e. a significant bit has been lost because the size of the result exceeded the capacity of its destination location. It is used for detecting errors in signed arithmetic operations.

PF flag : The parity flag is set when the result has even parity, i.e. even number of 1's.

Bits used for bank selection :

The **bits RS0 and RS1** are used for selecting one of the register banks as shown in Fig. 1.16.2.



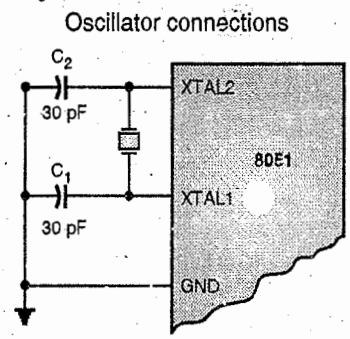
RS1	RS0	Bank Selection
0	0	Select register Bank 0
0	1	Select register Bank 1
1	0	Select register Bank 2
1	1	Select register Bank 3

m(19.6)**Fig. 1.16.2 : Program status word (PSW)**

User flags : The 8051 also has three general purpose user flags that can be set to 1 or cleared to 0 by the programmer as desired. The user flags are named F0, GF0 and GF1. The PSW contains user program flag F0, while GF0 and GF1 flags are stored in the PCON register.

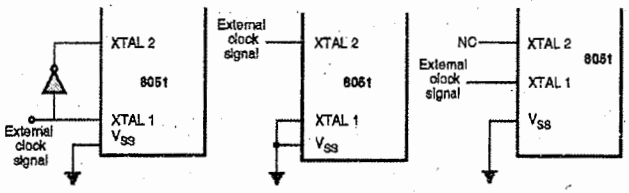
1.16.6 Clock and Oscillator

- **The 8051 has an on-chip oscillator, but needs an external clock to run it.**
- The oscillator circuit that generates the clock pulses so that all the internal operations are synchronized.
- The pins XTAL 1 and XTAL 2 are provided to connect a resonating network comprising of a crystal and capacitors to form an oscillator. Normally quartz crystal is used. The capacitors are used inorder to stabilize the network. Fig. 1.16.3 shows the oscillator circuit. Generally the capacitors C_1 and C_2 are of 30 pF.
- **The crystal frequency is the internal clock frequency of the microcontroller.**



m(19.7)**Fig. 1.16.3 : Connection of external crystal to 8051**

- The manufacturers provide 8051 microcontroller designs that run a frequencies ranging from 1 MHz to 16 MHz. Fig. 1.16.4 shows how 8051 is driven by an external clock signal.



m(19.8)**Fig. 1.16.4 : Using an external clock**

1.16.7 Program Counter (PC)

SPPU - May 14, Aug. 15

University Question

Q. Explain the use of the following register : PC **(May 2014, Aug. 2015(In Sem.), 2 Marks)**

Size : It is a **16 bit register.**

Uses : It is used to hold the address of a instruction in the memory.

Function : Its function is to keep the track of the execution of the program. The program instruction bytes are fetched from locations in memory that are addressed by the Program counter.

PC reset address : Whenever the power supply is switched on, the PC resets to 0000H. The PC is automatically incremented after fetching a byte from the program memory.

- In case of instructions like jump, call, interrupt the contents of PC may change.
- The PC is only a register that does not have an internal address.

1.16.8 Data Pointer (DPTR)

SPPU - May 14, Aug. 15

University Question

Q. Explain the use of the following register : DPTR

(May 2014, Aug. 2015(In Sem.), 2 Marks)

Size : The data pointer is a 16 bit register.

Uses : (i) It is used to hold the address of data in the memory.

(ii) It can be used as a 16 bit data register or two independent 8-bit registers.

Function : It serves as a base register in case of instructions handling look up tables and external data transfer.

- The DPTR register can be accessed separately as lower eight bits (DPL) and higher eight bits (DPH). DPL and DPH are 8 bit registers. The address of DPL is 82H and address of DPH is 83H.
- The DPTR does not have a single internal address instead DPL and DPH are each assigned a separate address.

1.16.9 The Stack and Stack Pointer

SPPU - May 13

University Question

Q. Explain the stack operation and stack pointer register of 8051. What is its reset value ?

(May 2013, 4 Marks)

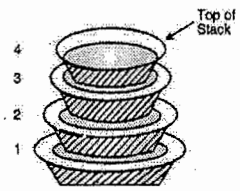
Function : The stack is a reserved area of the memory in RAM where temporary information may be stored. An 8-bit stack pointer is used to hold the address of the most recent stack entry. This location which has the most recent entry is called as the **top of the stack**.

- When the information is written on the stack, the operation is called **PUSH**. When the information is read from the stack, the operation is called **POP**. The stack works on the principle of **Last In First Out** or **First In Last Out**.
- The microcontroller stores the information/data like stacking plates. Fig. 1.16.5 shows stacked plates. If we want to remove the first stack plate, then we have to remove all the plates above the first i.e. we have to remove the fourth plate, third plate, second plate and then finally the first plate. This indicates that the first plate pushed onto the stack is the last one to be popped from the stack. This operation is called as **First In Last Out (FILO)**.

Size : The stack is implemented with the help of special memory pointer register called as the **stack pointer**. It is of 8 bit.

Reset address of stack : This indicates that it can take values from 00 to FFH. On **power up, SP contains the value 07H.**

- The stack pointer's contents are automatically adjusted to top of the stack. The memory location that is currently pointed by the stack pointer is called as **top of stack**. As shown in Fig. 1.16.5 the 4th stacked plate represents top of stack.
- When data is to be stored on the stack, the SP increments before storing the data on stack and when the data is to be retrieved/popped from the stack the SP decrements to point next available byte of stored data.



m(19.9)**Fig. 1.16.5 : Stacked plates**

- The stack array can reside anywhere in the on-chip RAM. However, its height is limited to the size of internal RAM.
- The locations 08H to 1FH in the 8051 RAM can be used for the stack. This is because the locations 20-2FH of RAM are reserved for bit-addressable memory and cannot be used by the stack.

- If in a program we need more than 24 bytes of stack, we can change SP to point to RAM locations 30-7FH
- The stack can overwrite data in the register banks, bit-addressable RAM, scratch-pad RAM areas. So, normally the stack is placed at a higher location in internal RAM to avoid conflict with the register and bit addressable internal RAM areas.

Uses

(i) The stack can be used to save the register contents.

(ii) The CPU uses the stack to save the address of the instruction just below the CALL instruction. This will indicate the CPU where to resume after it returns from the called subroutine.

1.16.9.1 Stack and Bank 1 Conflict

SPPU - Dec. 12.

University Question

Q. Explain why the stack pointer is initialized to 07h after a reset. **(Dec. 2012, 4 Marks)**

- After a reset, the stack pointer is initialized to 07H. So, the stack will begin at location 08H.
- The stack pointer register points to the current RAM location available from the stack. As data is pushed onto the stack, SP is incremented. Conversely it is decremented as data is popped off the stack into the registers. SP is incremented after the push instruction to check that the stack is growing towards RAM location 7FH from lower address to the upper address.
- If after the PUSH instruction we decrement the stack pointer then the RAM locations used are 07H, 06H, 05H. However, these locations represent registers R7 to R0 of register Bank 0.
- The incrementing of the stack pointer for push instructions confirms that the stack will not reach location 0 at bottom of RAM and stack will run out of space.
- However there is a conflict with default setting of stack. As SP = 07H when 8051 is powered up, the starting location of stack is RAM location 08H. This location belongs to register R0 of register bank 1 i.e. the register bank 1 and stack are using same memory space.

Syllabus Topic : Port Structure

1.17 Port Structure

SPPU - May 12, Dec. 12, Dec. 13, Aug. 15

University Questions

Q. With the help of port structure explain why it is necessary to send logic one on port pin before performing read operation. **(May 2012, 8 Marks)**

Q. Explain port structure of 8051 in detail and configure ports as input and output. **(Dec. 2012, 9 Marks)**

Q. Explain port structure of 8051 in detail. **(Dec. 2013, 8 Marks, Aug. 2015 (In Sem.), 4 Marks)**

No. of I/O ports and size : The microcontroller has four ports named P0, P1, P2 and P3. All these ports are bi-directional and of 8 bit. Each of these 8 bit ports consists of a D-type output latch, an output driver and an input buffer.

- All ports upon Reset are configured as input ports.
- As functions are multiplexed on same port pins, in order to decide which function is supported we need to see how the circuit is connected and what software commands are used to "program" the pin.
- The four ports are required for I/O operations. Out of the 40 pins, 32 pins are set aside for the four ports P0, P1, P2 and P3. Each port takes 8 pins.
- Each port is an 8 bit port.
- When zero is written to a port, it becomes an output. To make it an input port, 1 needs to be sent to the port.

Now let us see the ports one by one.

1.17.1 Port 0

SPPU - Aug. 14, Dec. 16

University Questions

Q. Explain structure of Port 0 of 8051. **(Aug. 2014 (In Sem.), 2 Marks)**

Q. Draw and explain PORT 0 structure of 8051 microcontroller. **(Dec. 2016, 4 Marks)**

Function and use : Port 0 is a multifunctioned port of microcontroller 8051. It can be used as simple input / output mode or for generating data and lower order address bus for external memory ($AD_0 - AD_7$).

- Fig. 1.17.1 shows Port 0 circuit. Its address is 80H. It is a bit addressable port.

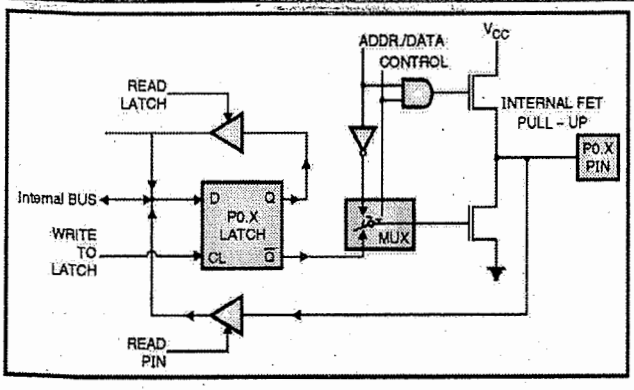


Fig. 1.17.1 : Port 0 circuit

- Port 0 does not have an internal pull up, but whenever port 0 is configured as an output port pull up is required.
- In order to use the pins of port 0 as input and output each pin must be connected to 10 KΩ pull up resistor as shown in Fig. 1.17.2. It is because Port 0 is an **open drain**.

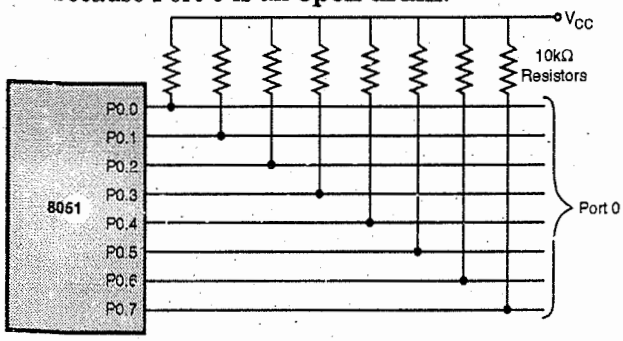


Fig. 1.17.2 : Port 0 with, Pull-up resistor

1.17.1.1 Port 0 as Simple Input Port

- When port 0 is used as an input port, '1' must be written to the corresponding port 0 latch that will cause both the output transistors to switch off and the pin **"floats"** in a high impedance state. Hence it is called as **"true bidirectional"** port because when configured as an input port it floats. When configured as input port the microcontroller provides two facilities :
  (i) Reads logic level on physical pin by asserting READ pin signal.
  (ii) Reads the contents of internal latch by asserting the READ LATCH signal. The latch is read when the instruction is a Read-Modify-Write type of instruction. A Read-Modify-Write instruction is one, wherein the instruction **Reads** the data from the port (latch) **Modifies** (performs some operation on) it and **Writes** to the port (latches and hence pins).

1.17.1.2 Port 0 as Simple Output Port

- When port 0 is configured as an output port, the latch pins that are programmed to 0 will cause the lower FET to turn on and pin is grounded.
- If a '1' is written on to the latch pin the FET will turn off and the pin is pulled HIGH by external pull up resistors.

1.17.1.3 Port 0 used as Multiplexed Address / Data bus (AD_0 – AD_7) for External Memory

- When micro-controller accesses external program memory or data memory the address for memory is generated by port 0 and port 2.
- Port 0 generates the lower order address $A_7 - A_0$, while port 2 generates the higher order address $A_8 - A_{15}$.
- When port 0 is used as an address bus to the external memory, the internal control signals switch the address lines to the gate of FETs.
- If a logic 1 is written onto the address bit, then the upper FET will turn on and the lower FET will turn off providing a logic HIGH at the pin. If a logic 0 is written onto the address bit, then the upper FET will turn off and the lower FET will turn on providing a logic LOW at the pin. Once, the 8 bit address is been formed and latched by the Address Latch Enable (ALE) pulse into the external circuits, the bus turns around to data bus. Port 0 can now read data from the external memory. For reading data it must be configured as a input port. Fig. 1.17.3 shows multiplexed address / data bus.

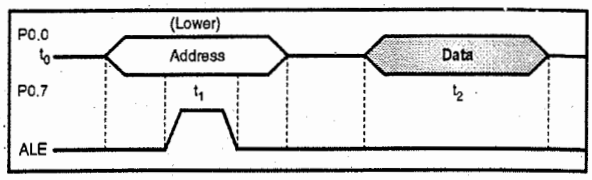


Fig. 1.17.3 : Multiplexed address / data bus

1.17.2 Port 1

SPPU - Aug. 14

University Question

Q. Explain structure of Port 1 of 8051.
(Aug. 2014 (In Sem.), 2 Marks)

- Port 1 is a simple I/O port of microcontroller. Its address is 90H. It does not have any extra function. Hence, the output latch is connected directly to the gate of the lower FET, consisting of a circuit labelled internal pull up. Fig. 1.17.4 shows port 1 circuit.

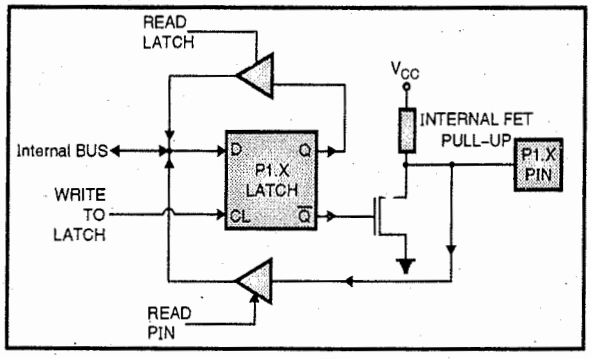


Fig. 1.17.4 : Port 1 circuit

1.17.2.1 Port 1 as Simple Input Port

- When port 1 is used as an input port, "1" must be written to the corresponding port 1 latch bit. This causes the lower FET to turn off. The pin and input to pin buffer are pulled to logic HIGH by the internal pull up load.
- Port 1 is called as **"quasi-bidirectional"** port as its output is pulled high with pull up resistors.

1.17.2.2 Port 1 as Simple Output Port

- When port 1 is used as an output port, the latch pins that are programmed to 0, will cause the lower FET to turn on, the internal pull up to turn off and input to the circuit is logic 0.
- If '1' is written onto the latch pin then it will drive the input of external circuit high through the pull up. The lower FET turns off.
- The internal FET pull up is used to help the port 1 to speed up when it is used as an output port. The internal FET pull up has another FET that is in parallel to lower FET. This FET is turned on for two clock periods when the transition on the pin is low to high. This arrangement provides a low impedance path to the positive supply voltage.

1.17.3 Port 2

SPPU - Dec. 16

University Question

Q. Draw and explain PORT2 structure of 8051 microcontroller. **(Dec. 2016, 4 Marks)**

Function and use : Port 2 of the microcontroller 8051 is a multifunctional port. **It can be used as a simple input / output port or for generating the upper order address bus for external memory ($A_8 - A_{15}$).** Fig. 1.17.5 shows port 2 circuit. Its address is A0H. It is a bit addressable port.

- Port 2 circuit contains D-type latch, multiplexer, two unidirectional buffer and FET output stage with pull up.

1.17.3.1 Port 2 as Simple Input Port

- When port 2 is used as an input port, "1" must be written to the corresponding port 2 latch bit. This causes the FET to turn off. The pin and input to pin buffer are pulled to logic HIGH by the internal pull up load.
- Port 2 is called as **"quasi-bi-directional port"** as its output is pulled high with pull up resistors.

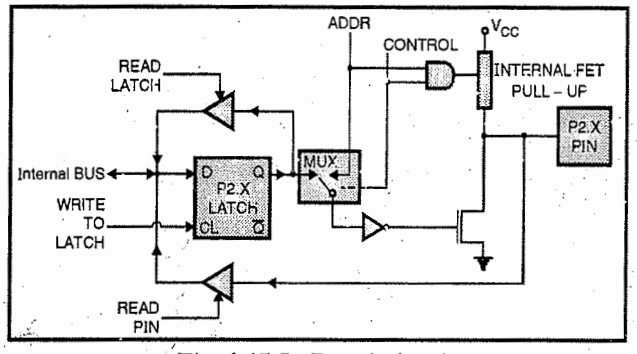


Fig. 1.17.5 : Port 2 circuit

1.17.3.2 Port 2 as Simple Output Port

- When port 2 is used as an output port, the latch pins that are programmed to 0, will cause the lower FET to turn on, the internal pull up to turn off and input to the circuit is logic 0.
- If "1" is written onto the latch pin then it will drive the input of external circuit high through the pull up. The lower FET turns off.

1.17.3.3 Port 2 used as Higher Order Address Bus ($A_8 - A_{15}$) for External Memory

- The port 2 pins are momentarily changed, due to the address control signals when it is supplying the higher order address. The latch remains stable because it does not have to turn around for data input as in port 0.

Fig. 1.17.6 shows how port 2 generates address in conjunction with ALE.

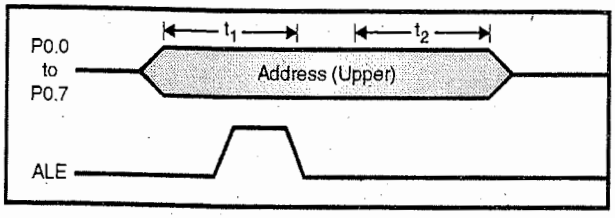


Fig. 1.17.6 : Port 2 generates higher order address in conjunction with ALE

1.17.4 Port 3

Use : Port 3 is a multifunctional port it can be used as a simple input / output port. The port 3 pins have special functions. Its address is B0H. It is a bit addressable port.

1.17.4.1 Functions of Port 3 Pins

Table 1.17.1 : Functions of port 3 pins

PIN	Symbol	SFR	Significance
P 3.0	RxD	SBUF	It is the receive data pin for serial port in UART mode.
P 3.1	TxD	SBUF	It is the transmit data pin for serial port in UART mode. It works as the clock output in the shift register mode.
P 3.2	$\overline{INT0}$	TCON 1	It is an external interrupt. It is low level or falling edge triggered.
P 3.3	$\overline{INT1}$	TCON 3	It is an external interrupt. It is low level or falling edge triggered.
P 3.4	T0	TL0.	External Timer / Counter 0 input pin, gives pulses to TL0 register of the timer 0 to increment by 1.
P 3.5	T1	TL1	External Timer / Counter 1 input pin, gives pulses to TL1 register of the timer 0 to increment by 1.
P 3.6	$\overline{WR}$	-	It is external memory write pulse. It is an active low pulse.
P 3.7	$\overline{RD}$	-	It is an external memory read pulse. Whenever data from memory is read this pulse is active low.

- Unlike the ports 0 and 2, where all the 8 bits simultaneously change for alternate use, each bit of port 3 can be programmed as I/O of to perform one of the functions listed above.
- Fig. 1.17.7 shows port 3 circuit.
- Port 3 bit contains D type latch, three unidirectional buffer, FET with internal pull-up. As the internal pull up is fixed port 3 is called as **"quasi-bidirectional"** port.

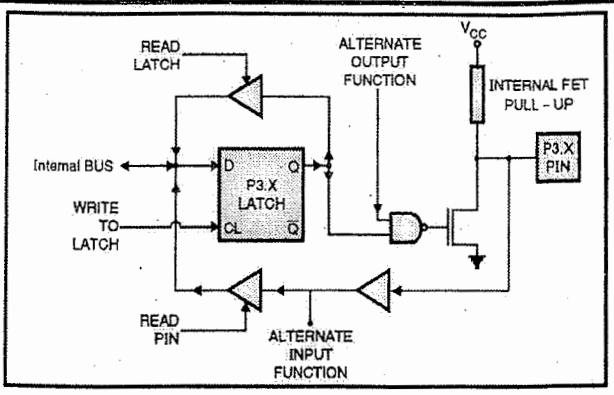


Fig. 1.17.7 : Port 3 circuit

1.17.4.2 Port 3 as Simple Input Port

When port 3 is used as an input port, "1" must be written to the corresponding port 3 latch bit. This causes the FET to turn off. The pin and input to pin buffer are pulled to logic HIGH by internal pull up load.

1.17.4.3 Port 3 as Simple Output Port

- When Port 3 is used as an output port, the latch pins that are programmed to 0, will cause the lower FET to turn on, the internal pull up to turn off and input to the circuit is logic 0.
- If "1" is written onto the latch pin then it will drive the input of external circuit high through the pull up. The lower FET turns off.

1.17.4.4 Port 3 as One of the Alternate Functions

- For achieving any one of the alternate output functions, another control signal called as "alternate output function" is available on Port 3. Depending on the logic level present on line "alternate output function" the FET will turn ON or OFF.
- When the latch bit of Port 3 is at logic 1, the output level is controlled by the control input.

1.17.4.5 Port Loading and Interfacing

- The output buffers of Ports 1, 2 and 3 can each drive 4 LS TTL inputs. Port 0 when used in external bus mode output buffer can drive 8 LS TTL inputs.
- In order to drive inputs these ports as port pins require external pull ups to drive any inputs.

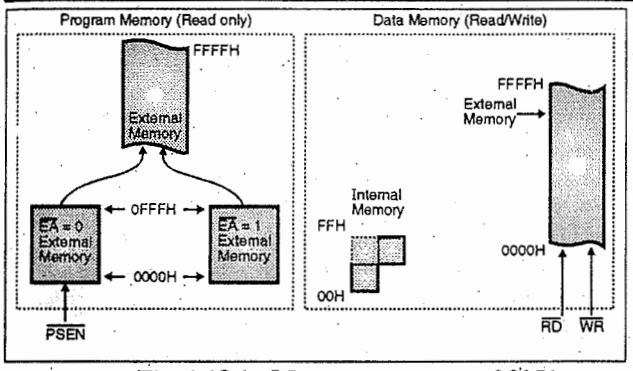
Syllabus Topic : Memory Organisation

1.18 Memory Organisation

SPPU - Dec. 12

University Question

Q. Explain structure of internal memory organization of 8051. **(Dec. 2012, 5 Marks)**



m(19.10)**Fig. 1.18.1 : Memory structure of 8051**

1.18.1 Structure of Internal Memory

- Fig. 1.18.1 shows the basic memory structure for microcontroller 8051. It can access upto **64 KB of program memory and 64 KB of data memory**. It has 4 KB of internal program memory and 256 bytes of internal data memory.
- Each memory has a separate addressing mechanism. It also has different control signals and different functions. Each memory will use the same address and data bus, but with different control signals.

1.18.1.1 Program Memory

- The microcontroller 8051 has 64 KB external and 64 KB internal program memory. Fig. 1.18.1(a) shows the memory map of program memory.
- If the $\overline{EA}$ pin is connected to V_{cc} then the program fetches the addresses from 0000H through 0FFFH are directed to the internal memory and the addresses from 1000H through FFFFH through the external ROM
- If the $\overline{EA}$ pin is grounded then all the addresses from 0000H through FFFFH are fetched by the 64 KB external ROM/EPROM. The $\overline{PSEN}$ signal activates the output enable signal for ROM/EPROM.

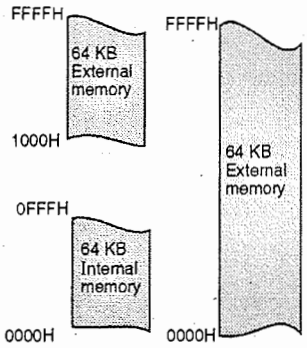


Fig. 1.18.1(a) : 8051 Program memory

1.18.1.2 Data Memory

- 8051 microcontroller can address upto 64 KB of external data memory and 256 bytes of internal data memory.
- Fig. 1.18.1(b) shows a memory map of the 8051 data memory.

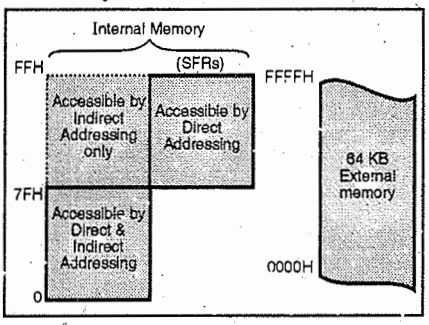


Fig. 1.18.1(b) : 8051 data memory map

- The instruction "MOVX" is used to access the external data memory. The internal data memory space is divided into three blocks lower 128 bytes (00H to 7FH), upper 128 bytes (80H to FFH) and the SFRs. The upper 128 bytes and SFRs occupy the same block of address space, although both are separate.

1.18.2 Internal Memory Organization

SPPU - Dec. 15

University Question

Q. Explain structure of internal memory organization of 8051. **(Dec. 2015, 6 Marks)**

- For the proper operation of a computer system, the system should have memory for program code. This memory for program code is normally in the ROM.
- The 8051 microcontroller has an internal RAM and ROM. If this memory is insufficient, then additional memory is externally added using suitable circuits.

1.18.2.1 Internal RAM SPPU - Oct. 16

University Question

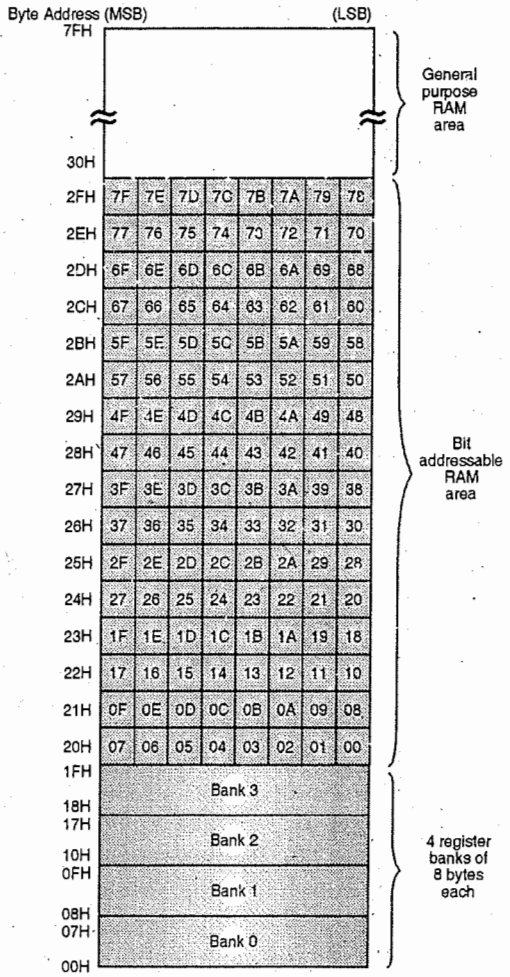
Q. Explain RAM organisation of 8051.

(Oct. 2016 (In Sem.), 5 Marks)

- The 8051 microcontroller has a **128 byte Internal RAM**. This internal RAM is organized in three distinct areas. They are :

(i) Four register banks of 8 bytes each
(ii) Bit addressable area of 16 bytes
(iii) General purpose RAM area (scratch pad area)

Fig. 1.18.2 shows the internal RAM structure with memory map.



m(19.11)**Fig. 1.18.2 : Internal RAM structure**

1.18.2.1(A) Four Register Banks of 8 Bytes each

Q. How many register banks are available in 128 byte RAM of 8051 microcontroller? How many registers are there in each bank?

Register bank organisation :

There are four register banks which are numbered 0 to 3 in the 128 byte RAM of 8051 microcontroller. Each bank is made up of eight registers named R0 to R7. In total (4 × 8 = 32) 32 bytes or 32 working registers from address 00H to 1FH organized as four register banks.

- RAM locations 00H to 07H are set aside for Bank 0 of R0 – R7, R0 is RAM location 00H, R1 is RAM location 01H, R2 is RAM location 02H and so on.
- The second bank of registers begins at 08H – 0FH. The third bank of R0 – R7 begins at 10H – 17H and finally the fourth bank of R0 – R7 at RAM locations 18H – 1FH as shown in Fig. 1.18.3.

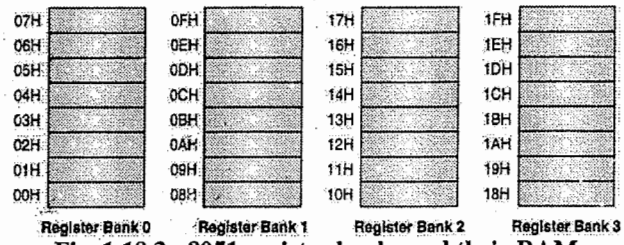


Fig. 1.18.3 : 8051 register banks and their RAM addresses

Register bank selection : For selecting a register bank two bits RS0 and RS1 are provided in the program status word (PSW) as shown in Fig. 1.16.2.

- If any of the register banks is not selected, then the programmer can use the area from 00H to 1FH as simple scratch pad RAM or general purpose RAM.
- Upon power on or reset register BANK 0 is selected by default.

Disadvantage : We know that the register bank 1 uses the same RAM space as the stack. This is a major problem while programming 8051. We must either not use the register bank 1 or allocate another area of RAM for stack.

Advantage of register banks

- The advantage of Register banks is that it reduces the latency period for a subroutine call or an interrupt.
- When the programmer is using a set of registers R0 to R7 of bank 0, and an interrupt occurs :
  (i) For a normal processor (without register banks), the ISR or subroutine begins with pushing the contents of all the registers onto the stack. This avoids overwriting of the data of the caller function. At the end of ISR or subroutine all the contents of these registers are to be popped. This pushing and popping of data is the extra work the processor has to do and the time required to do this is called as latency period.
  (ii) In case of 8051, the programmer need not push the data when an interrupt occurs or when a subroutine is called. Instead the programmer can switch to another set of registers (register bank). And hence the programmer need not even pop the data from stack. This saves the latency period and hence improves efficiency of the system.

Ex. 1.18.1

Write instructions for selecting bank 1 of 8051 Microcontroller.

Soln. :

Select Bank 1 :

```
SETB PSW.3
CLR PSW.4
```

Ex. 1.18.2 SPPU - May 2014, 2 Marks

Write code for selecting bank 2 of 8051.

Soln. :

```
Program to select Bank 2.
SETB PSW.4
SETB PSW.3
```

1.18.2.1(B) Bit Addressable Area of 16 Bytes

Q. What is the address range allocated to bit addressable area in RAM of 8051 ?

Size and address range : The microcontroller has reserved 16 bytes of internal RAM whose address ranges from 20H to 2FH. These 16 bytes provide us with 16 × 8 = 128 bits forming addressable bits. The microcontroller has given addresses to these bits ranging from 0 to 127 (decimal) or 00H to 7FH. Hence, these locations are called **bit addressable locations** as shown in Fig. 1.18.2.

Use : The addressable bits are useful if a binary event in the program needs to be remembered. e.g. open switch, close switch etc.

- Inorder to access the 128 bits of RAM locations and other bit addressable space of 8051, we can use only the single bit instructions. All the single bit instructions support only direct addressing mode.
- For single bit instructions there is no indirect addressing mode.

1.18.2.1(C) General Purpose RAM Area

- This RAM area is also called as **scratch pad area.**
- It lies above the bit addressable area and has address 30H to 7FH. This RAM area can be used as data RAM. The memory in this area is byte addressable.
- The programmer may declare stack in this area, provided sufficient number of bytes are available.

1.18.2.2 Uses of Internal RAM

The uses of internal RAM are :

(i) It is not essential to utilise the entire area of internal RAM of the microcontroller as scratch pad.

(ii) The microcontroller has instructions that allow the internal memory locations to be used as data pointers.

(iii) The registers R0 to R1 in the four registers banks (Bank 0 to Bank 3) can be used as a pointer to point the internal RAM as well as the external RAM. When the registers R0 or R1 are used to access the external RAM, these 8 bits will be treated as the LSB of 16 bit address and the MSB is taken to be 00H by default.

1.18.2.3 Internal ROM

- The microcontroller 8051 has a separate memory for the program and data. Both these memories have same address ranges.

Function : The internal ROM is used to store the internal program code. It occupies the address space ranging from address 0000 H to 0FFF H.

Address range : The PC can access program code bytes from 0000 H to FFFF H. The capacity of internal ROM is from 0000 H to 0FFF H. For addresses higher than 0FFF H the 8051 will automatically fetch the program code bytes from an external memory.

- The PC is not bothered of about where the program code resides, the programmer decides whether the code resides in the internal memory, external memory or combination of the internal and external memory.

1.18.3 External Memory

- External memory is used in cases when the internal ROM and RAM memory available on chip is not sufficient. Two separate external memory spaces are made available by the 16-bit PC and the DPTR and by different control pins for enabling external ROM and RAM chips.
- If the 128 bytes of internal RAM is insufficient, then external RAM is accessed by the DPTR. In the 8051 family, external RAM of upto 64 KB can be added to any chip.
- The external ROM of upto 64 KB can be added to any chip in the 8051 family.

1.18.4 Special Function Registers (SFRs)

- **Special function registers are placed in the address space immediately above the 128 bytes of RAM, from address 80H to FFH.**
- **The SFR memory consists of important registers like accumulator, B register, interrupt control registers, PSW, timer / counter, power control, four I/O ports, serial control. Some of these registers are bit addressable while remaining are byte addressable.**
- Fig. 1.18.4 shows the SFR structure.
- Some of the addresses i.e. locations in between 80H and FFH are not used. If we try to use one of these unused locations that are not defined or are empty, then we may get unpredictable results. When reading from such an unused location a random data will be given. When writing the data to such unused location the data will be lost, i.e. not stored anywhere.
- The PC is not a part of the SFR. The PC (program counter) does not have an internal RAM address.
- SFRs are referenced by their addresses such as 0E0 H, 87 H, 90 H, 0A0 H, 80 H etc.

Byte address	Bit address								
FF									
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
D0	D7	D6	D5	D4	D3	D2	—	D0	PSW
B8	—	—	—	BC	BB	BA	B9	B8	IP
B0	B7	B6	B5	B4	B3	B2	B1	B0	P3
A8	AF	—	—	AC	AB	AA	A9	A8	IE
A0	A7	A6	A5	A4	A3	A2	A1	A0	P2
99	not bit addressable								SBUF
98	9F	9E	9D	9C	9B	9A	99	98	SCON
90	97	96	95	94	93	92	91	90	P1
8D	not bit addressable								TH1
8C	not bit addressable								TH0
8B	not bit addressable								TL1
8A	not bit addressable								TL0
89	not bit addressable								TMOD
88	8F	8E	8D	8C	8B	8A	89	88	TCON
87	not bit addressable								PCON
83	not bit addressable								DPH
82	not bit addressable								DPL
81	not bit addressable								SP
80	87	86	85	84	83	82	81	80	P0

Special Function Registers

m(19.12)**Fig. 1.18.4 : SFR Structure**

1.18.5 CPU Timing and Machine Cycle

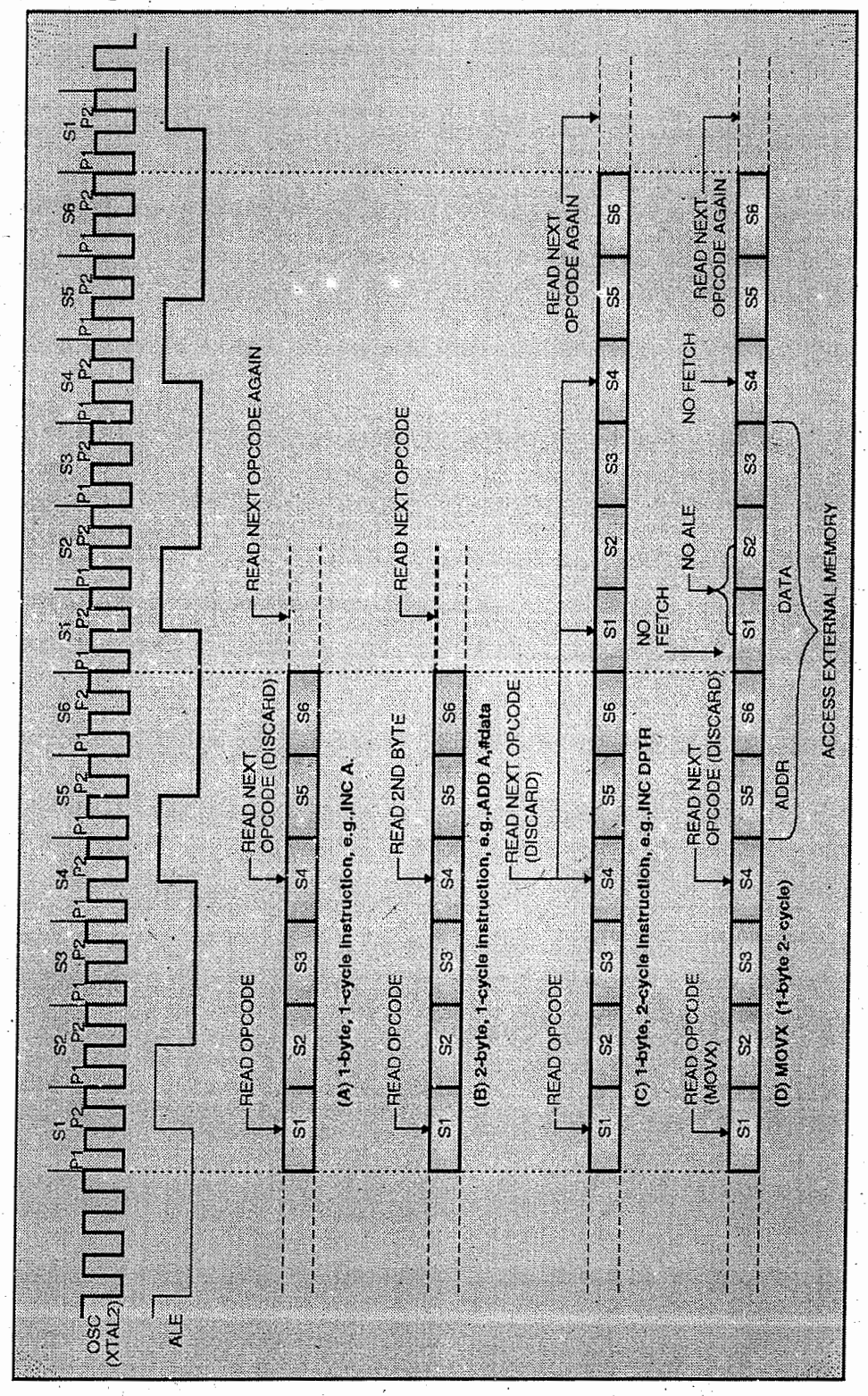


Fig. 1.18.5 : State sequence in MCS-51 device

- A machine cycle consists of sequence of 6 states, numbered S1 through S6. Each state time lasts for two oscillator periods i.e. each state has period 1 and period 2. Therefore, one machine cycle takes 12 oscillator clock periods.
- Each machine state is divided into two period : period 1 and period 2. During period 1, period 1 clock is active and period 2 clock is active during period 2. The machine cycle states will be numbered as S1P1 (State 1, period 1), S1P2 (State 1, period 2) ..... S6P2 (State 6, period 2).
- Each period lasts for one oscillator period. Typically, arithmetic and logical operations occur during period 1 and register to register transfers occur during period 2. Fig. 1.18.5 shows the fetch 1 execute sequence in states and phases for different instructions.
- In case of one byte two cycle instruction e.g. MOVX. The MOVX instruction accesses external data memory. Two fetches are skipped while the external data memory is being addressed and strobed. Fig. 1.18.5 shows the timing diagram.

Ex. 1.18.3

If oscillator frequency of 8051 is 10 MHz, then what is the time required for one machine cycle ?

Soln. : Clock frequency $= \frac{1}{12} \times$ oscillator frequency

$\therefore$ Clock frequency $= \frac{10}{12}$ **= 0.833 MHz.**

$\therefore$ Time required for one machine cycle $= \frac{1}{0.833 \text{ MHz}}$

$= 1.2\ \mu s.$

$\therefore$ **One machine cycle = 1.2 μs.**

1.18.6 Time for Execution of an Instruction

Instruction cycle helps in calculating the time required for executing an instruction e.g. : a two cycle instruction will need 2 μs. It is because one-cycle frequency is $\left(\frac{1}{12}\right) f_{osc}$, where f_{osc} is the XTAL oscillation frequency. (Assuming crystal frequency 12 MHz)

Adding the total number of instruction cycles in a program that will take on execution gives the total time.

Syllabus Topic : Interrupt Structure

1.19 Interrupt Structure

- Whenever the microcontroller is executing a program and if a user wants service to an I/O device then an external asynchronous input would inform the microcontroller that it should complete the execution of current instruction and then fetch a new routine that will service the requesting I/O device. Once, the I/O device is serviced, the microcontroller resumes operation from the point whenever it had stopped. The external asynchronous input applied to the microcontroller is termed as an **Interrupt**.
- In this section we will study the interrupt structure enabling and disabling interrupts, programming the interrupts and interrupt priority.

1.19.1 Interrupt Service Routine

- Each interrupt requires an **interrupt handler** or an **Interrupt Service Routine (ISR).**
- Whenever an interrupt is invoked, the microcontroller executes an interrupt service routine. Each interrupt has a fixed location in the memory that holds the address of ISR.
- The group of memory locations kept aside to hold the addresses of ISRs is called as **interrupt vector table.**

1.19.2 Steps in Executing an Interrupt

Q. Explain the steps in executing an interrupt.

Whenever an interrupt is invoked, the microcontroller performs the following steps :

Step I : The microcontroller executes the current instruction and saves the address of the next instruction on the stack.

Step II : The microcontroller saves the current status of all the interrupts internally.

Step III : The microcontroller jumps to a fixed location in the memory referred to as the interrupt vector table. The interrupt vector table holds the address of the interrupt service routine (ISR).

Step IV : The microcontroller jumps to the address of the ISR from interrupt vector table and jumps to it. The microcontroller executes the ISR till it reaches the last instruction of the subroutine i.e. RETI.

Step V : On receiving the RETI instruction, the microcontroller returns to the main program where it was interrupted. The microcontroller fetches the address of program counter from the stack. The microcontroller then executes the main program from that address.

Note : Step V is the critical role of the stack. Hence, while modifying the contents of the stack in the ISR, the number of pushes and pops must be equal.

1.19.3 8051 Interrupt Structure

SPPU - May 13, Aug. 14, Dec. 15, May 16

University Questions

Q. Explain the interrupt structure of 8051. **(May 2013, Aug. 2014 (In Sem.), 2 Marks)**

Q. Draw and explain the interrupt structure of 8051 in detail. **(Dec. 2015, May 2016, 6 Marks)**

- The microcontroller 8051 supports **five interrupts**. These interrupts are automatically generated by internal operations and two interrupts are generated by the external signals provided.
- Although there are five interrupts, the manufactures data sheets indicate that there are six interrupts. This is because they include reset.
- The **six interrupts** are :

(i) Reset
(ii) Timer 0 (TF0) and timer 1 (TF1) interrupt
(iii) External hardware interrupts, $\overline{INT0}$ and $\overline{INT1}$
(iv) Serial communication interrupts RI and TI.

- All the interrupt functions are under the program control. **The programmer is able to change the control bits in the Interrupt Enable register (IE), the interrupt priority register (IP), and the Timer control register (TCON). By setting or clearing the bits in these registers the program can block any or all of the interrupts.**

Fig. 1.19.1 shows the 8051 Interrupt structure and control system.

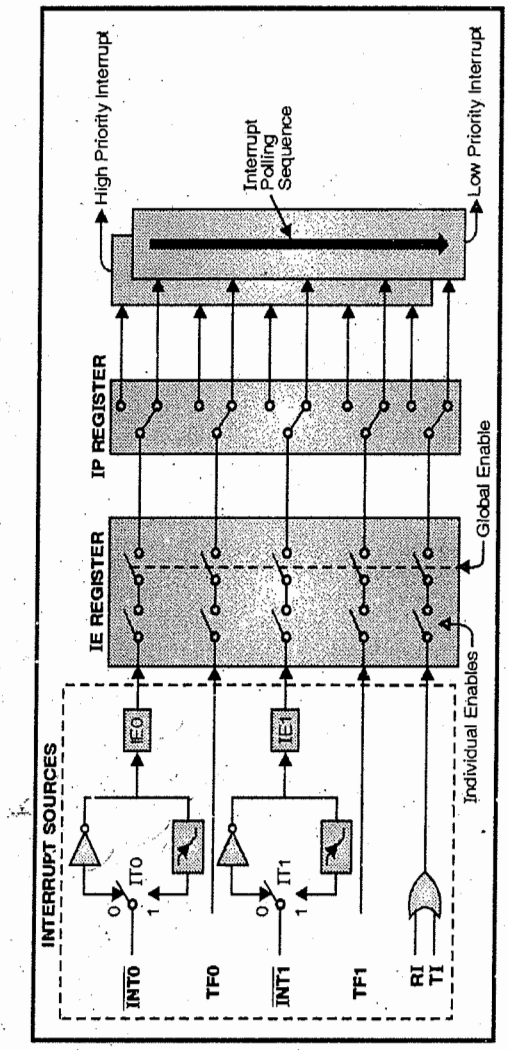


Fig. 1.19.1 : 8051 Interrupt structure and control system

1.19.3.1 Timer Flag Interrupts

When the timer / counter overflows, the corresponding timer flag TF0 or TF1 is set to 1. The flag is cleared to 0 when the interrupt generates program call to the timer subroutine in the memory.

1.19.3.2 Serial Port Interrupts

- The **serial port interrupt is generated because of RI or TI.** These bits are logically ORed, to provide a single interrupt to the processor.

- The TI bit in the SCON register is set when a data byte is transmitted and the RI bit in the SCON register is set whenever a data byte is received.
- The serial port Interrupts RI and TI are not cleared like the Timer interrupt when the interrupt generates program call. So, the program which deals with serial communication must reset or clear the RI or TI bits to 0 to enable next data communication operation.

1.19.3.3 External Interrupts

The two interrupts that are generated by external circuits are $\overline{INT0}$ and $\overline{INT1}$. The inputs on the pins of these interrupts sets the interrupt flags IE0 and IE1 in the TCON register. These interrupts may be edge triggered or they may be level triggered.

1.19.4 Interrupt Vector Addresses

SPPU - Aug. 14

University Question

Q. List the vector addresses. **(Aug. 2014(In Sem), 1 Mark)**

Table 1.19.1 depicts the vector location for different interrupt sources. e.g. If TF0 interrupt is generated by Timer 0, then the microcontroller branches to vector location 000BH. The program execution will continue from that location till a return instruction is encountered.

Table 1.19.1 : Interrupt vector table

Source	Interrupt vector address (ROM location)
Reset	0000H
External hardware interrupt 0 ($\overline{INT0}$)	0003 H
Timer 0 interrupt (TF0)	000BH
External hardware interrupt 1 ($\overline{INT1}$)	0013H
Timer 1 interrupt (TF1)	001BH
Serial communication interrupt (RI and TI)	0023 H

- A total of 8 bytes is set aside for each interrupt e.g. a total of 8 bytes from location 000BH to 0012H is set aside for Timer 0 interrupt (TF0).
- If the interrupt service routine is short enough to fit in the memory space allocated to it, it is placed in the interrupt vector table, otherwise an LJMP instruction is placed in the vector table to point to the address of the interrupt service routine. The remaining bytes allocated to the interrupt are unused.

1.19.5 Enabling and Disabling an Interrupt with the IE Register

SPPU - Dec. 13, May 15, Oct. 16

University Questions

Q. With the help of IE register explain interrupt structure of 8051. **(Dec. 2013, 4 Marks)**

Q. Explain Interrupt enable register ? **(May 2015, 3 Marks)**

Q. Explain IE register of 8051. **(Oct. 2016(In Sem.), 2 Marks)**

- When the 8051 microcontroller is reset, all the interrupts are disabled. Even if the interrupts are activated they will not be responded by the microcontroller upon reset.
- The interrupts must be enabled/activated by the software, so that the microcontroller can service the interrupts. The interrupts can be enabled or disabled by modifying the bit in the **IE register**. Fig. 1.19.2 shows the IE (Interrupt Enable Register).

(MSB)							(LSB)
EA	–		ES	ET1	EX1	ET0	EX0

Enable Bit = 1 enables the interrupt. Enable Bit = 0 disables it.

Symbol	Position	Function
EA	IE.7	disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
–	IE.6	Reserved
–	IE.5	Reserved
ES	IE.4	Serial Port interrupt enable bit.
ET1	IE.3	Timer 1 interrupt 1 enable bit.
EX1	IE.2	External interrupt 1 enable bit.
ET0	IE.1	Timer 0 interrupt enable bit.
EX0	IE.0	External interrupt 0 enable bit.

Fig. 1.19.2 : Interrupt Enable register (IE)

- IE is a bit addressable register. It contains a global disable bit EA that disables all the interrupts at once.

1.19.5.1 Steps in Enabling an Interrupt

- To enable an interrupt the following steps are to be considered :

Step I : The bit D_7 of the IE register must be set.

Step II : Set the corresponding bit in the IE register for enabling a particular interrupt. If the EA bit is not set, then no interrupt will be responded even if the bit in the IE register is set.

Ex. 1.19.1

Program the 8051 to

(i) Enable timer 0 interrupt, serial interrupt and external hardware interrupt (EX1)

(ii) Disable all the interrupts using a single instruction

(iii) Disable the Timer 1 interrupt.

Soln. :

(i) For enabling timer 0 interrupt, serial interrupt and external hardware interrupt (EX1) the IE is,

1	0	0	1	0	1	1	0

= 96 H

The instruction

MOV IE, #96H or

MOV IE, # '10010110B' will enable the interrupts.

(ii) If the EA bit is cleared, then all the interrupts are disabled. CLR IE.7 instruction will disable all the instructions.

(iii) We need to disable bit 3 for disabling the Timer 1 interrupt CLR IE.3 will disable Timer 1 interrupt.

1.19.6 Interrupt Priority in the 8051/52

SPPU - Aug. 14

University Question

Q. How priority can be changed ?

(Aug. 2014(In Sem.), 1 Mark)

- Upon power up, the priorities are assigned according to Table 1.19.2.

Table 1.19.2 : Interrupt priorities

No.	Interrupt Source	Name	Priority level
1	External interrupt 0	$\overline{INT0}$	Highest
2	Timer interrupt 0	TF0	↓
3	External Interrupt 1	$\overline{INT1}$	
4	Timer Interrupt 1	TF1	
5	Serial communication	RI + TI	
6	Timer 2 (8052 only)	TF2	lowest

- These priorities are assigned to the registers by default, but if the programmer wishes to change the priority, then the priority can be changed by the IP register.
- If two requests of the same priority level are received at the same time, then an internal polling sequence determines which request is serviced.
- If two requests of different priority are received at the same time, then the request having higher priority level will be serviced first.
- Thus, within each priority level there is a second priority structure determined by the polling sequence.

1.19.6.1 Setting Interrupt Priority with the IP Register

SPPU - May 15, Oct. 16

University Questions

Q. Explain interrupt priority register ?

(May 2015, 3 Marks)

Q. Explain IP register of 8051.

(Oct. 2016(In Sem.), 2 Marks)

- The interrupt priorities in Table 1.19.2 can be modified by programming a register called IP (Interrupt Priority) register.
- Fig. 1.19.3 shows the IP register.
- Upon reset the IP register contains all 0's. To assign a higher priority to any of the interrupts we make the corresponding bit in the IP register high.

(MSB)							(LSB)
–	–	–	PS	PT1	PX1	PT0	PX0
Priority bit = 1 assigns high priority, Priority Bit = 0 assigns low priority							

Symbol	Position	Function
–	IP.7	Reserved
–	IP.6	Reserved
–	IP.5	Reserved
PS	IP.4	Serial Port interrupt priority bit.
PT1	IP.3	Timer 1 interrupt priority bit.
PX1	IP.2	External interrupt 1 priority bit.
PT0	IP.1	Timer 0 interrupt priority bit.
PX0	IP.0	External interrupt 0 priority bit.

Fig. 1.19.3 : Interrupt priority register (IP)

Ex. 1.19.2

Program IP register to assign the highest priority to $\overline{INT1}$.

Soln. :

IP	0	0	0	0	0	1	0	0

External interrupt 1 priority bit

(i) SETB IP.2 or

(ii) MOV IP, #04H instructions will assign highest priority to $\overline{INT1}$

Ex. 1.19.3

If IP register is loaded with 0CH, write down the sequence in which interrupts are serviced.

Soln. :

If IP = 0CH, it sets the external interrupt $\overline{INT1}$ and Timer 1 (TF1) interrupt at a high priority level as compared to other interrupts. The sequence in which the interrupts are serviced is as follows :

	Interrupt source	Name	Priority level
Highest priority	External interrupt 1	($\overline{INT1}$)	↓
	Timer interrupt 1	(TF1)	
	External interrupt 0	($\overline{INT0}$)	
	Timer interrupt 0	(TF0)	
Lowest priority	Serial communication interrupt	(RI + TI)	

1.19.6.2 Nested Interrupts

- When the 8051 is executing an ISR inorder to provide service to an interrupt and if another interrupt is invoked, then in such a case if the newly invoked interrupt is a high priority interrupt then only it can interrupt the previously serviced low priority interrupt. It is an **interrupt inside interrupt** or **nested interrupt**.
- A low priority interrupt can be interrupted by a high priority interrupt, but not by any other low priority interrupt.
- All the 8051 interrupts are latched and kept internally. This allows low priority interrupts to be serviced after the high priority interrupts are being serviced.

1.19.7 Triggering the Interrupt by Software

- It is possible to trigger the interrupts by software. It can be achieved by simple instructions that set the interrupts and cause the microcontroller 8051 to jump to the interrupt vector table.
- e.g. : If the IE bit for Timer 0 is set, then the instruction SETB TF0 will interrupt the 8051 and force it to jump to the interrupt vector table to provide service to the interrupt.
- Thus, it is not required to wait for the Timer 0 to rollover to have an interrupt.
- We are using an instruction to raise the interrupt. This is called as **software triggering of the interrupt**.

1.19.8 Why We Cannot use RET Instead of RETI as the Last Instruction of on ISR ?

RET and RETI both these instructions return control to the main program from the subroutine/ISR. However, the RETI instruction clears the interrupt service flag after servicing the interrupt. This allows the 8051 microcontroller to accept a new interrupt.

If we use RET as the last instruction of an ISR, then after the initial interrupt all other interrupts will be blocked till the initial interrupt is serviced. Hence, RETI must be the last instruction of an ISR.

Syllabus Topic : Timers and its Modes

1.20 Timers and its Modes

SPPU - Dec. 16

University Question

Q. Explain counter operation in 8051 microcontroller. **(Dec. 2016, 6 Marks)**

Function and use : The **microcontroller 8051 has two 16 bit Timer / Counter registers namely Timer 0 (T0) and Timer 1 (T1).** Both these registers can be configured independently to operate as timer or an event counter. They can be used as timers to generate a time delay or as counters to count events happening outside the microcontroller.

- When used as a **"Timer"** the register is programmed to count the internal clock pulse. The internal clock pulses are generated from a constant clock generator, the count loaded in the register gives constant time. The register is incremented every machine cycle. One machine cycle consists of 12 oscillator periods and so the counting rate is 1/12 of the oscillator frequency.
- When used as a **"Counter"**, the microcontroller is programmed to count external pulses. The register is incremented in response to a high to low transition (↘) of the corresponding external input pin, T0 and T1. The external input does not have a constant frequency and hence it is not used for timing reference.

Maximum count rate : Hence, in order to recognize the high-low transition the microcontroller requires two machine cycles i.e. 24 oscillator periods. The maximum count rate is 1/24 of the oscillator frequency.

- There are no restrictions on the duty cycle of the external input signal, but it should be held high atleast for one machine cycle, to ensure that the input is sampled atleast once before it changes.
- The timer mode can also be used for **pulse width measurement**. When the gate bit is kept '1', the timer runs until the INTx pin is high. Hence the timer is counting the number of internal clock pulses for which the pulse on INTx pin is at logic '1'. For e.g. if the timer counts from 1 to 10 and the crystal is of 12 MHz (i.e. one machine cycle is 1μsec), it indicates the pulse on INTx pin was at logic '1' for 10 μsecs.

1.20.1 Timer 0 and Timer 1 Registers

Size of registers :

The counter / timer registers are divided into 8 bit registers called the timer low (TL0 and TL1) and timer high (TH0 and TH1). TL0 and TH0 together form the 16 bit Timer 0 and TL1 and TH1 forms the 16 bit Timer 1.

- Fig. 1.20.1 shows the timer 0 and timer 1 registers. These registers can be accessed like other registers A, B, R0, R1 etc.

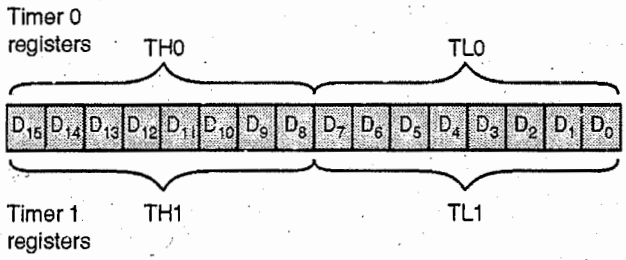


Fig. 1.20.1 : Timer 0 and Timer 1 registers

1.20.2 TMOD and TCON Registers

SPPU - Dec. 12, Aug. 15

University Questions

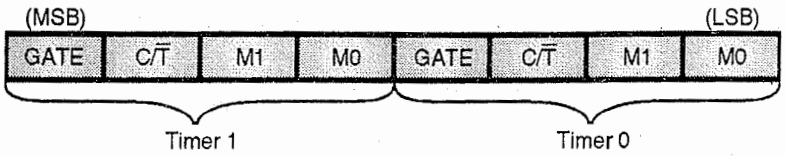
Q. Explain the format of TMOD register of 8051 μc. in detail. **(Dec. 2012, 9 Marks)**

Q. Explain the use of TMOD register of 8051. **(Aug. 2015 (In Sem.), 2 Marks)**

- **The counter action is controlled by the bits in the SFRs timer mode control register (TMOD) and the timer / counter control register (TCON) and some instructions.**

Size : They are 8 bit registers.

- Fig. 1.20.2 shows TMOD register and Fig. 1.20.3 shows the TCON register.



GATE — Gating control. When set Timer/Counter "x" is enabled only while "$\overline{INTx}$" pin is high and "TRx" control pin is set. When cleared, Timer "x" is enabled whenever "TRx" control bit is set. It is used for Pulse width measurement (PWM) of the pulse on $\overline{INTx}$ pin.

C/$\overline{T}$ — Timer or Counter selector. Cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin)

M1	M0	Operating Mode
0	0	8-bit Timer/Counter "THx" with "TLx" as 5-bit prescalar
0	1	16-bit Timer/Counter "THx" and "TLx" are cascaded, there is no prescaler
1	0	8-bit auto-reload Timer/Counter. "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
1	1	(Timer 1) Timer/Counter 1 stopped

GATE is used to program timers in PWM mode.

C/$\overline{T}$ will select either counter operation or timer operation.

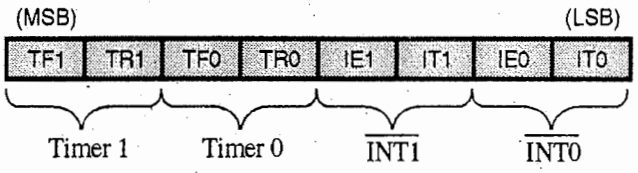
C/$\overline{T}$ = 0 → Timer; C/$\overline{T}$ = 1 → Counter.
Mode bits are going to select different modes of operation of the timer.

m(19.21)**Fig. 1.20.2 : Timer / Counter mode control register (TMOD)**

Use :

Both the timers T0 and T1 use the SFR called TMOD register to set various timer operation modes.

- TMOD is an 8 bit register. It uses the lower 4 bits for Timer 0 and upper 4 bits for Timer 1. In each case, the lower 2 bits are used to set the timer mode and upper 2 bits specify the operation.



Symbol	Position	Name & Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on timer Counter overflow. Cleared by hardware when processor vectors to interrupt routine
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on Timer counter overflow. Cleared by hardware when processor vectors to interrupt routine.
TR0	TCON.4	Timer 0 Run control bit. Set/ cleared by software to turn Timer/Counter on/off.

Symbol	Position	Name & Significance
IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
IT0	TCON.0	Interrupt 0 Type control bit Set / cleared by software to specify falling edge/low level triggered external interrupts.

m(19.22)**Fig. 1.20.3 : TCON Timer / Counter control register**

Function : The SFR TCON (Timer/Counter Control) register controls the timer/counter operations. The lower four inputs serve to the interrupt functions, but the upper four bits are for timer operations.

1.20.3 Clock Source for Timer

- If the $C/\overline{T} = 0$, then the crystal frequency attached to 8051 is the source of clock for timer. i.e. the size of the crystal frequency attached to 8051 decides the speed at which the timer of 8051 ticks.

The timer frequency is always $\frac{1}{12}^{th}$ of the frequency of the crystal attached to 8051.

1.20.4 How are Timers 0 and 1 Started and Stopped ?

- The 8051 timers can be started and stopped by hardware and software controls.
- The start and stop of the timer are controlled by the software method by the TR (timer start) bits TR0 and TR1 in the TCON register.
- It can be achieved by the instructions ***"SETB TR0"*** and ***"CLR TR0"*** for Timer 0 and ***"SETB TR1"*** and ***"CLR TR1"*** for Timer 1. The SETB instruction starts the timer and CLR instruction stops the timer. These instructions start and stop the timer till the GATE = 0 in the TMOD register.
- The hardware method of starting and stopping the timer by an external source can be obtained by making GATE = 1 in the TMOD register.

Syllabus Topic : Timer Modes of Operation

1.20.5 Timer Modes of Operation

SPPU - May 13, Dec. 13

University Questions

Q. Explain different timer/counter modes of 8051. **(May 2013, 8 Marks)**

Q. Explain in details the timer modes of 8051. **(Dec. 2013, 8 Marks)**

Depending on the mode control bits M1 and M0 in the SFR Timer / Counter mode control (TMOD) register, the timer can operate in any one of the four modes :

(i)	Mode 0	(ii)	Mode 1
(iii)	Mode 2	(iv)	Mode 3.

1.20.5.1 Mode 0

SPPU - May 12

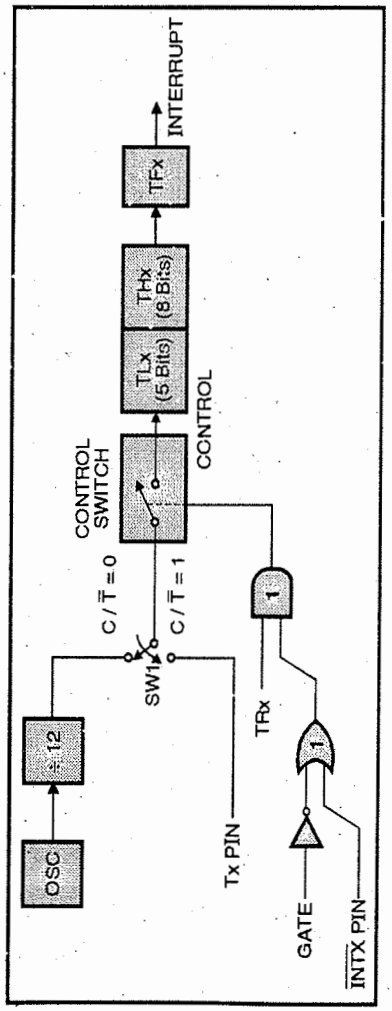
University Question

Q. Explain mode 0 of timer in 8051.

(May 2012, 4 Marks)

Size : In this mode the timer operates as a 13 bit register (TL – 5 bits and TH – 8 bits). The 5 bits in the TL sets the divide by 32 prescalar to the TH as an 8 bit counter.

- Fig. 1.20.4 shows the Timer / Counter in mode 0.



m(19.23) **Fig. 1.20.4 : Timer / Counter mode 0 : 13 bit counter**

- The Timer is enabled when TR = 1, GATE = 0 and $\overline{INTX}$ = 1. If the GATE = 1, then the timer is controlled by external input $\overline{INT}$, to allow pulse width measurement. TR is control bit in the TCON register.

Interrupt

- If the user desires some count in the register and gives command by TRx bit the timer / counter will start. If the timer count exceeds 13 bit i.e. 1FFFH the next count will be 0000H.
- It causes the microcontroller to generate an interrupt in order to inform the programmer whether one cycle is over.
- To indicate it, Timer overflow bit (TF1 and TF0) will be set.
- If the timer interrupt is enabled using the Interrupt Enable register (IE), the controller will vector to ISR routine.

1.20.5.2 Mode 1

SPPU - May 12, Aug. 15

University Question

Q. Explain mode 1 of timer in 8051.

(May 2012, 4 Marks, Aug. 2015 (In Sem.), 2 Marks)

Mode 1 of the Timer / Counter is same as mode 0. The only difference is that in mode 0 the timer / counter was 13 bit timer / counter, but in mode **1 it is a 16 bit timer / counter**. It values counts from 0000H to FFFFH to be loaded into the timer registers.

Mode 1 is 16 bit timer. Hence the biggest count that can be stored in the timer registers is FFFFH. Mode 2 is 8 bit Timer. Hence, the maximum count that can be stored in Timer register is FFH.

1.20.5.3 Mode 2

SPPU - Dec. 14, Aug. 15

University Question

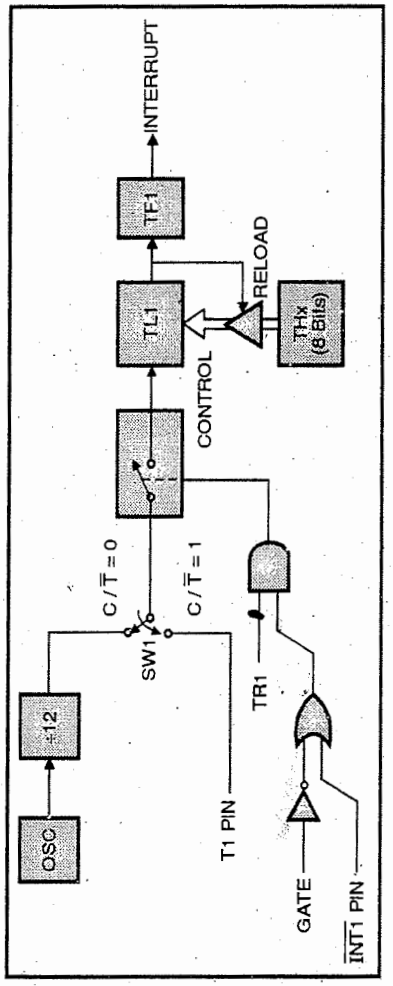
Q. Explain the timer mode 2 of 8051.

(Dec. 2014, 6 Marks, Aug. 2015 (In Sem.), 2 Marks)

- In this mode the Timer / Counter register is configured as an 8 bit counter (TL) with auto reload facility. Fig. 1.20.5 shows Timer / Counter 1 in mode 2. It allows values 00H-FFH to be loaded into TH register.

- TL1 acts as the basic timer / counter. In autoreload mode TH is loaded. When the timer starts working, it keeps on incrementing and it will overflow. This will set the timer interrupt flag TF1 and it also reloads the contents of TH1 to TL1. The reloading operation will not alter the contents of TH1. The operation is same for Timer 0.

Applications : Setting baud rate in serial communication, generating square wave of different frequencies, display count.

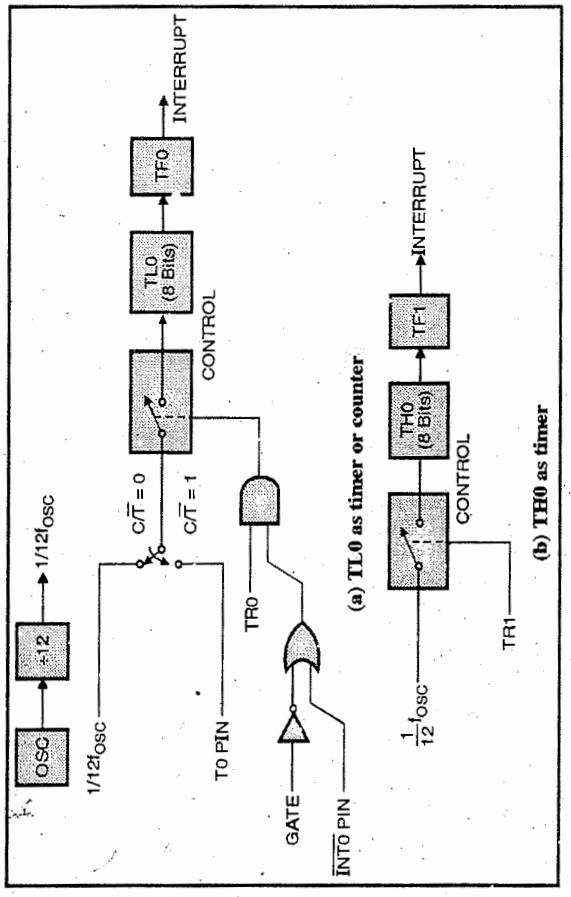


m(19.24)**Fig. 1.20.5 : Timer / Counter 1 mode 2 – 8 bit auto reload**

1.20.5.4 Mode 3

- In mode 3 Timer 1 is not used. Timer 1 therefore simply holds its count. The timer 0 registers TL0 and TH0 are configured as two separate 8 bit counters. Fig. 1.20.6 shows Timer / Counter 0 mode 3.

- The TL0 register uses the Timer 0 control bits $C/\overline{T}$, GATE, TR0, $\overline{INT0}$ and TF0. TH0 counts the machine cycles. It takes over the use of TR1 and TF1 from Timer 1.



m(19.25)**Fig. 1.20.6 : Timer / Counter 0 mode 3**

- When the Timer 0 is in mode 3, the Timer 1 may be used in modes 0, 1, and 2. In this case no interrupts will be generated because Timer 0 is using the TF1 flag.

Use : When the Timer 0 is in mode 3, Timer 1 can be used **as a baud rate generator for serial port.**

Table 1.20.1 : Summary of operating modes of Timers

Mode	Description
Mode 0	13 bit timer (TL - 5 bits and TH - 8 bits) counter overflow indicated by TI flag.
Mode 1	16 bit timer (TL - 8 bits and TH - 8 bits)
Mode 2	Reload mode 8 bit counter (TL - 8 bit) overflow from TL not only sets, but also reloads TL with the contents of TH.
Mode 3	Configures TL0 and TH0 as two separate counters.

1.20.6 Counter and Pulse Width Measurement (PWM)

- 8051's Timer/Counter can also be used as a counter when $C/\overline{T}$ = '1'. We will see this feature of 8051 in this section. This feature can also be used to measure the frequency by measuring the number of pulses in 1 second.
- Another feature of 8051's timer/counter is that it can be used to measure the width of a pulse. This can be implemented by programming the timer/counter in timer mode. The pulse to be measured is to be applied on the INT0 or INT1 pins for timer 0 and timer1 respectively. The GATE bit of the TMOD register is to be made '0'. The count in the timer registers indicate the number of machine cycles for which the pulse was at logic '1'. This when multiplied with the time period of 1 machine cycle gives the time period for which the pulse was at logic '1'.

Syllabus Topic : Programming the 8051 Timer / Counter in Mode 0 and 1

1.20.7 Programming the 8051 Timer / Counter in Mode 0 and 1

Program 1.20.1

Indicate the mode in which the timer will be operated after the execution of the following instructions :

(i) MOV TMOD, #20H (ii) MOV TMOD, 02H

Soln. :

Initially we will convert the hex values to binary.

(i) TMOD = 0010 0000. Hence mode 2 of timer 1 is selected.

(ii) TMOD = 0000 0010. Hence mode 2 of timer 0 is selected.

Program 1.20.2

Estimate the timer's clock frequency and its period, for the 8051 based system that has clock frequency of 16 MHz.

Soln. :

We know that the frequency for the timer is $\frac{1^{th}}{12}$ of the crystal frequency.

$$\therefore \text{ Clock frequency } = \frac{1}{12} \times 16 = 1.333 \text{ MHz}$$

$$\therefore \text{ Clock period } = \frac{1}{T} = \frac{1}{1.333} \text{ MHz}$$

$$= \mathbf{0.75\ \mu s} \qquad \textbf{...Ans.}$$

Program 1.20.3

Describe various modes of Timer in 8051. Find out Hex number to be loaded in TH0, to produce delay of 4.096 msec in mode '0' operation. Assume clock frequency of 12 MHz.

Soln. : f_{OSC} = 12 MHz

Hence the counter will count up every 1 μs.

$$\frac{4.096 \text{ ms}}{1\mu s} = 4096 \text{ clocks}$$

To achieve this need to load into TL and TH the value

$$65536 - 4096 = (61440)_{10} = \text{F000H.}$$

Hence **TH0 = F0H, TL0 = 00H** **...Ans.**

Syllabus Topic : Programming the Timer in Mode 1

1.20.8 Programming the Timer in Mode 1

Following are the steps to program the timer in mode 1.

Step I : Load the TMOD register.
Step II : Load the TL and TH registers with initial count values.
Step III : Start timer.
Step IV : Observe the timer flag (TF). Exit from the loop when the TF flag is set.
Step V : Stop timer.
Step VI : Clear the timer for next round.
Step VII : Go back to step II.

Program 1.20.4

Assuming that XTAL = 11.0592 MHz, write a program to generate a square wave of 2 KHz frequency on pin 1.5.

Soln. : The period of square wave is $\frac{1}{2 \text{ KHz}}$ = 500 μs

Let us assume that duty cycle of square wave is 50%. Hence, the square wave will be high for 250 μs and it will be low for 250 μs.

XTAL = 11.0592 MHz i.e. the counter will count up every 1.085 μs.

$\frac{250\ \mu s}{1.085\ \mu s}$ = 230, 1.085 μs intervals will make a 500 μs / 2 KHz pulse.

The values that should be loaded into TH and TL registers are

$$65536 - 230 = (65306)_{10} = \text{FF1AH}$$

$$\therefore \text{ TL} = \text{1AH and TH} = \text{FFH}$$

Program :

Label	Instruction	Comments
	MOV TMOD, #10H	Load TMOD register in timer 1, mode 1
L1 :	MOV TL1, #1AH	Load TL
	MOV TH1, #0FFH	Load TH
	SETB TR1	Start timer 1
L2 :	JNB TF1, L2	Remain until timer rolls over
	CLR TR1	Stop timer 1
	CPL P1.5	
	CLR TF1	Clear timer 1 flag
	SJMP L1	Reload timer as mode 1 is not auto reload

Syllabus Topic : Delay Using Timer

Program 1.20.5 SPPU - Oct. 2016, 5 Marks

Describe the values to be loaded in TH, TL and TMOD on register for delay calculations of 1 msec using timer 1. (Assume oscillatory frequency 10 MHz).

Soln. :

$$\text{Crystal frequency} = 10 \text{ MHz}$$

$$\therefore \text{ Time for 1 machine cycle} = \frac{12}{10 \text{ MHz}} = 1.2\ \mu \text{ sec.}$$

$$\text{Delay period} = 1 \text{ msec}$$

The count for mode 1 can be calculated as,

$$(\text{Count} - 1) = \text{Maximum value} - \text{required delay} \times \frac{\text{Crystal frequency}}{12}$$

$$(\text{Count} - 1) = \text{FFFF} - 1 \times 10^{-3} \times \frac{10 \times 10^6}{12}$$

$$(\text{Count} - 1) = 65535 - 833$$

$$\text{Count} = (64703)_{10} = \text{FCBFH}$$

$$\text{TH1} = \text{FCH}$$

$$\text{TL1} = \text{BFH}$$

$$\text{TMOD} = 10 \text{ H (Timer 1, mode 1)}$$

Program :

Label	Instruction	Comments
	MOV TMOD, #10H	Timer 1, mode 1
L1:	MOV TL1, #BFH	TL1 = BFH
	MOV TH1, # FCH	Load TH1 = FCH
	SETB TR1	Start Timer 1
L2:	JNB TF1, L2	Wait till timer rolls over
	CLR TR1	Stop Timer1
	CLR TF1	Clear Timer 1 flag
	SJMP L1	Reload Timer 1

Program 1.20.6 SPPU - Aug. 2014, 6 Marks

Calculate the hexadecimal values to be loaded in TH, TL and TMOD register for delay calculations of 1 msec using Timer 1 in mode 1. (Assume input frequency = 12 MHz).

Soln. :

$$\text{Crystal frequency} = 12 \text{ MHz}$$

$$\text{Delay} = 1 \text{ msec}$$

$$\text{Count} - 1 = \text{Maximum value} - \text{delay} \times \frac{\text{Crystal frequency}}{12}$$

$$\text{Count} - 1 = \text{FFFF} - 1 \times 10^{-3} \times \frac{12 \times 10^6}{12}$$

$$\text{Count} - 1 = 65535 - 1000$$

$$\text{Count} = (64536)_{10} = \text{FC18H}$$

$$\therefore \text{ TH1} = \text{FCH}$$

$$\text{TL1} = 18 \text{ H}$$

$$\text{TMOD} = 10 \text{ H (Timer 1, mode 1)}$$

Program

Label	Instruction	Comments
	MOV TMOD, #10H	Load TMOD register in timer 1, mode 1
L1:	MOV TL1, #18H	Load TL1 = 18H
	MOV TH1, #FCH	Load TH1 = FCH
	SETB TR1	Start Timer 1
L2:	JNB TF1, L2	Remain until timer rolls over
	CLR TR1	Stop timer 1
	CLR TF1	Clear timer 1 flag
	SJMP L1	Reload Timer 1 in mode 1

1.20.9 Mode 0 Programming

The programming of timer in mode 0 is same as the programming of mode 1. In mode 0 the timer is a 13 bit timer, so the maximum possible count ranges in values between 0000H to 1FFFH. Hence, when timer reaches to 1FFFH, it rolls over to 0000H and TF is raised.

1.20.10 Programming the Timer in Mode 2

Following are the steps to program the timer in mode 2.

Step I : Load the TMOD register.

Step II : Load the TH register with initial count.

Step III : Start the timer.

Step IV : Observe the TF flag. Exit from loop when the TF flag is set.

Step V : Clear the TF flag.

Step VI : Goto step IV, as mode 2 is auto reload.

Program 1.20.7

Compute the frequency of the square wave generated on P1.5 in the following program.

Label	Instruction
	MOV TMOD, #20H
	MOV TH1, #4H
	SET TR1
L1 :	JNB TF1, L1
BACK :	CPL P1.5
	CLR TF1
	SJMP BACK

Soln. :

Since TH1 is 8 bit in mode 2

$\therefore\ 256 - 4 = 252$ cycles

$252 \times 1.085\ \mu s$ the square wave will remain high and for $273.33\ \mu s$ the square wave will remain low.

$$\therefore \text{Period } T = 2 \times 272.33 = 544.67\ \mu s$$

$$\text{Frequency} = \frac{1}{T} = 1.8359 \text{ KHz.}$$

Program 1.20.8

Write an assembly language program for 8051 such that LED connected to port P1.0 will flash at a rate 0.5 sec rate when line P2.3 goes high use timer 0 for generating delay.

Soln. :

Let XTAL = 12 MHz.

$$\therefore \text{Timer clock frequency} = \frac{12\text{MHz}}{12} = 1 \text{ MHz}$$

$$\therefore T = 1\ \mu s.$$

Hence we can get a maximum delay of $65536 \times 1\ \mu s = 65.536$ ms.

To get a delay of 0.5 sec we will program timer 0 to give a delay of 50 ms. We will execute the delay for 10 times so that we will get a delay of 0.5 sec.

To obtain a delay of 50 ms, the values that should be loaded into TH and TL registers are :

$$(65536 - 50000)_{10} = (3CB0)_H$$

$$\therefore\ TL = B0H \text{ and } TH = 3CH$$

Program :

Delay routine

Label	Instruction	Comments
DELAY :	MOV R0, #0AH	Initialize counter to 10
L1 :	MOV TL0, #0BH	Load TL
	MOV TH0, #03CH	Load TH
	SETB TR0	Start timer 0
L2 :	JNB TF0, L2	Remain until timer rolls over
	CLR TR0	Stop timer 0
	CLR TF0	Clear timer 0 flag
	DJNZ R0, L1	Decrement R0 and if R0 ≠ 0 repeat
	RET	

Main program :

Label	Instruction	Comment
	MOV TMOD, #01	Timer 0, Mode 1
	MOV P2, #0FFH	Let P2 = input port
AL1 :	JB P2.3, AL1	Continue till P2.3 = 1
HERE :	CPL P1.0	Toggle P1.0
	ACALL DELAY	Call Delay
	SJMP HERE	Repeat

Program 1.20.9

Write an assembly program for counting the pulses on P3.5 pin (T1) and display the hex count on P0 (LSB) P1(MSB).

Soln. :

Program :

Label	Instruction
	org 0000H
	LJMP main
	org 1000H
main :	MOV P0, #00H
	MOV P1, #00H
	MOV TMOD, #50H
	MOV TL0, #00H
	MOV TH0, #00H
	SETB P3.5
	SETB TR0
wait :	JNB TF0, wait
	CLR TF1
	MOV P0, TL0
	MOV P1, TH1
	SJMP wait

Program 1.20.10

Write an assembly program to generate a square wave of 10 KHz with timer 0 on Port pin.

Soln. :

Frequency = 10 KHz.

$$\therefore \text{ 1 clock pulse } = \frac{1}{10 \text{ KHz}} = 100 \ \mu\text{sec.}$$

$\therefore$ 50 μsec is 'ON' time and 50 μsec is 'OFF' time

Hence, a delay of 50 μsec required

$$\therefore \text{ count } = \frac{50 \ \mu\text{sec}}{1\mu\text{sec}} \text{ (Assuming 12 MHz crystal)}$$

$$= 50.$$

Program :

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	by pass interrupt vector table
	ORG 000BH	Interrupt vector for Timer 0
	CPL P0.0	Complement P0.0 bit
	RETI	Return from ISR
	ORG 1000H	Start main program after interrupt vector table
main :	MOV TMOD, #02	Initialize timer 0 is mode 2
	MOV TH0, #50	Load timer count
	MOV TL0, #50	
	MOV IE, #82H	Enable Timer 0 interrupt
	SETB TR0	Start Timer 0
here :	SJMP here	Wait it timer rolls off

Program 1.20.11

Write a program to generate frequencies of 2 KHz and 10 KHz on pins P0.0 and P0.1 respectively. Assume crystal frequency = 12MHz.

Soln. : Timer clock frequency $= \dfrac{12 \times 10^6}{12} = 1$ MHz

For 2 KHz

On period is 0.25 msec and off period is 0.25 msec.

$$\therefore \text{ TH0 } = 256 - 0.25 \times 10^{-3} \times 1 \times 10^6$$

$$\text{TH0 } = (6)_{10} = 6\text{H}$$

For 10 KHz

On period is 0.05 msec and off period is 0.05 msec.

$$\therefore \text{ TH1 } = 256 - 0.05 \times 10^{-3} \times 1 \times 10^6$$

$$\text{TH1 } = (206)_{10} = \text{CEH}$$

Program :

Label	Instruction	Comments
	ORG 0000H	Avoid using interrupt vector table
	LJMP main	
	ORG 000BH	ISR for Timer 0 interrupt
	CPL P0.0	Complement bit P0.0
	RETI	Return from ISR
	ORG 001BH	ISR for Timer 1 interrupt complement bit return from ISR
	CPL P0.1	
	RETI	
main :	ORG 0030H	
	MOV TMOD, #22H	Initialize both timers in mode 2
	MOV IE, #8AH	Enable Timer 0 and Timer 1 interrupts
	MOV TH0, #06H	Count value for 2 KHz wave
	MOV TH0, #CEH	Count value for 10 KHz wave
	SETB TR0	Start Timer 0
	SETB TR1	Start Timer 1
here :	SJMP here	Wait till either timer rolls off
	END	

Program 1.20.12

Write an assembly program to switch 'on' or 'off ' a LED connected on P1.5 when external interrupt $\overline{\text{INT0}}$ is activated.

Soln. : Program :

Label	Instruction	Comments
	ORG 0000H	Bypass interrupt vector table
	LJMP MAIN	
	ORG 0003H	Interrupt vector for Interrupt 0
	SETB P1.5	Turn on LED
L1 :	MOV R0, #200	Wait for sometime
	DJNZ R0, L1	
	RETI	Return to main program
	ORG 0030H	
MAIN :	MOV IE, #81H	Enable INT0
L2:	SJMP L2	
	END	End program

Program 1.20.13

Write an assembly language program for 8051 to generate a delay of 1msec using 12MHz crystal.

Soln. :

Crystal frequency = 12 MHz.

$\therefore$ Time for 1 machine cycle $= \dfrac{12}{12\ \text{MHz}} = 1\ \mu\text{sec.}$

$\therefore$ Delay period = 1 m sec.

Hence a delay of 1 msec is required.

$\therefore$ We will require mode 2; and execute a delay of (lets say) 250 μsec four times.

Suppose we use timer 1, TMOD = 20 H

$$TH1 = TL1 = (256)_{10} - (250)_{10} = (6)_{10} = (6)_{16}$$

The ISR should just toggle a port pin of 8051.

Program :

Label	Instruction
Delay :	MOV TMOD, #20H
	MOV TH1, #06H
	MOV TL1, #06H
	SETB TR1
here :	JNB TF1, here
	RETI

Syllabus Topic : Delay Using Interrupt

Program 1.20.14

Write a program to generate a square wave of frequency 1KHz and 75% duty cycle at pin P1.0 using 8051 microcontroller. Assume microcontroller is operating at 6 MHz.

Soln. :

Crystal frequency = 6 MHz.

$\therefore$ Time for 1 machine cycle $= \dfrac{12}{6\ \text{MHz}} = 2\ \mu\text{sec.}$

$\therefore$ Square wave period = 1 msec. ($\because$ 1 KHz)

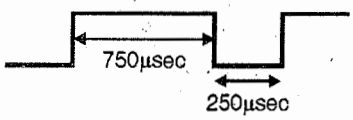


Fig. P. 1.20.14

Hence the delay required (Using timer 1 in Mode 1)

(i) For 'ON' period

$$\frac{750\ \mu\text{sec}}{2\ \mu\text{sec}} = 375$$

$\therefore$ Count = 65536 – 375 = $(65161)_{10} = (FE89)_{16}$

(ii) For 'OFF' period

$$\frac{250\ \mu\text{sec}}{2\ \mu\text{sec}} = 125$$

$\therefore$ Count = 65536 – 125 = $(65411)_{10}$

$= (FF83)_{16}$

Program :

Label	Instruction	Comments
	org 0000H	
	LJMP main	
	org 001BH	
	CPL P1.5	Toggle the output pin
	CLR TR1	
	MOV TL1, R1	Swap the counts as calculated above
	MOV 00, R3	
	MOV R3, 01	
	MOV R1, 00	
	MOV TH1, R2	
	MOV 00, R4	
	MOV R4, 02	
	MOV R2, 00	
	SETB TR1	
	CLR TF1	
	RETI	
	org 0100 H	
main :	SETB P1.5	
	MOV IE, #88H	
	MOV TMOD, #10H	
	MOV R1, #83H	
	MOV R2, #0FFH	
	MOV R3, #69H	
	MOV R4, #0FEH	
	MOV TH1, R4	
	MOV TL1, R3	
	SETB TR1	
here :	SJMP here	
	end	

Program 1.20.15

If the crystal frequency of 8051 is 12 MHz, write its assembly language program to do the following :

Generate a delay of 2 ms.

Soln. :

Generate a delay of 2ms.

Crystal frequency = 12 MHz

$$\therefore\ T = \frac{12}{12 \times 10^6} = 1\ \mu\text{s}$$

Let us determine the count to get a delay of 1 ms

$$\therefore\ \frac{1\ \text{ms}}{1\ \mu\text{s}} = 1000\ \text{clocks}$$

$\therefore$ Count = 65536 – 1000 = 64536

$\therefore$ Count = FC18 H

To get a delay of 2 msec we have to repeat the delay of timer 0, 2 times.

Program :

Label	Instruction	Comments
	MOV TMOD,#01H	Timer 1, mode 0
	MOV R0,#2	Count for running delay 2 times
BACK :	MOV TL0,18H	Load count in TL0
	MOV TH0,FCH	Load count in TH0
	SETB TR0	Start Timer 0
AGAIN :	JNB TF0, AGAIN	Stay until timer rolls over
	CLR TR0	Stop Timer 0
	CLR TF0	Clear Timer flag
	DJNZ R0, BACK	If R0 ≠ 0, reload timer
	END	

Program 1.20.16

Write a program to generate square wave with 50% duty cycle from P1.5 (use timer 0) (Total time period = 30.38 μs) (clock frequency 11.0592 MHz)

Soln. :

The period of square wave is 30.38 μs. The duty cycle of the square wave is 50%. Hence, the square wave will be high for 15.19 μs. XTAL = 11.0592 MHz i.e. counter will count up every 1.085 μs.

$\therefore \frac{15.19\ \mu s}{1.085\ \mu s} = 14$, 1.085 μs will make a 30.38 μs pulse.

The values that should be loaded into TH and TL registers are

$65536 - 14 = (65522)_{10} = FFF2\ H.$

$\therefore$ TL = F2H and TH = FFH

Program :

Label	Instruction	Comments
	MOV TMOD,#01	Load TMOD register in timer 0, mode 1
L1 :	MOV TL0,#1AH	Load TL0
	MOV TH0,#0FFH	Load TH0
	SETB TR0	Start Timer 0
L2 :	JNB TF0, L2	Remain until timer rolls over
	CLR TR0	Stop Timer 0
	CPL P1.5	
	CLR TF0	Clear timer 0 flag
	SJMP L1	Reload timer as mode 0 is not auto reload

Program 1.20.17

Write a following programs (Use 8051 μc)

(i) Create a square wave of 50% duty cycle on bit 0 of port 1.

(ii) Create a square wave of 66% duty cycle on bit 3 of port 1

Soln. :

(i) Create a square wave of 50% duty cycle on bit 0 of port 1.

50% duty cycle indicates that the on time and off time is same. Hence, we will toggle bit P1.0 with time delay in each state.

Program :

Label	Instruction	Comments
L1 :	SETB P1.0	Make P1.0 =1
	ACALL DELAY	Wait for sometime
	CLR P1.0	Make P1.0 = 0
	ACALL DELAY	Wait for sometime
	SJMP L1	Keep doing it.

(ii) Create square wave of 66% duty cycle on bit 3 of Port 1.

Let crystal frequency = 12 MHz.

$\therefore$ Time for 1 machine cycle $= \frac{12}{12\ MHz} = 1\ \mu sec.$

Let square wave period = 1 msec

Hence the delay required.

(i) For "ON" period

$\frac{660\ \mu sec}{1\ \mu sec} = 660$

$\therefore$ count $= 65536 - 660 = (64876)_{10}$

$\therefore$ count = FD6C H

(ii) For "OFF" period

$\frac{340\ \mu sec}{1\ \mu sec} = 340$

$\therefore$ count $= 65536 - 340 = (65196)_{10}$

$\therefore$ count = FEAC H

Program :

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	
	ORG 001BH	
	CPL P1.3	Toggle output pin
	CLR TR1	
	MOV TL1, R1	Swap the contents as calculated above

Label	Instruction	Comments
	MOV 00,R3	
	MOV R3,01	
	MOV R1,00	
	MOV TH1,R2	
	MOV 00,R4	
	MOV R2,00	
	SETB TR1	
	CLR TF1	
	RETI	
main :	ORG 0100H	
	SETB P1.3	
	MOV IE,#88H	
	MOV TMOD,#10H	
	MOV R1,#ACH	
	MOV R2,#FEH	
	MOV R3,#6CH	
	MOV R4,#FDH	
	MOV TH1,R4	
	MOV TL1,R3	
	SETB TR1	
here:	SJMP here	
	END	

Program 1.20.18

Assume crystal frequency of 12 MHz to 8051 microcontroller, write assembly language program to generate square wave of 1 KHz on port bit P1.0 using timer 1 interrupt.

Soln. :

$$\text{Frequency} = 1 \text{ KHz}$$

$$\therefore \quad 1 \text{ clock pulse} = \frac{1}{1 \text{ KHz}} = 1 \text{ msec}$$

$\therefore$ 500 μsec is "ON time" and 500 μsec is the "off" time

Hence, a delay of 500 μsec is required.

$$\therefore \quad \text{count} = \frac{500 \ \mu\text{sec}}{1 \ \mu\text{sec}}$$

(as crystal frequency = 12 MHz)

$$\therefore \quad \text{count} = 500$$

$$\therefore \quad \text{count} = 65536 - 500$$

$$= (65036)_{10} = \text{FE0CH}$$

$$\therefore \quad \text{TL1} = \text{0CH and TH1} = \text{FEH}$$

Program :

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	By pass interrupt service routine
	ORG 000BH	
	CPL P1.0	Complement P1.0 bit
	RETI	Return from ISR
	ORG 1000H	
main :	MOV TMOD,#20H	Initialize timer 1 as mode 2
	MOV TL1,#0CH	Load timer count
	MOV TH1,#FEH	
	MOV IE,88H	Enable Timer 1 interrupt
	SETB TR1	Start Timer 1
here :	SJMP here	

Program 1.20.19

Four outputs of a BCD switch are connected to pins of port P1 and corresponding seven segment code is to be displayed using port 2. Write assembly language program for 8051 to read switch (number) and display in seven segment format using look up table method. Assume look up table for code is located from address 4000H (ROM).

Soln. : Program :

Label	Instruction	Comments
	MOV DPTR, #4000H	initialize look up table pointer
L1:	MOV A, P1	read switch number in accumulator
	MOV DPL, A	A to point the seven segment code
	MOV A, @DPTR	get the code to be displayed
	MOV P2, A	display the code on port 2
	ACALL DELAY	call delay
	SJMP L1	
DELAY :	MOV R0, 0FFH	delay routine
L2 :	DJNZ R0, L2	continue till R0 = 00 H
	RET	

Program 1.20.20

Using Timer auto reload mode of 8051 generate a square wave of 2 KHz on port pin 1.0, using interrupt technique. Write an assembly language program for the same. Assume crystal frequency to be 11.0592 MHz.

Soln. : Let us assume that duty cycle of square wave is 50%. Hence the square wave will be high for 250 μs and low for 250 μs.

XTAL = 11.0592 MHz. i.e. counter will count up every 1.085 μs.

$$\frac{250}{1.085} = 230$$

1.085 intervals will make a $\frac{500\ \mu s}{2\ KHz}$ pulse

The values that should be loaded into TH and TL registers are

$$65536 - 230 = (65306)_{10} = FF1AH$$

$$\therefore \quad TL = 1AH \text{ and } TH = FFH$$

Program :

Label	Instruction	Comments
	ORG 0000H	
	LJMP main	Bypass interrupt service routine
	ORG 000BH	
	CPL P1.0	Complement P1.0 bit
	RETI	Return from ISR
	ORG 1000H	
main :	MOV TMOD, #20H	Initialize timer 1 in mode 2
	MOV TL1, #1AH	Load timer count LSB TL1 = 1AH
	MOV TH1, #FFH	TH1 = FFH
	MOV IE, 88H	start timer 1
here :	SJMP here	

Syllabus Topic : Serial Communication and Modes

1.21 Serial Communication and Modes

- For communication between two computer systems we can send and receive the data bits serially.
- **The microcontroller 8051 supports full duplex serial communication.** A full duplex asynchronous serial interface can be implemented on using serial interface control circuitry.
- It has **serial data communication circuit which uses a register in the SFR called SBUF to hold the data. The SFR register SCON controls the data communication, the SFR register PCON along with timer 1 controls the data rates.**

1.21.1 SBUF Register

Size and use : The SBUF is an 8 bit register used for serial communication in 8051.

- The SBUF register comprises of two registers physically; one of them is write only and is used to hold the data that is to be transmitted out from the microcontroller via the TxD pin, while the other is read only and holds the data that is received from the external sources via the RxD pin.
- A **double buffered receiver** is used in the circuit so that the receiver can receive a second character when the first one is in an intermediate register. Double buffering reduces the chances of an overrun error and complexity. But if the previous byte is not read when and the next byte is completed its reception, the first byte received will be lost.

1.21.2 SCON Register SPPU - Dec. 13, Aug. 14

University Question

Q. Explain SCON register in details.
(Dec. 2013, 8 Marks, Aug. 2014 (In Sem.), 3 Marks)

Size : It is an 8 bit register.

- Serial data communication is a relatively slow process. In order not to tie up with valuable processor time, serial data flags are included in SCON to aid in efficient data transmission and reception. The serial data flags in SCON, TI and RI, are set whenever a data byte is received (RI) or transmitted (TI). These flags are ORed together to produce an interrupt to the program, indicating that the byte is received / transmitted and hence to get ready for next byte (i.e. read the byte from SBUF in case of reception or write the next byte into the SBUF for serial transmission). The program must read these flags to find out which bit has created the interrupt and clear the bit.
- Transmission of serial data bits begins anytime data is written in to SBUF. TI is set to a 1 when the data has been transmitted and signifies that the SBUF is empty and another byte can be sent.
- Reception of serial data will begin if they receive enable bit (REN) in SCON is set to 1 for all modes. In addition, for Mode 0 only, RI must be cleared to 0. RI is set to 1 when a byte is received in all modes. REN is the only program control to prevent or to receive the serial data. (Fig. 1.21.1 shows the SCON register)

(MSB)							(LSB)
SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows :

SM0	SM1	Mode	Description	Baud Rate
0	0	0	Shift register	f_{osc} / 12
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	f_{osc} / 64 or f_{osc} / 32
1	1	3	9-bit UART	variable

- SM2 enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2 = 1 then RI will not be activated if a valid stop bit was not received. In mode 0, SM2 should be 0.
- REN enables serial reception. Set by software to enable reception. Clear by software to disable reception.
- TB8 is the 9th data bit that will be transmitted in modes 2 and 3. Set or clear by software as desired.
- RB8 in modes 2 and 3, is the 9th data bit that was received. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In mode 0, RB8 is not used. (concept of stop bit will be dealt with later in this section).
- TI is transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
- RI is receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

Fig. 1.21.1 : SCON - serial port control / status register

1.21.3 PCON Register

- The PCON register is not bit addressable, It is used for controlling the data rate. The baud rate can be doubled if the SMOD bit is changed from 0 to 1.

PCON register

(MSB)							(LSB)
SMOD	-	-	-	GF1	GF0	PD	$\overline{IDL}$

Table 1.21.1 : PCON register

Symbol	Position	Name and Function
SMOD	PCON.7	Double Baud rate bit. When set to a 1 and Timer 1 is used to generate baud rate, and the Serial Port is used in modes 1,2 or 3.
—	PCON.6	(Reserved)
—	PCON.5	(Reserved)
—	PCON.4	(Reserved)
GF1	PCON.3	General-purpose flag bit.
GF0	PCON.2	General-purpose flag bit.
PD	PCON.1	Power Down bit. Setting this bit activates power down operation.
$\overline{IDL}$	PCON.0	Idle mode bit. Setting this bit activates idle mode operation.

1.21.4 Modes of Serial Communication

- The serial interface can be operated in four different modes that can be configured using SCON (Serial port control status) register.
- The 4 modes behave as

(i) Mode 0 : Shift Register with fixed baud rate
(ii) Mode 1 : 8 bit UART with variable baud rate
(iii) Mode 2 : 9 bit UART with fixed baud rate.
(iv) Mode 3 : 9 bit UART with variable baud rate.

- The modes 2 and 3 have special provision for multiprocessor communication i.e. we can have master / slave configuration.
- Baud rate for serial mode can be adjusted and the clock source required for setting baud rate is provided on chip. The Baud rate is fixed for mode 0, while it is variable for modes 1, 2 and 3. Variable baud rates are determined by the Timer 1 overflow rate.

1.21.4.1 Mode 0

- This mode is a half duplex **synchronous** mode. It is referred as the shift register. The shift register has shift left, shift right operation. The shifting operation of data bits is sequentially done.

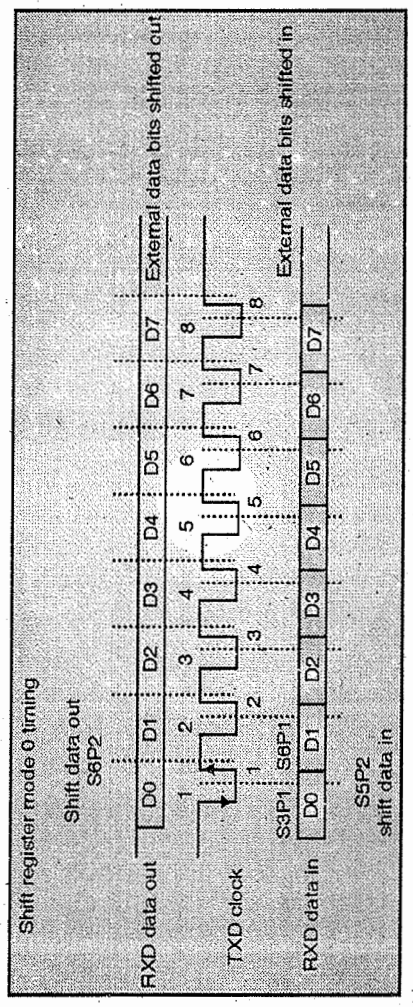


Fig. 1.21.2 : Shift register mode 0 timing

- The input to the shift register is data and clock. The serial data is transmitted and received through the RxD line. The clock is generated by the TxD line. The TxD shift clock is a square wave, low for states S3, S4 and S5 of the machine cycle, while high for S6, S1 and S2 as shown in Fig. 1.21.2.
- As eight bits are transmitted at a time, the start and stop bits are not required. The LSB of data is transmitted and received first.
- The **Baud rate for mode 0 is fixed.**

$$\textbf{baud rate} = \frac{\textbf{Oscillator frequency}}{\textbf{12}}$$

- The microcontroller chip has limited number of I/O lines available. In order to increase the I/O lines we can provide external I/O chips like shift register.

1.21.4.2 Mode 1

- It is a full duplex mode. It supports 8 bit asynchronous communication. Whenever there is a change in data 1 is transmitted, otherwise 0 is transmitted.
- Although the data bits are 8 bit, the number of transmitted bits are 10 i.e. 1 start bit, 1 stop bit and 8 data bits.
- Whenever a character is serially transmitted the transmitting and receiving device should satisfy this communication protocol.

Fig. 1.21.3 shows the UART data word.

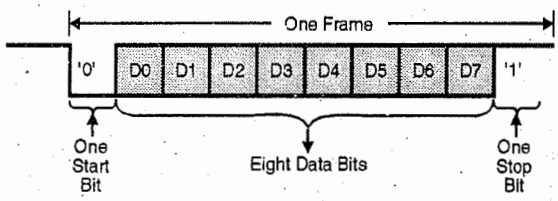


Fig. 1.21.3 : Transmission of 1 byte, formed by 1 start and 1 stop bit

- **The baud rate for mode 1 is variable.**

1.21.4.3 Mode 2

- **It supports full duplex transmission with supports 11 bit asynchronous communication.** Whenever there is change in data '1' is transmitted, otherwise zero is transmitted. Such an operation is called non-return to zero (NRZ) operation.

- In this mode the number of transmitted bits are 11 i.e. one start bit, nine data bits and one stop bit. The 9th data bit to be transmitted is to be stored in the TB8 bit in the SCON register and is stored in the RB8 bit of SCON when data is received. The start and stop bits are discarded.
- **The baud rate for mode 2 is fixed and can expressed as,**

$$\textbf{baud rate} = \frac{2^{SMOD}}{64} \times \textbf{Oscillator frequency}$$

If SMOD = 0,

$$\text{baud rate} = \frac{1}{64} \times \text{Oscillator frequency and}$$

If SMOD = 1,

$$\text{baud rate} = \frac{1}{32} \times \text{Oscillator frequency}$$

In comparison to mode 0, the baud rate in mode 2 is higher than standard communication rates. This is done because multiprocessor systems demand high data rates.

- Mode 2 is mainly used for **multiprocessor communication**. Multiprocessor communication means that 'N' number of microcontroller based systems are interfaced to each other through the serial communication channel. This helps to transfer the data from one system to another, like a LAN network.

1.21.4.4 Mode 3

Mode 3 is similar to mode 2 except that in mode 3 the baud rate is determined as in mode 1 using Timer 1.

1.21.4.5 Summary of Serial Port Operating Modes

Mode	Transmission format	Baud rate
Mode 0	8 data bits	$\frac{1}{12} \times$ oscillators frequency
Mode 1	10 data bits (1 start bit, 8 data bits, 1 stop bit)	variable
Mode 2	11 data bits (1 start bit, 9 data bits, 1 stop bit)	$\frac{1}{32} \times$ oscillator frequency Or $\frac{1}{64} \times$ oscillator frequency
Mode 3	11 data bits (1 start bit, 8 data bits, programmable 9th data bit, 1 stop bit)	Variable

1.21.4.6 Baud Rate

- In serial communication the rate at which the data bits are transmitted is called as **baud rate.**

  The baud rate is defined as bits/second or the changes in voltage levels/second.

  The typical baud rates are 110, 300, 600, 1200, 2400, 4800 and 9600.

1.21.5 Generating Baud Rates for Serial Port Operating Modes

SPPU - Aug. 15

University Question

Q. In serial communication, how baud rate is set ?

(Aug. 2015 (In Sem.), 4 Marks)

- Baud rate for serial modes can be adjusted. The clock source required for setting the baud rate is provided on the chip. The baud rate is fixed for mode 0 and 2 and it is variable for modes 1 and 3. The variable baud rates are determined by Timer 1 overflow rate.

1.21.5.1 Mode 0

- Mode 0 has a fixed baud rate.
- It is expressed as

$$\textbf{baud rate} = \frac{1}{12} \times \textbf{oscillator frequency}$$

1.21.5.2 Mode 1

SPPU - Aug. 14, Aug. 15

University Questions

Q. Calculate the hexadecimal count in TH1 when the baud rate of controller is 1200.

(Aug. 2014 (In Sem.), 3 Marks)

Q. Show calculation for 2400 baud rate.

(Aug. 15 (In Sem.), 2 Marks)

- The baud rate for Mode 1 is determined by timer 1 overflow rate. Typically Timer 1 is used in mode 2 as an auto reload 8 bit timer.

$$\text{baud rate} = \frac{2^{SMOD}}{32} \times \frac{\text{Oscillator frequency}}{12 \times (256 - (TH1))}$$

SMOD is a control bit in the PCON register. It can be '0' or '1'.

- If the timer 1 is not in mode 2 then the baud rate is

$$\text{baud rate} = \frac{2^{SMOD}}{32} \times (\text{Timer 1 overflow rate})$$

Assume that XTAL = 11.0592 MHz. Now we have to set the baud rate to :

(a) 9600 (b) 4800 (c) 2400 (d) 1200.

For setting the baud rate we need to get the value that is to be loaded into the TH1.

$$\text{Machine cycle frequency of 8051} = \frac{11.0592 \text{ MHz}}{12}$$

$$= 921.6 \text{ KHz}$$

$$\text{Frequency provided by UART} = \frac{921.6 \text{ KHz}}{32}$$

$$= 28{,}800 \text{ Hz}$$

($\because$ The UART circuitry divides the machine cycle frequency of 8051 by 32)

(a) To get baud rate of 9600

As $\frac{28800}{3} = 9600 \quad \therefore \quad$ **TH1 = FFH – 3 = FDH**

(b) To get baud rate of 4800

As $\frac{28800}{6} = 4800 \quad \therefore \quad$ **TH1 = FFH – 6 = FAH**

(c) To get baud rate of 2400

As $\frac{28800}{12} = 2400 \quad \therefore \quad$ **TH1 = FFH – $(12)_{10}$ = F4H**

(d) To get baud rate of 1200

As $\frac{28800}{24} = 1200 \quad \therefore \quad$ **TH1 = FFH – $(24)_{10}$ = E8H**

Baud rate	TH1 (Decimal value)	TH1 (Hex)
9600	–3	FDH
4800	–6	FAH
2400	–12	F4H
1200	–24	E8H

– The Table 1.21.2 shows the commonly used baud rates and the way in which they can be obtained from Timer 1. The oscillator frequency is specified.

Table 1.21.2

Baud Rate	f_{osc}	SMOD	Timer		
			C/$\bar{T}$	Mode	Reload value
Mode 0 maximum = 1 Mbps	12 MHz	×	×	×	×
Mode 1,3 maximum = 62.5 kbps	12 MHz	1	0	2	FFH
Mode 2 maximum = 375 kbps	12 MHz	1	×	×	×
19.2 kbps	11.059 MHz	1	0	2	FEH
9.6 kbps	11.059 MHz	0	0	2	FDH
4.8 kbps	11.059 MHz	0	0	2	FAH
2.4 kbps	11.059 MHz	0	0	2	F4H
1.2 kbps	11.059 MHz	0	0	2	E8H
137.5 bps	11.986 MHz	0	0	2	1DH
110 bps	6 MHz	0	0	2	72H
110 bps	12 MHz	0	0	1	FEEBH

1.21.5.3 Mode 2

– The baud rate for mode 2 is fixed. It is expressed as

$$\textbf{baud rate} = \frac{2^{SMOD}}{64} \times \textbf{oscillator frequency}$$

If SMOD = 0

$$\textbf{baud rate} = \frac{1}{64} \times \textbf{oscillator frequency}$$

If SMOD = 1

$$\textbf{baud rate} = \frac{1}{32} \times \textbf{oscillator frequency}$$

Timer 1 is used in mode 2 to carry out serial communication.

SMOD is a control bit in the PCON register. The PCON register is not bit addressable. Its address is 87H. To set the SMOD bit one way is logical ORing the PCON register i.e. ORL PCON, # 80H.

1.21.5.4 Mode 3

The baud rate in mode 3 is variable and sets up exactly similar to that in mode 1.

1.21.5.5 Multiprocessor Communication in 8051

– In 8051 serial modes 2 and 3 have a special provision for multiprocessor communications.

- In these modes, 9 bits of data are received. The 9th bit goes into RB8. Then there is a stop bit. The port can be programmed such that when the stop bit is received, the serial port interrupt is activated if RB8 is set i.e. RB8 = 1. This feature can be enabled by setting the SM2 bit in SCON register.
- A method to use the above feature in multiprocessor systems is given below.
- When the master processor wants to transmit a block of data to one of the may slaves, it sends an address that identifies the target slave.
- An address byte is different than the data byte i.e. the 9th bit in an address byte is 1 whereas the 9th bit in a data byte is '0'.
- If SM2 = 1 then no slave will be interrupted by a data byte.
- However, an address byte can interrupt all slaves, so that each slave can examine the received byte, to check whether it is being addressed.
- The addressed slave will clear the SM2 bit and prepare to receive the data bytes that are coming. The slaves that were not being addressed have SM2 = 1.

Syllabus Topic : Data Transmission using Serial Port

1.22 Data Transmission using Serial Port

Inorder to transfer data bytes serially, the following steps must be considered.

Step I : The TMOD register is loaded with the value 20 H. This will indicate the use of timer 1 in mode 2 to set the baud rate.

Assume that XTAL = 11.0592 MHz. Now we have to set the baud rate to (a) 9600 (b) 4800 (c) 2400 (d) 1200. For setting the baud rate we need to get the value that is to be loaded into the TH1.

The machine cycle frequency of

$$8051 = \frac{11.0592}{12} \text{ MHz} = 921.6 \text{ KHz}$$

The frequency provided by UART $= \frac{921.6}{32}$ KHz

= 28,800 Hz.

(a) To get baud rate 9600, $\because \frac{28,800}{3} = 9600$

where TH1 = FFH – 3 = FD H.

(b) To get baud rate 4800, $\because \frac{28,800}{6} = 4800$

where TH1 = FFH – 6 (decimal) = FA H.

(c) To get baud rate 2400, $\because \frac{28,800}{12} = 2400$

where TH1 = FFH – 12 (decimal) = F4 H

(d) To get baud rate 1200, $\because \frac{28800}{24} = 1200$

where TH1 = FFH – 24 (decimal) = E8 H

Summarizing

Baud Rate	TH 1 (Decimal)	TH 1 (Hex.)
9600	– 3	FD
4800	– 6	FA
2400	– 12	F4
1200	– 24	E8

Step II : Load TH1 with one of the values given is above table to set the baud rate for serial data transfer. (These values of baud rates are assuming XTAL = 11.0592 MHz.)

Step III : Load the SCON register with 50 H. This indicates serial mode 1.

Step IV : TR1 is set to 1 to start timer 1.

Step V : Clear TI bit.

Step VI : The character byte that is to be serially transmitted is written into the SBUF register.

Step VII : To check whether the character byte has been completely transmitted observe the TI bit.

Step VIII : To transfer next character byte, go to step V.

Program 1.22.1

Write the assembly language program for 8051, to send one byte of data serially with baud rate of 1200. The oscillator frequency is 12 MHz. Assume suitable mode for serial port.

Soln. : When 12 MHz crystal is used and a standard baud rate of 1200 Hz is required then, TH1 value will be

$$TH1 = 256 - \frac{k \times \text{Oscillator frequency}}{384 \times \text{Baud rate}}$$

$$TH1 = 256 - \frac{1 \times 12 \times 10^6}{384 \times 1200}$$

$$TH1 = 229.95$$

$$TH1 = E6H$$

Label	Instruction	Comment
	MOV TMOD, #20H	Timer 1, mode 2
	MOV TH1, #E6H	Baud rate 1200
	MOV SCON, #50H	
	SETB TR1	start timer 1
L2:	MOV SBUF, #"A"	transfer letter A serially
L1 :	JNB TI, L1	
	CLR TI	clear TI for next character
	SJMP L2	keep sending character

1.22.1 Importance of the TI Flag Bit

In order to understand the importance of TI flag consider that a data byte is to be transmitted through the TxD pin in the following sequence :

(i) Initially the data byte that is to be transmitted is loaded into the SBUF register.
(ii) Then the start bit is transferred.
(iii) The start bit is followed by data byte. The data is transferred bit by bit.
(iv) Then the stop bit is transferred. When the stop bit is transferred the 8051 raises the TI flag bit, indicating that the last character was transmitted and it is now ready to transmit next character.
(v) By monitoring the TI flag, we make sure that we are not overloading the SBUF register. If before raising the TI flag we write another byte into the SBUF register then the untransmitted part of the earlier data byte transmitted is lost.
(vi) After the SBUF is loaded with new byte, the TI flag must be forced to 0 by CLR TI instruction so that new data byte can be transmitted. The programmer can check the TI flag bit by "JNB TI, xx" instruction or by using an interrupt.

Program 1.22.2

Write an assembly language program to send message 'WELCOME' to COM port of PC at 4800 baud rate. Assume XTAL frequency = 11.0592 MHz.

Soln. :

Label	Instruction	Comment
	MOV TMOD, #20H	timer 1, mode 2
	MOV TH1, #FAH	4800 baud rate
	MOV SCON, #50H	8 bit, 1 stop, REN enabled.
	SETB TR1	start timer 1
L1 :	MOV A, #"W"	transfer "W"
	ACALL TRANS	
	MOV A, #"E"	transfer "E"
	ACALL TRANS	
	MOV A, #"L"	transfer "L"
	ACALL TRANS	
	MOV A, #"C"	transfer "C"
	ACALL TRANS	
	MOV A, #"O"	transfer "O"
	ACALL TRANS	
	MOV A, #"M"	transfer "M"
	ACALL TRANS	
	MOV A, #"E"	transfer "E"
	ACALL TRANS	
	SJMP L1	load SBUF
TRANS :	MOV SBUF, A	Put the data to be transmitted in SBUF
L2 :	JNB TI, L2	wait for last bit to transfer
	CLR TI	get ready for next character byte
	RET	

Program 1.22.3

Write a program to transmit letter 'A' to serial COM port using 8051 at 9600 baud rate. Assume XTAL = 11.0592 MHz.

Soln. :

Label	Instruction	Comments
	MOV TMOD, #20H	Timer 1, mode 2
	MOV TH1, #FDH	9600 baud rate
	MOV SCON, #50H	8 bit, 1 stop-bit, REN enabled
	SETB TR1	Start Timer 1
L2 :	MOV SBUF, # "A"	Letter A to be transmitted
L1 :	JNB TI, L1	Wait for last bit
	CLR TI	Clear TI for next character
	SJMP L2	Keep sending A

Syllabus Topic : Data Reception using Serial Port

1.23 Data Reception using Serial Port

Inorder to receive the data bytes serially, the following steps must be considered

Step I : The TMOD register is loaded with the value 20 H. This indicates the use of timer 1 in mode 2 to set the baud rate.

Step II : Load TH1 with value to set the baud rate for serial data transfer.

Step III : Load the SCON register with 50 H. This indicates serial mode 1.

Step IV : TR1 is set to 1 to start timer 1.

Step V : Clear RI bit.

Step VI : To check whether the entire character has been received, the RI bit is observed.

Step VII : If RI is set, then the byte is received in SBUF register. Save this character byte.

Step VIII : To receive next character, go to step V.

Program 1.23.1

Write an 8051 assembly language program to receive bytes serially with baud rate of 2400, 8 bit data and 1 stop bit. Simultaneously send the received character bytes to port 2.

Soln. :

Label	Instruction	Comment
	MOV TMOD, #20H	timer 1 mode 2.
	MOV TH1, #F4H	2400 baud rate
	MOV SCON, #50H	8 bit, 1 stop bit, REN enabled.
	SETB TR1	start timer 1
L1 :	JNB RI, L1	wait for character to be received completely
	MOV A, SBUF	save the received character
	MOV P2, A	send character to port 2
	CLR RI	Get ready to receive next character
	SJMP L1	Goto receive next character.

1.23.1 Importance of RI Flag

While receiving a data byte the microcontroller 8051 follows these steps :

(i) It receives the start bit indicating that the next bit is the first bit of the character byte it is about to receive.

(ii) Then the 8 bit data is received one character at a time. When the last bit is received, a byte is formed and placed in the SBUF register.

(iii) Then a stop bit is received. On reception of the stop bit, the 8051 raises the RI flag indicating that a data byte is received. It must be picked up before any other character is received.

(iv) Once the RI flag is raised we know that the character byte is received and it is present in the SBUF register. The SBUF contents must be loaded to some register or memory location before they are lost.

(v) After copying the SBUF contents, the RI flag bit must be forced to 0 by "CLR RI" instruction so that microcontroller can receive next byte of data. The programmer can check the RI flag bit by the instruction "JNB RI, xx" or by using interrupt.

1.23.2 Doubling the Baud Rate

There are two methods to increase the baud rate of data transfer in 8051. They are:

(i) Use a crystal of high frequency
(ii) Change a SMOD bit in the PCON register.

Program 1.23.2

Write an assembly program to take data from ports 0 and 1, one after the other and transfer data serially continuously.

Soln. :

Label	Instruction	Comment
	org 0000H	
	LJMP main	
	org 1000H	
main :		
	MOV TMOD, #20H	Timer 1 in Mode 2 for Baud rate
	MOV TH1, #0FDH	Timer 1 count for 9600 baud/sec
	MOV SCON, #50H	Mode 1 for serial communication
	MOV P0, #0FFH	P0 as input port

Label	Instruction	Comment
	MOV P1, #0FFH	P1 as input port
	SETB TR1	Run timer 1
here :	MOV SBUF, P0	Send data from P0 to serial port
WAIT :	JNB TI, WAIT	Wait for transmission to complete
	CLR TI	
	MOV SBUF, P1	Send data from P1 to serial port
WAIT1 :	JNB TI, WAIT1	
	CLR TI	
	SJMP here	

Program 1.23.3

Write an assembly program to receive the data which has been sent in serial form and send it out to port 2 in parallel form continuously.

Soln. :

Label	Instruction	Comment
	org 0000H	
	LJMP main	
	org 1000H	
main :	MOV TMOD, #20H	Timer 1 in Mode 2 for Baud rate
	MOV TH1, #0FDH	Timer 1 count for 9600 baud/sec
	MOV SCON, #50H	Mode 1 for serial communication
	SETB TR1	Run timer 1
here :	JNB RI, here	
	MOV P2, SBUF	
	CLR RI	
	SJMP here	

Program 1.23.4

Write an 8051 program to transfer WELCOME serially at 9600 baud rate (8-data bits and 1 stop bit) Do this continuously.

Soln. :

Label	Instruction	Comment
	MOV TMOD, #20H	timer 1, mode 2
	MOV TH1, #FDH	9600 baud rate
	MOV SCON, #50H	8 bit, 1 stop, REN enabled.
	SETB TR1	start timer 1
L1 :	MOV A, #"W"	transfer "W"
	ACALL TRANS	
	MOV A, #"E"	transfer "E"
	ACALL TRANS	
	MOV A, #"L"	transfer "L"
	ACALL TRANS	
	MOV A, #"C"	transfer "C"
	ACALL TRANS	
	MOV A, #"O"	transfer "O"
	ACALL TRANS	
	MOV A, #"M"	transfer "M"
	ACALL TRANS	
	MOV A, #"E"	transfer "E"
	ACALL TRANS	
	SJMP L1	load SBUF
L2 :	JNB TI, L2	wait for last bit to transfer
	CLR TI	get ready for next character byte
	RET	

Program 1.23.5

Write instructions to initialize serial port in mode 1 with baud rate of 4800 and crystal frequency of 11.059 MHz.

Soln. :

$$\text{Machine cycle frequency of 8051} = \frac{11.059 \text{ MHz}}{12}$$

$$= 921.6 \text{ KHz.}$$

$$\text{The frequency provided by UART} = \frac{921.6 \text{ KHz}}{32}$$

$$= 28800 \text{ Hz.}$$

$$\text{To get baud rate 4800, } \frac{28,800}{6} = 4800$$

$$\text{TH1} = \text{FFH} - 6 \text{ (decimal)} = \text{FA H}$$

Initialization :

Instruction	Comment
MOV TMOD,#20H	Timer 1, Mode 2
MOV SCON, #4CH	Initialize serial mode 1
MOV TH1, FAH	4800 baud rate

Program 1.23.6

Explain mode 1 and mode 2 of serial port in 8051. Write instructions to initialize serial port in mode 2 with baud rate of 9600.

Soln. :

The machine cycle frequency of 8051 = $\frac{11.059 \text{ MHz}}{12}$

= 921.6 KHz

The frequency provided by UART = $\frac{921.6 \text{ KHz}}{32}$

= 28800 Hz.

To get baud rate 9600, $\frac{28,800}{3}$ = 9600

where, TH1 = FFH – 3 = FDH

Program :

Instruction	Comment
MOV TMOD, #20H	Timer 1, Mode 2
MOV TH1 #0FDH	9600 baud rate
MOV SCON, #50H	8 bit, 1 stop, REN enabled

Program 1.23.7

Write an assembly language program to transmit "MMA" serially at baud rate 9600 continuously.

Soln. :

Label	Instruction	Comment
	MOV TMOD, #20H	Timer 1 mode 2
	MOV TH1, #FDH	9600 baud
	MOV SCON, #50H	8 bit, 1 stop, REN enabled.
	SETB TR1	Start Timer 1
L1 :	MOV A, #"M"	Transmit **M**
	ACALL TRANS	
	MOV A, #"M"	Transmit **M**
	ACALL TRANS	
	MOV A, #"A"	Transmit **A**
	ACALL TRANS	
	SJMP L1	
TRANS :	MOV SBUF A	Load SBUF
L2 :	JNB T1, L2	Wait for last bit to transfer
	CLR T1	
	RET	

Program 1.23.8

Write an assembly language program to transfer the message "HELLO" serially at 9600 baud rate, 9 bit data and 1 stop bit for 8051.

Soln. :

Label	Instruction	Comment
	MOV TMOD, #20H	
	MOV TH1, #FDH	9600 baud rate
	MOV SCON, #50H	
	SETB TR1	Start timer 1
	MOV A, #"H"	transfer "**H**"
	ACALL TRANSMIT	
	MOV A, #"E"	transfer "**E**"
	ACALL TRANSMIT	
	MOV A, #"L"	transfer "**L**"
	ACALL TRANSMIT	
	MOV A,#"L"	transfer "**L**"
	ACALL TRANSMIT	
	MOV A,#"O"	transfer "**O**"
	ACALL TRASMIT	
TRANSMIT :	MOV SBUF, A	Load SBUF
HERE :	JNB TI, HERE	Wait for last bit to transfer
	CLR TI	Clear TI for next character
	RET	

Program 1.23.9

Write a program to receive message from PC to 8051. Message string is "Hello". After this microcontroller sends message to PC "Fine".

Soln. : Fig. P. 1.23.9 shows the connections between 8051 and PC.

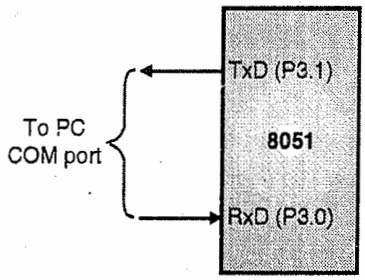


Fig. P. 1.23.9 : Connections between 8051 and PC

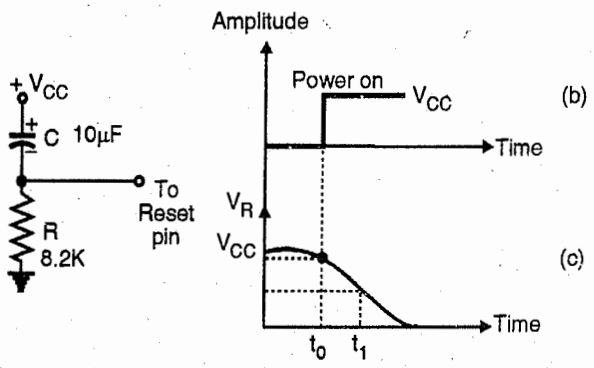
Label	Instruction	Comment
	MOV TMOD, #20H	Initialize Timer 1 in mode 2
	MOV TH1, #0FDH	9600 baud
	MOV SCON, #50H	8 bit, 1 stop, REN enabled
	SETB TR1	
	MOV DPTR,#1000H	Initialize memory pointer to save received data
	MOV R0,#05H	Counter to read 5 characters

Label	Instruction	Comment
REC :	JNB RI,REC	Wait for character
	MOV A,SUB	Read the character
	MOVX @DPTR,A	Save it in memory
	INC DPTR	
	CLR RI	get ready for next character
	DJNZ R0, REC	If not last character, repeat
	MOV DPTR, #My data	Initialize pointer for message
	CLR A	
	MOV R0, #4H	Initialize counter to send 4 characters
	MOVC A, @A+DPTR	Get character
	MOV SBUF,A	Load the data
L1 :	JNB TI,L1	Wait for complete byte transfer
	CLR TI	Get ready for next character
My data :	DB "Fine",0	
	END	

1.24 Reset

Q. Write note on 8051 Reset.

- The Reset pin for microcontroller is active HIGH. Whenever power is switched ON, positive going pulse should be present for two machine cycles (The smallest time interval of time that is required to execute an instruction is called as a machine cycle) on this pin.
- The Reset pin can also be considered as an interrupt because the program cannot block the signal on reset pin.
- Fig. 1.24.1 shows power on reset circuit for microcontroller.



(a) Reset circuit (b) Power on pulse (c) Across resister

m(19.29)**Fig. 1.24.1**

- At time t_0, the power supply is switched ON. The supply voltage V_{CC} appears across the RC network. The entire voltage appears across the resistor R, so V_R is approximately equal to V_{CC}. This resets the 8051.
- Once the capacitor charges, then V_R starts reducing and reaches approximately 0V. This removes reset signal. The oscillator $t_1 - t_0$ is reset time.
- The reset will force all the SFRs to 00 H, port latches are initialised to FF H, SP to 07 H and SBUF is undetermined. The internal RAM is not affected by reset. The internal RAM content is indeterminate.

Table 1.24.1 lists the values of Registers on Reset.

Table 1.24.1

Registers	Reset Value
PC	0000 H
ACC	00 H
B	00 H
PSW	00 H
SP	07 H
DPTR	0000 H
P0 – P3	FF H
IP	xxx00000B
IE	0xx00000B
TMOD	00 H
TCON	00 H
TH0	00 H
TL0	00 H
TH1	00 H
TL1	00 H
SCON	00 H
SBUF	Indeterminate
PCON	0xxxxx00 B

1.25 Power Saving Modes of Operation

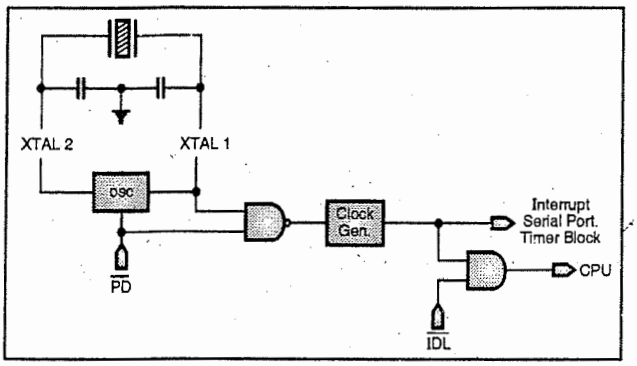
SPPU - Dec. 12

University Question

Q. Explain the need of power saving mode in Microcontroller. **(Dec. 2012, 4 Marks)**

- Power saving feature is available in CHMOS version of the microcontroller. The question now arises that in microcontroller how can one reduce power consumption. The power consumption reduces if some part of the IC is kept in working condition and some part of IC retains previous status and indefinitely stops.
- For applications where power consumption is critical the CHMOS version provides power reduced modes of operation as a standard feature.
- The advantages of reduced **power consumption** are :

(i) It allows the microcontroller to put more functionality into smaller space.
(ii) It allows the use of smaller and lighter power supplies. As less heat is generated, the packaging can be made dense. Also, the use of expensive fans and blowers for cooling is avoided.
(iii) The cooler running chip is more reliable, as most of the random and wear out failures of the microcontroller are related to temperature.



m(19.30)**Fig. 1.25.1 : Hardware to implement idle and power down modes**

- The microcontroller has two power saving features. They are idle and power down modes of operation. Fig. 1.25.1 shows the on chip hardware that implements the reduced power modes.
- The typical values of supply currents for 89C51 in these modes are given in Table 1.25.1.

Table 1.25.1

	Idle mode	Power down mode
89C51	(0.3 × frequency in MHz + 2) mA at 12 MHz I_{CC} = 5.6 mA	3 μA
89C51RD	(0.37 × frequency in MHz + 1) mA at 12 MHz I_{CC} = 5 mA	20 μA

1.25.1 Idle Mode

SPPU - Dec. 12, May 14

University Question

Q. Explain idle mode in detail.

(Dec. 2012, May 2014, 4 Marks)

- As shown in Fig. 1.25.1, the $\overline{IDL}$ is connected to the input of AND gate. The second input to the AND gate is from the clock generator. The output of the AND gate is given to the CPU.
- If $\overline{IDL}$ = 1, then the output of clock generator is given to CPU. But if $\overline{IDL}$ = 0, then output of clock generator will not be passed to the CPU, as output of the AND gate will be zero. So, in the idle mode pulse are not given to the CPU and hence the CPU is at standstill.
- The on chip peripherals i.e. timers, serial port, interrupts, RAM etc. continue to function as normal in the Idle mode.
- The stack pointer, program counter, program status word, accumulator, B register and all other registers maintain their data during idle state.
- The ALE and $\overline{PSEN}$ signals are at logic level high when the microcontroller operates in the Idle mode. Due to this external EPROM can be deselected, if its output is disabled.
- The Idle mode is invoked by setting $\overline{IDL}$ = 0. $\overline{IDL}$ is idle mode bit. It resides in the PCON register.
- The PCON register is not bit addressable, so the $\overline{IDL}$ bit has to be set with a byte operation like. ORL PCON, #01H.

- The general purpose flags GF0 and GF1 in the PCON register, give an indication about the interrupt, whether the interrupt occurred during normal operation or idle mode operation. An instruction that invokes idle mode operation causes either of the flags bits GF0 or GF1 to be set.

PCON register

(MSB)							(LSB)
SMOD	-	-	-	GF1	GF0	PD	$\overline{IDL}$

Table 1.25.2

Symbol	Position	Name and Function
SMOD	PCON.7	Double Baud rate bit. When set to a 1 and Timer 1 is used to generate baud rate, and the Serial Port is used in modes 1,2 or 3.
—	PCON.6	(Reserved)
—	PCON.5	(Reserved)
—	PCON.4	(Reserved)
GF1	PCON.3	General-purpose flag bit.
GF0	PCON.2	General-purpose flag bit.
PD	PCON.1	Power Down bit. Setting this bit activates power down operation.
$\overline{IDL}$	PCON.0	Idle mode bit. Setting this bit activates idle mode operation.

1.25.2 Termination / Exit from Idle Mode

There are two ways to terminate the Idle Mode.

(1) The activation of any enabled interrupt will cause the PCON.0 (i.e. 0th bit of PCON) to be cleared by the hardware, terminating the idle mode. After clearing the bit the CPU will be activated. This causes an interrupt service routine to be executed. After the execution of interrupt service routine, the program execution will continue from the next instruction that invoked the idle mode.

(2) The other way of termination from the Idle Mode is to Reset the system. As the clock oscillator is running, the hardware reset needs to be held active for two machine cycles (i.e. 24 oscillator periods) to complete the reset. The signal at RST pin of microcontroller clears the $\overline{IDL}$ bit. Finally, CPU resumes program execution from where it was left off i.e. at the instruction that follows the instruction that invoked the idle mode.

Note : The termination from Idle Mode writes 1s to all the ports, initializes all SFRs to their reset values and it restarts program execution from location 0000H.

1.25.3 Power Down Mode

SPPU - Dec. 12, May 14

University Question

Q. Explain power down in detail.

(Dec. 2012, May 2014, 4 Marks)

- To enter the power down mode, we have to set bit 1 (PCON.1) in the PCON register. This will cause the oscillator operation to stop. If the microcontroller was running from an external oscillator, it gates off the path to internal phase generators and no internal clock is generated even if the external oscillator is running. i.e. with the clock frozen / stopped all the functions are stopped.
- However as long as the supply voltage V_{CC} is maintained, the contents of internal RAM and SFRs are held.
- In power down mode the ALE and $\overline{PSEN}$ signals are at logic level low.
- The port pins output the values that are held by their respective SFRs.
- In power down mode each and every activity is stopped / frozen. So, an instruction that sets PCON.1 causes that to be the last instruction to be executed before going into the power down mode.

Table 1.25.3 summarizes the status of pins in Idle and power down modes.

Table 1.25.3

Pin	Internal Execution		External Execution	
	Idle	Power down	Idle	Power down
ALE	1	0	1	0
$\overline{PSEN}$	1	0	1	0
P0	SFR data	SFR data	High impedance	High impedance
P1	SFR data	SFR data	SFR data	SFR data
P2	SFR data	SFR data	PCH	SFR data

Pin	Internal Execution		External Execution	
	Idle	Power down	Idle	Power down
P3	SFR data	SFR data	SFR data	SFR data

SFR data : internal register data.

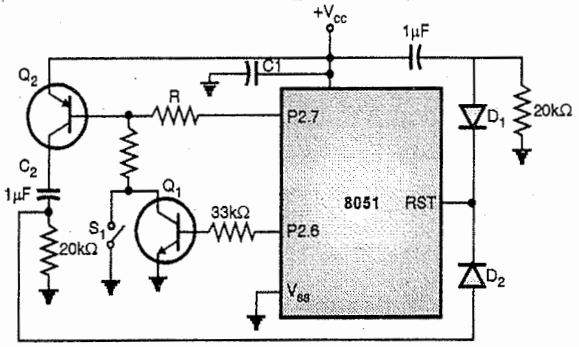
PCH : higher byte of program counter.

1.25.4 Termination from Power Down Mode

- The only way to come out of power down mode is hardware reset. As the oscillator was stopped / frozen in the power down mode the RST needs to be active for a long time for the oscillator to restart and stabilise.
- The SFRs are initialised to their reset values and program execution begins from 0000H. The contents on the on chip internal RAM are retained.

1.25.5 Using Power Down Mode

- The software invoked power down mode offers reduction in power consumption in some applications. Sometimes during the power down mode the V_{CC} may be turned off so that quiescent and standby currents are avoided. But care should be taken that before switching off the V_{CC} of a chip, all the chip signals should be logic low irrespective of the families HMOS, CMOS, TTL.
- To ensure safe power down operation it is essential that the V_{CC} for a secondary circuit should shut down after the microcontroller 8051 is in the power down mode **Fig. 1.25.2 shows a circuit for power switching in the power down mode.**
- When the V_{CC} is shut down, the capacitor C_1 provides power-on-reset. It writes 1s to all the port pins. Hence at port P 2.6 the transistor Q_1 turns on, and enables V_{CC} to the secondary circuit through Q_2.
- As soon reset operation is completed, port 2 emits PCH higher byte of program counter that results P 2.6 and P 2.7 to output zeros. The zero at P 2.7 ensures the continuation of V_{CC} to secondary circuit.



m(19.31)**Fig. 1.25.2 : Power switching circuit for power down mode**

- When the 8051 goes into the power down mode, the system software that had written a 1 to P 2.7 and 0 to P 2.6 while normal operation, appear across the port pins causing transistors Q_1 and Q_2 to shut off. This disables V_{CC} to secondary circuit.
- On closing the switch S_1, the secondary circuit can be reenergize and a reset through C_2 is given to switch on the microcontroller.
- The diodes D_1 is used to prevent C_1 from hogging current from C_2 during a secondary reset while diode D_2 is used to prevent C_2 from discharging through the RST pin. The V_{CC} of secondary circuit is then shut off.

Syllabus Topic : Overview of Instruction Set

1.26 Overview of Instruction Set

- In this section we will study the instruction set of Microcontroller 8051. These instructions treat different types of operands uniformly. Register, memory and immediate operands may be specified interchangeably in most instructions.
- The instruction set is divided into number of groups of functionally related instructions. The different groups are :

(1) Data transfer group
(2) Arithmetic group
(3) Bit manipulation group
(4) Program transfer instruction group
(5) Processor control group

Before studying the instruction set, it is essential to know how 8051 accesses the instruction operands in different ways i.e. the Addressing modes.

1.27 Addressing Modes

SPPU - May 12, May 13, Dec. 13, Dec. 14, May 15, Dec. 16

University Questions

Q. State and explain different addressing modes of 8051 with the help of example. **(May 2012, 10 Marks)**

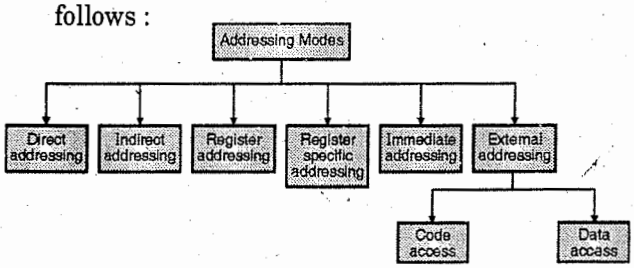
Q. State and explain with the help of examples addressing modes of 8051. **(May 2013, 10 Marks)**

Q. List and explain addressing modes of 8051 with the help of examples. **(Dec. 2013, May 2015, 6 Marks)**

Q. Explain the addressing modes of 8051 with example. **(Dec. 2014, 6 Marks)**

Q. Explain addressing modes of 8051 microcontroller. **(Dec. 2016, 6 Marks)**

- When the microcontroller executes an instruction, it performs specific function on data. The data is stored at some source location. This data must be moved or copied to destination location. The ways by which these addresses locations are specified are called as **Addressing Modes.**
- The Addressing modes for 8051 can be given as follows :



1.27.1 Direct Addressing Mode

- In this mode, the operand is specified by an 8 bit address field in the instruction. One can access all the 128 bytes of internal RAM and the SFRs directly, using the single byte address that is assigned to each RAM location and each special function register.
- The most significant bit in the address decides whether the location is within the on chip internal RAM or in the special function register. If MSB = 0, then the location is within on chip internal RAM. If MSB = 1, then the location is in the special function register.
- The internal RAM uses addresses from 00 H to 7FH to address each byte. The SFR addresses are from 80H to FFH.

Example :

MOV A, 40 H	;	Copy data from address 40 H into register A
MOV R0, 14 H	;	Copy the contents of memory location 14H, to register R0 of the selected bank.

1.27.2 Indirect Addressing Mode

- In this addressing mode, the instruction specifies a register which contains address of an operand i.e. the register holds the actual address that will be used in the data move operation. This address may be 8 bit or a 16 bit address.
- The R0 and R1 of each register bank can be used as an index or pointer register. R0 and R1 point to the contents in the RAM.
- The @ sign indicates the register acts as a pointer to memory location.

Example

MOV A, @R1 ;	Copy the contents of memory location, whose address is specified in R1 register of selected bank to the accumulator.

Note : @ indicates that the register acts like a pointer.
MOV @R0, 85 H ; Copy the data of address 85H to the memory location whose address is specified by the R0 register of selected bank.
Only registers R0 and R1 can be used for indirect addressing. If registers R2 to R7 are used, then in that case the instruction becomes an invalid instruction
e.g. : MOV @R2, A ; invalid instruction.

1.27.3 Register Addressing Mode

Each register bank consists of registers R0 to R7. To access these registers there are special instructions. In the instruction Opcode, 3 bits are reserved for specifying one of the eight registers from the selected register bank. For selecting the register bank, the user has to modify two bits in the PSW.

Example :

MOV A, R2 ; copy the data from register R2 of the selected register bank to register A.

1.27.4 Register Specific Addressing Mode

In this addressing mode the instructions refer to a specific register such as accumulator or data pointer DPTR.

Example :

DA A ; Decimal adjust accumulator for addition.

RR A ; Rotate the contents of accumulator to the right

SWAP A ; Swap the nibbles within the accumulator.

1.27.5 Immediate Addressing Mode

- This method is the simplest method to get the data. In this addressing mode the source operand is a constant rather than a variable. As the data source is a part of the instruction it is immediately available.
- The # sign indicates that the data followed is immediate operand.

Example :

MOV A, #30H ; Copy 30H immediately to accumulator.

MOV P1, #0FFH ; Copy FFH immediately to port 1.

MOV DPTR, #1234H ; Copy 1234H immediately to data pointer (DPTR).

1.27.6 External Addressing Mode

(a) Code access (External ROM access)

- Using these instructions only external program memory can be accessed.
- This addressing mode is preferred for reading look up tables in the program memory. Either the DPTR or PC (program counter) can be used as pointer.

Example :

MOVC A, @A + DPTR ; This instruction will load the accumulator with the byte from program memory. The byte from program memory is fetched from the sum of unsigned eight bit accumulator contents and contents of the DPTR.

(b) Data access (External RAM access)

- Using this addressing mode the programmer can access the external data memory.

Example :

MOVX @R0, A ; This instruction will copy the data from accumulator to the external memory location, whose address is given by register R0. Using Register R0 or R1, the programmer can access external data memory from location 00H to FFH. To access the memory beyond this, DPTR is used.

1.28 Data Transfer Instructions

1.28.1 MOV <dest –byte>, <src–byte>

Mnemonic	MOV <Dest–byte>, <src–byte>
Algorithm	destination = source.
Function	Move byte variable.
Operation MOV <dest–byte>, <src–byte>	– This instruction copies the contents of the source location to the destination location. The contents of source location are unchanged. – It is the most flexible operation. It allows fifteen combinations of source and destination addressing modes. – Depending on the type of transfer the number of bytes required may be 1, 2 or 3 and number of cycles required are 1 or 2.

Let us see the different combinations :

Case (i) : If the destination byte is Accumulator, then source byte may be

1) MOV A, Rn

Operation	MOV A, Rn.	This instruction copies the contents of the register Rn to the accumulator.
Example	MOV A, R1.	This instruction will copy the contents of register R1 of the selected register bank to the accumulator.

2) MOV A, direct

Operation	MOV A, direct	This instruction will copy the contents of direct address given in the instruction to the accumulator.
Example	MOV A, 40H	This instruction will copy the contents from memory location whose address is 40H to the accumulator.

3) MOV A, @Ri

Operation	MOV A, @Ri	This instruction will copy the contents of memory location whose address is specified in the register Ri of the selected bank to the accumulator
Example	MOV A, @R0	This instruction will copy the contents of memory location whose address is specified in the R0 register of the selected bank to accumulator.

4) MOV A, #data

Operation	MOV A, #data	This instruction will copy the immediate data to accumulator.
Example	MOV A, #31H	This instruction will move the data 31H immediately to the accumulator.

Case (ii) : If the destination is a register, then the source may be

5) MOV Rn, A

Operation	MOV Rn, A	This instruction will copy the contents of accumulator to the register Rn of selected register bank.
Example	MOV R5, A	This instruction will copy the contents of accumulator to register R5 of selected register bank.

6) MOV Rn, direct

Operation	MOV Rn, direct	This instruction will copy contents from direct address specified in the instruction to register Rn of selected register bank.
Example	MOV R3, 30H	This instruction will copy the contents from address 30 H to the register R3 of selected register bank.

7) MOV Rn, #data

Operation	MOV Rn, #data	This instruction will copy the immediate data to the register Rn of selected register bank.
Example	MOV R7, #20H	This instruction will copy immediate data 20H to the register R7 of the selected register bank.

Case (iii) : If the destination is a direct address, then source may be.

8) MOV direct, A

Operation	MOV direct, A	This instruction will copy the contents of accumulator to the direct address specified in the instruction.
Example	MOV 80H, A	80H is the address of port 0. This instruction will copy the contents of accumulator to the port 0 latch.

9) MOV direct, Rn

Operation	MOV direct, Rn	This instruction will copy the contents of register Rn of the selected register bank to the direct address specified in the instruction.
Example	MOV 30H, R5	This instruction will copy the contents of register R5 of the selected register bank to the memory location whose address is 30H.

10) MOV direct, direct

Operation	MOV direct, direct	This instruction will copy the contents of source direct address to the destination direct address.
Example	MOV 20H, 30H	This instruction will copy the contents of memory location whose address is 30H to the memory location whose address is 20H.

11) MOV direct, @Ri

Operation	MOV direct, @Ri	This instruction will copy the data from memory location whose address is specified in the Ri register of the selected register bank to the direct address
Example	MOV 20H, @R1	This instruction will copy the data from memory location whose address is given in R1 register of selected register bank to memory location whose address is 20H.

12) MOV direct, #data

Operation	MOV direct, #data	This instruction will copy the immediate data to direct address specified in the instruction.
Example	MOV 10H, #10H	This instruction will copy immediate data 10H to memory location whose address is 10H.

Case (iv) : If the destination byte is a memory location on pointed by Ri, then the source may be

13) MOV @Ri, A

Operation	MOV @Ri, A	This instruction will copy the contents of accumulator to the memory location pointed by Ri.
Example	MOV @R0, A	This instruction will copy the contents of accumulator to memory location whose address is pointed by the R0 register of selected register bank.

14) MOV @Ri, Direct

Operation	MOV @Ri, direct	This instruction will copy the contents of direct address specified in the instruction to the memory location pointed by Ri register of selected register bank
Example	MOV @R0, 30H	This instruction will copy data from memory location whose address is 30 H to the memory location pointed by register R0 of selected register bank.

15) MOV @Ri, #data

Operation	MOV @Ri, #data	This instruction will copy the immediate data to memory location pointed by register Ri of selected register bank.
Example	MOV @R0, #30H	This instruction will copy data immediate data 30H to memory location pointed by R0 register of selected register bank.

1.28.2 MOV DPTR, #data 16

Mnemonic	MOV DPTR, #data 16	**Function**	Load data pointer with a 16-bit constant
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	(DPTR) = #data$_{15-0}$ (DPH) = #data$_{15-8}$, (DPL) = #data $_{7-0}$
Addr. Mode	Immediate Addressing mode	**Flags**	No flags are affected.

Operation	MOV (DPTR) ← #data $_{15-0}$ OR (DPH) ← #data$_{15-8}$, (DPL) ← #data $_{7-0}$	– This instruction will load the data pointer with the 16 bit constant indicated. – The 16 bit constant is loaded into the second and third bytes of the instruction. The second bytes (DPH) holds the high-order byte. While the third byte (DPL) holds low-order byte.
Example	MOV DPTR, #2476 H	This instruction will load the immediate data 2476H into the Data Pointer. DPH will hold 24H, while DPL will hold 76H.

Note : This is the only instruction which moves 16 bits of data at once.

1.28.3 MOVC A, @A+ <base register>

SPPU - May 12, Dec. 13

University Questions

Q. Explain following instruction : MOVC **(May 2012, 2 Marks)**

Q. Explain : MOVC A, @A + DPTR **(Dec. 2013, 2 Marks)**

Mnemonic	MOVC A, @A + <base register>	**Function**	Move code byte.
Machine cycles	2	**Clock Pulses**	24
Bytes	1	**Algorithm**	A = ((A) + ((DPTR)) OR (PC) = (PC) + 1, A = (A) + (PC)
Addr. Mode	Indirect addressing mode	**Flags**	No flags are affected

Operation MOVC A, @A + DPTR (A) ← ((A) + (DPTR)) OR MOVC A, @A + PC (PC) ← (PC) + 1 (A) ← (A) + (PC)	– This instruction will load the accumulator with a code byte or constant from the program memory. – Hence, it is essential to generate 16 bit address, so that data can be fetched from that address. The address of the byte fetched is the sum of the original unsigned eight-bit accumulator contents and the contents of a sixteen bit base register. The 16 bit base register may be the Data pointer or the program counter (PC). – If the base register used is PC, then the PC is incremented to the address of the following instruction i.e. next instruction before being added with the Accumulator, otherwise the base register remains unaltered.

Example :

(i) MOVC A, @A + DPTR

Let (DPTR) = 1000 H, (A) = 8H. Contents of memory location (1008H) = 22H. This instruction will copy the contents of address A + DPTR → 8H+ 1000H → 1008H to the accumulator i.e. after execution of this instruction (A) = 22 H.

(ii) MOVC A, @A + PC

Let (PC) = 4000 H and (A) = 50 H.

Let contents of memory location 4051H = 52H.

Initially the 16 bit address is computed.

(PC) = (PC) + 1 (PC) = 4000 H + 1H

(PC) = 4001 H (A) + (PC) → 50 H + 4001H → 4051 H.

This instruction will copy the contents of memory location 4051H i.e. 52H to the accumulator.

Note : The DPTR and PC remain unchanged, the accumulator contains the code byte fetched from program memory.

1.28.4 MOVX <dest-byte>, <src-byte>

SPPU - May 12, Oct. 16

University Questions

Q. Explain following instruction : MOVX **(May 2012, 2 Marks)**

Q. Explain MOVX instruction. **(Oct. 2016 (In Sem.), 2 Marks)**

Mnemonic	MOVX <dest-byte>, <src-byte>	**Function**	Move External
Machine cycles	2	**Clock Pulses**	24
Bytes	1	**Algorithm**	dest-byte = src-byte
Addr. Mode	register indirect addressing mode.	**Flags**	No flags are affected.

Operation MOVX <dest-byte>, <src-byte>	– The MOVX instructions transfer data between the accumulator and a byte of external data memory, hence "X" is appended to MOV. – Depending on whether the indirect address provided to the external data RAM is eight bit or 16 bit, there are two types of instructions. – In the first type, the contents of register R0 or R1 of current register bank provide an 8 bit address that is multiplexed with data on port 0. Eight bits are sufficient for external I/O expansion decoding or for a relatively small RAM array. – In the second type, the Data Pointer is used for generating 16 bit address. This is done for larger RAM array. Port 2 outputs the high-order address bits (the contents of DPH), while Port 0 output the low-order address bits (contents of DPL) multiplexed with data.

Example

1) MOVX A, @R0	This instruction will copy the data from the 8 bit address pointed by register R0 of the selected register bank to the accumulator.
2) MOVX A, @DPTR :	This instruction will copy the contents of external data memory location, pointed by DPTR to the accumulator.
3) MOVX @Ri, A :	This instruction will copy the contents of accumulator to the external data memory location pointed by register Ri of selected register bank.
4) MOVX @DPTR, A :	This instruction will copy the contents of accumulator to the 16 bit address, pointed by DPTR.

1.28.5 PUSH <direct>

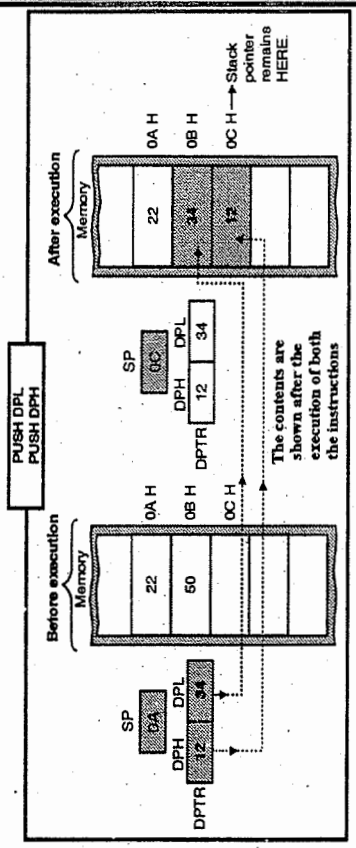
SPPU - May 13

University Question

Q. Explain following instruction : PUSH **(May 2013, 2 Marks)**

Mnemonic	PUSH <direct>	**Function**	Push onto stack
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	(SP) = (SP) + 1 ((SP)) = direct
Addr. Mode	direct addressing mode	**Flags**	No flags are affected.

Operation	PUSH (SP) ← (SP) + 1 ((SP)) ← direct	– This instruction copies the data from the source address onto the stack. – The stack pointer (SP) is incremented by 1 before the data is copied to the internal RAM location addressed by the stack pointer. – The stack will grow up in memory as data is pushed onto the stack. If the stack exceeds 7F H (i.e. top of internal RAM), then it results in errors.
Example	PUSH DPL PUSH DPH	Let SP = 0AH and Data Pointer=1234 H. The first instruction PUSH DPL will set the SP=0BH and store 34H in internal Ram location 0BH. The second instruction PUSH DPH will set the SP = 0CH and store 12H in internal RAM location 0BH. The stack pointer will remain at 0CH. Fig. 1.28.1 shows this.



m(21.2) Fig. 1.28.1

1.28.6 POP <direct>

SPPU - May 12

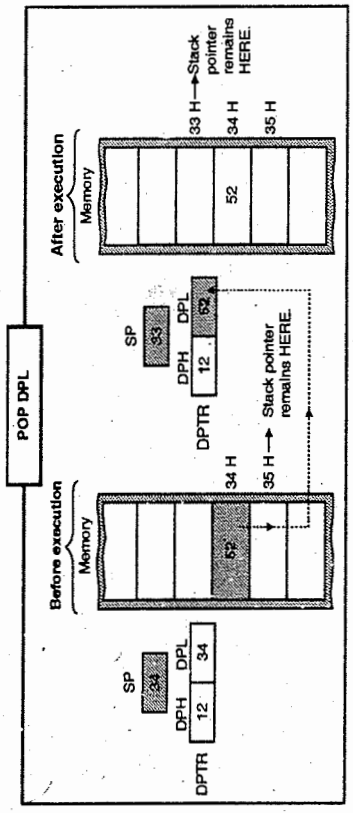
University Question

Q. Explain following instruction : POP **(May 2012, 2 Marks)**

Mnemonic	POP <direct>	**Function**	Pop from the stack
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	direct = ((SP)) (SP) = (SP) – 1
Addr. Mode	Direct addressing mode	**Flags**	No flags are affected.

Operation	POP direct (direct) ← ((SP)) (SP) ← (SP) – 1	– This instruction copies data from the stack to the destination location. – The SP is decremented by 1 after data is copied from the stack. RAM address to the direct destination address. This is done to ensure that the data placed on the stack is retrieved in the same order as it was stored.
Example	POP DPL	Let SP = 34H and data at internal RAM locations 34H be 52H. The instruction POP DPL will make the stack pointer to value 33H and DPL = 52 H. Fig. 1.28.2 shows this.

Note : The Push and Pop operation can Push /Pop a single byte only



m(21.3) Fig. 1.28.2

1.28.7 XCH A, <byte variable>

Mnemonic : XCH A, <byte variable >	**Function :** Exchange accumulator with byte variable.
Algorithm : (A) = (byte variable) (byte variable) = (A)	**Operation :** (A) ↔ (Byte variable)

Operation

- This instruction will load the accumulator with the contents of byte variable. At the same time the original accumulator contents are written to the byte variable.
- The source / destination operand can use register, direct, or register indirect addressing. Let us see the different combinations depending on the different addressing modes.

1) XCH A, Rn

Operation	(A) ↔ (Rn)	This instruction will exchange the contents of accumulator with the contents of register Rn of selected register bank.
Example	XCH A, R1	This instruction will load the contents of register R1 of selected register bank in the accumulator and at the same time the contents of original accumulator will be copied in register R1.

2) XCH A, direct

Operation	(A) ↔ (direct)	This instruction will exchange the contents of accumulator with the direct address.
Example	XCH A, 10H	This instruction will load the contents of memory location whose address is 10H to the accumulator and at the same time the contents of accumulator are transferred to memory location whose address is 10H.

3) XCH A, @Ri

Operation	XCH (A) ↔ ((Ri))	This instruction will exchange the contents of accumulator with the contents of memory pointed by register Ri i.e. contents of memory location pointed by Ri will be transferred to accumulator and at the same time contents of the accumulator are transferred to memory location pointed by Ri.
Example	XCH A, @R0	This instruction will load the contents of memory location pointed by register R0 of selected register bank to the accumulator and at the same time the contents of accumulator are copied to the memory location pointed by R0 register of the selected register bank.

1.28.8 XCHD A, @Ri

Mnemonic	XCHD @Ri
Machine cycles	1
Bytes	1
Addr. Mode	Register indirect addressing mode

Function	Exchange Digit
Clock Pulses	12
Algorithm	$(A_{(3-0)}) = ((Ri_{(3-0)}))$
Flags	No flags are affected.

Operation	XCHD $(A)_{(3-0)} \leftrightarrow ((Ri_{(3-0)}))$	– This instruction exchanges the lower nibble of accumulator (bits 3-0) with the lower nibble of the memory location indirectly addressed by the specified register R0/R1 of the specified register bank. – The upper nibble (bits 7-4) of each register remain unchanged.
Example	XCHD A, @R0	Let R0 contain address 37 H, accumulator contain 25H, and the internal RAM location 37H contain 47H, then the instruction XCHD A, @R0 will leave RAM location 37H holding the value 45H and accumulator holding the value 27H

1.29 Arithmetic Instructions

1.29.1 ADD A, <src-byte>

Mnemonic : ADD A, <src-byte>	**Function :** Add
Algorithm : (A) = (A) + <src-byte>	**Operation :** (A) ← (A) + <src-byte>

- This instruction adds the byte variable indicated to the accumulator. The result is contained in the accumulator.
- All the addressing modes can be used for source : an immediate number, a register, direct address, indirect address.
- Depending on the addressing modes, let us see the different combinations :

1) ADD A, Rn

Operation	(A) ← (A) + (Rn)	This instruction will add the byte in register Rn of the selected register bank with the byte in accumulator. The result is contained in the accumulator
Example	ADD A, R0	Let A = 42H (0100 0010 B) and R0 = 91 H (1001 0001B). then, ADD A, R0 will leave A = D3H (1101 0011 B) with the AC flag cleared, carry flag and overflow flag also cleared.

2) ADD A, direct

Operation	(A) ← (A) + (direct)	This instruction will add the contents of the memory to location whose direct address is specified in the instruction with the accumulator contents. The result of addition will be stored in the accumulator
Example	ADD A, 20H	Let the contents at memory location 20H be 45H (0100 0101B) and contents of accumulator be 77H (0111 0111B), then the instruction ADD A, 20H will leave. A = BC H (1011 1100 B) with auxiliary flag cleared, carry flag cleared and overflow flag is set.

3) ADD A, @Ri

Operation	ADD (A) ← (A) + ((Ri))	This instruction will add the contents of memory location whose address is pointed by register Ri of the selected register bank with contents of the accumulator. The result of addition is stored in the accumulator.
Example	ADD A, @R0	The instruction ADD A, @R0 will add the contents of memory location pointed by register R0 with the contents of accumulator. The result stored in the accumulator.

4) ADD A, #data

Operation	ADD (A) ← (A) + data	This instruction will add the immediate 8 bit data with data in the accumulator. The result of addition is stored in the accumulator.
Example	ADD A, #40H	The instruction ADD A, #40H will add the data 40H to contents of accumulator. The result is stored in accumulator.

1.29.2 ADDC A, <src-byte>

Mnemonic : ADDC A ,<src-byte>	**Function :** Add with carry
Algorithm : (A) = (A) + <src-byte> + carry	**Operation :** (A) ← (A) + <src-byte> + carry

- This instruction will add the byte variable indicated, the carry flag and the accumulator contents. The result of addition is stored in the accumulator.
- All the addressing modes can be used for source: an immediate number, a register, direct address, indirect address.
- Depending on the addressing modes, let us see the different combinations.

1. ADDC A, Rn

SPPU - Oct. 16

University Question

Q. Explain ADDC A, B. **(Oct. 2016 (In Sem.), 2 Marks)**

Operation	(A) ←(A) + (Rn) + (CY)	This instruction will add the contents of accumulator with the contents of register Rn of the selected register bank and carry flag. The result of addition is stored in accumulator
Example	i. ADDC A, R1	The instruction ADDC A, R1 will add the contents of accumulator with the contents of register R1 of the selected register bank and carry flag. The result is stored in accumulator.
	ii. ADDC A, B	The instruction ADDC A, B will add the contents of accumulator with the contents of register B and carry flag. The result is stored in accumulator.

2. ADDC A, direct

Operation	(A) ← (A)+(direct) + CY	This instruction will add the contents of memory location whose direct address is specified in the instruction with the contents of accumulator and carry. The result of addition is stored in the accumulator
Example	ADDC A, 10 H.	The instruction ADDC A, 10H will and the contents of accumulator, memory location whose address is 10H and the carry flag and stores the result in A.

3. ADDC A, @Ri

Operation	(A) ← (A) + ((Ri)) + CY.	– This instruction will add the contents of memory location pointed by register Ri of selected register bank with the accumulator and carry flag. – The result of addition is stored in the accumulator.

Example	ADDC A, @R0.	– The instruction ADDC A, @R0 will add the contents of A, memory location whose address is given by register R0 and the carry flag and stores the result in accumulator.

4. ADDC A, #data

Operation	(A) ← (A) + data + CY.	This instruction will add the contents of accumulator with immediate data specified in the instruction along with carry.
Example	ADDC A, #40H.	Let A = 50H, CY = 1 then the instruction ADDC A, #40H will leave the accumulator with 91H and the carry, overflow and auxiliary flags are cleared.

1.29.3 SUBB A, <src-byte>

Mnemonic : SUBB A, <src-byte>	**Function :** Subtract with borrow.
Algorithm : (A) = (A) – <src-byte> – CY	**Operation :** (A) ← (A) – <src-byte> – CY

This instruction subtracts the indicated byte variable and the carry flag contents together, from the accumulator. The result is stored in the accumulator.

Note : The carry flag is treated as the borrow flag.

- Carry (borrow) flag is set if a borrow is needed for bit 7 and clears CY otherwise.
- Auxiliary carry flag is set if a borrow is needed for bit 3 otherwise it is cleared.
- The overflow flag is set if a borrow is needed for bit 6, but not into bit 7 or if a borrow is needed into bit 7, but not bit 6.
- When signed integers are subtracted the overflow flag indicates a negative number produced when negative value is subtracted from a positive value, or a positive number produced when a positive number is subtracted from a negative number.
- All the addressing modes can be used as source : an immediate number, a register, direct address, indirect address.
- Depending on the addressing-modes, let us see the different combinations.

1. SUBB A, Rn

Operation	SUBB. (A) ← (A) – (Rn) – CY	This instruction will subtract the contents of register Rn of the current register bank and the contents of carry flag together, from the accumulator. The result is stored in the accumulator.
Example	SUBB A, R3	The instruction SUBB A, R3 will subtract the contents of register R3 of selected register bank and contents of carry flag from the accumulator. The result is stored in the accumulator.

2. SUBB A, direct

Operation	SUBB. (A) ← (A) – (direct) – CY.	This instruction will subtract the contents of memory location whose direct address is specified in the instruction and the contents of carry flag from the contents of accumulator. The result is stored in the accumulator.
Example	SUBB A, 45H.	This instruction subtracts the contents of memory location 45 H and carry from contents of accumulator and result is stored in accumulator.

3. SUBB A, @Ri

Operation	(A) ← (A) – ((Ri)) – CY	This instruction will subtract the contents of memory location pointed to by register Ri and contents of carry flag from the accumulator. The result of subtraction is stored in the accumulator.
Example	SUBB A, @R1.	Let R1 = 30H and the contents of memory location 30H be 54H (0101 0100 B), and contents of accumulator = C9H (1100 1001 B) and CY (borrow) = 1, then the instruction SUBB A, @R1 will leave the accumulator with 74H with the carry and auxiliary carry flag cleared and the overflow flag set.

4. SUBB A, #data

Operation	SUBB (A) ← (A) – data – CY.	This instruction will subtract the data specified in the instruction and contents of carry flag from the contents of accumulator. The result will be stored in the accumulator.
Example	SUBB A, #40H.	Let A = 50H, CY = 1 then the instruction SUBB A, #40H will leave the accumulator with 0FH.

1.29.4 INC <byte>

Mnemonic : INC <byte>	**Function :** Increment
Algorithm : <byte> = <byte> + 1	**Operation :** byte ← byte + 1.

- This instruction will increment the indicated variable by 1.
- If the byte value is FFH and if it is incremented, then the result will overflow to 00H.
- It supports three addressing modes, Register, direct and register-indirect.

Note : When the increment instruction operates on a port direct address, alter the latch of that port. (Since it is a read-modify-write operation)

Depending on different addressing modes let us see the different combinations.

1. INC Rn

Operation	(Rn) ← (Rn) + 1.	– This instruction will increment the contents of register Rn of selected register bank by 1. – The register Rn can also be accumulator.
Example	INC R5.	Let R5 = 0EH then instruction INC R5 will leave R5 = 0FH.

2. INC <direct>

Operation	<direct> ← <direct> + 1.	This instruction will increment the contents of memory location whose direct address is specified in the instruction by 1.
Example	INC 40H.	Let the contents of memory location 40H be 22H, then the instruction INC 40H will increment the contents of memory location whose direct address is 40H by 1. i.e. location 40H = 23H.

3. INC @Ri

Operation	((Ri)) ← ((Ri)) + 1.	This instruction will increment the contents of memory location that is pointed by register Ri by 1.
Example	INC @R1.	Let the contents of R1 = 45H and contents of memory location 45H = 56H, then the instruction INC @R1 will increment the contents of memory location pointed by register R1 i.e. 45H by 1. i.e. now memory location 45H will contain 57H

1.29.5 INC DPTR

Operation	(DPTR) ← (DPTR) + 1	– This instruction will opincrement the contents of Data Pointer by 1. – A 16 bit increment is performed. An overflow of the low-order byte of the data pointer (DPL) from 0FF H to 00 H will increment the high order byte (DPH). – DPTR is the only 16 bit register that is incremented.
Example	INC DPTR	Let contents of DPH = 15 H and contents of DPL = FFH. The instruction INC DPTR will cause DPH = 16H and DPL = 00H.

1.29.6 DEC <byte>

Mnemonic : DEC <byte>.	**Function :** Decrement.
Algorithm : <byte> = <byte> – 1. <byte> ← <byte> – 1.	**Operation :** DEC

- This instruction will decrement the indicated variable by 1.
- An original value of 00H will underflow to FFH.
- Three operand addressing modes are allowed register, direct and register indirect addressing modes.

Note : The decrement instruction when used to modify an output port, alters the latch for that port. Depending on different addressing mode, let us see the different combinations.

1. DEC Rn

Operation	(Rn) ← (Rn) – 1.	– This instruction will decrement the contents of register Rn of the selected register bank by 1. – The registers can be any of the registers R0 – R7 of the selected register bank or the accumulator.
Example	DEC R5	Let R5 = 0EH, DEC R5 will leave H5 = 0DH

2. DEC <direct>

Operation	<direct> ← <direct> – 1.	This instruction will decrement the contents of memory location whose direct address is specified in the instruction by 1.
Example	DEC 55H	Let the contents of memory location 55H = 14H, then the instruction DEC 55H will decrement the contents of memory location 55H by 1 i.e., contents of memory location 55H = 13H.

3. DEC @Ri

Operation	((Ri)) ← ((Ri)) – 1.	This instruction will decrement the contents of memory location pointed by register Ri by 1.
Example	DEC @R0.	Let R0 = 51H and contents of memory location 51 = 1AH, then the instruction DEC @R0 will leave the memory location 51H = 19H.

1.29.7 MUL AB

Mnemonic	MUL AB.
Machine cycles	4
Bytes	1
Addr. Mode	Register Addressing mode.

Function	Multiply.
Clock Pulses	48
Algorithm	$(A)_{7-0} = (A) \times (B)$ $(B)_{15-8}$
Flags	Flags are affected.

Operation	MUL $(A)_{7-0} \leftarrow (A) \times (B)$ $(B)_{15-8}$	– This instruction multiplies an eight bit unsigned integer in the Accumulator and the B register. The low-order byte of the sixteen bit product is left in the accumulator, and the high-order byte in B. – The largest possible product is FE01 H when A = FFH and B = FFH. Then A = 01H and B = FEH after multiplication.
Example	MUL AB.	Let A = 50H, B = A0H MUL AB (50H) × (A0H) = (3200H) B = 32H, A = 00H with the overflow flag set and carry flag cleared.

Note : (1) There is no comma between A and B in the MUL instruction.
(2) Original contents of registers A and B are lost.

– The overflow flag is set if the product is greater than 255 decimal (FFH), otherwise it is cleared.
– The carry flag is always cleared.

1.29.8 DIV AB

Mnemonic	DIV AB
Machine cycles	4
Bytes	1
Addr. Mode	Register Addressing mode.

Function	Divide.
Clock Pulses	48
Algorithm	(A ÷ B) A = quotient B = Remainder.
Flags	Flags are affected.

Operation	DIV A (quotient) ← A ÷ B B(remainder)	– This instruction divides the unsigned number in accumulator with the unsigned number in register B. – Accumulator contains the quotient of the result and register B contains the remainder. – The contents of A and B, when division by 0 is attempted, are undefined.

Example	DIV AB	Let A = FBH and B = 12H then DIV AB will result A = $(13)_{10}$ = 0DH (quotient), B = $(17)_{10}$ = 11H (remainder)

Note : (1) The original contents of A and B are lost. The registers A and B are used as source and destination for the division operation.
(2) There is no comma between A and B in the DIV instruction.

- The carry and overflow flags are always cleared. But if, A contains some number, B contains 00H. Then if an attempt is done DIV AB then the values contained in register A and B are undefined. The overflow flag will be set and carry flag will be cleared. i.e. overflow flag is set when a divide by zero is attempted.

1.29.9 DA A

Mnemonic	DA A	**Function**	Decimal Adjust Accumulator for Addition.
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	Contents of Accumulator are BCD if $[(A_{3-0}) > 9]$ or [AC = 1] then $(A_{3-0}) = (A_{3-0}) + 6$AND if $[(A_{7-4}) > 9]$ or [CY = 1] then $(A_{7-4}) = (A_{7-4}) + 6$
Addr. Mode	Register Specific Addressing mode.	**Flags**	Flags are affected.

Operation	DA A	– This instruction adjusts the sum of two packed BCD numbers to an eight bit value i.e. producing two four bit digits. – To perform the addition any ADD or ADDC instruction can be used. – The rules of BCD addition are (i) if the number is greater than 9, add 6 (ii) if the auxiliary carry or carry is generated, add 6. This is done in order to produce a valid BCD result.

Example	DA A	Let A = 56H (packed BCD) R3 = 67H (packed BCD) CY = 1 then, ADDC A, R3 A → 56H R3 → +67H CY → +1 BE H DA A As B > 9, E > 9, 6 should be added to both BEH + 66H 24 with CY = 1.

- DA A instruction is used for performing decimal addition using the instruction ADD or ADDC. The eight bit binary result that is present in the accumulator is adjusted to valid BCD to form two BCD digits of four bits each.
- If $(A_{3-0} > 9)$ or AC = 1 then 6 is added to accumulator, so that it produces a proper BCD digit in the lower nibble. This internal addition may set the auxiliary carry flag if there is a carry-out of the bit 3, propagating into higher order bits.
- If $(A_{7-4} > 9)$ or CY = 1 then 6 is added to produce proper BCD digit in the high-order nibble. If there is carry-out of high-order bits then carry flag will again be set.
- The carry flag if set thus, indicates whether the sum of two BCD numbers is greater than 99, allowing multiple precision decimal addition.
- The overflow flag is not affected.

Note : (1) All the above conversions are done in one instruction cycle.
(2) DA A performs decimal-conversions by adding 00H, 06H, 60H or 66H to the accumulator, depending on the accumulator and PSW (i.e. flags) conditions.
(3) DA A cannot simply convert a hexadecimal number in the accumulator to BCD notation.
(4) DA A does not apply to decimal subtraction.

1.30 Logical Instructions

1.30.1 ANL <dest-byte>, <src-byte>

Mnemonic : ANL <dest-byte>, <src-byte>	**Function :** Logical AND for byte variables
Algorithm : <dest-byte> = <dest-byte> ∧ <src-byte>	**Operation :** <dest-byte> ← <dest-byte> ∧ <src-byte>

- This instruction performs bit wise logical AND operation between the destination and the source byte. The result is stored at the destination byte.
- The source and destination support four addressing modes : register, direct, register-indirect and immediate addressing modes. These 4 addressing modes support six combinations.
- Let us see these combinations.

Note : When this instruction is used to modify an output port, the value used as original port data will be read from the output data latch, not the input pins, as it will be a read-modify-write instruction.

1. ANL A, Rn

Operation	ANL (A) ← (A) ∧ (Rn)	This instruction will perform bitwise logical AND operation between the contents of accumulator and register Rn of the selected register bank. The result will be stored in the accumulator.
Example	ANL A, R5	Let A = FFH, R5 = 15H, then the instruction ANL A, R5 will logically AND contents of accumulator with the contents of register R5. The result of ANDing is stored in accumulator i.e. 15H.

2. ANL A, Direct

Operation	ANL (A) ← (A) ∧ (direct)	This instruction will bitwise logically AND the contents of accumulator with the contents of memory location whose direct address is specified in the instruction. The result will be stored in the accumulator.
Example	ANL A, 50H.	Let A = 10H, contents of memory location 50H be 80H, then the instruction ANL A, 50H will logically AND the contents of accumulator bitwise with the contents of memory location 50H. The result in accumulator will be 00H.

3. ANL A, @Ri

Operation	ANL (A) ← (A) ∧ ((Ri))	This instruction will bitwise logically AND the contents of accumulator with the contents of memory location pointed by register Ri of the selected register bank. The result will be stored in the accumulator.
Example	ANL A, @R1.	Let A = 40H and R1 = 0AH, contents of memory location 0AH = CAH, then the instruction ANL A, @R1 will bitwise AND the contents of accumulator with the contents of memory location pointed by register R1 of the selected register bank i.e. CAH. The result in accumulator will be 40H.

4. ANL Direct, A

Operation	ANL (direct) ← (direct) ∧ A	This instruction will bitwise logically AND the contents of memory location whose direct address is specified in the instruction with the contents of the accumulator. The result will be stored in the memory location whose direct address is specified in the instruction.
Example	ANL 30H, A.	The instruction ANL 30H, A will bitwise logically AND the contents of memory location whose direct address is 30H with the contents of accumulator. The result is stored in memory location 30H.

5. ANL A, #data

SPPU - Dec. 13

University Question

Q. Explain ANL A, # data **(Dec. 2013, 2 Marks)**

Operation	ANL (A) ← (A) ∧ (data)	This instruction will bitwise logically AND the contents of accumulator with the immediate data specified in the instruction. The result will be stored in the accumulator.
Example	ANL A, #57H.	Let A = 22H (0010 0010 B), then the instruction ANL A,#57H will bitwise logically AND the contents of accumulator 22H with immediate data 57H. The result in accumulator is 02H.

6. ANL Direct, #data

Operation	ANL (direct) ← (direct) ∧ data.	This instruction will bitwise logically AND the contents of memory location whose direct address is specified in the instruction with the immediate data. The result will be stored in the memory location whose direct address is specified.

Example	ANL 54H, #33H	Let the contents of memory location 54H be 25H. The instruction ANL 54H, #33H will logically AND the contents of memory location 54H i.e. 25H with immediate data i.e. 33H. The result will be stored in the memory location 54H i.e. 21H.

1.30.2 ORL <dest-byte>, <src-byte>

Mnemonic : ORL <dest-byte>, <src-byte>	**Function :** Logical OR for byte variables
Algorithm : <dest-byte> = <dest-byte> ∨ <src-byte>	**Operation :** ORL <dest-byte> ← <dest-byte> ∨ <src-byte>

- This instruction will bitwise logically OR the contents of source byte with the contents of the destination byte. The result of logical ORing operation will be stored in the destination byte.
- The source and destination support four addressing modes : register, direct, register-indirect and immediate addressing modes. These four addressing modes support six combinations. Let us see these combinations.

Note : When this instruction is used to modify an output port, the value used as original port data will be read from the output data latch, not the input pins. Since, it is a read-modify-write operation.

1. ORL A, Rn

Operation	ORL (A) ← (A) ∨ (Rn)	This instruction will perform bitwise logical OR operation between the contents of accumulator and the register Rn of the selected register bank. The result will be stored in the accumulator
Example	ORL A, R5	Let A = FFH, R5 = 15H, then the instruction ORL A, R5 will logically OR the contents of accumulator with the contents of register R5. The result of ORing stored in the accumulator is FFH.

2. ORL A, Direct

Operation	ORL (A) ← (A) ∨ (Direct)	This instruction will perform bitwise logical OR operation between the contents of accumulator and the contents of memory location whose direct address is specified in the instruction. The result will be stored in the accumulator.
Example	ORL A, 50H.	Let A = 10H, contents of memory location 50H = 80H. The instruction ORL A, 50H will bitwise logically OR the contents of accumulator 10H with contents of memory location 50H i.e. 80H. The result stored in the accumulator will be 90H.

3. ORL A, @Ri

Operation	ORL (A)←(A) ∨ ((Ri))	This instruction will bitwise logically OR the contents of accumulator with the contents of memory location pointed by register Ri of the selected register bank. The result will be stored in the accumulator.
Example	ORL A, @R1	Let A = 40H, and R1 = 0AH. Let the contents of memory location 0AH = CAH. The instruction ORL A, @R1 will bitwise logically OR the contents of accumulator (40H) with the contents of memory location pointed by register R1 i.e. CAH. The result in accumulator will be CAH.

4. ORL Direct, A

Operation	ORL (Direct) ← (Direct) ∨ (A)	This instruction will bitwise logically OR the contents of memory location whose direct address is specified in the instruction with the contents of the accumulator. The result will be stored in the memory location whose direct address is specified in the instruction.
Example	ORL 30H, A	Let the contents of memory location 30H = 57H, contents of accumulator = 64H. The instruction ORL 30H, A, will bitwise logically OR the contents of memory location whose address is 30H i.e. 57H with the contents of accumulator (64H). The result in memory location 30H will be 77H.

5. ORL A, #data

Operation	ORL (A) ← (A) ∨ data	This instruction will bitwise logically OR the contents of accumulator with the immediate data specified in the instruction. The result will be stored in the accumulator.
Example	ORL A, #57H	Let A = 22H. The instruction ORL A, #57H will bitwise logically OR the contents of accumulator 22H with the immediate data 57H. The result in accumulator will be 77H.

6. ORL Direct, #data

Operation	ORL (Direct) ← (Direct) ∨ data	This instruction will bitwise logically OR the contents of memory location whose direct address is specified in the instruction with the immediate data. The result will be stored in the memory location whose direct address is specified.
Example	ORL 54H, #33H	Let the contents of memory location 54H be 25H. The instruction ORL 54H, #33H will logically OR the contents of memory location 54H i.e. 25H with immediate data i.e. 33H. The result stored in memory location 54H is 37H.

1.30.3 XRL <dest-byte>, <src-byte>

Mnemonic : XRL <dest-byte>, <src-byte>	**Function :** Logical Exclusive-OR for byte variables
Algorithm : <dest-byte> = <dest-byte> ⊕ <src-byte>	**Operation :** XRL <dest-byte> ← <dest-byte> ⊕ <src-byte>

- This instruction performs bitwise Exclusive-OR operation between the destination byte and the source byte. The result of operation will be stored in the destination byte.
- The source and destination support four addressing : register, direct, register-indirect and immediate addressing modes. The four addressing modes support six combinations. Let us see these combinations.

Note : When this instruction, is used to modify an output port, the value used as original port data will be read from the output data latch, not the input pins. Since it is read-modify-write operation.

1. XRL A, Rn

Operation	XRL (A) ← (A) ⊕ (Rn)	This instruction will perform bitwise logical Exclusive-OR operation between the contents of accumulator and the contents of register Rn of the selected register bank. The result will be stored in the accumulator.
Example	XRL A, R5	Let A = FFH, R5 = 15H, then the instruction XRL A, R5 will logically EX–OR the contents of accumulator with the contents of register R5. The result stored in accumulator is EAH.

2. XRL A, Direct

Operation	XRL (A) ← (A) ⊕ (Direct)	This instruction will perform bitwise logical EX-OR operation between the contents of accumulator and the contents of memory location whose direct address is specified in the instruction. The result will be stored in the accumulator.
Example	XRL A, 50H	Let A = 10H, contents of memory location 50H = 80H. The instruction XRL A, 50H will bitwise logically EX-OR the contents of accumulator 10H with the contents of memory location 50H i.e. 80H. The result stored in the accumulator will be 90H.

3. XRL A, @Ri

Operation	XRL (A) ← (A) ⊕ ((Ri))	This instruction will bitwise logically EX-OR the contents of accumulator with the contents of memory location pointed by register Ri of the selected register bank. The result will be stored in the accumulator.
Example	XRL A, @R1	Let A = 40H and R1 = 0AH. Let the contents of memory location 0A = CAH. The instruction XRL A, @R1 will bitwise logically EX-OR the contents of accumulator (40H) with the contents of memory location pointed by register R1 i.e. CAH. The result stored in the accumulator will be 8AH.

4. XRL Direct, A

Operation	XRL (Direct) ← (Direct) ⊕ (A)	This instruction will bitwise logically EX-OR the contents of memory location whose direct address is specified in the instruction with the contents of the accumulator. The result will be stored in the memory location whose direct address is specified in the instruction.
Example	XRL 30H, A	Let contents of memory location 30H = 57H, A = 64H. The instruction XRL 30H, A will bitwise logically EX-OR the contents of memory location whose address is 30H i.e. 57H with the contents of accumulator (64H). The result in memory location 30H will be 33H.

5. XRL A, #data

Operation	XRL (A) ← (A) ⊕ Data	This instruction will bitwise logically EX-OR the contents of accumulator with the immediate data specified in the instruction. The result will be stored in the accumulator.
Example	XRL A, #57H	Let A = 22H The instruction XRL A, #57H will bitwise logically EX-OR the contents of accumulator 22H with the immediate data 57H. The result in accumulator will be 75H.

6. XRL Direct, #data

Operation	XRL (Direct) ← (Direct) ⊕ Data	This instruction will bitwise logically EX-OR the contents of memory location whose direct address is specified in the instruction with the immediate data. The result will be stored in the memory location whose direct address is specified.
Example	XRL 54H, #33H	Let the contents of memory location 54H be 25H. The instruction XRL 54H, #33H will logically EX-OR the contents of memory location 54H i.e. 25H with immediate data i.e. 33H. The result stored in memory location 54H is 16H.

1.30.4 CLR A

Mnemonic	CLR A	**Function**	Clear Accumulator
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	A = 0
Addr. Mode	Register specific addressing mode.	**Flags**	No flags are affected

Operation	CLR A ← 0	This instruction will clear all the bits of accumulator to zero.
Example	CLR A	Let A = 77H. The instruction CLR A will leave the accumulator set to 00H.

1.30.5 CPL A

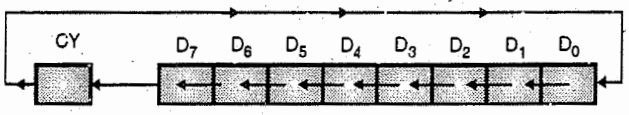
Mnemonic	CPL A	**Function**	Complement Accumulator
Machine cycles	1	**Clock Pulses**	12
Bytes	1	**Algorithm**	$A = \bar{A}$
Addr. Mode	Register specific addressing mode	**Flags**	No flags are affected

Operation	CPL $A \leftarrow \bar{A}$	– This instruction will complement all the bits of the accumulator i.e. 1's complement of the number in accumulator is taken. – The bits which previously contained one are changed to zero and vice versa.
Example	CPL A	Let A = 57H (0101 0111 B). The instruction CPL A will complement all the bits in accumulator. So, now the accumulator will contain (1010 1000 B) A8H.

