

Processor Architecture

(Code : 214451)

Semester IV - Information Technology (Savitribai Phule Pune University)

**Strictly as per the New Credit System Syllabus (2019 Course)
Savitribai Phule Pune University w.e.f. academic year 2020-2021**

Harish G. Narula

Formerly, Assistant Professor (Senior),
D. J. Sanghvi College of Engineering,
Mumbai. Maharashtra, India.

Khushboo Shah

Senior Solution Architect,
Department of Computer Engineering
Citus Tech, Mumbai.
Maharashtra, India.

For Multiple Choice Questions (MCQs) visit our website
https://techknowledgebooks.com/library/#student_downloads



 **TechKnowledge™**
Publications

PE108A Price ₹ 315/-



Processor Architecture

(Code : 214451)

Semester IV - Information Technology (Savitribai Phule Pune University)

**Strictly as per the New Credit System Syllabus (2019 Course)
Savitribai Phule Pune University w.e.f. academic year 2020-2021**

Harish G. Narula

Formerly, Assistant Professor (Senior),
D. J. Sanghvi College of Engineering,
Mumbai. Maharashtra, India.

Khushboo Shah

Senior Solution Architect,
Department of Computer Engineering
Citus Tech, Mumbai.
Maharashtra, India.

For Multiple Choice Questions (MCQs) visit our website
https://techknowledgebooks.com/library/#student_downloads



 **TechKnowledge**TM
Publications

PE108A Price ₹ 315/-



COURSE CONTENTS

Unit I : PIC Microcontroller Architecture (06 Hours)

Introduction : introduction to microcontroller, Brief history of microcontrollers, Difference between microprocessor and microcontroller, Criteria for selection of microcontroller,

PIC18FXXX : Features and architecture, comparison of PIC 18 series microcontrollers; PIC18F458/452 Pin out connection, Registers of PIC18F,

Program and data memory organization : The Program Counter and Programmable ROM space in the PIC, File register and Access bank, Bank switching in PIC18;

Addressing modes : Addressing modes with instruction example, Oscillator configurations, Reset operations, Brownout reset, Watchdog timer, Power down modes & Configuration registers.

(Refer Chapters 1, 2)

Unit II : PIC I/O Ports and Timer (06 Hours)

I/O Port: I/O Port structure with programming : I/O Port structure, I/O Port programming, I/O Bit manipulation Programming.

Timer/Counter : Registers used for Timer/Counter operation, Delay calculations, Programming of Timers using Embedded C.

(Refer Chapters 3, 4 and 5)

Unit III : PIC Interrupts & Interfacing-I (06 Hours)

PIC Interrupts : Interrupt Vs Polling, IVT, Steps in executing interrupt, Sources of interrupts; Enabling and disabling interrupts, Interrupt registers, Priority of interrupts,

Programming of : Timer using interrupts, External hardware interrupts, Serial communication interrupt;

Interfacing of LED, Interfacing 16X2 LCD (8 bits) and Key board (4 x 4 Matrix), Interfacing Relay & Buzzer.

(Refer Chapters 6 and 7)

Unit IV : PIC Interfacing-II (06 Hours)

CCP modes : Capture, Compare and PWM generation;

DC Motor speed control with CCP, Stepper motor interfacing with PIC,

Basics of Serial communication protocols : Study of RS232, I2C, SPI, UART, Serial communication programming using Embedded C.

(Refer Chapters 8 and 9)

Unit V : PIC Interfacing-III

(06 Hours)

Interfacing : Interfacing of ADC and DAC 0808 with PIC, Temperature sensor interfacing using ADC and I2C with PIC, Interfacing of RTC (DS1306) using I2C with PIC, Interfacing of EEPROM using SPI with PIC.,

(Refer Chapter 10)

Unit VI : Current Trends in Processor Architecture

(06 Hours)

ARM & RISC : ARM and RISC design philosophy, Introduction to ARM processor & its versions ARM 7, ARM 9, ARM 11, Features& advantages of ARM processor, Suitability of ARM processor in embedded applications, ARM 7 dataflow model, Programmers model. CPSR & SPSR registers, Modes of operation, Difference between PIC and ARM.

(Refer Chapter 11)





PIC Architecture

1.1 Introduction to Microprocessors

When we hear the word microprocessor, what comes to our mind is a small (i.e. micro) IC (integrated circuit) that can process data i.e. perform arithmetic and logical operations.

1.1.1 Some Basic Terms used in Microprocessors

1. Opcode

A binary code, that indicates the operation to be performed is called as an Opcode.

2. Operands

The data on which the operation is to be performed (as well as the result of an operation) are termed as operands.

3. Instruction

The combination of opcode and operands, that can be used to instruct a system, is called as an instruction.

4. Instruction set

A list of all the instructions that can be issued to a system, is called as instruction set of that system.

5. Program/subroutine/routine

A set of instructions written in a particular sequence, so as to implement a given task is called as a program. A subroutine in assembly language refers to function as of High level languages like C/C++. The term routine is used in a special case called as Interrupt Service Routine (ISR).