1.30.6 RL A

Mnemonic	RL A	Function	Rotate Accumulator left
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	$(A_{n+1}) = (A_n)$ where n = 0 to 6 $(A_0) = (A_7)$
Addr. Mode	Register Specific Addressing mode	Flags	No flags are affected

Operation	RL $(A_{n+1}) \leftarrow (A_n)$ where n = 0 to 6 $(A_0) \leftarrow A_7$	– This instruction will rotate the eight bits in the accumulator by one bit to the left. – Bit 7 is rotated into the Bit 0 position.
Example	RL A	Let A = 58H (0101 1000 B). The instruction RLA will rotate the accumulator by one bit to the left. So now accumulator contains (1011 0000 B) i.e. B0H



m(21.4)**Fig. 1.30.1 : RL A**

1.30.7 RLC A

SPPU - May 13

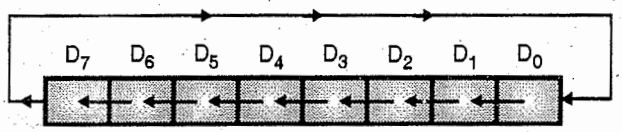
University Question

Q. Explain following instruction : RLC A.

(May 2013, 2 Marks)

Mnemonic	RLC A	Function	Rotate Accumulator Left through Carry
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	$(A_{n+1}) = (A_n)$ where n = 0 – 6 $(A_0) = (CY)$ $(CY) = A_7$
Addr. Mode	Register specific addressing mode	Flags	Except carry, no other flags are affected

Operation	RLC $(A_{n+1}) \leftarrow (A_n)$ where n = 0 – 6 $(A_0) \leftarrow (CY)$ $(CY) \leftarrow (A_7)$	– This instruction will rotate the eight bits in the accumulator and the carry flag together by one bit to the left. – Bit 7 will move into carry and original carry will move to Bit 0 position. Fig. 1.30.2 shows this
Example	RLC A	Let A = 72H (0111 0010 B), and CY = 1. The instruction RLC A leaves the accumulator holding the value (1110 0101 B) i.e. E5 H and CY = 0

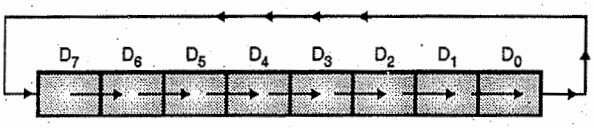


m(21.5)**Fig. 1.30.2 : RLC A**

1.30.8 RR A

Mnemonic	RR A	Function	Rotate Accumulator Right
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	$(A_n) = (A_{n+1})$ where n = 0 to 6 $(A_7) = (A_0)$
Addr. Mode	Register Specific Addressing mode	Flags	No flags are affected

Operation	RR $(A_n) \leftarrow (A_{n+1})$ where n = 0 to 6 $(A_7) \leftarrow A_0$	– This instruction will rotate the eight bits in the accumulator by one position to the right. – Bit 0 is rotated into bit 7 position. – Fig. 1.30.3 shows this.
Example	RR A	Let A = 75H (0111 0101 B). The instruction RR A will leave the accumulator holding value BAH (1011 1010 B)

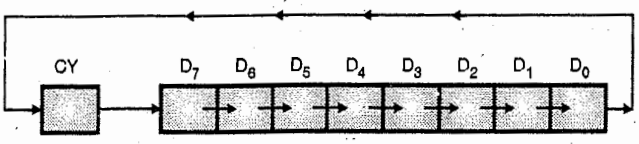


m(21.6)**Fig. 1.30.3 : RR A**

1.30.9 RRC A

Mnemonic	RRC A	Function	Rotate Accumulator Right through carry
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	$(A_n) = (A_{n+1})$ where n = 0 to 6 $(A_7) = (CY)$ $(CY) = A_0$
Addr. Mode	Register Specific Addressing mode	Flags	Except carry, no other flags are affected

Operation	RRC $(A_n) \leftarrow (A_{n+1})$ where n = 0 to 6 $(A_7) \leftarrow (CY)$ $(CY) \leftarrow A_0$.	– This instruction will rotate the eight bits in accumulator and the carry flag together by one bit position to the right. – Bit 0 moves into the carry flag, original carry flag contents move into bit 7. Fig. 1.30.4 shows this
Example	RRC A	Let A = 85H (1000 0101 B), and CY = 1 then the instruction RRC A will leave the accumulator holding C2H (1100 0010) and CY = 1.



m(21.7)**Fig. 1.30.4 : RRC A**

1.30.10 SWAP A

Mnemonic	SWAP A	Function	Swap nibbles within the Accumulator
Machine cycles	1	Clock Pulses	12
Bytes	1	Algorithm	$(A_{3-0}) = (A_{7-4})$ $(A_{7-4}) = (A_{3-0})$
Addr. Mode	Register Specific addressing mode	Flags	No flags are affected

Operation	SWAP $(A_{3-0}) \leftrightarrow (A_{7-4})$	– This instruction interchanges the low order and high order nibbles of the accumulator. – The operation can also be thought of as a four bit rotate instruction.
Example	SWAP A	Let A = 87H (1000 0111 B). The instruction SWAP A leaves the accumulator holding the value 78H (0111 1000 B).

1.31 Boolean Variable Manipulation Instructions

1.31.1 CLR Bit

Mnemonic	CLR bit	Function	Clear bit
Machine cycles	1	Clock Pulses	12
Bytes	1 if carry specific 2 if directly addressable bit.	Algorithm	(Bit) = 0
Addr. Mode	If operated on carry flag then register addressing mode otherwise direct addressing mode.	Flags	Except carry, no other flags are affected if it is carry specific. If it is direct no flags are affected.

Operation	CLR $(Bit) \leftarrow 0$	This instruction will clear the indicated bit. CLR can operate on carry flag or any directly addressable bit
Example	CLR P2.3	Let Port 2 has previously been written with ADH (1010 1101 B). The instruction CLR P2.3 will clear the 3rd bit of Port 2 leaving Port 2 with A5H (1010 0101 B)

1.31.2 SETB

Mnemonic	SETB	Function	Set Bit
Machine cycles	1	Clock Pulses	12
Bytes	1 if carry specific 2 if directly addressable bit.	Algorithm	(Bit) = 1
Addr. Mode	register addressing mode if carry specific otherwise direct addressing mode.	Flags	Except carry, no other flags are affected if it is carry specific. If it is direct no flags are affected.

Operation	SETB $(Bit) \leftarrow 1$	– This instruction will set the indicated bit. – SET can operate on carry flag or any directly addressable bit.
Example	SET P2.3	Let Port 2 has previously been written by A5H (1010 0101 B). The instruction SET P2.3 will set the 3rd bit of Port 2, leaving Port 2 with ADH (1010 1101 B).

1.31.3 CPL Bit

Mnemonic	CPL bit
Machine cycles	1
Bytes	1 if carry specific 2 if directly addressable bit
Addr. Mode	register addressing mode if carry specific otherwise direct addressing mode

Function	Complement bit
Clock Pulses	12
Algorithm	(Bit) = ($\overline{\text{Bit}}$)
Flags	Except carry, no other flags are affected if it is carry specific. If direct no flags are affected.

Operation	CPL (Bit) ← ($\overline{\text{Bit}}$)	– This instruction will complement the bit variable specified. A bit which had been a one is changed to zero and vice versa. – CPL can operate on carry flag or any directly addressable bit.
Example	CPL P2.5	Let Port 2 has previously been written by A5H (1010 0101 B). The instruction CPL P 2.5 will complement the 5th bit of Port 2. Port 2 will hold 85H (1000 0101 B)

1.31.4 ANL C, <src-bit>

Mnemonic	ANL C, <src-bit>
Machine cycles	2
Bytes	2
Addr. Mode	Direct Addressing mode

Function	Logical AND for bit variables
Clock Pulses	24
Algorithm	(C) = (C) ∧ (src-bit)
Flags	Except carry, no other flags are affected

Operation	ANL (C) ← (C) ∧ <src-bit>	– This instruction will logically AND the specified bit with the carry bit. The result is stored in the carry bit. – If the Boolean value of the source bit is zero, then this instruction will clear the carry flag, otherwise it will leave the carry flag in its current state. – A slash ("/") preceding the operand (i.e. ANL C, /<src-bit>) in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is unaffected.
Example	ANL C, ACC.4 ANL C, / ACC.4	– Let C = 1, A = 11H (0001 0001 B) then the instruction ANL C, ACC.4 will logically AND the contents of carry with accumulator Bit 4 leaving the C = 1. – The instruction ANL C,/ACC. 4 will logically AND the contents of carry with complement of accumulator Bit 4 leaving the carry C = 0.

1.31.5 ORL C, <src-bit>

Mnemonic	ORL C, <src-bit>
Machine cycles	2
Bytes	2
Addr. Mode	Direct Addressing mode

Function	Logical-OR for bit variables
Clock Pulses	24
Algorithm	(C) = (C) ∨ <src-bit>
Flags	Except carry, no other flags are affected

Operation	ORL (C) ← (C) ∨ <src-bit>	– This instruction will logically OR the specified bit with the carry bit. The result is stored in the carry bit. – If the Boolean value is a logical 1 then set the carry flag otherwise leave the carry flag in its current state. – A slash ("/") preceding the operand (i.e. ORL C, /<src-bit>) in the assembly language indicates that the logical complement of the addressed bit is used as the source value, but the source bit itself is unaffected.
Example	ORL C, ACC.4 ORL C, / ACC.4	– Let C = 1, A = 11H (0001 0001 B) then the instruction ORL C, ACC.4 will logically OR the contents of carry with accumulator Bit 4 leaving carry C = 1. – The instruction ORL C, / ACC.4 will logically OR the contents of carry with complement of accumulator Bit 4 leaving carry C = 1.

1.31.6 MOV <dest-bit>, <src-bit>

Mnemonic : MOV <dest-bit>, <src-bit>	**Function :** Move bit data
Algorithm : <dest-bit> = <src-bit>	**Operation :** MOV <dest-bit> ← <src-bit>

– This instruction will copy the source bit to the destination bit. One of the operands must be carry flag, the other operand may be any directly addressable bit. The different combinations are,

1. MOV bit, C

Operation	MOV (Bit) ← C	This instruction will copy the carry flag status into the Boolean variable whose address is specified in the instruction.
Example	MOV ACC.3,C	Let C = 1 A = 11H (0001 0001 B) then the instruction MOV ACC .3, C will copy the contents of carry to 3rd bit of accumulator leaving the accumulator with 19H (0001 1001 B).

2. MOV C, Bit

Operation	MOV C ← (Bit)	This instruction will copy the data from Boolean variable whose address is specified in the instruction to the carry flag.
Example	MOV C, ACC.4	Let C = 1, A = 01H (0000 0001 B), then the instruction MOV C, ACC.4 will load the contents of 4th bit of accumulator to carry flag, leaving C = 0.

1.32 Program and Machine Controls Instructions

1.32.1 ACALL addr11 SPPU - May 12, May 13

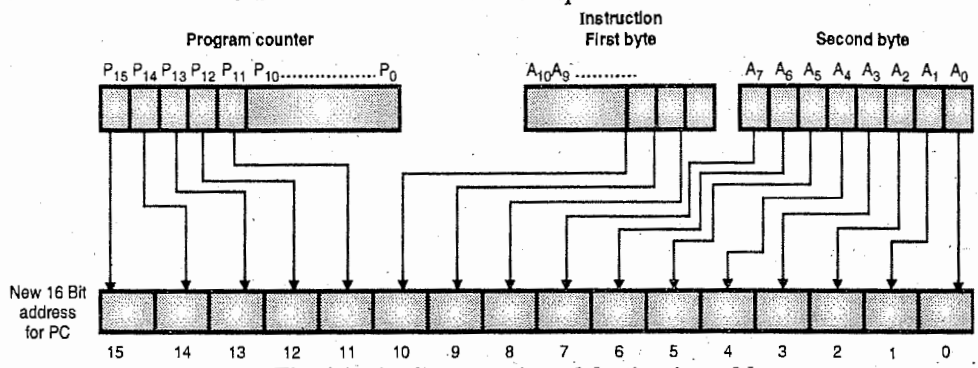
University Question

Q. Explain following instruction : ACALL **(May 2012, May 2013, 2 Marks)**

Mnemonic	ACALL addr11	**Function**	Absolute call
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	(PC) = ((PC) + 2) (SP) = (SP) + 1 ((SP)) = (PC_{7-0}) (SP) = (SP) + 1 ((SP)) = (PC_{15-8}) (PC_{10-0}) = Page address.

Operation	ACALL (PC) ← (PC) + 2 (SP) ← (SP) + 1 ((SP)) ← (PC_{7-0}) (SP) ← (SP) + 1 ((SP)) ← (PC_{15-8}) (PC_{10-0}) ← Page address.	– This instruction unconditionally calls a subroutine at the indicated address. At the end of subroutine the program will resume operation at the opcode address following the call instruction. – ACALL can be located anywhere in the program memory. It can be called/used a number of times in the same program. – The return address of the next instruction after the call instruction is in the program counter. The steps followed while executing ACALL instruction are : **Step 1 :** The PC increments by twice inorder to obtain the address of the next instruction after CALL. **Step 2 :** It then pushes the 16 bit result onto the stack. Initially (SP) increments by 1. Then the low order byte of PC is pushed onto the stack (SP) again increments by 1. Now the high-order byte of PC is pushed onto the stack. **Step 3 :** The destination address is computed by concatenating the high-order five bits of the incremented PC, 3 bits (A_{10}, A_9, A_8) from the first byte of instruction and the 8 bits (A_7 to A_0) second byte of instruction.
Example	ACALL add	Let SP = 0AH. PC = 0239H. The label "add" is at program memory location 0435 H. After the execution of instruction, ACALL add at location 0237 H, SP will contain 0CH, the internal RAM location 0BH will contain 39H and 0CH will contain 02H and the PC will contain 0435H.

Fig. 1.32.1 shows how the destination address is computed.



m(21.8)**Fig. 1.32.1 : Computation of destination address**

1.32.2 LCALL addr16

Mnemonic	LCALL addr16
Machine cycles	2
Bytes	3

Function	Long call
Clock Pulses	24
Algorithm	(PC) = (PC) + 3 (SP) = (SP) + 1 ((SP)) = (PC_{7-0}) SP) = (SP) + 1 ((SP)) = (PC_{15-8}) (PC) = $addr_{15-0}$

Operation	LCALL (PC) ← (PC) + 3 (SP) ← (SP) + 1 ((SP)) ← (PC_{7-0}) (SP) ← (SP) + 1 ((SP)) ← (PC_{15-8}) (PC) ← $addr_{15-0}$	– This instruction calls a subroutine that is located at the indicated address. The steps followed while executing LCALL instruction are : **Step 1 :** The PC increments by thrice inorder to generate the address of the next instruction. **Step 2 :** It then pushes the 16 bit PC onto the stack. Initially (SP) increments by 1. Then the low-order byte of PC is pushed onto the stack (SP) again increments by 1. Now the high-order byte of PC is pushed onto the stack. **Step 3 :** The PC will be loaded with the second and third bytes of the instruction.

Example	LCALL add	Let the Stack Pointer SP = 07 H and PC = 023AH. The label "add" is at program memory location 0435H. After the execution of instruction. LCALL add at location 0237H, the stack pointer will contain 09H, internal RAM location 08H will contain 3DH and location 09H will contain 02H, and the PC will contain 0435H

1.32.3 RET

Mnemonic	RET
Machine cycles	2
Bytes	1

Function	Return from Subroutine
Clock Pulses	24
Algorithm	(PC_{15-8}) = ((SP)) (SP) = (SP) – 1 (PC_{7-0}) = ((SP)) (SP) = (SP) – 1

Operation	RET (PC_{15-8}) ← ((SP)) (SP) ← (SP) – 1 (PC_{7-0}) ← ((SP)) (SP) ← (SP) – 1.	– When this instruction is executed, it informs the microcontroller to return back to the program, from where subroutine was called. – This instruction Pops the high and low order bytes of the PC successively from the stack. The stack pointer is decremented by two. – The program execution will resume from the resulting address, generally the address of the next instruction that follows the LCALL or CALL instruction.

Example	RET	The Stack Pointer contains value 09H. Internal RAM locations 08H and 09H contain 53H and 27H. The instruction, RET will leave the stack pointer equal to the value 07H. Program execution will continue at location 2753H.

1.32.4 RETI

Mnemonic	RETI	**Function**	Return from interrupt
Machine cycles	2	**Clock Pulses**	24
Bytes	1	**Algorithm**	$(PC_{15-8}) = ((SP))$ $(SP) = (SP) - 1$ $(PC_{7-0}) = ((SP))$ $(SP) = (SP) - 1$

Operation	RETI $(PC_{15-8}) \leftarrow ((SP))$ $(SP) \leftarrow (SP - 1)$ $(PC_{7-0}) \leftarrow ((SP))$ $(SP) \leftarrow (SP - 1)$	– This instruction informs the microcontroller that it is returning from an ISR routine. – This instruction pops high and low order bytes of PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. – The Stack Pointer is decremented by two. – The program execution resumes from the instruction that follows the point at which the interrupt request was detected. – If an interrupt of the same level is pending, then it is processed.
Example	RETI	Let SP = 09H, let us assume that an instruction RETI is detected at location 0322H. The internal RAM locations 08H contains 57H and location 09H contains 12H. The instruction RETI, will leave the SP = 07H and returns the program execution to location 1257H. i.e. PC = 1257H

Note : (i) If the RETI instruction is used at the end of subroutine, then it may enable the interrupt logic erroneously.
(ii) No other registers are affected.
(iii) PSW is not automatically restored to its pre-interrupt status.

– The only difference between the RET and RETI instructions is that whenever an interrupt logic is enabled RETI instruction is to be used.

1.32.5 AJMP addr11

Mnemonic	AJMP addr11	**Function**	Absolute jump
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	PC) = (PC) + 2 (PC_{10-0}) = Page address.

Operation	AJMP $(PC) \leftarrow (PC) + 2$ $(PC_{10-0}) \leftarrow$ Page address	– This instruction transfers the program execution to the indicated address i.e. AJMP label. – The limitation is that the label (destination) must be within the same 2KB block of program memory. The destination address is calculated by concatenating the higher order five bits of the PC i.e. $(P_{15} - P_{11})$ after the PC is incremented by twice, bits 5 – 7 of the first byte of instruction i.e. (A_{10}, A_9, A_8) and the second byte of instruction. Fig. 1.32.1 shows the address computation.
Example	AJMP L7	The label "L7" is at program memory location 0537H. The instruction AJMP L7 is at location 0671H and will load the PC with 0537H.

The jump range is the difference in bytes of the new address from the bytes in the program counter. E.g. if a jump instruction is at program address 1000H and jump causes program counter to be 1050H, then the jump range is 50H bytes.

The jump instructions have following ranges.

Relative jump range : + 127 to – 128

Absolute jump range : 0000H to 07FFH

Long jump range : 0000H - FFFFH

1.32.6 LJMP addr16

Mnemonic	LJMP addr16
Machine cycles	2
Bytes	3

Function	Long Jump
Clock Pulses	24
Algorithm	(PC) = $addr_{15-0}$

Operation	LJMP (PC) ← $addr_{15-0}$	– This instruction causes an unconditional branch to the indicated address, by loading the high order and low-order bytes of the PC respectively, with the second and third instruction bytes. – The destination may therefore be anywhere in the full 64K program memory address space, because it uses full 16 bits of two bytes i.e. 2nd and 3rd instruction bytes.
Example	LJMP L7	The label "L7" is assigned at memory location 5678H. The instruction LJMP L7 at location 0537H will load the program counter with 5678H.

1.32.7 SJMP rel

Mnemonic	SJMP rel
Machine cycles	2
Bytes	2

Function	Short jump
Clock Pulses	24
Algorithm	(PC) = (PC) + 2 (PC) = (PC) + rel

Operation	SJMP (PC) ← (PC) + 2 (PC) ← (PC) + rel	– The program control branches unconditionally to the address indicated. – The branch destination is computed by adding the signed i.e. relative displacement in the second instruction byte to the PC, after incrementing the PC by twice. – The only limitation is that the jump range is limited from – 128 to + 127 bytes. The destination range allowed is from 128 bytes preceding this instruction to 127 bytes following it.
Example	SJMP SUBA1	The label "SUBA1" is assigned to an instruction whose program memory location is 0478H. The instruction SJMP SUBA1 assembles into location 0401H. After the execution of this instruction the PC will contain 0478H.

1.32.8 JMP @A+DPTR

Mnemonic	JMP @A+DPTR
Machine cycles	2
Bytes	1

Function	Jump indirect
Clock Pulses	24
Algorithm	(PC) = (A) + (DPTR)

Operation	JMP (PC) ← (A) + (DPTR)	– This instruction adds the eight bit unsigned contents of the accumulator with the contents of sixteen bit Data Pointer. The result of addition is loaded into the program counter. This is the address from where the microcontroller will begin execution. – Neither the accumulator nor the DPTR contents are altered.

Example	JMP @A + DPTR	(i) Let (A) = 30H, (DPTR) = 1000 H, PC = 0500 H. The instruction JMP @A + DPTR will the PC will contain 1030H, and execution will begin from location 1030H instead of location 0500H. (ii) Another application of this instruction is in CASE jumps.
		MOV DPTR, #JMP_TBL ; DPTR points to jump table. MOV A, INDEX – NO JMP @A + DPTR. JMP_TBL : AJMP CASE – 0 AJMP CASE – 1 AJMP CASE – 2 AJMP CASE – 3 AJMP CASE – 4 Let say Index – No = 1 then DPTR = 1000H, A = 01H ∴ Execution begins from address 1001H i.e. CASE – 1.

1.32.9 JZ rel

Mnemonic	JZ rel	**Function**	Jump if accumulator zero
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	PC = (PC) + 2 if A = 0 then (PC) = (PC) + rel

Operation	JZ (PC) ← (PC) + 2 if (A) = 0 then (PC) ← (PC) + rel	– This instruction will branch i.e. jump to indicated address if all the bits in the accumulator are zero otherwise it will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second instruction byte to the PC, after incrementing the PC by two. – The accumulator remains unchanged.
Example	JZ LABEL5	Let A = 00H. The instruction, JZ LABEL5 will cause the program execution to continue at the instruction identified by label LABEL5.

1.32.10 JNZ rel

SPPU - May 12, May 13, Oct. 16

University Question

Q. Explain following instruction : JNZ

(May 2012, May 2013, Oct. 2016 (In Sem.), 2 Marks)

Mnemonic	JNZ rel	**Function**	Jump if Accumulator Not Zero
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	(PC) = (PC) + 2 if A ≠ 0 then PC = PC + rel

Operation	JNZ (PC) ← (PC) + 2 if A ≠ 0 then PC = PC + rel.	– This instruction will branch i.e. jump to indicated address if the bits in the accumulator are nonzero otherwise it will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second instruction byte to the PC, after incrementing the PC by two. – The accumulator remains unchanged.
Example	JNZ L1	Let A = 05H. The instruction JNZ L1 will continue program execution at label L1

1.32.11 JC rel

Mnemonic	JC rel	**Function**	Jump if carry is set
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	(PC) = (PC) + 2 if C = 1 then PC = PC + rel

Operation	JC (PC) ← (PC) + 2 if C = 1 then (PC) ← (PC) + rel	– This instruction will branch i.e. jump to the address indicated if the carry flag is set otherwise it will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second instruction byte to the PC, after incrementing the PC twice.

Example	JC L1	The instruction JC L1 will cause the program execution to continue at the instruction identified by Label L1 if the carry flag is set.

1.32.12 JNC rel

Mnemonic	JNC rel	**Function**	Jump if carry not set
Machine cycles	2	**Clock Pulses**	24
Bytes	2	**Algorithm**	(PC) = (PC) + 2 if C ≠ 1 (i.e. C = 0) then (PC) = (PC) + rel

Operation	JNC (PC) ← (PC + 2) if C ≠ 1 (i.e. C = 0) then (PC) ← (PC) + rel	– This instruction will branch i.e. jump to the address indicated if the carry flag is cleared otherwise continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the second byte of instruction to the PC, after the PC is incremented by twice
Example	JNC	Let the carry flag is set C = 1 The instruction sequence JNC L1 CPL C JNC L2 will clear the carry flag and program execution will resume from the instruction identified by Label L2.

1.32.13 JB bit, rel

Mnemonic	JB bit, rel	**Function**	Jump if bit set
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	(PC) = (PC) + 3 if (bit) = 1 then (PC) = PC + rel

Operation	JB (PC) ← (PC) + 3 if (bit) = 1 then (PC) ← (PC) + rel	– This instruction will jump to the address indicated, if the indicated bit in the instruction is 1, otherwise program will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to the first byte of next instruction – The bit indicated is unchanged.
Example	JB P2.1, L1	Let port 2 = 73 H (0111 0011 B). The instruction JB P2.1, L1 will cause program execution to jump to the instruction at label L1

1.32.14 JNB bit, rel.

Mnemonic	JNB bit, rel.	**Function**	Jump if bit not set
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	(PC) = (PC) + 3 if (bit) = 0 then (PC) = (PC) + rel.

Operation	JNB (PC) ← (PC) + 3 if (bit) = 0 then (PC) ← (PC) + rel.	– This instruction will jump to the address indicated, if the indicated bit in the instruction is 0, otherwise program will continue with the next instruction. – The branch destination is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction. – The bit remains unchanged.
Example	JNB P1.3, L3	Let port 2 = 73 H (0111 0011 B). The instruction JNB P1.3, L3 will cause the program execution to resume at the instruction at label L3

1.32.15 JBC bit, rel.

Mnemonic	JBC bit, rel.	**Function**	Jump if bit set and clear bit
Machine cycles	2	**Clock Pulses**	24
Bytes	3	**Algorithm**	(PC) = (PC) + 3 if (bit) = 1 then bit = 0 and (PC) = (PC) + rel.

Operation	JBC (PC) ← (PC) + 3 if (bit = 1) then (bit = 0) and (PC) ←(PC) + rel.	– This instruction will jump to the address indicated if the indicated bit is 1, otherwise the program will continue with the next instruction. – The destination is computed by adding the signed relative displacement in the third instruction byte to the PC, once the PC in incremented to point the first byte of the next instruction. – The contents of the indicated bit are changed if the bit is set otherwise of bit are unaltered the contents
Example	JBC P2.0, L1	Let port 2 = 75 H (0111 0101 B). The instruction JBC P2.0, L1 will cause program execution to resume form label L1 and port 2 = 74 H (0111 0100 B)

1.32.16 CJNE <dest-byte>, <src-byte>, rel. SPPU - May 13

University Question

Q. Explain following instruction : CJNE **(May 2013, 2 Marks)**

Mnemonic	CJNE <dest-byte>, <src-byte>, rel.

Algorithm	(PC) = (PC) + 3
	if (dest-byte) ≠ (src-byte)
	then (PC) = (PC) + rel.
	if (dest-byte) < (src-byte)
	then C = 1
	else
	C = 0

Function	Compare and Jump if not equal
Operation	– This instruction compares the magnitudes of the source bytes and the destination byte. If their values are unequal then it jumps to the address indicated otherwise program execution continues from the next instruction. – Neither source-byte nor the destination-byte is altered. The destination is calculated by adding the signed relative-displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction. – The carry flag is set if the dest-byte is less than the source byte, otherwise carry flag is cleared.

– This instruction supports different combinations. They are :

1. CJNE A, direct, rel. SPPU - Dec. 13

University Question

Q. Explain : CJNE A, B, label **(Dec. 2013, 2 Marks)**

Mnemonic	CJNE A, direct, rel.	**Function**	Compare and Jump if not equal.
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			if (A) ≠ (direct)
			then
			(PC) = (PC) + rel.
			if (A) < (direct)
			then
			C = 1
			else
			C = 0.

Addr. Mode	Direct addressing mode	**Flags**	Except carry, no other flags are affected.

Operation	CJNE A, direct, rel	– This instruction compares the magnitudes of accumulator and magnitude of memory location whose direct address is provided in the instruction and jumps to the indicated address if the magnitudes are unequal. – The carry flag is set if the contents of accumulator are smaller than the contents of memory location whose direct address is specified, otherwise it is cleared. – The destination address is calculated by adding the signed relative-displacement in the third instruction byte to the PC, once the PC is incremented to point to the first byte of the next instruction.

Example	(i) CJNE A, 60 H, L5	Let A = 75H, contents of memory location 60H = 44 H then the instruction CJNE A, 60H, L5 Will clear the carry flag as contents of A > contents of memory location 60H and jump to instruction at label L5.
	(ii) CJNE A, B, label	This instruction compares the magnitudes of accumulator and B register and jumps to the address indicated if the magnitudes are not equal.

2. CJNE A, #data, rel.

Mnemonic	CJNE A, #data, rel.	**Function**	Compare and jump if not equal
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			If (A) ≠ (data)
			then
			(PC) = (PC) + rel.
			if (A) < (data)
			C = 1
			else
			C = 0

Addr. Mode	Immediate addr mode	**Flags**	Carry flag is affected

Operation	(PC) ← (PC) + 3
	if (A) ≠ (data)
	(PC) ← (PC) + rel.
	if A < data
	C ← 1
	else
	C ← 0.
		– This instruction compares the magnitudes of accumulator with the magnitude of data specified in the instruction and jumps to the indicated address if the magnitudes are unequal.
		– The carry flag is set if the contents of accumulator are smaller than the data otherwise carry flag is cleared.
		– The destination address is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the point to the first byte of the next instruction.

Example	CJNE A, #60, L7	Let A = 44H, then the instruction CJNE A, #60H, L7 will set the carry flag and jump to instruction at Label L7.

3. CJNE Rn, #data, rel.

Mnemonic	CJNE Rn, #data, rel	**Function**	Compare and Jump if not equal
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			if (Rn) ≠ data
			then
			(PC) = (PC) + rel
			if (Rn) < data
			then C = 1
			else
			C = 0

Operation	CJNE Rn, #data, rel	– This instruction compares the magnitudes of register Rn of the selected register bank with the data specified in the instruction and jumps to the indicated address if the magnitudes are unequal. – The carry flag is set if the contents of register Rn are smaller than data, otherwise it is cleared. – The destination address is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction.
Example	CJNE R2, #60 H, L8	Let R2 = 44H then the instruction CJNE R2, #60H, L8 will set the carry flag and jump to instruction at Label L8.

4. CJNE @Ri, #data, rel.

Mnemonic	CJNE @Ri, #data, rel	**Function**	Compare and Jump if not equal
Machine cycles	2	**Clock Pulses**	24

Bytes	3	**Algorithm**	(PC) = (PC) + 3
			if ((Ri)) ≠ data
			then
			(PC) = (PC) + rel
			if ((Ri)) < data
			then
			C = 1
			else
			C = 0.

Operation	CJNE @Ri, #data, rel.	– This instruction compares the magnitudes of contents pointed by register Ri of the selected register bank with the data specified in the instruction and jumps to the indicated address if the magnitudes are unequal. – The carry flag is set if the contents pointed by register Ri are smaller than the data, otherwise the carry flag is cleared. – The destination address is calculated by adding the signed relative displacement in the third instruction byte to the PC, once the PC is incremented to point the first byte of the next instruction.
Example	CJNE @R1, #98, add	Let R1 = 57H and the contents of memory location 57H be 99H. then the instruction CJNE @R1, #98H, add will set the carry flag and jump to instruction at label add.

1.32.17 DJNZ <byte>, <ret-addr>

Mnemonic	DJNZ <byte>, <rel-addr>
Machine cycles	2
Bytes	2 if register addressing is used 3 if direct addressing is used

Function	Decrement specified memory location/register by 1 and jump if not zero
Clock Pulses	24
Algorithm	(PC) = (PC) + 2 (byte) = (byte) – 1 if (byte) ≠ 0 then (PC) = (PC) + rel

Operation	DJNZ (PC) ← (PC) + 2 (byte) ← (byte) – 1 if byte ≠ 0 then (PC) ← (PC) + (rel)	– This instruction decrements the specified register or memory location by 1 and branches to the address indicated by the second operand if the resulting value is nonzero. – The destination is calculated by adding the signed relative displacement value in the last instruction byte to the PC, after incrementing the PC to the first byte of the next instruction. – The original value of 00H will underflow to FFH if decremented. – It supports two addressing modes : register addressing mode and direct addressing mode.
Example	DJNZ R5, SUB	– Let R5 = 50 H the instruction DJNZ R5, SUB will cause jump to the instruction at label SUB and R5 = 4FH.

1.32.18 NOP

Mnemonic	NOP
Machine cycles	1
Bytes	1

Function	No operation
Clock Pulses	12
Algorithm	(PC) = (PC) + 1

Operation	NOP (PC) ← (PC) + 1	– No operation is performed. Only the PC is affected. The contents of PC are incremented by 1. – It is mainly used for inserting delay in programs.

1.32.19 Solved Examples

Ex. 1.32.1

Write instructions to

(i) Read from external program memory at address 0200H.

(ii) Write to external program memory at address 6000H in 8051 microcontroller.

Soln. :

(i) Read from external program memory at address 0200H.

```
MOV DPTR, #0200H      ; pointer to external program memory
MOVX A, @DPTR         ; read data from external program
                        memory.
```

(ii) Write to external program memory at address 6000H.

MOV DPTR, #6000H ; pointer to external program memory

MOVX @DPTR, A ; write data to external program memory

Ex. 1.32.2

Write an instruction to clear the bit, which has address 001h.

Soln. : CLR #001h

Ex. 1.32.3

Name four bit addressable instructions of 8051.

Soln. : Four addressable instructions are

(1) CLR Bit (2) SET Bit

(3) CPL Bit (4) ANL C, <bit>

Ex. 1.32.4

For a 8051 system of 11.0592 MHz, find how long it takes to execute each of the following instructions :

(a) DEC R3 (b) SJMP.

Soln. :

$$\text{Time required for 1 machine cycle} = \frac{12}{11.0592 \times 10^6}$$

$= 1.085\ \mu s.$

(i) The instruction DEC R3 needs one machine cycle.

$\therefore$ It takes 1.085 μs for its execution.

(ii) The instruction SJMP needs two machine cycles.

$\therefore$ It takes $2 \times 1.085 = 2.17\ \mu s$ for its execution.

Syllabus Topic : Sample Programs (Assembly)

1.33 Sample Programs (Assembly)

Program 1.33.1

Write assembly language program to add two sixteen bit numbers stored at locations 30 H and 32 H and store the result from location 40 H onwards.

Soln. :

Instruction	Comment
CLR C	CY = 0
MOV A, 30H	Load low byte of number at location 30H into A
ADD A, 32H	Add the lower bytes
MOV 40H, A	Store the lower byte of sum at location 40H.
MOV A, 31H	Load high byte of number into A.
ADDC A, 33H	Add the higher bytes with carry.
MOV 41H, A	Store the higher byte of sum at location 41H

Program 1.33.2 :

Write assembly language program for adding two, 16 bit numbers stored in external memory location from A000H and result to be stored in external memory locations from B000H.

Soln. :

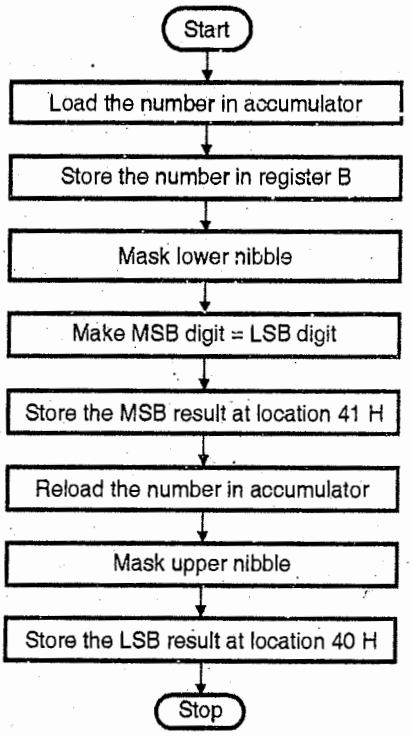
Instruction	Comment
CLR C	CY = 0
MOV DPTR, #A000H	Initialize DPTR from A000H
MOVX A, @DPTR	Load low byte of number at A000H into A
MOV R7, A	Load LSB of first number in R7
INC DPTR	Increment DPTR to next location
MOVX A, @DPTR	Load MSB of first number into A
MOV R6, A	Load MSB of first number in R6
INC DPTR	Increment DPTR to next location
MOVX A, @DPTR	Load LSB of second number in A
MOV R5, A	Load LSB of second number in R5
INC DPTR	Increment DPTR to next location
MOVX A, @DPTR	Load MSB of second number in A
MOV R4, A	Load MSB of second number in R4
MOV A, R7	Move lower byte of first number into accumulator
ADD A, R5	Add the two lower bytes
MOV DPTR #B000H	Initialize DPTR to destination memory location
MOVX @DPTR, A	Load the LSB result at B000H
INC DPTR	Increment DPTR to next memory location
MOV A, R6	Load the MSB of first number into accumulator
ADDC A, R4	Add the MSBs of two numbers with carry of LSB addition
MOVX @DPTR, A	Load the MSB result at location B001H
INC DPTR	Increment DPTR to next memory location
MOV A, #00H	
ADDC A, #00H	Add zero, plus carry from MSB addition
MOVX @DPTR, A	Load the result to highest byte of result
END	

Program 1.33.3

Write an assembly language program of 8051 to unpack the BCD no. stored at 30 H location. Store the MSB digit at memory location 41 H and LSB digit at memory location 40 H. Draw the flowchart for same.

Soln. :

Instruction	Comment
MOV A, 30H	Load the BCD number into accumulator
MOV B, A	Store the result in register B
ANL A, #0F0H	Mask lower nibble
SWAP A	Make MSB digit = LSB digit
MOV 41H, A	Store MSB digit at memory location 41H
MOV A, B	Load the number back in accumulator
ANL A, #0FH	Mask upper nibble
MOV 40H, A	Store LSB digit at memory location 40H



Flowchart 1

Program 1.33.4 :

Write an assembly language program for 8051 to unpack the BCD number stored at external memory location 3000H. Store the result in internal memory locations 40H (LSB) and 41H (MSB).

Soln. :

Instruction	Comment
MOV DPTR, #3000H	Initialize DPTR with 3000H
MOVX A, @DPTR	Load the BCD number in accumulator
MOV B, A	Store the number in register B
ANL A, #0F0H	Mask lower nibble
SWAP A	Make MSB digit = LSB digit
MOV 41H, A	Store MSB digit at memory location 41H
MOV A, B	Load the number back in accumulator
ANL A, #0FH	Mask upper nibble
MOV 40H, A	Store LSB digit at memory location 40H

Program 1.33.5

Program to multiply two unpacked BCD numbers

Program statement

- A two digit BCD number is stored in the register A. Write a program in the ALP of 8051 to unpack this BCD number. Store the MSB digit in register R1 and LSB digit in register R0 of the register bank 3. Find the product of these two digits and store the result in packed BCD form in accumulator.

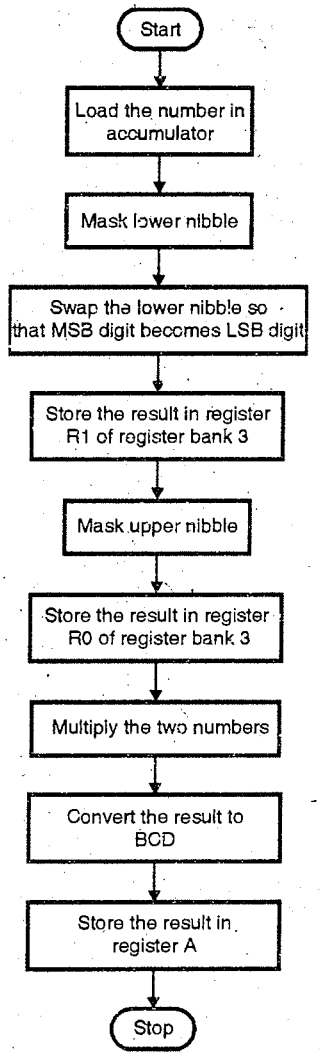
Explanation

- A digit BCD number is available in register A. We have to unpack this BCD number i.e. we have to separate the BCD digits.
- e.g. ; If the number = 92 H then in unpack form the two digits will 02 H and 09 H. i.e. we have to mask the lower nibble, first and rotate four times to the right to get the MSB digit or use the SWAP instruction.
- Then to get the LSB digit mask the upper nibble.
- Masking lower nibble means ANDing the number with 0F0 to get MSB.
- Store the obtained unpacked digits.
- We are supposed to find the product of the two unpacked digits. We will copy the unpacked digits in registers A and B and then using the MUL instruction we will find the product of the two numbers.
- Then convert the result from binary to packed BCD format.

Algorithm

Step I	:	Load number into register A.
Step II	:	Mask the lower nibble.
Step III	:	Swap the nibble to make MSB digit = LSB digit.
Step IV	:	Store the digit in register R1
Step V	:	Load number in register A.
Step VI	:	Mask upper nibble.
Step VII	:	Store the digit in register R0.
Step VIII	:	Multiply the two digits
Step IX	:	Convert the result of multiplication to BCD form
Step X	:	Stop

Flowchart : Refer Flowchart 2



Flowchart 2

Program

Instruction	Comment
MOV A, #92H	Load the number in accumulator.
MOV B,A	Store the number in register B.
ANL A, #0F0H	Mask lower nibble
SWAP A	Make the MSB digit = LSB digit
MOV 19 H, A	Store the result in register R1 of register bank 3. 19 H is the address of register R1 of register bank 3.
MOV A,B	Load the number back in accumulator.
ANL A, #0FH	Mask upper nibble
MOV A, 18H	Store the result in register R0 of register bank 3. 18 H is the address of register R0 of register bank 3.
MOV B, 19H	Load the second digit in register B
MUL AB	Find the product i.e. A × B
MOV B, #0AH	Load decimal 10 in B to find BCD equivalent
DIV AB	A = quotient , B = remainder
SWAP A	MSB digit in A = LSB digit in A and vice versa.
ORL A, B	Pack the two digits
END	End Program

Program 1.33.6

To add Block of data.

Program statement

- Write a program to add ten bytes in internal RAM. Assume that the starting location of the Block is 40 H. Assume sum to be 8 bit. Store the result in register R0 of bank 1.

Explanation

- Consider that a block of 10 bytes is present at source location i.e. 40 H.
- We have to add these 10 bytes.
- We will initialize this as count in the R0 register.
- The register R1 will act as pointer to point the block.
- Using ADD instruction add the contents, byte by byte of the block.
- Increment R1 to point to next element.
- Decrement the counter and continue till all the contents are added.
- Result is stored in A. This result is stored in register R0 of bank 1.

For example :

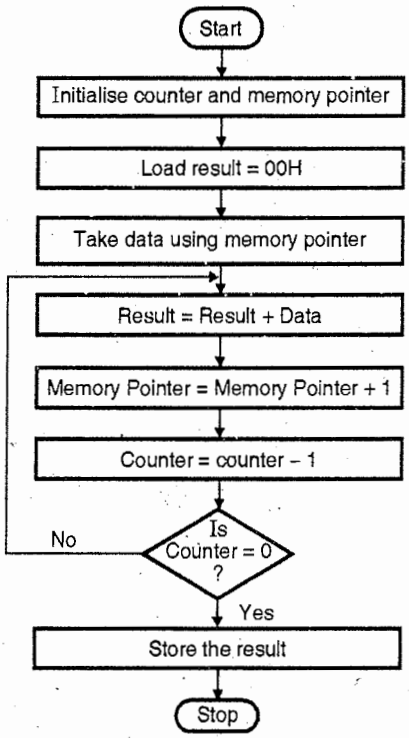
Block Data : 01 02 03 04 05 06 07 08 09 0A

Result : 01 + 02 + 03 + 04 + 05 + 06 + 07 + 08 + 09 + 0A = 37 H

Algorithm

Step I	:	Initialise R1 as pointer with source address.
Step II	:	Initialise R0 register with count.
Step III	:	Add data, byte by byte.
Step IV	:	Increment pointer.
Step V	:	Decrement counter.
Step VI	:	Check for count, if not zero go to step III else go to step VII.
Step VII	:	Store the result of addition.
Step VIII	:	Stop.

Flowchart : Refer Flowchart 3.



Flowchart 3

Program

Label	Instruction	Comment	Operation
	CLR PSW.3 CLR PSW.4	Select Register bank 0	Register Bank 0 selected
	MOV R0, #0AH	Initialize register R0 as counter.	R0 = 0AH
	MOV R1, #40H	Initialize register R1 as memory pointer.	R1= 40H
	MOV A, #00H	Initialize result = 00H	A= 00H
L1:	ADD A, @R1	Compute addition	A = A + @R1
	INC R1	Increment R1 to point next memory location	R1= R1 + 1
	DJNZ R0, L1	Check if count = 0 ?	Is R0 = 0, if not continue executing loop L1
	MOV R0, A	Store the result in register R0 of bank 1	**R0 = 37H (Result)**
	END	End Program	

Program 1.33.7

Write assembly language program to add 10, 8 bit nos. stored from starting location 50 H and store result at 65 H and 66 H.

Soln. : Program

Label	Instruction	Comment	Operation
	CLR PSW.3 CLR PSW.4	Select Register bank 0	Register bank 0 selected
	MOV R0 , #0AH	Initialize register R0 as counter.	R0 = 0AH
	MOV R1 , #50H	Initialize register R1 as memory pointer.	R1 = 50H
	MOV A , #00H	Clear accumulator	A = 00H
	MOV R3 , #00H	Clear register R3	R3 = 00H
L1:	ADD A , @R1	Compute addition	A = A+ @R1
	JNC L2	Check for carry	If CY= 1 go to L2
	INC R3	Increment register R3 if carry is present	If CY= 1 then R3= R3+1 **R3 = 03H (Result)**
	MOV 66H, R3	Load result at 66H	
L2 :	INC R1	Increment R1 to point next memory location	R1= R1+ 1
	DJNZ R0, L1	Check if count = 0 ?	If R0 ≠ 0 then go to L1 else continue
	MOV 65H, A	Store the result at location 65H	**65H = 54H (Result)**
	END	End Program	

Program 1.33.8

Write an assembly language program of 8051 to add 8 bytes. The numbers are stored in memory location starting from 80H onwards and store result at 60H and 61H.

Soln. : Program

Label	Instruction	Comment
	CLR PSW.3	Select Register Bank 0
	CLR PSW.4	
	MOV R0, #08H	Initialize R0 as counter for 8 bytes
	MOV R1, #80H	Initialize R1 as memory pointer
	MOV A, #00H	Clear accumulator
	MOV R3, #00H	Clear register R3
L1 :	ADD A, @R1	Compute addition
	JNC L2	Check for carry
	INC R3	Increment register R3 if carry is present
	MOV 61H, R3	Load result at 61H
L2 :	INC R1	Increment R1 to next memory location
	DJNZ R0, L1	Check for count? If R0 ≠ 0 go to L1 else continue
	MOV 60H, A	Store addition result at 60H
	END	

Program 1.33.9

Write an assembly language program to add 5 numbers stored in internal RAM starting from address 40H onwards. Store the result in location 60H (LSB) and 61H (MSB).

Soln. : Program

Label	Instruction	Comment
	CLR PSW.3	Select Register Bank 0
	CLR PSW.4	
	MOV R0, #05	Initialize R0 as counter for 5 numbers
	MOV R1, #40H	Initialize R1 as memory pointer
	MOV A, #00H	Clear accumulator
	MOV R3, #00H	Clear register R3
L1 :	ADD A, @R1	Compute addition
	JNC L2	Check for carry
	INC R3	Increment register R3 if carry is present
	MOV 61H, R3	Load result at 61H
	INC R1	Increment R1 to next memory location
	DJNZ R0, L1	Check for count? If R0 ≠ 0 goto L1 else continue
	MOV 60H, A	Store addition result at 60H
	END	

Program 1.33.10

Write a Program to add 10 bytes in external RAM. Assume starting location of block is 5000h. Assume the sum to be 8-bit. Store the result at memory location 6000h.

Soln. : Program

Label	Instruction	Comment
	ORG 0000h	Organize code from 0000h
	MOV R7, #0Ah	Load the Counter with 10 (decimal Ten)
	MOV R0, #00h	Initialize R0 to 00h.
	MOV DPTR, #5000h	Load DPTR with address where the numbers are stored – 5000h
Back:	MOVX A, @DPTR	Copy a Number from source location, into register A
	ADD A, R0	Add the number to cumulative sum stored in R0
	MOV R0, A	Move cumulative sum to R0 for next reference.
	INC DPTR	Increment DPTR – Source pointer
	DJNZ R7, Back	Decrement the counter and if not 0 then go to back for next.
	MOV DPTR, #6000h	Load DPTR with value of address where result is to be stored
	MOVX @DPTR,A	Store the result in the desired location
HERE:	SJMP HERE	Done
	END	

Program 1.33.11 **Lab Assignment**

Transfer a block of N bytes from source to destination (Non overlapped block transfer).

Program statement

– Write an ALP to move a block of N bytes of data from source to destination. Assume that the length of block is stored in register R2 of register bank 0. The source block starts from memory location 20H and the destination block begins from memory location 30 H.

Explanation

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- The number of bytes N are stored in register R2 of bank 0.
- For pointing the source address we will use the R0 register and for the destination address we will use the R1 register.
- Transfer the data byte by byte from source to destination block.

Algorithm

Step I : Initialize R0 and R1 with source and destination address.

Step II : Initialize R2 register with the count.

Step III : Transfer the data block byte by byte to destination.

Step IV : Decrement Count

Step V : Increment source and destination memory pointer.

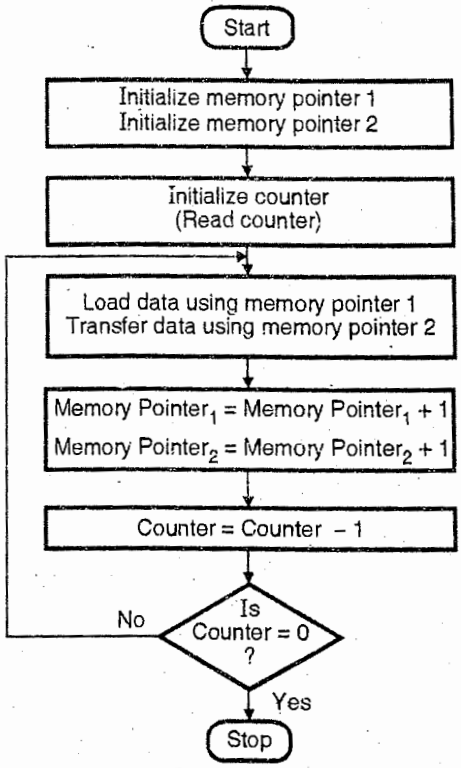
Step VI : Check for count in R2, if not zero goto step III else goto step VII.

Step VII : Stop.

Flowchart : Refer Flowchart 4.

Program

Label	Instruction	Comment
	CLR PSW.3	Select Register bank 0
	CLR PSW.4	
	MOV R0, #20H	Initialize register R0 as with source address.
	MOV R1,#30H	Initialize register R1 as with destination address.
L1 :	MOV A,@R0	Load accumulator with number from source block.
	MOV @R1,A	Store the data in desired memory location
	INC R0	Increment source memory pointer
	INC R1	Increment destination memory pointer
	DJNZ R2,L1	Decrement count in R2 , Check if count = 0 ? if not go to L1
	END	



Flowchart 4

Program 1.33.12

Transfer a block of N bytes from source in external memory to destination in internal memory. (Non overlapped block transfer)

Program statement

- Write an ALP to move a block of N bytes of data from source to destination. Assume that the length of block is stored in register R2 of register bank 0. The source block starts from memory location 2000 H and the destination block begins from memory location 40 H.

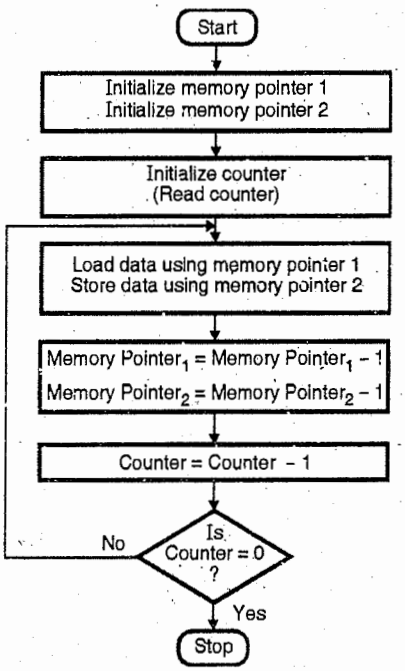
Explanation

- Consider that a block of data of N bytes is present at source location. Now this block of N bytes is to be moved from source location to a destination location.
- The number of bytes N are stored in register R2 of bank 0.
- Transfer the data byte by byte from source to destination block.

Algorithm

Step I : Initialise DPTR and R0 with source and destination address.
Step II : Initialise R2 register with the count.
Step III : Transfer the data block byte by byte to destination.
Step IV : Decrement Count
Step V : Increment source and destination memory pointer.
Step VI : Check for count in R2, if not zero goto step III else goto step VII.
Step VII : Stop.

Flowchart : Refer Flowchart 5.



Flowchart 5

Program

Label	Instruction	Comment
	CLR PSW.3	Select Register bank 0
	CLR PSW.4	
	MOV DPTR, #2000H	Initialize register DPTR as with source address.
	MOV R0, #40H	Initialize register R0 as with destination address.
L1 :	MOVX A, @DPTR	Load accumulator with number from source block.
	MOV @R0, A	Store the data at desired memory location
	INC DPTR	Increment source memory pointer
	INC R0	Increment destination memory pointer
	DJNZ R2, L1	Check if count = 0 ?
	END	

Program 1.33.13

Write an assembly language program to move 5 bytes of data stored at location 8000H onwards to the location C000H onwards and vice-versa.

Program

Label	Instruction	Comment
	CLR PSW.3	Select Register Bank 0
	CLR PSW.4	
	MOV R0, #05 H	Count = 05 H
	MOV DPL, #00H	Initialize memory pointer to 8000H
	MOV DPH, #80H	
L1 :	MOVX A, @DPTR	Load accumulator with number from source block
	MOV R1, A	Save it temporarily in register R1.
	MOV DPH, #C0H	Load the destination address of memory location
	MOVX A, @DPTR	Get the data in accumulator
	MOV R2, A	Save it in register R2
	MOV A, R1	Store data from source memory location in accumulator
	MOVX @DPTR, A	Store it at destination memory location
	MOV DPH, #80 H	Adjust memory pointer to point source memory location
	MOV A, R2	Get the data
	MOVX @DPTR, A	Store the data of destination memory to source memory
	INC DPTR	
	DJNZ R0, L1	If count #0, repeat

Program 1.33.14

Write assembly language program for 8051 to move 30 bytes of data from external RAM at address 2000 H to internal RAM located at 40 H as starting address.

Program

Label	Instruction	Comment
	CLR PSW.3	Select Register bank 0
	CLR PSW.4	
	MOV DPTR, #2000H	Initialize register DPTR as with source address.
	MOV R0, #40H	Initialize register R0 as with destination address.
	MOV R2, #30H	Initialize R2 with count i.e. 30.
L1 :	MOVX A, @DPTR	Load accumulator with number from source block.
	MOV @R0, A	Store the data at desired memory location
	INC DPTR	Increment source memory pointer
	INC R0	Increment destination memory pointer
	DJNZ R2, L1	Check if count = 0 ?
	END	

Program 1.33.15 **Lab Assignment**

ADD two 8 bit numbers.

Program statement

- Assuming two numbers are available in A and B registers, write a program in assembly language of 8051 to add two 8 bit numbers.

Explanation

- Consider that a byte of data is present in the A register and second byte of data is present in the B register.
- We have to add the byte in B from the byte in A. Using ADD instruction add the contents of two registers.
- Result will be stored in the A register.

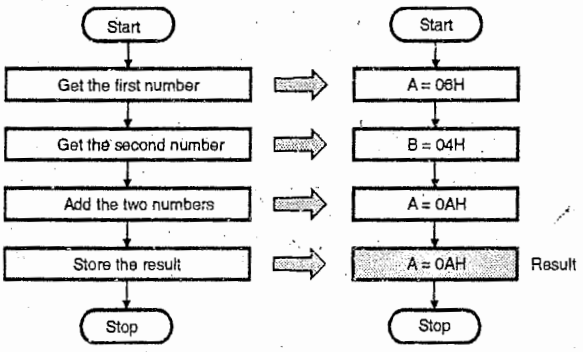
For example :

A = 06 H		06 H	$(06)_{10}$
B = 04 H	+	04 H	$(04)_{10}$
		0AH	$(10)_{10}$

Algorithm

Step I : Get the first number in A register.
Step II : Get the second number in B register.
Step III : Add the two numbers.
Step IV : Stop

Flowchart : Refer Flowchart 6.



Flowchart 6

Program

Instruction	Comment	Operation
MOV A, #06H	Load the first number in register A.	A = 06H
MOV B, #04H	Load the second number in register B.	B = 04H
ADD A, B	Compute addition. Result is stored in accumulator	**A = 0AH (Result)**
END	End program	End

Program 1.33.16 **Lab Assignment**

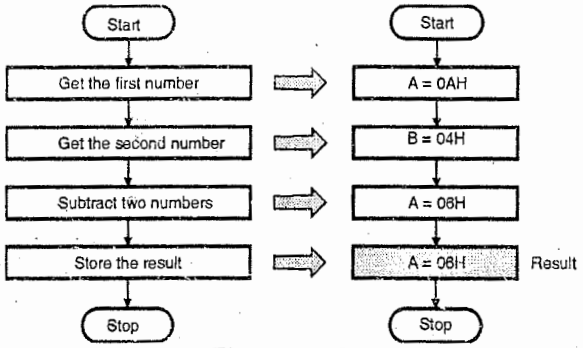
Subtract two 8 bit numbers.

Program statement

- Assuming two numbers are available in A and B registers, write a program in assembly language of 8051 to subtract two 8 bit numbers.

Explanation

- Consider that a byte of data is present in the A register and second byte of data is present in the B register.
- We have to subtract the byte in B from the byte in A. Using sub instruction subtract the contents of two registers.
- Result will be stored in the A register.



Flowchart 7

For example :

A = 0A H		0A H	$(10)_{10}$
B = 04 H	–	04 H	$(4)_{10}$
		06H	

Algorithm

Step I : Get the first number in A register.
Step II : Get the second number in B register.
Step III : Subtract the two numbers.
Step IV : Stop

Flowchart : Refer Flowchart 7.

Program

Instruction	Comment	Operation
MOV A, #0AH	Load the first number in register A.	A= 0AH
MOV B, #04H	Load the second number in register B.	B= 04H
SUBB A, B	Compute subtraction. Result is stored in accumulator	**A= 06H (Result)**
END	End program	End

Program 1.33.17

An array of 10 numbers is stored at location 5000H onwards. Write an assembly language program to arrange the numbers in ascending order in the same array.

Soln. : Program

Label	Instruction	Comment
	MOV R0, #10H	Initialize counter 1
START:	MOV DPTR, #5000H	Initialize memory pointer
	MOV R1, #10H	Initialize counter 2
BACK:	MOV R2, DPL	Save the address of lower byte
	MOVX A, @DPTR	Get the number in accumulator
	MOV R3, A	Store the number in R3
	INC DPTR	Increment memory pointer
	MOVX A, @DPTR	Get the next number in accumulator
	MOV B,A	
	MOV A,R3	
	CJNE A, B, NE	Compare number with next number
	AJMP SKIP	If less, don't interchange
NE:	JC SKIP	If equal, don't interchange
	MOV DPL, R2	
	MOV A,B	
	MOVX @DPTR, A	Otherwise swap the contents
	INC DPTR	Increment pointer to next memory location
	MOV A, R3	
	MOVX @DPTR, A	
SKIP :	DJNZ R1, BACK	If R1≠ 0 then goto BACK
	DJNZ R0, START	If R0≠ 0 then goto START
	END	

Program 1.33.18 Lab Assignment

Multiply two 8 bit numbers.

Program statement

Multiply two 8 bit numbers stored in external memory locations 3000 H and 3001 H. Store the result in memory locations 3020 H and 3021 H.

Explanation

- Consider that a byte is present at the memory location 3000 H and second byte is present at memory location 3001 H.
- We have to multiply the bytes present at the above two memory locations.
- We will multiply the numbers using MUL instruction. As MUL instruction operates only on the A and B register we will load the two numbers from memory locations 3000 H and 3001 H to the A and the B registers. The result of multiplication is stored in the A and B registers. LSB digit is stored in the accumulator while the MSB digit is stored in the B register.
- Result is stored at memory locations 3020 H and 3021 H with LSB stored at memory location 3020 H and MSB stored at memory location 3021 H.

For example : 3000 H = 09 H, 3001 H = 02 H

For example : A = 09 H, B = 02 H

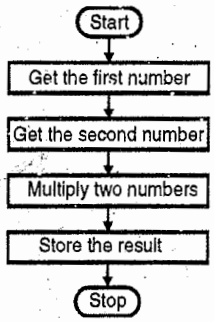
$$\begin{array}{r} 09\text{ H} \\ \times\ 02\text{ H} \\ \hline 0012\text{ H} \end{array}$$

Result = 0012 H
3020 H = 12 H
3021 H = 00 H

Algorithm

Step I : Get the first number.
Step II : Get the second number.
Step III : Multiply the two numbers.
Step IV : Store the result.
Step V : Stop.

Flowchart : Refer Flowchart 8.



Program **Flowchart 8**

Instruction	Comment	Operation
MOV DPTR, #3000H	Initialize DPTR as memory pointer	DPTR = 3000H
MOVX A, @DPTR	Get the first number in register A	A = 09H
MOV B, A	B = first number	B = 09H
INC DPTR	Increment memory pointer to point the second number	DPTR = 3001H
MOVX A, @DPTR	Get the second number in register A	A = 02H
MUL AB	Compute multiplication. A = LSB digit B = MSB digit after multiplication.	**A = 12H, B = 00H (Result)**

Instruction	Comment	Operation
MOV DPTR, #3020H	Initialize DPTR as memory pointer For storing result	DPTR = 3020H
MOVX @DPTR, A	Store the LSB digit obtained at location 3020H.	**3020H = 12H (LSB digit of result)**
INC DPTR	Increment memory pointer.	DPTR = 3021H
MOV A, B	Store the MSB digit obtained in register A.	A = 00H
MOVX @DPTR, A	Store the MSB digit obtained at location 3021H.	**3021H = 00H (MSB digit of result)**
END		

Program 1.33.19

Program for Binary-Gray conversion.

Program statement

- Write a program in the assembly language of 8051 to convert a given binary number into its Gray code equivalent. Draw flowchart. Store the Gray equivalent in register A.

Explanation

- For Binary-Gray conversion.
  (i) Record the MSB as it is.
  (ii) Add this bit to the next position, recording the sum and neglecting carry if any.
  (iii) Record successive sums until completed.
- e.g. to convert 0AH i.e. 1010 binary to gray.

Given binary no :- 1 0 1 0

Gray code :- 1 1 1 1 LSB = (0F)H

- For our program the logic that we will be using for Binary-Gray conversion is that first we will add the number with itself.
- Then we will X-OR this added number with the original number. Then we will shift this X-ORed number by 1 bit position to the right. This gives the equivalent gray code. Display this Gray code.
- e.g. number = 0A.

$$\begin{array}{r} 0\,A\,H \\ +\ 0\,A\,H \\ \hline \mathbf{14\,H} \end{array}$$

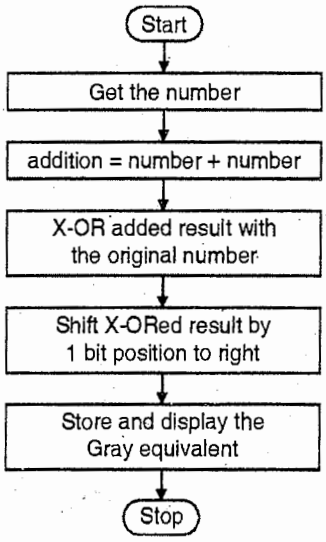
XOR added number	0	0	0	1	0	1	0	0	(14 H)
with original number	0	0	0	0	1	0	1	0	(0A H)
	0	0	0	1	1	1	1	0	
Shift by 1 bit to right	0	0	0	0	1	1	1	1	(0F H)

- (0FH) is the gray equivalent of 0A H.

Algorithm

Step I : Get the number whose gray code equivalent is be found.
Step II : Add number with itself.
Step III : XOR this added result with the original number.
Step IV : Shift the XORed number by 1 bit position to the right to get the Gray equivalent.
Step V : Store the result.
Step VI : Stop.

Flowchart : Refer Flowchart 9.



Flowchart 9

Program

1PT	Comment	Operation
MOV A, #0AH	Load number in register A	A=0AH
MOV B, A	Get the number in register B also.	B= 0AH
ADD A, A	Add contents of A with itself	A = 14H, PSW =40H
XRL A, B	XOR the added contents with number itself	A= 1EH, PSW = 40H
RR A	Roll by 1 bit to right to get gray equivalent	**A = 0FH (Result)**
END	END PROGRAM	

Program 1.33.20

Write a Program to transfer a string "PUNE" located at memory location 250h to memory location 350h.

Soln. :

- This program refers to external data memory (RAM) as addresses are greater than 8 bits.

- The Counter = 04h is to be loaded in R7.
- The source pointer is provided through DPTR and is loaded with 0250h.
- Destination pointer is done through P1 and R0 using external memory transfers and loaded with 03h and 50h respectively.

Program

Label	Instructions	Explanation
	ORG 0000h	Organize code from 0000h
	MOV R7, #04h	Load the Counter with 04
	MOV DPTR, #0250	Load DPTR with address -"PUNE" stored at address 0250h
	MOV P1, #03h MOV R0, #50h	Load Port 1 with 03 h and Load R0 with 50h to have total address ref. 0350h
Back :	MOVX A, @DPTR	Copy a character from source location, into register A
	MOVX @R0, A	Store the character to destination location
	INC DPTR	Increment Source pointer
	INC R0	Increment Destination pointer
	DJNZ R7, Back	Decrement the counter and if not 0 then go to back for next.
HERE:	SJMP HERE	Done
	END	

Program 1.33.21

Write an assembly language program to move a block of 20 bytes of data from source to destination. The source block start from memory location 30 H and destination block start from memory location 35 H. (Overlapped data transfer)

Soln. : Program

Label	Instruction	Comment
	CLR PSW.3 CLR PSW.4	Select Register bank 0
	MOV R0, #30H	Initialize register R0 as with source address.
	MOV R1, #35H	Initialize register R1 as with destination address.
	MOV A, R2	load accumulator with the size of block i.e. 20H.
	ADD A, R0	
	MOV R0, A	R0 contains address of the number after the last element in the source block.
	MOV A, R2	load accumulator with the size of block
	ADD A, R1	
	MOV R1, A	R1 contains address of the number after the last element in the destination block.
	DEC R0	R0 contains address of the last element of source
	DEC R1	R1 contains address of the last element of destination
L1 :	MOV A, @R0	Load accumulator with number from source block.
	MOV @R1,A	Store the data in desired memory location
	DEC R0	Decrement source memory pointer
	DEC R1	Decrement destination memory pointer
	DJNZ R2, L1	Check if count = 0 ?
	END	End Program

Program 1.33.22

Program for checking the parity of number is odd or even.

Program statement

- Given a number which is 8 bit. Write a program in ALP to find the parity of this number. If parity is even display 00 in the result and if parity is odd display 01 in the result. Draw flowchart.

Explanation

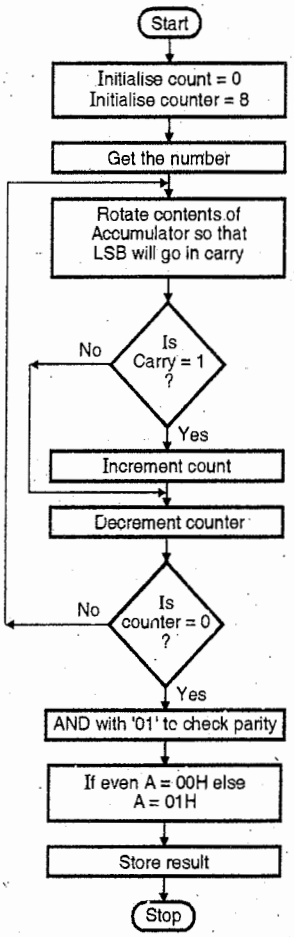
- We have a number given. Initially, we will count the number of 1's in that number. For this we will rotate the contents of number bit by bit to right, along with carry.
- If carry = 1 then increment the count for number of 1's. In this way we will count the number of 1's.
- Then AND the number of 1's with 01 H. If number is odd, ZF = 1.
- For even number ZF = 0.
- Store register A = 00 if number is even, otherwise store A = 01 H if the number is odd.
- Store the result.
- For e.g. if the number is 09 H (0000 1001 B) i.e. number has 2 ones. This means that this number has even parity.

Algorithm

Step I : Initialize counter = 8 for number of bits and count = 0.
Step II : Get the number.
Step III : Rotate the number by 1 bit to right alongwith carry.
Step IV : Check if carry = 1 ? If not goto step VI.
Step V : Increment count for number of 1's.
Step VI : Decrement counter.
Step VII : Check if count = 0 ? If not, goto step III.

Step VIII : AND with 01 H to check for parity.
Step IX : Store result.
Step X : Stop.

Flowchart : Refer Flowchart 10.



Flowchart 10

Program

Label	Instruction	Comment
	MOV A, #09H	Load the number whose parity is to be found in register A
	MOV R2, #08H	initialize counter
	MOV B, #00H	Initialize counter FOR COUNTING of number of 1's.
BACK:	RRC A	Rotate accumulator right through carry
	JNC SKIP	Check If Carry = 1
	INC B	Increment the count of number of number of 1's
SKIP:	DEC R2	Decrement count
	CJNE R2, #00H,BACK	If count ≠ 0 repeat
	MOV A,B	Store the count of number of 1's in register A
	ANL A, #01H	Check if number of 1's is even or odd to find parity
	JNZ NEXT	If number of 1's odd then jump to NEXT
	MOV A, #00H	Store 00 H in register A if number has even parity
	JMP ENDD	
NEXT:	MOV A, #01H	Store 01 H in register A if number has odd parity
ENDD:	END	

Program 1.33.23

To find the factorial of a number.

Program Statement

- Write a program in the assembly language of 8051 to find the factorial of a number.

Explanation

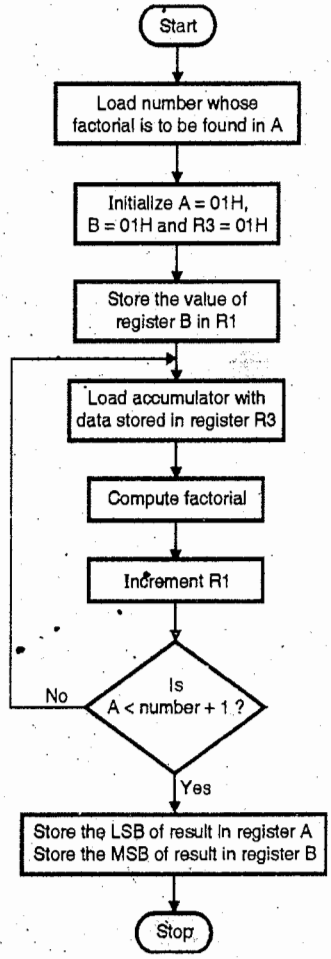
- To compute the factorial of a number, means to multiply the number say n with (n – 1) (n – 2) (n – 3) ...1.
  e.g. To compute 5 ! of a number.
  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.
- In our program, we will initialize the A and B registers with 1.
- We will load the number whose factorial is to be found in the A and register R0.
- Store the number in register B in register R1
- We will multiply number in A with number in B.
- Result of multiplication is stored in A register and B register.
- Increment R1 check that R1 is less than R0. If yes, continue the process till factorial is computed.
- Store the result.

Algorithm

Step I : Load the number whose factorial is to be computed in the R0 register.
Step II : Initialize A and B registers = 1.
Step III : Load accumulator with data in register R3 and R3 register store value of B in R1.
Step IV : Multiply A × B.
Step V : Increment R1.

Step VI : Check if A < (number+1), if no goto step VII
Step VII : Store the result.
Step VIII : Stop.

Flowchart : Refer Flowchart 11.



Flowchart 11

Program

Label	Instruction	Comment
	MOV R0, #05H	Load number in Register R0
	MOV A, #01H	initialize A=1
	MOV B, #01H	initialize B=1
	MOV R3, #01H	initialize R3=1
	MOV R1, B	
UP :	MOV A, R3	
	MUL AB	multiply numbers
	MOV R3, A	
	MOV R2 , B	
	INC R1	increment R1
	MOV B, R1	Restore the value of B
	MOV A,B	
	CJNE A, #06H, UP	multiply till B < R0
	MOV A,R3	store the LSB of result in Register A
	MOV B,R2	store the MSB of result in register B
	END	

Program 1.33.24

Write assembly language program for finding out factorial of decimal no. 8.

Soln. : Program

Label	Instruction	Comment
	MOV R0, #08H	Load number in Register R0
	MOV A, #01H	initialize A=1
	MOV B, #01H	initialize B=1
	MOV R3, #01H	initialize R3=1
	MOV R1, B	
UP :	MOV A, R3	
	MUL AB	multiply numbers
	MOV R3, A	
	MOV R2 , B	
	INC R1	increment R1
	MOV B, R1	Restore the value of B
	MOV A,B	
	CJNE A, #06H, UP	multiply till B < R0
	MOV A,R3	store the LSB of result in Register A
	MOV B,R2	store the MSB of result in register B
	END	

Program 1.33.25

Find the contents of Register A after execution of following set of instructions.

```
MOV A, #6AH
MOV R4, #6EH
XRL A, R4
CLR C
MOV A, #4DH
SWAP A
RRC A
RRC A
RRC A
RRC A
```

Program

Instruction Sequence	Contents of Register A
MOV A, #6AH	A = 6A
MOV R4, #6EH	(R4 has been loaded, no effect on A)
XRL A, R4	A = 01101010 (XOR) 01101110 = 00000100 = 04
CLR C	(Carry has become 0, No effect on A)
MOV A, #4DH	A = 4D
SWAP A	Nibbles swapped, so A = D4
RRC A	A = 11010100 – 1 bit right shift through carry = 01101010 = 6A
RRC A	A = 01101010 – 1 bit right shift through carry = 00110101 = 35
RRC A	A = 00110101 – 1 bit right shift through carry = 10011010 = 9A
RRC A	A = 10011010 – 1 bit right shift through carry = 01001101 = 4D

Therefore at the end of the execution sequence, the Register A would contain **4D (Hexadecimal)**

Program 1.33.26

Write an assembly language program to obtain 1's complement of the given number.

Soln. : Let the number be stored in memory location 30h and the result (1's complement of the number) is stored in memory location 31h.

Label	Instruction	Comment
	MOV A, 30h	Load the Number from the Internal Data Memory location 30h
	CPL A	Generate 1's complement of the number by CPL instruction
	MOV 31h, A	Store the result in memory location 31h
HERE :	SJMP HERE	Done
	END	

Program 1.33.27

Write an ALP to convert packed BCD number 35 into HEX number and store the result in memory locations 50h and 51h.

Soln. :

Algorithm steps

- We need to divide packed BCD number 35 by 16.
- This will result in MSB of the hex result being obtained from the Quotient of the Division. (A = 02h)
- The LSB of the Hexadecimal number would be obtained from the remainder. (B = 03h)
- Now, we need to swap nibbles of A (20h) and then OR the result with Register B. (so A = 23h)
- 23h is Hex result of 35 packed BCD, we store it at Memory locations 50h and 51h.

Label	Instruction	Comment
	MOV A, #35h	Load the Number 35 packed BCD in Register A
	MOV B, #10h	Load divisor 16 decimal = 10h in Register B
	DIV AB	Perform division, returning Quotient in A, Remainder in B
	SWAP A	Swap nibble of A
	ORL A, B or ORL A, 0F0h (B – SFR)	OR the contents of A and B
	MOV 50h, A	Move Hex result to 50h
	MOV 51h, A	Move Hex result to 51h
HERE:	SJMP HERE	Done
	END	

Program 1.33.28 **SPPU - Dec. 2012, 4 Marks.**

Write assembly code to select R7 register from bank 3 to store hex value 08h available in RAM memory location 20h.

Soln. :

Instruction	Comment
SETB PSW.4	Select register bank 3 by making RS1 = 1 and RS0 = 1
SETB PSW.3	
MOV R1, 20H	
MOV A, @R1	A = 08H
MOV R7, A	R7 = 08H

□□□

UNIT - II

CHAPTER 2

IO Port Interfacing - I

Syllabus Topic : Interfacing of LEDs

2.1 Interfacing of LEDs

It is a human oriented output peripheral. It is used to display result or operand. One may use CRT, LED or LCD displays.

A CRT is used to display large amount of data. LED and LCD displays are used to display small amount of data. The commonly used LED displays are numeric displays.

LED Displays

To drive a LED, there are two methods.

Method 1)

Connect the cathode of LED to ground. Connect the anode of LED to port pin of 8051, through a resistor as shown in Fig. 2.1.1.

This method requires 8051 to source a huge amount of current required by the LED i.e. around 20 mA. But 8051 is not capable of sourcing a current more than 2 mA. This will make the LED glow very dim.

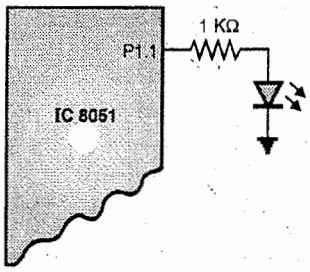


Fig. 2.1.1 : LED driven by 8051 port pin (Method 1)

Method 2)

Connect the anode of LED to V_{CC} through resistor. Connect the cathode of LED to the port pin of 8051 as shown in the Fig. 2.1.2.

This method requires 8051 to sink a huge current required by LED i.e. 20 mA. 8051 can sink huge currents and hence it makes LED glow brighter. Hence we will always use method 2, to interface LED.

Note : Source current is current to be sourced i.e. given or provided Sink current is the current to be sinked i.e. connected to ground or given a path to ground. Every microcontroller has a better current sinking capability than its current sourcing capability.

The LED displays are available in two common formats : seven segment display and 5 by 7 dot matrix displays.

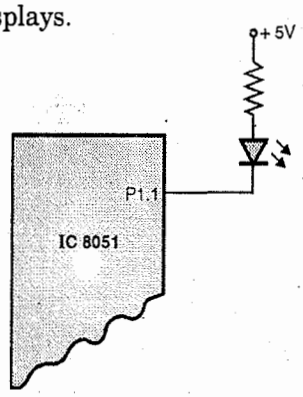


Fig. 2.1.2 : LED driven by 8051 port pin (Method 2)

Ex. 2.1.1

Port 1 of 8051 is to be connected to two on-off switches and two LEDs. It is required to sense the status of the switches and indicate it through the LEDs.

Write a program to achieve this task and give the essential interfacing details.

Soln. :

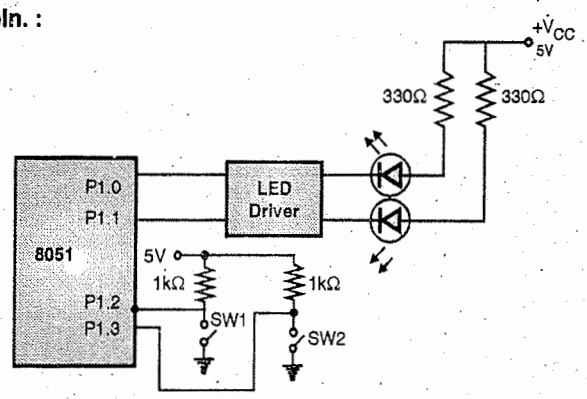


Fig. P. 2.1.1

P1.0 and P1.1 lines are used to connect the LEDs, while the P1.2 and P1.3 lines of Port 1 are used to connect the switches.

The status of the switches is sensed by the instruction BIT TEST instruction. The LEDs are driven by the instruction BIT SET.

Program

Label	Instruction	Comments
	MOV P1, #0CH	Initialize P1.2 and P1.3 lines as input and switch off the LEDS.
BACK:	JNB P1.2, L1	If bit is set, glow LED1
	CLR P1.0	
	SJMP over	
L1:	SETB P1.0	
over:	JNB P1.3,L2	If bit is set, glow LED2
	CLR P1.1	
	SJMP BACK	Keep polling
L2:	SETB P1.1	
	SJMP BACK	Keep polling

Ex. 2.1.2 **Lab Assignment**

Write an assembly language program to interface an LED to pin P1.0 and flash it after every 1 ms. Assume XTAL = 11.0592 MHz.

Soln. :

$$XTAL = 11.0592 \text{ MHz}$$

$$\therefore \text{Timer clock frequency} = \frac{11.0592 \text{ MHz}}{12}$$

$$= 921.6 \text{ KHz}$$

$$\therefore \text{Timer for 1 machine cycle} = \frac{1}{921.6 \text{ KHz}}$$

$$= 1.085 \ \mu\text{sec.}$$

$$\text{Delay period} = 1 \text{ msec.}$$

Let us determine count to get a delay of 250 μsec.

$$\therefore \text{we need } \frac{250 \ \mu\text{s}}{1.085 \ \mu\text{sec}} = 230 \text{ clocks}$$

$$\therefore \text{Count} = 65536 - 230 = 65306 = \text{FF1AH}$$

$$\text{TH} = \text{FFH, TL} = \text{1AH}$$

To get a delay of 1 msec, the delay loop of 250 μs is executed for 4 times.

Delay routine

Label	Instruction	Comments
Delay :	MOV R0, #04 H	Initialize counter to 4
	MOV TL0, #1A H	Load TL0
	MOV TH0, #FFH	Load TH0
	SETB TR0	Start-timer 0
L2 :	JNB TF0, L2	Remain until timer rolls over
	CLR TR0	Stop timer 0
	CLR TF0	Clear timer 0 flag
	DJNZ R0, L1	Decrement R0 and if R0 ≠ 0 repeat
	RET	

Main program

Label	Instruction	Comments
	MOV TMOD, #01 H	Timer 0, mode 1
HERE :	CPL P1.0	Toggle 1.0
	ACALL DELAY	Wait for 1 msec.
	SJMP HERE	

Ex. 2.1.3

Design a 8051 based system to blink a LED at a frequency of 1Hz using interrupts. Also write the corresponding assembly program.

Soln. : Design

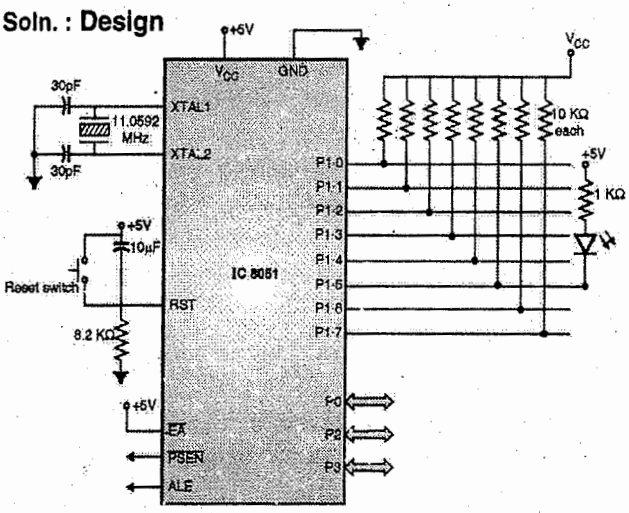


Fig. P. 2.1.3 : Interface diagram

Program

Since we want LED to be blinking at a frequency of 1Hz, we need to basically provide a square wave of frequency of 1Hz to it. i.e. 500 msec on time and 500 msec off time.

We need a delay of 500 msec as seen from the Fig. P. 2.1.3(a). The output should be logic '1' for 500 msec and should to logic '0' for 500 msec.

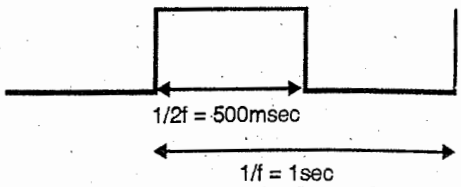


Fig. P. 2.1.3(a) : Required waveform

1 machine cycle = 1.085 μsec.

∴ for delay of 500 msec, no of machine cycles

$$= \frac{500 \text{ msec}}{1.085 \ \mu\text{sec}} \cong (460829)_{10} = (7081D)_{16}$$

But the 16 bit timer of 8051 can have a maximum of 16 bit count i.e. $(FFFF)_{16}$

Hence we will design a system with 50 msec delay and execute it 10 times to get a delay of 500 msec.

For delay of 50 msec no of machine cycle $= \frac{50 \text{ msec}}{1.085 \ \mu\text{sec}} \cong (46083)_{16} = (B403)_{16}$

$\therefore$ Counter is to be initialized as $(FFFF)_{16} - (B403)_{16}$
$= (4BFC)_{16}$

Algorithm

A) Main Program

Step I : Initialize the count to 10, for calling 50msec delay 10 times and hence getting a 500 msec delay.

Step II : Set P1.5 pin to logic 1

Step III : Enable timer 0 and global interrupts

Step IV : Initialize TMOD for timer 0 in timer mode 1

Step V : Load TL0 and TH0 with the count calculated

Step VI : Set TR0 bit to run timer 0

Step VII : Do nothing. Wait for interrupt.

B) Timer 0 ISR

Step I : If count has become 0, toggle P1.5 pin and reinitialize count to 10.

Step II : Stop timer 0 (i.e. TR0 = 0) and reload timer 0 with count calculated.

Step III : Set TR0 to restart timer and clear timer 0 overflow flag.

Step IV : Decrement the count

Registers value

1) Interrupt Enable (IE) Register

→ To enable global interrupt and Timer 1 interrupt.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	0	0	0	1	0

8 2

$\therefore$ IE = 0x82

2) Timer Mode (TMOD) Register

→ To initialize Timer / Counter 0 as timer

→ To initialize Timer 0 in mode 1

Timer 1				Timer 0			
D7	D6	D5	D4	D3	D2	D1	D0
GATE	$C/\bar{T}$	M1	M0	GATE	$C/\bar{T}$	M1	M0
0	0	0	0	0	0	0	1

0 1

$\therefore$ TMOD = 0x01

3) Count value = $(4BFC)_{16}$

$\therefore$ TL0 = 0xFC and TH0 = 0x4B

Assembly Program

```
org 0000H
LJMP    Start
org 000BH                   //ISR for timer 0.
MOV A,R5
CJNE A,#00,Next             //If 10 times 50msec delays
                            over toggle P1.5 pin and
                            reinitialize count to 10.
CPL P1.5
MOV R5,#0AH
Next:
CLR TR0                     //Clear Timer 0 run bit.
MOV TL0,#0FCH               //Load the count for a delay of
                              50msec in TL0 and TH0.
MOV TH0,#4BH
SETB TR0                    //Set the Timer 0 in run mode.
CLR TF0                     //Clear Timer 0 overflow flag.
DEC R5                      //Decrement the variable count
                            //    after every 50msec.
RETI
org 1000H
Start:
MOV R5,#0AH                 //Initialise the variable count to 10.
SETB P1.5                   //Set P1.5 pin to logic 1.
MOV IE,#82H                 //Enable Timer 0 interrupt and
                            // global interrupt.
MOV TMOD,#01H               //Initialize timer 0 as timer and
                              in mode 1.
MOV TH0, 4BH                //Load the count for 50msec in
                            // the TL1 and TH1 registers.
MOV TL0, 0FCH
SETB TR0                    //Set Timer 0 in run mode.
here: SJMP here             //Do nothing loop. Wait
                            //    for interrupts.
end
```

Output : The Fig. P. 2.1.3(b) shows the output of the above program.

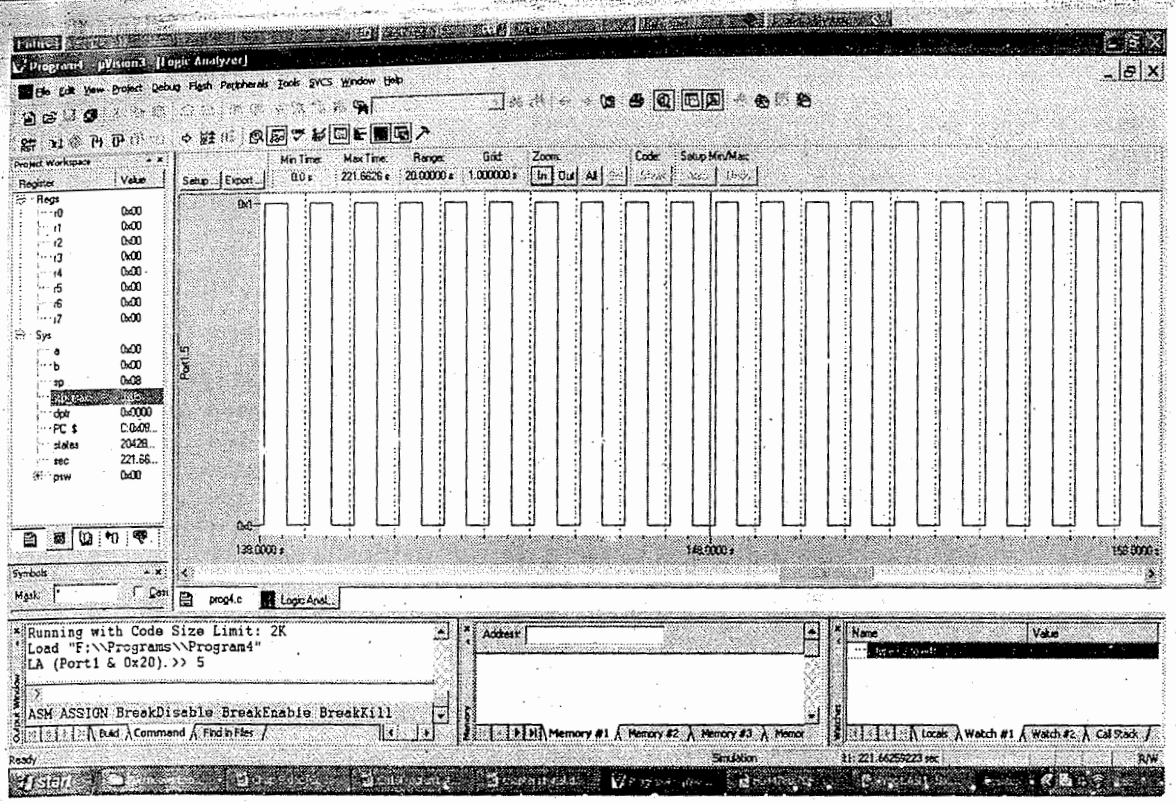


Fig. P. 2.1.3(b) : Output

2.2 Interfacing DIP Switches

- Generally a group of DIP switches comprises four or eight switches. An input port with eight pins e.g. : port 0, port 1, port 2 uses eight DIP switches.
- Fig. 2.2.1 shows the interfacing of DIP switches to 8051 microcontroller.

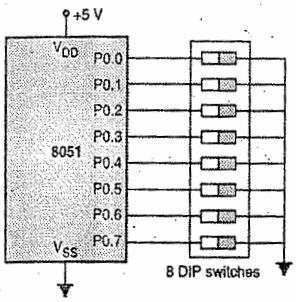


Fig. 2.2.1 : Interfacing 8 DIP switches to port 0 of 8051

2.3 Interfacing Push Button Switches to 8051

- Fig. 2.3.1(a) shows a single push button switch connected to P1.3 of microcontroller 8051. If key is not pressed, there is no connection between the points A and B. The port P1.3 pin is read as input high. This is because the 8051 ports P1, P2 and P3 have internal pull up registers.

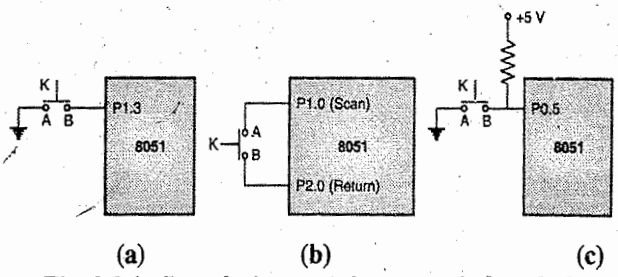


Fig. 2.3.1 : Interfacing push button switch to 8051

- If push button K is not pressed, P1.3 is read as logic 1.
- However if K is pressed the points A and B are shorted, connecting P1.3 pin to GND. Thus, by reading the status of port pin P1.3 we can determine whether the push button is pressed or not. If K is pressed (P1.3 = 0) and if not pressed (P1.3 = 1).
- The push button can also be connected as shown in Fig. 2.3.1(b) such that one line is the input line (Return line) and the other is output line (scan line). Let P1.0 be scan line and P2.0 is

the return line, that provides the status whether key is pressed or not. Whenever key K is to be checked P1.0 is grounded. If P2.0 = 0, key is not pressed and if P2.0 = 1 key is pressed. This method is useful when P1.0 = 1 then P2.0 = 1 for both conditions i.e. for matrix keyboard.

- The push button can also be connected to port 0 as shown in Fig. 2.3.1(c). Port 0 is an open drain port. Hence, we connect an external 10 KΩ resistor at point B.

Ex. 2.3.1

A switch is connected to pin P2.0 and an LED to pin P2.4. Write a program to get the status of the switch and send it to the LED.

Soln. : Program :

Label	Instruction	Comments
	SETB P2.0	P2.0 = input
L1:	MOV C, P2.0	Read the status of switch into CF
	MOV P2.4, C	Send the switch status to LED
	SJMP L1	Keep repeating

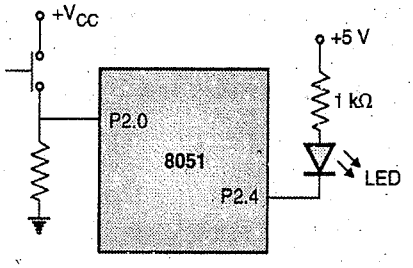


Fig. P. 2.3.1

Syllabus Topic : Interfacing of Keypad

2.4 Interfacing of Keypad

2.4.1 Keyboard/Keypad

Keyboard is a human oriented input peripheral. It is used to input data or program into the microcomputer. It consists of push button type switches. When a key is pressed, the microcontroller identifies key depression and then performs appropriate operation.

2.4.2 Key Switch Mechanism

Q. What is key debounce ?

The aim of this mechanism is to generate and transmit a code each time a key is pressed. The mechanism should send one and only proper code, when the key is pressed. Fig. 2.4.1 shows the general operation of a keyboard.

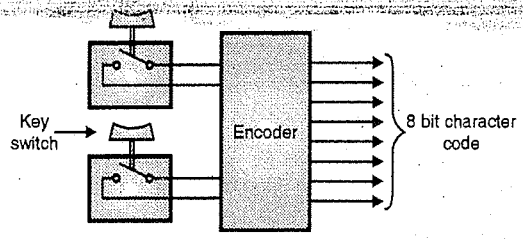


Fig. 2.4.1 : General operation of a keyboard

The input keyboard is composed of a set of labelled push button switches. Each switch makes electrical contact when pressed. The nature of the contact should be reliable, have long life and feel right.

In case of a push button key, the metal contact bounces few times, hence the voltage across the switch fluctuates and generates spikes in the signal. Therefore, it is necessary to debounce the mechanical switches. The key debouncing is done through hardware and software. Fig. 2.4.2 shows the bouncing of key switch.

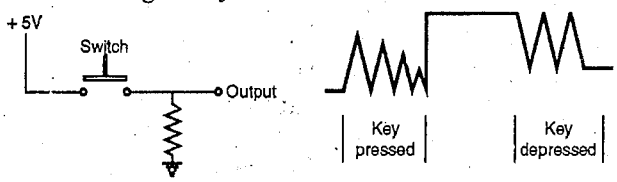


Fig. 2.4.2 : Bouncing of key switch

2.4.3 Hardware Key Debouncing

Q. Explain hardware key debounce technique.

It is implemented by using flip-flop or latch. Fig. 2.4.3 shows a circuit diagram of hardware key debouncing.

When the switch is connected to A, the output of the latch goes high. When the key makes contact with B, the output changes from logic 1 to logic 0. The wiper bounces many times on contact B, but the output does not fluctuate between logic 1 and logic 0. When the wiper is not connected either to A or B, the output of the latch remains constant.

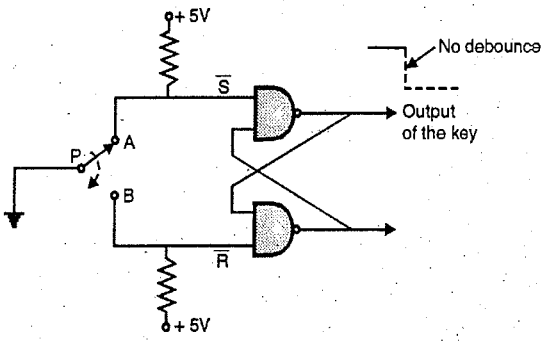


Fig. 2.4.3 : Hardware key debouncing

2.4.4 Software Key Debouncing

Q. Explain software key debounce technique.

In the software technique the microcontroller waits for 20 ms before it accepts the key as an input..

If after 20 ms the key is pressed the key is accepted by microcontroller.

The process of software key debouncing is as shown in Fig. 2.4.4.

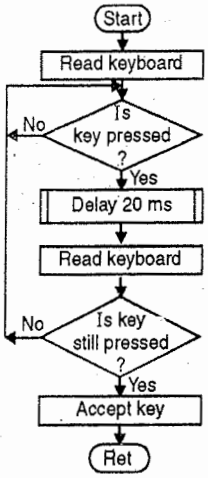


Fig. 2.4.4

2.4.5 Keyboard Interface Circuit

The keyboard is interfaced with microcontroller through input ports. The keyboard consists of mechanical switches. These switches are arranged in non-matrix or matrix form.

2.4.5(A) Non-matrix Type Keyboard

In non-matrix type keyboard, the key closure is identified by reading the port data, but it requires many port lines. The number of I/O lines is equal to number of keys. Fig. 2.4.5 shows the interfacing of octal non-matrix type keyboard.

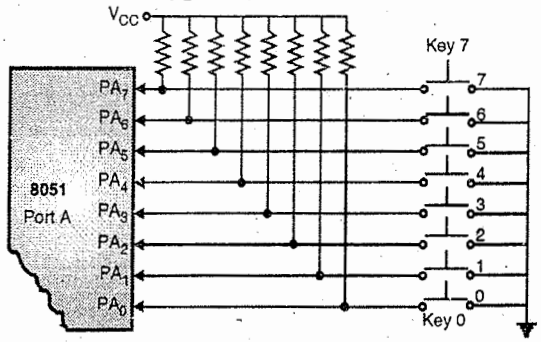


Fig. 2.4.5 : Non matrix type keyboard

To identify the key value the following three functions should be performed :

(1) Identifying a key closure.

(2) Debouncing the key.

(3) Encoding the key to an appropriate code like hexadecimal.

The above three functions can be performed through hardware as well as software. As an example we will see hardware technique for identification of key closure. The interfacing is as shown in Fig. 2.4.6.

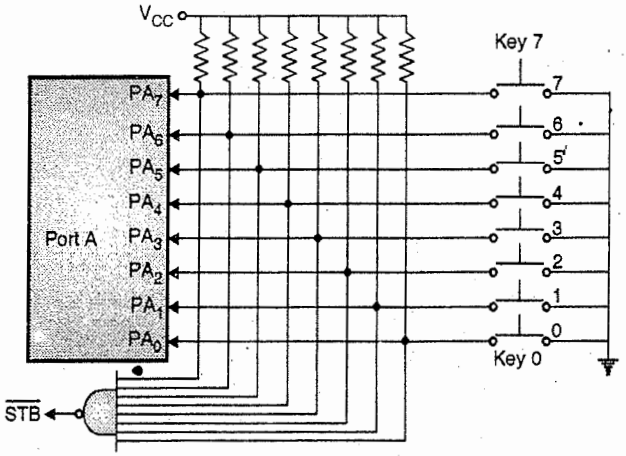


Fig. 2.4.6 : Hardware technique of identification

When all keys are open, the output of NAND gate ($\overline{STB}$) goes low. When one of the keys is pressed, the output of NAND gate ($\overline{STB}$) goes high. The $\overline{STB}$ is used to identify that the key is pressed. This $\overline{STB}$ signal can be used to interrupt the microcontroller.

2.4.5(B) Matrix Keyboard Interface

In a simple keyboard interface one input line is required to interface one key and this increases the number of keys. When a large number of keys are to be interfaced, this technique is not useful. Matrix method is used in such cases, so that the number of connections are reduced.

Fig. 2.4.7 shows 16 keys arranged in 4 rows and 4 columns. No connections is there, when the keys are open. If a key is pressed then there is connection between corresponding rows and columns. Such a matrix requires eight lines to complete the connections.

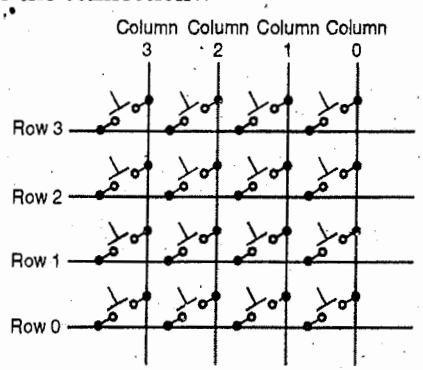


Fig. 2.4.7 : Matrix keyboard

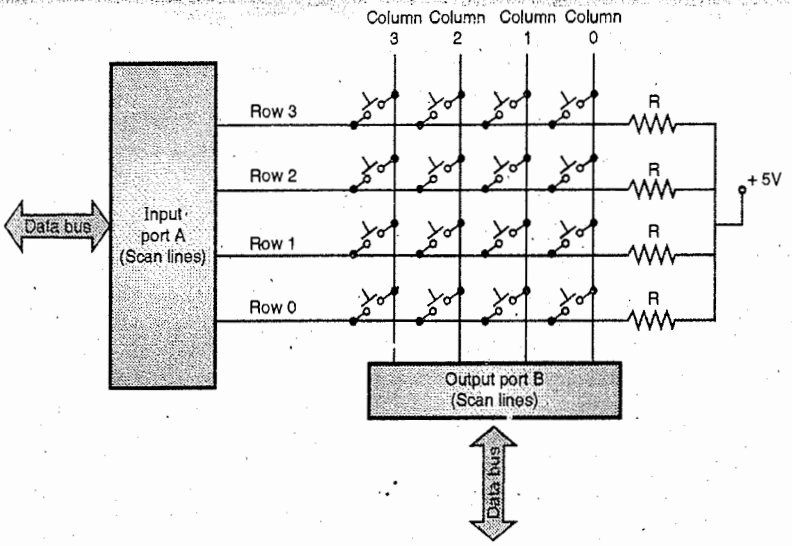


Fig. 2.4.8 : Matrix keyboard connections

If non matrix type connection is used then 16 lines will be required. So using method reduces the number of connections.

Fig. 2.4.8 shows the interfacing of a matrix keyboard, it requires two ports : an input port and an output port. The columns are referred to as scan lines and rows are referred to as return lines.

When a key is pressed, the corresponding row and column are connected i.e. they are shorted. If the output line of a row is high, then it makes the line of a column high and vice versa. The key is recognized by data which is sent on the output port and the input code that is received from the input port. The steps required to identify the pressed key are,

i) To identify if any key is pressed or not.
   (a) All the column lines are made zero by sending low on all the output lines. i.e. all the keys in the keyboard matrix are activated.
   (b) Read the status of rows i.e. return lines. If the status of all lines is logic high, the key is not pressed. Otherwise if the status of all lines is logic low, the key is pressed.

ii) Debouncing the key. (Using software debouncing as explained earlier)

iii) Identifying the pressed key.
   (a) Activate the keys from one column by making one column line zero.
   (b) Read the status of return lines. The zero on any return line indicates that key is pressed.
   (c) Activate the keys from next column and repeat steps (b) and (c) for all the columns.

Ex. 2.4.1

Interface a simple keyboard to microcontroller 8051.

Soln. :

Fig. P. 2.4.1 shows how a simple keyboard is interfaced to microcontroller 8051.

As shown in Fig. P. 2.4.1 eight keys are connected to port 1 pins. Each port pin gives the status of key that is connected to that pin.

If a pin shows logic 1 then the key is open otherwise the key is closed.

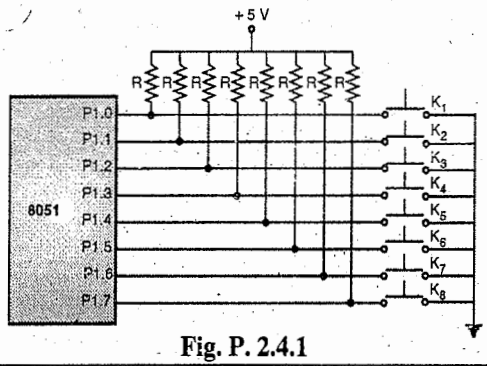


Fig. P. 2.4.1

Ex. 2.4.2

A 4 × 4 matrix keyboard is connected to two ports of 8051 microcontroller. Draw the flowchart for determining which key is pressed and obtain scan code from look-up table.

Soln. : Fig. P. 2.4.2 shows how a 4 × 4 matrix keyboard is connected to microcontroller 8051. The 4 × 4 matrix keyboard is connected to ports P0 and P1 of 8051. P1.0 to P1.3 are used as scan lines and P0.0 to P0.3 are used as return lines.

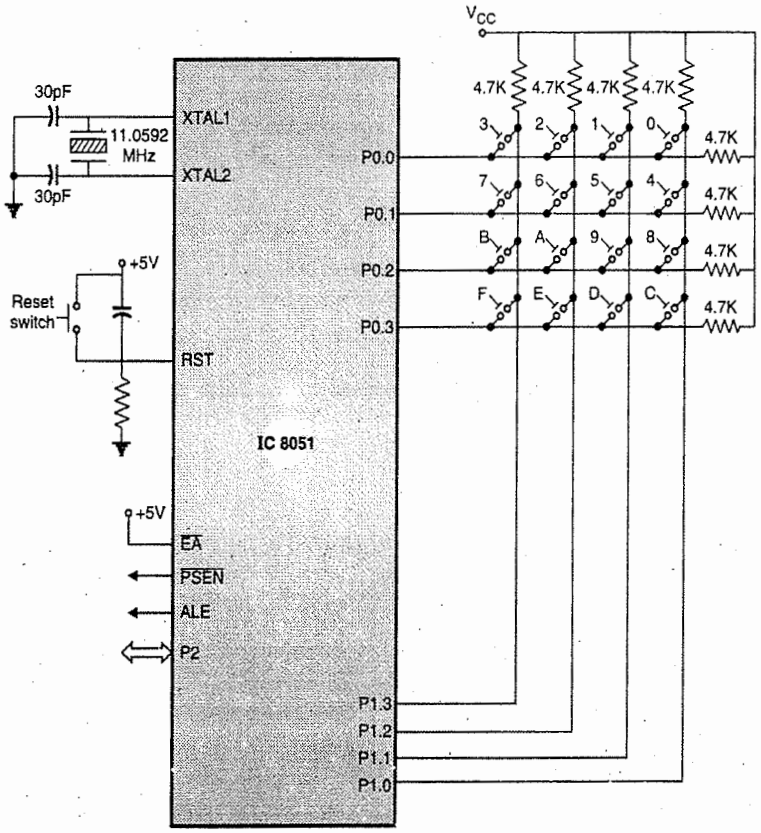


Fig. P. 2.4.2 : Interfacing 4 × 4 matrix keyboard to 8051

Program

Label	Instruction	Comments
	MOV P1, #0FFH	Make P1 an input port
L1 :	MOV P0, #00H	Ground all the rows
	MOV A, P1	Read all columns. Ensure all keys are open
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, L1	Check till all keys are released
L2 :	ACALL DELAY	Call 20 ms delay
	MOV A, P1	See if any key is pressed
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, L3	Key pressed await closure
	SJMP L2	Check if key is pressed
L3 :	ACALL DELAY	Wait for 20 ms debounce time
	MOV A, P1	Check key closure
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, L4	Key pressed, find row
	SJMP L2	If none, keep polling
L4 :	MOV P0, #11111110B	Ground row 0
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_0	Read row 0, find the column
	MOV P0, #11111101B	Ground row1
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_1	Key row 1, find the column
	MOV P0, #11111011B	Ground row 2
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_2	Key row 2, find the column
	MOV P0, #11110111B	Ground row 3
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_3	Key row 3, find the column
	LJMP L2	If none, false input, repeat.
ROW_0	MOV DPTR, #KEYCODE0	Set DPTR = start of row 0
	SJMP FIND	Find column, key belongs to
ROW_1	MOV DPTR, #KEYCODE1	Set DPTR = start of row 1
	SJMP FIND	Find column, key belongs to
ROW_2	MOV DPTR, #KEYCODE2	Set DPTR = start of row 2
	SJMP FIND	Find column, key belongs to
ROW_3	MOV DPTR, #KEYCODE3	Set DPTR = start of row 3

Label	Instruction	Comments
	SJMP FIND	Find column, key belongs to
FIND :	RRC A	See if CY bit is 0
	JNC MATCH	If zero, get ASCII code
	INC DPTR	Point to next column address
	SJMP FIND	Keep searching
MATCH:	CLR A	Set A = 0 (Match is found)
	MOVC A, @A + DPTR	Get ASCII code from table
	MOV P2, A	Display pressed key
	LJMP L1	

ASCII Look up table for each row

	ORG 0300H					
KEYCODE0	DB	'0'	'1'	'2'	'3'	ROW 0
KEYCODE1	DB	'4'	'5'	'6'	'7'	ROW 1
KEYCODE2	DB	'8'	'9'	'A'	'B'	ROW 2
KEYCODE3	DB	'C'	'D'	'E'	'F'	ROW 3
END						

Ex. 2.4.3

Explain how microcontroller can be interfaced with keyboard. Draw flowchart for reading the code of key that has been pressed.

Soln. :

Fig. P. 2.4.3 shows how a 4 × 8 matrix keyboard is connected to microcontroller 8051. The 4 × 8 matrix keyboard is connected to port P0 and P1 of 8051. The P1.0 to P1.3 are used as scan lines and P0.0 to P0.7 are used as return lines.

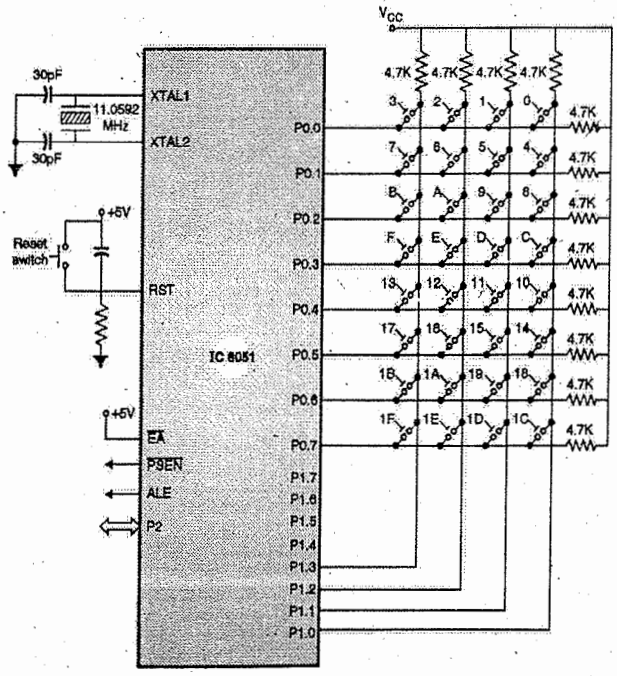


Fig. P. 2.4.3 : Interfacing 4 × 8 matrix keyboard to 8051

Program

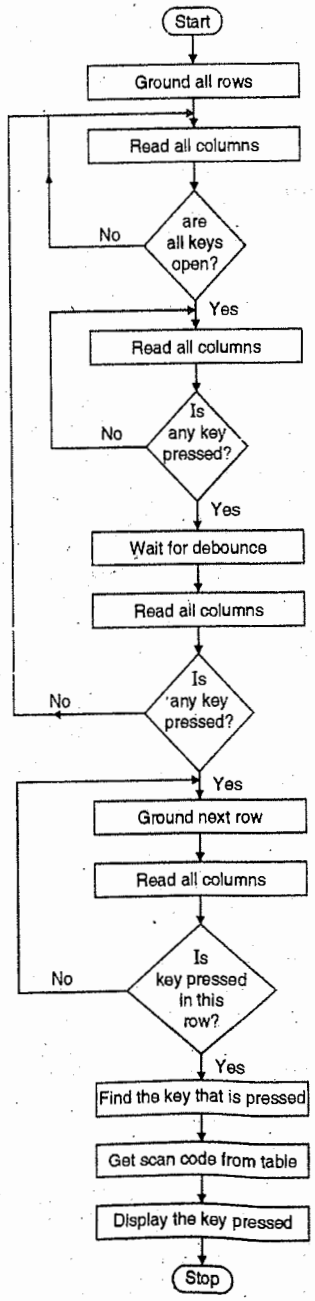
Label	Instruction	Comments
	MOV P1, #0FFH	Make P1 an input port
L1 :	MOV P0, #00H	Ground all the rows
	MOV A, P1	Read all columns. Ensure all keys are open
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, L1	Check till all keys are released
L2 :	ACALL DELAY	Call 20 ms delay
	MOV A, P1	See if any key is pressed
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, L3	Key pressed await closure
	SJMP L2	Check if key is pressed
L3 :	ACALL DELAY	Wait for 20 ms debounce time
	MOV A, P1	Check key closure
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, L4	Key pressed, find row
	SJMP L2	If none, keep polling
L4 :	MOV P0, #11111110B	Ground row 0
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_0	Read row 0, find the column
	MOV P0, #11111101B	Ground row1
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_1	Key row 1, find the column
	MOV P0, #11111011B	Ground row 2
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_2	Key row 2, find the column
	MOV P0, #11110111B	Ground row 3
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_3	Key row 3, find the column
	MOV P0, #11101111B	Ground row 4
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_4	Key row 4, find the column
	MOV P0, #11011111B	Ground row 5
	MOV A, P1	Read columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_5	Key row 5, find the column

Label	Instruction	Comments
	MOV P0, #10111111B	Ground row 6
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_6	Key row 6, find the column
	MOV P0, #01111111B	Ground row 7
	MOV A, P1	Read all columns
	ANL A, #00001111B	Mask unused bits
	CJNE A, #00001111B, ROW_7	Key row 7, find the column
	LJMP L2	If none, false input, repeat.
ROW_0	MOV DPTR, #KEYCODE0	Set DPTR = start of row 0
	SJMP FIND	Find column, key belongs to
ROW_1	MOV DPTR, #KEYCODE1	Set DPTR = start of row 1
	SJMP FIND	Find column, key belongs to
ROW_2	MOV DPTR, #KEYCODE2	Set DPTR = start of row 2
	SJMP FIND	Find column, key belongs to
ROW_3	MOV DPTR, #KEYCODE3	Set DPTR = start of row 3
	SJMP FIND	Find column, key belongs to
ROW_4	MOV DPTR, #KEYCODE4	Set DPTR = start of row 4
	SJMP FIND	Find column, key belongs to
ROW_5	MOV DPTR, #KEYCODE5	Set DPTR = start of row 5
	SJMP FIND	Find column, key belongs to
ROW_6	MOV DPTR, #KEYCODE6	Set DPTR = start of row 6
	SJMP FIND	Find column, key belongs to
ROW_7	MOV DPTR, #KEYCODE7	Set DPTR = start of row 7
FIND :	RRC A	See if CY bit is 0
	JNC MATCH	If zero, get ASCII code
	INC DPTR	Point to next column address
	SJMP FIND	Keep searching
MATCH :	CLR A	Set A = 0 (Match is found)
	MOVC A, @A + DPTR	Get ASCII code from table
	MOV P2, A	Display pressed key
	LJMP L1	

ASCII Look up table for each row

	ORG 0300H					
KEYCODE0	DB	'0'	'1'	'2'	'3'	ROW 0
KEYCODE1	DB	'4'	'5'	'6'	'7'	ROW 1
KEYCODE2	DB	'8'	'9'	'A'	'B'	ROW 2
KEYCODE3	DB	'C'	'D'	'E'	'F'	ROW 3
KEYCODE4	DB	'10'	'11'	'12'	'13'	ROW 4
KEYCODE5	DB	'14'	'15'	'16'	'17'	ROW 5
KEYCODE6	DB	'18'	'19'	'1A'	'1B'	ROW 6
KEYCODE7	DB	'1C'	'1D'	'1E'	'1F'	ROW 7
END						

Flowchart : Refer Flowchart 1.



Flowchart 1

2.5 Interfacing of Seven Segment Display Device (SSD)

- As shown in the Fig. 2.5.1 the seven segment display uses seven LEDs to make any digit. If all the LEDs are on, it shows the digit 8. There are two types SSDs available.

(i) Common cathode i.e. the cathode of all the LEDs are given as a common pin. In this case the anode is connected to the port pins. As already discussed in the section for LEDs, this method requires port pins to source large current. But 8051 cannot source current beyond 2 mA.

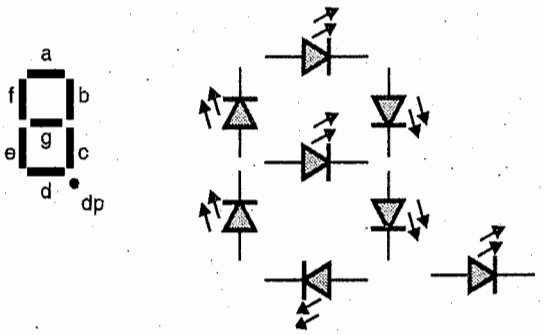


Fig. 2.5.1 : Structure of Seven Segment Display (SSD)

(ii) Common anode i.e. the anode of all the LEDs are given as a common pin. In this case the cathode is connected to the port pins. Hence port pins have to sink current of 20 mA.

Hence we use common anode SSD, and connect it to 8051 as shown in Fig. 2.5.2.

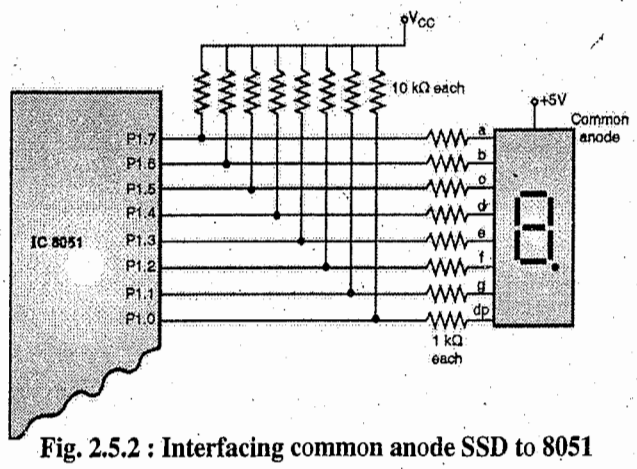


Fig. 2.5.2 : Interfacing common anode SSD to 8051

- The seven segments are labelled a to g and dp represents decimal point. By forward biasing different LED segments we can display the digits 0 through 9.

- The Table 2.5.1 illustrates the binary and hexadecimal data given to the pins of SSD, to display different digits. A pin should be given logic '0', to switch on the corresponding LED.

Table 2.5.1 : Code for 0 to F given to Common Anode SSD

Digit	a	b	c	d	e	f	g	dp	Hexadecimal form
0	0	0	0	0	0	0	1	1	0x03
1	1	0	0	1	1	1	1	1	0x9F
2	0	0	1	0	0	1	0	1	0x25
3	0	0	0	0	1	1	0	1	0x0D
4	1	0	0	1	1	0	0	1	0x99
5	0	1	0	0	1	0	0	1	0x49
6	0	1	0	0	0	0	0	1	0x41
7	0	0	0	1	1	1	1	1	0x1F
8	0	0	0	0	0	0	0	1	0x01
9	0	0	0	0	1	0	0	1	0x09
A	0	0	0	1	0	0	0	1	0x11
B	1	1	0	0	0	0	0	1	0xC1
C	0	1	1	0	0	0	1	1	0x63
D	1	0	0	0	0	1	0	1	0x85
E	0	1	1	0	0	0	0	1	0x61
F	0	1	1	1	0	0	0	1	0x71

Syllabus Topic : Interfacing of 7-segment Multiplexed Display

2.6 Interfacing of 7-segment Multiplexed Display

To drive multiple seven segment displays, it is difficult to spare separate port for each SSD. For example to interface 4 SSDs we will require 4 ports i.e. all the ports of 8051 will be used. In this case, no other device can be interfaced to 8051. Hence we multiplex the SSDs, as shown in the Fig. 2.6.1.

Now to control these SSDs we utilize one of the characteristic of human eye i.e. persistence of vision. We will give the data for the first SSD (i.e. thousands place) and enable its CA (Common Anode) pin, keeping the CA pin of remaining SSDs disabled. A PNP transistor is connected to each of the CA pin. Hence, when we provide a 0V (i.e. logic '0') to the base of the transistor it will be 'ON' and the corresponding SSD will be selected. For e.g. when we give P2 = 0x07, the first SSD is enabled, while others are disabled. Similarly we will give the data on P1 pins for second SSD and enable its CA pin, keeping the CA pin of others disabled. This can be achieved by giving 0x0B on P2. Similarly for third and fourth SSD.

This entire procedure will be repeated continuously after every 1/50th of a second (i.e. every 20 msec), so that human eye feels all the SSDs to be displaying continuously.

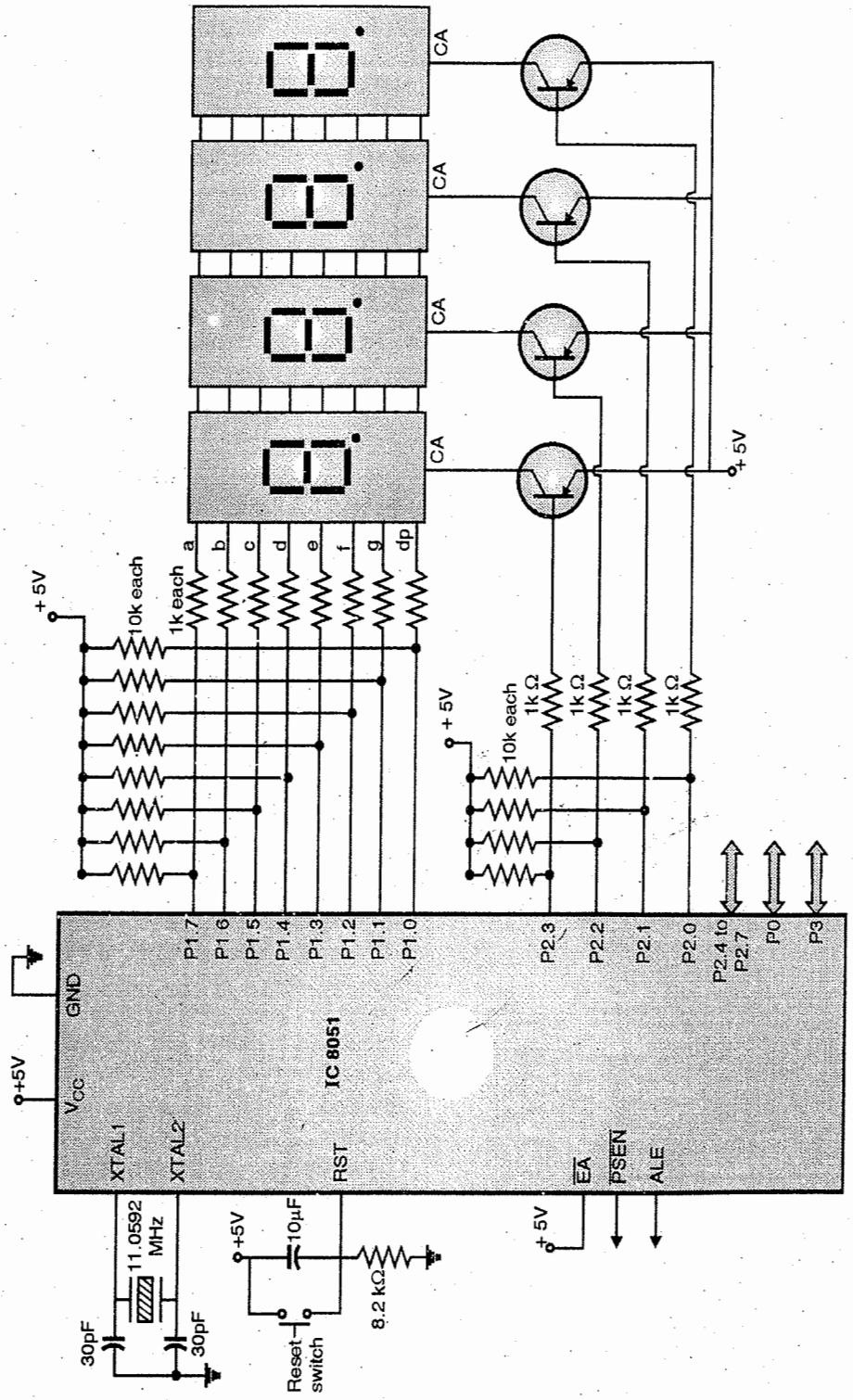


Fig. 2.6.1 : Interfacing 7 segment multiplexed display

Ex. 2.6.1 Lab Assignment

Write a program to display message on an 8 bit 7 segment LED display that is interfaced through Port 1 and Port 3 of 8051.

Soln. : Fig. P. 2.6.1 shows a multiplexed 8 digit 7 segment LED display connected in 8051 system using Port 1 and Port 3. Both the ports are used as output ports. Transistors are used to drive LED segments.

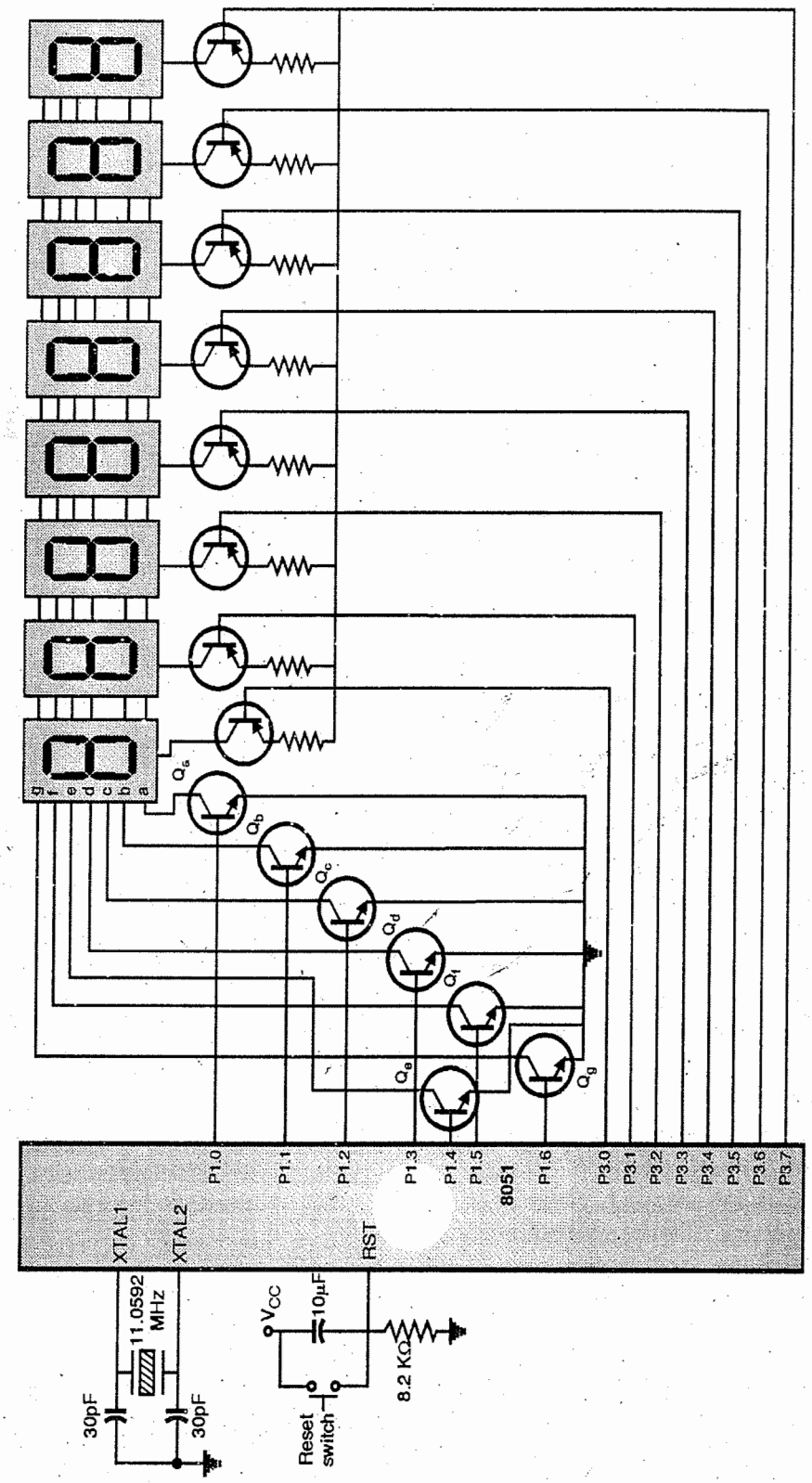


Fig. P. 2.6.1

Subroutine to display message :

Label	Instruction	Comments
	MOV R0, #08 H	Initialize counter for 8 digits
	MOV R1, #7FH	Load the select pattern
	MOV DPTR, #3000H	Load starting address of message to be displayed
L1 :	MOV P3, R1	Select digit
	MOVX A,@DPTR	Get send
	MOV P1, A	Send data to Port A for display
	LCALL DELAY	Wait sometime
	MOV A,R1	Adjust select in pattern
	RR A	
	MOV R1, A	
	INC DPTR	Increment message pointer
	DJNZ R0, L1	
	RET	

Syllabus Topic : Interfacing of LCD

2.7 Interfacing of Liquid Crystal Display (LCD)

- LCD displays are widely used because of its low current consumption as compared to SSD. Also that LCD can be used to display any character as it uses a 5 × 7 dot matrix to display.
  For e.g. to display '1' of LCD as shown in Fig. 2.7.1.
- An LCD allows the user to output a specific message making the application more user friendly and attractive.
- LCDs are invaluable for displaying status messages and information while a program is being debug.
- The LCDs generally use a common controller chip, Hitachi 44780 and common connector interface. Due to these factors, the alphanumeric LCDs range in size from 8 characters to 80 characters. All the characters are interchangeable without any hardware or software changes. They are arranged in 40 by 2 or 20 by 4 or 10 by 2 or 20 by 1 or 20 by 2. The first figure represents the number of characters in each line and second figure represents the number of lines the display has.

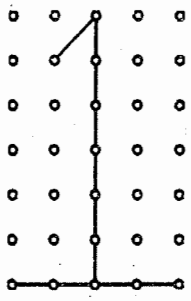


Fig. 2.7.1 : Displaying 1 on LCD display of a 5 × 7 dot matrix

A typical 16 by 2 (i.e. 16 characters and 2 such lines) LCD looks as shown in Fig. 2.7.2.

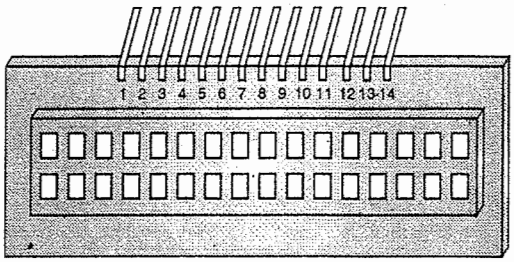


Fig. 2.7.2 : Structures of 16 × 2 LCD

2.7.1 LCD Pin Description

Q. Explain the functions of following pins of LCD :
i) RS ii) E iii) $\overline{R/W}$ iv) D0-D7

Some LCD have their pins on left or on bottom. The functions of the pins of LCD are listed in the Table 2.7.1.

Table 2.7.1 : Pin description of LCD

Pins	Symbol	Functions
1	V_{ss}	Ground
2	V_{dd} (or V_{cc})	+ 5 V supply
3	V_{EE}	Power supply to control contrast
4	RS	Should be '0' for instruction and '1' for data.
5	$\overline{R/W}$	Should be '0' to write and '1' to read
6	E	Enable display logic
7	D0	Data bus bit 0
8	D1	Data bus bit 1
9	D2	Data bus bit 2
10	D3	Data bus bit 3
11	D4	Data bus bit 4
12	D5	Data bus bit 5
13	D6	Data bus bit 6
14	D7	Data bus bit 7 (Also used as busy pin)

2.7.1(A) RS : Registers Select

- There are two registers of LCD viz. Instruction command code register and data register. The RS pin is used to select one of these registers.
- If RS = 0 the instruction command code register is selected and if RS = 1 the data register is selected, allowing user to send data to be displayed on the LCD.

2.7.1(B) R/$\overline{W}$: Read/Write

- R/$\overline{W}$ pin is used to read or write to LCD.
- If R/$\overline{W}$ = 0 the user can write information to the LCD and if R/$\overline{W}$ = 1 the user can read information.

2.7.1(C) Enable : E (It can also be denoted by EN)

- The enable pin E, is used to latch the data into the data or command register. When data is supplied, a high-to-low (negative edge) is required for LCD to latch the data.

2.7.1(D) D0 - D7 or DB0 - DB7

- The data pins D0 – D7 are used to send information to the LCD or read the contents of LCD internal registers.
- To display letters and numbers we send ASCII codes for letters A – Z, a – z and numbers 0 – 9 while making RS = 1.
- The ASCII code that is to be displayed is of 8 bits. It is send to the LCD in either nibbles or bytes i.e. 4 or 8 bits at a time.
- The two primary modes of operation to send parallel data are 4 or 8 bits.
- If four bit mode is used two nibbles of data are sent to do an 8 bit transfer. The "E" clock is used to initiate the data transfer. Atleast 6 I/O pins must be available for 4 bit mode.
- In 8 bit mode atleast 10 I/O pins must be used. This mode is used when application needs speed.

2.7.1(E) V_{CC}, V_{SS} and V_{EE}

V_{CC} – Provides +5V supply
V_{SS} – Ground
V_{EE} is used for controlling LCD contrast.

2.7.2 Cursor Addresses for LCDs

Table 2.7.2 gives the cursor addresses for common types of LCDs.

Table 2.7.2 : Cursor addresses for some LCDs

16 × 2 LCD	80	81	82	83	84	85	86	through	8F
	C0	C1	C2	C3	C4	C5	C6	through	CF
20 × 1 LCD	80	81	82	83	through	93			
20 × 2 LCD	80	81	82	83	through	93			
	C0	C1	C2	C3	through	D3			
20 × 4 LCD	80	81	82	83	through	93			
	C0	C1	C2	C3	through	D3			
	94	95	96	97	through	A7			
	D4	D5	D6	D7	through	E7			
40 × 2 LCD	80	81	82	83	through	A7			
	C0	C1	C2	C3	through	E7			
All data is in hex.									

2.7.3 LCD Command Codes

Q. List the LCD module commands.

- The list of commands that can be given to the LCD are as listed in the Table 2.7.3.

Table 2.7.3 : LCD commands

Hex command	Function
0x01	Clear display
0x02	Return cursor to home
0x04	Decrement cursor (i.e. shift cursor left)
0x06	Increment cursor (i.e. shift cursor right)
0x05	Shift display right
0x07	Shift display left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor on
0x0F	Display on, cursor on and blinking
0x10	Move cursor one position left
0x14	Move cursor one position right
0x18	Shift entire display left
0x1C	Shift entire display right
0x80	Move cursor to beginning of 1st line
0xC0	Move cursor to beginning of 2nd line
0x38	Initialize 2 line display of 5 × 7 matrix

The command codes can be used to display or force the cursor to home position or blink cursor.

Table 2.7.4 shows the command codes.

Table 2.7.4 : Command codes

Commands	RS	R/$\overline{W}$	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	Description
Clear display	0	0	0	0	0	0	0	0	0	1	It clears the entire display and sets display data RAM to address 0.
Return Home	0	0	0	0	0	0	0	0	1	–	It sets the display data RAM address to 0 It returns the cursor to home position. The display data RAM contents remain unchanged.
Entry Mode Set	0	0	0	0	0	0	0	1	1/D	S	It sets the direction for moving the cursor and specifies the shift of display. These operations are done during data read and data write.
	1/D = 1 increment 1/D = 0 decrement S = 1 accompanies display shift										
Display on/off control	0	0	0	0	0	0	1	D	C	B	It sets the entire display on/off, cursor on/off (0) and blink of cursor position character B
Cursor or Display shift	0	0	0	0	0	1	S/C	R/L	–	–	It moves cursor and display shifts without changing display data RAM contents.
	S/C = 1 display shift S/C = 0 cursor move R/L = 1 shift to the right R/L = 0 shift to the left										
Function Set	0	0	0	0	1	DL	N	F	–	–	It sets interface data length (DL), number of data lines (L) and character font (F)
	DL = 1, 8 bits DL = 0, 4 bits N = 1 2 lines N = 0 1 line			F = 0 : 5 × 7 dots F = 1 : 5 × 10 dots							
Set CG RAM address (character generator RAM)	0	0	0	1	ACG						Sets the character generator RAM address. The character generator RAM data is sent and received once this setting is done
	(ACG : CG RAM address)										
Set DD RAM address (display data RAM)	0	0	1	ADD							Sets the display data RAM address. The display data RAM data is sent and received after this setting is done
	(ADD : DD RAM address)										
Read Busy Flag and Address	0	1	BF	AC							Reads busy flag indicating if internal operation is being done and reads address counter contents
	AC : address counter for CG and DD RAM address BF = 0 can accept command or instruction BF = 1 busy in internal operation										
Write data to CG or DD RAM	1	0	write data								Writes data to CG or DD RAM
Read data from CG or DD RAM	1	1	read data								Reads data from CG or DD RAM

- In order to display a message on the LCD module we need to initialize the LCD. The LCD is initialized by writing command codes in the command register.
- Initialization comprises of command codes for clearing the display, shifting cursor automatically after writing a character, returning cursor home etc.
- After the initialization we can write data to the DD RAM or the CG RAM by issuing correct command and asserting the R/$\overline{W}$ signal low and RS signal high. The data is sent on the Port and a high to low pulse is applied on the E pin. The DD RAM stores the characters in their ASCII code. The CG RAM stores the character in its internally generated character code.
- Before sending the command or data it is essential to check busy flag i.e. whether the LCD is ready or not.

2.7.4 Initialization of LCD

The following algorithm is required to initialize and write data to LCD :

- Wait 1 second after power up for display to stabilize.
- Initialize the LCD by giving the instruction 0x38 to the command subroutine.

- Wait for 5 msec.
- Issue the command 0x0F to command subroutine for display on, cursor on and cursor blinking.
- Wait for 5 msec.
- Issue the command 0x01 for clearing display to command subroutine.
- Wait for 5 msec.
- Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.
- Wait for 5 msec.
- Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.
- Issue the data character one by one giving their ASCII values using data subroutine.

Command subroutine

- Give the instruction to the port connected to data bus of the LCD.
- Make RS = '0', to indicate instruction.
- Make R/$\overline{W}$ = '0', to indicate write.
- Make E = '1' } To give a high-to-low
- Wait for 120 µsec. } pulse on E pin so as
- Make E = '0' } to latch the command
- Return.

Data subroutine

- Check if LCD is ready by calling ready subroutine.
- Give the data to the port connected to the data bus of the LCD.
- Make RS = '1', to indicate data
- Make R/$\overline{W}$ = '0', to indicate write
- Make E = '1' } To give a high-to-low
- Wait for 120 µsec. } pulse on E pin so as
- Make E = '0' } to latch the data
- Return

Ready subroutine

- Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.
- Make RS = '0' to indicate instruction.
- Make R/$\overline{W}$ = '1', to indicate read.
- Make E = 0
- Make E = 1
- Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.
- Return.

2.7.5 Interfacing LCD Module with 8051

Q. Draw and explain interfacing diagram for 20 × 2 LCD module with 8051. List the LCD module command.

- Fig. 2.7.3 shows the interfacing of a 20 character × 2 line LCD module with 8051. As shown in Fig. 2.7.3 the data lines are connected to Port 1 of 8051. The control lines RS, R/$\overline{W}$ are driven by Port 3 pins P3.2, P3.3 and P3.4. The voltage at V_{EE} pin is adjusted by potentiometer to adjust the contrast of the LCD.

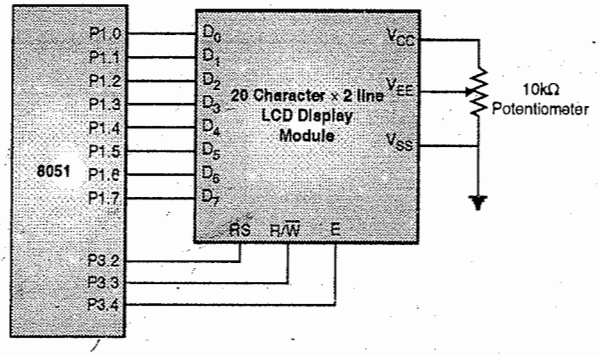


Fig. 2.7.3 : Interfacing 20 character × 2 line LCD module with 8051

Ex. 2.7.1

Interface 2 line, 16 character LCD display to 8051 / 8951 using only one port. Write assembly language program to display message 'HELLO' on line 2 of LCD.

Soln. :

Fig. P. 2.7.1 shows the interfacing of a 16 character × 2 line LCD module with the microcontroller 8051. The data lines are connected to Port 1 of 8051. The control lines RS, R/$\overline{W}$ and E are driven by P3.2, P3.3 and P3.4. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

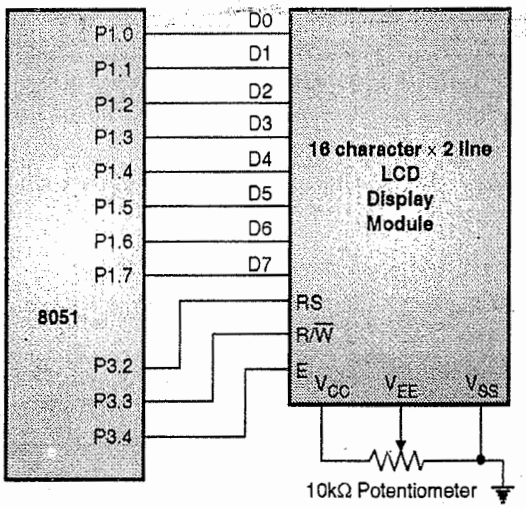


Fig. P. 2.7.1 : Interfacing 16 × 2 LCD to 8051

Let us write assembly language program to display message "HELLO"

Program

Label	Instruction	Comments
	MOV 81 H, #30H	Initialize stack pointer
	MOV A, #3CH	Command code for 5 × 10 dots, DL = 8 bits, N = 2 lines
	LCALL COMMAND	
	MOV A, #0EH	Command for setting display cursor on
	LCALL COMMAND	
	MOV A, #01H	Command for clearing display
	LCALL COMMAND	
	MOV A, #06H	Shift cursor right
	LCALL COMMAND	
	MOV A, #C0H	Cursor line 2, position 0
	LCALL COMMAND	
	MOV A, #'H'	Display Letter **H**
	LCALL DISPLAY	
	MOV A, #'E'	Display Letter **E**
	LCALL DISPLAY	
	MOV A, #'L'	Dispiay Letter **L**
	LCALL DISPLAY	
	MOV A, #'L'	Display Letter **L**
	LCALL DISPLAY	
	MOV A, #'O'	Display Letter **O**
	LCALL DISPLAY	
HERE:	SJMP HERE	Loop here after displaying message

Command Routine

Instruction	Comments
LCALL READY	Check if LCD is ready
MOV P1, A	Issue command code
CLR P3.2	Make RS = 0 to issue command
CLR P3.3	Make R/$\overline{W}$ = 0 to enable writing
SETB P3.4	Make E = 1
CLR P3.4	Make E = 0
RET	Return

Display Routine

Instruction	Comments
LCALL READY	Check if LCD is ready
MOV P1, A	Give data
SETB P3.2	RS = 1 to get data
CLR P3.3	R/$\overline{W}$ = 0 to enable writing
SETB P3.4	E = 1
CLR P3.4	E = 0
RET	Return

Ready Routine

Label	Instruction	Comments
	CLR P3.4	Disable display
	CLR P3.2	RS = 0 inorder to access command register
	MOV P1, #0FFH	Configure P1 as input port
	SETB P3.3	R/$\overline{W}$ = 1 to enable writing
L1 :	SETB P3.4	Make E = 1
	JB P1.7, L1	Check D7 bit. If 1, LCD is busy wait till it becomes 0.
	CLR P3.4	Make E = 0 to disable display

Ex. 2.7.2 Lab Assignment

Write a program to display message "MICRO" using busy flag check method on line 1.

OR

Draw and explain interfacing diagram for 20 × 2 LCD module. Write a program to display 'MICRO' message on LCD module.

Soln. :

Fig. P. 2.7.2 shows interfacing of 20 × 2 LCD module with LCD.

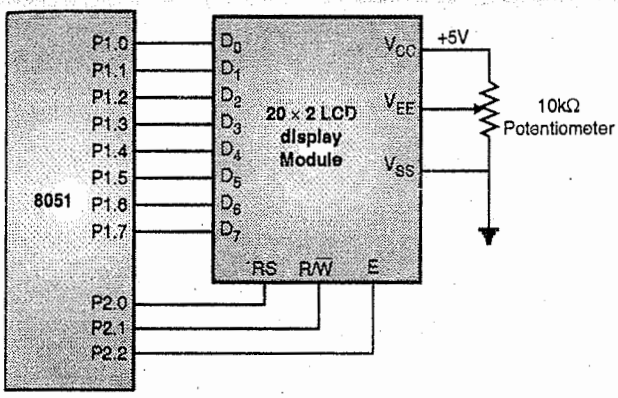


Fig. P. 2.7.2 : Interfacing 8051 to LCD

Label	Instruction	Comments
	MOV A, #38H	Initialize LCD 2 lines, 5×7 matrix
	ACALL COMMAND	Issue command
	MOV A, #0EH	LCD on, cursor on
	ACALL COMMAND	Issue command
	MOV A, #01H	Clear LCD command
	ACALL COMMAND	Issue command
	MOV, #06H	Shift cursor right
	ACALL COMMAND	Issue command
	MOV A, #80H	Cursor : Line 1 position 0
	ACALL COMMAND	Issue command
	MOV A, #'M'	Display letter " M"
	ACALL DISPLAY	
	MOV A, #'I'	Display letter "I"
	ACALL DISPLAY	
	MOV A, #'C'	Display letter "C"
	ACALL DISPLAY	
	MOV A, #'R'	Display letter "R"
	ACALL DISPLAY	
	MOV A, #'O'	Display letter "O"
	ACALL DISPLAY	
L1 :	SJMP L1	

Command Routine

Label	Instruction	Comments
COMMAND :	ACALL READY	Is LCD = ready ?
	MOV P1, A	Issue command code
	CLR P2.0	RS = 0 for command
	CLR P2.1	R / $\overline{W}$ = 0 to write to LCD
	SETB P2.2	E = 1 for high-to-low pulse
	CALL DELAY	Wait for sometime
	CLR P2.2	E = 0, latch in
	RET	

Display Routine

Label	Instruction	Comments
DISPLAY :	ACALL READY	Is LCD = ready
	MOV P1, A	Issue data
	SETB P2.0	RS = 1 for data
	CLR P2.1	R / $\overline{W}$ = 0 to write to LCD
	SETB P2.2	E = 1 for H to L pulse
	ACALL DELAY	Wait for sometime
	CLR P2.2	E = 0
	RET	

Ready Routine

Label	Instruction	Comments
READY :	SETB P1.7	Make P1.7 input port
	CLR P2.0	RS = 0 access command register
	SETB P2.1	R / $\overline{W}$ = 1 read command register
// Read command register and busy flag check		
BACK :	CLR P2.2	E = 0 for low to high pulse
	ACALL DELAY	Wait for sometime
	SETB P2.2	E = 1 for low to high pulse
	JB P1.7, BACK	Stay till busy flag = 0
	RET	

Delay Routine

Label	Instruction	Comments
	MOV R3, #10	
L1 :	MOV R4, #250	
L2 :	DJNZ R4, L2	
	DJNZ R3, L1	
	RET	
	END	

Ex. 2.7.3

Interface intelligent LCD module to 8951 / 8051 microcontroller. Explain interface signals. Write assembly language program to display "UNIVERSITY" on line 1 and "OF PUNE" on line 2 of LCD.

Soln. :

Fig. P. 2.7.3 shows the interfacing diagram.

- Fig. P. 2.7.3 shows the interfacing of a 20 character × 2 line LCD module with 8051. As shown in Fig. P. 2.7.3 the data lines are connected to Port 1 of 8051. The control lines RS, R/$\overline{W}$ are driven by Port 3 pins P3.2, P3.3 and P3.4. The voltage at V_{EE} pin is adjusted by potentiometer to adjust the contrast of the LCD.

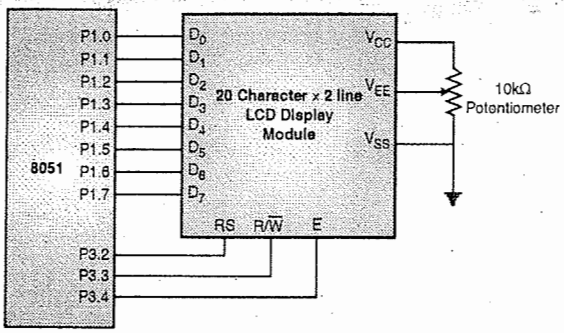


Fig. P. 2.7.3 : Interfacing 8051 to LCD module

Program

Label	Instruction	Comments
	MOV 81 H, #30H	Initialize stack pointer
	MOV A, #3CH	Command code for 5 × 10 dots, DL = 8 bits, N = 2 lines
	LCALL COMMAND	
	MOV A, #0EH	Command for setting display cursor on
	LCALL COMMAND	
	MOV A, #01H	Command for clearing display
	LCALL COMMAND	
	MOV A, #06H	Command for moving cursor to the right
	LCALL COMMAND	
	MOV A, #80H	Cursor line1,position 0
	LCALL COMMAND	
	MOV A, #'U'	Display Letter U
	LCALL DISPLAY	
	MOV A, #'N'	Display Letter N
	LCALL DISPLAY	
	MOV A, #'I'	Display Letter I
	LCALL DISPLAY	
	MOV A, #'V'	Display Letter V
	LCALL DISPLAY	
	MOV A, #'E'	Display Letter E
	LCALL DISPLAY	
	MOV A, #'R'	Display Letter R
	LCALL DISPLAY	
	MOV A, #'S'	Display Letter S
	LCALL DISPLAY	
	MOV A, #'I'	Display Letter I
	LCALL DISPLAY	
	MOV A, #'T'	Display Letter T
	LCALL DISPLAY	
	MOV A, #'Y'	Display Letter Y
	LCALL DISPLAY	
	MOV A,#C0 H	Cursor line 2, position 0
	LCALL COMMAND	
	MOV A, #'O'	Display Letter O
	LCALL DISPLAY	
	MOV A, #'F'	Display Letter F
	LCALL DISPLAY	
	MOV A, #'P'	Display Letter P
	LCALL DISPLAY	
	MOV A, #'U'	Display Letter U
	LCALL DISPLAY	
	MOV A, #'N'	Display Letter N
	LCALL DISPLAY	
	MOV A, #'E'	Display Letter E
	LCALL DISPLAY	
HERE :	SJMP HERE	Loop here after displaying message

Command Routine

Instruction	Comments
LCALL READY	Check if LCD is ready
MOV P1, A	Issue command code
CLR P3.2	Make RS = 0 to issue command
CLR P3.3	Make R/ $\overline{W}$ = 0 to enable writing
SETB P3.4	Make E = 1
CLR P3.4	Make E = 0
RET	Return

Display Routine

Instruction	Comments
LCALL READY	Check if LCD is ready
MOV P1, A	Give data
SETB P3.2	RS = 1 to get data
CLR P3.3	R/ $\overline{W}$ = 0 to enable writing
SETB P3.4	E = 1
CLR P3.4	E = 0
RET	Return

Ready Routine

Label	Instruction	Comments
	CLR P3.4	Disable display
	CLR P3.2	RS = 0 inorder to access command register
	MOV P1, #0FFH	Configure P1 as input port
	SETB P3.3	R/ $\overline{W}$ = 1 to enable writing
L1 :	SETB P3.4	Make E = 1
	JB P1.7, L1	Check D7 bit. If 1, LCD is busy wait till it becomes 0.
	CLR P3.4	Make E = 0 to disable display
	RET	

Ex. 2.7.4 Lab Assignment

Interface a 2 line , 16 character LCD display to 89C51 using four data pins only. Write a program to display " LCD Interfacing " on line 2 of LCD using busy check flag.

Soln. : When $R/\overline{W} = 0$, $\overline{RS} = 0$ and interface length sets DL bit ($B_4 = 0$) when $B_7\ B_6\ B_5 = 001$, N = 1 (two display uses), f = 0 (5 × 7) font). Then, the command instruction written to controller is 00101000 (28 H). Let controller pins RS connect to P3.7, $R/\overline{W}$ pin to P3.5 and E pin to P1.6. Let P1.7 – P1.4 connect to pins $D_7 - D_4$.

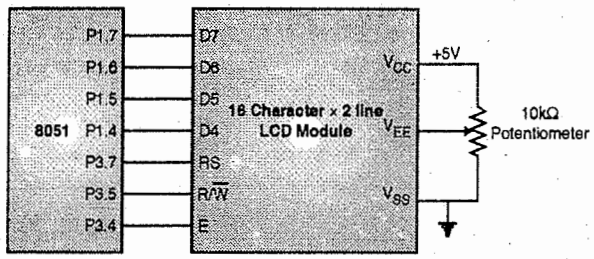


Fig. P. 2.7.4 : Interfacing 8051 to 16 character × 2 line LCD module

Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP MAIN	
	ORG 0030H	
MAIN :	NOP	
	E EQU P3.7	
	RS EQU P3.5	
	RW EQU P3.4	
	DAT EQU P1	
	LCALL LCD_INT	
AGAIN :	LCALL CLEAR	
	LCALL LINE2	
	MOV DPTR,#MYDAT	
	LCALL LOOP	
	SJMP AGAIN	
W_NIB :	PUSH A	Save A for low nibble
	ORL DAT,#0F0h	Bits 4..7 <– 1
	ORL A,#0Fh	Don't affect bits 0-3
	ANL DAT,A	High nibble to display
	SETB E	
	CLR E	
	POP A	Prepare to send
	SWAP A	...second nibble
	ORL DAT,#0F0h	Bits 4...7 <– 1
	ORL A,#0Fh	Don't affect bits 0...3
	ANL DAT,A	Low nibble to display
	SETB E	
	CLR E	
	RET	
LCD_INT :	CLR RS	
	CLR RW	
	CLR E	
	SETB E	
	MOV DAT,#028h	
	CLR E	
	LCALL SDELAY	
	MOV A,#28h	
	LCALL COM	
	MOV A,#0Ch	
	LCALL COM	
	MOV A,#06h	
	LCALL COM	
	LCALL CLEAR	
	MOV A,#080H	
	LCALL COM	
	RET	
CLEAR :	CLR RS	
	MOV A,#01H	
	LCALL COM	
	RET	
DATAW :	SETB RS	
	CLR RW	
	LCALL W_NIB	
	LCALL LDELAY	
	RET	
SDELAY :	MOV R6,#1	
HERE2 :	MOV R7,#255	
HERE :	DJNZ R7,HERE	
	DJNZ R6,HERE2	
	RET	
LDELAY :	MOV R6,#100	
HER2 :	MOV R7,#255	
HER :	DJNZ R7,HER	
	DJNZ R6,HER2	
	RET	
COM :	CLR RS	
	CLR RW	
	LCALL W_NIB	
	LCALL SDELAY	
	RET	
LINE2 :	MOV A,#0C0H	
	LCALL COM	
	RET	
LOOP :	CLR A	
	MOVC A,@A+DPTR	
	JZ GO_B2	
	LCALL DATAW	
	LCALL SDELAY	
	INC DPTR	
	SJMP LOOP	
GO_B2 :	RET	
MYDAT :	DB " LCD INTERFACING ",0	
	END	

Ex. 2.7.5

Interface a 2 line ,20 character LCD display to 89C51 using four data pins only. Write a program to display "Ohmic Memory Avail" on line 2 of LCD using busy check flag.

Soln. :

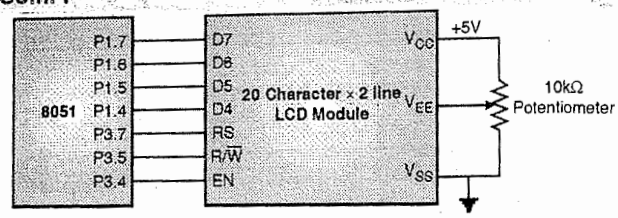


Fig. P. 2.7.5 : Interfacing 8051 to 20 character × 2 line LCD module

Program

Label	Instruction	Comment
	ORG 0000H	
	LJMP MAIN	
	ORG 0030H	
MAIN :	NOP	
	E EQU P3.7	
	RS EQU P3.5	
	RW EQU P3.4	
	DAT EQU P1	
	LCALL LCD_INT	
AGAIN :	LCALL CLEAR	
	LCALL LINE2	
	MOV DPTR,#MYDAT	
	LCALL LOOP	
	SJMP AGAIN	
W_NIB :	PUSH A	Save A for low nibble
	ORL DAT,#0F0h	Bits 4..7 <- 1
	ORL A,#0Fh	Don't affect bits 0-3
	ANL DAT,A	High nibble to display
	SETB E	
	CLR E	
	POP A	Prepare to send
	SWAP A	...second nibble
	ORL DAT,#0F0h	Bits 4...7 <- 1
	ORL A,#0FH	Don't affect bits 0...3
	ANL DAT,A	Low nibble to display
	SETB E	
	CLR E	
	RET	
LCD_INT :	CLR RS	
	CLR RW	
	CLR E	
	SETB E	
	MOV DAT,#028h	
	CLR E	
	LCALL SDELAY	
	MOV A,#28h	
	LCALL COM	
	MOV A,#0Ch	
	LCALL COM	
	MOV A,#06h	
	LCALL COM	
	LCALL CLEAR	
	MOV A,#080H	
	LCALL COM	
	RET	
CLEAR :	CLR RS	
	MOV A,#01h	
	LCALL COM	
	RET	
DATAW :	SETB RS	
	CLR RW	
	LCALL W_NIB	
	LCALL LDELAY	
	RET	
SDELAY :	MOV R6,#1	
HERE2 :	MOV R7,#255	
HERE :	DJNZ R7,HERE	
	DJNZ R6,HERE2	
	RET	
LDELAY :	MOV R6,#100	
HER2 :	MOV R7,#255	
HER :	DJNZ R7,HER	
	DJNZ R6,HER2	
	RET	
COM :	CLR RS	
	CLR RW	
	LCALL W_NIB	
	LCALL SDELAY	
	RET	
LINE2 :	MOV A,#0C0H	
	LCALL COM	
	RET	
LOOP :	CLR A	
	MOVC A,@A+DPTR	
	JZ GO_B2	
	LCALL DATAW	
	LCALL SDELAY	
	INC DPTR	
	SJMP LOOP	
GO_B2 :	RET	
MYDAT :	DB " Ohmic Memory Avail ",0	
	END	

Syllabus Topic : Interfacing of ADC 0809

2.8 Interfacing of ADC 0809

2.8.1 ADC 0808/0809

- A large number of ADC ICs have been produced by the manufacturing companies like National semiconductors, Motorola, Intersil etc. to meet various demands such as speed of response resolution, compatibility and ease of interfacing with microprocessors etc.
- The National semiconductor produces ADC 0809 which is an 8-bit ADC whereas Intersil produces IC7109. ICL 7109 which is a 12-bit ADC. A serial 8-bit ADC is available

i.e. MAX1112. This ADC gives the digital data out in serial form i.e. 1 bit at a time. Let us discuss the ADC 0808/0809 in detail.

2.8.1(A) Principle of A to D Conversion in ADC 0808/0809

- The ADC 0809 operates on the successive approximation technique of A to D conversion.
- It is a CMOS device with 8-analog inputs, an 8 channel multiplexer and microprocessor compatible control logic.
- As the number of bits n = 8, it includes a 256 resistor voltage divider, a group of analog switches and a successive approximation register (SAR).
- As there are 8-analog channels, we can connect upto 8 analog inputs to this IC.
- However due to the use of a multiplexer, at a time only one analog input will be converted into an equivalent 8-bit digital output. The analog input channels can be selected using the three address lines A, B and C.

2.8.1(B) Features of ADC 0808/0809

- Inbuilt 8 analog channels with multiplexer.
- Zero or Full scale adjustment is not required.
- 0 to 5 V input voltage range with a single polarity 5 V supply.
- Output is TTL compatible.
- High speed.
- Low conversion time (100 μs).
- High accuracy.
- 8-bit resolution.
- Low power consumption (less than 15 mW).
- Easy to interface with all microprocessors.
- Minimum temperature dependence.

2.8.1(C) Pin Configuration of ADC 0808/0809

Table 2.8.1 : Selection of one of the analog inputs using the address lines A, B and C

Selected analog channel	Address		
	C	B	A
IN 0	0	0	0
IN 1	0	0	1
IN 2	0	1	0
IN 3	0	1	1
IN 4	1	0	0
IN 5	1	0	1
IN 6	1	1	0
IN 7	1	1	1

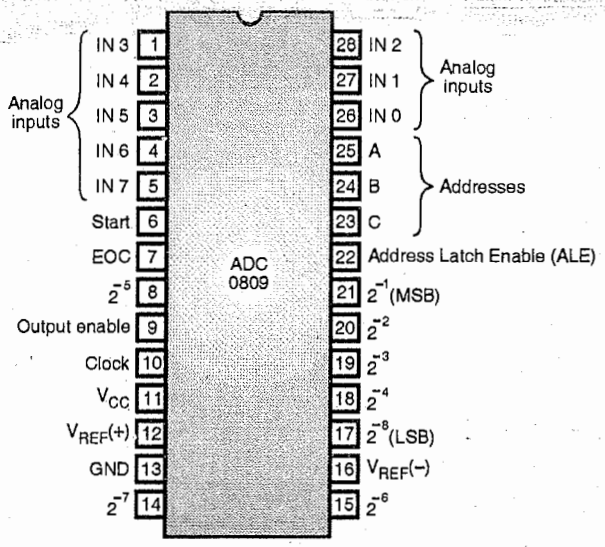


Fig. 2.8.1 : Pin configuration of IC ADC 0808/0809

Description

(i) Analog Inputs (IN 0 to IN 7)

Pin numbers 1 to 5 and 26 to 28, designated as IN 0 to IN 7 are the eight analog inputs of this IC. We can connect signals coming from eight different transducers to these inputs. Each one of these inputs will be converted to an 8-bit equivalent digital (binary word). However these inputs are converted into digital form one by one and not all at a time. Hence, one of these eight inputs should be selected for conversion. This selection is done by means of the address pins A, B and C.

(ii) Address Pins A, B, C (Pin 23, 24, 25)

These pins will decide or select one out of the eight analog inputs, for conversion into digital form. For example if CBA = 010 then the "IN 2" is selected and the analog signal at this input is converted to equivalent digital form.

(iii) Reference Voltage [V_{REF} (+) and V_{REF} (–)]

Depending on the desired polarity of the reference voltage, we can connect a positive or negative reference voltage externally to these pins. (2^{-1} to 2^{-8}).

(iv) ALE and Output Enable

As shown in the functional block diagram of ADC 0808/0809 (Fig. 2.8.2), the address latch enable (ALE) input is useful in enabling the address latch which stores the address on lines A, B and C. The output enable pin, when activated will make the digital output available on the output pins.

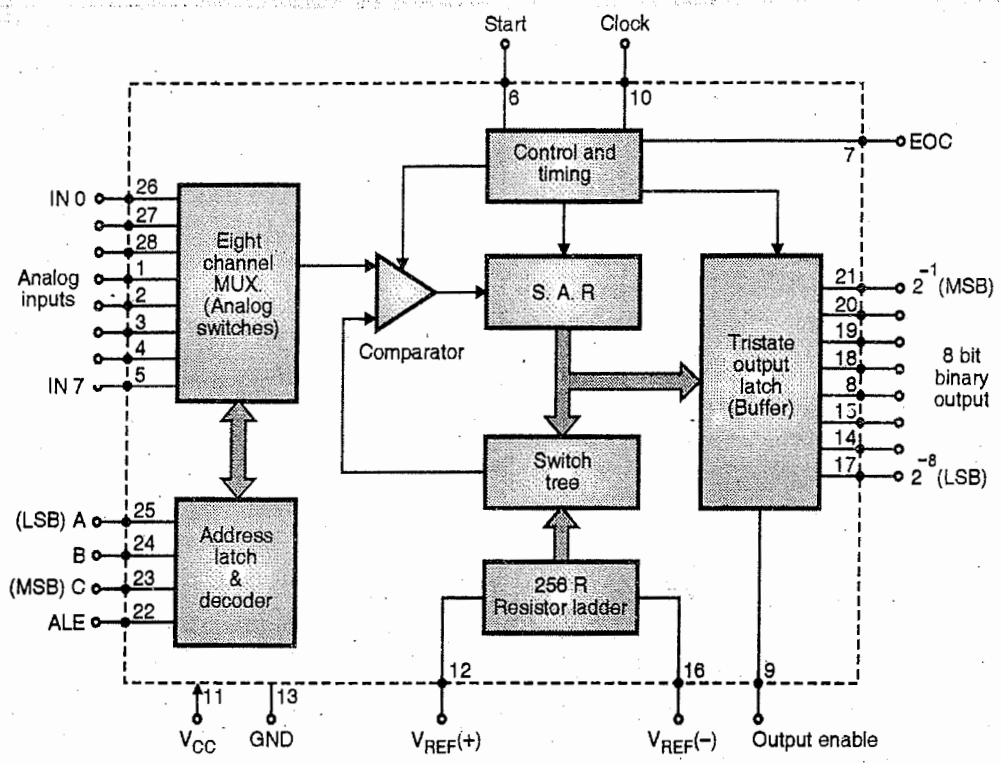


Fig. 2.8.2 : Functional block diagram of IC ADC 0808/0809

(v) Start and EOC

We have to enable the start input to begin the A to D conversion. A pulse is to be given on start pin to start conversion. The end of conversion is indicated by EOC (End of Conversion) output.

(vi) Digital Outputs [2^{-1} to 2^{-8}]

The digital output is available to these pins. 2^{-1} represents the MSB and 2^{-8} represents the LSB of digital output. Fig. 2.8.3 shows typical interfacing of 8 channel, 8 bit ADC to 8051.

2.8.2 Interfacing ADC 0808 to 8051

Fig. 2.8.3 shows interfacing of 8 channels, 8 bit ADC to 8051.

- ADC 0808 has eight input channels. Hence in order to select an input channel, it is essential to send 3 bit address on C, B and A inputs.
- The address of the desired channel is sent to the address inputs through port pins P2.0, P2.1 and P2.2.
- After 50 ns the address must be latched. It can obtained by sending ALE signal. After 2.5μs, SOC must be made high and then low to start the conversion.
- To indicate end of conversion (EOC) signal must be activated.
- The 8051 pins P2.6 and P2.7 are connected to SOC and EOC pins of ADC 0808/0809.
- After the conversion is over, 8 bit digital data is present on D_0 – D_7 lines. 8051 accepts data through port 1.
- Fig. 2.8.3 the clock source to ADC0808/0809 is from the crystal oscillator. But this frequency is very high for the A/D converter. Hence, four D flip-flops are used for dividing the frequency.

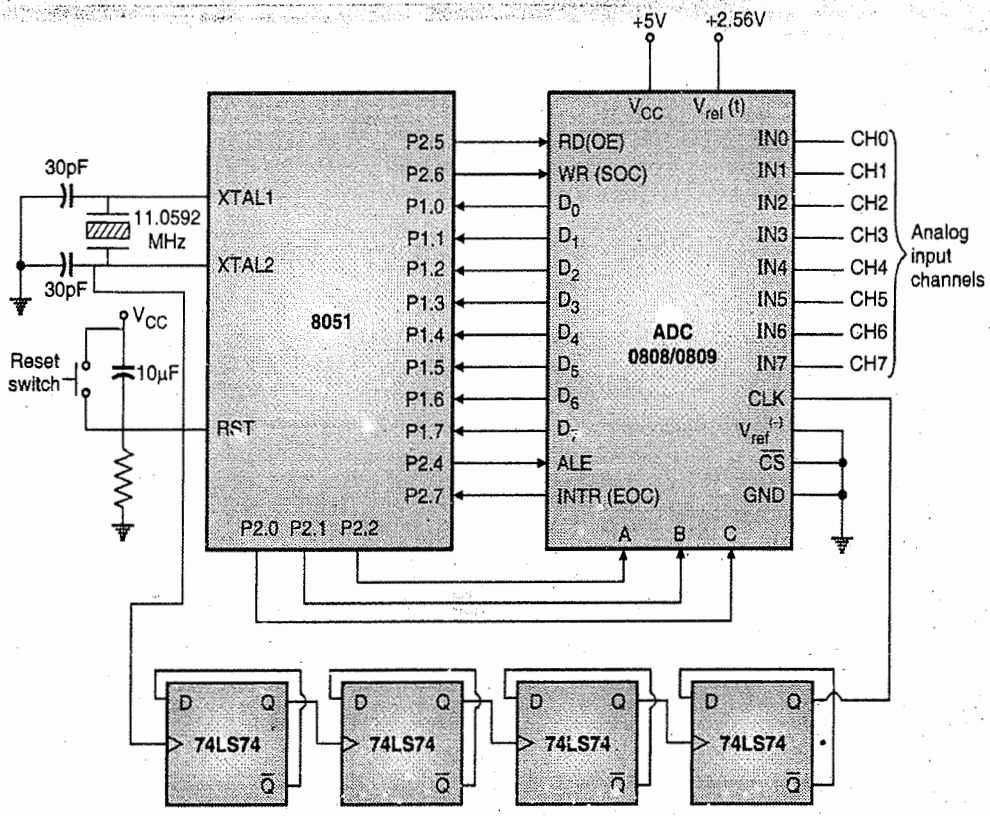


Fig. 2.8.3 : Interfacing 8 channel 8 bit ADC to 8051

Fig. 2.8.4 shows timing diagram for selecting a channel and read timing for ADC 0809.

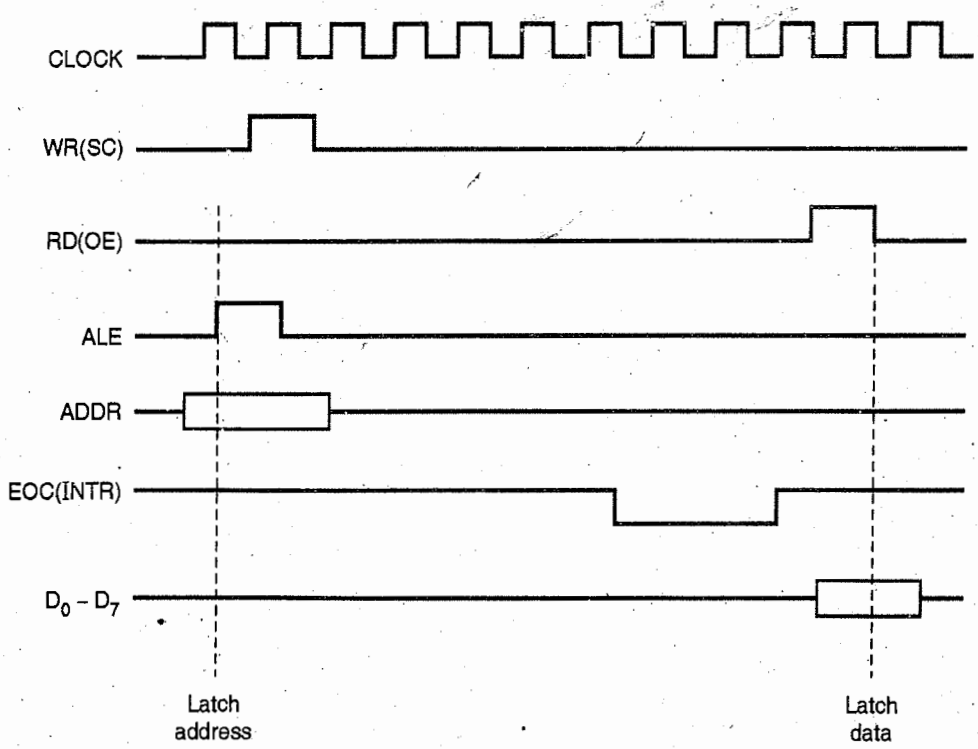


Fig. 2.8.4 : Timing diagram

2.8.3 Steps to Program ADC 0808/0809

Step I : By properly selecting ABC inputs, select the required analog channel.

Step II : Enable the ALE pin. For latching the address a low to high pulse is given to the ALE pin.

Step III : Enable SOC pin by L-to-H (low to high pulse) to begin conversion.

Step IV : Observe for the EOC (end of conversion) to check whether conversion is complete.

Step V : For reading the data from ADC chip active the OE signal. A low-to high pulse on OE pin will fetch data from the chip.

Ex. 2.8.1

Write an assembly language to perform A/D conversion ADC 0808.

Soln. :

Label	Instruction	Comments
	A BIT P3.0	
	B BIT P3.1	
	C BIT P3 2	
	ALE BIT P1.4	
	OE BIT P1.5	
	SOC BIT P1.6	
	EOC BIT P1.7	
	DAT EQU P2	
	ORG 00H	
	MOV DAT, #0FFH	P2 = input port
	MOV R2, #00H	
	SETB EOC	EOC = 1
	CLR ALE	ALE = 0
	CLR SOC	SOC = 0
	CLR OE	OE = 0
BACK :	CLR A	A = 0 } select channel 3
	SETB B	B = 1 } select channel 3
	SETB C	C = 1 } select channel 3
	ACALL DELAY	Delay
	SETB ALE	ALE = 1(Latch Address)
	ACALL Delay	Delay
	SETB SOC	Start conversion
	ACALL DELAY	
	CLR ALE	
L1 :	JB EOC, L1	
L2 :	JNB EOC, L2	Wait till conversion is done
	SETB OE	
	ACALL Delay	
	MOV A, DAT	Read Data
	MOV 50H, A	Store result at address 50H
	CLR OE	$\overline{RD} = 0$
	SJMP BACK	

Ex. 2.8.2

Interface 8 bit, 8 channel ADC to 8051. Write assembly language program to convert CH0, CH3 and CH7 and store the result in external memory location starting from C000 H. Repeat procedure for every 1 sec.

Soln. :

Fig. P. 2.8.2 shows interfacing of 8 channels, 8 bit ADC to 8051.

- ADC 0808 has eight input channels. Hence inorder to select an input channel, it is essential to send 3 bit address on C, B and A inputs.
- The address of the desired channel is sent to the address inputs through port pins P2.0, P2.1 and P2.2.
- After 50 ns the address must be latched. It can obtained by sending ALE signal. After 2.5μs, SOC must be made high and then low to start the conversion.
- To indicate end of conversion (EOC) signal must be activated.
- The 8051 pins P2.6 and P2.7 are connected to SOC and EOC pins of ADC 0808/0809.
- After the conversion is over, 8 bit digital data is present on $D_0 - D_7$ lines. 8051 accepts data through port 0.

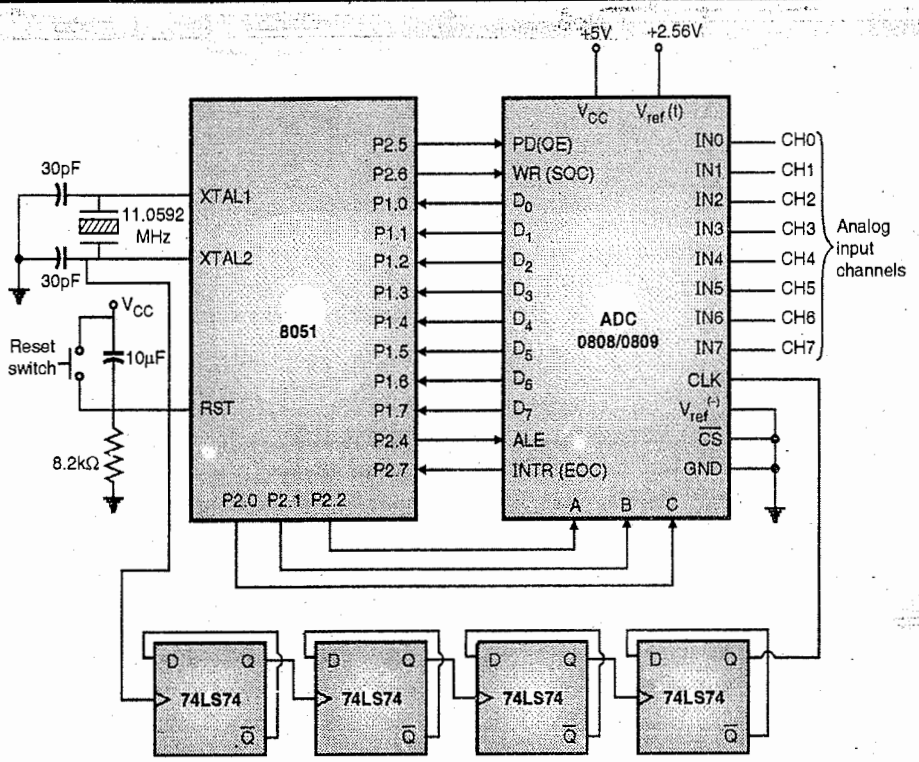


Fig. P. 2.8.2 : Interfacing 8 channel 8 bit ADC to 8051

Program

Label	Instruction	Comments
	CLR P2.6	Make SOC Low
	CLR P2.4	Make ALE Low
	MOV P0, #FFH	Configure port 0 as input
	MOV P1, #FFH	Configure port 1 as input
L1 :	MOV DPTR, #C000H	Initialize memory pointer
	MOV A, #00H	Set address for channel 0
	ACALL A_D	Call ADC routine
	MOVX @DPTR, A	Increment memory pointer
	MOV A, #03H	Set address for channel 3
	ACALL A_D	Call ADC routine
	MOVX @DPTR, A	Save digital value
	INC DPTR	Increment memory pointer
	MOV A, #07H	Set address for channel 7
	ACALL A_D	Call ADC routine
	MOVX @DPTR, A	Save digital value
	ACALL DELAY	Wait for 1sec
	SJMP L1	Repeat

Analog to Digital Conversion Routine

Label	Instruction	Comments
A_D :	MOV P2, A	Get the channel number and set its address
	SET P2.4	Send ALE
	NOP	
	CLR P2.4	
	SET P2.6	Send SOC
	NOP	
	CLR P2.6	
WAIT :	JB P2.7, WAIT	Wait for EOC signal
WAIT 1 :	JNB P2.7, WAIT1	
	MOV A, P1	Get digital data
	RET	Return

Delay Routine

Label	Instruction	Comments
DELAY :	MOV TMOD, #01	Timer 0, mode 1 (16 bit mode)
	MOV R0, #1H	Initialize counter to 20
L2 :	MOV TL0, #B0H	
	MOV TH0, #3CH	
	SETB TR0	
L3 :	JNB TF0, L3	Check timer 0 flag till it rolls over
	CLR TR0	Clear timer 0
	CLR TF0	Clear timer 0 flag
	DJNZ R0, L2	Decrement counter and if not zero repeat
	RET	

Ex. 2.8.3

Write assembly language program to take 20 samples from ADC and store it in RAM location starting at 50H address.

Soln. :

Label	Instruction	Comments
	ALE BIT P2.4	
	OE BIT P2.5	
	SOC BIT P2.6	
	EOC BIT P2.7	
	ADD_A BIT P2.0	
	ADD_B BIT P2.1	
	ADD_C BIT P2.2	
	ORG 00H	
	MOV P1, #0FFH	Make P1 an input port
	MOV R0, #50H	Store address 50H
	MOV R1, #20	Count for 20 samples
	SETB EOC	Make EOC an input
	CLR ALE	Clear ALE
	CLR SOC	Clear $\overline{WR}$
	CLR OE	Clear $\overline{RD}$
BACK :	CLR ADD_C	C = 0 } (Select channel 1)
	CLR ADD_B	B = 0 } (Select channel 1)
	SETB ADD_A	A = 1 } (Select channel 1)
	ACALL DELAY	Ensure that address is stable
	SETB ALE	Latch address
	ACALL DELAY	
	SETB SOC	Start conversion
	ACALL DELAY	
	CLR ALE	
	CLR SOC	
HERE :	JB EOC, HERE	Wait until done
HERE 1:	JNB EOC, HERE 1	Wait until done
	SETB OE	Enable $\overline{RD}$
	ACALL DELAY	Wait
	MOV A, P1	Read data
	MOV @R0, A	Store the samples from 50H
	INC R0	Increment pointer to next memory location
	DEC R1	Decrement count
	CLR OE	Clear $\overline{RD}$ for next sample
	SJMP BACK	
DELAY :	MOV R3, #250	
HERE 2 :	NOP	
	NOP	
	NOP	
	NOP	
	DJNZ R3, HERE 2	
	RET	

Syllabus Topic : Programming Environment

2.9 Programming Environment

The basic development tools used for programming the microcontroller based systems are :

(i) Software development tools
(ii) Hardware development tools

The software development tools comprise of

(i) assemblers (ii) editors
(iii) compilers (iv) simulators
(v) debuggers
(vi) IDE (Integrated development environment)

- The hardware development tools comprise of
  (i) Emulators (ii) Demo boards
  (iii) Logic analyzers
  (iv) Digital storage oscilloscopes (DSO)
  (v) Logic probes

In this chapter we will study these hardware and software development tools.

Syllabus Topic : Study of Software Development Tool Chain (IDE)

2.10 Study of Software Development Tool Chain (IDE)

- An integrated development environment (IDE) is a software that allows the user to develop an application. It provides :

(i) Source code editor / text editor
(ii) Compiler and interpreter
(iii) Debugger
(iv) Emulator
(v) Programmer
(vi) Simulator

that is needed to develop an application program.

- An IDE comprises every software tool that is needed to develop an application program. 8051 / IDE is IDE used for 8051 microcontroller.
- For PIC18xxx family of microcontrollers MPLAB IDE is the IDE used.

2.10.1 Editor

- It is a program that allows the programmer to enter and edit our programs.
- An editor is basically a software (i.e. a program).
- It helps the user to create a file that contains the assembly language statements.
- The examples of editors used for the assembly language programs are Wordstar, Edit, WordPad, Notepad etc.
- The job of the editor is to store the ASCII codes for the letters and numbers in the successive RAM locations.
- As the typing of program is over, this file is stored on a floppy or hard disk.
- This file is called as the "source file" and an ASM extension is given to it.
- The source file is then processed using an assembler.
- The microchip MPLAB IDE has a text editor that allows the user to enter programs and also edit them.

2.10.2 Assembler

SPPU - May 12, Dec. 12; Dec. 14; Aug. 15

University Question

Q. Explain assembler.

(May 2012, Dec. 2012, Dec. 2014, 2 Marks, Aug. 2015(In sem.), 3 Marks)

- Each assembly level instruction has a mnemonic. For example in the instruction MOVLW 0x50, MOVLW represents the mnemonic.
- An assembler is a program which translates the assembly language mnemonics into corresponding binary codes.
- The assembler reads the source file more than once.

Assembler operation

- The assembler first reads the source file of program.
- Then it determines the displacement of data items, offsets of labels etc. and puts this information into a symbol table.
- Then it produces the binary codes for each assembly language instruction and detects syntax errors if any. Then it inserts the offsets etc. calculated earlier.

File Generation in assembler

- An assembler generates two files namely the **object file** and the **assembler list file.**
- The object file is given extension **.OBJ** whereas the assembler list file is given extension **.LST.**
- **Object file :** It contains the binary codes of the program instructions and the information about the addresses of instructions.
- **List file :** It contains the assembly language statements, the binary codes for each instruction and the offset of each instruction.
- Any typing or syntax errors are indicated in the assembly listing if we take a print out of **.LST** file.

Error detection and correction

- The assembler is capable of only finding the syntax errors.
- To check if our program is working, we have to test and run the program.
- The errors indicated by the assembler should be edited using the editor.
- This edit-assemble loop should be executed till all the errors are corrected.
- MPLAB supports **MPASM** assembler for PIC18xxx microcontroller families.
- It has following features :
  - i) All the source codes can be translated into object codes.
  - ii) It produces the .obj, .lst files needed for debugging with the emulator systems.
  - iii) It has macro assembly capability.
  - iv) It supports decimal, Hex and octal source as well as listing formats.

2.10.3 Linker

- Linker is a program which is used for joining many object files into one large object file.
- When a large program is being written, it is always advisable to break it into small modules, so that each module can be separately tested and debugged. Then finally link their object modules together to form a large working program. e.g. the display routine can be kept in the library file and linked into the other programs when required.
- The linker produces a link file which contains the binary codes for all the combined modules.
- The linker produces a link map file. It contains the address of all the linked files.

- It is important to note that the linker does not assign absolute addresses. It only assigns relative addresses to the program starting from zero.
- It is relocatable. The linkers with MASM or TASM produce link files with . EXE extension.
- The microchips **MPLINK** is a relocatable linker for MPASM and C-18 compiler. It can link relocatable objects.
- MPLINK allows the user to generate a reusable code with MPASM and C18.
- It combines several object modules generated by MPASM or C18 compiler to single executable file.

2.10.4 Compiler

SPPU - Dec. 14, Aug. 15

University Questions

Q. Write short note on : Compiler. **(Dec. 2014, 3 Marks)**

Q. Explain software development tool : Compiler. **(Aug. 2015(In sem.), 3 Marks)**

- **Compiler** is a program that translates the high level language source program to a machine language program. The program written in high level language is called as the source program and the program that is compiled on the machine is called as object code.
- A compiler translates the source program into relocatable object modules. The object modules are linked by the linker. Locator loads the complete program in memory where it can be executed.
- However, the drawback of using the compiler is that if any error is detected we need to correct the source program and repeat the compilation process.

2.10.5 Cross Assembler and Cross Compiler

SPPU - May 13

University Question

Q. Explain cross assembler. **(May 2013, 2 Marks)**

- The special feature of the cross assembler is that it is not written in the same language that is used by the microcontroller that executes the machine code that is generated by the assembler.
- The cross assembler is usually written in a high level language like FORTRAN, C, PASCAL etc that makes it machine independent.
- An assembler that runs on one type of computer and assembles the source code for a different target computer is called as a **cross assembler.**
- For example,
  (i) An assembler that runs on an Intel x86 machine and generates object code for Motorola's 68HC05.
  (ii) 8086 assembler may be written in C and then the assembler may be executed on some other machine like Motorola 6800.
- For a PIC18XXX microcontroller the cross assemblers and cross compilers generate an executable code that can be placed in the ROM, EPROM, flash memory or EEPROM.

2.10.6 Debugger

- Debugger is a software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables.
- If a program is directly accessible from the microcomputer and does not need any external hardware, then we can use a debugger to run and test the program.
- Debugger is basically a program which permits the user to load object code program into the system memory, execute the program and debug it.
- The debugger also permits the change in register contents, memory locations and rerun the program.
- With the help of the debugger, we can stop the program execution after each instruction so that we can check or alter the memory and register contents.
- In other words we can put breakpoints in the program and execute the program from one breakpoint to the other.
- It is possible to examine the register and memory contents after partial execution of program between the breakpoints.
- We can use the debugger to check and correct the program till all the errors are corrected.
- For most IBM PC type computers the basic debugger comes by default.
- The MPLAB IDE allows the user to debug programs using source files for PIC18xxx. family of microcontrollers.
- They make the debugging easier and allow the user to see the contents of registers and memory locations as the program is executed.

2.10.7 Emulator SPPU - May 13, Dec. 13, Oct. 16

University Questions

Q. Explain emulator. **(May 2013, Dec. 2013, 2 Marks)**

Q. Explain the role of Emulator as development tool. **(Oct. 2016(In sem.), 2 Marks)**

- The **emulator** is used to test and debug the hardware and software of an external system such as the Microcontroller based system.
- Emulator is a combination of hardware and software.
- An emulator consists of a multi-wire cable that connects the host system to the external system.
- Through this cable the software of the emulator allows the user to download the object code program into RAM in the external system being developed.
- Like the debugger the emulator also allows the user to load the programs to be tested, run the, programs check and modify the contents of various registers and memory locations and also insert the breakpoints.
- As each instruction in the assembly language program is executed, the emulator as if takes "snapshot" of the register contents, activities on the address and data buses and the state of flag register.
- The emulator stores this data as "trace data".
- It is possible to take out the print cut of the trace data so as to analyse the results produced in the program on the step by step basis.

2.10.8 Simulator SPPU - Oct. 16

University Question

Q. Explain the role of simulator as development tool. **(Oct. 2016(In Sem.), 2 Marks)**

- A **Simulator** is often used to execute a program that has to run on some inconvenient type of computer. For example, simulators are usually used to debug a microprogram.
- Since the operation of the computer is simulated, all of the information about the computer's operation is directly available to the programmer, and the speed and execution of the simulation can be varied as per the user's wish.
- Simulators may also be used to interpret **fault trees**, or test logic designs before they are constructed.
- Many video games are also simulators, that are implemented inexpensively.
- Simulator is a software package that functions like a hardware without acquiring hardware. The 8051 microcontroller simulator gives the user an 8051 environment on the PC.
- It performs the functions of the 8051 microcontroller / 8085 microprocessor without using it.
- It also performs a simulation of different peripherals that are used with the microcontroller / microprocessor in any application.
- It provides facilities that help the user to find logical errors, facilities to help the user learn about the initialization of different peripherals and get an insight of the microcontroller functioning.
- It does acts as a direct replacement of the costly 8051 microcontroller kit.
- It shows all internal registers, entire memory and peripherals on the monitor.
- It supports breakpoint and single stepping facilities that help the user to debug their programs.

2.10.8.1 Computer Configuration Required to Run the Simulator

The 8051 simulator runs on the PC. The requirement for PC is

(a) RAM of 512 KB. (b) DOS Compatibility

(c) IBM Mono, CGA, EGA, VGA or Compatible Monitor.

(d) Two Disk Drives.

2.10.8.2 Features

- Easy to operate.
- Allows Simulation of peripherals.
- Allows Simulation of Interrupts.
- Provides continuous display of the system register values.
- The Program can be easily modified.
- The core can be viewed and the corresponding instruction makes it easy to understand the logic of the program while the program executes.
- It is a powerful debugging tool with different high level debugging facilities.
- It saves the development time.
- It allows checking of Software before the hardware is available to the user.
- It has the ability for the user to construct screens that show various parts of the 8051

system. Each screen is made up of separate windows that display internal CPU registers, code and Data Memory areas.

- The contents of any location in the code ROM and Internal RAM (including SFR's) and External Memory can be changed as the program executes.
- The I/O ports may be simulated by changing the value of the port SFR's.
- Interrupts are simulated by striking the function on the keyboard.

For executing a simulation, the screen set file is loaded into the simulator first. This file is followed by a program that is written in the object code format. The program is then executed using these simulator commands.

1. Reset the Program Counter to 0000H.
2. Single Step the Program
3. Free run the program.
4. Free run until breakpoint is reached
5. Stop Free run

2.10.8.3 Modes of Simulator

The simulator has three modes of operation. They are :

1. **Idle mode :** This is a non executing mode. This mode allows the user to set the configuration i.e. loading the program file, saving required memory contents on the disk, change default directory or drive etc.
2. **Execute Mode :** This mode is a continuous execution mode. In this mode the program that the user wishes is continuously executed.
3. **Single Step Mode :** In this Mode the user program is executed step by step When the user presses the key F2 an instruction is executed and the PC points to the next instruction.

2.10.9 Comparison of Assembler and Compiler

SPPU - Dec. 16

University Question

Q. Compare Assembler and compiler. **(Dec. 2016, 6 Marks)**

Assembler	Compiler
An assembler is a program which translates the assembly language mnemonics into corresponding binary codes. The assembler reads the source file more than once.	**Compiler** is a program that translates the high level language source program to a machine language program. The program written in high level language is called as the source program and the program that is compiled on the machine is called as object code.
The assembler first reads the source file of program. Then it determines the displacement of data items, offsets of labels etc. and puts this information into a symbol table. Then it produces the binary codes for each assembly language instruction and detects syntax errors if any. Then it inserts the offsets etc. calculated earlier.	A compiler translates the source program into relocatable object modules. The object modules are linked by the linker. Locator loads the complete program in memory where it can be executed.
The assembler is capable of only finding the syntax errors. To check if our program is working, we have to test and run the program. The errors indicated by the assembler should be edited using the editor.	The drawback of using the compiler is that if any error is detected we need to correct the source program and repeat the compilation process.

Syllabus Topic : Hardware Debugging Tools

2.11 Hardware Debugging Tools

It is a difficult task to develop a microprocessor based system. In absence of developing tool, the task is time consuming and hectic. The tools are used for the development of a microprocessor / microcontroller based system are :

(i) In-circuit emulator.
(ii) Programmers
(iii) Programmer development board
(iv) Digital storage oscilloscope
(v) Logic analyzer.

Let us study them one by one

2.11.1 An In Circuit Emulator

SPPU - Oct. 16

University Question

Q. Explain role of emulator as a development tool. **(Oct. 2016(In Sem.), 2 Marks)**

- An **in-circuit emulator** (ICE) is a hardware device used during the development of **microcontroller based systems.** Virtually all such systems have a hardware element and a software element, which are separate but highly interdependent.
- The ICE allows the software element to be run and tested on the actual hardware on which it is to run, but still allows programmer conveniences such as source-level **debugging** and single-stepping, etc.

- Without an ICE, the development of microcontroller based systems can be extremely difficult, since if something does not function correctly, it is often very hard to tell what went wrong without some sort of monitoring system to oversee it.
- Most ICEs consist of an adaptor unit that sits between the host computer and the system to be tested. A large header and cable assembly connects this unit to where the actual **CPU or microcontroller** mounts within the system to be tested.
- The unit emulates the CPU, such that from the system's point of view, it has a real processor fitted. From the host computer's point of view, the system under test is under full control, allowing the developer to debug and test code directly.
- The emulator is used to test RAM, I/O ports and control functions of the development system by replacing the microprocessor IC on the board by the in-circuit emulator. E.g. with the help of the in-circuit emulator a user could stop in between during the execution of a program for examining the contents of memory locations and registers. This helps the user to see the intermediate results obtained at a particular condition, by which the user can come to know why the hardware is not performing upto the expectations.
- The MPLAB IDE can run upto four in-circuit emulators on the same PC.

2.11.2 Programmer

- The software program should be loaded on the microcontroller ROM before it is tested.
- For loading the program the microchip uses two programmers.

  They are :

i) **PICSTART® PLUS :** It is a low cost programmer that is used for PIC18 microcontrollers and has a 40 pin socket.
   - By adding different adapters we can program PIC18xxx microcontrollers that have more pins.
   - It is driven by the MPLAB IDE software. We can connect the PICSTART® PLUS programmer to the computer with the serial port.

ii) **PROMATE® II :** This programmer can be used for programming all the PIC microcontrollers manufactured by MICROCHIP.
   - It also uses adapters to program microcontrollers with more pins.
   - The MPLAB IDE drives the PROMATE® II programmer. With the help of serial port we can connect PROMATE® II programmer to the PC (computer).

2.11.3 Development Board

- For learning the microcontroller operation and testing the software before finalizing the hardware the development boards are very useful.
- The development board is a printed circuit board that comprises the microcontroller and supporting logic like I/O circuits, clock generator, RAM, ROM, UART, Timers etc. So that a programmer can learn the circuits, be acquainted with the board and also learn to program it.
- A well designed development board should allow the programmer to test each and every peripheral function.

Syllabus Topic : Logic Analyzer (Timing Analysis using Logic Analyzer)

2.11.4 Logic Analyzer (Timing Analysis using Logic Analyzer)

SPPU - May 12, Dec. 12, May 13, Dec. 13

University Question

Q. With the help of neat block diagram explain the operation of logic analyzer.
(May 2012, Dec. 2012, May 2013, Dec. 2013, 8 Marks)

Suppose we have to analyze a memory chip which is soldered on a PC board. In order to analyze the memory chip the analog analyzers are incapable because the number of signals to be observed are very larger in number and they are digital in nature i.e. we should have a device which is capable of displaying a number of signals, which is compatible with the TTL and CMOS logic levels. It should be able to sense and display the address bus, data bus, memory read, memory write, I/O read, I/O write etc. signals. All these tasks can be achieved by a Logic Analyzer. The normal oscilloscope deals with time domain, spectrum analyzer with frequency domain and the logic analyzer with digital domain.

2.11.4.1 Features

1. The logic analyzer displays signals from many inputs at a time. The number of inputs may be 16, 32, 48, 64 or even more.
2. The logic analyzer shows the display in sequence of instructions that have occurred.
3. It responds to the logic levels i.e. it only displays logic 0 or 1 state of input signal.
4. It takes input samples and stores the samples in its own memory.
5. It has capability of displaying the words that occurred before and after the trigger.
6. The logic analyzer displays data in four different ways. They are :
   (i) Timing diagram method
   (ii) Logic state method
   (iii) Hexadecimal method
   (iv) Map method
7. It is used to identify a malfunctioning system, with the help of its map display. In this method the map of system is compared with map of a good system.
8. The logic analyzer has a programmable trigger facility, which allows the user to take input sample at any instant.

2.11.4.2 Types of Logic Analyzers

SPPU - Aug. 15

University Question

Q. Write note on logic analyzers.
(Aug. 2015(In Sem), 5 Marks)

The logic analyzers are of two types :

(a) **Logic timing analyzer :** In this analyzer, the data is sampled as the clock signal is generated and at regular intervals. The data sampled is stored in the memory and this stored information is displayed. It is preferred for troubleshooting of the problems related to the computer hardware. It is an asynchronous measurement method.

(b) **Logic state analyzer :** In this analyzer, the data is sample when the clock signals are synchronized with the measured device. The data sampled is stored in the memory and this stored information is displaying in the binary or hexadecimal format. This method is preferred for troubleshooting of software problems. It is a synchronous measurement method.

- The logic analyzer is basically a multichannel oscilloscope. Fig. 2.11.1 shows the block diagram of a logic analyzer.
- The probes connect the logical analyzer to system which is under test. The probes operate as voltage divides, the lowest possible slew rate can be selected by dividing the input signal. This helps the device to capture high speed signals.
- The different logic families i.e. TTL, CMOS, NMOS etc. have different threshold voltages and hence adjustable threshold comparators are used. Each signal is connected to each line of the logic analyzer. The reference signal of each comparator is set to a voltage which is equal to the logic threshold voltage of the logic family under test.
- The logic analyzer memory consists of a RAM. The clock signals i.e. internal or external clock input is connected to the memory. On receiving clock signal, the logic analyzer samples the data present on input signals. These samples are stored in the memory. For each input channel the analyzer can store from 256 to 1024 samples.
- When the memory receives a trigger signal then the samples are stored in it and displayed on the CRT display. This trigger signal may be provided externally or it may be provided from the word recognizer circuitry.
- We can set a binary word using switches or through keyboard in the word recognizer circuit. The word recognizer circuit compares this word with the binary input word. When the two words match it sends a trigger signal to the memory. When the memory receives a trigger signal, it sends the samples to a CRT display. There are three types of displays, depending on the trigger signal.
  (i) Pre-trigger display (ii) Post-trigger display
  (iii) Center trigger display

(i) In **pretrigger mode**, the memory acts as a loop. The samples before the trigger event are captured which represent the time before event. It is useful to find the causes of malfunctioning of a circuit. It is also called as negative time capturing.

(ii) In the **post trigger mode**, the memory displays the samples which are captured after the occurrence of trigger signal.

(iii) In the **center trigger mode** half of samples are taken before the trigger signal and the other half samples are taken after the trigger signal.

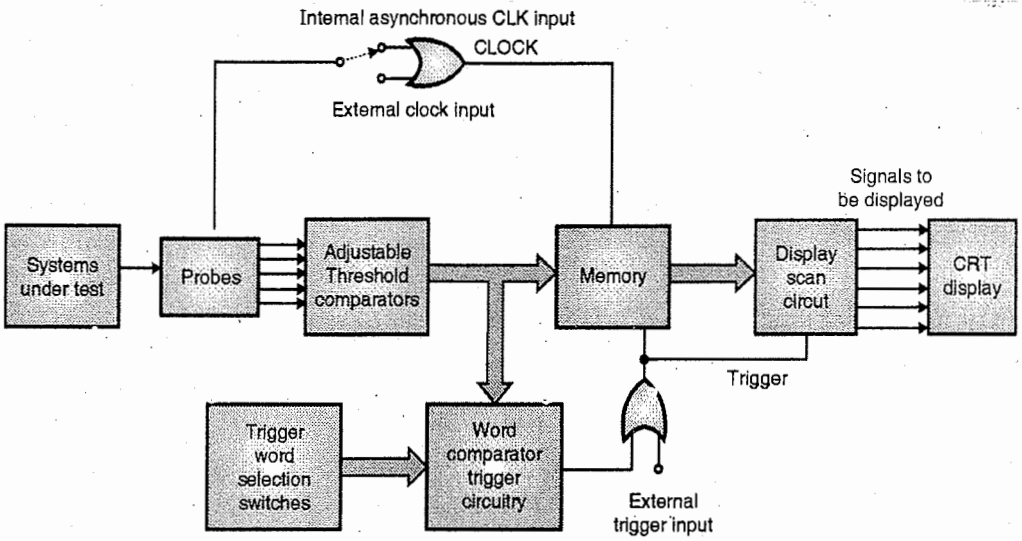


Fig. 2.11.1 : Block diagram of logic analyzer

2.11.4.3 Display Methods / Formats

The logic analyzer allows the user to display the samples it stores in the memory in four ways.

(i) Timing diagram method
(ii) Logic state method
(iii) Hexadecimal method
(iv) Map method

(i) Timing diagram method

In this method a particular portion could be zoomed in or zoomed out on the timing diagram. Such a display is best suitable for finding out glitches and displaying long data sequences.

Fig. 2.11.2 shows this method.

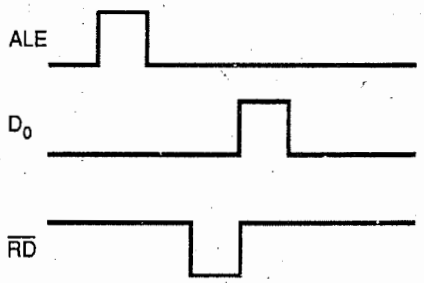


Fig. 2.11.2 : Timing diagram method

(ii) Logic state method

In this method the sampled data is displayed in logic state format of 1's and 0's. This format is very easy to read. Fig. 2.11.3 shows display of data sampled using logic state method.

A_0	0	0	1	0	0	1	0	0	1	1	1	1	1	1	1	1
A_1	0	1	0	0	0	1	1	0	0	1	1	1	0	0	0	0
A_2	1	0	1	0	1	0	0	1	1	0	1	1	1	1	0	1
A_3	0	1	0	1	1	0	1	1	0	0	0	1	1	0	1	1
A_4	1	0	1	1	0	1	0	0	1	1	0	1	0	0	1	1
A_5	0	1	0	1	0	1	1	0	0	1	0	0	1	0	0	1
A_6	0	0	1	0	1	0	0	1	1	0	1	1	1	1	1	1

Fig. 2.11.3

(iii) Hexadecimal method

This method is used whenever it is essential to improve the readability of sampled data. The sampled data is displayed in hexadecimal format. Fig. 2.11.4 shows this. Whenever the contents of address and data bus are to be observed, this method is used.

A_0 - A_7	20	50	7F	30	31	2E	3B	5A
D_0 - D_7	54	76	8E	2B	1A	3C	4D	7E

Fig. 2.11.4

(iv) Map method

In this method the data sampled is sampled in the form of a map. This method is most difficult method in regards to reading the data, but it is very useful inorder to identify a malfunctioning system. This is done by comparing its map with that of a good system.

2.11.4.4 Applications of a Logic Analyzer

- They are used for the troubleshooting and analysis of complex digital systems.
- As it has the facility of asynchronous internal clocking, the activities in the system which is under the test can be seen on real time basis.
- It can be used to observe up to 64 signals at a time, while the oscilloscope can be used to observe 4 channels at a time.
- It has compatibility with RS 232 and IEEE 488 and can be attached to printer, for taking hard copy of the signals.
- It is able to detect and trigger on signal glitches. A glitch is a signal which makes a transition through the threshold voltage two or more times between the successive clock samples. Glitches are unwanted signals. They can cause malfunctions in the system.

❑❑❑

UNIT - III

CHAPTER 3

Parallel Port Interfacing-II

Syllabus Topic : Interfacing of DAC

3.1 Interfacing of DAC

3.1.1 DAC 0808

The DAC 0808 is an 8-bit current output monolithic DAC manufactured by the National semiconductor corporation. It is a 16 - pin IC available in dual in line DIP plastic package. The analog output is available in the form of current I_o. That means I_o is proportional to the 8-bit digital input. The important features of DAC 0808 are as follows :

3.1.1.1 Features of DAC 0808

Fast settling time	150 nsec. typically
Power supply voltage range	± 4.5 mW at ± 5V
Low power consumption.	33 mW at ± 5V.
High speed multiplying input slew rate	8 mA / μ sec.
Interfaces directly with TTL, DTL and CMOS logic levels.	

3.1.1.2 Pin Configuration and Functional Block Diagram

- The pin configuration and functional block diagram of DAC 0808 are as shown in Figs. 3.1.1(a) and (b) respectively.

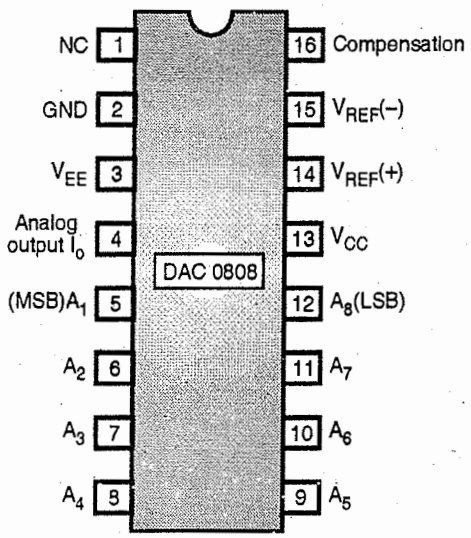


Fig. 3.1.1(a) : Pin configuration of DAC 0808

- The internal block diagram shows that DAC 0808 consists of R-2R ladder along with current switches and reference current amplifier.
- A_1 to A_8 are the 8-digital input lines with A_1 as the most significant bit and A_8 as the least significant bit.
- The analog output is available in the form of current I_o, therefore we need to use an external current to voltage converter if the analog output in the form of voltage is required.
- DAC 0808 requires a dual polarity (±) supply voltage, typically ± 15V, for its operation. The reference voltage can be either positive or negative.
- An external reference voltage should be applied to either V_{REF} (+) or V_{REF} (–) depending on the polarity of reference voltage.

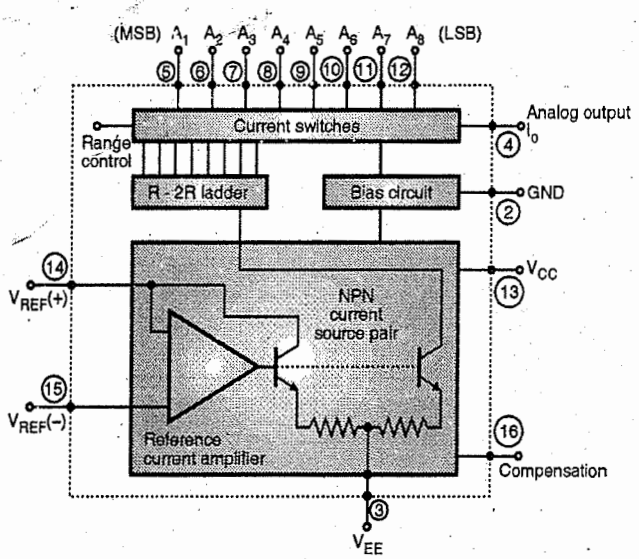


Fig. 3.1.1(b) : Functional block diagram of DAC 0808

3.1.2 Interfacing DAC to 8051

Q. Interface DAC 0808 to 8051 microcontroller with timing diagram.

Fig. 3.1.2 shows the interfacing of DAC 0808 to 8051.

- The **output of DAC** is a current which is converted into voltage using opamp based current-to-voltage (I-V) converter.

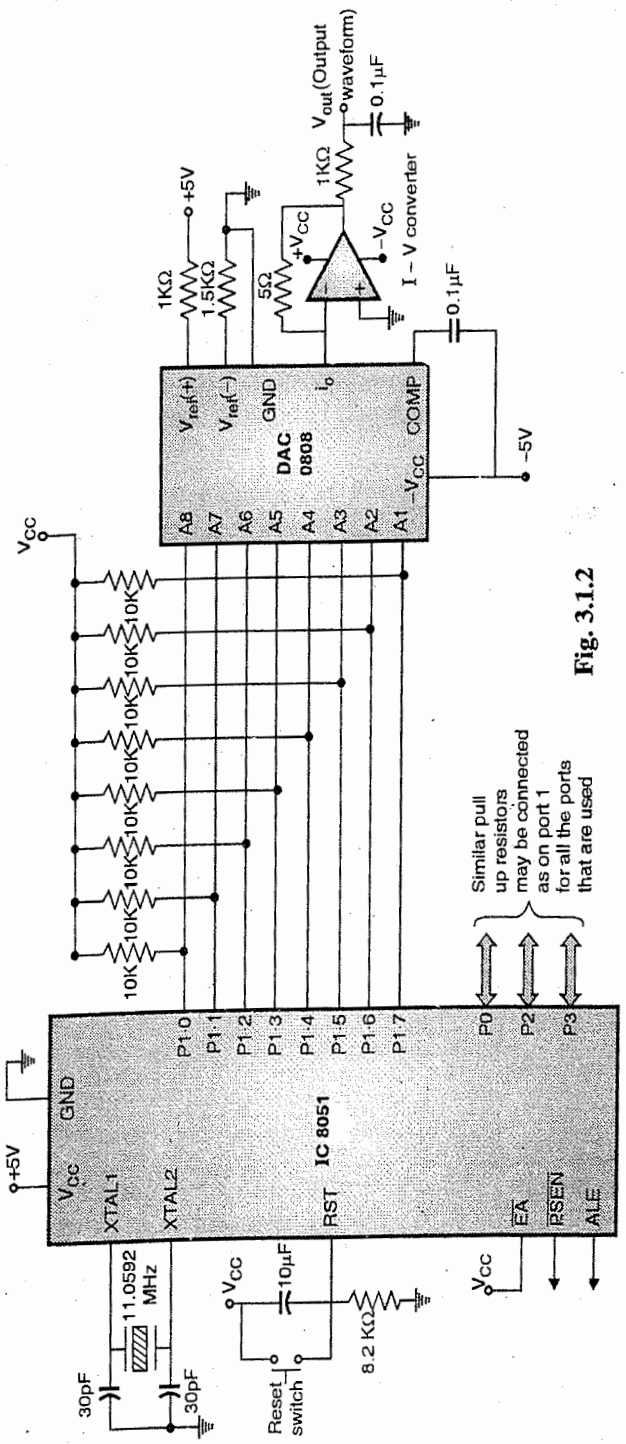


Fig. 3.1.2

– The analog output current I_{out} of the DAC depends on the I_{ref} flowing into the V_{ref} terminal and the status of the D_0 – D_7 bits. It is expressed as,

$$I_{out} = I_{ref}\left(\frac{D_7}{2}+\frac{D_6}{4}+\frac{D_5}{8}+\frac{D_4}{16}+\frac{D_3}{32}+\frac{D_2}{64}+\frac{D_1}{128}+\frac{D_0}{256}\right)$$

where, D_7 = MSB

I_{ref} depends on V_{ref} voltage and the resistors 1 KΩ and 1.5 KΩ connected.

$$I_{ref} = \frac{V_{ref}}{1\text{ K} + 1.5\text{ K}} = \frac{5\text{ V}}{2.5\text{ K}\Omega} = \mathbf{2\ mA}$$

(a) If $D_0 - D_7$ = FFH then

$$I_{out} = 2\text{ mA}\times\left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\frac{1}{256}\right)$$

$$= 2\text{ mA}\times\frac{255}{256}$$

$$= 1.99\text{ mA}$$

$$V_{out} = 1.99\text{ mA}\times 5\text{ K}\Omega$$

$$= \mathbf{9.96\ V}$$

(b) If $D_0 - D_7 = 80$ H then

$$I_{out} = 2\text{ mA}\times\left(\frac{1}{2}+\frac{0}{4}+\frac{0}{8}+\frac{0}{16}+\frac{0}{32}+\frac{0}{64}+\frac{0}{128}+\frac{0}{256}\right)$$

$$= 1\text{ mA}$$

$$V_{out} = 1\text{ mA}\times 5\text{ K}\Omega = 5\text{ V}$$

(c) If $D_7 - D_0 = 00$ H then $\mathbf{I_{out} = 0, V_{out} = 0V}$

DAC is commonly used in waveform generation as shown in examples.

Ex. 3.1.1 Lab Assignment

Design a 8051 based system to interface DAC. Write the C and an assembly language programs to generate.

(i) Triangular wave

(ii) Sinusoidal wave

(iii) Trapezoidal wave

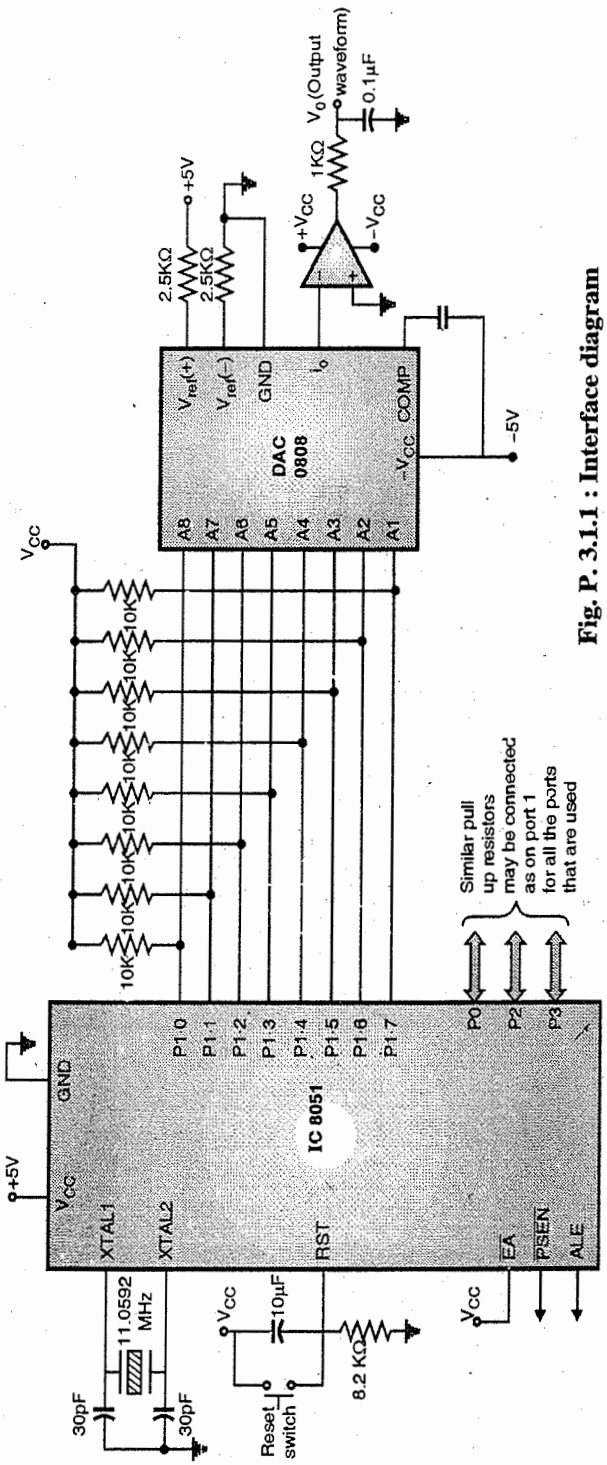


Fig. P. 3.1.1 : Interface diagram

Soln. :

(i) Triangular wave generation

Algorithm

Main Program

Step I : Initialize port 1 to all 0s

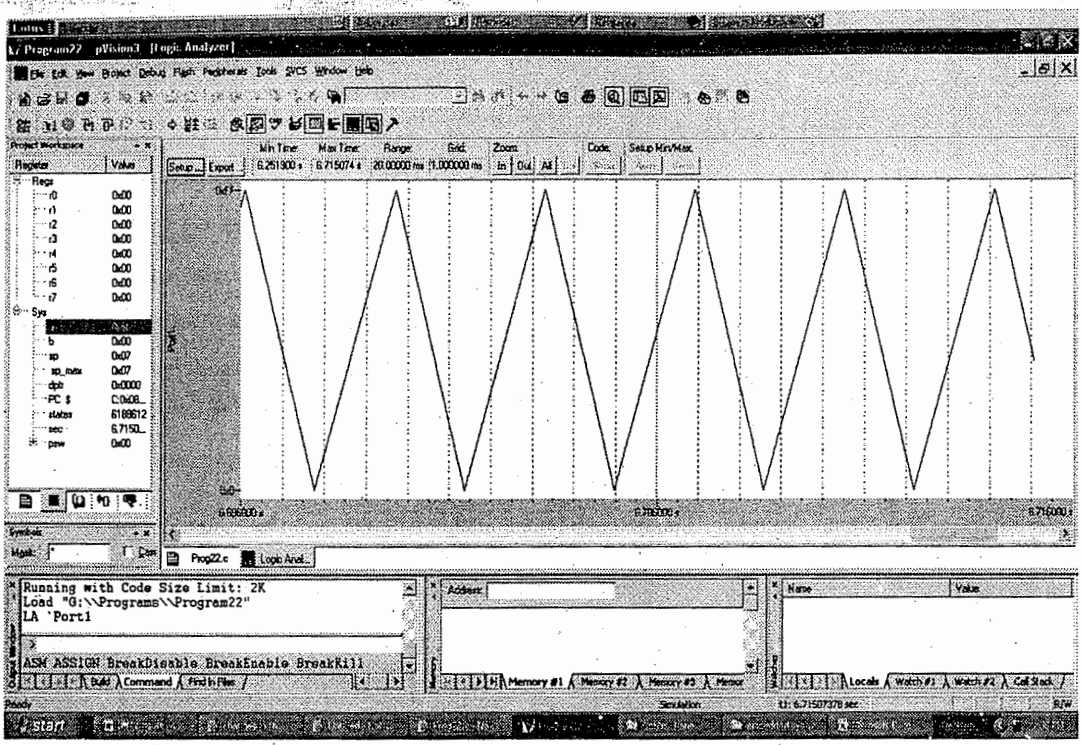
Step II : Increment the data in P1 till it reaches the maximum i.e. FFH.

Step III : Once it reaches the maximum, decrement the data in P1 till it reaches minimum i.e. 00H.

Assembly program

Label	Instruction	Comments
	ORG 0000H	
	LJMP START	
	ORG 0100H	
START :	MOV A, #00H	Initialise Port 1 to all 0's
HERE :	MOV P1, A	
	INC A	
	CJNE A, #0FFH,HERE	Increment the data in port 1 till it reaches the maximum i.e. FFH
NXT:	MOV P1, A	
	DEC A	
	CJNE A, #00H,NXT	Once it reaches maximum, decrement the data in P1 till it reaches minimum i.e. 00H
	SJMP HERE	
	END	

Output



(ii) Sinusoidal wave generation

We need to calculate the values for sinusoidal waveform between 00H and FFH. This is done by the following table. Here we have divided the entire cycle of 360° into 48 parts each of incremental 7.5°. Hence the angles are 0°, 7.5 °, 15°, 22.5°....

We cannot have negative voltage in the output of our microcontroller. Hence for Sin 0, we need the output to be in centre i.e. 2.5V, treating it as x-axis. Hence we add 2.5V to every calculation as seen in the fourth row of the table below. Then we find the corresponding value for each voltage by multiplying it with the maximum count and dividing it by maximum voltage as given in the fifth column of the table.

Note: If you still reduce the step size from 7.5°, you may get a better waveform. For the program in assembly, we have taken the angles at an interval of 30°. This gives less accurate output.

Calculation of array elements

Sr. No.	Angle in degrees (θ)	sin (θ)	Count = 2.5+(2.5 * sin θ)	count*256/5
0	0	0	2.5	128
1	7.5	0.130578428	2.826446071	145
2	15	0.258920827	3.147302068	161
3	22.5	0.382829457	3.457073643	177
4	30	0.500182502	3.750456255	192
5	37.5	0.608970405	4.022426013	206
6	45	0.707330278	4.268325695	219
7	52.5	0.793577803	4.483944508	230
8	60	0.866236075	4.665590188	239
9	67.5	0.924060891	4.810152227	246
10	75	0.966062056	4.915155141	252
11	82.5	0.991520342	4.978800856	255
12	90	0.9999998	4.9999995	256
13	97.5	0.991355227	4.978388068	255
14	105	0.965734654	4.914336634	252
15	112.5	0.923576807	4.808942018	246
16	120	0.8656036	4.664008999	239
17	127.5	0.792807767	4.482019417	229

Sr. No.	Angle in degrees (θ)	sin (θ)	Count = 2.5+(2.5 * sin θ)	count*256/5
18	135	0.706435867	4.266089666	218
19	142.5	0.607966935	4.019917336	206
20	150	0.499087156	3.74771789	192
21	157.5	0.381660992	3.45415248	177
22	165	0.257699252	3.144248131	161
23	172.5	0.129324662	2.823311654	145
24	180	0.001264489	2.496838778	128
25	187.5	0.131831986	2.170420034	111
26	195	0.260141988	1.849645029	95
27	202.5	-0.38399731	1.540006725	79
28	210	0.501277049	1.246807379	64
29	217.5	0.609972902	0.975067745	50
30	225	0.708223559	0.729441104	37
31	232.5	0.794346571	0.514133573	26
32	240	0.866867165	0.332832087	17
33	247.5	0.924543497	0.188641258	10
34	255	0.966387914	0.084030214	4
35	262.5	0.991683872	0.02079032	1
36	270	0.999998201	4.9999	256
37	277.5	0.991188527	0.022028682	1
38	285	0.965405707	0.086485732	4
39	292.5	0.923091247	0.192271883	10
40	300	-0.86496974	0.337575649	17
41	307.5	0.792036463	0.519908843	27
42	315	0.705540326	0.736149186	38
43	322.5	0.606962492	0.98259377	50
44	330	0.497991012	1.25502247	64
45	337.5	0.380491917	1.548770208	79
46	345	0.256477265	1.858806837	95
47	352.5	0.128070688	2.17982328	112
48	360	0.002528976	2.50632244	128

Algorithm

(A) Main Program

Step I : Initialize data according to the above table

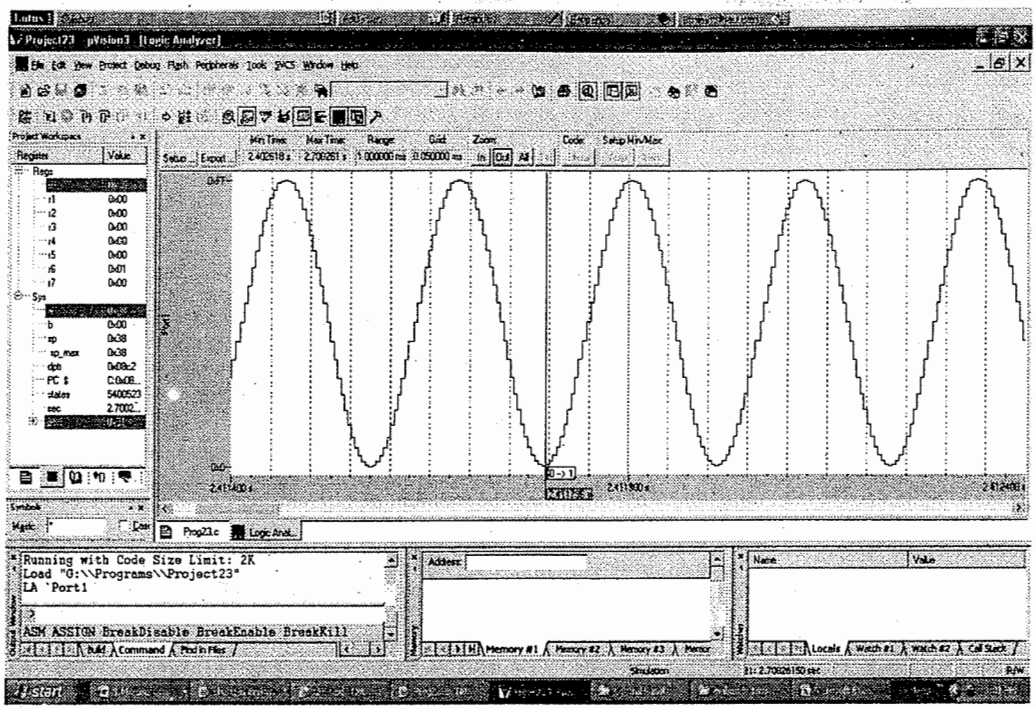
Step II : Issue the data one by one from the array.

Step III : Repeat the above step continuously to get a continuous waveform

Assembly Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP START	
	ORG 0100	
START:	MOV R0, #30H	data according to above calculation
	MOV @R0, #128	
	INC R0	
	MOV @R0, #192	
	INC R0	
	MOV @R0, #238	
	INC R0	
	MOV @R0, #255	
	INC R0	
	MOV @R0, #238	
	INC R0	
	MOV @R0, #192	
	INC R0	
	MOV @R0, #128	
	INC R0	
	MOV @R0, #64	
	INC R0	
	MOV @R0, #17	
	INC R0	
	MOV @R0, #0	
	INC R0	
	MOV @R0, #17	
	INC R0	
	MOV @R0, #64	
	MOV R0, #30H	
HERE:	MOV P1, @R0	Issue the data one by one from the array
	INC R0	
	CJNE R0, #3CH,HERE	If last data is issued, repeat the procedure from beginning
	MOV R0, #30H	
	SJMP HERE	
	END	

Output



(iii) Trapezoidal wave generation

Algorithm

Step I : Initialize port 1 to all 0s

Step II : Increment the data in P1 till it reaches the maximum i.e. FFH

Step III : Small software delay for flat top

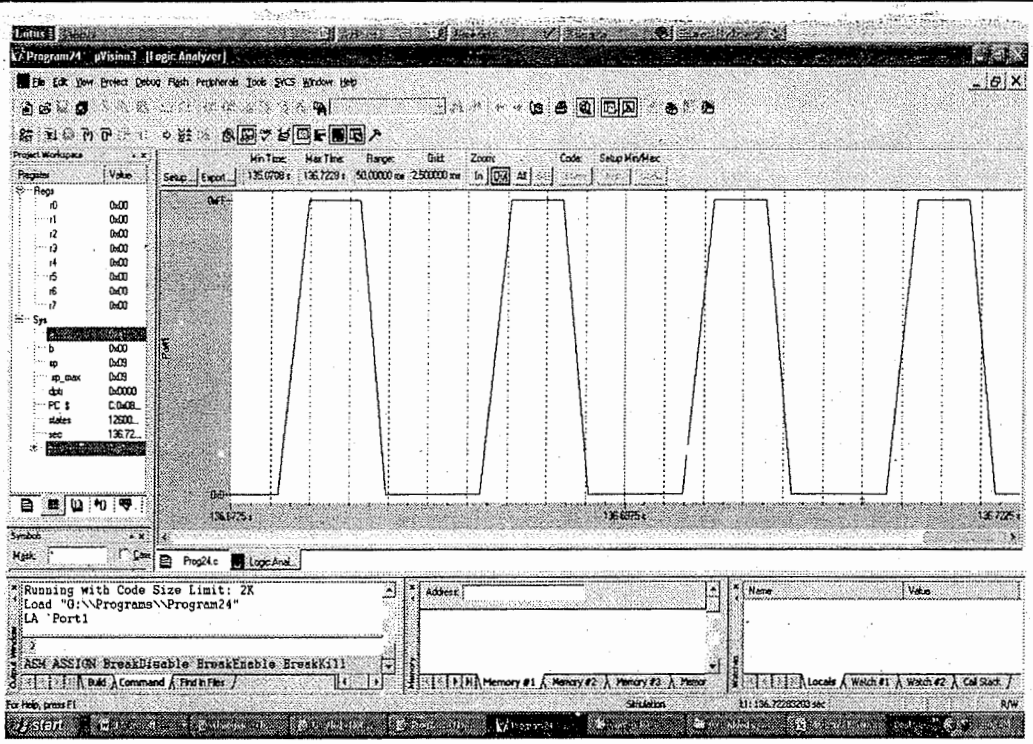
Step IV : Once it reaches the maximum, decrement the data in P1 till it reaches minimum i.e. 00H

Step V : Small software delay for flat bottom

Assembly Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP START	
	ORG 0100H	
START:	MOV A, #00H	Initialize Port 1 to all 0's
HERE:	MOV R7, #0FFH	Small software delay for flat top
HERE1:	DJNZ R7, HERE1	
AGAIN:	MOV P1, A	
	INC A	
	CJNE A, #0FFH,AGAIN	Increment the data in port 1 till it reaches the maximum i.e. FFH
	MOV R7, #0FFH	Small software delay for flat bottom
HERE2:	DJNZ R7, HERE2	
NXT:		
	MOV P1, A	
	DEC A	
	CJNE A, #00H,NXT	Once it reaches maximum, decrement the data in P1 till it reaches minimum i.e. 00H
	SJMP HERE	
	END	

Output



Ex. 3.1.2

Draw the hardware interfacing connections between 8051 and DAC0808. Write an assembly language program to obtain a five step staircase waveform at analog output of DAC.

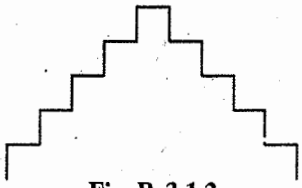


Fig. P. 3.1.2

Soln. : For interfacing diagram, refer Fig. P. 3.1.1.

Program :

Label	Instruction
L1 :	MOV DPTR, #Table
	MOV R2, #Count
BACK :	CLR A
	MOVC A, @A+DPTR
	MOV P1, A
	INC DPTR
	DJNZ R2, BACK
	SJMP L1
	ORG 0100 H
	Table dB 01H, 02H, 03H, 04H, 05H, 04H, 03H, 02H, 01H

Ex. 3.1.3

Write an assembly language program for square wave generation.

OR

Draw the block diagram for interfacing DAC 0808 with 8051 microcontroller. Write an assembly language program to generate square wave.

Soln. : A square wave has only two amplitudes, a minimum say 0 V (00H) and a maximum of 10 V (FFH). To generate square wave we have to output 00H and then FFH to Port 1 of 8051. The Port 1 is connected as input to DAC 0808. According to the frequency requirement delay is provided between the outputs. Fig. P. 3.1.1 shows the interfacing diagram for a 1 KHz square assuming 50% duty cycle, the time period.

$$T = \frac{1}{f} = 1 \text{ ms and } T_{ON} = T_{OFF} = 0.5 \text{ ms}$$

Assembly Program

Label	instruction	Comments
AGAIN :	MOV A, #00H	
	MOV P1, A	Output low
	ACALL DELAY	0.5 ms delay
	MOV A, #0FFH	
	MOV P1, A	Output high
	ACALL DELAY	
	SJMP AGAIN	
DELAY :	MOV R0, #0FFH	
HERE :	DJNZ R0, HERE	
	RET	
	END	

Ex. 3.1.4

Write a program to generate a sawtooth waveform for 8051. Draw flowchart.

Soln. : The interfacing diagram is same as shown in Fig. P. 3.1.1. A sawtooth (ramp) waveform is shown in Fig. P. 3.1.4. It has an amplitude that is continuously increasing to maximum value.

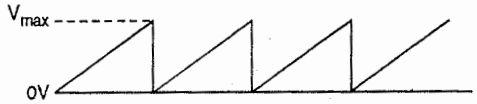


Fig. P. 3.1.4 : Sawtooth waveform

From an initial value say 00H, the value at P1 is continuously incremented. (P1 increments from 00 to FFH and the next increment makes FFH to 00H. Thus, a sawtooth waveform is generated)

Assembly Program

Label	Instruction	Comments
	MOV A, #00H	Load initial value in A
AGAIN :	MOV P1, A	Output value on P1
	INC A	Increment value in A
	SJMP AGAIN	

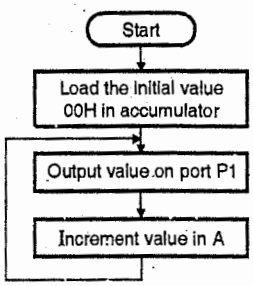


Fig. P. 3.1.4(a)

Syllabus Topic : Interfacing of Temperature Sensor (LM35)

3.2 Interfacing of Temperature Sensor (LM35)

- Temperature is the most-measured process variable in industrial automation. Most commonly, a temperature sensor is used to convert temperature value to an electrical value. Temperature Sensors are the key to read temperatures correctly and to control temperature in industrials applications.
- The LM34 are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Fahrenheit temperature.
- The LM35 are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Celsius (Centigrade) temperature.
- The LM34/LM35 thus has an advantage over linear temperature sensors calibrated in degrees Kelvin, as the user is not required to subtract a large constant voltage from its output to obtain convenient Fahrenheit scaling.
- LM35 does not require any external calibration or trimming to provide typical accuracies of ±1/4°C at room temperature and ± 3/4°C over a full –55 to +150°C temperature range.
- The LM35 is rated to operate over a – 55°C to +150°C temperature range.
- As shown in the Fig. 3.2.1 the connection for LM35 is very simple. Also the output scale is linear with a change of 10mV/°C.

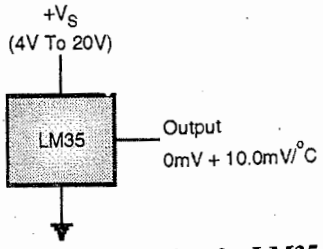


Fig. 3.2.1 : Circuit diagram for the LM35 basic temperature sensor (+2° C to +150° C)

Ex. 3.2.1

Assume P1 is an i/p port connected to a temperature sensor. Write a program to read the temperature and test it for the value 75. According to the rest results, place the temperature value into the resisters indicated by the following :

If T = 75 then A = 75 ; If T < 75 then R1 = T ;

If T > 75 then R2 = T

Soln. :

Label	Instruction	Comments
	MOV P1,#0xFF	Initialize port 1 as input port
	MOV R7,P1	Read temperature from port 1 to the register R7
	CJNE R7,#0x4B,nxt	Compare R7 with 75 (4BH)
	MOV A,R7	Move the data in acc. if temperature is 75
	SJMP over	
nxt:	JNC over1	Check if temperature reading is greater than 75
	MOV R1,R7	If less than 75, move the temperature in R1
	SJMP over	
over1:	MOV R2,R7	If more than 75, move the temperature in R1
over:	SJMP over	

Ex. 3.2.2

Initialize port 0 as an input port and write a program to read the temperature from temperature sensor connected to P0.

Soln. :

```
MOV P0, #0FFH      ; Configure port 0 as input port
MOV A, P0          ; Read the temperature from port 0
```

Ex. 3.2.3

Design a 8051 based system to interface LM35 using ADC. Write the corresponding Assembly program to display the temperature on LCD.

Soln. : Part (a) : Design

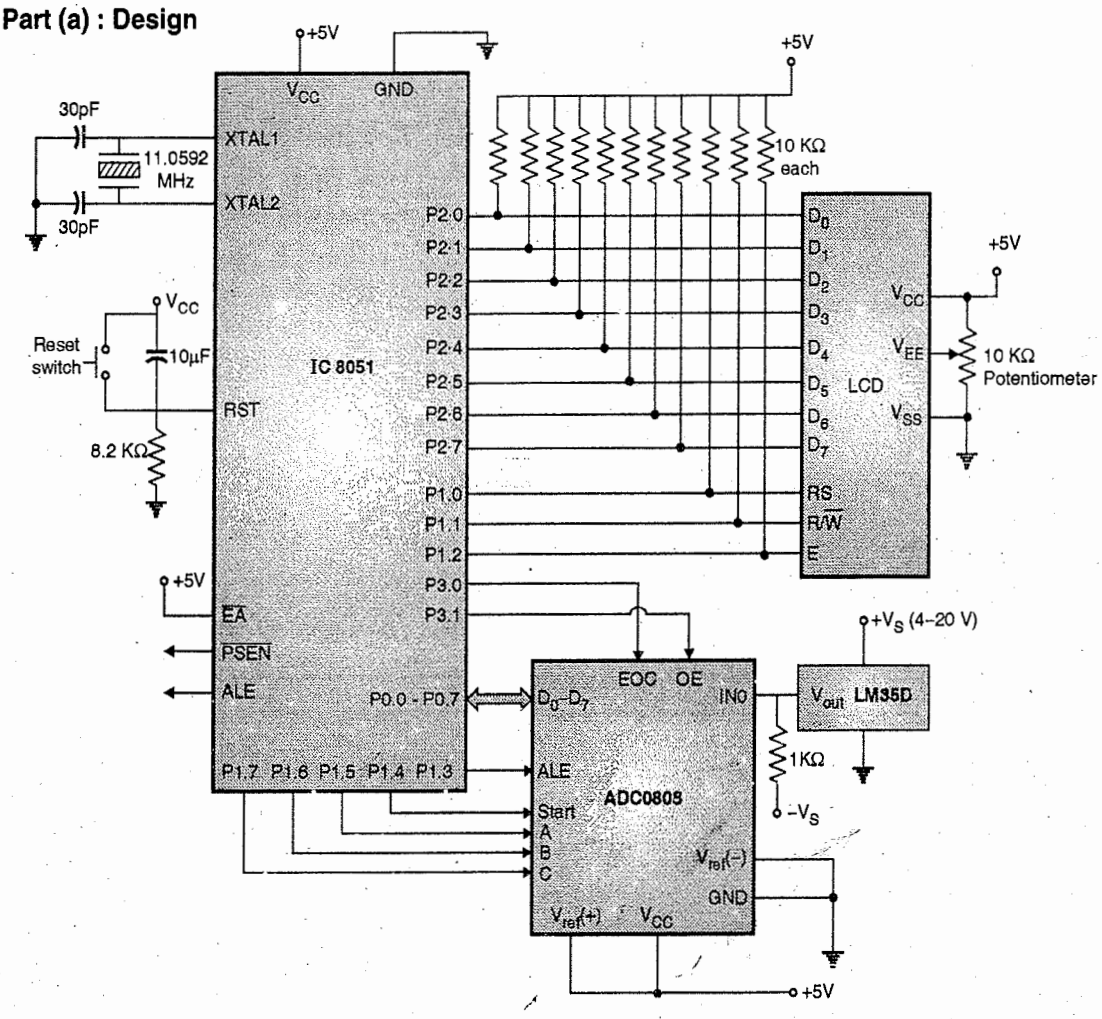


Fig. P. 3.2.3 : Interface diagram

Part (b) Program

Algorithm

A) Main Program

Step I : Initialize Port 1 as input port.

Step II : Initialize pin used for End of conversion (EOC) as input.

Step III : Clear the OE (used for read), start (that indicates start of conversion to ADC) and ALE (used to latch the address).

Step IV : Issue the address of channel 0 on the CBA pins.

Step V : Issue pulse on ALE to latch the address.

Step VI : Initialize the LCD.

Step VII : Wait for some software delay after power up for display to stabilize.

Step VIII : Initialize the LCD by giving the instruction 0x38 to the command subroutine.

Step IX : Wait for small software delay.

Step X : Issue the command 0x0E to command subroutine for display on, cursor on.

Step XI : Wait for small software delay.

Step XII : Issue the command 0x01 for clearing display to command subroutine.

Step XIII : Wait for small software delay.

Step XIV : Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.

Step XV : Wait for small software delay.

Step XVI : Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.

Step XVII : Issue start of conversion.

Step XVIII : Wait for EOC to become 0 and then wait for it to become 1. This indicates the ADC has completed the conversion.

Step XIX : Calculate the digits and display it on LCD.

Step XX : Also display the unit of temperature.

B) Command subroutine

Step I : Give the instruction to the port connected to data bus of the LCD.

Step II : Make RS = '0', to indicate instruction.

Step III : Make R/$\overline{W}$ = '0', to indicate write.

Step IV : Make E = '1'

Step V : Wait for 120 µsec.

Step VI : Make E = '0'

(Step IV – Step VI: To give a high-to-low pulse on E pin so as to latch the command)

Step VII : Return.

C) Data subroutine

Step I : Check if LCD is ready by calling ready subroutine.

Step II : Give the data to the port connected to the data bus of the LCD.

Step III : Make RS = '1', to indicate data

Step IV : Make R/$\overline{W}$ = '0', to indicate write

Step V : Make E = '1'

Step VI : Wait for 120 µsec.

Step VII : Make E = '0'

(Step V – Step VII: To give a high-to-low pulse on E pin so as to latch the data)

Step VIII : Return

D) Ready subroutine

Step I : Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.

Step II : Make RS = '0' to indicate instruction.

Step III : Make R/$\overline{W}$ = '1', to indicate read.

Step IV : Make E = 0

Step V : Make E = 1

Step VI : Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.

Step VII : Return.

Assembly Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP Start	
	ORG 0100H	Function to write the command to LCD
COMMAND:		
	MOV P2, R3	Write the command on the Port 2 so as to issue it to LCD.
	CLR P1.0	RS=0, Indicates instruction.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on en pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0200H	
DISPLAY:		Function to write data to LCD
	LCALL READY	Check if the LCD is ready by calling the ready function.
	MOV P2, R3	Write the data on the Port 2 so that it is given to the LCD.
	SETB P1.0	RS=1, Indicates data.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on en pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0300H	Function to check if LCD is busy or ready.

Label	Instruction	Comments
READY:	SETB P2.7	Making the P2.7 pin as input pin by writing a'1' on t.
	CLR P1.0	RS=0, Indicates instruction and not data
	SETB P1.1	RW=1, Indicates read and not write
WAIT:	CLR P1.2	A low to high going pulse on en pin.
	LCALL DELAY	
	SETB P1.2	
	JB P2.7, WAIT	Wait till the LCD is busy.
	RET	
	ORG 0400H	
DELAY:		A subroutine to implement small software delay
	MOV R5, #1CH	
REP:	DJNZ R5,REP	
	RET	
	ORG 1000H	
Start:	MOV R4, #0FFH	Wait for some time for LCD to stabilize when power-on.
AGAIN1:	MOV R5, #0FFH	
AGAIN:	DJNZ R5,AGAIN	
	DJNZ R4,AGAIN1	
	MOV R3, #38H	Issue the command to initialize 16 × 2 LCD.
	LCALL COMMAND	
	MOV R3, #0FH	Issue the command for display on, cursor on and cursor blinking.
	LCALL COMMAND	
	MOV R3, #01H	Issue the command to clear display.
	LCALL COMMAND	
	MOV R3, #06H	Issue the command to increment cursor position on every character written.
	LCALL COMMAND	Issue the command to position the cursor on the first position on line 1.
	MOV R3, #80H	
	LCALL COMMAND	
	MOV P1, #0FFH	P1 as input
	SETB P3.0	EOC as input
	CLR P1.3	Clear ALE
	CLR P3.1	Clear OE
	CLR P1.5	Select Channel 0
	CLR P1.6	
	CLR P1.7	
	LCALL DELAY	
	SETB P1.3	Latch address
	LCALL DELAY	
	CLR P1.3	

Label	Instruction	Comments
HERE:	SETB P1.4	Start of conversion
AGAIN3:	JB P3.0, AGAIN3	
AGAIN4:	JNB P3.0, AGAIN4	Wait for end of conversion
	SETB P3.1	Enable read
	LCALL DELAY	
	MOV R7, P0	Take input from ADC
	CLR P3.1	Disable read
	MOV A, R7	
	MOV B, #100	Calculate and display the count
	DIV AB	
	MOV R3, A	
	LCALL DISPLAY	
	MOV A,B	
	MOV B, #10	
	DIV AB	
	MOV R3, A	
	LCALL DISPLAY	
	MOV R3, B	
	LCALL DISPLAY	
	MOV R3, #'°'	The count is directly in degree Celsius.
	LCALL DISPLAY	
	MOV R3, #'C'	
	LCALL DISPLAY	
	MOV R3, #0x80	
	LCALL COMMAND	
	SJMP HERE	
	END	

Syllabus Topic : Interfacing of Stepper Motor

3.3 Interfacing of Stepper Motor

- A stepper motor is a device that translates electrical pulses to mechanical movement. It is used in applications like robotics, dot matrix printers, disk drives for position control.
- Stepper motors have a permanent magnet rotor surrounded by a stator.
- Generally the stepper motors have four stator windings that are paired with a common center tap as shown in Fig. 3.3.1.
- Such a stepper motor is called as four phase or unipolar stepper motor.
- With the help of center tap the current direction in each of the two coils can be changed when a winding is grounded. This results in a polarity change of stator.

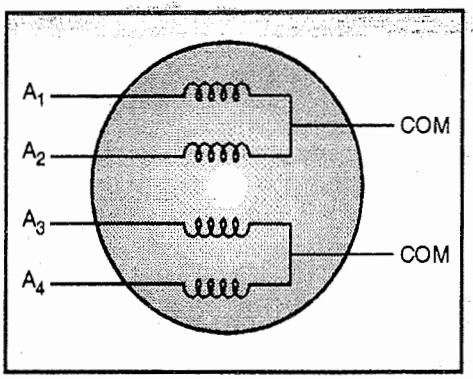


Fig. 3.3.1 : Stator windings configuration

- The stator is responsible for the direction of rotation by the stator poles. The stator poles are determined by the current sent through the wire coils. As the direction of current is changed, the polarity is also changed causing the reverse motion of the stepper motor.
- The stepper motor has 6 leads, 4 leads representing four stator windings and 2 leads for 2 commons for the center tapped leads. On application of power to the stator the rotor rotates.
- There are many sequences for rotation. Each sequence has a different degree of precision.

Table 3.3.1 shows a 2 phase 4 step stepping sequence.

Table 3.3.1 : 4 step sequence (Full stepping sequence)

Step	Winding A_1	Winding A_2	Winding A_3	Winding A_4
1	1	0	0	1
2	1	1	0	0
3	0	1	1	0
4	0	0	1	1

Counter clockwise ↑ ↓ Clockwise

- Although we can start with any sequence, we must continue in proper order. e.g. if we start with step 2 then the sequence of steps is 3,4,1 etc.
- Although we can start with any sequence, we must continue is proper order. e.g. if we start with step 2 then the sequence of steps is 3,4,1 etc.

Step angle

- It is the minimum degree of rotation associated with a single step.

Table 3.3.2 gives some step angles.

Table 3.3.2 : Stepper motor step angles

Step angle	Steps per revolution
0.72	500
1.8	200
2.0	180
2.5	144
5.0	72
7.5	48
15	24

The total number of steps required to rotate one complete rotation of 360° is called as **steps per revolution.**

$$\text{Steps per second} = \frac{\text{rpm} \times \text{Steps per revolution}}{60}$$

Ex. 3.3.1

Draw 8051 connection to stepper motor and code a program to continuously rotate it.

Soln. :

Fig P. 3.3.1 shows 8051 connection to stepper motor. The four leads of the stator winding are controlled by port 1 bits P1.0 – P1.3. The 8051 does not have sufficient current to drive the stepper motor windings. Hence, a driver like ULN 2003 is required to energize the stator.

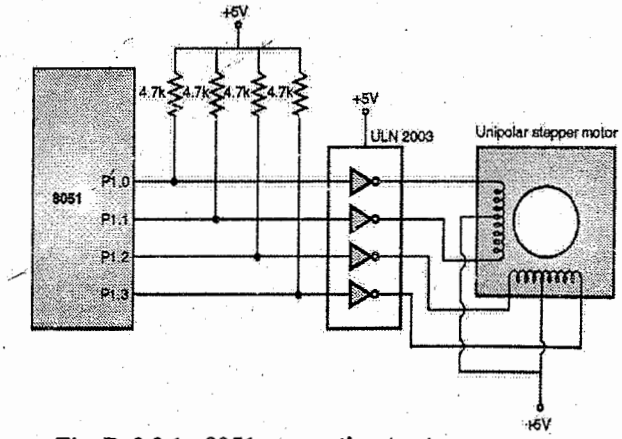


Fig. P. 3.3.1 : 8051 connection to stepper motor

Program :

Label	Instruction	Comments
	MOV A, #66 H	Load step sequence
L1 :	MOV P1, A	Give sequence to motor
	RR A	Rotate right clockwise
	ACALL DELAY	Wait for sometime
	SJMP L1	Continue doing
DELAY :	MOV R2, #100	
L3 :	MOV R3, #255	

Label	Instruction	Comments
L2 :	DJNZ R3, L2	
	DJNZ R2, L3	
	RET	

3.3.1 Four Step Sequence and 8 Step Sequence

- Table 3.3.1 shows 4 step sequence. As after switching four steps the same two windings will be 'on'.
- After completing four steps, the rotor moves only one tooth pitch. Hence, in a stepper motor with 180 steps per revolution, the rotor has 45 teeth as 4 × 45 = 180 steps are required to complete one revolution.
- The minimum step angle in function of number of teeth on rotor.
- The four step sequence is also called as **full stepping** sequence.
- To allow finer resolutions all stepper motors use an 8 step switching sequence. It is also called as **half stepping.**
- Each step is the half of the normal step angle. Table 3.3.3 shows a half stepping sequence.

Table 3.3.3 : Half stepping sequence

Step	Winding A_1	Winding A_2	Winding A_3	Winding A_4
1	1	0	0	1
2	1	0	0	0
3	1	1	0	0
4	0	1	0	0
5	0	1	1	0
6	0	0	1	0
7	0	0	1	1
8	0	0	0	1

Counter clockwise ↑ ↓ Clockwise

3.3.2 Using Transistors as Drivers

Fig. 3.3.2 shows an interface to unipolar stepper motor using transistors.

- Diodes are used to reduce the back EMF spike created when the coils are energized and deenergized in the same manner as the electromechanical relays.
- TIP 120 Darlington transistors can be used to supply higher current to the motors.

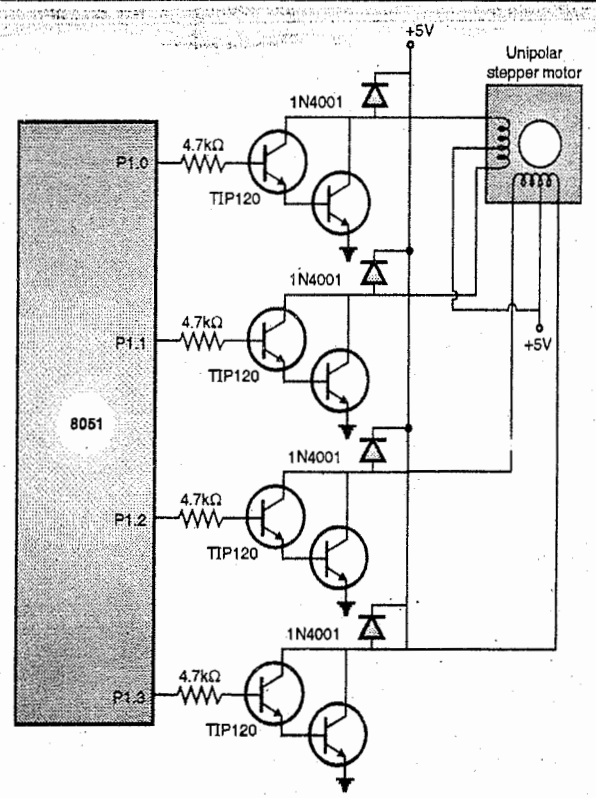


Fig. 3.3.2 : Using transistor drivers for stepper motor

3.3.3 Controlling Stepper Motor Via Opto-isolator

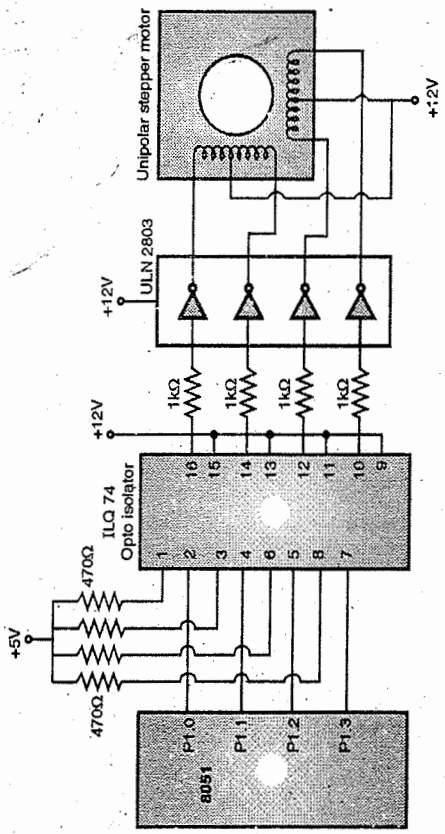


Fig. 3.3.3 : Controlling stepper motor with opto-isolator

- The opto-isolators are widely used to isolate the stepper motors EMF voltage and keep it from damaging the microcontroller system.

Fig. 3.3.3 shows stepper motor control using opto-isolator

Ex. 3.3.2

Write an assembly program to rotate a motor 117° in clockwise direction. The motor has a step angle of 1.8°

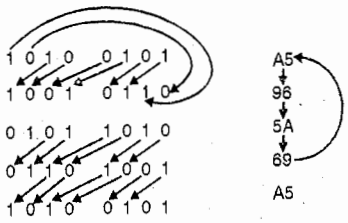
Soln. : 1 step = 1.8°

x steps = 117°

$$x = \frac{117}{1.8}$$

$$= 65$$

∴ 65 steps are to be given to stepper motor.



The sequence to be given is 0x05, 0x06, 0x04, 0x0A and 0x09. If we load 0xA5 in Accumulator and rotate it left twice, it gives the next code in LSBs. If we repeat it we get the third code and so on.

Hence we will rotate the data left twice, to generate the next step code in the above manner.

Label	Instruction
	org 0000H
	LJMP main
	org 1100H
delay:	
	MOV R4,#250
here :	DJNZ R4,here
	RET
	org 1000H
main :	MOV R0, #65
	MOV A, #0A5H
next :	MOV P0,A
	ACALL delay
	RL A
	RL A
	DJNZ R0, next
	END

Ex. 3.3.3

Write an assembly program to rotate the stepper motor clockwise if P1.0 = '1' and anticlockwise if P1.0 = '0'.

Soln. :

Label	Instruction
	org 0000H
	LJMP main
	org 1100H
delay:	
	MOV R4,#250
here :	DJNZ R4,here
	RET
	org 1000H
main :	MOV A,#0A5H
cw :	JNB P1.0, acw
	MOV P0,A
	ACALL delay
	RL A
	RL A
	SJMP cw
acw :	JB P1.0,cw
	MOV P0,A
	ACALL delay
	RR A
	RR A
	END

Ex. 3.3.4

A switch is connected to pin P2.7. Write Assembly language program to rotate 4-winding stepper motor.

(1) If switch = 0, the stepper motor rotate in clockwise

(2) If switch = 1, the stepper motor rotate in anticlockwise.

Soln. :

Label	Instruction
	org 0000H
	LJMP main
	org 1100H
delay:	
	MOV R4,#250
here :	DJNZ R4,here
	RET
	org 1000H
main :	MOV A,#0A5H
acw :	JNB P2.7, cw
	MOV P0,A
	ACALL delay
	RL A
	RL A
	SJMP cw
cw :	JB P2.7, acw
	MOV P0,A
	ACALL delay
	RR A
	RR A
	END

3.3.4 Wave Drive 4 Step Sequence

In addition to the 4 and 8 step sequences there is a sequence called as wave drive 4 step sequence as shown Table 3.3.4. The 8 step sequence is combination of wave drive and 4 step sequence.

Table 3.3.4

Step	Winding A_1	Winding A_2	Winding A_3	Winding A_4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1

Counter clockwise ↑ ↓ Clockwise

Ex. 3.3.5

Write a program to rotate the stepper motor clockwise using the wave drive 4 step sequence. Use the sequence values saved in program ROM locations.

Soln. : We assume that the sequence values are saved in ROM locations starting from 0300 H.

Label	Instruction	Comments
	ORG 0000H	
START :	MOV R0, #04 H	Initialize counter to excitation code sequence
	MOV DPTR, #0300H	
L1 :	CLR A	
	MOVC A, @A+DPTR	
	MOV P1, A	
	ACALL DELAY	
	INC DPTR	
	DJNZ R0, L1	
	SJMP START	
	ORG 0300H	
	DB 8,4,2,1	Code sequence for clockwise rotation
	END	

3.3.5 Identifying Stepper Motor Interface

- There are three types of stepper motor : universal, unipolar and bipolar.
- Depending on the number of motor connections the stepper motor can be identified. Usually a universal stepper motor has 8 connections, a unipolar stepper motor has 6 connections and bipolar stepper motor has 4 connections.
- The universal stepper motor can be used in any mode. The unipolar stepper can be used in unipolar or bipolar mode. The bipolar stepper motor cannot be configured in other modes. It needs H bridge circuitry and high operational current.

3.3.6 Design Problems

Ex. 3.3.6 **Lab Assignment**

Design a 8051 based system to interface stepper motor. Write the corresponding assembly language program to control the stepper motor using keyboard of the PC

- 'c' key to START motor in clockwise direction.
- 't' key STOP the motor.
- 'a' key to START motor in anticlockwise direction.
- 'f' key for increasing the speed (fast).
- 's' key for decreasing the speed (slow).

Soln. :

Part (a) : Design

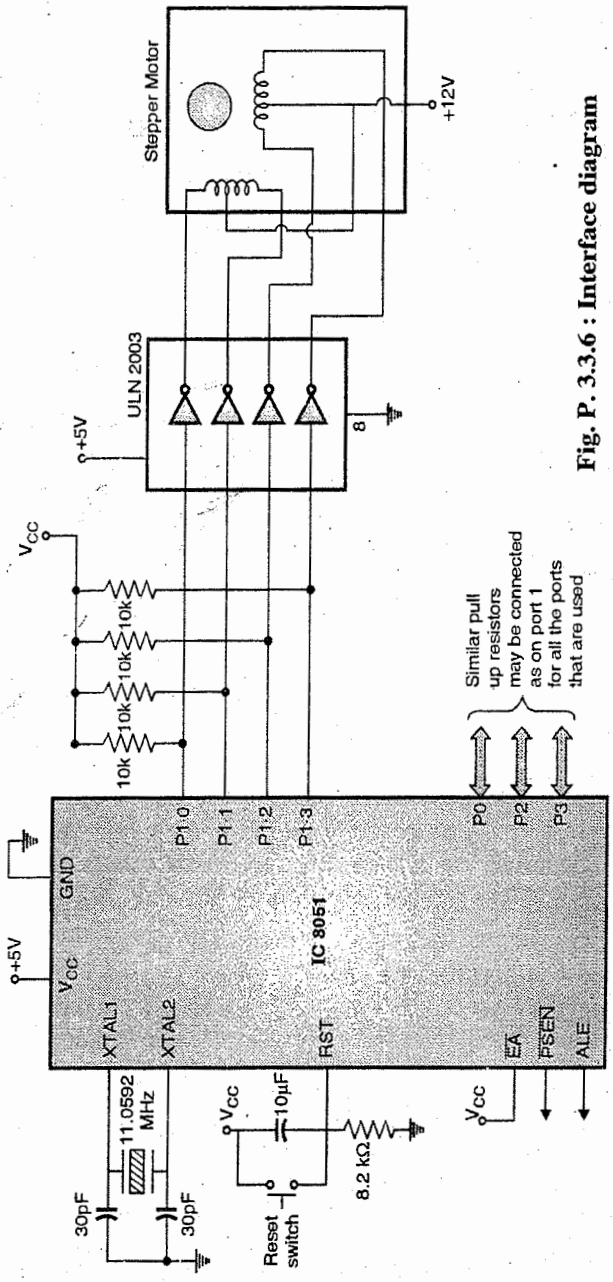


Fig. P. 3.3.6 : Interface diagram

Part (b) Program

We will implement the following state diagram (as shown in Fig. P. 3.3.6) to decide the next state of the system i.e. clockwise rotation, anticlockwise rotation or stop the motor.

Algorithm

A) Main Program

Step I : Initialize serial port in mode 1 and enable reception.

Step II : Initialize TH1 for the required baud rate.

Step III : Enabie serial and global interrupts.

Step IV : Initialize timer 1 in mode 2.

Step V : Switch on the timer 1 run bit.

Step VI : Implementation of state diagram, so initial state is 0.

Step VII : If state is 0, check if key is 1 or 2 and change state accordingly to key pressed and transmitted from PC.

Step VIII : If state is 1 and If stop key is pressed change state to 0. Call subroutine for clockwise motion.

Step IX : If state is 2 and If stop key is pressed change state to 0. Call subroutine for anticlockwise motion.

Step X : Goto to step VII

B) ISR of serial interrupt

Step I : If interrupt is because of TI then clear TI and return

Step II : If interrupt is because of RI then copy the data into a temp register and change the value of a variable (say key) to indicate the next state of the state diagram system. If speed is to be increased or decreased, decrease or increase the delay respectively.

Step III : Clear RI flag

C) Subroutine for Clockwise rotation

Step I : Give the data in the sequence for forward motion.

Step II : Give a delay after every data.

D) Subroutine for Anticlockwise rotation

Step I : Give the data in the sequence for reverse motion.

Step II : Give a delay after every data.

Registers value

1) Interrupt Enable (IE) Register

→ To enable global and serial interrupts.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	1	0	0	0	0

9 0

∴ IE = 0x90

2) Timer Mode (TMOD) Register

→ To initialize Timer 1 in mode 2 as timer

TIMER 1 TIMER 0

D7	D6	D5	D4	D3	D2	D1	D0
GATE	$\overline{C/T}$	M1	M0	GATE	$\overline{C/T}$	M1	M0
0	0	1	0	0	0	0	0

2 0

∴TMOD = 0x20

3) Serial Control (SCON) Register

→ To enable reception and initializing serial communication in mode 1

D7	D6	D5	D4	D3	D2	D1	D0
SM0	SM1	SM2	REN	TB8	RB8	TI	RI
0	1	0	1	0	0	0	0

5 0

∴ SCON= 0x50

4) Timer 1 registers are to be initialized to 0xFD, so that a baud rate of 9600 can be achieved

∴ TH1 = 0xFd

Assembly Program

Label	Instruction	Comment
	ORG 0000H	
	LJMP START	
	ORG 0023H	Serial port ISR
	JNB RI,NXT	If interrupt is because of reception
	CLR RI	Clear RI
	MOV A,SBUF	change the value of key i.e. R1 according to the data received
	CJNE A,#'c',A1	c indicates Clockwise motion, make key=1
	MOV R1,#01H	
	SJMP NXT	
A1:	CJNE A,#'a',A2	a indicates Anticlockwise motion, make key=2

Label	Instruction	Comment
	MOV R1,#02H	
	SJMP NXT	
A2:	CJNE A,#'t',A3	t indicates stop, make key=3
	MOV R1,#03H	
	SJMP NXT	
A3:	CJNE A,#'s',A4	s indicates slow, increase delay in R3 and R4 together
	MOV R2,A	
	MOV A,R3	
	ADD A,#50	
	MOV R3,A	
	MOV A,R2	
	JNC A5	
	INC R4	
A5:	SJMP NXT	
A4:	CJNE A,#'f',NXT	f indicates fast, decrease delay in R3 and R4 together
	MOV R2,A	
	MOV A,R3	
	CLR C	
	SUBB A,#50	
	MOV R3,A	
	MOV A,R2	
	JNC NXT	
	DEC R4	
NXT:	JNB TI,NXT1	If interrupt is because of transmission return & clear TI
	CLR TI	
NXT1:	RETI	
	ORG 0100H	Subroutine for clockwise movement
CLOCKWISE:	MOV R0,#30H	Give the data in the sequence for forward motion with the delay according to the speed
STEP:	MOV P1,@R0	
	MOV 06,R3	
	MOV 07,R4	
AGAIN1:	MOV 04,R7	
AGAIN:	DJNZ R4,AGAIN	
	DJNZ R3,AGAIN1	
	MOV 03,R6	
	MOV 04,R7	
	INC R0	
	CJNE R0,#34H,STEP	
	RET	
	ORG 0200H	

Label	Instruction	Comment
ANTICLOCKWISE:	MOV R0,#40H	Give the data in the sequence for reverse motion with the delay according to the speed
STEP1:	MOV P1,@R0	
	MOV 06,R3	
	MOV 07,R4	
AGAIN3:	MOV 04,R7	
AGAIN2:	DJNZ R4,AGAIN2	
	DJNZ R3,AGAIN3	
	MOV 03,R6	
	MOV 04,R7	
	INC R0	
	CJNE R0,#44H,STEP1	
	RET	
	ORG 1000H	
START:	MOV R3,#100	Initialize counter for delay
	MOV R4,#50	
	MOV R0,#30H	Store the sequence in RAM for clockwise movement
	MOV @R0,#09H	
	INC R0	
	MOV @R0,#0AH	
	INC R0	
	MOV @R0,#06H	
	INC R0	
	MOV @R0,#05H	
	MOV R0,#40H	Store the sequence in RAM for anticlockwise movement
	MOV @R0,#05H	
	INC R0	
	MOV @R0,#06H	
	INC R0	
	MOV @R0,#0AH	
	INC R0	
	MOV @R0,#09H	
	MOV SCON,#50H	Initialize serial port in mode 1 and enable reception
	MOV TH1,#0FDH	Initialize TH1 for baud rate of 9600
	MOV IE,#90H	Enable global and serial interrupt
	MOV TMOD,#20H	Enable timer 1 in mode 2 for baud rate generation
	SETB TR1	Switch on timer 1 Run bit
	MOV R5,#00H	implementation of state diagram, so initial state (Register R5) is 0.

Label	Instruction	Comment
HERE:	CJNE R5,#00H,B1	If state is 0, check if key is 1 or 2 and change state accordingly to key pressed and transmitted from PC.
	CJNE R1,#01H,B3	
	MOV R5,#01	
B3:	CJNE R1,#02H,B1	
	MOV R5,#02	
	SJMP HERE	
B1:	CJNE R5,#01H,B2	
	CJNE R1,#03H,B4	If stop key is pressed goto state 0.
	MOV R5,#00	
	SJMP HERE	
B4:	LCALL CLOCKWISE	Call subroutine for clockwise motion.
	SJMP HERE	
B2:	CJNE R5,#02H,HERE	
	CJNE R1,#03H,B5	If stop key is pressed goto state 0.
	MOV R5,#00	
	SJMP HERE	
B5:	LCALL ANTICLOCKWISE	Call subroutine for anticlockwise motion.
	SJMP HERE	
	END	

Syllabus Topic : Interfacing of Motion Detector

3.4 Interfacing of Motion Detector

- Passive infrared sensors (PIR) sensors are widely used for motion detection for intruder alarm systems.
- The PIR sensor measures the infrared energy radiated by the object that is placed infront of it.
- PIR sensors are made up of pyro-electric material that generates energy when it is exposed to radiation. Gallium Nitride is the commonly used pyroelectric material. The energy is converted to equivalent output voltage.
- The output of PIR sensor will be high when it detects motion and low when there is no motion.

Ex. 3.4.1

Interface 8051 to PIR sensor, switch on LED connected to P2.3 on if motion is detected and switch off LED if no motion is detected.

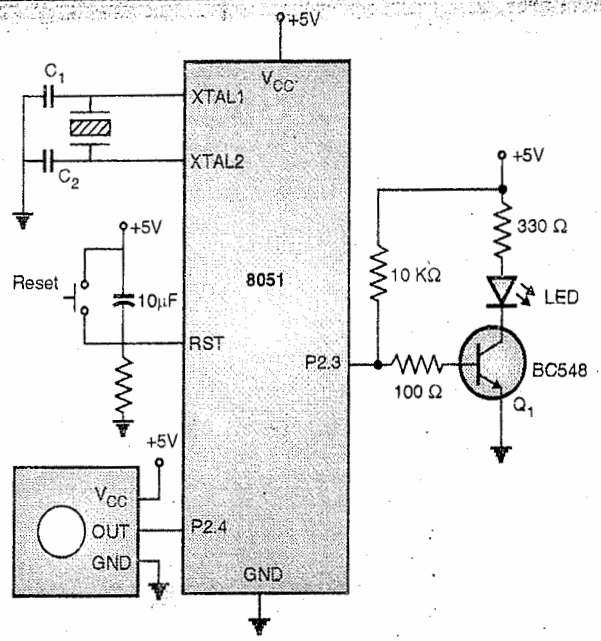


Fig. P. 3.4.1 : Interfacing PIR sensor to 8051 for detecting motion

Fig. P. 3.4.1 shows the interfacing of PIR sensor to 8051. If the output of PIR sensor is high i.e. motion is detected the LED connected to P2.3 will switch ON. If no motion is detected the output of PIR sensor will be low and the LED connected to P2.3 will switch off.

The transistor Q is used for switching LED. The 100 Ω resistor limits the base current of transistor and 330Ω resistor limits the current through LED.

Program :

Label	Instruction	Comments
	PIR EQU P2.4	
	LED EQU P2.3	
	ORG 00H	
	CPL P2.3	
	SETB P2.4	
L1:	JNB PIR, L1	Check if output of PIR is high.
	SETB LED	if yes, switch on LED
L2:	JB PIR, L2	Check if output of PIR is low (no motion detected)
	CLR LED	switch off LED
	SJMP L1	
	END	

Ex. 3.4.2

Design Microcontroller based path follower.

Soln. :

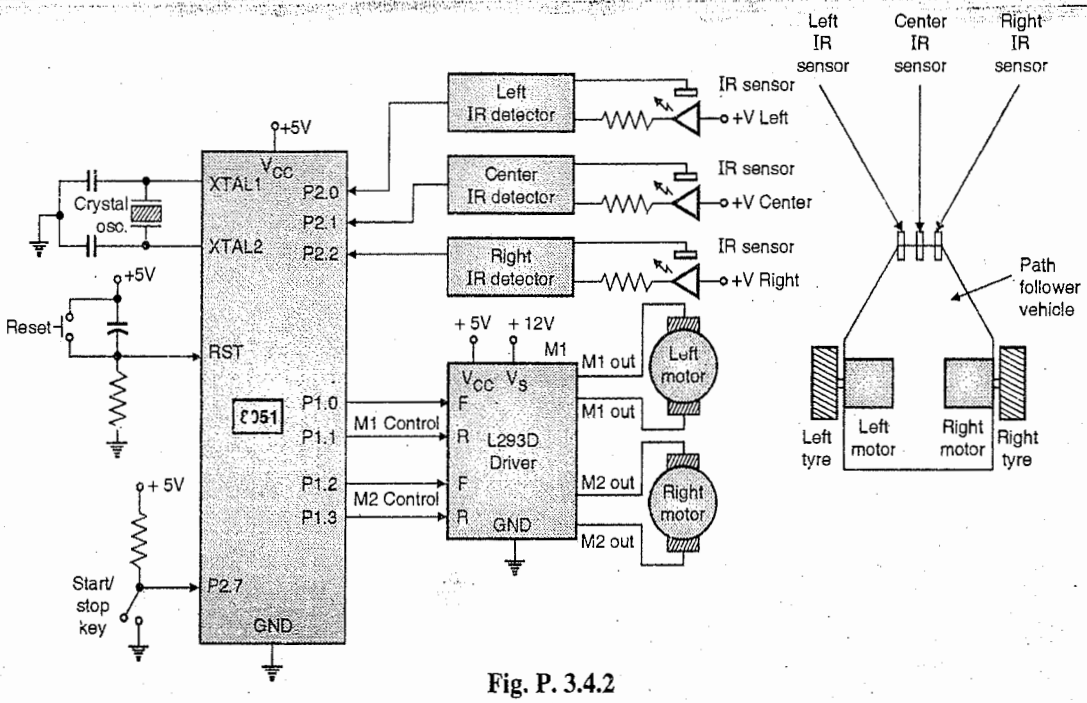


Fig. P. 3.4.2

- Let us use 89C51 Microcontroller, which is a 8051 family Microcontroller with 4 KB internal flash memory. The Microcontroller has 4 i/O Ports.
- Path follower requires set of 3 Infrared Diode and Detector pairs. Detector output is sensed on 3 bits of the Microcontroller, P2.0 – Left IR, P2.1- Centre IR and P2.2 – Right IR, as shown in Fig. P. 3.4.2.
- It would also need a vehicle base with two wheels driven independently by two DC motors, which in turn driven by L293D current drivers. Each motor is controlled by 2 bits (00 – No Motion, 01- Forward Motion 10 – Reverse Motion) for the wheel motion. Say P1.0 and P1.1 for Left Motor and P1.2 and P1.3 for Right Motor.
- An appropriate program is fed into the Microcontroller which senses the input from infrared photo detectors and accordingly controls the DC motors, in order to follow the white line path.
- The Operating logic :
  1. Input IR sensors
  2. If (Left IR = White AND Centre IR = Black) then Left Motor = 00; Right Motor = 01
  3. else if (Right IR = White AND Centre IR = Black) then Left Motor = 01; Right Motor = 00
  4. else if (Right IR = Left IR = White AND Centre IR = White) then Left Motor = Right Motor = 01
  5. else Left Motor = 00; Right Motor = 00
  6. go to step 2
  7. end
- Fig. P. 3.4.2 shows the detailed circuit of the interface.

Syllabus Topic : Interfacing Relays

3.5 Interfacing Relays

- A relay is an electrical switch that opens and closes under the control of another electrical circuit. In the original form, the switch is operated by an electromagnet to open or close one or many sets of contacts.
- It consists of a coil of wire surrounding a soft iron core, an iron yoke, which provides a low reluctance path for magnetic flux, a moveable iron armature, and a set or sets of contacts.
- The armature is hinged to the yoke and mechanically linked to a moving contact or contacts. It is held in place by a spring so that when the relay is de-energized there is an air gap in the magnetic circuit. In this condition, one of the two sets of contacts in the relay is closed, and the other set is open.

- When an electric current is passed through the coil, the resulting magnetic field attracts the armature, and the consequent movement of the movable contact breaks a connection with a fixed contact and makes connection with the other contact.
- Fig. 3.5.1 illustrates the symbol of relay and interfacing of a relay with 8051. Simply making the pin '0' or '1' will switch ON/OFF the relay.

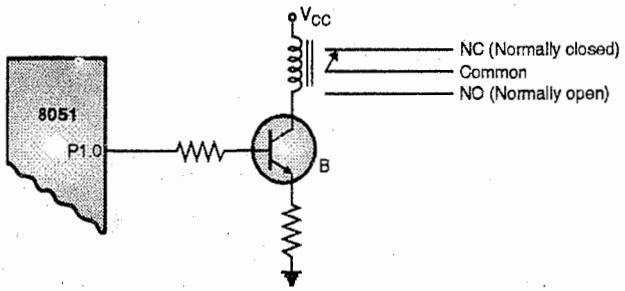


Fig. 3.5.1 : Interfacing relay with 8051 diagram

3.6 Solid State Relay

- A solid state relay (SSR) is a solid state electronic component that provides a similar function to an electromechanical relay but does not have any moving components, increasing long-term reliability.
- With early SSR's, the trade-off came from the fact that every transistor has a small voltage drop across it. This voltage drop limited the amount of current a given SSR could handle. As transistors improved, higher current SSR's, able to handle 100 to 1,200 amps, have become commercially available.
- Compared to electromagnetic relays, they may be falsely triggered by transients.
- The types of SSR are photo-coupled SSR, transformer-coupled SSR, and hybrid SSR.
- A photo-coupled SSR is controlled by a low voltage signal which is isolated optically from the load. The control signal in a photo-coupled SSR typically energizes an LED which activates a photo-sensitive diode.
- The diode turns on a back-to-back thyristor, silicon controlled rectifier, or MOSFET transistor to switch the load.
- Fig. 3.6.1 illustrates the internal structure of a solid state relay and its interfacing with 8051

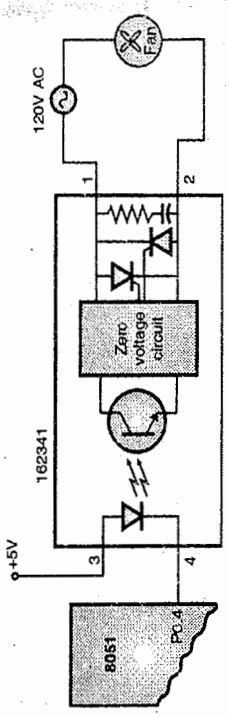


Fig. 3.6.1 : Internal structure of Solid State Relay (SSR) and its interfacing with 8051

Syllabus Topic : Interfacing of Buzzer

3.7 Interfacing of Buzzer

- A buzzer is an electronic device that converts the electronic signal into buzzing noise, that is given to it. A buzzer can be used as an electronic bell or alarm.

Ex. 3.7.1

Let bit P3.1 be an input bit. It represents the condition of a microwave oven. If bit is set, it indicates that the oven is hot. Monitor the bit continuously. Whenever it goes high, send a high-to-low pulse to port P1.2 to turn on a buzzer.

Soln. : Program :

Label	Instruction	Comments
L1:	JNB P3.1, L1	Check if P3.1 is high
	SETB P1.2	If high set bit P1.2 = 1
	CLR P1.2	Send a high to low pulse to P1.2 to turn on buzzer
	SJMP L1	Continue

Syllabus Topic : Opto-isolator

3.8 Opto-isolator

- An opto-isolator (or optical isolator, optocoupler, photocoupler, or photoMOS) is a device that uses a short optical transmission path to transfer a signal between elements of a circuit, typically a transmitter and a receiver, while

keeping them electrically isolated - since the signal goes from an electrical signal to an optical signal back to an electrical signal, electrical contact along the path is broken.

- The solid state relay has an opto-isolator at its input as shown the Fig. 3.8.1.
- The LED at the input triggers the transistor, optically and hence no electrical path exists between the LED and transistor.
- Fig. 3.8.1 illustrates internal structure of an opto-isolator and its interfacing with 8051.

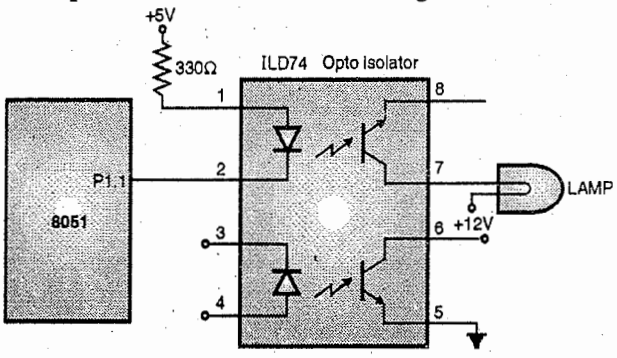


Fig. 3.8.1 : Internal structure of opto-isolator and its interfacing with 8051

Syllabus Topic : Design of Data Acquisition System

3.9 Design of Data Acquisition System

- Computer are the tools that are used for processing of information. However, before the processing begins, we need to collect the data upon which a computer can act. It needs the acquisition of data or the required information from its source.
- This information should be converted into suitable form whereby further analysis and manipulation can be carried out.
- The data that is to be acquired is contained in physical variables like pressure, temperature, force, humidity etc.
- The data is collected then it is converted to the digital form so that it can be transmitted, displayed and stored.
- The data acquisition system is related to the process of acquiring the data in digital form quickly, accurately and economically.
- The **objectives of a data acquisition system** are :

(i) It must acquire essential data at correct speed and at correct time.

(ii) The data acquired must be efficiently used so that the operator knows the state of the system.

(iii) Inorder to maintain an online optimum and safe operation of the system, the system must be continuously monitored.

(iv) The data acquisition system must be able to collect, summarise and store the data for the diagnosis of operation and record purpose.

(v) An effective human system needs to provided by the data acquisition system. It must be able to identify the problem areas, thereby minimizing the unit availability. It maximizes the unit output through a point at minimum cost.

(vi) It must be able to compute the unit performance indices using on-line, real time data.

(vii) It must be reliable.

(viii) It must be flexible and capable for further enhancements.

Fig. 3.9.1 shows the generalized block diagram of a data acquisition system.

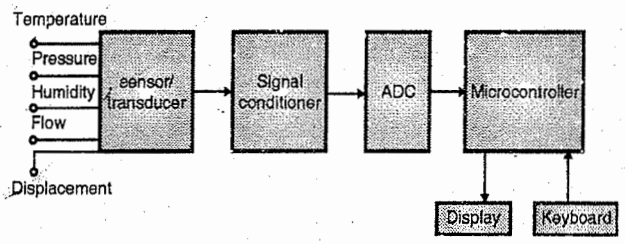


Fig. 3.9.1 : Generalized block diagram of data acquisition system

Sensor / Transducer

They are used to convert the physical parameters into electrical signals. The typical physical parameters include pressure, temperature, humidity, flow, displacement, velocity, acceleration etc.

- The transducers like strain gauge, LVDT, load cells, RTDs, thermocouples are used to convert the physical parameters into an equivalent electrical signal.

- This electrical signal is applied to the signal conditioning circuit.

Signal conditioner

- This block is used to produce conditions so that the circuits works properly.
- Signal conditioning may include :
  - (i) Amplifying the weak signal to a required level.
  - (ii) Modifying the data.
  - (iii) Providing excitation and balancing the circuits.
  - (iv) Filter out the unwanted noise.
- For designing a signal conditioning circuit the person must be more convergent with the OPAMP design. It is a critical stage to design because the input signal is weak and has a lot of noise in it.

ADC (Analog to Digital Converter)

It converts the analog voltage to digital equivalent. It may contain a multiplexer that accepts multiple inputs and sequentially connects them to the converter.

- The digital ADC chips that are widely used in the industry are :
  - (i) ADC 0808 / 0809 (8 bit successive approximation ADC)
  - (ii) ICL 7109 (12 bit resolution, conversion time 33 msec)
  - (iii) AD 574 (12 bit resolution, conversion time 35 μsec max)
  - (iv) LM 331 is used if input frequency is to be measured and it is calibrated in terms of physical parameter.
- The output of the ADC is given to the microcontroller.

Microcontroller : The output of the ADC is given to the microcontroller. The microcontroller will accept the digital output and calibrate the same as per the requirement of the user.

For interface it provides the keyboard and display.

3.9.1 Instruments used for Measuring Temperature

A temperature measuring instruments requires one of the following elements :

(a) A **thermocouple** contains two wires of dissimilar metals. One junction is at the temperature to be measured and the other is a reference temperature (that surrounds the room temperature or ice temperature 0°C). Fig. 3.9.2 shows the interfacing diagram.

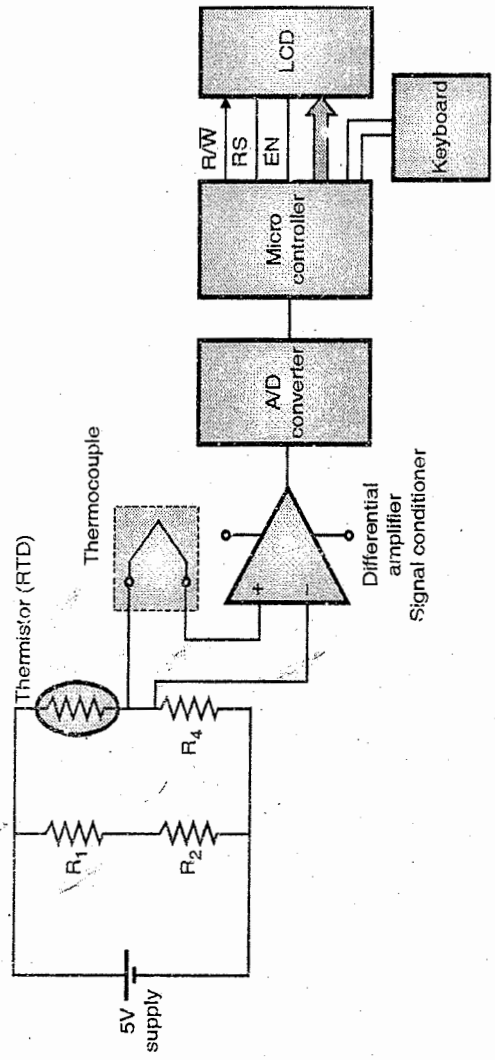


Fig. 3.9.2 : Prototype instrument for thermocouple based temperature measurement using microcontroller

(b) **Resistance temperature detector (RTD)** like platinum resistance temperature detector resistance change about 0.4 Ω/°C for a 100 Ω (at 0°C) resistor unit.

(c) **Negative temperature coefficient (NTC)** or **positive temperature coefficient (PTC)** or **linearlised thermistor.** Fig. 3.9.3 shows a prototype instrument for a PTC or NTC based temperature measurement using the MCU.

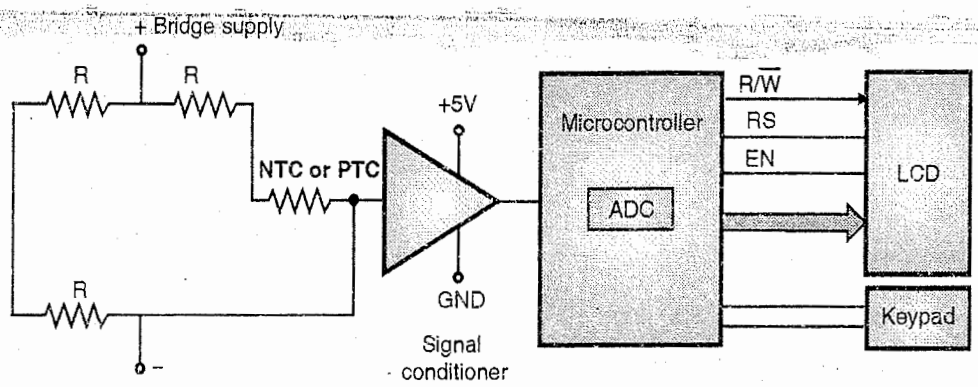


Fig. 3.9.3 : Prototype instrument for a PTC or NTC based temperature measurement using the microcontroller

(d) IC based semiconductor temperature sensor e.g. AD590 from analog devices. Fig. 3.9.4 shows a prototype instrument for measuring temperature using microcontroller.

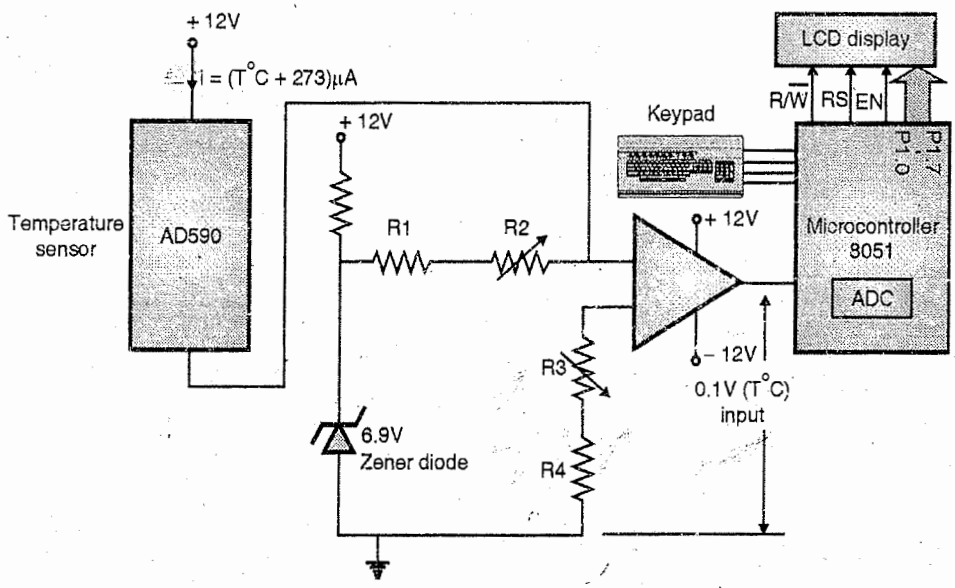


Fig. 3.9.4 : Prototype instrument for IC based sensor temperature measurement using microcontroller

3.9.2 Instruments for Measuring Current using a Current Sensor

- A current passing through a semiconductor in the surrounding of a perpendicular magnetic field generates a voltage that can be measured by an appropriate Hall sensor.
- A Hall sensor measures the voltage across the semiconductor e.g. bismuth. The voltage V developed along the x axis is proportional to the current i along the y axis when the magnetic field H appears along x axis.

3.9.3 Instruments for Measuring Voltage using a Voltage Sensor Thermistor

A voltage sensing thermistor can be used for measuring the DC or AC peak voltage.

3.9.4 Instruments using Resistive Sensors

When the sensor resistance is in one of the arms of Wheatstone bridge then several parameters like load (measured by load cell), strain (force in a perpendicular direction), moisture (as measured by compressed moist sample resistance) and pressure (as measured by bismuth alloy strip / wire resistance) can be measure. The instrument design is similar to that shown in Fig. 3.9.2.

3.9.5 Instruments based on Position Sensor by Proximity Detection

- A metal detector detects eddy currents when the object is in close proximity.

- A capacitive sensor detects the change in the capacitance when an object approaches it.
- A Hall sensor detects the magnetic field when a magnetic object approaches it.
- A optical sensor employs a phototransistor when an object obstructs light from a LED nearby.
- A position sensor for a moving object can be based on the LVDT (Linear Variable Differential Transformer). It can be used to measure the linear position of a moving object or shaft.
- Differential transformer is formed by two primary coils connected in series. They are close to each other along a common axis. They carry AC currents in opposite direction i.e. one carries current in clockwise direction and other carries current in anticlockwise direction.
- The secondary coil (pick up coil) measures no voltage when there is no mechanical object in the surrounding. As soon as an object approaches in proximity along the coil axis, the secondary voltage is sensed.
- The voltage varies along with the position of the object LVDT can also detect the position and shaft movement along the axis in addition to the proximity.
- There are many proximity sensors

  e.g.

  (i) Counting coins.

  (ii) Counting the capped bottles passed in proximity by checking the metallic cap on a bottle in a bottling plant.

  (iii) Position index sensing.

  (iv) Wall or obstacle sensing.

  (v) Broken part detection.

  (vi) Motion detection.

Fig. 3.9.5 shows microcontroller based instrument interfacing circuit for measuring the moisture in multiple type of grains, beans, coffee, milk powder.

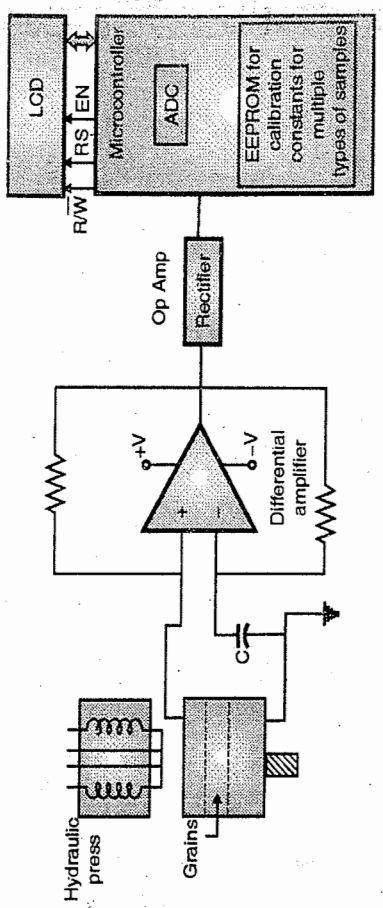


Fig. 3.9.5 : Microcontroller based instrument interface for measuring moisture in multiple type of grains, beans, coffee, milk powder

Fig. 3.9.6 shows microcontroller based instrument for the level of reacting elements in a tank at a cement plant.

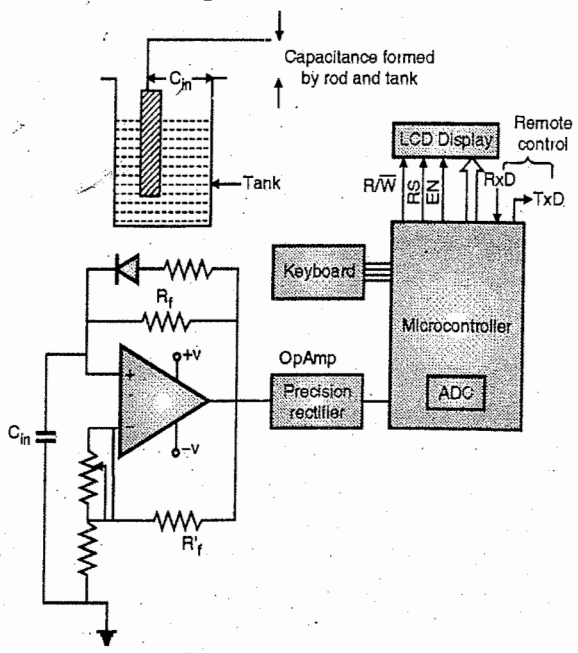


Fig. 3.9.6 : Microcontroller based instrument for the level of reacting elements in a tank at a cement plant

Ex. 3.9.1

Design a Digital Thermometer to display the Temperature in Celsius. The range of temperature is from 0 to 100°C.

Soln. :

- Let us use 89C51 Microcontroller, which is a 8051 family Microcontroller with 4 KB internal flash memory. The Microcontroller has 4 I/O Ports.
- Let us use LM35 Temperature sensor with 10 mA/°C output. Let the signal conditioning Instrumentation Amplifier be used so as to amplify the generated electrical signal to 0-2 V for this range.
- Let us use ADC0800 to convert the analog voltage (measured temperature) to the corresponding digital value. The reference voltage V+ is kept at 5V. Negative reference is kept at 0 V (Ground) This would give us digital value of Temperature to be 00h (for 0°C) to 64h ie 100 decimal (for 100 C). The circuit is fine tuned to achieve this calibration, by adjusting the gain of Instrumentation amplifier accurately.
- The ADC is connected to 89C51 over Ports P1 (ADC Data) and P2.0, P2.1 being control signals SC(Start Conversion), OE (Output Enable) of the ADC, respectively. Status signal EOC (End Of Conversion) is interfaced on P3.2.
- A LCD Display is attached to show the Temperature Output on the Display. The LCD is interfaced to 89C51 over P0 (LCD Data) and P2.2, P2.3, P2.4 being control signals RS(Register Select), R/#W (Read/Write) and EN (Enable) respectively.
- The Operating Logic is :
  1. Initialize Stack
  2. Initialize LCD Display
  3. Start ADC Conversion
  4. If EOC = 0 then wait else proceed
  5. Receive ADC Temperature Data
  6. Convert the Data into it's ASCII Equivalent
  7. Send Data to LCD for Display
  8. Go to step 3 (Continuously loop)
  9. End
- The interface is as shown in Fig. P. 3.9.1.

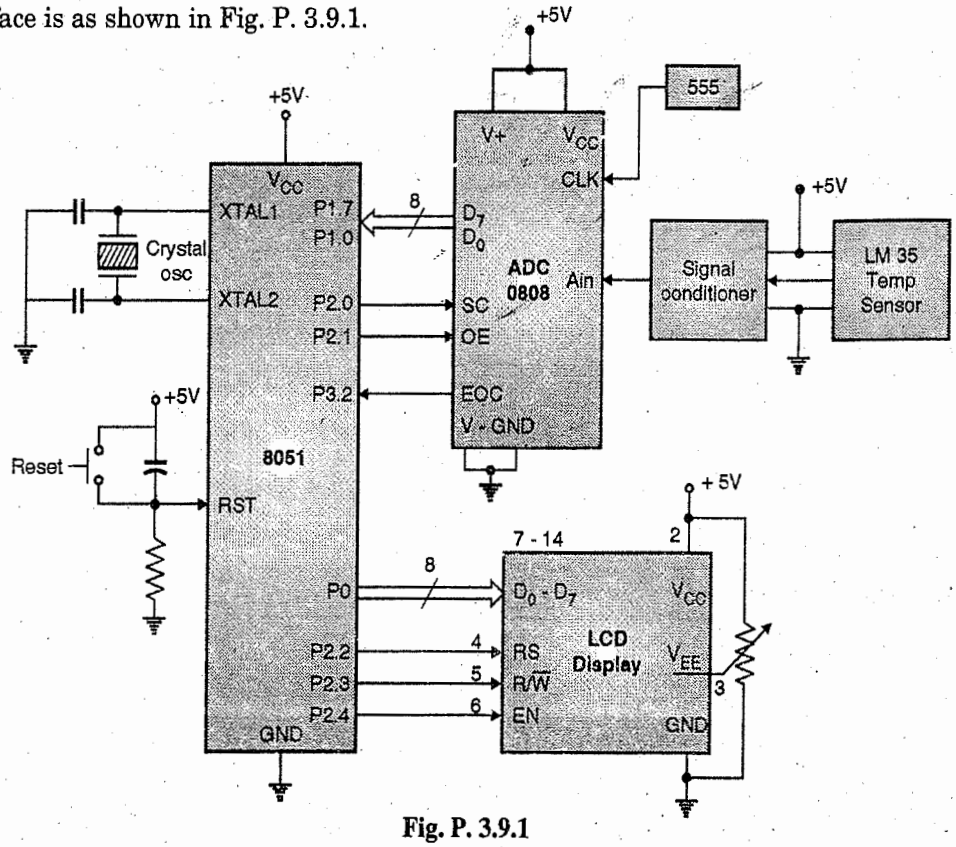


Fig. P. 3.9.1

3.9.6 Load Cell

- Load cell is used to measure mechanical force. The strain gauges are generally used for force measurement.
- The resistance of a strain gauge increases if it is stretched.
- The relation between the elongation and resistance change is given by the gauge factor :

$$G = \frac{\Delta R / R}{\Delta L / L}$$

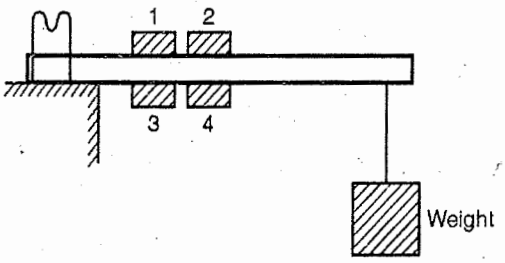
R : Original resistance of the strain gauge

L : Original length

ΔR : Change in resistance of strain gauge

ΔL : Small change in length, due to application of stress

- The gauge factor is a dimensionless quantity.
- Semiconductor strain gauges are more sensitive with gauge factors around 50 against the typical gauge factor of 2 for bonded strain gauges.
- The strain gauges are cemented over mechanical structure whose deformation under the influence of stress is to be measured.
- Fig. 3.9.7 shows a cantilever beam with four strain gauges.



1,2,3,4 Locations of 4 strain gauges

Fig. 3.9.7 : Force measurement using strain gauge

- The strain gauges 1 and 2 are mounted such that after applying load those will be under tension.
- The strain gauges 3 and 4 are under compression and loaded conditions.
- The strain gauges are used in a full bridge to give bridge output proportional to the force applied.
- To maximum the sensitivity of the bridge the strain gauges are connected as shown in Fig. 3.9.8.
- Under the loaded conditions, the resistance of strain gauges 1 and 2 increases and that of 3 and 4 decreases. Hence, a potential at point X of bridge is elevated as much compared to Y.
- If the gauge factor of strain gauge the Young's modulus of cantilever material and spring constant are known then we can find out the change in resistance and hence the bridge output for a given load.
- The bridge output is double ended. It can be converted using a difference amplifier or can be given to an ADC having differential input.
- Fig. 3.9.8 shows connections using MAX111 serial ADC. It is a 14 bit serial ADC that accepts differential inputs. It supports the serial SPI bus.
- The output of bridge is in mV. The amplifiers provide essential gain to obtain a full scale differential output of 2 V.
- A single + 5 V supply is sufficient.
- The ADC reference voltages REF+ and REF– are obtained from the + 5 V supply. The supply is also used to excite the strain gauge bridge.

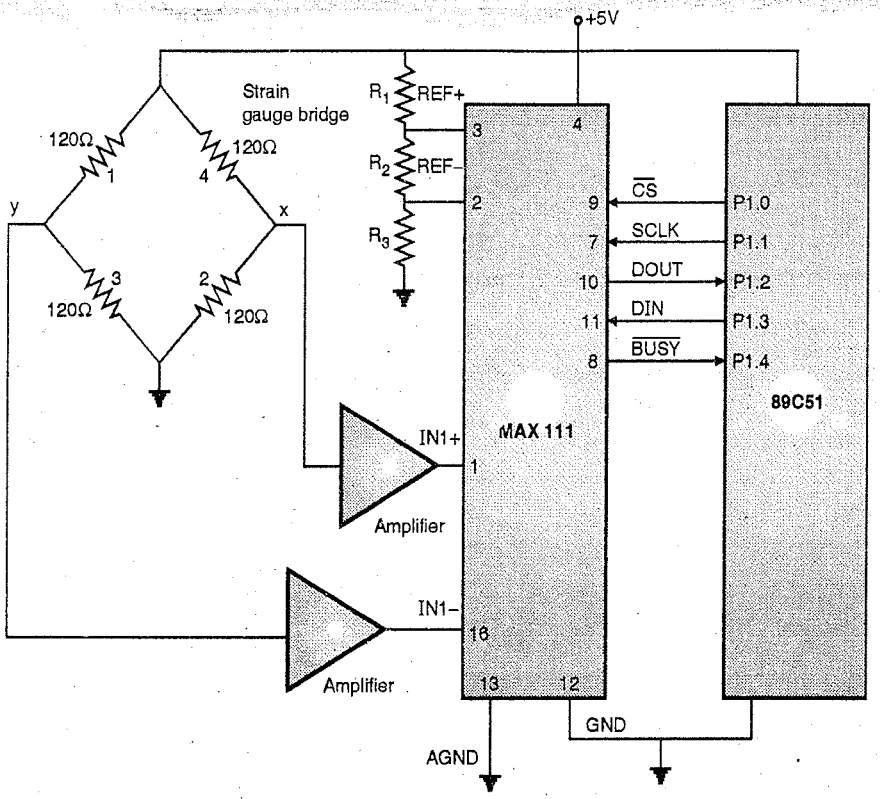


Fig. 3.9.8 : Strain gauge load cell interface with 89C51

Ex. 3.9.2

Design a system to calculate and display the weight using load cell using 89C51/PIC microcontroller along with suitable signal conditioning circuit. Display the weight on LCD interfaced to the microcontroller. Draw the complete block diagram and flow chart. Also write the algorithm and program for the system.

Soln. :

Algorithm

(A) Main Program

Step I : Initialize Port 1 as input port

Step II : Initialize pin used for End of conversion (EOC) as input

Step III : Clear the OE (used for read), start (that indicates start of conversion to ADC) and ALE (used to latch the address)

Step IV : Issue the address of channel 0 on the CBA pins

Step V : Issue pulse on ALE to latch the address.

Step VI : Initialize the LCD.

Step VII : Wait for some software delay after power up for display to stabilize.

Step VIII : Initialize the LCD by giving the instruction 0x38 to the command subroutine.

Step IX : Wait for small software delay.

Step X : Issue the command 0x0E to command subroutine for display on, cursor on.

Step XI : Wait for small software delay.

Step XII : Issue the command 0x01 for clearing display to command subroutine.

Step XIII : Wait for small software delay.

Step XIV : Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.

Step XV : Wait for small software delay.

Step XVI : Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.

Step XVII : Issue start of conversion

Step XVIII : Wait for EOC to become 0 and then wait for it to become 1. This indicates the ADC has completed the conversion.

Step XIX : Calculate the digits and display it on LCD

Step XX : Also display the unit of weight.

(B) Command subroutine

Step I : Give the instruction to the port connected to data bus of the LCD.

Step II : Make RS = '0', to indicate instruction.

Step III : Make R/$\overline{W}$ = '0', to indicate write.

Step IV : Make E = '1'
Step V : Wait for 120 μsec.
Step VI : Make E = '0'
} To give a high-to-low pulse on E pin so as to latch the command

Step VII : Return.

(C) Data subroutine

Step I : Check if LCD is ready by calling ready subroutine.

Step II : Give the data to the port connected to the data bus of the LCD.

Step III : Make RS = '1', to indicate data

Step IV : Make R/$\overline{W}$ = '0', to indicate write

Step V : Make E = '1'
Step VI : Wait for 120 μsec.
} To give a high-to-low pulse on E pin so as to latch the data

Step VII : Make E = '0'

Step VIII : Return

(D) Ready subroutine

Step I : Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.

Step II : Make RS = '0' to indicate instruction.

Step III : Make R/$\overline{W}$ = '1', to indicate read.

Step IV : Make E = 0

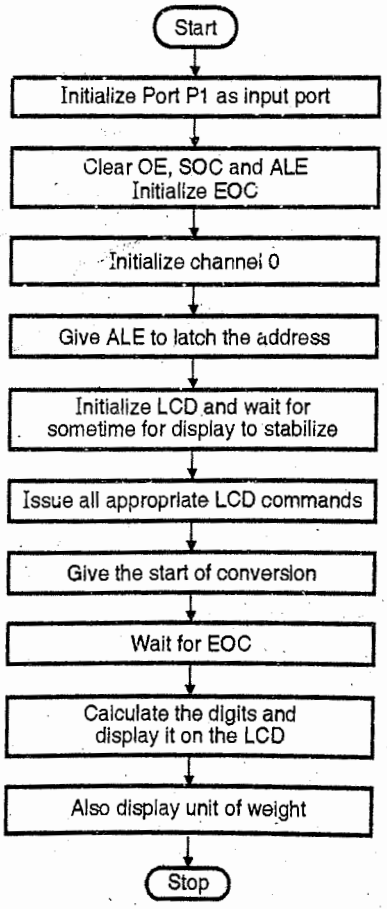
Step V : Make E = 1

Step VI : Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.

Step VII : Return.

Flowchart :

Refer Flowchart P. 3.9.2



Flowchart P. 3.9.2

Interfacing diagram

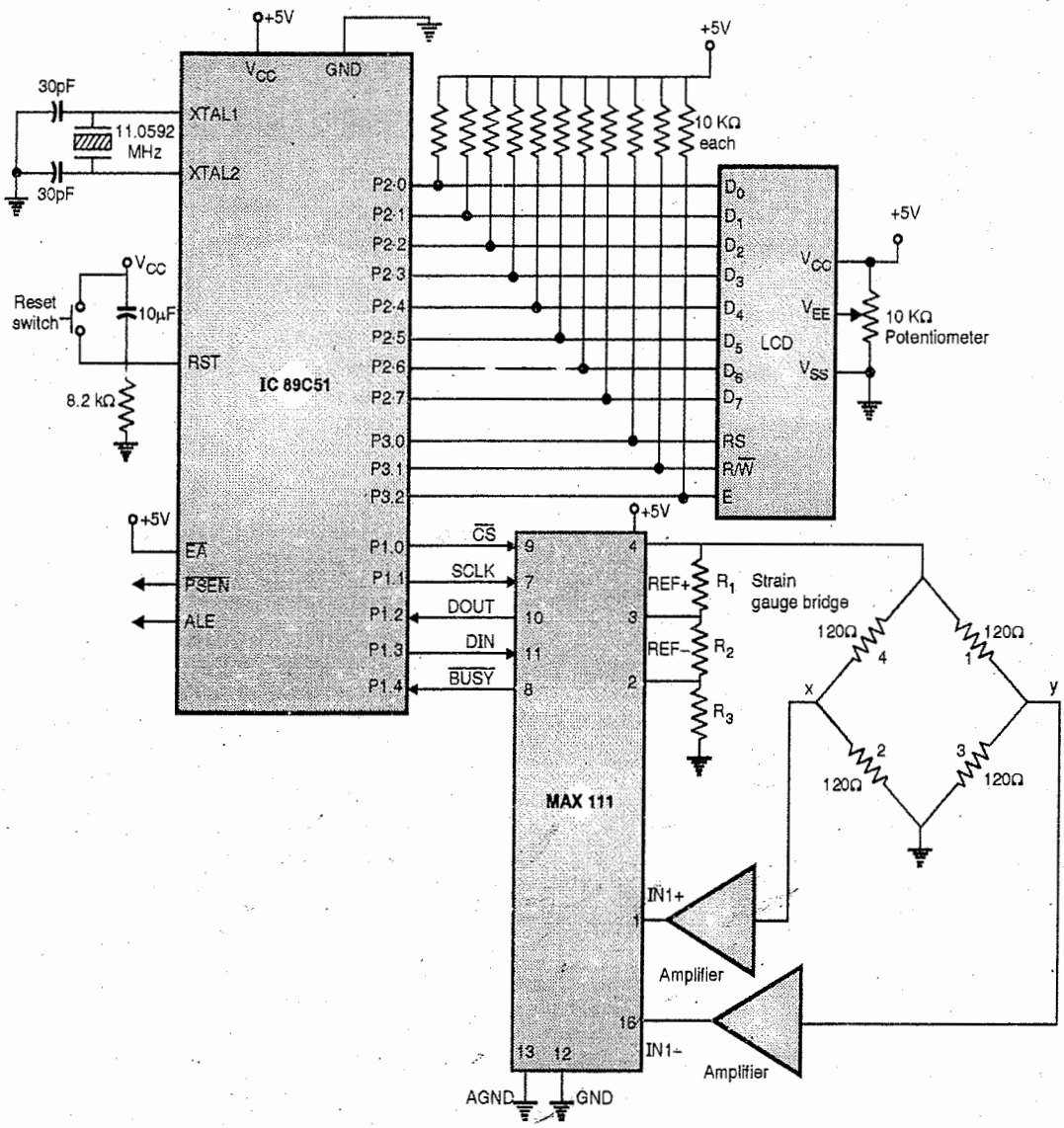


Fig. P. 3.9.2 : Interfacing diagram

Assembly Program

Label	Instruction	Comments
	ORG 0000H	
	LJMP Start	
	ORG 0100H	Function to write the command to LCD
COMMAND:		
	MOV P2, R3	Write the command on the Port 2 so as to issue it to LCD.
	CLR P3.0	RS=0, Indicates instruction.
	CLR P3.1	RW=0, Indicates Write.
	SETB P3.2	A high to low pulse on en pin to latch the command
	LCALL DELAY	

Label	Instruction	Comments
	CLR P3.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0200H	
DISPLAY:		Function to write data to LCD
	LCALL READY	Check if the LCD is ready by calling the ready function.
	MOV P2, R3	Write the data on the Port 2 so that it is given to the LCD.
	SETB P3.0	RS=1, Indicates data.
	CLR P3.1	RW=0, Indicates Write.
	SETB P3.2	A high to low pulse on en pin to latch the command

Label	Instruction	Comments
	LCALL DELAY	
	CLR P3.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0300H	Function to check if LCD is busy or ready.
READY:	SETB P2.7	Making the P2.7 pin as input pin by writing a'1' on t.
	CLR P3.0	RS=0, Indicates instruction and not data
	SETB P3.1	RW=1, Indicates read and not write
WAIT:	CLR P3.2	A low to high going pulse on en pin.
	LCALL DELAY	
	SETB P3.2	
	JB P2.7, WAIT	Wait till the LCD is busy.
	RET	
	ORG 0400H	
DELAY:		A subroutine to implement small software delay
	MOV R5, #1CH	
REP:	DJNZ R5,REP	
	RET	
	ORG 1000H	
Start:	MOV R4, #0FFH	Wait for some time for LCD to stabilize when power-on.
AGAIN1:	MOV R5, #0FFH	
AGAIN:	DJNZ R5,AGAIN	
	DJNZ R4,AGAIN1	
	MOV R3, #38H	Issue the command to initialize 16×2 LCD.
	LCALL COMMAND	
	MOV R3, #0FH	Issue the command for display on, cursor on and cursor blinking.
	LCALL COMMAND	
	MOV R3, #01H	Issue the command to clear display.
	LCALL COMMAND	
	MOV R3, #06H	Issue the command to increment cursor position on every character written.
	LCALL COMMAND	Issue the command to position the cursor on the first position on line 1.
	MOV R3, #80H	
	LCALL	

Label	Instruction	Comments
	COMMAND	
	CS BIT P3.0	
	SCLK BIT P3.1	
	DIN BIT P3.3	
	DOUT BIT P3.2	
HERE :	MOV A, #9E	channel 1 selection
	MOV R3, #08H	Load count.
	CLR CS	
	CLR C	
I1 :	RLC A	
	MOV DIN, C	
	CLR SCLK	
	ACALL DELAY	
	SETB SCLK	Latch data
	ACALL DELAY	delay
	DJNZ R3, 11	repeat for all 8 bits.
	SETB CS	deselect ADC and starts conversion.
	CLR SCLK	SCLK = 0 during conversion.
	MOV B, #100	Calculate and display the count
	DIV AB	
	MOV R3, A	
	LCALL DISPLAY	
	MOV A,B	
	MOV B, #10	
	DIV AB	
	MOV R3, A	
	LCALL DISPLAY	
	MOV R3, B	
	LCALL DISPLAY	
	MOV R3, #'g'	The count is in gms
	LCALL DISPLAY	
	MOV R3, #'m'	
	LCALL DISPLAY	
	MOV R3, # 's'	
	LCALL DISPLAY	
	MOV R3, #0×80	
	LCALL COMMAND	
	SJMP HERE	
	END	

Ex. 3.9.3 Lab Assignment

Design a system to measure a speed of DC Motor. Display the speed on Seven-Segment Display.

Soln. :

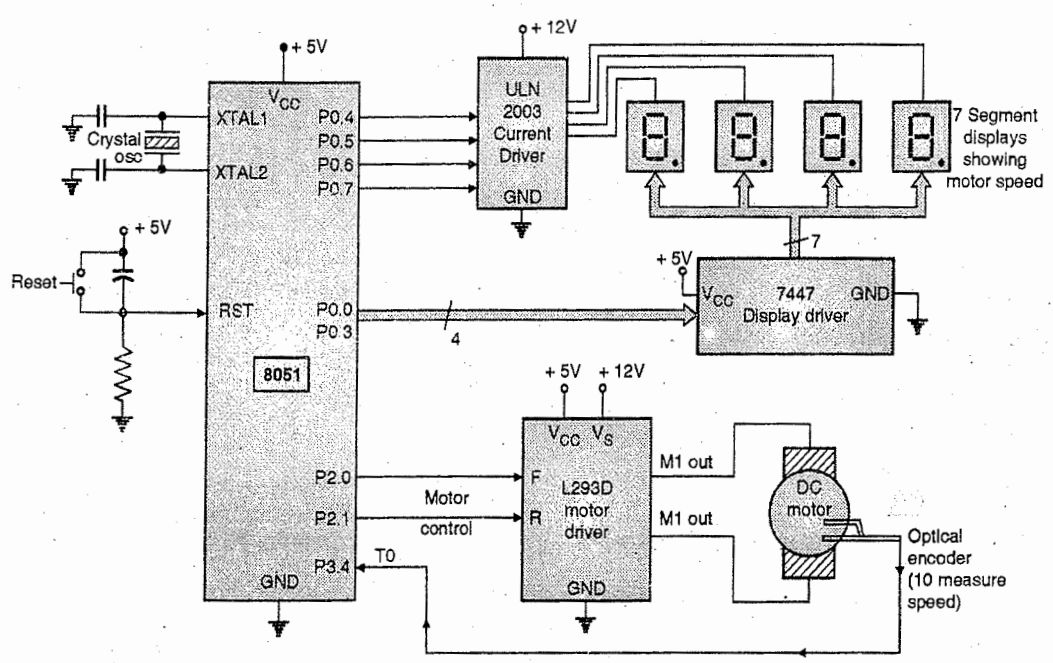


Fig. P. 3.9.3

- Let us use 89C51 Microcontroller, which is a 8051 family Microcontroller with 4 KB internal flash memory. The Microcontroller has 4 I/O Ports.
- Let us connect the DC Motor, using appropriate current drivers L293D. Let the Motor be controlled by 2 bits supplied to L293D using P2.0 and P2.1
- Let us use an Infrared Photo Diode and Photo Detector pair connected across the Fans of DC Motor. A Rotary Shaft encoder is used to calculate the pulses. As the motor rotates 360°, shaft encoder cuts once across the Photo Detector, thus generating a Pulse on the detector output. This pulse is fed to 89C51 Microcontroller over timer input pin P3.4 (T0) so that the pulse can be counted per unit time, using the Counter mode of Timer0.
- Let us connect 4 Digit, 7-Segment Display to 89C51. The Data lines of LCD Display are driven through IC 7447 (BCD to 7-Segment Driver) using P0.0 to P0.3 lines and Display is run in the scanned mode using common Anode mode. The driving of each display digit is done on lines P0.4 to P0.7.
- The Operating Logic is :
  1. Initialize Stack
  2. Initialize Timer
  3. Count Pulses from Photo Detector for 1 Sec.
  4. Multiply by 60 (3Ch) – To get Pulses per min i.e. Speed in RPM.
  5. Convert into Decimal
  6. Display the RPM value on the 7-Segment Display
  7. Scan Display and show digits one by one
  8. Reset the timer
  9. Go to step 3 (Continuous Loop showing Motor speed in RPM)
  10. End
- The interface is as shown in the Fig. P. 3.9.3.

Syllabus Topic : Design of Frequency Counter

3.10 Design of Frequency Counter

Design 3.10.1

Design a 8051 based counter to count the number of pulses on the timer pin in one second, and hence get the frequency. Display the count on LCD. Write the corresponding assembly program.

Soln. : Part (a) : Design

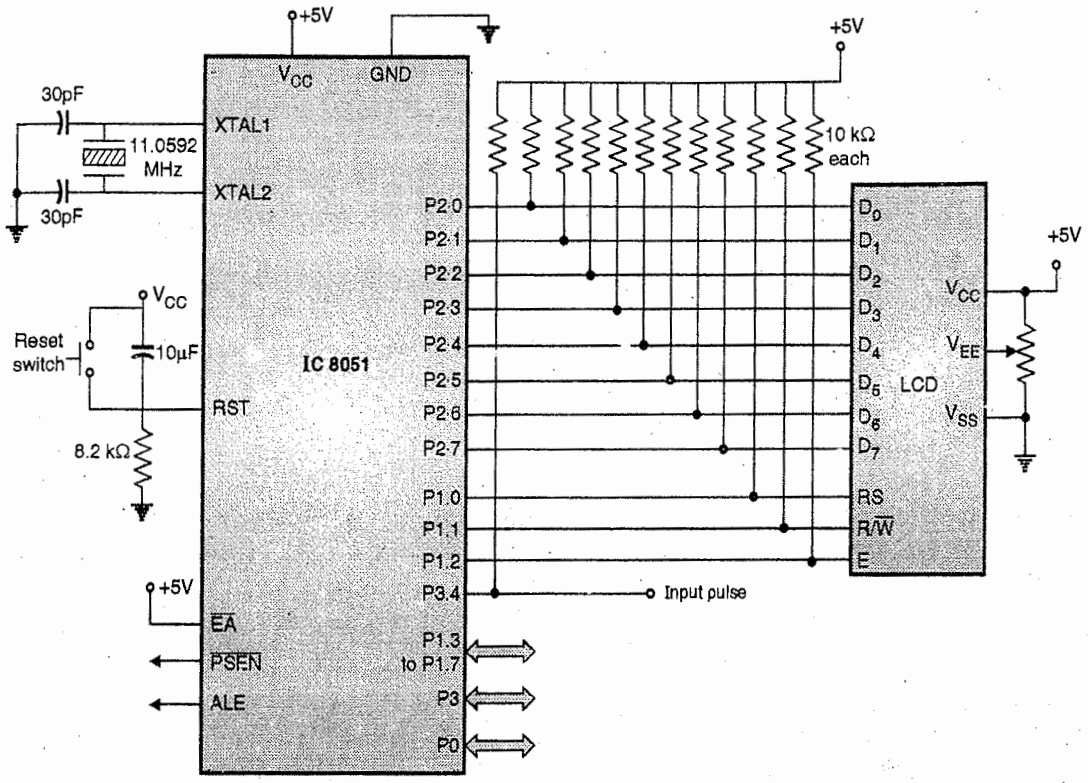


Fig. P. 3.10.1 : Interface diagram

Part (b) : Program

Algorithm

(A) Main Program

Step I : Initialize the LCD.

Step II : Wait for some software delay after power up for display to stabilize.

Step III : Initialize the LCD by giving the instruction 0x38 to the command subroutine.

Step IV : Wait for small software delay.

Step V : Issue the command 0x0E to command subroutine for display on, cursor on.

Step VI : Wait for small software delay.

Step VII : Issue the command 0x01 for clearing display to command subroutine.

Step VIII : Wait for small software delay.

Step IX : Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.

Step X : Wait for small software delay.

Step XI : Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.

Step XII : Initialize a port 2 as input port for keys by writing 0xF0 on it.

Step XIII : Initialise the output port for keyboard to all 0's

Step XIV : Initialise TMOD to 0x26 i.e. timer 0 in mode 2 as timer, while timer 1 has timer mode 2.

Step XV : Initialize global and timer 0 and timer 1 interrupt

Step XVI : Initialize the TL0 and TH0 registers to maximum count so that when a pulse occurs on the T) pin it overflows and generates a interrupt and Initialize TL1 and TH1 to generate a delay of 217µsec. This delay will be executed 4608 times to get a delay of 1 sec.

Step XVII : Initialize the LCD to 1st line 1st character. Calculate the thousands digit and display it

Step XVIII : Calculate the hundreds digit and display it

Step XIX : Calculate the tens digit and display it

Step XX : Calculate the units digit and display it. Also display the characters H and z for unit of frequency i.e. Hertz

Step XXI : Repeat the steps XVII onwards to continuously display the refreshed count

(B) Command subroutine

Step I : Give the instruction to the port connected to data bus of the LCD.

Step II : Make RS = '0', to indicate instruction.

Step III : Make R/$\overline{W}$ = '0', to indicate write.

Step IV : Make E = '1'

Step V : Wait for 120 μsec.

(To give a high-to-low pulse on E pin so as to latch the command)

Step VI : Make E = '0'

Step VII : Return.

(C) Data subroutine

Step I : Check if LCD is ready by calling ready subroutine.

Step II : Give the data to the port connected to the data bus of the LCD.

Step III : Make RS = '1', to indicate data

Step IV : Make R/$\overline{W}$ = '0', to indicate write

Step V : Make E = '1'

Step VI : Wait for 120 μsec.

(To give a high-to-low pulse on E pin so as to latch the data)

Step VII : Make E = '0'

Step VIII : Return

(D) Ready subroutine

Step I : Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of 8051 as input port.

Step II : Make RS = '0' to indicate instruction.

Step III : Make R/$\overline{W}$ = '1', to indicate read.

Step IV : Make E = 0

Step V : Make E = 1

Step VI : Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.

Step VII : Return.

(E) ISR of timer 0 interrupt

Step I : Increment the count until it is 9999 after every pulse on T0 pin. Once the count reaches 9999 stop incrementing it.

(F) ISR of timer 1 interrupt

Step I : Decrement the count until it is 0000 after every 217 μsec. When 217μsec delay is executed 4608 times, it completes a delay of 1 sec.

Step II : Once the entire delay of 1 sec is over disable interrupts so as to stop counting the pulses on T0 pin.

Registers value

(1) Interrupt Enable (IE) Register

To enable global, timer 1 and timer 0 interrupts.

D7	D6	D5	D4	D3	D2	D1	D0
EA	-	-	ES	ET1	EX1	ET0	EX0
1	0	0	0	1	0	1	0
8				A			

∴ IE = 0x8A

(2) Timer Mode (TMOD) Register

To initialize Timer 0 as counter in mode 2

To initialize Timer 1 as timer in mode 2

TIMER 1				TIMER 0			
D7	D6	D5	D4	D3	D2	D1	D0
GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0
0	0	1	0	0	1	1	0
2				6			

∴TMOD = 0x26

(3) Timer 0 will be to count number of pulses on T0 pin. Hence it is initialized to 0xFF, so that whenever it overflows will cause an interrupt and hence increment the count. Since it is in mode 2, also the count 0xFF will be reloaded into it.

$\therefore$ TL0 = 0xFF and TH0 = 0xFF

Timer 1 will be used to generate a delay of 1 sec, hence it will be initialized to 55, so that it generates a delay of 217μsec and for 4608 times will result in 1 sec.

$\therefore$ TL1 = 55 and TH1 = 55

Assembly Program

Label	Instruction	Comment
	ORG 0000H	
	LJMP Start	
	ORG 000BH	ISR for timer 0
	LJMP PULSE	
	ORG 0700H	
PULSE:	MOV A,DPL	
	ANL A,DPH	
	CJNE A,#99H,NXT	If count is reached to maximum i.e. 9999 then stop incrementing DPTR
	SJMP NXT	else increment the counter DPTR in decimal form using DA A
CONTINUE:	MOV A,DPL	
	ADD A,#01H	
	DA A	
	MOV DPL,A	
	MOV A,DPH	
	ADDC A,#00H	
	DA A	
	MOV DPH,A	
NXT:	RETI	
	ORG 001BH	ISR for Timer 1
	DJNZ R7,NOTDONE	If 1 sec delay not over then return
	MOV R7,#144	R7 and R6 together are used to generate the delay
	DJNZ R6,NOTDONE	of 217μsec for 4608 times for a delay of 1 sec.
	CLR EA	If one second delay is over then disable all interrupts.
NOTDONE:	RETI	
	ORG 0100H	Function to write the command to LCD
COMMAND:	MOV P2,R3	Write the command on the Port 2 so as to issue it to LCD.
	CLR P1.0	RS=0, Indicates instruction.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on EN pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0200H	
DISPLAY:		Function to write data to LCD
	LCALL READY	Check if the LCD is ready by calling the ready function.
	MOV P2,R3	Write the data on the Port 2 so that it is given to the LCD.
	SETB P1.0	RS=1, Indicates data.
	CLR P1.1	RW=0, Indicates Write.
	SETB P1.2	A high to low pulse on EN pin to latch the command
	LCALL DELAY	
	CLR P1.2	
	LCALL DELAY	Wait for some time, software delay
	RET	
	ORG 0300H	Function to check if LCD is busy or ready.
READY:	SETB P2.7	Making the P2.7 pin as input pin by writing a '1' on it.
	CLR P1.0	RS=0, Indicates instruction and not data
	SETB P1.1	RW=1, Indicates read and not write
WAIT:	CLR P1.2	A low to high going pulse on EN pin.
	LCALL DELAY	
	SETB P1.2	
	JB P2.7,WAIT	Wait till the LCD is busy.
	RET	
	ORG 0400H	
DELAY:		A subroutine to implement small software delay
	MOV R5,#1CH	
REP:	DJNZ R5,REP	
	RET	
	ORG 1000H	

Label	Instruction	Comment
Start:	MOV R6,#144	Count for 4608 times delay of 217μ sec using R6 and R7 i.e. 144 x 32
	MOV R7,#32	
	MOV DPTR,#0000	Initialize number of pulses on Timer pin as 0
	SETB P3.4	Make T0 pin as input pin
	MOV R4,#0FFH	Wait for some time for LCD to stabilize when power-on.
AGAIN1:	MOV R5,#0FFH	
AGAIN:	DJNZ R5,AGAIN	
	DJNZ R4,AGAIN1	
	MOV R3,#38H	Issue the command to initialize 16 × 2 LCD.
	LCALL COMMAND	
	MOV R3,#0FH	Issue the command for display on, cursor on and cursor blinking.
	LCALL COMMAND	
	MOV R3,#01H	Issue the command to clear display.
	LCALL COMMAND	
	MOV R3,#06H	Issue the command to increment cursor position on every character written.
	LCALL COMMAND	
	MOV R3,#80H	Issue the command to position the cursor on the first position on line 1.
	LCALL COMMAND	
	MOV IE,#8AH	Enable global, timer 1 and timer 0 interrupts
	MOV TMOD,#26H	Timer 0 is programmed in mode 2 as Counter and timer 1 as timer mode 2
	MOV TL0,#FFH	Initialize count to maximum count in TH0 and TL0 so that, when a pulse occurs on the T0 pin it overflows and generates an interrupt.
	MOV TH0,#FFH	
	MOV TL1,#55	Initialize count in TH0 and TL0 so as to give a delay of 217μ sec.
	MOV TH1,#55	
	SETB TR0	Set timer 0 in run mode
	SETB TR1	Set timer 1 in run mode
HERE:	MOV R3,#80H	Initialize the LCD to 1st line 1st character
	LCALL COMMAND	
	MOV A,DPH	Separate thousands, convert it to ASCII digit and display it on LCD
	ANL A,#0F0H	
	MOV 0F0H,#10H	
	DIV AB	
	ADD A,#30H	
	MOV R3,A	
	LCALL DISPLAY	
	MOV A,DPH	Separate hundreds digit, convert it to ASCII and display it on LCD
	ANL A,#0FH	
	ADD A,#30H	
	MOV R3,A	
	LCALL DISPLAY	
	MOV A,DPL	Separate tens digit, convert it to ASCII and display it on LCD
	ANL A,#0F0H	
	MOV 0F0H,#10H	
	DIV AB	
	ADD A,#30H	
	MOV R3,A	
	LCALL DISPLAY	
	MOV A,DPL	Separate units digit, convert it to ASCII and display it on LCD
	ANL A,#0FH	
	ADD A,#30H	
	MOV R3,A	
	LCALL DISPLAY	
	MOV R3,#'H'	Display character H
	LCALL DISPLAY	
	MOV R3,#'z'	Display character z
	LCALL DISPLAY	
	SJMP HERE	
	END	

❑❑❑

UNIT - IV

CHAPTER 4

PIC Microcontroller Architecture

4.1 Introduction

- PIC microcontrollers are developed by Microchip Technology Inc.
- **PIC 18FXX** belongs to a class of 8-bit microcontrollers of RISC (Reduced Instruction Set Computing) architecture.
- **8051** supports CISC (Complex Instruction Set Computer) architecture.
- **RISC (Reduced Instruction Set Computer),** as the name says has less number of instructions.
- The CISC processors have complex instructions while RISC have simple instructions.
- Complex instructions are combination of multiple simple instructions.
- Simple instructions are those that perform only one operation i.e. either memory access or ALU operation etc. e.g. MOVLW 0x30 is a simple instruction.
- Complex instructions are those that perform multiple operations i.e. one instruction accesses memory as well as performs ALU operation.
- Since in RISC the number of instructions are lesser, it has lesser addressing modes, simpler instructions etc, its control unit can be implemented using a Hardwired control unit, that makes the decoding faster. Whereas CISC requires a microprogrammed control unit.

4.1.1 Harvard Architecture of PIC Microcontroller

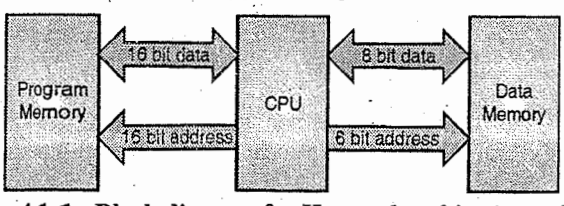


Fig. 4.1.1 : Block diagram for Harvard architecture of PIC microcontrollers

- Like 8051, PIC microcontrollers have Harvard organization of memory. But here it is a special case wherein the number of data lines for data memory and program memory are different. Also the number of address lines for program and data are different.
- Harvard architecture is a newer concept than von-Neumann's. It rose out of the need to speed up the work of a microcontroller. In Harvard architecture, data bus and address bus are separate. Thus a faster flow of data is possible through the central processing unit, and of course, a greater speed of work. Separating a program from data memory makes it further possible for instructions not to have to be 8-bit words. Fig. 4.1.1 shows block diagram for Harvard architecture in PIC controllers.
- PIC uses 16 bits for instructions which allows for all instructions to be one/two word instructions.
- Instructions are fetched from program memory using buses that are different from the ones that are used to access the data memory.
- 16-bit wide data bus for program memory fetches an entire instruction in a single access.

Syllabus Topic : Features of PIC Family of Microcontrollers

4.1.2 Features of PIC Family of Microcontrollers

SPPU - Aug. 14

University Question

Q. What are the features of PIC microcontroller ? **(Aug. 2014(In. sem.), 4 Marks)**

The features of PIC microcontroller are :

1. They use Harvard architecture and are high performance RISC processors.
2. The register files/data memory can be addressed directly and indirectly. All the SFRs including the PC are mapped in data memory.
3. It consists of an instruction set with 35 instructions. Most of the instructions are completed in a single cycle.
4. The PIC microcontroller has a built in power-on-reset.
5. There are three timers. They are used to characterize the inputs, control outputs and provide internal timing for program execution.
6. It can control upto 12 independent interrupt sources.
7. It also supports analog to digital conversion function.

8. There is a built in serial peripheral interface.
9. The PIC microcontroller has a brownout reset. Whenever the supply voltage drops below βV_{DD} brownout feature causes reset of the microcontroller.
10. For clock generation an RC circuit, a quartz crystal or a ceramic resonator can be used. The oscillator clock can be stopped at any instant and can be restored back.
11. It allows serial programming.
12. It consumes low power.

4.1.2.1 RISC Architecture in the PIC Microcontrollers

Following are some features of RISC architecture in the PIC microcontrollers :

(i) The RISC processors have a instruction size that is fixed. Hence, the CPU can decode the instructions at a faster speed and efficiently. In a CISC microcontroller e.g. 8051 the instruction size is variable. The instructions can be 1 byte (SWAP A), 2 byte (ADD A, #50H) or 3 byte (LJMP dest_address). The variable instruction size makes it difficult for the decoder as the size of the next instruction cannot be predicted.

(ii) The number of registers in a system affect the performance of a system. One main characteristic of the RISC processors is they have a large number of registers. Generally, all RISC architectures have 32 registers. With a large number of registers the need of stack to store parameters is avoided.

(iii) The RISC processors have a small instruction set. They support the basic instructions like ADD, SUB, MUL, LOAD, STORE, AND, OR, EX-OR, JUMP, CALL etc. The limitation on the number of instructions is a criticism levelled at the RISC processor because it makes the job of programmers difficult. Hence, RISC is commonly used in high-level language environments like C programming. The RISC programs are large because of limitation on the number of instructions. Though these programs need more memory, it is not a problem because the memory is cheap. In PIC16 there are around 35 instructions and PIC18 has 75 instructions.

(iv) The most important characteristic of the RISC processor is that more than 95% of the instructions are executed with one clock cycle.

(v) The RISC processors have separate buses for data and code.

There are four set of buses. They are :

(a) A set of data buses for carrying data (operands) in and out of the CPU.

(b) A set of address buses for accessing the data.

(c) A set of buses to carry the opcodes.

(d) A set of address buses to access the opcodes.

The use of separate buses for code and data operands is called as Harvard architecture.

(vi) In RISC processors the instruction set is small, so they are implemented using hardwired method. It takes no more than 10% of the transistors. While in CISC processors the instruction set is large and with so many addressing modes, microinstructions are used to implement them. The implementation of microinstructions inside the CPU takes 40% to 60% of the transistors.

(vii) RISC processors use load/store architecture. The instructions can load from external memory into registers or store registers into external memory locations.

Syllabus Topic : Comparison and Selection of PIC Series as per Application

4.2 Comparison and Selection of PIC Series as per Application

- The microchip technology corporation in 1989, came up with the first 8 bit microcontroller called **PIC (Peripheral Interface Controller).**
- Fig. 4.2.1 shows the block diagram of PIC microcontroller.

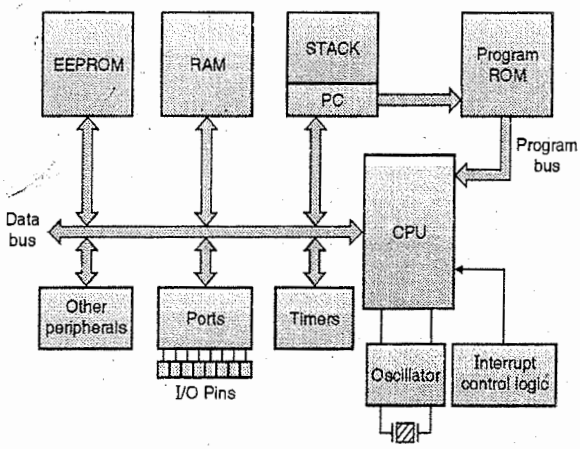


Fig. 4.2.1 : Block diagram of PIC microcontroller

- As shown in Fig. 4.2.1 the PIC microcontroller consists of on-chip ROM, RAM, Timers, I/O ports.
- The PIC family of microcontrollers introduced an array of 8 bit microprocessors. They comprise PIC10Fxx, PIC12Fxx, PIC16Fxx, PIC17Fxx and PIC18Fxx families. All these microcontroller families are 8 bit microcontroller families. It indicates that at a time each PIC microcontroller can process 8 bit data.

- The different 8 bit microcontrollers developed by microchip are :
  (i) PIC10xxx : 8 pin, 12 bit instruction format. (ii) PIC12xxx : 28 pin, 12 or 14 bit instruction format.
  (iii) PIC16Cxx : 14 bit instruction format. (iv) PIC17 : 16 bit instruction format.
  (v) PIC18 : 16 bit instruction format.
- Every PIC microcontroller supports different number of instructions and has a slightly different instruction format. Also the design of their peripheral functions is different.

Drawback : Hence, a drawback of PIC family is that the microcontroller are not totally compatible in software when we change from one family to another. e.g. PIC12Fxx have 12 bit wide instructions, PIC16Fxx have 14 bit wide instructions whereas PIC18Fxx have 16 bit wide instructions.

- The PIC18 family members share the same peripheral functions, instruction set and have 8 to 80 functional pins. Hence, it used for new designs as we can upgrade the PIC18 application to powerful chip versions. Also they are software compatible.
- If we need a small package then PIC10xxx – PIC16xxx family of microcontrollers is used.

4.2.1 Comparison of Features of PIC10, PIC12, PIC16, PIC18 Families SPPU - Aug. 16

University Question

Q. Compare PIC10, PIC12, PIC16 and PIC18 series. **(Aug. 2016 (In Sem.), 5 Marks)**

Features	PIC10	PIC12	PIC16	PIC18
Number of pins	6-40	8-64	8-64	18-80
Instruction size	12 bits	12/14 bits	14 bits	16 bits
Number of instructions	33	35	49	83
Timers	1	1	3	4
Parallel slave Port	---	--	Yes	Yes
Performance	5 MIPS	5 MIPS	8 MIPS	Up to 16 MIPS
Hardware stack	2 level	8 level	16 level	32 level
Other features Supported:	– Comparator – 8-bit ADC – Data Memory – Internal Oscillator	– SPI/I2C – Comparator – 8-bit ADC – Data Memory – Internal Oscillator – UART – PWMs – LCD – 10-bit ADC	– SPI/I2C – Comparator – 8-bit ADC – Data Memory – Internal Oscillator – UART – PWMs – LCD – 10-bit ADC	– SPI/I2C – Comparator – 8-bit ADC – Data Memory – Internal Oscillator – UART – PWMs – LCD – 10-bit ADC – 8×8 Hardware Multiplier – CAN – USB – Ethernet
Applications	(1) Security systems (2) Personal care appliances (3) Low power remote transmitters and receivers	(1) Security systems (2) Personal care appliances (3) Low power remote transmitters and receivers	(1) Sensing Robot Arm position (2) Measurement of angular speed (3) Data acquisition systems (4) Control of DC motors (5) Stepper motor control (6) PID controllers (7) Low power remote transmitters and receivers (8) Real time clocks (9) Alarm systems	(1) Data acquisition Systems (2) Home automation systems (3) Lighting control (4) DC motor control (5) Stepper motor control (6) Alarm systems (7) Industrial controllers (8) Embedded systems (9) PID controllers in process control systems (10) Security systems

4.3 PIC 18F458 Features

SPPU : Dec. 12, Dec. 13, Aug. 15, Dec. 16

University Questions

Q. State features of the PIC 18F458. **(Dec. 2012, 4 Marks)**

Q. List features of PIC18Fxx microcontroller. **(Dec. 2013, 6 Marks)**

Q. List various features of PIC18. **(Aug. 2015 (In Sem.), 5 Marks)**

Q. Write feature of PIC18FXX microcontroller over PIC16FXXX. **(Dec. 2016, 6 Marks)**

4.3.1 Flash Technology

1. It has a PIC microcontroller with low-power, and high-speed Enhanced Flash technology.
2. It supports a completely static design.
3. It supports a wide operating voltage ranging from 2.0V to 5.5V.
4. It supports operation in the industrial and Extended temperature ranges.

4.3.2 Advanced Analog Features

1. It has a on – chip 10-bit, 8-channel. Analog-to-Digital Converter.
2. It has two analog comparators.
3. It supports programmable Brown-out Reset (BOR).
4. It has a programmable Low-Voltage Detection (LVD) module that invokes interrupts when a low voltage is detected.

4.3.3 Peripheral Features

1. It has a high current sink/source 25 mA/25 mA
2. It supports three external interrupt pins INT0, INT1 and INT2.
3. It supports four timer such that
   (a) Timer 0 module is an 8-bit/16-bit timer/counter with 8-bit programmable prescaler.
   (b) Timer 1 module is a 16-bit timer/counter
   (c) Timer 2 module: 8-bit timer/counter with 8-bit period register (time base for PWM)
   (d) Timer 3 module: 16-bit timer/counter
4. It supports a Capture/Compare/PWM (CCP) module that can be used in any of the following modes :
   (a) Capture mode
   (b) Compare mode
   (c) PWM generation
5. The Enhanced CCP module supports all the features of the standard CCP module and has additional features for advanced motor control. They are :
   (a) 1, 2 or 4 PWM outputs
   (b) Selectable PWM polarity
   (c) Programmable PWM dead time
6. It supports a Master Synchronous Serial Port (MSSP) module that supports two operating modes :
   (a) SPI
   (b) I2C
7. It has an addressable USART module for serial communication.
8. It supports a CAN bus module.

4.3.4 Special Microcontroller Features

1. It supports power-on Reset (POR)
2. It supports Power-up Timer (PWRT)
3. It supports Oscillator Start-up Timer (OST).
4. It has a watchdog Timer (WDT) that has its own on-chip RC oscillator.
5. It provides programmable code protection.
6. It supports the power-saving Sleep mode.
7. We can select the oscillator as per the application .
8. It supports in-Circuit Serial Programming.

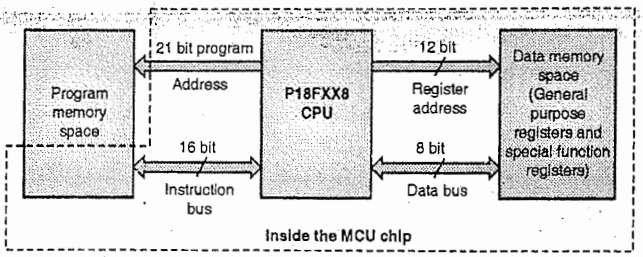


Fig. 4.9.2 : PIC18 Data and Program Memory Spaces

4.9.1 Program Memory Organization

SPPU - May 12, Dec. 12, May 13, Dec. 13, Aug. 14, May 15, Aug 15

University Questions

Q. Draw and explain program memory map of PIC microcontroller. **(May 2012, 4 Marks)**

Q. Explain Program memory organization of PIC in detail. **(Dec. 2012, 4 Marks)**

Q. Describe in details memory organization of 18Fxxx. **(May 2013, 8 Marks)**

Q. Explain memory organization of PIC microcontroller. **(Dec. 2013, 8 Marks)**

Q. Draw and explain structure of program memory map of PIC. **(Aug. 2014(In Sem.), Aug. 2015(In Sem.), 5 Marks)**

Q. Explain memory mapping of PIC18F Microcontroller ? **(May 2015, 8 Marks)**

- The PIC18F458 has a 21-bit program counter. Through this program counter we can access **2-Mbyte program memory space.**
- The program memory contains instructions for execution and data tables for storing fixed data. Fig. 4.9.3 shows the diagram for the program memory map and stack for the PIC18F458.
- As shown in Fig. 4.9.3 the microcontroller has a 32 level stack . This stack holds the return addresses for interrupts and subroutine calls . This return address stack is not part of the program memory space.
- As shown in Fig. 4.9.3, the address 0000 H is allocated assigned to the **Reset vector.** This address is the address at which a after reset a microcontroller program will begin.
- The PIC18 has two interrupt priority levels ; high and low priority interrupts .
- The high-priority interrupt service routine is allocated address 0008 H. The PIC18 microcontroller assigns sixteen bytes to the high-priority ISRs (interrupt service routine) by default.
- The low-priority interrupt service routine is allocated address 0018 H. The PIC18 microcontroller low-priority ISRs can be of any size. After the execution of the ISR the microcontroller should continue with the main program execution.

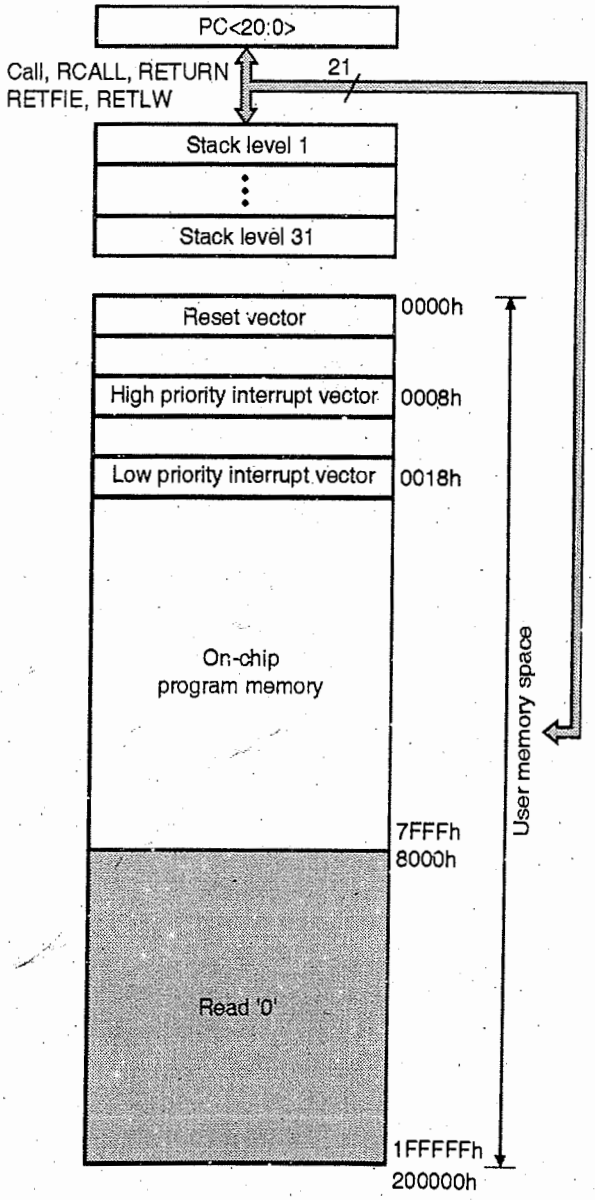


Fig. 4.9.3: Program memory map and stack for PIC18F458

4.9.1.1 Program Counter

Q. Draw and explain the program counter of PIC microcontroller.

- The Program Counter (PC) is a **21 bit register** that holds the address of an instruction in memory. Fig. 4.9.4 shows the program counter.
- Its **function** is to keep the track of program execution. The program instruction bytes are fetched from locations in memory that are addressed by the program counter.

- As shown in Fig 4.9.4 the lower byte of program counter is called as the **PCL register**. This register is directly accessible to the user.
- It is an 8 bit register. This register is readable and writable.
- The program counter higher byte is called as the **PCH register**. It is an 8 bit register contains the PC15 – PC 8bits. This register is not directly readable or writable.

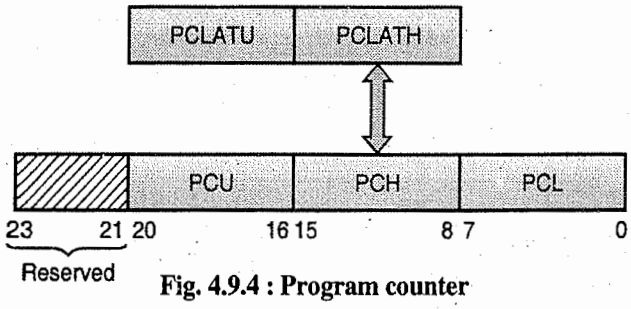


Fig. 4.9.4 : Program counter

- The PCLATH register is used to do updates on the PCH register .
- PCU is the upper byte of the 21 bit program counter. It has bits PC20 – PC16. The PCU is not directly readable or writable. The PCLATU register is used for updating the PCU register.
- The data bytes in program memory are accessed by the PC.
- With the help of the branching instructions like RCALL, CALL, GOTO and BRA use write can directly to the program counter directly.

4.9.2 Data Memory Organization

SPPU - May 12, Dec. 12, May 13, Dec. 13, Dec. 14, May 15, Dec. 15, May 16, Aug. 16

University Questions

Q. Explain data memory organization of PIC microcontroller. **(May 2012, 3 Marks)**

Q. Explain data memory organization of PIC in detail. **(Dec. 2012, 8 Marks)**

Q. Describe in details memory organization of 18Fxxx **(May 2013, 8 Marks)**

Q. Explain memory organization of PIC microcontroller. **(Dec. 2013, 8 Marks)**

Q. Draw and explain the data memory map of PIC18fxx series. **(Dec. 2014, 8 Marks)**

Q. Explain memory mapping of PIC18F Microcontroller. **(May 2015, 8 Marks)**

Q. Explain data memory organization with details description of GPRs and SFR in PIC18F458. **(Dec. 2015, 8 Marks)**

Q. Explain in detail data memory map of PIC18F with GPR and SFRs. **(May 2016, 8 Marks)**

Q. Explain data memory organization of PIC18. **(Aug. 2016(In sem), 5 Marks)**

- The data memory is implemented as static RAM (SRAM).
- Every location in the data memory is called as a **File Register or register**.
- The PIC18 microcontroller supports **4 KB of data memory** such that every data register has a 12 bit address.
- Fig. 4.9.5 shows the data memory organization for the PIC18FXX8 devices.
- The PIC18 instructions are of 16 bits .Hence, for specifying the file register only 8 bits are used. As a result the 4096 file registers are divided into 16 banks.
- Every bank has (4096/16) = 256 bytes. For accessing a particular bank the lower 4 bits of the BSR (bank select register) are used.
- At a time only one register bank can be active. For changing the register bank we need to modify the lower 4 bits of BSR. The upper four bits of BSR are unimplemented.
- **Two types of registers** are available in the data memory as shown in Fig. 4.9.5. They are :

1. General-purpose registers (GPRs)
2. Special-function registers (SFRs).

- The **General Purpose Registers (GPRs)**are used storing the data for different program modules and applications .
- The **Special Function Registers (SFRs)**are used by the microcontroller and the different peripherals in order to control the device operation.
- Bank 0 to Bank 14 is allocated for GPRs. Bank 15 is allocated to SFRs.
- The data memory may be accessed directly or indirectly.
- Direct addressing may or may not the use of the BSR register.
- However indirect addressing needs 12 bit the File Select Register (FSR) for accessing a location in the memory map .
- All banks the banks can be accessed with the help of PIC18FXXX instructions.
- An **Access Bank** is used to group the bank 0 GPR and bank 15 SFR so that they can be accessed in a single cycle.

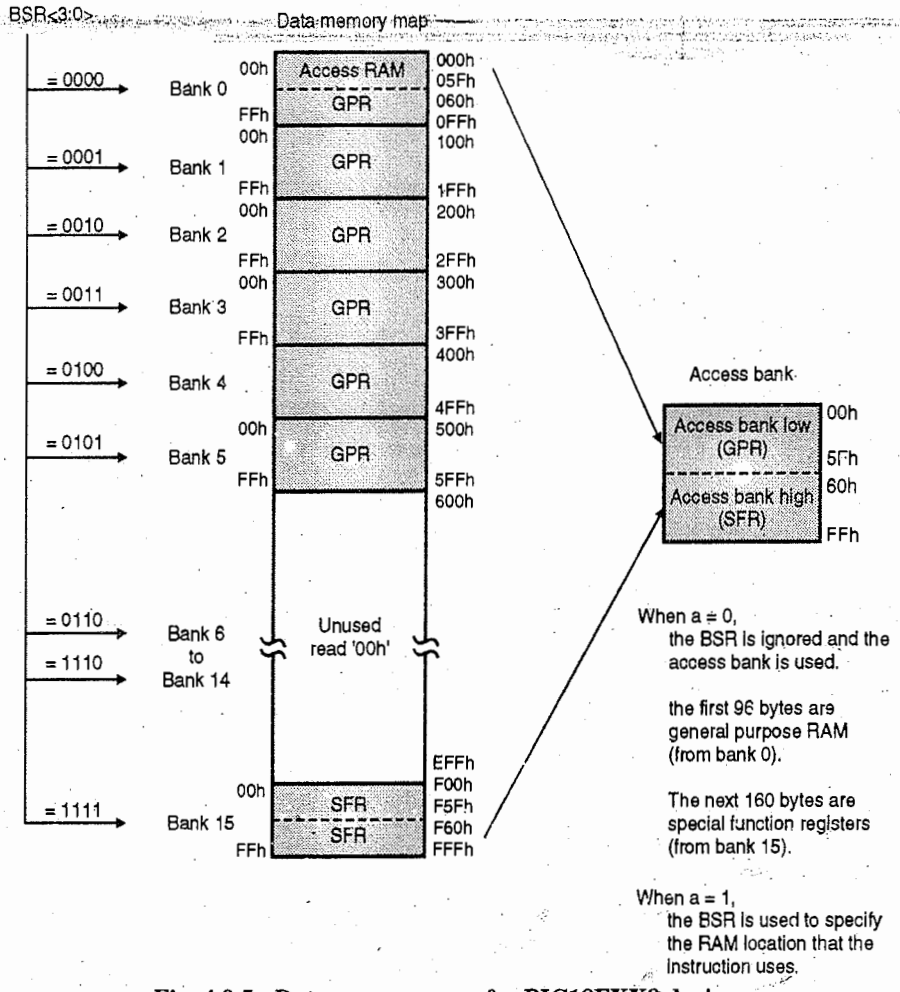


Fig. 4.9.5 : Data memory map for PIC18FXX8 device

4.9.2.1 Access Bank

- As shown in Fig. 4.9.5 the **Access Bank** is made up of Bank 0 GPR lower part and Bank 15 SFR higher part .It ensures that the commonly used registers can be accessed in a single cycle.
- As shown in Fig. 4.9.5 the two parts are called as Access bank low (GPRs) and Access bank high (SFRs).
- Of the 256 bytes of Access bank 128 bytes are allocated to GPRs and 128 bytes are allocated to SFRs.
- An "**A bit** "(access bit) in the instruction indicates whether the operation will take place in the Access Bank or the BSR. If A=0 then access bank is default access bank.
- The Access bank is useful for testing status flags, modifying control bits, software stacks, and context saving of registers.

4.9.2.2 Bank Select Register (BSR)

SPPU - May 12

University Question

Q. Explain use of bank select register. **(May 2012, 2 Marks)**

- An "A bit "(access bit) in the instruction indicates whether the operation will take place in the Access Bank or the BSR. If A=1 then (BSR) bank select register is used for selecting the desired bank.
- BSR is an 8 bit SFR. The lower 4 bits are used for selecting the bank and remaining 4 bits are zero.
- The 4 bit BSR along with 16 banks of 256 bytes accesses 16x 256 = 4KB of data memory.
- On reset BSR is set to 0. If BSR = 1 bank 1 is selected. Similarly if BSR = 2, bank 2 is selected.

4.9.2.3 Data Memory Registers

SPPU - Dec. 15

University Question

Q. Explain data memory organization with details description of GPRs and SFR in PIC18F458. **(Dec. 2015, 8 Marks)**

Two types of registers are available in the data memory. They are :

1. General-purpose registers (GPRs) 2. Special-function registers (SFRs).

4.9.2.3(A) General Purpose Register File

- The **general purpose registers** are a group of RAM locations in the file register. Fig. 4.9.5 shows the general purpose registers. They are located from Bank 0 to Bank 14 in the data memory.
- The general purpose registers are mainly used for storing the data.
- The general purpose registers are of 8 bit.
- The register file can be accessed either directly or indirectly. Indirect addressing is done with the File Select Registers (FSR).
- They are not initialized on Reset. Their contents are unmodified on other resets.

4.9.2.3(B) Special Function Registers

- The **Special Function Registers (SFRs)** are registers used by the CPU and peripherals for controlling the desired operation of the device e.g. serial communication, timers, counters, PWM etc.
- The SFRs are implemented as static RAM.
- Each and every SFR is an 8 bit register. Fig. 4.9.6 shows the PIC18 registers .
- The SFRs can either be accessed by their names or their addresses.
- The SFRs can be classified into two types. They are :
  (i) The SFRs related with the "core" function.
  (ii) The SFRs related to the peripheral functions. Eg TMR0,TMR1 They have addresses from F80 H to FFF H .
- The unused SFR locations are read as '0's.

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2	FBCh	ECCPR1H	F9Ch	–
FFBh	PCLATU	FDBh	PLUSW2	FBBh	ECCPR1L	F9Bh	–
FFAh	PCLATH	FDAh	FSR2H	FBAh	ECCP1CON	F9Ah	–
FF9h	PCL	FD9h	FSR2L	FB9h	–	F99h	–
FF8h	TBLPTRU	FD8h	STATUS	FB8h	–	F98h	–
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	ECCP1DEL	F97h	–
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	ECCPAS	F96h	TRISE
FF5h	TABLAT	FD5h	T0CON	FB5h	CVRCON	F95h	TRISD
FF4h	PRODH	FD4h	–	FB4h	CMCON	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	–
FF0h	INTCON3	FD0h	RCON	FB0h	–	F90h	–
FEFh	INDF0[2]	FCFh	TMR 1H	FAFh	SPBRG	F8Fh	–
FEEh	POSTINC0	FCEh	TMR 1L	FAEh	RCREG	F8Eh	–
FEDh	POSTDEC0	FCDh	T1CON	FADh	TXREG	F8Dh	LATE
FECh	PREINC0	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD
FEBh	PLUSW0	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	–	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	–
FE7h	INDF1	FC7h	SSPSTAT	FA7h	EECON2	F87h	–
FE6h	POSTINC1	FC6h	SSPCON1	FA6h	EECON1	F86h	–
FE5h	POSTDEC1	FC5h	SSPCON2	FA5h	IPR3	F85h	–
FE4h	PREINC1	FC4h	ADRESH	FA4h	PIR3	F84h	PORTE
FE3h	PLUSW1	FC3h	ADRESL	FA3h	PIE3	F83h	PORTD
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	–	FA0h	PIE2	F80h	PORTA

Fig. 4.9.6 : Special function register map

4.10 Status Register

SPPU - Dec. 12, May 13, Dec. 13, May 14, Aug. 14, Aug. 15

University Questions

Q. Explain status register of PIC. **(Dec. 2012, 4 Marks)**

Q. Draw and explain status register of PIC microcontroller. **(May 2013, 8 Marks)**

Q. Explain status register of PIC microcontroller. **(Dec. 2013, 6 Marks)**

Q. Draw and explain status register of PIC controller. **(May 2014, 8 Marks)**

Q. What is the function of status register in PIC microcontroller. Explain in detail. **(Aug. 2014(In sem), 3 Marks)**

Q. Explain status register in PIC18. **(Aug. 2015(In sem), 2.5 Marks)**

- The Status register is shown in Fig. 4.10.1. It is an 8 bit register. It is also called as **flag register.**
- There are five conditional flags as shown in Fig. 4.10.1.

-	-	-	N	OV	Z	DC	C
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

bit 7-5 Unimplemented : Read as '0'

bit 4 **N : Negative bit**
1 = Result was negative, 7th MSB bit of result is 1
0 = Result was positive, 7th MSB bit of result is zero.

bit 3 **OV : Overflow bit**
1 = Overflow occurred for signed arithmetic
0 = No overflow occurred

bit 2 **Z : Zero bit**
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero.

bit 1 **DC : Digit Carry bit**
For ADDWF, ADDLW, SUBLW and SUBWF instructions.
1 = A carry-out from the 4th low-order bit of the result occurred.
0 = No carry-out from the 4th low-order bit of the result.

bit 0 **C : Carry bit**
For ADDWF, ADDLW, SUBLW and SUBWF instructions :
1 = A carry-out from the Most Significant bit of the result occurred
0 = No carry-out from the Most Significant bit of the result occurred.

Fig. 4.10.1 : Status register

- Many instructions affect the status flags. They are 1 bit registers provided to store the result of some instructions .e.g. a carry flag is set if there is a carry out of the MSB bit of result.
- The instructions BSF, BCF, MOVFF, MOVWF and SWAPF do not affect the Z, C, DC, OV and N bits of the STATUS register. Hence, they can be used to change the STATUS register.
- If the STATUS register is used as destination then write to N, OV, Z, DC and C bits is disabled. Depending on the microcontroller logic these flag bits are set or cleared.

4.11 RCON Register

- The Reset Control (RCON) register contains flag bits that allow differentiation between the sources of a device reset.
- These flags include the $\overline{TO}$, $\overline{PD}$, $\overline{POR}$, $\overline{BOR}$ and $\overline{RI}$ bits.
- This register is readable and writable.

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 (PIC16CXXX Compatibility mode)

bit6-5 Unimplemented : Read as '0'

bit 4 $\overline{RI}$: RESET Instruction Flag bit
1 = The RESET instruction was not executed
0 = The RESET instruction was executed causing a device Reset (must be set in software after a Brown-out Reset occurs)

bit 3 $\overline{TO}$: Watchdog Time-out Flag bit
1 = After power-up, CLRWDT instruction or SLEEP instruction
0 = A WDT time-out occurred.

bit 2 $\overline{PD}$: Power-down Detection Flag bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 1 $\overline{POR}$: Power-on Reset Status bit
1 = A Power-on Reset has not occurred
0 = A Power-on reset occurred (must be set in software after a Power on Reset occurs)

bit 0 $\overline{BOR}$: Brown-out reset status bit
1 = A Brown-out Reset has not occurred.
0 = A Brown-out Reset occurred (must be set in software after a Brown-out Reset occurs)

Fig. 4.11.1 : RCON register

4.12 Stack and Stack Pointer

SPPU - May 12

University Question

Q. Draw and explain stack of PIC microcontroller.

(May 2012, 3 Marks)

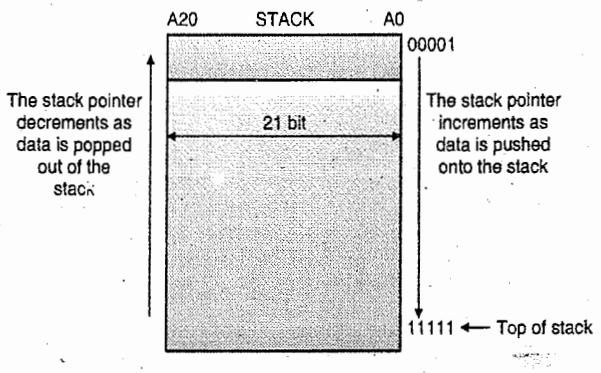


Fig. 4.12.1 : PIC stack

Function : The stack is a reserved RAM area of memory where temporary data or address can be stored. A **5 bit stack pointer** is used to hold the address of the most recent entry.

- The stack is of 21 bit and can acquires values from 000000H to 1FFFFFH. The size of stack is 21 bit because the program counter is of 21 bits.

Size : The **5 bit stack pointer** can access $2^5 = 32$ locations making the stack a **32 level deep stack**. Each stack location is **21 bits wide** as shown in Fig. 4.12.1. The stack pointer can take values from 00H to 1FH.

- While the information is written on the stack, the operation is called **PUSH**. When the information is read from the stack, the operation is called **POP**.

Reset address : On reset, the SP register is initialized to 0.

- The stack pointer register is not readable or writable.
- Whenever a CALL instruction is executed the contents are PC are loaded or PUSHED onto the stack and after a RETURN, RETFIE or RETLW instruction the contents of PC are poped off from the stack.
- The last location pointed by the stack pointer is called as **Top of stack.**
- Stack pointer is incremented by 1 when data is pushed onto the stack and when the data is to be retrieved/popped from the stack, SP decrements by 1.
- **Stack is mainly used for interrupts and calls.**

Syllabus Topic : Oscillator Options (CONFIG)

4.13 Oscillator Options (CONFIG)

SPPU - Aug. 14

University Question

Q. What are the various oscillator options ? How they can be selected using config register.

(Aug. 2014(In Sem.), 5 Marks)

- We need the system clock for operating the microcontroller system and peripherals and for executing the programs properly.
- One instruction cycle comprises of four device **system clock periods** (T_{SCLK}).
- T_{SCLK} is derived from an external system clock. The oscillator mode is decided by the configuration bits that are programmed.
- The different operating modes of the oscillator are :
  1. **LP :** Low Frequency (Power) Crystal
  2. **XT :** Crystal/Resonator
  3. **HS :** High Speed Crystal/Resonator
  4. **HS4 :** High Speed Crystal/Resonator with 4x frequency PLL multiplier enabled
  5. **RC :** External Resistor/Capacitor
  6. **EC :** External Clock
  7. **ECIO :** External Clock with I/O pin enabled
  8. **RCIO :** External Resistor/Capacitor with I/O pin enabled
- OSC1 is the default oscillator. Fig. 4.13.1 shows the oscillator clock sources.

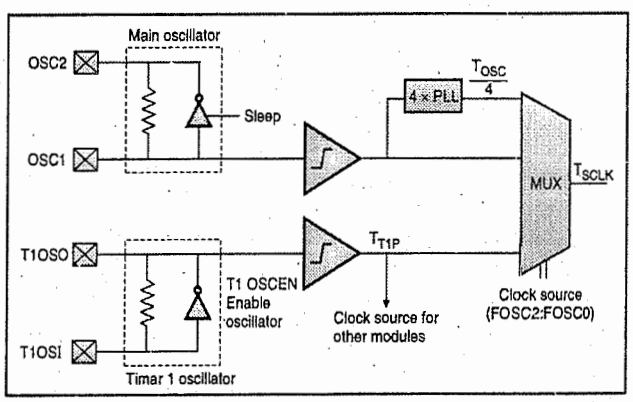


Fig. 4.13.1 : Oscillator clock sources

- The Timer 1 oscillator operates at 32 KHz T_{T1P} is the output got from Timer 1 oscillator as shown in Fig. 4.13.1.
- The different oscillator options are used for different applications.

Syllabus Topic : CONFIG1H Register

4.13.1 CONFIG1H Register

- The CONFIG1H register is used for selecting the clock oscillator.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
–	–	$\overline{\text{OSCSEN}}$	–	–	FOSC2	FOSC1	FOSC0

bit 7 – 6 : Unimplemented.

Bit 5 : $\overline{\text{OSCSEN}}$ (Oscillator system clock switch Enable Bit)
1 : Oscillator system clock switch disabled.
0 : Oscillator system clock switch enabled.

bit 4 - 3 : unimplemented.

bit 2 : 0 : FOSC2 : FOSC1 Oscillator Selection Bits.

FOSC2:FOSC0 Configuration Bits 2-0	Oscillator Mode
000	LP : Low Frequency (Power) Crystal
001	XT : Crystal/Resonator
010	HS : High Speed Crystal/Resonator
011	RC : External Resistor/Capacitor
100	EC : External Clock
101	ECIO : External Clock with I/O pin enabled
110	HS4 : High Speed Crystal/Resonator with 4x frequency PLL multiplier enabled
111	RCIO :External Resistor/Capacitor with I/O pin enabled

Fig. 4.3.2 : CONFIG1H Register

4.13.2 OSCCON Control Register

Fig. 4.13.2(a) shows the OSCON control Register. This register controls the clock switching between the main oscillator and Timer 1 oscillator.

bit 7	bit 1	bit 0
Unimplemented		SCS

bit 7 – 1 unimplemented Read as 0

bit 0 SCS – system clock switch bit

If $\overline{\text{OSCEN}}$ = '0' and T1OSCEN = 1
1 = Switch to Timer 1 oscillator/clock pin.
0 = Use primary oscillator/clock pin

If $\overline{\text{OSCEN}}$ and T1OSCEN have other values,
SCS = 0

Fig. 4.13.2(a) : OSCON control register

4.13.3 Types of Oscillators

- By programming the configuration bits FOSC2:FOSC0 a programmer can select one of the eight operating modes. The eight oscillator operating modes are as follows :

1. **LP** : Low Frequency (Power) Crystal
2. **XT** : Crystal/Resonator
3. **HS** : High Speed Crystal/Resonator
4. **HS4** : High Speed Crystal/Resonator with 4x frequency PLL multiplier enabled
5. **RC** : External Resistor/Capacitor
6. **EC** : External Clock
7. **ECIO** : External Clock with I/O pin enabled
8. **RCIO** : External Resistor/Capacitor with I/O pin enabled

FOSC2:FOSC0 Configuration Bits	Oscillator Mode	Oscillator Function
000	LP : Low Frequency (Power) Crystal	It takes the least current and is suitable for low frequency or low power applications.
001	XT : Crystal/Resonator	-
010	HS : High Speed Crystal/Resonator	It takes the maximum current of all the modes. It is used for high frequency applications.
011	RC : External Resistor/Capacitor	It provides a clock out such that the oscillator frequency is divided by 4.
100	EC : External Clock	It provides a clock out such that the oscillator frequency is divided by 4.
101	ECIO : External Clock with I/O pin enabled	I/O
110	HS4 : High Speed Crystal/Resonator with 4x frequency PLL multiplier enabled	It takes the maximum current of all the modes. It is used for high frequency applications. The PLL multiplies the external clock frequency by 4.
111	RCIO :External Resistor/Capacitor with I/O pin enabled	I/O

4.13.4 Crystal Oscillators/Ceramic Resonators for LP, XT, HS and HS4 Modes

- Fig. 4.13.3 shows the crystal oscillator for LP, XT, HS and HS4 modes.

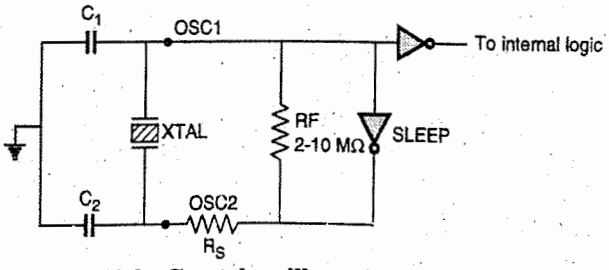


Fig. 4.13.3 : Crystal oscillator for XT, LP, HS and HS4 modes

- As shown in Fig. 4.13.3 the crystal oscillator is connected to the OSC1 and OSC2 pins. A series or parallel cut crystal is used.

4.13.5 Crystal Oscillators for EC or ECIO Modes

- Fig. 4.13.4 shows the EC oscillator. It uses an external clock. In this mode the OSC1 pin is driven by CMOS drivers and at OSC2 pin the oscillator frequency is divided by 4 for testing and synchronization applications .

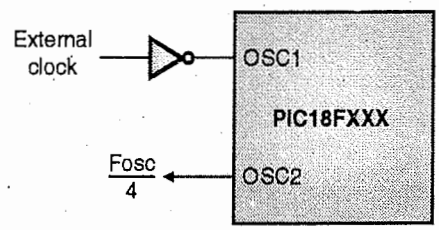


Fig. 4.13.4 : External clock for EC mode

- After a brown-out Reset there is some delay for power up .After a power on Reset if the PWRT is disabled there is no time out.
- Fig. 4.13.5 shows the oscillator for ECIO mode. It also uses an external clock .In this mode the OSC1 pin is driven by CMOS drivers and OSC2 pin is multiplexed with an I/O pin .

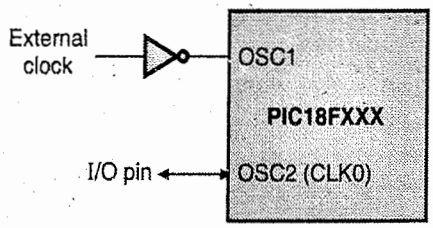


Fig. 4.13.5 : External clock input operation

- After a brown-out Reset there is some delay for power up . After a power on Reset if the PWRT is disabled there is no time out.

4.13.6 External RC Oscillator

Fig. 4.13.6 shows the External RC oscillator.

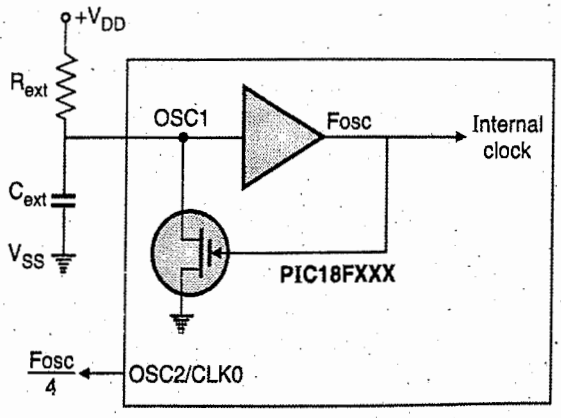


Fig. 4.13.6 : RC oscillator mode

- Its oscillation frequency is a function of :
  (a) R_{EXT}
  (b) C_{EXT}
  (c) Supply voltage (d) Operating temperature
- It is used for applications that are not dependent on time.

4.13.7 External RC Oscillator with I/O Enabled

- Fig. 4.13.7 shows the External RC oscillator I/O mode.
- Its oscillation frequency is a function of :
  (a) R_{EXT}
  (b) C_{EXT}
  (c) Supply voltage
  (d) Operating temperature
- It is used for applications that are not dependent on time. The OSC2 pin is set up as an I/O pin.

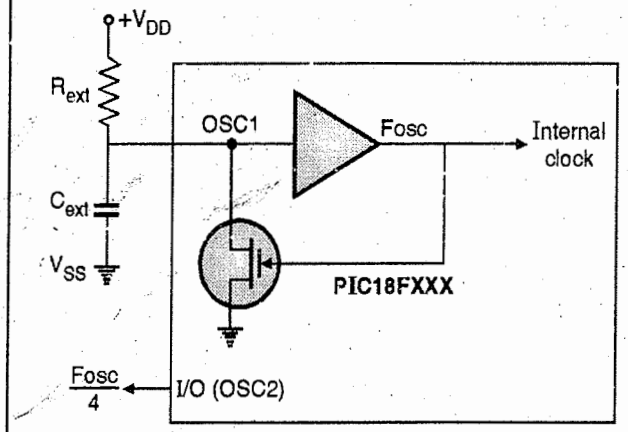


Fig. 4.13.7

4.14 Clock / Instruction Cycle

- The clock input (from OSC1) is internally divided by four in order generate four non-overlapping quadrature clocks, namely Q1, Q2, Q3 and Q4.
- After every instruction cycle the Program Counter (PC) is incremented by 1 .
- Fig. 4.14.1 shows the clocks and instruction execution.
- In the clock cycle Q4 , generally the instruction is fetched from the program memory then the instruction is latched into the instruction register.

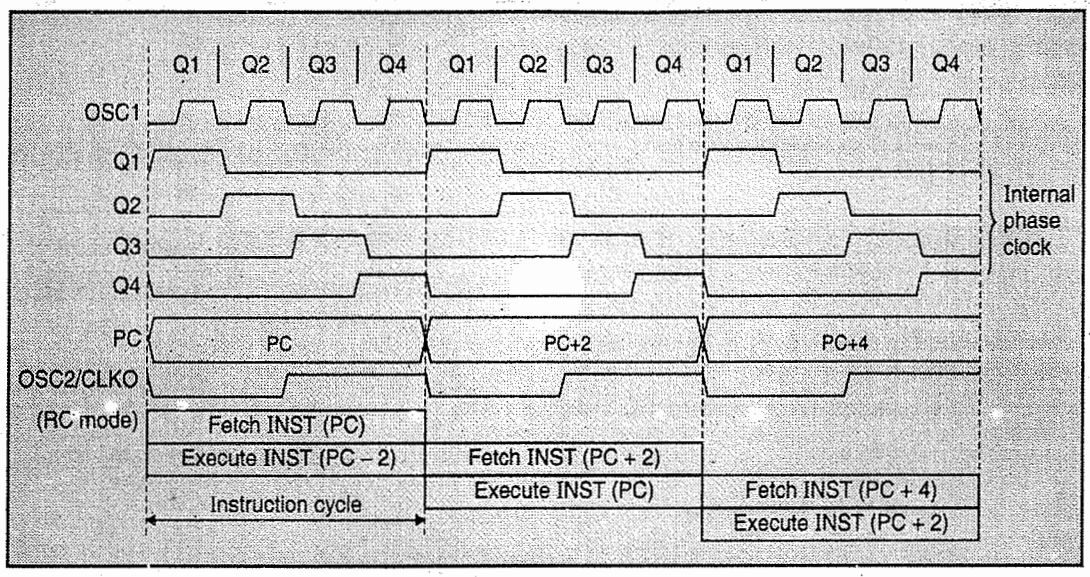


Fig. 4.14.1 : Clock/Instruction cycle

4.14.1 Instruction Flow/Pipelining

SPPU - Dec. 12, Dec. 15

University Questions

Q. Explain instruction pipeline structure of PIC with the help of example. **(Dec. 2012, 8 Marks)**

Q. Explain instruction pipeline flow in PIC18F. **(Dec. 2015, 8 Marks)**

- An **"Instruction Cycle"** consists of four Q cycles (Q1, Q2, Q3 and Q4) as shown in Fig. 4.14.1.
- The PIC18 microcontroller needs four takes 4 clock periods of the oscillator for executing an instruction.

Q_1	Q_2	Q_3	Q_4
DECODE	READ	PROCESS	WRITE TO

Fig. 4.14.2 : Pipeline flow after instruction is fetched

- During the first clock period Q_1 the instruction that is fetched is decoded and placed in the instruction queue.
- During the second clock period Q_2 the microcontroller fetches the operand from the file register that is specified.
- During the third and fourth clock periods Q_3 and Q_4 the instruction execution is completed and the result is placed in the desired destination register.
- **Pipelining** is defined as the method of fetching the next instruction when the current instruction is being executed. Pipelining is possible because of queue.
- The instruction fetch and execute are pipelined such that fetch takes one instruction cycle, while decode and execute take another instruction cycle.
- However, because of pipelining, each instruction except the branch instructions executes in one cycle.
- If a branch instruction is executed the contents of the program counter are modified (e.g. GOTO, BRA, CALL) then for executing the instruction we need two cycles .
- Fig. 4.14.3 shows an instruction pipeline flow.