6. Bus

A group of lines, pins or signals having common function is termed as bus.

The functional grouping of signals results in,

- (a) Data bus i.e. signals used to carry data.
- (b) Address bus i.e. signals to address or select a memory or I/O location.
- (c) Control bus are signals to issue and receive control operations.

7. Register

It is a flip-flop based circuit used to store the data or as the name says, register the data into it. It is as good as a PIPO (Parallel In Parallel Out) register. These are present inside the processor to store data temporarily. These are few in count.

1.1.2 Microprocessor Characteristics and Functions

1.1.2(A) Characteristics of Microprocessor

The computing power of the microprocessor can be determined by certain characteristics listed as follows :

1. The processor size or number of bits

For example, processors are 8-bit, 16-bit, 32-bit, 64-bit etc. A processor is n-bit processor implies that,

- (a) Its ALU is n-bit, i.e. the ALU can perform n-bit operation simultaneously.
- (b) Its internal data bus and register size is n-bit.
- (c) Its external data bus is also n-bit (in most of the cases).

2. Processing capability

It depends on the number of instructions supported by a processor. It also depends on the different types of data supported for execution by an instruction like binary, BCD, ASCII etc.

3. Clock frequency

The different operations that a microprocessor does on the external bus are memory read, memory write, input read and output write. The internal operations involve transfer of data between storage, arithmetic operations, logical operations etc. All these operations are synchronized with the clock. The frequency of this clock decides the speed of the processor.



4. Width of the address bus

The address bus width decide the number memory locations and I/O locations it can access. For a microprocessor, access refers to reading and writing a location.

5. Interrupt capability

The number of hardware and software interrupts for a processor depends on the number of interrupt pins and number of interrupt instructions respectively. Different types of interrupts like vectored/non-vectored, maskable/non-maskable have their importance, we will study the same later in this chapter.

1.1.2(B) Functions of a Microprocessor

- **Microprocessor** is an IC that can fetch, decode and execute an instruction. Thus, microprocessor is a semiconductor chip, which can fetch or get the instruction from the memory.
- The instruction fetched is in binary form and hence, it decodes the instruction. After decoding, the operation as specified in the instruction is executed by the processor.

The basic functions of a microprocessor are :

1. Fetching instruction from the memory location pointed by the program counter. (Fetching means to bring)
2. Decoding the instruction fetched. Since the instruction is coded in binary form, it is to be decoded to find out the operation to be performed.
3. Executing the instruction. Once the instruction is decoded, the required operation is to be performed.

The various operations performed by the processor are :

- (a) Transferring a data from one location to another in the system. This could be transferring of data from memory to the processor, memory to the Input/Output device, from processor to memory etc.
- (b) Performing different arithmetic operations like addition, subtraction, increment, decrement etc.

- (c) Performing various logical operations like AND, OR, EXOR etc.

- (d) Performing branching operations.

There are instructions required to perform the above functions.

1.1.3 Block Diagram of a Microcomputer

A computer or a microcomputer mainly consists of three components, namely

1. Microprocessor or the CPU
2. Memory
3. Input/Output Devices

Besides the above three, it also consists of buses that connect the three components of the microcomputer.

Fig. 1.1.1 shows the block diagram of a Microcomputer.

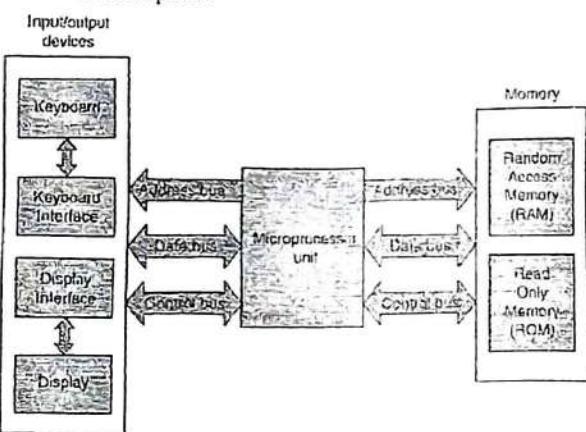


Fig. 1.1.1 : Block diagram of a microcomputer

1. Microprocessor Unit

- The microprocessor unit or the microprocessor contains Arithmetic and Logical Unit (ALU) as well as a Control Unit (CU).
- The ALU as the name says is responsible for all the arithmetic and logical operations done by the processor.
- The CU controls not only the components inside the microprocessor unit, but also outside the microprocessor in the system. It produces control signals to be issued to the memory as well as the Input/Output (I/O) devices.

2. Memory

- It mainly contains Random Access Memory (RAM) and Read Only Memory (ROM) as shown in the Fig. 1.1.1.
- RAM is normally used to store temporary data as RAM is volatile i.e. the data is lost as soon as the supply goes off. ROM is used to store permanent data as it retains data even after the system is switched off. Hard disk, CD are various examples of ROM.