	$T_{CY}0$	$T_{CY}1$	$T_{CY}2$	$T_{CY}3$	$T_{CY}4$	$T_{CY}5$
1. MOVLW 30H	Fetch 1	Execute 1				
2. MOVWF PORTC		Fetch 2	Execute 2			
3. BRA L1			Fetch 3	Execute 3		
4. BSF PORTB,2				Fetch 4	Flush	
5. Instruction @ address L1					Fetch L1	Execute L1

Fig. 4.14.3 : Instruction pipeline flow

Note : Except the branch instructions , All instruction are are executed in single cycle.

4.14.2 Advantage of Pipelining

The execution unit always reads the next instruction byte from the queue. This process is faster than sending out the address to memory and waiting for next instruction to arrive. Pipelining eliminates the execution unit waiting time and speeds up the processing.

Syllabus Topic : RESET Operations

4.15 RESET Operations

- The main function of Reset circuit is place values in the registers such that the Microcontroller can start or restart without any intervention.
- The PIC18FXX8 supports different types of RESET. They are :

(a) Power-on Reset (POR)	(b) $\overline{MCLR}$ Reset during normal and Sleep
(c) Programmable Brown-out Reset (PBOR)	(d) Stack Full Reset
(e) RESET Instruction	(f) Watchdog Timer (WDT) Reset during normal operation
(g) Stack Underflow Reset	

- Fig. 4.15.1 shows the PIC18 microcontroller Reset circuit. The $\overline{MCLR}$ signal is the manual reset signal that when activated resets the PIC18 microcontroller.
- Most of the PIC18 registers are not affected by a reset. However some of the registers will be forced to any "RESET state" by the RESET instruction execution.
- After RESET the microcontroller begins program execution from memory address 0x0000 H. This address is called **reset vector**.

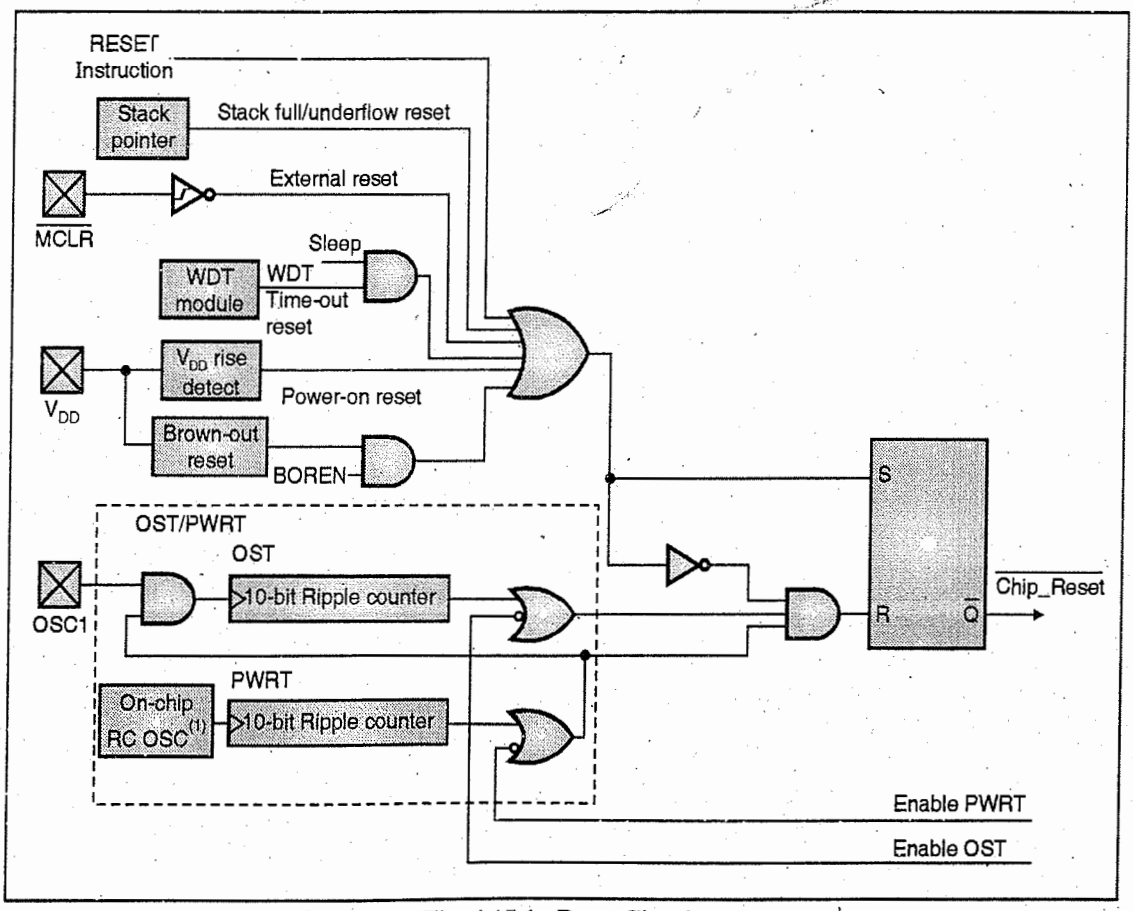


Fig. 4.15.1 : Reset Circuit

4.15.1 Power-on Reset (POR)

- A **power-on reset (POR)** pulse is generated when the microcontroller detects a rise in the V_{DD}.
- The $\overline{MCLR}$ is connected through a 1 KΩ to 10 KΩ resistor to V_{DD}. This eliminates the external RC components that are required for generating the delay.

4.15.2 $\overline{MCLR}$

- PIC18FXX8 devices have a noise filter in the $\overline{MCLR}$ Reset path. The filter will detect and ignore small pulses.
- $\overline{MCLR}$ pin uses an RC network for power on reset for ESD protection as shown in Fig. 4.15.2.

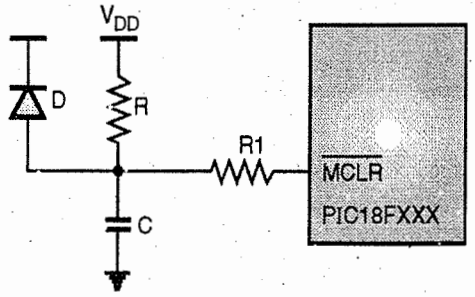


Fig. 4.15.2 : External power-on reset circuit (For Slow V_{DD} Power-Up)

4.15.3 Power-up Timer (PWRT)

- On power-up through POR, the PWRT (Power-up Timer) supplies a fixed nominal time-out. This timer operates on an internal RC oscillator.
- Till the PWRT is active the microcontroller is in RESET condition.
- PWRTEN is a configuration bit that is used for enabling /disabling the PWRT.

4.15.4 Oscillator Start-up Timer (OST)

- After the PWRT delay, the Oscillator Start-up Timer (OST) provides a 1024 oscillator cycle from OSC1 input.
- This delay guarantees the crystal oscillator or resonator is stabilized and has begun operation.
- The OST time-out is invoked only on Power-on Reset or wake-up from Sleep or for XT, LP, HS and HS4 modes of the oscillator.

4.15.5 PLL Lock Time-out

- PLL Lock Time-out is the fixed time supplied by the Power Up Timer for locking the oscillator frequency. This PLL lock time-out (TPLL) is typically 2 ms.

Syllabus Topic : Brown-out Reset (BOR) or Brown-out Detection (BOD)

4.15.6 Brown-out Reset (BOR) or Brown-out Detection (BOD)

SPPU - Dec. 2014, Aug. 2015

University Questions

Q. Explain the BOD mode of PIC18FXXX. **(Dec. 2014, 4 Marks)**

Q. Write a short note on BOD. **(Aug. 2015(In Sem.), 5 Marks)**

- **"Brown-out"** is a condition where the power level of the microcontroller temporarily becomes low.
- **BOREN** is a configuration bit used for enabling/disabling the Brown-out Reset circuit.
- If V_{DD} falls below the desired parameter the device may RESET. The microcontroller chip remains in Brown-out Reset till V_{DD} rises above V_{BOR} (trip voltage).
- If the power-up timer is enabled, then it will be activated after V_{DD} rises above V_{BOR}. The microcontroller will remain in RESET condition for an additional period.

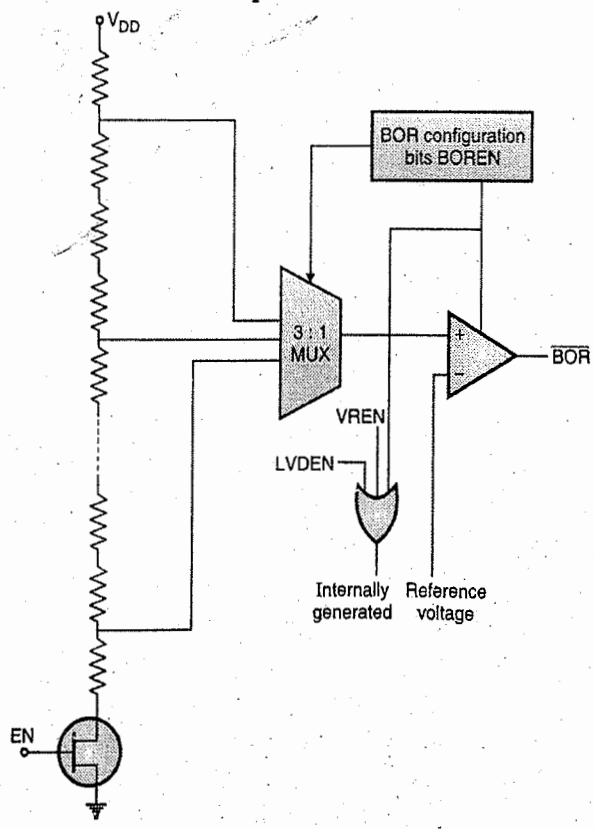


Fig. 4.15.3 : Block diagram of brown out detection circuit

Fig. 4.15.3 shows a circuit for brown out detection.

- The brownout circuit has four reset trip voltages.
- Fig. 4.15.4(a) and 4.15.4(b) shows external brown-out protection circuits. The internal brown out detection circuit must be disabled if any of circuits shown in Figs. 4.15.4(a) or (b) are used.

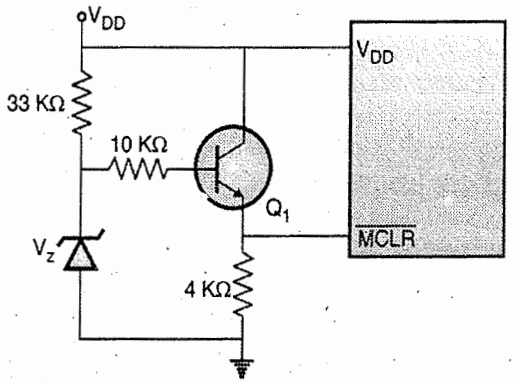


Fig. 4.15.4(a) : External Brown out protection circuit that resets when V_{DD} decreases below ($V_z \geq + 0.7$ V)

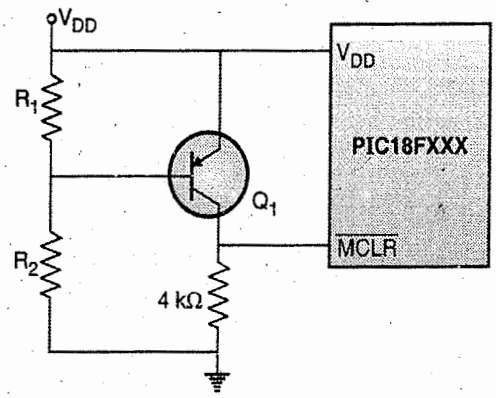


Fig. 4.15.4(b) : External brown-out protection

4.15.7 Reset Instruction

The reset instruction is used to reset all the registers and flags affected due to the MCLR reset.

4.16 Watchdog Timer (WDT)

SPPU - May 13, Aug. 16

University Questions

Q. Explain watchdog timer. **(May 2013, 8 Marks)**

Q. Explain the use of watch dog timer. **(Aug. 2016(In Sem.), 5 Marks)**

Use : The Watchdog Timer (WDT) can be used for doing a microcontroller RESET or making the microcontroller return to operating mode. This feature improves and enhances the overall microcontroller system operation.

- The Watchdog Timer is a free running, on-chip RC oscillator. This RC oscillator is different than the RC oscillator that is present at the OSC1/CLKI pin.
- Even if the system clock oscillator has stopped the WDT will run, even if a SLEEP instruction is executed by the microcontroller. When the watchdog Timer operates normally and if a time out occurs then in that condition the watch dog Timer generates a device Reset.
- When the watch dog timer is in the SLEEP mode and if a time out occurs then the microcontroller wakes up and continues normal operation.
- When a watch dog Timer time out occurs the $\overline{TO}$ bit in the RCON register is cleared i.e. mode 0.
- The WDTEN configuration bit can be used for enabling /disabling the watchdog Timer. However if the WDTEN bit is disabled, then the watchdog timer can be enabled / disabled with the SWDTEN instruction.

4.16.1 WDTCON : Watchdog Timer Control Register

- Fig. 4.16.1 shows the Watchdog timer control register.

–	–	–	–	–	–	–	SWDTEN
bit 7							bit 0

bit 7 – bit 1 Unimplemented : Read as '0'

bit 0 : SWDTEN : Software Controlled Watchdog Timer Enable bit

1 = Watchdog Timer is on

0 = Watchdog Timer is turned off if the WDTEN configuration bit in the Configuration register = 0

Fig. 4.16.1 : WDTCON : Watchdog Timer Control Register

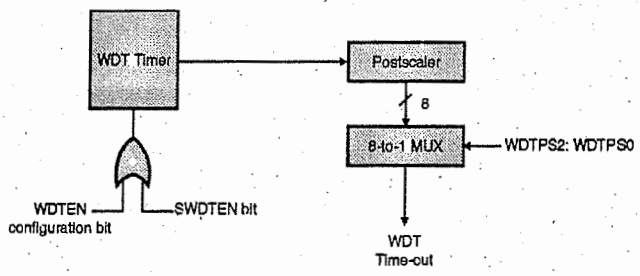


Fig. 4.16.2 : Watchdog timer

Syllabus Topic : Brief Summary of Peripheral Support

4.19 Brief Summary of Peripheral Support

- The different peripherals supported by PIC18 microcontroller are as follows :

1. MSSP module SPI/I2C interface
2. Analog Comparators
3. 10-bit, 8 channel ADC
4. UART for serial communication
5. CCP module
6. 8×8 Hardware Multiplier
7. 4 Timers : Timer 0 , Timer 1 , Timer 2 and Timer 3
8. 5 I/O ports : PORT A , PORT B , PORT C , PORT D and PORT E .
9. Interrupts

- Let us see these peripherals in brief. They are discussed in detail in further chapters.

4.19.1 I/O Ports

- **PIC 18F458** is a 40 pin IC having **5 I/O ports** viz. **Port A, Port B, Port C, Port D** and **Port E.**
- Some of the pins of I/O ports are multiplexed with an alternate function from the peripheral features on the device.
- Whenever a peripheral is enabled that pin cannot be used as a general purpose I/O pin.
- For using the ports as an input or output we need to program the ports.
- Each PORT has three SFRs for its operation. They are :

(i) PORTx (reads the levels on the pins of the device)
(ii) TRISx (data direction register)
(iii) LATx (output latch)

Eg. for Port D we have PORTD, TRISD and LATD SFRs. TRISx register is used for making the port an input port or output port.

- The LATx register is used for read-modify-write operations.
- Table 4.19.1 lists the SFR addresses of ports A-E for PIC18F458.

Table 4.19.1 : SFR addresses of ports A-E for PIC18F458

Port	Address
PORT A	F80H
PORT B	F81H
PORT C	F82H
PORT D	F83H
PORT E	F84H
LATA	F89H
LATB	F8AH
LATC	F8BH
LATD	F8CH
LATE	F8DH
TRISA	F92H
TRISB	F93H
TRISC	F94H
TRISD	F95H
TRISE	F96H

4.19.2 Timers / Counters

Function and use : The PIC18 family has **two to five timers** depending on the family member. The timers are called as Timer 0, Timer 1, Timer 2, Timer 3 and Timer 4. The PIC18F458 supports 4 timers Timer 0, Timer 1, Timer 2 and Timer 3. They can be used as timers in order to generate a time delay or as counters to count events occurring outside the microcontroller.

- When the timer is used as the timer, the PIC18's crystal is used as the frequency source. $\frac{1^{th}}{4}$ of crystal oscillator frequency is given to the timer. When the timer is used as counter, it is a pulse outside the circuit that increments the TMRxH and TMRxL registers. Additional registers used in this mode are T0CON, TMR0H, TMR0L.
- The T0CS (timer 0 clock source) bit in the T0CON register decides the source of clock for the timer. if bit=0 then the timer operates as a timer with pulses from OSC1 and OSC2 pins. If bit =1 then the timer operates like a counter and gets the pulses outside from PIC18.

4.19.4.2 Serial Port Interrupts

- The serial port interrupt is generated because of TXIF or RCIF. One interrupt is used for sending the data byte and the other whenever a data byte is received.
- If the TXIE or RCIE flags in the PIE register are enabled when TXIF or RCIF are 1, the PIC18 microcontroller is interrupted and goes to 0008H location for executing the interrupt service routine.

4.19.4.3 External Hardware Interrupts

- The PIC18 supports **three external hardware interrupts**. They are INT0, INT1 and INT2. These interrupts are located on pins RB0, RB1 and RB2. They are directed to vector location 0008H.
- These interrupts can be enabled by setting the INTxIE bit in the INTCON and INTCON3 registers.
- On reset, the microcontroller configures these interrupts as positive edge triggered interrupts.

4.19.4.4 PORT B-Change Interrupts

- The pins RB_4 – RB_7 of port B can invoke an interrupt whenever any modifications are detected on the respective pin. The interrupt is called **"PORTB-change interrupt"**.
- These interrupts have a single interrupt flag RBIF in the INTCON register. They can be enabled by the RBIE bit in the INTCON register.
- Even though PORT-B change interrupt can use four port B pins, it is considered to be a single interrupt.
- This interrupt is used mainly for keyboard interfacing.

4.19.4.5 Enabling and Disabling an Interrupt

- On reset, all the interrupts are disabled. Even if the interrupts are activated they will not be responded by the microcontroller on reset.
- These interrupts need to activated by software so that the microcontroller can service the interrupts.
- The interrupts can be enabled or disabled by modifying the GIE bit in the INTCON register.

4.19.5 PIC18F458 ADC

- ADC is most commonly used in data acquisition systems. Hence, the PIC microcontrollers have an on-chip ADC.
- The Analog - to - Digital (A/D) converter for PIC18 has following features.

(i) It is a 10 bit ADC.

(ii) Depending on the PIC18 family member, the chip can have 5 to 15 channels. In PIC18F458 there are 8 inputs RA_0-RA_7 of port A. They are used as 8 analog channels.

(iii) The A/D module has four registers. They are :

(a) ADRESH (A/D Result High Register)

(b) ADRESL (A/D Result Low Register)

(c) A/D control register 0 (ADCON0)

(d) A/D control register 1 (ADCON1)

(iv) The ADRESH and ADRESL registers hold the result of the A/D conversion and give a 16 bit output.

However, as the PIC has a 10 bit A/D converter, 6 of the 16 bits will remain unused. The upper 6 or lower 6 bits can be left unused.

(v) The ADCON0 is a A/D control register used for setting the conversion time. It can also be used to select the analog input channel. The ADCON1 is a A/D control register used for setting V_{ref} voltage.

(vi) The analog reference voltage V_{ref} can be selected by using V_{DD} or voltage level on the V_{REF+} or V_{REF-} pins.

$V_{ref} = V_{ref}(+) - V_{ref}(-)$

(vii) The A/D conversion time is decided by the crystal oscillator connected to OSC1 and OSC2 pins of PIC18F458. It has to be greater than 1.6 ms.

Fig. 4.19.2 shows the PIC18 ADC Block Diagram.

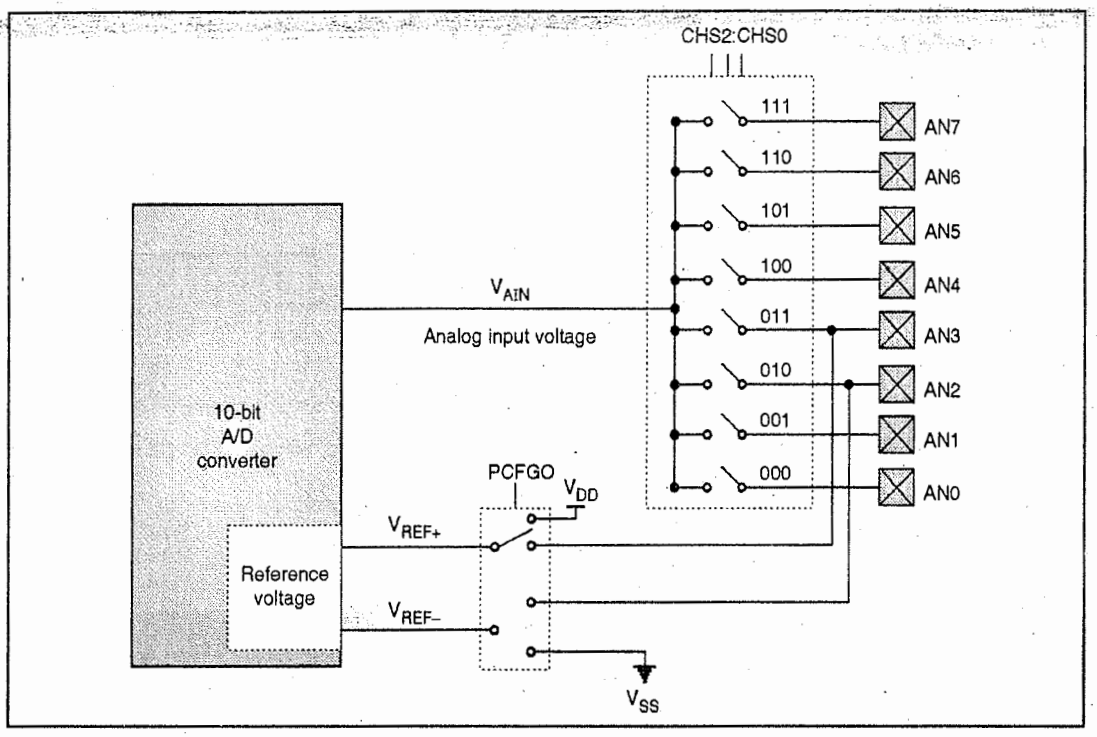


Fig. 4.19.2 : PIC ADC Block Diagram

4.19.6 MSSP with SPI/I2C

- The **Master Synchronous Serial Port (MSSP)** of PIC18 is a serial interface. It is used for communicating the PIC microcontroller with the different peripheral devices like A/D converts, D/A converters, EEPROMS, RTCs, shift registers, display drivers, SD cards, temperature sensors, USB devices etc.
- For data transmission and reception, the MSSP needs a common clock signal for the transmitter and the receiver.
- The MSSP module supports two operating modes. They are :
  (i) Serial Peripheral Interface (SPI)
  (ii) Inter-Integrated Circuit (I2C)
- Both the SPI and I2C are serial interface protocols as studied in chapter 12.
- SPI protocol was developed by Motorola. This serial interfacing method today has become an industry standard because of its easy use and flexibility.
- I2C protocol was developed by Philips. This serial interfacing method supports data transfers at 100 Kbps, 400 Kbps and high speed data transfers at 3.4 Mbps.
- Both the SPI and I2C protocols share the same signal pins. However, both these protocols cannot be active at the same time. The pins used by SPI and I2C MSSP module are :
  (i) Serial Data Clock (SCK) → RC3/SCK/LVDIN
  (ii) Serial Data In (SDI) → RC4/SDI/SDA.
  (iii) Serial Data Out (SDO) → RC5/SDO

4.19.7 Serial Port and USART

- The PIC18FXXX contains the Universal Synchronous Asynchronous Receiver Transmitter (USART) module.
- The USART can be operated in the following modes :
  (i) Asynchronous mode
  (ii) Synchronous mode
- The asynchronous mode is used for communicating with peripheral devices like personal computers, CRT terminals. The synchronous mode is used for communicating with peripheral devices like ADCs, serial EEPROMs.
- The registers that are responsible for serial communication and handling UART are :

(1) SPBRG register (Serial Port Baud Rate Generator)
(2) TXREG (Transfer register)
(3) RCREG (Receive register)
(4) TXSTA (Transmit Status and Control Register)
(5) RCSTA (Receive Status and Control Register)
(6) PIR1 (Peripheral Interrupt Request Register PIR1)

4.19.7.1 USART Asynchronous Mode

- In the asynchronous mode, the PIC18UART sends 8 bits or 9 bits along with start and stop bits. It uses a non return-to-zero (NRZ) format.
- For achieving different baud rates there is an on-chip dedicated 8-bit baud rate generator.
- The UART is responsible for serially transmitting and receiving the data.
- Both the UART transmitter and receiver sections are functionally independent. However they have same baud rates and employ the same data format.
- By clearing the SYNC bit in the TXSTA register we can select the UART asynchronous mode.
- Depending on the BRGH bit in the TXSTA register, the baud rate generator is responsible for producing a clock of either x16 or x64 of the bit shift rate.
- The UART does not support parity .However a 9th bit can be added as parity using software.
- Asynchronous mode is stopped during SLEEP.
- The USART Asynchronous Mode comprises of the following :
  1. Sampling circuit
  2. Baud Rate Generator
  3. Asynchronous Transmitter
  4. Asynchronous Receiver
- In section 12.13 we have seen how the baud rate generator is used to generate different baud rates.
- The sampling circuit samples data on the RC7/RX pin three times to check the signal level present at the RX pin i.e. high or low.

❑❑❑

UNIT - IV

CHAPTER 5

PIC18F458 Instruction Set

5.1 PIC18F458 Addressing Modes

- When the microcontroller executes an instruction, it performs a specific function on the data. The data is stored at some source location. This data must be moved or copied to a destination location. The methods by which these address locations are specified are called as **addressing modes.**
- The PIC18 microcontroller supports **4 addressing modes.** They are :

(i) Immediate addressing mode
(ii) Direct addressing mode
(iii) Register indirect addressing mode
(iv) Indexed ROM addressing mode

5.1.1 Immediate Addressing Mode

- **Immediate addressing mode** is the simplest addressing mode to get data in this addressing mode the source operand is a constant rather than a variable.
- When the instruction is assembled, the operand comes after the opcode. The operand is called as a **literal. The letter L indicates immediate.**
- The immediate addressing mode is used to load data into the PIC registers and WREG register. However, it cannot be used to load data into any of the file register.

Example :

1.	MOVLW 0x40	Load 40H into WREG register
2.	ANDLW 50H	And 50H with WREG
3.	IORLW 20H	Logically OR 20H with the WREG register and store result in WREG register.

5.1.2 Direct Addressing Mode

- In the direct addressing mode, the 8 bit data is in a RAM memory location whose address is specified in the instruction.

Use : The direct addressing mode is used for accessing the RAM file register.

- The letter **"F"** used in the instruction indicates the address of file register location.

Example :

1.	MOVLW 0x23 MOVWF 0x10	WREG = 23H This instruction will copy the contents of WREG register i.e. 23H to the file register RAM location 10H.
2.	MOVFF 0x30, PORTC	This instruction will copy the contents of memory location 30H to PORTC.
3.	MOVFF PORTB, PORTC	This instruction will copy the contents of PORTB to PORTC.

5.1.3 Register Indirect Addressing Mode

Use : The register indirect addressing mode is used for accessing data stored in the RAM part of the file register.

- In register indirect addressing mode a register is used as a pointer to the ROM location of the file register.
- For register indirect addressing mode three **file select registers** (FSRs) are used. They are **FSR0, FSR1** and **FSR2.**
- The FSR is a 12 bit register. It covers the complete 4KB RAM space of the PIC18.
- PIC18 is a 8 bit microcontroller. Hence, FSR registers are split into 8 bit so that they can be placed in the SFR space.
- The FSRs have low byte parts (FSRxL) and high byte parts (FSRxH) e.g. FSR0L and FSR0H are low and high byte parts of FSR0. Only lower 4 bits of FSRxH are used.

- Each FSR register is associated with an indirect register INDFx. These registers are INDF0, INDF1 and INDF2.

Example :

(i) LFSR 1,0x55 Load FSR1 with 55H.

(ii) MOVWF INDF1 Copy the contents of WREG to RAM location FSR1 points to.

- The advantage of register indirect addressing mode is that it makes the addressing dynamic.

5.1.4 Indexed ROM Addressing Mode

Use : Indexed ROM addressing mode is used for accessing the data from look-up tables that reside in the PIC18 program ROM.

Syllabus Topic : Overview of Instruction Set

5.2 Overview of Instruction Set

- The PIC18 instruction set adds many enhancements to the previous PIC micro instruction sets, while maintaining an easy migration from these PIC micro instruction sets.
- Most instructions are a single program memory word (16 bits) but there are three instructions that require two program memory locations.
- Each single-word instruction is a 16-bit word divided into an opcode, which specifies the instruction type and one or more operands, which further specify the operation of the instruction.
- The instruction set is highly orthogonal and is grouped into four basic categories :

(a) Byte-oriented operations
(b) Bit-oriented operations
(c) Literal operations
(d) Control operations

- The PIC18 instruction set summary in Table 5.2.2 lists **byte-oriented**, **bit-oriented**, **literal** and **control** operations. Table 5.2.1 shows the opcode field descriptions.
- Most **byte-oriented** instructions have three operands :
  1. The file register (specified by 'f')
  2. The destination of the result (specified by 'd')
  3. The accessed memory (specified by 'a')
- The file register designator 'f' specifies which file register is to be used by the instruction.
- The destination designator 'd' specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the WREG register. If 'd' is one, the result is placed in the file register specified in the instruction.
- All **bit-oriented** instructions have three operands :
  1. The file register (specified by 'f')
  2. The bit in the file register (specified by 'b')
  3. The accessed memory (specified by 'a')
- The bit field designator 'b' selects the number of the bit affected by the operation, while the file register designator 'f' represents the number of the file in which the bit is located.
- The **literal** instructions may use some of the following operands :
  (a) A literal value to be loaded into a file register (specified by 'k')
  (b) The desired FSR register to load the literal value into (specified by 'f')
  (c) No operand required (specified by '—')
- The **control** instructions may use some of the following operands :
  (a) A program memory address (specified by 'n')
  (b) The mode of the CALL or RETURN instructions (specified by 's')
  (c) The mode of the table read and table write instructions (specified by 'm')
  (d) No operand required (specified by '—')
- All instructions are a single word, except for three double-word instructions. These three instructions were made double-word instructions so that all the required information is available in these 32 bits. In the second word, the 4 MSBs are '1's. If this second word is executed as an instruction (by itself), it will execute as a NOP.
- All single-word instructions are executed in a single instruction cycle, unless a conditional test is true or the program counter is changed as a result of the instruction. In these cases, the

execution takes two instruction cycles, with the additional instruction cycle(s) executed as a NOP.

- The double-word instructions execute in two instruction cycles. One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 μs.
- If a conditional test is true, or the program counter is changed as a result of an instruction, the instruction execution time is 2 μs. Two-word branch instructions (if true) would take 3 μs.

Table 5.2.1 : Opcode Field Descriptions

Field	Description
A	RAM access bit : a = 0 : RAM location in Access RAM (BSR register is ignored) a = 1 : RAM is specified by BSR register
bbb	Bit address within an 8-bit register (0 to 7)
BSR	Bank Select Register. Used to select the current RAM bank
d	Destination select bit : d = 0 : store result in WREG d = 0 : store result in file register f
destination	Destination either the WREG register or the specified register file location
f	8-bit register file address (0x00 to 0xFF)
f_s	12-bit register file address (0x000 to 0xFFF). This is the source address
f_d	12-bit register file address (0x000 to 0xFFF). This is the destination address
k	Literal field, constant data or label (may be either an 8-bit, 12-bit or a 20-bit value)
label	Label name.
PRODH	Product of Multiply High Byte
PRODL	Product of Multiple Low Byte.
s	Fast call / return mode select bit : s = 0 : do not update into / from shadow registers s = 1 certain register loaded into / from shadow registers (Fast mood)
u	Unused or unchanged.

Field	Description
WREG	Working register (accumulator).
x	Don't care (0 or 1). The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all microchip software tools.
TBLPTR	21-bit Table pointer (points to a program memory location).
TABLAT	8-bit Table Latch
TOS	Top of Stack
PC	Program Counter
PCL	Program Counter Low Byte
PCH	Program Counter High Byte
PCLATH	Program Counter High Byte Latch
PCLATU	Program Counter Upper Byte Latch
GIE	Global Interrupt Enable bit
WDT	Watchdog Timer
$\overline{TO}$	Time-out bit.
$\overline{PD}$	Power Down bit.
C, DC, Z, OV, N	ALU status bits : Carry, Digit Carry, Zero, Overflow, Negative
[ ]	Optional
( )	Contents
→	Assigned to
< >	Register bit field
∈	In the set of
Italics	User defined term (font is courier).

Table 5.2.2 : PIC18FXXX Instruction Set

Mnemonic, Operands		Description	Cycles	16-bit Instruction MSB		LSB		Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d, a	Add WREG and f	1	0010	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
ADDWFC	f, d, a	Add WREG and Carry bit to f	1	0010	00da	ffff	ffff	C, DC, Z, OV, N	1, 2
ANDWF	f, d, a	AND WREG with f	1	0001	01da	ffff	ffff	Z, N	1, 2
CLRF	f, a	Clear f	1	0110	101a	ffff	ffff	Z	2
COMF	f, d, a	Complement f	1	0001	11da	ffff	ffff	Z, N	1, 2
CPFSEQ	f, a	Compare f with WREG, skip =	1 (2 or 3)	0110	001a	ffff	ffff	None	4
CPFSGT	f, a	Compare f with WREG skip >	1 (2 or 3)	0110	010a	ffff	ffff	None	4
CPFSLT	f, a	Compare f with WREG, skip <	1 (2 or 3)	0110	000a	ffff	ffff	None	1, 2
DECF	f, d, a	Decrement f	1	0000	01da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
DECFSZ	f, d, a	Decrement f, skip if 0	1 (2 or 3)	0010	11da	ffff	ffff	None	1, 2, 3, 4
DCFSNZ	f, d, a	Decrement f, skip if not 0	1 (2 or 3)	0100	11da	ffff	ffff	None	1, 2
INCF	f, d, a	Increment f	1	0010	10da	ffff	ffff	C, DC, Z, OV, N	1, 2, 3, 4
INCFSZ	f, d, a	Increment f, skip if 0	1 (2 or 3)	0011	11da	ffff	ffff	None	4
INFSNZ	f, d, a	Increment f, skip if Not 0	1 (2 or3)	0100	10da	ffff	ffff	None	1, 2
IORWF	f, d, a	Inclusive OR WREG with f	1	0001	00da	ffff	ffff	Z, N	1, 2
MOVF	f, d, a	Move f	1	0101	00da	ffff	ffff	Z, N	1
MOVFF	f_s, f_d	Move f_s (source) to 1st word f_d (destination). 2nd word	2	1100 1111	ffff ffff	ffff ffff	ffff ffff	None	
MOVWF	f, a	Move WREG to f	1	0110	111a	ffff	ffff	None	
MULWF	f, a	Multiply WREG with f	1	0000	001a	ffff	ffff	None	
NEGF	f, a	Negate f	1	0110	110a	ffff	ffff	C, DC, Z, OV, N	1, 2
RLCF	f, d, a	Rotate left f through carry	1	0011	01da	ffff	ffff	C, Z, N	
RLNCF	f, d, a	Rotate left f (No carry)	1	0100	01da	ffff	ffff	Z, N	1, 2
RRCF	f, d, a	Rotate right f through carry	1	0011	00da	ffff	ffff	C, Z, N	
RRNCF	f, d, a	Rotate right f (No carry)	1	0100	00da	ffff	ffff	Z, N	
SETF	f, a	Set f	1	0110	100a	ffff	ffff	None	
SUBFWB	f, d, a	Subtract f from WRGE with borrow	1	0101	01da	ffff	ffff	C, DC, Z, OV, N	1, 2
SUBWF	f, d, a	Subtract WREG from f	1	0101	11da	ffff	ffff	C, DC, Z, OV, N	
SUBWFB	f, d, a	Subtract WREG from f with borrow	1	0101	10da	ffff	ffff	C, DC, Z, OV, N	1, 2
SWAPF	f, d, a	Swap nibbles in f	1	0011	10da	ffff	ffff	None	4
TSTFSZ	f, a	Test f, skip if 0	1(2 or 3)	0110	011a	ffff	ffff	None	1, 2
XORWF	f, d, a	Exclusive OR WREG with f	1	0001	10da	ffff	ffff	Z, N	
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b, a	Bit Clear f	1	1001	bbba	ffff	ffff	None	1, 2
BSF	f, b, a	Bit Set f	1	1000	bbba	ffff	ffff	None	1, 2
BTFSC	f, b, a	Bit Test f, Skip if Clear	1(2 or 3)	1011	bbba	ffff	ffff	None	3, 4
BTFSS	f, b, a	Bit Test f, Skip if Set	1(2 or 3)	1010	bbba	ffff	ffff	None	3, 4
BTG	f, d, a	Bit Toggle f	1	0111	bbba	ffff	ffff	None	1, 2
CONTROL OPERATIONS									
BC	n	Branch if Carry	1 (2)	1110	0010	nnnn	nnnn	None	
BN	n	Branch if Negative	1 (2)	1110	0110	nnnn	nnnn	None	
BNC	n	Branch if Not Carry	1 (2)	1110	0011	nnnn	nnnn	None	

Mnemonic, Operands		Description	Cycles	16-bit Instruction MSB		LSB		Status Affected	Notes
BNN	n	Branch if Not Negative	1 (2)	1110	0111	nnnn	nnnn	None	
BNOV	n	Branch if Not Overflow	1 (2)	1110	0101	nnnn	nnnn	None	
BNZ	n	Branch if Not Zero	2	1110	0001	nnnn	nnnn	None	
BOV	n	Branch if Overflow	1 (2)	1110	0100	nnnn	nnnn	None	
BRA	n	Branch Unconditionally	1 (2)	1101	0nnn	nnnn	nnnn	None	
BZ	n	Branch if Zero	1 (2)	1110	0000	nnnn	nnnn	None	
CALL	n, s	Call Subroutine 1st word 2nd word	2	1110 1111	110s kkkk	kkkk kkkk	kkkk kkkk	None	
CLRWDT	–	Clear Watchdog Timer	1	0000	0000	0000	0100	$\overline{TO}$, $\overline{PD}$	
DAW	–	Decimal Adjust WREG	1	0000	0000	0000	0111	C	
GOTO	n	Go to address 1st word 2nd Word	2	1110 1111	1111 kkkk	kkkk kkkk	kkkk kkkk	None	
NOP	–	No Operation	1	0000	0000	0000	0000	None	
NOP	–	No Operation	1	1111	xxxx	xxxx	xxxx	None	4
POP	–	Pop top of return stack (TOS)	1	0000	0000	0000	0110	None	
PUSH	–	Push top of return stack (TOS)	1	0000	0000	0000	0101	None	
RCALL	n	Relative Call	2	1101	1nnn	nnnn	nnnn	None	
RESET		Software device Reset	1	0000	0000	1111	1111	All	
RETFIE	s	Return from interrupt enable	2	0000	0000	0001	000s	GIE/GIEH, PEIE/GIEL	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
RETURN	s	Return from subroutine	2	0000	0000	0001	001s	None	
SLEEP	–	Go into Standby mode	1	0000	0000	0000	0011	$\overline{TO}$, $\overline{PD}$	
LITERAL OPERATIONS									
ADDLW	k	Add literal and WREG	1	0000	1111	kkkk	kkkk	C, DC, Z, OV, N	
ANDLW	k	AND literal with WREG	1	0000	1011	kkkk	kkkk	Z, N	
IORLW	k	Inclusive OR literal with WREG	1	0000	1001	kkkk	kkkk	Z, N	
LFSR	f, k	Move literal (12-bit) 2nd word to FSRx 1st word	2	1110 1111	1110 0000	00ff kkkk	kkkk kkkk	None	
MOVLB	k	Move literal to BSR<3:0>	1	0000	0001	0000	kkkk	None	
MOVLW	k	Move literal to WREG	1	0000	1110	kkkk	kkkk	None	
MULLW	k	Multiply literal with WREG	1	0000	1101	kkkk	kkkk	None	
RETLW	k	Return with literal in WREG	2	0000	1100	kkkk	kkkk	None	
SUBLW	k	Subtract WREG from literal	1	0000	1000	kkkk	kkkk	C, DC, Z, OV, N	
XORLW	k	Exclusive OR literal with WREG	1	0000	1010	kkkk	kkkk	Z, N	
DATA MEMORY ↔ PROGRAM MEMORY OPERATIONS									
TBLRD*		Table Read	2	0000	0000	0000	1000	None	
TBLRD*+		Table Read with post-increment		0000	0000	0000	1001	None	
TBLRD*-		Table Read with post-decrement		0000	0000	0000	1010	None	
TBLRD+*		Table Read with pre-increment		0000	0000	0000	1011	None	
TBLWT*		Table Write	2 (5)	0000	0000	0000	1100	None	5
TBLWT*+		Table Write with post-increment		0000	0000	0000	1101	None	5
TBLWT*-		Table Write with post-decrement		0000	0000	0000	1110	None	5
TBLWT+*		Table Write with pre-increment		0000	0000	0000	1111	None	5

5.3 Instruction Descriptions

5.3.1 ADDLW

Mnemonic	ADDLW k		**Function**	ADD Literal to WREG.
Operands	0 <k < 255		**Clock Cycles**	1
Words	1		**Algorithm**	(WREG) + k → WREG
Addr. Mode	Immediate Addressing Mode.		**Flags**	N, OV, C, DC, Z flags are affected.

Operation	WREG + k → WREG	– This instruction adds the contents of the WREG register with the 8-bit literal 'k' and the result is placed in WREG register. – It is used for signed and unsigned numbers.
Example	MOVLW 0x20 ADDLW 0x25	WREG = 20 H – WREG = 45 H (20 H + 25 H) This instruction adds the contents of the WREG register i.e. 20 H with the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register.

5.3.2 ADDWF

Mnemonic	ADDWF f,d,a		**Function**	ADD WREG and file register.
Flags	N, OV, C, DC, Z flags are affected.		**Clock Cycles**	1
Words	1		**Algorithm**	(W) + (f) → destination

Operation	(W) + (f) → destination	– This instruction adds the contents of the WREG register with the contents of the file register and the result is placed in WREG register if d=0 or file register if d=1. – It is used for signed and unsigned numbers.
Example	MYREG SET 0x10 MOVLW 0x25 MOVWF MYREG MOVLW 0x20 ADDWFMYREG	– Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H WREG = 20 H WREG = 45 H (20 H + 25 H) This instruction adds the contents of the WREG register i.e. 20 H with contents of file register i.e. 25 H and the result is placed in WREG register.

5.3.3 ADDWFC

Mnemonic	ADDWFC f,d,a		**Function**	ADD WREG and carry bit to the file register.
Clock Cycles	1		**Flags**	N, OV, C, DC, Z flags are affected.
Words	1		**Algorithm**	(W) + (f) + (CY) → destination

Operation	(W)+ (f) + (CY) → destination	– This instruction adds the contents of the WREG register and carry flag with the contents of the file register and result is placed in destination location.
Example	MYREG SET 0x10 MOVLW 0x25 MOVWF MYREG BCF STATUS , C MOVLW 0x20 ADDWFC MYREG,F	Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H Carry = 0 WREG = 20 H MYREG = 45 H (20 H + 25 H+ 0) This instruction adds the contents of the WREG register i.e. 20 H with carry i.e. 0 and contents of file register i.e. 25 H and the result is placed in MYREG.

5.3.4 ANDLW

Mnemonic	ANDLW k		**Function**	AND Literal with WREG.
Operands	0 <k < 255		**Clock Cycles**	1
Words	1		**Algorithm**	(WREG) AND k → WREG
Addr. Mode	Immediate Addressing Mode.		**Flags**	N, Z flags are affected.

Operation	(WREG) AND k → WREG	– This instruction ANDs the contents of the WREG register with the 8-bit literal 'k' and the result is placed in WREG register.
Example	MOVLW 0x20 ANDLW 0x25	WREG = 20 H WREG = 20 H 0010 0000 25H 0010 0101 WREG = 20 H 0010 0000 This instruction ANDs the contents of the WREG register i.e. 20 H with the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register.

5.3.5 ANDWF

Mnemonic	ANDWF f,d,a	**Function**	AND WREG and file register.
Flags	N and Z flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(W) AND (f) → destination

Operation	(W) AND (f) → destination	This instruction ANDs the contents of the WREG register with the contents of the file register and the result is placed in WREG register if d=0 or file register if d=1 .
Example	MYREG SET 0x10 MOVLW 0x25 MOVWF MYREG MOVLW 0x20 ANDWF MYREG ,W	– Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H WREG = 20 H WREG = 20 H 0010 0000 MYREG = 25H 0010 0101 WREG = 20 H 0010 0000 – This instruction ANDs the contents of the WREG register i.e. 20 H with the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register.

5.3.6 BC

Mnemonic	BC target address	**Function**	Branch if carry = 1.
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if CY=1

Operation	Jump to target address if CY = 1	– This instruction branches to the target address if the carry flag bit =1. – If the instruction branches to the target address it becomes a two cycle instruction.
Example	MOVLW 0x25 ADDLW 0x05 BC L1	– WREG = 25 H – Add 05 to WREG – If CY =1 branch to L1

5.3.7 BCF

Mnemonic	BCF f, b, a	**Function**	Bit Clear File Register
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	0 → f <b>

Operation	0 → f <b>	This instruction clears a bit of the file register .The file register bit indicated may be directly addressable.
Example	1) BCF PORTC , 5	This instruction will clear the bit 5 of PORTC.
	2) BCF MYREG , 2	This instruction will clear the B2 bit of MYREG.

5.3.8 BN

Mnemonic	BN target address	**Function**	Branch if Negative.
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if N=1

Operation	Jump to target address if N=1	– This instruction branches to the target address if the carry flag bit =1. – If the instruction branches to the target address it becomes a two cycle instruction. – This instruction is used for addition of signed numbers.

5.3.9 BNC

Mnemonic	BNC target address	**Function**	Branch if No Carry
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if CY=0

Operation	Jump to target address if CY=0	– This instruction branches to the target address if the carry flag bit =0. – If the instruction branches to the target address it becomes a two cycle instruction.
Example	MOVLW 0x25 ADDLW 0x05 BNC L1	WREG = 25 H Add 05 to WREG If CY =0 branch to L1

5.3.10 BNN

Mnemonic	BNN target address	**Function**	Branch if Not Negative
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if N=0

Operation	Jump to target address if N = 0	– This instruction branches to the target address if the negative flag bit = 0. – If the instruction branches to the target address it becomes a two cycle instruction. – This instruction is used for addition of signed numbers.

5.3.11 BNOV

Mnemonic	BNOV target address	**Function**	Branch if NO Overflow
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if OV=0

Operation	Jump to target address if OV = 0	– This instruction branches to the target address if the overflow flag bit =0. – If the instruction branches to the target address it becomes a two cycle instruction. – This instruction is used for addition of signed numbers.

5.3.12 BNZ

Mnemonic	BNZ target address	**Function**	Branch if Not Zero
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if Z=0

Operation	Jump to target address if Z=0	– This instruction branches to the target address if the zero flag bit =0. – If the instruction branches to the target address it becomes a two cycle instruction.
Example	LOC EQU 0x30 MOVF LOC , F BNZ L1	 – Copy LOC to itself – Branch if LOC is not zero

5.3.13 BOV

Mnemonic	BOV target address	**Function**	Branch if Overflow
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if OV=1

Operation	Jump to target address if OV=1	– This instruction branches to the target address if the overflow flag bit = 1. – If the instruction branches to the target address it becomes a two cycle instruction. – This instruction is used for addition of signed numbers.

5.3.14 BRA

Mnemonic	BRA target address	**Function**	Branch Unconditional
Flags	No flags are affected.	**Clock Cycles**	2
Words	1	**Algorithm**	Branch Unconditionally

Operation	Branch Unconditionally	– This instruction transfers program control to the target address unconditionally.
Example	BSF PORTC , 5 L1: BTFSC PORTC, 5 BRA L1	– This instruction will set the bit 5 of PORTC. – Skip if RC5 = 0 – Branch unconditionally

5.3.15 BSF

Mnemonic	BSF f, b, a	**Function**	Bit Set File Register
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	1 → f <b>

Operation	1 → f <b>	This instruction sets a bit of the file register The file register bit indicated may be directly addressable.
Example	1) BSF PORTC, 5	This instruction will set the bit 5 of PORTC.
	2) BSF MYREG, 2	This instruction will set the B2 bit of MYREG.

5.3.16 BTFSC

Mnemonic	BTFSC f, b, a	**Function**	Bit TEST file Register, skip if clear
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	Skip if f<b> = 0

Operation	Skip if f <b> = 0	– This instruction tests a bit of the file register. If the file register bit is zero then it skip the next instruction. – The file register bit indicated may be directly addressable.
Example	BSF TRISC , 2 L1: BTFSC PORTC,2 BRA L1	– This instruction will set the bit 2 of PORTC. – If RC2 = 0, skip the next instruction.

5.3.17 BTFSS

Mnemonic	BTFSS f, b, a	**Function**	Bit TEST file Register , skip if Set
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	Skip if f <b> = 1

Operation	Skip if f <b> = 1	– This instruction tests a bit of the file register. If the file register bit is Set then skip to the next instruction. – The file register bit indicated may be directly addressable.
Example	BSF TRISC , 2 L1: BTFSS PORTC,2 BRA L1	– This instruction will set the bit 2 of PORTC. – If RC2=1, skip the next instruction.

5.3.18 BTG

Mnemonic	BTG f, b, a	**Function**	Bit Toggle file Register
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	$(\overline{f<b>}) \rightarrow f<b>$

Operation	$(\overline{f<b>}) \rightarrow f<b>$	– This instruction toggles a bit of the file register. – The file register bit indicated may be directly addressable.
Example	BSF TRISC , 2 BTG PORTC,2	– This instruction will set the bit 2 of PORTC. – Toggle bit RC2

5.3.19 BZ

Mnemonic	BZ target address	**Function**	Branch if Zero
Flags	No flags are affected.	**Clock Cycles**	1 (2)
Words	1	**Algorithm**	Jump to target address if Z=1

Operation	Jump to target address if Z=1	– This instruction branches to the target address if the zero flag bit =1. – If the instruction branches to the target address it becomes a two cycle instruction.
Example	LOC EQU 0x30 MOVF LOC , F BZ L1	– Copy LOC to itself – Branch if LOC is zero

5.3.20 CALL

Mnemonic	CALL k ,s	**Function**	Transfer Control to a Subroutine
Flags	No flags are affected.	**Clock Cycles**	2
Words	2	**Algorithm**	_____

Operation	– This instruction is used to transfer control to a subroutine whose address is specified in the instruction. – It is a 4 byte instruction such that first 12 bytes are given for opcode and remaining bits for the target address. – The steps to be followed when the program transfers control to the subroutine are as follows : **Step 1 :** The contents of PC are pushed onto the stack. SP = SP + 1 **Step 2 :** PC is loaded with new address and control is transferred to the subroutine. **Step 3 :** Return instruction is executed at the end of subroutine. **Step 4 :** Then the contents of PC are popped off the stack. **Step 5 :** The program then transfers control to the next instruction after CALL instruction.
Example	CALL DELAY

5.3.21 CLRF

Mnemonic	CLRF f, a	Function	Clear File Register
Flags	Z flag is affected.	Clock Cycles	1
Words	1	Algorithm	00 → f

Operation	00 → f	– This instruction clears the complete byte in file register.
Example	CLRF TMR0H	– This instruction clears the timer 0 high register.

5.3.22 CLRWDT

Mnemonic	CLRWDT	Function	Clear Watchdog Timer
Flags	Z flag is affected.	Clock Cycles	1
Words	1	Algorithm	00 → WDT

Operation	00 → WDT	– This instruction clears the Watchdog Timer.
Example	CLRWDT	– This instruction clears the Watchdog Timer.

5.3.23 COMF

Mnemonic	COMF f,d,a	Function	Complement the File Register
Flags	N and Z flags are affected.	Clock Cycles	1
Words	1	Algorithm	$(\bar{f}) \rightarrow$ destination

Operation	$(\bar{f}) \rightarrow$ destination	– This instruction complements the complete byte in file register. The result is placed in WREG register if d=0 or file register if d=1.
Example	MOVLW 0xFF MOVWF PORTD COMF PORTD , W	– WREG = FF H – PORT D = FF H – TOGGLE or complement PORT D, WREG = 00 H

5.3.24 CPFSEQ

Mnemonic	CPFSEQ f, a	Function	Compare the File Register with WREG and skip if Equal (F= W)
Flags	No flags are affected.	Clock Cycles	1 (2)
Words	1	Algorithm	Skip next instruction if f= W

Operation	Skip next instruction if f = W	– This instruction compares the contents of file register with the contents of WREG register. If the contents are equal the next instruction is skipped.
Example	SETF TRISC MOVLW 0xFF L1 : CPSEQ PORTC BRA L1	– Make PORT C an input port – WREG = FF H – Skip next instruction if PORT C = FF H – Keep checking

5.3.25 CPFSGT

Mnemonic	CPFSGT f, a	Function	Compare the File Register with WREG and skip if Equal (F> W)
Flags	No flags are affected.	Clock Cycles	1 (2)
Words	1	Algorithm	Skip next instruction if f> W

Operation	Skip next instruction if f> W	– This instruction compares the contents of file register with the contents of WREG register. If the contents of file register are greater than contents of WREG register then the next instruction is skipped.
Example	SETF TRISC MOVLW 0x52 L1 : CPFSGT PORTC BRA L1	– Make PORT C an input port – WREG = 52 H – Skip next instruction if PORT C > 52 H – Keep checking

5.3.26 CPFSLT

Mnemonic	CPFSLT f, a	Function	Compare the File Register with WREG and skip if Equal (F < W)
Flags	No flags are affected.	Clock Cycles	1 (2)
Words	1	Algorithm	Skip next instruction if f< W

Operation	Skip next instruction if f< W	– This instruction compares the contents of file register with the contents of WREG register. If the contents of file register are less than contents of WREG register then the next instruction is skipped.

Example	SETF TRISC MOVLW 0x52 L1 : CPFSLT PORTC BRA L1	– Make PORT C an input port – WREG = 52 H – Skip next instruction if PORT C < 52 H – Keep checking

5.3.27 DAW

Mnemonic	DAW	**Function**	Decimal Adjust WREG after addition
Flags	CY flag is affected.	**Clock Cycles**	1
Words	1	**Algorithm**	Contents of WREG are BCD if [($WREG_{3-0}$) > 9] or [DC = 1] then ($WREG_{3-0}$) = ($WREG_{3-0}$) + 6 AND if [($WREG_{7-4}$) > 9] or [CY = 1] then ($WREG_{7-4}$) = ($WREG_{7-4}$) + 6

Operation	DAW	– This instruction adjusts the sum of two packed BCD numbers to an eight bit value i.e. producing two four bit digits. – To perform the addition any ADDLW, ADDWF or ADDWFC instruction can be used. – The rules of BCD addition are (i) if the number is greater than 9, add 6 (ii) if the DC =1 or CY =1, add 6 to the upper or lower nibble . – This is done in order to produce a valid BCD result.
Example	MOVLW 0x57 ADDLW 0x67 DAW	– WREG = 57 H – WREG = 1011 1110 (invalid BCD) – WREG = 24 H with CY=1 (valid BCD) 57H 0101 0111 + 67H 0110 0111 BE 1011 1110 (invalid BCD) + DAW 66 0110 0110 24H 0010 0100 with CY = 1 **(valid BCD Result)**

5.3.28 DECF

Mnemonic	DECF f,d,a	**Function**	Decrement the File Register
Flags	N, OV, C, DC, Z flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(f) - 1→ destination

Operation	(f) - 1→ destination	– This instruction is used to subtract 1 from the contents of the file register. The result is placed in WREG register if d=0 or file register if d=1.
Example	Count SET 0x20 MOVLW 0xD4 MOVWF Count DECF Count, F	Set location 20 H for count WREG = D4 H Count = D4 H Count =D3 H, WREG = D4 H

5.3.29 DECFSZ

Mnemonic	DECFSZ f,d,a	**Function**	Decrement the File Register and skip if zero
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(f) – 1 → destination , skip the next instruction if result = 0

Operation	(f) - 1→ destination, skip the next instruction if result = 0	This instruction is used to subtract 1 from the contents of the file register and skips the next instruction if the result is zero.
Example	Count SET 0x20 MOVLW 0x10 MOVWF Count MOVLW 0X55 MOVWF PORTD COMF PORTD,F L1 : DECFSZ Count, F BRA L1	Set location 20 H for count WREG = 10 H Count = D4 H WREG = 55 H PORTD = 55 H Toggle PORT D Decrement count and skip if zero

5.3.30 DECFSNZ

Mnemonic	DECFSNZ f,d,a	**Function**	Decrement the File Register and skip if not zero
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(f) - 1→ destination, skip the next instruction if result ≠ 0

Operation	(f) - 1→ destination, skip the next instruction if result ≠ 0	– This instruction is used to subtract 1 from the contents of the file register and skips the next instruction if the result is not zero.
Example	Count SET 0x20 L1 : MOVLW 0x10 MOVWF Count MOVLW 0X55 MOVWF PORTD COMF PORTD,F L2 : DECFSNZ Count , F BRA L1 BRA L2	Set location 20 H for count WREG = 10 H Count = D4 H WREG = 55 H PORTD = 55 H Toggle PORT D Decrement count and skip if not zero

5.3.31 GOTO

Mnemonic	GOTO k	**Function**	Unconditional Branch
Flags	No flags are affected.	**Clock Cycles**	2
Words	2	**Algorithm**	_____

Operation	– There are two types of unconditional jumps. They are : **BRA** : we have seen this instruction in section 5.3.14. It is a short jump. **GOTO** : This instruction is used for unconditional branching or unconditional jumps. It is a long jump. – It is a 4 byte instruction such that first 12 bytes are given for opcode and remaining bits for the target address. – Its drawback is that it needs 4 bytes. Hence BRA instruction is commonly used in programs as it needs only 2 bytes .
Example	GOTO MYREG

5.3.32 INCF

Mnemonic	INCF f,d,a	**Function**	Increment the File Register
Flags	N, OV, C, DC, Z flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(f) + 1→ destination

Operation	(f) + 1→ destination	– This instruction is used to Increment 1 to the contents of the file register. The result is placed in WREG register if d=0 or file register if d =1.
Example	Count SET 0x20 CLRF TRISD MOVLW 0xD4 MOVWF Count INCF Count , F	Set location 20 H for count WREG = D4 H Count = D4 H Count =D5 H , WREG = D4 H

5.3.33 INCFSZ

Mnemonic	INCFSZ f,d,a	**Function**	Increment the File Register and skip if zero.
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(f) + 1→ destination , skip the next instruction if result = 0.

Operation	(f) + 1→ destination, skip the next instruction if result = 0	– This instruction is used to increment 1 to the contents of the file register and skips the next instruction if the result is zero.
Example	Count SET 0x20 CLRF TRISD MOVLW 0x10 MOVWF Count MOVLW 0X55 MOVWF PORTD COMF PORTD,F L1 : INCFSZ Count , F BRA L1	Set location 20 H for count WREG = 10 H Count = D4 H WREG = 55 H PORTD = 55 H Toggle PORT D Increment count and skip if zero

5.3.34 INCFSNZ

Mnemonic	INCFSNZ f,d,a	**Function**	Increment the File Register and skip if not zero
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	(f) + 1→ destination, skip the next instruction if result ≠ 0

Operation	(f) + 1→ destination , skip the next instruction if result ≠ 0	– This instruction is used to increment 1 to the contents of the file register and skips the next instruction if the result is not zero.
Example	Count SET 0x20 CLRF TRISD L1: MOVLW 0x10 MOVWF Count MOVLW 0X55 MOVWF PORTD COMF PORTD,F L2 : INCFSNZ Count, F BRA L1 BRA L2	Set location 20 H for count WREG = 10 H Count = D4 H WREG = 55 H PORTD = 55 H Toggle PORT D Increment count and skip next instruction if not zero

5.3.35 IORLW

Mnemonic	IORLW k
Operands	0 <k < 255
Words	1
Addr. Mode	Immediate Addressing Mode.

Function	OR Literal with WREG.
Clock Cycles	1
Algorithm	(WREG) OR k→WREG
Flags	N and Z flags are affected.

Operation	(WREG) OR k →WREG	This instruction ORs the contents of the WREG register with the 8-bit literal 'k' and the result is placed in WREG register.
Example	MOVLW 0x20 IORLW 0x25	WREG = 20 H WREG = 25 H This instruction will logically OR the contents of the WREG register i.e. 20 H with the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register. 20 H 0010 0000 25 H 0010 0101 25 H 0010 0101

5.3.36 IORWF

Mnemonic	IORWF f,d,a
Flags	N and Z flags are affected.
Words	1

Function	OR WREG and file register.
Clock Cycles	1
Algorithm	(W) OR (f) → destination

Operation	(W) OR (f) → destination	This instruction logically ORs the contents of the WREG register with the contents of the file register and the result is placed in WREG register if d=0 or file register if d=1.
Example	MYREG SET 0X10 MOVLW 0x25 MOVWF MYREG MOVLW 0x20 IORWF MYREG ,W	Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H WREG = 20 H WREG = 25 H This instruction ORs the contents of the WREG register i.e. 20 H with contents of file register i.e. 25 H and the result is placed in WREG register .

5.3.37 LFSR

Mnemonic	LFSR f, k
Operands	0 <k < 4095 0≤f ≤2
Words	2

Function	Load FSR
Clock Cycles	2
Algorithm	k → FSRf
Flags	No flags are affected.

Operation	K → FSRf	This instruction is used to load the 12-bit literal 'k' to file register. k can be loaded in FSR0, FSR1 and FSR2.
Example	LFSR0 0x456	FSR0H = 04 H and FSR0L = 56 H. This instruction will load 0456 H to FSR0 file register.

5.3.38 MOVFW

Mnemonic	MOVFW f, d , a
Flags	N and Z flags are affected.
Words	1

Function	Move file register to WREG.
Clock Cycles	1
Algorithm	(f) →destination

Operation	(f) → destination	This instruction copies the contents of the file register to WREG register.
Example	MOVFW PORTC	This instruction will copy the contents of PORTC to the WREG register.

5.3.39 MOVFF

Mnemonic	MOVFF fsource,fdestination
Flags	No flags are affected.
Words	2

Function	Move file register to file register
Clock Cycles	2 (3)
Algorithm	(fsource) → fdestination

Operation	(fsource) → fdestination	– This instruction copies the contents of the source file register to destination file register. – The source and destination can be a file register location, I/O ports or any SFR. – It is a 4 byte instruction.
Example	MOVFF PORTC, PORTB	This instruction will copy the contents of PORTC to the PORTB.

5.3.40 MOVLB

Mnemonic	MOVLB k
Flags	No flags are affected.
Words	1

Function	Move Literal to low nibble of BSR
Clock Cycles	1
Algorithm	k → BSR

Operation	k → BSR	This instruction loads the literal value into the bank selected register (BSR).
Example	MOVLB 0x3	Use bank 3

5.3.41 MOVLW k

Mnemonic	MOVLW k	**Function**	Move Literal to WREG register
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	k → W

Operation	k → W	This instruction loads the literal value into the WREG register.
Example	MOVLW 0x30	WREG = 30 H. This instruction loads the literal value i.e. 30 H into the WREG register.

5.3.42 MOVWF

Mnemonic	MOVWF f,a	**Function**	Move WREG register to file register
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	W → f

Operation	W → f	This instruction loads the WREG value into the file register. It is used with the MOVLW instruction.
Example	MOVLW 0x30 MOVWF ADCON1	WREG = 30 H. This instruction loads the literal value i.e. 30 H into the WREG register. ADCON1 = 30

5.3.43 MULLW

Mnemonic	MULLW k	**Function**	Multiply Literal with WREG
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	W x k → PRODH : PRODL

Operation	W x k → PRODH: PRODL	– This instruction multiplies the WREG value with literal value and the result is stored in the PRODH: PRODL register pair. The higher byte of result is stored in PRODH register and lower byte of result is placed in the PRODL register.
Example	MOVLW 0x30 MULLW 0x10	WREG = 30 H. This instruction multiplies the WREG valuei.e.30 H with literal value 10 H and the result is stored to the PRODH: PRODL register pair. PRODH = 03 H, PRODL = 00 H as 10 H × 30 H = 300 H

5.3.44 MULWF

Mnemonic	MULWF f, a	**Function**	Multiply WREG with file register
Flags	No flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	W x f PRODH : PRODL

Operation	W x f → PRODH: PRODL	– This instruction multiplies the unsigned bytes in WREG register and file register and result is stored in PRODH.PRODL register pair.
Example	COUNT SET 0x50 MOVLW 0x10 MOVWF COUNT MOVLW 0x05 MULWF COUNT	Set count location = 50 WREG = 10 H . COUNT = 10 H WREG = 05 H This instruction multiplies the WREG value i.e.05 H with COUNT 10 H and the result is stored to the PRODH: PRODL register pair PRODH = 00 H , PRODL = 50 H as 05 H x 10 H = 0050 H

5.3.45 NEGF

Mnemonic	NEGF f, a	**Function**	Negate file register
Flags	N, OV, C, DC, Z flags are affected.	**Clock Cycles**	1
Words	1	**Algorithm**	$(\bar{f}) + 1 \rightarrow f$, $1 \rightarrow Z$

Operation	$(\bar{f}) + 1 \rightarrow f$, $1 \rightarrow Z$	– This instruction is used to take 2's complement of the contents of the file register and result is stored in the file register
Example	COUNT SET 0x50 MOVLW 0x10 MOVWF COUNT NEGF	WREG = 10 H. COUNT = 10 H This instruction will compute 2's complement of the value in Count. Count 10 H 0001 0000 2's complement of 10H 1111 0000 Result in Count = F0 H

5.3.46 NOP

Mnemonic	NOP	Function	No Operation
Flags	No flags are affected.	Clock Cycles	1
Words	1	Algorithm	---

Operation	– This instruction does not do any operation . – It is mainly used in time delays. The PC executes the next instruction after NOP – It is a 2 byte instruction.

5.3.47 PUSH

Mnemonic	PUSH	Function	Push Top of Stack
Flags	No flags are affected.	Clock Cycles	1
Words	1	Algorithm	(PC + 2) = Top of Stack

Operation	This instruction places the program counter on the stack. The SP is incremented by 1.

5.3.48 POP

Mnemonic	POP	Function	Pop Top of Stack
Flags	No flags are affected.	Clock Cycles	1
Words	1	Algorithm	---

Operation	– This instruction pops out the value from top of stack and ignores it. The stack pointer is decremented by 1. – Now the value that the stack pointer points is the value that is last pushed onto the stack.

5.3.49 RCALL

Mnemonic	RCALL target address	Function	Relative Call
Flags	No flags are affected.	Clock Cycles	2
Words	1	Algorithm	---

Operation	– It is a 2 byte instruction used to transfer control to a subroutine whose target address is within 1 KB of the current program counter. The sequence of operation is as follows : **Step 1 :** The contents PC are pushed onto the stack. **Step 2 :** The stack pointer is incremented by 1. **Step 3 :** PC is loaded with the target address and program control is transferred to the subroutine. **Step 4 :** After executing the subroutine a RETURN instruction is executed to transfer control back to the executing program.

5.3.50 RESET

Mnemonic	RESET	Function	Reset through software
Flags	All flags are affected.	Clock Cycles	1
Words	1	Algorithm	Reset all registers and flags that are affected by a $\overline{MCLR}$ Reset

Operation	This instruction resets all the PIC18 registers and flags that are affected by a $\overline{MCLR}$ Reset.

5.3.51 RETFIE

Mnemonic	RETFIE S	Function	Return from Interrupt Exit
Flags affected	GIE/GIEH, PEIE/GIEL	Clock Cycles	2
Words	1	Algorithm	---

Operation	– This instruction is executed at the end of every ISR to transfer control back to the current program being executed. When this instruction is executed the sequence of operation is as follows : **Step 1 :** The address of program counter is popped off from the stack . **Step 2 :** Program Control is transferred to the new address stored in PC **Step 3 :** The stack pointer is decremented by 1.

5.3.52 RETLW

Mnemonic	RETLW k	Function	Return Literal in WREG register
Flags affected	None	Clock Cycles	2
Words	1	Algorithm	---

Operation	– This instruction copies the literal value 'k ' in the WREG register . When this instruction is executed the sequence of operation is as follows : **Step 1 :** The address of program counter is popped off from the stack . **Step 2 :** The stack pointer is decremented by 1 . – This instruction is used in look up tables to return the required look table element in the WREG register
Example	– RETLW D'4' ; WREG = 4 i.e. the literal 4 is copied to the WREG register.

5.3.53 RETURN

Mnemonic	RETURN s	Function	Return from Subroutine
Flags affected	None	Clock Cycles	2
Words	1	Algorithm	---

Operation	– This instruction is executed at the end of subroutine called through CALL or RCALL instructions to transfer control back to the current program being executed. When this instruction is executed the sequence of operation is as follows : **Step 1 :** The address of program counter is popped off from the stack . **Step 2 :** Program Control is transferred to the new address stored in PC. **Step 3 :** The stack pointer is decremented by 1.

5.3.54 RLCF

Mnemonic	RLCF f, d, a	Function	Rotate left the file register through carry
Flags	N, C, Z flags are affected.	Clock Cycles	1
Words	1	Algorithm	$(f_{n+1}) \leftarrow (f_n)$ where n = 0 – 6 $(f_0) \leftarrow (CY)$ $(CY) \leftarrow (f_7)$

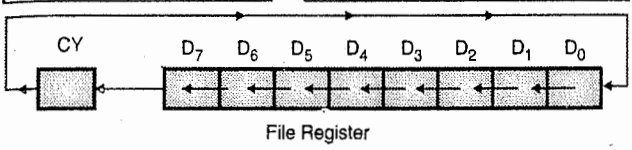


Fig. 5.3.1 : RLCF

Operation	$(f_{n+1}) \leftarrow (f_n)$ where n = 0 – 6 $(f_0) \leftarrow (CY)$ $(CY) \leftarrow (f_7)$	– This instruction will rotate the eight bits in the file register and the carry flag together by one bit to the left. – Bit 7 will move into carry and original carry will move to Bit 0 position as shown in Fig. 5.3.1.
Example	MYREG SET 0x50 MOVLW 0x10 MOVWF MYREG BSF STATUS, C RLCF MYREG, F	Set location 50 for MYREG WREG = 10 H. MYREG = 10 H = 0001 0000 C = 1 This instruction will ROTATE the contents of file register and carry to the left by 1. – MYREG = 0010 0001 AND CY=0

5.3.55 RLNCF

Mnemonic	RLNCF f, d, a	Function	Rotate left the file register NOT through carry
Flags	N, Z flags are affected.	Clock Cycles	1
Words	1	Algorithm	$(f_{n+1}) \leftarrow (f_n)$ where n = 0 to 6 $(f_0) \leftarrow f_7$

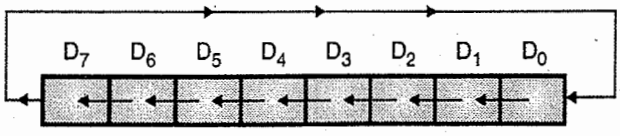


Fig. 5.3.2 : RLNCF

Operation	$(f_{n+1}) \leftarrow (f_n)$ where n = 0 – 6 $(f_0) \leftarrow f_7$	– This instruction will rotate the eight bits in the file register by one bit to the left. – Bit 7 is rotated into the Bit 0 position.
Example	MYREG SET 0x50 MOVLW 0x10 MOVWF MYREG RLNCF MYREG ,F	Set location 50 for MYREG WREG = 10 H. MYREG = 10 H = 0001 0000 This instruction will ROTATE the contents of file register to the left by 1. MYREG = 0010 0000 = 20 H

5.3.56 RRCF

Mnemonic	RRCF f, d , a	Function	Rotate right the file register through carry
Flags	N, C, Z flags are affected.	Clock Cycles	1
Words	1	Algorithm	$(f_n) = (f_{n+1})$ where n = 0 to 6 $(f_7) = (CY)$ $(CY) = f_0$

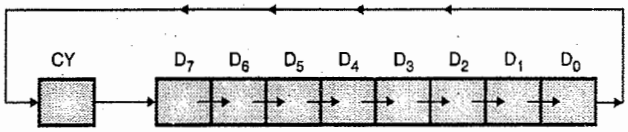


Fig. 5.3.3 : RRCF

Operation	$(f_n) \leftarrow (f_{n+1})$ where n = 0 to 6 $(f_7) \leftarrow (CY)$ $(CY) \leftarrow f_0$	– This instruction will rotate the eight bits in the file register and the carry flag together by one bit to the right. – Bit 0 will move into carry and original carry will move to Bit 7 position of the file register as shown in Fig. 5.3.3.
Example	MYREG SET 0x50 MOVLW 0x10 MOVWF MYREG BSF STATUS ,C RRCF MYREG ,F	Set location 50 for MYREG WREG = 10 H. MYREG = 10 H = 0001 0000 C=1 This instruction will ROTATE the contents of file register and carry to the right by 1 bit leaving. MYREG = 1000 1000 AND CY=0

5.3.57 RRNCF

Mnemonic	RRNCF f, d , a	Function	Rotate Right the file register NOT through carry
Flags	N, Z flags are affected.	Clock Cycles	1
Words	1	Algorithm	$(f_n) = (f_{n+1})$ where n = 0 to 6 $(f_7) = (f_0)$

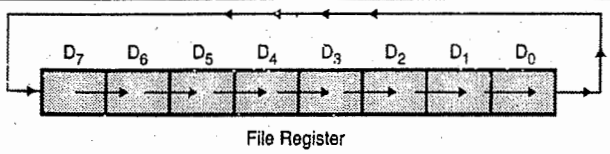


Fig. 5.3.4 : RRNCF

Operation	$(f_n) \leftarrow (f_{n+1})$ where n = 0 to 6 $(f_7) \leftarrow f_0$	This instruction will rotate the eight bits in the file register by one position to the right. Bit 0 is rotated into bit 7 position.
Example	MYREG SET 0x50 MOVLW 0x10 MOVWF MYREG RRNCF MYREG ,F	Set location 50 for MYREG WREG = 10 H . MYREG = 10 H = 0001 0000 This instruction will ROTATE the contents of file register to the right by 1, so that MYREG = 0000 1000

5.3.58 SETF

Mnemonic	SET f, a		**Function**	SET file register
Flags	No flags are affected.		**Clock Cycles**	1
Words	1		**Algorithm**	FF H → f

Operation	FF H → f	This instruction is used to Set all the bits of a file register to 1.
Example	SETF TRISB	This instruction will set all the bits of PORTB making it an input port.

5.3.59 SLEEP

Mnemonic	SLEEP		**Function**	Enter the Sleep Mode
Flags	$\overline{TO}$, $\overline{PD}$ are affected.		**Clock Cycles**	1
Words	1		**Algorithm**	00h → WDT, 1 → $\overline{TO}$, 0 → $\overline{PD}$

Operation	– This instruction is used to put the microcontroller into the SLEEP mode. The oscillator stops operating . The watchdog timer is reset and the power down status bit $\overline{PD}$ is cleared. The Time out status bit $\overline{TO}$ is set . – For coming out of the SLEEP mode there are two methods. They are : (1) Activate the Watchdog timer interrupt. (2) Hardware interrupt.

5.3.60 SUBLW

Mnemonic	SUBLW k		**Function**	Subtract WREG from Literal
Operands	0 <k < 255		**Clock Cycles**	1
Words	1		**Algorithm**	k – (WREG) → WREG
Addr. Mode	Immediate Addressing Mode.		**Flags**	N, OV, C, DC, Z flags are affected.

Operation	k - (WREG) → WREG	– This instruction subtracts the contents of the WREG register from the 8-bit literal 'k' and the result is placed in WREG register. The steps followed for subtraction are as follows : **Step 1 :** Take 2's complement of the WREG **Step 2 :** Add k to the 2's complement **Step 3 :** Observe the result and discard carry if any.
Example	MOVLW 0x20 SUBLW 0x25	WREG = 20 H WREG = 05 H (25 H – 20 H). This instruction subtracts the contents of the WREG register i.e. 20 H from the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register.

5.3.61 SUBWF

Mnemonic	SUBWF f,d,a		**Function**	SUBTRACT WREG from file register.
Flags	N, OV, C, DC, Z flags are affected.		**Clock Cycles**	1
Words	1		**Algorithm**	(f) - (W) → destination

Operation	(f) - (W) → destination	– This instruction subtracts the contents of the WREG register from the contents of the file register and the result is placed in WREG register if d=0 or file register if d=1. – The steps followed for subtraction are as follows : **Step1 :** Take 2's complement of the WREG **Step2 :** Add file register contents to the 2's complement **Step 3 :** Observe the result and discard carry if any.
Example	MYREG SET 0x10 MOVLW 0x25 MOVWF MYREG MOVLW 0x20 SUBWF MYREG, F	Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H WREG = 20 H MYREG = 05 H (25 H - 20 H) This instruction subtracts the contents of the WREG register i.e. 20 H from contents of file register i.e. 25 H and the result is placed in MYREG file register .

5.3.62 SUBWFB

Mnemonic	SUBWFB f,d,a	Function	Subtract WREG and borrow bit from the file register.
Clock Cycles	1	Flags	N, OV, C, DC, Z flags are affected.
Words	1	Algorithm	(f) - (W) - (CY) → destination

Operation	(f) – (W) – (CY) → destination	This instruction subtracts the contents of the WREG register and borrow/carry flag from the contents of the file register.
Example	MYREG SET 0x10 MOVLW 0x25 MOVWF MYREG BCF STATUS , C MOVLW 0x20 SUBWFB MYREG,F	Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H Carry = 0 WREG = 20 H MYREG = 05 H (25 H – 20 H – 0) This instruction subtracts the contents of the WREG register i.e. 20 H and carry from contents of file register i.e. 25 H and the result is placed in MYREG .

5.3.63 SUBFWB

Mnemonic	SUBFWB f,d,a	Function	Subtract File register and borrow bit from the WREG register.
Clock Cycles	1	Flags	N, OV, C, DC, Z flags are affected.
Words	1	Algorithm	(W) - (f) - (CY) → destination

Operation	W- (f) - (CY) → destination	This instruction subtracts the contents of the file register and carry (borrow) flag from the contents of the WREG register and the result is stored in WREG register if d= 0 and file register if d=1.
Example	MYREG SET 0x10 MOVLW 0x20 MOVWF MYREG BCF STATUS , C MOVLW 0x25 SUBFWB MYREG	Allocate Location 10 H for MYREG WREG = 20 H MYREG = 20 H Carry = 0 WREG = 25 H WREG = 05 H (25 H – 20 H – 0) This instruction subtracts the contents of the file register i.e. 20 H and carry from contents of WREG register i.e. 25 H and the result is placed in WREG register.

5.3.64 SWAPF

Mnemonic	SWAPF f,d,a	Function	Swap nibbles in file Register
Clock Cycles	1	Flags	No flags are affected.
Words	1	Algorithm	$(f_{3-0}) = (f_{7-4})$ $(f_{7-4}) = (f_{3-0})$

Operation	$(f_{3-0}) \rightarrow (f_{7-4})$ $(f_{7-4}) \rightarrow (f_{3-0})$	This instruction interchanges the low order and high order nibbles of the file register and the result is stored in WREG register if d= 0 and file register if d=1.
Example	MYREG SET 0x10 MOVLW 0x20 MOVWF MYREG SWAPF MYREG	Allocate Location 10 H for MYREG WREG = 20 H MYREG = 20 H The instruction SWAPF leaves the WREG Register holding the value 02 H (0000 0010 B).

5.3.65 TBLRD

Mnemonic	TBLRD* (Table Read) TBLRD*+ (Read and then increment TBLPTR) TBLRD*- (Read and then decrement TBLPTR) TBLRD+* (increment TBLPTR and read)	Function	Table Read
Clock Cycles	2	Flags	No flags are affected.
Words	1	Algorithm	Read a byte from ROM into the TABLAT register

Operation	– This instruction is use to load the data byte into the program ROM in the TABLAT register .This data is entered in the arrays , look- up tables strings and is read by the microcontroller. – The TBLPTR register holds the address of the byte to be read .

5.3.66 TBLWT

Mnemonic	TBLWT* (Table Write) TBLWT*+ (write and then increment TBLPTR) TBLWT*- (Write and then decrement TBLPTR) TBLWT+* (increment TBLPTR and write)	Function	Table Write
Clock Cycles	2	Flags	No flags are affected.
Words	1	Algorithm	Write a byte to flash ROM .

Operation	– This instruction is use to write the data byte into the flash program ROM. – The TBLPTR register holds the address of the byte to be written.

5.3.67 TSTFSZ

Mnemonic	TSTFSZ f, a	Function	Test file Register and Skip if it is zero
Clock Cycles	1(2)	Flags	No flags are affected.
Words	1	Algorithm	_____

Operation	– This instruction is used to test the contents of a file register. If the file register has value zero then the next instruction is skipped.	
Example	Count SET 0x20	Set location 20 H for count
	MOVLW 0x10	WREG = 10 H
	MOVWF Count	Count = 10 H
	MOVLW 0X55	WREG = 55 H
	MOVWF PORTD	PORTD = 55 H
	COMF PORTD,F	Toggle PORT D
	L2 : DECF Count , F	decrement count
	TSTFSZ Count	test Count for zero
	BRA L2	Repeat it

5.3.68 XORLW

Mnemonic	XORLW k	Function	EX-OR Literal with WREG.
Operands	0 <k < 255	Clock Cycles	1
Words	1	Algorithm	(WREG) EX-OR kF → WREG
Addr. Mode	Immediate Addressing Mode.	Flags	N, Z flags are affected.

Operation	(WREG) EX-OR k → WREG	This instruction EX-ORs the contents of the WREG register with the 8-bit literal 'k' and the result is placed in WREG register .
Example	MOVLW 0x20 XORLW 0x25	WREG = 20 H WREG = 20 H 0010 0000 XOR 25H 0010 0101 WREG = 05 H 0000 0101 This instruction EX-ORs the contents of the WREG register i.e. 20 H with the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register.

5.3.69 XORWF

Mnemonic	XORWF f,d,a	Function	EX-OR WREG and file register.
Flags	N and Z flags are affected.	Clock Cycles	1
Words	1	Algorithm	(W) EX-OR (f) → destination

Operation	(W) EX-OR (f) → destination	This instruction EX-ORs the contents of the WREG register with the contents of the file register and the result is placed in WREG register if d=0 or file register if d=1.
Example	MYREG SET 0X10 MOVLW 0x25 MOVWF MYREG MOVLW 0x20 XORWF MYREG ,W	– Allocate Location 10 H for MYREG WREG = 25 H MYREG = 25 H WREG = 20 H WREG =20 H 0010 0000 MYREG = 25H 0010 0101 WREG = 20 H 0000 0101 – This instruction EXORs the contents of the WREG register i.e. 20 H with the 8-bit literal 'k' i.e. 25 H and the result is placed in WREG register.

❑❑❑

UNIT - IV

CHAPTER 6

PIC Programming in C

6.1 Introduction

- The different compilers generate hex files that can be downloaded into the ROM of the microcontroller.
- The size of the hex file is the main reason of worry for the microcontroller programmers. This is because :
  (i) The on-chip ROM of the microcontrollers is limited.
  (ii) The code space for the PIC18 is limited to 2 MB.
- The size of the assembly language program is much smaller in comparison to the C language code. The assembly language program is tedious. It is time consuming.
- The C language program is much easier to write. It is less time consuming. However the hex file produced by C program is larger.
- The various reasons for writing programs in C are

  (i) C is easier to modify and update.
  (ii) Code available in the libraries can be used by users.
  (iii) The C code is portable with other microcontrollers with little or no changes.
  (iv) C is easy and less time consuming to write than assembly.

- Various companies develop C compiler for PIC microcontroller. We need to include the file P18F458.h for C programs.

6.2 Data Types and Time Delays in C

Table 6.2.1 lists some data types used by PIC.

Table 6.2.1 : Data types used by PIC

Data Type	Size	Data Range
Unsigned char	8 bit	0-255
Char	8 bit	–128 to +127
Unsigned int	16 bit	0 to 65,535
Int	16 bit	– 32768 to + 32767
Unsigned short	16 bit	0 to 65535
Short	16 bit	– 32768 to + 32767
Unsigned long short	24 bit	0 to 16777215
Short long	24 bit	– 8388608 to 8388607
Unsigned long	32 bit	0 to 4294, 967295
Long	32 bit	– 2147483648 to + 2147483648

Unsigned char

- It is an 8 bit data type that takes value in range of 0-255 i.e. (00H-FFH).
- It is widely used in applications like setting a counter value where signed data is not required.
- C compilers use signed char by default. If we want to use unsigned char then the keyword **unsigned** must be placed in front of char.
- It can also be used for a string of ASCII characters, including extended ASCII characters.

Program 1

Write a C program to send values 00H-FFH to port A.

```
# include <P18F548.h >        // For TRISA and
                              // PORT A declaration
void main (void)
{
    unsigned char i ;
    TRISA = 0 ;               // Make Port A as output
    for (i = 0 ; i < = 255, i ++)
    PORTA = i ;
    while (1) ;
}
```

Program 2

Write a C program to send hex values for ASCII characters 0, 2, 3, 5, 6, A, B, C, D to Port B.

```
# include < P18F458.h>
void main (void)
{
unsigned char myno [ ] = "02356ABCD" ;
                          //Data is stored in RAM
unsigned char i ;
TRISB = 0 ;               //Make Port B as output port
for (i = 0 ; i < 9 ; i ++)
PORTB = myno [i] ;
while (1) ;               //Stay here forever
}
```

On running the program Port B displays ASCII values 30H, 32H, 33H, 35H, 36H, 41H, 42H, 43H and 44H.

Program 3

Write a C program to send values of – 5 to + 5 to Port A.

```
# include < P18F458.h >
void main (void)
{
    char data [ ] = {+1, – 1, + 2, – 2, + 3, – 3, + 4, – 4,
                    + 5, – 5} ;
    unsigned char i ;
    TRISA = 0;                    // Port A as output port
    for (i = 0 ; i < 10 ; i ++)
    {  PORTA = data [i] ;
}
while (1) ;
    }
```

Program 4

Write a C program to toggle all bits of Port C 50,000 times.

```
    # include < P18F458.h >
    void main (void)
    {
        unsigned short long x ;
        TRISC = 0 ;
        for (x = 0 ; x < = 50000 ; x ++)
        {
          PORTC = 0x55 ;
          PORTC = 0xAA;
        }
        while (1) ;
    }
```

6.3 Time Delays in C

- Time delays can be created by two methods in C. They are :
  (i) Using a simple for loop.
  (ii) Using PIC18 timers.
- While creating time delay using a simple for loop we cannot get exact delay because of the following reasons :
  (i) The crystal frequency is connected to the OSC1-OSC2 input pins. The duration of the clock period for the machine cycle is a function of the crystal frequency.
  (ii) In C programs the C compiler converts the C statements and functions to assembly language instructions. As a result different compilers produce different codes. Hence the instructions executed in a loop may vary with different compilers.

Note : Software delays can be implemented by the use of simple for loops. But to get exact amount of delay one needs to use try and test method. Here one can check the delay on a Digital Storage Oscilloscope (DSO) or even a Cathode Ray Oscilloscope (CRO). To obtain accurate delays without try and test method we need to use hardware delays.

Program 5 SPPU- Dec. 2013, 4 Marks

Write a C program to toggle all the bits of Port A continuously with a 250 ms delay. Assume that the system is PIC18F458 with XTAL = 10 MHz.

```
    # include < P18F458.h>
    void Delay (unsigned int) ;
    void main (void)
    {
        TRISA = 0 ;      // Port A is output port forever
        while (1)
        {
        PORTA = 0x55 ;
        Delay (250) ;
        PORTA = 0xAA ;
        Delay (250) ;
        }
    }
    void Delay (unsigned int xtime)
    {
        unsigned int i ;
        unsigned char j ;
    for (i = 0 ; i < xtime ; i ++)
    for (j = 0 ; j < 165 ; j ++) ;
    }
```

When we use functions like DELAY in the C program we need to know following points :

(i) Before the main function, the declaration :
void Delay (unsigned int)
will tell the compiler that there will be a function called DELAY. It is called **function prototype**.

(ii) The functions are generally defined immediately after the main program ends as in program 8. The first line of function declaration must be exactly same as its function prototype.

Program 6 SPPU - Dec. 2012, May 2013, 8 Marks

Write embedded C program to blink LED connected to port of PIC.

Soln. :

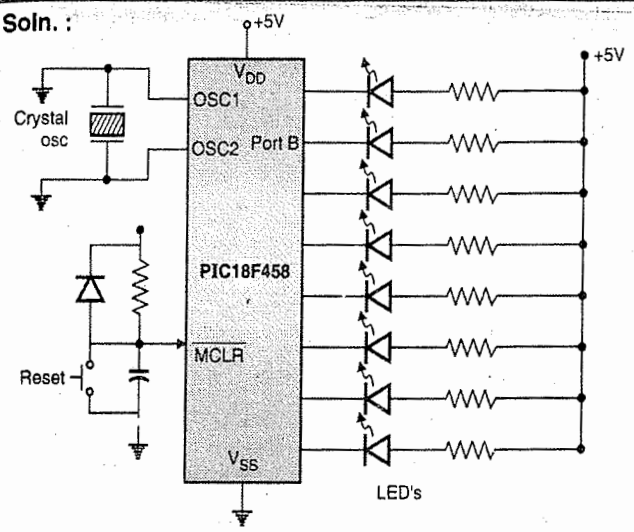


Fig. P. 6

```
# include < P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISA = 0 ;      // Port A is output port forever
    while (1)
    {
    PORTB = 0x00H ;      //
    Delay (250) ;        //
    PORTB = 0xFFH ;      //  } Blink LEDs
    Delay (250) ;        //
    }
}
void Delay (unsigned int xtime)
{
    unsigned int i ;
    unsigned char j ;
for (i = 0 ; i < xtime ; i ++)
for (j = 0 ; j < 165 ; j ++) ;
}
```

Program 7 **SPPU - May 2014, 8 Marks**

Write a C program to toggle all bits of Ports B, C, D continuously with a 250 ms delay. Assume XTAL = 10 MHz.

Soln. :

```
# include <P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISB = 0 ;  } Make Ports A, B, C and D as
    TRISC = 0 ;  } output ports forever
    TRISD = 0 ;  }
while (1)
    {
        PORTB = 0x55 ;    //Toggle ports A, B, C, D
        PORTC = 0x55 ;
        PORTD = 0x55 ;
        Delay (250) ;
        PORTB = 0xAA ;
        PORTC = 0xAA ;
        PORTD = 0xAA ;
        Delay (250) ;
    }
}
void Delay (unsigned int xtime)
{
    unsigned int i ;
    unsigned char j ;
    for (i = 0 ; i < xtime ; i ++)
    for (j = 0 ; j < 165 ; j ++ ) ;
}
```

Program 8 **SPPU - May 2012, 8 Marks**

Write a program in C for PIC to toggle alternate the bits of port B continuously with a 250 ms delay.

```
# include <P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISB = 0 ;      //Make Port B as output port

while (1)
    {
        PORTB = 0x55 ;    // Toggle port B
        Delay (250) ;
        PORTB = 0xAA ;
        Delay (250) ;
    }
}
void Delay (unsigned int xtime)
{
    unsigned int i ;
    unsigned char j ;
    for (i = 0 ; i < xtime ; i ++)
    for (j = 0 ; j < 165 ; j ++ ) ;
}
```

Program 9

Write a program in C for PIC to toggle all the bits of port C continuously with a 200 ms delay.

```
# include <P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISC = 0 ;          // Port C as output port
    while (1)
    {
        PORTC = 0x55 ;   // Toggle port C
        Delay (200) ;
        PORTC = 0xAA ;
```

```
        Delay (200) ;
    }
}
void Delay (unsigned int xtime)
{
    unsigned int i ;
    unsigned char j ;
    for (i = 0 ; i < xtime ; i ++)
    for (j = 0 ; j < 165 ; j ++ ) ;
}
```

6.4 I/O Programming in C

6.4.1 Byte Size I/O

- The Port A - Port D are byte accessible. We use the Port A - Port D labels as defined in the C18 header file.

Program 10

LEDs are connected to the bits in Port A and Port D. Write a C18 program that shows the count from 0 to FFH on the LEDs.

```
# include < P18F458.h>
# define LED PORTD
void main (void)
{
    TRISA = 0 ;      // Port A as an output port
    TRISD = 0 ;      // Port D as an output port
    PORTA = 00 ;     // Clear Port A
    LED = 0 ;        // Clear Port D
    for (; ;)        // repeat forever
    {
        PORTA++ ;        // increment Port A
        LED ++ ;         // increment Port D
    }
}
```

Program 11

Write a C18 program to read the data from Port A and give it to Port B after a small delay.

```
# include < P18F458.h >
unsigned char i, x ;
void main ( )
    {
        TRISA = 0xFF ;     // Port A as input port
        TRISB = 0   ;      // Port B as output port
    while (1)
    {
        x = PORTA ;        //get byte from Port A
        for (i = 0 ; i < = 100 ; i ++ )
                           //wait for sometime
        PORTB = x ;        //send it to Port B
    }
}
```

Program 12 SPPU - May 2012, 8 Marks

Write a C program to get a byte of data from Port C. If it is less than 1000, send it to Port B otherwise send it to Port D.

```
# include < P18F458.h >
void main (void)
{
    unsigned char i ;
    TRISC = 0xFF ;          // Port C as input port
    TRISB = 0 ;             // Port B as output port
    TRISD = 0 ;             // Port D as output port
    while (1)
    {
    i = PORTC ;       // read data from PORTC
    if (i < 1000 )
    PORTB = i ;       // if i < 1000 then send it to
                         PORTB
    else
    PORTD = i ;       // otherwise send it to PORTD
    }
}
```

6.4.2 Bit Addressable I/O Programming

- The I/O ports of PIC18FXX are bit addressable. We can access a single bit without disturbing the remaining port bits.
- We use PORTxbits.Rxy to access a single bit of Port x, where x is the port A, B, C or D and y is the bit (0-7) of that port. e.g. PORTAbits.RA5 indicates PORTA.5.
- We access the TRISx register in the same manner where TRISCbits.TRISC7 indicates the RC7 bit of PORTC.
- Table 6.4.1 shows the single bit addresses of PIC18.

Table 6.4.1 : Single bit address of PIC18F458 ports

PORT A	PORT B	PORT C	PORT D	PORT E	Port's Bit
RA0	RB0	RC0	RD0	RE0	D0
RA1	RB1	RC1	RD1	RE1	D1
RA2	RB2	RC2	RD2	RE2	D2
RA3	RB3	RC3	RD3		D3
RA4	RB4	RC4	RD4		D4
RA5	RB5	RC5	RD5		D5
	RB6	RC6	RD6		D6
	RB7	RC7	RD7		D7

Port bits structure : Fig. 6.4.1 shows the structure of Port C bits as given by C18 compiler. The structure of ports can be found in microcontroller header file.

```
extern volatile near unsigned char PORTB ;
extern volatile near union {
struct {
        unsigned RC0 : 1;
        unsigned RC1 : 1 ;
        unsigned RC2 : 1 ;
        unsigned RC3 : 1 ;
        unsigned RC4 : 1 ;
        unsigned RC5 : 1 ;
        unsigned RC6 : 1 ;
        unsigned RC7 : 1 ;
} ;
struct {
        unsigned INT0 : 1 ;
        unsigned INT1 : 1 ;
        unsigned CANTX : 1 ;
        unsigned CANRX : 1 ;
        unsigned : 1 ;
        unsigned PGM : 1 ;
        unsigned PGC : 1 ;
        unsigned PGD : 1 ;
        } ;
} PORTCbits ;
```

Fig. 6.4.1 : Port C bit structure

Program 13

Write a C program to toggle only bit RA3 continuously without disturbing the rest of bits of Port A.

```
# include < P18F458.h >
# define mybit PORTAbits. RA3        // declare RA3
void main (void)
{
        TRISAbits.TRISA3 = 0; // Make RA3 as output
        while (1)
        {
        mybit = 1 ;                // turn on RA3
        mybit = 0 ;                // turn off RA3
        }
}
```

Program 14

Write a C18 program to monitor bit PC5. If it is high send 55H to Port D otherwise send AAH to Port B.

```
# include < P18F458.h >
# define mybit PORTCbits.RC5
void main (void)
{
        TRISCbits.TRISC5 = 1 ;  // Make RC5 as input
        TRIS B = 0 ;            // Make Port B as output
        TRIS D = 0 ;            // Make Port D as output
        while (1)
        {
        if (mybit = = 1)
        PORTD = 0x55 ;
        else
        PORTB = 0xAA ;
        }
}
```

Program 15

Write a C program to turn bit 6 of Port D on and off 50,000 times.

```
# include < P18F458.h >
# define Mybit PortDbits. RD6
void main (void)
{
        unsigned int i ;
        TRISDbits.TRISD6 = 0 ; // Make RD6 an output
        for (i = 0 ; i < 50,000 ; i ++)
        {
         mybit = 1 ;
         mybit = 0 ;
        }
        while (1) ;            // Stay here forever
}
```

Program 16

Write a C program to get the status of bit RB6 and send it to RC7 continuously.

```
# include < P18F458.h >
# define inbit PORTBbits.RB6
# define outbit PORTCbits.RC7
void main (void)
{
        TRISBbits.TRISB6 = 1 ;    // Make RB6 as input
        TRISC bits. TRISC7 = 0 ; // Make RC7 as output
        while (1)
        {
         outbit = inbit ;
        }
}
```

Program 17

A door sensor is connected to the RD0 pin and a buzzer is connected to RC6. Write a C18 program to monitor the door sensor and when it opens, sound the buzzer. The buzzer can be sound by sending a square wave of few hundred Hz to it.

```
# include < P18F458.h >
# define sensor PORTDbits.RD0
# define buzzer PORTCbits.RC6
void main ( )
{
        unsigned char i ;
        TRISDbits.TRISD0 = 1 ;
                                // Make PORTD.0 as input
        TRISCbits.TRISC6 = 0 ;
                                // Make PORTC.6 as output
        while (sensor = = 1)
        {
           buzzer = 0 ;
           for (i = 0 ; i < 5000 ; i + +)
                                // software delay
           buzzer = 1 ;         // sound buzzer
           for (i = 0 ; i < 500 ; i + +) // software delay
        }
        while (1) ;
}
```

Program 18

Write embedded C program to implement HEX counter on port and display the count.

```
title "HEX Count on LED Display"

#include < P18F458.h>
#define Led_Disp PORTA

void main(void)
{
    TRISA = 0;                          // declares PORTA as output
    Led_Disp =0x00;                     // initiates count to 0
    int j;
    for (j=0x00; j<0x0F; j++)
    {
        Led_Disp = j;                   // display incremented Hex count on LED.
        delay(200);
    }
    while(1);
}

void delay(int delayval)
{
unsigned int m,n;
for (m=0x00; m<0x0F; m++)               //looping around to introduce software delay
    for (n=0x00; n<0x0F; n++);
}
```

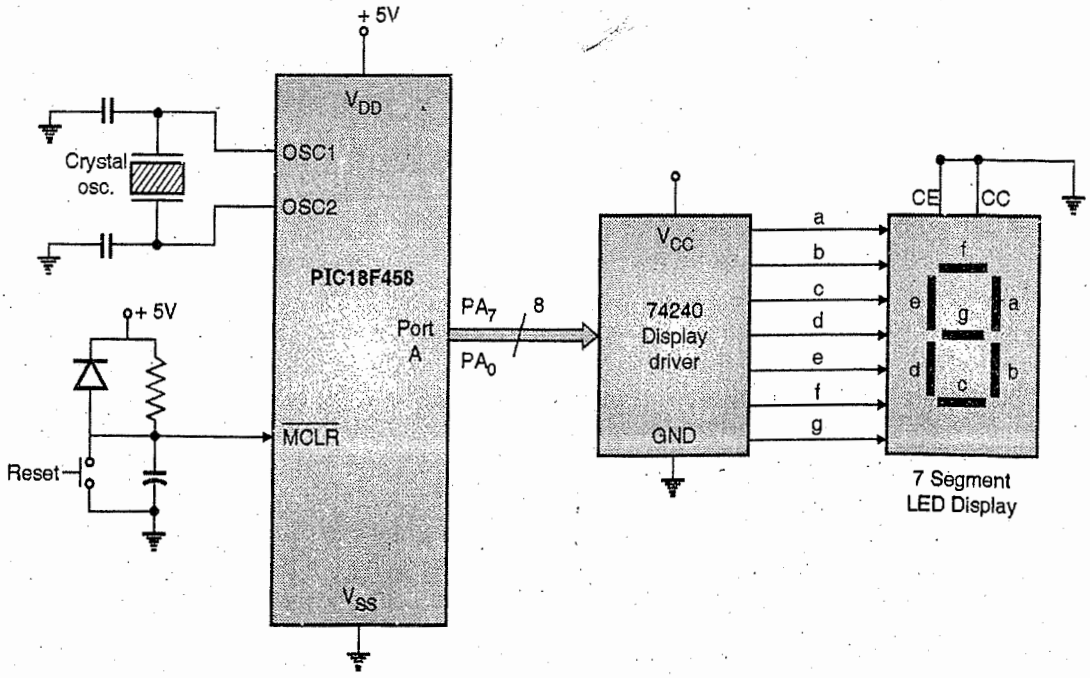


Fig. P. 18

6.5 Logic Operations in C

The different logical operators used in C are AND (&&), OR (| |) and NOT (|). The programmers are less familiar with bitwise operators AND (&), OR(|), EX-OR(^), Inverter (~), Shift right (>>) and shift left (<<). The bitwise operators are used in software engineering for embedded systems and control. Table 6.5.1 shows bitwise logic operators for C.

Table 6.5.1 : Bitwise logic operators for C

		AND	OR	EX-OR	Inverter
A	B	A & B	A | B	A^B	Y = ~B
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Program 19

Write a C program to toggle all the bits of Port B, Port C and Port D continuously with a delay. Use the inverting operator.

```
# include <P18F458.h >
void main (void)
{
    TRISB = 0 ;   // make ports B, C and D as output
    TRISC = 0 ;
    TRISD = 0 ;
    PORTB = 0x55 ;
    PORTC = 0x55 ;
    PORTC = 0x55 ;
    while (1) ;
    {
    for (i = 0 ; i < = 100 ; i + +)   // Software delay
    PORTB = ~ PORTB ;
    PORTC = ~ PORTC ;
    PORTD = ~ PORTD ;
    }
}
```

Program 20 : Write a C program to toggle all the bits of Port B, C and D continuously with a delay. Use EX-OR operator.

```
# include < P18F458.h >
void Delay (unsigned int) ;
void main (void)
{
    TRISB = 0 ;
    TRISC = 0 ;   } Make Ports B, C and D as output
    TRISD = 0 ;
    PORTB = 0x55 ;
    PORTC = 0x55 ;
    PORTD 0x55;
    while (1)
    {
    PORTB = PORTB ^ 0xFF ;
    PORTC = PORTC ^ 0xFF ;
    PORTD = PORTD ^ 0xFF;
    Delay (300);
    }
}
void Delay (unsigned int count)
{
    unsigned int i ;
    unsigned char j ;
    for (i = 0 ; i < count ; i + + )
    for (j = 0 ; j < 165 ; j + +) ;
}
```

Program 21 : Write a C program to get bit RC0 and send it to RD3 after inverting it.

```
# include < P18F458.h >
# define inbit PORTCbits.RC0
# define outbit PORTDbits.RD3
void main void
{
    TRISCbits.TRISC0 = 1 ;
                    // Make Port C.0 as input
    TRISDbits.TRISD3 = 0 ;
                    // Make Port D.3 as output
    while (1)
    {
        outbit = ~ inbit ;    // Get a bit from RC0
    }
}
```

Program 22

Write a C program to read RC0 and RC1 bits an issue an ASCII character to Port B according to the following table :

RC1	RC0	
0	0	Send '0' to Port B
0	1	Send '1' to Port B
1	0	Send '2' to Port B
1	1	Send '3' to Port B

```
# include < P18F458.h >
void main (void)
{
    unsigned char i ;
    TRISC = 0xFF ;          // Make Port C an input
    TRISB = 0 ;             // Make Port B an outut
    while (1)
    {
        i = PORTC ;         // Read Port C
```

```
        i = i & 0x3 ;          // Mask the unused bits
        switch (i)
        {
            case (0) :
            {
            PORTB = '0' ;      // Issue ASCII 0
            break ;
            }
            case (1) :
            {
            PORTB = '1' ;      // Issue ASCII 1
            break ;
            }
            case (2) :
            {
            PORTB = '2' ;      // Issue ASCII 2
            break ;
            }
            case (3) :
            {
            PORTB = '3' ;      // Issue ASCII 3
            break ;
            }
        }
    }
}
```

6.6 Data Conversion Programs in C

- The advanced microcontrollers have real time clock (RTC) that displays data and time even when the power is switched off.
- The RTC provides the time and date in packed BCD. To display it the data must be converted to ASCII.

6.6.1 ASCII Numbers

On the ASCII keyboards when key "0" is activated "011 0000" (30H) is given to the computer. If the key "1" is pressed 31H is given. Table 6.6.1 lists the ASCII codes for digits 0-9.

Table 6.6.1 : ASCII code for digits 0-9

Key	ASCII (hex)	Binary	BCD (unpacked)
0	30	011 0000	0000 0000
1	31	011 0001	0000 0001
2	32	011 0010	0000 0010
3	33	011 0011	0000 0011
4	34	011 0100	0000 0100
5	35	011 0101	0000 0101
6	36	011 0110	0000 0110
7	37	011 0111	0000 0111
8	38	011 1000	0000 1000
9	39	011 1001	0000 1001

6.6.2 Packed BCD to ASCII Conversion

- The RTC provides the time of day in hours; minutes; second and date (year : month : day) continuously irrespective of power is on/off. This data is in packed BCD.
- To convert packed BCD to ASCII it needs to be converted to unpacked BCD. The unpacked BCD is the added with 30H.

e.g.	Packed BCD	Unpacked BCD	ASCII
	0x91	0x09, 0x01	0x39, 0x31

6.6.3 ASCII to Packed BCD Conversion

To convert ASCII to packed BCD, it is first converted to unpacked BCD and then combined to make packed BCD.

e.g. 3 and 5 on keyboard give 33H and 35H.

To get 35H is our aim.

Key	ASCII	Unpacked BCD	Packed BCD
3	33	0000 0011	
5	35	0000 0101	0011 0101 = 35H

After conversion the packed BCD numbers are processed and result will be in packed BCD format.

Program 23 SPPU - Dec. 2013, 4 Marks

Write a C program to convert packed BCD 0x49 to ASCII and display the bytes on PORTB and PORTC.

```
# include < P18F458 . h >
void main (void)
{
    unsigned char i, j ;
    unsigned char mbyte = 0x49 ;
    TRISB = 0 ;
    TRISC = 0 ;
    i = mbyte & 0x0F ;        // Mask upper 4 bits
    PORTB = x | 0x30 ;        // Make it ASCII
    j = mbyte & 0xF0 ;        // Mask lower 4 bits
    j =j >> 4 ;               // Shift it to lower 4 bits
    PORT C = j | 0x30 ;       // Make it ASCII
}
```

Program 24

Write a C program to convert ASCII digits of '4' and '9' to packed BCD and display it PORTC.

```
# include <P18F458.h>
void main (void)
{
    unsigned char bbyte ;
    unsigned char i = '4' ;
    unsigned char j = '9' ;
```

```
        TRISC = 0 ;                // Port C as output
        i = i & 0x0F ;             // Mask 3
        i =i << 4  ;
                                   // Shift left to make upper BCD digit
        j = j & 0x0F ;             // Mask 3
        bbyte = i | j ;  // Combine to make packed BCD
        PORTC = bbyte ;
    }
```

6.6.4 Checksum Byte

Checksum byte is used to maintain the integrity of data. To calculate the checksum byte of a set of data the following steps are to be taken :

Step I : Add all the bytes

Step II : Discard carry if any

Step III : Take 2's complement.

For example : The checksum byte of data 25H, 35H, 45H, 65H. can be calculated as ;

Step I : 25H + 35H + 45H + 65H = $\boxed{1}$ 04H → Carry

Step II : Discard carry.

∴ Result = 04H

Step III : Take 2's complement of 04H (0000 0100)

2's complement = (1111 1100) = FCH.

∴ checksum byte = FCH

Program 25

Write a C program to calculate the checksum byte of the data 25H, 35H, 45H and 65H and display the result on Port B.

```
    # include < P18F458.h >
    void main (void)
    {
        unsigned char dat1[ ] = {(0x25, 0x35, 0x45,
0x65) } ;
        TRISB = 0 ;
        PORTB = ~ (dat1[0] + dat1[1] + dat1[2]
                  + dat1[3]) + 1;  // Checksum byte
        while (1)
    }
```

6.6.5 Binary (hex) to Decimal and ASCII Conversion in C18

- The print function in C can convert data from binary (hex) to decimal or vice-versa. But it requires a lot of memory space. This increases the hex file size. Hence, in PIC18 microcontroller systems it is recommended to use a separate conversion function.
- Binary to decimal conversion is widely used in ADCs, DACs. In some RTCs the time and dates are provided in binary. To display binary data we have to convert it to decimal and then ASCII. The hex format is an easy way to represent the binary data.
- The binary data 00-FFH is converted to decimal to give 000-255. It is done by dividing it by 10 and saving the remainder. FFH is decimal 255.

e.g. :

	Quotient	Remainder	
FF/0A	25	5	Low digit (LSD)
25/0A	2	5	Middle digit
		2	High digit (MSD)

= (255)

Program 26

Write a C program to convert FFH to decimal and display the digits on PORTB, PORTC and PORTD

```
# include <P18F458.h>
void main ()
{
unsigned char i ,
unsigned char binbyte, b1, b2, b3;
TRISB = 0 ;                  // Port B as output port.
TRISC = 0 ;                  // Port C as output port.
TRISD = 0 ;                  // Port D as output port
binbyte = 0xFF ;             // binary (hex) value
i = binbyte / 10 ;   // divide by 10 (Least significant digit)
b1 = binbyte % 10 ;          // find remainder
b2 = i % 10 ;                // middle digit
b3 = i / 10 ;                // most significant digit
PORTB = b1 ;
PORTC = b2 ;
PORTD = b3 ;
}
```

6.7 Data Serialization in C

Data serialization is a method by which the data can be transmitted or received bit by bit using a single pin of microcontroller. There are two methods of transferring data serially. They are

(i) Using the serial port.

(ii) Serialization i.e. transfer the data one bit at a time and control the sequence of data and spaces between them.

Program 27

Write a C program to send out the value 22H serially one bit at a time via RB0. The LSB should go out first.

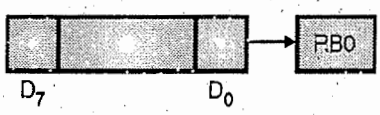


Fig. P. 27 : LSB going out first

```
# include <P18F458.h>
# define PB0 PORTB bits. RB0
void main (void)
{
    unsigned char bbyte  = 0x22 ;
    unsigned char regLSB ;
    unsigned char i ;
    regLSB = bbyte ;
    TRISBbits. TRISB0 = 0          // Make RB0 as output
    for (i = 0; i < 8 ; i++)
    {
    PB0 = reg LSB & 0x01 ;
    reg LSB = reg LSB > > 1 ;
    }
}
```

Program 28

Write a C program to send out the value 22 H serially one bit at a time through RC0. The MSB should go out first.

```
# include <P18F458.h>
# define PC0 PORTC bits. RC0
void main (void)
{
    unsigned char mybyte  = 0x22 ;
    unsigned char regMSB ;
    unsigned char i ;
    regMSB = mybyte ;
    TRISCbits. TRISC0 = 0 ;        // Make RC0 as output
    for (i = 0; i < 8 ; i++)
    {
    PC0 = (regMSB >> 7) & 0x01
    regMSB = regMSB << 1 ;
    }
}
```

Program 29

Write a C program to bring in a byte of data serially one bit at a time via RB0 pin. The MSB should come in first.

```
# include <P18F458.h>
# define PB0 PORTBbits.RB0
void main (void)
{
    unsigned char i ;
    unsigned char ACC = 0 ;
    TRISBbits.TRISB0 = 1 ;         // Make RB0 as input
    TRISD = 0 ;                    // Make Port D as output
    for (i = 0; i < 8; i++)
    {
    ACC = ACC << 1 ;
    ACC| = PB0 & 0x01 ;
    }
    PORTD= ACC ;
}
```

Program 30 : Write a C program to receive byte serially one bit at a time via RB0 pin. Place the byte on Port D. The LSB should come in first.

```
# include <P18F458.h>
# define PB0 PORTBbits.PB0
void main (void)
{
    unsigned char i ;
    unsigned char ACC = 0 ;
    TRISBbits.TRISB = 1 ;          // Make RB0 as input
    TRISD = 0 ;                    // Make PortD as output
    for (i = 0; i < 8 ; i++)
    {
    ACC = ACC >> 1 ;
    ACC| = (PB0 & 0x01) << 7 ;
    }
PORTD = ACC ;
}
```

6.8 Program ROM Allocation in C

In PIC18 there are two spaces for storing the data. They are as follows :

(i) The 4KB data RAM space with address range 000 - FFFH. Many PIC18F XX chips have less than 4 KB for the file register data RAM.

(ii) The 2 MB of code (program) space with addresses of 000000 - 1FFFFFH. This 2 MB space is used for storing programs. Hence, it is directly under the control of program counter (PC). There is one problem in using the program code space for storage of fixed data. The more the code space we use for data, the less is the space left for program code. e.g. : If we use PIC18F252 with 4 KB on chip ROM and we use 1 KB to store look up table only 3 KB is left for program. It can create a problem for some applications. Hence, microchip has added EEPROM memory to PIC microcontroller for data storage.

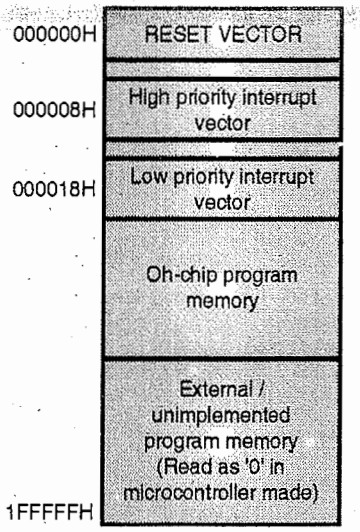


Fig. 6.8.1 : PIC18 Program ROM space

Fig 6.8.1 shows PIC program ROM space. Table 6.8.1 lists program ROM size for some PIC18FXX family members.

Table 6.8.1

	On chip code ROM (Bytes)	Code Address Range (Hex)
PIC18F2220	4 KB	00000 – 00FFF
PIC18F2410	16 KB	00000 – 03FFF
PIC18F458/4580	32 KB	00000 – 07FFF
PIC18F6680	64 KB	00000 – 0FFFF
PIC18F8722	128 KB	00000 – 1FFFF

6.8.1 Allocating Program Space to Data

- In C18 example we have studied, the byte size variables are stored in the data RAM. To make the C compiler use program (code) ROM space for fixed data, we use the codeword rom as shown below :

```
rom char mydat [ ]  = "Holiday"
                    ;        // Use code space for data
rom char dat 1  = 2, data 2 = 3 ;
                             // Use code space for data
```

6.8.2 NEAR and FAR for Code

- PIC18 has a maximum of 2 MB of on-chip program ROM space although there are exceptions e.g. some PIC18 chips come with 4 KB and 128 KB of program ROM.
- Inorder to make efficient use of the code space, the C compiler uses near and far storage qualifiers.
- The near qualifier tells that a program memory data variable is placed in the first 64 KB of program ROM. While a far qualifier indicates that the data variable in program ROM can be placed anywhere in 2 MB ROM space. In our programs far qualifier is the default qualifier. Hence we need not specify it.

Table 6.8.2 : Near and far usage for ROM

Storage qualifier	ROM
Near	In program space of 0000 – FFFFH (64 kB)
Far	In program space of 000000 – 1FFFFFH (2 MB)

```
e.g.   near rom const char mydat [ ] ="Timepass" ;
                                   // program ROM data
       far rom const chardat1 [ ] = "Hello" ;
                                   // program ROM data
```

6.8.3 Pragma and Allocating a Fixed Address to Data and Code

- Data or code can be placed at a specific ROM address using the ORG directive.
- In C programming we use #pragma section directive to do the same thing "section" is part of an application (code or data) that can be assigned a specific memory location.
- For on-chip ROM program memory we have two alternatives
  (i) code (ii) ROM data
- The #pragma code directive is used for the program code. This is because it has executable instructions. The #pragma ROM data directive is used for fixed data such as look up tables, strings etc.

6.9 Data RAM Allocation in C

- PIC18 family members can have a maximum of 4 KB of data RAM. All family members do not have 4 KB RAM. The data RAM size can vary from 256 bytes to 4096 bytes depending on the microcontroller chip that is used.
- PIC18 family members have atleast one bank of RAM. This bank is called as the **access bank**.
- 128 bytes of data RAM are used for SFRs while remaining are used for scratch pad. The C18 compiler allocates the RAM (leaving SFR) to variables and stack used in the program.

```
e.g.   unsigned char i = 2, j = 5 ;
                   // Uses data RAM to store data
```

- We require successive RAM locations for a array elements i.e. the size of array is limited to size of data RAM in a given PIC18 chip.

```
e.g.   unsigned char dat1 = "0123456789"
                   // Uses RAM space to store data
```

6.9.1 NEAR and FAR for Data

- Near and far are two storage qualifiers that are used by PIC18, C compiler for data RAM allocation.
- They signify which parts of data RAM are to be used for the storing the variables that are declared.
- The keyword near limits the C compiler usage of RAM to the access bank for data declaration. The keyword far puts the complete data RAM at disposal of C compilers.

Table 6.9.1 : Near and far usage for data RAM

Storage qualifier	RAM
near	in access bank
far	anywhere in data RAM file register (default).

- The Table 6.9.1 indicates that programs written for the PIC18 chips with limited data RAM cannot have too many arrays with larger number of elements.

e.g. near unsigned char dat1[100] ;
// 100 byte space in RAM
far unsigned char dat2[50] ;
// 50 byte space in RAM.

6.9.2 Putting Data in a Specific RAM Address

- The MOVLW and MOVWF instructions allow user to place data at a specific RAM address.
- #pragma directive is used for placing the data at a specific RAM address in the C18 compiler.
- There are two alternatives for #pragma when it is used for data RAM.

(i) i data (initialized data)
(ii) u data (uninitialized data)

They are used by C to assign an explicit address in RAM data.

6.9.3 Overlay Storage Class

- In order to use the data space of PIC18 efficiently the C compiler introduces an overlay storage class.
- It reserves the memory such that two variables can have the same physical address provided that both the variables are not simultaneously active.

e.g.

```
unsigned char bina(void)
{
    overlay unsigned char x = 0
    x = x + 1 ;
    return x ;
}
```

and

```
unsigned char binb(void)
{
overlay unsigned char y = 0;
y = y + 2;
return y ;
}
```

- The x and y variables are not active at the same time. The C compiler uses the same physical address location in the file register for both of them.
- If the keyword "overlay" is removed the C compiler allocates different locations to the two variables.
- The C compiler allocates two different physical locations for the variables when the variables are simultaneously active and depend on each other.

e.g.

```
unsigned char binc(void)
{
overlay unsigned char x = 0 ;
x = bind ( );
return x ;
}
```

and

```
unsigned char bind(void)
{
overlay unsigned char y = 0 ;
y = y + 2 ;
return y;
}
```

- In the program the C compiler allocates a different RAM locations to x and y even through we use the keyword overlay. This is because binc calls function bind, both the variables are simultaneously active and depend on each other. Thus, **overlay** is a new storage class that can be applied on different local variables.

□□□

UNIT - IV

CHAPTER 7

Timers and its Programming

Syllabus Topic : Timers / Counters

7.1 Timers / Counters

Function and use : The PIC18 family has **two to five timers** depending on the family member. The timers are called as Timer 0 , Timer 1 , Timer 2 , Timer 3 and Timer 4. They can be used as timers in order to generate a time delay or as counters to count events occurring outside the microcontroller.

- When the timer is used as the timer, the PIC18's crystal is used as the frequency source. $\frac{1^{th}}{4}$ of crystal oscillator frequency is given to the timer. When the timer is used as counter, it is a pulse outside the circuit that increments the TMRxH and TMRxL registers. Additional registers used in this mode are TCON, TMRxH and TMRxL.
- E.g. For Timer 0 the T0CS (timer 0 clock source) bit in the T0CON register decides the source of clock for the timer. If bit=0 then the timer operates as a timer with pulses from OSC1 and OSC2 pins. If bit =1 then the timer operates like a counter and gets the pulses outside from PIC18.
- Timer 0, Timer 2, Timer 3 are 16 bit timers while Timer 2 and Timer 4 are 8 bit timers.
- Timer 2 and Timer 4 use the instruction cycle clock as their clock source while the other timers may use internal or external clock signals as their clock source.

7.2 Prescaling of PIC Timers

SPPU - May 13

University Question

Q. Describe prescaling. **(May 2013, 8 Marks)**

- PIC Timers can be optionally **Pre-scaled**. The timer operates on the input clock signal. The timer counts with reference to the clock input. The system clock is usually running at a very high speed (In MHz range) If the timers are operated with this clock input as it is, they would reach their terminal counts/ overflow very fast.
- In real time in many conditions we need a much slower time base to be calculated / elapsed. To support this PIC optionally allows this system clock to be internally divided by stage (e.g. ÷ by 2, 4, 8)
- This internal division to slow down the clocking input to the Timers derived from the higher speed system clock, is called as **Pre-Scaling**. The circuit using the fixed, variable or programmable ÷ by stages in order to achieve the pre-scaling, is called as **Pre-Scalar.**

Timer 0	:	Programmable using T0PS2-T0PS0 bits of T0CON register from 1:2 to 1:256
Timer 1	:	Variable 1:1, 1:2, 1:4, 1:8 configured by T1CKPS1, T1CKPS0 of T1CON Register.
Timer 2	:	Variable 1:1, 1:4, 1:16 configured by T2CKPS1, T2CKPS0 of T2CON Register.
Timer 3	:	Variable 1:1, 1:2, 1:4, 1:8 configured by T3CKPS1, T3CKPS0 of T3CON Register.

7.3 Timer 0

- The Timer 0 has the following features :

1. Software selectable as an 8-bit or 16-bit timer/counter.
2. It is readable and writable.
3. Dedicated 8-bit software programmable prescaler.
4. Clock source selected can be external or internal.
5. Interrupt-on-overflow from FFh to 00h in 8-bit mode and FFFFh to 0000h in 16-bit mode.
6. Edge select for external clock.

7.3.1 Timer 0 Block Diagram

SPPU - Dec. 14, May 15

University Question

Q. Explain timer 0 mode and its applications of PIC18XX in detail. **(Dec. 2014, 2 Marks, May 2015, 8 Marks)**

Fig. 7.3.1 shows a simplified block diagram of the Timer 0 in 8-bit mode and Fig. 7.3.2 shows a simplified block diagram of the Timer 0 in 16-bit mode.

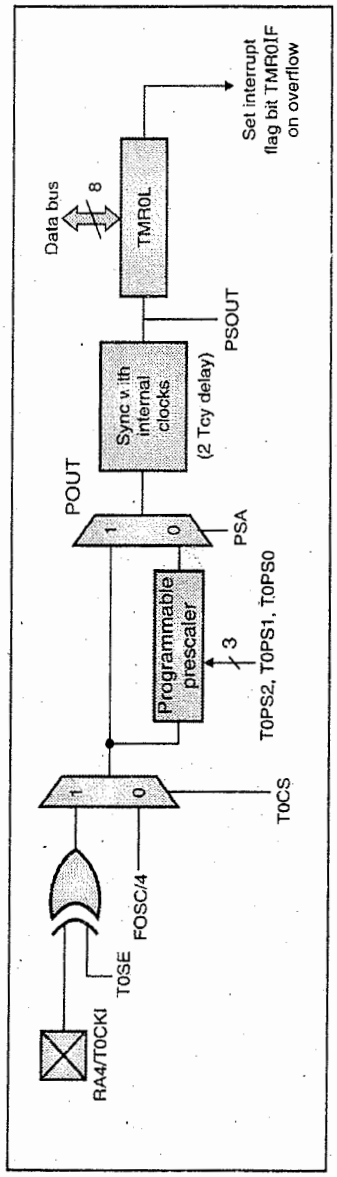


Fig. 7.3.1 : TIMER 0 block diagram in 8-BIT mode

- Timer 0 can be used as a timer or counter.
- Timer 0 operates as a timer if the clock source is from the instruction cycle clock $(1/4)^{th}$ of the crystal oscillator frequency is fed to the timer.
- If the TOCS bit of T0CON register is set (TOCS = 1), the timer 0 operates like a counter and gets the pulse from T0CKI pin (PORTA.4 or RA.4). Depending on the user selection of rising or falling edge of T0CKI signal the TMR0L and TMR0H registers are incremented.
- As shown in Fig. 7.3.2, the TMR0H register behaves as a buffer between the internal data bus and TMR0 high byte. When the TMR0L register is read by the microcontroller, the contents of the TMR0 high byte are sent to the TMR0H register. When the microcontroller reads the high byte, it is read from TMR0H register.

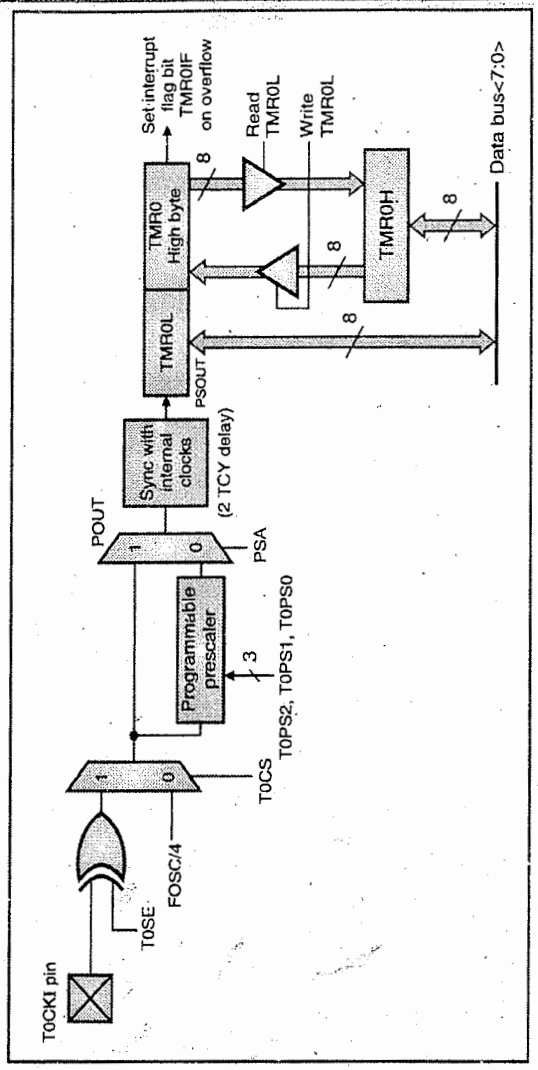


Fig. 7.3.2 : TIMER 0 block diagram in 16-BIT mode

- The **main application** of Timer 0 is **creating time delays.** A wide range of delays can be generated by selecting the prescaler. Another application is measuring the frequency of unknown signal.

Timer modes :

- Timer 0 can operate in 2 modes.
  (i) 8 bit mode (ii) 16 bit mode
- In the 8 bit mode, values 00 to FFH can be loaded into the TMR0L register. On reaching FFH, it rolls over to 00H setting the TMR0IF flag bit in the INT0CON register.
- In the 16 bit mode, values 0000H to FFFFH can be loaded into the TMR0L and TMR0H registers.
- Timer 0 can be set in 16-bit mode by clearing the T08BIT in T0CON. Registers TMR0H and TMR0L are used to access the 16-bit timer value.

  The TMR0 interrupt is generated when the TMR0 register overflows from FFh to 00h in 8-bit mode or FFFFh to 0000h in 16-bit mode. This overflow sets the TMR0IF bit. The interrupt can be masked by clearing the TMR0IE bit.

7.3.2 Timer 0 Registers

- TIMER 0 can be accessed as low byte and high byte if used as 16 bit timer. The low byte register is called as TMR0L (timer 0 low byte) and high byte register is called TMR0H (timer 0 high byte). Like the other SFRs both these registers can be accessed.

Size : The registers TMR0L and TMR0H are of 8 bit.

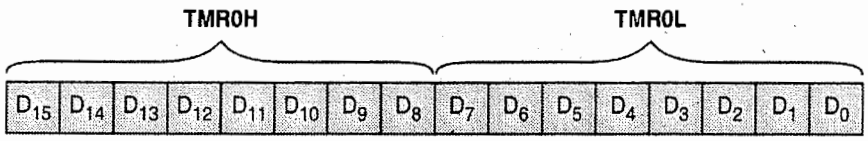


Fig. 7.3.3 : Timer 0 high and low registers

7.3.3 T0CON (Timer 0 Control) Register

Each timer has a control register. It is called T0CON (Timer 0 control) register.

Use : It is used for setting different timer operating modes.

Fig. 7.3.4 shows the T0CON register. The T0CON register is a readable and writable register that controls all the aspects of Timer 0, including the prescaler selection.

Size : It is an 8 bit register.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
TMR0ON	T08BIT	T0CS	T0SE	PSA	T0PS2	T0PS1	T0PS0

bit 7 **TMR0ON :** Timer 0 On/ Off Control bit
1 = Enables Timer 0
0 = Stops Timer 0

bit 6 **T08BIT :** Timer 0 8-bit/16-bit Control bit
1 = Timer 0 is configured as an 8-bit timer/counter
0 = Timer 0 is configured as a 16-bit timer/counter

bit 5 **T0CS :** Timer 0 Clock Source Select bit
1 = Transition on T0CKI/RA4 pin
0 = Internal Instruction cycle clock $\left(\frac{f_{osc}}{4}\right.$ from crystal oscillator$\left.\right)$

bit 4 **T0SE :** Timer 0 Source Edge Select bit
1 = Increment on high-to-low transition on T0CKI pin
0 = Increment on low-to-high transition on T0CKI pin

bit 3 **PSA :** Timer 0 Prescaler Assignment bit
1 = Timer 0 prescaler is not assigned. Timer 0 clock input bypasses prescaler.
0 = Timer 0 prescaler is assigned. Timer 0 clock input comes from prescaler output.

bit 2-0 **T0PS2 : T0PS0 :** Timer 0 Prescaler Select bits
111 = 1:256 Prescale value (f_{osc}/4/256)
110 = 1 :128 Prescale value (f_{osc}/4/128)
101 = 1 : 64 Prescale value (f_{osc}/4/64)
100 = 1 : 32 Prescale value (f_{osc}/4/32)
011 = 1 :16 Prescale value (f_{osc}/4/16)
010 = 1:8 Prescale value (f_{osc}/4/8)
001 = 1:4 Prescale value (f_{osc}/4/4)
000 = 1:2 Prescale value (f_{osc}/4/2)

Fig. 7.3.4 : T0CON (Timer 0 control) Register

7.3.4 TMR0IF Flag Bit

The INTCON register has the Timer 0 interrupt flag bit (TMR0IF). When the timer is used as 8 bit, and if the timer reaches its maximum value FFH, it rolls over to 00H and TMR0IF bit is set. When the timer is used as 16 bit timer, once it reaches its maximum value FFFFH, it rolls over to 0000H and TMR0IF flag bit is set. Before loading the Timer 0 registers TMR0L and TMR0H, the TMR0IF bit is observed.

7.4 Timer 1

SPPU - Dec. 14, May 15

University Question

Q. Explain timer 1 and its applications of PIC18XX in detail. **(Dec. 2014, May 2015, 2 Marks)**

- The Timer 1 module timer/counter has the following features :

1. 16-bit timer/counter (two 8-bit registers : TMR1H and TMR1L)
2. Both the registers are Readable and writable.
3. Internal or external clock select
4. Interrupt-on-overflow from FFFFh to 0000h when TMR1IF flag bit goes high.
5. Reset from CCP module special event trigger

7.4.1 Timer 1 Block Diagram

Fig. 7.4.1 shows a simplified block diagram of the Timer 1.

- Timer 1 is a 16 bit Timer/counter.
- Timer 1 operates as a 16 bit timer if the clock source is from the instruction cycle clock $(1/4)^{th}$ of the crystal oscillator frequency is fed to the timer.
- Depending on the external clock signal, the Timer 1 can work as a synchronous counter or asynchronous counter.
- If TMR1CS = 1 in T1CON register, Timer 1 increments on rising edge of external clock input. (RC0/T1OS0/T1CKI).
- If the Timer 1 oscillator is enabled (T1OSCEN = 1 of T1CON), the T1OS0 and T1OS1 provide clock input to the microcontroller. Usually a 32 kHz crystal is connected to T1OS1 and T1OS0 pins. It is mainly utilized for power saving in the sleep mode as the Timer 1 cannot be disabled by the SLEEP instruction. This facilitates the Timer 1 to implement on chip Real Timer clock (RTC).
- Timer 1 can work in 16 bit mode only as
  (i) 16 bit timer (ii) Synchronous counter
  (iii) Asynchronous counter

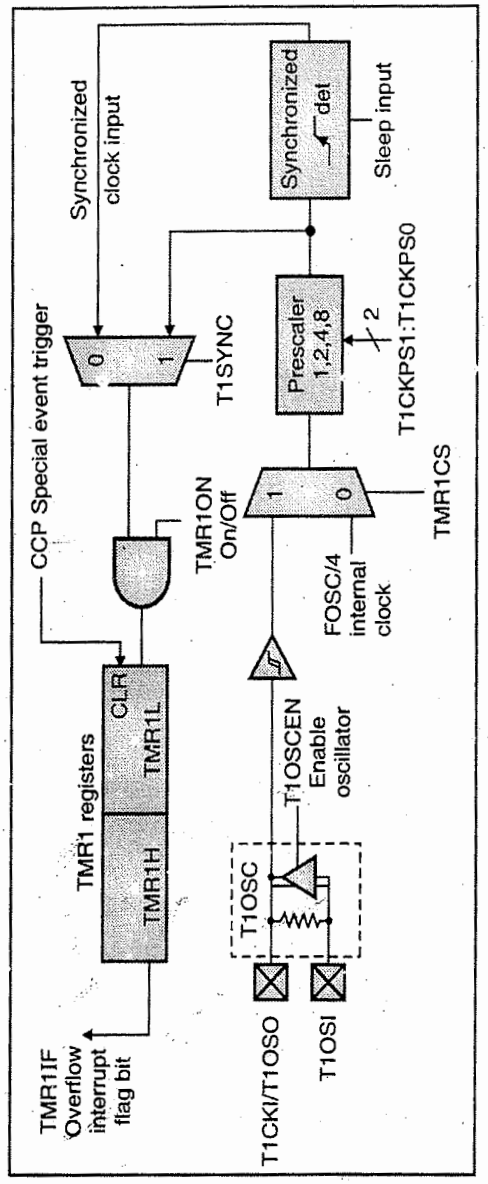


Fig. 7.4.1 : Timer 1 Block Diagram

Applications :

(i) To generate time delays.
(ii) Frequency measurement
(iii) Implementing real time clock
(iv) Generating square waves with certain duty cycles.
(v) Generate special event trigger in compare mode of CCP module.

7.4.2 Timer 1 Registers

- Timer 1 is a 16 bit Timer. It can be accessed as low byte and high byte. The low byte register is called Timer 1 low byte (TMR1L) register and high byte register is called Timer 1 high byte (TMR1H) register.

Size : Both these registers are of 8 bit.

Mode : Timer 1 operates in 16 bit mode only. It does not support 8 bit mode like Timer 0. Fig. 7.4.2 shows Timer 1 registers.

TMR1H								TMR1L							
D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

Fig. 7.4.2 : Timer 1 low and high registers

7.4.3 Timer 1 Control Register (T1CON)

SPPU - Dec. 15

University Question

Q. Explain T1CON register in PIC18FXXX. **(Dec. 2015, 4 Marks)**

Size : It is an 8 bit register that controls the operation of Timer 1.

- Fig. 7.4.3 shows the Timer 1 Control register. This register controls the operating mode of the Timer 1, and also has the Timer 1 Oscillator Enable bit (T1OSCEN). Timer 1 can be enabled/disabled by setting/clearing control bit, TMR1ON in (T1CON register) shown in Fig. 7.4.3.
- Timer 1 has a prescaler option. It supports factors 1:1, 1:2, 1:4,1:8.

RD16	–	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR1ON
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

bit 7 **RD16 :** 16-bit Read/Write Mode Enable bit
1 = Enables register read/write of Timer 1 in one 16-bit operation
0 = Enables register read/write of Timer 1 in two 8-bit operations

bit 6 **Not used**

bit 5-4 **T1CKPS1 :** T1CKPS0 : Timer 1 Input Clock Prescale Select bits
11 = 1:8 Prescale value,
10 = 1:4 Prescale value
01 = 1:2 Prescale value,
00 = 1:1 Prescale value

bit 3 **T1OSCEN :** Timer 1 Oscillator Enable bit
1 = Timer 1 oscillator is enabled, 0 = Timer 1 oscillator is switched off

bit 2 **T1SYNC :** Timer 1 Synchronization
When TMR1CS = 1, Counter mode synchronizes to external clock input
T1SYNC bit is ignored if TMR1CS = 0

bit 1 **TMR1CS :** Timer 1 Clock Source Select bit
1 = External clock from pin RC0/T1OSO/T1CKI (on the rising edge),
0 = Internal clock (FOSC/4 from crystal oscillator).

bit 0 **TMR1ON :** Timer 1 On bit
1 = Starts Timer 1,
0 = Stops Timer 1

Fig. 7.4.3 : T1CON-Timer 1 Control Register

7.4.4 TMR1IF Flag Bit

The **TMR1IF (Timer 1 interrupt flag) bit** is in the PIR1 register. Once the Timer 1 reaches its maximum value FFFFH, it rolls over to 0000H and TMR1IF flag is set. Before loading the TMR1 registers TMR1L and TMR1H, the TMR1IF flag is observed.

7.5 TIMER 2

SPPU - Dec. 14, May 15

University Question

Q. Explain timer 2 and its applications of PIC18XX in detail. **(Dec. 2014, 2 Marks, May 2015, 8 Marks)**

- The Timer 2 module timer has the following features:

1. It is 8-bit timer (TMR2 register)
2. It has an 8-bit period register called (PR2)
3. Both registers are readable and writable
4. Software programmable prescaler (1:1, 1:4, 1:16)
5. Software programmable postscaler (1:1 to 1:16)
6. Interrupt on TMR2 match of PR2
7. SSP module that uses TMR2 output for generating the clock shifts.

7.5.1 Timer 2 Block Diagram

- Fig. 7.5.1 shows the block diagram of Timer 2. As shown in Fig. 7.5.1 $\frac{f_{osc}}{4}$ is the clock source for Timer 2. As no external source is used for Timer 2, Timer 2 cannot be used as a counter.
- Timer 2 will increment from 00H till it matches with the value in PR2. The EQ signal will set the TMR2IF flag and TMR2 will be reset to 00H. The comparator output is divided by postscale factor.
- The prescaler and postscaler counters are cleared if
  - (i) The device is reset
  - (ii) TMR2 register is written
  - (iii) T2CON register is written.
- Operating mode Timer 2 works as 8 bit timer with clock soruce as the instruciton cycle clock.

Applications :

(i) Generating periodic interrupts.

(ii) Generating time delays.

(iii) Generating pulses with variable widths (i.e. generating PWM in CCP module).

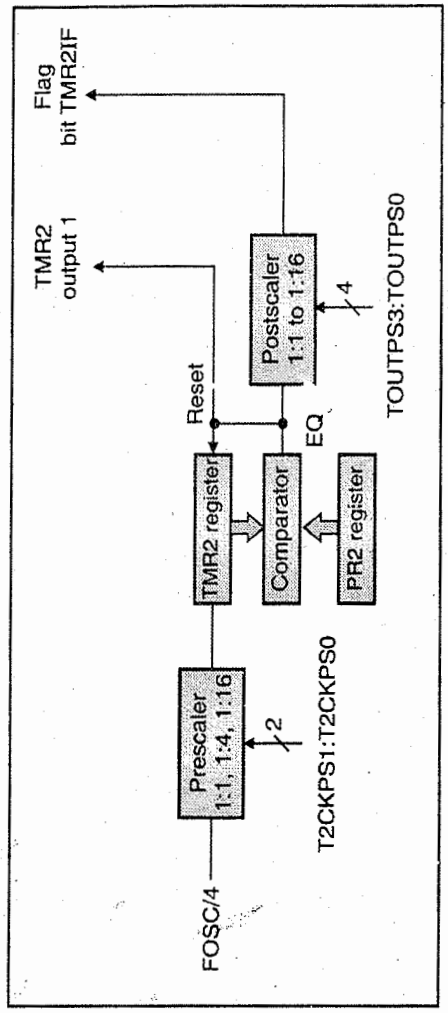


Fig. 7.5.1 : Timer 2 block diagram

7.5.2 Timer 2 Registers and TMR2IF Flag

Size : Timer 2 has two **8 bit registers** called **TMR2 and PR2(period register).**

- The period register (PR2) can be set to a fixed value. Then the Timer 2 increments till it matches with the value in PR2. This will set the TMR2IF flag. The TMR2 will reset to 0, once the TMR2IF flag is set. TMR2IF flag is part of the PIR1 register.

7.5.3 Timer 2 Control Register (T2CON)

SPPU - Dec. 15

University Question

Q. Explain T2CON register in PIC18FXXX. **(Dec. 2015, 4 Marks)**

Size : It is an 8 bit register that controls the Timer 2 operation.

- Fig. 7.5.2 shows the Timer 2 Control register. Timer 2 can be switched off by clearing control bit TMR2ON
- The prescaler and postscaler selection of Timer 2 are controlled by T2CON register of Timer 2.

–	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

bit7 **Not used**

bit6-3 **TOUTPS3 :** TOUTPS0 : Timer 2 Output Postscale Select bits
0000 = 1 : 1 Postscale
0001 = 1 : 2 Postscale
.
.
.
1111 = 1 : 16 Postscale

bit2 **TMR2ON :** Timer 2 On bit
1 = Start Timer 2,
0 = Stop Timer 2

bit1-0 **T2CKPS1 : T2CKPS0 :** Timer 2 Clock Prescale Select bits
00 = Prescaler is 1,
01 = Presclaer is 4,
1x = Prescaler is 16

Fig. 7.5.2 : Timer 2 control register (T2CON)

7.6 Timer 3

SPPU - Dec. 14, May 15

University Question

Q. Explain timer 3 and its applications of PIC18XX in detail. **(Dec. 2014, 2 Marks, May 2015, 8 Marks)**

– The Timer 3 timer/counter has the following features:

1. Timer 3 is a 16-bit timer/counter. It has two 8-bit registers: TMR3H and TMR3L
2. Both registers are readable and writable
3. It has internal or external clock select
4. It has a software programmable prescaler (1:1, 1:2, 1:4, 8, 1:16)
5. It supports Interrupt-on-overflow from FFFFh to 0000h
6. It can be reset by a module trigger from CCP1/ECCP1.

7.6.1 Timer 3 Block Diagram

Fig. 7.6.1 shows the block diagram of Timer 3. It is a 16 bit Timer. After the timer reaches its maximum value FFFF it rolls over to 0000H and TMR3IF (Timer 3 interrupt flag) is set. TMR3IF flag bit is present in the PIR2 register of the PIC microcontroller.

– Timer 3 can be operated in 3 modes.
  (i) 16 bit Timer.
  (ii) Synchronous counter
  (iii) Asynchronous counter.
– Timer 3 operates as a timer if the clock source is from the internal clock source.
– Timer 3 operates as counter if the clock source is T1CKI/RC0 or crystal oscillator connected to T1OS0 and T1OS1.

Applications :

(i) To generate time delays.
(ii) Frequency measurement
(iii) Implementing RTC.
(iv) Generate special event trigger in compare mode of CCP module.
(v) Generate square waves with variable duty cycles.

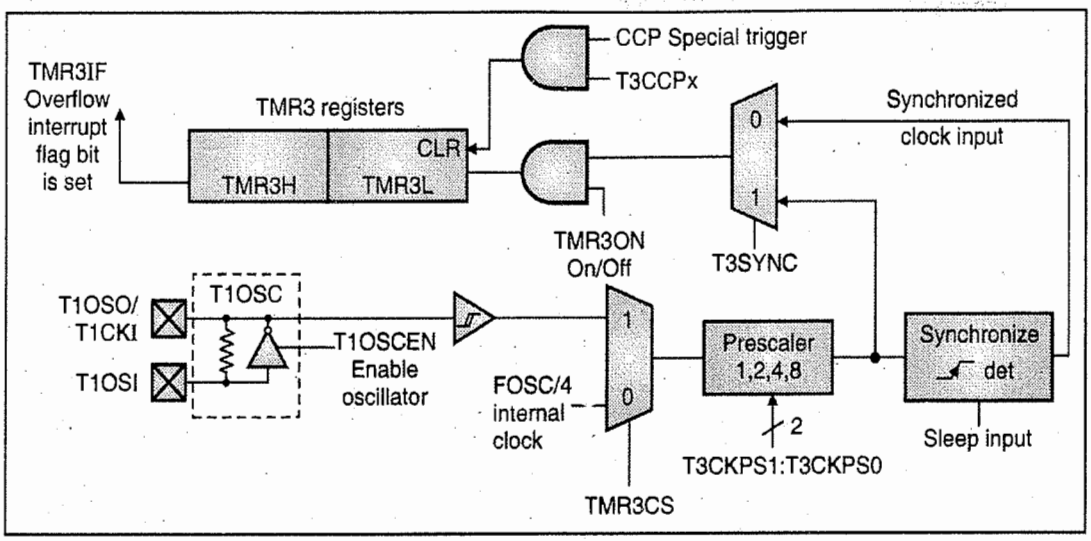


Fig. 7.6.1 : Block diagram of Timer 3

7.6.2 Timer 3 Control Register (T3CON)

Size : T3CON register is of 8 bit

Fig. 7.6.2 shows the Timer 3 Control (T3CON) register. This register controls the operating mode of the Timer 3 module and sets the CCP1 and ECCP1 clock source.

RD16	T3CCP2	T3CKPS1	T3CKPS0	T3CCP1	T3SYNC	TMR3CS	TMR3ON
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

bit 7 **RD16 :** 16-bit Read/Write Mode Enable bit
1 = Enables register read/write of Timer 3 in one 16-bit operation
0 = Enables register read/write of Timer 3 in two 8-bit operations.

bit 6-3 **T3CCP2 : T3CCP1 :** Timer 3 and Timer 1 to CCP Enable bits
1x = Timer 3 is the clock source for compare/capture CCP module.
01 = Timer 3 is the clock source for compare/capture of CCP2.
Timer 1 is the clock source for compare/capture of CCP1
00 = Timer 1 is the clock source for compare/capture of CCP.

bit 5-4 **T3CKPS1 : T3CKPS0 :** Timer 3 Input Clock Prescale Select bits
11 = 1 : 8 Prescale value
10 = 1 : 4 Prescale value
01 = 1 : 2 Prescale value
00 = 1 : 1 Prescale value

bit 2 **T3SYNC :** Timer 3 External Clock Input Synchronization Control bit
(Not usable if the system clock comes from Timer 1/Timer 3).
When TMR3CS = 1
1 = Do not synchronize external clock input
0 = Synchronize external clock input
When TMR3CS = 0 :
This bit is ignored. Timer 3 uses the internal clock when TMR3CS = 0.

bit 1 **TMR3CS :** Timer 3 Clock Source Select bit
1 = External clock input from Timer 1 oscillator (T1OS1) or T1CKI
(on the rising edge after the first falling edge)
0 = Internal clock (FOSC/4)

bit 0 **TMR3ON :** Timer 3 On bit
1 = Starts Timer 3
0 = Stops Timer 3

Fig. 7.6.2 : Timer 3 control register (T3CON)

Syllabus Topic : Programming the PIC18 Timers

7.7 Programming the PIC18 Timers

Ex. 7.7.1

Find the timer's clock frequency and its period for different PIC18 based systems with the following crystal frequencies assuming no prescaler is used.

(a) 2 MHz (b) 10 MHz (c) 16 MHz

Soln. : The PIC timers use $\frac{f_{osc}}{4}$, in addition to prescalers as their clock frequency.

(a) For 2 MHz

$$\text{Timers clock frequency} = \frac{f_{osc}}{4} = \frac{2}{4} = \mathbf{0.5\ MHz}$$

$$\text{Period T} = \frac{1}{0.5\ \text{MHz}} = \mathbf{2\ \mu s}$$

(b) For 10 MHz

$$\text{Timer clock frequency} = \frac{f_{osc}}{4} = \frac{10}{4} = \mathbf{2.5\ MHz}$$

$$T = \frac{1}{2.5\ \text{MHz}} = \mathbf{0.4\ \mu s}$$

(c) For 16 MHz

$$\text{Timer clock frequency} = \frac{f_{osc}}{4} = \frac{16}{4} = \mathbf{4\ MHz}$$

$$T = \frac{1}{4\ \text{MHz}} = \mathbf{0.25\ \mu s}$$

7.7.1 Programming the Timer 0 in 8/16 Bit Mode

Step I :	Load to T0CON register with 8 or 16 bit mode and prescale.
Step II :	Load the TMR0H and TMR0L registers with the initial count (for 8 bit mode load the TMR0L register with initial count)
Step III :	Start the Timer.
Step IV :	Observe the TMR0IF flag. Exit from loop when the TMR0IF flag is set.
Step V :	Stop the timer.
Step VI :	Clear the TMR0IF flag.
Step VII:	Goto back to step II

Ex. 7.7.2

Assuming XTAL = 10 MHz, write a program to generate a square wave of 2 KHz on port B.5.

Soln. :

We need to generate a square wave of 2 KHz on pin RB5.

$$\therefore \text{ The period of square wave} = \frac{1}{2\ \text{KHz}} = 500\ \mu s$$

Let us assume the duty cycle of square wave is 50%. Hence, the square wave will be high for 250 μs and it will be low for 250 μs

$$\text{XTAL} = 10\ \text{MHz}$$

$$\therefore \text{ Timer clock frequency} = \frac{10\ \text{MHz}}{4} = 2.5\ \text{MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5\ \text{MHz}} = 0.4\ \mu s.$$ i.e. counter will count up every 0.4 μs.

$$\therefore \frac{250\ \mu s}{0.4\ \mu s} = 625,$$

0.4 μs intervals will make a 2 KHz pulse.

The values to be loaded into TMR0H and TMR0L are :

$$65536 - 625 = (64911)_{10} = \text{FD8FH}$$

$$\therefore \text{ TMR0H} = \text{FDH}$$

$$\text{TMR0L} = \text{8FH}$$

C18 program :

```
#include <P18F458.h>
void delay (void)
#define mybit PORTBbits.RB5
void main (void)
  {
      TRISBbits.TRISB5 = 0 ;     // RB5 = output bit
      while (1)
      {
         mybit = ~ mybit ;       // toggle bit RB5
         delay ( ) ;
      }
  }
  void delay ( )
  {
      T0CON = 0x08 ;     // Timer 0, 16 bit mode, no
                              prescaler
      TMR0H = 0xFD ;          // Load TMR0H
      TMR0L = 0x8F ;          // Load TMR0L
      T0CONbits.TMR0ON = 1 ;     // Start Timer 0
      while (INTCONbits.TMR0IF = = 0) ;
                        // wait for TMR0IF to roll over
      T0CONbits.TMR0ON = 1 ;     // stop Timer 0
      INTCONbits.TMR0IF = 0 ;
                              // Clear TMR0IF flag
  }
```

Ex. 7.7.3

Write a C18 program to toggle all the bits of PORTC continuously with some delay. Use Timer 0, 16 bit mode, no prescaler to generated the delay.

Soln. :

```
#include<P18F4580.h>
void delay (void) ;
void main (void)
  {
      TRISC = 0 ;              // Port C = output port
      while (1)
      {
         PORTC = 0xFF ;   //Toggle all bits of port C
         Delay ( ) ;
         PORTC = 0x00 ;   // Toggle all bits of port C
         Delay ( ) ;
      }
  }
  void Delay ( )
  {
      T0CON = 0x08 ;          // Timer 0, 16 bit mode,
                                 no prescaler
      TMR0H = 0x50 ;          // Load TMR0H
      TMR0L = 0x00 ;          // Load TMR0L
      T0CONbits. TMR0ON = 1 ;     // Start Timer 0
      while (INTCONbits.TMR0IF = = 0) ;
                        // wait till TMR0IF flag rolls over
      T0CONbits.TMR0ON = 0 ; // Stop Timer 0
      INTCONbits.TMR0IF = 0 ; // Clear TMR0IF flag
  }
```

Ex. 7.7.4

Write a C18 program to toggle RB5 continuously every 50 ms. Use Timer 0, 16 bit mode, 1 : 4 prescaler to create the delay. Let XTAL = 10 MHz.

Soln. :

$$\text{XTAL} = 10 \text{ MHz}$$

$$\text{Prescaler} = 1 : 4$$

$$\therefore \text{ Timer clock frequency} = \frac{f_{osc}}{4} \times \frac{1}{4} = 0.625 \text{ MHz}$$

$$\therefore \text{ Timer clock period} = 1.6 \ \mu s$$

To get delay of 50 msec,

$$\frac{50 \text{ msec}}{1.6 \ \mu\text{sec}} = 31250$$

$$\therefore \text{ Count} = 65536 - 31250 = (34286)_{10} = (85EE)_H$$

$$\therefore \text{ TMR0H} = 85H$$

$$\text{TMR0L} = EEH$$

Program :

```
#include<P18F4580.h>
void delay (void) ;
# define bit 5 PORTBbits.RB5
void main (void)
  {
      TRISBbits.TRISB5 = 0 ;
                        // RB5 = 0 i.e. output pin.
  while (1)
  {
      bit5 = ~ bit5 ;     // RB5 = 1 toggle bit
      Delay ( ) ;
  }
}
  void delay ( )
  {
      T0CON = 0x01 ;       //Timer 0, 1 : 4 prescaler,
                              16 bit mode
      TMR0H = 0x85 ;          // Load TMR0H
      TMR0L = 0xEE ;          // Load TMR0L
      T0CONbits. TMR0ON = 1 ;     // Start Timer 0
      while (INTCONbits.TMR0IF = = 0) ;
  //Wait till Timer 0 flag rolls over
      T0CONbits.TMR0ON = 0 ;      // Stop Timer 0
      INTCONbits.TMR0IF = 0 ;
                              //Clear TMR0IF flag
}
```

Ex. 7.7.5

A switch is connected to pin PORTC.4. Write a C18 program to monitor the switch and create the following frequencies on pin PORTC 0.

SW = 0 : 500 Hz

SW = 1 : 750 Hz

Use Timer 0 with prescaler 1 : 64. Assume XTAL = 10 MHz.

Soln. :

$$\text{XTAL} = 10 \text{ MHz}$$

$$\text{Prescaler} = 1 : 64$$

$$\therefore \text{ Timer clock frequency} = \frac{f_{osc}}{4} \times \frac{1}{64}$$

$$= \frac{10 \text{ MHz}}{4} \times \frac{1}{64} = 39.0625 \text{ KHz}$$

$$\therefore \text{ Timer clock period} = \frac{1}{39.0625 \times 10^3}$$

$$= 2.56 \times 10^{-5} \text{ s}$$

Assuming duty cycle 50% ON period is same as off period so it is divided by 2.

For 500 Hz i.e. 2ms,

$$\frac{2 \text{ ms}}{2.56 \times 10^{-5} \times 2} = 39.0625$$

$$\therefore \text{ Count} = 65536 - 39 = (65497)_{10} = FFD9H$$

$$\therefore \text{ TMR0H} = FFH$$

$$\text{TMR0L} = D9H$$

For 750 Hz i.e. 1.33 ms

$$\frac{1.33 \text{ ms}}{2.56 \times 10^{-5} \times 2} = 26.041$$

$$\therefore \text{ Count} = 65536 - 26.04 = (65510)_{10} = FFE6H$$

$$\therefore \text{ TMR0H} = FFH$$

$$\text{TMR0L} = E6H$$

Program :

```
#include<P18F4580.h>
void delay (void)
#define mybit PORTBbits.RB5
void main (void)
  {
      TRISBbits.TRISB5 = 0 ;     //RB5 = output bit
      while (1)
      {
         mybit = ~ mybit ;       // toggle bit RB5
         delay ( ) ;
      }
  }
void delay ( )
  {
      T1CON = 0x30 ;          // Timer 1, 16 bit, 1 : 8
                                 prescaler
      TMR1H = 0xF3 ;              // Load TMR1H
      TMR1L = 0xCB ;              // Load TMR1L
      T1CONbits.TMR1ON = 1 ;      // Start Timer 1
      while (PIR1bits.TMR1IF = = 0) ;
                          // Wait for TMR1IF to roll over
      T1CONbits.TMR1ON = 0 ;      // Stop Timer 1
      PIR1bits. TMR1IF = 0 ;  // Clear TMR1IF flag
  }
```

Ex. 7.7.8 **SPPU - Dec. 2016, 8 Marks**

Write a program to generate 100 msec delay using Timer 1. What are the values to be loaded in T1CON, TMR1H, TMR1L ? Assume that XTAL = 8 MHz ?

Soln. :

$$\text{XTAL} = 8 \text{ MHz}$$

$$\text{Assuming prescaler} = 1:4$$

$$\text{Timer clock frequency} = \frac{8 \text{ MHz}}{4} \times \frac{1}{4} = 0.5 \text{ MHz.}$$

$$\therefore \text{ Timer clock period} = \frac{1}{0.5 \text{ MHz}} = 2 \ \mu\text{s}$$

$$\therefore \frac{100 \text{ ms}}{2 \ \mu\text{sec}} = 50000$$

$$\therefore \text{ Count} = 65536 - 50000$$

$$= (15536)_{10} = (3\text{CB0})_{\text{H}}$$

$$\therefore \text{ TMR1H} = 3\text{CH}$$

$$\text{TMR1L} = \text{B0H}$$

Program :

```
#include <P18F4580.h>
void delay (void) ;
void delay ( )
{
    T1CON = 0x20 ;              // Load T1CON Timer 1,
                                   1 : 4 prescaler, 16 bit mode
    TMR1H = 0x3C ;              // Load TMR1H
    TMR1L = 0xB0H ;             // Load TMR0L
    T1CONbits.TMR1ON = 1 ;      // Start Timer 1
    while (PIR1bits.TMR1IF = = 0)
                          // wait for TMR1IF flag to roll over
    T1CONbits.TMR1ON = 0 ;      // Stop Timer 1
    PIR1bits.TMR1IF = 0 ;       // Clear TMR1IF
}
```

7.7.3 Counter and Pulse Width Measurement

- The PIC18 Timers can also be used as counters. This feature is used to count the events occurring outside the PIC18 microcontroller. This feature can also be used to measure the frequency by measuring the number of pulses in 1 second.
- Another feature is that it can be used to measure the width of a pulse by programming the Timer.
- However, when the Timer is used as counter, a pulse outside the PIC18 microcontroller increments the TMRxH and TMRxL registers.
- If the T0CS bit in T0CON register is set, the timer 0 works as a counter and counts pulses fed at T0CKI (Timer 0 clock input) pin (RA4).
- If the TMR1CS = 1 in T1CON, the Timer 1 works as counter and counts pulses from RC0/T1CKI (Timer 1 clock input) pin.

Ex. 7.7.9

Assume that a 1 Hz external clock is fed to RA4 pin. Write a C18 program for counter 0 in 8 bit mode to count up and display the count on port D. Begin the count at 00H.

Soln. :

Program :

```
# include <P18F458.h>
# define mybit PORTAbits.RA4
void main (void)
  {
      TRISAbits.TRISA4 = 1 ;  // Make RA4 = 1
                                 i.e. input pin.
      TRISD = 0 ;             // Make Port D = output
      T0CON = 0x68 ;          // Timer 0, 8 bit,
                              external clock, no prescaler
      TMR0L = 0 ;             // Initialize count = 0
      while (1)
      {
      do
      {
         T0CONbits.TMR0ON = 1 ; // Start Timer 0
```

UNIT - V

CHAPTER 8

I/O Port Programming

Syllabus Topic : Port Structure with Programming

8.1 Port Structure with Programming

SPPU - Dec. 14, May 16

University Questions

Q. Explain PIC18FXXX port structure. **(Dec. 2014, 8 Marks)**

Q. Explain the function of port of PIC in detail. **(May 2016, 8 Marks)**

- The general purpose I/O ports available on the PIC 18FXX devices depends on the number of pins on the chip.
- Table 8.1.1 shows the number of ports in PIC18 Family members.

Table 8.1.1 : Number of ports in PIC18 family members

PIC Chip	Number of Pins	Number of Ports	Ports Available
PIC 18F1220	18 Pin	2	Port A, Port B
PIC 18F2220	28 Pin	3	Port A, Port B, Port C
PIC 18F458	40 Pin	5	Port A, Port B, Port C, Port D, Port E
PIC 18F6525	64 Pin	9	Port A, Port B, Port C, Port D, Port E, Port F, Port G, Port H, Port J
PIC 18F8525	80 Pin	11	Port A, Port B, Port C, Port D, Port E, Port F, Port G, Port H, Port J, Port K, Port L

- **PIC 18F458** is a 40 pin IC having **5 I/O ports** viz. **Port A, Port B, Port C, Port D** and **Port E.**
- Some of the pins of I/O ports are multiplexed with an alternate function from the peripheral features on the device.
- Whenever a peripheral is enabled that pin cannot be used as a general purpose I/O pin.
- For using the ports as an input or output we need to program the ports.
- Each PORT has three SFRs for its operation. They are :

(i) PORTx (reads the levels on the pins of the device)
(ii) TRISx (data direction register)
(iii) LATx (output latch)

Eg. for Port D we have PORTD, TRISD and LATD SFRs. TRISx register is used for making the port an input port or output port.

- The LATx register is used for read-modify-write operations.
- Table 8.1.2 lists the SFR addresses of ports A-E for PIC 18F458.

Table 8.1.2 : SFR addresses of ports A-E for PIC 18F458

Port	Address
PORT A	F80H
PORT B	F81H
PORT C	F82H
PORT D	F83H
PORT E	F84H
LATA	F89H
LATB	F8AH
LATC	F8BH
LATD	F8CH
LATE	F8DH
TRISA	F92H
TRISB	F93H
TRISC	F94H
TRISD	F95H
TRISE	F96H

8.2 TRIS Register

- The PIC 18F458 Ports A-E can be used as input ports or output ports.

Use : The SFR TRISx is used for **configuring** a PIC 18F458 **port as input or output port**. Eg. To make a port as input port we need to write 1s to the TRISx register and to make a port as output port we need to write 0s to the TRISx register.

- All the PIC 18F458 ports have values FFH on reset i.e. **on reset all the ports are configured as input ports**.

8.2.1 Reading a Pin WhenTRISx = 1 (Input)

- We know that to make a port an input, we need to write a 1 into the TRISx register. Fig. 8.2.1 shows reading a 0 from a PIC18 pin and Fig. 8.2.2 shows reading a 1 from the PIC18 pin when TRIS = 1. Fig. 8.2.1 shows the structure of a PIC18 port along with its components.

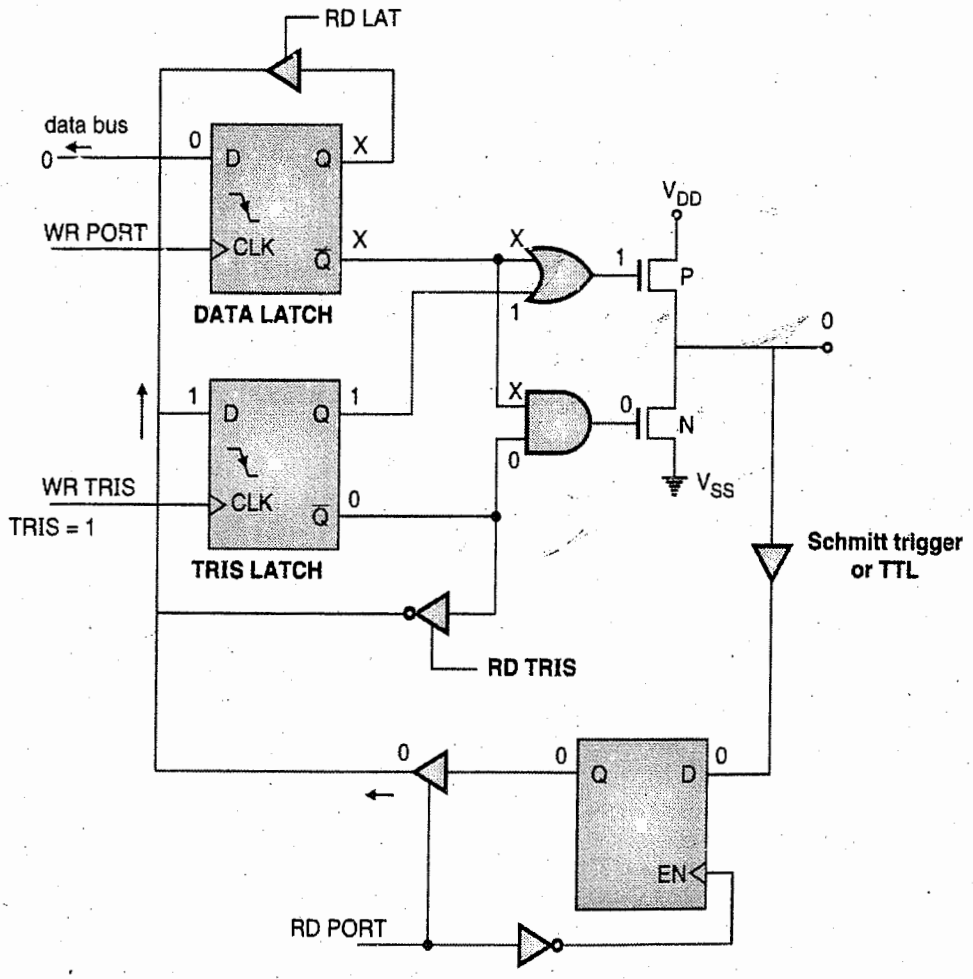


Fig. 8.2.1 : Reading a 0 from PIC18 port 1 pin

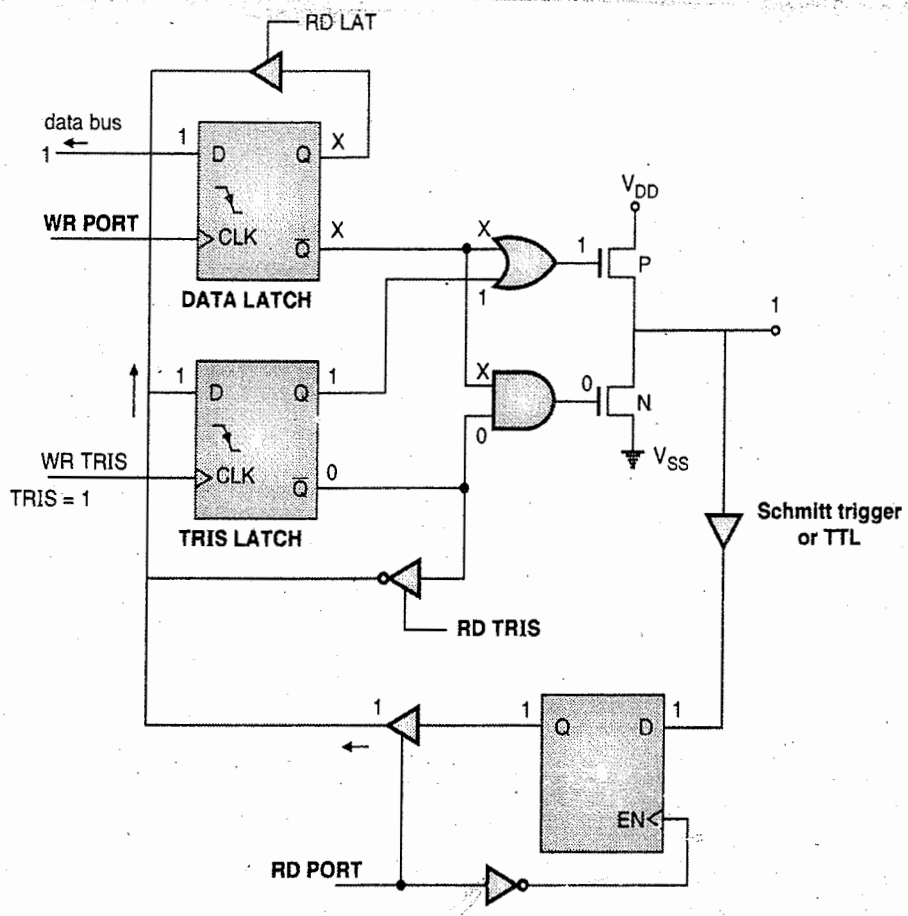


Fig. 8.2.2 : Reading a 1 from PIC18 port pin

The sequence of events while reading at the port pin are as follows :

Step I : Write a 1 on the TRIS Latch. This will make its Q output high i.e. 1 and $\overline{Q}$ output will be low.

Step II : As Q = 1, the P transistor will turn off and as $\overline{Q}$ = 0, transistor N will turn off. As both transistors are off, they will block the path towards V_{DD} or ground and the input signal passes to the buffer.

Step III : Thus, while reading the data from input port we are reading the data at that pin.

Note :

While reading data from the input port there are two possibilities.

(i) Some instructions read the status of internal latch referred as LATx.

(ii) Some instructions read the status of input port pin. Hence, to avoid errors in programming we must be able to differentiate between the two instruction possibilities.

– The instructions that read the status of input port are as follows :

(i) MOVFW PORTx (ii) BTFSS f, b (iii) BTFSC f, b (iv) TSTFSZ f (v) CPFSEQ f

8.2.2 Writing to a Pin When TRISx = 0 (Output)

We know that for making a port an output port we need to write a 0 into the TRISx register of PIC18F458. Fig. 8.2.3 shows writing a 0 to PIC 18 pin. The sequence of events while writing data on the port pin are as follows :

Step I : Write a 0 on the data latch. This will make Q = 0 and $\overline{Q}$ output will be 1.

Step II : As $\overline{Q}$ = 1, the P transistor will turn off and N transistor will turn on. As the N transistor is on, it provides a path towards ground to the input pin. Thus, if the programmer tries to read data from the PIC 18 pin, it will receive a low signal.

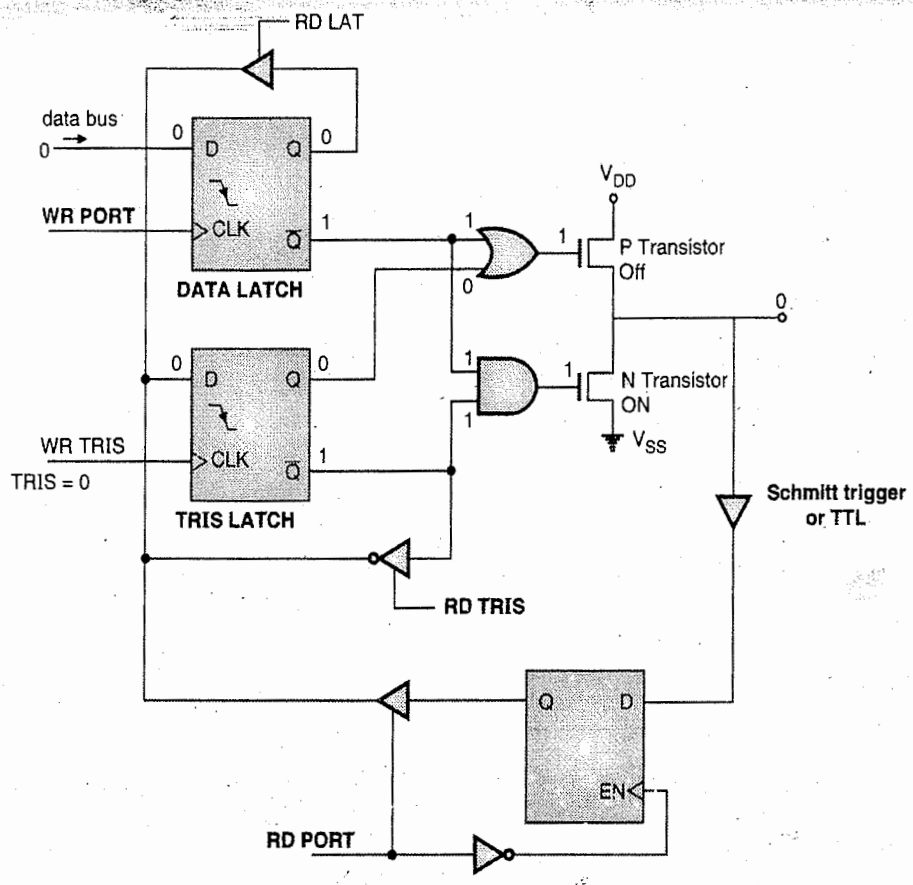


Fig. 8.2.3 : Writing a 0 to PIC18 pin

8.3 Reading LATx for PORTS

Use : The LATx SFR is used for read-modify-write operations.

- We know that some instructions read the status of internal latch. Such instructions generally read a latch value, do the required operation and rewrite the value back to the port latch. Table 8.3.1 lists these instructions. They are called as **Read-write- modify instructions.**

Table 8.3.1 : Read-write-modify instructions

Instruction	Function
1) ADDWF f, [d[,a]]	Add Wreg and file Reg.
2) BSF f, b[,a]	Set bit
3) BCF f, b[,a]	Clear bit of a file register
4) INCF f, [d,[a]]	Increment
5) COMF f[d,[a]]	Complement a file register
6) XORWF f[d,[a]]	Logical exclusive OR WREG and file register
7) SUBWF f[d,[a]]	Subtract Wreg from file register

8.4 Port A

SPPU - Dec. 14, May 16

University Questions

Q. Explain Port A of PIC18FXXX. **(Dec. 2014, 8 Marks)**

Q. Explain function of port A of PIC in detail. **(May 2016, 8 Marks)**

Number of pins : Port A is a 7 bit wide bidirectional port with 7 pins RA0 to RA6.

- The port A pins are multiplexed with the analog to digital converter. Hence, it has alternate functions as listed in Table 8.4.1.

Table 8.4.1 : Port A Alternate functions

Port Bit	Function
RA0	AN0 / CV_{ref}
RA1	AN1
RA2	AN2 / V_{ref}^-
RA3	AN3 / V_{ref}^+
RA4	T0CKI
RA5	AN4 / SS / LVDIN
RA6	OSC2 / CLK0

8.4.1 Port A as Simple Input Port

- When port A is used as simple input port "1" must be written in all the bits of the TRISA register.
- In the following code Port A is configured as an input port by writing 1s to the TRISA register. The data obtained will be stored in RAM location of the file register.

8.4.2 Port A as Output

- If we write 0's to all the bits in TRISA register, Port A will be configured as an output port.

8.5 Port B

SPPU - Dec. 14, May 16

University Questions

Q. Explain port B of PIC18FXXX. (Dec. 2014, 8 Marks)

Q. Explain function of port B of PIC in detail. (May 2016, 8 Marks)

Number of pins : Port B is an 8 bit bidirectional port with 8 pins RB0-RB7.

- In order to minimize the pins, the PIC18 Port B pins are multiplexed to some other functions. Table 8.5.1 lists the alternate functions of port B.

Table 8.5.1 : Port B alternate functions

Port Bit	Function
RB0	INT0
RB1	INT1
RB2	INT2 / CANTX
RB3	CANRX
RB4	–
RB5	PGM
RB6	PGC
RB7	PGD

8.5.1 Port B as Input Port

- When port B is used as a simple input port "1" must be written in the TRISB register

8.5.2 Port B as Output

- If we write 0's to all the bits in the TRISB register, Port B will be configured as an output port.

8.6 Port C

SPPU - Dec. 14, May 16

University Questions

Q. Explain port C of PIC18FXXX. (Dec. 2014, 8 Marks)

Q. Explain function of port C of PIC in detail. (May 2016, 8 Marks)

Number of pins : Port C is an 8 bit wide bidirectional port with 8 pins RC0-RC7.

- Table 8.6.1 lists the alternate functions of port C.

Table 8.6.1 : Port C alternate functions

Port Bit	Function
RC0	T10S0 / T1CK1
RC1	T10SI
RC2	CCP1
RC3	SCK/SCL
RC4	SDI/SDA
RC5	SDO
RC6	TX/CK
RC7	RX / DT

8.6.1 Port C as Input

- When port C is used an input port "1" must be written in all the bits of TRISC register

8.6.2 Port C as Output

- If we write 0's to all the bits in the TRISC register, port C will be configured as an output port.

8.7 Port D

SPPU - Dec. 14, May 16

University Questions

Q. Explain port D of PIC18FXXX. (Dec. 2014, 8 Marks)

Q. Explain function of port D of PIC in detail. (May 2016, 8 Marks)

Number of pins : PORT D is an 8 bit bidirectional port with 8 pins RD0-RD7.

- Table 8.7.1 lists the alternate functions of PORT D

Table 8.7.1 : Port D alternate functions

Port Bit	Function
RD0	PSP0 / C1IN +
RD1	PSP1 / C1IN–
RD2	PSP2 / C2IN+
RD3	PSP3/C2IN–
RD4	PSP4 / ECCP1 / P1A
RD5	PSP5 / P1B
RD6	PSP6 / P1C
RD7	PSP7 / P1D

8.7.1 Port D as Input

- When Port D is used as input port a "1" must be written to all the bits in the TRISD register.

8.7.2 Port D as Output

- If we write 0's to all the bits in the TRISD register, Port D will be configured as an output port.

8.8 Port E

SPPU - Dec. 14, May 16

University Questions

Q. Explain port E of PIC18FXXX. **(Dec. 2014, 8 Marks)**

Q. Explain function of port E of PIC in detail. **(May 2016, 9 Marks)**

Number of pins : Port E is a 3-bit wide, bidirectional port. **Port E has three pins (RE0/AN5/RD, RE1/AN6/WR/C1OUT and RE2/AN7/CS/C2OUT)** which are individually configurable as inputs or outputs. These pins have Schmitt Trigger input buffers.

- Read-modify-write operations on the LATx register, read and write the latched output value for Port E. The corresponding Data Direction register for the port is TRISE.
- Setting a TRISE bit (= 1) will make the corresponding Port E pin an input (i.e., put the corresponding output driver in a high-impedance mode).
- Clearing a TRISE bit (= 0) will make the corresponding PORTE pin an output (i.e., put the contents of the output latch on the selected pin).
- The TRISE register also controls the operation of the Parallel Slave Port through the control bits in the upper half of the register. When the Parallel Slave Port is active, the Port E pins function as its control inputs.

8.9 Port Status Upon Reset

- All the PIC18F458 ports have value FF H on their TRIS register on reset. **i.e. all the ports are configured as input ports on Reset.** Table 8.9.1 gives the Reset values of TRIS registers for PIC18F458.

Table 8.9.1 : Reset value of TRIS registers for PIC18

Register	Reset Value (Binary)
TRIS A	11111111
TRIS B	11111111
TRIS C	11111111
TRIS D	11111111

Program 1

Write a C18 program to toggle all the bits of port C continuously.

```
#include <P18F458.h >
void main (void)
{
    TRISC = 0          // Make Port C as output port
    for (; ;)          // repeat forever
    {
    PORTC = 0x55 ;
                       // 0x indicates data is in hex (binary)
    PORTC = 0xAA ;
    }
}
```

Program 2

Write a C program to toggle all the bits of Port D continuously.

```
# include < P18F458.h>
void main (void)
{
    TRISD = 0 ;        // Make Port D an output port
    for (; ;)
    {
    PORTD = 0x55 ;
    PORTD = 0xAA ;
    }
}
```

Program 3

Write a C program to toggle all bits of Port A 10,000 times.

```
# include < P18F458.h>
void main (void)
{
    unsigned int i ;
    TRISA = 0;         // Port A is output port
    for (i = 0 ; i < = 10,000 ; i ++)
    {
        PORTA = 0x55 ;
        PORTA = 0xAA ;
    }
    while (1) ;
}
```

Program 4 SPPU- Dec. 2013, 4 Marks

Write a C program to toggle all the bits of Port A continuously with a 250 ms delay. Assume that the system is PIC18F458 with XTAL = 10 MHz.

```
# include < P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISA = 0 ;        // Port A is output port forever
    while (1)
    {
    PORTA = 0x55 ;
```

```
        Delay (250) ;
        PORTA = 0xAA ;
        Delay (250) ;
        }
    }
    void Delay (unsigned int xtime)
    {
        unsigned int i ;
        unsigned char j ;
    for (i = 0 ; i < xtime ; i ++)
    for (j = 0 ; j < 165 ; j ++) ;
    }
```

When we use functions like DELAY in the C program we need to know following points :

(i) Before the main function, the declaration :

void Delay (unsigned int)

will tell the compiler that there will be a function called DELAY. It is called **function prototype**.

(ii) The functions are generally defined immediately after the main program ends as in program 4. The first line of function declaration must be exactly same as its function prototype.

Program 5 SPPU - May 2014, 8 Marks

Write a C program to toggle all bits of Ports B, C, D continuously with a 250 ms delay. Assume XTAL = 10 MHz.

```
# include <P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISB = 0 ;
    TRISC = 0 ;   } Make Ports A, B, C and D as
    TRISD = 0 ;     output ports forever
while (1)
    {
        PORTB = 0x55 ;   // Toggle ports A, B, C, D
        PORTC = 0x55 ;
        PORTD = 0x55 ;
        Delay (250) ;
        PORTB = 0xAA ;
        PORTC = 0xAA ;
        PORTD = 0xAA ;
        Delay (250) ;
    }
}
void Delay (unsigned int xtime)
{
    unsigned int i ;
    unsigned char j ;
    for (i = 0 ; i < xtime ; i ++)
    for (j = 0 ; j < 165 ; j ++ ) ;
}
```

❑❑❑

UNIT - V

CHAPTER 9

Interrupt Structure of PIC18F

9.1 Interrupts Vs Polling

- Whenever more than one I/O devices are connected to a microcontroller based system, any one of I/O devices can ask service at a given time. There are two methods in which the microcontroller can service the I/O devices. One method is to use the **polling routine**, while the other uses **interrupt method**.
- In the polling routine the microcontroller checks if any of the I/O devices needs service.
- The polling routine is a simple program that checks the status of the I/O devices and if the condition is satisfied, it provides service. e.g. let us assume that the polling routine is servicing the I/O ports in sequence. Initially the polling routine will transfer the status of port A to the WREG register. It then checks the contents of WREG register to determine if the service request bit is set. If the bit is set then I/O port 1 service routine is called, otherwise the polling routine will move forward to check if port B is requesting service. On completion of service to port A, the polling routine will test port B. The process is repeated till all the ports are tested and offered service. After completion of the polling routine, the microprocessor will continue the program execution.
- However, the **drawback of the polling routine** is that it can assign priority to the I/O devices as they are checked one after the other, secondly, it wastes the time by polling for devices that don't require service.
- Another method that allows the microcontroller to stop with the execution of the current program and give service to the I/O devices is called as **interrupt**.

Definition : An **interrupt** is an external asynchronous input that informs the microcontroller to the complete the instruction that it is currently executing and fetch a new routine inorder to offer service to the I/O device. Once the I/O device is serviced, the microcontroller will continue with the execution of the program from where it was interrupted.

- The **advantages** of interrupt method are :
  (i) The microcontroller can service many devices depending on the priority assigned to these devices.
  (ii) The microcontroller can ignore a device request for service.

9.2 Interrupt Service Routine (ISR)

SPPU – Dec. 16

University Question

Q. What are peripheral interrupts, IVT and ISR ? **(Dec. 2016, 2 Marks)**

- Each interrupt requires an **interrupt handler** or an **Interrupt Service Routine (ISR).**
- Whenever an interrupt is involved, the microcontroller executes an interrupt service routine. Every interrupt has a fixed location in memory that holds the address of the ISR. The group of memory locations for holding the addresses of the ISRs is called as **Interrupt Vector Table (IVT).**
- An interrupt generated by a peripheral (Timer, ADC etc.) is called as peripheral interrupt.

PIC18 locations for IVT :

PIC18 microcontroller has only two locations 0008H and 0018H for the interrupt vector table.

Table 9.2.1 : IVT for PIC18

Interrupt	ROM location
Power on reset	0000H
High priority interrupt	0008H
Low priority interrupt	0018H

9.3 Steps in Executing an Interrupt

Whenever an interrupt is invoked, the microcontroller performs the following steps :

Step I : The microcontroller executes the current instruction and saves the address of next instruction on the stack.

Step II : The microcontroller saves the current status of all the interrupts internally.

Step III : The microcontroller jumps to a fixed location in the memory called as interrupt vector table. The interrupt vector table holds the address of the interrupt service routine (ISR).

Step IV : The microcontroller jumps to the address of ISR from interrupt vector table. The microcontroller executes the ISR till it reaches the last instruction of the subroutine i.e. RETFIE (return from interrupt exit).

Step V : On receiving the RETFIE instruction, the microcontroller returns to the main program where it was interrupted. The microcontroller fetches the address of program counter from the stack. The microcontroller then executes the main program from that address.

Note : Step V is the critical role of the stack. Hence ,while modifying the contents of stack in the ISR, the number of pushes and pops must be equal.

Syllabus Topic : Interrupt Structure (Legacy and Priority Mode) of PIC with SFRs

9.4 Interrupt Structure (Legacy and Priority Mode) of PIC with SFRs

SPPU : Dec. 15, Dec. 16

University Questions

Q. Explain interrupt structure of PIC18FXXX. **(Dec. 2015, 8 Marks)**

Q. Draw and explain interrupt structure for PIC18FXXX microcontroller. **(Dec. 2016, 6 Marks)**

The PIC18 interrupts are :

(i) **Reset.**
(ii) **Timer 0 (TMR0IF), Timer 1 (TMR1IF), Timer 2 (TMR2IF) and Timer 3 (TMR3IF) interrupt.**
(iii) **External hardware interrupts INT0, INT1 and INT2.**
(iv) **Serial communication interrupts TXIF and RCIF.**
(v) **PORTB – change interrupt RBIF.**
(vi) **ADC interrupt- ADIF.**
(vii) **Compare capture pulse width modulation (CCP) interrupts : CCP1IF**

Fig. 9.4.1 shows the PIC18 interrupts.

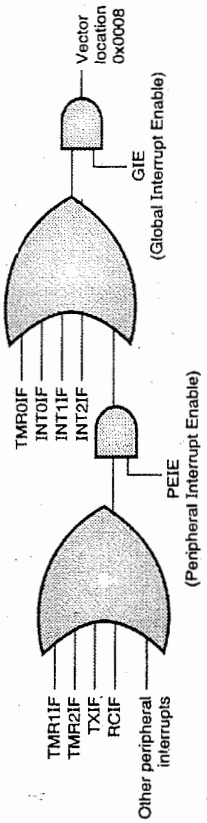


Fig. 9.4.1 : PIC18 interrupts

- The PIC18FXX8 devices have multiple interrupt sources. Every interrupt source can be assigned either a low priority or high priority. On reset all interrupts are assigned a high priority.
- For controlling the interrupt operating, the SFR registers used are :

1. **RCON**
2. **INTCON**
3. **INTCON2**
4. **INTCON3**
5. **PIR1, PIR2, PIR3**
6. **PIE1, PIE2, PIE3**
7. **IPR1, IPR2, IPR3**

- It is recommended that the Microchip header files, supplied with MPLAB® IDE, be used for the symbolic bit names in these registers. This allows the assembler/compiler to automatically take care of the placement of these bits within the specified register.
- Inorder to control the interrupt operation, every interrupt has three sources. The sources are responsible for :

(a) The interrupt flag bit indicates an interrupt event that took place.
(b) Interrupt enable bit that allows program execution to branch to the interrupt vector address when the flag bit is set
(c) Interrupt priority bit to assign high priority or low priority to the interrupts.

- The interrupt priority feature is enabled by setting the IPEN bit in the RCON register.
- When interrupt priority is enabled, there are two bits that enable interrupts globally.

9.4.1 Timer Flag Interrupts

- When the timer/counter overflows, the corresponding timer flag **TMR0IF, TMR1IF, TMR2IF and TMR3IF.** The flag is cleared to 0, when the interrupt generates a program call to the timer subroutine in the memory.

9.4.2 Serial Port Interrupts

- The serial port interrupt is generated because of **TXIF or RCIF**. One interrupt is used for sending the data byte and the other whenever a data byte is received.
- If the TXIE or RCIE flags in the PIE register are enabled when TXIF or RCIF are 1, the PIC18 microcontroller is interrupted and goes to 0008H location for executing the interrupt service routine.

9.4.3 External Hardware Interrupts

- The PIC18 supports **three external hardware interrupts**. They are INT0, INT1 and INT2. These interrupts are located on pins RB0, RB1 and RB2. They are directed to vector location 0008H.
- These interrupts can be enabled by setting the INTxIE bit in the INTCON and INTCON3 registers.
- On reset, the microcontroller configures these interrupts as positive edge triggered interrupts.

9.4.4 PORT B-Change Interrupts

- The pins RB_4 – RB_7 of port B can invoke an interrupt whenever any modifications are detected on the respective pin. The interrupt is called **"PORTB-change interrupt"**.
- These interrupts have a single interrupt flag RBIF in the INTCON register. They can be enabled by the RBIE bit in the INTCON register.
- Even though PORT-B change interrupt can use four port B pins, it is considered to be a single interrupt.
- This interrupt is used mainly for keyboard interfacing.

9.5 Enabling and Disabling an Interrupt

- On reset, all the interrupts are disabled. Even if the interrupts are activated they will not be responded by the microcontroller on reset.
- These interrupts need to activated by software so that the microcontroller can service the interrupts.
- The interrupts can be enabled or disabled by modifying the GIE bit in the INTCON register.

Size : Fig. 9.5.1 shows the INTCON register. It is a 8 bit addressable register. It contains GIE bit, that disables all the interrupts at once.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
GIE	PEIE	TMR0IE	INT0IE				

GIE (Global Interrupt Enable)

If GIE = 1, interrupts are enabled. Every interrupt source is enabled by setting the respective interrupt enable bit.

If GIE = 0, all interrupts are disabled, and none of them will be acknowledged.

TMR0IE : Timer 0 interrupt enable

If TMR0IE = 0 Disable Timer 0 overflow interrupt
TMR0IE = 1 Enable Timer 0 overflow interrupt

INT0IE : External Interrupt 0 enable or disable

INT0IE = 0 Disable external interrupt zero
INT0IE = 1 Enable external interrupt 0

Fig. 9.5.1 : INTCON (interrupt control) register

- For servicing an interrupt, the IE (interrupt enable) flag bit for that interrupt must be set with the GIE bit.
- After the interrupt is activated the GIE bit is cleared. This ensures that the microcontroller will not service any other interrupt.
- After servicing the interrupt, the RETFIE instruction will activate the GIE bit, so that microcontroller can service other interrupts also.
- The PEIE (Peripheral Interrupts Enable) bit must also be enabled if peripherals like Timers, Serial port etc. are used.

9.6 Steps in Enabling an Interrupt

- To enable an interrupt the following steps are to be considered :

Step I :	Set the GIE bit of INTCON to enable interrupts.
Step II :	Set the corresponding IE (interrupt enable) bit for that interrupt e.g. INT0IE will enable external interrupt 0. If the GIE bit is not set, then no interrupt will be responded even if the bit in the INTCON register is set.

Step III : Set the PEIE bit for peripheral interrupts like TMR0IF, TMR1IF, TMR2IF, TXIF. The GIE bit must also be set.

9.7 PIC18 Interrupt Programming in C Using C18 Compiler

- The C18 compiler uses the **#pragma directive** to place code at a particular ROM address.
- For transferring control to the ISR we need to use assembly language instruction GOTO, because the C18 compiler does not place ISR at the interrupt vector table automatically. It is done as :

```
#pragma code hi_priori = 0x0008
                        // high priority interrupt location
void hi_priori (void)
  {
      _asm
      GOTO high_isr
      _endasm
  }
      #pragma code          //end code
```

To compute the interrupt source we are directed from location 0008H to other program using keyword interrupt as follows :

```
# pragma interrupt high_isr
void high_isr (void)
{
}
```

Syllabus Topic : Use of Timers with Interrupts

9.8 Use of Timers with Interrupts

- The Timer 0, Timer 1, Timer 2 and Timer 3 interrupts are generated by TMR0IF, TMR1IF, TMR2IF and TMR3IF. These bits are set by rollover in the respective Timer registers.
- When a timer interrupt is generated, the flag that generated it, is cleared by the on chip hardware when the interrupt service routine is vectored.
- In earlier chapters, we have seen the use of Timers 0, 1, 2 and 3 with the polling method. The TMRxIF flag is set whenever the timer rolls over.
- In the polling method TMRxIF flag is monitored and the user has to wait till the TMRxIF flag is activated. This is the main drawback of the polling method.
- This drawback can be overcome by the interrupt method. If the timer interrupt is set, then the TMRxIF flag is set whenever the timer is rolled over and the PIC microcontroller is interrupted. Thus, PIC18 can perform any operation till it is interrupted. Upon interruption, the PIC18 is busy executing the interrupt service routine.

Table 9.8.1 : Timer interrupt flag bits and their registers

Timer	Enable bit	Register	Interrupt flag bit	Register
Timer 0	TMR0IE	INTCON	TMR0IF	INTCON
Timer 1	TMR1IE	PIE 1	TMR1IF	PIR1
Timer 2	TMR2IE	PIE 1	TMR2IF	PIR1
Timer 3	TMR3IE	PIE 2	TMR3IF	PIR3

Ex. 9.8.1 Lab assignment

Write a program to generate square wave 2 KHz with Timer 0 on pin PORTB.5 with interrupt.

Soln. :

The period of square wave $= \frac{1}{2\text{ KHz}} = 500\ \mu s$

Let us assume the duty cycle of square wave is 50%. Hence, the square wave will be high for 250 μs and low for 250 μs.

Let XTAL = 10 MHz.

$\therefore$ Timer clock frequency $= \frac{f_{osc}}{4} = \frac{10\text{ MHz}}{4} = 2.5\text{ MHz}$

Timer clock period $= \frac{1}{2.5\text{ MHz}}$

= 0.4 μs i.e. counter will count up every 0.4 μs.

$\therefore \frac{250\ \mu s}{0.4\ \mu s} = 625$

$\therefore$ Count = 65536 – 625 = $(64911)_{10}$ = FDF8H

$\therefore$ TMR0H = FDH

$\therefore$ TMR0L = F8H

C18 Program :

```
#include <P18F4580.h>
#define mybit PORTBbits.RB5
void Timer 0_ISR (void) ;

#pragma interrupt my_isr
void my_isr (void)
  {
      if (INTCONbits.TMR0IF = = 1)
                        // if Timer 0 caused interrupt
                        execute Timer 0 ISR
      Timer 0_ISR ( ) ;
  }
      #pragma code hipriori_int = 0x08
                        // high priority interrupt
      void hipriori_int (void)
      {
        _asm
        GOTO my_isr
```

```
        _endasm
    }
    # pragma code
    void main (void)
    {
        TRISB bits.TRISB5 = 0 ; // Make RB5 output
        T0CON = 0x08 ;      // Timer 0, internal
                            clock, 16 bit mode, no
                            prescaler
        TMR0H = 0xFD ;      // Load TMR0H
        TMR0L = 0xF8 ;      // Load TMR0L
        INTCONbits.TMR0IF = 0 ;
                            // Clear TMR0IF flag
        INTCONbits. TMR0IE = 1 ;
                            // Enable Timer 0 interrupt
        T0CONbits. TMR0ON = 1 ;  // Start Timer 0
        INTCONbits.GIE = 1 ;
                            // Enable all interrupts globally
        INTCONbits.PEIE = 1 ;
                            // Enable all peripheral interrupts
}
void Timer 0_ISR (void)
{
    mybit = ~ mybit ;       // Toggle RB5
    TMR0H = 0xFD ;          // Load TMR0H
    TMR0L = 0xF8 ;          // Load TMR0L
    INTCONbits.TMR0IF = 0 ; // Clear TMR0IF flag
}
```

Ex. 9.8.2

Write a program to generate frequencies of 2 KHz and 10 KHz on pins RB3 and RB4 respectively. Assume crystal frequency = 10 MHz.

Soln. :

$$\text{Timer clock frequency} = \frac{f_{osc}}{4} = \frac{10\text{ MHz}}{4} = 2.5\text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5\text{ MHz}} = 0.4\ \mu\text{sec}$$

For 10 KHz (period = 0.1 msec)

∴ On period is 0.05 msec and off period is 0.05 msec, assuming 50% duty cycle.

$$\therefore \frac{0.05\text{ msec}}{0.4\ \mu\text{sec}} = 125$$

$$\text{Count} = 65536 - 125 = (65411)_{10} = \text{FF83H}$$

∴ TMR1H = FFH

TMR0L = 83H

For 2 KHz (period = 0.5 msec)

∴ On period is 0.25 msec and off period is 0.25 msec, assuming 50% duty cycle.

$$\therefore \frac{0.25\text{ msec}}{0.4\ \mu\text{sec}} = 625$$

$$\therefore \text{Count} = 65536 - 625 = (64911)_{10} = \text{FD8FH}$$

∴ TMR0H = FDH

TMR0L = 8FH

Program :

```
# include <P18F4850.h>
# define PB3 PORTBbits.RB3
#define PB4 PORTBbits.RB4
void Timer 0 _ISR (void) ;
void Timer 1_ISR (void) ;
#pragma interrupt my_isr          // high priority interrupt
void my_isr (void)
  {
      if (INTCONbits.TMR0IF = = 1)
                          // if Timer 0 caused interrupt
      Timer0_ISR ( ) ;          // execute Timer 0 ISR.
      if (PIRbits.TMR1IF = = 1)
                          // Else if Timer 1 caused interrupt
      Timer1_(ISR) ;            //execute Timer 1 ISR.
  }
  #pragma code hi_priori = 0x08
                          // high priority interrupt
  void hi_priori (void)
  {
      _asm
      GOTO my_isr
      _endasm
  }
  # pragma code
  void main (void)
  {
      TRISBbits.TRISB3 = 0 ;  // Make RB3 an output
      TRISBbits.TRISB4 = 0 ;  // Make RB4 an output
      T0CON = 0x08 ;          // Timer 0, 16 bit mode,
                              no prescaler
      TMR0H = 0xFDH ;         // Load TMR0H
      TMR0L = 0x8FH ;         //Load TMR0L
      T1CON = 0x88 ;          // Timer 1, 16 bit mode,
                              no prescaler
      TMR1H = 0xFF ;          // Load TMR1H
      TMR1L = 0x83 ;          // Load TMR1L
      INTCONbits.TMR0IF = 0 ;     //Clear Timer 0
                                  interrupt flag
      PIR1bits.TMR1IF = 0 ;       // Clear Timer 1
                                  interrupt flag
      INTCONbits.TMR0IE = 1 ;     // Enable Timer
                                  0 interrupt
      PIE1bits.TMR1IE = 1 ;       // Enable Timer 1
                                  interrupt
      T0CONbits.TMR0ON = 1 ;      //Start Timer 0
      T1CONbits.TMR1ON = 1 ;      // Start Timer 1
      INTCONbits.PEIE = 1 ;       // Activate all
                                  peripheral interrupts
      INTCONbits.GIE = 1 ;        // Globally enable all
                                  interrupts
      while (1) ;             // keep looping till interrupt is
                              recognized
```

```
}
void Timer0_ISR (void)
{
    PB3 = ~ PB3 ;          // Toggle RB3
    TMR0H = 0xFD ;         // Load TMR0H
    TMR0L = 0x8F ;         // Load TMR0L
    INTCONbits.TMR0IF = 0 ;
                           // Clear Timer 0 interrupt
}
void Timer1_ISR (void)
{
    PB4 = ~ PB4 ;          // Toggle RB4
    TMR1H = 0xFF ;         // Load TMR1H
    TMR1L = 0x83 ;         // Load TMR1L
    PIR1bits.TMR1IF = 0 ;// Clear Timer 1 interrupt
}
```

Ex. 9.8.3

Write a program that displays a value of 'Y' at port C and 'N' at Port D. It also generates a square wave of 5 KHz with Timer 1 at port pin RB6. Use XTAL = 10 MHz.

Soln. : Timer clock frequency = $\frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$

Timer clock period $= \frac{1}{2.5 \text{ MHz}} = 0.4\ \mu\text{sec}$

For 5 KHz (period = 0.2 msec.)

On period = 0.1 msec, off period = 0.1 msec, assuming 50% duty cycle.

$$\therefore \frac{0.1 \text{ msec}}{0.4\ \mu\text{sec}} = 250$$

$$\therefore \text{Count} = 65536 - 250 = (65286)_{10} = \text{FF06H}$$

$$\therefore \text{TMR1H} = \text{FFH}$$

$$\text{TMR1L} = \text{06H}$$

Program :

```
# include <P18F458.h>
# define PB6 PORTBbits.RB6
void Timer1_ISR ( ) ;
# pragma interrupt my_isr
void my_isr (void)
  {
      if (PIR1bits.TMR1IF = = 1)
                        // If Timer 1 caused interrupt
      Timer1_(ISR) ;          // execute Timer 1 ISR
  }
  #pragma code hi_priori = 0x08
  void hi_priori (void)
  {
      _asm
      GOTO my_isr
      _endasm
  }
      # pragma code
void main (void)
  {
      TRISBbits.TRISB6 = 0 ;          // RB6 = output
      PORTC = 0 ;     // Make Port C an output port
      PORTD = 0 ;     // Make Port D an output port
      T1CON = 0x88 ;
                  // 16 bit mode, Timer 1, no prescaler
      TMR1H = 0xFF ;          // Load TMR1H
      TMR1L = 0x06 ;          // Load TMR1L
      PIR1bits.TMR1IF = 0 ;
                          // Clear Timer 1 interrupt
      PIE1bits.TMR1IE = 0 ; // Enable Timer 1
                              interrupt
      T1CONbits.TMR1ON = 1 ;      // Start Timer 1
      INTCONbits.PEIE = 1 ;   // Enable peripheral
                              interrupts
      INTCONbits.GIE = 1 ;  //Enable interrupts
                              globally
      while(1)            //keep looping till interrupt
                            is recognized.
  {
      PORTC = 'Y'             //Display 'Y' at Port C
      PORTD = 'N'             //Display 'N' at Port D
  }
}
  void Timer1_ISR (void)
{
  PB6 = ~ PB6;                //toggle bit RB6
  TMR1H = 0xFF;               //Load TMR1H
  TMR1L = 0x06;               //Load TMR1L
  PIR1bits.TMR1IF = 0;        //Clear Timer 1 interrupt
}
```

9.9 Programming External Hardware Interrupts

- The PIC18 microcontroller has three external interrupts. They are INT0, INT1 and INT2.
- Whenever they are invoked, the microcontroller gets interrupted. The microcontroller stops the current program execution and jumps to the vector table to provide service to the interrupt.
- The interrupts INT0, INT1 and INT2 are located on pins RB0, RB1 and RB2 of port B. They are directed to vector location 0008H.
- The INT0 interrupt can be enabled/disabled using INT0IE bit of INTCON register.
- The INT1 and INT2 interrupts can be enabled and disabled using the INTxIE bits of the INTCON3 register.
- On reset the external hardware interrupts are positive edge triggered interrupts.

Table 9.9.1 : INT0, INT1, INT2 interrupt flag bits and their registers

Interrupt	Interrupt enable bit	Register	Interrupt flag bit	Register
INT0 (RB0)	INT0IE	INTCON	INT0IF	INTCON
INT1 (RB1)	INT1IE	INTCON3	INT1IF	INTCON3
INT2 (RB3)	INT2IE	INTCON3	INT2IF	INTCON3

- Inorder to make the interrupts negative edge triggered we need to program the INTEDGx bits in the INTCON2 register.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
	INTEDG0	INTEDG1	INTEDG2				

INTEDGx : External Hardware Interrupt Edge Trigger Bit
0 : Interrupt on negative edge
1 : Interrupt on positive edge

Fig. 9.9.1 : INTCON2 register

- As shown in Fig. 9.9.1 bits D_6, D_5 and D_4 hold the INTEDG0, INTEDG1, INTEDG2 bits. If bit = 0, the interrupt is negative edge triggered otherwise the interrupt is positive edge triggered.
- On reset all the INTEDGx bits are 1, to indicate positive edge triggered interrupts. To make an interrupt negative edge triggered we need to make the INTEDGx bit of that interrupt low i.e. 0.
- In case of an edge triggered external interrupt, the external source has to hold the request pin high for atleast two instruction cycles and then hold it low for atleast two instruction cycles to ensure that the transition is seen so that the interrupt request flag is set. When the interrupt service routine is called, the INTxIF bits must be cleared to indicate that the interrupt is service and PIC18 microcontroller can respond to another interrupt.

Assuming XTAL = 10 MHz

1 instruction cycle time = 0.4 μs = 400 ns

For detecting edge triggered interrupts we need two instruction cycles high and two instruction cycles low.

∴ Minimum pulse duration for detecting edge triggered interrupts = 2 instruction cycles (0.8 μs) + 2 instuction cycles (0.8 μs)

Minimum pulse duration for detecting edge triggered interrupts = 1.6 μs

Ex. 9.9.1

Write a C18 program to generate a square wave that is half the frequency of the signal applied at INT0 on pin PORTB.5.

Soln. :

Program :

```
# include <P18F4580.h>
# define PB5 PORTBbits.RB5
void my_isr (void);
void INT0_ISR(void) ;
#pragma interrupt my_isr
void my_isr(void)
  {
      if (INTCONbits.INT0IF = =1)
                //if external hardware interrupt 0
                 caused interrupt  execute INT0 ISR.
      INT0_ISR();
  }
      # pragma code hi-priori =0x08
            //interrupt location of high priority interrupt
  void hi_priori (void)
  {
      _asm
      GOTO my_isr
      _endasm
  }
# pragma code
void main (void)
  {
      TRISBbits.TRISB5 = 0 ;        // RB5 = output
      TRISBbits.TRISB0 = 1 ;        // INT0 = input
      INTCONbits.INT0IF = 0 ;
                        // clear External interrupt 0
      INTCONbits.INT0IE = 1 ;
                        // Enable External interrupt 0
      INTCONbits.GIE = 1 ;
                        // Enable all interrupts globally
      while(1) ;   // Keep looping till interrupt comes
  }
  void INT0_ISR (void)
  {
      PB5 = ~PB5;          // Toggle bit PB5
      INTCONbits. INT0IF = 0;
                        // Clear External interrupt 0 flag
  }
```

Ex. 9.9.2

Write a C18 program to switch "on" or "off" a LED connected to port B.6 when external interrupt INT1 is activated. Let INT1 be negative edge triggered.

Soln. : Program :

```
# include <P18F4580.h>
# define PB6  PORTBbits.RB6
void my_isr (void);
void Interrupt1_ISR(void) ;
#pragma code hi_priori=0x0008          // High priority
                                       interrupt location
void  hi_priori (void)
   {
        _asm
        GOTO my_isr
        _endasm
   }
   # pragma code
   #pragma interrupt my_isr
   void my_isr (void)
   {
        if (INTCON3bits.INT1IF ==1)
                        // If external interrupt 1 caused
        Interrupt1_ISR();        //Interrupt execute
                                 interrupt1_ISR
   }
   void main (void)
   {
        TRISBbits.TRISB6 = 0 ;          // RB6 = output
        TRISBbits.TRISB1 = 1 ;
                                 // RB1 i.c. INT1 = input
        INTCON3bits.INT1IF = 0 ;
                              // clear External interrupt 1
        INTCON3bits.INT1IE = 1 ;
                              // Enable External interrupt 1
        INTCON2bits.INTEDG1 = 0;
                     // Make INT1 negative edge triggered
        INTCONbits.GIE = 1;
                              // Enable all interrupts globally
   while (1) ;   // Keep looping till interrupt is recognized
   }
void Interrupt1_ISR(void)
   {
        PB6 = ~ PB6;            // Turn LED on and off
        INTCON3bits.INT1IF = 0;
                              // Clear External interrupt 1.
   }
```

9.10 Programming the Serial Communication Interrupts

- The PIC18 microcontroller has two interrupts reserved for serial communication TXIF and RCIF.
- The TXIF bit in the PIR1 register is set when a data byte is transmitted and RCIF bit in the PIR1 register is set when a data byte is received.
- The TXIF (Transfer interrupt) flag is set if the last bit of framed date (i.e. stop bit) is transmitted. It indicates that TXREG register is ready to transmit the next data byte.
- The RCIF (Received Interrupt) flag is set if a byte is present in the RCREG register. RCIF is set to indicate that the byte needs to be picked up before it is lost by new arriving serial data.
- All the above concepts are applied equally with polling or interrupts. The difference arises in the manner in which the serial communication interrupts are served.
- In the polling method, the TXIF or RCIF is monitored. We need to wait till the particular bit is set.
- In the interrupt method, when the microcontroller is ready to transmit or has received a byte we are notified. Any other task can also be completed while the serial communication is done.
- If the TXIE or RCIE bit in the PIE1 register is enabled, when the TXIF or RCIF are high the microcontroller gets interrupted. The PIC18 microcontroller the branches to memory location 0008H for executing the ISR.

Table 9.10.1 : Serial Port interrupt flag bits and their registers

Serial port interrupt	Enable bit	Register	Interrupt flag bit	Register
TXIF	TXIE	PIE1	TXIF	PIR1
RCIF	RCIE	PIE1	RCIF	PIR1

Ex. 9.10.1

Write a C18 program to take data from Port D and transfer it serially continuously. Use serial interrupt.

Soln. :

Program :

```
# include <P18F458.h>
void my_isr (void);
void transmit_ISR(void);
#pragma interrupt my_isr
void  my_isr (void)
{
    if  (PIR1bits.TXIF==1)        //If transmit interrupt
                                  execute transmit_ISR
    transmit_ISR( ) ;
}
#pragma code hi_priori = 0x08          //High priority
                                       interrupt.
```

```
void hi_priori (void)
  {
      _asm
      GOTO my_isr
      _endasm
  }
#pragma code
void main(void)
  {
      TRISD = 0xFF;          //Port D = Input.
      TRISCbits.TRISC6 = 0;
                             //Make TX pin output pin.
      TXSTA=0x20;            //Enable transmit
      SPBRG = 15 ;           //XTAL=10MHz, baud
                             rate = 9600
      RCSTAbits.SPEN = 1;    //Enable serial port
      TXSTAbits.TXEN=1;      //Enable transmit
      PIE1bits.TXIE =1;      //Enable transmit
                             interrupt.
      INTCONbits.PEIE = 1;   //Enable all peripheral
                              interrupt
      INTCONbits.GIE = 1;    //Enable all interrupts
                              globally.
      while (1);             //Keep looping till interrupt is
                             recognized.
  }
  void transmit_ISR(void)
  {
      TXREG = PORTD;    //Send data from Port B
                        serially to TXREG register.
  }
```

Ex. 9.10.2

Write a program that continuously read 8 bit data from port D and transmits it serially, while the incoming data from serial port is sent to port B. Let XTAL = 10 MHz and baud rate = 9600

Soln. : Program :

```
# include <P18F458.h>
void my_isr (void);
void transmit_isr (void);
void receive_isr (void);
#pragma interrupt my_isr
void  my_isr (void)
  {
  if (PIR1bits.TXIF==1)        // if transmit interrupt,
                               execute transmit_isr

  transmit_isr( );
  if (PIR1bits.RCIF ==1)       // if receive interrupt,
                               execute receive_isr.

  receive _isr( );
  }
```

```
  #pragma code hi_priori = 0x08        // high priority
                                        interrupt.
  void hi_priori (void)
  {
  _asm
  GOTO my_isr
  _endasm
  }
#pragma code
void main(void)
  {
      TRISD = 0xFF;          //port D = Input.
      TRISB = 0x00;          //Port B = Output.
      TRISCbits.TRISC6 = 0;          //Make transmit
                                     pin = output
      TRISCbits.TRISC7=1;    //Make receive pin input
      TXSTA=0x20;            //Enable transmit
      SPBRG = 15 ;           //Set baud rate = 9600
      RCSTAbits.CREN = 1;    //Enable receive
      RCSTAbits.SPEN =1;     //Enable serial port.
      TXSTAbits.TXEN=1;      //Enable transmit
      PIE1bits.TXIE =1;      //Enable transmit interrupt.
      PIE1bits.RCIE=1;       //Enable receive interrupt
      INTCONbits.PEIE = 1;   //Enable all
                              peripheral interrupts
      INTCONbits.GIE = 1;    //Globally enable all
                              interrupts
      while (1);
  }
void transmit_isr(void)
  {
      TXREG = PORTD;         //Send Port D value
                             serially
  }
void receive_isr (void)
  {
      PORTB=RCREG;           // Receive serial data
                             at Port B
  }
```

9.11 Port B-Change Interrupt

- If a change is observed on any of the pins RB4-RB7 of port B, an interrupt is generated. This interrupt is called as **Port B change interrupt.**
- It can be enabled and disabled by the RBIE bit in the INTCON register. RBIF is the port B change interrupt flag bit located in INTCON register.
- Although Port B change interrupt uses four pins, it is considered to be a **single interrupt.**
- Mostly Port B change interrupt is used for keyboard interfacing.

Ex. 9.11.1

Two switches are connected to pins RB6 and RB7 and two LEDs are connected to Port D bits RD6 and RD7. Write a program that will change the state of two LEDs depending on the activation of switches.

Soln. :

Program :

```
# include <P18F458.h>
#define LED1 PORTDbits.RD6
#define LED2 PORTDbits.RD7
void PortBInt_ISR( );
void n.y_isr( );
#pragma interrupt my_isr
void  my_isr (void)
{
   if (INTCONbits.RBIF = =1)
                        //if Port B change interrupt then
                          execute PortBInt_ISR program
   PortBInt_ISR( );
}
#pragma code hi_priori = 0x0008
void hi_priori (void)
   {
        _asm
        GOTO my_isr
        _endasm
   }
#pragma code
void main(void)
   {
        TRISDbits.TRISD6 = 0;          //RC6 = output
        TRISDbits.TRISD7 = 0;          //RC7 = output
        TRISBbits.TRISB6 = 1;   //RB6 = 1 i.e. input
                                  for interrupt
        TRISBbits.TRISB7 = 1;  //RB7 = 1 i.e. input
                                 for interrupt
        INTCONbits.RBIF = 0;   //clear RBIF flag
        INTCONbits.RBIE = 1; //Enable port B-change
                                 interrupt
        INTCONbits.GIE = 1;   //Enable all interrupts
                                 globally
        while (1);          //keep looping till interrupt is
                              recognized.
   }
void PortBInt_ISR(void)
   {
        LED1 =PORTBbits.RB6; //  change the state
                             //  of LEDs
        LED2 = PORTBbits.RB7;
        INTCONbits.RBIF = 0;  //  Clear RBIF
   }
```

9.12 Setting the Interrupt Priority

- The PIC18 microcontroller has only **two interrupt priority levels**.

  They are :

  (i) **Low priority interrupts :** These interrupts are directed to vector location 00018H.

  (ii) **High priority interrupts :** These interrupts are directed to vector location 00008H.
- The addresses 00008H and 00018H are assigned to interrupts to make them compatible with the earlier PIC microcontrollers.
- On reset all the interrupts are assigned high priority, making it a single priority system.
- In order to make the system a two priority level system we need to set the IPEN bit in the RCON register. If IPEN = 1, then the interrupts can be assigned a low or high priority. On reset, IPEN = 0 i.e. all interrupts are assigned high priority.
- By programming the IP bits, we can assign low priority to any interrupt. A low priority interrupt will be directed to address 00018H. If IP bit = 0, interrupt will be assigned low priority otherwise interrupt will have a high priority.
- Table 9.12.1 shows the interrupt Flag Bits for the PIC18 interrupts.

Table 9.12.1 : Interrupt flag bits for PIC18 and their registers

Type of Interrupt	Interrupt	Interrupt enable bit (Register)	Interrupt flag bit (Register)	Interrupt priority (Register)
Timer Interrupts	Timer 0	TMR0IE (INTCON)	TMR0IF (INTCON)	TMR0IP (INTCON2)
	Timer 1	TMR1IE (PIE1)	TMR1IF (PIR1)	TMR1IP (IPR1)
	Timer 2	TMR2IE (PIE1)	TMR2IF (PIR1)	TMR2IP (IPR1)
	Timer 3	TMR3IE (PIE2)	TMR3IF (PIR3)	TMR3IP (IPR2)
External Hardware Interrupts	INT1	INT1IE (PIE1)	INT1IF (PIR1)	INT1IP (INTCON3)
	INT2	INT2IE (PIE1)	INT2IF (PIR1)	INT2IP (INTCON)
Serial Communication Interrupts	TXIF	TXIE (PIE1)	TXIF (PIR1)	TXIP (IPR1)
	RCIF	RCIE (PIE1)	RCIF (PIR1)	RCIP (IPR1)
Port B Change Interrupts	RB_INT	RBIE (INTCON)	RBIF (INTCON)	RBIP (INTCON2)

Note : INT0 is a single priority interrupt. It has high priority level and is directed to vector location 0008H. Its priority cannot be modified, like other interrupts.

9.13 Interrupt Inside an Interrupt

- When the PIC18 microcontroller is executing an ISR in order to provide service to an interrupt and if another interrupt is invoked, then in such a case the newly invoked interrupt is a high priority interrupt and only then it can interrupt the previously serviced low priority interrupt. It is an **interrupt inside interrupt** or **nested interrupt.**
- A low priority interrupt can be interrupted by a high priority interrupt, but not by any other low priority interrupt.
- All the PIC18 interrupts are latched and kept internally. This allows low priority interrupts to be serviced after the high priority interrupts are being serviced.
- If a low priority interrupt is directed to vector location 0018H, the GIE1 bit in the INTCON register is disabled to indicate that a low priority interrupt is being serviced and all other interrupts will be blocked. After servicing the interrupt the GIEL bit is set to allow another low priority interrupts to be accepted.
- If a high priority interrupt is directed to vector location 0008H, the GIEH bit in the INTCON register is disabled to indicate that interrupt is being serviced. After servicing the interrupt the GIE bit is set to allow other interrupts.
- If two interrupts have the same priority then they will be serviced depending on the sequence in which program checks them in the interrupt vector table.

9.14 Triggering the Interrupt by Software

- It is possible to trigger the interrupts by software. By using simple instructions that set the interrupts and cause the microcontroller to jump to the interrupt vector table we can trigger the interrupts.

  Eg. BSF INTCON, RBIF will interrupt the PIC18 microcontroller and force it to jump to the interrupt vector table to provide service to the interrupt.
- We are using an instruction that activates the interrupt. This is called as **software triggering of the interrupt.**

9.15 Interrupt Latency

Definition :

Interrupt Latency is defined as the time from when an interrupt is activated to the time the microcontroller begins executing the code directed at vector address 0008H or 0018H.

- Depending on the type of interrupt and instruction the interrupt latency can be 2 to 4 instruction cycles.

9.16 Fast Context Saving in Task Switching

- In real time operating systems, the system processes one task at a time and completes it before it processes the next task.
- The execution of a task involves the execution of an interrupt service routine. Also while executing the task the access to the CPU resources is very important as the task needs to be completed efficiently and quickly.
- Previously the systems had very few registers. Hence, before executing a new task the programmers had to save the contents of CPU on the stack. This mechanism of saving the contents of CPU before switching to a new task is called as **context saving** or **context switching.**
- The PIC18 microcontroller has many registers. Hence, we need not save the CPU contents on stack.
- The PIC18 uses three registers : WREG, BSR and STATUS registers for executing each task. These registers are saved in shadow registers if a high priority interrupt is recognized. At the end of ISR we need to use "RETFIE 0x01" instruction. This is called **fast context saving in task switching** in PIC18 systems.
- However, fast context saving is not allowed for interrupts with low priority.
- The depth of shadow registers is one. So if two or three high priority interrupts are recognized, only the first ISR will use fast context saving.

□□□

UNIT - V

CHAPTER 10

Interfacing of Switches, Keyboard, LED and LCD

10.1 Interfacing of Switches

- Generally a group of DIP switches comprises four or eight switches. An input port with eight pins e.g. : PORT B, PORT C, PORT D uses eight DIP switches.
- Fig. 10.1.1 shows the interfacing of DIP switches to PIC microcontroller.

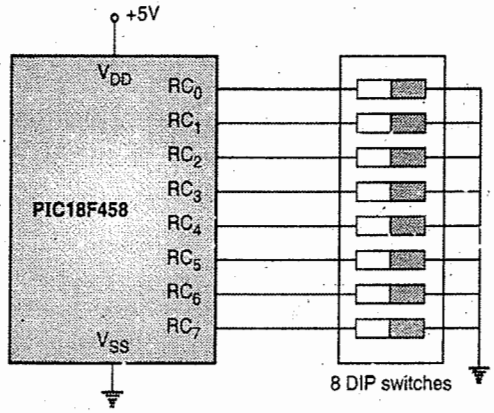


Fig. 10.1.1 : Interfacing 8 DIP switches to port C of PIC18F458

Syllabus Topic : Interfacing of Keyboard

10.2 Interfacing of Keyboard

It is a human oriented input peripheral. It is used to input data or program into the microcomputer. It consists of push button type switches. When a key is pressed, the microcontroller identifies key depression and then performs appropriate operation.

10.2.1 Key Switch Mechanism and Key Debouncing

The aim of this mechanism is to generate and transmit a code each time a key is pressed. The mechanism should send one and only proper code, when the key is pressed. Fig. 10.2.1 shows the general operation of a keyboard.

The input keyboard is composed of a set of labeled push button switches. Each switch makes electrical contact when pressed. The nature of the contact should be reliable, have long life and feel right.

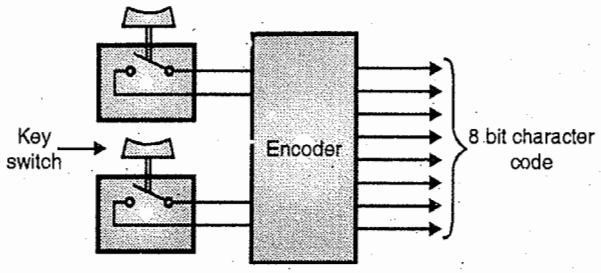


Fig. 10.2.1 : General operation of a keyboard

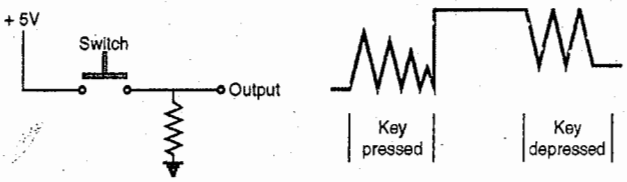


Fig. 10.2.2 : Bouncing of key switch

In case of a push button key, the metal contact bounces few times, hence the voltage across the switch fluctuates and generates spikes in the signal. Therefore, it is necessary to debounce the mechanical switches. This is called **key debouncing**. The key debouncing is done through **hardware and software**. Fig. 10.2.2 shows the bouncing of key switch.

10.2.2 Hardware Key Debouncing

It is implemented by using flip-flop or latch. Fig. 10.2.3 shows a circuit diagram of hardware key debouncing.

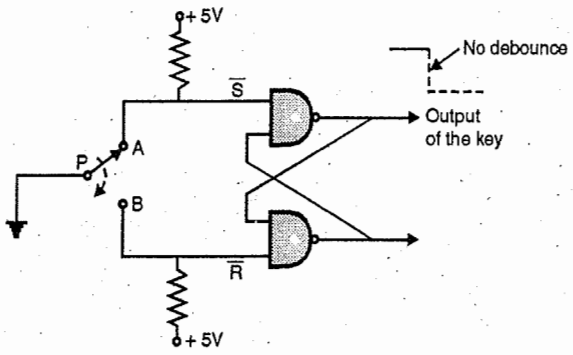


Fig. 10.2.3 : Hardware key debouncing

When the switch is connected to A, the output of the latch goes high. When the key makes contact with B, the output changes from logic 1 to logic 0. The wiper bounces many times on contact B, but the output does not fluctuate between logic 1 and logic 0. When the wiper is not connected either to A or B, the output of the latch remains constant.

10.2.3 Software Key Debouncing

In the software technique the microcontroller waits for 20 ms before it accepts the key as an input. If after 20 ms the key is pressed the key is accepted by microcontroller. The process of software key debouncing is as shown in Fig. 10.2.4.

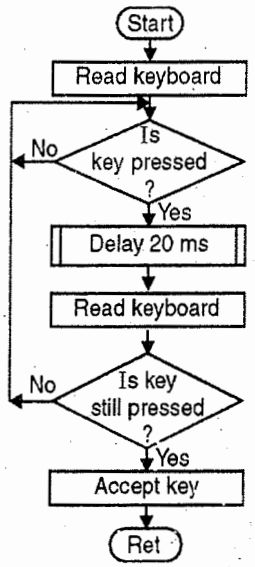


Fig. 10.2.4 : Software key debouncing

10.3 Keyboard Interface Circuit

The keyboard is interfaced with microcontroller through input ports. The keyboard consists of mechanical switches. These switches are arranged in **non-matrix** or **matrix form**.

10.3.1 Non-matrix Type Keyboard

- In non-matrix type keyboard, the key closure is identified by reading the port data, but it requires many port lines. The number of I/O lines is equal to number of keys. Fig. 10.3.1 shows the interfacing of octal non-matrix type keyboard.
- To identify the key value the following three functions should be performed :
  (1) Identifying a key closure.
  (2) Debouncing the key.
  (3) Encoding the key to an appropriate code like hexadecimal.

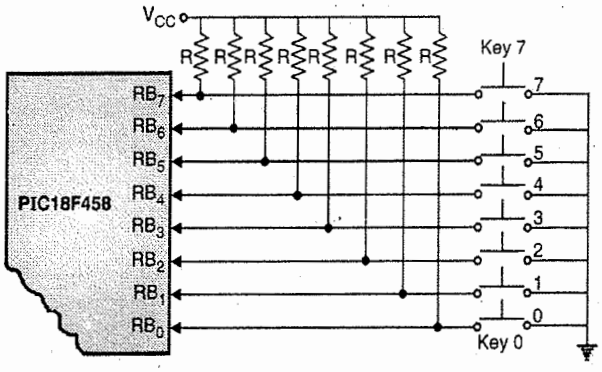


Fig. 10.3.1 : Non matrix type keyboard

- The above three functions can be performed through hardware as well as software. As an example we will see hardware technique for identification of key closure. The interfacing is as shown in Fig. 10.3.2.

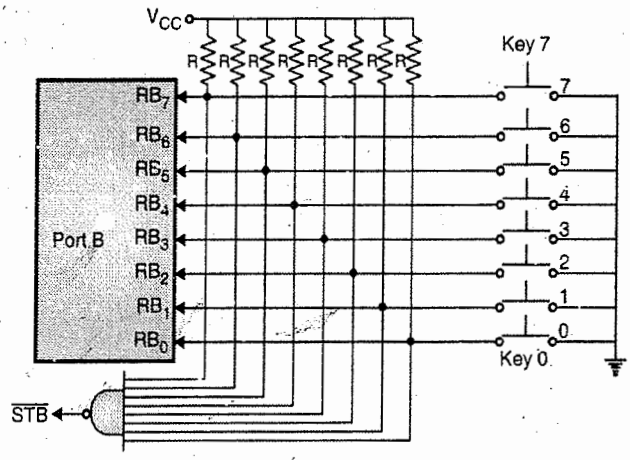


Fig. 10.3.2 : Hardware technique of identification

- When all keys are open, the output of NAND gate ($\overline{STB}$) goes low. When one of the keys is pressed, the output of NAND gate ($\overline{STB}$) goes high. The $\overline{STB}$ is used to identify that the key is pressed. This $\overline{STB}$ signal can be used to interrupt the microcontroller.

Ex. 10.3.1

Interface a simple keyboard to microcontroller PIC18.

Soln. : Fig. P. 10.3.1 shows how a simple keyboard is interfaced to PIC18 microcontroller.

As shown in Fig. P. 10.3.1 eight keys are connected to PORTB pins. Each port pin gives the status of key that is connected to that pin.

If a pin shows logic 1 then the key is open otherwise the key is closed.

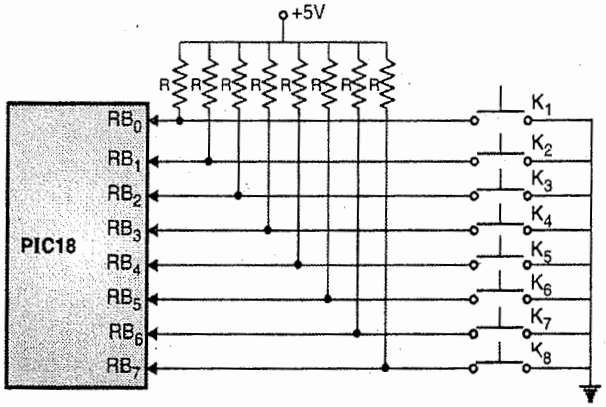


Fig. P. 10.3.1

Following Table P. 10.3.1 gives the keycodes for keys from PORTB.

Table P. 10.3.1

Key	Keycode							
	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
K_1	1	1	1	1	1	1	1	0
K_2	1	1	1	1	1	1	0	1
K_3	1	1	1	1	1	0	1	1
K_4	1	1	1	1	0	1	1	1
K_5	1	1	1	0	1	1	1	1
K_6	1	1	0	1	1	1	1	1
K_7	1	0	1	1	1	1	1	1
K_8	0	1	1	1	1	1	1	1

10.4 Matrix Keyboard Interface

- In a simple keyboard interface one input line is required to interface one key and this increases the number of keys.
- When a large number of keys are to be interfaced, this technique is not useful. Matrix method is used in such cases, so that the number of connections are reduced.
- Fig. 10.4.1 shows 16 keys arranged in 4 rows and 4 columns. No connections are there, when the keys are open.
- If a key is pressed then there is connection between corresponding rows and columns. Such a matrix requires eight lines to complete the connections.

If non-matrix type connection is used then 16 lines will be required. So using method reduces the number of connections.

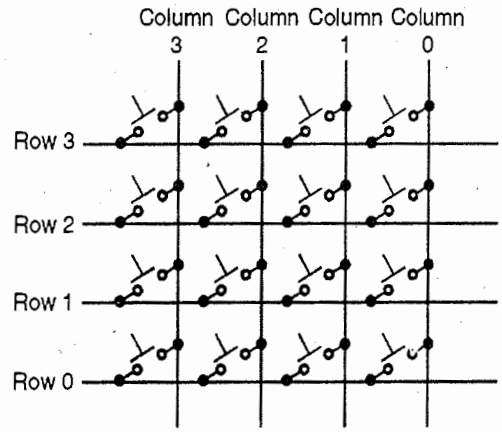


Fig. 10.4.1 : Matrix keyboard

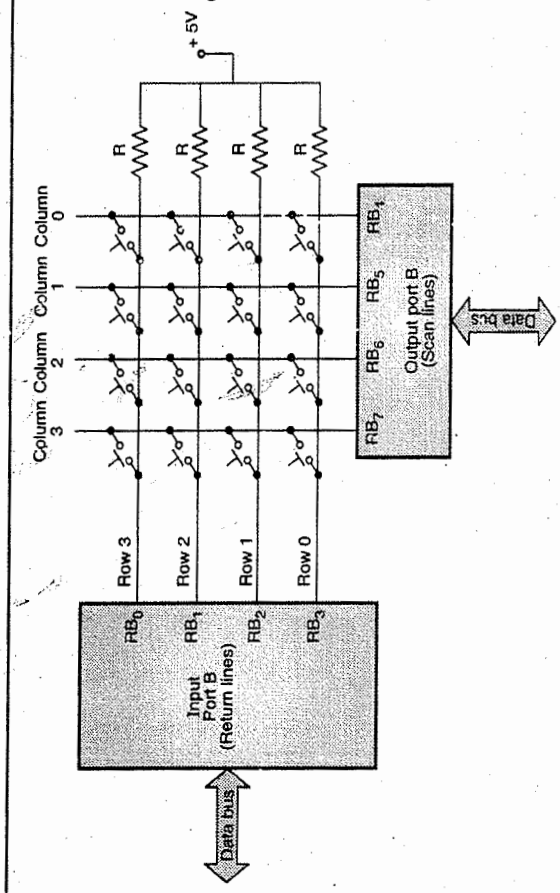


Fig. 10.4.2 : Matrix keyboard connections

- Fig. 10.4.2 shows the interfacing of a matrix keyboard, it requires two ports : an input port and an output port. The columns are referred to as **scan lines** and rows are referred to as **return lines**.

- When a key is pressed, the corresponding row and column are connected i.e. they are shorted. If the output line of a row is high, then it makes the line of a column high and vice versa. The key is recognized by data which is sent on the output port and the input code that is received from the input port. The steps required to identify the pressed key are,
  - (i) To identify if any key is pressed or not.
    - (a) All the column lines are made zero by sending low on all the output lines. i.e. all the keys in the keyboard matrix are activated.
    - (b) Read the status of rows i.e. return lines. If the status of all lines is logic high, the key is not pressed. Otherwise if the status of all lines is logic low, the key is pressed.
  - (ii) Debouncing the key. (Using software debouncing as explained earlier)
  - (iii) Identifying the pressed key.
    - (a) Activate the keys from one column by making one column line zero.
    - (b) Read the status of return lines. The zero on any return line indicates that key is pressed.
    - (c) Activate the keys from next column and repeat steps (b) and (c) for all the columns.

Ex. 10.4.1

SPPU : Dec. 2015, May 2016, 8 Marks, Lab assignment

Interface a 4 × 4 matrix keyboard to PIC18F458. Display keypressed on hyper terminal.

OR

Draw and explain interfacing of 4*4 matrix key pad with PIC18FXXX microcontroller using interrupt Write code in 'C'.

OR

Draw an interfacing diagram for 4*4 matrix key board and display the Key pressed on LED. Write a code.

Soln. :

Fig. P. 10.4.1 shows the interfacing of a 4 × 4 matrix keyboard to PIC18F458.

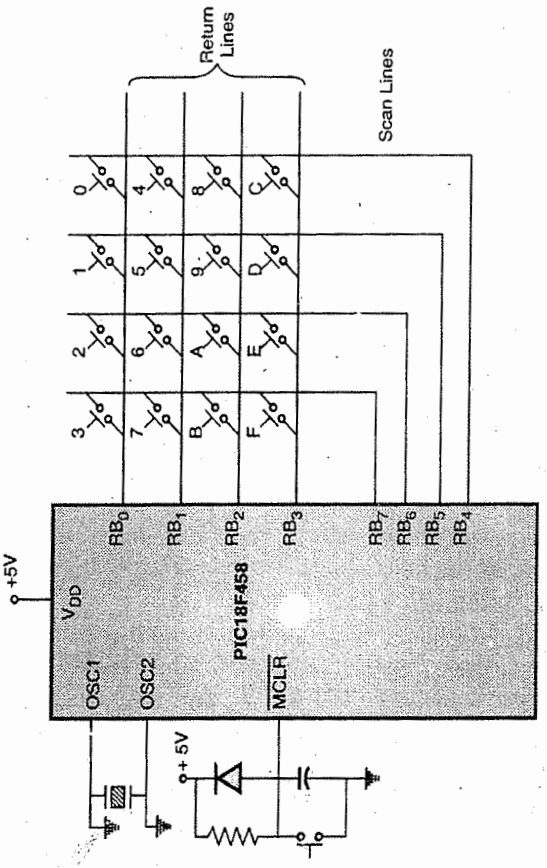


Fig. P. 10.4.1 : Interfacing a 4 × 4 matrix keyboard to PIC18F458

The 4 × 4 matrix keyboard is connected to the port B of PIC microcontroller. $RB_0 - RB_3$ are rows or return lines while pins $RB_7 - RB_4$ are columns or return lines. For detecting the key presses two methods can be used. They are

(i) Interrupt method

(ii) Scanning method

In this program we use the interrupt method for detecting the key pressed.

Fig. P. 10.4.1(a) shows the flowchart for determining the key pressed.

Flowchart :

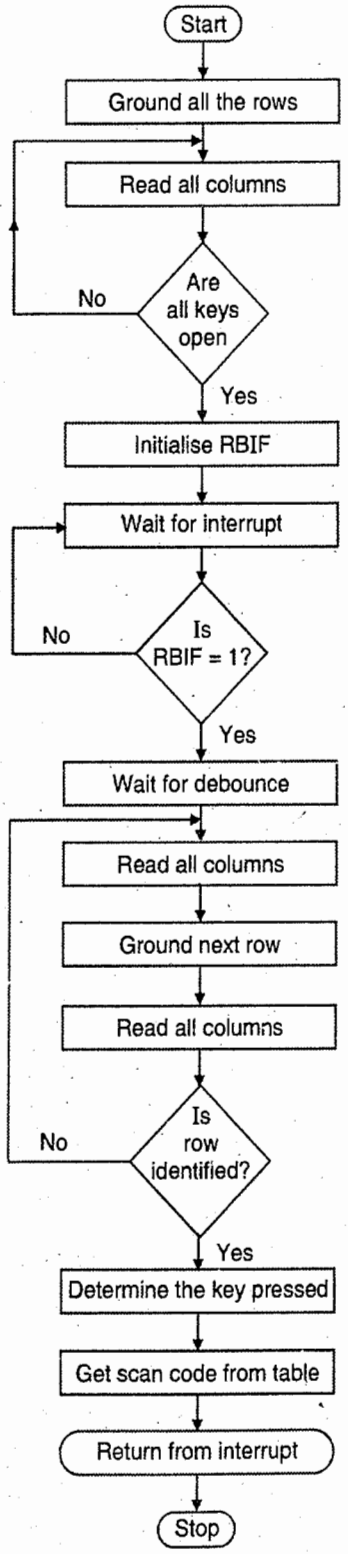


Fig. P. 10.4.1(a)

```
#include<P18F458.h>
        void serial (unsigned char x) ;
        void RBIF_ISR (void) ;
        void delay (unsigned int ms) ;
        unsigned char keypad [4] [4] = {'0', '1', '2', '3', '4',
'5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'} ;
# pragma code hi_int = 0x0008    // high priority interrupt
        void hi_int (void)
        {
            _asm
            Goto my_isr
            _endasm
        }
#pragma code
# pragma interrupt my_isr
void my_isr (void)
        {
            if (INTCONbits.RBIF = = 1)  // Is RBIF = 1 ?
            RBIF_ISR ( ) ;
        }
# pragma code
void main ( )
        {
            TRISD = 0 ;        // Make Port D an output port
            INTCON2bits.RBPU = 0 ;
                               //Enable PORTB pull up resistors
            TRISB = 0xF0;
            PORTB = 0xF0;
            while (PORTB! = 0xF0) ;
                               // Wait until key is not pressed.
            TXSTA = 0x20 ;
            SPBRG = 15 ;           // Baud rate 9600
            TXSTA bits. TXEN = 1 ;   //Enable transmit
            RCSTA bits. SPEN = 1 ;  //Enable serial port
            INTCONbits. RBIE = 1;
                               //Enable PORTB interrupt on change
            INTCONbits. GIE = 1;
                               //enable interrupts globally
            while (1) ;            // Wait till key is pressed.
        }
void RBIF_ISR (void)           //Identify the key pressed
        {
            unsigned char x, COL = 0, ROW = 4 ;
            delay (15) ;
            x = PORT B ;           //get column
```

```
        x ^ = 0xF0 ;          //invert the high nibble
    if (!x) return ;
    while (x << 1) COL++ ;    //determine the column
    PORTB = 0xFE;             //ground row 0
    if (PORTB! = 0xFE)        // is there a nibble
                              change yes then row = 0
    ROW = 0 ;
    else
    {
        PORTB = 0xFD ;   //ground row 1
        if (PORTB! = 0xFD)// is there a nibble change
        ROW = 1 ;         //yes then row = 1
        else
    {
        PORTB = 0xFB) ;  //Ground row 2
        if (PORTB! = 0xFB)//is there a nibble change
        ROW = 2;          //yes then row = 2
        else
        PORTB = 0xF7 ;   //Ground row 3
        if (PORTB! = 0xF7) // is there nibble change ?
        ROW = 3 ;         //yes then row = 3
        }
    }
}
if (ROW < 4)              // is valid row found ?
serial (keypad [ROW] [COL]) ;  // then send character
while (PORTB ! = 0xF0)
    PORTB = 0xF0 ;            //Wait for release
    INTCONbits.RBIF = 0 ;     //Reset flag
    }
    void serial (unsigned char i)    // send character
    {
        while (P1R1 bits. TXIF! = 1) ;  //wait till ready
    TXREG = i ;   // send key pressed to
                  serial port / hyper terminal
    }
void delay (unsigned int ms)
    {
        unsigned int x, y ;
        for (x = 0 ; x < ms ; x++)
        for (y = 0 ; y < 165 ; y ++) ;
    }
```

Syllabus Topic : Interfacing of LED

10.5 Interfacing of LED

It is a human oriented output peripheral. It is used to display result or operand. One may use CRT, LED or LCD displays.

A CRT is used to display large amount of data. LED and LCD displays are used to display small amount of data. The commonly used LED displays are numeric displays.

10.5.1 LED Displays

To drive a LED, there are two methods.

Method 1)

Connect the cathode of LED to ground. Connect the anode of LED to port pin of PIC18Fxxx, through a resistor as shown in Fig. 10.5.1.

This method requires PIC18FXX to source a huge amount of current required by the LED i.e. around 25 mA.

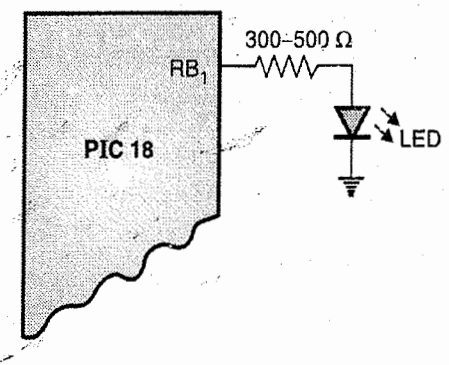


Fig. 10.5.1 : LED driven by PIC18 output (Method 1) port pin in current source mode

Method 2)

Connect the anode of LED to V_{CC} through resistor. Connect the cathode of LED to the port pin of PIC18F458 as shown in the Fig. 10.5.2.

This method requires PIC microcontroller to sink a huge current required by LED i.e. 25 mA. PIC microcontroller can sink huge currents and hence it makes LED glow brighter.

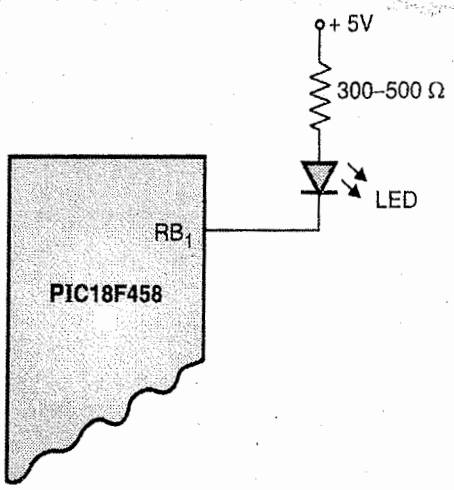


Fig. 10.5.2 : LED driven by PIC18F458 port pin in current sink mode (Method 2)

The LED displays are available in two common formats : seven segment display and 5 by 7 dot matrix displays.

Ex. 10.5.1

Write an embedded C program to blink LED connected to port of PIC.

Soln. :

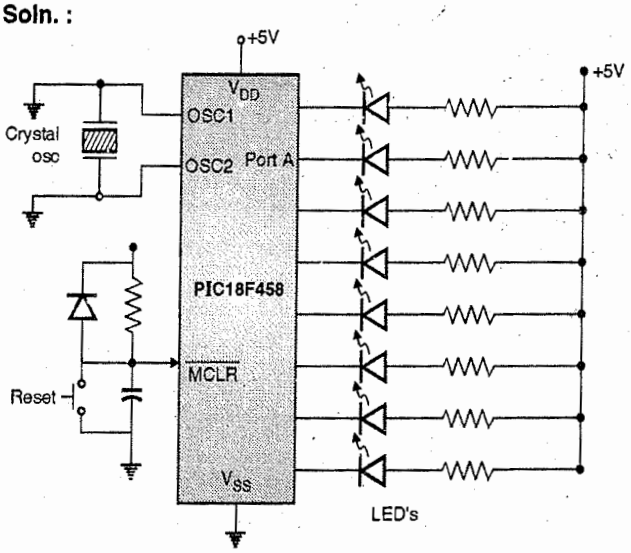


Fig. P. 10.5.1

```
# include < P18F458.h>
void Delay (unsigned int) ;
void main (void)
{
    TRISA = 0 ;      // Port A is output port forever
    while (1)
    {
        PORTA = 0x00H ;
        Delay (250) ;
        PORTA = 0xFFH ;
        Delay (250) ;
    }
}
void Delay (unsigned int xtime)
{
    unsigned int i ;
    unsigned char j ;
for (i = 0 ; i < xtime ; i++)
for (j = 0 ; j < 165 ; j++) ;
}
```

Ex. 10.5.2 SPPU - Dec. 2014, 8 Marks

An LED is connected to each pin of port D. Write a C program that will turn on each LED from pin D0 to D7. Call a delay module before turning on the next LED.

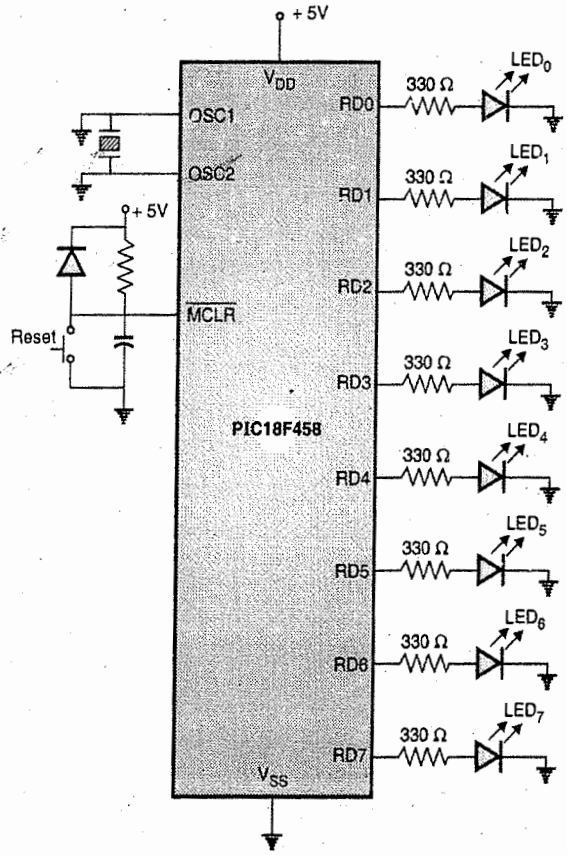


Fig. P. 10.5.2

Soln. :

```
# include < P18F458.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 ;        //Port D is output port
        while (1)
        {
        PD0 = 1 ;                // turn on RD0
        Delay (250) ;
        PD1 = 1 ;                //turn on RD1
        Delay (250) ;
        PD2 = 1 ;                // turn on RD2
        Delay (250);
        PD3 = 1 ;                //turn on RD3
        Delay (250) ;
        PD4 = 1 ;                // turn on RD4
        Delay (250) ;
        PD5 = 1 ;                // turn on RD5
        Delay (250) ;
        PD6 = 1 ;                //turn on RD6
        Delay (250) ;
        PD7 = 1 ;                // turn on RD7
        Delay (250) ;
        }
    }
    void Delay (unsigned int xtime)
    {
        unsigned int i ;
        unsigned char j ;
        for (i = 0 ; i < xtime ; i++)
        for (j = 0 ; j < 165 ; j++) ;
    }
```

Ex. 10.5.3

A switch is connected to pin RC0 and LED to pin RB6. Write a program to get the status of switch and send it to the LED.

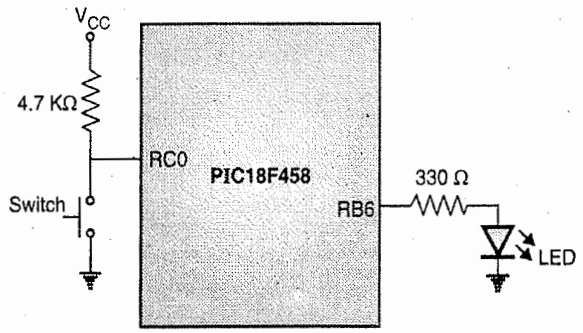


Fig. P. 10.5.3

Soln. : C program :

```
# include <P18F458.h>
# define switch PORTCbits.RC0
# define LED PORTBbits.RB6
    void main (void)
    {
        TRISCbits.TRISC0 = 1  //Make RC0 an input pin
        TRISBbits.TRISB6 = 0 // Make RB6 an output pin
        while (1)
        {
            if (switch = = 1)
            LED = 1 ;            // Turn on LED
            else
            LED = 0 ;            //Turn off
        }
    }
```

Ex. 10.5.4

LEDs are connected to the bits in PORT C and PORT D. Write a C18 program that shows the count from 0 to FFH on the LEDs.

Soln. :

```
#include <P18F458.h>
#define LED PORTD
    void main (void)
    {
        TRISC = 0;        //Port C as an output port
        TRISD = 0 ;       // Port D as an output port
        PORTC = 00 ;      // Clear Port C
        LED = 0           // Clear Port D
        for (;;)
        {
            PORTC++;  // increment port C
            LED ++ ;  // increment port D
        }
    }
```

10.5.2 Seven Segment Display (SSD)

- As shown in the Fig. 10.5.3 the seven segment display uses seven LEDs to make any digit. If all the LEDs are on, it shows the digit 8. There are two types SSDs available.

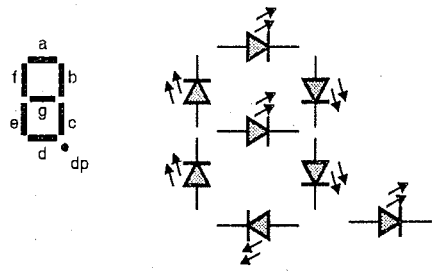


Fig. 10.5.3 : Structure of seven segment display (SSD)

(i) Common cathode i.e. the cathode of all the LEDs are given as a common pin. In this case the anode is connected to the port pins. As already discussed in the section for LEDs, this method requires port pins to source large current.

(ii) Common anode i.e. the anode of all the LEDs are given as a common pin. In this case the cathode is connected to the port pins.

Ex. 10.5.5

Write embedded C program to implement HEX counter on port and display the count.

Soln. :

```
title "HEX Count on LED Display"
#include < P18F458.h>
#define Led_Disp PORTA

void main(void)
{
    TRISA = 0;              // declares PORTA as output
    Led_Disp =0x00;         // initiates count to 0
    int j;
    for (j=0x00; j<0x0F; j++)
    {
            Led_Disp = j;       //  display  incremented
                                    Hex count on LED.
            delay(200);
    }
    while(1);
}
void delay(int delayval)
{
unsigned int m,n;
for (m=0x00; m<0x0F; m++)
                //looping around to introduce software delay
    for (n=0x00; n<0x0F; n++);
}
```

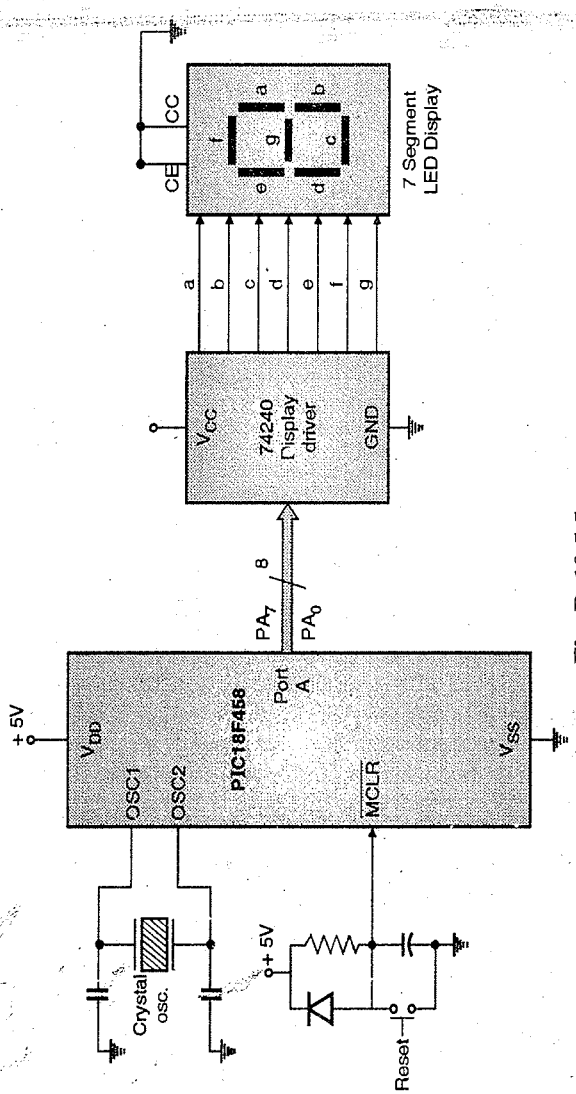


Fig. P. 10.5.5

10.5.3 Multiplexed Display

- Fig. 10.5.4 shows the interfacing of 4 seven segment display. The 4 digits are connected using the time multiplexing method.
- As shown in Fig. 10.5.4 the common cathode of a seven segment display is connected to the collector of an NPN transistor.
- The circuit uses two ports Port B and Port D. Port D is used to drive the LED segments and Port B is used to turn on the respective cathodes, so that the digits can be displayed.
- The NPN transistor will operate in saturation region, if the voltage at pin B is high. The common cathode will be made low, so that display can be lighted.

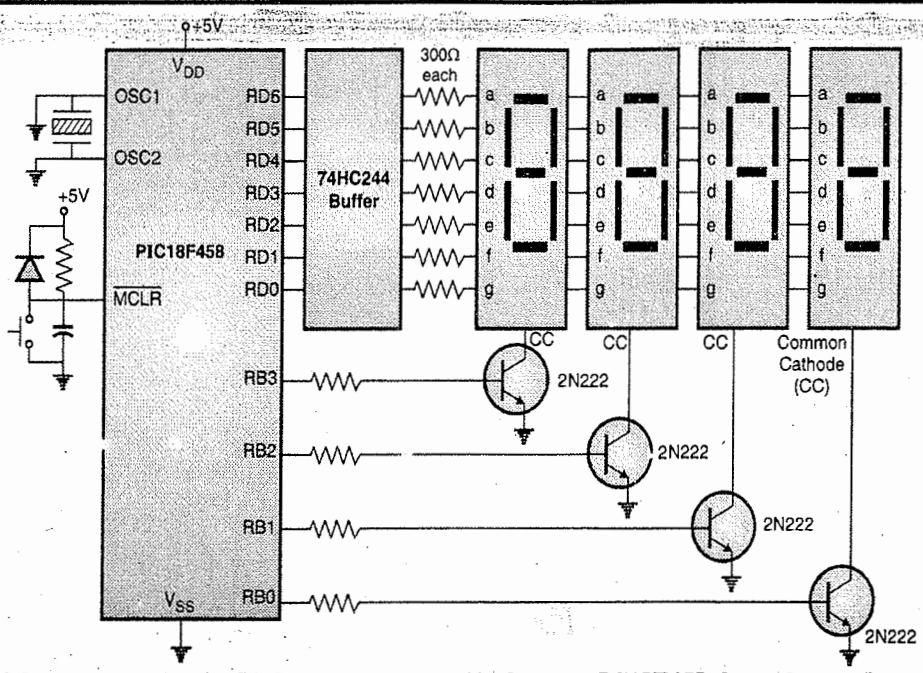


Fig. 10.5.4 : Interfacing multiple seven segment displays to PIC18F458 through Port B and Port D

- Looking at the Fig. 10.5.4 one question may arise in your mind that will all the 4 digits display the same number. The answer is that if all the digits are turned on at the same time then they will show the same number. But, in case of multiplexed display the segment information is sent for all digits on common lines, but only one display digit is turned on at a time.
- The digit driver i.e. NPN transistors are connected in series with the common cathode of each digit.
- By turning the four NPN transistors on and off turn by turn many times in a second, multiple digits can be displayed.
- Multiplexing gives a large saving in power and hardware components.
- Fig. 10.5.5 shows interfacing a common cathode seven segment display through buffer chip 74HC244 to PIC18F458.

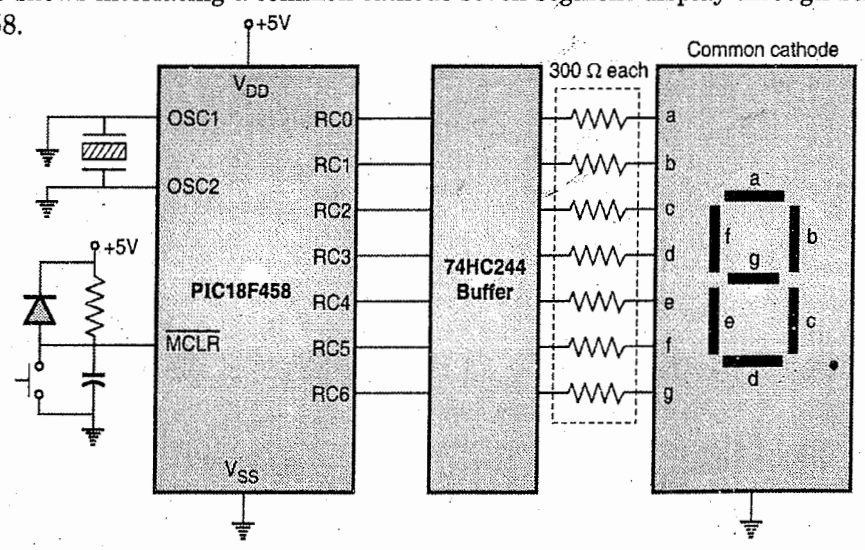


Fig. 10.5.5 : Interfacing common cathode seven segment display through buffer chip 74HC244 to PIC18F458

- The seven segments are labelled a to g. Table 10.5.1 shows the binary and hexadecimal data given to the seven segment display to display digits 0 through 9.
- A pin should be given logic '1' to switch on the corresponding LED.

Table 10.5.1 : Code for 0 to 9 given to common cathode SSD

Decimal digit	Segments							Hexadecimal code
	a	b	c	d	e	f	g	
0	1	1	1	1	1	1	0	0x7E
1	0	1	1	0	0	0	0	0x30
2	1	1	0	1	1	0	1	0x6D
3	1	1	1	1	0	0	1	0x79
4	0	1	1	0	0	1	1	0x33
5	1	0	1	1	0	1	1	0x5B
6	1	0	1	1	1	1	1	0x5F
7	1	1	1	0	0	0	0	0x70
8	1	1	1	1	1	1	1	0x7F
9	1	1	1	1	0	1	1	0x7B

Ex. 10.5.6

Write an instruction sequence to display 8 on the 4 seven segment displays shown in Fig. P. 10.5.6.

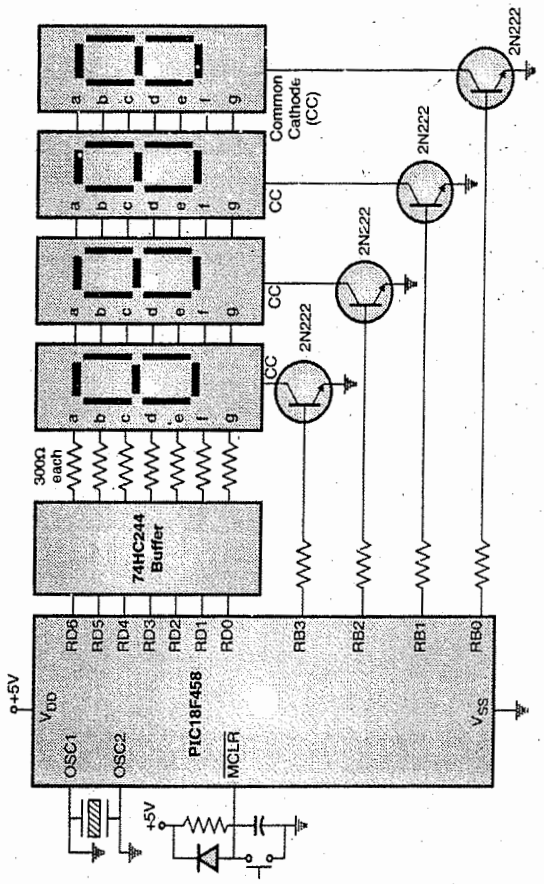


Fig. P. 10.5.6 : Interfacing multiple seven segment displays to PIC18F458 through Port B and Port D

Soln. :

Program :

The code for displaying 8 on the 7 segment display is 0x7FH.

C program :

```
# include <P18F458.h>
    void main (void)
    {
        TRISB = 0 ;        //Make port B an output port
        TRISD = 0 ;        // Make port D an output port
        PORTD = 0x7F ;     //Output the segment
                             pattern of 8
        PORTB = 0x10       //activate 4, seven
                             segment displays
    }
```

Ex. 10.5.7

Write a program to display 54321 on a five seven segment display assuming that PIC18F458 is used.

Soln. :

Fig. P. 10.5.7 shows the interfacing diagram.

The digits 54321 will be displayed on the five seven segment digits. The codes for displaying the digits are given in table below.

Seven segment display digit No.	Displayed digit	Port D	Port B
4	5	0x5B	0x10
3	4	0x33	0x08
2	3	0x79	0x04
1	2	0x6D	0x02
0	1	0x30	0x01

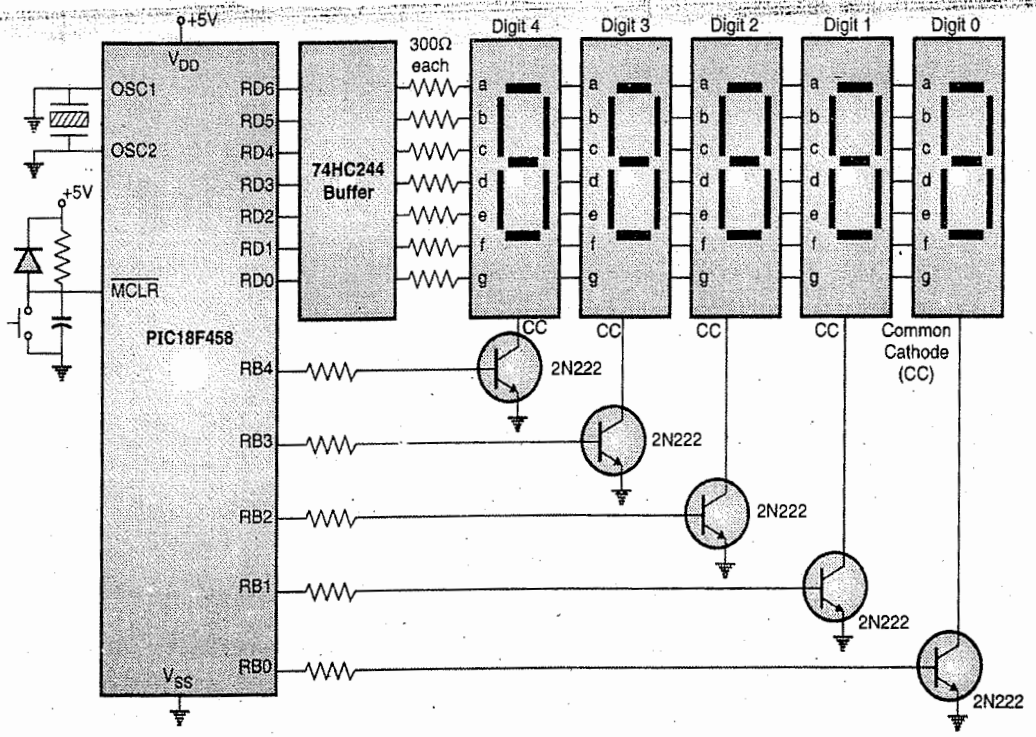


Fig. P. 10.5.7 : Interfacing five seven segment displays with PIC18F458

Program :

```
#include <P18F458.h>
    void delay (unsigned int) ;
    char disp [5] [2] = {{0x5B, 0x10}, {0x33, 0x08},
{0x79, 0×04}, {0x6D,0x02}, {0x30, 0x01}}
    void main (void)
    {
        int i ;
        TRISB = 0 ;      //Make port B an output port
        TRISD = 0 ;      //Make Port D an output port
        while (1)
        {
            for (i = 0 ; i < 5 ; i++)
            {
             PORTD = disp[i] [0] ;      // output display
                                          pattern
            PORTB = disp[i] [1] ;     //turn on one display
            Delay (250) ;
        }
    }
    void delay (unsigned int itime)
    {
        unsigned int i ;
        unsigned int j ;
        for (i = 0 ; i < itime ; i++)
        for (j = 0 ; j < 165 ; j++);
    }
```

Syllabus Topic : Interfacing Liquid Crystal Display (LCD) to PIC18F458

10.6 Interfacing Liquid Crystal Display (LCD) to PIC18F458

LCD displays are widely used because of its low current consumption as compared to SSD. Also that LCD can be used to display any character as it uses a 5 × 7 dot matrix to display.

For e.g. to display '1' of LCD as shown in Fig. 10.6.1.

- An LCD allows the user to output a specific message making the application more user friendly and attractive.
- LCDs are invaluable for displaying status messages and information while a program is being debug.

- The LCDs generally use a common controller chip, Hitachi 44780 and common connector interface. Due to these factors, the alphanumeric LCDs range in size from 8 characters to 80 characters. All the characters are interchangeable without any hardware or software changes. They are arranged in 40 by 2 or 20 by 4 or 10 by 2 or 20 by 1 or 20 by 2. The first figure represents the number of characters in each line and second figure represents the number of lines the display has.

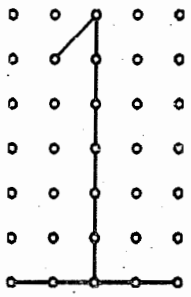


Fig. 10.6.1 : Displaying 1 on LCD display of a 5 × 7 dot matrix

A typical 16 by 2 (i.e. 16 characters and 2 such lines) LCD looks as shown in Fig. 10.6.2.

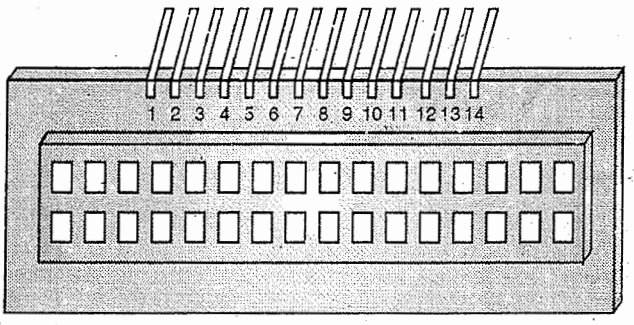


Fig. 10.6.2 : Structures of 16 × 2 LCD

10.6.1 LCD Pin Description

Some LCD have their pins on left or on bottom. The functions of the pins of LCD are listed in the Table 10.6.1.

Table 10.6.1 : Pin description of LSD

Pins	Symbol	Functions
1	V_{ss}	Ground
2	V_{dd} (or V_{cc})	+ 5 V supply
3	V_{EE}	Power supply to control contrast
4	RS	Should be '0' for instruction and '1' for data.
5	$R/\overline{W}$	Should be '0' to write and '1' to read
6	E	Enable display logic
7	D0	Data bus bit 0
8	D1	Data bus bit 1
9	D2	Data bus bit 2
10	D3	Data bus bit 3
11	D4	Data bus bit 4
12	D5	Data bus bit 5
13	D6	Data bus bit 6
14	D7	Data bus bit 7 (Also used as busy pin)

10.6.1.1 RS : Registers Select

- There are two registers of LCD viz. Instruction command code register and data register. The RS pin is used to select one of these registers.
- If RS = 0 the instruction command code register is selected and if RS = 1 the data register is selected, allowing user to send data to be displayed on the LCD.

10.6.1.2 $R/\overline{W}$: Read/Write

- $R/\overline{W}$ pin is used to read or write to LCD.
- If $R/\overline{W}$ = 0 the user can write information to the LCD and if $R/\overline{W}$ = 1 the user can read information.

Enable : E

- The enable pin E, is used to latch the data into the data or command register. When data is supplied, a high-to-low (negative edge) is required for LCD to latch the data.

D0 – D7

- The data pins D0 – D7 are used to send information to the LCD or read the contents of LCD internal registers.
- To display letters and numbers we send ASCII codes for letters A – Z, a – z and numbers 0 – 9 while making RS = 1.

- The ASCII code that is to be displayed is of 8 bits. It is send to the LCD in either nibbles or bytes i.e. 4 or 8 bits at a time.
- The two primary modes of operation to send parallel data are 4 or 8 bits.
- If four bit mode is used two nibbles of data are sent to do an 8 bit transfer. The "E" clock is used to initiate the data transfer. Atleast 6 I/O pins must be available for 4 bit mode.
- In 8 bit mode atleast 10 I/O pins must be used. This mode is used when application needs speed.

10.6.2 Cursor Addresses for LCDs

Table 10.6.2 gives the cursor addresses for common types of LCDs.

Table 10.6.2 : Cursor addresses for some LCDs

16 × 2 LCD	80	81	82	83	84	85	86	through	8F
	C0	C1	C2	C3	C4	C5	C6	through	CF
20 × 1 LCD	80	81	82	83	through	93			
20 × 2 LCD	80	81	82	83	through	93			
	C0	C1	C2	C3	through	D3			
20 × 4 LCD	80	81	82	83	through	93			
	C0	C1	C2	C3	through	D3			
	94	95	96	97	through	A7			
	D4	D5	D6	D7	through	E7			
40 × 2 LCD	80	81	82	83	through	A7			
	C0	C1	C2	C3	through	E7			
All data is in hex.									

10.6.3 LCD Command Codes

- The list of commands that can be given to the LCD are as listed in the Table 10.6.3.

Table 10.6.3 : LCD commands

Hex command	Function
0x01	Clear display
0x02	Return cursor to home
0x04	Decrement cursor (i.e. shift cursor left)
0x06	Increment cursor (i.e. shift cursor right)
0x05	Shift display right
0x07	Shift display left
0x08	Display off, cursor off
0x0A	Display off, cursor on
0x0C	Display on, cursor off
0x0E	Display on, cursor on
0x0F	Display on, cursor on and blinking
0x10	Move cursor one position left
0x14	Move cursor one position right
0x18	Shift entire display left
0x1C	Shift entire display right
0x80	Move cursor to beginning of 1st line
0xC0	Move cursor to beginning of 2nd line
0x38	Initialize 2 line display of 5 × 7 matrix

The command codes can be used to display or force the cursor to home position or blink cursor.

Table 10.6.4 shows the command codes.

Table 10.6.4 : Command codes

Commands	RS	R/$\overline{W}$	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	Description
Clear display	0	0	0	0	0	0	0	0	0	1	It clears the entire display and sets display data RAM to address 0.
Return Home	0	0	0	0	0	0	0	0	1	–	It sets the display data RAM address to 0 It returns the cursor to home position. The display data RAM contents remain unchanged.
Entry Mode Set	0	0	0	0	0	0	0	1	1/D	S	It sets the direction for moving the cursor and specifies the shift of display. These operations are done during data read and data write.
	1/D = 1 increment 1/D = 0 decrement S = 1 accompanies display shift										
Display on/off control	0	0	0	0	0	0	1	D	C	B	It sets the entire display on/off, cursor on/off (0) and blink of cursor position character B
Cursor or Display shift	0		0	0	0	0	1	S/C	R/L	–	–
	S/C = 1 display shift S/C = 0 cursor move R/L = 1 shift to the right R/L = 0 shift to the left										It moves cursor and display shifts without changing display data RAM contents.
Function Set	0	0	0	0	1	DL	N	F	–	–	It sets interface data length (DL), number of data lines (L) and character font (F)
	DL = 1, 8 bits DL = 0, 4 bits N = 1 2 lines N = 0 1 line			F = 0 : 5 × 7 dots F = 1 : 5 × 10 dots							
Set CG RAM address (character generator RAM)	0	0	0	1	ACG						Sets the character generator RAM address. The character generator RAM data is sent and received once this setting is done
	(ACG : CG RAM address)										
Set DD RAM address (display data RAM)	0	0	1	ADD							Sets the display data RAM address. The display data RAM data is sent and received after this setting is done
	(ADD : DD RAM address)										
Read Busy Flag and Address	0	1	BF	AC							Reads busy flag indicating if internal operation is being done and reads address counter contents
	AC : address counter for CG and DD RAM address BF = 0 can accept command or instruction BF = 1 busy in internal operation										
Write data to CG or DD RAM	1	0	write data								Writes data to CG or DD RAM
Read data from CG or DD RAM	1	1	read data								Reads data from CG or DD RAM

- Inorder to display a message on the LCD module we need to initialize the LCD. The LCD is initialized by writing command codes in the command register.
- Initialization comprises of command codes for clearing the display, shifting cursor automatically after writing a character, returning cursor home etc.
- After the initialization we can write data to the DD RAM or the CG RAM by issuing correct command and asserting the R/$\overline{W}$ signal low and RS signal high. The data is sent on the Port and a high to low pulse is applied on the E pin. The DD RAM stores the characters in their ASCII code. The CG RAM stores the character in its internally generated character code.
- Before sending the command or data it is essential to check busy flag i.e. whether the LCD is ready or not.

10.6.4 Initialization of LCD

The following algorithm is required to initialize and write data to LCD :

- Wait 1 second after power up for display to stabilize.
- Initialize the LCD by giving the instruction 0x38 to the command subroutine.
- Wait for 5 msec.
- Issue the command 0x0F to command subroutine for display on, cursor on and cursor blinking.
- Wait for 5 msec.
- Issue the command 0x01 for clearing display to command subroutine.
- Wait for 5 msec.
- Issue the command 0x06 for making LCD in increment mode i.e. cursor should increment after every character is written to command subroutine.
- Wait for 5 msec.
- Issue the command 0x80 (to command subroutine), to position the cursor at 1st line 1st character.
- Issue the data character one by one giving their ASCII values using data subroutine.

Command subroutine

- Give the instruction to the port connected to data bus of the LCD.
- Make RS = '0', to indicate instruction.
- Make R/$\overline{W}$ = '0', to indicate write.
- Make E = '1' } To give a high-to-low
- Wait for 120 μsec. } pulse on E pin so as
- Make E = '0' } to latch the command
- Return.

Data subroutine

- Check if LCD is ready by calling ready subroutine.
- Give the data to the port connected to the data bus of the LCD.
- Make RS = '1', to indicate data
- Make R/$\overline{W}$ = '0', to indicate write
- Make E = '1' } To give a high-to-low
- Wait for 120 μsec. } pulse on E pin so as
- Make E = '0' } to latch the data
- Return

Ready subroutine for sending data or command to LCD using busy flag.

- Make the busy pin (i.e. data bus bit 7) = '1', to program the corresponding port pin of PIC18 as input port.
- Make RS = '0' to indicate instruction.
- Make R/$\overline{W}$ = '1', to indicate read.
- Make E = 0
- Make E = 1
- Check if busy pin = '0'. If it is '1', indicates LCD is busy, hence again make E = '0', then E = '1' and check busy pin. Repeat this until busy pin = '0'.
- Return.

10.6.5 Interfacing LCD Module with PIC18F458

- Fig. 10.6.3 shows the interfacing of a LCD module with PIC18F458. As shown in Fig. 10.6.3 the data lines are connected to Port D of PIC18F458. The control lines RS, R/$\overline{W}$ are driven by Port B pins RB0, RB1 and RB2. The

voltage at V_{EE} pin is adjusted by potentiometer to adjust the contrast of the LCD.

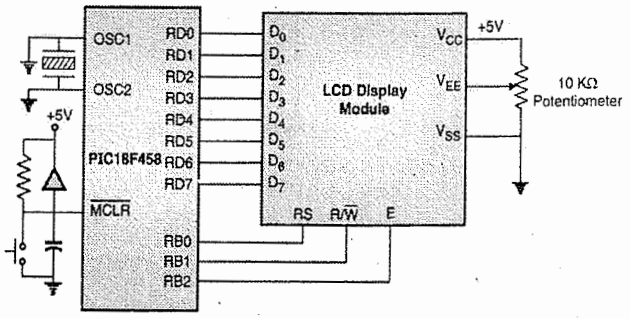


Fig. 10.6.3 : Interfacing LCD module with PIC18F458 (8 bit bus)

Ex. 10.6.1

Interface 2 line, 16 character LCD display to PIC18F458 using only one port. Write C and assembly language program to display message 'HELLO' on line 2 of LCD.

Soln. :

Fig. P. 10.6.1 shows the interfacing of a 16 character × 2 line LCD module with the PIC microcontroller. The data lines are connected to Port D of microcontroller. The control lines RS, R/$\overline{W}$ and E are driven by Port B lines RB0, RB1 and RB2. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

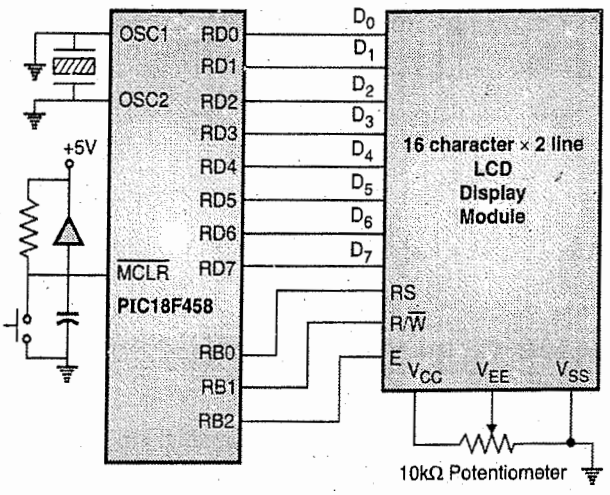


Fig. P. 10.6.1 : Interfacing 16 × 2 LCD to PIC18

Let us write C program to display message "HELLO"

C Program :

```
# include <P18F458.h>
# define lcddatal PORTD              // PORTD LCD pins
# define RS PORTBbits.RB0            //RS = RB0

#define RW PORTBbits.RB1             //R/W = RB1
#define E PORTBbits.RB2              //EN = RB2
    void main (void)
    {
        TRISB = 0;        // Make port B an output port
        TRISD = 0 ;       // Make port D an output port
        E = 0 ;
        Delay (250) ;
        Lcdcmd (0x38) ;          // Initialize LCD 2 lines,
                                    5×7 matrix
        Delay (250) ;
        Lcdcmd (0x0E) ;          // Display on, cursor on
        Delay (250);
        Lcdcmd (0x01) ;          // Clear LCD
        Delay (250) ;
        Lcdcmd (0x06) ;          // Shift cursor right
        Delay (250) ;
        Lcdcmd (0×C0) ;          //Line 2, position 0
        Delay (250) ;
        Lcddata ('H') ;          //display letter 'H'
        Delay (250) ;
        Lcddata ('E') ;          //display letter 'E'
        Delay (250) ;
        Lcddata ('L');           //display letter 'L'
        Delay (250);
        Lcddata ('L');           // Display letter L
        Delay (250) ;
        Lcddata ('O')            //Display letter O
        Delay (250) ;
    }
    void Lcdcmd (unsigned char value) //command routine
    {
        Lcddatal = value;
        RS = 0;
        RW = 0 ;
        E = 1 ;
        Delay (250) ;
        E = 0 ;
    }
    void Lcddata (unsigned char value) //display routine
    {
        Lcddatal = value ;
        RS = 1 ;
```

```
        RW = 0 ;
        E = 1;
        Delay (250) ;
        E = 0 ;
    }
    void Delay (unsigned int itime)
    {
        unsigned int i, j ;
        for (i = 0 ; i < itime ; i++)
        for (j = 0; j < 165 ; j++) ;
    }
```

Ex. 10.6.2

Write a program to display message "MICRO" using busy flag check method on line 1.

Soln. : Fig. P. 10.6.1 shows the interfacing diagram.

C Program :

```
# include <P18F458.h>
# define Ldata PORTD
# define RS PORTBbits.RB0
# define RW PORTBbits.RB1
# define EN PORTBbits.RB2
# define BUSY PORTDbits.RD7
    void main (void)
    {
        TRISB = 0 ;         // Make Port B an output port
        TRISD = 0 ;         // Make port D an output port
        E = 0 ;             // E = 0
        Delay (250) ;
        Lcdcommand (0x38) ;   //Initialize LCD 2 lines,
                                5 × 7 matrix
        Delay (250) ;
        Lcdcommand (0x0E) ;   //display on, cursor on
        Lcdready ( ) ;        //check the busy flag
        Lcdcommand (0x01) ;   //clear display
        Lcdready ( ) ;        //check the busy flag
        Lcdcommand (0x06) ;   //shift cursor right
        Lcdready ( ) ;        //check the busy flag
        Lcdcommand (0x80) ;   //line 1, position 0
        Lcdready ( ) ;
        Ldisplay ('M') ;      //Display 'M'
        Lcdready ( ) ;        // Check the busy flag
        Ldisplay ('I') ;      // Display 'I'
        Lcdready ( ) ;        // Check the busy flag
        Ldisplay ('C') ;      // Display 'C'
        Lcdready ( ) ;        // Check the busy flag
        Ldisplay ('R') ;      // Display 'R'
        Lcdready ( ) ;        // Check the busy flag
        Ldisplay ('O')        // Display 'O'
    }
        void Lcdcommand (unsigned char value)
    {
        Ldata = Value ;
        RS = 0 ;
        RW = 0 ;
        E = 1 ;
        Delay (250) ;
        E = 0 ;
    }
        void Ldisplay (unsigned char value)
    {
        Ldata = value ;
        RS = 1 ;
        RW = 0 ;
        E = 1 ;
        Delay (250) ;
        E = 0 ;
    }
    void Lcdready ( )
    {
        TRISD = 0xFF ;
        RS = 0 ;
        RW = 1 ;
        do
    {
        E = 1 ;
        Delay (250) ;
        E = 0 ;
    }
    while (busy = = 1) ;
    TRISD = 0 ;
    }
    void delay (unsigned int itime)
    {
        unsigned int x, y ;
        for (x = 0 ; x < itime ; x ++)
        for (y = 0 ; y < 165 ; y ++) ;
    }
```

Ex. 10.6.3 SPPU - May 2015, 8 Marks

Draw and explain the interfacing of LCD with Port D and Port E of PIC18F xxx microcontroller. Write C code to display 'WELCOME'.

Soln. :

Fig. P. 10.6.3 shows the interfacing of a 16 character × 2 line LCD module with the PIC microcontroller. The data lines are connected to Port D of microcontroller. The control lines RS, R/$\overline{W}$ and E are driven by Port E lines RE0, RE1 and RE2. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

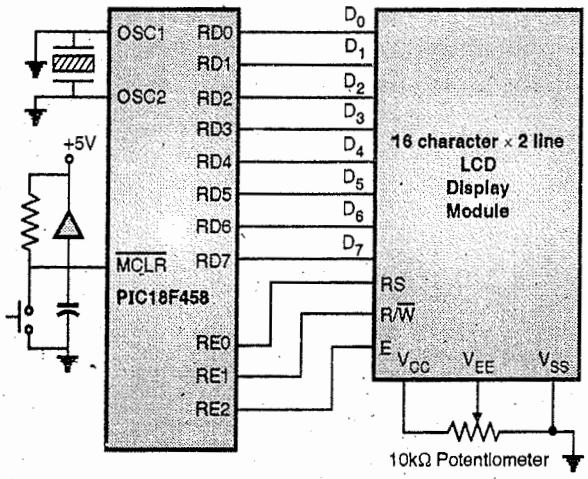


Fig. P. 10.6.3 : Interfacing 16 × 2 LCD to PIC18

C Program

```
# include <P18F458.h>
# define lcddatal PORTD              // PORTD LCD pins
# define RS PORTEbits.RE0            //RS = RE0

#define RW PORTEbits.RE1             //R/W = RE1
#define E PORTEbits.RE2              //EN = RE2
        void main (void)
        {
            TRISB = 0;        // Make port B an output port
            TRISD = 0 ;       // Make port D an output port
            E = 0 ;
            Delay (250) ;
            Lcdcmd (0x38) ;          // Initialize LCD 2 lines,
                                       5×7 matrix
            Delay (250) ;
            Lcdcmd (0x0E) ;          // Display on, cursor on
            Delay (250);
            Lcdcmd (0x01) ;          // Clear LCD
            Delay (250) ;
            Lcdcmd (0x06) ;          // Shift cursor right
            Delay (250) ;
            Lcdcmd (0×C0) ;          //Line 2, position 0
            Delay (250) ;
            Lcddata ('W') ;          //display letter 'W'
            Delay (250) ;
            Lcddata ('E') ;          //display letter 'E'
            Delay (250) ;
            Lcddata ('L');           //display letter 'L'
            Delay (250);
            Lcddata ('C');           // Display letter C
            Delay (250) ;
            Lcddata ('O')            //Display letter O
            Delay (250) ;
            Lcddata ('M')            //Display letter M
            Delay (250) ;
            Lcddata ('E')            //Display letter E
            Delay (250) ;
        }
        void Lcdcmd (unsigned char value) //command routine
        {
            Lcddatal = value;
            RS = 0;
            RW = 0 ;
            E = 1 ;
            Delay (250) ;
            E = 0 ;
        }
        void Lcddata (unsigned char value) //display routine
        {
            Lcddatal = value ;
            RS = 1 ;
            RW = 0 ;
            E = 1;
            Delay (250) ;
            E = 0 ;
        }
        void Delay (unsigned int itime)
        {
            unsigned int i, j ;
            for (i = 0 ; i<itime ; i++)
            for (j = 0; j < 165 ; j++) ;
        }
```

Ex. 10.6.4

SPPU - Dec. 2014, May 2015, May 2016, 8 Marks

Draw and explain the interfacing of LCD with port D and port E of PIC18XXXL microcontroller without Busy flag. Write C code to display 'S.P.P.U Pune'.

Soln. : Fig. P. 10.6.4 shows the interfacing of a 16 character × 2 line LCD module with the PIC microcontroller. The data lines are connected to Port

D of microcontroller. The control lines RS, R/$\overline{W}$ and E are driven by Port E lines RE0, RE1 and RE2. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

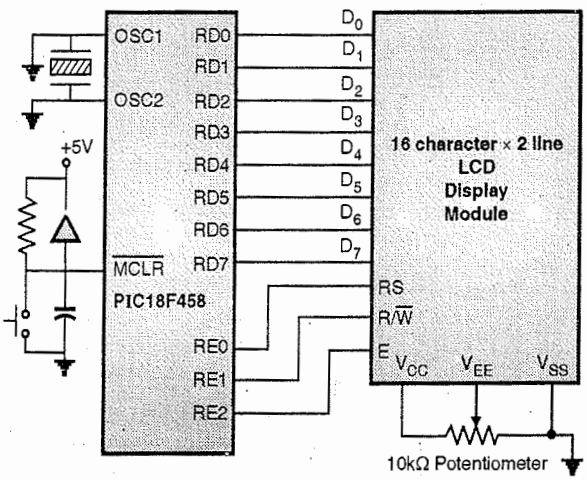


Fig. P. 10.6.4 : Interfacing 16 × 2 LCD to PIC18

C Program :

```
# include <P18F458.h>
# define lcddatal PORTD              // PORTD LCD pins
# define RS PORTEbits.RE0            //RS = RE0

#define RW PORTEbits.RE1             //R/W = RE1
#define E PORTEbits.RE2              //EN = RE2
        void main (void)
        {
            TRISB = 0;        // Make port B an output port
            TRISD = 0 ;       // Make port D an output port
            E = 0 ;
            Delay (250) ;
            Lcdcmd (0x38) ;          // Initialize LCD 2 lines,
                                       5×7 matrix
            Delay (250) ;
            Lcdcmd (0x0E) ;          // Display on, cursor on
            Delay (250);
            Lcdcmd (0x01) ;          // Clear LCD
            Delay (250) ;
            Lcdcmd (0x06) ;          // Shift cursor right
            Delay (250) ;
            Lcdcmd (0×C0) ;          //Line 2, position 0
            Delay (250) ;
            Lcddata ('S') ;          //display letter 'S'
            Delay (250) ;
            Lcddata ('.') ;          //display letter '.'
            Delay (250) ;
            Lcddata ('P');           //display letter 'P'
            Delay (250);
            Lcddata ('.');           //Display letter '.'
            Delay (250) ;
            Lcddata ('P')            //Display letter 'P'
            Delay (250) ;
            Lcddata ('.')            //Display letter '.'
            Delay (250) ;
            Lcddata ('U')            //Display letter 'U'
            Delay (250) ;
            Lcddata (' ')            //Display ' '
            Delay (250) ;
            Lcddata ('P')            //Display letter 'P'
            Delay (250) ;
            Lcddata ('U')            //Display letter 'U'
            Delay (250) ;
            Lcddata ('N')            //Display letter 'N'
            Delay (250) ;
            Lcddata ('E')            //Display letter 'E'
            Delay (250) ;
        }
        void Lcdcmd (unsigned char value) //command routine
        {
            Lcddata = value;
            RS = 0;
            RW = 0 ;
            E = 1 ;
            Delay (250) ;
            E = 0 ;
        }
        void Lcddata (unsigned char value) //display routine
        {
            Lcddata = value ;
            RS = 1 ;
            RW = 0 ;
            E = 1;
            Delay (250) ;
            E = 0 ;
        }
        void Delay (unsigned int itime)
        {
            unsigned int i, j ;
            for (i = 0 ;i<itime ; i++)
            for (j = 0; j <165 ;j++) ;
        }
```

Ex. 10.6.5 SPPU - Dec. 2016, 8 Marks

Draw an interfacing diagram and write an Embedded C program to interface 16 × 2 LCD with PIC 18FXX Microcontroller to display the "My College" message. Use 8 bit interface mode.

Soln. :

Fig. P. 10.6.5 shows the interfacing of a 16 character × 2 line LCD module with the PIC microcontroller. The data lines are connected to Port D of microcontroller. The control lines RS, R/$\overline{W}$ and E are driven by Port E lines RE0, RE1 and RE2. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.

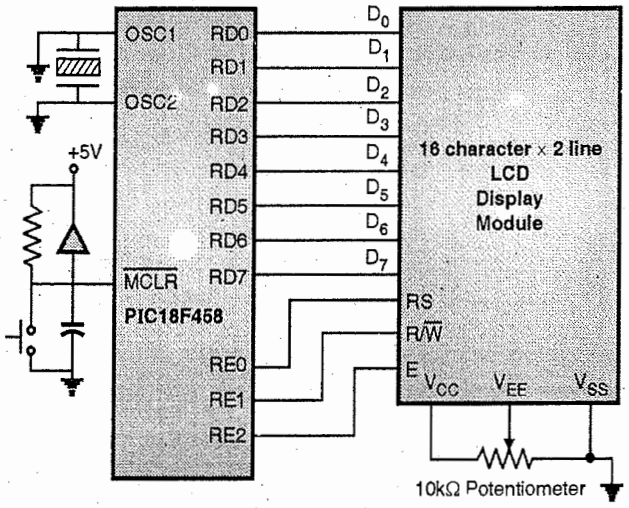


Fig. P. 10.6.5 : Interfacing 16 × 2 LCD to PIC18

C Program :

```
# include <P18F458.h>
# define lcddatal PORTD          // PORTD LCD pins
# define RS PORTEbits.RE0        //RS = RE0

#define RW PORTEbits.RE1         //R/W = RE1
#define E PORTEbits.RE2          //EN = RE2
        void main (void)
        {
            TRISB = 0;       // Make port B an output port
            TRISD = 0 ;      // Make port D an output port
            E = 0 ;
            Delay (250) ;
            Lcdcmd (0x38) ;          // Initialize LCD 2 lines,
                                       5×7 matrix
            Delay (250) ;
            Lcdcmd (0x0E) ;          // Display on, cursor on
            Delay (250);
            Lcdcmd (0x01) ;          // Clear LCD
            Delay (250) ;
            Lcdcmd (0x06) ;          // Shift cursor right
            Delay (250) ;
            Lcdcmd (0×C0) ;          //Line 2, position 0
            Delay (250) ;
            Lcddata ('M') ;          //display letter 'M'
            Delay (250) ;
            Lcddata ('y') ;          //display letter 'y'
            Delay (250) ;
            Lcddata ('C');           //display letter 'C'
            Delay (250);
            Lcddata ('o');           //Display letter 'o'
            Delay (250) ;
            Lcddata ('l')            //Display letter 'l'
            Delay (250) ;
            Lcddata ('l')            //Display letter 'l'
            Delay (250) ;
            Lcddata ('e')            //Display letter 'e'
            Delay (250) ;
            Lcddata ('g')            //Display 'g'
            Delay (250) ;
            Lcddata ('e')            //Display letter 'e'
            Delay (250) ;
        }
        void Lcdcmd (unsigned char value) //command routine
        {
            Lcddata = value;
            RS = 0;
            RW = 0 ;
            E = 1 ;
            Delay (250) ;
            E = 0 ;
        }
        void Lcddata (unsigned char value) //display routine
        {
            Lcddata = value ;
            RS = 1 ;
            RW = 0 ;
            E = 1;
            Delay (250) ;
            E = 0 ;
        }
        void Delay (unsigned int itime)
        {
            unsigned int i, j ;
            for (i = 0 ;i<itime ; i++)
            for (j = 0; j <165 ;j++) ;
        }
```

Ex. 10.6.6 SPPU - May 2016, 10 Marks

Design a frequency counter for counting number of pulses and display same on LCD.

OR

SPPU - Dec. 2016, 12 Marks

Design frequency counter for the range from DC to 5 MHz frequency using PIC18FXXX. Design and draw interfacing circuit. Also explain required flowchart.

Soln. :

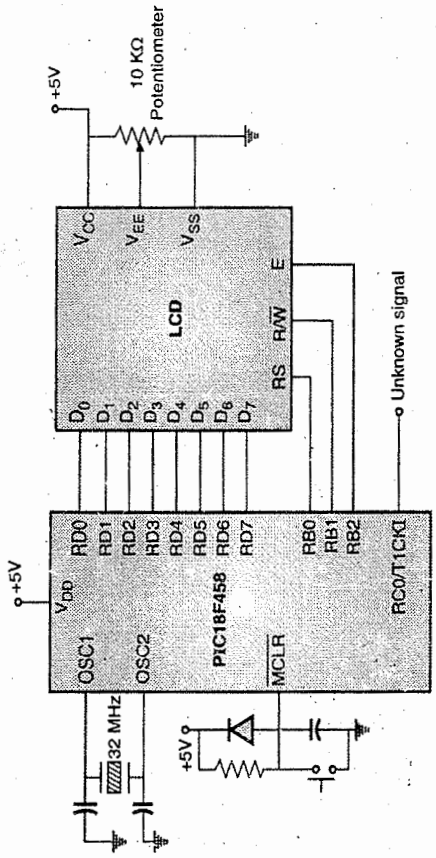


Fig. P. 10.6.6 : Interfacing diagram for frequency counter with display on LCD

A 1 second delay can be created by executing a delay of 100 msec, 10 times.

Program :

```
#include <P18F458.h>
unsigned int i = 0, t1overflow_cnt ;
unsigned short log frequency ;
#define Ldata PORTD
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define E PORTBbits.RB2
        void high_ISR (void) ;
        void low_ISR (void) ;
        #pragma code high_vector = 0x08
                                        // high priority interrupt
        void high_interrupt void        //begin at 0x08H
        {
            _asm
            gotohigh_ISR
            _endasm
        }
        #pragma code low_vector = 0x18H
                                        //low priority interrupt
        {
            _asm
            gotolow_ISR
            _endasm
        }
            # pragma code
            # pragma interrupt high_ISR
            void high_ISR (void)        // ISR of high priority
                                        interrupt
        {
            if (PIR1bits.TMR1IF)
            {
                PIR1bits.TMR1IF = 0 ;       // clear the Timer
                                            1 interrupt flag
                t1overflow_cnt++ ;
        //increment Timer 1 roll over count
            }
}
#pragma code
#pragma interrupt low_ISR           // ISR of low priority
                                    interrupt
void low_ISR (void)
        {
            _asm
            retfie 0            //Return if not Timer 0 interrupt
            _endasm
        }
            void delay (char value) ;
void ready ( )                  // function to check if LCD is
                                busy or ready.
{
        TRISD = 0xFF ;      // Port D = input port
        RS = 0 ;
```

```
        RW = 1 ;
        do
        {
            E = 0 ;
            for (i = 0 ; i< 25 ; i ++) ;
            E = 1 ;
        } while (busy = = 1) ;
        TRISD = 0 ;
}
void Lcdcommand (unsigned char d)  //command routine
{
            Ldata = d ;
            RS = 0 ;
            RW = 0 ;'
            E = 1 ;
            for (i = 0 ; i< 25 ; i ++) ;
            E = 0 ;
}
void Lcddata (unsigned char x)        // data routine
{
            ready ( ) ;
            Ldata = x ;          // write data on port D
            RS = 1 ;
            RW = 0 ;
            E = 1 ;
            for (i = 0 ; i< 25, i ++)
            E = 0 ;
}
void main (void)
{
            char t0_count, temp, temp1 ;
            for (i = 0 ; i< 5000 ; i ++)
                                // wait for sometime for LCD to
                                stabilize on power up
            Lcdcommand (0x38) ;
                                //issue command to initialize
                                16 × 2 LCD, 5 × 7 matrix.
            for (i = 0 ; i< 25, i ++) ;
                                // wait for sometime
                                using software delay
            Lcdcommand (0x0E) ;    //display on, cursor on
            for (i = 0, i< 25 ; i ++) ; // wait for sometime
                                using software delay
            Lcdcommand (0x01) ;    // clear display
            for (i = 0, i< 25 ; i ++) ; //wait for sometime
                                using software delay
            Lcdcommand (0x06) ;    //shift cursor right
            for (i = 0, i< 25 ; i ++) ;   // wait for sometime
                                using software delay
            Lcdcommand (0x80) ;    // issue command to
                                position cursor to first
                                line position 0
            for (i = 0; i< 25 ; i ++) ;
                        // wait for sometime using software delay
            t1overflow_cnt = 0 ;
                        // initialize Timer 1 overflow count to 0.
            frequency = 0 ;        // initialize frequency to 0.
            TMR1H = 0 ;            //   initialize Timer 1 to 0
            TMR1L = 0 ;            //
            PIR1bits.TMR1IF = 0    // clear Timer 1 interrupt
                                flag
            RCONbits.IPEN = 1 ;    // Enable priority
                                interrupt
            IPR1bits.TMR1IP = 1 ;     // Set Timer 1 interrupt
                                to high priority
            PIE1bits.TMR1IE = 1 ; // Enable Timer 1 roll
                                over interrupt
            T1CON = 0x83 ;    // Enable Timer 1 with T1CKI
                                and prescaler 1
            INTCON = 0xC0 ;        //Enable global and
                                peripheral interrupts
            delay (10) ;        // Create one second delay and
                                wait for interrupt
            INTCONbits.GIE = 0 ;  // Disable global
                                interrupts.
            temp = TMR1L ;         // save frequency low byte
            frequency = t1overflow_count * 65536 + TMR1H
* 256 + temp ; // compute frequency
            Lcdcommand (0x80);
            temp1 = TMR1H ;
            unsigned char x1, y, z ;
            x1 = temp1 & 0x0F ;     // mask upper 4 bits
            z = x1|0x30 ;           // make it ASCII
            PORTD = z ;
            Lcddata (z) ;       // Display the thousands digit
            y = temp1 and 0xF0 ;    // mask lower 4 bits
            y = y >>4 ;             // shift it to lower 4 bits
            z = y|0x30 ;            // Make it ASCII
            Lcddata (z) ;       // display the hundreds digit
```

```
        temp = TMR1L;
        x1 = temp & 0x0F ;          // mask upper 4 bits
        z = x1 | 0x30 ;             // Make it ASCII
        Lcddata (z) ;               // display the tens digit
        y = temp &0xF0 ;            // Mask lower 4 bits
        y = y >>4 ;                 // shift it to lower 4 bits
        z = y | 0x30 ;              // make it ASCII
        Lcddata (z) ;               // display the units digit
        Lcddata (" ") ;             // display single space
        Lcddata ("H") ; //  } display unit of frequency Hz
        Lcddata ("Z") ; //
while (1) ;
        }
        void delay (char value)
                    // delay of 100ms is executed 10 times
                       to get 1 second delay
    {
        int j ;
        T0CON = 0x83 ;              // Enable Timer 0 to set
                                    instruction clock, set
                                    prescaler to 16.
        for (i = 0 ; i< value ; i ++ )
    {
        TMR0H = 0x3C ;              // Put 15535 in Timer 0
                                    high byte and low
                                // byte so that it will roll over in
                                   50000 clock cycles
        TMR0L = 0xAF
        INTCONbits.TMR0IF = 0 ;     // Clear TMR0IF
                                    flag
    while (! (INTCONbits.TMR0IF)) ;
                                // wait till Timer 0 rolls over
    }
        Return ;
}
```

Ex. 10.6.7 **SPPU - May 2015, 16 Marks**

Design of DAS system for pressure monitoring system (use any suitable sensor).

Soln. : ASCX30AN is a 0-30 psi pressure transducer.

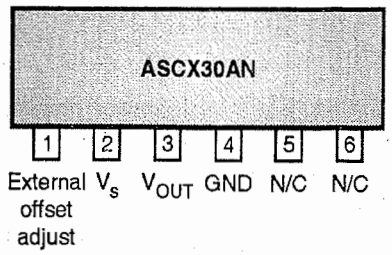


Fig. P. 10.6.7 : ASCX30AN pin diagram

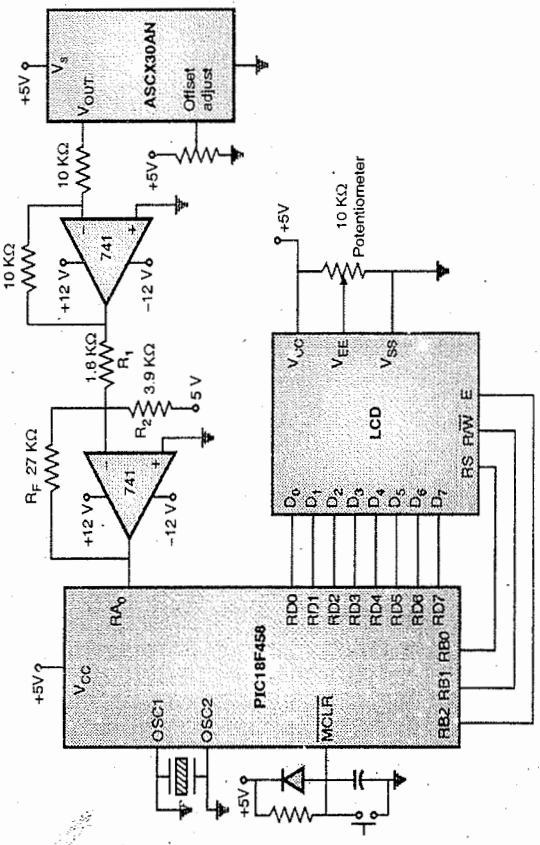


Fig. P. 10.6.7(a) : Interfacing

The output of ASCX30AN is 0.15 V/psi for a voltage range from 2.06V to 2.36V. This range is small. Hence, we need a level shifting and scaling circuit as shown in Fig. P. 10.6.7(a) to cover the voltage range from 0 to +5V. The offset adjustment is done with a potentiometer to 0.25V. Barometric pressure is from 948 to 1083.8 mbar. Inorder to find the barometric pressure we need to divide A/D result by 7.53V.

$$\text{Barometric pressure} = 948 + \frac{\text{A/D result}}{7.53}$$

$$= 48 + \frac{\text{A/D result} \times 100}{753}.$$

Algorithm :

(A) Main Program :

Step I : Initialize port B and port D as output ports.

Step II : Initialize the LCD.

Step III : Wait for some software delay after power up for display to stabilize.

Step IV : Initialize the LCD by giving 0x38 instruction to the command subroutine.

Step V : Wait for some software delay.
Step VI : Issue the command 0x0E to command subroutine for display on, cursor on.
Step VII : Wait for small software delay.
Step VIII : Check the LCD busy flag.
Step IX : Issue the command 0x01 for clearing display to command subroutine.
Step X : Wait for small software delay.
Step XI : Check the LCD busy flag.
Step XII : Issue the command 0x06 for shifting cursor right to command subroutine.
Step XIII : Wait for small software delay.
Step XIV : Check the LCD busy flag.
Step XV : Issue the command 0x80 to position cursor at line 1, position 1 using command subroutine.
Step XVI : Wait for small software delay.
Step XVII : Check the LCD busy flag.
Step XVIII : Issue 0x81 to ADCON0 to turn on ADC.
Step XIX : Issue 0xC5 to ADCON1.
Step XX : Issue start of conversion.
Step XXI : Wait for end of conversion i.e. DONE bit = 1. It indicates that ADC has completed conversion.
Step XXII: Save the result in ADRESL and ADRESH registers.
Step XXIII : Divide the ADC output by 7.53 to get barometric pressure.
Step XXIV: Send ADC output to port D for displaying it on LCD.
Step XXV: Calculate the digits and display it on LCD.
Step XXVI : Also display unit of pressure (mbar).

(B) Command subroutine :

Step I : Give the instruction to port D connected to data bus of LCD.
Step II : Make RS = '0' for command.
Step III : Make R/$\overline{W}$ = '0' to indicate write.
Step IV : Make E = '1'
Step V : Wait for 120 μsec
Step VI : Make E = '0'
Step VII : Return.
} High-to-low pulse on E pin to latch the command.

(C) Display subroutine :

Step I : Check if LCD is ready by calling ready subroutine.
Step II : Give data to port D, connected to data bus of LCD.
Step III : Make RS = '1' to indicate data.
Step IV : Make R/$\overline{W}$ = '0' to indicate write.
Step V : Make E = '0'
Step VI : Wait for 120 μsec
Step VII : Make E = '0'
Step VIII : Return.
} High-to-low pulse on E pin to latch the data

(D) Ready subroutine :

Step I : Make RD7 = 1.
Step II : Make RS = '0' to indicate instruction.
Step III : Make R/$\overline{W}$ = '1' to indicate read.
Step IV : Make E = 1.
Step V : Make E = 0.
Step VI : Check if busy pin = '0'. If it is '1' it indicates that LCD is busy, hence again make E = '0' then E = '1' and check busy pin. Repeat this till busy pin = 0.
Step VII : Return.

Program :

```
#include <P18F458.h>
unsigned int i = 0 ;
#define Ldata PORTD
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define E PORTBbits.RB2
#define temp PORTAbits.RA0
#define Vref PORTAbits.RA3
#define busy PORTDbits.RD7
    void ready ( )                  // function to check if
                                    LCD is busy or ready
    {
      TRISD = 0xFF ;                // Port D = input port
      RS = 0 ;
      RW = 1 ;
      do                            // wait for busy flag
    {
      E = 0 ;
      for (i = 0 ; i < 25 ; i++) ;
      E = 1 ;
    } while (busy = = 1) ;
    TRISD = 0 ;
    }
    void Lcdcommand (unsigned char value)
    {
    Ldata = value ;
    RS = 0 ;
```

```
    RW = 0 ;
    E = 1 ;
    for (i = 0 ; i < 25 ; i++) ;
    E = 0 ;
    }
    void Lcddisplay (unsigned char d)
    {
        ready ( ) ;
        Ldata = d ;        // write data on port D so that it is
                              given to LCD for displaying.
        RS = 1 ;
        RW = 0 ;
        E = 1 ;
        for (i = 0 ; i < 25 ; i++) ;
        E = 0 ;
    }
    void main ( )
    {
        unsigned char ADC_output, Lo_byte, Hi_byte ,
output;
        for (i = 0 ; i < 5000 ; i++) ;
                              //wait for sometime for LCD
                                to stabilize on power up.
        Lcdcommand (0x38) ;   //issue command to
                              initialize 16 × 2 LCD, 2 lines,
                              // 5 × 7 matrix
        for (i = 0 ; i < 25 ; i++) ;     //wait for sometime
                                using software delay
        Lcdcommand (0x0E) ;   // display on, cursor on
        for (i = 0 ; i < 25 ; i++) ;     //wait for sometime
                                using software delay
        Lcdready ( ) ;
        Lcdcommand(0x01)      //clear display
        for (i = 0 ; i < 25 ; i ++) ;    //wait for sometime
                                using software delay
        Lcdready ( ) ;
        Lcdcommand (0x06) ;           // shift cursor right
        for (i = 0 ; i < 25 ; i++) ;  // wait for sometime
        Lcdready ( ) ;
        Lcdcommand (0x80) ;   // issue the command to
                                position the
                              //  cursor on the first
                              position of line 1.
        for (i = 0 ; i < 25 ; i++) ;
                  // wait for sometime using software delay
        Lcdready ( ) ;
        TRISD = 0 ;       //make port D an output port
        TRISAbits.TRISA0 = 1 ; // Make RA0 an input pin
        TRISAbits.TRISA3 = 1 ; // RA3 = 1 for Vref input
        ADCON0 = 0x81 ;       //Channel 0, ADC on, fosc/64
        ADCON1 = 0xC5 ;       // fosc/64, right justified, AN0
                                = analog input, AN3 = Vref+
    while (1)
        {
        for (i = 0 ; i < 25 ; i++) ;
                  //wait for sometime using software delay
        ADCON0bits.GO = 1          //start conversion
        while (ADCON0bits.DONE = = 1) ;
                              // wait for end of conversion
        Lo_byte = ADRESL ;         // save low byte
        Hi_byte = ADRESH ;         // save high byte
        ADC_output = Lo_byte | High byte ;
        output = 948 + (ADC_output * 100) / 7.53 ;
    PORTD = output ;
        Lcddisplay (output/100) ;
        Lcddisplay ((output %100)/10);  // Display
        Lcddisplay (output % 10) ;       the
        Lcddisplay ("m") ;               pressure
                                         count
    Lcddisplay ("b") ;    //The pressure is in mbar
    Lcddisplay ("a") ;
    Lcddisplay ("r") ;
    Lcdcommand (0x80) ;
        }
    }
```

Ex. 10.6.8 **SPPU - Dec. 2014, 16 Marks**

Design a PIC18 based data acquisition system for temperature measurement using LM35. Write the corresponding program to display the temperature on LCD.

Soln. :

The LM35 operates over a temperature range of – 55°C to + 150°C with an output of 10 mV for each degree centigrade. eg for 80°C temperature the output will be 80 × 10 = 800 mV.

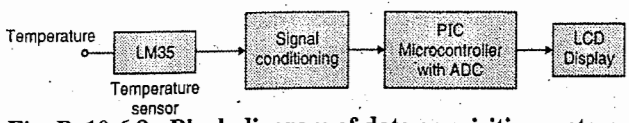


Fig. P. 10.6.8 : Block diagram of data acquisition system used for temperature measurement

- The A/D converter of PIC18F458 is of 10 bit resolution and maximum number of steps = 2^{10} = 1024 Steps. For each °C LM35 gives 10 mV output.

- For a step size of 10 mV,

  V_{out} = 10 mV × 1024 steps = 10240 mV
  = 10.24 V

  However, this much voltage is not acceptable.
- If a step size of 2.5 mV is selected,

  V_{out} = 2.5 mV × 1024
  V_{out} = 2560 mV = 2.56 V

  The binary output of the ADC is $\left(\frac{10\text{ mV}}{2.5\text{ mV}} = 4\right)$

  i.e. 4 times the real temperature. We select V_{ref} = 2.56 V.
- The real temperature can be got by dividing the A/D output by 4.
- Fig. P. 10.6.8(a) shows the interfacing diagram of PIC18F458 with LM35 temperature sensor.

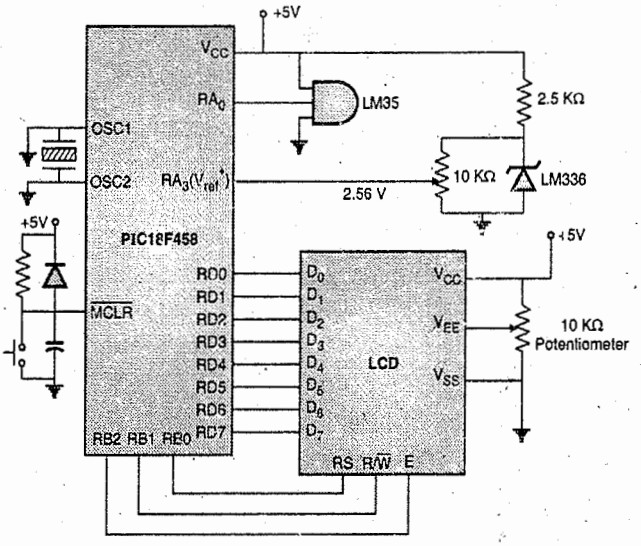


Fig. P. 10.6.8(a) : Interfacing diagram of PIC18F458 based data acquisition system for measurement of temperature

- The LM336 zener diode is used for maintaining V_{ref} = 2.56 V and to overcome the fluctuations in power supply.
- Table P. 10.6.8 gives the 10 bit output values for ADC and the temperature with V_{ref} = 2.56 V.

Table P. 10.6.8

Temperature °F	V_{in} (mV)	Number of steps $\frac{V_{in}}{2.5}$	10 bit A/D output	Real temperature in binary A/D output / 4
0	0	0	0000000000	0000000000
1	10	4	0000000100	0000000001
2	20	8	0000001000	0000000010
5	50	20	0000010100	0000000101
10	100	40	0000101000	0000001010
20	200	80	0001010000	0000010100
30	300	120	0001111000	0000011110
40	400	160	0010100000	0000101000
50	500	200	0011001000	0000110010
60	600	240	0011110000	0000111100
70	700	280	0100011000	0001000110
80	800	320	0101000000	0001010000
90	900	360	0101101000	0001011010
100	1000	400	0110010000	0001100100
110	1100	440	0110111000	0001101110
120	1200	480	0111100000	0001111000
130	1300	520	1000001000	0010000010
140	1400	560	1000110000	0010001100
150	1500	600	1001011000	0010010110

Algorithm :

(A) Main Program :

Step I : Initialize port B and port D as output ports.

Step II : Initialize the LCD.

Step III : Wait for some software delay after power up for display to stabilize.

Step IV : Initialize the LCD by giving 0x38 instruction to the command subroutine.

Step V : Wait for some software delay.

Step VI : Issue the command 0x0E to command subroutine for display on, cursor on.

Step VII : Wait for small software delay.

Step VIII : Check the LCD busy flag.

Step IX : Issue the command 0x01 for clearing display to command subroutine.

Step X : Wait for small software delay.

Step XI : Check the LCD busy flag.

Step XII : Issue the command 0x06 for shifting cursor right to command subroutine.

Step XIII : Wait for small software delay.

Step XIV : Check the LCD busy flag.

Step XV : Issue the command 0x80 to position cursor at line 1, position 1 using command subroutine.

Step XVI : Wait for small software delay.

Step XVII : Check the LCD busy flag.

Step XVIII : Issue 0x81 to ADCON0 to turn on ADC.

Step XIX : Issue 0xC5 to ADCON1.

Step XX : Issue start of conversion.

Step XXI : Wait for end of conversion i.e. DONE bit = 1. It indicates that ADC has completed conversion.

Step XXII : Save the result in ADRESL and ADRESH registers.

Step XXIII : Divide the ADC output by 4 to get real temperature i.e.

(i) right shift ADRESL by 2 bits

(ii) rotate ADRESH 2 bits

(iii) OR ADRESH with ADRESL to get 8 bit temperature output

Step XXIV : Send ADC output to port D for displaying it on LCD.

Step XXV : Calculate the digits and display it on LCD.

Step XXVI : Also display unit of temperature (°C).

(B) Command subroutine :

Step I : Give the instruction to port D connected to data bus of LCD.

Step II : Make RS = '0' for command.

Step III : Make R/$\overline{W}$ = '0' to indicate write.

Step IV : Make E = '1'

Step V : Wait for 120 μsec

Step VI : Make E = '0'

Step VII : Return.

} High-to-low pulse on E pin to latch the command.

(C) Display subroutine :

Step I : Check if LCD is ready by calling ready subroutine.

Step II : Give data to port D, connected to data bus of LCD.

Step III : Make RS = '1' to indicate data.

Step IV : Make R/$\overline{W}$ = '0' to indicate write.

Step V : Make E = '0'

Step VI : Wait for 120 μsec

Step VII : Make E = '0'

} High-to-low pulse on E pin to latch the data

Step VIII : Return.

(D) Ready subroutine :

Step I : Make RD7 = 1.

Step II : Make RS = '0' to indicate instruction.

Step III : Make R/$\overline{W}$ = '1' to indicate read.

Step IV : Make E = 1.

Step V : Make E = 0.

Step VI : Check if busy pin = '0'. If it is '1' it indicates that LCD is busy, hence again make E = '0' then E = '1' and check busy pin. Repeat this till busy pin = 0.

Step VII : Return.

Program :

```
#include <P18F458.h>
unsigned int i = 0 ;
#define Ldata PORTD
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define E PORTBbits.RB2
#define temp PORTAbits.RA0
#define Vref PORTAbits.RA3
#define busy PORTDbits.RD7
void ready ()   // function to check if LCD is busy or ready
{
        TRISD = 0xFF ;              // Port D = input port
        RS = 0 ;
        RW = 1 ;
        do                          // wait for busy flag
        {
            E = 0 ;
            for (i = 0 ; i < 25 ; i++) ;
            E = 1 ;
        } while (busy = = 1) ;
        TRISD = 0 ;
}
void Lcdcommand (unsigned char value)
```

```
{
        Ldata = value ;
        RS = 0 ;
        RW = 0 ;
        E = 1 ;
        for (i = 0 ; i < 25 ; i++) ;
        E = 0 ;
}
void Lcddisplay (unsigned char d)
{
        ready ( ) ;
        Ldata = d ;      // write data on port D so that it is
                         given to LCD for displaying.
        RS = 1 ;
        RW = 0 ;
        E = 1 ;
        for (i = 0 ; i < 25 ; i++) ;
        E = 0 ;
}
void main ( )
{
        unsigned char ADC_output, Lo_byte, Hi_byte ;
        for (i = 0 ; i < 5000 ; i++) ;
           //wait for sometime for LCD to stabilize on power up.
        Lcdcommand (0x38) ;
              //issue command to initialize 16 × 2 LCD,
              2 lines, 5 × 7 matrix
        for (i = 0 ; i < 25 ; i++) ;
                    //wait for sometime using software delay
        Lcdcommand (0x0E) ;        // display on, cursor on
        for (i = 0 ; i < 25 ; i++) ;
                    // wait for sometime using software delay
        Lcdready ( ) ;
        Lcdcommand(0x01)           //clear display
        for (i = 0 ; i < 25 ; i ++) ;
                    // wait for sometime using software delay
        Lcdready ( ) ;
        Lcdcommand (0x06) ;                // shift cursor right
        for (i = 0 ; i < 25 ; i++) ; // wait for sometime
        Lcdready ( ) ;
        Lcdcommand (0x80) ;        // issue the command to
                                      position the
                                      cursor on the first
                                      position of line 1.
        for (i = 0 ; i < 25 ; i++) ;
                    // wait for sometime using software delay
        Lcdready ( ) ;
        TRISD = 0 ;          //make port D an output port
        TRISAbits.TRISA0 = 1 ;     // Make RA0 an
                                      input pin
        TRISAbits.TRISA3 = 1 ;     // RA3 = 1 for Vref input
        ADCON0 = 0x81 ;            //Channel 0, ADC on, fosc/64
        ADCON1 = 0xC5 ;            // fosc/64, right justified, AN0
                                   = analog input, AN3 = Vref+
while (1)
{
        for (i = 0 ; i < 25 ; i++) ;
                    //wait for sometime using software delay
        ADCON0bits.GO = 1          //start conversion
        while (ADCON0bits.DONE = = 1) ;
                                   // wait for end of conversion
        Lo_byte = ADRESL ;         // save low byte
        Hi_byte = ADRESH ;         // save high byte
        Lo_byte >> = 2 ;   // shift right by
                           //2 bits
        Lo_byte & = 0x3F;  // mask 2 upper
                           // bits
        Hi_byte << = 6 ;   //shift 6 times
                           //to the left
        Hi_byte & = 0xC0 ; // mask the
                           // lower 6 bits
        ADC_output = Lo_byte|High byte ;
        PORTD = ADC_output ;
        Lcddisplay (ADC_output/100) ;
        Lcddisplay ((ADC_output%100)/10) ;
        Lcddisplay (ADC_output % 10) ;
        Lcddisplay (“ ”) ;
        Lcddisplay (‘C’) ;     //The temperature is in °C
        Lcdcommand (0x80) ;
        }
}
```

10 bit ADC output is divided by 4 to obtain real temperature

Display the temperature count

Ex. 10.6.9 **Lab Assignment**

Write a program for interfacing button, LED, relay & buzzer as follows

A. On pressing button1 relay and buzzer is turned ON and LED's start chasing from left to right

B. On pressing button2 relay and buzzer is turned OFF and LED start chasing from right to left.

Soln. :

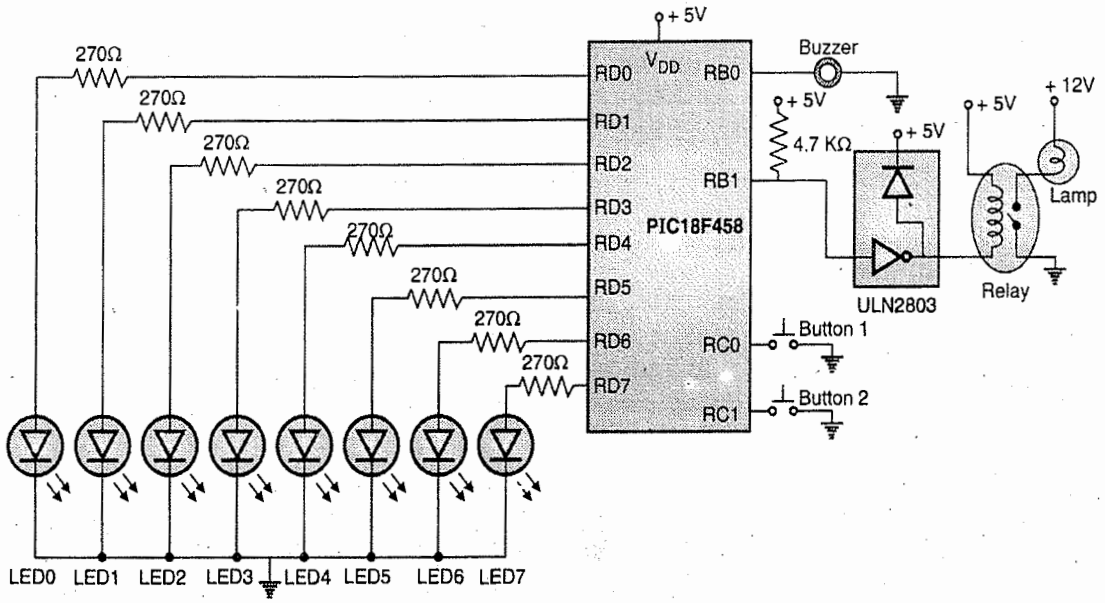


Fig. P. 10.6.9 : Interfacing button, LEDs, relay and buzzer to PIC18F458

Fig. P. 10.6.9 shows the interfacing of a buzzer, relay, button and LEDs to PIC18F458 microcontroller. The buzzer is connected to pin RB0. The relay is connected to pin RB1. For energizing the relay ULN2803 driver is used. Two buttons are connected to pins RC0 and RC1. Eight LEDs are connected to Port D pins RD0 to RD7.

Program :

```
# include <P18F458.h>
# define buzzer PORTBbits.RB0
# define relay PORTBbits.RB1
# define button1 PORTCbits.RC0
# define button2 PORTCbits.RC1
# define LED0 PORTDbits.RD0
# define LED1 PORTDbits.RD1
# define LED2 PORTDbits.RD2
# define LED3 PORTDbits.RD3
# define LED4 PORTDbits.RD4
# define LED5 PORTDbits.RD5
# define LED6 PORTDbits.RD6
# define LED7 PORTDbits.RD7
void delay (unsigned int) ;
void main (void).
    {
        TRISCbits.TRISC0 = 1 ;    // Make RC0 = 1
                                     i.e. an input pin
        TRISCbits.TRISC1 = 1 ;    // Make RC1 = 1
                                     i.e. an input pin
        TRISB = 0 ;               // Make port B an output port
        TRISD = 0 ;               // Make port C an output port
    while (1)
    {
        if (button1 = = 0)
        {
            relay = 1 ;           // Turn on Relay
            buzzer = 1 ;          // Turn on buzzer
            LED0 = 1 ;            // Turn on LEDs from left
                                  to right i.e. LED0 to LED7
            delay (10) ;
            LED1 = 1 ;            // Turn on LED1
            delay (10) ;
            LED2 = 1 ;
            delay (10) ;
            LED3 = 1 ;
            delay (10) ;
            LED4 = 1 ;
            delay (10) ;
            LED5 = 1 ;
            delay (10) ;
            LED6 = 1 ;
            delay (10) ;
            LED7 = 1 ;
            delay (10) ;
```

UNIT - V

CHAPTER 11

CCP Programming

Syllabus Topic : CCP Modes

11.1 CCP Modes

- The PIC18 family supports 0 to 5 **Compare/ Capture / PWM (CCP)** modules depending on the PIC family member .The modules are called as CCP1 , CCP2 , CCP3 , CCP4 and CCP5 respectively.
- Mostly the PWM feature of this module is used for controlling the DC Motors .
- The PIC18F458 microcontroller has one CCP MODULE and one Enhanced ECCP module.

11.2 CCP Timer Resources

- Table 11.2.1 shows the PIC18 Timers used for the different CCP modes. For selecting the timer for CCP module for compare and capture modes the T3CON register is used.

Table 11.2.1 : PIC18 Timers used for the different CCP modes

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

11.3 CCP Registers

Every CCP module supports 3 registers. They are :

1. CCP1CON control register
2. 16 bit register CCPR1H : CCPR1L.
3. 10 bit duty cycle register in the PWM mode.

11.3.1 CCP1CON Control Register

SPPU - Dec. 14

University Question

Q. Explain SFR CCP1CON register in detail.

(Dec. 2014, 4 Marks)

- Fig 11.3.1 shows the CCP1CON control register. It is used for selecting the operating mode of the CCP module. It also controls the operation of the CCP1 module.

-	-	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0

bit 7-6 Unimplemented : Read as '0'

bit 5-4 DC1B1 : DC1B0 :

Capture mode and Compare mode : Unused.

PWM mode :

These bits are the two LSBs of the 10-bit PWM duty cycle. The upper eight bits (DC1B9 : DC1B2) of the duty cycle are found in CCPR1L register.

bit 3-0 CCP1M3 : CCP1M0 : CCP1 Mode Select bits

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

0001 = Reserved

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

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

0100 = Capture mode, every falling edge

0101 = Capture 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 special event (CCP1IF bit is set ; CCP resets TMR1 or TMR3 and starts an A/D conversion if the A/D module is enabled)

11xx = PWM mode

Fig. 11.3.1 : CCP1CON register

11.3.2 CCPR High and Low Registers

- CCPR in compare and capture mode can be accessed as a 16 bit capture or 16 bit compare register .The low byte register is called as CCPRIL and high byte register is called CCPR1H. Like the other SFRs both these registers can be accessed.

Size : The registers CCPR1L and CCPR1H are of 8 bit.

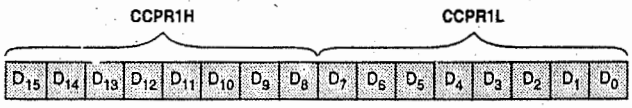


Fig. 11.3.2 :CCP1 high and low registers

11.3.3 10 bit Duty Cycle Register

- A 10 bit register comprising of 8 bits from CCPR1L register and 2 bits from CCP1CON register is used for setting the duty cycle.
- Of the 10 bits, the upper 8 bits are from the CCPR1L register and lower 2 bits are DC1B2 : DC1B1 of CCP1CON register. The lower two bits are for the decimal part of duty cycle as shown in Table 11.3.1.

Table 11.3.1

DC1B2	DC1B1	Decimal point
0	0	0
0	1	0.25
1	0	0.5
1	1	0.75

Syllabus Topic : Compare Mode

11.4 Compare Mode

SPPU - Dec. 14, Dec. 16

University Question

Q. Explain compare mode of operation of PIC18XXX in detail. **(Dec. 2014, Dec. 2016, 4 Marks)**

- By programming the CCP1CON register bits the compare mode of CCP module can be selected.
- An event that is external to the PIC18 microcontroller can also be created by the compare mode.
- This can be achieved by starting an A/D conversion or turning on the device that is connected to the RC2 pin. i.e. CCP pin.
- In this the 16-bit CCPR1(CCPR1H: CCPR1L) register value is constantly compared with the values in of the Timer1 register (TMR1H: TMR1L) or the timer 3 register values (TMR3H : TMR3L). In case the values are equal or match then in such conditions an event is generated.
- In case an event is generated , the CCP1 pin can do one of the following :

1. It can drive CCP1 pin low.
2. It can drive the CCP1 pin high.
3. The CCP1 pin can remain unchanged.
4. The CCP1 pin can toggle its state.
5. The CCP1 pin can trigger a special event with hardware interrupt and then clear the timer.

- The control bits CCP1M3:CCP1M0 in the CCP1CON register are used for selecting these actions. The interrupt flag bit CCP1IF in the PIR1 register is set.
- Only special event trigger clears the timer. For all other actions we need to clear the Timer. Fig. 11.4.1 shows the block diagram in Compare mode.

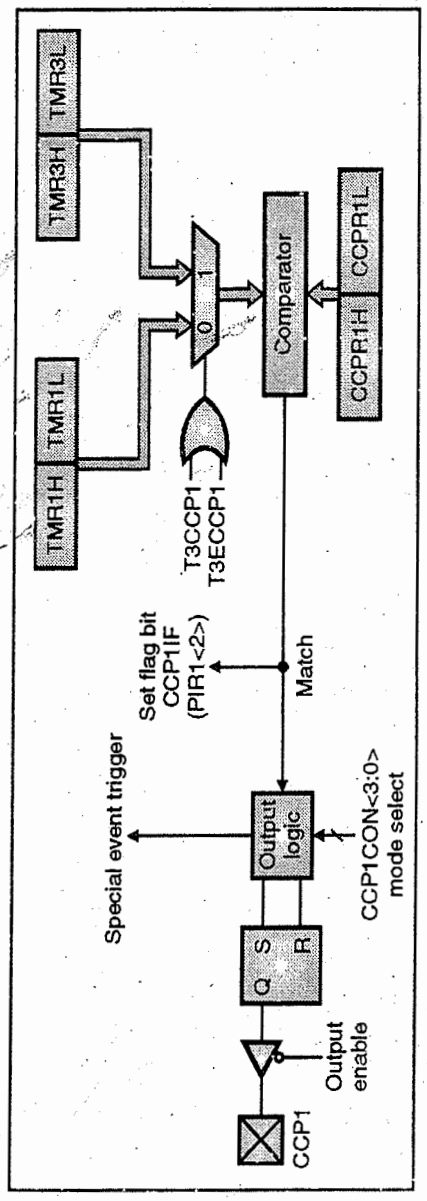


Fig. 11.4.1 : Compare Mode Block diagram

11.4.1 Steps for Programming in Compare Mode

- Following are the steps for programming in Compare mode :

Step I	:	Initialize the CCP1CON register for compare mode operation.
Step II	:	Initialize the T3CON Register for selecting the desired timer (Timer 1 or Timer 3).
Step III	:	Initialize the CCPR high and low registers for compare mode operation.
Step IV	:	Initialize the CCP1 pin i.e. RC2 pin as an output pin.
Step V	:	Initialize the Timer 1 or 3 register values for compare mode operation.
Step VI	:	Start the selected Timer.
Step VII	:	Keep checking the CCP1IF flag .

Ex. 11.4.1

A 1 Hz pulse is given to timer 1 (T1CKI) and an LED is connected to the CCP1 pin. Write a program that counts the number of pulses given to Timer 1 and when the count reaches 50, the LED toggles.

Soln. : C18 program :

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0x02;     // Initialize CCP1CON register
                        //  for compare mode.
    T3CON = 0x00 ;      // Initialize T3CON for Timer 1
                        //  operation
    T1CON = 0x02;       // Timer 1 , 16 bit , external
                        clock , no prescaler
    CCPR1L = 50;        // load CCPR1L
    CCPR1H=00;          // LOAD CCPR1H
    TRISCbits.TRISC2 = 0;       //make CCP1 pin output
    TRISCBITS.TRISC0=1          // make T1CKI an input
while(1)
    {
     TMR1H = 0;
     TMR1L=0;
     PIR1bits. CCP1IF = 0;      //clear CCP1IF flag
     T1CONbits.TMR1ON = 1;  //start timer 1
     while (PIR1bits. CCP1IF= = 0);
       //wait FOR CCP1IF  (when match occurs LED toggles
                                // and CCP1IF =1
     T1CONbits.TMR1ON=0;    // stop Timer 1
    }
}
```

Ex 11.4.2

Write a program to generate a square wave with frequency 10 KHz and 50% duty cycle on the CCP1 pin. Use Timer 1.

Soln. :

Square wave frequency = 10 KHz

$$\therefore \quad 1 \text{ clock pulse} = \frac{1}{10 \text{ KHz}} = 100\ \mu\text{sec}$$

$\therefore$ 50 μsec is the 'ON' time and 50 μsec is 'OFF' time assuming a duty cycle of 50%.

Hence, a delay of 50 μsec is needed.

$$\therefore \quad \text{Count} = \frac{50\ \mu\text{s}}{0.4\ \mu\text{s}} \quad \text{(assuming XTAL = 10 MHz)}$$

$\therefore$ Count = 125

Count = 65536 – 125 = $(65411)_{10}$

= FF83HCCPR1H = FFH

CCPR1L = 83H

C18 program :

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0x02;     // Initialize  CCP1CON register
                         for compare mode.
    T3CON = 0x 09       // Initialize T3CON for
                       //  Timer 1 operation
    T1CON = 0x00;       // Initialize T1CON internal
                        //clock , 1: 1 prescaler
    CCPR1L = 83;        // load CCPR1L
    CCPR1H=FF;                  // LOAD CCPR1H
    TRISCbits.TRISC2 = 0;       //make CCP1 pin output
while(1)
    {
     TMR1H = 0;
     TMR1L=0;
     PIR1bits. CCP1IF = 0;      //clear CCP1IF flag
     T1CONbits.TMR1ON = 1;  //start timer 1
     while (PIR1bits. CCP1IF= = 0); //wait FOR CCP1IF
     T1CONbits.TMR1ON=0;    // stop Timer 1
    }
}
```

Ex 11.4.3 SPPU - Dec. 15, 8 Marks

Write a program to generate a square wave with frequency 2.5 KHz and 50% duty cycle on the CCP1 pin Using timer 3.

Soln. :

Let XTAL = 10 MHz

$$\text{Timer clock frequency} = \frac{f_{osc}}{4} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5 \text{ MHz}} = 0.4\ \mu\text{s}$$

Frequency required = 2500 Hz

$$\text{Period} = \frac{1}{2500} = 400\ \mu\text{s}$$

Assuming 50% duty cycle, period = 200 μs

$$\therefore \frac{200\ \mu\text{s}}{0.4\ \mu\text{s}} = 500$$

$\therefore$ Count = 65536 – 500 = $(65036)_{10}$
= FE0CH
CCPR1H = FE H
CCPR1L = 0C H

C18 program :

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0x02;     // Initialize CCP1CON
                        // register for compare mode.
    T3CON = 0x 42 ;     // Initialize T3CON for Timer 3
                        // operation
    CCPR1L = 0C;            // load CCPR1L
    CCPR1H=FE;              / LOAD CCPR1H
    TRISCbits.TRISC2 = 0;   //make CCP1 pin output
    TRISCBITS.TRISC0=1      // make T1CKI an input
while(1)
    {
     TMR3H = 0;
     TMR3L=0;
     PIR1bits. CCP1IF = 0;      //clear CCP1IF flag
     T3CONbits.TMR3ON = 1;  //start timer 3
     while (PIR1bits. CCP1IF= = 0); //wait FOR CCP1IF
     T3CONbits.TMR3ON=0;            // stop Timer 3
    }
}
```

Ex 11.4.4

A 1 Hz pulse is given to timer 1 (T1CKI) and an LED is connected to the CCP1 pin. Write a program that counts the number of pulses given to Timer 3 and when the count reaches 100, the LED toggles.

Soln. : C18 program :

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0x02;     // Initialize CCP1CON register
                        // for compare mode.
    T3CON = 0x 42 ;     // Initialize T3CON for Timer 3
                        //operation
    CCPR1L = 100;       // load CCPR1L
    CCPR1H=00;          // LOAD CCPR1H
    TRISCbits.TRISC2 = 0;   //make CCP1 pin output
    TRISCBITS.TRISC0=1      // make T1CKI an input
while(1)
    {
     TMR3H = 0;
     TMR3L=0;
     PIR1bits. CCP1IF = 0;      //clear CCP1IF flag
     T3CONbits.TMR3ON = 1;  //start timer 3
     while (PIR1bits. CCP1IF= = 0); //wait FOR CCP1IF
     T3CONbits.TMR3ON=0;            // stop Timer 3
    }
}
```

Syllabus Topic : Capture Mode

11.5 Capture Mode

SPPU - Dec. 16

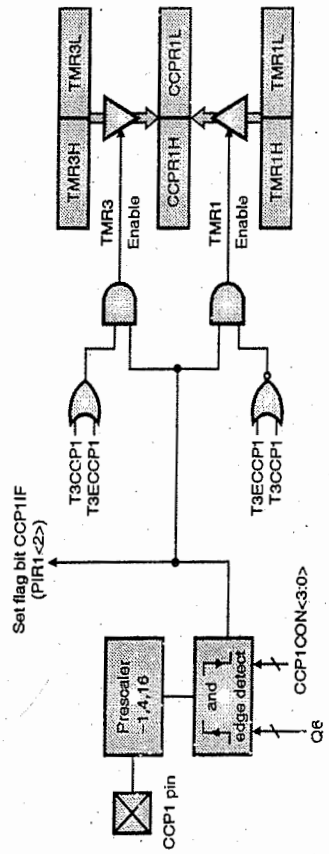
University Question

Q. Explain capture mode of PIC18FXXX in detail. **(Dec. 2016, 4 Marks)**

- By programming the CCP1CON register bits the Capture mode of CCP module can be selected.
- An event that is detected at the CCP1 pin will load the contents of Timer 1 or Timer 3 into the CCPR1H: CCPR1L register.
- For capture mode the CCP1 pin operates as an input pin for capturing the events.
- When an event on CCP1 is detected in Capture mode, the 16 bit values from the TMRxH: TMRxL registers of Timer 1 or Timer 3 is captured into CCPR1H:CCPR1L register .The event can be detected on :

1. Every rising edge 2. Every falling edge
3. Every 4th rising edge 4. Every 16th rising edge

- If another capture occurs before the value in register CCPR1 register is read, the old captured value will be lost.
- The control bits CCP1M3:CCP1M0 in the CCP1CON register are used for selecting the event. The interrupt flag bit CCP1IF in the PIR1 register is set.



Note : I/O pins have diode protection to V_{DD} and V_{SS}.

Fig. 11.5.1: Capture Mode Block Diagram

11.5.1 Steps for Programming in Capture mode

Following are the steps for programming in Capture mode :

Step I :	Initialize the CCP1CON register for capture mode operation
Step II :	Initialize the T3CON Register for selecting the desired timer (Timer 1 or Timer 3).
Step III :	Initialize the CCP1 pin i.e. RC2 pin as an input pin.
Step IV :	Read the Timer 1 or Timer 3 values on first rising edge and store it
Step V :	Read the Timer 1 or Timer 3 values on second rising edge and store it
Step VI :	Subtract the value obtained in step 4 from value obtained in step 5 .

11.5.2 Measuring Period of a Pulse

- Fig. 11.5.2 shows how capture mode is used to measure the period of a pulse applied to the CCP1 pin of the CCP module in terms of the system clock period. Generally the measurement is done with respect to T_{SCLK}(system clock) i.e. $\frac{f_{osc}}{4}$.

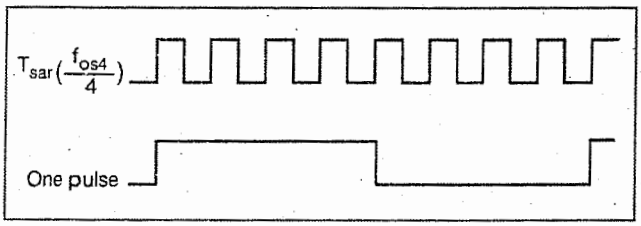


Fig. 11.5.2 : Measuring period of pulse in capture mode

Ex 11.5.1 SPPU - May 16, 9 Marks

A pulse is given to the CCP1 pin on its rising edge. Write a program that measures the period of the pulse and sends it to PORT B and PORT D.

Soln. : C18 program :

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0x05;     // Initialize CCP1CON
                        // register for capture mode.
    T3CON = 0x 00       // Initialize T3CON for
                        // Timer 1 operation
    T1CON = 0x00;       // Timer 1 , 16 bit,
                        // interual clock , 1:1prescaler
    CCPR1L = 00;        // Load CCPR1L
    CCPR1H=00;          // LOAD CCPR1H
    TRISB=0;            // make PORT B an output port
    TRISD=0;            // make PORT D an output port
    TRISCbits.TRISC2 = 1;   //make CCP1 pin input
while(1)
    {
      TMR1H = 0;
      TMR1L=0;
      PIR1bits. CCP1IF = 0;   //clear CCP1IF flag
      while (PIR1bits. CCP1IF= = 0);
                              // wait for 1st rising edge
      T1CONbits.TMR1ON = 1;  //start timer 1
      PIR1bits. CCP1IF= = 0 ;  // Clear CCP1IF for next
                               rising edge
    while (PIR1bits. CCP1IF= = 0);
                              // wait for 2ndt rising edge
      T1CONbits.TMR1ON=0     // stop Timer 1
      PORTD=CCPR1H;
      PORTB=CCPR1L;
    }
}
```

Syllabus Topic : PWM Mode

11.6 PWM Mode

SPPU - Dec.-15

University Question

Q. Explain PWM generation with example.

(Dec. 2015, 8 Marks)

- Pulse width modulation is a feature of the CCP (Capture/Compare/pulse width modulation). This feature allows the programmer to generate pulses that are having variable widths.
- In the earlier sections we have seen how to generate PWM using timers. However, if we use the CCP module for PWM generation then programming becomes easier.
- Mostly PWM is used for DC motor control.
- Two important factors considered during PWM generation are :
  (i) Duty cycle of the pulse.
  (ii) Period of the pulse.
- The **duty cycle** of a pulse is the part for which the pulse is high for the given period of time. Generally, it is expressed in terms of percentage eg : if a pulse of 8 ms, is high for 4 ms then duty cycle = $\frac{4\text{ ms}}{8\text{ ms}} = 50\ \%$

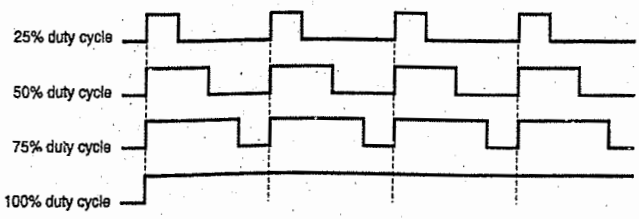


Fig. 11.6.1 : Period and duty cycle

11.6.1 Period of PWM

- The Timer 2 and register PR2 are used by the CCP module for PWM generation. The PR2 register is used for setting the period of PWM as,

$$T_{PWM} = [(PR2) + 1]\, 4 \times N \times T_{OSC}$$

Where T_{PWM} = required PWM period

N = Prescaler set by T2CON (Timer 2 control register)

i.e. 1, 4 or 16

$$T_{osc} = \frac{1}{F_{OSC}} \quad \therefore \quad PR2 = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1$$

If PR2 = 255 and N = 16, then we get maximum value of T_{PWM}

i.e. $T_{PWM} = [(255 + 1)] \times 4 \times 16 \times T_{OSC}$

$$T_{PWM} = 16382\, T_{OSC} \quad \text{or} \quad F_{PWM} = \frac{F_{OSC}}{16382}$$

Ex. 11.6.1

Find the PR2 value for following frequencies assuming XTAL = 10 MHz and prescaler = 4.

a) 10 KHz b) 20 KHz

Soln. :

(a) $$\text{PR2 value} = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1$$

$$\text{PR2 value} = \left[\frac{10 \times 10^6}{10 \times 10^3 \times 4 \times 4}\right] - 1 = 62 - 1$$

PR2 value = 61

(b) $$\text{PR2 value} = \left[\frac{10 \times 10^6}{20 \times 10^3 \times 4 \times 4}\right] - 1$$

PR2 value = 31 – 1

PR2 value = 30

11.6.2 Duty Cycle of PWM

- The duty cycle of a pulse is the part for which the pulse is high for the given period of time. A 10 bit register comprising of 8 bits from CCPR1L register and 2 bits from CCP1CON register is used for setting the duty cycle.
- Of the 10 bits, the upper 8 bits are from the CCPR1L register and lower 2 bits are DC1B2 : DC1B1 of CCP1CON register. The lower two bits are for the decimal part of duty cycle as shown in Table 11.6.1.

Table 11.6.1

DC1B2	DC1B1	Decimal point
0	0	0
0	1	0.25
1	0	0.5
1	1	0.75

Example :

1. If PR2 = 100 and we need a duty cycle of 20% then, CCPR1L = 20% × 100 = 20
   here DC1B2 : DC1B1 = 00
2. If PR2 = 50 and we need a duty cycle of 25% then, CCPR1L = 25% × 50 = 12.5
   So, CCPR1L = 12 and DC1B2 : DC1B1 = 10 representing 0.5, decimal part.

Ex. 11.6.2

Find the values of registers PR2, CCPR1L and DC1B2 : DC1B1 for the following PWM frequencies if we need a 50% duty cycle. Let XTAL = 10 MHz.

(a) 1 KHz (b) 2.5 KHz

Soln. :

(a) For 1 KHz

$$PR2 = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1, \quad \text{Let } N = 16$$

$$\therefore \quad PR2 = \left[\frac{10 \times 10^6}{1 \times 10^3 \times 4 \times 16}\right] - 1 = 155$$

For duty cycle of 50%

155 × 50% = 77.5

∴ CCPR1L 77 and DC1B2 : DC1B1 = 10 for 0.5 part.

(b) For 2.5 KHz

$$PR2 = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1$$

$$PR2 = \left[\frac{10 \times 10^6}{2.5 \times 10^3 \times 4 \times 16}\right] - 1 = 62$$

For duty cycle of 50%.

$$\frac{62 \times 50}{100} = 31$$

∴ CCPR1L = 31 and DC1B2 : DC1B1 = 00

11.6.3 Steps for Programming the CCP Module for PWM Generation

Step I : Load the PR2 register for setting the PWM period.

Step II : Load the CCPR1L register with higher 8 bits for setting the duty cycle.

Step III : Set the CCP1pin i.e. RC2 pin output.

Step IV : Set the timer 2 prescalar value using the T2CON register.

Step V : Clear the TMR2 register.

Step VI : Set the CCP1CON register for PWM, DC1B2 : DC1B1 for decimal part of the duty cycle.

Step VII : Start timer 2.

Ex. 11.6.3 **SPPU - Dec. 15, 4 Marks**

Write a program to create a 1 KHz PWM frequency with a 50% duty cycle on CCP1 pin. Assume XTAL = 10 MHz. Let N = 16.

Soln. : As seen in Ex. 11.6.2 the values of PR2, CCPR1L and DC1B2 : DC1B1 for 1 KHz frequency are

PR2 = 155, CCPR1L = 77 and DC1B2 : DC1B1 = 10.

C18 program :

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0;          //clear CCP1CON register.
    PR2 = 155;            //PR2 = (10 × 10^6)/(1 × 10^3 × 4 × 16) − 1 = 155
    CCPR1L = 77;                  //50% duty cycle
    TRISCbits.TRISC2 = 0;         //make CCP1 pin output
    T2CON = 0x03;   //timer 2, 16 prescale, no postscaler
    CCP1CON = 0x2C;     //PWM mode,
                        // DC1B2 : DC1B1 = 10 for 0.5
    TMR2 = 0;                     //clear timer 2
    T2CONbits.TMR2ON = 1;         //start timer 2
    while(1)
    {
      PIR1bits. TMR2IF = 0;       //clear timer 2 flag
      while (PIR1bits. TMR2IF= = 0);
                                  //wait till end of period.
    }
}
```

Ex. 11.6.4 **SPPU - May 2015, May 2016, 8 Marks**

Write a program for 1 KHz 10% duty cycle PWM waveform.

Soln. : For 1 KHz

$$PR2 = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1, \quad \text{Let } N = 16$$

$$\therefore \quad PR2 = \left[\frac{10 \times 10^6}{1 \times 10^3 \times 4 \times 16}\right] - 1 = 155$$

For duty cycle of 10%

$155 \times 10\% = 15.5 \quad \therefore CCPR1L = 15$

and DC1B2 : DC1B1 = 10 for 0.5 part.

PR2 = 155, CCPR1L = 15 and

DC1B2 : DC1B1 = 10.

```
#include < P18F458.h>
void main(void)
{
    CCP1CON = 0;          //clear CCP1CON register.
    PR2 = 155;            //PR2 = (10 × 10^6)/(1 × 10^3 × 4 × 16) − 1 = 155
    CCPR1L = 15;                  //10% duty cycle
    TRISCbits.TRISC2 = 0;         //make CCP1 pin output
    T2CON = 0x03;         //timer 2, 16 prescale, no
                          // postscaler
    CCP1CON = 0x2C;       //PWM mode, DC1B2 :
                          // DC1B1 = 10 for 0.5
    TMR2 = 0;                     //clear timer 2
    T2CONbits.TMR2ON = 1;         //start timer 2
    while(1)
    {
      PIR1bits. TMR2IF = 0;       //clear timer 2 flag
      while (PIR1bits. TMR2IF= = 0);
                                  //wait till end of period.
    }
}
```

Ex. 11.6.5

Write a program to create a 2.5 KHz PWM frequency with 50% duty cycle on CCP1 pin. Assume XTAL = 10 MHz. Let N = 16.

Soln. :

$$PR2 = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1$$

$$PR2 = \left[\frac{10 \times 10^6}{2.5 \times 10^3 \times 4 \times 16}\right] - 1 = 62$$

For 50% duty cycle

$$\frac{62 \times 50}{100} = 31$$

$\therefore$ CCPR1L = 31 and DC1B2 : DC1B1 = 00

C18 program :

```
#include <P18F458.h>
void main (void)
{
    CCP1CON = 0 ;         // Clear CCP1CON register
    PR2 = 62 ;    // PR2 = (10 × 10^6)/(2.5 × 10^3 × 4 × 16) − 1 = 62
    CCPR1L = 31 ;                 // 50% duty cycle
    TRISCbits.TRISC2 = 0 ;        // make CCP1 pin output
    T2CON = 0x03 ;  // Timer 2, 16 prescaler, no postscaler
    CCP1CON = 0x0C ;    // PWM mode,
                        //DC1B2 : // DC1B1 = 00
    TMR2 = 0 ;                    // Clear Timer 2
    T2CONbits.TMR2ON = 1 ;        // Start Timer 2
    while (1)
    {
      PIR1bits.TMR2IF = 0 ;       // Clear timer 2 flag
      while (PIR1bits.TMR2IF = = 0) ;
                                  // wait till end of period
    }
}
```

Ex. 11.6.6 **SPPU - Dec. 14, 8 Marks**

Create a 2 KHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL = 10 MHz.

Soln. :

For 2 KHz

$$PR2 = \left[\frac{F_{OSC}}{F_{PWM} \times 4 \times N}\right] - 1$$

Let N = 16

$$\therefore \quad PR2 = \left[\frac{10 \times 10^6}{2 \times 10^3 \times 4 \times 16}\right] - 1 = 78 - 1 = 77$$

For 25% duty cycle,

$$77 \times \frac{25}{100} = 19.25$$

$$\therefore \quad CCPR1L = 19 \text{ and } DC1B2 : DC1B1 = 01$$

C program :

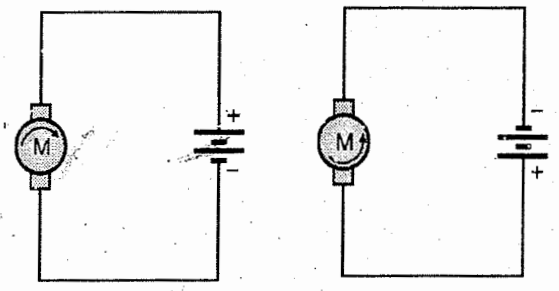
```
#include <P18F458.h>
void main (void)
{
    CCP1CON = 0 ;          // Clear CCP1CON register
    PR2 = 77 ;             // PR2 = (10 × 10^6)/(2 × 10^3 × 4 × 16) − 1 = 77
    CCPR1L = 19 ;                  // 25% duty cycle
    TRISCbits.TRISC2 = 0 ;         // make CCP1 pin output
    T2CON = 0x03 ; // Timer 2, 16 prescaler, no postscaler
    CCP1CON = 0x1C ;               // PWM mode, DC1B2 :
                                   // DC1B1 = 01
    TMR2 = 0 ;                     // Clear Timer 2
    T2CONbits.TMR2ON = 1 ;         // Start Timer 2
    while (1)
    {
      PIR1bits.TMR2IF = 0 ;        // Clear timer 2 flag
      while (PIR1bits.TMR2IF = = 0) ;
                                   // wait till end of period
    }
}
```

11.7 DC Motor Control

- The DC motor (direct current) motor rotates continuously. It is a device that translates electrical pulses into mechanical movement.
- It has two terminals i.e. positive and negative.
- By connecting DC power supply to these terminals the motor rotates in one direction. If the polarity of power supply is reversed the direction of rotation reverses.
- The speed of DC motor is measured in revolutions per minute (RPM).
- As the supply voltage increases, the speed of the DC motor increases. However we cannot exceed the supply voltage beyond rated voltage.
- The speed of DC motor depends on load. At no-load the speed of DC motor is highest. As the load is increased the speed decreases.
- Overloading of the DC motor can damage it because of excessive heat generated due to high current consumption.

11.7.1 Unidirectional Control

- Fig. 11.7.1 shows the DC motor rotation for clockwise and anticlockwise rotation. The direction of rotation reverses as the polarity of supply voltage polarity reverses.



Clockwise rotation **Anticlockwise rotation**

Fig. 11.7.1 : DC motor rotation

11.7.2 Bidirectional Control

By using switches for charging the power supply polarity we can control the direction of rotation of DC motor as shown in Fig. 11.7.2.

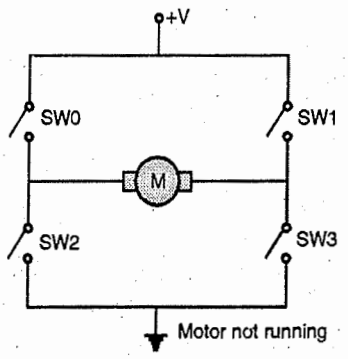


Fig. 11.7.2 : Bidirectional control of DC motor

Table 11.7.1 : Switch configurations

SW0	SW1	SW2	SW3	Motor operation
ON	ON	ON	ON	OFF
OFF	ON	ON	OFF	Clockwise
ON	OFF	OFF	ON	Counter clockwise
OFF	OFF	OFF	OFF	Invalid

11.7.3 Interfacing DC Motor using L293 H-Bridge

- Since the motor requires high current, it cannot be driven from the pins of PIC18 microcontroller directly. As already seen a power transistor is to be connected to drive the motor, but it cannot drive the motor in both the directions. Hence to drive motor in either directions we need an H-bridge.
- Fig. 11.7.3(a) shows the pin diagram while Fig. 11.7.3(b) shows the block diagram of L293.

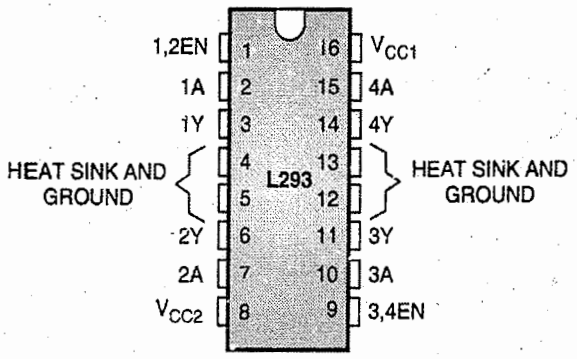


Fig. 11.7.3(a) : Pin Diagram of L293

- The L293 is a quadruple high-current half-H driver.
- The L293 is designed to provide bidirectional drive currents of up to 1 A at voltages from 4.5 V to 36 V.
- It is designed to drive inductive loads such as relays, solenoids, dc and bipolar stepping motors, as well as other high-current/high-voltage loads in positive-supply applications.
- All inputs are TTL compatible. Each output has a Darlington transistor sink and a pseudo-Darlington source.
- Drivers are enabled in pairs, with drivers 1 and 2 enabled by 1,2EN and drivers 3 and 4 enabled by 3,4EN. When an enable input is high, the associated drivers are enabled, and their outputs are active and in phase with their inputs.
- When the enable input is low, those drivers are disabled, and their outputs are off and in the high-impedance state.
- With the proper data inputs, each pair of drivers forms a full-H (or bridge) reversible drive suitable for solenoid or motor applications.

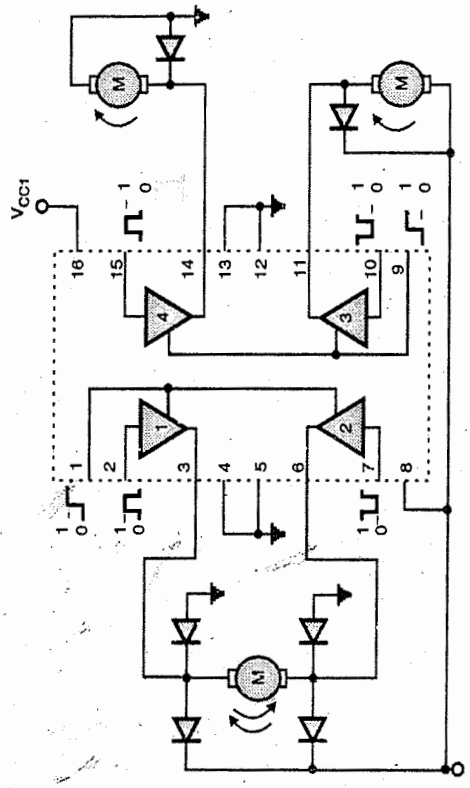


Fig. 11.7.3(b) : Block diagram

- The Table 11.7.2 shows how an output can be enabled

Table 11.7.2 : Function Table (each driver)

Inputs		Output Y
A	EN	
H	H	H
L	H	L
X	L	Z

H = high level, L = low level, X = irrelevant, Z = high impedance (off)

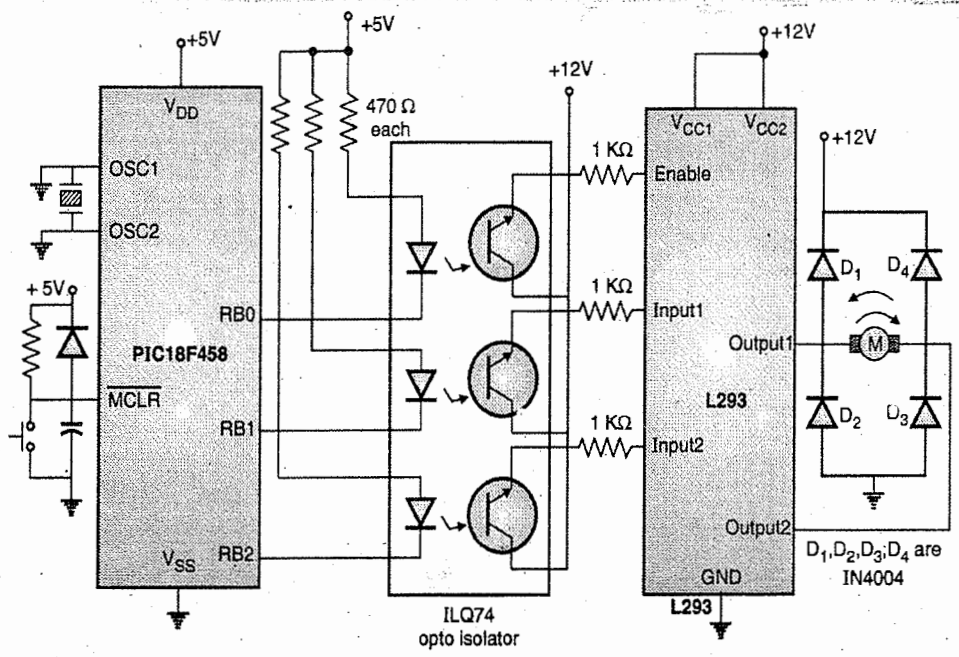


Fig. 11.7.4 : Interface DC motor with L293 and PIC18F458 control the direction of motor

Ex. 11.7.1

A switch is connected to pin RD3. Write a program to monitor the status of switch and do the following :

(a) if switch = 0, DC motor moves clockwise

(b) If switch = 1, DC motor moves anticlockwise

Soln. :

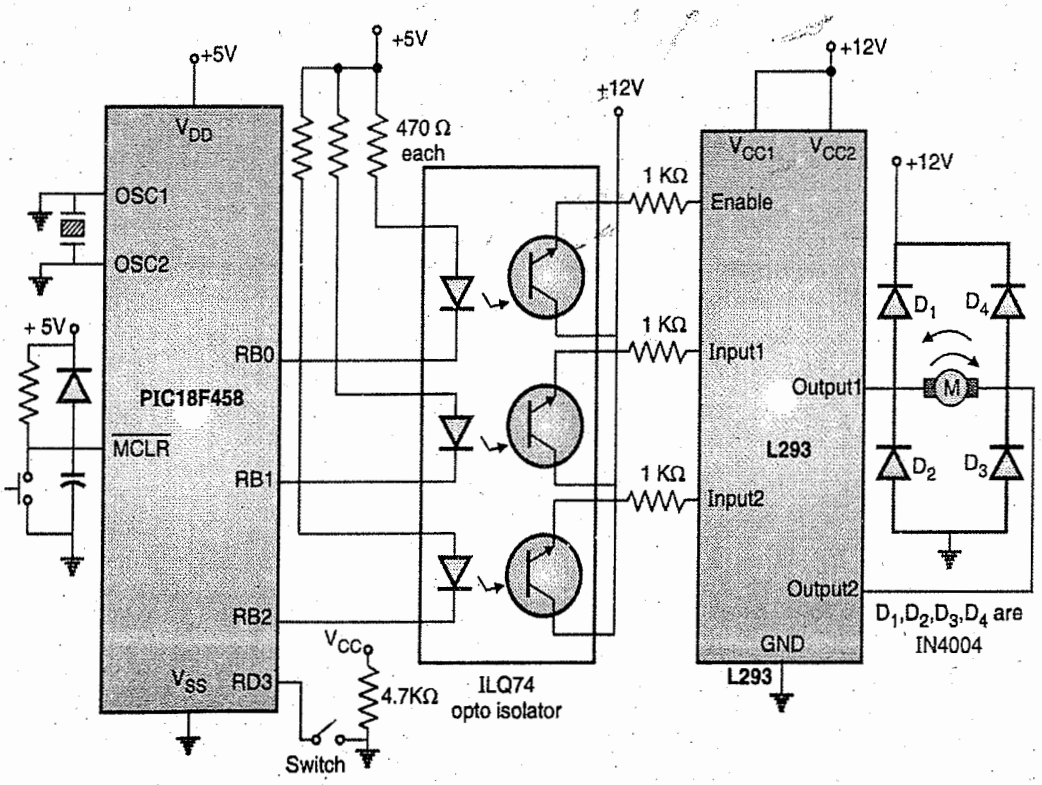


Fig. P. 11.7.1 : Interfacing DC motor with L293 and PIC18F458 to control the direction of motor

C program :

```
# include <P18F458.h>
# define SW PORTDbits.RD3
# define ENABLE PORTBbits.RB0
# define M_1 PORTBbits.RB1
#define M_2 PORTBbits.RB2
void main ( )
{
    TRISD = 0x04 ;              // Make RD3 input pin
    TRISB = 0x00 ;              // Make PORTB output
    SW = 1 ;
    ENABLE = 0;
    M_1 = 0 ;
    M_2 = 0 ;
    while (1)
    {
      ENABLE = 1 ;
      if (SW = = 1)
      {
        M_1 = 1 ;          // Rotate motor anticlockwise
        M_2 = 0 ;
      }
    else
      {
        M_1 = 0 ;          // Rotate motor clockwise
        M_2 = 1 ;
      }
    }
}
```

11.7.4 Pulse Width Modulation

SPPU - Dec. 16

University Question

Q. How the speed of the DC motor is controlled by PWM, explain in brief ? **(Dec. 2016, 6 Marks)**

- The DC motor speed is dependent on three factors
  (i) voltage (ii) current and (iii) load.
- For a fixed load steady speed can be maintained by using a technique called **pulse width modulation (PWM).**
- By changing (modulating) the width of the applied pulse to the DC motor we can increase or decrease the amount of power provided to the motor, thereby increasing or decreasing the motor speed.
- Although the voltage has a fixed amplitude, its duty cycle varies. The wider the pulse greater is the motor speed.
- Hence PWM is widely used in DC motor control.
- The ability to control the speed of DC motor using PWM is one of the reasons why DC motors are preferable over AC motors.
- **AC motor speed** is dictated by AC frequency of voltage applied to motor and frequency is fixed. Hence, we cannot control the speed of AC motors when load is increased.
- Fig. 11.7.5 shows pulse width modulation comparisons.

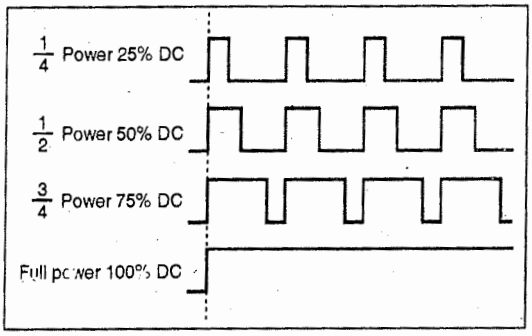


Fig. 11.7.5 : Pulse width modulation comparison

11.7.5 DC Motor Control using Opto-Isolator

SPPU - Dec. 15

University Question

Q. Design a DC Motor control using PWM. **(Dec. 2015, 8 Marks)**

Fig. 11.7.6 shows dc motor control using bipolar transistor and MOSFET as a switch. It uses opto-isolator that allows the motor and PIC18F458 to use separate power supplies. The circuits shown are protected from EMI interference produced because of motor brushes. A decoupling capacitor is connected across the motor to protect circuit from EMI interference.

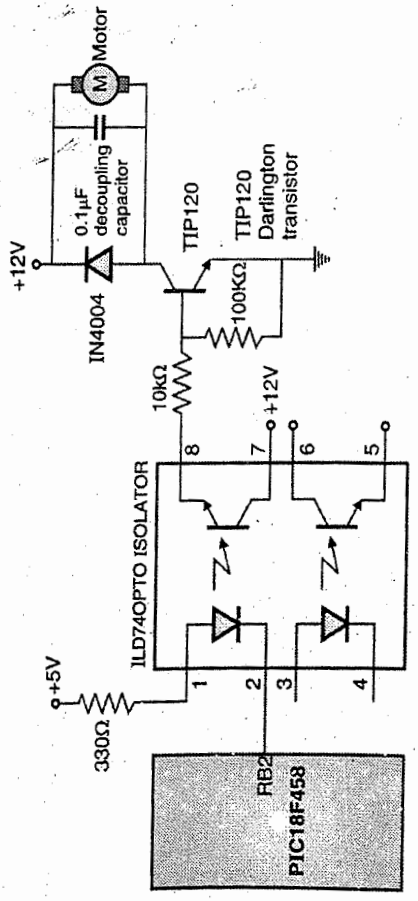


Fig. 11.7.6 : DC motor connection using bipolar transistor

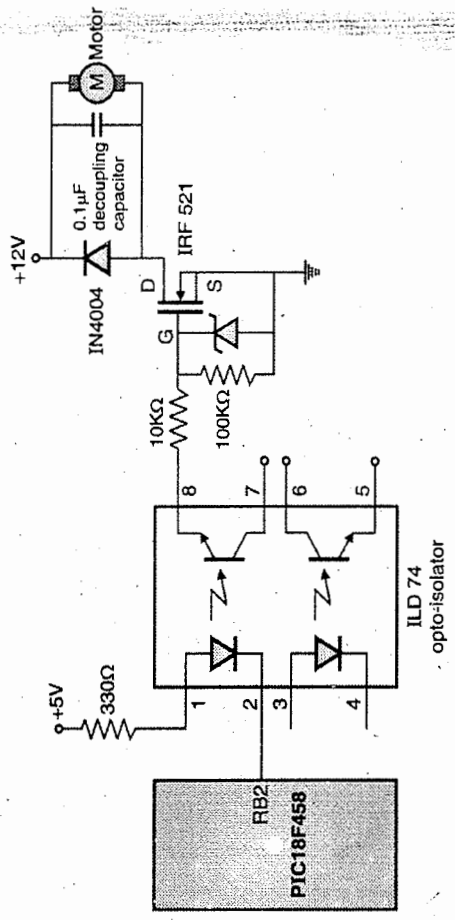


Fig. 11.7.7 : DC motor connection using MOSFET

- Port RB2 controls the switching of DC motor. When RB2 is low, motor is switched ON and when RB2 is high motor is switched off.
- In MOSFET circuit, the zener diode keeps the gate voltage below rate maximum gate voltage for MOSFET.

Ex. 11.7.2 **SPPU - May 15, Dec. 15, May 16, 10 Marks, Lab assignment**

Draw an interfacing diagram and write an algorithm for DC Motor speed controller using PIC18xxx.

OR

Design a PIC based system to interface DC motor to monitor the status of switch connected to pin RC2 and do the following :

(a) If SW = 0, the DC motor moves with 50% duty cycle pulse.

(b) If SW = 1, the DC motor moves with 25% duty cycle pulse

Soln. :

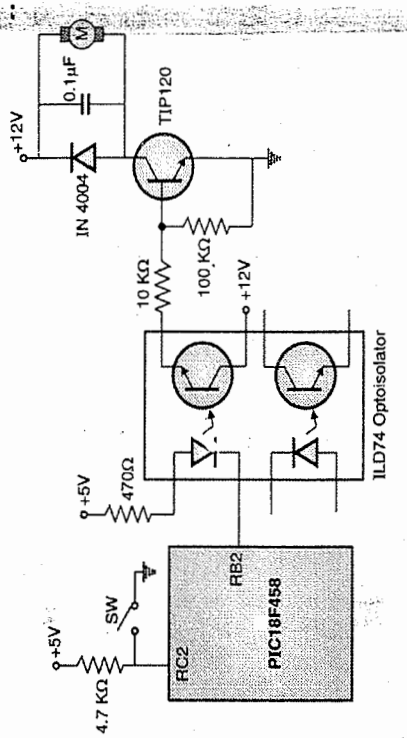


Fig. P. 11.7.2 : Interfacing DC motor to PIC18F 458 using PWM

Algorithm :

Step I : Initialize RB2 (Port B port pin 2) as output.

Step II : Initialize RC2 (Port C pin 2) as input

Step III : Switch off DC motor

Step IV : Check if SW= 1 (i.e. check RC2) if yes move the DC motor with 25% duty cycle. Otherwise goto step VII.

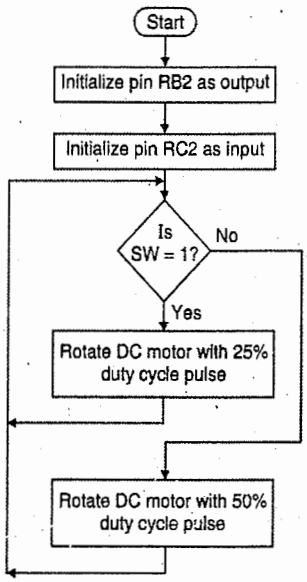
Step V : Wait for sometime till motor rotates with 25% duty cycle.

Step VI : Go to Step IV

Step VII : Rotate DC motor with 50% duty cycle i.e. RB2 is high for two time periods and low for two time periods.

Step VIII : Go to step IV

Flowchart :



Flowchart 1

C18 program :

```
#include <P18F458.h>
#define SW PORTCbits.RC2
#define MOTOR PORTBbits.RB2
    void delay (unsigned int value) ;
    void main (void)
    {
      TRISCbits. TRISC2 = 1 ;    //Make RC2 an input
      TRISBbits. TRISB2 = 0 ;    // Make RB2 an output
    while (1)
    {
      if (SW = = 1)
    {
      MOTOR = 1 ;  // Rotate motor with 25% duty cycle
      Delay (25) ;
      MOTOR = 0 ;
      Delay (75) ;
    }
    else
    {
      MOTOR  = 1 ;  //Rotate motor with 50% duty cycle
      Delay (50) ;
      MOTOR = 0 ;
      Delay (50) ;
      }
    }
}
    void delay (unsigned int value)
    {
      unsigned char i, j ;
      for (i = 0 ; i < 1275 ; i + +)
      for (j = 0 ; j < value ; j + +) ;
    }
```

Ex. 11.7.3

Design a PIC based system to interface DC motor to monitor two switches that are connected to pins RC1 and RC2. Write a C program to monitor the status of both the switches and do the following :

SW2 (RC1)	SW1 (RC0)	
0	0	DC motor moves slowly (25% duty cycle)
0	1	DC motor moves moderately (50% duty cycle)
1	0	DC motor moves fast (75% duty cycle)
1	1	DC motor moves very fast (100% duty cycle)

Soln. : Fig. P. 11.7.2 shows the interfacing of DC motor to PIC18F458 using PWM.

C18 program :

```
#include <P18F458.h>
#define SW2 PORTCbits.RC1
#define SW1 PORTCbits.RC0
#define MOTOR PORTBbits.RB2
    void delay (unsigned int value) ;
    void main (void)
    {
      unsigned char z ;
      TRISCbits.TRISC1 = 1 ;     // RC1 = input
      TRISCbits.TRISC0 = 1 ;     // RC0 =  input
      TRISBbits.TRISB2 = 0 ;     // RB2 = output
    while (1)
    {
      z= PORTC ;                 //Read RC1 and RC0
      z = z & 0x03 ;             // Mask the unused bits
      switch (z)                 // make decision
      {
case (0) :
    {
      MOTOR = 1 ;  //Rotate motor with 25% duty cycle
      Delay (25) ;
      MOTOR= 0 ;
      Delay (75) ;
      break ;
    }
case (1) :
    {
      MOTOR = 1 ; // DC motor moves with 50% duty cycle
      Delay (50) ;
      MOTOR = 0 ;
      Delay (50) ;
      break ;
    }
case (2) :
      {
        MOTOR = 1 ;              // DC motor moves with
                                 //  75% duty cycle
        Delay (75) ;
        MOTOR = 0 ;
        Delay (25) ;
        break ;
      }
case (3) :
    {
      MOTOR = 1 ;                // DC motor moves with
                                 //100% duty cycle
      Delay (100) ;
      break ;
        }
      }
    }
}
    void delay (unsigned int value)
    {
      unsigned char i, j ;
      for (i = 0, i < 1275, i + +)
      for (j = 0 ; j < value ; j + +) ;
    }
```

Syllabus Topic : DC Motor Control with CCP

11.7.6 DC Motor Control with CCP

- The PWM feature is used for controlling the DC motors.
- We program the PR2, TMR2 to get PWM.
- Fig. 11.7.8 shows the DC motor control with CCP1 pin.
- Now we will write a program that monitors the switch. If switch is low, PIC18 creates a duty cycle of 50% PWM and if the switch is high a 75% duty cycle PWM is created.

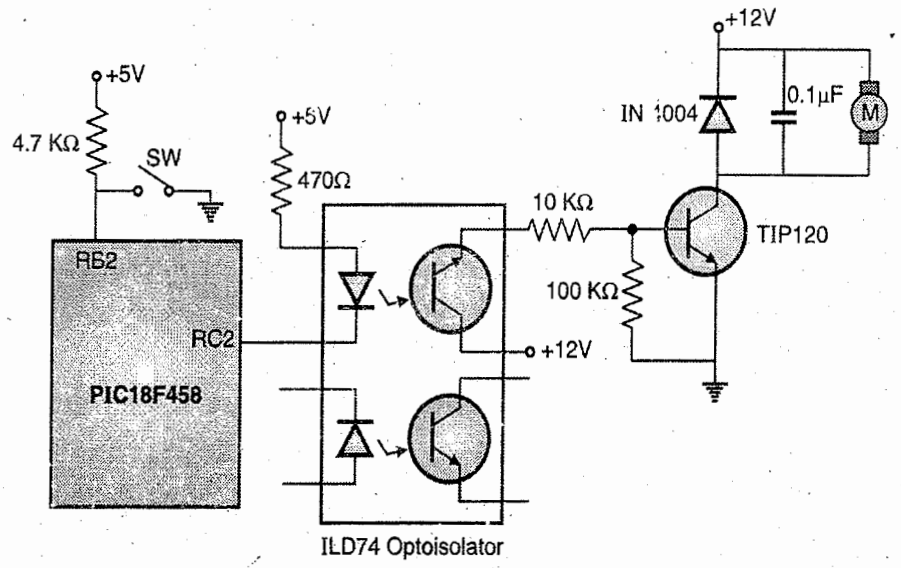


Fig. 11.7.8 : Interfacing DC motor to PIC18F458 using PWM

C18 program :

```
#include <P18F458.h>
#define switch PORTBbits.RB2
void main ( )
    {
     TRISCbits.TRISC2 = 0 ;                    //RC2 = output
     TRISBbits.TRISB2 = 1 ;                    //RB2 = input
     CCP1CON = 0x0C ;                          // PWM mode
     PR2 = 100 ;                               // Timer 2, 16 prescaler, no postscaler
     T2CON = 0x03 ;
    while (1)
     {
        if (switch = = 0)
        CCPR1L = 50 ;                          // 50% duty cycle
    else
     CCPR1L = 75 ;                             // 75% duty cycle
     TMR2 = 0 ;                                // Clear Timer 2
     PIR1bits.TMR2IF = 0 ;                     // Clear Timer 2 interrupt flag
     T2CONbits.TMR2ON = 1 ;                    // Start Timer 2
     while (PIR1bits.TMR2IF = = 0) ;           // wait for end of period
     }
    }
```

□□□

UNIT - VI

CHAPTER 12

MSSP Structure, UART and Interfacing Serial Port

12.1 Basics of Serial Communication Protocol

- The word, *communication* specifies, data transfer between two points.
- The data may be a digital or analog in nature.
- We will consider only digital data transfer because microprocessor is digital circuit. Suppose you want to transfer data from Point A to Point B. There are two possible ways of doing it :
  (1) Parallel data transfer
  (2) Serial data transfer.
- In parallel data transfer 8 data lines are needed to transfer a byte of data. The data transferred is 8 bits at a time.
- In serial data transfer one bit is transferred at a time over a single line.
- For transmitting data over long distances using parallel communication is impractical because of the increase in cost of cabling. Parallel data transfer cannot be used for devices like cassette tapes, CRT terminal etc. In such cases serial communication is used.

Now let's compare the specified 2 methods of data transfer.

Sr. No.	Parallel data transfer	Serial data transfer
1.	8 bits of data is transferred at a time.	One bit of data is transferred at a time.
2.	9 lines required to be connected between 2 points.	Only 2 lines required to be connected.
3.	Data transfer is fast.	Data transfer is slow.
4.	More advantageous over small distances only.	More advantageous over long distances.

12.1.1 Types of Communication Systems

The communication systems are classified on the basis of transmission :

1. Simplex 2. Duplex

1. Simplex

- The simplex is one way transmission.
- The connection exists such that data transfer takes place only in one direction.
- There is no possibility of data transfer in the other direction.
- System A is transmitter and system B is receiver only.

2. Duplex

The duplex is two way transmission. It is further divided in 2 groups :

(a) Half duplex (b) Full duplex.

(a) Half duplex

- It is a connection between two terminals such that, data may travel in both the directions, but transmission activated in one direction at a time.
- This indicates that the line has to turn around after communication is complete in one direction.

(b) Full duplex

It is a connection between two terminals such that, data may travel in both the directions simultaneously. So it will contain one way transmission or two way transmission at a time. Fig. 12.1.1 shows the simplex, half and full duplex data transfers.

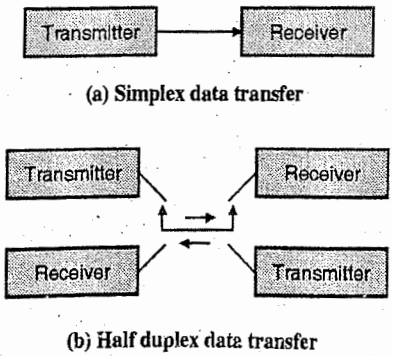
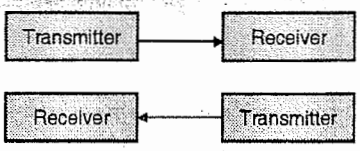


Fig. 12.1.1 : Simplex, half and full-duplex data transfers



(c) Full duplex data transfer

Fig. 12.1.1 : Simplex, half and full-duplex data transfers

12.2 Serial Transmission Formats

In serial communication, two formats of data transfer are used. These two formats are :

1. Asynchronous 2. Synchronous

12.2.1 Asynchronous Data Transfer

- It is a *'character'* oriented data transfer. In this one data byte is transferred serially at a time. When data is not transferred output line stays HIGH.
- The start of data is indicated by a low start bit. The start bit is used to synchronise transmitter and receiver. After the start bit, data bits are transferred serially D_0, D_1 ... D_7 (least significant bit first).
- The data bits are followed by a one or more high stop bits. After the stop bits same logical level is maintained on output line i.e. marking state for next data byte.
- At the receiver end the receiver will check for marking state. When it detects a low on data line for 1 bit time it is treated as valid start bit.
- After start bit it will start accepting data bits D_0 to D_7. At the end it will check stop bits and go back to initial marking state. This process is repeated for next data byte.
- The asynchronous data format consists of a start bit, data bits and stop bits (also called as frame) is as shown in Fig. 12.2.1.
- The clock to both transmitter and receiver are separate. But the operating frequency for both is maintained same.
- The bit period (baud rate) is pre-decided to maintain synchronization between the transmitter and receiver.

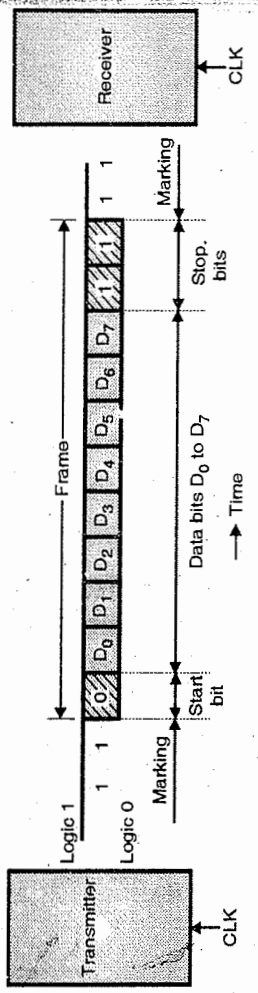


Fig. 12.2.1 : Asynchronous data format

The Table 12.2.1 gives the list of asynchronous data format standards.

Table 12.2.1

Specification	Typical values
Data bits/character	5, 6, 7 or 8
Stop bits	1, 1½ or 2
Parity bit	Odd or even parity
Baud rates	75, 100, 150, 300, 600, 1200, 2400, 4800, 9600, 19200.

12.2.2 Synchronous Data Transfer

- It is a ***'block or packet of data'*** oriented data transfer. The block(s) of data bytes are transferred serially.
- Packet of data is of size *'n'*. The data bytes in a block are transferred one after the other. As shown in Fig. 12.2.2 D_0 to D_7 data bits of $data_1$, $data_2$, ... $data_n$; is transferred serially.

- Before sending data, special characters are transferred by transmitter to achieve synchronization between transmitter and receiver.

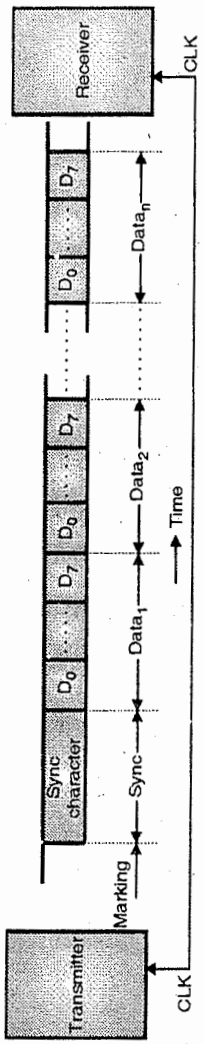


Fig. 12.2.2 : Synchronous data format

- This special character is called as ***Sync character***.
- Normally the output line of transmitter is HIGH i.e. marking state.
- To start transmission, the sync character bits are sent by transmitter followed by data bits.
- At receiver end, the receiver will go on checking input line.
- The checking operation is comparing ***previous 8 bits*** of data received with receivers sync character. When match occurs i.e. both the bit patterns are exactly equal then it is considered that the receiver is in synchronization with transmitter.
- When the receiver is synchronized, it will go on accepting data bits in same sequence i.e. D_0 to D_7 and so on.
- The term, ***previous 8 bits*** is used for sync character.
- It specifies previously accepted 7 bits and present bit will be 8th bit.
- The sync character may be any digital bit pattern which is used for transmitter and receiver. Refer Fig. 12.2.2.
- The clock to both transmitter and receiver is same. If there is little difference between clock synchronization the synchronous method will accept wrong data bits so the same clock is used.

12.2.3 Comparison of Asynchronous and Synchronous Format

Sr. No.	Asynchronous data transfer	Synchronous data transfer
1.	It is used to transfer one character at a time.	It is used to transfer a block of characters at a time.
2.	Used for data transfer rates ≤ 20 k bits / second.	Used for high data transfer rates ≥ 20 k bits / second.
3.	Sync characters are not transmitted along with characters.	Sync characters are transmitted along with the group of characters.
4.	Start bit and stop bit for each character is present which forms a frame.	No start and stop bit is used.
5.	Two separate clock inputs can be used for transmitter and receiver.	One clock is used for both transmitter and receiver.
6.	No synchronization is required hence hardware and software implementation is possible.	Since synchronization is involved, this can be implemented by using hardware only.
7.	Speed is less, because overheads are more per byte (start bit, stop bit(s) etc.)	Speed is high because overheads are spread over a frame or packet of data bytes.

12.2.4 Baud Rate

In serial communication the rate at which data bits are transmitted generates a term **Baud rate.** The Baud rate is defined as bits / second or the changes in voltage levels / second.

$$\text{Baud} = \frac{\text{Bits transmitted}}{\text{Second}}$$

Typical Baud rates are 110, 300, 600, 1200, 2400, 4800 and 9600.

A teletypewriter typically uses 110 Baud.

Transmission rate = 110 Baud = 110 bits / sec.

$$\text{Time for one bit} = \frac{1}{110} = 9.1 \text{ ms.}$$

Syllabus Topic : Study of RS 232

12.3 Study of RS 232

- A standards working committee that is today called as the Electronic Industries Association (EIA) in 1990, for data communication equipment introduced a common interface standard.
- During that phase, the data communications was referred to be a method of exchanging the digital data between a mainframe computer and a remote computer terminal, or without a computer to exchange the data between two remote terminals.
- These devices where the digital data was to be transferred were connected through the telephone voice lines. These telephone lines needed a modem at the transmitter and the receiver end for the signal translation.
- This system was simple to implement. However many data were observed while the data was transmitted through the analog channel. This made the system design complex.
- Then it was recommended that there must be a standard that will ensure reliable communication between the nodes. Also the standard must be compatible of making connections with the equipments that are produced by different manufacturers. The above two parameters will support mass production of the standard in competition to the different market standards. With all these ideas into mind, the RS232 standard was developed. The RS232 standard specifies the signal voltages, signal timing, signal function and mechanical connectors. RS232 is communication protocol that is used for exchanging data between two devices e.g. two computers, a mobile and a computer etc.
- In 1963 the RS232 standard was modified to RS232A. In 1965 and 1969 the standard was modified to RS232B and RS232C.
- Today, it is called as RS232 and is the most widely used standard for serial I/O interfacing.
- However, the input and output voltage levels of the RS232 standard are not TTL compatible. This is because the standard was set before the TTL logic family was introduced.
- In RS232 standard a bit 0 is representated by +3 to +25V, while a bit 1 is representated by – 3 to – 25V.
- Hence, for connecting any RS232 to a microcontroller we **need voltage converters like MAX232** to convert the TTL logic levels to RS232 voltage levels and vice-versa in serial communication.
- MAX232 is a line driver (voltage converter) that converts RS232 voltage levels to TTL voltage levels and vice-versa.
- This interface is useful for point-to-point communication at slow speeds.
- Ex. COM1 port in a PC can be used for a mouse, COM2 port can be used for a modem.

RS-232 serial communications

- The EIA RS-232C serial communication standard is a universal standard, originally used to connect teletype terminals to modem devices.
- Fig. 12.3.1 shows a PC connected to a device such as a modem or a serial printer using the RS-232C connection. In a modern PC the RS-232C interface is referred to as a COM port.
- The COM port uses a 9-pin D-type connector to attach to the RS-232 cable. The RS-232 standard defines a 25-pin D-type connector but in newer systems the connector is reduced to a 9-pin device so as to reduce the size.
- Fig. 12.3.2 also shows a simple three wire connection to support full-duplex serial communication.
- The RS-232C (COM port) standard includes additional signal wires for "hand-shake" purposes, but the fundamental serial communication can be achieved with just three wires as shown.

- The serial data is transmitted at a predefined rate, referred to as the baud rate. The term baud rate refers to the number of state changes per second which is the same as the bit rate for this particular communication scheme.
- Typical baud rates are: 150, 300, 600, 1200, 1800, 2400, 3600, 4800, 7200, 9600 and 19,200 bps or baud.

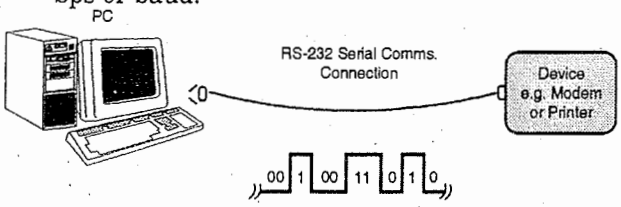


Fig. 12.3.1

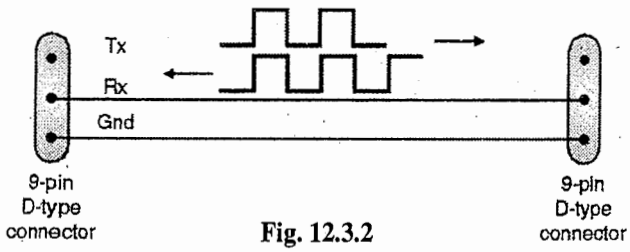


Fig. 12.3.2

12.3.1 Signals used in RS232

Table 12.3.1 : Pin Description of RS-232 DB-9 Connector

Pin	Description
1	Data carrier detect ($\overline{DCD}$)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready ($\overline{DTR}$)
5	Signal ground (GND)
6	Data set ready ($\overline{DSR}$)
7	Request to send ($\overline{RTS}$)
8	Clear to send ($\overline{CTS}$)
9	Ring indicator (RI)

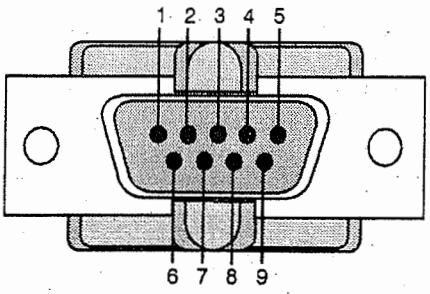
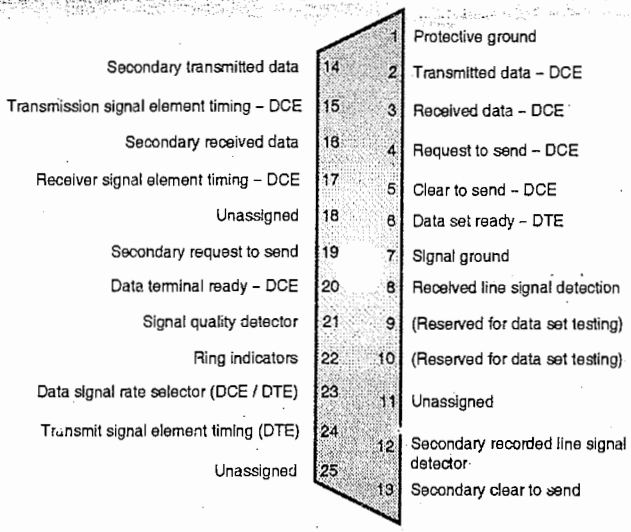


Fig. 12.3.3 : RS 232 DB – 9 connector



m(14.1)**Fig. 12.3.4 : Pins on RS 232 C interface DB-25 connector**

Table 12.3.2 : Pin description of DB-25 connector

Pin Number	Common Name	RS 232 C Name	Description	Signal Direction on DCE
1		AA	Protective ground	
2	TxD	BA	Transmitted data	IN
3	RxD	BB	Received data	OUT
4	$\overline{RTS}$	CA	Request to send	IN
5	$\overline{CTS}$	CB	Clear to send	OUT
6	$\overline{DSR}$	CC	Data set ready	OUT
7	GND	AB	Signal ground (common return)	
8	$\overline{CD}$	CF	Received line signal detector	OUT
9			(Reserved for data set testing)	
10			(Reserved for data set testing)	
11			Unassigned	
12		SCF	Secondary recorded line signal detector	OUT
13		SCB	Secondary clear to send	OUT

Pin Number	Common Name	RS 232 C Name	Description	Signal Direction on DCE
14		SBA	Secondary transmitted data	IN
15		DB	Transmission signal element timing (DCE source)	OUT
16		SBB	Secondary received data	OUT
17		DD	Receiver signal element timing (DCE source)	OUT
18			Unassigned	
19		SCA	Secondary request to send	IN
20	$\overline{DTR}$	CD	Data terminal ready	IN
21		CG	Signal quality detector	OUT
22		CE	Ring indicator	OUT
23		CH/CI	Data signal rate selector (DTE/DCE source)	IN/OUT
24		DA	Transmit signal element timing (DTE source)	IN
25			Unassigned	

- It is a bus standard used for transmitting and receiving Serial Data.
- The format specifies handshake as well as communication signals, and gives hardware specifications.

12.3.2 Hardware Specifications or Features

1. Voltage levels +3 V ... +25 V for logic 0 and –3 V ... –25 V for logic 1.
2. Transmission line should be a Twisted pair wire with capacitance between 300 ... 2500 pF.
3. Max length of the line = 50 ft at maximum baud rate of 20 K. For lower baud rates the distance can be increased from 2000 to 3000 feet.
4. The circuit should be single end, bi-polar, voltage un-terminated.
5. Maximum common mode output voltage ± 25 V.
6. Driver slew rate is 30 V/μs.

12.3.3 Signal Specifications

The Data Terminal Equipment (DTE) wants to send data through the Data Communication Equipment (DCE), to a Remote MODEM. The signals used for this purpose are as follows :

(1) $\overline{DTR}$ (Data Terminal Ready)

- The DTE alerts the DCE for data communication by sending this signal.
- In response to this signal, the DCE prepares itself for the communication.

(2) $\overline{DSR}$ (Data Set Ready)

- The DCE sends this signal to the DTE to inform the DTE that it is ready for communication.

(3) $\overline{RTS}$ (Request To Send)

- Once DTE receives the DSR, it prepares itself for the data transfer.
- It initializes the Memory pointer and the Byte Counters.
- It then sends the $\overline{RTS}$ signal to the DCE, to inform that it is ready for the transfer.

(4) $\overline{DCD}$ (Data Carrier Detect)

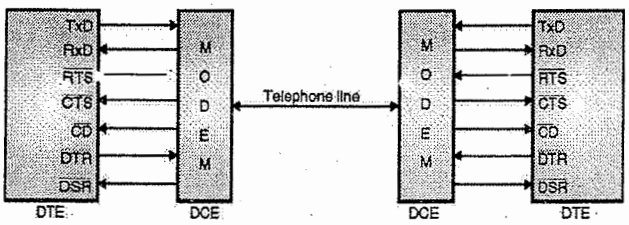
- In response to the $\overline{RTS}$, DCE tries to make contact with the Remote MODEM.
- If the Carrier is detected it informs the DTE about it through the $\overline{DCD}$ signal.

(5) $\overline{CTS}$ (Clear To Send)

- Once a stable contact is established with the Remote MODEM, the DCE sends the $\overline{CTS}$ to DTE.
- Due to this, the transmission begins.

12.3.4 Data Transmission and Reception using RS232C

Modems and other devices used to send serial data, are called as DCE i.e. Data Communication Equipments. The computers that transfer or receive serial data are called as DTE i.e. Data Terminal Equipments. The signals and handshake signals are transferred between DTE and DCE.



m(8.5)**Fig. 12.3.5 : DTE and DCE generalized connection diagram**

Operation

(i) When DTE is turned ON, after performing self checking, it sends $\overline{DTR}$ (Data terminal ready) to the $\overline{MODEM}$.

(ii) Upon receiving $\overline{DTR}$, MODEM responds by giving $\overline{DSR}$ (Data set ready) to indicate it is active for taking part in data transfer.

(iii) The MODEM then sends signal to other end and upon receiving yes from other side, makes $\overline{CD}$ (Carrier Detect low).

(iv) The DTE then sends a signal $\overline{RTS}$ (Request To Send) signal to the MODEM.

(v) The MODEM when ready to transfer data sends signal $\overline{CTS}$ (Clear To Send.)

The DTE then transmits data on TxD line. Similar handshake signals are exchanged between DTE and MODEM for receiving data on RxD line.

Fig. 12.3.6 shows the flowchart for handshaking process when an RS232C data port is used to exchange data with the modem.

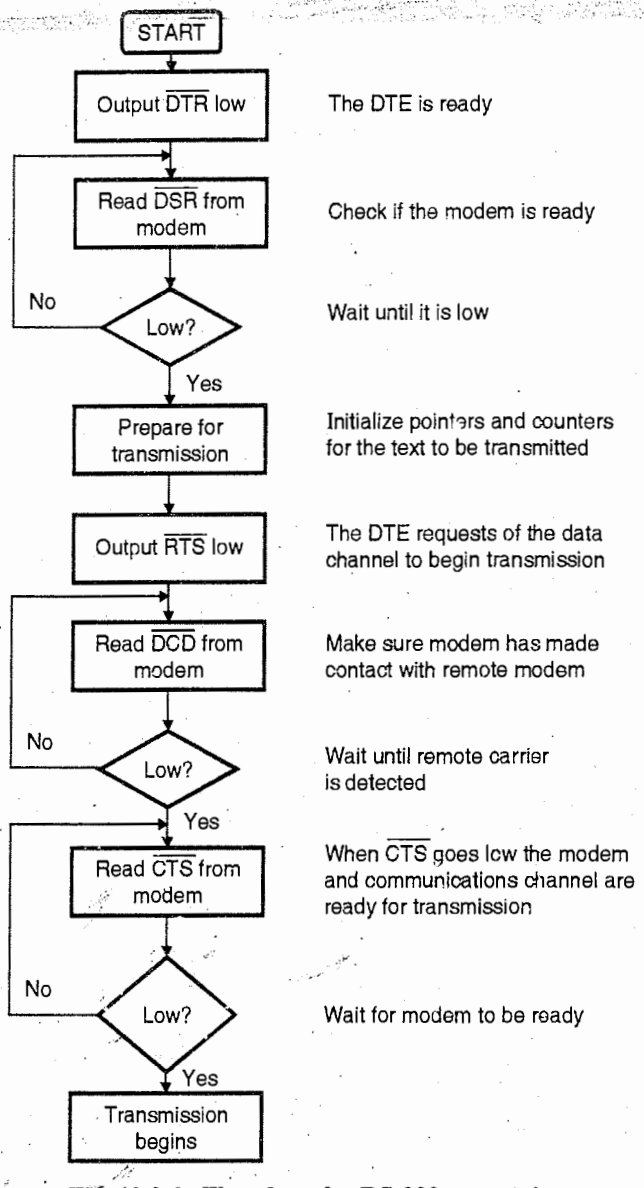


Fig. 12.3.6 : Flowchart for RS-232 connection

Syllabus Topic : RS-485

12.4 Study of RS-485

Q. Explain the following bus standard : RS485

– EIA standard RS-485 was introduced in 1983, is an upgraded version of EIA RS422-A. Increasing use of balanced data transmission lines in distributing data to several system components and peripherals over long lines brought about the need for multiple driver / receiver combinations on a single twisted pair line.

- It considers RS422-A requirements for balanced line transmission along with added features allowing multiple drivers and receivers.
- Fig. 12.4.1 shows an application with multiple drivers and receivers.

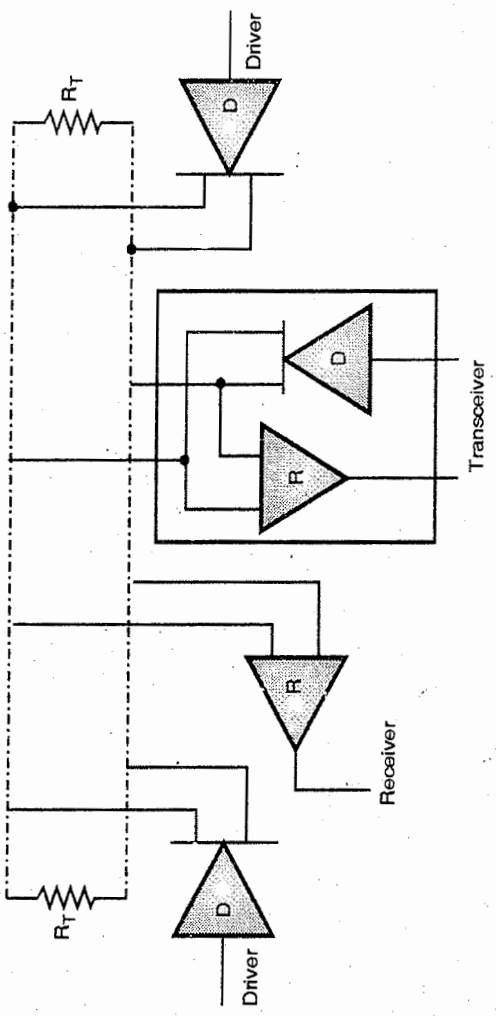


Fig. 12.4.1 : Multiple drivers and receivers interface using EIA RS485 standard

- Standard RS-485 differs from the other standards i.e. RS232, RS422-A in the features that allow **multipoint communication**.

12.4.1 Features for Drivers

- The **features** for drivers are :
  (i) A driver can drive upto 32 unit loads and total line termination resistance of 60 Ω or more (one unit load is typically one passive driver and one receiver). Thus, RS485 supports 32 drivers and 32 receivers (supporting bi-directional- half duplex-multi-drop communication).
  (ii) The driver output, off leakage current must be 100 μA or less with line voltage from – 7 V to + 7 V.
  (iii) The driver must be capable of providing differential output voltage of 1.5 V to 5 V with common-mode line voltages from – 7 V to + 12 V.
  (iv) The drivers must have self protection against contention. (Multiple drivers contending for the transmission line at same time). No driver damage will occur when outputs are connected to a voltage source of – 7 V to + 12 V irrespective of the state of binary output (i.e. 0 or 1)

12.4.2 Features for Receivers

- The **features** for receiver are :
  (i) High receiver input resistance of 12 KΩ is minimum.
  (ii) Differential input sensitivity of ± 200 mV over a common mode range of – 7 V to + 12 V
  (iii) A receiver input common mode range of – 7 V to + 12 V
- The SN75172B and SN75174B drivers exhibit a differential output transition time of 75 ns maximum. This parameter is the key to maximum speed at which the driver can operate in accordance with EIA RS-485 specifications.

12.4.3 Specifications of the RS485

Specifications	RS485
Mode of Operation	Differential
Total Number of Drivers and Receivers on One Line DRIVER	32 Drivers 32 Receivers
Maximum Cable Length	4000 ft.
Maximum Data Rate	10Mb/s
Maximum Driver Output Voltage	– 7V to +12V
Driver Output Signal Level (Loaded Min.) (Loaded)	+/– 1.5V
Driver Output Signal Level (Unloaded Max)(Unloaded)	+/– 6V
Driver Load Impedance (Ohms)	54
Max. Driver Current in High Z State (Power On)	+/– 100μA
Max. Driver Current in High Z State (Power Off)	+/– 100μA
Slew Rate (Max.)	N/A
Receiver Input Voltage Range	– 7V to +12V
Receiver Input Sensitivity	+/– 200mV
Receiver Input Resistance (Ohms)	> = 12k Ω

12.4.4 Versions of RS 485

RS 485 exists in two versions: Single Twisted Pair or Dual Twisted Pairs.

(1) Single Twisted Pair RS 485

- In this version, all devices are connected to a single Twisted Pair. So all the signals of RS 485 must have drivers with tri-state outputs (including the Master).
- The communication precedes over the single line in both directions.

(2) Double Twisted Pair RS 485

- In this version, the Master does not have to have tri-state output. The Slave devices transmit over the second twisted pair, which is intended for sending data from Slave to Master.
- This technique allows the user to implement multipoint communication in systems, which were originally designed for the RS 232.
- The course, Master software needs to be modified. The Master periodically sends query packets to all Slave devices.
- RS 485 system can also work in a point-to-point system. The high impedance state of the RS 485 output driver is not used.

12.4.5 Advantages of RS485 Over RS232

The advantages of RS485 over RS232 are as follows :

(i) It requires just a single +5V (or lower) supply to generate the required minimum 2V difference at the differential outputs. But RS232 needs dual supply or an expensive interface chip that generates the supplies. RS485 drivers and receivers are hence inexpensive.

(ii) RS485 is a multi-drop interface that can have multiple drivers and receivers. But RS232 is limited to two devices.

(iii) RS485 link can extend upto 1200 m while RS232 can go upto 15 m.

(iv) RS485 can transfer at the rate upto 10 Mbps.

(v) RS485 is a robust standard for use in industrial environment.

(vi) Several microcontrollers have built in UARTs that support RS485 communication.

Syllabus Topic : Study of I^2C Bus Standard

12.5 Study of I^2C Bus Standard

Q. Explain operation of I^2C protocol.

- Communication systems have grown in size as well as complexity. They have high speed ICs with critical operating parameters. They must give an extremely reliable service.
- To maintain the performance of the communication systems environmental monitoring needs to be done in order to avoid failure.
- To support all these requirements Philips has developed a simple bi-directional 2 wire bus called inter IC bus (integrated circuit bus) or I^2C bus in 1970.
- It uses a comprehensive protocol to ensure reliable transmission and reception of data within the devices that are connected.
- Each device can operate like a transmitter or receiver. The address of each and every device is unique.
- It is a multi-master bus. It indicates that more than one IC is capable of initiating data transfer can be connected to it.
- The I^2C protocol states that the IC that initiates a data transfer on the bus is called as the **bus master.** Consequently all other ICs are regarded as **bus slaves.**

12.5.1 Features

(i) It comprises of two lines: (i) clock line (SCL) and (ii) data line (SDA). The clock line is utilized to strobe the data from SDA line or to the master that has control over the bus.

(ii) Each device connected to the bus has its own unique address no matter it is a microcontroller, LCD driver, memory or keyboard. Each of the chips can behave like a transmitter or receiver, depending on the functionality. The LCD driver is a receiver while memory or I/O chip can be both transmitter as well as receiver.

(iii) It allows sharing of network resources between the processors. The process of sharing network resources between processors is called **multi-mastering.**

(iv) It includes collision detection and arbitration to avoid data corruption if two or more masters simultaneously initiate data transfer.

(v) The on-chip filtering rejects spikes on data bus inorder to preserve data integrity.

(vi) The data transfer is bidirectional, serial and 8 bit. They can be done at :

(a) upto 100 KB/s in standard mode

(b) upto 400 KB/s in fast-mode

(c) upto 3.4 MB/s in High-speed mode

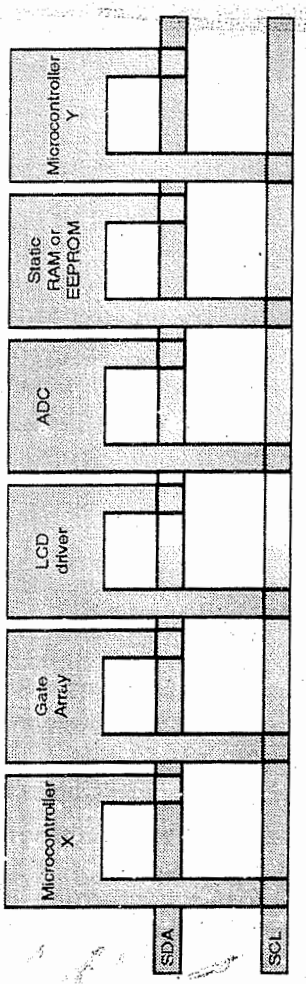
(vii) The number of ICs that can be connected to same bus is limited by a maximum bus capacitance of 400 pF.

12.5.2 I²C Bus Terminology

Transmitter	The device that sends the data to the bus.
Receiver	It is the device that receives the data from the data bus.
Master	It is the device that initiates the transfer, generates the clock. It also terminates the transfer.
Slave	It is addressed by a master.
Arbitration	It is a process that guarantees that only one master devices will control the bus. It ensures that the transfer data does not get corrupted.
Multi-Master	It comprises of more than one master in the system. The masters try to control the bus at the same time without corrupting the message.
Synchronization	It is a method where the clock signals of two or more devices are synchronized.

12.5.3 I²C Bus Configuration

- The I²C bus is a multi master bus i.e. more than one device capable of controlling the bus can be connected to it.
- Consider two microcontrollers connected to I²C bus as shown in Fig. 12.5.1.



m(14.9) **Fig. 12.5.1 : Two microcontrollers connected to I²C bus**

The transfer of data would proceed as follows :

1. Suppose microcontroller X wants to send information to microcontroller Y :
   - microcontroller X (master), addresses microcontroller Y (slave)
   - microcontroller X (master-transmitter), sends data to microcontroller Y (slave-receiver)
   - microcontroller X terminates the transfer.
2. If microcontroller X wants to receive information from microcontroller Y :
   - microcontroller X (master) addresses microcontroller Y (slave)
   - microcontroller X (master-receiver) receives data from microcontroller Y (slave-transmitter) microcontroller X terminates the transfer.
   - In condition 2, the master (microcontroller X) will generate the required timing signal. Then the microcontroller X will end the

transfer. The probability that more than one microcontroller can be connected to the I²C-bus indicates more masters can attempt to initiate the data transfer simultaneously.

- An event arbitration procedure is developed in order to eliminate the simultaneous data transfer. The event arbitration procedure is dependant on the wired-AND connection of all I²C interfaces to the I²C-bus.
- If two or more masters make an attempt to place the data onto the bus, the first master produces a 'one' while the other master produces a 'zero'. In this case the arbitration is lost. The clock signals are synchronized. At the time of arbitration the clock signals will be a combination of the clocks that are generated by the masters with the help of the wired-AND connection made to the SCL line. Generally the masters are held responsible for generating the clock signals on the I²C-bus. Every master generates its own clock signals whenever the master wishes to transmit or receive data on the I²C-bus. The Bus clock signals that the master generates can only be modified in two ways either when they are pulled by a slow-slave device that holds-down the clock line or if arbitration occurs by another master microcontroller.

12.5.4 I²C Signals Conditions

The I2C signal conditions are as follows :

(i) **Start condition :** High to Low transition of the SDA while SCL line is high.

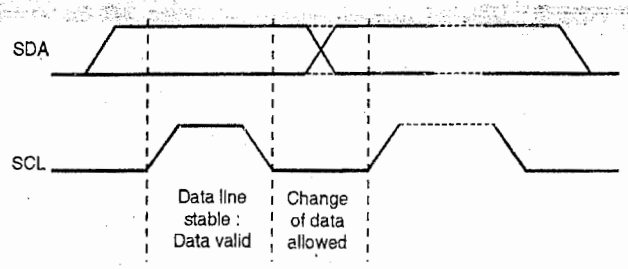
(ii) **Stop condition :** Low to High transition of the SDA while SCL line is high.

(iii) **ACK :** RECEIVER pulls SDA line low while the transmitter allows it to flow high.

(iv) **Data valid :** Transition occurs when SCL is low.

Data validity

Whenever the clock signal is high the data that is present on the SDA line should be stable. The HIGH or LOW transition of the data line can be modified whenever the clock signal on the SCL line is LOW as shown in Fig 12.5.2.

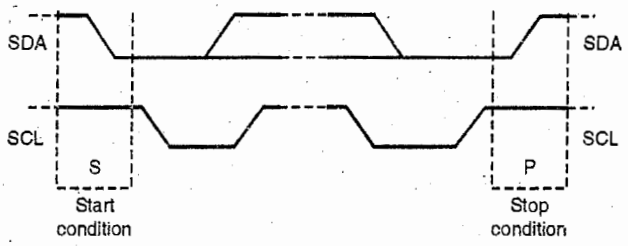


m(14.10)**Fig. 12.5.2 : Bit transfer on I²C bus**

12.5.5 Start and Stop Conditions

Q. Explain operation of I²C protocol with timing diagram for START and STOP.

- Fig. 12.5.3 shows start and stop conditions. Within the procedure of the I²C-bus, unique situations arise which are defined as START and STOP conditions.
- If the SCL is at logic 1 i.e. high then a HIGH to LOW transition on the SDA line shows the beginning of a **START** condition.
- Whereas if the SCL is at logic 1 i.e. high then a LOW to HIGH transition on the SDA line shows the **STOP** condition.
- The master is always responsible for generating the START and STOP conditions.
- Once the START condition has begun, the bus is assumed to be busy. After the STOP condition has ended the bus is again considered to be free.
- If the devices contain correct interfacing hardware then the START and STOP conditions can be easily detected. However, if the devices do not have necessary interfacing hardware then in order to sense the transition the devices need to sample the SDA line at least twice per clock period.



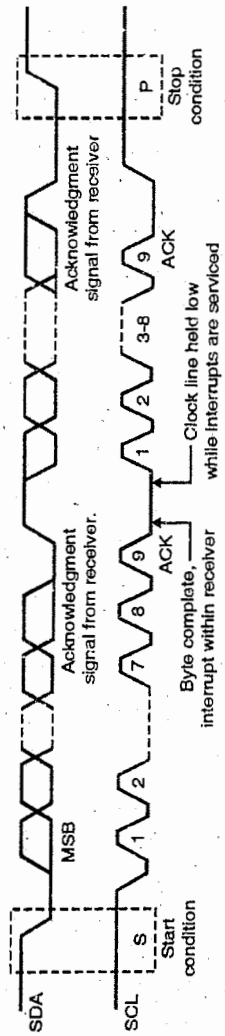
m(14.11)**Fig. 12.5.3 : Start and Stop conditions**

12.5.6 Transferring Data

12.5.6.1 Byte Format

- Every byte that can be placed on the SDA line should be of 8-bits. PER transfer any unlimited number of bytes can be transmitted.

- However every byte that is transmitted follows an acknowledge bit as shown in Fig. 12.5.4. Initially the data is transferred with the most significant bit (MSB).
- If a slave is unable receive a complete data byte till it completes some other task, for example if the slave is busy in servicing an internal interrupt, then the slave can hold the SCL clock line LOW. This will force the transmitter to enter into a wait state. Once the SCL clock line is released and the receiver becomes ready to accept the data byte the data transfer will resume again.
- In some cases, it's permitted to use a different format from the I^2C-bus format. 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.

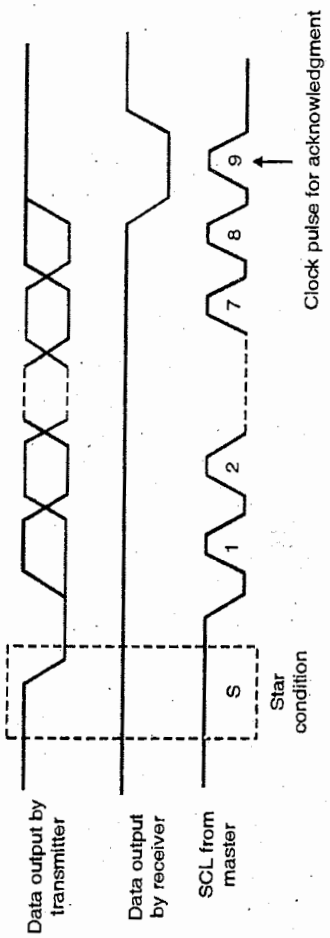


m(14.12)**Fig. 12.5.4 : Data transfer on the I^2C-bus**

12.5.6.2 Acknowledge

Q. Explain operation of I^2C protocol with timing diagram for AKNOWLEDGE.

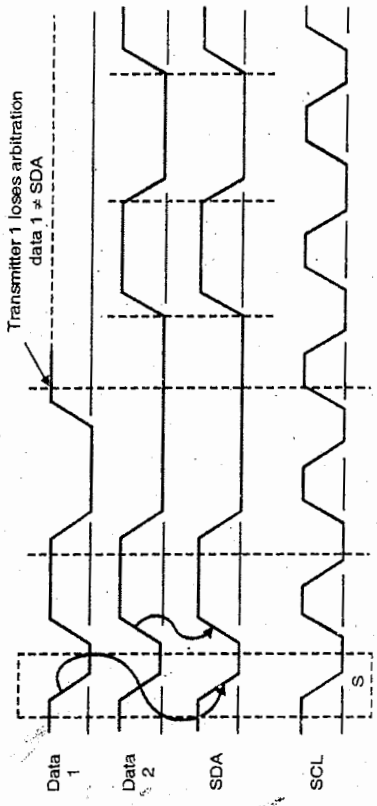
- Every byte of data that is transferred must have an acknowledge. Acknowledgement form the receiver is a compulsion. The master device generates the clock pulses that are required for acknowledging the-correct byte data transfer .
- At the time the acknowledge clock pulse is given the transmitter will assert the SDA line (HIGH). The receiver then deactivates the SDA line (LOW) at the time of acknowledge clock pulse as shown in Fig. 12.5.5.
- The set-up and hold times must also be considered. Generally, a receiver which has been addressed is obliged to generate an acknowledge after each byte has been received.
- If a slave-address is not acknowledged by the slave receiver as the receiver is busy in servicing some other functions then the data line should remain HIGH. In such a situation if the master wishes to quit the data transfer, the master can generate a STOP condition .
- However if after sometime the slave-receiver acknowledges the slave address it is not able receive the data bytes. For receiving the data bytes the master should terminate the data transfer. In such situation the slave device does not acknowledge the first byte of the data transfer and leaves the data line HIGH . Then in response the master device will generate the STOP condition.
- If the data transfer comprises a master-receiver then on the last data byte that is transmitted by the slave , the slave-transmitter will not transmit an acknowledge . Also the data line must be released by the slave-transmitter so that the master device can generate the repeated START or STOP conditions.



m(14.13)**Fig. 12.5.5 : Acknowledge on I²C bus**

12.5.7 Arbitration

- A master device can initiate a data transfer provided that the I²C bus is free. Many masters can be used for generating the START condition. The START condition that is generated should be within the minimum hold time ($t_{HD;STA}$) . This yields in a defined START condition .
- Arbitration occurs on the SDA line. At the time of Arbitration the SCL line is HIGH. This allows one master to transmit at a HIGH level, while the other master transmits a LOW level. Such a data transfer at high as well as low level will switch off the DATA output stage. This is because the level on the bus is not stable and results in Arbitration.
- Arbitration can continue till the level of data on the bus becomes stable. In the first stage of Arbitration address bits are compared as see whether the same address is requested by the different master devices. If the address is same as the device address then arbitration will still continue. During the arbitration process no information loss will occur as only the address and data information is used .



m(14.14)**Fig. 12.5.6 : Arbitration procedure of two masters**

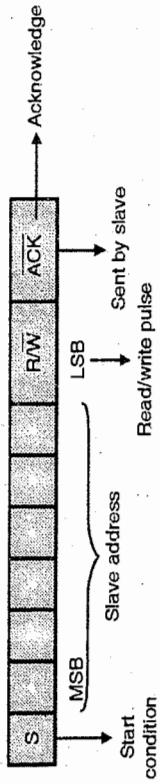
- A master which loses the arbitration can generate clock pulses until the end of the byte in which it loses the arbitration. If a master also incorporates a slave function and it loses arbitration during the addressing stage, it's possible that the winning master is trying to address it. The losing master must therefore switch over immediately to its slave-receiver mode.
- Fig. 12.5.6 shows the arbitration method for two masters. At the instant we observe a change in difference in the internal data level of the master that generates the DATA 1 and the actual level on the SDA line, the data output will be switched off. This indicates that a HIGH output level is connected to the bus. This does not disturb the data transfer operation.
- If the arbitration process is in progress then we need to observe the instants at which START condition or STOP conditions are transmitted

on the I²C-bus. If such repeated conditions are observed then the masters needs to transmit the repeated START condition or STOP condition at the same position in the format frame.

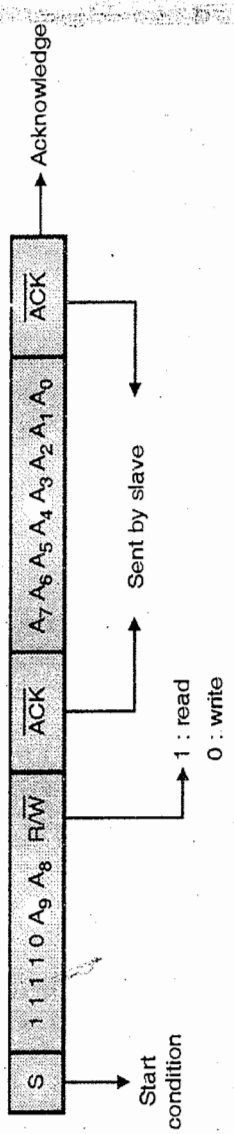
- In other words, arbitration isn't allowed between :
  1. A repeated START condition and a data bit
  2. A STOP condition and a data bit
  3. A repeated START condition and a STOP condition.

12.5.8 Addressing I²C Devices

- There are two address formats : **7 bit and 10 bit.**
- The 7 bit format is the simplest format with a R/$\overline{W}$ bit.
- For the 10 bit addressing mode two bytes must be transmitted with the first five bits specifying it to be 10 bit address. It supports up to 1024 slave addresses. It does not affect the 7 bit addressing.
- Devices with 7 and 10 bit addresses can be connected to the I²C bus.
- Figs. 12.5.7 and 12.5.8 show the 7 and 10 bit address format.



m(14.15)**Fig. 12.5.7 : 7 bit address format**

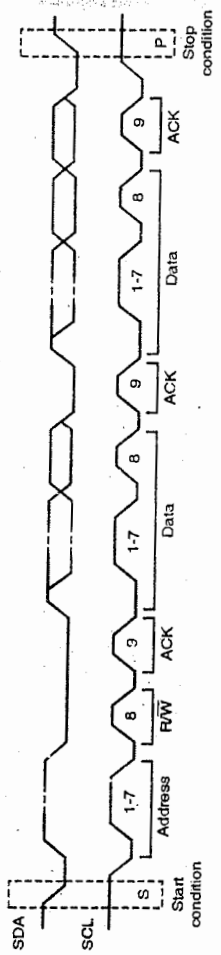


m(14.16)**Fig. 12.5.8 : 10 bit Addressing**

12.5.9 Complete Data Transfer

Q. Explain operation of I²C protocol with timing diagram for Send address.

- Data transfers follow the format shown in Fig. 12.5.9.
- On the START condition (S), a slave address is sent. This address is 7 bits long followed by an eighth bit which is a data direction bit (R/$\overline{W}$) a 'zero' indicates a transmission (WRITE), a 'one' indicates a request for data (READ). A data transfer is always terminated by a STOP condition (P) generated by the master.

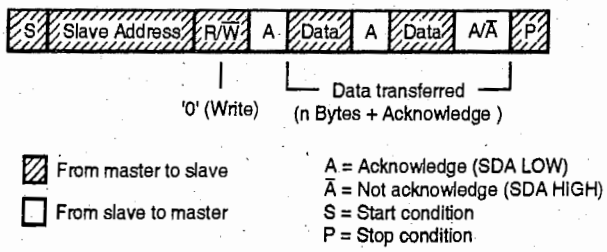


m(14.17)**Fig. 12.6.9 : Complete Data Transfer**

- However, if a master device wants to transmit/receive the data on the bus, the master device generate a repeated START condition (Sr). Also without generating a STOP condition the master device can address some other slave device.
- Various combinations of read/write formats are then possible within such a transfer. Possible data transfer formats are :

1. **Master-transmitter transmits to slave-receiver.**

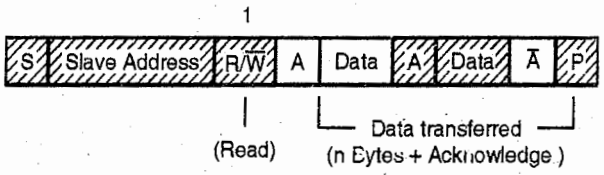
The transfer direction is not changed (Fig. 12.6.10)



m(14.18)**Fig. 12.5.10 : A master-transmitter addresses a slave receiver with a 7-bit address. The transfer direction is not changed**

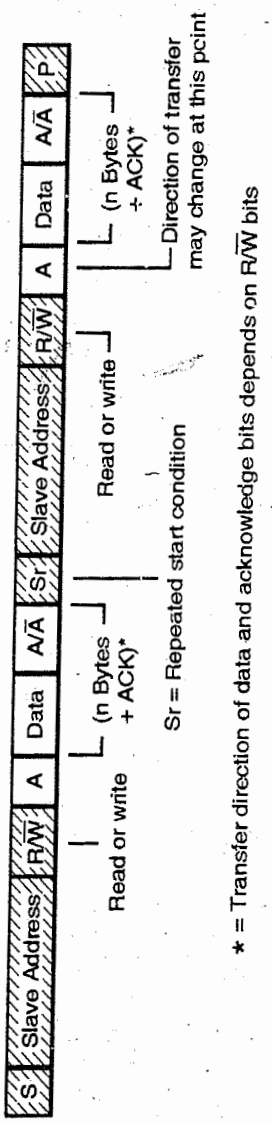
2. **Master reads slave immediately after first byte** (Fig. 12.5.11).

At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This acknowledges is still generated by the slave. The STOP condition is generated by the master



m(14.19)**Fig. 12.5.11 : A master reads a slave immediately after the first byte**

3. **Combined format** (Fig. 12.5.12).



m(14.20)**Fig. 12.5.12 : Combined format**

- During a transition change at the time of data transfer, the START condition as well as and the slave address are both repeated . However the R/$\overline{W}$ bit reversed i.e. if bit was 1 it will be 0 and vice-versa. If a repeated START condition is sent by the master-receiver, then initially we have received a not acknowledge (A).

Notes :

1. Combined formats can be used, for example, to control a serial memory. During the first data byte, the internal memory location has to be written. After the START condition and slave address is repeated, data can be transferred
2. All decisions on auto-increment or decrement of previously accessed memory locations etc. are taken by the designer of the device.
3. Each byte is followed by an acknowledgement bit as indicated by the A or A blocks in the sequence.
4. I^2C-bus compatible devices must reset their bus logic on receipt of a START or repeated START condition such that they all anticipate the sending of a slave address.

12.5.10 Extensions to the I^2C-Bus Specification

- The I^2C-bus with a data transfer rate of up to 100 kbit/s and 7-bit addressing has now been in existence for more than ten years with an unchanged specification. The concept is accepted world-wide as a de facto standard and hundreds of different types of I^2C-bus compatible ICs are available from Philips and other suppliers.
- The I^2C-bus specification is now extended with the following features:
  1. A **fast-mode** which allows a fourfold increase of the bit rate to 0 to 400 kbit/s
  2. **10-bit addressing** which allows the use of up to 1024 additional slave addresses.
  3. **High Speed mode (HS mode)** with a bit rate of 3.4 MBits/s.

12.5.11 Fast-Mode

In the fast-mode of the I^2C-bus, the protocol, format, logic levels and maximum capacitive load for the SDA and SCL lines quoted in the previous I^2C-bus specification are unchanged. Changes to the previous I^2C-bus specification are:

1. The maximum bit rate is increased to 400 kbit/s
2. Timing of the serial data (SDA) and serial clock (SCL) signals has been adapted. There is no need for compatibility with other bus systems such as CBUS because they cannot operate at the increased bit rate
3. The inputs of fast-mode devices must incorporate spike suppression and a Schmitt trigger at the SDA and SCL inputs
4. The output buffers of fast-mode devices must incorporate slope control of the falling edges of the SDA and SCL signals
5. If the power supply to a fast-mode device is switched off, the SDA and SCL I/O pins must be floating so that they don't obstruct the bus lines
6. The external pull-up devices connected to the bus lines must be adapted to accommodate the shorter maximum permissible rise time for the fast-mode I^2C-bus.

12.5.12 10-BIT Addressing

- The 10-bit addressing does not change the format in the I^2C-bus specification. Using 10 bits for addressing exploits the reserved combination 1111XXX for the first seven bits of the first byte following a START (S) or repeated START (Sr) condition.
- 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, and both 7-bit and 10-bit addressing can be used in a standard-mode system (up to 100 kbit/s) or a fast-mode system (up to 400 kbit/s).
- We know that the reserved address bits has eight possible combinations of 1111XXX. However for 10 bit addressing we can only use the four combinations 11110XX. For further enhancements of the bus the other four combinations are reserved and Inaccessible to the user.

12.5.13 HS Mode (High-speed mode)

- High-speed mode (HS-mode) devices provide a quantum leap in I^2C-bus transfer speeds.
- The devices that support the HS -mode can transfer data at speeds of up to 3.4 Mbit/s, The devices are completely compatible with the other modes i.e. the Fast- or Standard-mode (F/S-mode) that are used for the bi-directional communication .
- In this mode of data transfer arbitration and clock synchronization is not done. Depending on

the nature of data transfer application, the new devices that are used support the Fast or Hs-mode I²C-bus interface. The devices that support this mode are preferred to be used in a variety of applications. To obtain the data transfer at high speeds the enhancements added to the existing I²C-bus specifications are :

- The master devices supporting the high speed mode of operation comprise an open-drain output buffer for the SDAH signal. On the SCHL output they also provide a combination of an open-drain pull-down and current-source pull-up circuit. The current-source circuit will decrease the SCLH signals rise time. For an operation at a time only a master current source is selected provided the device is working in the HS mode.
- In this mode of data transfer arbitration and clock synchronization is not done. This helps in speeding up in the data handling. After the data transmission in the F/S mode by the master device the process of arbitration finishes.
- The master devices operating in the high speed master devices will produce a serial clock signal that has HIGH to LOW ratio. Generally this ratio is of 1 to 2.
- The master devices working in this mode can support a built-in bridge. This bridge is responsible for separating the SDAH and SCLH signals. The signals are separated from the SDA and SCL lines of devices supporting the F/S-mode. This will decrease the capacitive load of the signals . Also it will help in providing faster rise and fall times.
- The operating speed is the only distinguishing parameter between F/S mode and HS mode slave devices.
- The HS mode devices comprise of Schmitt trigger at the SDAH and SCLH inputs. And also spike suppression. For the falling edges of the SDAH and SCLH signals slope control is provided by the output buffers.

12.5.14 Advantages

(1) Inorder to establish full-fledged I²C bus , two bus lines are more than sufficient.

(2) Each slave device has a different slave address. It is connected uniquely.

(3) The devices incorporating I²C bus can select either a short 7 bit addressing or 10 bit addressing scheme.

(4) As the master is responsible for driving the clock, no specific baud rate is specified. True multimaster support with up to 8 masters in a single bus system.

(5) I²C is a very simple protocol. It can be easily emulated even if the microcontroller does not have an integrated I²C peripheral device.

(6) It is cheap

(7) I²C bus supports data transfer at speeds up to 3.4 Mbits/sec.

12.5.15 Disadvantages

(1) I²C bus that allows the 7 bit addressing will supports very few devices.

(2) The addresses of the different devices that are produced by the different manufacturers are hard coded slave address. This can cause the address clashes.

(3) It does not support automatic bus configuration neither it supports plug and play.

12.5.16 Applications

I²C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed. Common applications of the I²C bus are:

- Reading configuration data from SPD EEPROMs on SDRAM, DDR SDRAM, DDR2 SDRAM memory sticks (DIMM) and other stacked PC boards
- Supporting systems management for PCI cards, through a SMBus 2.0 connection.
- Accessing NVRAM chips that keep user settings.
- Accessing low speed DACs and ADCs.
- Changing contrast, hue, and color balance settings in monitors (Display Data Channel).
- Changing sound volume in intelligent speakers.
- Controlling LED /LCD displays, like in a cell-phone.
- Reading hardware monitors and diagnostic sensors, like a CPU thermostat and fan speed.
- Reading real time clocks.
- Turning on and turning off the power supply of system components.

Syllabus Topic : Study of SPI (Serial Peripheral Interface)

12.6 Study of SPI (Serial Peripheral Interface)

SPPU - Dec. 16

University Question

Q. How SPI is better than I2C Bus. **(Dec. 2016, 2 Marks)**

- The **SPI (Serial Peripheral Interface)** provides a bi-directional, synchronous serial communication between the microcontrollers and peripherals.
- This protocol is based on a master-slave protocol where the master is the device that drives the clock signal.
- A number of masters and slaves are allowed on the bus.

Fig. 12.6.1 shows a single master and two slave peripherals. Master drives the slave signals only one slave can be accessed at a given time.

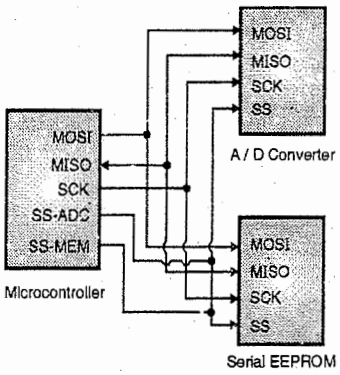


Fig. 12.6.1 : One master and two slave peripherals

12.6.1 Features of SPI Bus

(i) It is defined by Motorola on the MC68HCxx line of microcontrollers.
(ii) The data word size is of 8 bits.
(iii) It is a synchronous serial data link operating at full duplex mode.
(iv) It provides hold facility that allows the transmitter to suspend transfer.
(v) It is a synchronous serial data link operating at full duplex mode.
(vi) The data can be transferred in multiple bytes called as blocks or pages.

12.6.2 Standard Naming Convention in SPI

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
SCK : Serial clock (It is always by the master)

- The data transfer on an SPI bus can be implemented using a large shift register shared between the master and slave devices.
- The data is clocked IN at the same time it is clocked OUT of the devices. The clock is symmetrical i.e. its ON and OFF period is same.

Fig. 12.6.2 shows internal SPI registers.

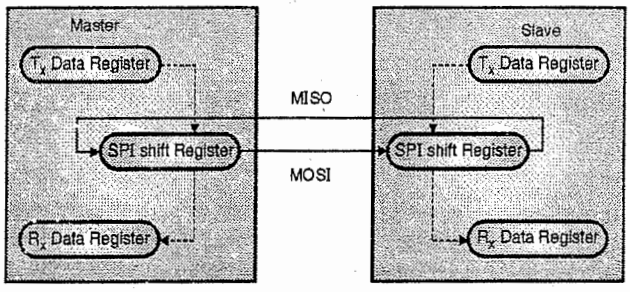


Fig. 12.6.2 : Internal SPI registers

12.6.3 Clock Polarity and Clock Phase

- Serial peripheral interface needs that the master and the slave devices must agree regarding the idle state of the clock signal and the manner in which the data will be clocked during the SPI transfer.

Clock Phase (CPHA)

This parameter indicates when the data must be presented and when the data must be sampled.

CPHA = 0 : The first edge on the SCK line is used to sample the first data bit and second edge is used to present it.

CPHA = 1 : The first edge on the SCK line is used to present the data and the second edge to sample it.

Clock Polarity (CPOL) :

This parameter indicates the level of clock signal when it is idle.

CPOL = 0 Clock idle state is low.
CPOL = 1 Clock idle state is high.

Fig. 12.6.3 shows the clock polarity and clock phase setting.

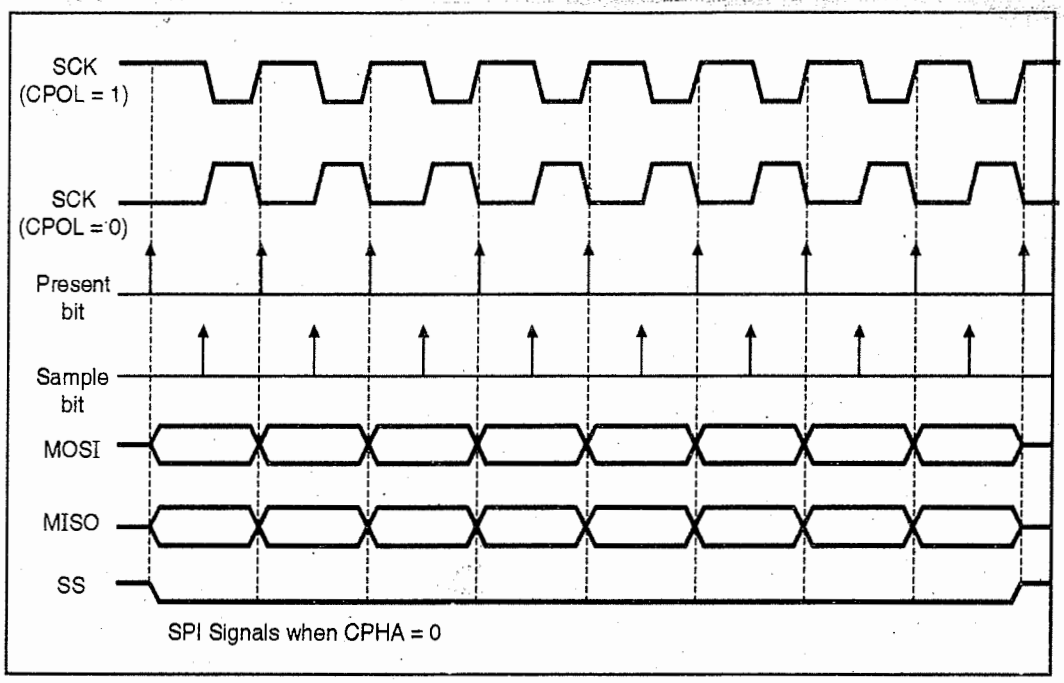


Fig. 12.6.3(a) : Clock polarity

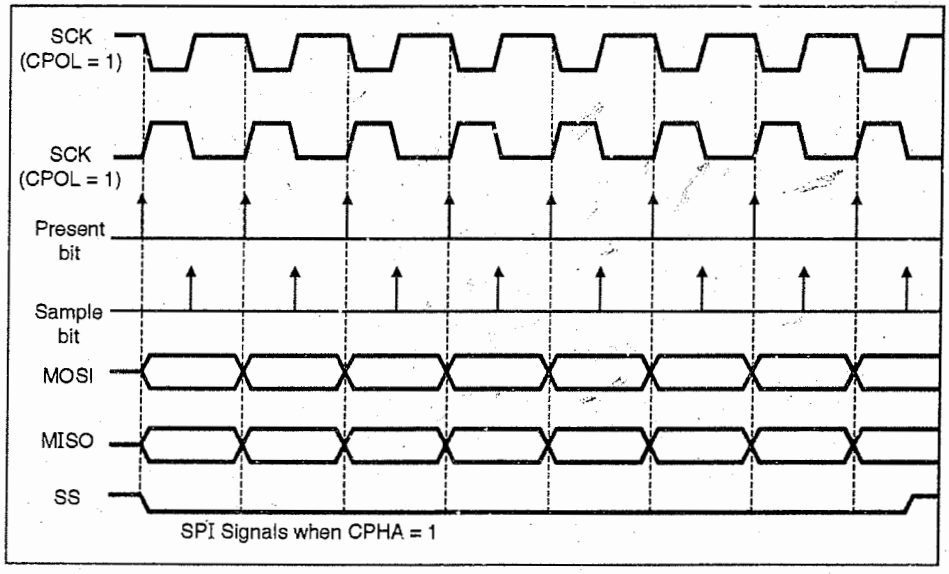


Fig. 12.6.3(b) : Clock phase settings

12.6.4 Applications

(i) Full duplex capability is support by SPI bus. Hence it can be used in communication between a codec and a digital signal processor.
(ii) It is used in EEPROM, flash and real time clocks.
(iii) ADC converters.

12.7 Comparison between I²C and SPI SPPU - May 15, Dec. 15, May 16

University Question

Q. Compare SPI and I²C protocol.
(May 2015, Dec. 2015, May 2016, 8 Marks)

Sr. No.	SPI	I²C
1.	For point to point communication it is simple and efficient.	For point to point communication it is complex.
2.	SPI system is full duplex.	I²C system is simplex.
3.	Less overhead due to lack of addressing.	It has more overheads.
4.	For multiple slaves, each slave needs a separate slave select signal hence more hardware is required.	For multiple slaves, same slave select signal can be given, hence less hardware is required.

Syllabus Topic : MSSP Structure

12.8 MSSP Structure

SPPU - Dec. 14, May 15, Dec. 15, May 16

University Question

Q. Draw and explain MSSP structure of PIC18FXX.
(Dec. 2014, May 2015, Dec. 2015, May 2016, 8 Marks)

- The **Master Synchronous Serial Port (MSSP)** of PIC18 is a serial interface. It is used for communicating the PIC microcontroller with the different peripheral devices like A/D converters, D/A converters, EEPROMS, RTCs, shift registers, display drivers, SD cards, temperature sensors, USB devices etc.
- For data transmission and reception, the MSSP needs a common clock signal for the transmitter and the receiver.
- The MSSP module supports two operating modes. They are :
  (i) Serial Peripheral Interface (SPI)
  (ii) Inter-Integrated Circuit (I2C)
- SPI protocol was developed by Motorola. This serial interfacing method today has become an industry standard because of its easy use and flexibility.
- I2C protocol was developed by Philips. This serial interfacing method supports data transfers at 100 Kbps, 400 Kbps and high speed data transfers at 3.4 Mbps.
- Both the SPI and I2C protocols share the same signal pins. However, both these protocols cannot be active at the same time. The pins used by SPI and I2C MSSP module are :

(i) Data Clock (SCK) → RC3/SCK/LVDIN
(ii) Serial Data In (SDI) → RC4/SDI/SDA.
(iii Serial Data Out (SDO) → RC5/SDO

Syllabus Topic : MSSP-SPI Mode

12.9 MSSP-SPI Mode

SPPU - Dec. 14, May 15, Dec. 15, May 16, Dec. 16

University Questions

Q. Draw and explain MSSP structure of PIC18FXX.
(Dec. 2014, May 2015, Dec. 2015, May 2016, 8 Marks)

Q. Explain the MSSP with SPI mode.
(Dec. 2016, 8 Marks)

- The SPI mode supports 8 bit data transmission and reception simultaneously. One device behaves like a master while the remaining devices are slaves in a system using SPI interface.
- The master device is responsible for generating the clock signal and also synchronizing the data transfer.
- For SPI mode operation the MSSP supports four registers. They are :
  (i) MSSP Status Register (SSPSTAT)
  (ii) MSSP Control Register 1 (SSPCON1)
  (iii) Serial transmit / receive buffer (SSPBUF)
  (iv) MSSP shift register (SSPSR) - not accessible to the programmer.

Fig. 12.9.1 shows the MSSP structure in the SPI mode.

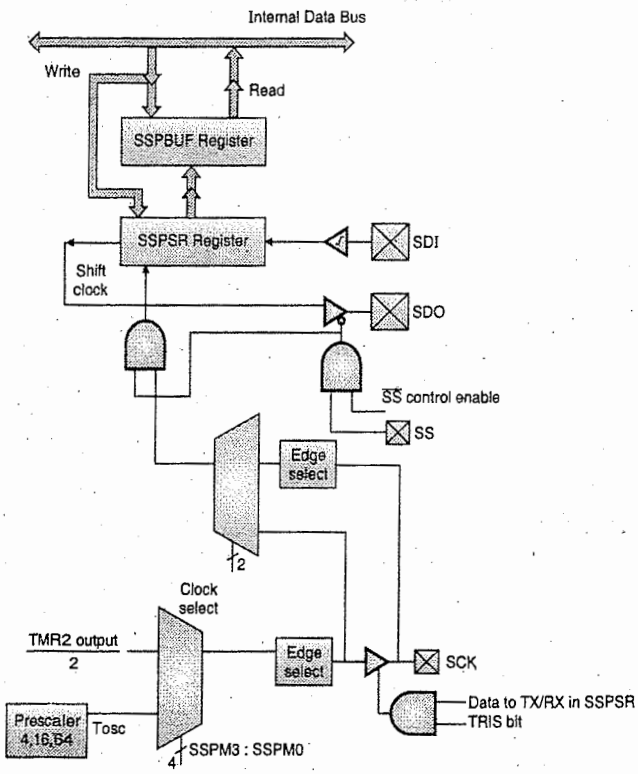


Fig. 12.9.1 : MSSP structure in SPI mode

12.9.1 MSSP Status Register (SSPSTAT)

Fig. 12.9.2 shows the MSSP Status Register SSPSTAT (SPI mode).

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF

SMP : Sample bit
1 = Input data sampled at the end of data output time.
0 = Input data sampled at middle of data output time.

CKE : SPI clock edge select bit
1 : Data transmitted on rising edge of SCK.
0 : Data transmitted on falling edge of SCK.

D/$\overline{A}$: Data / Address bit : This bit is used in I2C mode only.

P : Stop bit : This bit is used in I2C mode only.

S : Start bit : This bit is used in I2C mode only.

R/$\overline{W}$: Read / write Bit : This bit is used in I2C mode only.

UA : Update Address : This bit is used in I2C mode only.

BF : Buffer Full Status bit : (Receive mode only for SPI)
1 = receive not complete, SSPBUF full.
0 = Receive not complete, SSPBUF empty.

Fig. 12.9.2 : SSPSTAT register (SPI mode)

SSPSTAT is the status register for SPI mode operation. The lower six bits of SSPSTAT are read only while upper two bits are readable/writable.

12.9.2 SSPCON 1 : MSSP Control Register 1

WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0

WCOL : Write Collision Detect Bit.
0 = no collision
1 = The SSPBUF register is written while it is transmitting the earlier word.

SSPOV : Receive Overflow Indicator Bit.
SPI master mode : This bit is not set
SPI slave mode
0 : No overflow
1 : A new byte is received when previous data is in SSPBUF register. The data in SSPSR is lost if there is overflow.

SSPEN : Synchronous Serial Port Enable Bit.
0 : disables serial port and configures the pins as I/O pins.
1 : enables serial port and configures SCK, SDI, SDO.

CKP : Clock polarity select bit
0 : Idle state for clock is low level
1 : Idle state for clock is high level

SSPM3 : SSPM0 : Synchronous serial port mode select bits

0000 : SPI master mode, clock = $\frac{f_{osc}}{4}$

0001 : SPI master mode, clock = $\frac{f_{osc}}{16}$

0010 : SPI master mode, clock = $\frac{f_{osc}}{64}$

0011 : SPI master mode, clock = $\frac{\text{TMR 2 output}}{2}$

0100 : SPI slave mode, clock = SCK Pin · $\overline{SS}$ pin control enabled.

0101 : SPI slave mode, clock = SCK Pin · $\overline{SS}$ pin control disabled.

Fig. 12.9.3 : SSPCON 1 (SPI mode)

12.9.3 SSPBUF and SSPSR Registers

- While transferring data, the data is read/written into the SSPBUF register. The SSPSR register is not accessible to the programmer.
- The SSPSR register shifts data in or out of the device. Both the SSPBUF and SSPSR registers create a double buffered receiver for implementing receive functions. Because of this before reading the first byte received, the reception of next data byte begins. The data byte is transferred to the SSPBUF register after the byte is received. Also the SSPIF flag bit is set.
- When the data is transmitted, SSPBUF register is not double buffered.

12.9.4 SPI Operation

Fig. 12.9.4 shows SPI master-slave connection.

- While connecting the master and slave devices, the SDO pin of one device should be connected to SDI pin of other SPI device and vice-versa.
- The SCK signal is driven by SPI master. This signal is responsible for controlling the data shifting for SPI transfer.
- The data transfer is initiated by sending the SCK signal to the slave device. On the programmed clock edges data is shifted out of the shift registers. On the opposite clock edge this shifted data is latched.
- The clock for both the master and slave devices must be same so that they can simultaneously send and receive data.

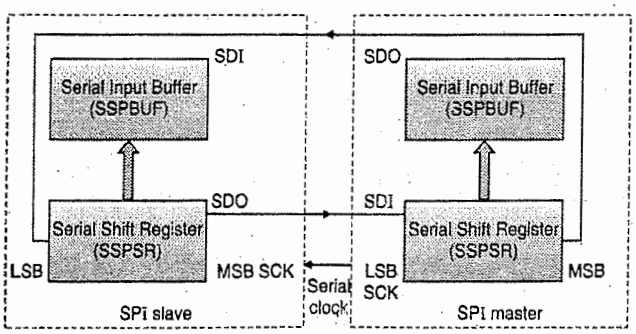


Fig. 12.9.4 : SPI master and SPI slave connection

- The three conditions for data transfer are :
  (i) Master sends data – slave sends dummy data.
  (ii) Master sends dummy data-slave sends data.
  (iii) Master sends data-slave sends data.

12.9.5 SPI Master Mode

- The master device is responsible for controlling the clock SCK. Hence, it can start the data transfer at any moment.
- Once data is written to SSPBUF register the master device starts transmitting / receiving the data.
- By programming the CKP bit the clock polarity is selected. Fig. 12.9.5 shows the SPI waveform for master mode. The MSB is first transmitted as shown in Fig. 12.9.5.
- The clock rate can be programmed to be one of the following :
  (i) T_{cy} or $\frac{F_{osc}}{4}$
  (ii) $4\,T_{cy}$ or $\frac{F_{osc}}{16}$
  (iii) $16\,T_{cy}$ or $\frac{F_{osc}}{64}$
  (iv) $\frac{\text{Timer 2 output}}{2}$
- By programming the SSPM3 : SSPM0 bits of SSPCON1 register the clock rate can be programmed.
- The highest data rate for a 40 MHz crystal is 10 Mbps.
- By setting the CKP bit of SSPCON1 register we can set the CLK signal to be idle high or idle low.
- Clock edge selection is done with the CKE bit of SSPSTAT register.
- However, actual clock edge selection is done by combining the CKE and CKP signals. This gives us 4 clock modes as shown in Fig. 12.9.5. Generally the idle low clock state is used.

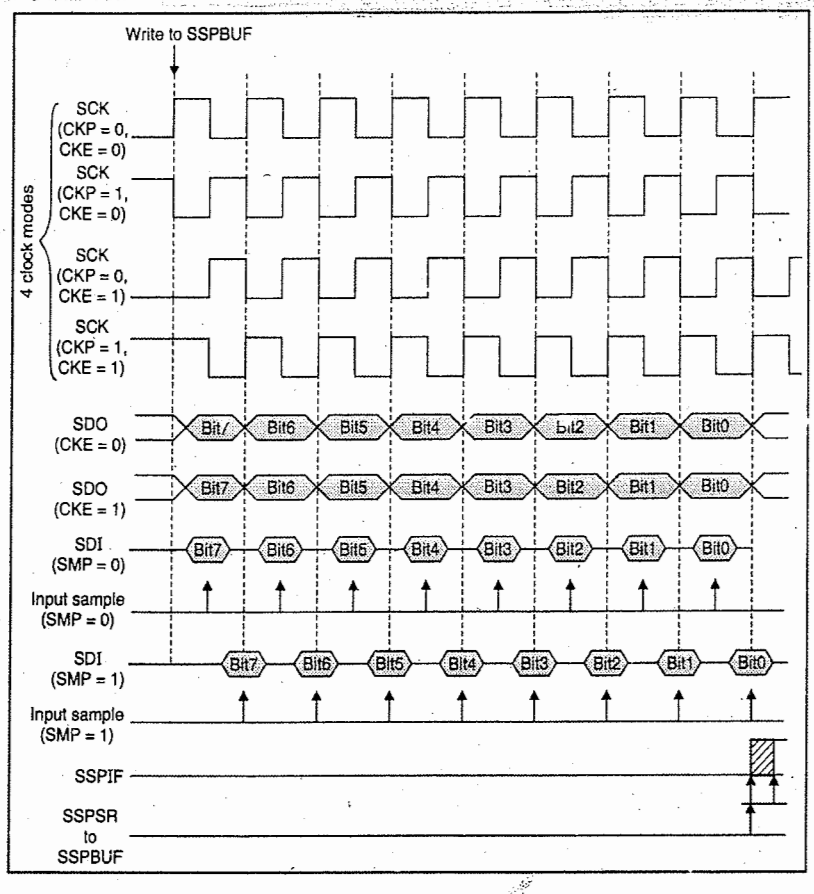


Fig. 12.9.5 : SPI Mode waveform (Master mode)

12.9.6 SPI Slave Mode

- In the SPI slave mode when external clock pulses arrive on the SCK pin the data is transmitted and received. When the last bit is latched, the SSPIF interrupt flag bit is set.
- Fig. 12.9.6 and 12.9.7 show the SPI slave mode waveforms with CKE = 0 and CKE = 1

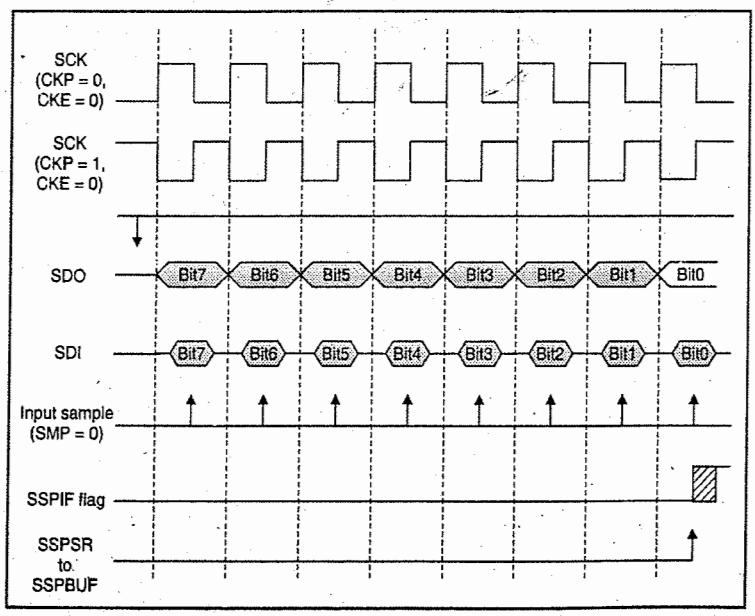


Fig. 12.9.6 : SPI slave mode (CKE = 0)

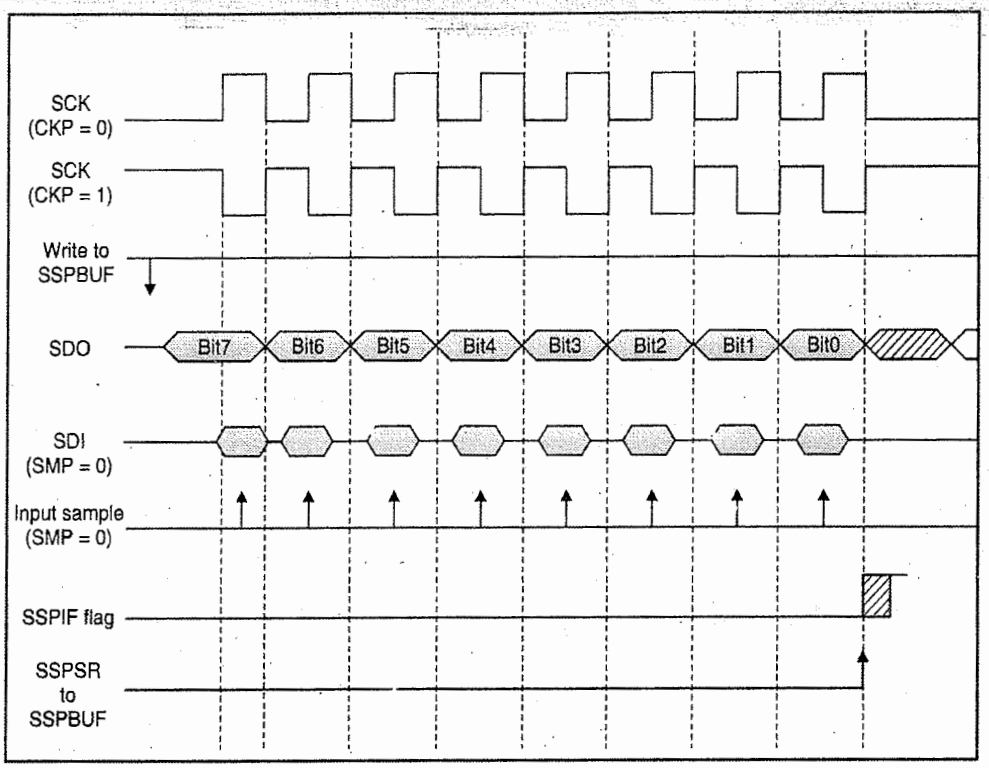


Fig. 12.9.7 : SPI slave mode (CKE = 1)

Syllabus Topic : MSSP – I2C Mode

12.10 MSSP – I2C Mode

SPPU - Dec. 14, May 15, Dec. 15, May 16, Dec. 16

University Questions

Q. Draw and explain MSSP structure of PIC18FXX.
(Dec. 2014, 8 Marks)

Q. Draw and explain I2C protocol of PIC18FXX.
(Dec. 2014, May 2015, Dec. 2015, May 2016, 8 Marks)

Q. Explain MSSP for I2C master mode.
(Dec. 2016, 6 Marks)

- The I2C mode implements all the master and slave functions. It also provides interrupts on the start and stop pins.
- I2C mode supports both 7 and 10 bit addressing. Fig. 12.10.1 shows the I2C slave mode and Fig. 12.10.2 shows the I2C Master mode block diagram.
- SCL and SDA are two pins used in MSSP : I2C mode. Both these pins need to be configured as inputs.

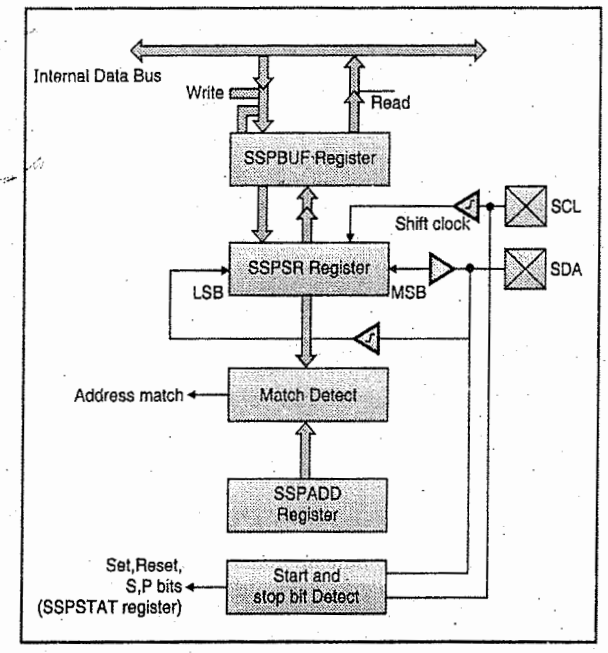


Fig. 12.10.1 : MSSP structure in I2C slave mode

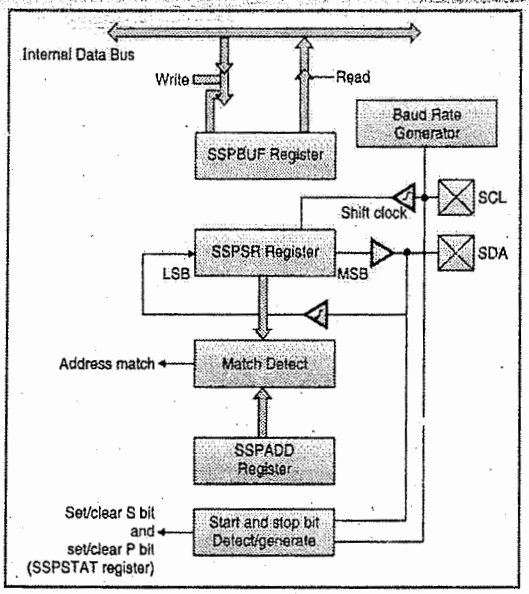


Fig. 12.10.2 : MSSP structure in I2C master mode

12.10.1 Registers for I2C Operation

For I2C operation, the MSSP supports six registers. They are :

(i) MSSP status register (SSPSTAT)
(ii) MSSP control register 1 (SSPCON1)
(iii) MSSP control register 2 (SSPCON2)
(iv) MSSP transmit/receive buffer (SSPBUF)
(v) MSSP address register (SSPADD)
(vi) SSP shift register (SSPSR)-not accessible to the programmer.

12.10.2 MSSP Transmit/Receive Buffer (SSPBUF) and MSSP Shift Register (SSPSR)

- While transferring data, the data is read/written into the SSPBUF register. The SSPSR register is not accessible to the programmer.
- The SSPSR register shifts data in or out of the device.
- Both the SSPBUF and SSPSR registers create a double buffered receiver for receive functions. Because of this before reading the first byte of data received, the reception of next data byte starts. The data byte is transferred to the SSPBUF register after the byte is received. Also the SSPIF flag bit is set.
- If before the SSPBUF register is read, another byte is received then the SSPOV bit is set indicating that the receiver has overflowed,. The data byte in SSPSR is lost.

12.10.3 MSSP Status Register (SSPSTAT)

Fig. 12.10.3 shows the SSPSTAT register. This register indicates the status of data transfer.

Bit 7							Bit 0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF

SMP : Sample bit
1 : Slew rate control disabled for standard speed mode (100 KHz and 1 MHz).
0 : Slew rate control enabled for high speed mode (400 KHz).

CKE : SMbus select bit
0 : disable SMbus specific inputs.
1 : enable SMbus specific inputs.

D/$\overline{A}$: Data address bit
In master mode : reserved
In slave mode :
1 : indicates that the last byte transmitted or received was data.
0 : indicates that the last byte transmitted or received was address.

P : Stop bit
1 : last bit detected was a stop bit
0 : stop bit was not detected.

S : Start bit
1 : a start bit has been detected
0 : start bit is not detected.

R/$\overline{W}$: Read write information
For slave mode
1 : Read
0 : Write
For master mode
1 : transmit is in progress
0 : transmit is not in progress

UA: Updates address bit (10 bit I2C mode)
1 : address needs to be updated in SSPADD register.
0 : address needs not be updated

BF: Buffer full status bit
In transmit mode
0 : SSPBUF is empty. Data transmit complete
1 : SSPBUF is full. Data transmit is in progress.
In receive mode
0 : SSPBUF is empty. Receive not complete.
1 : SSPBUF is full. Receive complete.

Fig. 12.10.3 : SSPSTAT register (I2C mode)

12.10.4 MSSP Control Register 1 (SSPCON 1)

Fig. 12.10.4 shows the MSSP control register 1 (SSPCON 1 : I2C Mode)

WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0

WCOL : Write collision detect

Master mode

0 : no collision

1 : A write to SSPBUF register was attempted while I2C conditions were not valid to start a transmission.

Slave Mode

0 : no collision.

1 : SSBUF register is written while it is transmitting the earlier word.

SSPOV : **Receive overflow bit**

0 : no overflow

1 : A byte is received when SSPBUF register holds the earlier byte

SSPEN : **Synchronous serial port enable bit**

0 : Disables serial port and configures the SCL and SDA pins as I/O port pins.

1 : Enables the serial port and configures the two pins as serial port pins.

CKP : **SCK release control bit**

In slave mode

1 : enable clock

0 : holds clock low

This bit is unused in master mode

SSPM3 – SSPM0 : Synchronous serial port mode select bits

0110 : I^2C slave mode, 7 bit address

0111 : I^2C slave mode, 10 bit address

1000 : I^2C master mode, clock = $\frac{f_{osc}}{[4 \times (SSPADD + 1)]}$

1001 : Reserved

1010 : Reserved

1011 : I^2C firmware controlled master mode

1110 : I^2C slave mode, 7 bit address with start and stop bits interrupts enabled.

1111 : I^2C slave mode, 10 bit address with start and stop bits interrupts enabled.

Fig. 12.10.4 : SSPCON1 register (I2C mode)

- The SSPCON1 register is both readable and writable. The WCOL bit is set if the PIC18 microcontroller writes to the SSPBUF register when one of following operations are done.
  (i) The ACK (acknowledge) condition is not complete.
  (ii) The SSPSR is shifting the data in/out.
  (iii) The START condition is not complete.
  (iv) The STOP condition is not complete.

12.10.5 SSPCON2 Register

It is a readable and writable register. Fig. 12.10.5 shows the SSPCON2 register.

GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN

GCEN : **General call enable bit**

0 : general call address disabled

1 : enable interrupt when general call address is received in SSPSR

ACKSTAT : **Acknowledge status bit**

0 : Acknowledge was received from slave
1 : acknowledge not received from slave
} Master transmit mode only

ACKDT : **Acknowledge data bit**

0 : acknowledge
1 : no acknowledge
} Master mode only

ACKEN : **Acknowledge sequence enable bit**

0 : acknowledge sequence idle
1 : acknowledge cleared by hardware
} Master receive mode only

Initiate acknowledge sequence on SCL and SDA pins. Transmit ACKDT data bit.

RCEN : **Receive enable bit**

0 : receive idle
1 : enables receive mode for I2C
} Master mode only

PEN : **Stop condition enable bit**

0 : Stop condition idle
1 : initiate stop condition on the SCL and SDA pins
} Master mode only

RSEN : **Repeated start condition enabled bit** (master mode only)

0 : repeated start condition idle

1 : initiate repeated start condition on SCL and SDA pins.

SEN : **Start condition enabled bit** (master mode only)

0 : start condition idle

1 : initiate start condition on SCL and SDA pins.

Fig. 12.10.5 : SSPCON2 register (I2C mode)

12.10.6 MSSP Address Register

- When the MSSP is configured in the I2C slave mode the SSPADD register holds the address of the slave device.
- If 10 bit addressing is used then the programmer needs to write the upper byte of 10 bit address into SSPADD register. After the

upper byte address matches, the lower byte is written into the SSPADD register.

- In the I2C master mode, the SSPADD register bit D0 - D6 behave as reload value of the baud rate generator.

12.10.7 I2C Master Mode

- By setting/clearing correct bits in the SSPCON1 register we can enable the I2C in master mode. The master mode can be operated either in the firmware controlled mode or interrupt enable mode by programming bits SSPM3 : SSPM0 of SSPCON1 register.
- In the firmware controlled master mode i.e. SSPM3 : SSPM0 = 1011 the programmer performs all the I2C bus operations depending on the START and STOP bit conditions.
- In the interrupt enabled master mode i.e. SSPM3 : SSPM0 = 1000 the data transfer is done by interrupt generation when START or STOP conditions are detected.
- The MSSP I2C module clears the START (S) and STOP (P) bits on reset.
- If the following conditions are detected, the **SSPIF flag bit** is set.
  (i) START condition
  (ii) Repeated START condition
  (iii) Transmitting a data byte
  (iv) Receiving a data byte
  (v) Acknowledge transmit
  (vi) STOP condition
- If the I2C is configured in the master mode the programmer can do the following :
  (i) Assert a START condition on the SCL and SDA pins.
  (ii) Assert a STOP condition on the SCL and SDA pins
  (iii) Assert a repeated START condition on the SCL and SDA pins.
  (iv) Assert an acknowledge condition when a data byte is received.
  (v) Set the I2C port for receiving the data.
  (vi) Write to the SSPBUF register for initializing data/address transfer.
- The master device generates the clock pulses, START and STOP conditions.
- A data transfer operation begins with a START condition and ends with a STOP or repeated START condition.

12.10.8 Baud Rate Generator

- Fig. 12.10.6 shows the block diagram of baud rate generator.
- The MSSP module in I2C master mode places the baud rate generator reload value in lower 7 bits of SSPADD register.
- After loading the value in baud rate generator (BRG), the baud rate generator counts down to 0. The BRG stops after a value is reloaded.
- Per instruction cycle the baud rate generator count is decremented twice.

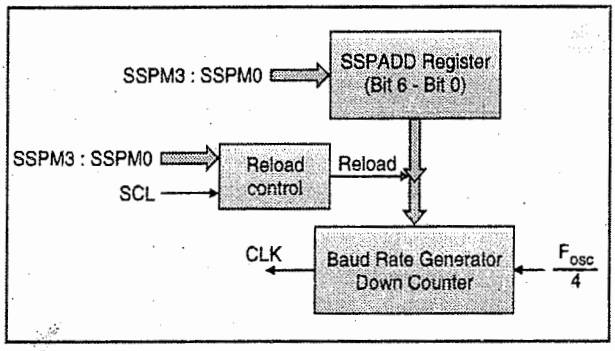


Fig. 12.10.6 : Block diagram of baud rate generator

12.10.9 I2C Master Mode Start Condition Timing

- The SEN bit in the SSPCON2 register is set to begin a start condition.
- Fig. 12.10.7 shows the start condition timing.
- The baud rate generator is loaded with count from SSPADD register if the SCL and SDA pins are sampled high. The baud rate generator will then count down.
- However, if when the baud rate generator times out and the pins are sampled then SDA pin is driven low while SCL remains high. This indicates a **Start condition** and sets the S bit in the SSPSTAT register.
- After the initiation of start condition, the baud rate generator is reloaded with the count and begins down counting. After the BRG times out the start condition enable (SEN) bit is cleared by hardware. This indicates that the start condition has been completed.

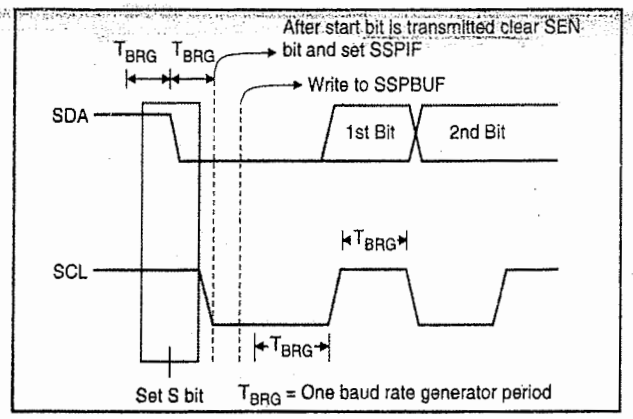


Fig. 12.10.7 : Start condition timing

12.10.10 I2C Master Mode Repeated Start Condition Timing

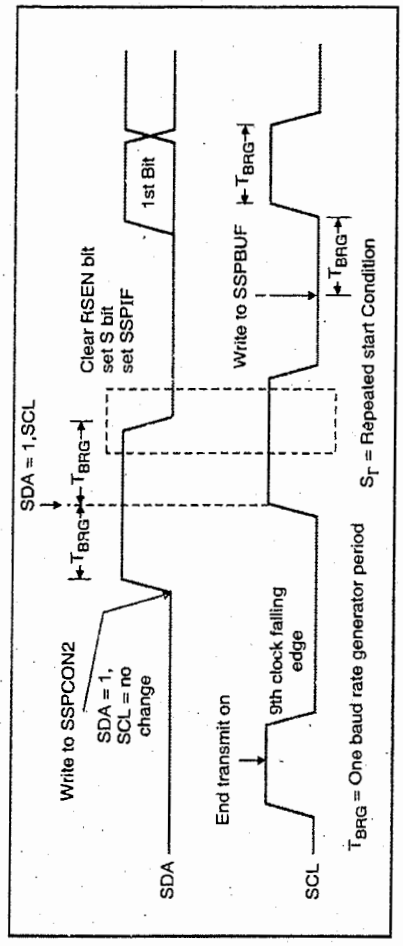


Fig. 12.10.8 : Repeated start condition timing

- When the RSEN bit is set high, a repeated start condition occurs.
- The baud rate generator is loaded with SSPADD contents. When the SCL pin is asserted low and starts down counting. If the SDA is asserted low when the baud rate generator times out, the SCL pin is pulled high. The baud rate generator is again reloaded with SSPADD and starts counting.
- Both the SCL and SDA pins must be asserted high for one T_{BRG}. Then the RSEN bit will be cleared by hardware, repeated start condition will be detected. The S bit in the SSPSTAT is set. After the baud rate generator times out the SSPIF flag will be set.
- When a repeated start condition is in progress and if the user writes to SSPBUF register then the WCOL bit is set. The SSPBUF contents remain unchanged.

12.10.11 Master Mode Transmission

- The **transmission** steps in master mode are as follows :

Step I	:	Generate the START condition by setting the SEN bit in SSPCON2 register.
Step II	:	The SSPIF flag bit is set. Before any operation preceeds the MSSP module waits for the start time needed.
Step III	:	The address that is to be transmitted is loaded in the SSPBUF register.
Step IV	:	The MSSP module shifts out the address from the SDA pin.
Step V	:	The ACK bit is shifted from the slave device and its value is written to the SSPCON2 register.
Step VI	:	At the end of 9th clock cycle, an interrupt is generated by the MSSP module. The interrupt is generated by setting the SSPIF flag bit.
Step VII	:	The programmer loads SSPBUF with eight bit data.
Step VIII	:	The MSSP module shifts out data from the SDA pin.
Step IX	:	The ACK bit is shifted from the slave device and its value is written to the SSPCON2 register.

Step X	:	At the end of 9th clock cycle, an interrupt is generated by the MSSP module. The interrupt is generated by setting the SSPIF flag bit.
Step XI	:	A stop condition is generated by setting the PEN bit in the SSPCON2 register.
Step XII	:	After the completion of stop condition, the MSSP module generates an interrupt.

12.10.12 Master Mode Reception

- If the RCEN (receive enable bit) in the SSPCON2 register is set the I2C master mode reception is enabled.
- The steps for reception in master mode are as follows :

Step I	:	The baud rate generator starts counting after RCEN bit in SSPCON2 register is enabled.
Step II	:	On every rollover of baud rate generator, the SCL pin changes its status. Shift data into the SSPSR register.
Step III	:	On the 9th clock edge, load SSPBUF with the contents of SSPSR. The receive enable flag is cleared by hardware.
Step IV	:	Set the BF flag, SSPIF flag.
Step V	:	SCL pin is asserted low. The baud rate generator stops down counting. MSSP enters idle state.
Step VI	:	The BF flag is cleared, when contents of SSPBUF register are read.
Step VII	:	By setting ACKEN bit in SSPCON2 send an acknowledge bit to indicate end of reception.

12.10.13 Stop Condition Timing :

Fig. 12.10.9 shows the stop condition timing.

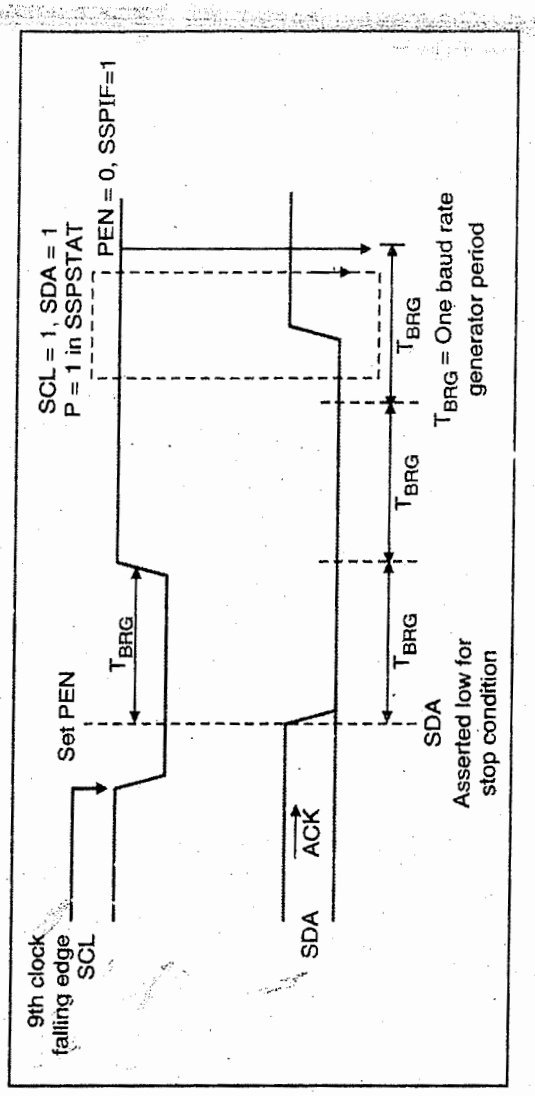


Fig. 12.10.9 : Stop condition timing

- By setting the PEN bit in the SSPCON2 register we can initiate the STOP condition at the end of data transmission or reception.
- On the falling edge of 9th clock pulse, the SCL line is pulled low. The master device then pulls the SDA line low.
- When both the SCL and SDA are low, the baud rate generator is reloaded is counts down to 0.
- The SCL pin is pulled high when the baud rate generator times out. After one T_{BRG} SDA is also pulled high. Now the P bit is set to indicate that a stop bit is detected. After that the PEN bit is cleared. This sets the SSPIF flag.

12.10.14 I2C Slave Mode

- Both the SCL and SDA pins are inputs in the I2C slave mode.
- On the address match the I2C slave mode generates an interrupt.
- An acknowledge pulse is generated whenever an address is matched. The SSPBUF registers is loaded with the contents of SSPSR. If any one of these conditions are observed the acknowledge pulse is generated.
  (i) Before the transfer was received the overflow bit SSPOV is set.
  (ii) Before the transfer was received the buffer full BF bit is set. Both these conditions will set the SSPIF. Also SSPBUF will not be loaded with the contents of SSPSR register. We need to clear the SSPOV bit by programming.

12.10.15 Slave Addressing

- After the MSSP module is configured in the slave mode, it has to wait for a start condition to take place.
- After the start condition, a byte of data is shifted to the SSPSR register.
- The rising edge of SCL is used to sample all the bits.
- On the falling edge of eight clock pulse the SSPSR register value is compared with the SSPADD register value. If both these addresses match then the SSPOV and BF bits are cleared and following steps take place.

Step I	:	On the falling edge of eight clock pulse SSPBUF register is loaded with SSPSR register
Step II	:	On the falling edge of eight clock pulse, BF is set.
Step III	:	An acknowledge $\overline{ACK}$ pulse is generated.
Step IV	:	On the falling edge of ninth clock pulse the SSPIF flag is set.

- If 10 bit addressing is used then the slave is required to receive two address bytes. The first five MSB bits indicate whether the address is a 10 bit address. The steps for a 10 bit address are as follows :

Step I	:	Receive the first byte of address.	
Step II	:	Load SSPADD register with low byte of address.	
Step III	:	Read the SSPBUF register. Clear the SSPIF flag.	
Step IV	:	The low byte of address is received.	
Step V	:	Load the SSPADD register with high byte of address.	
Step VI	:	Read the SSPBUF register. Clear SSPIF flag.	
Step VII	:	Receive repeated start condition.	For slave transmitter
Step VIII	:	Receive high byte of address.	For slave transmitter
Step IX	:	Read SSPBUF register. Clear SSPIF flag.	For slave transmitter

12.10.16 Reception in Slave Mode

- The MSSP I2C module receives data when the R/$\overline{W}$ is low and an address match occurs. This loaded SSPBUF register with the address received. To acknowledge this the SDA line is pulled low.
- An acknowledge pulse will not be given, if there is an overflow in the address byte. The overflow will set the BF flag or SSPOV flag.
- For every data byte received an interrupt is generated by setting the SSPIF flag. The status of the received byte can be found out by SSPSTAT register.

12.10.17 Transmission in the I2C Slave Mode

- If an address match occurs and if R/$\overline{W}$ of the incoming address byte is set then the R/$\overline{W}$ of SSPSTAT register will be set. The SSPBUF register is loaded with this received address.
- On the 9th clock pulse an acknowledge will be sent and the SCL line is pulled low.
- SSPBUF register is then loaded with the data to be transmitted. The CKP bit must be set for releasing the SCL signal. Now the master device can shift data out on the falling edge of SCL to the SSPSR register.

- On the rising edge 9th SCL pulse the acknowledge pulse from master-receiver is latched. The data transfer is complete if the SDA line is high.
- If the slave latches the not acknowledge, the slave logic will be reset. It waits for another start condition to commence.
- If the master device acknowledges, the slave device must load the next data byte into SSPBUF register for transmission.
- The CKP pin must be set by the slave for releasing the SCL signal.
- For each data byte that is transmitted an interrupt is generated by setting the SSPIF flag on falling edge of 9th clock pulse.

12.10.18 Multi-Master Mode

- In multi-master mode an interrupt is generated when the start and stop condition is detected. By this we can know when the I2C bus is free and when it is busy.
- When the MSSP module is disabled or on reset the start and stop bits are cleared.
- When the P bit in SSPSTAT register is set, we can access control over the I2C bus.
- The SDA line must be monitored in the multimaster mode to check whether is signal level as expected. i.e. arbitration is seen.
- A master device can lose bus arbitration in cases of
  (i) Data transfer
  (ii) Address transfer
  (iii) Start condition
  (iv) Acknowledge condition
  (v) Repeated start condition
- If master loses arbitration in I2C bus, the bus collision flag BCLIF is set. In a bus collision a 0 is detected on the SDA line, even if master sends a 1. This sets the BCLIF flag and resets the I2C to its reset state.
- If a START, repeated START, ACK or STOP conditions are in progress when bus collision took place, the condition is aborted. The SCL and SDA lines are pulled low and the control bits in SSPCON2 register are cleared.

Fig. 12.10.10 shows bus collision for transmit and acknowledge.

- If a transmission is going on and bus collision takes place, then the transmission is stopped. The SCL and SDA lines are pulled low. The BF flag is cleared. After the microcontroller executes the bus collision interrupt service routine and if then the I2C bus is free, the user can resume communication again by initiating a start condition.

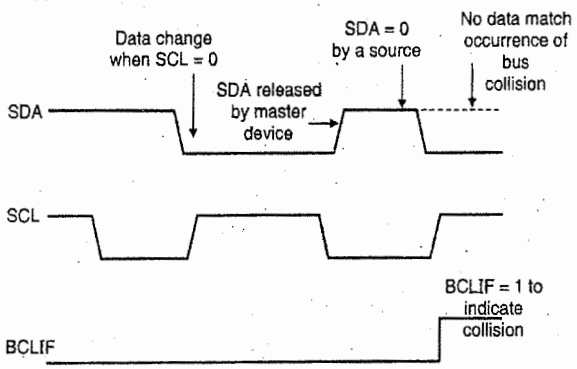


Fig 12.10.10 : Bus collision timing for transmission and acknowledge

12.11 PIC18 Connection to RS232

- We know that RS232 is not compatible with the TTL logic levels and hence line drivers and receivers are used to interface RS232 and the TTL devices. Instead of using line driver MC1488 and line driver MC1489, we can also use MAX232. The **MAX232** converts the voltage levels from RS232 to TTL voltage levels and vice versa.
- Fig. 12.11.1 shows the connection of PIC18 to the RS232.
- PIC18 has two pins of Port C RC6 (TxD) and RC7 (RxD) for the transmission and reception of serial data.

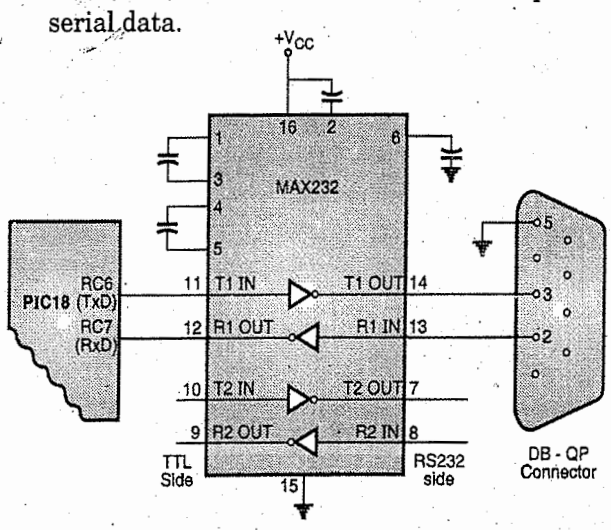
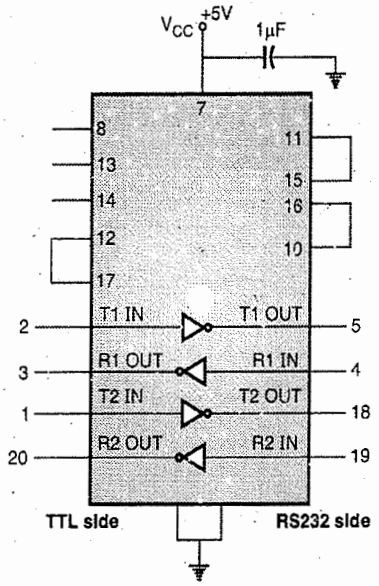
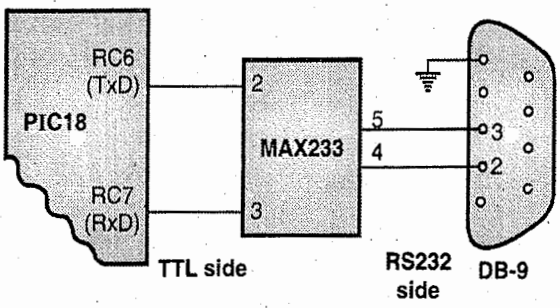


Fig. 12.11.1 : PIC18 connection to RS232 (Null modem)

- The pins RC6 and RC7 are TTL compatible and need a line driver and receiver to make them RS232 compatible.
- The MAX232 has two sets of line drivers and receivers for transmitting and receiving the data. The line drivers that are used for transmitting serial data are T1 and T2, while R1 and R2 are used for line receivers.
- MAX232 needs four capacitors that range from 1 to 22 μF.
- For saving some onboard space, some designers use MAX233. MAX233 does the same work as MAX232 but eliminates the need for capacitors. Note that MAX233 and MAX232 are not pin compatible. So we cannot use MAX233 on a MAX232 board.
- The standard interface diagram for MAX233 and its connection with PIC18 is shown in Fig. 12.11.2.



(a) MAX233



(b) PIC18 connection to MAX233 (Null modem)

Fig. 12.11.2 : MAX233 and its connection to PIC18

12.12 Interfacing PC to PIC18 using RS232 Standard

- The PCs are based on 8086, 80186, 80286, 80486, Pentium, core i3, core i5, core i7 microprocessors. Generally they have two COM ports.
- Both the COM ports have RS232 type connectors. Some PCs use the DB-25 connectors while some use the DB-9 connectors.
- The COM ports are called as COM1 and COM2.
- Nowadays one of the COM ports is been replaced by USB. So COM1 is the only serial port.
- For serial communication applications we can interface the PIC18 to PC using RS232 standard as shown in Fig. 12.12.1.

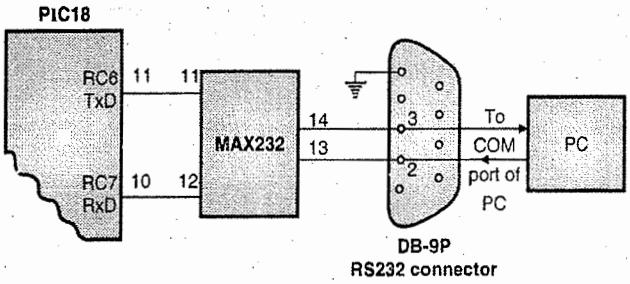


Fig. 12.12.1 : Interfacing PIC18 to PC using RS232 connector

Syllabus Topic : Interfacing Serial Port and USART (UART)

12.13 Interfacing Serial Port and USART (UART)

- The PIC18FXXX contains the Universal Synchronous Asynchronous Receiver Transmitter (USART) module.
- The USART can be operated in the following modes :

(i)	Asynchronous mode
(ii)	Synchronous mode

- The asynchronous mode is used for communicating with peripheral devices like personal computers, CRT terminals. The synchronous mode is used for communicating with peripheral devices like ADCs, serial EEPROMs.

- The registers that are responsible for serial communication and handling UART are :

(1) SPBRG register (Serial Port Baud Rate Generator)
(2) TXREG (Transfer register)
(3) RCREG (Receive register)
(4) TXSTA (Transmit Status and Control Register)
(5) RCSTA (Receive Status and Control Register)
(6) PIR1 (Peripheral Interrupt Request Register PIR1)

12.13.1 SPBRG Register

- The baud rate generator supports both the synchronous as well as asynchronous mode of serial communication.
- The microcontroller PIC18 transfers and receives the data serially at different baud rates. With the help of SPBRG the baud rate can be programmed.

Size and use : The SPBRG register is an **8 bit register.** It is used mainly for programming the baud rates.

- The baud rate can be calculated by the given formula :

$$\textbf{Baud rate} = \frac{F_{osc}}{64\,(X+1)} \quad \text{...(12.13.1)}$$

Where f_{osc} = Crystal frequency of PIC18 microcontroller

X = value loaded in SPBRG register

- The instruction cycle frequency for PIC18 microcontroller $= \frac{f_{osc}}{4} = \frac{10\text{ MHz}}{4} = 2.5\text{ MHz}$

For setting the baud rate the PIC18's UART circuit divides this instruction cycle frequency by 16.

∴ Assuming XTAL = 10 MHz

$$\text{Desired baud rate} = \frac{10\text{ MHz}}{4 \times 16\,(X+1)}$$

$$\therefore\ X = \left\{\left[\frac{156250}{\text{desired baud rate}}\right] - 1\right\}$$

Table 12.13.1 gives the different SPBRG values for different baud rates, assuming XTAL = 10 MHz, BRGH = 0.

Table 12.13.1

Baud rate	SPBRG (X Decimal value) $\left(X = \frac{156250}{\text{baud rate}} - 1\right)$	SPBRG Hex value
1200	129	81H
2400	64	40H
4800	32	20H
9600	15	FH
19200	7	7H
38400	3	3H

Ex. 12.13.1

Find the SPBRG value for following baud rates with f_{osc} = 4 MHz.

(a) 1200 (b) 2400 (c) 4800 (d) 9600 (e) 19200

Soln. :

$$f_{osc} = 4\text{ MHz}$$

$$\text{Instruction cycle frequency} = \frac{4\text{ MHz}}{4} = 1\text{ MHz}$$

It is divided by 16, before it is used by UART.

$$\therefore\ X = \frac{1\text{ MHz}}{16 \times \text{desired baud rate}} - 1$$

$$= \frac{62500}{\text{desired baud rate}} - 1$$

(a) For baud rate = 1200

The SPBRG value is, $X = \frac{62500}{1200} - 1 = (51)_{10}$

= **33 H** is loaded into SPBRG register

(b) For baud rate = 2400

$$X = \frac{62500}{2400} - 1 = (25)_{10}$$

= **19H** is loaded into SPBRG register

(c) For baud rate = 4800

$$X = \frac{62500}{4800} - 1 = (13)_{10}$$

= **0CH** is loaded into SPBRG register

(d) For baud rate = 9600

$$X = \frac{62500}{9600} - 1 = (5)_{10}$$

= **05H** is loaded into the SPBRG register

(e) For baud rate = 19200

$$X = \frac{62500}{19200} - 1 = (2)_{10}$$

= **02H** is loaded into the SPBRG register

12.13.2 TXREG Register

Size and use : It is an 8 bit special function register used for serial communication.

- Whenever a data byte is transmitted through the Tx pin, the data byte should be placed in the TXREG register.
- Once the data is written to TXREG register it is placed into **transmit shift register (TSR)**. The TSR register is not accessible to the user. It is a **parallel-in-serial-out-shift register.**
- The TSR register adds start and stop bits to the 8 bit data making it 10 bit data. This 10-bit data is serially transmitted through the TX pin of the PIC18 microcontroller.

12.13.3 RCREG Register

- When the 10 bit data is received at the RX pin, the PIC18 microcontroller reframes the data to 8 bits by eliminating the start and stop bits. The 8 bit data is placed into the RCREG register.

  e.g. : MOVFF RCREG, PORTC ; This instruction will copy the data in RCREG register to Port C.

12.13.4 TXSTA (Transmit Status and Control Register)

Size and use

It is an 8 bit register. It is used to select one of the serial communication modes i.e. synchronous / asynchronous, data frame size.

Fig. 12.13.1 shows the TXSTA register.

- We know that the TSR register transmits 10 bit data through the TX pin. When the TSR register transmits the 10th bit i.e. the stop the TRMT flag bit of TXSTA register is raised to indicate that the TSR register is empty. The TSR register can accept the next data byte. When the TSR register gets data from TXREG, the TRMT flag is cleared to indicate that the TSR register has data.

Bit 7							Bit 0
CSRC	TX9	TXEN	SYNC	0	BRGH	TRMT	TX9D

Bit 7 : CSRC Clock Source Select
It is not used in asynchronous mode ∴ D7 = 0

Bit 6 : Tx9 9 bit Transmit Enable
1 = Selects 9 bit transmission
0 = Selects 8 bit transmission

Bit 5 : TXEN Transmit Enable Bit
1 = Transmit enabled
0 = Transmit disabled

Bit 4 : SYNC : USART Mode Select
1 : Synchronous mode
0 : Asynchronous mode

Bit 3 : 0

Bit 2 : BRGH High Baud Rate Select
When SYNC = 0
BRGH = 1 High speed
BRGH = 0 Low speed
When SYNC = 1
In synchronous mode it is unused.

Bit 1 : TRMT : Transmit Shift Register Status
1 = TSR empty
0 = TSR full

Bit 0 : Tx9D 9th bit of transmit data.
This bit can be used as address / data or parity bit.

Fig. 12.13.1 : TXSTA register

12.13.5 RCSTA (Receive Status and Control Register)

Size and use : It is an 8 bit special function register used for serial communication for enabling the serial port to receive a data byte.

Fig. 12.13.2 shows the RCSTA register.

Bit 7							Bit 0
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D

Bit 7 : SPEN Serial Port Enable Bit.
1 : Serial port enabled. It makes TX and RX 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
It is not used in asynchronous mode

Bit 4 : CREN Continuous Receive Enable bit.
1 = enable continuous reception } asynchronous mode
0 = disable continuous reception }

Bit 3 : ADDEN Address Detect Enable Bit.
When SYNC = 0 and RX9 = 1
1 = enable address detection
0 = disables address detection and 9th bit can be used as parity bit

Bit 2 : FERR : Framing Error Bit.
1 = Framing error
0 = no framing error

Bit 1 : OERR : Overrun Error Bit.
1 = Overrun error
0 = no overrun error

Bit 0 : RX9D 9th bit of Received Data.
1 = 9th bit received was '1'
0 = 9th bit received was '0'

Fig. 12.13.2 : RCSTA register

12.13.6 PIR1 Register

Bit 7							Bit 0
		RCIF	TXIF				

RCIF : Receive Interrupt Flag Bit
1 = Data is received into RCREG register and when this data is read, the RCIF flag is cleared so that next data byte can be received.
0 = RCREG register is empty

TXIF : Transmit Interrupt Flag bit
1 : TXREG is empty
0 : TXREG is full

Fig. 12.13.3 : PIR1 register

- Bits D4 and D5 of PIR1 register are used by the UART. They are TXIF (Transmit Interrupt Flag) and RCIF (Receive Interrupt Flag) as shown in Fig. 12.13.3.

12.14 USART Asynchronous Mode

- In the asynchronous mode, the PIC18UART sends 8 bits or 9 bits along with start and stop bits. It uses a nonreturn-to-zero (NRZ) format.
- For achieving different baud rates there is an on-chip dedicated 8-bit baud rate generator.
- The UART is responsible for serially transmitting and receiving the data.
- Fig 12.15.1 shows block diagram the UART transmitter and Fig 12.15.2 shows the UART receiver. Both these sections are functionally independent. However they have same baud rates and employ the same data format.
- By clearing the SYNC bit in the TXSTA register we can select the UART asynchronous mode.
- Depending on the BRGH bit in the TXSTA register, the baud rate generator is responsible for producing a clock of either x16 or x64 of the bit shift rate.
- The UART does not support parity. However a 9th bit can be added as parity using software.
- Asynchronous mode is stopped during SLEEP.
- The USART Asynchronous Mode comprises of the following :
  1. Sampling circuit
  2. Baud Rate Generator
  3. Asynchronous Transmitter
  4. Asynchronous Receiver
- In Section 12.13.1 we have seen how the baud rate generator is used to different baud rates.
- The sampling circuit samples data on the RC7/RX pin three times to check the signal level present at the RX pin i.e. high or low.
- In the next sections we will see the asynchronous transmitter and receiver.

12.15 UART Asynchronous Transmitter

- Fig. 12.15.1 shows the block diagram of UART transmitter.

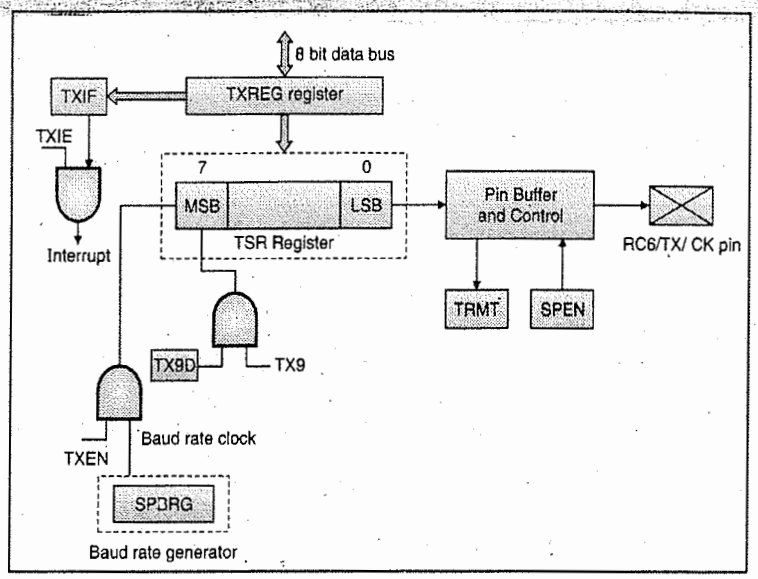


Fig. 12.15.1 : Block diagram of UART transmitter

- **TSR (Transmit Shift Register)** is the heart of the transmitter. This register is loaded after stop bit of previous data is transmitted. The TRMT flag is set to indicate TSR register is empty.
- The TSR register transmits 10 bit data (8 bit data from TXREG + 1 start bit + 1 stop bit) through the TX pin.
- After the data is transferred to the TSR register, the TXREG is made empty and TXIF flag is set.
- The TXIF can be enabled/disabled by TXIE bit. The TRMT flag indicates the status of TSR whether it is empty or full.
- By setting the TXEN bit in TXSTA register we can enable transmission.
- After the TXREG is loaded with data and desired baud rate is generated, the transmission will start. At this time the TSR register is empty. After the data is transmitted the TSR register is full and TXREG is empty.
- If the TXEN bit in the TXSTA register is cleared, then the ongoing transmission will be aborted. This clearing will reset the transmitter and the RC6/TX pin will be in high-impedance state.
- If we want 9 bit transmission, then the TX9 bit in the TXSTA register should be set indicating select 9 bit transmission. Then the 9th bit should be written in bit 0 i.e. TX9D bit of TXSTA register.
- For 9 bit transmission before writing data to TXREG register, the 9th bit must be written. Otherwise if the TSR is empty incorrect data will be transmitted.

12.15.1 Setting up Asynchronous Transmission

Following are the steps to be followed for asynchronous transmission.

Step I	:	For setting the baud rate, initialize the SPBRG register. For high speed, set the BRGH bit in TXSTA register.
Step II	:	Select asynchronous transmit mode by clearing the SYNC bit in TXSTA register and setting the SPEN in the RCSTA register.
Step III	:	Set the TXIE, PEIE, GIE bits if interrupt operation is needed.
Step IV	:	Set the TX9 bit in TXSTA register, if 9 bit transmission is needed.
Step V	:	Set the TXEN bit of TXSTA register to start transmission and set TXIF flag.
Step VI	:	Load TX9D bit i.e. 9th bit if 9 bit transmission is selected.
Step VII	:	Start transmission i.e. load the data into the TXREG register.

Fig. 12.15.2 shows the timing diagram for 8 or 9 bit transmission in asynchronous mode.

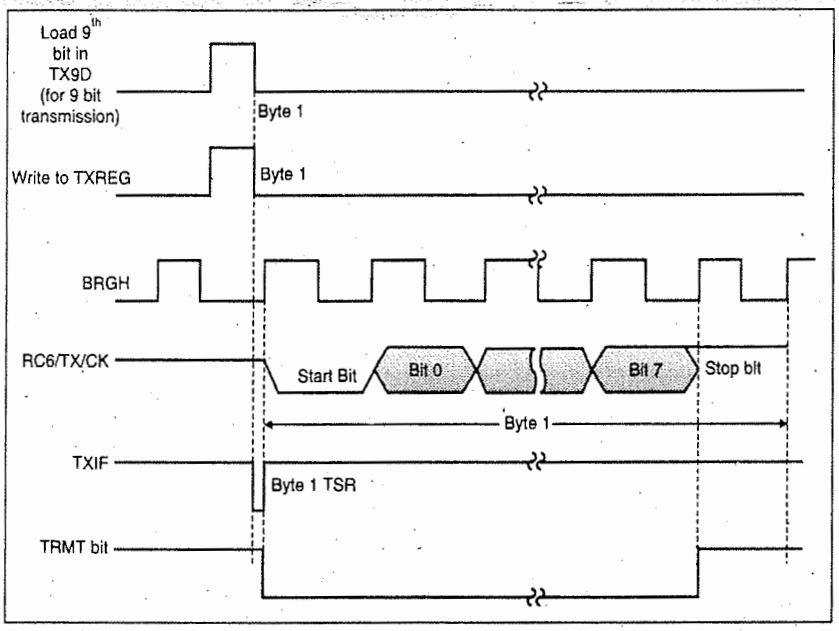


Fig. 12.15.2 : Timing diagram for 8/9 bit transmission in asynchronous mode

12.16 UART Asynchronous Receiver

Fig. 12.16.1 shows the UART receiver block diagram.

- The data is received on the RC7/RX/DT pin. This data is responsible for driving the data recovery block.
- **The RSR (Receive shift register)** is the heart of the receiver. The CREN bit of RCSTA register must be set to enable data reception.
- Once the RC7/RX pin is sampled for the STOP bit, the data that is received is sent to the RCREG register. After the data transfer is complete the RCIF is set.
- The RCIF flag can be enabled/disabled by the RCIE bit.
- After the RCREG is empty, the RCIF flag is cleared.
- With the help of FIFO, we can receive two bytes and load them in FIFO.
- If the third byte is received and the RCREG is full then OERR (overrun error) bit is set. This results in a loss in data byte in RSR register.
- The OERR bit has to be cleared by software.
- For retrieving the two bytes of data in FIFO, the RCREG can be read twice.
- If a stop bit is detected having low level, then the FERR bit (framing error bit) is set.
- For receiving the 9th bit RX9 bit of RCSTA register should be set.
- Before reading the RCREG register, the programmer must read the RCSTA register so that is not lost.
- The 9th bit is placed in RX9D bit of RCSTA register.
- The ADDEN bit can program the serial port such that if it receives a stop bit, it will interrupt the serial port provided the RX9D bit is set.
- If the receiver is configured in 9 bit mode, then only the ADDEN bit can be used. If it is set, after the stop bit no data will be loaded in the receive buffer and no interrupt will occur.

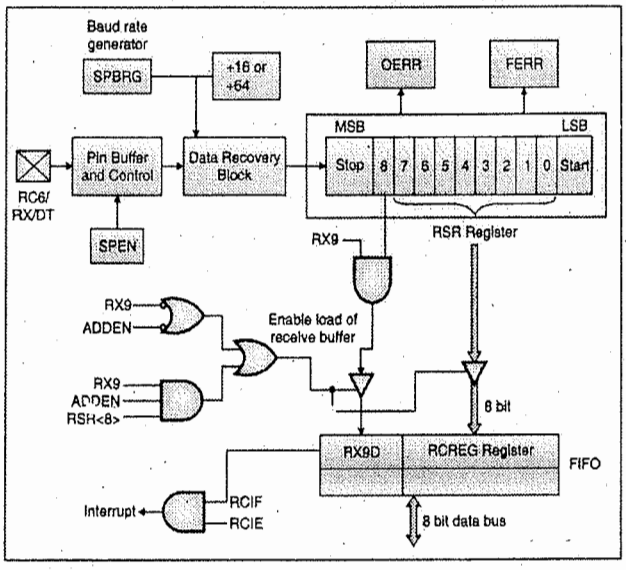


Fig. 12.16.1 : UART receiver block diagram

12.16.1 Setting up Asynchronous Reception without Address Detect Mode

Following are the steps to be followed for asynchronous mode reception without ADDRESS detect.

Step I	:	For setting the baud rate, initialize the SPBRG register. For high speed, set the BRGH bit in TXSTA register.
Step II	:	Select asynchronous receive mode by clearing the SYNC bit in TXSTA register and setting the SPEN bit in RCSTA register.
Step III	:	Set the RCIE, PEIE, GIE bits, if interrupt operation is needed.
Step IV	:	Set the RX9 bit in RCSTA register, if 9 bit reception is needed.
Step V	:	Set the CREN bit in RCSTA register to enable data reception.
Step VI	:	When the reception is complete the RCIF flag is set.
Step VII	:	If 9 bit reception is enabled, read RCSTA register and check if any error occurred while data was received.
Step VIII	:	Read the RCREG register for reading 8 bit data received.
Step IX	:	Clear the error if any by clearing the CREN bit.

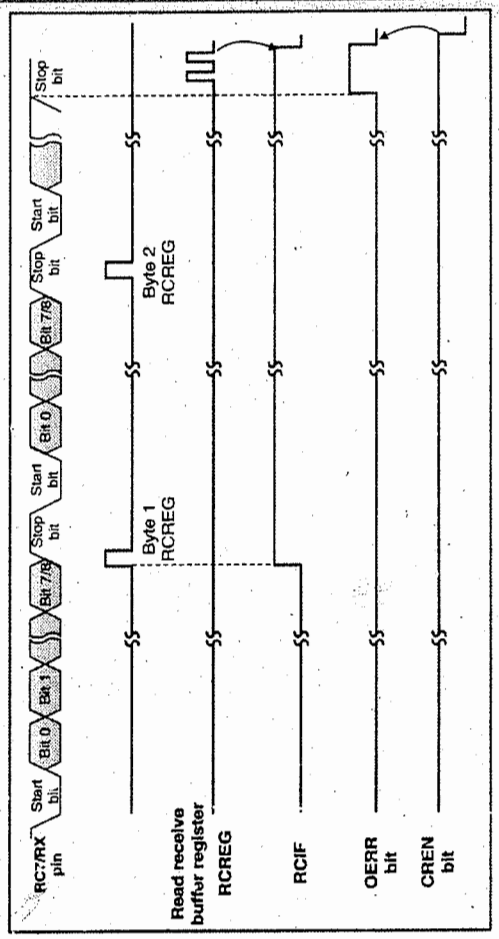


Fig. 12.16.2 : Timing diagram for 8/9 bit reception in asynchronous mode without address detect

12.16.2 Setting up Asynchronous Reception with Address Detect

Fig. 12.16.3 timing diagram for 8/9 bit reception in asynchronous mode with address detect. The data byte is followed by a address byte in the timing diagram.

The steps to be followed for asynchronous mode reception with address detect are as follows :

Step I	:	For setting the baud rate, initialize the SPBRG register. For high speed set BRGH bit in TXSTA register.
Step II	:	Select asynchronous receive mode by clearing the SYNC bit in the TXSTA register and setting the SPEN bit in the RCSTA register.
Step III	:	Set the RCIE, PEIE, GIE bits if interrupt operation is needed.
Step IV	:	Set the RX9 bit in RCSTA register, if 9 bit reception is needed.
Step V	:	To enable address detect, set the ADDEN bit.

Step VI : Set the CREN bit in RCSTA register to enable data reception.

Step VII : When the reception is complete, the RCIF flag is set. Depending on the RCIE/PEIE/GIE bits an interrupt can be generated.

Step VIII : If 9 bit reception is enabled, read the RCSTA register and check if any error occurred while data was received.

Step IX : Read the RCREG register for reading the 8 bit data received to find if the device is addressed.

Step X : Clear the error if any found, by clearing the CREN bit in RCSTA register.

Step XI : Clear the ADDEN bit if the UART is being addressed. This allows the data and address bytes to be read into the receive buffer register and the microcontroller can be interrupted.

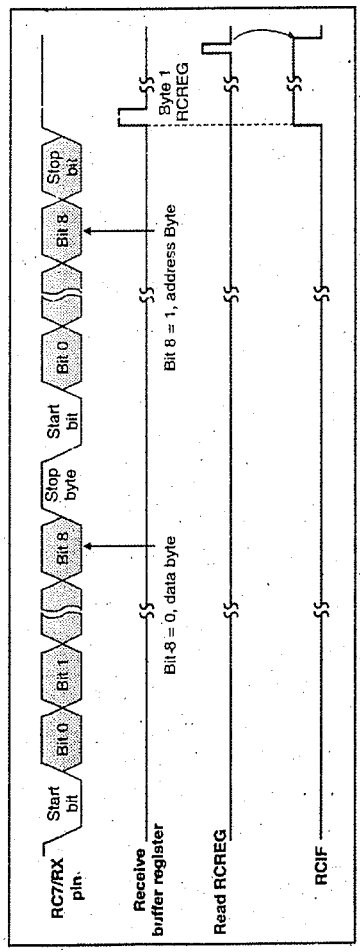


Fig. 12.16.3

12.17 Programming the PIC18 to Transfer Data Serially

Following are the steps for transferring data bytes serially :

Step I : Load 20 H into TXSTA register. 20 H indicates 8 bit data, asynchronous mode, low baud rate and transmit enabled.

Step II : Make pin RC6 i.e. TX pin an output pin.

Step III : Load SPBRG register for setting the baud rate for transferring data serially.

Step IV : Enable serial port by setting the SPEN bit in RCSTA register.

Step V : Load the character byte to be transmitted into TXREG register.

Step VI : Observe the TXIF flag to check whether UART is ready for transferring next data byte.

Step VII : For transferring next character, go to step V.

Ex. 12.17.1

Write a PIC18 program to transfer the letter 'A' serially at 9600 baud continuously. Let XTAL = 10 MHz.

Soln. :

C18 program :

```
#include <P18F458.h>
void main (void)
  {
    TXSTA = 0x20 ;          // 8bit, asynchronous mode
    SPBRG = 15 ;   // 9600 baud rate, XTAL = 10 MHz
    TXSTAbits.TXEN = 1 ;      // Enable transmit
    RCSTAbits.SPEN = 1 ;      // Enable serial port
    TRISCbits.TRISC6 = 0 ;        // TX = output
    while (1)
  {
    TXREG = 'A' ;
    while (PIR1bits.TXIF = = 0) ;
  }
}
```

Ex. 12.17.2

Write a program to transmit message "YES" serially at 9600 baud rate, 8 bit data and 1 stop bit. Do this continuously.

Soln. :

Program :

```
#include <P18F458.h>
void serialTX(unsigned char) ;
void main (void)
  {
     TXSTA = 0x20 ;          // 8 bit, asynchronous mode
     SPBRG = 15 ;            // 9600 baud rate with XTAL
                             = 10 MHz
     TRISCbits.TRISC6 = 0 ;        // TX = output
     TXSTAbits.TXEN = 1 ;          // Enable Transmit
     RCSTAbits.SPEN = 1 ;          // Enable serial port
     while (1)
  {
     serialTX ('Y') ;
     serialTX ('E') ;
     serialTX ('S') ;
  }
}
void serialTX (unsigned char x)
  {
     while (PIR1bits.TXIF = = 0) ;
     TXREG = x ;
  }
```

Ex. 12.17.3

Write a program to send the message "University of Pune" to the serial port continuously. Assume a SW is connected to pin RB2. Monitor its status and set the baud rate as follows :

SW = 0 9600 baud rate

SW = 1 38400 baud rate

Assume XTAL = 10 MHz

Soln. :

By making the BRGH = 1 in TXSTA register we can quadruple the baud rate with same value in SPBRG register.

Program :

```
#include <P18F458.h>
#define mybit PORTBbits.RB2
void main (void)
  {
     unsigned char x ;
     unsigned char mess [ ] = "University of Pune" ;
     TRISBbits.TRISB2 = 1 ;     // Make RB2 an input
     TXSTA = 0x20 ;          // 8 bit, asynchronous mode
     SPBRG = 15 ;                // 9600 baud rate
     TRISCbits.TRISC6 = 0 ;     // TX = output
     TXSTAbits.TXEN = 1 ;       // Enable Transmit
     RCSTAbits. SPEN = 1 ;      // Enable serial port
     if (mybit = = 0)
  {
     for (x = 0 ; x < 18 ; x++)
  {
     while (PIR1bits.TXIF = = 0) ;    // wait for transmit
     TXREG = mess [x] ;          // Put value in buffer
     }
  }
  else
     {
        TXSTA = TXSTA|0x04 ;    // quadruple baud
                                   rate = 38400
        for (x = 0 ; x < 18 ; x++)
     {
        while (PIRbits.TXIF = = 0) ;   //wait for transmit
        TXREG = mess [x] ;      // place value in buffer
     }
  }
  while (1) ;
}
```

Ex. 12.17.4

Write a C18 program to send different strings to the serial port. Assuming that a switch is connected to pin port B.6, monitor its status and if.

SW = 0 ; send your first name

SW = 1 ; send your last name

Assume XTAL = 10 MHz, baud rate of 9600 and 8 bit data.

Soln. : Program :

```
#include <P18F458.h>
#define mybit PORTBbits.RB6
void main (void)
  {
     unsigned char x ;
     unsigned char firstname [ ] = "URVASHI" ;
     unsigned char lastname [ ] = "SHAH" ;
     TRISBbits.TRISB6 = 1 ;          //RB6 = 1
     TXSTA = 0x20 ;
     SPBRG = 15 ;                    // 9600 baud rate
     TXSTAbits.TXEN = 1 ;            // Enable transmit
     RXSTAbits.SPEN = 1 ;            / Enable serial port
     TRISCbits.TRISC6 = 0 ;          // TX = output
     if (mybit = = 0)
     {
        for (x = 0 ; x < 7 ; x++)
                                   // If SW = 0 send first name
```

```
{
            while (PIR1bits.TXIF = = 0) ;
            TXREG = firstname [x] ;
        }
    }
    else
    {
        for (x = 0 ; x < 4 ; x++)
// if SW = 1 send last name
        {
            while (PIR1bits.TXIF = = 0) ;
            TXREG = lastname [x] ;
        }
    }
    while (1) ;
  }
```

Ex. 12.17.5

Assume a switch is connected to pin RD7. Write a program to monitor its status and send two messages to the serial port continuously as follows :

SW = 0 send "NO"

SW = 1 send "YES"

Assume XTAL = 10 MHz, set baud rate = 9600.

Soln. :

```
#include<P18F458.h>
#define SW PORTDbits.RD7
void main (void)
  {
    unsigned char x ;
    unsigned char mess1 [ ] = "NO" ;
    unsigned char mess2 [ ] = "YES" ;
    TRISDbits.TRISD7 = 1 ;          // RD7 = input
    TRISCbits.TRISC6 = 0 ;          // TX = output
    TXSTA = 0x20 ;          // asynchronous mode, 8 bit
    SPBRG = 15 ;            // baud rate = 9600
    TXSTAbits.TXEN = 1 ;        // Transmit enabled
    RCSTAbits.SPEN = 1 ;        // Enable serial port
    if (SW = = 0)               // Check switch
    {
        for (x = 0 ; x < 2 ; x++ // if SW = 0, send NO
        {
            while (PIR1bits.TXIF = = 0) ;
                                        // wait for transmit
            TXREG = mess1[x] ;    // put character in
                                        buffer
        }
    }
    else
    {
        for (x = 0 ; x < 3 ; x++) // if SW = 1, send YES
        {
            while (PIR1bits.TXIF = = 0) ;
                                        // wait for transmit
            TXREG = mess 2 [x] ;
                                        // put character in buffer
        }
    }
    while (1) ;
  }
```

12.18 Programming the PIC18 to Receive Data Serially

– Following steps need to be considered for receiving data bytes serially.

Step I	:	Load 90H into RCSTA register indicating continuous reception.
Step II	:	Load TXSTA with 00H.
Step III	:	Load SPBRG for setting the correct baud rate.
Step IV	:	Make RX pin i.e. RC = 1 input for receiving data.
Step V	:	Monitor the RCIF flag to see if data is received.
Step VI	:	If RCIF = 1, the RCREG holds a data byte and its contents are transferred or saved.
Step VII	:	Goto step V, to receive the next character.

Ex. 12.18.1

Write a PIC18 program for receiving data bytes serially and placing them on port D. Set baud rate 4800 8 bit data and 1 stop bit.

Soln. :C18 program :

```
#include <P18F458.h>
void main (void)
  {
    TRISD = 0 ;                 // Port D = output port
    RCSTA = 0x90 ;   // Enable serial port and receiver
    SPBRG = 20 ;                // Baud rate = 4800
    TRISCbits.TRISC7 = 1 ;      // RX = input
    while (1)
    {
    while (PIR1bits.RCIF = = 0) ;
    PORTD = RCREG ;             // send received data
                                   bytes to port D
    }
  }
```

Ex. 12.18.2

Write a PIC18 program to receive data serially and display it on LEDs on Port B. Set the baud rate of 1200. Assume XTAL = 10 MHz.

Soln. : Program :

```
#include <P18F458.h>
void main (void)
  {
      TRISB = 0 ;            // Port B = output port
      RCSTA = 0x90 ;         / Enable continuous reception
      SPBRG = 81 ;           / XTAL = 10 MHz,
                              baud rate = 1200
      TRISCbits.TRISC7 = 1 ;     // RC7 i.e. RX = 1
      while (1)
      {
          while (PIR1bits.RCIF = = 0) ;
          PORTB = RCREG ;        // Display received
                                    data on LEDs on port B
      }
  }
```

Ex 12.18.3 SPPU - Dec. 16, 8 Marks

Write a program to read only numbers from input UART string.

Soln. :

```
# define CR 0x00          //define carriage return character
void gets_usart (char*ptr)
{
      char i ;
      while (1)
      {
          i = gets_usart() ;              // read number
          if (i = = CR)
          *ptr = '0' ;
      }
}
```

Ex. 12.18.4 Lab assignment

Interfacing serial port with PC both side communication.

Soln. :

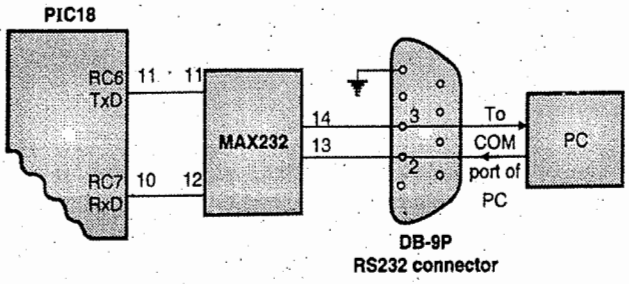


Fig. P. 12.18.4

Program :

```
# include <P18F458.h>
void my_isr (void);
void transmit_isr (void);
void receive_isr (void);
#pragma interrupt my_isr
void  my_isr (void)
  {
  if (PIR1bits.TXIF==1)      // if transmit interrupt,
                                execute transmit_isr
  transmit_isr( );
  if (PIR1bits.RCIF ==1)     // if receive interrupt,
                                execute receive_isr.
  receive _isr( );
  }
  #pragma code hi_priori = 0x08      // high priority
                                        interrupt.
  void hi_priori (void)
  {
  _asm
  GOTO my_isr
  _endasm
  }
#pragma code
void main(void)
  {
      TRISD = 0xFF;              //port D = Input.
      TRISB = 0x00;              //Port B = Output.
      TRISCbits.TRISC6 = 0; //Make transmit pin = output
      TRISCbits.TRISC7=1;  //Make receive pin input
      TXSTA=0x20;                //Enable transmit
      SPBRG = 15 ;               //Set baud rate = 9600
      RCSTAbits.CREN = 1;        //Enable receive
      RCSTAbits.SPEN =1;         //Enable serial port.
      TXSTAbits.TXEN=1;          //Enable transmit
      PIE1bits.TXIE =1;         //Enable transmit interrupt.
      PIE1bits.RCIE=1;          //Enable receive interrupt
      INTCONbits.PEIE = 1;       //Enable all peripheral
                                    interrupts
      INTCONbits.GIE = 1;        //Globally enable all
                                    interrupts
      while (1);
  }
void transmit_isr(void)
  {
      TXREG = PORTD;        //Send Port D value serially
  }
void receive_isr (void)
  {
      PORTB=RCREG;   // Receive serial data at Port B
  }
```

12.19 Quadrupling the Baud Rate in PIC18

- There are two methods to increase the baud rate in PIC18. They are :
  - (i) To use a crystal of high frequency. However this method is not suitable because the crystal frequency is fixed.
  - (ii) Modify the BRGH bit in the TXSTA register keeping the crystal frequency unmodified. This method is used for quadrupling the baud rate.

12.19.1 Baud Rates with BRGH = 0

For BRGH = 0

The instruction cycle frequency for $= \frac{f_{osc}}{4}$

$= \frac{10 \text{ MHz}}{4} = 2.5$ MHz PIC18 microcontroller.

For setting the baud rate the PIC18's UART circuit divides this instruction cycle frequency by 16.

$$\therefore \text{ Desired baud rate } = \frac{10 \text{ MHz}}{4 \times 16 (X + 1)}$$

$$\therefore \text{ Desired baud rate } = \frac{156250}{(X + 1)}$$

156250 Hz is the frequency used by UART to set baud rate when BRGH = 0.

Table 12.19.1 gives the SPBRG for different baud rates.

12.19.2 Baud Rates with BRGH = 1

For BRGH = 1

The instruction cycle frequency $= \frac{f_{osc}}{4} = \frac{10 \text{ MHz}}{4}$

= 2.5 MHz for PIC18 microcontroller.

For setting the baud rate PIC18's UART circuit divides this instruction cycle frequency by 4.

$$\therefore \text{ Desired baud rate } = \frac{2.5 \text{ MHz}}{4 (X + 1)} = \frac{625000}{(X + 1)}$$

625000 Hz is the frequency used by UART for setting the baud rate when BRGH = 1.

Table 12.19.1 gives the SPBRG values for different baud rates.

Table 12.19.1 : SPBRG values for different baud rates (XTAL = 10 MHz)

Baud Rate	BRGH = 0 SPBRG (Decimal)	SPBRG = $\frac{156250}{\text{baud rate}} - 1$ SPBRG (HEX)	BRGH = 1 SPBRG (Decimal)	SPBRG = $\frac{625000}{\text{baud rate}} - 1$ SPBRG (HEX)
1200	129	81H	519	207H
2400	64	40H	259	103H
4800	32	20H	129	81H
9600	15	0FH	64	40H
19200	7	07H	32	20H
38400	3	03H	15	0FH
57600	2	02H	10	0AH

Table 12.19.2 shows how the baud rate are quadrupled if BRGH = 1. While SPBRG value remains same. XTAL = 10 MHz from Table 12.19.1.

Table 12.19.2

SPBRG value Decimal	BRGH = 0	BRGH = 1
15	9600	38400
32	4800	19200
64	2400	9600
129	1200	4800

Ex. 12.19.1

Write a program for PIC18 microcontroller to transfer letter 'H' serially at 57600 baud rate continuously. Assume XTAL = 10 MHz. Use BRGH = 1.

Soln. :

Program :

```
#include <P18F458.h>
void main (void)
  {
    TXSTA = 0x20 ;          // asynchronous mode, 8 bit
    SPBRG = 0x10 ;          // baud rate = 57600
    TRISCbits.TRISC6 = 0 ;  // TX = output pin
    TXSTAbits.TXEN = 1 ;    // Enable Transmit
    RCSTAbits.SPEN = 1 ;    // Enable serial port
    while (1)
    {
      while (PIR1bits.TXIF = = 0) ;
      TXREG = 'H' ;         // Send letter 'H'
    }
  }
```

Ex. 12.19.2

Write a program to send message "Welcome to Pune" to the serial port continuously. Assume a SW is connected to pin RB0. Monitor its status and set the baud rate as follows :

SW = 0 1200 baud rate

SW = 1, 4800 baud rate

Assume XTAL = 10 MHz

Soln. :

Program :

```
#include <P18F458.h>
#define SW PORTBbits. RB0
void main (void)
  {
     unsigned char x ;
     unsigned char mess [ ] = "Welcome to Pune" ;
     TRISBbits.TRISB0 = 1 ;          // RB0 = input
     TRISCbits.TRISC6 = 0 ;          //TX = output
     TXSTA = 0x20;           // 8 bit, asynchronous mode
     SPBRG = 129 ;           // baud rate = 1200
     TXSTAbits.TXEN = 1 ;       // Transmit enable
     RCSTAbits.SPEN = 1 ;       //Enable serial port
     if (SW = = 0)           // if switch = 0 send message
                             at baud rate 1200
     {
        for (x = 0; x < 15 ; x++)
        {
           while (PIR1bits.TXIF = = 0) ;
                                   // wait for transmit
           TXREG = mess [x] ;
                                   // put the value in TXREG
        }
     }
  else
  {
     TXSTA = TXSTA|0x4 ;          // if switch = 1
     for (x = 0 ; x < 15 ; x + + )
                          // Quadruple baud rate = 4800
                           and send message
  {
     while (PIR1bits.TXIF = = 0) ;
     TXREG = mess [x] ;
     }
  }
  while (1) ;
}
```

Ex. 12.19.3

Write a program to send two messages "Low speed" and "High speed" to the serial port. Assume that a switch is connected to pin RB5, monitor its status and set the baud rate as follows :

SW = 0 9600 baud rate

SW = 1 38400 baud rate

Assume XTAL = 10 MHz.

Soln. : Program :

```
#include <P18F458.h>
#define SW PORTBbits.RB5
void main (void)
  {
     unsigned char x ;
     unsigned char mess1 [ ] = "Low speed" ;
     unsigned char mess2 [ ] = "High speed" ;
     TRISBbits.TRISB5 = 1 ;          //RB5 = input
     TRISCbits.TRISC6 = 0 ;          // TX = output
     TXSTA = 0x20 ;          // asynchronous mode, 8 bit
     SPBRG = 15 ; // baud rate = 9600, XTAL = 10 MHz
     TXSTAbits.TXEN = 1 ;          // Enable transmit
     RCSTAbits.SPEN = 1 ;          //Enable serial port
     if (SW = = 0)
     {
        for (x = 0 ; x < 9 ; x + +)
        {
           while (PIR1bits.TXIF = = 0) ;
           TXREG = mess1 [x] ;
                                   // send message "low speed"
        }
     }
  else
  {
     TXSTA = TXSTA|0x4 ;      // Quadruple baud rate to
                              38400 by making BRGH = 1
     for (x = 0 ; x < 10 ; x + +)
     {
        while (PIR1bits.TXIF = = 0) ;
        TXREG = mess 2 [x] ;
                              // send message "High speed"
     }
  }
     while (1) ;
  }
```

12.20 Baud Rate Error Calculation

- While calculating the baud rates we only consider the integer part, the decimal part is not considered. This introduces an error in the baud rate.
- For finding this error we use the formula :

$$\text{error} = \frac{\text{calculated value of SPBRG} - \text{Integer part}}{\text{Integer part}}$$

e.g. : for BRGH = 0, XTAL = 10 MHz, baud rate = 2400

$$\text{SPBRG} = \frac{156250}{2400} - 1 = 65.1 - 1 = 64.1 = 64$$

$$\therefore \text{ error} = \frac{64.1 - 64}{65} = 0.15\%$$

The baud rate can also be calculated as,

$$\text{error} = \frac{\text{calculated baud rate} - \text{desired baud rate}}{\text{desired baud rate}}$$

e.g. : For BRGH = 0, XTAL = 10 MHz,
baud rate = 2400

$$\text{SPBRG value} = \frac{156250}{2400} - 1 = 65.1 - 1$$

$$= 64.1 = 64$$

$$\therefore \text{ Calculated baud rate} = \frac{156250}{(64 + 1)} = 2403$$

$$\therefore \text{ error} = \frac{2403 - 2400}{2400} \times 100\%$$

$$\text{error} = 0.125\%$$

Table 12.20.1 lists the SPBRG values and % error for different baud rate, assuming XTAL = 10 MHz.

Table 12.20.1

Baud rate	BRGH = 0		BRGH = 1	
	SPBRG	%Error	SPBRG	%Error
1200	129	0.15%	519	0.16%
2400	64	0.15%	259	0.125%
4800	32	1.3%	129	0.15%
9600	15	1.7%	64	0.15%
19200	7	1.7%	32	1.3%
38400	3	1.5%	15	1.7%

□□□

UNIT - VI

CHAPTER 13

Interfacing ADC-DAC

13.1 Interfacing DAC, ADC and Sensors

- Most of the physical quantities such as temperature, pressure, displacement, vibrations etc. are available in analog form. These quantities are represented accurately in analog form but it is difficult to process, store or transmit the analog signal because noise easily introduces error.
- Hence to reduce these errors it is always better to express these physical quantities in the digital form.
- The digital representation of a signal makes storage possible, processing simpler and transmission easier.
- Therefore A to D conversion is necessary. Now once the processing, transmission etc. is done the signal should be brought back to its analog form, for which the D to A conversion is essential.
- Both ADC and DAC circuits are called as data converters and they are available in the IC form.
- Fig. 13.1.1 shows a A/D and D/A converter application.

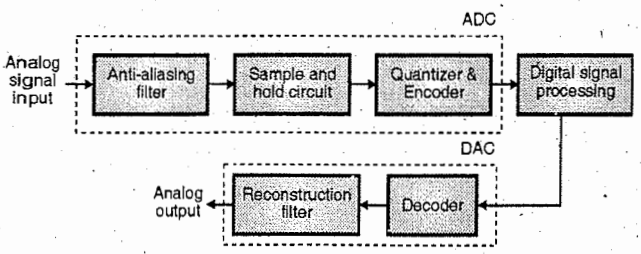


Fig. 13.1.1 : A/D and D/A converter application

As shown in the Fig. 13.1.1 the A/D conversion involves band limiting the signal, sampling the signal, quantizing it and encoding it into a suitable digital format before transmission. Once the signal is transmitted, it is received and converted back to analog form by the decoder and the reconstruction filter.

13.2 DAC 0808 and its Interfacing with PIC Microcontroller

The DAC 0808 is an 8-bit current output monolithic DAC manufactured by the National semiconductor corporation. It is a 16 - pin IC available in dual in line DIP plastic package. The analog output is available in the form of current I_o. That means I_o is proportional to the 8-bit digital input. The important features of DAC 0808 are as follows :

13.2.1 Features of DAC 0808

Fast settling time	150 nsec. Typically
Power supply voltage range	± 4.5 mW at ± 5V
Low power consumption.	33 mW at ± 5V.
High speed multiplying input slew rate	8 mA / μ sec.
Interfaces directly with TTL, DTL and CMOS logic levels.	

13.2.2 Pin Configuration and Functional Block Diagram

- The pin configuration and functional block diagram of DAC 0808 are as shown in Fig. 13.2.1(a) and (b) respectively.
- The internal block diagram shows that DAC 0808 consists of R-2R ladder along with current switches and reference current amplifier.
- A_1 to A_8 are the 8-digital input lines with A_1 as the most significant bit and A_8 as the least significant bit.
- The analog output is available in the form of current I_o, therefore we need to use an external current to voltage converter if the analog output in the form of voltage is required.
- DAC 0808 requires a dual polarity (±) supply voltage, typically ± 15V, for its operation. The reference voltage can be either positive or negative.
- An external reference voltage should be applied to either V_{REF} (+) or V_{REF} (–) depending on the polarity of reference voltage.

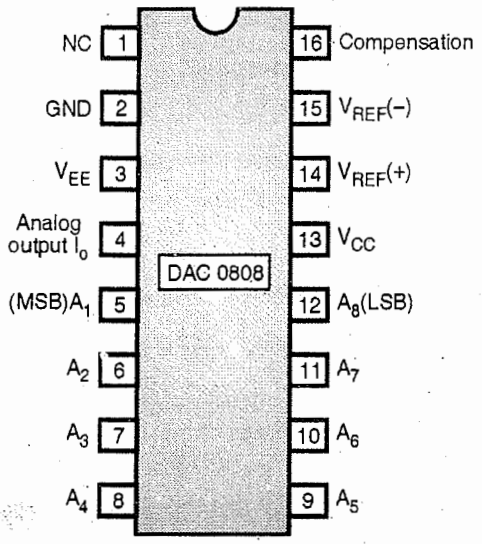


Fig. 13.2.1(a) : Pin configuration of DAC 0808

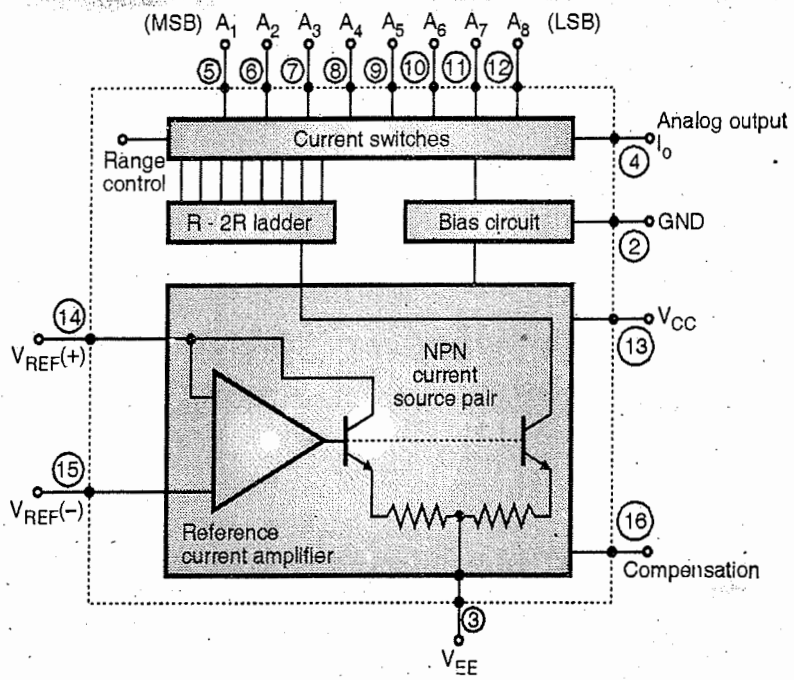


Fig. 13.2.1(b) : Functional block diagram of DAC 0808

13.2.3 Interfacing DAC to PIC18

Fig. 13.2.2 shows the interfacing of DAC 0808 to PIC18.

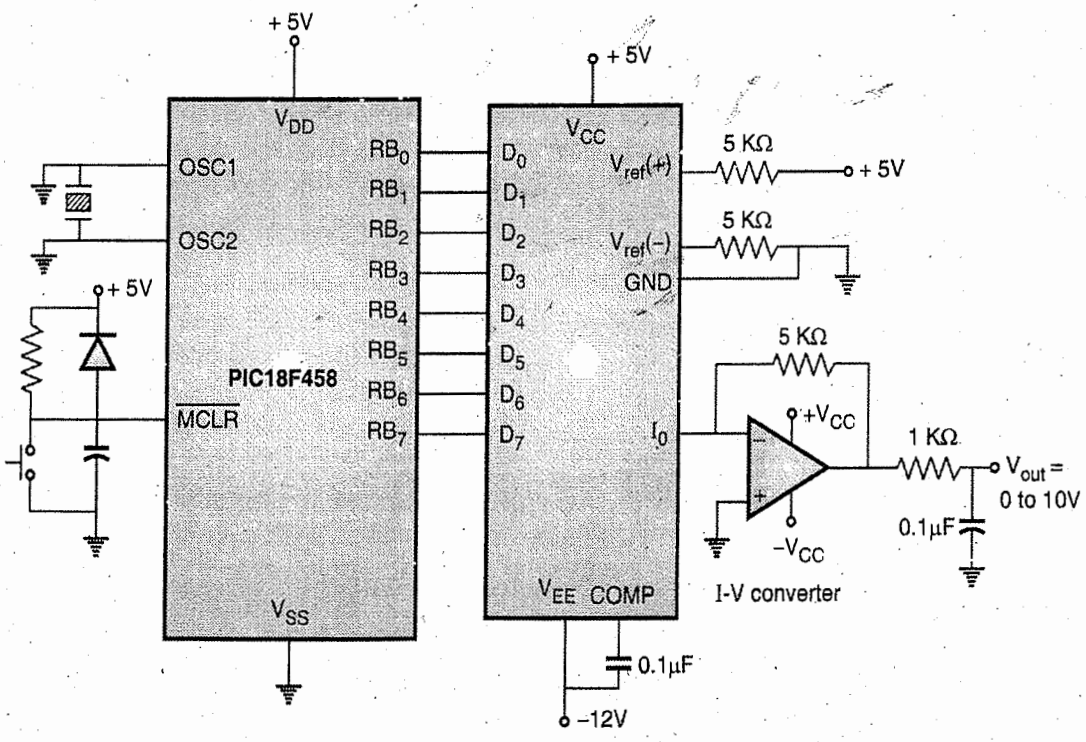


Fig. 13.2.2 : Interfacing of DAC 0808 to PIC18F458

- The output of DAC is a current which is converted into voltage using opamp based current-to-voltage (I-V) converter.
- The analog output current I_{out} of the DAC depends on the I_{ref} flowing into the V_{ref} terminal and the status of the $D_0 - D_7$ bits. It is expressed as,

$$I_{out} = I_{ref}\left(\frac{D_7}{2}+\frac{D_6}{4}+\frac{D_5}{8}+\frac{D_4}{16}+\frac{D_3}{32}+\frac{D_2}{64}+\frac{D_1}{128}+\frac{D_0}{256}\right)$$

where, D_7 = MSB

I_{ref} depends on V_{ref} voltage and the resistors 1 KΩ and 1.5 KΩ connected.

$$I_{ref} = \frac{V_{ref}}{1\text{ K} + 1.5\text{ K}} = \frac{5\text{ V}}{2.5\text{ K}\Omega}$$

$$I_{ref} = 2\text{ mA}$$

(a) If $D_0 - D_7 =$ FFH then

$$I_{out} = 2\text{ mA} \times \left(\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\frac{1}{16}+\frac{1}{32}+\frac{1}{64}+\frac{1}{128}+\frac{1}{256}\right)$$

$$I_{out} = 2\text{ mA} \times \frac{255}{256} = 1.99\text{ mA}$$

$$V_{out} = 1.99\text{ mA} \times 5\text{ K}\Omega = 9.96\text{ V}$$

(b) If $D_0 - D_7 = 80$ H then

$$I_{out} = 2\text{ mA} \times \left(\frac{1}{2}+\frac{0}{4}+\frac{0}{8}+\frac{0}{16}+\frac{0}{32}+\frac{0}{64}+\frac{0}{128}+\frac{0}{256}\right)$$

$$I_{out} = 1\text{ mA}$$

$$V_{out} = 1\text{ mA} \times 5\text{ K}\Omega = 5\text{ V}$$

(c) If $D_7 - D_0 = 00$ H then $I_{out} = 0$, $V_{out} = 0$V

DAC is commonly used in waveform generation as shown in examples.

Ex. 13.2.1

Design a PIC18 based system to interface DAC. Write the C and an assembly language programs to generate a sine wave.

Soln. :

Fig. 13.2.2 shows the interfacing diagram for interfacing a PIC18 based system to DAC 0808.

We need to calculate the values for sinusoidal waveform between 00H and FFH. This is done by the following table. Here we have divided the entire cycle of 360° into 48 parts each of incremental 7.5°. Hence the angles are 0°, 7.5 °, 15°, 22.5°....

We cannot have negative voltage in the output of our microcontroller. Hence for Sin 0, we need the output to be in centre i.e. 2.5V, treating it as x-axis. Hence we add 2.5V to every calculation as seen in the fourth row of the table below. Then we find the corresponding value for each voltage by multiplying it with the maximum count and dividing it by maximum voltage as given in the fifth column of the table.

Note : If you still reduce the step size from 7.5°, you may get a better waveform. For the program in assembly, we have taken the angles at an interval of 30°. This gives less accurate output.

Calculation of array elements

Sr. No.	Angle in degrees (θ)	sin (θ)	Count = 2.5 +(2.5 * sin θ)	count*256/5
0	0	0	2.5	128
1	7.5	0.130578428	2.826446071	145
2	15	0.258920827	3.147302068	161
3	22.5	0.382829457	3.457073643	177
4	30	0.500182502	3.750456255	192
5	37.5	0.608970405	4.022426013	206
6	45	0.707330278	4.268325695	219
7	52.5	0.793577803	4.483944508	230
8	60	0.866236075	4.665590188	239
9	67.5	0.924060891	4.810152227	246
10	75	0.966062056	4.915155141	252
11	82.5	0.991520342	4.978800856	255
12	90	0.9999998	4.9999995	256
13	97.5	0.991355227	4.978388068	255
14	105	0.965734654	4.914336634	252
15	112.5	0.923576807	4.808942018	246
16	120	0.8656036	4.664008999	239
17	127.5	0.792807767	4.482019417	229

Sr. No.	Angle in degrees (θ)	sin (θ)	Count = 2.5 +(2.5 * sin θ)	count*256/5
18	135	0.706435867	4.266089666	218
19	142.5	0.607966935	4.019917336	206
20	150	0.499087156	3.74771789	192
21	157.5	0.381660992	3.45415248	177
22	165	0.257699252	3.144248131	161
23	172.5	0.129324662	2.823311654	145
24	180	0.001264489	2.496838778	128
25	187.5	0.131831986	2.170420034	111
26	195	0.260141988	1.849645029	95
27	202.5	-0.38399731	1.540006725	79
28	210	0.501277049	1.246807379	64
29	217.5	0.609972902	0.975067745	50
30	225	0.708223559	0.729441104	37
31	232.5	0.794346571	0.514133573	26
32	240	0.866867165	0.332832087	17
33	247.5	0.924543497	0.188641258	10
34	255	0.966387914	0.084030214	4
35	262.5	0.991683872	0.02079032	1
36	270	0.999998201	4.9999	256
37	277.5	0.991188527	0.022028682	1
38	285	0.965405707	0.086485732	4
39	292.5	0.923091247	0.192271883	10
40	300	-0.86496974	0.337575649	17
41	307.5	0.792036463	0.519908843	27
42	315	0.705540326	0.736149186	38
43	322.5	0.606962492	0.98259377	50
44	330	0.497991012	1.25502247	64
45	337.5	0.380491917	1.548770208	79
46	345	0.256477265	1.858806837	95
47	352.5	0.128070688	2.17982328	112
48	360	0.002528976	2.50632244	128

Algorithm

Main Program

Step I : Initialize data according to the above table

Step II : Issue the data one by one from the array.

Step III : Repeat the above step continuously to get a continuous waveform

C Program

```
# include <P18F458.h>
    char array [ ] = {128, 145, 161, 177, 192, 206, 219, 230,
239, 246, 252, 254, 255, 254, 252, 246, 239, 229, 219, 206,
192, 177, 161, 145, 128, 111, 95, 79, 64, 50, 37, 26, 17, 10,
4, 1, 0, 1, 4, 10, 17, 26, 37, 50, 64, 79, 95, 111};
                // data according to above calculation in table.
    void main (void)
{
    unsigned char i ;
    TRISB = 0 ;
    while (1)
      {
            for (i = 0 ; i < 48 ; i++)
            PORTB = array [ i ] ;
      }
}
```

13.3 PIC18F458 ADC Features and Programming

- ADC is most commonly used in data acquisition systems. Hence, the PIC microcontrollers have an on-chip ADC.
- The Analog - to - Digital (A/D) converter for PIC18 has **following features**.

(i) It is a 10 bit ADC.

(ii) Depending on the PIC18 family member, the chip can have 5 to 15 channels. In PIC18F458 there are 8 inputs RA0-RA7 of port A. They are used as 8 analog channels.

(iii) The A/D module has four registers. They are :

(a) ADRESH (A/D Result High Register)

(b) ADRESL (A/D Result Low Register)

(c) A/D control register 0 (ADCON0)

(d) A/D control register 1 (ADCON1)

(iv) The ADRESH and ADRESL registers hold the result of the A/D conversion and give a 16 bit output.

However, as the PIC has a 10 bit A/D converter, 6 of the 16 bits will remain unused. The upper 6 or lower 6 bits can be left unused.

(v) The ADCON0 is a A/D control register used for setting the conversion time. It can also be used to select the analog input channel. The ADCON1 is a A/D control register used for setting V_{ref} voltage.

(vi) The analog reference voltage V_{ref} can be selected by using V_{DD} or voltage level on the V_{REF+} or V_{REF-} pins.

$$V_{ref} = V_{ref}(+) - V_{ref}(-)$$

(vii) The A/D conversion time is decided by the crystal oscillator connected to OSC1 and OSC2 pins of PIC18F458. It has to be greater than 1.6 ms. Fig. 13.3.1 shows the PIC18 ADC Block Diagram.

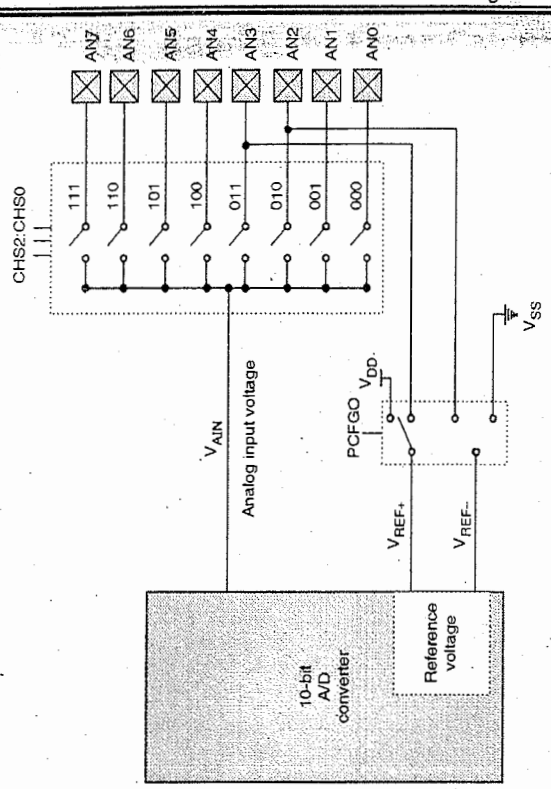


Fig. 13.3.1 : PIC ADC Block Diagram

13.3.1 ADCON0 Register

Size : It is an 8 bit register.

Use : ADCON0 register is used for setting the conversion time. It is also used for selecting the analog channel.

- Fig. 13.3.2 shows the ADCON0 register.
- On power up, the ADC is turned off so as to decrease the power consumption. The ADON bit of ADCON0 register is used to turn on the ADC.
- To start and observe the A/D conversion the GO/DONE bit is used.
- The ADCS0 and ADCS1 bits are used to set the conversion time. ADCS2 bit belongs to ADCON1 register.

ADCON0 : A/D CONTROL REGISTER 0

ADCS1	ADCS0	CHS2	CHS1	CHS0	$\overline{\text{GO/DONE}}$	–	ADON
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

bit 7-6 **ADCS1 : ADCS0 :** A/D Conversion Clock Selects bits.

ADCON1 < ADCS2>	ADCON0 <ADCS1: ADCS0>	Source for clock conversion
0	00	FOSC/2
0	01	FOSC/8
0	10	FOSC/32
0	11	FRC (clock derived from the internal A/D RC oscillator)
1	00	FOSC/4
1	01	FOSC/16
1	10	FOSC/04
1	11	FRC (clock derived from the internal A/D RC oscillator)

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

000 = Channel 0 (AN0)
001 = Channel 1 (AN1)
010 = Channel 2 (AN2)
011 = Channel 3 (AN3)
100 = Channel 4 (AN4)
101 = Channel 5 (AN5)
110 = Channel 6 (AN6)
111 = Channel 7 (AN7)

bit 2

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

When ADON = 1 :

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

0 = A/D conversion not in progress

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

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

1 = A/D converter module is powered up

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

Fig. 13.3.2 : ADCON0-A/D control register 0

13.3.2 ADCON1 Register

Size : It is an 8 bit register.

Function : The ADCON1 register is used to select the V_{ref} voltage. Fig. 13.3.3 shows the ADCON1 register.

ADCON1 : A/D CONTROL REGISTER 1

ADFM	ADCS2	–	–	PCFG3	PCFG2	PCFG1	PCFG0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	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'

bit 6 **ADCS2 :** A/D Conversion Clock Select bit.

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

bit 5, bit 4 **Unimplemented :** Read as '0'

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

PCFG	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}	/60
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

Note : Shaded cells indicate channels available only on PIC18F4X8 devices.

Fig. 13.3.3 : ADCON1 register

- We know that the 16 bit result of the A/D conversion is in ADRESH and ADRESL registers. For making this result either left or right justified (i.e. not using the higher or lower 6 bits) the ADFM bit of ADCON1 register is used.

	ADRESH		ADRESL	
Left Justified ADFM = 0	9	2	1 0	UNUSED
Right Justified ADFM = 1	UNUSED 9	8	7	0

Fig. 13.3.4 : ADRESH, ADRESL registers and ADFM bit

- **The PCFG bit is used for managing the port configuration of the A/D channel.**

Ex. 13.3.1

For a PIC18 based system we have V_{ref} = 5 V.

Determine

(a) Step size,

(b) ADCON1 value if we require 3 channels and ADRESH : ADRESL are Left justified.

Soln. :

(a) Step size = $\frac{V_{ref}}{2^n} = \frac{5}{2^{10}} = \frac{5}{1024}$ = **4.8 mV.**

(b) ADCON1 = 0x000100 as 100 gives 3 analog input channels. ADFM = 0 for left justified. ADCS2 bits is x as it is decided by conversion speed.

13.3.3 A/D Conversion Time

- The A/D conversion is defined in terms of T_{ad} such that T_{ad} is the conversion time per bit.

nding T_{ad}, a clock source of $\frac{f_{osc}}{2}, \frac{f_{osc}}{4}, \frac{f_{osc}}{8}$,

f_{osc}

is used.

of PIC18 chip.

conversion

Soln. : For ADCON register.

(i) $\frac{f_{osc}}{2} = \frac{5}{2} = 2.5$ MHz

$T_{ad} = \frac{1}{2.5 \text{ MHz}} = 800$ ns.

It is invalid because it is faster than 1.6 μs

(ii) $\frac{f_{osc}}{8} = \frac{5}{8}$ MHz = 625 KHz

$T_{ad} = \frac{1}{625 \text{ KHz}} = 1.6$ μs.

The conversion time = 12 × 1.6 μs = 19.2 μs

(iii) $\frac{f_{osc}}{32} = \frac{5}{32}$ MHz = 156.25 KHz

$T_{ad} = \frac{1}{156.25 \text{ KHz}} = 6.4$ μs.

The conversion time = 12 × 6.4 μs = 76.8 μs

13.3.4 A/D Converter Programming using Polling

The steps for A/D converter programming using polling are :

Step I	:	Using the ADON bit of ADCON0 register, turn on the ADC.
Step II	:	Select the ADC input channel with the BSF TRISA.x instruction.
Step III	:	Select the V_{ref} voltage.
Step IV	:	Using ADCON0 and ADCON1 registers select the conversion speed.
Step V	:	Wait for the needed acquisition period.
Step VI	:	With the GO/DONE bit of ADCON0 register start conversion.
Step VII	:	Check the GO/DONE bit to see if conversion is complete by polling.
Step VIII	:	After the conversion is complete read the ADRESH and ADRESL registers to obtain the digital output.
Step IX	:	Goto step V

ency

$= 2 \times T_{ad}$

MHz. Cal

Ex. 13.3.3

Interface ADC to PIC18 microcontroller. Write a program to get data from channel 0 of ADC and display the result on port C and port D every quarter of a second.

Soln. :

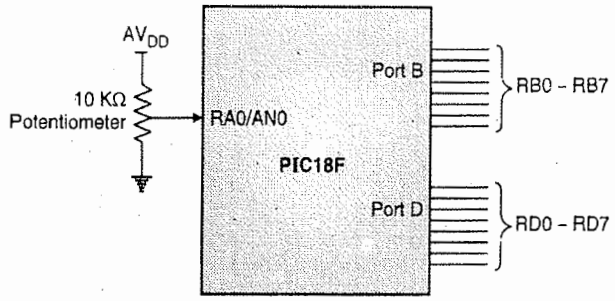


Fig. P. 13.3.3 : Interfacing ADC to PIC microcontroller

C Program :

```
# include<P18F458.h>
    void delay (unsigned int) ;
    void main (void)
    {
        TRISB = 0 ;        //Make port B output port
        TRISD = 0 ;        //Make Port D output port
        TRISAbits.TRISA0 = 1;          //RA0 = 1
        ADCON0 = 0x81 ;    //fosc/64, channel 0, ADC on
        ADCON1= 0xCE ;
                           //fosc/64, AN0 input, right justified
    while (1)
    {
        ADCON0bits.GO = 1 ;            // start conversion
        while (ADCON0bits.DONE == 1) ;
                                //check end of conversion
        PORTB = ADRESL ;     //send low byte result to
                                Port B
        PORTD = ADRESH ;
                            //Send high byte result to Port D
        Delay (250) ;       // Quarter second delay
        }
    }
    void delay (unsigned int xtime)
    {
        unsigned int i ;
        unsigned char j;
        for (i=0; i<xtime; i++)
        for (j=0; j<165; j++) ;
    }
```

13.3.5 A/D Programming using Interrupts

- For programming the A/D converter using interrupts we need to set the ADIE (A/D interrupt enable) flag in the PIE1 register. If this bit is set, then after the conversion the A/D interrupt flag (ADIF) is set in the PIR1 register. The ADIF flag forces the microcontroller to read binary outputs.

Ex. 13.3.4

Write a program using interrupts to get data from channel 0 of ADC and display the result on Port B and Port D every quarter of a second.

Soln. : C Program :

```
# include <PIC18F458.h>
# pragma code hi_int = 0x0008     //high priority interrupt
    void delay (unsigned int);
    void hi_int (void)
    {
        my_isr ( ) ;
    }
    # pragma code          //end high-priority interrupt
    # pragma interrupt my_isr
    void my_isr (void)
    {
        if (PIR1bits.ADIF = = 1)
                                // Did ADC cause interrupt ?
        ADC_ISR ( ) ;     // if yes execute ISR
    }
    void main (void)
    {
        TRISB = 0 ;       //Make Port B output port
        TRISD = 0 ;       // Make Port D output port
        TRISAbits.TRISA = 1;  //RA0 = 1
        ADCON0 = 0x81 ;
        ADCON1 = 0xCE ;
        PIR1 bits.ADIF = 0 ;     //ADIF = 0
        PIE1bits. ADIE = 1 ;     //Activate ADIE interrupt
        INTCONbits.PEIE = 1 ;  //Activate peripheral
                                  interrupts
        INTCONbits.GIE = 1 ;  //Activate interrupts
                                  globally
        while (1)
        {
            Delay (250) ;
            ADCON0bits. GO = 1;          //Start conversion
        }
    }
```

```
ADC ISR
void ADC_ISR (void)
{
PORTB = ADRESL ;  //send low byte result to Port C
PORT D = ADRESH ; //Send high byte result to
                  Port D
Delay (250);          // quarter second delay
PIR1bits.ADIF = 0 ;        // ADIF = 0
}
void delay (unsigned int xtime)
{
    unsigned int i,j;
    for (i=0; i<xtime; i++)
    for (j=0; j<165; j++);
}
```

Ex. 13.3.5

SPPU - Dec. 2014, Dec. 2015, 10 Marks, Dec. 2016, 8 Marks, Lab Assignment

Draw and explain interfacing of ADC for analog input 0-5V and write a C code.

OR

Write a embedded C program for reading single analog input range from 0V to 5V and display it on LCD.

Soln. : Fig. P. 13.3.5 shows the interfacing diagram. The microcontroller ADC module converts the analog signal to 10 bit binary data. The analog input to PIC is from 0 to 5V. This value is measured converted to volts and displayed on the LCD. If we use a step size of 5 mV,

V_{out} = Number of steps × step size.

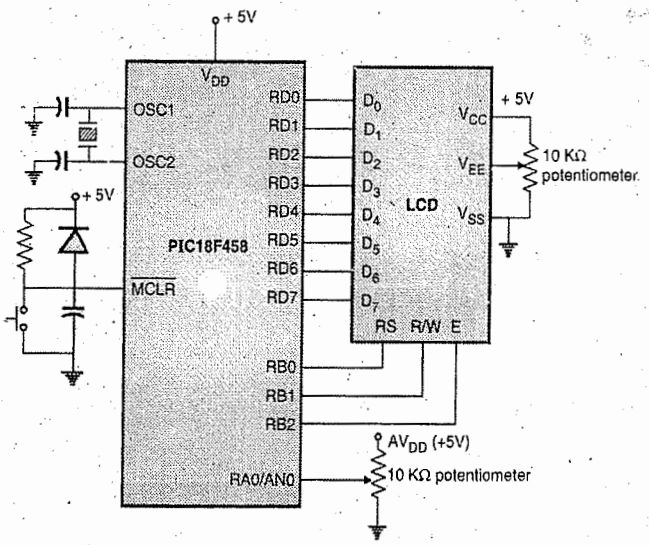


Fig. P. 13.3.5 : Interfacing PIC18F458 with analog voltage

V_{out} = 1024 × 5 mV = 5.12 V for full scale output. i.e. the binary output number for A/D is the real voltage. To convert it to mV we multiply by $\frac{5}{1023}$ and then again convert it to volts and is displayed on the LCD.

Program

```
# include <P18F458.h>
# define Ldata PORTD                // port D = LCD data pins
# define RS PORTBbits.RB0           // RS = PORTB.0
# define RW PORTBbits.RB1           // RW = PORTB.1
# define E PORTBbits.RB2            // E = PORTB.2
void delay (unsigned int);
void Lcdcmd (unsigned char value);
void Lcddata (unsigned char value);
void Lcdinit ( );
void main ( )
{
    TRISD = 0;                      // port D = output port
    TRISB = 0;                      // port B = output port
    Unsigned long voltage;
    unsigned int i ;
    char digit [ ] = "0.000 Volts";
    TRISAbits.TRISA0 = 1;           // RA0 = 1
    ADCON0 = 0x81;                  // fosc/64, channel 0,
                                    ADC ON
    ADCON1 = 0xCE;                  // fosc/64, AN0 input,
                                    right justified
    Lcd ( );
    while (1)
    {
        ADCON0bits.GO = 1;          // start conversion
        while (ADCON0bits.DONE = = 1);
                                    //check for end of conversion
        voltage = ADC_Read (0);
                    // Get the 10 bit result of A/D conversion
        voltage = voltage × 5000/1023   // convert it to mV
        digit [0] = (voltage/1000) + 48 ;
        Lcdddata (digit [0]);
        Lcddata ("·") ;
        digit [1] = (voltage
                    %1000/10) + 48;
        Lcddata (digit [1]);
        digit [2] = ((voltage%1000)
                    %100/10) + 48;
        Lcddata (digit [2]);
        digit [3] = (((voltage%100)
                    %100)%10) + 48;
        Lcddata (digit [3]);
        Lcddata ("V");
```

Display voltage on LCD in V

```
        }
}
void Lcdinit (void)
{
        Lcdcmd (0x38);   // initialize LCD2 lines, 5×7 matrix.
        Delay (250);
        Lcdcmd (0x0E)        // display on, cursor on
        Delay (250);
        Lcdcmd (0x01);       // clear LCD
        Delay (250);
        Lcdcmd (0x06);       // shift cursor right
        Delay (250);
        Lcdcmd (0x80);       // Line 1, position 0
        Delay (250);
        }
void Lcdcmd (unsigned char value)    // command routine
        {
            Ldata=value;                // get value on pins
            RS = 0;
            RW = 0;
            E = 1;                      // strobe enable pin
            Delay (1);
            E = 0;
        }
void Lcddata (unsigned char value)
{
        Ldata = value;                  // get value on pins
        RS =1;
        RW = 0;
        E = 1;
        Delay (1);
        E = 0;
}
void delay (unsigned int xtime)
{
        unsigned int i;
        unsigned char j ;
        for (i=0; i<xtime; i++)
        for (j=0; j<165; j++);
}
```

Ex. 13.3.6 **SPPU - May 2016, 8 Marks**

Draw an interfacing of temperature Sensor to PIC using Serial ADC and indicate excess temperature when exceed the set point by LED.

Soln. :

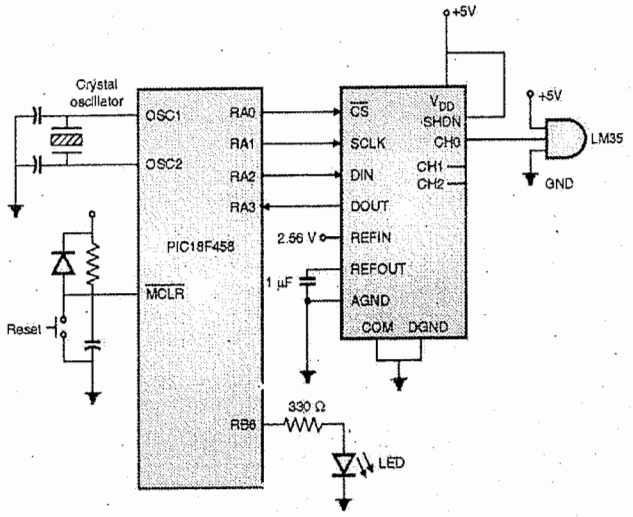


Fig. P.13.3.6

Program

```
#include <PIC18F458.h>
#define CS PORTAbits.RA0
#define SCK PORTAbits.RA1
#define DIN PORTAbits.RA2
#define DOUT PORTAbits.RA3
#define MSBWREG = WREG ^ 7
#define LSBWREG = WREG ^ 0;
#define LED PORTBbits.RB6
void main(void)
{
        unsigned char control_byte = 0x8E;  // channel 0
        unsigned char 1, mybyte;
        while (1)
        {
            WREG = control_byte ;          // select channel 0
            CS = 0
            for (i = 0 ; i < 8; i++)
            {
                SCLK = 0;
                DIN = MSBWREG;
                Delay ( );
                SCLK = 1;                  // Latch bit
                Delay ( );
                WREG = WREG << 1,          // next bit
            }
            CS = 1;                        // Deselect serial ADC
```

```
        SCK = 0;            // Make SCLK low during
                               conversion
        CS = 0;             // Read data
        SCK = 1;
        Delay ( );
        SCK = 0;            // Get all 8 bits
        Delay ( );
        for (i = 0; i < 8; i++)
        {
            SCK = 1;
            Delay ();
            SCK = 0;
            Delay ();
            LSBWREG = DOUT;   // Get data from DOUT
            WREG = WREG << 1;
                        // Keep shifting for all 8 bits of data
        }
        CS = 1;                 // Deselect ADC
        PORTB = WREG;           // 8 bit data (temperature
                                   from sensor)
        TRISBbits.TRISB6 = 0;   // RB6 = output pin
        while (1)
        {
            mybyte = PORTB;             // Get data
            if (mybyte > 65)
            LED = 1;            // Turn on LED if temp
                                   exceeds set point
            else
            LED = 0;        // Turn off LED if temp is below
                               set point
        }
    }
}
```

Syllabus Topic : Temperature Sensor Interfacing using ADC

13.4 Temperature Sensor Interfacing using ADC

- Temperature is the most-measured process variable in industrial automation. Most commonly, a temperature sensor is used to convert temperature value to an electrical value. Temperature Sensors are the key to read temperatures correctly and to control temperature in industrials applications.
- The LM34 are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Fahrenheit temperature.
- The LM35 are precision integrated-circuit temperature sensors, whose output voltage is linearly proportional to the Celsius (Centigrade) temperature.
- The LM34/LM35 thus has an advantage over linear temperature sensors calibrated in degrees Kelvin, as the user is not required to subtract a large constant voltage from its output to obtain convenient Fahrenheit scaling.
- LM35 does not require any external calibration or trimming to provide typical accuracies of ±1/4°C at room temperature and ± 3/4°C over a full –55 to +150°C temperature range.
- The LM35 is rated to operate over a – 55°C to +150°C temperature range.
- As shown in the Fig. 13.4.1 the connection for LM35 is very simple. Also the output scale is linear with a change of 10mV/°C.

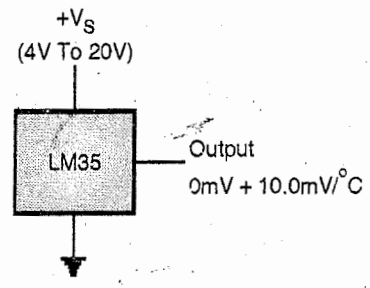


Fig. 13.4.1 : Circuit diagram for the LM35 basic temperature sensor (+2° C to +150° C)

Ex. 13.4.1

Design a PIC18F458 based system to interface LM35 using ADC0848. Write the corresponding C program for reading and displaying temperature.

Soln. :

Fig. P. 13.4.1 shows PIC18F458 connection to temperature sensor.

- The LM35 is connected to channel 0 (RA0 pin). The channel AN2 (RA2 pin) is connected to V_{ref} of 2.56 V. It makes PCFG = 0010 for ADCON1 register.
- The 10 bit output of A/D converter is divided by 4 to get the real temperature.

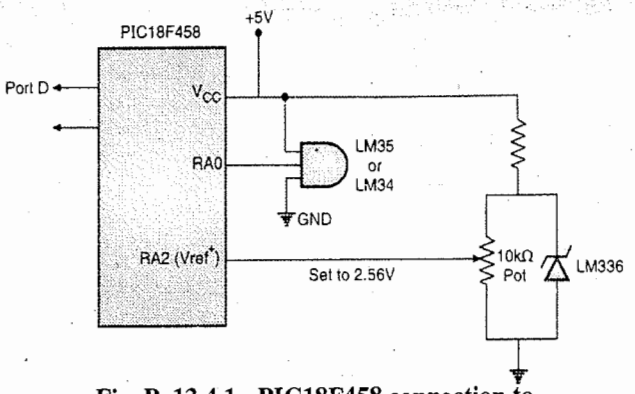


Fig. P. 13.4.1 : PIC18F458 connection to temperature sensor

C Program

```
# include <PIC18F458.h>
void delay (void)
void main (void)
{
        unsigned char Lo_byte, Hi_byte, bin_temp;
        TRISD = 0;                  // port D as o/p
        TRISAbits.TRISA0 = 1;       // RA input
        TRISAbits.TRISA2 = 1;  // RA2 = input for Vref input
        ADCON0 = 0x81;    // fosc/64, channel 0
        ADCON1 = 0xC5;    // right justified, fosc/64,
        while (1)                   // AN0 = analog, AN3 = Vref+
        {
            Delay (1);    // give A/D channel time to sample
            ADCON0bits.GO = 1;    // start converting
            while (ADCON0bits.DONE = = 1);
                                          //wait for EOC
            Lo_byte = ADRESL;     // save low byte
            Hi_byte = ADRESH;     // save high byte
            Lo_byte >> = 2;       // shift right
            Lo_byte & = 0x3F;     // mask upper 2 bits
            Hi_byte << 6;         // shift left 6 times
            Hi_byte & = 0xC0;     // mask lower 6 bits
            bin_temp = Lo_byte / Hi_byte;
                                  // get the real temperature
            PORTD = bin_temp;
                        // display the temperature on Port D
        }
}
void delay (unsigned int xtime)
{
        unsigned int i;
        unsigned char j;
        for (i=0; i<xtime; i++)
        for (j=0; j<165; j++);
}
```

❑❑❑

UNIT - VI

CHAPTER 14

Interfacing RTC and EEPROM with I2C and SPI

Syllabus Topic : DS1307 RTC with I2C

14.1 DS1307 RTC with I2C

- The DS1307 Serial Real-Time Clock is a device used to provide real-time clock/calendar for various applications.
- It has an additional 56 bytes of NV SRAM.
- DS1307 uses I2C i.e. 2-wire, bi-directional bus for transferring the address and data serially.
- The clock/calendar provides seconds, minutes, hours, day, date, month, and year information including leap year.
- It supports 12 hour and 24 hour clock modes with AM and PM indications .
- It has an in built power sense circuit for detecting the power failures and automatically switch to battery supply.

14.2 Features of DS1307 RTC

SPPU - May 16

University Question

Q. What are the features of RTC ? **(May 2016, 4 Marks)**

The features of DS 1307 RTC are as follows :

1. It keeps a track of seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation valid up to 2100.
2. It supports a I2C two wire serial interface .
3. For storing the data it has a 56-byte, battery-backed, non-volatile (NV) RAM.
4. It has a programmable square-wave output signal
5. It has an in built power sense circuit for detecting the power failures and automatically switch to battery supply.
6. It operates in temperature range : 40°C to +85°C.
7. It needs current less than 500nA in battery backup mode with oscillator running.
8. Available in 8-pin DIP or SO.

14.3 Pin Diagram of DS1307 RTC

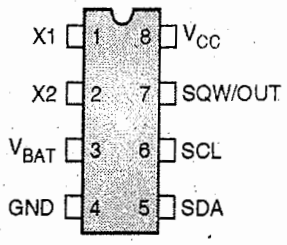


Fig. 14.3.1 : Pin diagram of DS 1307 RTC

- V_{CC}- Primary Power Supply
- X1, X2 - 32.768kHz Crystal Connection
- VBAT - +3V Battery Input
- GND - Ground
- SDA - Serial Data
- SCL - Serial Clock
- SQW/OUT - Square Wave/Output Driver

Pin Name	Description
V_{CC} - Primary Power Supply	– It is the +5V input. The data can be read and written when 5V is applied to the RTC within normal limits. – Reads and writes to the device are restricted if it is connected to a 3V battery and if V_{CC} is below 1.25 x V_{BAT}. – Even in low voltage the timekeeping function remains unchanged. – If V_{CC} decreases below V_{BAT} ,the timekeeper and RAM switch over to external power supply.
VBAT - +3V Battery Input	– It is battery input for any standard 3V lithium cell or other energy source. For proper operation of the RTC the battery voltage should be between 2.0V and 3.5V .
X1, X2 - 32.768kHz Crystal Connection	A standard 32.768kHz quartz crystal is connected to the X1, X2 pins.
GND	Ground
SDA - Serial Data	It is the input/output pin for the 2-wire serial interface. It needs an external pull-up resistor as it is open drain.

Pin Name	Description
SCL - Serial Clock	– This pin is used to synchronize data movement on the serial interface.
SQW/OUT- Square Wave/Output Driver	– If the SQWE bit = 1, the SQW/OUT pin outputs one of four square wave frequencies (1Hz, 4kHz, 8kHz, 32kHz). – It needs an external pull-up resistor as it is open drain. – SQW/OUT will operate with either Vcc or Vbat applied.

14.4 DS1307 Operation

- Fig 14.4.1 shows the block diagram of DS1307 RTC .On the I2C bus the RTC behaves like a slave device.
- By giving a START condition and providing a device identification code followed by a register address, we can access the RTC. The registers can be accessed sequentially till a STOP condition is executed.

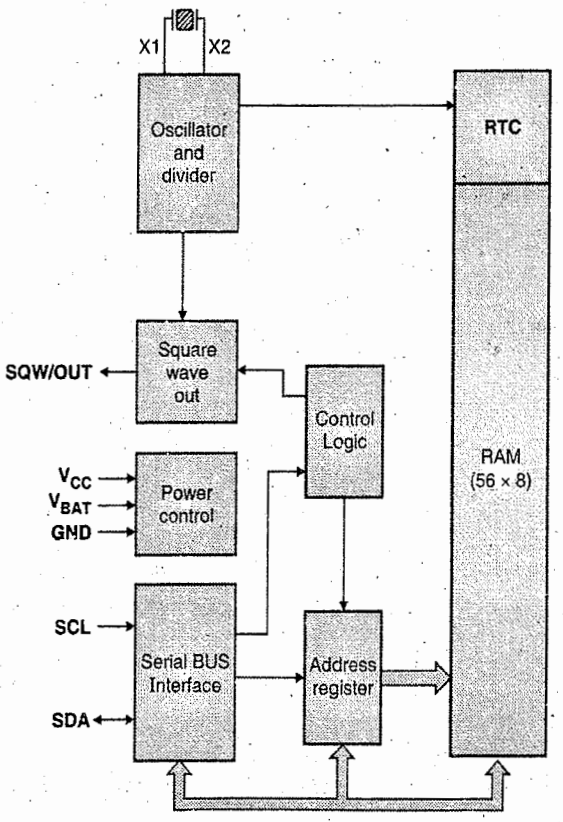


Fig 14.4.1 : Block diagram of DS1307 RTC

14.5 RTC and RAM Address MAP

- The address map for the RTC and RAM registers of the DS1307 is shown in Fig. 14.5.1
- The address locations 00h to 07H are used for the RTC registers. The address locations 08H to 3FH are used for the RAM registers.
- During a multi-byte access, when the address pointer reaches 3FH,i.e.the end of RAM space, it wraps around to location 00h, indicating the starting of the clock space.

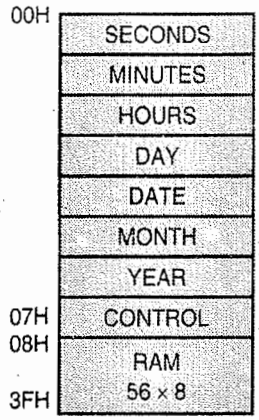


Fig. 14.5.1 : DS1307 Address map

14.6 Clock and Calendar

- The time and calendar information is obtained by reading the appropriate register bytes. The RTC registers are illustrated in Fig.14.6.1.
- By writing the appropriate registers, we can set or initialize the time and calendar . The contents of the both these registers i.e. the time and calendar are in the BCD format.
- If bit 7 i.e. the clock halt (CH) bit is set , the oscillator is disabled. Otherwise the oscillator is enabled.

Note : On power–up the state of registers is undefined. Hence, we need to enable the oscillator.

- The DS1307 RTC can operate in 12-hour or 24-hour mode. For selecting the 12/24 hour mode bit 6 of hour register is used . If bit 6 of hour register is low it indicates 24 hour mode , otherwise it is 12 hour mode.
- In the 12-hour mode, bit 5 of hour register indicates AM/PM .if bit 5 = 0 it indicates AM and if bit 5 = 1 it indicates the time in PM.
- In the 24-hour mode, bit 5 represents the second 10 hour bit (20- 23 hours).

	BIT7	BIT6	BIT5	BIT4	BIT3	BIT2	BIT1	BIT0	
00H	CH	10 Seconds			Seconds				00-59
01H	0	10 Minutes			Minutes				00-59
02H	0	12 / 24	10 HR / A/P	10 HR	Hours				01-12 00-23
03H	0	0	0	0	0	Day			1-7
04H	0	0	10 Date		Date				01-28/29 01-30 01-31
05H	0	0	0	10 Month	Month				01-12
06H	10 Year				Year				00-99
07H	Out	0	0	SQWE	0	0	RS1	RS0	

Fig. 14.6.1 : Timekeeper registers

14.7 Control Register

- Fig 14.7.1 shows the control register of DS1307 RTC . Its function is to control the operation of the SQW/OUT pin.

Size : It is of 8 bits.

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

Fig. 14.7.1 : Control Register

Pin	Function
OUT (Output control)	– It is responsible for controlling the output level of the SQW/OUT pin in cases when the square wave output is disabled. – If SQWE = 0, then the SQW/OUT pin =1 if OUT = 1 and – if SQWE = 0, then the SQW/OUT pin =0 if OUT = 0.
SQWE (Square Wave Enable)	– The oscillator output is enabled if SQWE = 1 . – The value of bits RS0 and RS1 decides the square wave output frequency . – If the bits RS0 and RS1 are 00 then the square wave output is 1Hz. The clock registers generally update on the falling edge of the square wave.
RS (Rate Select)	– If the square wave output is enabled the RS0 and RS1 bits decide the square wave output frequency . – Table 14.7.1 lists the square wave frequencies that can be selected. – Square wave Output Frequency

Table 14.7.1

RS1	RS0	SQW Output Frequency
0	0	1 Hz
0	1	4.096 kHz
1	0	8.192 kHz
1	1	32.768 kHz

14.8 2-Wire Serial Data Bus (I2C)

- The DS1307 supports a bi-directional, 2-wire bus and data transmission protocol called I2C .
- The device that transfers /sends data onto the bus is called as the **transmitter** while the device that receives the data is called as the **receiver.**
- The **master** is referred to the device that controls the message and a master. The **slaves** are the devices that are controlled by the master.
- The master devices function is to generate the serial clock (SCL) signal, START and STOP conditions and also control the bus accesses.

- The DS1307 behaves like a slave on the 2-wire bus. Fig 14.8.1 shows a typical bus configuration using this 2-wire protocol.
- If the bus is free then only the data transfer can be initiated.

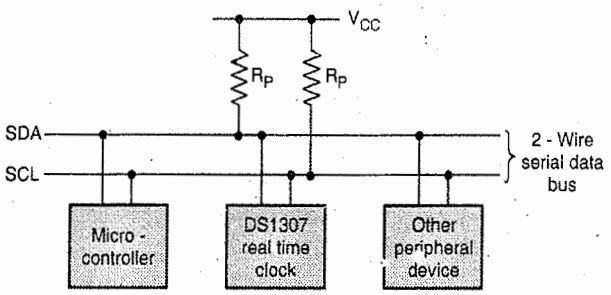


Fig. 14.8.1 : Typical 2 wire bus configuration

- At the time of data transfer, if the clock signal is high then the data line should remain stable. If the clock signal is high and there is a change in the status of the data line then the signals will be considered to be control signals.
- For 2 wire bus configuration the following bus conditions have been defined:
  1. **Bus not busy :** In this situation both the clock and data lines are HIGH.
  2. **Start data transfer :** If the data line changes its state from HIGH to LOW, while the clock line is HIGH, then this condition is defined as a **START condition**.
  3. **Stop data transfer :** If the data line changes its state from LOW to HIGH, while the clock line is HIGH, then this condition is defined as a **STOP condition**.
  4. **Data valid :** If after a START condition, the data line is stable during the HIGH portion of the clock signal then in such a condition the data line represents **valid data**.
- If the clock signal is LOW then the data lines must be modified .There is one clock pulse per bit of data.
- Each and every data transfer operation begins with a START condition and ends with a STOP condition.
- The master device computes the number of data bytes that are transferred between START and STOP conditions .
- The data is transferred byte-by- byte and every receiver acknowledges the received byte with a ninth bit.

14.8.1 Acknowledge

- After every byte is received an acknowledgement needs to be generated indicating that the transmitted byte is correctly received .
- For generating the acknowledgement ,the master device should generate an extra clock pulse i.e. acknowledge pulse that is associated with the acknowledge bit.
- During the acknowledge clock pulse the SDA line should be pulled LOW such that it remains LOW for the HIGH period of the acknowledge clock pulse.
- We need to also consider the setup and hold times .
- A master device should indicate an end of data to the slave by STOP condition and not by generating an acknowledge bit on the last byte that has been clocked out of the slave.

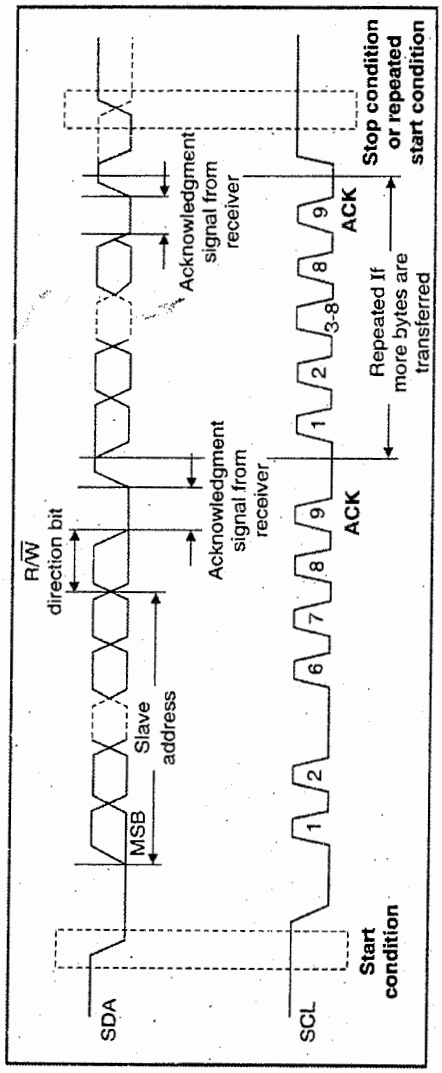


Fig.14.8.2 :Data Transfer On 2-Wire Serial Bus

14.8.2 Types of Data Transfer

Depending upon the state of the R/$\overline{W}$ bit, two types of data transfer are possible :

1. **Data transfer from a master transmitter to a slave receiver :** In this type of data transfer the first byte transmitted by the master is the address of the slave , followed by the number of data bytes. After each received byte the slave returns an acknowledge bit. The MSB (most significant bit) is transmitted first in this type of data transfer.
2. **Data transfer from a slave transmitter to a master receiver :** In this type of data transfer the first byte transmitted by the master is the address of the slave , followed by the acknowledge bit from the slave . Then the number of data bytes to be transmitted by the slave. The master returns an acknowledge bit after all received bytes other than the last byte. At the end of the last received byte, a "not acknowledge" is returned.
   - All the START and STOP conditions and the serial clock pulses are generated by the master device.
   - A data transfer ends with either a repeated START condition or a STOP condition. Data is transferred with the most significant bit (MSB) first.

14.9 Operating Modes

The DS1307 may operate in the following two modes:

1. **Slave transmitter mode (DS1307 read mode)**
   - In the slave transmitter mode the *direction bit indicates the transfer direction.
   - The DS1307 RTC transmits serial data through the SDA line and SCL is the serial clock input .
   - Fig 14.9.1 shows how the START and STOP conditions are recognized as the beginning and end of a serial transfer .
   - After the master generates the Start condition , the first byte that is received is the address byte. It has the 7-bit address of DS1307, which is 1101000, followed by the *direction bit (R/$\overline{W}$). The direction bit is 1 for a read operation.
   - Once the address byte is received and decoded , an acknowledge pulse is sent on the SDA line. The DS1307 RTC then begins to transmit data beginning with the register address pointed to by the register pointer.
   - The register pointer should be written prior to the initialization of the read mode. Otherwise the first address that is read is the last address that is stored in the register pointer. For ending a read operation ,the DS1307 should receive a "not acknowledge" signal .

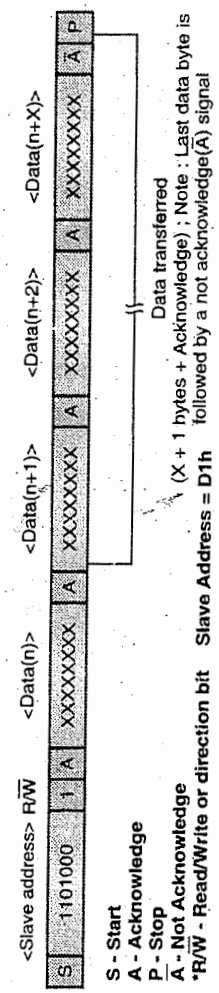


Fig. 14.9.1 :Data Read – Slave Transmitter Mode

2. **Slave receiver mode (DS1307 write mode) :**
   - The slave receiver receives the serial data and clock through SDA and SCL signals. An acknowledge bit is transmitted after each byte is received .

- Fig 14.9.2 shows how the START and STOP conditions are recognized as the beginning and end of a serial transfer .
- After the master generates the Start condition , the first byte that is received is the address byte. It has the 7-bit address of DS1307, which is 1101000, followed by the *direction bit (R/$\overline{W}$). The direction bit is 0 for a write operation.
- Once the address byte is received and decoded , an acknowledge pulse is sent on the SDA line.
- Then the DS1307 acknowledges the slave address + write bit. After that the master device transmits a register address to the DS1307 , setting its register pointer.
- An acknowledge bit is transmitted by DS 1307 after each byte transmitted by the master is received .For ending the data write operation a stop condition is generated by the master device.

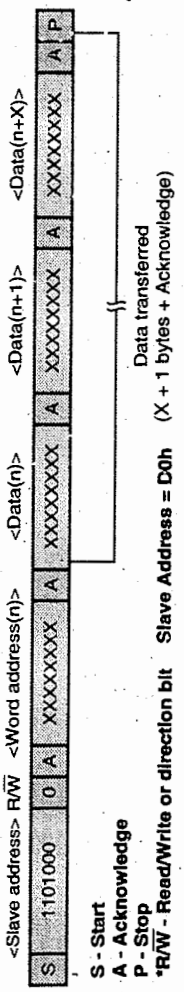


Fig. 14.9.2: Data Write – Slave Receiver Mode

14.10 Interfacing DS1307 with PIC18

SPPU - May 15, Dec. 15, May 16, Dec. 16

University Questions

Q. Draw interfacing diagram and write a program for I2C based RTC with PIC 18Fxxx. **(May 2015, 10 Marks)**

Q. Draw and explain interfacing of I2C based RTC with PIC18FXXX. Write a code in C. **(Dec. 2015, 10 Marks)**

Q. Draw an interfacing diagram to interface RTC with PIC. **(May 2016, 4 Marks)**

Q. Draw and explain interfacing RTC with PIC18FXXX ? **(Dec. 2016, 4 Marks)**

Fig. 14.10.1 shows the interfacing of DS1307 with PIC18 microcontroller.

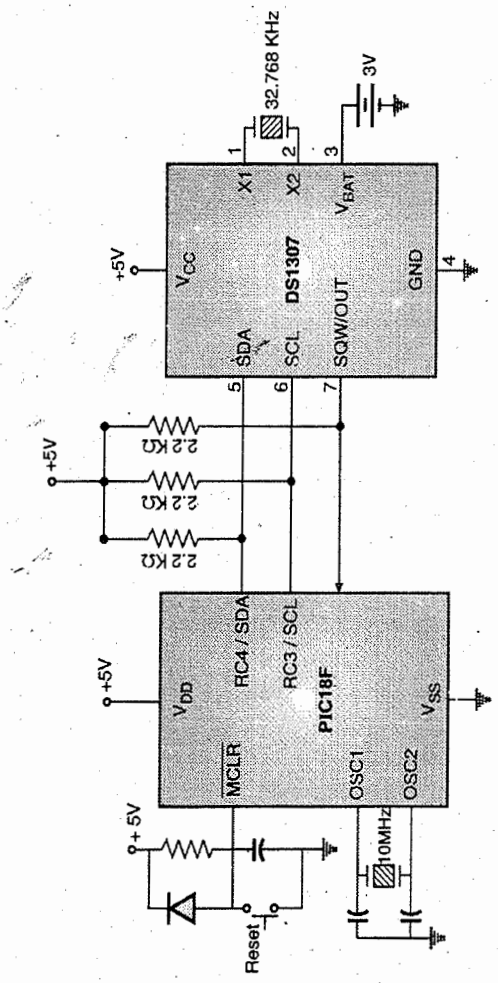


Fig. 14.10.1

The interfacing of I2C-RTC with PIC18F458 is very simple. The date and time can be read in RTC by using the I2C and the value is displayed on the LCD.

Ex. 14.10.1 **SPPU - Dec. 14, 10 Marks, Dec. 16, 8 Marks**

Interfacing DS1307 RTC chip using I2C and display date and time on LCD. **OR**

Draw and explain interfacing of RTC with PIC18FXXX ? Also write embedded C program to update date.

Soln. : Fig. P. 14.10.1 shows the interfacing of PIC18F458 with DS1307 RTC chip using I2C and LCD for reading the time and date and display it on the LCD.

Program

```
//Interfacing RTC DS1307 with PIC18F458 Micro-Controller
and Displaying TIME and DATE on LCD
#include <PICF458.h>
#define XTAL_FREQ 10000000        // XTAL = 10 MHz
#define LCD_DATA PORTD            // define LCD pins
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define E PORTBbits.RB2
void lcd_init();
void lcdcmd(unsigned char value);
void lcddata(unsigned char value);
void i2c_init();
void i2c_start(void);
void i2c_restart(void);
void i2c_stop(void);
void i2c_send(unsigned char dat);
unsigned char i2c_read(void);
unsigned char rtc1307_read(unsigned char address);
                                //RTC DS1307 Read Function
unsigned char BCD2UpperCh(unsigned char bcd);
unsigned char BCD2LowerCh(unsigned char bcd);
unsigned char sec,min,hour,date,month,year;
void main()
    {
        TRISB = 0x00;       // Make port B an output port
        TRISC = 0xFF;       // Make port C an input port
        TRISD = 0x00;       // Make port D an output port
        lcd_init();         // initialize LCD
        i2c_init();         //To Generate the Clock of
                              100Khz
        i2c_start();        // start the I2C protocol
        i2c_send(0xD0);
        i2c_send(0x00);
        i2c_send(0x80);         //CH = 1 Stop oscillator
        i2c_send(0x00);         //Minute
        i2c_send(0x01);         //Hour
        i2c_send(0x06);         // Friday
        i2c_send(0x11);         //11Date
        i2c_send(0x07);         //7 July
        i2c_send(0x14);         //2014
        i2c_stop();             //Stop the I2C Protocol
                                //Start the Clock again
        i2c_start();
        i2c_send(0xD0);
        i2c_send(0x00);
        i2c_send(0x00);         //start Clock and set the
                                second hand to Zero
        i2c_stop();
        lcdcmd(0x01);   //Infinite Loop For Reading
                        Time and Date and displaying it
                        on LCD
        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);
            delay(1);
            lcdcmd(0x80);       // display time on line1 of
                                LCD in hours:mins: secs
            lcdstr("Time:");
            lcddata(BCD2UpperCh(hour));
            lcddata(BCD2LowerCh(hour));
            lcddata(':');
            lcddata(BCD2UpperCh(min));
            lcddata(BCD2LowerCh(min));
            lcddata(':');
            lcddata(BCD2UpperCh(sec));
            lcddata(BCD2LowerCh(sec));
            lcdcmd(0xC0);       // display date on line 2 of
                                LCD in date/month/year
            lcdstr("DATE:");
            lcddata(BCD2UpperCh(date));
            lcddata(BCD2LowerCh(date));
            lcddata('/');
            lcddata(BCD2UpperCh(month));
            lcddata(BCD2LowerCh(month));
            lcddata('/');
            lcddata(BCD2UpperCh(year));
            lcddata(BCD2LowerCh(year));
            delay(1000);
        }
    }
    // LCD initialization
    void lcd_init()
```

```
    {
        lcdcmd(0x38);          //initialize LCD 2 lines ,
                                5×7 matrix
        delay(1);              // wait for sometime
        lcdcmd(0x0E);          //  display on ,cursor on
        delay(1);              // wait for sometime
        lcdcmd(0x01);          // Clear LCD
        delay(1);              // wait for sometime
        lcdcmd(0x06);          // Shift cursor right
        delay(1);              // wait for sometime
    }
void lcdcmd(unsigned char value)
{
    LCD_DATA = value;
    RS = 0;
    RW = 0;
    E = 1;
    delay(1);
    E = 0;
}
void lcddata(unsigned char value)
{
    LCD_DATA = value;
    RS = 1;
    RW = 0;
    E = 1;
    delay(1);
    E= 0;
}
void delay (unsigned int  x)
{
    unsigned int i,j;
    for (i=0;i< x ; i++)
    for (j=0;j<135;j++);
}
void i2c_init()
{
    SSPSTAT &= 0x3F;       // enable slew rate control
                            in high speed
    SSPADD = 0x13;       //set baud rate to 400 KHz
    SSPCON1 = 0x00;        //clear to reset state
    SSPCON2 = 0x00;        // clear to reset state
    SSPCON1 = 0x28;        //enable serial port and
                            select I2C master mode
    TRISCbits.RC3 = 1;     //configure SCL for input
    TRISCbits.RC4 = 1;     // configure SDA for input
    }
void i2c_start(void)
{
    SSPCON2bits.SEN = 1;
}
void i2c_restart(void)
{
    SSPCON2bits.RSEN = 1;
}
void i2c_stop(void);
{
    SSPCON2bits.PEN = 1;
}
void i2c_send(unsigned char dat);
{
SSPBUF = dat;
while(SSPSTATbits.BF); // wait until data is shifted out
}
unsigned char i2c_read(void);
{
SSPCON2bits.RCEN = 1;
while(!SSPSTATbits.BF);      // wait until a byte is
                                received //
return (SSPBUF);
}
```

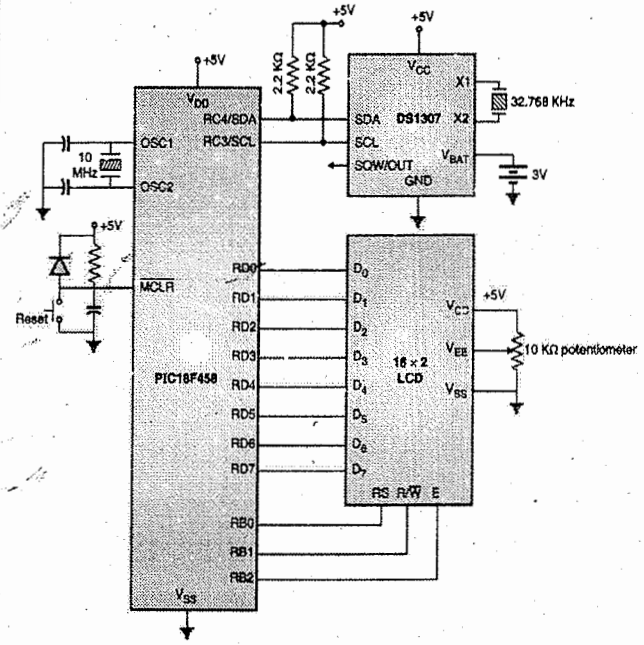


Fig. P. 14.10.1 : Interfacing DS1307 RTC chip using I2C to display date and time on LCD

Syllabus Topic : DS1306 RTC with SPI

14.11 DS1306 RTC with SPI

- DS1306 is a real time clock (RTC) used to provide real time clock/calendar for many applications.

- It keeps a track of "seconds, minutes, hours, days, days of the week, date, month and year with leap year compensation upto 2099."
- It uses BCD format for representing time, calendar and alarm.
- It supports 12 hour and 24 hour clock modes with AM and PM in 12 hour mode.
- It does not support Daylight savings time option.
- It has 128 bytes of non-volatile RAM.
- 28 bytes of RAM are used for clock/calendar and control register while for storing the general purpose data 96 bytes of RAM are used.
- Fig. 14.11.1 shows the block diagram of DS1306.

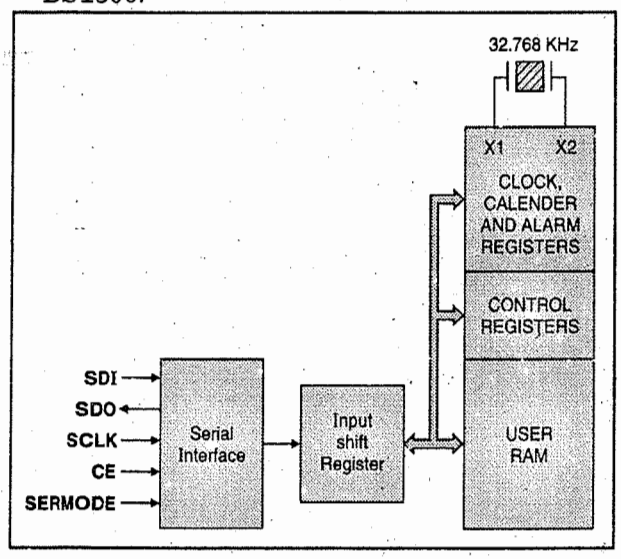


Fig. 14.11.1 : Block diagram of DS1306

14.11.1 Pin Description of DS1306

Fig. 14.11.2 shows the pin diagram of DS1306.

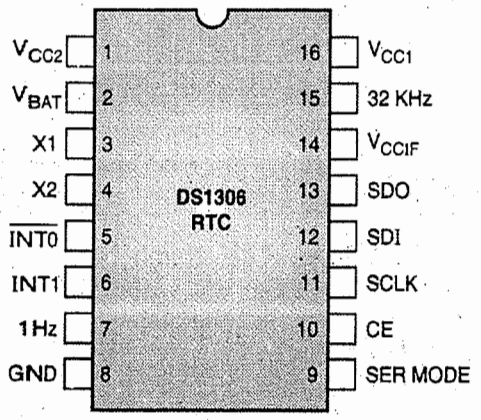


Fig. 14.11.2 : Pin diagram of DS1306

Pin	Function
V_{CC2}	This pin is responsible for providing external back up supply voltage to the chip. It is connected to external power source called trickle charger.
V_{BAT}	This pin is connected to +3V external lithium battery. If pin is not used it should be grounded.
V_{CC1}	It provides a supply voltage of + 5V to the RTC. If V_{CC1} voltage falls below V_{BAT} voltage, the RTC switches to V_{BAT} and provides power to the DS1306 RTC.
GND	Ground
SDI (serial Din)	The function of this pin is to provide path to get data into the chip, one bit at a time.
SDO (Serial Dout)	The function of this pin to get data out of the RTC serially one bit at a time.
X1 - X2	They provide clock source to the chip by connecting a external crystal oscillator of 32.768 KHz.
32 KHz	It is an output pin that always has a frequency of 32.768 KHz
SCLK (serial clock)	It is an input pin. It provides serial clock. This serial clock synchronizes the data transfers between PIC18 microcontroller and DS1306.
CE (Chip Enable)	It is an active high input pin that must be enable during the DS1306 read and write options.
$\overline{INT0}$	It is an active low output signal. For using this signal the interrupt enable bit in RTC control register should be pulled high.
INT1	It is an active high output signal. For using this signal the interrupt enable bit in the RTC control register should be pulled high.
1Hz	The DS1306 automatically creates a square wave of frequency 1 Hz at this output pin by enabling the required bits in the DS1306 control register.
V_{CCIF} (Interface logic power supply input)	It allows DS1306 to be interfaced with 3V logic in mixed supply systems.
SER MODE (serial mode selection)	This pin is an input pin. If the SPI mode is selected the SER MODE pin = 1. If SER MODE = 0, then 3 wire mode is used.

14.11.2 Address Map of DS1306

Table 14.11.1 shows the DS1306 registers.

Table 14.11.1

Hex Address		D7	D6	D5	D4	D3	D2	D1	D0	Range in Hex
Read	Write									
0x00	0x80	0	10 sec.			Seconds				00 – 59
0x01	0x81	0	10 min			Minutes				00 – 59
0x02	0x82	0	24/12	20 hour P/A	10 hour	Hours				00 – 23 01 –12 P/A
0x03	0x83	0	0	0	0	0	Day			01 – 07
0x04	0x84	0	0	10 date		Date				01 – 31
0x05	0x85	0	0	10 Month		Month				01 – 12
0x06	0x86	0	10 year			year				00 – 99
0x07	0x87	M	10 sec Alarm 0			Sec Alarm 0				00 – 59
0x08	0x88	M	10 min Alarm 0			Min Alarm 0				00 – 59
0x09	0x89	M	24/12	20 hours P/A	10 hour	Hour Alarm 0				00 – 23 01 – 12 P/A
0x0A	0x8A	M	0	0	0	0	Day Alarm 0			01 – 07
0x0B	0x8B	M	10 sec Alarm 1			Sec Alarm 1				00 – 59
0x0C	0x8C	M	10 min Alarm 1			Min Alarm 1				00 – 59
0x0D	0x8D	M	24/12	20 hour P/A	10 hour	Hour Alarm 1				00 – 23 01 – 12 P/A
0x0E	0x8E	M	0	0	0	0	Day Alarm 1			01 – 07
0x0F	0x8F	Control Register								
0x10	0x90	Status register								
0x11	0x91	Trickle charger register								
0x12 – 0x1F	0x92 – 0x9F	Reserved								

- Table 14.11.1 shows the address map of DS1306 RTC. The address map consists of 128 bytes of user RAM with addresses 00 – 7FH.
- First fifteen bytes of RAM are for RTC time, calendar, alarm and data. They are assigned addresses 00 – 0EH.
- The next three bytes 0FH, 10 H and 11H are used for control register, status register and trickle charger register.
- The bytes 12H – 1FH cannot be used. They are reserved.
- The remaining 96 bytes from 20H to 7FH can be used for storing data.
- All the 128 bytes of RAM can be directly written or read except for the reserved memory locations 12H – 1FH.

14.11.3 Time and Date Address Location

- The time and data information can be obtained by reading the correct memory bytes.

Table 14.11.2 : Time and Date Address locations for DS1306

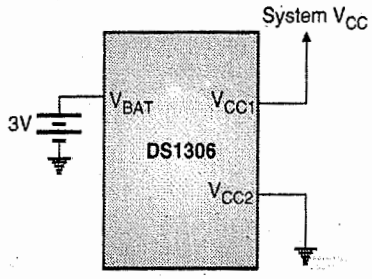
Address location		Function	Data mode range	
Read	Write		BCD	Hex
00	80	Seconds	00 – 59	00 – 59
01	81	Minutes	00 – 59	00 – 59
02	82	Hours, 12 hour mode	01 – 12	41 – 52 AM
		Hours, 12 hour mode	01 – 12	61 – 72 PM
		Hours, 24 hour mode	00 – 23	00 – 23
03	83	Day of the week Sunday = 1	01 – 07	01 – 07
04	84	Day of the Month	01 – 31	01 – 31
05	85	Month	01 – 12	01 – 12
06	86	Year	00 – 99	00 – 99

- The DS1306 RTC provides data in BCD format. The bytes addresses 00H to 06H are kept for the time and date as shown in Table 14.11.2.
- By modifying the bit 6 of hour location 02 we can select 12 hour or 24 hour mode. If Bit 6 = 0,

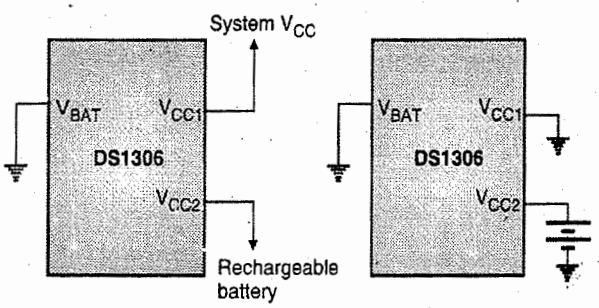
then 24 hour mode is selected. If Bit 6 = 1 then 12 hour mode is selected. Bit 5 decides AM/PM. If bit D5 = 0 the time is in AM otherwise time is in PM.

14.11.4 Power Supply Configurations

- The DS1306 has three power input pins. Hence, different power configurations as shown in Fig. 14.11.3 are allowed.



(a) Back up supply is non rechargeable lithium battery



(b) Back up supply is rechargeable battery or super capacitor **(c) Battery operated mode**

Fig. 14.11.3 : DS1306 power supply configurations

14.11.5 Control Register

- Fig. 14.11.4 shows the DS1306 control register. It is of 8 bit. The control register has address 0FH for read and 8FH for write operations.

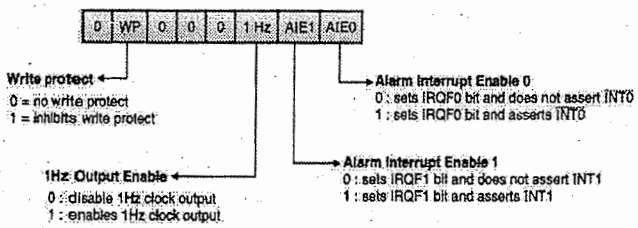


Fig. 14.11.4 : DS1306 control register

The RTC control **register has four functions.** They are :

(i) Enable/Disable write protection : WP bit must be cleared on power-up, so that we can write data to the DS1306 registers.

(ii) Enable/disable 1 Hz clock output

(iii) Enable/disable Alarm 0 interrupt

(iv) Enable/disable Alarm 1 interrupt

14.11.6 Status Register

- Fig. 14.11.5 shows the status register of DS1306 RTC.
- If the AIE1 = 1 and IRQF1 = 1 then the INT1 pin will output a pulse of 62.5 ms.
- If AIE0 = 1 and IRQF0 = 1 the $\overline{INT0}$ is activated.
- The IRQF1 and IRQF0 flags are cleared when the address pointer goes to any one of the Alarm 1 and Alarm 0 registers during a read or write operation.
- If device is powered by V_{CC2} or V_{BAT} IRQF1 can be activated and if the device is powered by V_{CC1}, V_{CC2} or V_{BAT}, IRQF0 can be activated.

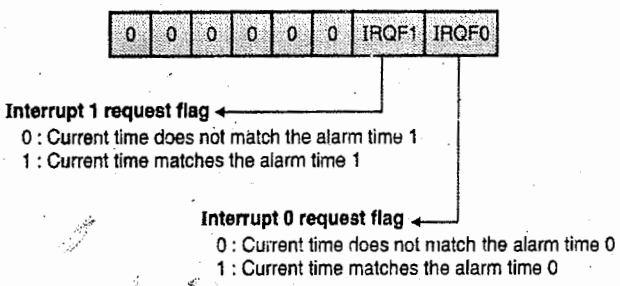


Fig. 14.11.5 : DS1306 RTC status register

14.11.7 Interfacing DS1306 with PIC18 Microcontroller using MSSP Module

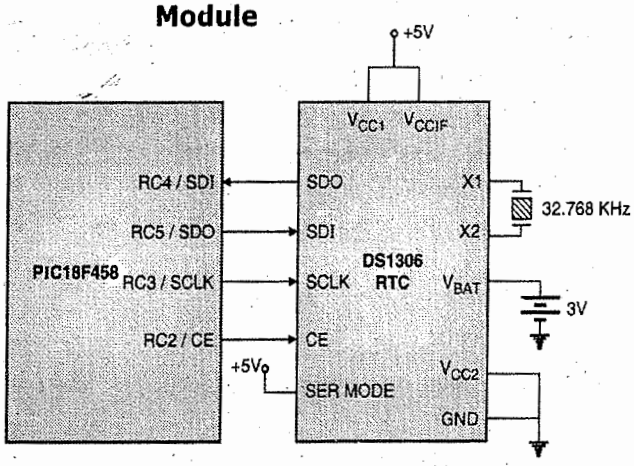


Fig. 14.11.6 : Interfacing DS1306 with PIC18 using SPI

Fig. 14.11.6 shows the interfacing of DS1306 with PIC18 using SPI.

- The DS1306 RTC can communicate using 3 wire interface or SPI interface.

- If SER MODE bit = 1, then the SPI mode is selected.
- The difference between SPI mode and three wire mode is as follows :
  (i) In three wire mode for incoming and outgoing data, there is only one I/O pin whereas in SPI mode SDI and SDO are two separate pins for incoming and outgoing data.
  (ii) In three wire mode each bit is shift such that LSB is shifted first whereas in SPI mode the MSB is shifted first.
- In both the modes an address byte is first written to the DS1306 RTC. The address byte is followed by single or multiple data bytes. As PIC18 supports only SPI mode, we will study it. **After each read or write the RAM register is incremented till the CE signal goes low.**
- The DS1306 determines the clock polarity by sampling the SCLK input when the CE signal is active.
- The MSSP module with SPI consists three registers. They are :
  (i) SSPBUF (ii) SSPSTAT
  (iii) SSPCON1
  We have studied these registers in section 12.9.
- For transferring/receiving data we put in the SSPBUF register.
- The SSPCON1 register is used for selecting the SPI mode operation. For using SPI mode, the SSPEN bit of SSPCON1 register should be set. The SPI master mode should also be selected with the help of SSPM3 : SSPM0 bits of SSPCON1 register. Generally SSPM3 : SSPM0 = 0010 ($f_{osc}/64$) is selected for obtaining best data transfer performance.

Ex. 14.11.1

Write a C18 program for setting the time and date we initialize the clock at 12 : 48 : 52 using 24 hour clock mode and date is set to be 7th July 2014. Firstly we will place the data to be transferred in the SSPBUF register. After writing data to SSPBUF register, we will monitor the BF flag bit of the SSPSTAT register to check whether complete byte is transferred.

Soln. :

Program :

```
#include <P18F458.h>
unsigned char SPI (unsigned char) ;
void delay (int ms) ;
void main ( )
    {
        SSPSTAT = 0 ;
        SSPCON1 = 0x22 ;      // Enable Master SPI, fosc/64
        TRISC = 0 ;       // Make port C an output port
        TRISCbits.TRISC4 = 1 ;        // Set SDI = 1
        TRISCbits.TRISC7 = 1 ;        // Enable RX
        PORTCbits.RC2 = 1 ;     // CE = 1 to enable RTC
        delay (1) ;
        SPI (0x8F) ;       // address of control register
        SPI (0x00) ;       // clear WP bit for write operation
        PORTCbits.RC2 = 1 ;   // Start write operation
        SPI (0x80) ;       // Address of seconds register
        SPI (0x52) ;       // 52 seconds
        SPI (0x48) ;       // 48 minutes
        SPI (0x12) ;       // 12 hour
        SPI (0x2) ;        // Monday
        SPI (0x07) ;       // 7th of month
        SPI (0x07) ;       // July
        SPI (0x14) ;       // 2014
        PORTCbits.RC2 = 0 ;   // end write operation
                              i.e. CE = 0
        delay (1) ;
    }
                              // SPI subroutine
    unsigned char SPI (unsigned char x)
    {
        SSPBUF = x ;   // for data transfer load SSPBUF
        while (!SSPSTATbits.BF) ;       // check BF flag
        return SSPBUF ;          // Return received byte
    }
    void delay (int ms)
    for (i = 0 ; i < ms ; i ++)
    for (j = 0 ; j < 165 ; j ++)
```

Ex. 14.11.2

Write a C18 program for reading the time and date and sending it to the PC screen through the serial port.

Soln. :

```
#include <P18F458.h>
unsigned char SPI (unsigned char) ;
void BCD_ASCII (unsigned char) ;
void delay (int ms) ;
void Serialtransfer (unsigned char) ;
void main ( )
    {
        unsigned char mydata [7] ;
                                    // holds the date and time
        unsigned char x ;
        int a ;
        SSPSTAT = 0 ;
                                    fosc
        SSPSCON1 = 0x22 ;     // SPI master, ----
                                    64
        TRISC = 0 ;       // Make port C an output port
        TRISCbits.TRISC4 = 1 ;          //SDI = 1
        TRISCbits.TRISC7 = 1 ;          // RX = 1
        TXSTA = 0x20 ;          // Enable serial transmit
        SPBRG = 15 ;            // baud rate  = 9600
        RCSTAbits.SPEN = 1 ;  // Enable the serial port
        Serialtransfer (0x0A) ;
        Serialtransfer (0x0D) ;   // New line
        while (1)
        {
            PORTCbits.RC2 = 1 ;  // CE = 1
                                    //start multibyte read
            delay (1) ;
            SPI (0x00) ;   // address of seconds register
            for (a = 0, a < 7 ; a++)
            {
                data [a] = SPI (0x00) ;  // get time/date
                                            and save it.
            }
    PORTCbits.RC2 = 0 ;          //CE = 0 End
                                 //multibyte read
    BCD_ASCII (mydata [4]) ;  // display date
    BCD_ASCII (mydata [5]) ;  // display month
    BCD_ASCII (mydata[6]) ;   // display year
    BCD_ASCII (mydata [2]) ;  // display hour
    BCD_ASCII (mydata [1]) ;  // display minutes
    BCD_ASCII (mydata [0]) ;  // display seconds.
    Serialtransfer (0x0D) ;
        }
    }
    unsigned char SPI (unsigned char x)
    {
        SSPBUF = x ;
        while (! SSPSTATbits.BF) ;       // check BF flag
        return SSPBUF ;
    }
    void Serialtransfer (unsigned char b)
    {
        while (! PIR1bits.TXIF) ;
        TXREG = b ;              // Load the value to be
                                    sent on serial port.
    }
    void BCD_ASCII (unsigned char C)
    {
        x = C ;
        x = x & 0xF0 ;           // Mask lower nibble
        x = x >> 4 ;
        x = x | 0x30 ;           // Make it ASCII
        Serialtransfer (x) ;     // display it.
        x = C ;
        x = x & 0x0F ;           // Mask upper nibble
        x = x|0x30 ;             // Make it ASCII
        Serialtransfer (x) ;     // display it
        Serialtransfer (':')     // display :
    }
    void delay (int ms)
    {
        unsigned int p, q ;
        for (p = 0 ; p < ms ; p++)
        for (q = 0 ; q < 135 ; q++)
    }
```

Syllabus Topic : EEPROM 24LC128 Using I2C

14.12 EEPROM 24LC128 Using I2C

- The EEPROM 24LC128 is a 16K × 8 i.e. 128 Kbit serial electrically erasable programmable ROM (EEPROM).
- It supports a 2 wire serial interface (I2C).
- Fig. 14.12.1 shows the block diagram of 24LC128.
- A device that sends the data on the I2C bus is called as a transmitter while a device that receives the data from the bus is called as a receiver.
- Master device controls the bus. It is also responsible for generating the serial clock (SCL), start and stop conditions.

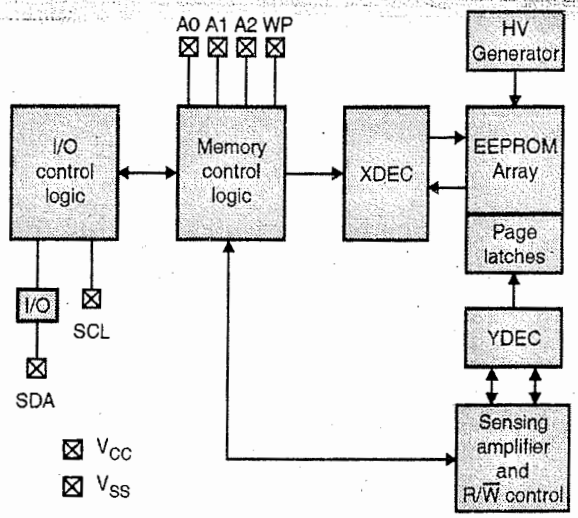


Fig. 14.12.1 : Block diagram of 24LC128

14.12.1 Features of 24LC128

(1) It is a 128 Kbit serial Electrically Programmable (PROM).
(2) It can operate on a single supply ranging from 2.5V to 5.5V.
(3) It supports 2 wire serial interface that is I2C compatible.
(4) It has a 64 byte page write buffer. The typical page write time is 5 ms.
(5) It supports random and sequential reads upto 128 Kbits.
(6) It has a clock compatibility of 100 KHz and 400 KHz.
(7) It supports a self timed erase/write cycle.
(8) 8 devices can be cascaded and handled on the same I2C upto 1MB address space.
(9) It has hardware write protection.
(10) It supports more than 1 million erase/write cycles.
(11) It is available in 8 lead PDIP, SOIC, TSSOP, MSOP, TDFN packages.

14.12.2 Pin Description

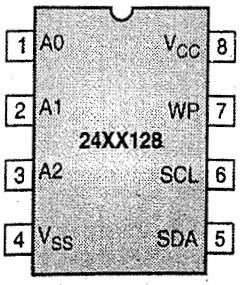


Fig. 14.12.2 : Pin diagram of 24LC128 EEPROM

Pin	Function
V_{SS}	Ground
V_{CC}	+ 2.5V to 5.5V single supply
A0, A1, A2 (Address Inputs)	– These inputs are used for multiple operations on the EEPROM. The input levels of these pins are compared with the corresponding levels of the slave pins. If match is found, the chip is selected. – By using different chip select combinations upto 8 devices can be cascaded on the same bus. – The address lines A0, A1 and A2 are set to logic '0' or '1'.
WP (Write Protect)	– This pin should be connected to V_{SS} or V_{CC}. Write operates are enabled, if WP is connected to V_{SS}. If WP is connected to V_{SS}, write operations are not allowed. However the read operations remain unaffected.
SCL (Serial Clock)	– This input signal is used for synchronizing the data transfer from the EEPROM.
SDA (Serial Data)	– It is a bidirectional pin. It can transfer data and addresses to /from the EEPROM. – It needs an external pull-up resistor as it is open drain. – In conditions of normal data transfer SDA changes its state while SCL is low. – However, state changes when SCL is high are reserved for the START and STOP condition generation.

14.12.3 I2C Bus Protocol

The bus protocol is defined as :

(i) If the bus is free then only data transfer can be initiated.

(ii) Whenever the clock line is high the data line must remain stable for transferring the data. If while the clock line is high and the state of data line changes then it indicates a **start or stop condition**.

Fig. 14.12.3 shows the sequence of data transfer on the bus.

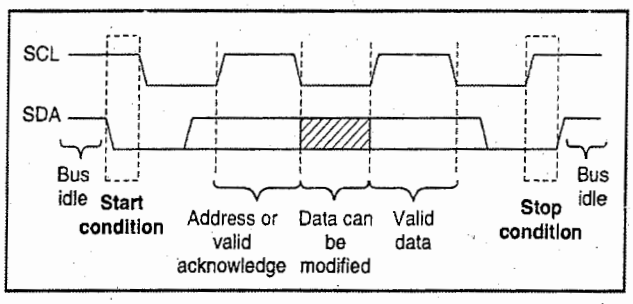


Fig. 14.12.3 : Data transfer sequence on the serial bus

The bus conditions are defined as follows :

1. **Bus not busy :** When the bus is idle both the SCL and SDA lines are high as shown in Fig. 14.12.3 before the start and after the stop condition bus is free.
2. **Start condition :** If there is a change in the state of SDA line when SCL is high, then it is a start condition. All the bus operations begin with a start condition.
3. **Data valid :**If after the start condition, the data line SDA is stable when SCL is high, then it represents valid data as shown in Fig. 14.12.3. If the clock line goes low the data line should be changed.
4. **Stop condition :** If when the clock line is high, the state of the data line changes from low to high then this condition is called as a STOP condition. All data transfer operations end with a STOP conditions.
5. **Acknowledge condition :**
   - After each data byte is received, the master device should generate an additional clock pulse to indicate acknowledgement of the byte received.
   - This action will pull down the SDA line during the acknowledgement clock pulse.
   - At the time of a read operation, the master device should indicate end of data to the slave by NOT generating an acknowledgement bit. The SDA line is left high by slave EEPROM so that master EEPROM can generate the stop condition.

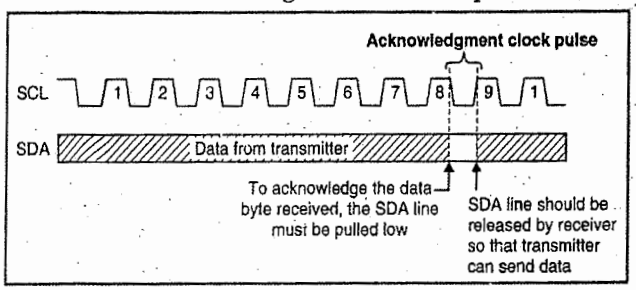


Fig. 14.12.4 : Acknowledge condition

Note : If the internal programming cycle is in process then the 24LC128 EEPROM will not generate any acknowledge bits.

14.12.4 Control Byte Format

- After the start condition, the first byte received is the control byte. Fig. 14.12.4(a) shows the control byte format.

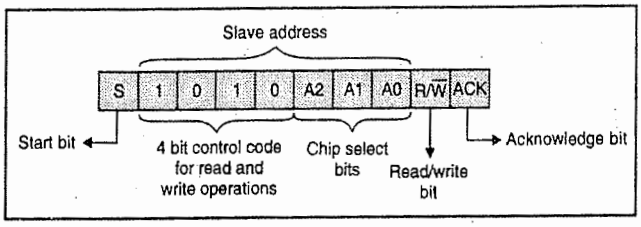


Fig. 14.12.4(a) : Control byte format

- As shown in Fig. 14.12.4(a) the control byte has a 4 bit control code. For 24xx128 EEPROM this code is 1010 for the read and write operations.
- The chip select bits A2, A1, A0 can use upto 8, 24xx128 devices on the same bus, depending on the combination selected.
- The last bit R/$\overline{W}$ bit decides the operation to be done. If R/$\overline{W}$ = 1 then read operation is selected, otherwise write operation is selected. After the control byte the next bytes that are received indicate the 13 bit address of first data byte.

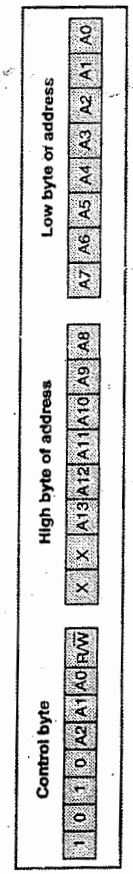


Fig. 14.12.5 : Address assignment of first 3 bytes that are received

After the start condition, the EEPROM 24LC128 observes the data line SDA. When the 4 bit control code and address select bits A2, A1, A0 are received on the slave, it sends an acknowledge signal to the SDA line. Then depending on the R/$\overline{W}$ bit in the control byte, the 24LC128 EEPROM will select a read or write operation.

14.12.5 Byte Write Operation

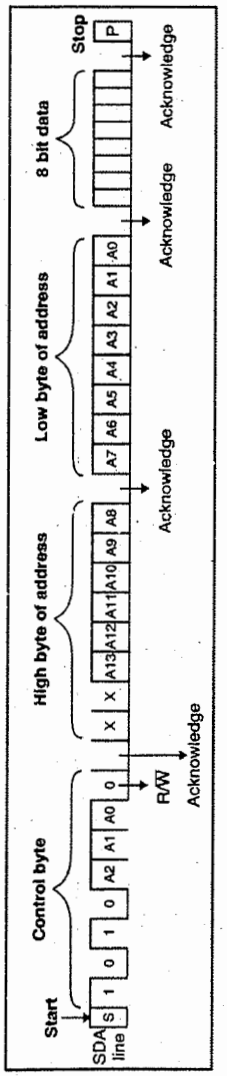


Fig. 14.12.6 : Byte write

- After the start condition, the control byte is transmitted by the EEPROM master transmitter. After the 9th bit i.e. acknowledgement bit the master transmitter will transmit the high byte of address.
- This high byte of address will be written on EEPROM 24LC128's address pointer.
- The high byte of address is followed by acknowledge bit. On receiving the acknowledge bit the master transmitter lower byte of address.
- On receiving the acknowledge bit the master transmitter will transmit the data byte to be written to addressed memory location.
- After receiving the acknowledgement bit the master generates a stop condition.
- This initializes the write cycle and EEPROM 24LC128 will not generate any acknowledgement signals during the time the write cycle is in progress.
- If WP = 1, and an attempt is done to write to the array then the command will be acknowledged. However, no write cycle will proceed and data will not be written. The master device will accept the new command.
- The internal address counter points to the written address location after the byte write command.

14.12.6 Page Write

- The control byte and first data byte are transmitted in the page write operation similar to the byte write operation. However, in the page write operation after the first data byte, 63 more additional bytes can be transmitted as shown in Fig. 14.12.7. Then the stop bit is transmitted. On transmission of stop bit the write cycle will begin and data will be written.
- The 63 bytes are stored in the on-chip page buffer. After the STOP condition is transmitted by the master 24xx128, the bytes will be written on to the desired memory locations.
- After every word is received, the address pointers lower six bits are automatically incremented by '1'.
- If it is desired for the master device to transmit more than 64 bytes, then before the stop condition is generated the counter will over. The data that was received will be over written.
- If WP = 1, and an attempt is done to write to the array then the command will be acknowledged. However, no write cycle will proceed and data will not be written. The master device will accept the new command.

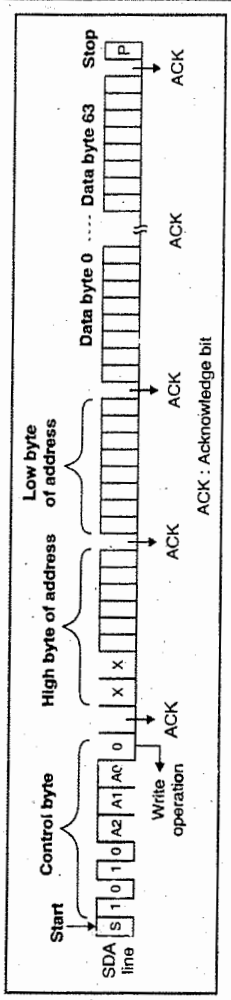


Fig. 14.12.7 : Page write operation

14.12.7 Write Protection

- If WP = 1 i.e. it is tied to V_{CC} the user can write protect the complete array from 0000-3FFFH. However, if WP = 0 i.e. it is tied to V_{SS} the write protection is disabled.
- For each write command, the WP pin is sampled at every stop bit.

14.12.8 Read Operation

- The EEPROM 24xx128 supports three read operations. They are :

(i) Current address read (ii) Random read
(iii) Sequential read

14.12.8.1 Current Address Read

- The EEPROM 24LC128 has an address counter. The **function** of the address counter is to hold the address of the last word that is accessed. It is automatically incremented by 1.
- When the control byte is received the EEPROM master acknowledges it and sends the 8 bit data word. This transfer is not acknowledged and the master 24xx128 stops transmission.

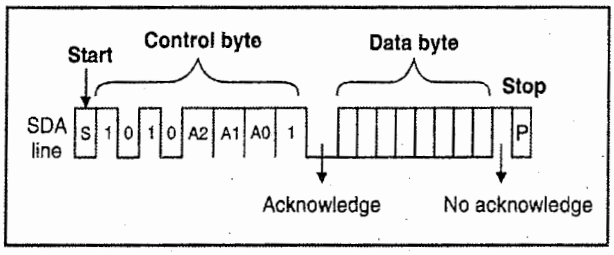


Fig. 14.12.8 : Current Address Read

14.12.8.2 Random Read

Fig. 14.12.9 shows the random read operation. This operation allows the master device to access any memory location randomly.

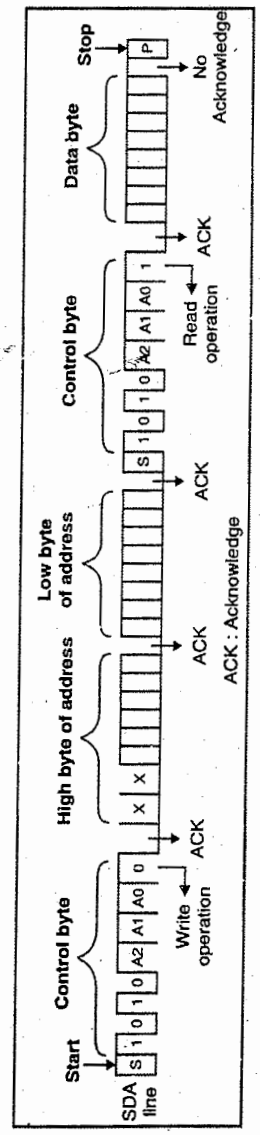


Fig. 14.12.9 : Random read operation

- The word address should be set for doing a random read operation. The word address is set by byte write operation as shown in Fig. 14.12.9.
- After the word address is sent, a start condition is again generated by the master. The master sends control byte and sets $R/\overline{W}$ = 1 to indicate read operation.
- The EEPROM master sends acknowledge bit and transmits the data byte. The master will not acknowledge the transfer and generates a stop condition for ending the data transfer.
- The address counter points to the address location next after the address location that is read.

14.12.8.3 Sequential Read

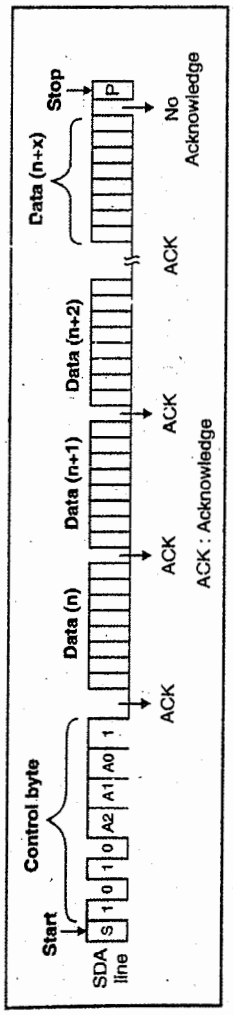


Fig. 14.12.10 : Sequential read operation

- Like the random read operation in sequential read operation the EEPROM transmits the first data byte and receives an acknowledgement.
- On receiving the acknowledgement the master EEPROM transmits the next 8 bit data words. On transmitting the last data word the master does not acknowledge the transfer and generates a STOP condition for indicating the end of data transfer.
- After each read operation the address counter is incremented by 1.
- When master acknowledges the byte received at array address 03FFFH. It rolls over to the address counter to 0000H.

14.13 PIC Interfacing to EEPROM 24LC128 using MSSP Module

Fig. 14.13.1 shows the PIC interfacing with EEPROM.

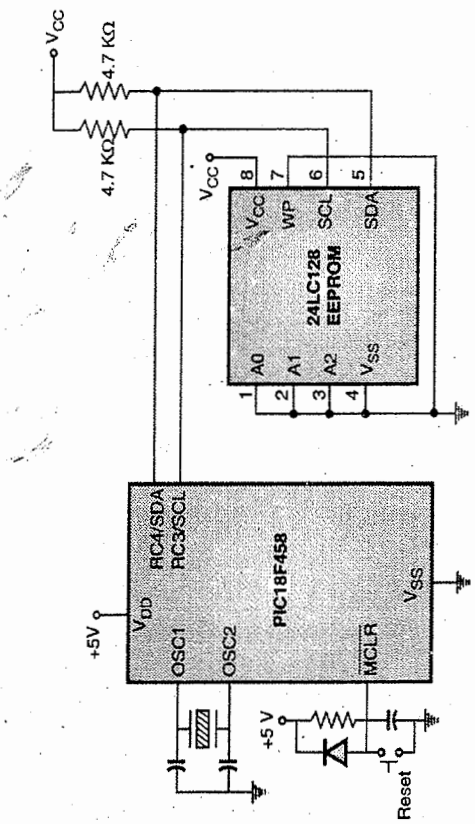


Fig. 14.13.1 : Interfacing PIC18 to EEPROM 24LC128 with I2C

- As shown in Fig. 14.13.1 the SCL and SDA pins are open drain. Hence, they need pull up registers.

- The MSSP module inside the PIC18 supports I2C bus protocol. The registers associated with this protocol are SSPBUF, SSPCON1, SSPCON2, SSPSTAT, SSPADD and SSPSR as studied in section 12.10.

Ex. 14.13.1 **Lab assignment**

Interface EEPROM 24C128 using I2C to store and retrieve data.

Soln. : Fig. 14.13.1 shows the interfacing diagram.

Program :

```
#include<P18F458.h>
void idleI2C ( ) ;
void startI2C ( ) ;
void ReadI2C( ) ;
void writeI2C (unsigned char dat)
void stopI2C ( ) ;
void RestartI2C ( ) ;
unsigned char EEPROMwrite (unsigned char control,
unsigned char address, unsigned char data)
        void main (void)
    {
        SSPSTAT & = 0x3F ;
        TRISCbits.RC3 = 1 ; // SCL = input   } Initialize I2C
        TRISCbits.RC4 = 1 ; // SDA = input   }
        SSPCON1 = 0x00 ;
        SSPCON2 = 0x00;
        SSPCON1 = 0x28 ; // Enable serial communication
                         // and select I2C master mode
        idleI2C ( ) ;        // Ensure that I2C bus is idle.
        startI2C ( ) ;       // Initiate a start condition
        while (SSPCON2bits.SEN) ;      // Wait till start
                                 // condition is over
        if (PIR2bits.BCLIF)      // Check for bus collision
        Return (– 1) ;           // Return with bus
                                 collision error
        WriteI2C (control) ; // Write control word

                         // R/W = 0 for write operation
        idleI2C( ) ;         // insure if mode is idle
        if (SSPCON2bits.ACKSTAT)
                         // Check for acknowledge status bit
        return (– 2) ;           // Return NACK error
        idleI2C ( ) ;
        writeI2C (address) ;     // Write word address i.e.
                         // store address on EEPROM
        idleI2C ( ) ;
        if (SSPCON2bits.ACKSTAT)
        return (– 2) ;
        writeI2C (data) ;        // Store data on EEPROM
        idleI2C ( ) ;
        stopI2C ( ) ;            // Send stop condition
        while (SSPCON2bits.PEN) ;
                         // Wait till stop condition is over.
        RestartI2C ( ) ;     // assert a I2C bus restart
                         condition.
        readI2C( ) ;
        while (SSPCON2bits.RCEN) ;          //
        NotACKI2C ( ) ;          // Send not acknowledge
                                 condition
        while (SSPCON2bits.ACKEN) ;
                 // wait till acknowledge sequence is over
        stopI2C ( ) ;            // send stop condition
        while (SSPCON2bits.PEN) ;
                         // wait till stop condition is over
        return ((unsigned int) SSPBUF) ; // Retrieve data
    }
    void NotACKI2C( )
    {
        SSPCONbits.ACKDT = 1 ;
        SSPCONbits.ACKEN = 1 ;
    }
        void idleI2C (void)
        {
            while ((SSPCON2 & 0x1F) |
(SSPSTATbits.R_W)) ;
        }
        void startI2C ( )
        {
            SSPCON2bits. SEN = 1;
        }
        void ReadI2C ( )
        {
            SSPCON2bits.RCEN = 1 ;
            while (! SSPSTATbits.BF) ;
                         // wait till a byte is received
            return (SSPBUF) ;
        }
    void writeI2C (unsigned char dat)
    {
    idleI2C ( ) ;
    SSPBUF = dat ;
    while (SSPSTATbits.BF) ;
    }
    void stopI2C ( )
    {
    SSPCONbits.PEN = 1 ;
    }
    void RestartI2C ( )
    {
        SSPCON2.RSEN = 1 ;  //assert a repeated
                                 start condition.
    }
```

Syllabus Topic : Design PIC Test Board

14.14 Design PIC Test Board

Fig. 14.14.1 shows PIC18F458 development test board that comprises of the following features:

- 32KB internal Flash Program Memory
- Breadboard for design , testing and development
- 10 MHz crystal frequency
- I/O pins for external connections
- In-circuit programming through computer with the help of download cable .
- RS232 Communication with on-board MAX232 or equivalent
- Test LED for testing the programs that are run on board
- Power LED and Reset Button
- Download Software

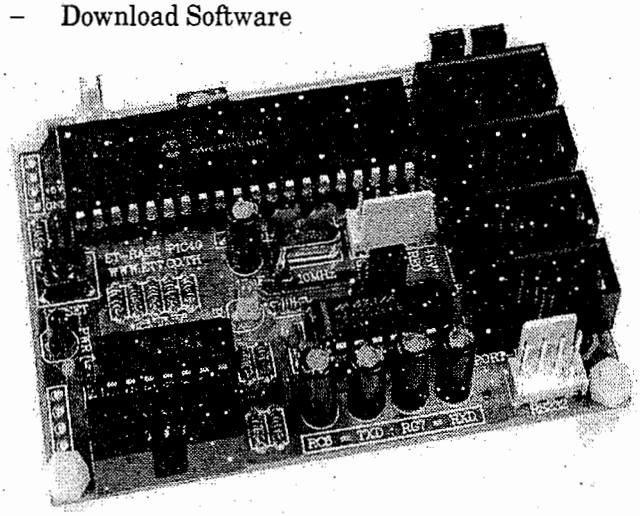


Fig. 14.14.1

Syllabus Topic : Home Protection System

14.15 Home Protection System

- Home protection systems provide protection against damage, loss, danger and crime Homes that do not have security or protection systems serve as target to property crimes. Thefts can occur during the daytime when we are not at home or homes can be broken while you are inside. Installing a home protection / security system ensures us to be safe at any time of the day / night.
- Fig. 14.15.1 shows a simple security alarm system that uses PIR (Passive Infrared Sensor). The sensor detects human motion by sensing the atmospheric temperature variations. The PIR sensor can operate in darkness also. The human motion detected by the PIR sensor will activate the PIR sensor output which in turn will activate the external lamps, or alarm sirens. In this way the burglar / intruder who entered the home will be exposed.
- The output of PIR sensor module is given to RC0 pin of PIC18F458. When motion is sensed, this output is approximately 3.3V which turns on the relay and buzzer. The buzzer is connected to RB0. The relay is connected to pin RB1 for energizing the relay ULN2803 driver is used.

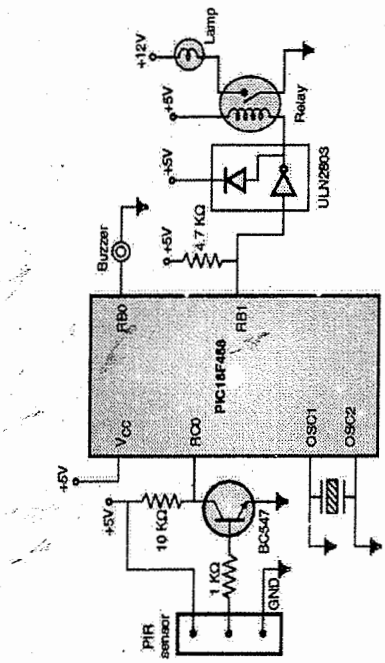


Fig. 14.15.1 : Home Protection System Using PIC

Program

```
# include < P18F458.h >
# define PIRsensor PORTCbits. RC0
# define relay PORTBbits. RB1
# define buzzer PORTBbits. RB0
        void delay (unsigned int) ;
        void main (void)
{
        TRISCbits, TRISC0 = 1 ;    // Make RC0 an input
        TRISB = 0 ;                // Make Port B an output port
        while (1)
        {
            if (PIRsensor = = 1)
```

```
{
        buzzer = 1 ;                    // Turn on buzzer
        relay = 1 ;                     // Turn on delay
        delay (10) ;
}
        elseif (PIRsensor = = 0)
{
        buzzer = 0 ;                    // Turn off buzzer
        relay = 0 ;                     // Turn off relay
```

```
        }
    }
}
void delay (unsigned int itime)
{
        unsigned int x ;
        unsigned char y ;
        for (x = 0 ; x < itime ; x + + )
        for (y = 0 ; y < 165 ; y + + ) ;}
```

❑❑❑

Lab Manual

List of Experiments

Program 1 : Simple programmes on Memory transfer.
Ans. : **Please refer Program 1.33.11 (Pg. 1-93), Program 1.13.15(Pg. 1-96), Program 1.13.16(Pg. 1-96), Program 1.13.18 (Pg. 1-97)**

Program 2 : Parallel port interacting of LEDs-Different programs(flashing, Counter, BCD, HEX, Display of Characteristic)
Ans. : **Please refer Program Ex. 2.1.2 (Pg. 2-2) and Ex. 2.6.1 (Pg. 2-13)**

Program 3 : Waveform Generation using DAC
Ans. : **Please refer Ex. 3.1.1 (Pg. 3-2)**

Program 4 : Interfacing of Multiplexed 7-segment display (counting application)
Ans. : **Please refer Ex. 3.9.3 (Pg. 3-31)**

Program 5 : Interfacing of LCD to 8051 (4 and 8 bit modes)
Ans. : **Please refer Ex. 2.7.2 (Pg. 2-18) and Ex. 2.7.4 (Pg. 2-20)**

Program 6 : Interfacing of Stepper motor to 8051- software delay using Timer
Ans. : **Please refer Ex. 3.3.6 (Pg. 3-15)**

Program 7 : Write a program for interfacing button, LED, relay and buzzer as follows
A. On pressing button1 relay and buzzer is turned ON and LED's start chasing from left to right
B. On pressing button2 relay and buzzer is turned OFF and LED start chasing from right to left .
Ans. : **Please refer Ex. 10.6.9 (Pg. 10-29)**

Program 8 : Interfacing 4 × 4 keypad and displaying key pressed on LCD.
Ans. : **Please refer Ex. 10.4.1 (Pg. 10-4)**

Program 9 : Generate square wave using timer with interrupt
Ans. : **Please refer Ex. 9.8.1 (Pg. 9-4)**

Program 10 : Interfacing serial port with PC both side communication.
Ans. : **Please refer Ex. 12.18.4 (Pg. 12-43)**

Program 11 : Interfacing EEPROM 24C128 using I2C to store and retrieve data
Ans. : **Please refer Ex. 14.13.1 (Pg. 14-19)**

Program 12 : Interface analog voltage 0-5V to internal ADC and display value on LCD
Ans. : **Please refer Ex. 13.3.5 (Pg. 13-10)**

Program 13 : Generation of PWM signal for DC Motor control.
Ans. : **Please refer Ex. 11.7.2 (Pg. 11-12)**

❑❑❑

SUBJECT CODE : 304184

Choice Based Credit System

SAVITRIBAI PHULE PUNE UNIVERSITY - 2019 SYLLABUS

T.E. (E&Tc) Semester - V

MICROCONTROLLER

(For END SEM Exam - 70 Marks)

Dr. Dnyaneshwar S. Mantri
Ph.D. (Wireless Communication)
Professor,
Department of E&TC Engg.
Sinhgad Institute of Technology, Lonavala
Senior Member IEEE, LMISTE, FIETE

FEATURES

- ☑ Written by Popular Authors of Text Books of Technical Publications
- ☑ Covers Entire Syllabus
- ☑ Question - Answer Format
- ☑ Exact Answers and Solutions
- ☑ Important Points to Remember
- ☑ Chapterwise Solved SPPU Questions May-2011 to Oct.-2022

SOLVED SPPU QUESTION PAPERS

- May - 2018
- Aug. - 2018
- Dec. - 2018
- May - 2019
- Aug. - 2019
- May - 2022
- Aug. - 2022
- Oct. - 2022

DECODE®

A Guide For Engineering Students

DECODE®

A Guide For Engineering Students

MICROCONTROLLER

(For END SEM Exam - 70 Marks)

SUBJECT CODE : 304184

T.E. (Electronics & Telecommunication Engineering) Semester - V

Published by :

TECHNICAL PUBLICATIONS® SINCE 1993 An Up-Thrust for Knowledge

Amit Residency, Office No.1, 412, Shaniwar Peth,
Pune - 411030, M.S. INDIA Ph.: +91-020-24495496/97
Email : info@technicalpublications.in
Website : www.technicalpublications.in

Printer :
Yogiraj Printers & Binders, Sr.No. 10/1A,Ghule Industrial Estate, Nanded Village Road,
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-93-5585-134-5

9 789355 851345

SPPU 19

SYLLABUS

Microcontroller - (304184)

Credit :	Examination Scheme :
03	End Sem (Theory) : 70 Marks

Unit III PIC 18F XXXX Microcontroller Architecture

Comparison of PIC family, Criteria for Choosing Microcontroller, features, PIC18FXXXX architecture with generalized block diagram. MCU, Program and Data memory organization, Bank selection using Bank Select Register, Pin out diagram, Reset operations, Watch Dog Timers, Configuration registers and oscillator options (CONFIG), Power down modes, Overview of instruction set. **(Chapter - 4)**

Unit IV Peripheral Support in PIC 18FXXXX

Brief summary of Peripheral support, Timers and its Programing (mode 0 &1), Interrupt Structure of PIC18FXXXX with SFR, PORTB change Interrupts, use of timers with interrupts, CCP modes : Capture, Compare and PWM generation, DC Motor speed control with CCP, Block diagram of in-built ADC with Control registers, Sensor interfacing using ADC : All programs in embedded C. **(Chapter - 5)**

Unit V Real Word Interfacing With 18FXXXX

Port structure with programming, Interfacing of LED, LCD and Key board, Motion Detectors, Gas sensors, IR sensors, Design of PIC test Board and debugging, **Home protection System :** All programs in embedded C. **(Chapter - 6)**

Unit VI Serial Port Programming interfacing with 18FXXXX

Basics of Serial Communication Protocol : Study of RS232, RS 485, I2C, SPI, MSSP structure (SPI & I2C), USART (Receiver and Transmitter), interfacing of RTC (DS1307) with I2C and EEPROM with SPI. Design of Traffic Light Controller; All programs in embedded C. **(Chapter - 7)**

TABLE OF CONTENTS

Unit III

4 PIC 18FXXXX Microcontroller Architecture

Q.1 Compare the different families of PIC microcontroller

[SPPU : In Sem : Aug.-16, Marks 5]

Ans. : Comparison of PIC families : The PIC microcontroller series has Harvard architecture and supports for RISC instruction set rather than CISC. It is single chip microcontroller developed by microchip and specifically used in embedded system development. Most of the instructions used in PIC are either 2 or 4 bytes instead of 1 and 3 bytes.

The variation and comparison of various family members of PIC according to number of bits, manufactures and supporting components as RAM, ROM, ADC channels, ports is given in Table Q.1.1 and Q.1.2.

Device	Pins	Digital I/O	ADC Channel	EPROM × 12 words	RAM [Bytes]
16C54	18	12	None	512	25
16C55	28	20	None	512	24
16C56	18	12	None	1 K	25
16C57	28	20	None	2 K	72
17C42A	40	33	None	2 K	232
17C43	40	33	None	4 K	454
17C44	40	33	None	8 K	454
16C7	18	13	4 (8 bit ADC)	1 K×14	36
17C752	40	33	12 (10 bit ADC)	8 K×16	678

Table Q.1.1 : Comparison of PIC 16 & 17 Family

Device	Program Memory		Data Memory		I/O	10-bit A/D (channel)	Comparators	CCP/ ECCP (PWM)	MSSP		USART	Timers 8/16 Bit
	Flash (bytes)	Single word Instructions	SRAM (bytes)	EEPROM (bytes)					SPI	Master I²C		
PIC18 F248	16 K	8192	768	256	22	5	-	1/0	Y	Y	Y	1/3
PIC18 F258	32 K	16,384	1536	256	22	5	-	1/0	Y	Y	Y	1/3
PIC18 F448	16 K	8192	768	256	33	8	2	1/1	Y	Y	Y	1/3
PIC18 F458	32 K	16,384	1536	256	33	8	2	1/1	Y	Y	Y	1/3

Table Q.1.2 : Comparison of PIC18FXXX

Q.2 Draw and explain architecture of PIC 18FXXXX.

☞ **[SPPU : Dec-17, Marks 8, May-16, 15, 14, Marks 8]**

Ans. : It is CMOS flash-based 8-bit microcontroller developed by Microchip's and packed into 40 or 44-pin package and is upwards compatible with the PIC16C5X, PIC12CXXX, PIC16CXX and PIC17CXX with higher levels of hardware integration. It is high-performance, enhanced flash microcontrollers with CAN. It uses Harvard architecture with RISC instruction set architecture. The PIC18F4520 features a 'C' compiler friendly development environment, 256 bytes of EEPROM, self-programming, an ICD, 2 capture/compare/PWM functions, 8 channels of 10-bit Analog-to-Digital (A/D) converter, the synchronous serial port can be configured as either 3-wire Serial Peripheral Interface (SPI) or the 2-wire Inter-Integrated Circuit (I^2C) bus and Addressable Universal Asynchronous Receiver Transmitter (AUSART).

All of these features make it ideal for manufacturing equipment, instrumentation and monitoring, data acquisition, power conditioning, environmental monitoring, telecom and consumer audio/video applications. It also supports for pipelining functionality with increased speed of operation. It has powerful instruction set with limited addressing modes ? It works on the clock ranging from DC to 40 MHz and requires only one cycles to execute the instructions.

The generalized block diagram of each PIC microcontrollers are designed using the Harvard architecture as shown in Fig. Q.2.1 which includes :

- Microprocessor unit (MPU) -- Accumulator along with general purpose register and file select registers
- Program memory for instructions - ROM
- Data memory for data-- RAM
- I/O ports -- A, B, C, D, E
- Support devices such as timers/counters, ADC, USART etc.

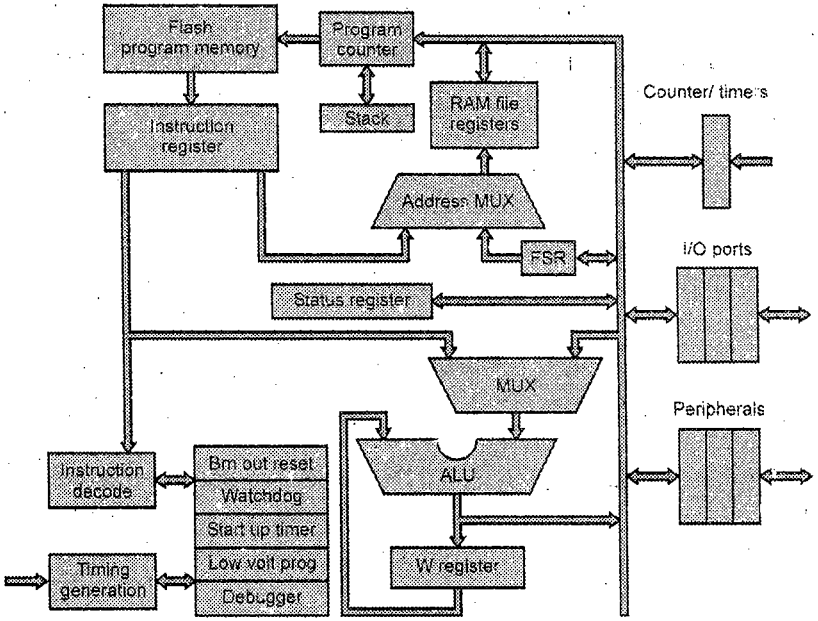


Fig. Q.2.1 : General architecture of PIC 18FXXXX

A. Microprocessor unit (MPU) : It consists of working register (W) - 8 bit long, along with the status register. Instruction decoder to perform the operation on data and speed up the operation. In all PIC has 15 banks and accessed by use of Bank

Select Register (BSR). Data can be stored in the file register or working register. The control unit synchronizes the operation of ALU while 21 bit program counter is used to access the 2 Mbytes of ROM.

B. **Program memory for instructions :** 21 bit PC is used to access the program ROM and stores the 16 bits of the instructions. In general instructions used by PIC are of 2 or 4 bytes long.

C. **Data memory for data :** The machine code for a PIC18 instruction has only 8 bits for a data memory address which needs 12 bits. The Bank Select Register (BSR) supplies the other 4 bits. The total data memory is organized into 15 banks with each of 256 bytes storage capacity. Upper 128 bytes of bank 15 and lower 128 bytes of bank 0, are used as access banks irrespective of bank selection. Data memory can be addressed directly or indirectly.

D. **I/O ports :** 33 I/O lines amongst 5 ports are used for communication with outside environment. These ports are A (6 - Bits), B, C and D (8 - Bits), E (3 - bits), they have direct locations in the memory as SFRS.

E. **Support devices such as timers :** Has four timers and PWM mode working on 8 or 16 bit and used for the application as delay generation.

F. **Support for serial communication :** It has the USART port to receive and communicate the data serially. It is also supported by many other devices as 10 bit ADC, CCP modes and so on.

The ALU plays important role in functioning of PIC, according to Fig. Q.2.1, The data from file register available in ROM is added with working register and may be stored in file or W register. The device can be reset by internal or external interrupt generated due to low voltage, in-sufficient clock, Power on reset circuit not providing

required delay or due to watch dog timers. The instruction register provide the code to fetch the data from data memory or file register. The PLL circuit provides the option of multiplying up the oscillator frequency to speed up the overall operation. The watch dog timer can be used to restart the controller under program crash or uneven execution of subroutines. The in-circuit debugger helps during program development to diagnostic data and communicate to processor.

Q.3 State features of the PIC 18FXXXX.

[SPPU : May-22, Marks 6, May-17, Marks 3, In Sem : Aug.-16, Marks 5]

Ans. : **Features of PIC18FXX** : The general features of PIC 18FXXX Microcontroller are as follows :

- Data Bus : 8-bit CPU With RISC architecture
- Clock : DC to 20 MHz
- Instructions : 16 bits
- Memory : 2 Mbytes of program ROM [21 Address line]
  - 4 kbytes of data RAM [12 Address lines]
  - 32 K flash ROM
  - 1536 bytes SRAM - Scratch Pad
  - 256 bytes - EEPROM - For storing critical information
- I/O Ports : 5 [A(6), B,C,D (8), E(3)] - 33 bidirectional and individually addressable I/O lines.
- Timers : Five timers with 8 and 16-bit operation
- Has 15 bank registers with 256 entries
- Has GPR [variable] and SFR [fixed locations]
- Supports for USART operation
- 10 bit, 8 channel ADC
- CCP modules.
- I^2C/SPI serial port

- High current sink/source 25 mA/25 mA
- 4x Phase Lock Loop (PLL) of primary oscillator
- In-circuit debugger
- Max. PWM freq. @ : 8-bit resolution = 156 kHz
  10-bit resolution = 39 kHz
- Supports SPI and I2C mode for serial communication
- Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
- Wide operating voltage range (2.0 V to 5.5 V)

Q.4 Explain with example functioning of ALU in PIC18 for transfer of data. ☞ **[SPPU : May-22, Marks 6, May-18, 17, Marks 8]**

Ans. : Arithmetic and Logic Unit (ALU) : The function of register is to support the functionality of ALU for performing various arithmetic and logical operations. It is a general-purpose register used for storing intermediate results obtained during Arithmetic Logic Unit (ALU) :

- WREG - Working register-Not addressable.
- Status register that stores flags.
- Instruction decoder - when the instruction is fetched it goes into the ID.
- The function of the ALU is shown in the Fig. Q.4.1. While performing operation on ALU one of the operand is from the program data memory multiplexed with other input from SFRS and other is working register (W). The result of operation may be stored either in working register or in file register according to the direction bit'd'. If d = 0, result will be stored in working register while d = 1, it will be stored in file register. The sample is explained with example as,

```
ADDWF  F, d, a    ; Add WREG to File (Data) Reg.
                  ; Save result in W if d = 0
                  ; Save result in F if d = 1
```

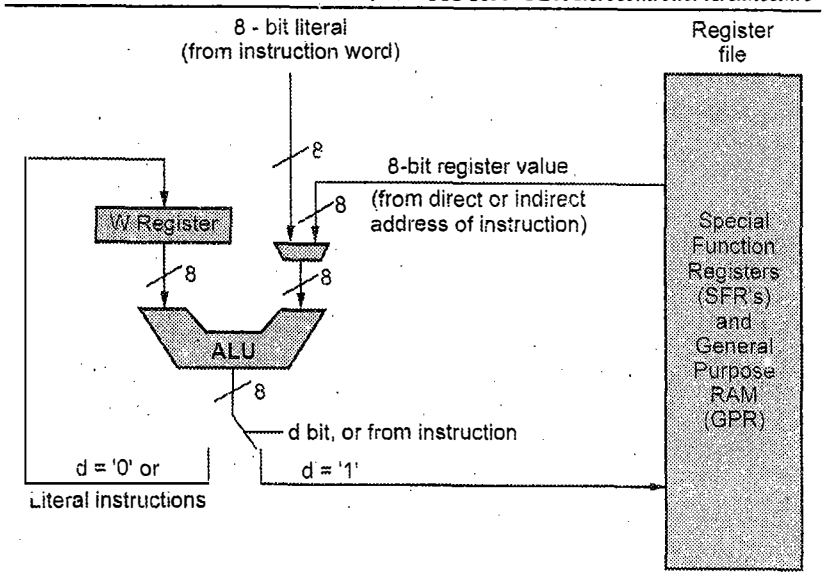



Fig. Q.4.1 : ALU function

Product : 16-bit Product of 8-bit by 8-bit Multiply :

- 8x8 hardware multiplier in ALU is used for Multiplication operation and stores 16 bit result in product register pair(PRODH : PRODL) without affecting flags.
- The CPU fetches instructions from memory, decodes them and passes them to the ALU for execution.
- The Arithmetic Logic Unit (ALU) is responsible for adding, subtracting, shifting and performing logical operations.
- The ALU operates in conjunction with :–
  - A general purpose register called 'W' register.
  - And 'F' register that can be any location in data memory.
  - Literal embedded in the instruction code.

Q.5 Explain the flag structure (PSW) of PIC in detail

☞ **[SPPU : Aug.-15, May-14, Marks 6, April-13, Marks 8]**

Ans. : Program Status Word (PSW) **:** Flags are 1-bit registers used to store the result of program. The Program Status Word (PSW) contains status bits that reflect the current CPU state after arithmetic and logical operations. PIC has 5 math flags (N, OV, Z, DC, and C). The general structure of PSW is as shown in Fig. Q.5.1.

PSW Register of PIC

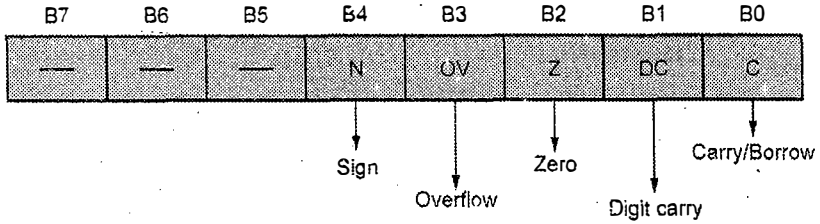


Fig. Q.5.1 : PSW of PIC

- **N (Negative flag) :** Used to indicate the result of an arithmetic/logic operation.(B7 = 1 : result Negative, B7 = 0 result Positive)
- **OV (Overflow flag) :** Indicate the operation of sign magnitude numbers, set when goes beyond 7-bits result of an operation of signed numbers.
- **Z (Zero flag) :** Set when result of an arithmetic and logical operation is zero.
- **DC (Digit Carry flag) (Half Carry) :** Set when carry generated from bit 3 to bit 4 in an arithmetic operation. It is used by BCD arithmetic instructions.
- **C (Carry flag) :** Set when an addition of unsigned number generates a carry (ADDLW, ADDWF, SUBLW, and SUBWF). For subtraction operation polarity is reversed and performed by adding 2's complement of second operand.

Q.6 Explain data memory organization of PIC, comment on bank select register and access banks.

[SPPU : Dec.-17, Marks 8, May-19, Marks 6, Nov.-15, May-16,17, Marks 8, In Sem : 16, Marks 5, April 12, 13]

Ans. : Data Memory Organization : The PIC 18F4550 has 4 KB of data memory organized in 16 banks, the banks are access by Bank Select Register (BSR). The data bus is of 8 bit and will be stored in memory, as file register. The detailed data organization of data memory is shown in Fig. Q.6.1.

- Data memory up to 4 kbytes : **Data register map - with 12-bit address bus 000-FFF :**
- Divided into 16 banks each of 256-byte (FFF = 2^{12} = 16 × 256 = 4096 = 4 K)
- Half of bank 0 and half of bank 15 form a virtual bank that is accessible no matter which bank is selected
- Data memory also known as register file, these registers are always accessible regardless of which bank is selected - acting as virtual memory.
- BSR holds 4 bit bank address 0-F and remaining 8 bits points to 256 locations in selected bank.
- The BSR can be loaded directly by using the MOVLB instruction.
- In the core PIC18 instruction set, only the MOVFF instruction fully specifies the 12-bit address of the source and target registers access bank.
- While the use of the BSR, with an embedded 8-bit address, allows users to address the entire range of data memory, it also means that the user must always ensure that the correct bank is selected.
  - To streamline access for the most commonly used data memory locations, the data memory is configured with an access bank, which allows users to access a mapped block of memory without specifying a BSR.

- The access bank is used by core PIC18 instructions that include the access RAM bit (the "a" parameter in the instruction).
- When "a = 1", the instruction uses the BSR and the 8-bit address included in the opcode for the data memory address.
- When "a = 0", the instruction is forced to use the access bank address map; the current value of the BSR is ignored entirely.

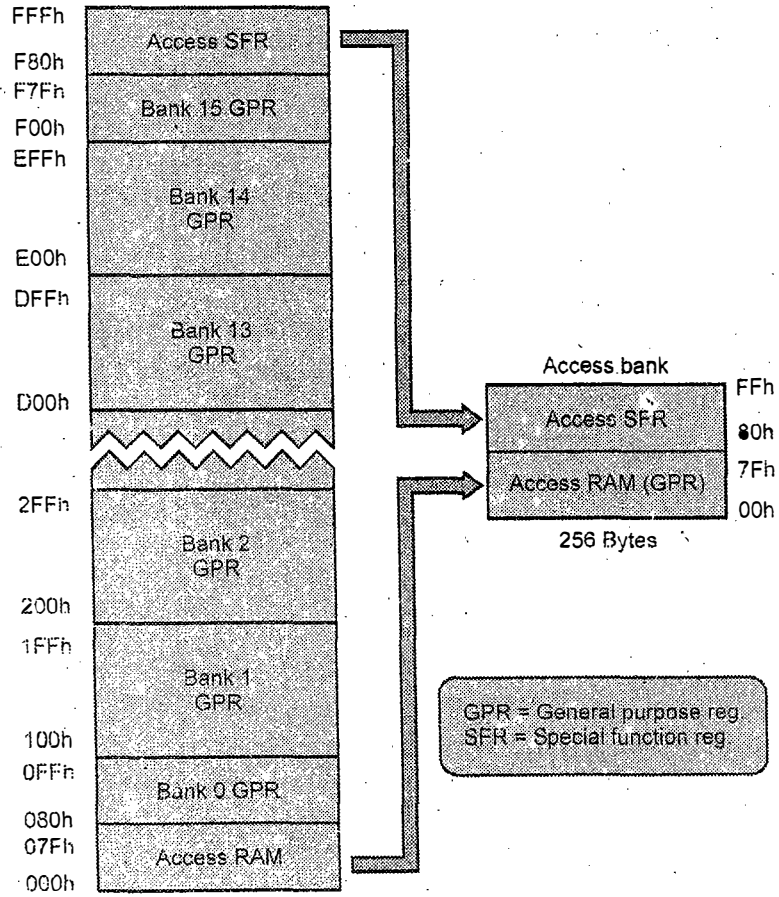


Fig. Q.6.1 : Data memory organization

Accessing data memory

- The machine code for a PIC18 instruction has only 8 bits for a data memory address which needs 12 bits. The Bank Select Register (BSR) supplies the other 4 bits as shown in Fig. Q.6.2.

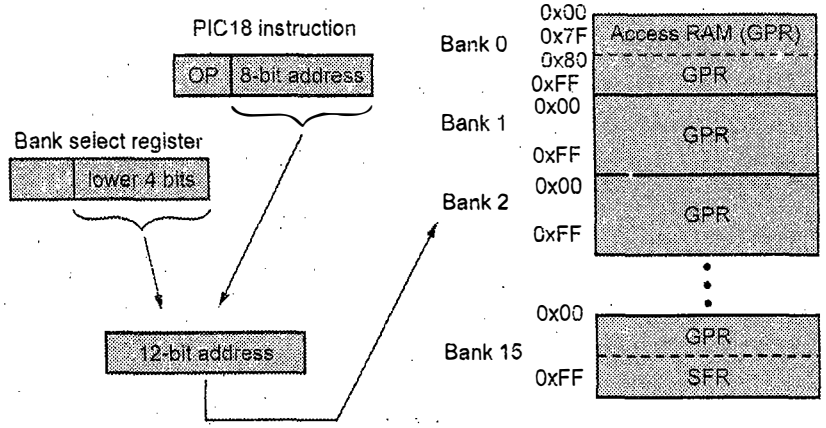


Fig. Q.6.2 : Use of BSR for bank selection

- Memory can be addressed directly and indirectly as shown in Fig. Q.6.3 and Fig. Q.6.4.

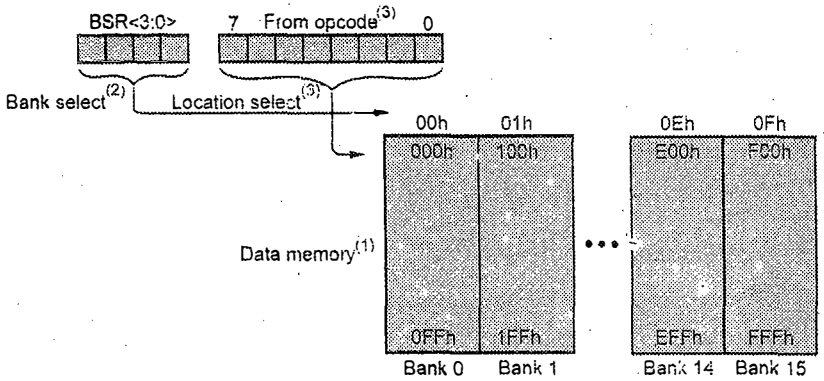


Fig. Q.6.3 : Memory addressing

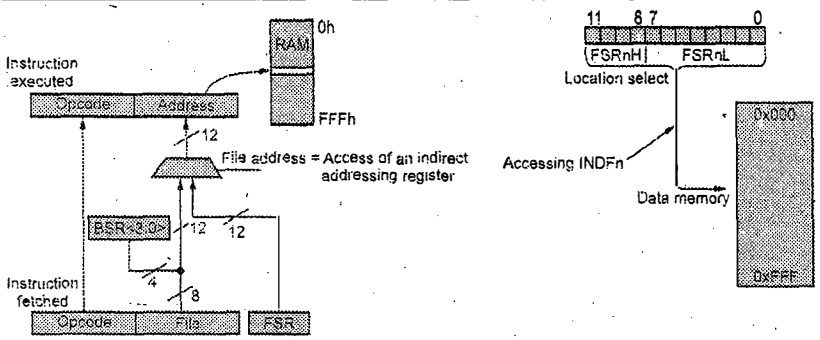


Fig. Q.6.4 : Indirect addressing

Data memory addressing - Direct

- 8 bits of the 16-bit instruction specify any one of 256 locations.
- The 9th bit specifies either the access bank (= 0) or one of the banks (= 1).

Data memory addressing - Indirect

- 3 File Select Registers (FSR) as a pointer to the data memory location that is to be read or written.
- Each FSR has an INDF register associated with it.
- The INDFn register is not a physical register. Addressing INDFn actually addresses the register whose address is contained in the FSRn register.

  FSR0 : Composed of FSR0H : FSR0L

  FSR1 : Composed of FSR1H : FSR1L

  FSR2 : Composed of FSR2H : FSR2L

Q.7 Draw and explain the program memory map and stack of PIC microcontroller.

☞ **[SPPU : May-22, Marks 6, In Sem : Aug-22, Marks 5, April-12]**

Ans. : Program Memory Organization

- PIC18 microcontrollers implement a 21-bit program counter, which is capable of addressing a 2-Mbyte program memory space. Accessing a location between the upper boundary of the physically implemented memory and the 2-Mbyte address will return all '0's (a NOP instruction).
- PIC18 devices have two interrupt vectors. The Reset vector address is at 0000H and the interrupt vector addresses are at 0008H and 0018H as shown in Fig. Q.7.1.
- There may be programming situations that require the creation of data structures, or look-up tables, in program memory. For PIC18 devices, look-up tables can be implemented in two ways : 1. Computed GOTO, 2. Table Reads.
- The Program Counter (PC) specifies the address of the instruction to fetch for execution.
- The PC is 21 bits wide and is contained in three separate 8-bit registers.
  - The low byte, known as the PCL register, is both readable and writable.
  - The high byte, or PCH register, contains the PC<15 : 8> bits; it is not directly readable or writable
  - The upper byte is called PCU. This register contains the PC<20 :16> bits; it is also not directly readable or writable.

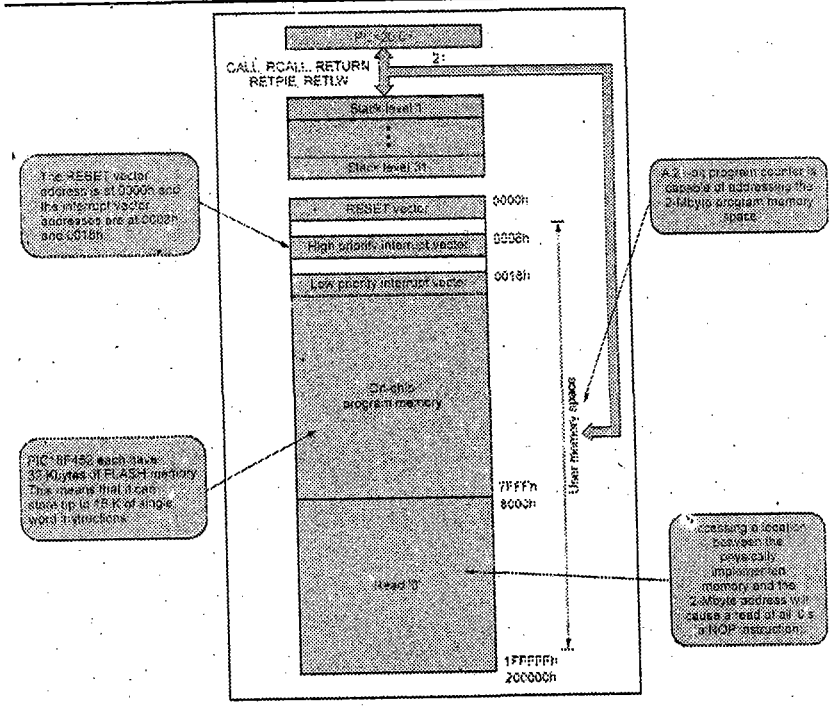


Fig. Q.7.1 : Program memory organization of PIC

STACK : The stack operates as a 31-word by 21-bit RAM and a 5-bit Stack Pointer, STKPTR. The stack space is not part of either program or data space. The stack organization is shown in Fig. Q.7.2.

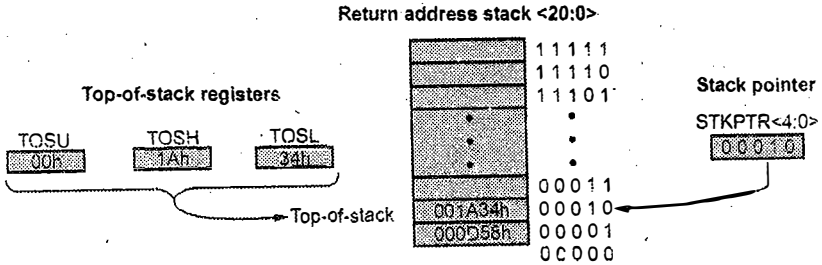


Fig. Q.7.2 : Stack indication

Q.8 Draw and explain the programming model of PIC 18FXXXX.

[SPPU : May-22, Marks 6]

Ans. : <u>Programming Model of PIC18FXX</u> : It supports for the smooth functioning of the microcontroller, all components are the parts of programming model contribute directly or indirectly. The components of general programming model are ALU, PSW, Pointers, RAM, Data and program ROM, Timers, SFRS and ports as shown in Fig. Q.8.1. It also represent the internal architecture of a microprocessor necessary to write assembly language programs. It has been divided into two groups as

- Arithmetic Logic Unit (ALU) and Registers from Microprocessor Unit (MPU).
- Special Function Registers (SFRs) : From Data (File) Memory.

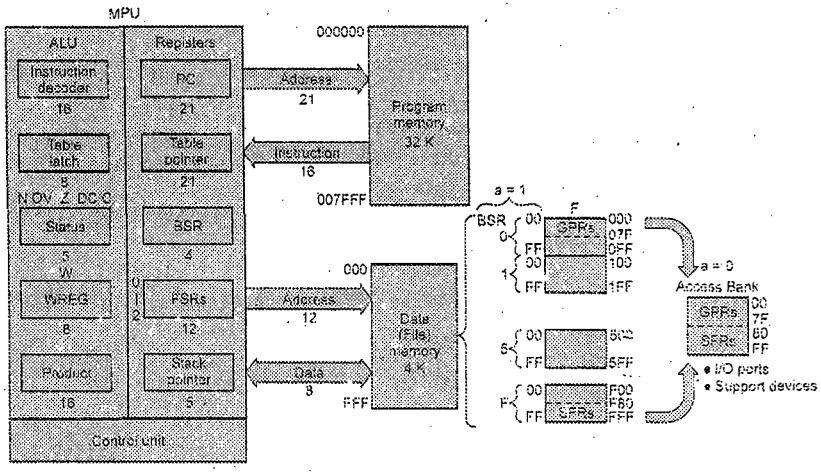


Fig. Q.8.1 : Programming model of PIC

Registers and Pointers

- **Working Register (W) :** 8 bit accumulator.
- **Program Counter (PC) :** 21-bit register functions as a pointer to program memory during program execution.

- **Table Pointer :** 21-bit register used as a memory pointer to copy bytes between program memory and data registers.
- **Stack Pointer (SP) :** 5-bit register used to point to the stack.
- **Stack :** 31 registers used for temporary storage of memory addresses during execution of a program.
- **BSR :** Bank Select Register (0 to F) : 4-bit register
  Provides upper 4-bits of 12-bit address of data memory.
- **FSR :** File Select Registers : FSR0, FSR1 and FSR2
  FSR : Composed of two 8-bit registers : FSRH and FSRL
  Used as pointers for data registers
  Holds 12-bit address of data register

Special Function Registers

These registers are used to control the operation of Timers/Counters, interrupts, serial interface and ports. The location of each SFR is fixed and accessed by use of direct addressing mode. Data registers associated with I/O ports, support devices and processes of data transfer.

- I/O Ports (A to E)
- EEPROM
- Timers
- Analog-to-Digital (A/D) converter
- Interrupts
- Serial I/O
- Capture/Compare/PWM (CCP)

Program and Data Memory

- Data memory up to 4 kbytes : Data register map - with 12-bit address bus 000-FFF. The total memory is divided into 16 banks each of 256-byte (FFF = 2^{12} = 16 × 256 = 4096 = 4K)
- PIC18 microcontrollers implement a 21-bit program counter, which is capable of addressing a 2-Mbyte program memory space. Accessing a location between the upper boundary of the physically implemented memory and the 2-Mbyte address will return all '0's (a NOP instruction).

Q.9 Enlist the steps in selection of PIC Microcontroller.

Ans. : Selection Criteria :

The selection of each microcontroller used for specific application depends on following factors.

- Data handling capacity - Bits, Nibble bytes, words, double words, quad words etc.
- Speed - depends on clock.
- Amount of RAM/ ROM/ EPROM/ flash/ static.
- Number of I/O pins, timers - All SFRS.
- Power consumption - Based on the modes.
- Packaging - 40 PIN DIP,/ QFP/ other - important - Space, assembly, prototyping the end product
- Added features like ADC/ DAC/ CCP, bus support like CAN, SPI, I2C, USB.
- Watchdog timer, Timer modes, data EEPROM etc.
- Easy to upgrade - Higher performances or low power operations.

Q.10 Draw the pin out diagram of PIC 18F4550 and explain function of each.

Ans. : Pin Functions of PIC 18F4550

- It uses the dual in-line package with forty pins as shown in Fig. Q.10.1. The total forty pins and divided into different parts as port pins, reset, memory control and power supply. Out of 40 pins, 33 pins are dedicated to I/O functions with five ports with alternate functions. Rest of the pins are VDD, GND, OSC1, OSC2, and MCLR.

Pin No.	Functions	Explanation
01	MCLR/VPP/RE3	It is input and active low, often referred as POR / Port E.
02-07	RA0-5/AN0-AN4/Cvref	Port A / ADC input channels.
08-10	RE0-3/AN5-7	Port E/,ADC input with other functions.
11, 32	VDD	Supply voltage, +5 V, low in order to reduce the noise and power dissipation, values can be set using configuration register.
12, 31	VSS (GND)	More pins for VDD and VSS help to reduce the noise.
13	OSC1/CLK1	Oscillator connect pin.
14	OSC2/CLK0/RA6	Oscillator connect/ Port A, Often the quartz crystal is connected between these pins, values of crystal are decided according to its configuration registers PIC have speed from 0 to 40 MHz.
15-18	RC0-2, 17-CCP1,18-USB	Port C and CCP mode configuration with USB.
19-22	RD0-RD3/ SPP 0-3	Port D and parallel slave ports.
23-26	RC4-7, D-/D*/TX/RX/SDO	Port C and SPI data bus and UART communication pins.
27-30	RD4-7/ SPP4-7	Port D and parallel slave ports.

33-40	RB0-7,/AN8-12/ INT0-2, CCP2	Port D , ADC input and external interrupt and CCP2 pins SCK and SDA used in I2C communications

- **Note :** For more details of pin assignment please refer Fig. Q.10.1. Also according to device number pins are assigned with increased multiple functions.

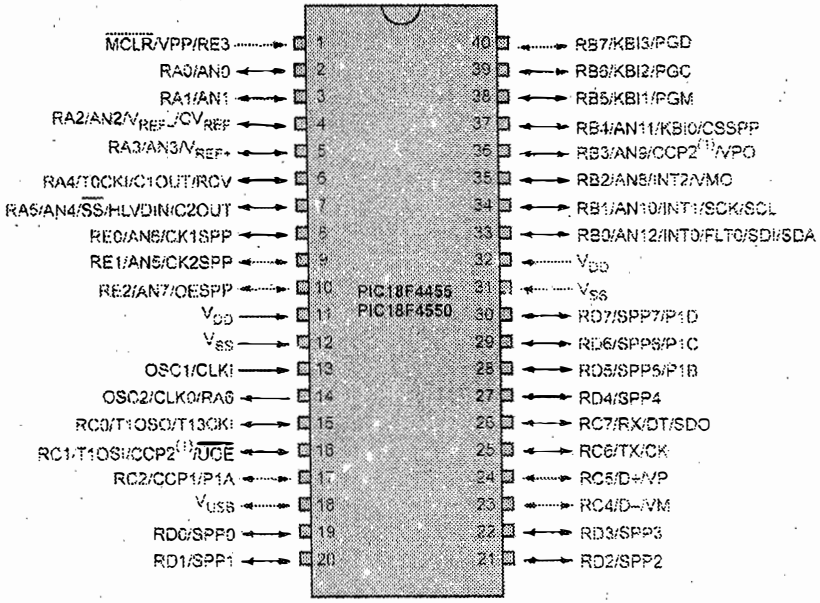


Fig. Q.10.1 : Pin out diagram

PORTA[7]	PORTB[8]	PORTC[8]	PORTD[8]	PORTE[4]
ADC CH [0-4]	Universal [ADC 8-12]	Timer clock	Parallel operation	ADC CH [5-7]
Ref voltage	Interrupt, I^2C , CCP2	CCP1, serial,I^2C		MCLR, CK1 CK2,OE

Q.11 Explain with block schematic function of RESET in PIC 18FXXXX.

[SPPU: May-22, Marks 8, May-18, Marks 8,Dec-17, Marks 8]

Ans. : Reset Operation : The main function of reset is to initialize the operation. After reset the processor is initialized for the initial value (by default, Like all ports are configured as input).

- A reset puts the PIC in a well-defined initial state so that the processor starts executing code from the first instruction.
- Reset will cause all current data to be lost.
- Reset can results from :
  - Power-on Reset (POR)
  - MCLR reset during normal operation(External reset by MCLR pulled down)
  - MCLR reset during power-managed modes
  - Watchdog Timer (WDT) reset (during execution Watchdog timer overflow)
  - Programmable Brown-out Reset (BOR)
  - RESET instruction
  - Stack full reset
  - Stack underflow reset
  - Reset on power supply brown-out
  - The detailed combination of reset operation along with functional diagram is shown in Fig. Q.11.1.

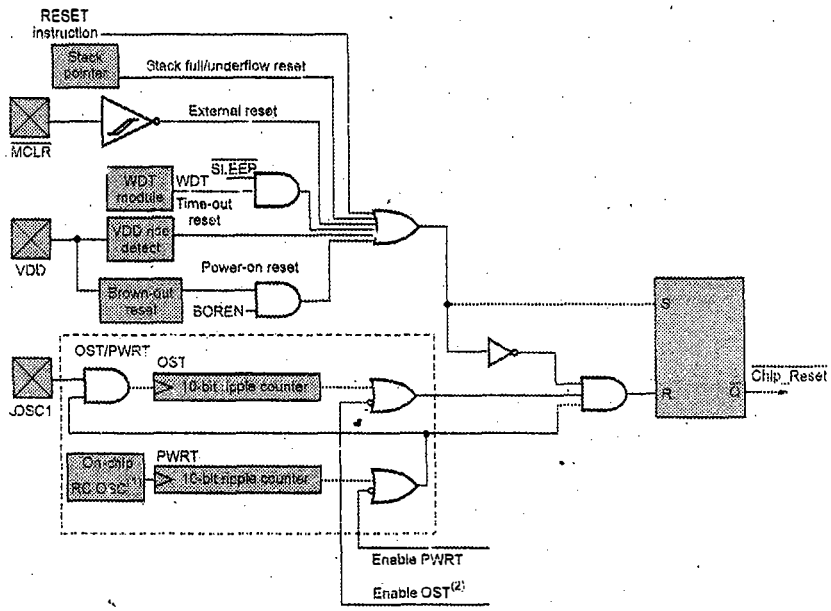


Fig. Q.11.1 : Reset operational diagram

- Reset can be of,
  1. Power-on Reset (POR)
  2. Power-up Timer (PWRT)
  3. Oscillator Start-up Timer (OST)

RCON : Reset Control Register

Device reset events are tracked through the RCON register as shown in Fig. Q.11.2. The lower five bits of the register indicate that a specific reset event has occurred. In most cases, these bits can only be cleared by the event and must be set by the application after the event. The state of these flag bits, taken together, can be read to indicate the type of Reset that just occurred. The RCON register also has control bits for setting interrupt priority (IPEN) and software control of the BOR (SBOREN).

R/W-0	R/W-1[1]	U-0	R/W-1	R-1	R-1	R/W-0[2]	R/W-0
IPEN	SBOREN	–	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

Fig. Q.11.2 : Reset control register

- The $\overline{MCLR}$ pin provides a method for triggering an external reset of the device.
- A reset is generated by holding the pin low.
- These devices have a noise filter in the reset path which detects and ignores small pulses.
- The pin is not driven low by any internal resets, including the WDT.
- In PIC18F2455/2550/4455/4550 devices, the $\overline{MCLR}$ input can be disabled with the MCLRE configuration bit. When $\overline{MCLR}$ is disabled, the pin becomes a digital input.

Q.12 Explain POR, PWRT, BOD modes of RESET in PIC 18FXXXX.

Ans. : <u>Reset Modes</u> : The reset operation is affected by the following modes depending on the way of using parameters.

A. Power-on Reset (POR)

- A Power-on Reset pulse is generated on-chip when VDD rise is detected above threshold. The power on circuit is shown in Fig. Q.12.1.
- To take advantage of the POR circuitry, just tie the MCLR pin directly (or through a resistor of 1 kΩ to 10 kΩ) to VDD. This will eliminate external RC components usually needed to create a power-on reset delay.

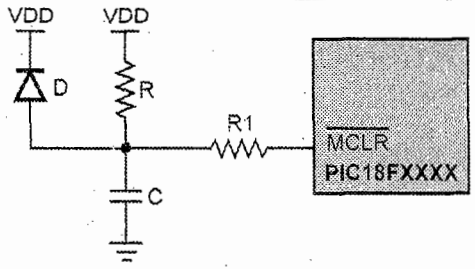


Fig. Q.12.1 : Power on reset

- When the device starts normal operation (i.e., exits the RESET condition), device operating parameters (like voltage, frequency, temperature, etc.) must be met to ensure operation. If these conditions are not met, the device must be held in RESET until the operating conditions are met.
- POR events are captured by the POR bit (RCON<1>). The state of the bit is set to '0' whenever a POR occurs; it does not change for any other reset event. POR is not reset to '1' by any hardware event. To capture multiple events, the user manually resets the bit to '1' in software following any POR.
- External power-on reset circuit is required only if the VDD power-up slope is too slow.
- The diode D helps discharge the capacitor quickly when VDD powers down. R < 40 kΩ is recommended to make sure that the voltage drop across R does not violate the device's electrical specification.
- R1 ≥ 1 kΩ will limit any current flowing into MCLR from external capacitor C, in the event of MCLR/VPP pin breakdown, due to Electro - Static Discharge (ESD) or Electrical Overstress (EOS).

B. Power-up Timer (PWRT)

- The power-up timer provides a fixed nominal time-out only on power-up from the POR. The power-up timer operates on an internal RC oscillator.
- The chip is kept in RESET as long as the PWRT is active.
- The PWRT's time delay allows VDD to rise to an acceptable level.

- A configuration bit is provided to enable/disable the PWRTEN.
- The power-up time delay will vary from chip-to-chip due to VDD, temperature and process variation.
- The Oscillator Start-up Timer (OST) provides a 1024 oscillator cycle (from OSC1 input) delay after the PWRT delay is over.
- This ensures that the crystal oscillator or resonator has started and stabilized.
- The Oscillator Start-up Timer (OST) provides a 1024 oscillator cycle (from OSC1 input) delay after the PWRT delay is over.
- This ensures that the crystal oscillator or resonator has started and stabilized.
- The Power-up Timer (PWRT) of the PIC18F2455/2550/4455/4550 devices is an 11-bit counter which uses the INTRC source as the clock input. This yields an approximate time interval of 2048 × 32 μs = 65.6 ms. While the PWRT is counting, the device is held in reset.

C. Brown Out Detect (RESET)

- Mostly all microcontrollers have built in Brown-out Detection (BOD) circuit, which monitors supply voltage level during operation. BOD circuit in Fig. Q.12.2 is nothing more than comparator, which compares supply voltage to a fixed trigger level.
- The BOR is controlled by the BORV1:BORV0 and BOREN1:BOREN0 configuration bits in CONFIG 2 L register.: Ref (Microchip manual 18F4550).

U-0	U-0	R/P-0	R/P-1	R/P-1	R/P-1	R/P-1	R/P-1
—	—	VREGEN	BORV1(1)	BORV0(1)	BOREN1(2)	BOREN1(2)	$\overline{\text{PWRTEN}}$ (3)
bit 7							bit 0

- The BOR threshold is set by the BORV1:BORV0 bits. If BOR is enabled, any drop of VDD below VBOR for greater than threshold

will reset the device. A reset may or may not occur if VDD falls below VBOR for less than threshold. The chip will remain in Brown-out Reset until VDD rises above VBOR.

- o Overcome the fluctuations in VDD brown on reset voltage provided.

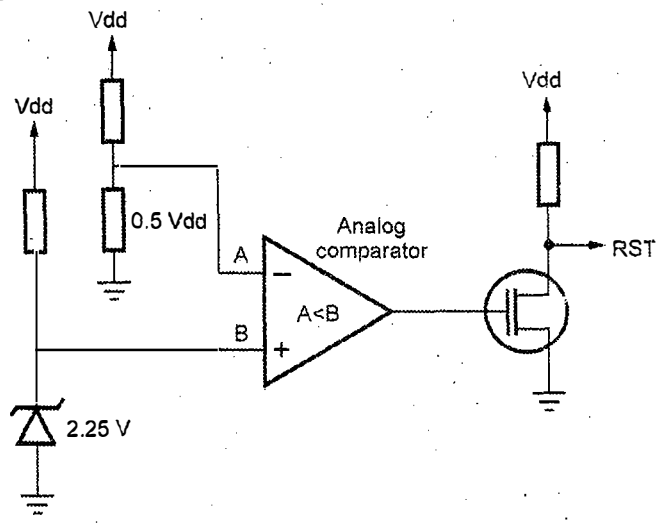


Fig. Q.12.2 : Brown on detect

- o CONFIG 2L allows to set minimum VDD required, if falls below CPU goes into reset state. It is set according to oscillator frequency connected to OSC1 and OSC2 pins.
- o For 40 MHz – VBOR = 4.5 V

  2 MHz – VBOR = 2.0 V

- If the power-up timer is enabled, it will be invoked after VDD rises above VBOR; it then will keep the chip in reset for an additional time delay, PWRT.
- If VDD drops below VBOR while the power-up timer is running, the chip will go back into a brown-out reset and the power-up timer will be initialized. Once VDD rises above VBOR, the power-up timer will execute the additional time delay.

- SOFTWARE ENABLED BOR : When BOREN1:BOREN0 = 01, the BOR can be enabled or disabled by the user in software. This is done with the control bit, SBOREN (RCON<6>). Setting SBOREN enables the BOR to function as previously described. Clearing SBOREN disables the BOR entirely. The SBOREN bit operates only in this mode; otherwise, it is read as '0'.
- Hardware BOR : When BOREN1: BOREN0 = 10, the BOR remains under hardware control and operates as previously described. Whenever the device enters sleep mode, however, the BOR is automatically disabled.
- When the device returns to any other operating mode, BOR is automatically re-enabled.

Note : BOR and the Power-on Timer (PWRT) are independently configured. Enabling BOR reset does not automatically enable the PWRT.

Q.13 Explain watch dog timer mode of RESET in PIC 18FXXXX.

Ans. : Watchdog Timer

- For PIC18F4550 devices, the WDT is driven by the INTRC source.
- The nominal WDT period is 4 ms and has the same stability as the INTRC oscillator. The period of the WDT is multiplied by a 16-bit postscaler.
- Any output of the WDT postscaler is selected by a multiplexer, controlled by bits in configuration register 2H shown below

U - 0	U - 0	U - 0	U - 0	R/P - 1	R/P - 1	R/P - 1	R/P1
				WDTPS2	WDTPS1	WDTPS0	WDTEN

- Available periods range from 4 ms to 131.072 seconds (2.18 minutes).

- The WDT and post scaler are cleared when any of the following events occur : a SLEEP or CLRWDT instruction is executed, the IRCF bits (OSCCON<6:4>) are changed or a clock failure has occurred according to oscillator control bits.
- The general block diagram of Watchdog timer is shown in Fig. Q.13.1.

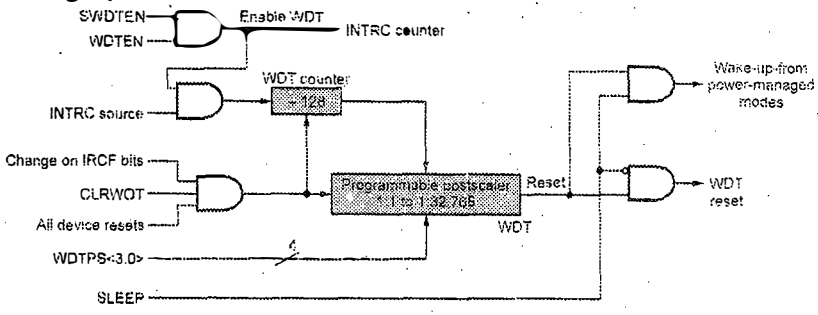


Fig. Q.13.1 Watchdog timer

Also the SFRS used for enabling the Watchdog timer are,

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
RCON	IPEN	SBOREN(1)	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
WDTCON	—	—	—	—	—	—	—	SWDTEN

Bit 0 SWDTEN : Software controlled Watchdog timer Enable bit : This bit has no effect if the configuration bit, WDTEN, is enabled in CONFIG 2H register

1 = Watchdog timer is on

0 = Watchdog timer is off

1. The CLRWDT and SLEEP instructions clear the WDT and postscaler counts when executed.
2. Changing the setting of the IRCF bits (OSCCON<6:4>) clears the WDT and postscaler counts.
3. When a CLRWDT instruction is executed, the postscaler count will be cleared.

Q.14 Explain different oscillator options with bit settings.

☞ [SPPU: May-19, Marks 8]

Ans. : Oscillator options : Essentially, there are three clock sources for the PIC microcontroller to operate in different modes of operation :

Primary oscillators : The primary oscillators include the external crystal and resonator modes, the external clock modes and the internal oscillator block. The particular mode is defined by the FOSC3:FOSC0 configuration bits.

Secondary oscillators : The secondary oscillators are those external sources not connected to the OSC1 or OSC2 pins. These sources may continue to operate even after the controller is placed in a power-managed mode.

Internal oscillator block : In addition to being a primary clock source, the internal oscillator block is available as a power-managed mode clock source. The INTRC source is also used as the clock source for several special features, such as the WDT and Fail-Safe Clock Monitor.

In short for the PIC

- Four crystal modes, using crystals or ceramic resonators.
- Two external clock modes, offering the option of using two pins (oscillator input and a divide-by-4 clock output) or one pin (oscillator input, with the second pin reassigned as general I/O).
- Two external RC oscillator modes with the same pin options as the external clock modes.
- The internal oscillator circuit is used to generate the device clock. The device clock is required for the device to execute instructions and for the peripherals to function. Four device clock periods generate one internal instruction clock (TCY) cycle.
- An internal oscillator block which provides an 8 MHz clock and an INTRC source (approximately 31 kHz), as well as a range of 6 user

selectable clock frequencies, between 125 kHz to 4 MHz, for a total of 8 clock frequencies. This option gives the two oscillator pins for use as additional general purpose I/O.

- A Phase Lock Loop (PLL) frequency multiplier, available to both the high-speed crystal and internal oscillator modes, which allows clock speeds of up to 40 MHz used with the internal oscillator, the PLL gives users a complete selection of clock speeds, from 31 kHz to 32 MHz - all without using an external crystal or clock circuit.

Configuration Registers for Oscillator Options

- Many features of the PIC can be selected using the bits in configuration registers reducing the cost for external components.
- These configuration registers are located at address 300000 H which is outside the 000000-1FFFFF H (4 MB) of range ROM.
- The configuration register can be assessed from the user program using table read and writes.
- Writing 8 bit values with CONFIG directive will be loaded in register.
- Incorrect programming of configuration register can cause system to fail.
- Configuration registers are of reset, clock source and VDD source. Table Q.14.1 gives the addressed location with details.

Address (HEX)	NAME	General Description
300001	CONFIG1H	Oscillator selection
300002	CONFIG2L	Brown out
300003	CONFIG2H	Watch dog enable
300006	CONFIG4L	Background debugger and ISCP

Table Q.14.1 : Address location of CONFIG registers

Configuration Bit Settings

- The configuration bits can be programmed (read as '0'), or left un-programmed (read as '1'), to select various device configurations. These bits are mapped starting at program memory location 300000H.
- The user will note that address 300000H is beyond the user program memory space. In fact, it belongs to the configuration memory space (300000H - 3FFFFFH), which can only be accessed using Table Reads and Table Writes.
- Programming the configuration registers is done in a manner similar to programming the FLASH memory. The only difference is the configuration registers are written a byte at a time. The configuration bit settings are shown in Table Q.14.2.

File Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Default / Un-programmed Value
300001H	CONFIG1H	IESO	FCMEN	—	—	FOSC3	FOSC2	FOSC1	FOSC0	00-- 0111
300002H	CONFIG2L	—	—	—	BORV1	BORV0	BOREN1	BOREN0	PWRTEN	---1 1111
300003H	CONFIG2H	—	—	—	WDTPS3	WDTPS2	WDTPS1	WDTPS0	WDTEN	---1 1111
300005H	CONFIG3H	MCLRE	—	—	—	—	LPT 1 QSC	PBADEN	CCP2MX	1--- 011
300006H	CONFIG4L	DEBUG	XINST	—	—	—	LVP	—	STVREN	10-- -1-1
300008H	CONFIG5L	—	—	—	—	CP3(1)	CP2(1)	CP1	CP0	---- 1111
300009H	CONFIG5H	CPD	CPB	—	—	—	—	—	—	11-- ----
30000AH	CONFIG6L	—	—	—	—	WRT3(1)	WRT2(1)	WRT1	WRT0	---- 1111
30000BH	CONFIG6H	WRTD	WRTB	WRTC	—	—	—		—	111- ----
30000CH	CONFIG7L	—	—	—	—	EBTR3(1)	EBTR2(1)	EBTR1	EBTR0	---- 1111
30000DH	CONFIG7H	—	EBTRB	—	—	—	—	—	—	-1-- ----
3FFFFEH	DEVID1	DEV2	DEV1	DEV0	REV4	REV3	REV2	REV1	REV0	xxxx xxxx(2)
3FFFFFH	DEVID2	DEV10	DEV9	DEV8	DEV7	DEV6	DEV5	DEV4	DEV3	xxxx xxxx(2)

Table Q.14.2 : Configuration bit setting

Q.15 Explain in detail oscillator control register OSCCON.

Ans. : OSCCON : Oscillator Control register

- The OSCCON register controls several aspects of the device clock's operation, both in full-power operation and in power-managed modes. The system clock select bits, SCS1:SCS0, select the clock source. The oscillator control register is shown in Fig. Q.15.1.
- The available clock sources are the primary clock (defined by the FOSC3:FOSC0 configuration bits in CONFIG 1H register), the secondary clock (Timer1 oscillator) and the internal oscillator block.
- The internal oscillator frequency select bits, IRCF2 : IRCF0, select the frequency output of the internal oscillator block to drive the device clock.
- When an output frequency of 31 kHz is selected (IRCF2:IRCF0 = 000), users may choose which internal oscillator acts as the source.

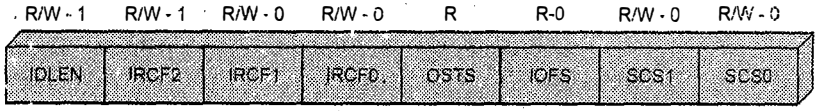


Fig. Q.15.1 : Oscillator control register

Bit 7 : IDLEN : Idle Enable bit

1 = Device enters Idle mode on SLEEP instruction

0 = Device enters Sleep mode on SLEEP instruction

Bit 6-4 : IRCF2:IRCF0: Internal oscillator frequency select bits

111 = 8 MHz (INTOSC drives clock directly)

110 = 4 MHz

101 = 2 MHz

100 = 1 MHz(3)

011 = 500 kHz

010 = 50 kHz

001 = 125 kHz

000 = 31 kHz (from either INTOSC/256 or INTRC directly)

Bit 3 : OSTS : Oscillator Start-up Time-out Status bit

1 = Oscillator start-up timer time-out has expired; primary oscillator is running

0 = Oscillator start-up timer time-out is running; primary oscillator is not ready

Bit 2 : IOFS : INTOSC Frequency Stable bit

1 = INTOSC frequency is stable

0 = INTOSC frequency is not stable

Bit 1-0 : SCS1 : SCS0 : System Clock Select bits

Q.16 Explain various power down (Managed) modes of PIC18FXXX.

[SPPU : May-22, Marks 6, May-18,19, Marks 8]

Ans. : **Power Managed (Down) Modes :**

- PIC18F2455/2550/4455/4550 devices offers total of seven operating modes for more efficient power management.
- There are three categories of power-managed modes :
  - o Run modes
  - o Idle modes
  - o Sleep mode.
- These categories define which portions of the device are clocked and sometimes, what speed.

- The Run and Idle modes may use any of the three available clock sources (primary, secondary or internal oscillator block); the Sleep mode does not use a clock source.
- Selecting a power-managed mode requires two decisions: If the CPU is to be clocked or not and the selection of a clock source.
- The IDLEN bit (OSCCON<7>) controls CPU clocking, while the SCS1 : SCS0 bits (OSCCON<1:0>) select the clock source.

OSCCON : Oscillator Control Register

R/W - 1	R/W - 1	R/W - 0	R/W - 0	R	R-0	R/W - 0	R/W - 0
IDLEN	IRCF2	IRCF1	IRCF0	OSTS	IOFS	SCS1	SCS0

Selection of power down modes

1. **Selection of Clock Sources :** Primary Clock OSC2 : OSC0, The secondary clock (the Timer 1 oscillator) and the internal oscillator block (for RC modes).
2. **Entering into power down modes :**
   a. Switching from one power-managed mode to another begins by loading the OSCCON register. The SCS1: SCS0 bits select the clock source and determine which Run or Idle mode is to be used. Changing these bits causes an immediate switch to the new clock source, assuming that it is running.
   b. Entry to the power-managed Idle or Sleep modes is triggered by the execution of a SLEEP instruction. The actual mode that results depends on the status of the IDLEN bit.
   c. Depending on the current mode and the mode being switched to, a change to a power-managed mode does not always require setting all of these bits.
   d. Executing a SLEEP instruction does not necessarily place the device into Sleep mode. It acts as the trigger to place the controller into either the Sleep mode, or one of the idle modes,

depending on the setting of the IDLEN bit.

e. Available power managed modes is shown in Table Q.16.1.

Mode	OSCCON<7, 1:0> IDLEN(1)	SCS1:SCS0	Module Clocking CPU	Peripherals	Available Clock and Oscillator Source
Sleep	0	N.A.	Off	Off	None - all clocks are disabled
PRI_RUN	N.A.	00	Clocked	Clocked	Primary - all oscillator modes. This is the normal full-power execution mode.
SEC_RUN	N.A.	01	Clocked	Clocked	Secondary - Timer 1 oscillator
RC_RUN	N.A.	1x	Clocked	Clocked	Internal oscillator BLOCK(2)
PRI_IDLE	1	00	Off	Clocked	Primary - all oscillator modes
SEC_IDLE	1	01	Off	Clocked	Secondary - Timer 1 oscillator
RC_IDLE	1	1x	Off	Clocked	Internal oscillator BLOCK(2)

Table Q.16.1 : Power managed modes

Q.17 Explain RUN power down (Managed) mode of PIC18FXXX.

☞ **[Marks 6]**

Ans. : Run modes : In the run modes, clocks to both the core and peripherals are active. The difference between these modes is the clock source.

A. PRI_RUN MODE : This is also the default mode upon a device Reset. It is the normal, full-power execution mode of the microcontroller. Depending on the primary clock source the IOFS bit may be set in oscillator control register.

B. SEC_RUN MODE :

- The SEC_RUN mode is the compatible mode to the "clock switching" feature offered in other PIC18 devices.
- In this mode, the CPU and peripherals are clocked from the Timer 1 oscillator.
- This gives users the option of lower power consumption while still using a high-accuracy clock source.

- SEC_RUN mode is entered by setting the SCS1:SCS0 bits to '01'. The device clock source is switched to the Timer 1 oscillator and the primary oscillator is shut down.
- On transitions from SEC_RUN mode to PRI_RUN, the peripherals and CPU continue to be clocked from the Timer 1 oscillator while the primary clock is started.

C. RC_RUN MODE :

- In RC_RUN mode, the CPU and peripherals are clocked from the internal oscillator block using the INTOSC multiplexer; the primary clock is shut down.
- When using the INTRC source, this mode provides the best power conservation of all the Run modes while still executing code.
- It works well for user applications which are not highly timing sensitive or do not require high-speed clocks at all times.
- If the primary clock source is the internal oscillator block (either INTRC or INTOSC), there are no distinguishable differences between the PRI_RUN and RC_RUN modes during execution. However, a clock switch delay will occur during entry to and exit from RC_RUN mode. Therefore, if the primary clock source is the internal oscillator block, the use of RC_RUN mode is not recommended.
- On transitions from RC_RUN mode to PRI_RUN mode, the device continues to be clocked from the INTOSC multiplexer while the primary clock is started.
- The IDLEN and SCS bits are not affected by the switch. The INTRC source will continue to run if either the WDT or the Fail-Safe Clock Monitor is enabled.

Q.18 Explain SLEEP power down (Managed) mode of PIC18FXXX.

☞ **[Marks 6]**

Ans. : Sleep Modes

- The power-managed Sleep mode in the PIC18F2455/2550/4455/4550 devices is identical to the legacy Sleep mode offered in all other PIC devices.
- It is entered by clearing the IDLEN bit (the default state on device Reset) and executing the SLEEP instruction.
- This shuts down the selected oscillator. All clock source status bits are cleared.
- Entering the Sleep mode from any other mode does not require a clock switch.

This is because no clocks are needed once the controller has entered Sleep. If the WDT is selected, the INTRC source will continue to operate. If the Timer 1 oscillator is enabled, it will also continue to run.

- When a wake event occurs in Sleep mode (by interrupt, Reset or WDT time-out), the device will not be clocked until the clock source selected by the SCS1 : SCS0 bits becomes ready or it will be clocked from the internal oscillator block if either the Two-Speed Start-up or the Fail-Safe Clock Monitor are enabled.
- In either case, the OSTS bit is set when the primary clock is providing the device clocks. The IDLEN and SCS bits are not affected by the wake-up.

Q.19 Explain IDLE power down (Managed) mode of PIC18FXXX.

☞ [Marks 6]

Ans. : Idle Modes

- The Idle modes allow the controller's CPU to be selectively shut down while the peripherals continue to operate. Selecting a particular idle mode allows users to further manage power consumption.

- If the IDLEN bit is set to "1" when a SLEEP instruction is executed, the peripherals will be clocked from the clock source selected using the SCS1: SCS0 bits; however, the CPU will not be clocked. The clock source status bits are not affected.
- Setting IDLEN and executing a SLEEP instruction provides a quick method of switching from a given Run mode to its corresponding idle mode.
- If the WDT is selected, the INTRC source will continue to operate. If the Timer 1 oscillator is enabled, it will also continue to run.
- Since the CPU is not executing instructions, the only exits from any of the idle modes are by interrupt, WDT time-out or a Reset.
- When a wake event occurs, CPU execution is delayed by an interval of TCSD while it becomes ready to execute code.
- When the CPU begins executing code, it resumes with the same clock source for the current idle mode.
- The IDLEN and SCS bits are not affected by the wake-up.
- While in any Idle mode or Sleep mode, a WDT time-out will result in a WDT wake-up to the Run mode currently specified by the SCS1: SCS0 bits.

A. PRI_IDLE MODE :

- It is unique among the three low-power Idle modes which does not disable the primary device clock during timing sensitive applications, this allows for the fastest resumption of device operation, with its more accurate primary clock source.
- PRI_IDLE mode is entered from PRI_RUN mode by setting the IDLEN bit and executing a SLEEP instruction.
- If the device is in another Run mode, set IDLEN first, then clear the SCS bits and execute SLEEP.

- Although the CPU is disabled, the peripherals continue to be clocked from the primary clock source specified by the FOSC3:FOSC0 configuration bits. The OSTS bit remains set.
- When a wake event occurs, the CPU is clocked from the primary clock source. A delay of interval TCSD is required between the wake event and when code execution starts. It is required to allow the CPU to become ready to execute instructions.
- After the wake-up, the OSTS bit remains set. The IDLEN and SCS bits are not affected by the wake-up.

B. SEC_IDLE MODE :

- In this mode, the CPU is disabled but the peripherals continue to be clocked from the Timer 1 oscillator. This mode is entered from SEC_RUN by setting the IDLEN bit and executing a SLEEP instruction.
- If the device is in another Run mode, set IDLEN first, then set SCS1:SCS0 to '01' and execute SLEEP. When the clock source is switched to the Timer 1 oscillator, the primary oscillator is shut down, the OSTS bit is cleared and the T1RUN bit is set.
- When a wake event occurs, the peripherals continue to be clocked from the Timer 1 oscillator. After an interval of TCSD following the wake event, the CPU begins executing code being clocked by the Timer 1 oscillator.
- The IDLEN and SCS bits are not affected by the wake-up; the Timer 1 oscillator continues to run.

C. RC_IDLE MODE :

- In RC_IDLE mode, the CPU is disabled but the peripherals continue to be clocked from the internal oscillator block using the INTOSC multiplexer. This mode allows for controllable power conservation during Idle periods.

- From RC_RUN, this mode is entered by setting the IDLEN bit and executing a SLEEP instruction. If the device is in another Run mode, first set IDLEN, then set the SCS1 bit and execute SLEEP. Although its value is ignored, it is recommended that SCS0 also be cleared; this is to maintain software compatibility with future devices.
- The INTOSC multiplexer may be used to select a higher clock frequency by modifying the IRCF bits before executing the SLEEP instruction. When the clock source is switched to the INTOSC multiplexer, the primary oscillator is shut down and the OSTS bit is cleared.
- If the IRCF bits are set to any non-zero value, or the INTSRC bit is set, the INTOSC output is enabled.
- If the IRCF bits were previously at a non-zero value, or INTSRC was set before the SLEEP instruction was executed and the INTOSC source was already stable, the IOFS bit will remain set.
- If the IRCF bits and INTSRC are all clear, the INTOSC output will not be enabled, the IOFS bit will remain clear and there will be no indication of the current clock source.
- When a wake event occurs, the peripherals continue to be clocked from the INTOSC multiplexer. After a delay of TCSD following the wake event, the CPU begins executing code being clocked by the INTOSC multiplexer.
- The IDLEN and SCS bits are not affected by the wake-up. The INTRC source will continue to run if either the WDT or the Fail-Safe Clock Monitor is enabled.

Q.20 Explain with diagram peripheral support in PIC18FXXX.

Ans: Brief Summary of Peripheral Support

The PIC 18F452 has the following peripherals : As shown in Fig. Q.20.1.

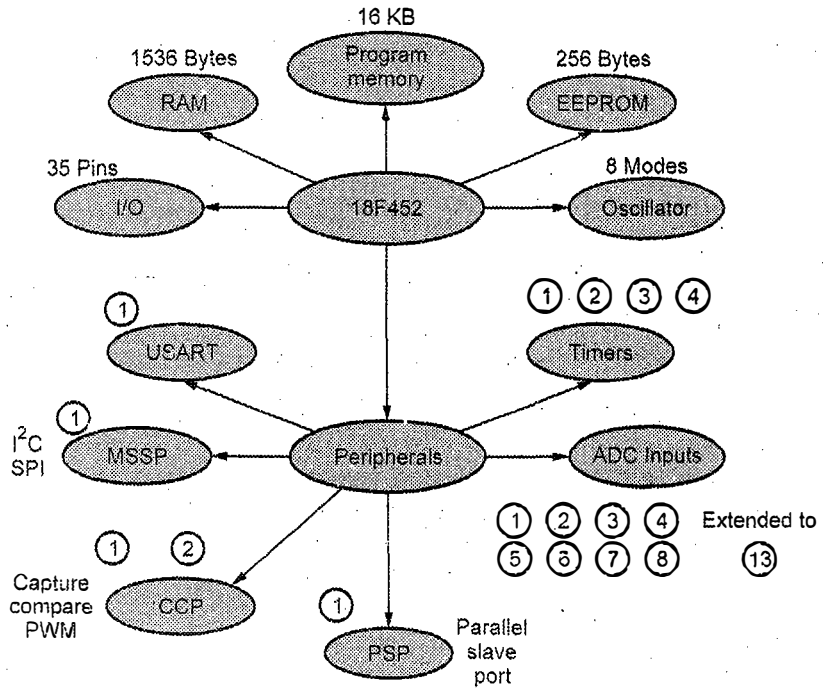


Fig. Q.20.1: Peripheral support in PIC

- Data ports :
  - A (7-bits)
  - B, C and D (8-bits)
  - E (4- bits)
- Counter/Timer modules :
  - Modules 0,2 (8-bits)
  - Modules 1,3 (16-bits)
- CCP modules :
  - I2C/SPI serial port.
  - USART port.

- ADC 10-bits with 13 Channel.
- EEPROM 256 bytes

a. Five I/O ports

- PORT A through PORT E
- Most I/O pins are multiplexed
- Generally have eight I/O pins with a few exceptions
- Addresses already assigned to these ports in the design stage
- Each port is identified by its assigned Special Function Registers (SFR)

  PORTA (address of F80)

  PORTB (address of F81)

  → these are part of data memory or register file.

b. Capture-Compare-Pulse Width Modulation (CCP)

- The compare mode can cause an event like simply turning on the device when the contents of timer matches with CCP register.
- In capture mode, an event at CCP pin will cause contents of timer to be loaded in CCP register.
- Pulse width modulation feature allows to create pulses of variable duty cycle.
- The main difference between enhanced CCP module and standard CCP is that it allows four pins for implementation of H bridge or half H bridge for DC motor control. 1, 2 or 4 PWM outputs

c. Timer module

- The timer 0 module timer/counter which can work as timer / counter has the following features :

- 8-bit or 16 bit timer/counter
- 8-bit software programmable pre-scaler
- Internal or external clock
- Select interrupt on overflow from FFH to 00H
- Edge select for external clock
- Timer 1 is 16 bit timer/counter and cannot be operated in 8 bit.
- Timer 2 is an 8-bit timer with a pre-scaler. It can be used as the PWM time-base for the PWM mode of the CCP module(s).
- Timer 3 is 16 bit timer/ counter and cannot be operated in 8 bit. It also works in CCP mode.

d. Master Synchronous Serial Port (MSSP) module

- The Master Synchronous Serial Port (MSSP) module is a serial interface useful for communicating with other peripheral or microcontroller devices. These peripheral devices may be serial EEPROMs, shift registers, display drivers, A/D converters, etc. The MSSP module can operate in one of two modes :
  1. Serial Peripheral Interface (SPI)
  2. Inter-Integrated Circuit (I2C).

e. Enhanced universal synchronous asynchronous receiver transmitter

- The EUSART can be configured in the following modes :
  1. Asynchronous (full duplex) with :
     - Auto-wake-up on character reception
     - Auto-baud calibration
     - 12-bit break character transmission.
  2. Synchronous - Master (half duplex) with selectable clock polarity

  3. Synchronous - Slave (half duplex) with selectable clock polarity

f. Parallel slave port

- In addition to its function as a general I/O port, PORTD can also operate as an 8-bit wide Parallel Slave Port (PSP) or microprocessor port.
- PSP operation is controlled by the four upper bits of the TRISE register.
- Setting control bit, PSPMODE (TRISE<4>), enables PSP operation as long as the enhanced CCP module is not operating in dual output or quad output PWM mode. In slave mode, the port is asynchronously readable and writable by the external world.
- The PSP can directly interface to an 8-bit microprocessor data bus. The external microprocessor can read or write the PORTD latch as an 8-bit latch.

Important Points to Remember

1. PIC uses Harvard architecture with RISC instruction set architecture
2. Clock is DC- 40 MHz
3. Has in build Timer/Counter, Serial port ADC,
4. Instructions are of 2, 4 bytes long
5. Families are not upward compatible.
6. Incorrect programming of Configuration register can cause system to fail
7. 21-bit address bus for program memory addressing capacity : 2 MB of memory
8. 12-bit address bus for data memory addressing capacity: 4 KB of memory

9. Has 77 Instructions
10. 16-bit instruction/data bus for program memory
11. 8-bit data bus for data memory
12. Has four timer, timer 0 (8 and 16 bit) timer 1 and 3(16 bit), timer 2 and 4 (8 bits only)
13. MCLR plays important role in reset operation of PIC
14. Watchdog timer is used to force reset operation if anything goes wrong
15. Prescaler in timer is used to increase the time delay
16. Sources of oscillator may be internal or external
17. Internal oscillator clock frequency is divided by four
18. OSC1 and OSC2 pins are used to connect the crystal or ceramic resonator, OSC2 pin can be used for alternate function as RA6 for I/O operation
19. Has five ports with 35 I/O pins [A (7), B, C, D (8), E (4), USB]
20. Port B pins have alternate function of accessing the external interrupts
21. Has various power managed modes (idle, sleep, run) which saves the power.
22. Configuration registers are used to set the operational conditions of PIC as reset voltage decision, Watch dog timer, background debugger etc .
23. Selection of supply voltage is important to avoid malfunctioning
24. Timer 0, 16 bit, TMR0H is loaded first then TMR0L
25. Highest delay will be generated when TMR0H = TMR0L = 00h

END...

Unit IV

5 Peripheral Support in PIC 18FXXXX

Q.1 Draw and explain functional diagram of timer 0 of PIC and differentiate between operating functions of timer 0, 1 and 2.

[SPPU : May-17, Dec-17, Marks 8]

Ans. : Timer 0 functional diagram :

The main use of timer is to generate the delay. Timer 0 is 8-bit or 16-bit timer with 8-bit software programmable prescaler.

Timer 0 : 8 bit mode :

- Timer 0 can operate as either a timer or a counter; the mode is selected by clearing the T0CS bit (T0CON<5>).
- In timer mode, the module increments on every clock by default unless a different prescaler value is selected.
- If the TMR0 register is written to, the increment is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the TMR0 register.
- The general configuration of Timer 0 - 8 bit is shown in Fig. Q.1.1.

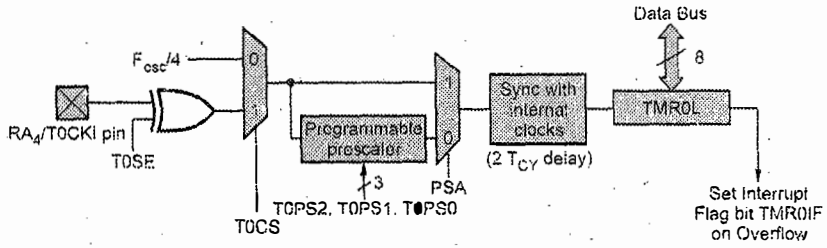


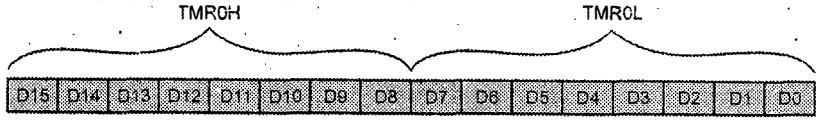
Fig. Q.1.1 : Timer 0, 8-bit

- The bock diagram is divided into three parts as 1. Selection of Clock, 2. Use of Prescaler, 3. Loading the Timer and checking for the Flag.
- The counter mode is selected by setting the T0CS bit (= 1). In counter mode, Timer 0 increments either on every rising or falling edge of pin RA4/T0CKI/C1OUT/RCV.
- The incrementing edge is determined by the Timer 0 Source Edge Select bit, T0SE (T0CON<4>); clearing this bit selects the rising edge.
- Restrictions on the external clock input : An external clock source can be used to drive Timer 0; however, it must meet certain requirements to ensure that the external clock can be synchronized with the internal phase clock (TOSC). There is a delay between synchronization and the onset of incrementing the timer/counter.
- An 8-bit counter is available as a pre-scaler for the Timer 0 module. The pre-scaler is not directly readable or writable; its value is set by the PSA and T0PS2:T0PS0 bits (T0CON<3:0>) which determine the pre-scaler assignment and pre-scale ratio.
- Clearing the PSA bit assigns the pre-scaler to the Timer0 module. When it is assigned, pre-scale values from 1:2 through 1:256, in power-of-2 increments, are selectable.
- When assigned to the Timer 0 module, all instructions writing to the TMR0 register (e.g., CLRF TMR0, MOVWF TMR0, BSF TMR0, etc.) clear the pre-scaler count.
- TMR0 interrupt is generated when the TMR0 register overflows from FFH to 00H in 8-bit mode, or from FFFFH to 0000H in 16-bit mode. This overflow sets the TMR0IF flag bit in INTCON register. The interrupt can be masked by clearing the TMR0IE bit (INTCON<5>). Before re-enabling the interrupt,

the TMR0IF bit must be cleared in software by the Interrupt Service Routine.

Timer 0 : 16 bit mode :

- 16-bit timer, 0000 to FFFFH.
- After loading TMR0H and TMR0L, the timer must be started.
- Count up till it reaches FFFFH, then it rolls over to 0000 and activate TMR0IF bit.
- Then TMR0H and TMR0L must be reloaded with the original value and deactivate TMR0IF bit.



MOVWF TMR0L : Loads 'W' register into TMR0L

MOVFF TMR0L, PORTB : Loads TMR0L value into PORT B.

- TMR0H is not the actual high byte of Timer 0 in 16-bit mode. It is actually a buffered version of the real high byte of Timer0 which is not directly readable nor writable.

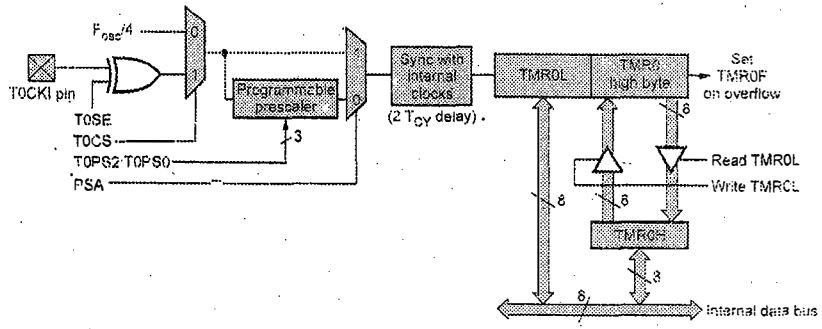


Fig. Q.1.2 : Timer 0 : 16 bit mode

- TMR0H is updated with the contents of the high byte of Timer 0 during a read of TMR0L. This provides the ability to read all 16 bits of Timer 0 without having to verify that the read of the

high and low byte were valid, due to a rollover between successive reads of the high and low byte.

- Similarly, a write to the high byte of Timer 0 must also take place through the TMR0H Buffer register. The high byte is updated with the contents of TMR0H when a write occurs to TMR0L. This allows all 16 bits of Timer 0 to be updated at once.

Note : Load TIMER0H first and then TIMER0L since TIMER0H will be kept in temporary reg. to avoid the errors during counting if TIMER0ON flag is set to high.

Comparison of Timers used in PIC

	Timer 0	Timer 1 & 3	Timer 2 & 4
Size of register	8-bits or 16-bits	16-bits	8-bits
Clock source (Internal)	$F_{osc}/4$	$F_{osc}/4$	$F_{osc}/4$
Clock source (External)	T0CKI pin	T13CKI pin or Timer 1 oscillator (T1OSC)	None
Clock scaling available (Resolution)	Prescaler 8-bits (1:2→1:256)	Prescaler 2-bits (1:1,1:2,1:4,1:8)	Prescaler (1:1,1:4,1:16) Postscaler (1:1→1:16)
Interrupt event	On overflow FF→00h	On overflow FFFFh→0000h	TMR REG matched PR2
Can wake PIC from sleep?	No	Yes	No

Table : Q.1.1

Q.2 Draw and explain functional diagram of timer 1 of PIC and differentiate between operating functions of timer 0, 1 and 2.

[SPPU : May-22, Marks 9, Dec-18, May-17, Marks 8]

Ans. : Timer1 : 16 bit operation

The Timer 1 module incorporates following features

1. Software programmed in 16-bit mode only and does not support 8-bit mode.
2. It has 2 bytes named as TMR1L and RMR1H which are readable and writable [It can count up 65.535 pulses in a single cycle].
3. Selectable clock source (internal or external) with device clock or Timer 1 oscillator internal options.
4. Has four prescale values [1:1, 1:2,1:4,1:8].
5. It has SFR as T1CON and TMR1IF.
6. The module incorporates its own low-power oscillator to provide an additional clocking option.
7. Used as a low-power clock source for the microcontroller in power-managed operation.
8. Interrupt : Generates an interrupt or sets a flag when it overflows.

   TMR1IF : Flag must be cleared to start the timer again.
9. Resetting timer 1 using CCP module,

   CCP1 in the Compare mode

   timer 1 and CCP1 compared at every cycle

   When a match is found, timer 1 is reset.

   - Timer TMR1 has a completely separate prescaler which allows 1, 2, 4 or 8 divisions of the clock input. The prescaler is not directly readable or writable. However, the prescaler counter is automatically cleared upon write to the TMR1H or TMR1L register. A simplified block diagram of the timer 1 module is shown in Fig. Q.2.1.

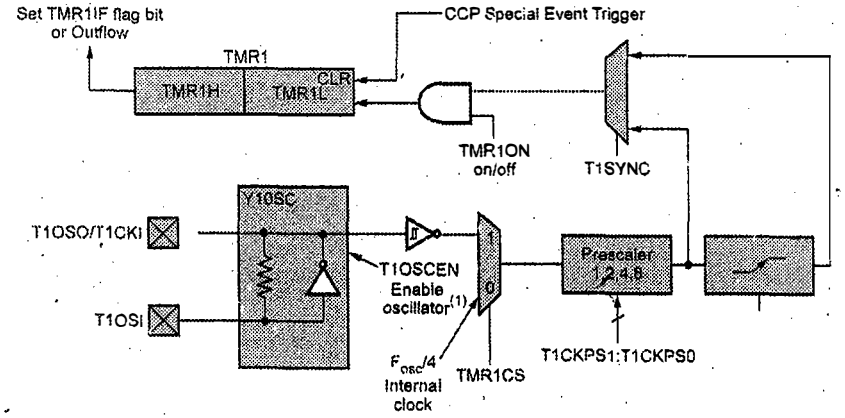


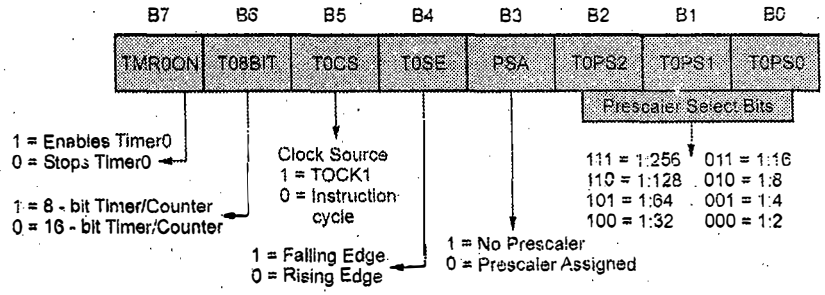
Fig. Q.2.1 : Timer 1 block diagram

- Timer 1 has the prescaler option but only supports for 1:1, 1:2,1:4,1:8.
- The module incorporates its own low-power oscillator to provide an additional clocking option. The timer 1 oscillator can also be used as a low-power clock source for the microcontroller in power-managed operation.
- Timer 1 can also be used to provide Real-Time Clock (RTC) functionality to applications with only a minimal addition of external components and code overhead.
- *Timer1 is controlled through the T1CON Control register shown in Fig. Q.19.1. It also contains the timer 1 oscillator enable bit (T1OSCEN). timer 1 can be enabled or disabled by* setting or clearing control bit, TMR1ON (T1CON<0>).

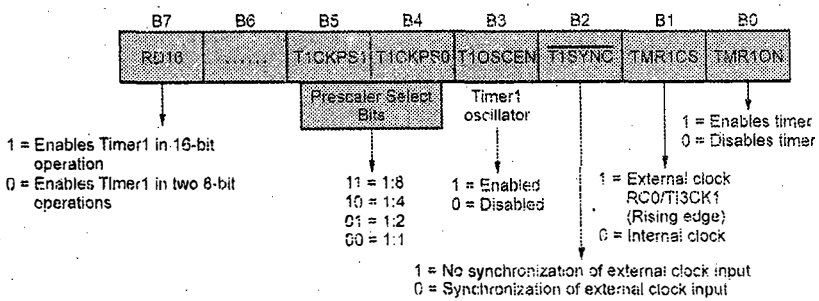
For comparison of timer 0, 1 and 3 refer Table Q.1.1

Q.3 Explain operation of T0CON and T1CON registers of PIC 18FXXX. ☞ **[Nov.-15, Marks 8]**

Ans. : T0CON Reg - Timer control Register-8 bit



T1CON : Timer 1 Control Register



Q.4 Explain operation of timer 2 of PIC 18FXXX.

☞ **[Nov.-15, Marks 8]**

Ans. : Timer 2 : 8 bit operation

- *It is an 8 bit register with 8-bit period register (PR2)- Fixed value*
- TMR2 and PR2 are readable and writable.
- TMR2 increments from 00 to the value equal to PR2.
- TMR2IF flag from PIR1 reg. is raised and TMR2 reset to 00.
- The clock source for TMR2 is Fosc/4 for both prescaler and postscaler options.
- There is no external clock source, hence cannot be use as counter.
- Three prescale values (Bit 1 - Bit 0) and 16 postscale values (Bit 6 - Bit 3) in T2CON register are used to calculate the delay.

- Flag (TMR2IF) is set when TMR2 matches PR2 : Can generate an interrupt.
- A simplified block diagram of timer 2 is shown in Fig. Q.4.1.

  Note : When using the TMR2 timer : one should know :

  1. Upon power-on, the PR2 register contains the value FFh;
  2. Both prescaler and postscaler are cleared by writing to the TMR2 register;
  3. Both prescaler and postscaler are cleared by writing to the T2CON register;
  4. On any reset, both prescaler and postscaler are cleared.

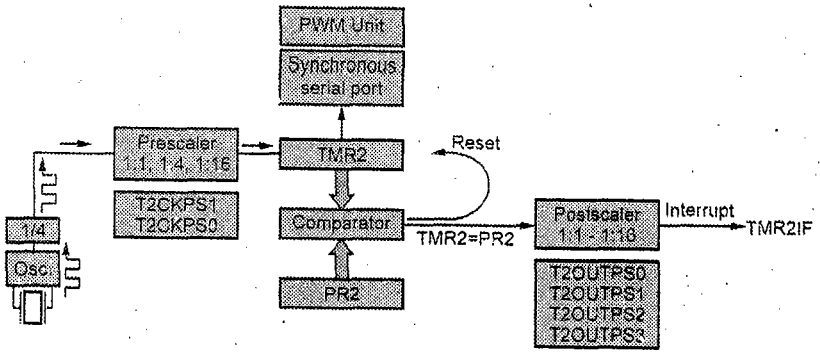


Fig. Q.4.1 : Timer 2 block diagram

- The module is controlled through the T2CON register Fig. Q.4.2, which enables or disables the timer and configures the prescaler and postscaler. Timer 2 can be shut off by clearing control bit, TMR2ON (T2CON<2>), to minimize power consumption. A range of 16 postscale options (from 1:1 through 1:16 inclusive) can be selected with the postscaler control bits,T2OUTPS3:T2OUTPS0 (T2CON<6:3>).

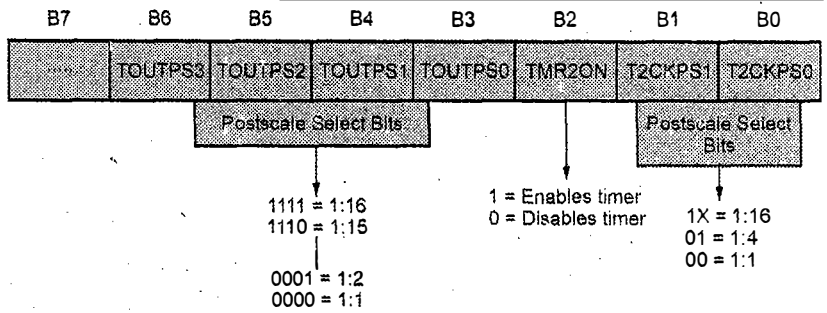
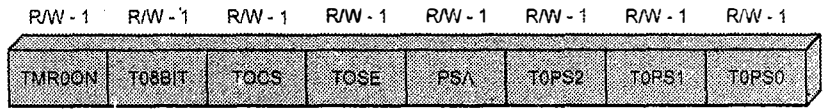


Fig. Q.4.2 : Timer 2 control register

- Timer 2 can also generate an optional device interrupt. The Timer2 output signal (TMR2 to PR2 match) provides the input for the 4-bit output counter/postscaler. This counter generates the TMR2 match interrupt flag which is latched in TMR2IF (PIR1<1>). The interrupt is enabled by setting the TMR2 Match Interrupt Enable bit, TMR2IE (PIE1<10).
- The unscaled output of TMR2 is available primarily to the CCP modules, where it is used as a time base for operations in PWM mode.
- Timer 2 can be optionally used as the shift clock source for the MSSP module operating in SPI mode.

Q.5 Write a C18 program to toggle all bits of port B continuously with delay of 10 ms using timer 0, 16 bit and no presclar.

[SPPU : May-18, May-22, Marks 8]



Ans. : Calculation of TMR0H and TMR0L values

1. Assume that crystal frequency = 10 MHz
2. Internal time delay = 4/ $(10*10^6)$ = 0.4 μs
3. N = 10 ms/0.4 μs = 25000

4. Count = 65536-25000 = $(40536)_{10}$
5. Hex value to be loaded = $(9E\ 58)_{16}$
6. Load TMR0H = 9EH and TMR0L = 58H

Program

```
#include <P18FXXXX.h>
void T0Delay(void);
void main(void)
{
  TRISB=0;                          // configure Port B as output
  While(1)
  {
    PORTB= 0x55;                    // Load bit patterns
    T0Delay ( );
    PORTB= 0xAA;
    T0Delay ( );
  }
}

void T0Delay ( )
{
    T0CON=0x08;                     // Timer0, 16 bit, no prescaler
    TMR0H=0x9E;                     // Load Higher byte in TMR0H
    TMR0L= 0x58;                    // Load Lower byte to TMR0L
    T0CONbits.TMR0ON=1;             // Start the timer for upcount
    while (INTCONbits.TMR0IF==0);   // Check for overflow
    T0CONbits.TMR0ON=0;             // Turnoff timer
    INTCONbits.TMR0IF==0;           // Clear the Timre0 flag
}
```

Q.6 Write a C18 program to generate square wave of 50 Hz continuously using timer 0, 16 bit and no prescaler.

Ans. : Square wave has 50 % duty cycle i.e equal on and off period

1. For 50 Hz frequency, Total time T = 1/ 50 Hz = 20 ms

   i.e. $T_{on} = T_{off} = 10$ ms

2. The calculations used for program 1 are same hence

   TMR0H = 9EH and TMR0L = 58 H

```
#include <P18FXXXX.h>
void T0Delay(void);
void main(void)
{
  TRISB=0;                          // configure port B as output
  While(1)
  {
    PORTBbits.RB1= 0;               // set RB1 bit high
    T0Delay ( );
    PORTBbits.RB1= 1;
    T0Delay ( );
  }
}

void T0Delay ( )
{
  T0CON = 0 x 08;                   // Timer0, 16 bit, no prescaler
  TMR0H = 0 x 9E;                   // Load Higher byte in TMR0H
  TMR0L= 0x58;                      // Load Lower byte to TMR0L
  T0CONbits.TMR0ON=1;               // Start the timer for upcount
  while(INTCONbits.TMR0IF==0);      // Check for overflow
  T0CONbits.TMR0ON=0;               // Turnoff timer
  INTCONbits.TMR0IF==0;             // Clear the Timer 0 flag
}
```

Q.7 Write a C18 program to generate frequency of 250 Hz on all PORTD lines continuously using Timer 0, 16 bit and no presclar.

Ans. : For 250 Hz frequency, Total time T = 1/ 250 Hz = 0.004 s i.e. $T_{on} = T_{off} = 0.002$ s

Calculation of TMR0H and TMR0L values

1. Assume that Crystal frequency = 10 MHz

2. Internal time delay = $4/(10*10^6) = 0.4\ \mu s$
3. N = 0.002/0.4 μs = 5000
4. 5000/20 = 250
5. Count = 65536-250 = $(65286)_{10}$
6. Hex value to be loaded = $(FF\ 06)_{16}$
7. Load TMR0H = FFH and TMR0L = 06H

```
#include <P18FXXXX.h>
 void T0Delay(void);
 void main(void)
 {
   Unsigned char x;
   TRISD=0;                        // configure Port D as output
   PORTD=0x55;
   While(1)
    {
      PORTD=~PORTD;                // Toggkle all bits of port D
      For(x=0; x<20; x++)
      T0Delay ( );
    }
  }

void T0Delay ( )
{
  T0CON=0x08;                      // Timer0, 16 bit, no prescaler
  TMR0H=0xFF;                      // Load Higher byte in TMR0H
  TMR0L= 0x06;                     // Load Lower byte to TMR0L
  T0CONbits.TMR0ON=1;              // Start the timer for upcount
  while(INTCONbits.TMR0IF==0);     // Check for overflow
  T0CONbits.TMR0ON=0;              // Turnoff timer
   INTCONbits.TMR0IF==0;           // clear the Timre0 flag
}
```

Note : If same program require to run with 1:4 prescaler then value for TMR0H and TMR0L are

1. Assume that crystal frequency = 10 MHz

2. Internal time delay = 4*4/(10*106) = 1.6 μs
3. N = 0.002/1.6 μs = 1250
4. Count = 65536 – 1250= $(64286)_{10}$
5. Hex value to be loaded = $(FB1E)_{16}$
6. Load TMR0H = FB H and TMR0L = 1EH
7. T0CON = 01

Q.8 Write a C18 program to generate frequency of 2500 Hz on all PORTC.2 continuously using timer 1 , 16 bit and no prescalar.

Ans. : For 2500 Hz frequency, Total time T = 1/ 2500 Hz = 400 μs i.e. $T_{on} = T_{off} = 200\ \mu s$

	R/W - 0	R-0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0	R/W - 0
T1CON	RD16	T1RUN	T1CKPS1	T1CKPS0	T1OSCEN	T1SYN	TMR1SC	TMR1ON

1. Calculation of TMR1H and TMR1L values,
   1. Assume that crystal frequency = 10 MHz
   2. Internal time delay = $4/(10*10^6) = 0.4\ \mu s$
   3. N = 200/0.4 μs = 500
   4. Count = 65536 – 550 = $(65036)_{10}$
   5. Hex value to be loaded = $(FE\ 1C)_{16}$
   6. Load TMR1H = FF H and TMR1L = 06 H
2. Program :

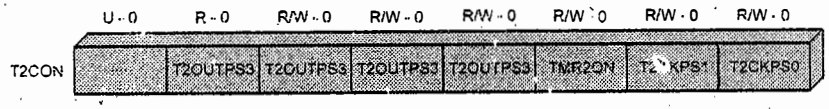
```
#include <P18FXXXX.h>
void T1Delay(void);
#define mybit PORTCbits.RC2
void main (void)
{
      TRISCbits.TRISC2=0;
      While(1)
      {
        mybit ^ =1
        T1Delay ( );
      }
```

```
}

void T1Delay ( )
{
  T1CON=0x00;                    // Timer1, 16 bit, no prescaler
  TMR1H=0xFE;                    // load Higher byte in TMR1H
  TMR1L= 0x06;                   // Load Lower byte to TMR1L
  T1CONbits.TMR1ON=1;            // Start the timer for upcount
  while(PIR1bits.TMR1IF==0);     // Check for overflow
  T1CONbits.TMR1ON=0;            // Turn off timer
  PIR1bits.TMR1IF==0;            // Clear the Timer 1 flag
}
```

Q.9 Using pre scaler and post scaler, find the largest time delay that can be generated using timer 2.

Ans. : Time delay using timer-2:




```
#include <P18FXXXX.h>
void T1Delay(void);
 #define mybit PORTCbits.RC2
void main (void)
{
        TRISCbits.TRISC2=0;
        T2CON=0X7B;                    // Timer 2, prescale=post=16
        TMR2=0X00;
        While(1)
        {
          PR2=255;                     // Load PR2 for highest value
         T2CONbits.TMR2ON=1;           // Start the timer
         While(PIR1bits.TMR2IF==0);    // Check for Timer 2 flag
         mybit=~mybit;                 // Toggle the bits
        T2CONbits.TMR2ON=0;            // Turn off timer
        PIR1bits.TMR2IF==0;            // Clear the Time 1 flag
        }
 }
```


Q.10 Draw and explain the interrupt structure of PIC18FXXX, what are peripheral Interrupts, IVT and ISR.

[SPPU : May-22, Nov.-16, Marks 8]

Ans. : Interrupt structure of PIC

- An interrupt is a communication process to provide services to different internal and external devices by executing ISR (Interrupt Service Routine).
  - A device
  - Requests the MPU to stop processing
  - Internal or external.
- The MPU
  - Acknowledges the request
  - Attends to the request
  - Goes back to processing where it was interrupted.
- Interrupts are events that cause your program to stop what it is doing in order to run an Interrupt Service Routine which will handle the event by taking whatever action is required before finally returning control to your main program.
- PIC18 has two vectors : High priority Interrupt (0008 H) and Low priority (0018H)
- Interrupts require special functions to service the events that cause them.

Types of Interrupts

The classification of interrupts in general is shown in Fig. Q.10.1.

PIC18 Interrupt Sources

- Interrupt Sources
- 3 or 4 External Interrupts (INT0-INT3)
  - Edge triggered
  - Rising or falling selected in INTCON2 register.

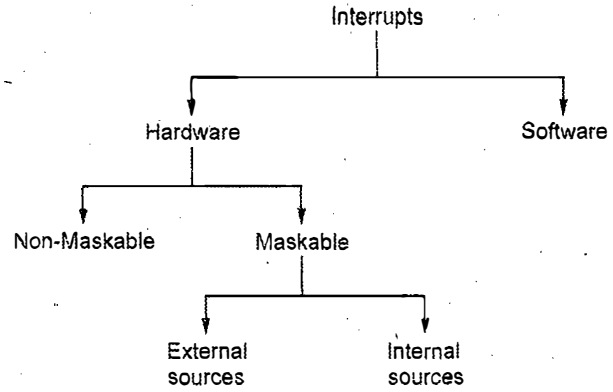


Fig. Q.10.1 Interrupt types

- PORTB interrupt on change (RB4 - RB7)
- Timer rollover/overflow events
- Comparator output change
- A/D conversion complete
- Communication channel events
- Other peripheral events.

Interrupt Structure : The general block schematic of Interrupt structure is shown in Fig. Q.10.2.

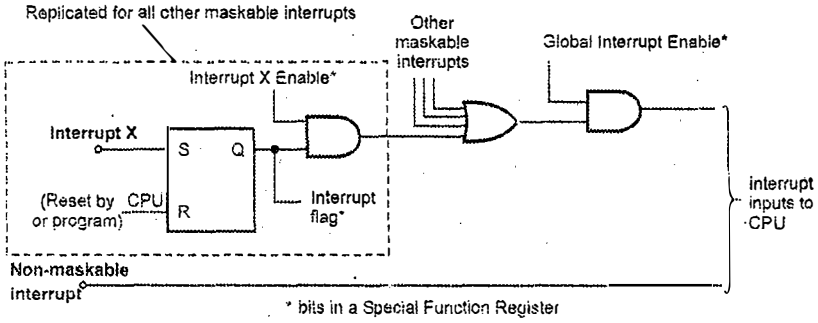


Fig. Q.10.2 General interrupt structure

According to the block schematic, the interrupt may occur due to reset by CPU or any other program execution used by different peripherals. The external interrupt on PORTB Lines (INT0-3) may be of high propriety and interrupts due to internal functions and peripherals are of low priority. When the CPU is interrupted either by external or internal interrupt, it sets the flag or theses interrupts are enabled by use of control registers. These interrupts are maskable interrupts. The vectored, internal or external interrupts causes the reset and all are enabled by use of Global interrupt enable bit in INCON0 control registers.

Q.11 Draw and explain INTCON register used in Interrupt of the PIC18F4550.

Ans. : INTCON register

The INTCON registers are readable and writable registers, which contain various enable, priority and flag bits.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

- Bit 7 GIE/GIEH : Global Interrupt Enable bit,

  When IPEN = 0 :

  1 = Enables all unmasked interrupts

  0 = Disables all interrupts

  When IPEN = 1 :

  1 = Enables all high priority interrupts

  0 = Disables all high priority interrupts

- Bit 6 PEIE/GIEL : Peripheral Interrupt Enable bit

  When IPEN = 0 :

  1 = Enables all unmasked peripheral interrupts

0 = Disables all peripheral interrupts

When IPEN = 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)
  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

Q.12 Draw and explain PIR1 register used in Interrupt of the PIC18F4550.

Ans. : PIR Registers

The PIR registers contain the individual flag bits for the peripheral interrupts. Due to the number of peripheral interrupt sources, there are two Peripheral Interrupt Flag Registers (PIR1, PIR2).

PIR1 : PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

- Bit 7 PSPIF(1) : Parallel slave port read/write interrupt flag bit
  1 = A read or a write operation has taken place
  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 SSPIF : 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 = MR1 register did not overflow

Q.13 Draw and explain the legacy and priority mode of PIC interrupts. **[SPPU : May-18, Marks 8]**

Ans. : The PIC microcontroller has the two types of interrupts as High priority and low priority depending on the internal and external sources.

Interrupt structure (legacy mode) (internal operations)

1. Operates on the internal operations as reset (MCLR), data (RB7) and clock (RB6) signals. MCLR is used for device reset and RB6 for serial clock, RB7 for serial data.
2. Even when the dedicated port is enabled, the ICSP functions remain available through the legacy port. When VIHH is seen on the MCLR/VPP/RE3 pin, the state of the ICRST/ICVPP pin is ignored.
3. The ICPRT Configuration bit can only be programmed through the default ICSP port (MCLR/RB6/RB7).
4. The power-managed Sleep mode in the PIC18F2455/2550/4455/4550 devices is identical to the legacy Sleep mode offered in all other PIC devices.

Fig. Q.13.1 shows interrupt structure in legacy mode.

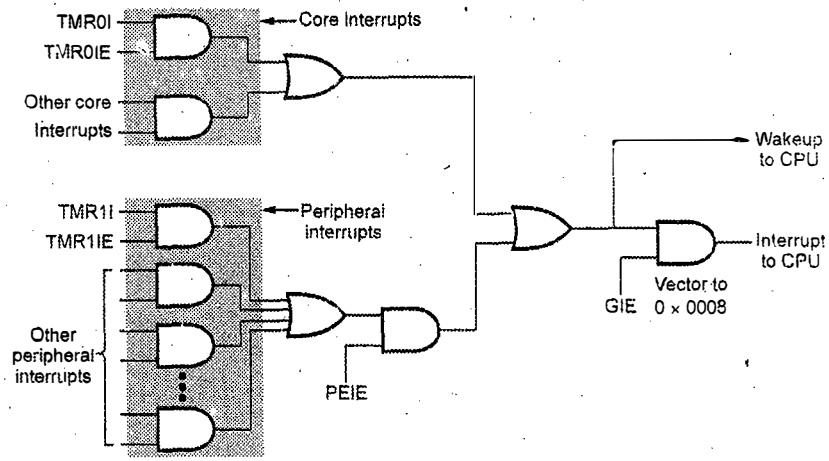


Fig. Q.13.1 Interrupt structure (legacy mode)

The CPU is interrupted by either by use of software interrupt as timer 0 enabled by T0CON register of any other core interrupts. The other sources used TIMER1 in CCP modes will be enabled by TMR1IE bit of T1CON register. Also the other internal peripherals such as ADC,

USART and so on. These interrupt requires the authentication and will be enabled by uses of PEIE bit in INTON0. The CPU is interrupted only when GIE bit of INCON0 is set either for low priority.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

Interrupt structure (Priority mode) (External operation)

The PIC18FXX2 devices have multiple interrupt sources and an interrupt priority feature that allows each interrupt source to be assigned a high priority level or a low priority level. The high priority interrupt vector is at 0008h and the low priority interrupt vector is at 0018h. High priority interrupt events will over-ride any low priority interrupts that may be in progress. The general block diagram of Priority mode is shown in Fig. Q.13.2.

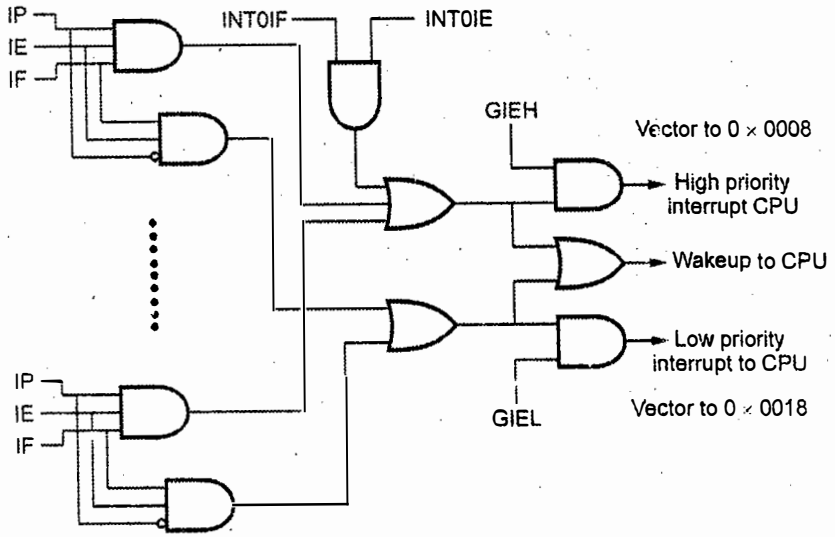


Fig. Q.13.2 Interrupt structure (priority mode)

- Each interrupt source, except INT0, has three bits to control its operation. The functions of these bits are :
  1. Flag bit to indicate that an interrupt event occurred.
  2. Enable bit that allows program execution to branch to the interrupt vector address when the flag bit is set.
  3. Priority bit to select high priority or low priority.

Interrupt handling steps :

- The interrupt priority feature is enabled by setting the IPEN bit (RCON<7>). When interrupt priority is enabled, there are two bits which enable interrupts globally.
- Setting the GIEH bit (INTCON<7>) enables all interrupts that have the priority bit set. Setting the GIEL bit (INTCON<6>) enables all interrupts that have the priority bit cleared.
- When the interrupt flag enable bit and appropriate global interrupt enable bit are set, the interrupt will vector immediately to address 000008h or 000018h, depending on the priority level.
- Individual interrupts can be disabled through their corresponding enable bits.
- When the processor receives the interrupt request, it completes the current instruction before jumping to the interrupt vector. Instruction execution in the PIC microcontroller can be one or two cycles long and when added to the two-instruction delay for calling the interrupt handler, the total delay (which is known as interrupt latency) is three or four instruction cycles.

Q.14 Write note on PORTB interrupt on Change.

Ans. : PORTB Interrupt-on-Change

- An input change on PORTB<7:4> sets flag bit, RBIF (INTCON<0>). The interrupt can be enabled/disabled by setting/clearing enable bit, RBIE (INTCON<3>). Interrupt priority for PORTB interrupt-on-change is determined by the value contained in the interrupt priority bit, RBIP (INTCON2<0>).

- External interrupts on the RB0/AN12/INT0/FLT0/SDI/SDA, RB1/AN10/INT1/SCK/SCL and RB2/AN8/INT2/VMO pins are edge-triggered. If the corresponding INTEDGx bit in the INTCON2 register is set (= 1), the interrupt is triggered by a rising edge; if the bit is clear, the trigger is on the falling edge. When a valid edge appears on the RBx/INTx pin, the corresponding flag bit, INTxIF, is set. This interrupt can be disabled by clearing the corresponding enable bit, INTxIE. Flag bit, INTxIF, must be cleared in software in the Interrupt Service Routine before re-enabling the interrupt.
- All external interrupts (INT0, INT1 and INT2) can wake-up the processor from the power-managed modes if bit, INTxIE, was set prior to going into the power-managed modes. If the Global Interrupt Enable bit, GIE, is set, the processor will branch to the interrupt vector following wake-up.
- Interrupt priority for INT1 and INT2 is determined by the value contained in the interrupt priority bits, INT1IP (INTCON3<6>) and INT2IP (INTCON3<7>). There is no priority bit associated with INT0. It is always a high priority interrupt source.

Q.15 Write an embedded C program to generate the delay of 100 ms using Timer Interrupt Program for Fosc = 48 MHz.

Ans. :

Step1 : CALCULATIONS of Delay (use Tmer 0 ,16 bit mode)

```
* required delay = 100ms
* TMR value = 0xFFFF - [(required time)/(4*Tosc*Prescaler)]
*           = 0xFFFF - [(0.1*48000000)/(4*256)]
*           = 0xFFFF - 0x124F
* TMR0 = 0xEDB0
* TMRH = 0xED
* TMRL = 0xB0
```

or

Note : SFRS used - No need in the exam.

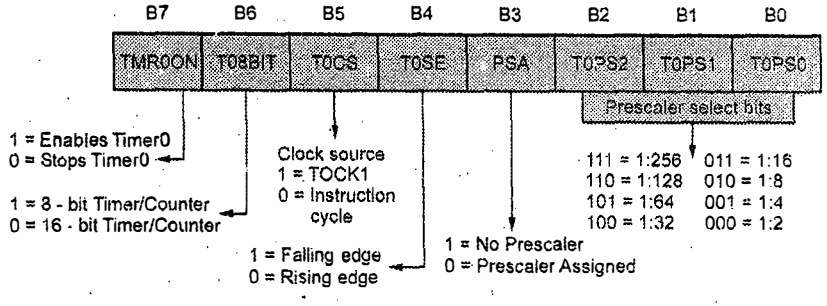
RCON Register

IPEN	SBOREN	—	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$

INTCON

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF

T0CON : Timer 0 control register-8 bit



Program :

```
void main(void)
{
  TRISB = 0x00;                  // Port B as output
  LATB = 0xFF;
  RCONbits.IPEN = 1;             // Priority Enable
  INTCONbits.GIEH = 1;           // High priority Interrupt
  INTCONbits.GIEL = 1;           // Enable Low priority interrupts
  INTCONbits.TMR0IE = 1;         // Enable Timer 0 interrupts
  INTCONbits.TMR0IF = 0;         // Disable Timer flag
  INTCON2bits.TMR0IP = 0;        // Disable Tier0 Priority
  T0CON = 0x07;                  // Stop the timer, Run in 16-bit mode,
                                 //Use system clock,Use a 1:256 prescaler
  TMR0H = 0xED;                  // Load Timer
  TMR0L = 0xB0;
  T0CONbits.TMR0ON = 1;          // Start the timer
  while(1);
```

```
 }
void interrupt low_priority timerinterrupt(void)
{
   if(TMR0IF == 1)//If timer 0 interrupt flag is set.....
    {
       T0CONbits.TMR0ON = 0;   // Stop the timer
       INTCONbits.TMR0IF = 0;
       TMR0H = 0xED;
       TMR0L = 0xB0;
        LATB =~LATB;
       T0CONbits.TMR0ON = 1;   // Start the timer
  }
}
```

Q.16 Explain the concept of CCP module with CCPX Control register in detail.

Ans. : Capture, Compare and PWM (CCP) Modules :

Each CCP (Capture/Compare/PWM) module contains a 16-bit register which can operate as a 16-bit Capture register, as a 16-bit Compare register or as a PWM Master/Slave Duty Cycle register.

- Capture, Compare and Pulse Width Modulation (PWM) module is associated with a control register (CCPxCON) and a data register (CCPRx).
- The data register in turn consists of two 8-bit register : CCPRxL and CCPRxH.
- The CCP modules utilize Timers 1, 2, 3, or 4, depending on the module selected.
- CCPR1H (high) and CCPR1L (low)
- 16-bit Capture register
- 16-bit Compare register
- Duty-cycle PWM register
- Timer 1 used as clock for Capture and Compare

- Timer 2 used as clock for PWM
- The assignment of a particular timer to a module is determined by the bit 6 and bit 3 of the T3CON register
- Following table shows the assignment of timers for CCP modes

CCP Mode	Timer Resource
Capture	Timer 1 or Timer 3
Compare	Timer 1 or Timer 3
PWM	Timer 2

CCPxCON register

	7	6	5	4	3	2	1	0
	–	–	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
value after reset	0	0	0	0	0	0	0	0

DCxB1 : DCxB0 : PWM duty cycle bit 1 and bit 0 for CCP module x

Capture mode : unused

Compare mode : unused

PWM mode : These two bits are the LSB's (bit 1 and bit 0) of the 10-bit PWM duty cycle.

CCPxM3 : CCPxM0 : CCP module x mode select bits

0000 = Capture/compare/PWM disabled (resets CCPx module)

0001 = Reserved

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

0100 = Capture mode, every falling edge

0101 = Capture 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 (CCPxIF bit is set)

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

1010 = Compare mode, generate software interrupt on compare match (CCP pin unaffected, CCPxIF bit is set).

1011 = Compare mode, trigger special event (CCPxIF bit is set)

For CCP1 and CCP2 : Timer 1 or Timer 3 is reset on event

For all other modules : CCPx pin is unaffected and is configured as an I/O port.

11xx = PWM mode

Q.17 Explain in detail with block schematic capture mode of CCP module. **[SPPU : Dec-17, Marks 8, Nov-16, Marks 6]**

Ans. : CCP in the capture mode

- CCPR1 captures the 16-bit value of Timer 1 : When an event occurs on pin RC2/CCP1.
- Interrupt request flag bit CCP1IF is set : Must be cleared for the next operation
- To capture an event
  - Set up pin RC2/CCP1 of PORTC as the input
  - Initialize Timer 1 : T1CON register
  - Initialize CCP1 : CCP1CON register
- The PIC18 event can be one of the following :
  1. Every falling edge
  2. Every rising edge

3. Every 4th rising edge
4. Every 16th rising edge

The block diagram of CCP in capture mode is shown in Fig. Q.17.1

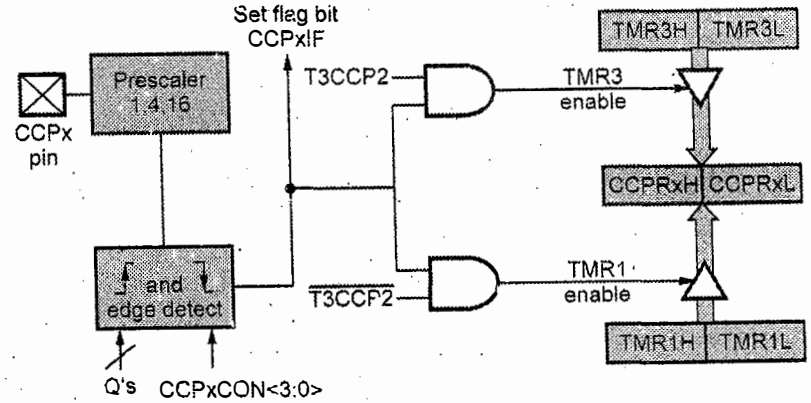


Fig. Q.17.1 Capture mode

- When a capture is made, the interrupt flag bit, CCPxIF is set. [PIR1 register]
- The CCPxIF flag must be cleared by software.
- In capture mode, the CCPx pin must be configured for input.
- The timer to be used with the capture mode must be running in timer mode or synchronous counter mode.
- To prevent false interrupt, the user must disable the CCP module when switching pre-scaler.
- The contents of TMR3H : TMR3L OR TMR1H : TMR1L are loaded into CCPRX register.
- PIR1 register

B7	B6	B5	B4	B3	B2	B1	B0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF

Q.18 Explain the compare mode of PIC in details

[SPPU : May -18, Marks 8, Nov.-16, Marks 6]

Ans. : CCP in compare mode

- In compare mode, the 16-bit CCPR1 (CCPR2) register value is constantly compared against either the TMR1 register pair value, or the TMR3 register pair value. When a match occurs, the RC2/CCP1 (RC1/CCP2) pin is :
  1. Driven high
  2. Driven low
  3. Toggle output
  4. Remains unchanged [Interrupt flag bit CCP1IF is set].
- The action on the pin is based on the value of control bits CCP1M3:CCP1M0 (CCP2M3:CCP2M0). At the same time, interrupt flag bit CCP1IF (CCP2IF) is set.
- Fig. Q.18.1 shows block diagram of CCP in compare mode.

To set up CCP1 in compare mode

1. Set up pin RC2/CCP1 of PORTC as output
2. Initialize timer 1 or 3 and CCP1
3. Stores the sum in the CCPRxH : CCPRxL register pair : Clear the flag CCP1IF.

Special Event Trigger

- The CCP1 and CCP2 modules can also generate this event to reset TMR1 or TMR3 depending on which timer is the base timer. The basic difference between capture and compare mode is that, in capture mode event is detected and then according to the flag contents of timer 1 or 3, is transferred to CCPRx register. But in compare mode the contents of timer 1 or 3 are compared with CCPRx register and when match found flag is raised to detect the event on CCPx pin.

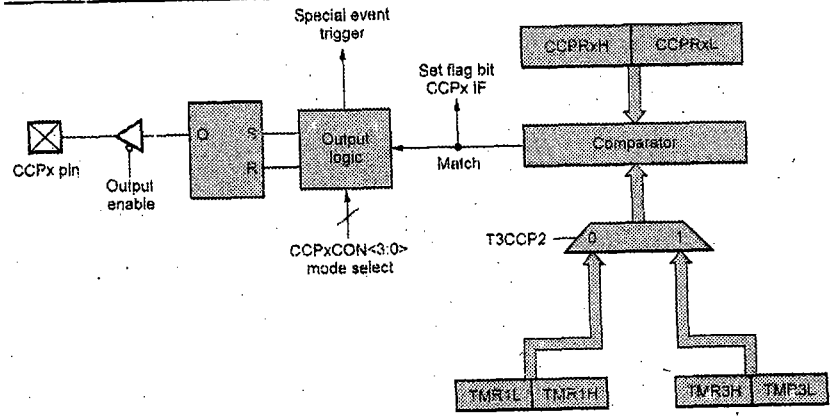


Fig. Q.18.1 CCP compare mode

Q.19 Explain in detail with block schematic PWM mode in CCP module.

[SPPU : May-17, Nov.-15, Marks 8]

Ans. : PWM mode

- CCP in PWM mode uses the tmer 2 to generate a pulse wave form for a given frequency/duty cycle. Duty cycle is the key performance measurement factor. It uses the CCPRx and PR2 register. PR2 is an 8-bit period register (PR2) whose value remains fixed. TMR2 increments from 00 to the value equal to PR2, TMR2IF flag from PIR1 reg. is raised and TMR2 reset to 00.The clock source for TMR2 is Fosc/4 for both prescaler and postscaler options. Fig. Q.19.1 shows block diagram for PWM mode. (Refer Q.19.1 on next page)
- To begin with the PWM mode the CCPx CON < 0:3> bits are set to 11XX and Pin RC2/CCP1 of PORTC is set high. The required duty cycle is obtained using CCPxCON<4:5> bits according to the waveforms shown in Fig. Q.19.2. (Refer Q.19.2 on next page)

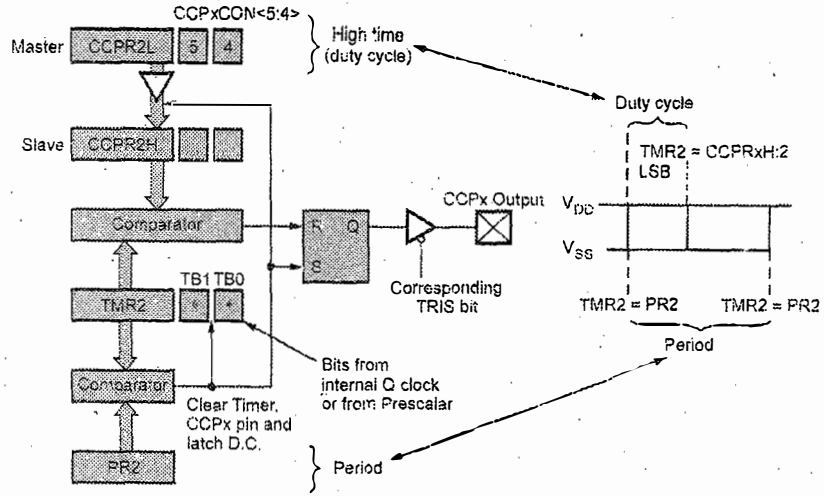


Fig. Q.19.1 PWM Mode

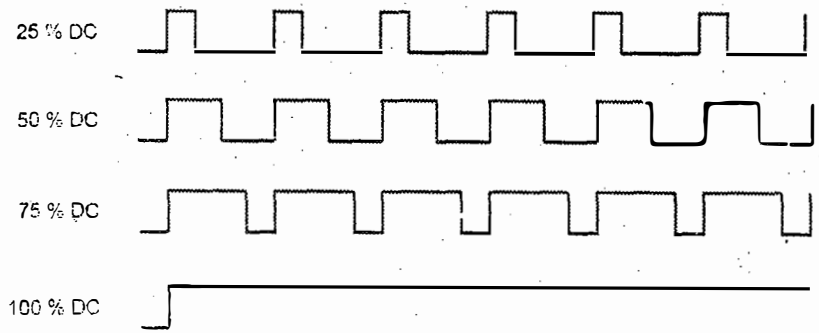


Fig Q.19.2 Duty cycle variations (Need to draw)

- The PWM mode uses timer 2 whose value is compared with the period register PR2 when match is found sets the FF and PWM wave is obtained on RC2 pin. At the same time it is set by match with contents of CCPR1 register. The PWM mode uses different SFRs as CCPxCON, T2CON, PIR1 and TMR2 is cleared CCPRxL. The configuration of PWM mode follows the following steps.

Step 1 : Calculation of PR2 = [Tpwm / * 4 * TOSC * N] –1

N = Presale factor 2,4,16 (if not mentioned by default 16 and TOSC=0.1 μs]. Choose Pre scaler [TMR2PRE] to ensure that PR2 is in the range of 0 to 255 for the desired PWM frequency.

Step 2 : Calculating the CCPRL1 value (Lower 8 bits) = % D * PR2

Step 3 : PWM duty cycle = (CCPRxL:CCPxCON<5:4>) * TOSC * N

DCpwm = %DC*Twpm = Desired PWM Duty Cycle (time, not %)

Step 4 : CCPxCON<5:4> for duty cycle bits to be set in CCPxCON register

Bits	% Duty cycle
0 0	0
0 1	25
1 0	50
1 1	75

Q.20 Write an embedded C program for 2.5 kHz and 75 % duty cycle PWM generation with N = 4

☞ **[SPPU : Dec-18, Marks 8, Dec-17, Marks 8]**

Ans. : **PWM Generation**

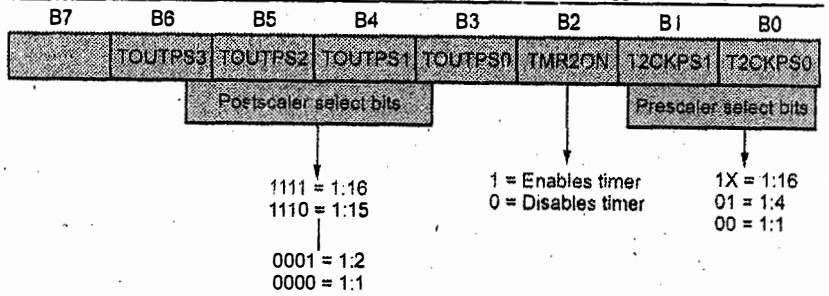
Assume that Fosc = 10 MHz

Step 1: Find value of PR2

PR2= [fosc/fpwm*4*N] –1=[10 MHz/2.5 kHz*4*4] –1 = 249;

Step 2 : Find Value of CCPR1L

CCPRxL= PR2*DC= 249*0.75= 186.75~186;

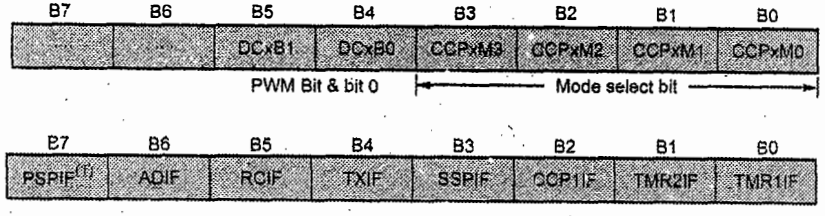


Step 3 : Set the TMR2 pre-scaler value, then enable TMR2 by writing to T2CON

T2CON=0x01;(pre-scaler = 4 00-1:1, 01-1:4; and 1X-1:16) 00000001

Step 4 : Configure the CCPx module for PWM mode set DC1B2 and DC1B1 for decimal portion of the duty cycle.

CCP1CON=0x3C;(CCPxCON<5:4> =11 for 75% DC&11XX--PWM)



Program :

```
#include <P18F458.h>
Void main (void)
{
CCP1CON=0;                  //clear the Reg
PR2=249;                    // load the PR2 value
CCPR1L=186;                 // 10% DC
TRISCbits.TRISC2=0;         // make PWM pin output
T2CON=0x01;                 // Timer 2, 4 prescalar, no post scalar
CCP1CON=0x3C;               // PWM mode 11 for DC1B1:DC1BO  for
                            // 75% duty cycle
TMR2=0;                     // Clear timer 2
T2CONbits.TIMER2ON=1;       // START TIMER 2
While (1)
{                           // Check for the timer flag
```



```
PIR1bits.TMR2IF=0;          // clear timer 2 flag.
While (PIR1bits.TMR2IF==0);  //wait for end of period
}
}
```

Q.21 Write an embedded C program for 1 kHz and 10 % duty cycle PWM generation. **[SPPU : May-22, Marks 8,May-15,16, Marks 8]**

Ans. : PWM Generation

Assume that Fosc = 10 MHz, If presclar is not given then N = 16

Step 1 : Find value of PR2

PR2 = [fosc/fpwm*4*N] –1= [10 MHz /1 kHz * 4 * 16] –1 =156;

Step 2 : Find value of CCPR1L

CCPRxL = PR2*DC= 155*0.1= 15.6~16;

Step 3 : Set the TMR2 pre-scaler value, then enable TMR2 by writing to T2CON

T2CON = 0x02; (pre-scaler=16 00- 1:1,01-1:4; and 1X-1:16)0000001x

Step 4 : Configure the CCPx module for PWM mode set DC1B2 and DC1B1 for decimal portion of the duty cycle.

CCP1CON = 0x0C; (CCPxCON<5:4> = 00 for 10% DC and 11XX-PWM)

Program :

```
#include <P18F458.h>
Void main (void)
{
CCP1CON=0;                  // clear the Reg
PR2=155;                    // load the PR2 value
CCPR1L=16;                  // 10 % DC
TRISCbits.TRISC2=0;         // make PWM pin output
T2CON=0x02;                 // Timer2, 16 prescalar, no post scalar
CCP1CON=0x0C;               // PWM mode 00 for DC1B1:DC1BO for
                            // 10 % duty cycle
TMR2=0;                     // Clear timer2
T2CONbits.TIMER2ON=1;       // START TIMER2
While (1)
{                           // Check for the timer flag
```

```
PIR1bits.TMR2IF=0;              // clear timer2 flag.
While (PIR1bits.TMR2IF==0);     //wait for end of period
}
}
```

Q.22 Explain various techniques to interface the DC motor for speed control with PIC controller.

Ans. : Speed control of DC motor

PWM (Pulse Width Modulation) :

- The speed of the motor depends on three factors; load, voltage and current. For a given fixed load we can maintain a steady speed by using a method called **Pulse Width Modulation (PWM)**. By changing the width of the pulse applied to the DC motor we can increase or decrease the amount of power provided to the motor, thereby increasing or decreasing the speed.
- Pulse-Width Modulation (PWM) or duty-cycle variation methods are commonly used in speed control of DC motors. The duty cycle is defined as the percentage of digital 'high' to digital 'low' plus digital 'high' pulse-width during a PWM period. Fig. Q.22.1 shows the 5 V pulses with 0 % through 50 % duty cycle.

Fig. Q.22.1 Pulses with 0 % through 50 % duty cycle

- The average DC voltage value for 0 % duty cycle is zero; with 25 % duty cycle the average value is 1.25 V (25 % of 5 V). With 50 % duty cycle the average value is 2.5 V and if the duty cycle is 75 %, the average voltage is 3.75 V and so on. The maximum duty cycle can be 100 %, which is equivalent to a DC waveform. Thus by varying the pulse-width, we can vary the average voltage across a DC motor and hence its speed. The speed of DC motor can be controlled using variety of methods as shown in Fig. Q.22.2 to Fig. Q.22.4, out of which H-bridge is mostly preferred.

- **NPN transistor speed control** : When motor is off, parallel diode acts as freewheeling for dissipation of energy.

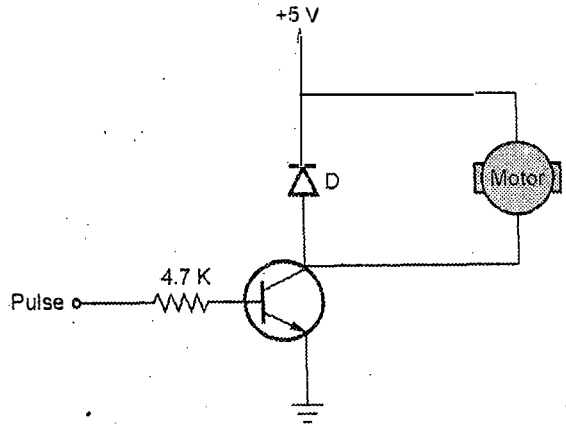


Fig. Q.22.2 NPN transistor speed control

- **Low value register to control the speed** : The series resistance controls the drop and effective flow of current.
- Fig. Q.22.3 Low value register to control the speed

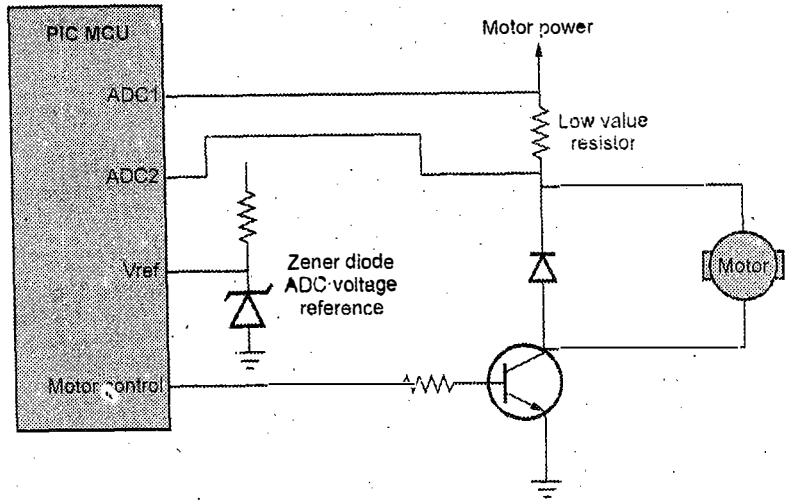


Fig. Q.22.3 : Transistor control

- **H- bridge speed control**

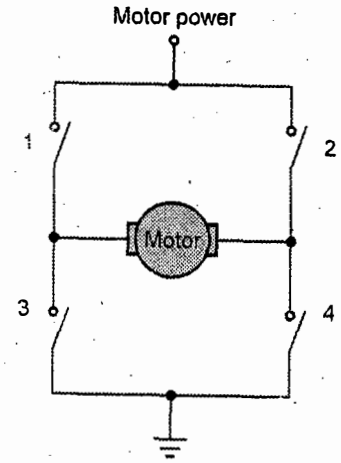


Fig. Q.22.4 H-bridge control

- In this application a fairly complex control application is used which allows forward and reverse, as well as speed control, of a dc motor using the full H-bridge circuit as shown in Fig. Q.22.4.

Q.23 Explain various steps used in PWM control of DC motor interfaced with PIC using H bridge

Ans.: Setup for PWM DC motor control

- The following steps should be taken when configuring the CCP module for PWM operation :

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. Make the CCP1 pin an output by clearing the TRISC<2> bit.
4. Set the TMR2 prescale value and enable timer 2 by writing to T2CON.
5. Configure the CCP1 module for PWM operation

- Usually H-bridge is preferred way of interfacing a DC motor. These days many IC manufacturers have H-bridge motor drivers available in the market like L293D is most used H-Bridge driver IC. H-bridge can also be made with the help of transistors and MOSFETs etc. rather of being cheap, they only increase the size of the design board, which is sometimes not required so using a small 16 pin IC is preferred for this purpose.

Working theory of H-bridge :

- The name "H-bridge" is derived from the actual shape of the switching circuit which controls the motion of the motor. It is also known as "full bridge". Basically there are four switching elements in the H-bridge as shown in the Fig. Q.23.1.

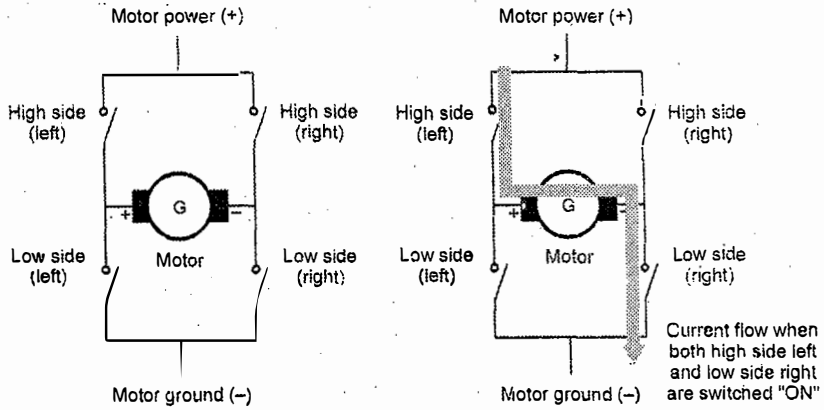


Fig. Q.23.1 H bridge PWM speed control

- As can seen in the Fig. Q.23.1. there are four switching elements named as "high side left", "high side right", "low side right", "low side left". When these switches are turned on in pairs, motor changes its direction accordingly. Like, if we switch on high side left and Low side right then motor rotate in forward direction, as current flows from power supply through the motor coil goes to ground via switch low side right. When you switch on low side left

and high side right, the current flows in opposite direction and motor rotates in backward direction. This is the basic working of H-bridge.

Uses the PWM mode to control the speed

- Uses the timer 2 with increased frequency either internal or external. The transistorized speed control is shown in Fig. Q.23.2.

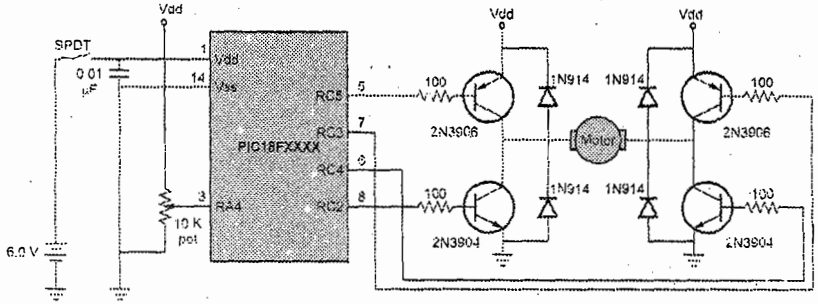


Fig. Q.23.2 PWM speed control

Q.24 Draw an interfacing diagram to interface the DC motor with PIC 18FXXX for speed control using PWM, Also write an embedded C program for increasing and decreasing speed with interrupt using key. ☞ **[SPPU : May-15, Marks 8]**

Ans. : Interfacing of DC motor with PIC

The interfacing diagram of DC motor with speed control using PWM mode of CCP is shown in Fig Q.24.1.

Program :

Assume that Fosc = 10 MHz, Use presclar N = 4, Duty cycle = 10% PWM frequency = 2.5 kHz

*/

Step 1 : Find value of PR2

PR2= [fosc/fpwm*4*N] – 1= [10 MHz/2.5 kHz * 4 * 4] – 1
= 249 or

0xF9;

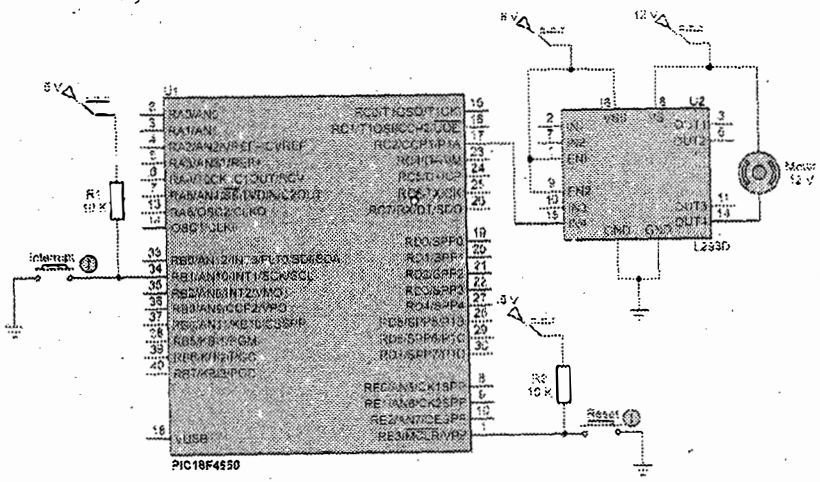


Fig. Q.24.1 DC motor PWM speed control using IC 293

Step 2 : Find value of CCPR1L

CCPRxL= PR2*DC= 249*0.1= 24.9=24 or 0x18

Step 3 : Set the TMR2 pre-scaler value, then enable TMR2 by writing to T2CON

T2CON = 0x01; (pre-scaler= 00- 1:1, 01- 1:4; and 1X- 1:16)] 0000001x

Step 4 : Configure the CCPx module for PWM mode set DC1B2 and

DC1B1 for decimal portion of the duty cycle.

CCP1CON=0x0C; (CCPxCON<5:4> =00 for 10% DC and 11XX-- PWM)

```
*/
#include<p18f4550.h>
unsigned char count=0;
bit TIMER,SPEED_UP;
```

```
void interrupt timer interrupt(void)
{
   if (INT1IF)                          // If the external interrupt flag is 1,
do .....
   {
     INT1IF =  0;                       // Reset the external interrupt flag
     if(SPEED_UP)
     {
       if(count < 8)
       {
         count++;
         CCPR1L = 0x18 + (count * 25);  //Increment duty cycle
       }
        else SPEED_UP = 0;
     }
     else
     {
       if(count > 0)
       {
         count--;
         CCPR1L = 0x18 + (count * 25);  //Decrement duty cycle
       }
       else SPEED_UP = 1;
     }
   }
}
void main(void)
{
   TRISBbits.TRISB1   = 1;              // interrupt pin as input
   TRISCbits.TRISC2   = 0;              //CCP1 pin as output
   CCP1CON =  0b00001100;               //Select PWM mode
   CCPR1L  =  0x18;                     //Duty cycle 10 %
   T2CON   =  0b00000001;               //Prescalar = 4; timer 2 OFF
   PR2     =  0xF9;                     //Period Register

   INT1IE =  1;                         //Enable external interrupt INT1
```



```
   INTEDG1 =  0;                        //Interrupt on falling edge
   GIE     =  1;                        // Enable global interrupt
   SPEED_UP = 1;
   TMR2ON = 1;                          //Timer 2 ON
   while (1);                           //Loop forever
}
```

Q.25 Draw and explain the block schematic of ADC in PIC 18FXXXX with features

☞ [SPPU : May-22, Marks 9, Dec.-17,18, Marks 8]

Ans. : **Features of ADC in PIC 18F4550**

- It has 10 bit ADC (Resolution - 10 bit)
- The ADC module can 12 channels associated with port A,E,B
- The converted binary output data is held in two registers ADRESL and ADRESH
- Vdd can be used as source for Vref or connecting to external device source
- The conversion time is decided by Fosc- cannot be shorter than 1.6 ms (40 MHz)
- It allows the differentiation of Vref + and Vref –
- All the features are programmed by ADCON0, ADCON1 and ADCON2
- The A/D allows conversion of an analog input signal to a corresponding 10-bit digital number.
- The A/D module has five registers.

  A/D Result High Register (ADRESH)

  A/D Result Low Register (ADRESL)

  A/D Control Register 0 (ADCON0)

  A/D Control Register 1 (ADCON1)

  A/D Control Register 2 (ADCON2)

- The A/D converter has a unique feature of being able to operate while the device is in SLEEP mode. To operate in SLEEP, the A/D conversion clock must be derived from the A/D's internal RC oscillator.
- The output of the sample and hold is the input into the converter, which generates the result via successive approximation.
- The functional diagram of ADC in PIC is as shown in Fig. Q.25.1

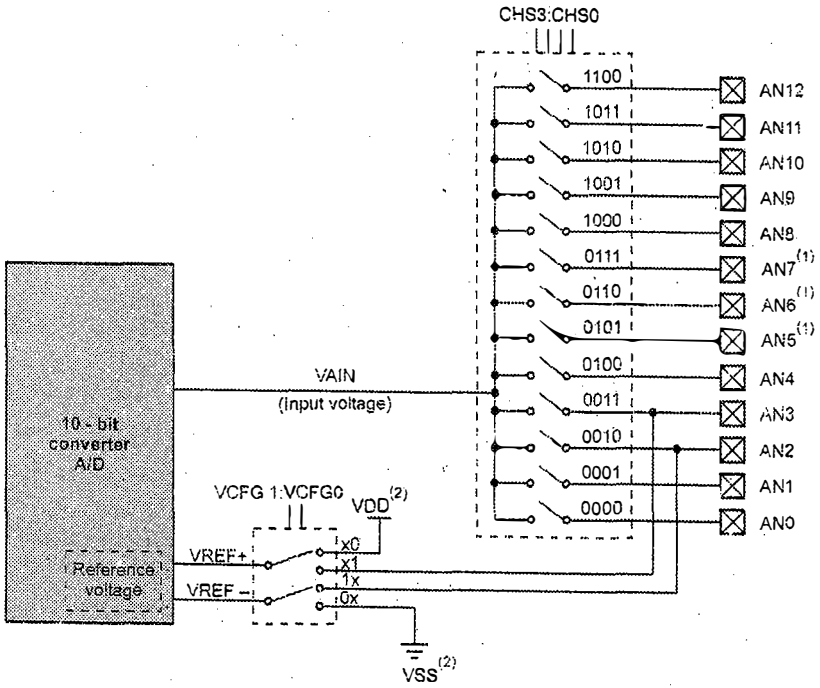


Fig. Q.25.1 ADC functional diagram

- A device RESET forces all registers to their RESET state. This forces the A/D module to be turned off and any conversion is aborted.

- Each port pin associated with the A/D converter can be configured as an analog input (RA3 can also be a voltage reference) or as a digital I/O.
- The ADRESH and ADRESL registers contain the result of the A/D conversion. When the A/D conversion is complete, the result is loaded into the ADRESH / ADRESL registers, the GO/DONE bit (ADCON0<2>) is cleared, and A/D interrupt flag bit, ADIF is set.
- Channels are selected by use of CHS2:CHS0 of DADCON0 register
- After the A/D module has been configured as desired, the selected channel must be acquired before the conversion is started. The analog input channels must have their corresponding TRIS bits selected as an input.
- An acquisition time can be programmed to occur between setting the GO/DONE bit and the actual start of the conversion.

Q.26 Explain various registers used by ADC in PIC 18FXXXX.

Ans. : Various registers of ADC:

1. ADCON0 : A/D control register 0

The ADCON0 register controls the operation of the A/D module. ADCON0 register is used to set the conversion time and select the channels. For power saving ADC feature is turned off when power up. And turned on with ADON bit when required. GO/DOWN bit is used for start and monitor the end of conversion.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$	ADON
bit 7							bit 0

ADC control register 0

2. ADCON1 : A/D control register 1

The ADCON1 register, configures the functions of the port pins. It is used to set the reference voltages.

U-0	U-0	R/W-0	R/W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

ADC control register 1

3. ADCON2 : A/D control register 2

The ADCON2 register, configures the A/D clock source, programmed acquisition time and justification. After conversion data in the ADRESL and ADRESH is right or left justified by ADFM bit

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	---	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7							bit 0

ADC control register 2

Block schematic of ADC in PIC 18F4550

- The analog reference voltage is software selectable to either the device's positive or negative supply voltage (VDD and VSS), or the voltage level on the RA3/AN3/ VREF+ pin and RA2/AN2/VREF – pin.

Q.27 Explain the block schematic of ADC result register in PIC 18FXXXX with features

Ans. : ADC result register

- The ADRESH : ADRESL register pair is the location where the 10-bit A/D result is loaded at the completion of the A/D conversion. This register pair is 16-bits wide.

- The A/D module gives the flexibility to left or right justify the 10-bit result in the 16-bit result register. The A/D format select bit (ADFM) controls this justification.
- The operation of the A/D result justification. The extra bits are loaded with '0's. When an A/D result will not overwrite these locations (A/D disable), these registers may be used as two general purpose 8-bit registers.
- Result registers are shown in Fig. Q.27.1

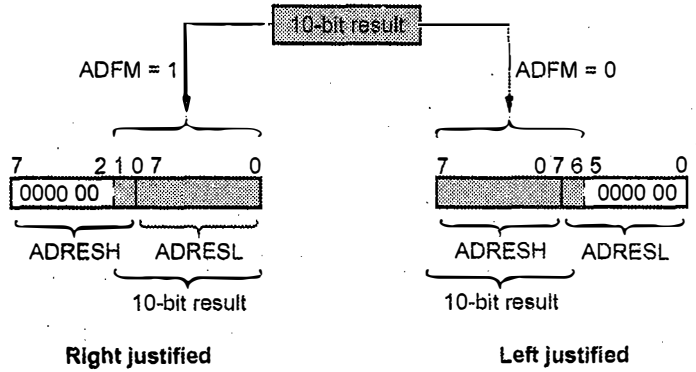


Fig. Q.27.1 ADC result register

Q.28 Explain the use of PIC ADC to interface the temp sensor for measuring temperature of room and display on LCD

[SPPU : May-17, Marks 8]

Or Draw an interfacing diagram to interface ADC and write an embedded C program to accept data on any channel.

[SPPU : Nov.-15, Marks 8]

Or Draw an interfacing of temp sensor to PIC using serial ADC and indicate excess temperature when exceeds the set point by LED.

[SPPU : May-16, Marks 8]

Note : Solution problem statements modified --- Interfacing diagram will change.

Ans. : Sensor interface to ADC

LM35 sensor is used for measurement of temperature. It is a precision integrated-circuit temperature sensor, whose output voltage is linearly proportional to the Celsius (Centigrade) temperature. It also gives the linearity of +10 mV/°C change in temperature. The LM35 does not require any external calibration or trimming to provide typical accuracies of ±1.4°C at room temperature. It can be used with single power supplies of 4 V to 30 V. It has very low self-heating as it draws only 60 μA from its supply. The LM35 is rated to operate over a – 55 °C to +150 °C temperature range. Normal temperature of cabin is as good as room temperature.

The interfacing of LM 35 with PIC is shown in Fig. Q.28.1

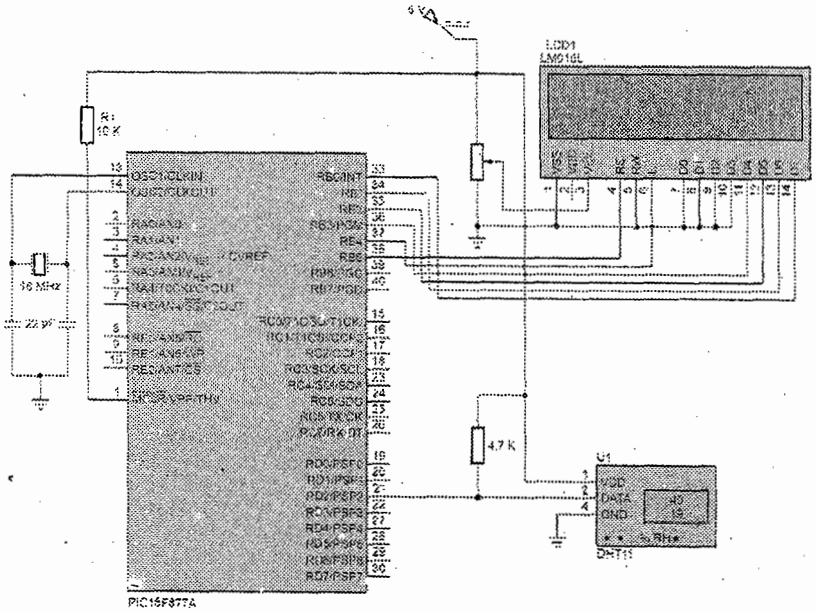


Fig. Q.28.1 (a) Interface with LCD display

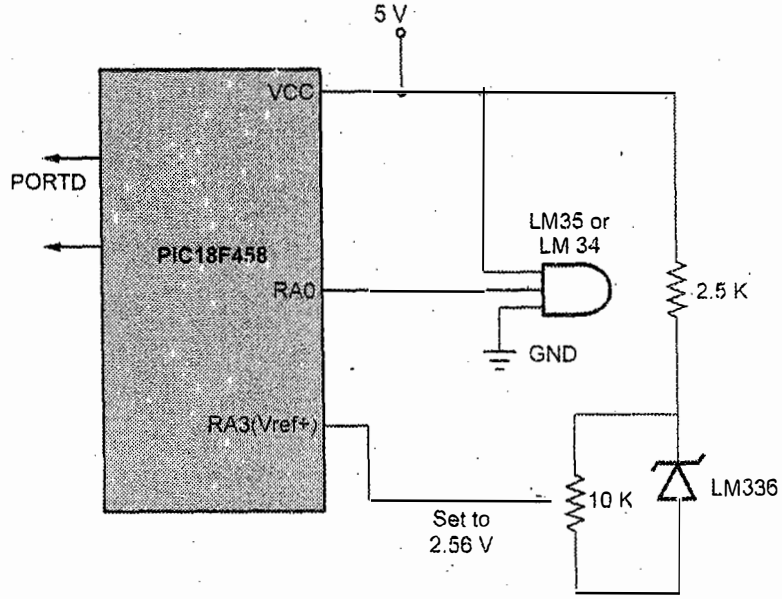


Fig. Q.28.1 (b) Sensor interface to PIC

Program

```
#include <p18f4550.h>
#include <stdio.h>
#include "LCD_SIT.h"
const unsigned char LCD_Data1[ ]={"ADC Value="};
void DisplayResult(unsigned short hexVal);

void main()
{
  unsigned char z=0,hexVal;
    TRISEbits.RA0 = 1;          // ADC channel 6 input
    TRISEbits.RA1 = 1:          // ADC channel 7 input
    Init_LCD();
   ADCON1=0x0E;                 // Voltage Reference Configuration bit?
                                // VSS, A/D
                                // Port Configuration Control bits -
                                // DDDDAAAAAAAA
    ADCON2=0xAE;     //A/D Result Format Select bit -
                     //Right justified,
```

```
                    //A/D Acquisition Time Select bits - 12
                    //TAD, ADC conversion Clock Select bits -
                    //FOSC/64
    Lcdcmd (0x80);
    MSdelay (50);
    for (z=0;z<10;z++)
    {
    lcddata(LCD_Data1[z]);
    MSdelay(50);
    }
    while (1)
    {
    ADCON0bits.ADON = 1;        //A/D On bit- on
    ADCON0bits.CHS = 0x00;      //Analog Channel Select bits?
                                // ch n0-7(ADC1)
    ADCON0bits.GODONE = 1;      //A/D Conversion Status bit --
                                //conversion in progress
    while(ADCON0bits.GO_DONE == 1 );
    ADCON0bits.ADON = 0; //A/D On bit- off
    DisplayResult(ADRES);

    }
}

void DisplayResult(unsigned short hexVal)
{
  unsigned char i,text[16];
  unsigned short tempv;
    tempv = hexVal;
    hexVal = (5200/1024)*tempv;
    lcdcmd(0x8A);
    MSdelay(50);
    sprintf(text,"%04dmv",hexVal);
    for(i=0;i<6;i++)
    {
    lcddata(text[i]);
    MSdelay(50);      }

    lcdcmd(0xC0);
    MSdelay(50);
    for(i=0;i<10;i++)
    {
```



```
    if(tempv & 0x200)
    {
    lcddata('1');
    MSdelay(50);
    }
    else
    {
    lcddata('0');
    MSdelay(50);
    }
    tempv<<=1;
    }
}
```

Important Points to Remember

1. PIC18F has 4 timers, timer 0 : 8 or 16 bit, timer 1 and 3 : 16 bits, timer 2 : 8 bit
2. Each timer has its control register TCONx
3. Timer is used to find the time delay.
4. To increase the required delay pre and post scaling is used.
5. Timer 0, 16 bit, TMR0H is loaded first then TMR0L
6. Highest delay will be generated when TMR0H=TMR0L= 00h
7. PIC18F4550 has 3 External Interrupts (INT0-2)
8. Interrupts are categorized as low priority and high priority
9. Interrupts can be edge or level triggered.
10. INTCON is used to enable or disable the interrupts globally
11. Port B bits (RB0-2) are specifically used for detect any interrupt change.
12. Capture : The CCP pin can be set as an input to record the arrival time of a pulse. In this CCP module may use either timer 1 or timer 3 to operate.
13. Compare : The CCP pin is set as an output and at a given count, it can be driven low, high or toggled

14. Pulse Width Modulation (PWM) : The CCP pin is set as an output and the duty cycle of a pulse can be varied. In PWM mode, either timer 2 or timer 4 may be used
15. The speed of the motor depends on three factors; load, voltage and current. For a given fixed load we can maintain a steady speed by using a method called **Pulse Width Modulation (PWM)**.
16. PIC 18F4550 has 10 bit resolution and 13 channels [Port A, B and E].
17. ADFM bit of ADCON2 is used to display the results in ADRESH : ADRESL. Left justified or right justified
18. Required reference voltages are obtained using ADCON1.
19. ADCON0 is used to select the required channel.
20. Sensors can be interfaced to PIC 18F4550 using ADC.

Registers and SFRS used

Special function registers for interrupts

The INTCON Registers are readable and writable registers, which contain various enable, priority and flag bits.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE/GIEH	PEIE/GIEL	TMR0IE	INT0IE	RBIE	TMR0IF	INT0IF	RBIF
bit 7							bit 0

INTCON2 Register

	R/W-1	R/W-1	R/W-1	U-0	R/W-1	U-0	R/W-1
	INTEDG0	INTEDG1	INTEDG2	–	TMR0IP	–	RBIP

ITCON3 Register

R/W-1	R/W-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
INT2IP	INT1IP	–	INT2IE	INT1IE	–	INT2IF	INT1IF
bit 7							bit 0

PIR Registers

The PIR registers contain the individual flag bits for the peripheral interrupts. Due to the number of peripheral interrupt sources, there are two Peripheral Interrupt Flag Registers (PIR1, PIR2).

PIR1 : PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

PIR2 : PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 2

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
–	–	–	EEIF	BCLIF	LVDIF	TMR3IF	CCP2IF
bit 7							bit 0

PIE1 : PERIPHERAL INTERRUPT ENABLE REGISTER 1

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

RCON REGISTER :

R/W-0	U-0	U-0	R/W-1	R-1	R-1	R/W-0	R/W-0
IPEN	–	–	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$
bit 7							bit 0

ADCON0 : A/D control register 0

The ADCON0 register controls the operation of the A/D module. ADCON0 register is used to set the conversion time and select the channels. For power saving ADC feature is turned off when power up. and turned on with ADON bit when required. GO/DOWN bit is used for start and monitor the End of conversion.

U-0	U-0	R/W-0	R/W-0	W-0	R/W-0	W-0	R/W-0
—	—	CHS3	CHS2	CHS1	CHS0	GO/ $\overline{\text{DONE}}$	ADON
bit 7							bit 0

ADCON1 : A/D control register 1

The ADCON1 register configures the functions of the port pins. It is used to set the reference voltages.

U-0	U-0	W-0	W-0	R/W-0(1)	R/W(1)	R/W(1)	R/W(1)
—	—	VCFG1	VCFG0	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

ADCON2 : A/D control register 2

The ADCON2 register configures the A/D clock source, programmed acquisition time and justification. After conversion data in the ADRESL and ADRESH is right or left justified by ADFM bit

R/W-0	U-0	W-0	W-0	W-0	W-0	W-0	R/W-0
ADFM	---	ACQT2	ACQT1	ACQT0	ADCS2	ADCS1	ADCS0
bit 7							bit 0

END...

Unit V

6 Real Word Interfacing with PIC18FXXXX

Q.1 Draw and explain port structure of PIC18 microcontroller with different registers used in programming

[SPPU : May-22, Marks 9, May-16,17, Dec-17, Marks 8]

Ans. : Port structure of PIC

The PIC18 family has five ports Port A (6), B (8), C (8), D (8), E (3) with 33 I/O lines. These lines can be used for data transfer between working register 'W' and peripherals and vice a versa. Each port can be configured as input or output. All Ports are bidirectional ports. Each port has some other functions such as timer, ADC, interrupts and serial communication. Port pins are assigned with multiple task and one of them will be available and active at a time. All the functions cannot be active simultaneously.

- Some ports have 8 bits, while others may not.
- Each port has three registers for its operation
- **TRISx register (Data direction register) :** For most ports, the I/O pin's direction (input or output) is controlled by the data direction register TRISx (x = A,B,C,D,E) : a '1' in the TRIS bit corresponds to that pin being an input, while a '0' corresponds to that pin being an output.
- **PORT register :** The PORTx register is the latch for the data to be output. Reading PORTx register read the status of the pins, whereas writing to it will write to the port latch.
- **LAT register (Output latch) :** The data latch register is useful for read-modify-write operations on the value that the I/O pins are driving.

Upon reset all ports are configured as input-TRISx register has 0FFh.

The detailed structure of PIC port is shown in Fig. Q.1.1 with SFR's used for data and direction control.

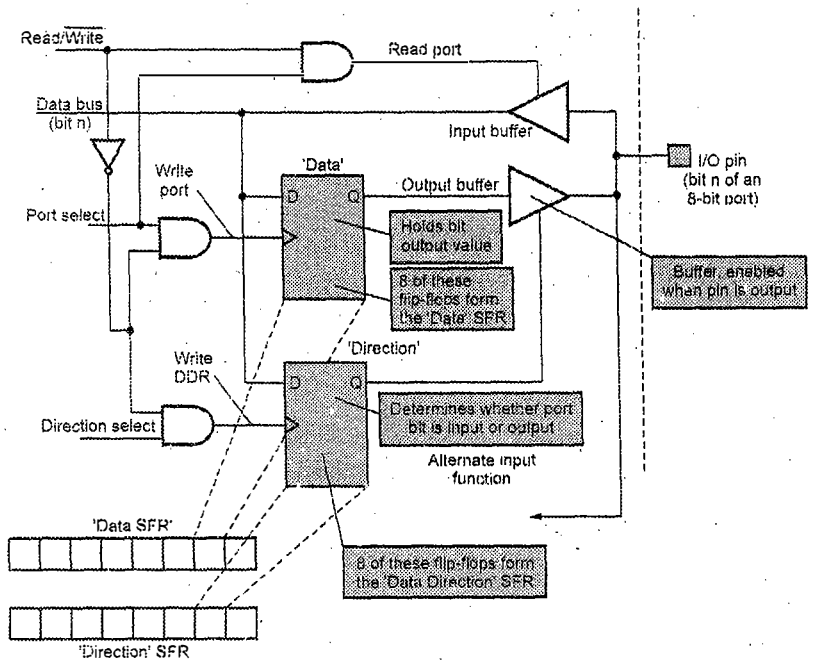


Fig. Q.1.1 Input / Output combined port structure

Each port in the PIC has three internal D flip-flops (latches)

- Data latch to hold the output data
- TRIS latch to setup data direction (Read or Write into or from the pin)
- Input latch for input data
- The Port can be configured for write operation according to the setting of Direction Control register Called as TRISx = 0x00 i.e.

port is configured as output. (Data transfer from data latch to port). The data written may be 0 or 1.

- The Port can be configured for read operation according to the setting of Direction Control register Called as TRISx = 0xFF i.e. port is configured as input. (Data transfer from port to data lat h). The data read may be 0 or 1.
- **During port read operation :** TRISx = 0xFF, R/W-1, Port select = 1, input buffer activated by read port line signal, write port and output buffer is disabled, Data from port pin either 0, 1 is placed on data line and stores in input data latch.
- **During port write operation :** TRISx = 0x00, R/W-0, Port select = 1, input buffer deactivated by read port line signal, write port and output buffer is enabled, Data from input buffer is placed on port pin either 0, 1.

Q.2 Draw and explain the port structure of PIC 18FXXXX for writing 0, 1 to port pins.

Ans. : Port structure for reading writing data

The PIC18 family has five ports Port A (6), B (8), C (8), D (8), E (3) with 33 I/O lines. These lines can be used for data transfer between working register 'W' and peripherals and vice a versa. Each port can be configured as input or output. Each port has three registers for its operation :

- **TRISx register (Data direction register) :** For most ports, the I/O pin's direction (input or output) is controlled by the data direction register **TRISx** (x = A,B,C,D,E) : a '0' corresponds to that pin being an output.
- **PORT register :** The PORTx register is the latch for the data to be output. Reading PORTx register read the status of the pins, whereas writing to it will write to the port latch.

- **LAT register (output latch) :** The data latch register is useful for read-modify-write operations on the value that the I/O pins are driving.
- **Upon reset all ports are configured as input-TRISx register has 0FFh.**
- Each port in the PIC has three internal D flip-flops (latches)
  - Data latch to hold the output data
  - TRIS latch to setup data direction
  - Input latch for input data

The configuration of port for writing 1 or 0 to port pin is shown in Fig. Q.2.1 and Q.2.2.

Writing 0 to Port pin

TRISx = 0 x 00 port is configured as output.

Data latch output : Data bus = 0, Write port pin used to clock the data, Q = 0, $\overline{Q}$ = 1, OR gate = 1, P Gate = OFF, Port pin = 0

TRIS latch output : Data bus = 0, Write TRIS pin used to clock the data, Q = 0, $\overline{Q}$ = 1, AND gate = 1, N Gate = ON, Port pin = 0

Writing 1 to Port pin

TRISx = 0 x 00 port is configured as output.

Data latch output : Data bus = 1, Write port pin used to clock the data, Q = 1, $\overline{Q}$ = 0, OR gate = 0, P Gate = ON, Port pin = 1

TRIS Latch output : Data bus = 1, Write TRIS pin used to clock the data, Q = 0, $\overline{Q}$ = 1, AND gate = 0, N Gate = OFF port pin = 0

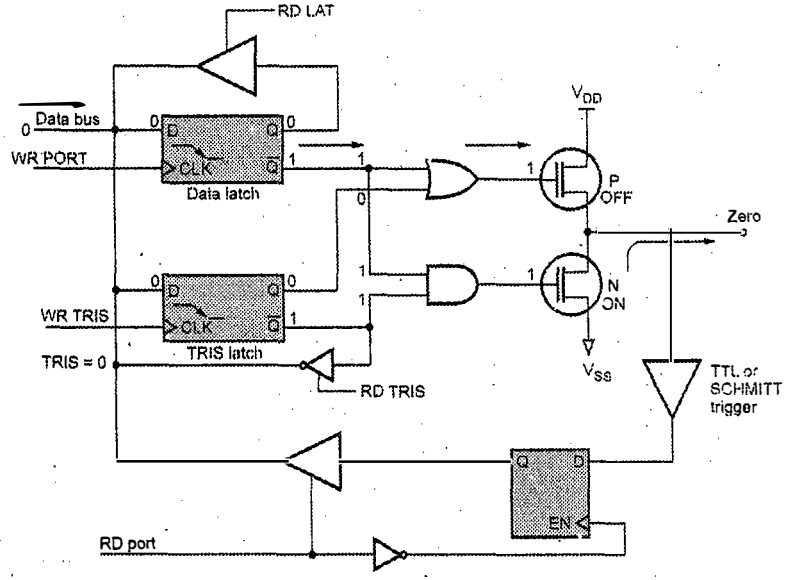


Fig. Q.2.1 Output '0' to pin in the PIC [writing]

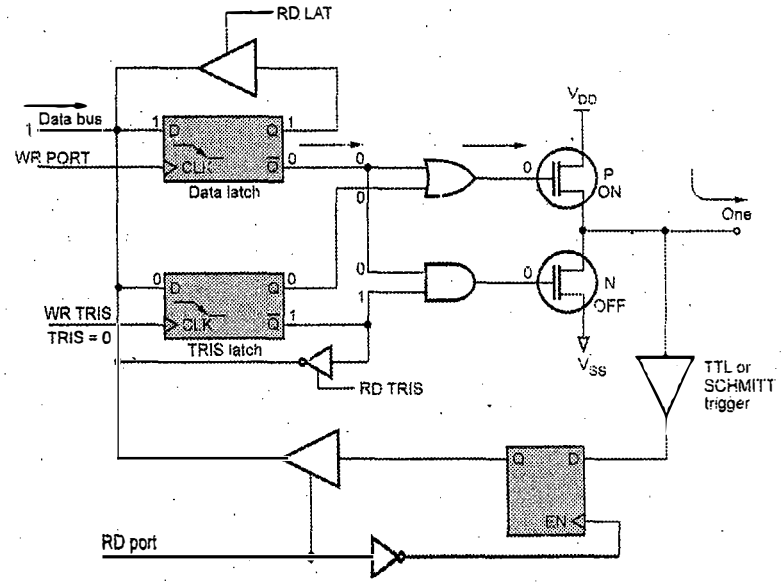


Fig. Q.2.2 Output '1' to pin in the PIC [writing]

Q.3 Draw and explain the port structure of PIC 18FXXXX for reading 0, 1 from port pins

Ans. : Reading from port pin : The PIC18 family has five ports Port A (6), B (8), C (8), D (8), E (3) with 33 I/O lines. These lines can be used for data transfer between working register 'W' and peripherals and vice a versa. Each port can be configured as input or output. Each port has three registers for its operation :

- **TRISx register (Data direction register) :** For most ports, the I/O pin's direction (input or output) is controlled by the data direction register **TRISx** (x = A,B,C,D,E): a '1' in the TRIS bit corresponds to that pin being an input.
- **PORT register :** The **PORTx** register is the latch for the data to be output. Reading PORTx register read the status of the pins, whereas writing to it will write to the port latch.
- **LAT register (output latch) :** The data latch register is useful for read-modify-write operations on the value that the I/O pins are driving.
- **Upon reset all ports are configured as input-TRISx register has 0FFh.**
- Each port in the PIC has three internal D flip-flops (latches)
  - Data latch to hold the output data
  - TRIS latch to setup data direction
  - Input latch for input data
- The configuration of port for reading 1 or 0 from port pin is shown in Fig. Q.3.1 and Q.3.2.

Reading 0 from port pin

TRISx = 0 x FF port is configured as input

Data latch output : Data bus = 0, Write port pin used to clock the data, Q = X, $\overline{Q}$ = X, OR gate = 1, P Gate = OFF.

TRIS latch output : Data bus = 0, Write TRIS pin used to clock the data, Q = 1, $\overline{Q}$ = 0, AND gate = 0, N Gate = OFF.

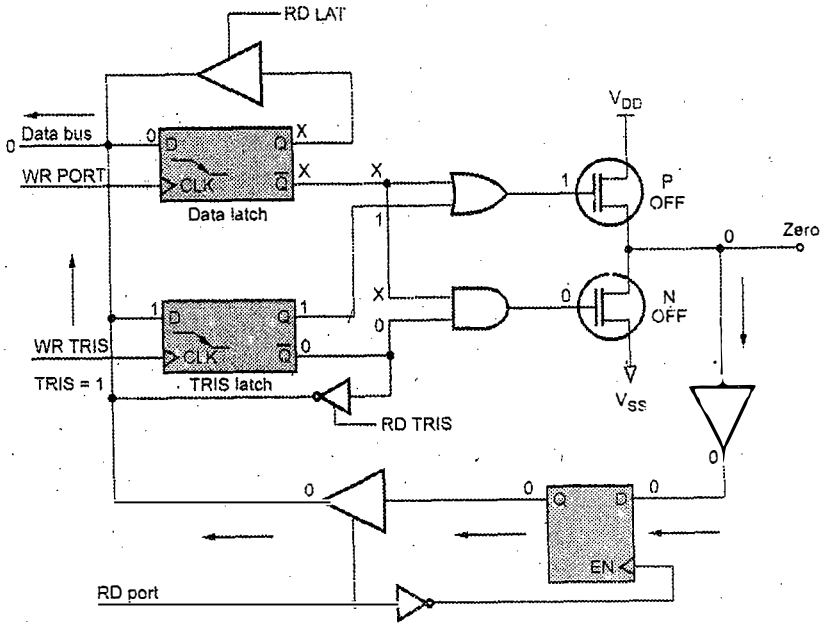


Fig. Q.3.1 Input '0' to pin in the PIC [reading]

Reading 1 from Port pin

TRISx=0xFF port is configured as input.

Data latch output : Data bus = 1, Write port pin used to clock the data, Q = X, $\overline{Q}$ = X, OR gate = 1, P Gate = OFF.

TRIS latch output : Data bus = 1, Write TRIS pin used to clock the data, Q = 1, $\overline{Q}$ = 0, AND gate = 0, N Gate = OFF,

Both the gates are off, hence data available on Port line (0, 1) is passed through the schmitt trigger and input latch.

The input buffer is enabled by R/$\overline{W}$ = 1, pin and data is placed on the data line.

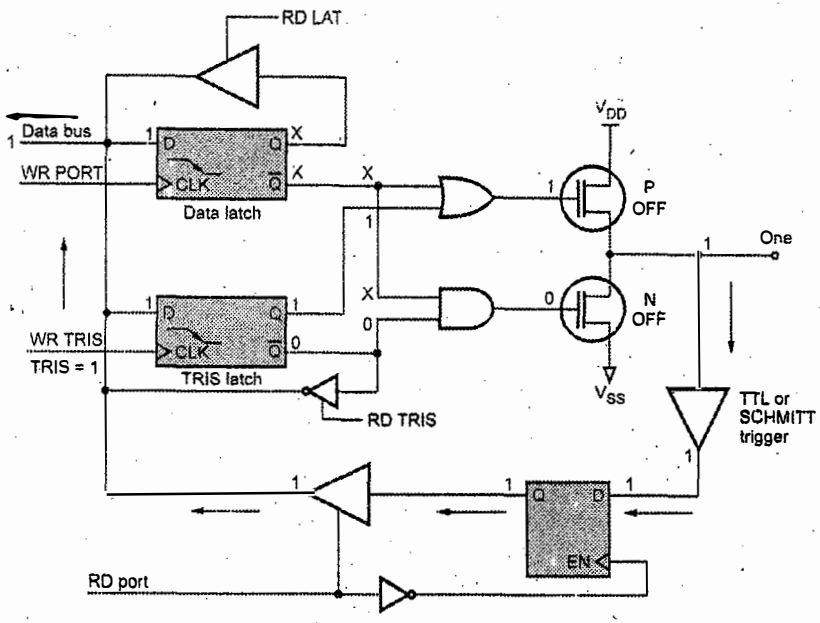


Fig. Q.3.2 Input '1' to pin in the PIC [reading]

Port Programming : I/O Port Programming in PIC

In 'C' program ports are configured as I/P and O/P as :

Direction	TRISB = 0 × 00 - output
	TRISB = 0 × FF - input
	TRISBbits.RB0 = 1;
Latch	LATB = 0;
	LATBbits.LATB0 = 0;
Port	PORTx = 0
	PORTBbits.RB0;

Q.4 Draw an interfacing diagram of LED with PIC18F and write an embedded C program for flashing of LEDs.

☞ [SPPU : May-22, Marks 9, Dec-18, Marks 8]

Ans. : Interfacing of LED

The interfacing diagram for LED to port C is shown in Fig. Q.4.1.

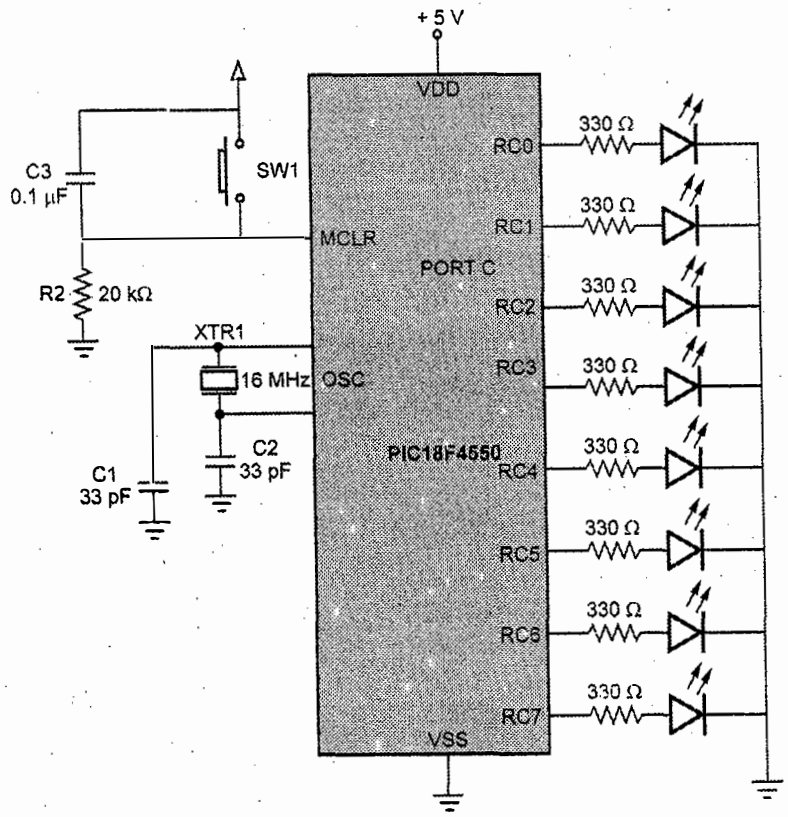


Fig. Q.4.1 LED Interfacing

Program :

```
#include <P18F458.h>
void MSDelay (unsigned int);
Void main(void)
{
    TRISB=0;
    While(1)
        {
```

```
            PORTB=0x55;
            MSDelay(250);
            PORTB=0xAA;
            MSDelay(250);
        }
}
void MSDelay(unsigned int itime)
{
unsigned int i; unsigned char j;
for (i=0; i<itime;i++)
for (j=0;j<100;j++);
}
```

Q.5 Write an embedded C program to read status of SW connected at RD0 and RD1. If RD0 = 0 then S1 = 0 and RD1 = 0 the S2 = 1 then : (a) If S1 = 0 then Relay on buzzer off and LED will glow LSB to MSB continuously. (b) If S2= 0 then Relay off buzzer on and LED will glow MSB to LSB.

Ans. : Interfacing Diagram : The interfacing diagram for Switch, LED, Relay and Buzzer is shown in Fig. Q.5.1.

Program :

```
#include<P18F4550.h>
void delay(void);
void main()
{
    unsigned char i, S1,S2;
    TRISB = 0x00;    //LED pins as output
    LATB = 0x00;
    TRISDbits.TRISD0 = 1;   //set RD0 as input
    TRISDbits.TRISD1 = 1;   //set RD1 as input
    TRISDbits.TRISD2 = 0;   //set buzzer pin RD2 as output
    TRISAbits.TRISA4 = 0;   //set relay pin RA4 as output
     while(1)
    {
        LATDbits.LD0 = 1;
        LATDbits.LD1 = 1;
        if(PORTDbits.RD0 == 0)
```

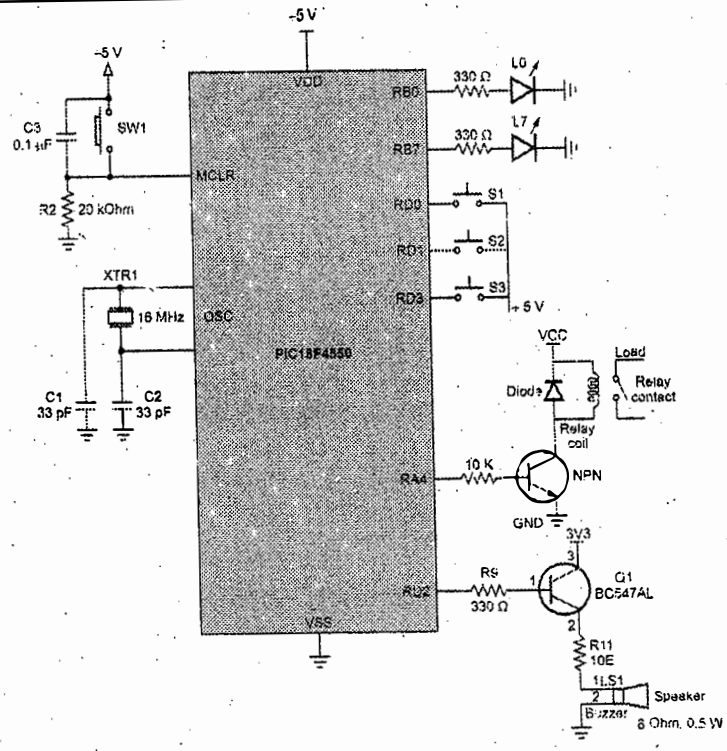



Fig. Q.5.1 Interfacing of switch, LED, relay and buzzer

```
S1 =0;
if(PORTDbits.RD1 == 0)
S2 =1;
if(S1 == 0)
{                          // loop for relay on , Buzzer OFF, LED  R-L
    LATAbits.LATA4 = 1;
    LATDbits.LATD2 = 1;
    for(i=0;i<8;i++)
{
    LATB = 0X01<<i;
    delay();
    LATB = 0x00;
    delay();
}
```

```
        }
        if(S2 == 0)
        {                        // loop for relay OFF , Buzzer ON, LED L-R
            LATAbits.LATA4 = 0;
            LATDbits.LATD2 = 0;
            for(i=0;i<8;i++)
            {
            LATB = 0X80>>i;
            delay();
            LATB = 0x00;
            delay();
            }
        }
    }
}
void delay()
{
unsigned int j,k;
for(j=0;j<=10;j++)
for(k=0;k<=3000;k++);
}
```

Q.6 Draw an interfacing diagram of LED with PIC18F and write program embedded C program for ring counter.

[SPPU : Dec-18, Marks 8]

Ans. : **Interfacing of LED :**

For interfacing refer Fig. Q.5.1.

Program :

```
#include<P18F4550.h>
void delay(void);
void main()
{
    unsigned char i,
    TRISB = 0x00;    //LED pins as output
    LATB = 0x00;

    while(1)
    {
```



```
        for(i=0;i<8;i++)
        {
            LATB = 0X01<<i;
            delay();
            LATB = 0x00;
            delay();
        }
    }

}
void delay()
{
unsigned int j,k;
for(j=0;j<=10;j++)
for(k=0;k<=3000;k++);
}
```

Q.7 Draw a neat interfacing diagram to display 'SPPU' on 4th position in line one and 'UNIVERSITY' at 5th position on second line, write an embedded C program.

[SPPU : May-22, marks 9,May-17, Marks 8]

Note : Similar Question is asked every Semester of the year

Ans. : **LCD Interfacing 8 bit mode :** The interfacing diagram of 16 × 2 Character LCD module with PIC18Fxx microcontroller is shown in Fig. Q.7.1.

Program :

```
#include <p18f4550.h>
#define en LATCbits.LC1
#define rs LATCbits.LC0
#define LCDPORT LATB
const unsigned char LCD_Data1[]={" SPPU"};
const unsigned char LCD_Data2[]={"UNIVERSITY"};
void lcdcmd(unsigned char value);
void lcddata(unsigned char value);
void MSdelay(unsigned int itime);
void main()
```

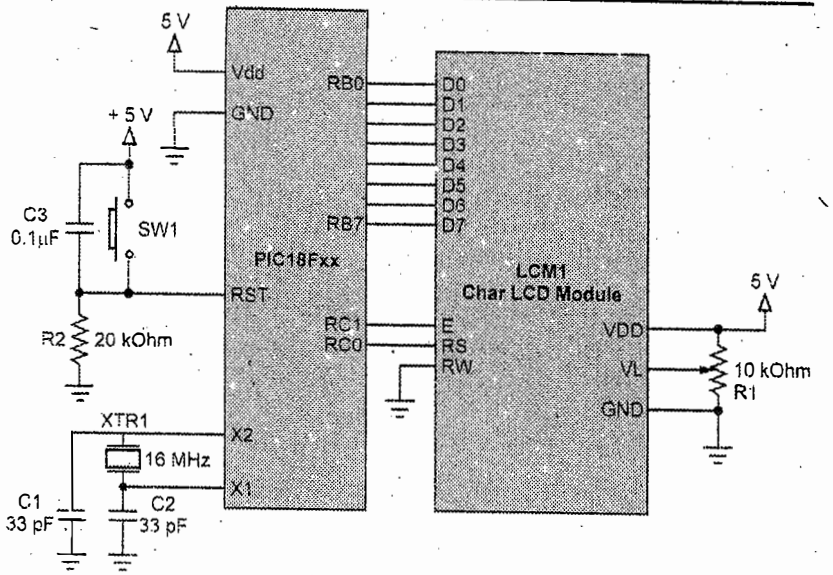


Fig. Q.7.1 LCD interfacing 8 bit mode

```
{
  unsigned char z=0;
   TRISB=0;
   TRISC=0;
  en=0;
  MSdelay(50);
  lcdcmd(0x38); //2line and 5X7 matrix
  MSdelay(50);
  lcdcmd(0x0e); //Display on, Cursor blinking
  MSdelay(50);
  lcdcmd(0x01); //Clear display screen
  MSdelay(50);
  lcdcmd(0x06); //Shiftcursor to right
  MSdelay(50);
```



```
  while(1)
   {
     lcdcmd(0x84);
     MSdelay(50);
    for (z=0;z<4;z++)
     {
       lcddata(LCD_Data1[z]);
       MSdelay(50);
     }
     lcdcmd(0xC5);
     MSdelay(50);
     for (z=0;z<10;z++)
     {
       lcddata(LCD_Data2[z]);
       MSdelay(50);
     }
   }
}
void lcdcmd(unsigned char value)
 {                                  // Routine for Command word
   LCDPORT = value;
   rs = 0;
   en = 1;
   MSdelay(1);
   en = 0;
   return;
 }
 void lcddata(unsigned char value)
 {                                  // Routine for Command word
   LCDPORT = value;
   rs = 1;
   en = 1;
   MSdelay(1);
   en = 0;
   return;
 }
```

```
void MSdelay(unsigned int itime)
 {
 unsigned int i,j;
 for(i=0;i<itime;i++)
 for(j=0;j<1200;j++);
}
```

Note : According to problem statement, change the interfacing diagram and do changes in the program.

Q.8 Write an embedded C program for interfacing of 4 × 4 matrix keyboard and display the number of key closed on LCD.

[SPPU : May-22, Marks 9, May-18, Marks 8, Dec.-17, Marks 8]

Ans. : Keyboard interfacing : The interfacing diagram of 4×4 matrix keyboard is shown in Fig. Q.8.1. The microcontroller accesses both rows and columns through the port D.

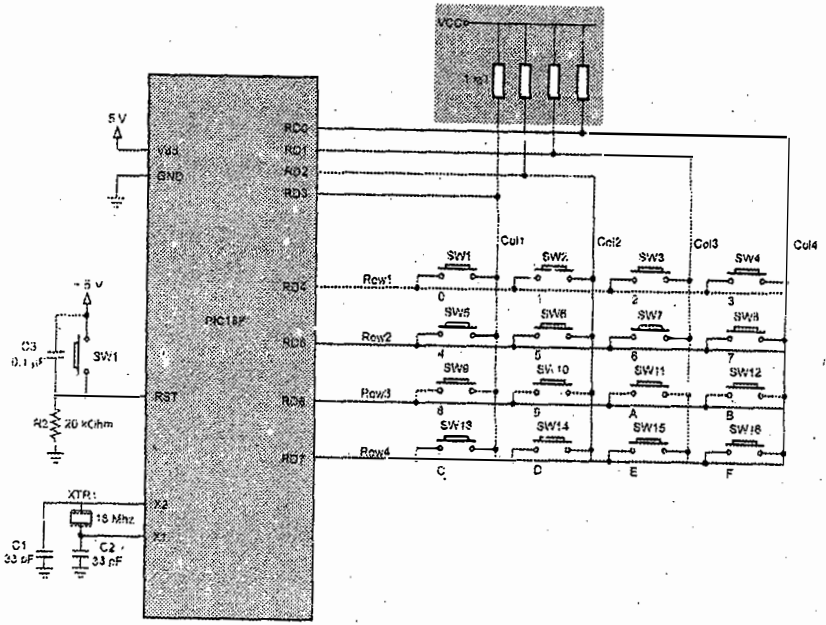


Fig. Q.8.1 Keyboard interfacing

Program :

```
#include <p18f4550.h>
#include "LCD_SIT.h"
const unsigned char LCD_Data1[]={"KEY="};
const unsigned char KeyLookupTbl[]= {'0','4','8','C',
                                     '1','5','9','D',
                                     '2','6','A','E',
                                     '3','7','B','F'};
unsigned char get_keyval(unsigned char col, unsigned char row);
void main(void)
{
    unsigned char z=0,row,val;
    TRISD = 0xF0;  //rows as inputs and cols as output
    LATD = 0xFF;
    Init_LCD();
    lcdcmd(0x80);
    MSdelay(50);
    for (z=0;z<4;z++)
    {
      lcddata(LCD_Data1[z]);
      MSdelay(50);
    }
      while(1)
    {
      LATD = 0xF0;
       MSdelay(5);
      row = PORTD;
       row >>=4;
       if((row & 0x0F)!= 0x0F)
       {
           LATDbits.LATD0 = 0;
          MSdelay(5);
           row = PORTD;
           row >>=4;
          if((row & 0x0F)!= 0x0F)
           {
               val = get_keyval(0,row);
               LATDbits.LD0 = 1;
```

```
            }
            LATDbits.LATD1 = 0;
            MSdelay(5);
            row = PORTD;
            row >>=4;
            if((row & 0x0F)!= 0x0F)
            {
                val = get_keyval(4,row);
                LATDbits.LD1 = 1;
            }
            LATDbits.LATD2 = 0;
            MSdelay(5);
            row = PORTD;
            row >>=4;
            if((row & 0x0F)!= 0x0F)
            {
                val = get_keyval(8,row);
                LATDbits.LD2 = 1;
            }
            LATDbits.LATD3 = 0;
            MSdelay(5);
            row = PORTD;
            row >>=4;
            if((row & 0x0F)!= 0x0F)
            {
                val = get_keyval(12,row);
                LATDbits.LD3 = 1;
            }
            MSdelay(10);
            lcdcmd(0x84);
            MSdelay(10);
            lcddata(val);
            MSdelay(10);
        }
    }
}
/*finds the value of the key*/
unsigned char get_keyval(unsigned char col, unsigned char row)
{
```



```
    unsigned int i=0;
    for(i=0;i<4;i++)
    {
        if((row & 0x01)!= 0x01)
        {
            return(KeyLookupTbl[(col)+i]);
        }
        else
        row >>=1;
    }
}
```

Q.9 What is the use of motion sensor ? Explain with classifications.

Ans. : Motion sensors :

- Motion sensors are commonly used in security systems. They work based on a wide variety of principles and are used in a wide variety of applications.
- Typical usage could be in the exterior doorways or windows of a building for monitoring the area around the building. Upon detecting motion, they generate an electrical signal based on which some actions are taken. Some operate in much the same way as a military radar scanner, while others work based on vibration, infrared radiation and, even sound.
- All of these different types of sensors have different strengths and weaknesses, which are important to take into account when making a decision to choose a particle motion detection sensor.

Classifications of Motion Sensors

Active detectors :

- Active detectors are also known as radar - based motion sensors. The active detector sensors emit the radio waves / microwaves across a room or other place, which strike on nearby objects and reflect it back to the sensor detector.

- When an object moves in motion sensor controlled area at this time, the sensor looks for a doppler (frequency) shift in the wave when it returns to the sensor detector, which would indicate that the wave has hit a moving object.
- Active motion sensors are not best suitable for outdoor lighting or similar applications as a movement of random objects such as windblown things, smaller animals and even larger insects can be detected by the active sensor and lightning will be triggered.

Passive detectors :

- Passive motion sensors are opposite to active sensors, they do not send out anything, but it simply detects the infrared energy. Infrared (heat) energy levels are sensed by passive detectors. Passive sensors scan the room or area, it is installed for infrared heat that is radiated from living beings.
- These sensors would not be effective if they could get activated by a small animal or insect that moves in the detection range, however, most passive sensors can be adjusted to pick up the motion of an object with a certain level of emitted heat, for example adjusting the sensor to pick up movement only by humans.

Combined or Hybrid detectors :

- Combined or hybrid technology motion sensor is a combination of both active and passive sensors. It activates light or alarm only in such a case when motion is detected by both active and passive sensors. Combined sensors are useful for alarm systems to reduce the possibility of false alarm triggers.
- However, this technology also has its disadvantages. It cannot provide the same level of safety as separate PIR and microwave sensors because the alarm is triggered only when motion is detected by both sensors.

- However, this technology also has its disadvantages. It cannot provide the same level of safety as separate PIR and microwave sensors because the alarm is triggered only when motion is detected by both sensors.
- The motion sensors come in different shapes and sizes. Here we are explaining below a couple of examples

Passive Infrared Detectors (PIR) :

- These are one of the widely used sensors nowadays and can be found in many home security systems. Passive infrared detectors are looking the changes of infrared energy level that caused by movement of objects (human, pets... etc.).
- PIR motion detector is very easily obstructed by the variability of heat sources and sunlight, so PIR motion detector is more suitable for the indoor movement detection within the closed environment.

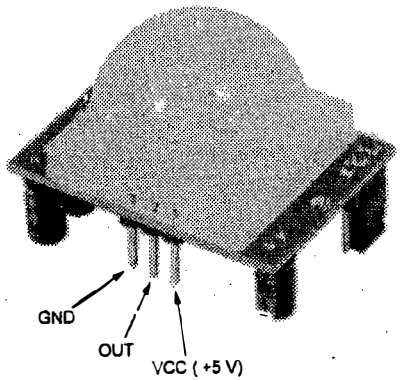


Fig. Q.9.1 : Passive Infrared Detectors (PIR)

- The top view of passive infrared detector is shown in Fig. Q.9.1

Active Infrared Detectors :

- Active infrared detectors use a dual beam transmission as structure, one side of a transmitter for emitting infrared ray and the other side with a receiver for receiving the IR, it is suitable for the outdoor point to point interruption detection.

- Active infra-red beam motion sensors are mainly installed outside, due to it adopts transmitter and receiver theory for detection. It is important that the beam must go through the detection area and reach the receiver.

Ultrasonic Detectors :

- These motion sensors are available in both active and passive types. In theory, an ultrasonic detector sends out high-frequency sound waves that are reflected back to the sensor. If any interruption occurs in the sound waves, the active ultrasonic sensor may sound the alarm. The mini ultrasonic motion detector is shown in Fig. Q.9.2.

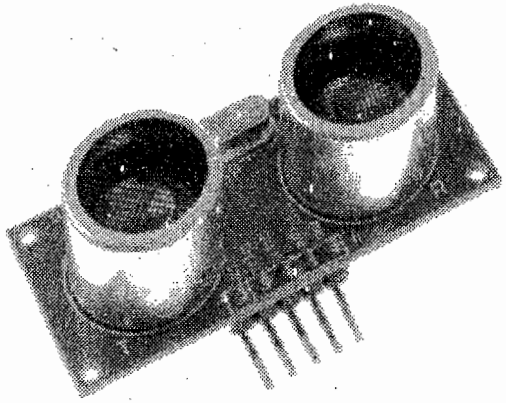


Fig. Q.9.2 Mini ultrasonic motion detector

Q.10 Draw an Interfacing diagram of PIR sensor with PIC 18F4550 and write the embedded C program to detect the motion.

Ans. : **Interfacing of PIR Sensor**

- The PIC 18F4550 considers any voltage between 2 and 5 V at its port pin as HIGH and any voltage between 0 to 0.8 V as LOW.

- Since the output of the PIR sensor module has only two stages (HIGH (3.3 V) and LOW (0 V)), it can be directly interfaced to the PIC 18F4550 microcontroller.
- The circuit diagram for interfacing PIR sensor to PIC 18F4550 microcontroller is shown in Fig. Q.10.1.

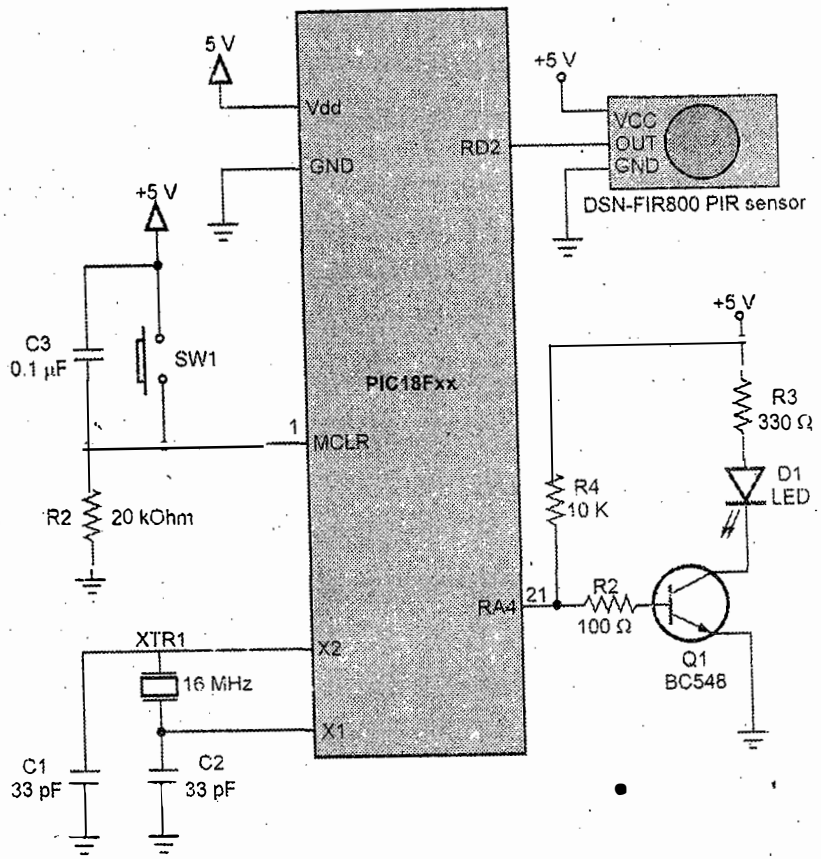


Fig. Q.10.1 Interfacing of ultrasonic motion detector

- The circuit shown in Fig. Q.10.1 will read the status of the output of the PIR sensor and switch ON the LED when there is a motion

detected and switch OFF the LED when there is no motion detected.

- Output pin of the PIR sensor is connected to port RD2 pin of the PIC 18F4550. Resistor R1, capacitor C1 and push button switch S1 forms the reset circuit. Capacitors C3, C4 and crystal X1 are associated with the oscillator circuit. C2 is just a decoupling capacitor.
- LED is connected through port 2.0 of the microcontroller. Transistor Q1 is used for switching the LED. R2 limits the base current of the transistor and R3 limits the current through the LED.

Program

```
#include <P18F4550.h>
    #define _XTAL_FREQ 20000000 //Specify the XTAL crystal FREQ
    #define PIR RD2
    #define LED RA4
    void T0Delay(void);
    void main(void)
    {
        TRISA=0X00;
        TRISD=0XFF
        TRISA=0X00;             // Make all output of RA4 low
        while(1)                // get into infinite loop
        {
            If (PIR==1)
            {
                LED=1;
                    delay ();   // wait for some time
                }
                else
            {
                LED=0;
                    delay();        // wait for some time
            }
```



```
            }
        }
    void T0Delay ( )
{
    T0CON=0x08;                     // Timer0, 16 bit, no prescaler
    TMR0H=0x9E;                     // load Higher byte in TMR0H
    TMR0L= 0x58;                    // Load Lower byte to TMR0L
    T0CONbits.TMR0ON=1;             // start the timer for up count
    while(INTCONbits.TMR0IF==0);    // Check for overflow
    T0CONbits.TMR0ON=0;             // Turn off timer
    INTCONbits.TMR0IF==0;           // clear the Timre0 flag
}
```

Q.11 State features of MQ-2 gas sensors. Draw an interfacing diagram and embedded C code

Ans. : Features of MQ-2 Gas Sensor :

- Operating voltage is +5 V
- Can be used to measure or detect LPG, Alcohol, Propane, Hydrogen, CO and even methane
- Analog output voltage : 0 V to 5 V
- Digital output voltage : 0 V or 5 V (TTL Logic)
- Preheat duration 20 seconds
- Can be used as a digital or analog sensor
- The sensitivity of digital pin can be varied using the potentiometer

Program

```
#include<P18F4550.h>
#define _XTAL_FREQ 20000000 //Specify the XTAL crystal FREQ
        #define GAS RD2
        #define LED RA4
        void delay (void);
        void main (void)
```

```
{
    TRISA=0X00;
    TRISD=0XFF
    TRISA=0X00;     // Make all output of RA4 low
    while(1)        // get into infinite loop
```

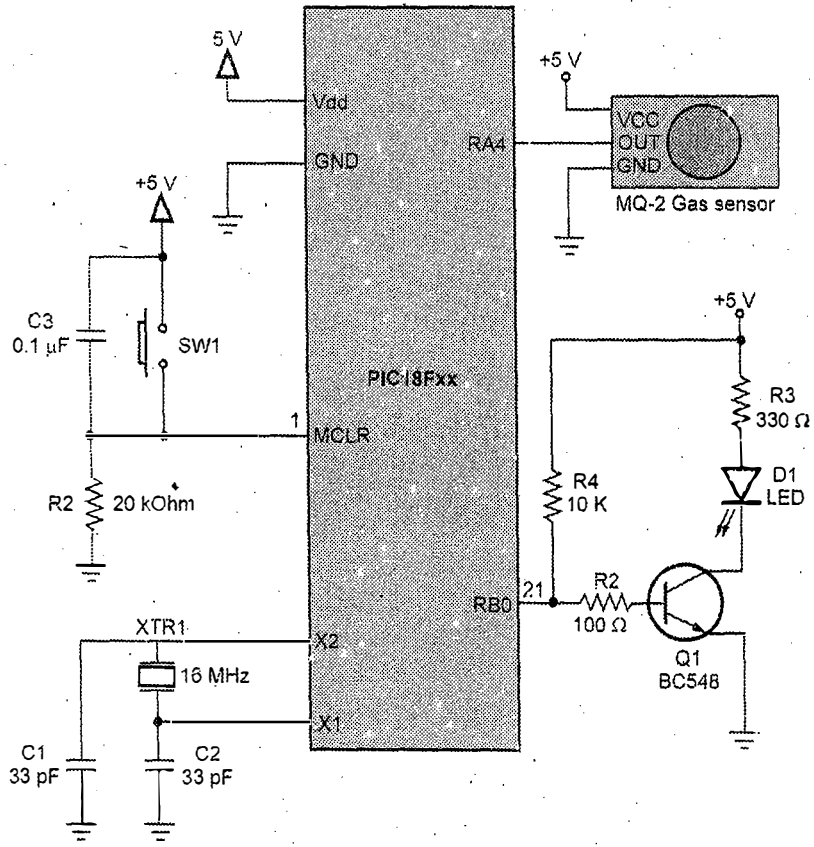


Fig. Q.11.1 Interfacing of MQ-2 Gas Sensor

```
    {
        If (GAS==1)
        {
        LED=1;
```



```
            delay ();   // wait for some time
        }
            else
            {
                LED=0;
                delay();   // wait for some time
            }

            }
    }
    void delay()
    {
        unsigned int j,k;
        for(j=0;j<=10;j++)
        for(k=0;k<=3000;k++);
    }
```

Q.12 Explain use of IR sensor with various applications.

Ans. : IR sensors :

- IR sensor is an electronic device that emits the light in order to sense some object of the surroundings. An IR sensor can measure the heat of an object as well as detects the motion. Usually, in the infrared spectrum, all the objects radiate some form of thermal radiation. These types of radiations are invisible to our eyes, but infrared sensor can detect these radiations.
- The emitter is simply an IR LED (Light Emitting Diode) and the detector is simply an IR photodiode . Photodiode is

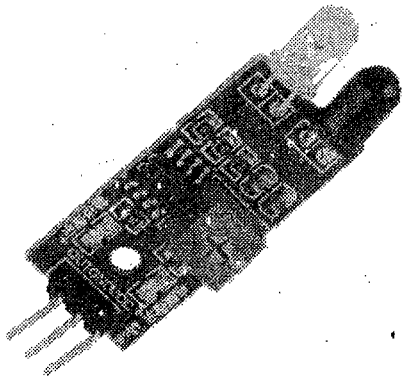


Fig. Q.12.1 IR sensor

sensitive to IR light of the same wavelength which is emitted by the IR LED. When IR light falls on the photodiode, the resistances and the output voltages will change in proportion to the magnitude of the IR light received.

- There are five basic elements used in a typical infrared detection system : An infrared source, a transmission medium, optical component, infrared detectors or receivers and signal processing. Infrared lasers and Infrared LED's of specific wavelength used as infrared sources.
- The three main types of media used for infrared transmission are vacuum, atmosphere and optical fibers. Optical components are used to focus the infrared radiation or to limit the spectral response.

Applications :

Night Vision Devices, Radiation Thermometers, Infra-red tracing, IT image Devising. Other key application areas that use infrared sensors include :

- Climatology
- Photobiomodulation
- Gas detectors
- Moisture analyzers
- Petroleum exploration
- Gas analyzers
- Meteorology
- Flame monitors
- Water analysis
- Anesthesiology testing
- Rail safety

Q.13 Design a PIC 18 FXXXX based to test the LED, Buzzer and relay connected to ports with control using keys.

[SPPU : May-22, Marks 9]

Ans. : Design of PIC test Board : Fig. Q.13.2 shows the design of PIC test board for verifying the LED connected to port B, with switches and buzzer to port D, with relay interface for Port A.

Step 1 : Design of power supply :

The microcontroller selected is PIC 18F4550 which works on the frequency of oscillator ranging from 0 to 20 MHz and requires power supply of ± 5 or ±12 V. A simple circuit design for +5 V is shown in Fig. Q.13.1

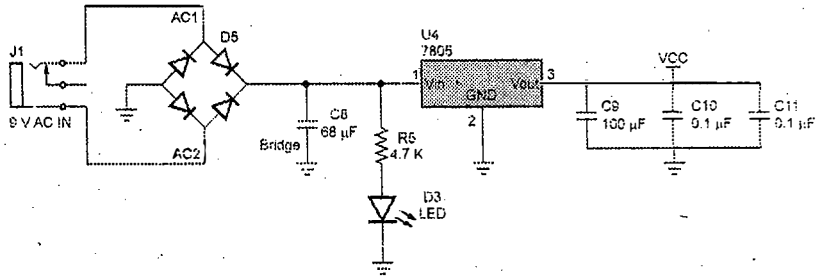


Fig. Q.13.1 Sample for power supply design

Step 2 : Design of clock circuit :

The Quartz crystal is connected to OSC1 and OSC2 pin in order to synchronize the operation of all components connected with internal and external means. The values for C1 and C2 are selected according to the crystal frequency for stabilizing the oscillator pulses. In general with quartz crystal 22 - 33 µF is preferred.

Step 3 : Design of reset circuit :

The RC high pass filter with C = 0.1µF along with 20 kΩ register is connected to MCLR pin. When high pulse appear on it, resets the contents of inter registers and SFRS to initial value.

Step 4 : Configuration of port :

PIC has 33 I/O lines which can be configured as input and output using TRISX register as

if TRISX = 0 - Ports (A-E) are configured as output ports

if TRISX = 1 - Ports (A-E) are configured as input ports.

Step 5 : Connection diagram

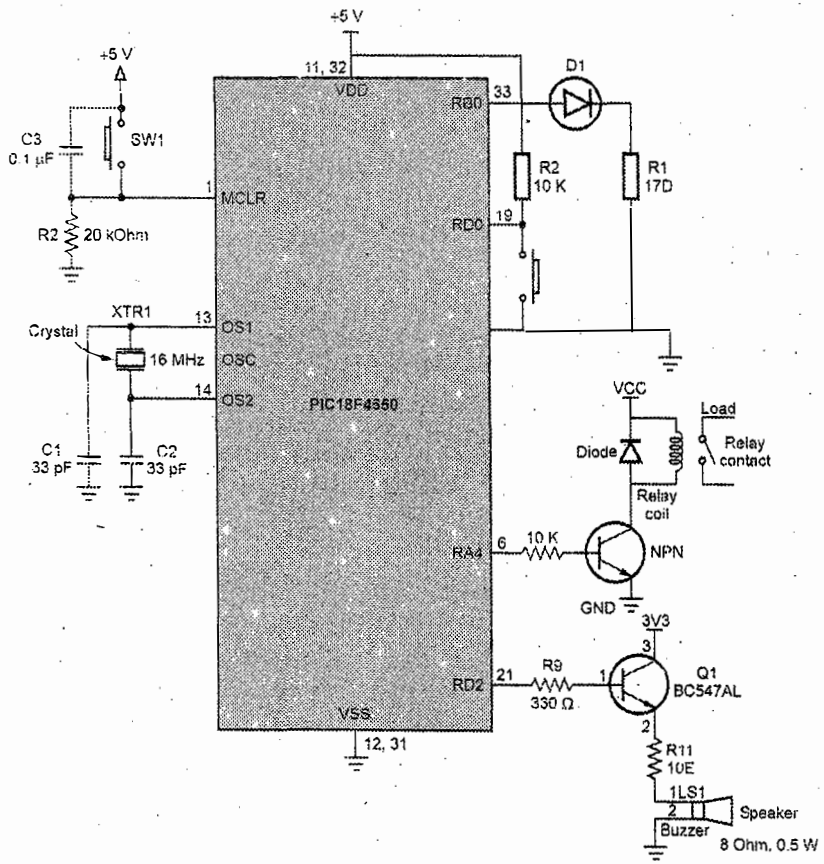


Fig. Q.13.2 Minimum component test system

Step 6 : Algorithm

Step 1 : Initialize TRIS SFR for direction control

Step 2 : Check status of switch

Step 3 : If closed, load bit pattern to glow LED, switch on buzzer and energizes the relay.

Step 4 : Otherwise wait till closer

Step 5 : Transfer content to port register

Step 6 : Wait for some time i.e. delay

Step 7 : Load same or different data sequence of LED glowing

Step 8 : Continue Go to step 2

Step 7 : Program :

```
#include<P18F4550.h>
void delay(void);
void main()
  {
        unsigned char i, key;
        TRISB = 0x00;            //LED pins as output
        LATB = 0x00;
        TRISDbits.TRISD0 = 1;    //set RD0 as input
        TRISDbits.TRISD1 = 1;    //set RD1 as input
        TRISDbits.TRISD2 = 0;    //set buzzer pin RD2 as output
        TRISAbits.TRISA4 = 0;    //set relay pin RA4 as output
    while(1)
    {
      LATDbits.LD0 = 1;
      LATDbits.LD1 = 1;
      if(PORTDbits.RD0 == 0)
      key =0;
      if(PORTDbits.RD1 == 0)
      key =1;
      if(key == 0)
       {
        LATAbits.LATA4 = 1;
        LATDbits.LATD2 = 0;
        for(i=0;i<8;i++)
        {
         LATB = 0X01<<i;
         delay();
         LATB = 0x00;
         delay();
        }
```

```
        }
if(key == 1)
{
  LATAbits.LATA4 = 0;
  LATDbits.LATD2 = 1;
  for(i=0;i<8;i++)
  {
   LATB = 0X80>>i;
  delay();
  LATB = 0x00;
  delay();
  }
}
}
}
void delay()
{
unsigned int j,k;
for(j=0;j<=10;j++)
for(k=0;k<=3000;k++);
}
```

Q.14 Design Home protection system for indicating various parameters like temperature, door open / closed, internal apparatus on, which will give alert by indicator, display and sounding alarm if exceed the set point. Also make provision to store few current records in the serial memory for analysis. ☞ **[SPPU : 10 to 12 Marks]**

Ans. : Design of home protection system

- The microcontroller selected is PIC 18F4550 which works on the frequency of oscillator ranging from 0 to 20 MHz and requires power supply of ± 5 or ±12 V. A sample circuit design for +5 V is shown in Fig. Q.14.1.

- The Quartz crystal is connected to OSC1 and OSC2 pin in order to synchronize the operation of all components connected with internal and external means. The values for C1 and C2 are selected according to the crystal frequency for stabilizing the oscillator pulses. In general with quartz crystal 22 - 33 μF is preferred.

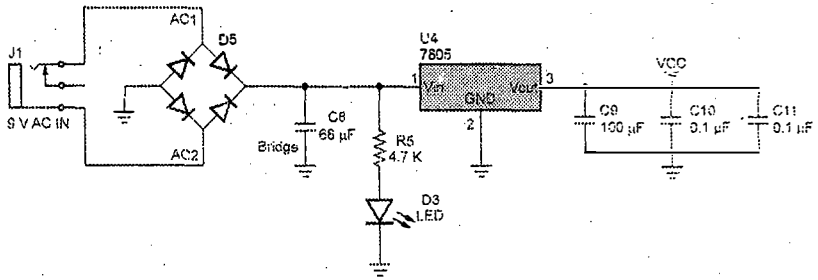


Fig. Q.14.1 Sample for power supply design

- Reset : The RC high pass filter with C = 0.1μF along with 20 kΩ register is connected to MCLR pin. When high pulse appear on it, resets the contents of inter registers and SFRS to initial value.
- PIC has 33 I/O lines which can be configured as input and output using TRISX register.

General block diagram

- The general block diagram of any security system without in built ADC is shown in Fig. Q.14.2 Some of the modern processor like PIC has the in-build ADC and require only signal conditioning circuit. The sample signal conditioning circuit is shown in Fig.Q.14.3 The signal conditioning circuit for any analog signal varies from signal to signal. For any low level signals an Instrumentation amplifier is best choice.

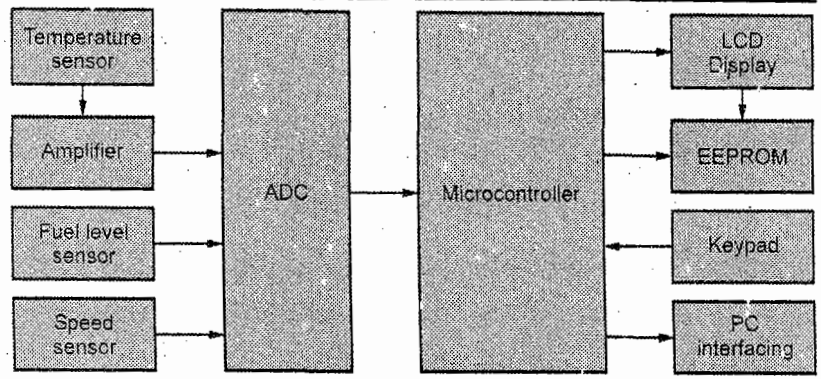


Fig. Q.14.2 The general block diagram

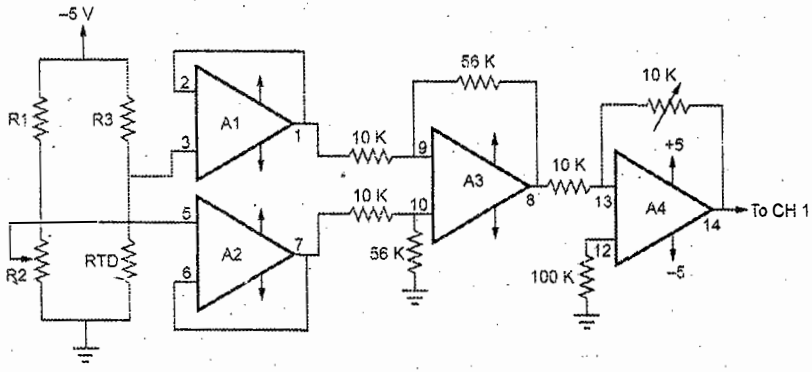


Fig. Q.14.3 Signal conditioning circuit

R1 = R3 = 2.2 K, R2 = 220, R4 = RTD, OPAMP LM324

Algorithm :

1. Initialize the PIC ports, LCD etc.
2. Provide set points for various controlling actions
3. Accept the signals (Analog or digital) on ADC pins.
4. Check the set conditions
5. If not met or exceed get the indication by beep, display on LCD and making devices on and Off

6. Store the current records
7. If everything is set right, continues from step 3.

Flowchart : The general flowchart for the above problem statement is shown in Fig. Q.14.4

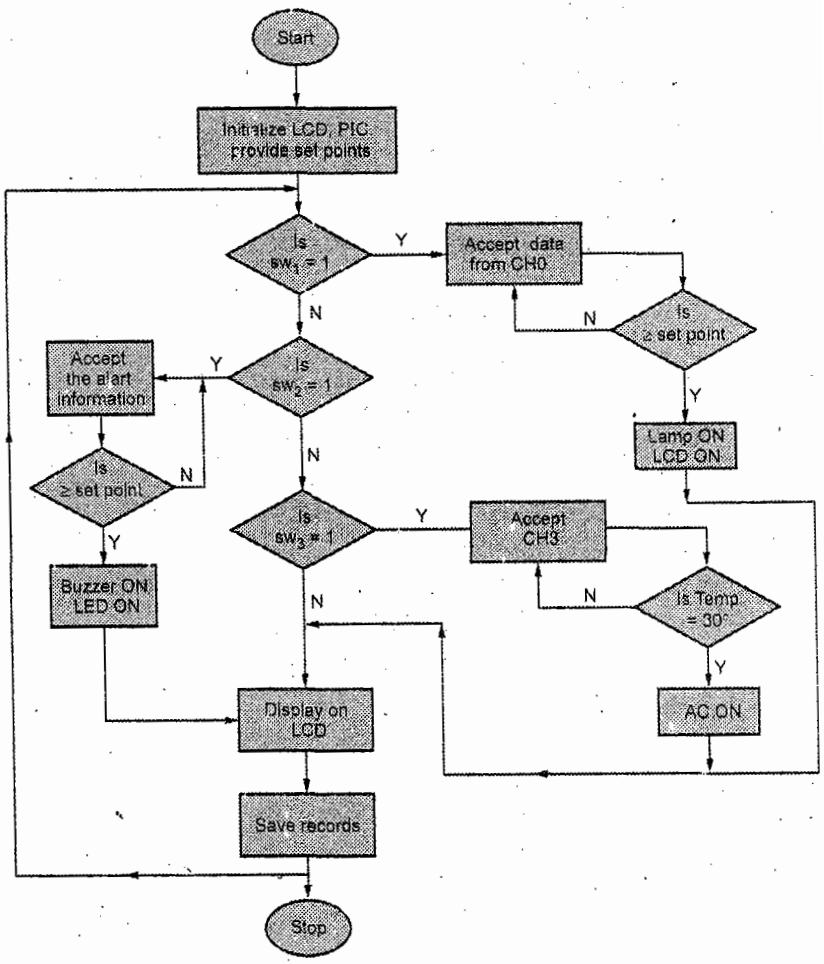


Fig. Q.14.4 Flow chart to access the information

Complete schematic : The complete schematic diagram is shown in Fig Q.14.5

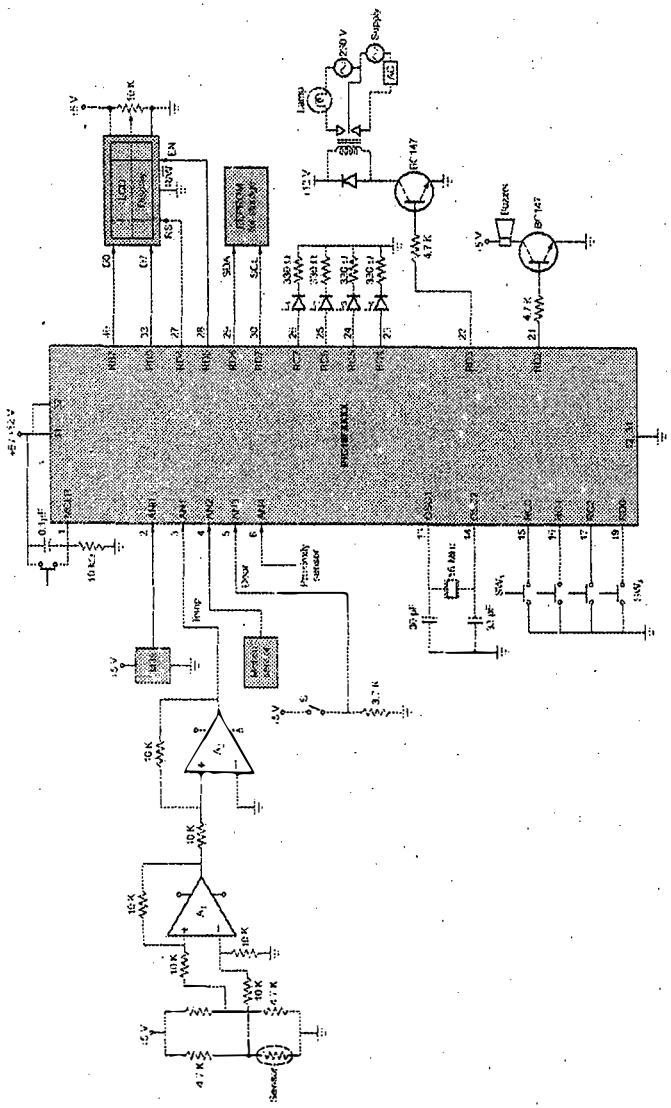


Fig. Q.14.5 Complete schematic for home protection system

Important Points to Remember

1. Before interfacing any input/output device to microcontroller refer to their specifications listed in data sheets by the manufacturers to fulfil the optimum working conditions.
2. List the hardware and software requirements depends on type of device to be interfaced.
3. Study the port structure of microcontroller to avoid damage to its port pins. Refer to the Fan in and out conditions carefully.
4. Pull up resistors are required to avoid damage to port pins.
5. PIC18F4550 has Five port 9A,B,C,D,E - 35 I/O lines) and USB
6. Port B is used for the external interrupts
   Each port has multiplexed for many functions and some pins are exclusively used as digital Input
7. By default on power on reset all ports are configured as input.
8. LEDs are most popularly used indication device. Take its forward voltage into consideration which depends on its colour composition and calculate the value of current limiting resistor using simply Ohm's law.
9. LEDs can be connected in common anode or common cathode configuration.
10. LCD is useful display device to display alphanumeric characters with reduced power consumption.
11. RS- register select R/$\overline{W}$ read/write and EN are the important control signal of LCD along with data lines referred while interfacing to microcontroller.
12. The busy flag DB7 gives updates about current status of LCD.
13. The LCD can be interfaced in two possible ways 4 bit and 8 bit mode. The 4 bit mode saves 4 I/O lines but increases the execution time as nibble by nibble the data is send to LCD.
14. Keys when connected in matrix form reduces number of I/O lines used.

15. Keyboard matrix even saves space and make it compact.
16. Key debounce - a state before key settles can be taken care either in hardware by using SR flip-flops or in software by introducing suitable delay.
17. Key detection can be taken care by using look-up table approach.
18. Motion detectors are type of PIR sensors
19. Gas sensors are used in home protection system
20. While designing any system make sure of VCC, GND, Reset and clock is connected

Reference for LCD Command words

LCD commands framing

BIT	RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Function
	0	0	0	0	0	0	0	0	0	1	Clear LCD and memory, home cursor
	0	0	0	0	0	0	0	0	1	0	Clear and home cursor only
	0	0	0	0	0	0	0	1	I/0	S	Screen action as display character written S = I/0 : Shift screen/cursor I/0 = I/0 : Cursor R/L, screen L/R
	0	0	0	0	0	0	1	D	C	B	D = I/0 : Screen on/off C = I/0 : Cursor on/off B = I/0 : Cursor Blink / Noblink

	0	0	0	0	0	1	S/C	R/L	0	0	S/C = I/0 : Screen / Cursor R/L = I/0 : Shift one space R/L
	0	0	0	0	1	DL	N	F	0	0	DL = I/0 : 8/4 Bits per character N = I/0: 2/1 Rows of characters F = I/0; 5 × 10/5 × 7 Dots / Character
	0	0	0	1	Character address						Write to character RAM address after the commands
	0	0	1	Display data address							Write to display RAM address after the commands
	0	1	BF	Current address							BF = I/0 : Busy / Not busy
	1	0		Character byte							Write byte to last RAM chosen
	1	1		Character byte							Read byte from last RAM chosen

END...

Unit VI

7 Serial Port Programming and Interfacing with PIC18FXXXX

Q.1 State features of RS232 with signal characteristics and frame format [SPPU : Marks 8]

Ans. : RS232 serial port : RS232 is an asynchronous serial communication protocol widely used in computers and digital systems. It is called asynchronous because there is no separate synchronizing clock signal as there are in other serial protocols like SPI and I2C. The protocol is such that it automatically synchronize itself. It has following features

- Developed by Electronic Industry Association (EIA) in 1960 and updated in 1969
- Most widely used serial I/O interface standard asynchronous communication [TxD , RxD and GND]
- Accepted to transfer characters over short distances of 50 feet.
- Low data rates - kbps
- Input and output voltage levels are not TTL compatible.
- Logic 1 : –3 to –25 volt - Negative logic.
- Logic 0 : 3 to 25 volt
- Can operate in a full duplex manner, supporting concurrent data flow in both directions.
- MAX 232 IC is normally termed as line drivers.
- It is designed around transmission of characters (of 7 bits of length). Sends each bit in exactly the same length of time

- PC standard baud rate (see hyper terminal configuration) 150, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 33600, 57600

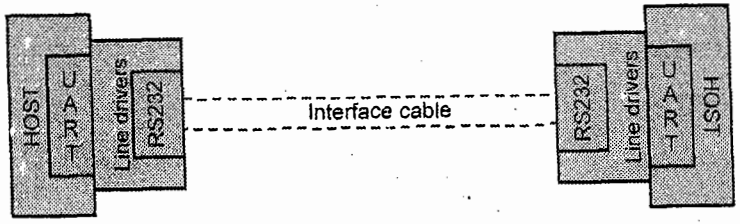


Fig. Q.1.1 Serial communication

- It uses DB9 or DB25 connector to interface the PC serial port with external devices as Data Terminal Equipment (DTE) and Data Communication Equipment (DCE).
- The RS-232 interface works in combination with UART universal asynchronous receiver/transmitter. It is a piece of integrated circuit integrated inside the processor or controller. It takes bytes and transmits the individual bits in a sequential fashion in a frame as shown in Fig. Q.1.1.

Signal characteristics of RS232 : This is the equivalent circuit for an EIA232 signal line and applies to signals originating at either the DTE or DCE side of the connection. "Co" is not specified in the standard, but is assumed to be small and to consist of parasitic elements only. "Ro" and "Vo" are chosen so that the short-circuit current does not exceed 500 mA. The cable length is not specified in the standard; acceptable operation is experienced with cables that are less than 25 feet in length as shown in Fig. Q.1.2.

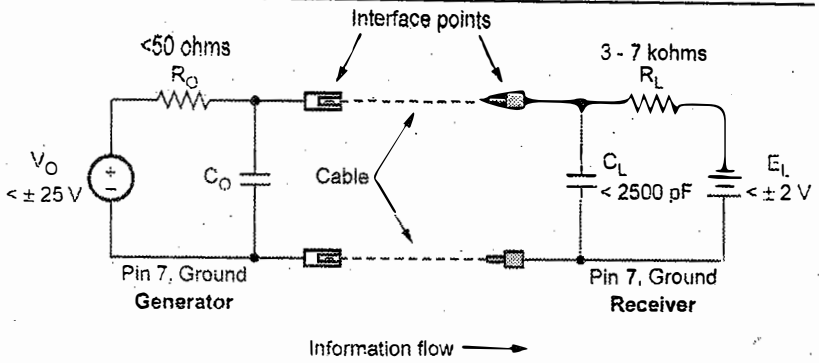


Fig. Q.1.2 Signal characteristics of RS232

The frame format used for communication is shown in Fig. Q.1.3.

The transmission rate of serial devices is called baud. It is the number of changes in the signal per second - Baud rate is the important property of any serial communication.

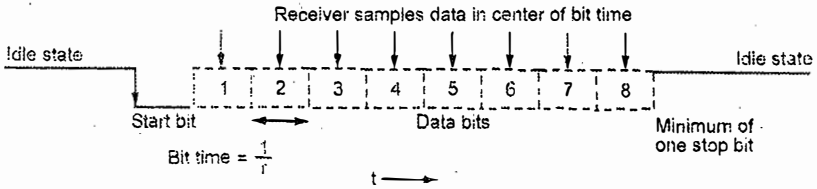


Fig. Q.1.3 Asynchronous frame format

Asynchronous serial transmission is widely used for the character oriented transmission. It has the speed limitations due to the fact that RS-232 is analog, therefore it is slow (in computing terms). The **Computer baud rates :** 110, 300,600, 1200, 2400, 4800, 9600, 19200 etc

Q.2 State features of RS485 with network topology used in communication. **[SPPU : May-19, Marks 8]**

Ans. : RS485 Protocol :

Silent features

1. Uses balanced differential configuration with Multi-drop.
2. Has 32 line drivers and receivers, can extend up to 256.
3. High data speed : 35 Mbit/s up to 10 m and 100 Kbit/s at 1200 m.

- RS-485, is a standard defining the electrical characteristics of drivers and receivers for use in balanced digital multipoint systems.
- The standard is published by the ANSI Telecommunication Industry Association/Electronic Industries Alliance (TIA/EIA).
- Digital communications networks implementing the EIA-485 standard can be used effectively over long distances and in electrically noisy environments.
- Multiple receivers may be connected to such a network in a linear, multi-drop configuration. These characteristics make such networks useful in industrial environments and similar applications.
- EIA-485 enables the configuration of inexpensive local networks and multi-drop communications links. It specifies electrical characteristics of the driver and the receiver. It does not specify or recommend any data protocol.
- It offers high data transmission speeds (35 Mbit/s up to 10 m and 100 kbit/s at 1200 m).
- Since it uses a differential balanced line over twisted pair it can span relatively large distances (up to 4000 feet or just over 1200 meters).

- EIA-485 drivers need to be put in transmit mode explicitly by asserting a signal to the driver.
- The equipment located along a set of EIA-485 wires are interchangeably called nodes, stations and devices. The network topology used by EIA 485 is shown in Fig. Q.2.1.
- The **RS485** network must be designed as one line with multiple drops, not as a star. Although total cable length may be shorter in a star configuration, adequate termination is not possible anymore and signal quality may degrade significantly.

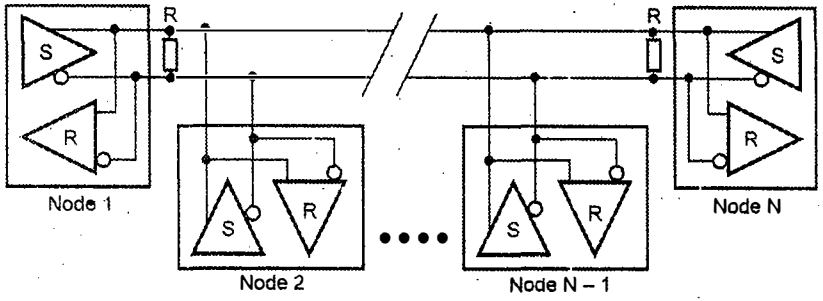


Fig. Q.2.1 Network topology of EIA485

- **RS485** is currently a widely used communication interface in data acquisition and control applications where multiple nodes communicate with each other. The communication media used is twisted pair of wires to reduce the effects of noise and avoid malfunctioning in selection of node as shown in Fig. Q.2.2.

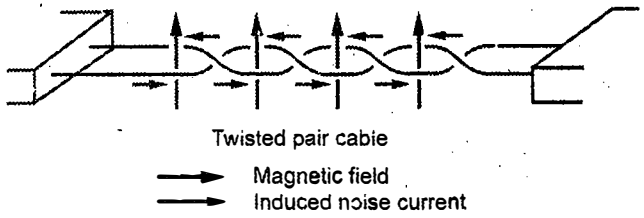


Fig. Q.2.2 Data communication In RS485

- Network topology is probably the reason why **RS485** is now the favourite of the four mentioned interfaces in data acquisition and control applications.
- **RS485** repeaters are also available which make it possible to increase the number of nodes to several thousands, spanning multiple kilometers.
- And that with an interface which does not require intelligent network hardware: the implementation on the software side is not much more difficult than with **RS232**.
- It is the reason why **RS485** is so popular with computers, PLCs, micro controllers and intelligent sensors in scientific and technical applications.

Q.3 Differentiate between RS232 and RS485 serial communication protocols. ☞ [SPPU : May-16, 17, Marks 6]

Ans. : Comparison between RS232 and RS485 : The comparison between RS232 and RS485 is given in Table Q.3.1.

Table Q.3.1 Comparison

	RS-232	RS-485
Cabling	Single ended	Balanced - differential
Number of devices	1 transmit 1 receive	64 transmitters (1/2 load Rx ±) 64 receivers
Communication mode	Full duplex	Half duplex
Maximum distance	50 feet (at 19.2 kbps)	4000 feet (at 100 kbps)
Maximum data rate	19.2 kbps (for 50 feet) 115 kps (for 6 feet)	10 MB/s (for 50 feet)

Signalling	Unbalanced	Balanced
Mark (data 1)	–5 VDC minimum –15 VDC maximum	1.5 VDC minimum (B>A) 5 VDC maximum (B>A)
Space (data 0)	5 VDC minimum 15 VDC maximum	1.5 VDC minimum (A>B) 5 VDC maximum (A>B)
Input level minimum	+/–3 VDC	0.2 VDC difference
Output current	500 mA (Note : driver ICs normally used in PCs are limited to 10 mA)	250 mA

Q.4 Explain in depth use of I2C protocol with features

[SPPU : May-17, Marks 6]

Ans. : Inter Integrated Circuit (I2C) : • I2C is a synchronous serial bus protocol uses two wires, serial data (SDA) and serial clock (SCL) to carry information between the devices connected to the bus.

- Each device is recognized by a unique address and can operate as either a transmitter (Master) or receiver (Slave).
- A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.
- Both SDA and SCL are bi-directional lines and connected to a positive supply voltage via a current-source or pull-up resistor.
- When the bus is free, both lines are HIGH.
- Devices connected to the bus must have an open drain or open collector output for serial clock and data.

- It is typically used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication as shown in Fig. Q.4.1.

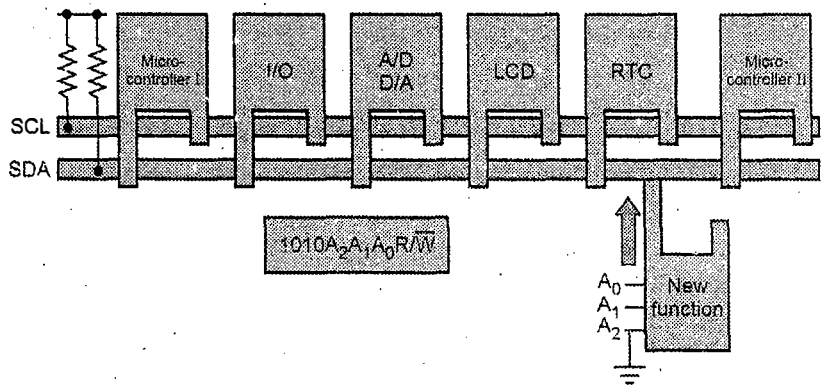


Fig. Q.4.1 Connection to serial devices on I2C bus

Features :

- A 2 wire serial protocol with data and control bus (SDA and SCL)
- Unique start and stop conditions with bi-directional data transfer.
- Acknowledgement after each byte transferred.
- No limit on the number of bytes transferred
- Has clock synchronization.
- Transmission speed : Normal : 100 kHz, Fast mode: 400 kHz and HS-mode: 3400 kHz.
- Maximum bus length of 4 meters.
- Maximum drive capacity of 400 pF.
- Real multi-master capability.
- Compatible with most IC technologies (TTL, CMOS, etc.).
- Has arbitration procedure.
- I2C is a synchronous serial bus protocol

- Two wires, serial data (SDA) and serial clock (SCL), carry information between the devices connected to the bus.
- Each device is recognized by a unique address and can operate as either a transmitter (Master) or receiver (Slave).
- A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave.
- Both SDA and SCL are bi-directional lines,
- Connected to a positive supply voltage via a current-source or pull-up resistor.
- When the bus is free, both lines are HIGH
- Devices connected to the bus must have an open drain or open collector output for serial clock and data

Q.5 Explain Operation of I2C protocol with Start, Stop, data valid condition etc. and device addressing for data transfer

Ans. : I2C operation for Data Transfer :

- Data transfer can be initiated only when the bus is not busy.
- During data transfer, the data line must remain stable whenever the clock line is HIGH.
- Changes in the data line while the clock line is high will be interpreted as control signals.
- Accordingly, the following bus conditions have been defined :
- **Bus not busy :** Both data and clock lines remain HIGH.
- **START data transfer :** A change in the state of the data line, from HIGH to LOW, while the clock is HIGH, defines a START condition

- **STOP data transfer :** A change in the state of the data line, from LOW to HIGH, while the clock line is HIGH, defines the STOP condition.
- **Data valid :** The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.
- The data on the line must be changed during the LOW period of the clock signal. There is one clock pulse per bit of data.
  - Each data transfer is initiated with a START condition and terminated with a STOP condition.
  - The number of data bytes transferred between START and STOP conditions is not limited, and is determined by the master device.
  - The information is transferred byte-wise and each receiver acknowledges with a ninth bit.
  - Within the I2C bus specifications a standard mode (100 kHz clock rate) and a fast mode (400 kHz clock rate) are defined.
- **Acknowledge :** The acknowledge-related clock pulse is generated by the master. The transmitter releases the SDA line (HIGH) during the acknowledge clock pulse. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable LOW during the HIGH period of this clock pulse. Each receiving device, when addressed, is obliged to generate an acknowledgement after the reception of each byte.
- The master device must generate an extra clock pulse which is associated with this acknowledge bit.
- A device that acknowledges must pull down the SDA line during the acknowledge clock pulse in such a way that the SDA line is

stable LOW during the HIGH period of acknowledge related clock pulse.

- N number of data can be transferred between start and stop conditions.
- The detailed sequence of data transfer and initial conditions are shown in Fig. Q.5.1, Q.5.2 and Fig. Q.5.3.

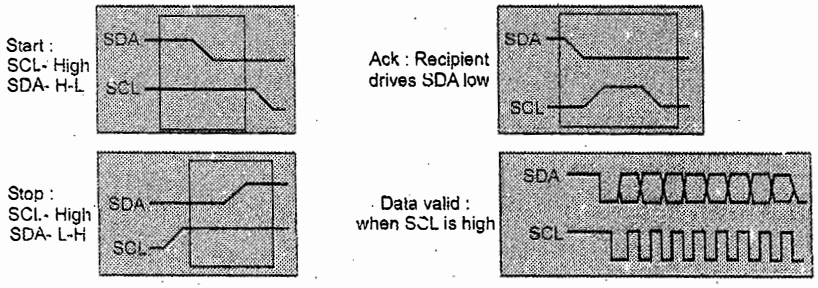


Fig. Q.5.1 Start, Stop, ACK and data valid condition

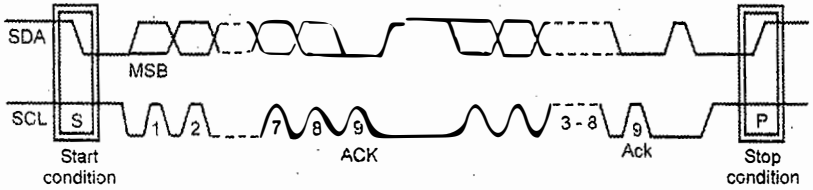


Fig. Q.5.2 Data transfer frame in I2C

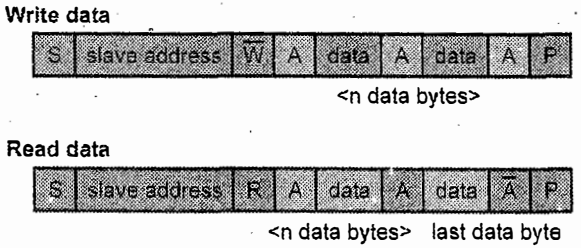


Fig. Q.5.3 Data transfer format

S = Start Condition, R/$\overline{W}$ = Read / Write, A = Acknowledgement, P = Stop

Q.6 Compare SPI, I2C and USART protocols.

[SPPU : Nov.-15, May-16, Marks 8]

Ans. : Comparison between I2C and SPI : The comparison between synchronous and asynchronous protocol is given in Table Q.6.1.

Table Q.6.1 SPI, I2C and USART protocol comparison

Types	Synchronous		Asynchronous
Peripherals	SPI	I2C	UART
Max bit rate	10 Mbits/s	1 Mbits/s	500 kbits/s
Max bus size	Limited No. of Pins	128 Devices	Point to point RS232, 256 devices
No. of pins	3+ n x CS	2	2
Pros	Simple, Low cost, High speed	Allows multiple masters	Longer distance, improved noise immunity
Cons	Single master, short distance	Slowest short distance	Interface with terminals and used in DAS
Typical applications	Direct connections to ASIC and other peripherals on same PCB	Bus connections on same peripherals	Interface with terminals, PCs, modems
Examples	Serial EPROMS 25CXXX series	Serial EPROMS 24CXXX series	RS33, RS485

Q.7 Draw and explain the block diagram of MSSP, SPI mode in detail. **[SPPU : May-16,17, Dec-17, Nov.-15,16, Marks 8, May-22, 9 Marks]**

Ans. : Master Serial Synchronous Port (MSSP) - SPI bus :

- The Master Synchronous Serial Port (MSSP) module is a serial interface useful for communicating with other peripheral or microcontroller devices. These peripheral devices may be serial EEPROMs, shift registers, display drivers, A/D converters, etc.
- The SPI mode allows 8-bits of data to be synchronously transmitted and received, simultaneously.
- To accomplish communication, typically three pins are used :

  Serial Data Out (SDO) - RC5/SDO

  Serial Data In (SDI) - RC4/SDI/SDA

  Serial Clock (SCK) - RC3/SCK/SCL/LVDIN
- Slave Select (SS) - RA5/SS/AN4 (Additionally, a fourth pin may be used when in a Slave mode of operation)
- The MSSP module has four registers for SPI mode operation. These are :

  1. MSSP Control Register1 (SSPCON1) - read and write, used to select the oscillator frequency, enable operation, and check for low and high level of clock.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
bit 7							bit 0

  2. MSSP Status Register (SSPSTAT) - used for SPI and I2C modes with checking of edge for clocking and check for transfer of receive the data.

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	$D/\overline{A}$	P	S	$R/\overline{W}$	UA	BF
bit 7							bit 0

  3. Serial Receive/Transmit Buffer (SSPBUF) - Utilized which data bytes are written to or read from controller.
  4. MSSP Shift Register (SSPSR) - Not directly accessible : SSPSR is the shift register used for shifting data in or out.
- In receive operations, SSPSR and SSPBUF together create a double buffered receiver. When SSPSR receives a complete byte, it is transferred to SSPBUF and the SSPIF interrupt is set.
- During transmission, the SSPBUF is not double buffered. A write to SSPBUF will write to both SSPBUF and SSPSR.
- The block diagram of MSSP - SPI is shown in Fig. Q.7.1. The block diagram is divided into three parts 1. Data bus for communication along with SSPBUF and SSPSR register, 2.Data and clock selection for Master and Slave (DI, DO, and SS pin and 3. Clock for shifting the data in and out.
- The detailed explanation is summarized as

  1. When initializing the SPI, several options need to be specified. This is done by programming the appropriate control bits (SSPCON1<5:0>) and SSPSTAT<7:6>. These control bits allow the following to be specified :
     - Master mode (SCK is the clock output)
     - Slave mode (SCK is the clock input)
     - Clock polarity (IDLE state of SCK)
     - Data input sample phase (middle or end of data output time)
     - Clock edge (output data on rising / falling edge of SCK)
     - Clock rate (Master mode only)
     - Slave select mode (Slave mode only)

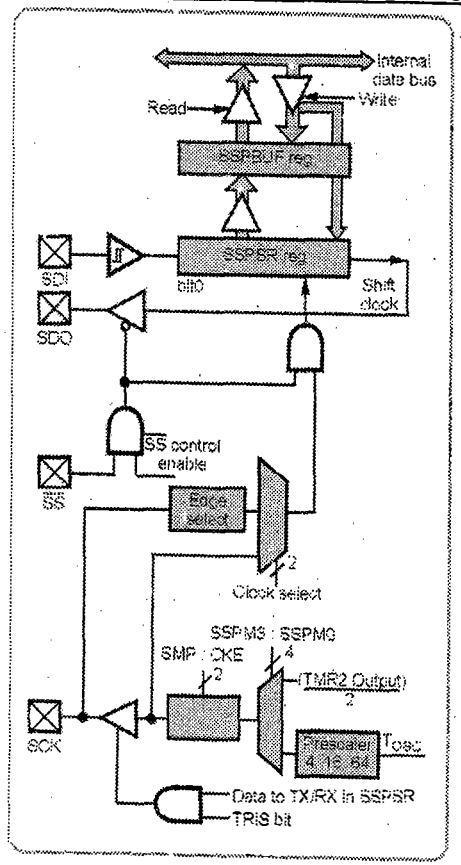


Fig. Q.7.1 MSSP SPI Mode

2. The MSSP consists of a transmit / receive shift register (SSPSR) and a buffer register (SSPBUF). The SSPSR shifts the data in and out of the device, MSB first.
3. The SSPBUF holds the data that was written to the SSPSR, until the received data is ready. Once the 8 bits of data have been received, that byte is moved to the SSPBUF register. Then the buffer full detect bit, BF (SSPSTAT<0>) and the interrupt flag bit, SSPIF are set.

4. This double buffering of the received data (SSPBUF) allows the next byte to start reception before reading the data that was just received.
5. Any write to the SSPBUF register during transmission/reception of data will be ignored, and the write collision detect bit, WCOL (SSPCON1<7>), will be set.
6. User software must clear the WCOL bit so that it can be determined if the following write(s) to the SSPBUF register completed successfully.
7. The SSPSR is not directly readable or writable and can only be accessed by addressing the SSPBUF register.
8. Additionally, the MSSP Status register (SSPSTAT) indicates the various status conditions.

Q.8 Explain function of MSSP, I2C mode with registers.

Ans. : MSSP structure I2C BUS

- The MSSP module in I2C mode fully implements all master and slave functions and provides interrupts on start and stop bits in hardware to determine a free bus (multi-master function).
- The MSSP module implements the standard mode specifications, as well as 7-bit and 10-bit addressing.
- Two pins are used for data transfer :

  Serial clock (SCL) - RB1/AN10/INT1/SCK/SCL

  Serial data (SDA) - RB0/AN12/INT0/FLT0/SDI/SDA
- The user must configure these pins as inputs by setting the associated TRIS bits.
- The I2C protocol supports either a 7-bit addressing mode, or a 10-bit addressing mode, permitting 128 or 1024 physical devices to be on the bus.

- According to the I2C specification, all changes on the SDA line must occur while the SCL line is low. This restriction allows two unique conditions to be detected on the bus; Start and Stop
- A START sequence occurs when the master device pulls the SDA line low while the SCL line is high.
- The I2C protocol also permits a Repeated Start condition (RS), which allows the master device to execute a START sequence without preceding it with a STOP sequence

A. I2C Bus registers

- The MSSP module has six registers for I2C operation these are :
- MSSP Control Register1 (SSPCON1)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
bit 7							bit 0

- MSSP Control Register2 (SSPCON2)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
bit 7							bit 0

- MSSP Status Register (SSPSTAT)

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$ (2,3)	UA	BF
bit 7							bit 0

- Serial Receive/Transmit Buffer (SSPBUF)
- MSSP Shift Register (SSPSR) - Not directly accessible
- MSSP Address Register (SSPADD)
- SSPCON1, SSPCON2 and SSPSTAT are the control and status registers in I2C mode operation. The SSPCON1 and SSPCON2 registers are readable and writable.

- The lower 6 bits of the SSPSTAT are read only. The upper two bits of the SSPSTAT are read/write.
- SSPSR is the shift register used for shifting data in or out. SSPBUF is the buffer register to which data bytes are written to or read from.
- SSPADD register holds the slave device address when the SSP is configured in I2C slave mode.
- When the SSP is configured in master mode, the lower seven bits of SSPADD act as the baud rate generator reload value.
- In receive operations, SSPSR and SSPBUF together, create a double buffered receiver. When SSPSR receives a complete byte, it is transferred to SSPBUF and the SSPIF interrupt is set.
- During transmission, the SSPBUF is not double buffered. A write to SSPBUF will write to both SSPBUF and SSPSR.

Q.9 Draw and explain the block diagram of MSSP, I2C slave mode in detail. ☞ [SPPU: May-22, Marks 9, May-18,19, Nov.-18, Marks 8]

Ans. : MSSP I2C Slave Mode

- The MSSP module in I2C mode fully implements all master and slave functions and provides interrupts on start and stop bits in hardware to determine a free bus (multi-master function).
- The MSSP module implements the standard mode specifications, as well as 7-bit and 10-bit addressing.
- Two pins are used for data transfer :

  Serial clock (SCL) - RB1/AN10/INT1/SCK/SCL

  Serial data (SDA) - RB0/AN12/INT0/FLT0/SDI/SDA
- The MSSP module functions are enabled by setting MSSP Enable bit, SSPEN (SSPCON1<5>). The SSPCON1 register allows control of the I2C operation. Four mode selection bits

(SSPCON1<3:0>) allow one of the following I2C modes to be selected :

- I2C Master mode, clock
- I2C Slave mode (7-bit address)
- I2C Slave mode (10-bit address)
- I2C Slave mode (7-bit address) with start and stop bit interrupts enabled
- I2C Slave mode (10-bit address) with start and stop bit interrupts enabled

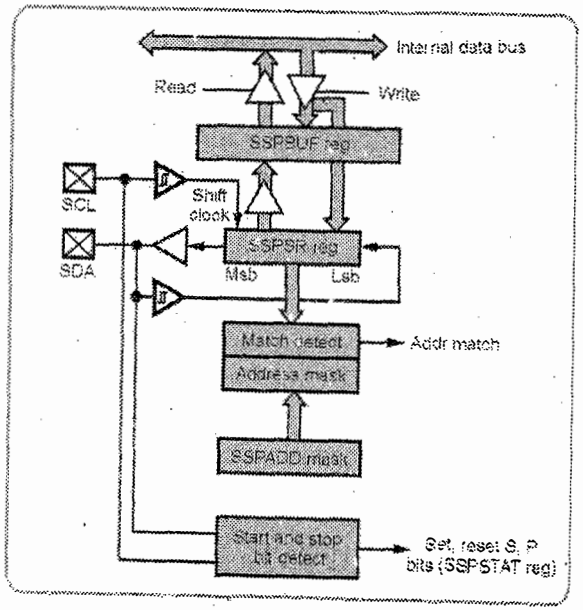


Fig. Q.9.1 I2C block diagram slave mode

- I2C Firmware controlled master mode, slave is Idle Selection of any I2C mode with the SSPEN bit set forces the SCL and SDA pins to be open-drain, provided these pins are programmed as inputs by setting the appropriate TRISC or TRISD bits. To ensure

proper operation of the module, pull-up resistors must be provided externally to the SCL and SDA pins. A simplified block diagram of I2C mode is shown in Fig. Q.9.1.

- In Slave mode, the SCL and SDA pins must be configured as inputs (TRISC<4:3> set). The MSSP module will override the input state with the output data when required (slave-transmitter).
- The I2C Slave mode hardware will always generate an interrupt on an address match. Address masking will allow the hardware to generate an interrupt for more than one address (up to 31 in 7-bit addressing and up to 63 in 10-bit addressing).
- Through the mode select bits, the user can also choose to interrupt on start and stop bits. When an address is matched, or the data transfer after an address match is received, the hardware automatically will generate the Acknowledge (ACK) pulse and load the SSPBUF register with the received value currently in the SSPSR register.
- Any combination of the following conditions will cause the MSSP module not to give this ACK pulse :
- The Buffer Full bit, BF (SSPSTAT<0>), was set before the transfer was received.
- The overflow bit, SSPOV (SSPCON1<6>), was set before the transfer was received.
- In this case, the SSPSR register value is not loaded into the SSPBUF, but a bit, SSPIF, is set. The BF bit is cleared by reading the SSPBUF register, while bit, SSPOV, is cleared through software.
- Once the MSSP module has been enabled, it waits for a start condition to occur. Following the start condition, the 8 bits are shifted into the SSPSR register. All incoming bits are sampled with the rising edge of the clock (SCL) line. The value of register SSPSR<7:1> is compared to the value of the SSPADD register.

- The address is compared on the falling edge of the eighth clock (SCL) pulse. If the addresses match and the BF and SSPOV bits are clear, the following events occur :
  1. The SSPSR register value is loaded into the SSPBUF register.
  2. The Buffer Full bit, BF, is set.
  3. An ACK pulse is generated.
  4. The MSSP Interrupt Flag bit, SSPIF, is set (and interrupt is generated, if enabled) on the falling edge of the ninth SCL pulse.
- Reception : When the R/W bit of the address byte is clear and an address match occurs, the R/W bit of the SSPSTAT register is cleared. The received address is loaded into the SSPBUF register and the SDA line is held low (ACK).
- Transmission: When the R/W bit of the incoming address byte is set and an address match occurs, the R/W bit of the SSPSTAT register is set. The received address is loaded into the SSPBUF register. The ACK pulse will be sent on the ninth bit and pin RB1/AN10/INT1/SCK/SCL is held low regardless of SEN.

Q.10 Draw and explain the block diagram of MSSP, I2C Master mode in detail. **[SPPU : Nov.-16, Marks 8]**

Ans. : **MSSP I2C Master Mode**

- Master mode is enabled by setting and clearing the appropriate SSPM bits in SSPCON1 and by setting the SSPEN bit. In Master mode, the SCL and SDA lines are manipulated by the MSSP hardware if the TRIS bits are set.
- Master mode operation is supported by interrupt generation on the detection of the start and stop conditions. The Stop (P) and Start (S) bits are cleared from a Reset or when the MSSP module is disabled.

- Control of the I2C bus may be taken when the P bit is set or the bus is Idle, with both the S and P bits clear.
- In Firmware Controlled Master mode, user code conducts all I2C bus operations based on start and stop bit conditions.
- Once master mode is enabled, the user has six options :
  1. Assert a start condition on SDA and SCL.
  2. Assert a repeated start condition on SDA and SCL.
  3. Write to the SSPBUF register initiating transmission of data / address.
  4. Configure the I2C port to receive data.
  5. Generate an acknowledge condition at the end of a received byte of data.
  6. Generate a stop condition on SDA and SCL.
- The following events will cause the MSSP Interrupt Flag bit, SSPIF, to be set (and MSSP interrupt, if enabled) :
- Start condition, Stop condition, Data transfer byte transmitted / received, Acknowledge transmit, Repeated Start.
- Fig. Q.10.1 Shows the MSSP I2C master mode configuration. (Refer Fig. Q.10.1 on next page)

Operation of master mode :

The master device generates all of the serial clock pulses and the start and stop conditions. A transfer is ended with a stop condition or with a repeated start condition. Since the repeated start condition is also the beginning of the next serial transfer, the I2C bus will not be released. In Master Transmitter mode, serial data is output through SDA, while

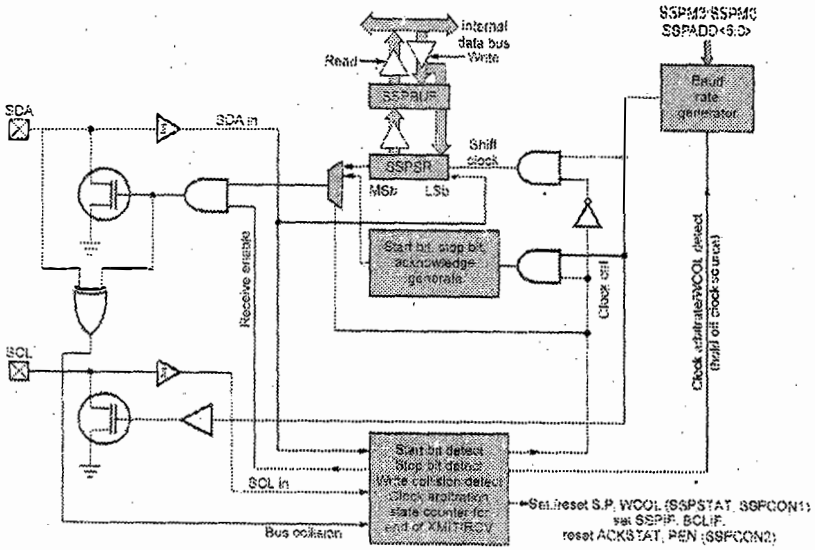


Fig. Q.10.1 MSSP I2C master mode

SCL outputs the serial clock. The first byte transmitted contains the slave address of the receiving device (seven bits) and the Read/Write (R/W) bit. In this case, the R/W bit will be logic '0'. Serial data is transmitted eight bits at a time. After each byte is transmitted, an Acknowledge bit is received. Start and stop conditions are output to indicate the beginning and the end of a serial transfer. In master receive mode, the first byte transmitted contains the slave address of the transmitting device (7 bits) and the R/W bit. In this case, the R/W bit will be logic '1' Thus, the first byte transmitted is a 7-bit slave address followed by a '1' to indicate the receive bit. Serial data is received via SDA, while SCL outputs the serial clock. Serial data is received eight bits at a time. After each byte is received, an Acknowledge bit is transmitted. Start and Stop conditions indicate the beginning and end of transmission. The baud rate Generator used for the SPI mode operation is used to set the SCL clock frequency for either 100 kHz, 400 kHz or 1 MHz I2C operation.

A typical transmit sequence would go as follows :

1. The user generates a start condition by setting the Start Enable bit, SEN (SSPCON2<0>).
2. SSPIF is set. The MSSP module will wait the required start time before any other operation takes place.
3. The user loads the SSPBUF with the slave address to transmit.
4. Address is shifted out the SDA pin until all eight bits are transmitted.
5. The MSSP module shifts in the ACK bit from the slave device and writes its value into the SSPCON2 register (SSPCON2<6>).
6. The MSSP module generates an interrupt at the end of the ninth clock cycle by setting the SSPIF bit.
7. The user loads the SSPBUF with eight bits of data.
8. Data is shifted out the SDA pin until all eight bits are transmitted.
9. The MSSP module shifts in the ACK bit from the slave device and writes its value into the SSPCON2 register (SSPCON2<6>).
10. The MSSP module generates an interrupt at the end of the ninth clock cycle by setting the SSPIF bit.
11. The user generates a Stop condition by setting the Stop Enable bit, PEN (SSPCON2<2>).
12. Interrupt is generated once the Stop condition is complete.

Q.11 Explain the use of BRGH register for calculation of Baud rates in USART ☞ **[SPPU : May-22, Marks-9, May-17,18, Dec.-18, Marks 8]**

Ans. : **USART Baud Rate Generator (BRG)**

- The BRG is a dedicated 8-bit, or 16-bit, generator that supports both the asynchronous and synchronous modes of the USART. By

default, the BRG operates in 8-bit mode. Setting the BRG16 bit (BAUDCON<3>) selects 16-bit mode.

- It is a dedicated 8-bit baud rate generator. The SPBRG register controls the period of a free running 8-bit timer. In Asynchronous mode, bit BRGH (TXSTA<2>) also controls the baud rate.
- Calculation of baud rate for different USART modes, which only apply in master mode (internal clock).
- The data on the RC7/RX/DT pin is sampled three times by a majority detect circuit to determine if a high or a low level is present at the RX pin. For example
- Fosc = 10 MHz, Desired Baud Rate = 9600, BRGH = 0, SYNC = 0
- Given the desired baud rate and Fosc, the nearest integer value for the SPBRGH : SPBRG registers can be calculated using the equation 1. From this, the error in baud rate can be determined. An example

Desired baud rate = Fosc / (64(X+1)) ... (1)

$$X = [Fosc / (Desired\ baud\ rate \times 64)] - 1$$

$$\text{Find value of } X = [Fosc / (Desired\ baud\ rate \times 64)] - 1$$

$$= [10 \times 10^6 / (9600 \times 64)] - 1$$

$$= [156250 / 9600] - 1$$

$$= 15.27 = 15$$

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.

Writing a new value to the SPBRG register causes the BRG timer to be reset (or cleared). This ensures the BRG does not wait for a timer overflow before outputting the new baud rate. The calculation of low and high baud rate is shown in Table Q.11.1

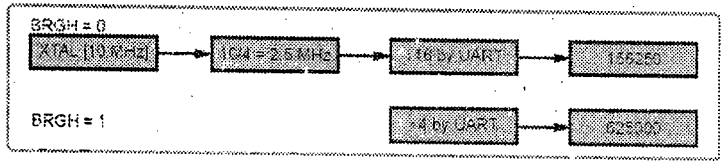
Table Q.11.1 Calculation of BRG value

Configuration Bits			BRG/EUSART Mode	Baud rate formula
SYNC	BRG16	BRGH		
0	0	0	8-bit/Asynchronous	Fosc/[64(n + 1)]
0	0	1	8-bit/Asynchronous	Fosc/[64(n + 1)]
0	1	0	16-bit/Asynchronous	
0	1	1	16-bit/Asynchronous	Fosc/[4(n + 1)]
1	0	x	8-bit/Asynchronous	
1	1	x	16-bit/Asynchronous	

In short, the baud rate formulas are

Sync	BRGH = 0 Low speed	BRGH = 1 High speed
0	Asynchronous BR = Fosc/(64(X+1))	BR = Fosc/(16(X+1))
1	Synchronous BR = Fosc/(4(X+1))	NA

X = Value in SPBRG (0 to 255)



Q.12 Find the value to be loaded into SPBRG register for baud rate of 1200, 2400, 4800, 9600, 19200 and 38400 for 10 MHz oscillator. BRGH = 0 and 1

Ans. : Value to be loaded in SPBRG register for BR of 1200 is BRGH = 0

$$X = [Fosc/ (Desired\ baud\ rate \times 64)] - 1$$

$$= [156250 / Desired\ BR] - 1$$

$$= [156250 / 1200] - 1$$

$$= (129)_{10} = (81)_{16}$$

Value to be loaded in SPBRG register for BR of 1200 is BRGH = 1

$$X = [Fosc / (Desired\ baud\ rate \times 16)] - 1$$
$$= [625000 / Desired\ BR] - 1$$
$$= [625000 / 1200] - 1$$
$$= (520)_{10} = (208)_{16}$$

Baud rate	BRGH = 0		BRGH = 1	
	Decimal	HEX	Decimal	HEX
57600	2	2	10	0A
38400	3	3	15	0F
19200	7	7	32	20
9600	15	F	64	40
4800	32	20	129	81
2400	64	40	259	103
1200	129	81	520	208

Q.13 Explain concept of USART trans - receiver with TXSTA and RCSTA regisers.

Ans. : **Universal asynchronous receiver transmitter**

- The Universal Synchronous Asynchronous Receiver Transmitter (USART) module is one of the two serial I/O modules. (USART is also known as a Serial Communications Interface or SCI.)
- The USART can be configured as a full duplex asynchronous system that can communicate with peripheral devices, such as CRT terminals and personal computers, or it can be configured as a half-duplex synchronous system that can communicate with peripheral devices, such as A/D or D/A integrated circuits, serial EEPROMs, etc.

- The USART can be configured in the following modes :
  - Asynchronous (full-duplex), Synchronous - Master (half-duplex), Synchronous - Slave (half-duplex)
- In order to configure pins RC6/TX/CK and RC7/RX/DT as the Universal Synchronous Asynchronous Receiver Transmitter :
  - bit SPEN (RCSTA<7>) must be set (= 1),
  - bit TRISC<6> must be cleared (= 0) and
  - bit TRISC<7> must be set (=1).

1. Transmit Status and Control Register (TXSTA)

It is 8 bit register used to select synchronous/asynchronous mode and data framing size. The various bits used in TXSTA has different functions, D6 bit decides the data size to be communicated. The D2 bit BRGH is used to select the higher speed of transmission. The default is lower baud rate transmission in asynchronous mode.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-0
CSRC	TX9	TXEN(1)	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

Transmit control register (TXSTA)

2. Receive Status and Control Register (RCSTA) :

It is 8 bit register, used to enable the serial port to receive data.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-1	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

Receiver status and control register

Q.14 Draw and explain block diagram of USART Transmitter

[SPPU : Dec-18, May-17,18, Marks 8]

Ans. : USART Transmitter

- The USART transmitter block diagram is shown in Fig. Q.14.1 The heart of the transmitter is the Transmit (Serial) Shift Register (TSR).
- The shift register obtains its data from the Read/Write Transmit Buffer register, TXREG. The TXREG register 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 (occurs in one TCY), the TXREG register is empty and the TXIF flag bit (PIR1<4>) is set.
- This interrupt can be enabled or disabled by setting or clearing the interrupt enable bit, TXIE (PIE1<4>).
- When TSR fetches the data from TXREG, it clears the TMRT flag bit indicating it is full. TSR is parallel in serial out shift register and not accessible to the programmer. Only write to TXREG automatically load the TSR.
- TXIF will be set regardless of the state of TXIE; it cannot be cleared in software. TXIF is also not cleared immediately upon loading TXREG, but becomes valid in the second instruction cycle following the load instruction.

- Polling TXIF immediately following a load of TXREG will return invalid results.
- While TXIF indicates the status of the TXREG register, another bit, TRMT (TXSTA<1>), shows the status of the TSR register. TRMT is a read-only bit which is set when the TSR register is empty. No interrupt logic is tied to this bit so the user has to poll this bit in order to determine if the TSR register is empty.

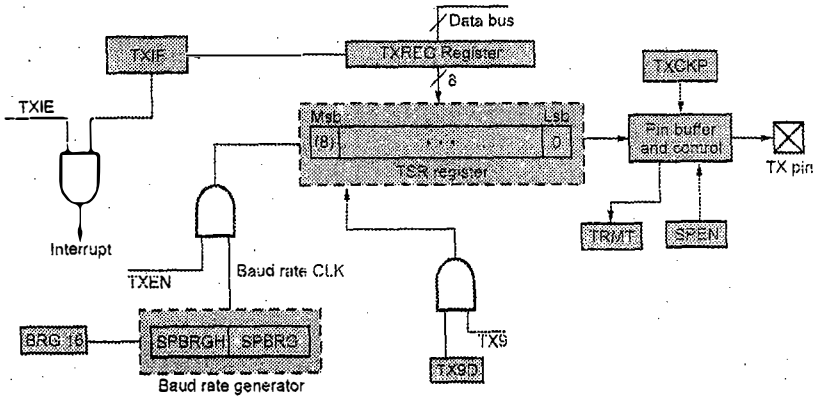


Fig. Q.14.1 USART transmitter block diagram

- The TXCKP bit (BAUDCON<4>) allows the TX signal to be inverted (polarity reversed). Devices that buffer signals from TTL to RS-232 levels also invert the signal. Inverting the polarity of the TX pin data by setting the TXCKP bit allows for use of circuits that provide buffering without inverting the signal.

Q.15 Draw and explain block diagram of USART Receiver

Ans. : USART Receiver

The receiver block diagram is shown in Fig. Q.15.1 The data is received on the RX 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. The RXDTP bit (BAUDCON<5>) allows the RX signal to be inverted (polarity reversed). Devices that buffer signals from RS-232 to TTL levels also perform an inversion of the signal (when RS-232 = positive, TTL = 0). Inverting the polarity of the RX pin data by setting the RXDTP bit allows for the use of circuits that provide buffering without inverting the signal.

To set up an asynchronous reception :

1. Initialize the SPBRGH : SPBRG registers for the appropriate baud rate. Set or clear the BRGH and BRG16 bits, as required, to achieve the desired baud rate.
2. Enable the asynchronous serial port by clearing bit, SYNC and setting bit, SPEN.
3. If the signal at the RX pin is to be inverted, set the RXDTP bit.
4. If interrupts are desired, set enable bit, RCIE.
5. If 9-bit reception is desired, set bit, RX9.
6. Enable the reception by setting bit, CREN.
7. Flag bit, RCIF, will be set when reception is complete and an interrupt will be generated if enable bit, RCIE, was set.
8. Read the RCSTA register to get the 9th bit (if enabled) and determine if any error occurred during reception.
9. Read the 8-bit received data by reading the RCREG register.

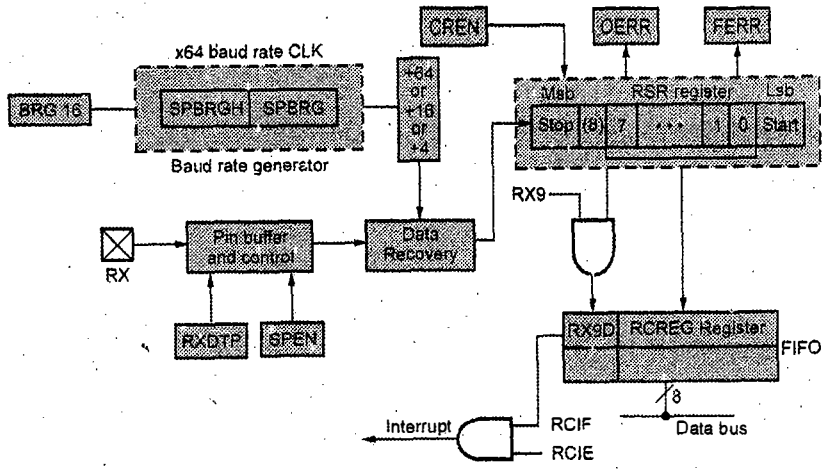


Fig. Q.15.1 USART receiver block diagram

10. If any error occurred, clear the error by clearing enable bit, CREN.
11. If using interrupts, ensure that the GIE and PEIE bits in the INTCON register (INTCON<7:6>) are set.

Q.16 Write an embedded C program to interface serial port with PC for both side communication and displaying key pressed on LCD and hyper terminal

Ans. : Serial Communication Program

```
/*Baud Rate GENERATION
 * n  => required baudrate
 * BRGH   =  0
 * SPBRG  =  (Fosc / (64 * n)) -1
 * For 9600 baudrate, SPBRG ~=77
 */ #include<p18F4550.h>
    # include<stdio.h>
    #include "LCD_SIT.h"
    #define Fosc 48000000UL
    void InitUART(unsigned int baudrate);
```

```
    void SendChar(unsigned char data);
    void putch(unsigned char data);
    unsigned char GetChar(void);

void main(void)
{
   InitUART(9600);
   printf("\r\nHello, Enter any Key from Keyboard\r\n");
   Init_LCD();
   lcdcmd(0x80);
   MSdelay(50);
  while(1)
  {
   printf("%c",GetChar()); //Receive character from PC and echo back
    lcddata(RCREG);
     MSdelay(50);
   }
}

void InitUART(unsigned int baudrate)
{
    TRISCbits.RC6 = 0;                  //TX pin set as output
    TRISCbits.RC7 = 1;                  //RX pin set as input
    SPBRG =(unsigned char)(((Fosc /64)/baudrate)-1);
     BAUDCON = 0b00000000;              //Non-inverted data; 8-bit baud
                                        //rate generator
TXSTA = 0b00100000;                     //Asynchronous 8-bit; Transmit
                                        //enabled; Low speed
                                        // baudrate select
RCSTA = 0b10010000;                     //Serial port enabled; 8-bit data;
                                        // single receive enabled
}

  void SendChar(unsigned char data)
 {
```



```
  while (TXSTAbits.TRMT = = 0);  //Wait while transmit register is
                                 //empty
  TXREG = data; //Transmit data
}
void putch(unsigned char data)
{
SendChar(data);
}
unsigned char GetChar(void)
}
```

Q.17 State features of RTC and draw an interfacing diagram to interface with PIC.

☞ [SPPU : May-22, Marks 9, May-18,17, Dec.-17,18, Nov.-16]

Ans. : Real Time Clock (RTC) Interface with I2C

Real time clock is used to synchronize all the operations of CPU and avoid the malfunction in real time considerations.

Features of RTC DS1307 : The DS1307 serial Real-Time Clock (RTC) is a low-power, full binary-coded decimal (BCD) clock/calendar.

- 56-Byte, Battery-Backed, Non-volatile (NV) SRAM for data storage
- The crystal frequency is 32.768 kHz
- Address and data are transferred serially through an I2C™, bidirectional bus.
- Real-Time Clock (RTC) clock/calendar counts seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation valid up to 2100
- The end of the month date is automatically adjusted for months with fewer than 31 days, including corrections for leap year.
- The clock operates in either the 24-hour or 12-hour format with AM/PM indicator.

- Programmable square-wave output signal (1,4,8,64 kHz output)
- Signal automatic power-fail detect and switch circuitry
- Consumes less than 500 nA in battery - backup Mode with oscillator running
- Optional industrial temperature range : – 40 °C to + 85 °C

Operational Tips for DS1307

- The DS1307 operates as a slave device on the I2C bus. Access is obtained by implementing a START condition and providing a device identification code followed by a register address
- Subsequent registers can be accessed sequentially until a STOP condition is executed. When VCC falls below 1.25 × VBAT, the device terminates an access in progress and resets the device address counter.
- Inputs to the device will not be recognized at this time to prevent erroneous data from being written to the device from an out-of-tolerance system.
- When VCC falls below VBAT, the device switches into a low-current battery-backup mode.

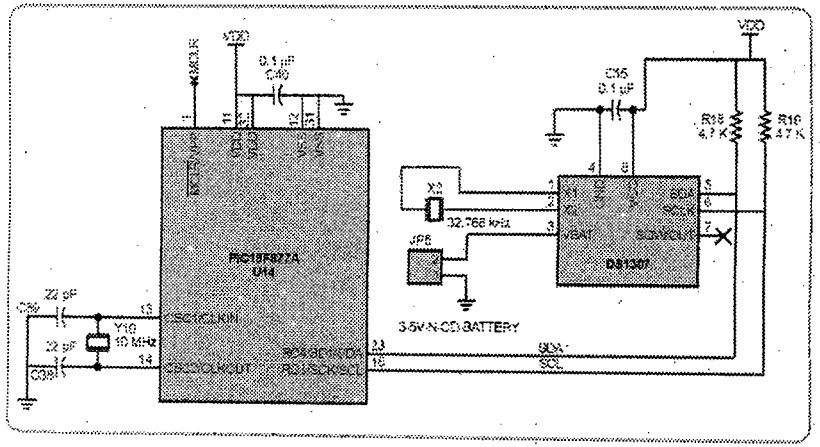


Fig. Q.17.1 RTC Interface to PIC18FXXXX

- Upon power-up, the device switches from battery to VCC when VCC is greater than VBAT +0.2 V and recognizes inputs when VCC is greater than 1.25 × VBAT.
- The complete interface diagram is shown in Fig. Q.17.1

Q.18 Draw an interfacing diagram of RTC with PIC ? Write an initialization Program.

[SPPU : May-22 Marks 9, May-15,17,16, Nov.-15,16, April-13, Marks 8]

OR Draw an interfacing diagram to interface the RTC and write an embedded C program to display the time on LCD. (Only remove the features)

Ans. : Interfacing diagram of RTC

The Complete interfacing diagram of RTC is same is Fig. Q.17.1.

Program ;

Initialization program to display time on LCD display using RTC

```
#include <p18f4550.h>
#include<stdio.h>
#include "LCD_SIT.h"
#include "I2C_SIT.h"

void set_time(unsigned char address, unsigned char x);
unsigned char get_time(unsigned char address);
void decode(unsigned char val);
void init_data(void);

unsigned char str[16];

void main()
{
   Unsigned char sec,min,hrs,date,month,year,unit,ten;
   i2c_interface_init();
   SSPADD = 126;        //set i2c clock
```

```
  SSPCON1= 0b00001000;   //I2C Master mode,
  clock = FOSC/(4 * (SSPADD + 1))
  SSPSTATbits.SMP = 1;  //Slew rate control disabled
  for Standard Speed mode (100 kHz and 1 MHz)
    SSPCON1bits.SSPEN = 1; //Enables the serial port and
                            configures the SDA
                            and SCL pins as the serial port pins
    init_data();           //ROUTINE TO SET TIME.
                           //Uncomment after time set to the RTC

    sec = get_time(0);
    min = get_time(1);
    hrs = get_time(2);
    date = get_time(4);
    month = get_time(5);
    year = get_time(6);

    i2c_interface_deinit();

    Init_LCD();

while(1)
  {
   sprintf(str,"Date:%02x/%02x/%02x",date,month,year);
   lcdcmd(0x80);
   MSdelay(50);
   LCDDisplayStr(str);
   sprintf(str,"Time:%02x:%02x:%02x",hrs,min,sec);
   lcdcmd(0xC0);
   MSdelay(50);
   LCDDisplayStr(str);
  }
}

void set_time(unsigned char address, unsigned char x)  // sets time
 {
```



```
  i2c_start();
  i2c_write(RTC_ADD);
  i2c_write(address);
  i2c_write(x);
  i2c_stop();
  return;
}
unsigned char get_time(unsigned char address) //Gets time
{
 unsigned char data;
  i2c_start();
  i2c_write(RTC_ADD);
  i2c_write(address);
  i2c_restart();
  i2c_write(RTC_ADD | 0x01);
  SSPCON2bits.ACKDT=1;
  data=i2c_read();
  i2c_stop();
  return (data);
}
void decode(unsigned char val) // Function separates the variable into
higher and lower nibble
{
 unsigned char tens,units;
  tens=val>>4;
  units=val&0x0F;
  MSdelay(50);
  lcddata(tens+48);
  MSdelay(50);
  lcddata(units+48);
  MSdelay(50);
  return;
}
void init_data(void)
 {
     set_time(0x00,0x45);    // seconds
```

```
    set_time(0x01,0x57);   // Minutes
    set_time(0x02,0x11);   // Hours along with 12hours and
                           // AM/PM selection
    set_time(0x03,0x06);   // Day
    set_time(0x04,0x19);   // Date
    set_time(0x05,0x08);   // Month
    set_time(0x06,0x14);   // Year
}
```

Q.19 State features of EEPROM, Draw an interfacing diagram EEPROM using SPI protocol with PIC 18FXXXX

[SPPU : May-22, Marks 9]

Ans. : EEPROM Interface with I2C and SPI

- The AT24C02A/04A provides 2048/4096 bits of serial electrically erasable and programmable read-only memory (EEPROM) organized as 256/512 words of 8 bits each.

Features

- Write protect pin for hardware data protection: utilizes different array protection compared to the AT24C02A/04A
- Medium-voltage and standard-voltage operation : 5.0 (VCC = 4.5 V to 5.5 V) and 2.7 (VCC = 2.7 V to 5.5 V)
- Internally organized 256 × 8 (2 K), 512 × 8 (4 K)
- Two-wire serial interface.
- Schmitt trigger, filtered inputs for noise suppression.
- Bidirectional data transfer protocol.
- 400 kHz (2.7 V, 5 V) Clock rate.
- 8-byte Page (2 K), 16-byte Page (4 K) write modes.
- Partial page writes allowed.
- Self-timed write cycle (5 ms Max).
- High reliability.
- Endurance : One million write cycles.
- Data Retention : 100 Years.

- Lead-Free/Halogen-Free devices available.
- 8-lead PDIP, 8-lead JEDEC SOIC, and 8-lead TSSOP Packages.

Interfacing of EEPROM with SPI Bus :

- Serial Peripheral Interface (SPI) is a synchronous serial data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over short distances. It can also be used for communication between two microcontrollers
- EEPROM (electrically erasable programmable read-only memory) is user-modifiable read-only memory (ROM) that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage.
- It is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration.
- With an SPI connection there is always one master device (usually a microcontroller) which controls the peripheral devices. Typically there are three lines common to all the devices,
  - Master In Slave Out (MISO) - The Slave line for sending data to the master,
  - Master Out Slave In (MOSI) - The Master line for sending data to the peripherals,
  - Serial Clock (SCK) - The clock pulses which synchronize data transmission generated by the master and
  - Slave Select pin - the pin on each device that the master can use to enable and disable specific devices. When a device's Slave Select pin is low, it communicates with the master. When it's high, it ignores the master.
  - The SPI Controller here acts as a master device and controls EEPROM which acts as a slave. The read-write operations are

accomplished by sending a set of control signals including the address and/or data bits. The control signals must be accompanied with proper clock signals.

- The basic operation of the SPI based EEPROM's is to send a command, such as WRITE, followed by an address and the data. In WRITE operation, the EEPROM to store the data.
- Four numbers of EEPROM lines are controlled by SPI Enabled drivers. The SPI Lines Chip Select of CS (PORTC.0), serial clock of CLK (PORTC.3), serial input data of MISO (PORTC.4) and serial output data of MOSI (PORTC.5) connected to the SPI based serial EEPROM IC. The EEPROM read & write operations are done in PIC 18F interface by using these CS, CLK, MOSI, MISO SPI lines. As shown in Fig. Q.19.1

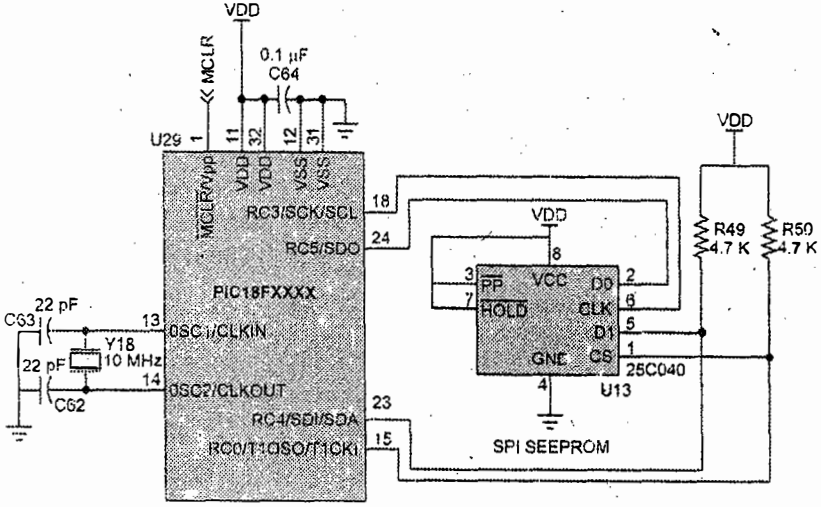


Fig. Q.19.1 EEPROM interface with PIC18FXXXX

Q.20 Write an PIC18 C program to read, write and erase contents of SPI - EEPROM.

Ans. : Program

```
#include<pic.h>
#include<stdio.h>

__CONFIG(0x3f72);  //HS oscillator,BODEN,PWRT and disable others

#define FOSC      10000  //10Mhz==>10000Khz
#define BAUD_RATE  9.6   //9600 Baudrate
#define BAUD_VAL   (char)(FOSC/ (16 * BAUD_RATE )) - 1;
//Calculation For 9600 Baudrate @10Mhz
//SPI lines
#define CS  RC0        //Chip select ON RC2
#define SI  RC5        //Master Out Slave In
#define SO  RC4        //Master in slave out
#define SCK RC3        //Clock

/*SPI_COMMANDS*/
#define READ  0x03
#define WRITE 0x02
#define WRDI  0x04
#define WREN  0x06
#define RDSR  0x05
#define WRSR  0x01

unsigned char i,a,j;
unsigned char Msg[]="SPI TEST Program";
void Serial_init(void);
void SPi_init(void);
void SPi_WRITE(unsigned char);
unsigned char SPi_RDSR(void);
unsigned char SPi_READ(unsigned char);
void DelayMs(unsigned int);

void main()
{
```

```
  unsigned char x;
  TRISC=0xd0;  //Enable RX,TX pin and Set MISO as input
  TRISD=0;     //and set the remaining pins as output
  Serial_init();//Setup the serial port
  SPi_init();
  DelayMs(10);
  while(!SPi_RDSR()); //SPI ready?
  SPi_WRITE(0x00);   //Send initialisation Command
  DelayMs(10);
  while(1)
  {
    x=0;
    while(x<16)
    {
      TXREG=PORTD=SPi_READ(x); //Read byte from 25c040 and
                                 send
                               // via Usart
      ++x;
      DelayMs(50);
    }
  }
}

void SPi_init()
{
  CS=1;            //Make CS pin high
  SI=0;            //Clear input pin
  SCK=0;           //Clock low
}
unsigned char SPi_RDSR()
{
  unsigned char Data=0x05;
  CS=0;     //Initiate transmission by pulling CS pin low
  for(i=0;i<8;i++)
  {
    SI=(Data & 0x80)?1:0;
```



```
  //Send Read Status Register Command bit by bit(MSB) first
    SCK=1;
    Data=Data<<1;
    SCK=0;
  }
  for(i=0; i<8; i++)   //wait for 0x00--device not busy
  {
     SCK=1:
     Data|=((SO & 1)?1:0);
    Data=Data<<1;
     SCK=0;
  }
  CS=1;                  //Pull up
  return !Data;
}

void SPi_WRITE(unsigned char Addr)
{
  unsigned char Data=WREN;
  int AH=WRITE;
  AH=(AH<<8)+Addr;
  CS=0;
  for(i=0;i<8;i++)     //Send Write Enable
  {
    SI=(Data & 0x80)?1:0;
    SCK=1;
    Data=Data<<1;
    SCK=0;
  }
  CS=1;                //Rise CS and pull down again
  CS=0;
  for(i=0;i<16;i++)   //Send WRITE command and Addr
  {
    SI=(AH & 0x8000)?1:0;
    SCK=1;
    AH=AH<<1;
```

```
    SCK=0;
  }
  for(i=0;i<16;i++)   //Send Data's
  {
    Data=Msg[i];
    for(j=0;j<8;j++)
    {
      SI=(Data & 0x80)?1:0;
      SCK=1;
      Data=Data<<1;
      SCK=0;
    }
  }
  CS=1;
}
unsigned char SPi_READ(unsigned char Addr)
{
  int Data=READ;
  unsigned char RData=0;
  Data=(Data<<8)|Addr;
  while(!SPi_RDSR());     //Device Ready?Proceed to next statement
                          //else wait
  CS=0;                   //Pull down CS
  for(i=0;i<16;i++)       //Send READ command and Addr
  {
    SI=(Data & 0x8000)?1:0;
    SCK=1;
    Data=Data<<1;
    SCK=0;
  }
  for(i=0; i<8; i++) //Read a Byte
  {
    RData=RData<<1;
    SCK=1;
    RData|=((SO & 1)?1:0);
    SCK=0;
```



```
  }
  CS=1;
  return RData;
}
void Serial_init()
{
  TXSTA=0x24;          //Transmit Enable
  SPBRG=BAUD_VAL;      //9600 baud at 10Mhz
  RCSTA=0x90;          //Usart Enable, Continuous receive enable
  TXREG=0x00;          //Dummy transmission
  printf("\033[2J");   //Clear the Hyphenterminal;
}
void putch(unsigned char character)
{
  while(!TXIF);          //Wait for the TXREG register to be empty
  TXREG=character;       //Display the Character
}

void DelayMs(unsigned int Ms)
{
  int delay_cnst;
  while(Ms>0)
  {
    Ms--;
    for(delay_cnst = 0;delay_cnst <220;delay_cnst++);
  }
}
```

Q.21 Design a PIC 18 F4550 based traffic light controller.

☞ **[SPPU : 10-12 Marks]**

Ans. : Design of PIC test Board : Fig. Q.21.2 shows the design of PIC test board for verifying the LED connected to port B, with switches and Buzzer to port D, with relay interface for Port A.

Step 1 : Design of power supply :

The microcontroller selected is PIC 18F4550 which works on the frequency of oscillator ranging from 0 to 20 MHz and requires power supply of ± 5 or ±12 V. A simple circuit design for +5 V is shown in Fig. Q.21.1

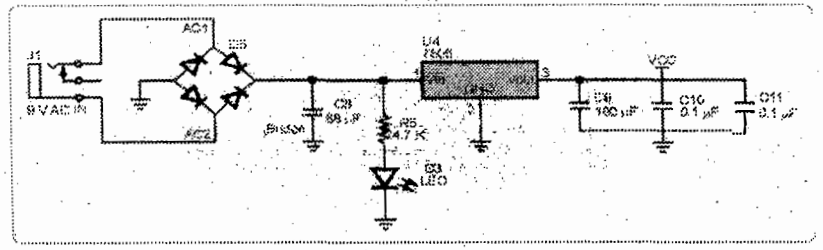


Fig. Q.21.1 Sample for power supply design

Step 2 : Design of clock circuit :

The Quartz crystal is connected to OSC1 and OSC2 pin in order to synchronize the operation of all components connected with internal and external means. The values for C1 and C2 are selected according to the crystal frequency for stabilizing the oscillator pulses. In general with quartz crystal 22-33 µF is preferred.

Step 3 : Design of reset circuit :

The RC high pass filter with C=0.1µF along with 20 kΩ register is connected to MCLR pin. When high pulse appear on it, resets the contents of inter registers and SFRS to initial value.

Step 4 : Configuration of Port :

PIC has 33 I/O lines which can be configured as input and output using TRISX register as

if TRISX = 0 - Ports (A - E) are configured as output ports

if TRISX = 1 - Ports (A - E) are configured as input ports.

LM 293D (Motor Driver) : The L293D is a quadruple high-current half-H driver designed to provide bidirectional drive currents of up to 600 - mA at voltages from 4.5 V to 36 V. It is designed to drive inductive loads such as relays, solenoids, dc and bipolar stepping motors, as well as other high-current/high-voltage loads in positive-supply applications. When an enable input is high, the associated drivers are enabled and their outputs are active and in phase with their inputs.

- **Logic and implementation details**

Road	Port Line	Traffic lights
	RD0	Red
Traffic Lane 1	RD1	Yellow
	RD2	Green
	RD4	Red
Traffic Lane 2	RD5	Yellow
	RD6	Green
	RC0	Red
Traffic Lane 3	RC1	Yellow
	RC2	Green
	RC4	Red
Traffic Lane 4	RC5	Yellow
	RC6	Green

Connection diagram :

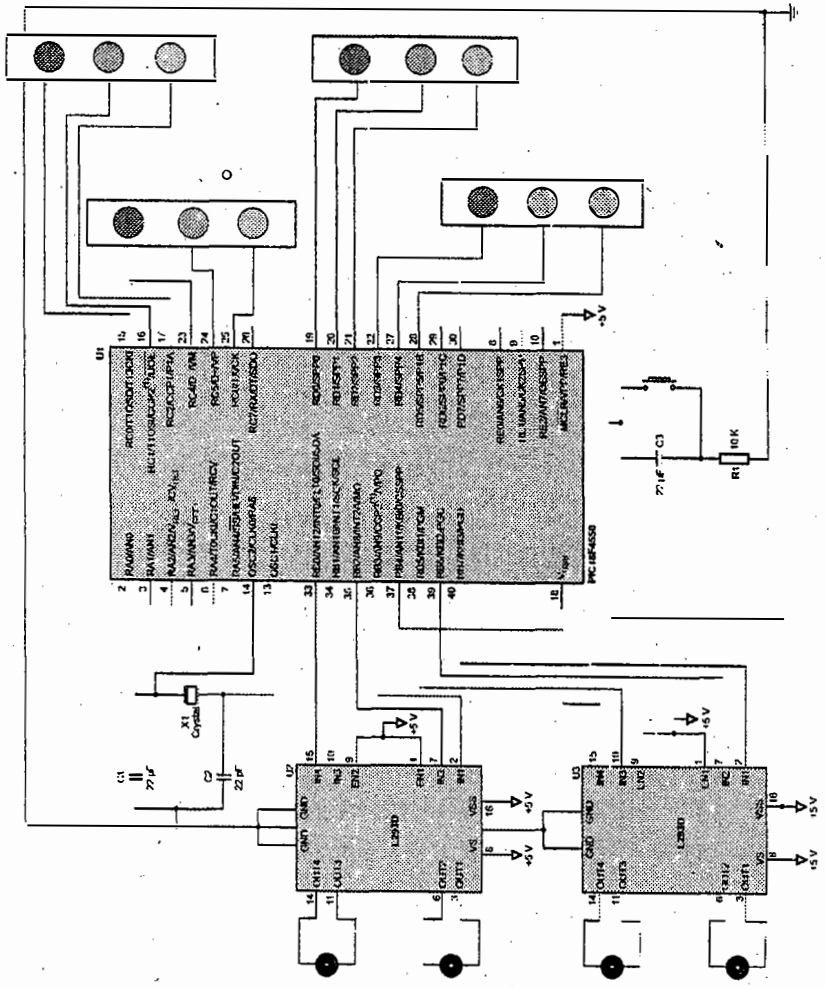


Fig. Q.21.2 : Traffic light controller system

Program

```
#include<p18f4550.h>
#include<stdio.h>
void MSdelay(unsigned int itime);
void main()

{
TRISD=0x00;
TRISB=0x00;
while(1)
{
LATD=0x56;
MSdelay(3000);
LATD=0x59;
MSdelay(3000);
LATD=0x65;
LATB=0x01;
MSdelay(1000);
LATB=0x00;
MSdelay(3000);
LATB=0x02;
MSdelay(1000);
LATB=0x00;
LATD=0x95;
LATB=0x08;
MSdelay(1000);
LATB=0x00;
MSdelay(3000);
LATB=0x04;
MSdelay(1000);
LATB=0x00;
}
}
void MSdelay(unsigned int itime)
{
unsigned int i,j;
for(i=0;i<itime;i++)
for(j=0;j<1200;j++)
}
```

Important Points to Remember

1. Serial communication is cost effective than parallel.
2. RS232 uses asynchronous communication.
3. I2C is synchronous.
4. MSSP structure has both SPI and I2C Mode.
5. I2C Operates in master and slave mode.
6. Data on I2C bus is communicated between START, STOP conditions defined with SDA and SCL lines
7. ACK signal is must in I2C.
8. SPI bus uses three signals MOSI, MISO, SCK.
9. Each module has status and control register.
10. Flash bits in PIR 1 register are used to indicate the data transmission and reception during serial communication.
11. BRGH register is used to change the baud rate. If BRGH=0, USART uses ÷16 and if BRGH = 1, it is ÷ 4.
12. TSR register is not accessible to programmer.
13. RTC DS1307 uses I2C interfacing while DS1306 uses SPI.
14. EEPROM uses SPI control.

SFRS USED :

1. **MSSP Control Register1 (SSPCON1) -** Read and write, used to select the oscillator frequency, enable operation, and check for low and high level of clock.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM2	SSPM1	SSPM0
bit 7							bit 0

2. **MSSP Status Register (SSPSTAT) -** Used for SPI and I2C modes with checking of edge for clocking and check for transfer of receive the data.

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF
bit 7							bit 0

3. **MSSP Control Register2 (SSPCON2)**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GCEN	ACKSTAT	ACKDT(1)	ACKEN(2)	RCEN(2)	PEN(2)	RSEN(2)	SEN(2)
bit 7							bit 0

4. **Transmit Status And Control Register (TXSTA)**
It is 8 bit register used to select synchronous/asynchronous mode and data framing size. D6 bit decides the data size to be communicated. The D2 bit BRGH is used to select the higher speed of transmission. The default is lower baud rate transmission in asynchronous mode.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-1	R/W-0
CSRC	TX9	TXEN(1)	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

5. **Receive Status And Control Register (RCSTA) :** It is 8 bit register, used to enable the serial port to receive data.

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-1	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

END...

MAY-2022 (END SEM) [5870] - 1064	Solved Paper

Course 2019

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

N. B. :

i) Attempt Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8.

ii) Neat diagrams must be drawn wherever necessary.

ii) Figures to the right side indicate full marks.

iv) Assume suitable data, if necessary.

Q.1 A) *Explain in depth the programming model of PIC 18Fxxxx microcontroller.* **(Refer Q.8 of Chapter - 4)** **[6]**

B) *Explain the power down modes of PIC l8Fxxxx.* **(Refer Q.3 of Chapter - 4)** **[6]**

C) *Enlist features of PIC l8Fxxx microcontroller.* **(Refer Q.3 of Chapter - 4)** **[6]**

OR

Q.2 A) *Draw and explain the reset functional diagram of PIC 18Fxxxx.* **(Refer Q.11 of Chapter - 4)** **[6]**

B) *Explain with example functioning of ALU in PIC 18Fxxxx.* **(Refer Q.4 of Chapter - 4)** **[6]**

C] *Draw and explain the program memory of PIC 18Fxxxx.* **(Refer Q.7 of Chapter - 4)** **[6]**

Q.3 A) *Enlist specifications of ADC used, also draw the interfacing diagram of temperature sensor with PIC 18Fxxxx with initialization program.* **(Refer Q.25 (For features of ADC) and Q.28 (For Interfacing of temp sensor) of Chapter - 5)** **[9]**

B) *Explain interrupt structure of PIC l8Fxxxx with reasons.* **(Refer Q.10 of Chapter - 5)** **[8]**

OR

Q.4 A) *Draw and explain the Timer 1 operation of PIC l8Fxxxx in details, compare the Timer 0,1 and 2.* **(Refer Q.5 of Chapter - 5)** **[9]**

B) *Write a program for 1 kHz and 10 % duty cycle PWM generation with Fosc = 10 MHz.* **(Refer Q.21 of Chapter - 5)** **[8]**

Q.5 A) *Write an embedded C program to blink LED connected to port B of PICl8Fxxxx with delay of 1 msec using Timer 0, 16 bit.* **[9]**

Ans. : Refer Q.6 of Chapter - 4 for interfacing and program, Refer Q.7 of Chapter - 5 - For generation of Delay

Calculation of TMR0H and TMR0L values

1. Assume that Crystal frequency = 10 MHz
2. Internal time delay = $4/(10 \times 10^6) = 0.4\ \mu s$
3. N = 1 ms/0.4 μs = 2500
4. Count = 65536 – 2500 = $(63036)_{10}$
5. Hex value to be loaded = $(F63C)_{16}$
6. Load TMR0H = 3CH and TMR0L = F6H

```
#include <P18FXXXX.h>
void T0Delay(void);
void main(void)
{
unsigned char x;
TRISC=0;                          // configure Port D as output
PORTC=0x55;
while(1)
{
PORTC=~PORTC;                     // Toggle all bits of port C
T0Delay ( );
}
}
void T0Delay ( )
{
T0CON=0x08;                       // Timer0, 16 bit, no prescaler
TMR0H=0xF6;                       // Load Higher byte in TMR0H
TMR0L= 0x3C;                      // Load Lower byte to TMR0L
T0CONbits.TMR0ON=1;               // Start the timer for upcount
while(INTCONbits.TMR0IF==0);      // Check for overflow
T0CONbits.TMR0ON=0;               // Turn off timer
INTCONbits.TMR0IF==0;             // clear the Timre0 flag
}
```

B] *Design a PIC 18Fxxxx test board with facility of status indication on LED, Buzzer and lamp connected PIC 18Fxxxx through relay, write embedded C program* **(Refer Q.13 of Chapter - 6)** **[9]**

OR

Q.6 A) *Draw and explain port structure of PIC 18Fxxxx microcontroller with different registers in programming.* **(Refer Q.1 of Chapter - 6)** **[9]**

B) *Interface 2 lines, 16 characters LCD to PIC 18Fxxxx microcontroller, write embedded C program to display a message "HELLO" on LCD at second line.* **(Refer Q.7 (Note only change the name and number of charters accordingly with starting address line 2 is C0) of Chapter - 6)** **[9]**

Q.7 A) *Explain with block diagram I2C mode of MSSP structure in detail.* **(Refer Q.9 of Chapter - 7)** **[9]**

B) *State features of RTC and draw an interfacing diagram with PIC 18Fxxxx.* **(Refer Q.17 of Chapter - 7)** **[8]**

OR

Q.8 A) *Explain the use of BRG register for calculation for baud rate with UART receiver block diagram.* **(Refer Q.11(Calculation of BRGH) and Q.15 (USART receiver block diagram) of Chapter - 7)** **[9]**

B) *Draw an interfacing diagram of EEPROM with PIC 18Fxxxx using SPI protocol with initialization program.* **(Refer Q.19 of Chapter - 7)** **[8]**

END...