3. Input / Output devices

- Each input device or output device normally works at a very slower speed than the microprocessor. Also each input or output device follows a different method of data transfer.
- Hence, for each input or output device a separate interface is required as shown in the Fig. 1.1.1. Keyboard interface to interface the keyboard to microprocessor unit and similarly display interface to interface display to the microprocessor unit.

4. Buses

- To connect the different components of a microcomputer, we need various connections.
- These connections having common functions when combined together form a bus.
- Thus, we have three busses namely data bus, address bus and the control bus.

1.1.4 Buses and Memory Accessing

1.1.4(A) Address Bus

- A set of lines that are used to select a memory or I/O location forms a bus (group of lines) called as the address bus.
- The address lines are used to select the location of memory or input/output device to be read or to be written on.
- The number of locations accessible by a processor depends on the number of address lines. Thus if there are 'n' lines, the processor can access 2^n memory locations.
- As processor selects the location it wants to access, the address bus is a unidirectional bus (indicated by the arrow).

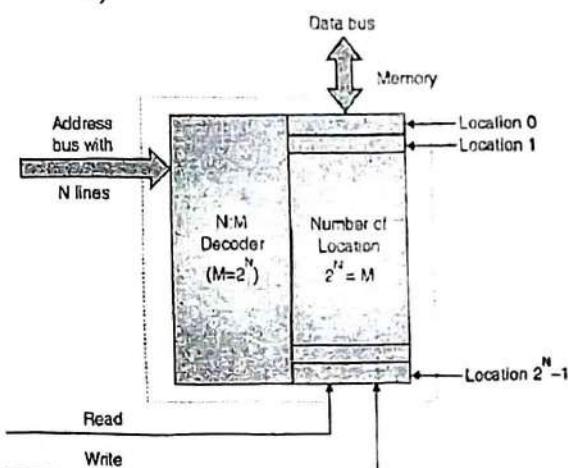


Fig. 1.1.2 : Structure of memory

- If the number of address lines, $N = 16$ then it can address $2^{16} = 65,536$ or 64K memory locations

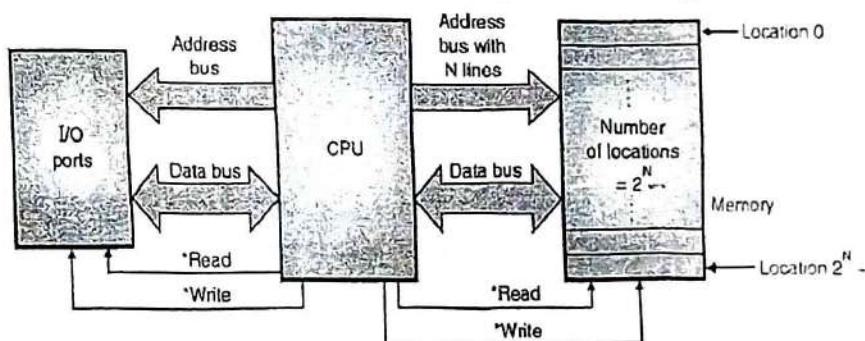


Fig. 1.1.3 : The three types of buses and their utility



$2^0 = 1$	$2^1 = 2$
$2^2 = 4$	$2^3 = 8$
$2^4 = 16$	$2^5 = 32$
$2^6 = 64$	$2^7 = 128$
$2^8 = 256$	$2^9 = 512$
$2^{10} = (1024) 1\text{ K (1 Kilo)}$	$2^{20} = 1\text{ K} \times 1\text{ K} = 1\text{ M (1 Mega)}$
$2^{30} = 1\text{ G (1 Giga)}$	$2^{40} = 1\text{ T (1 Tera)}$

Ex. 1.1.1 : A processor has 24 address lines, how many memory locations it can access.

Soln. :
$$\begin{aligned}2^{24} &= 2^4 \times 2^{20} = 16 \times 1\text{ M} \\&= 16\text{ MB}\end{aligned}$$

Ex. 1.1.2 : A processor has 4 GB memory, how many address lines are required to access this memory.

Soln. :
$$4\text{ GB} = 4 \times 1 = 2^2 \times 2^{30} = 2^{32}$$

Hence, 32 address lines are required.

1.1.4(B) Data Bus

- These lines are used to carry data between memory, processor and I/O devices.
- The size of data bus decides the number of bits that the processor can access simultaneously from memory or I/O device.
- The address lines select a location to be accessed (read/write), the data read or to be written is available on data bus.

1.1.4(C) Control Bus

- Since the same address lines are used for memory as well as I/O, there has to be some signalling that indicates the address is to access memory or I/O.
- Similarly, since same data bus is used to read or write data, some control signalling is required to indicate the operation to be carried out is read or write.
- Such signals are called as control signals. Some of the control bus signals are as follows :
 1. Memory read
 2. Memory write
 3. I/O read
 4. I/O write

Combined utilization of the three buses

- In practice the CPU needs to make use of all the buses in order to perform a desired operation.
- For example, if a byte (8 bits) is to be read from the memory location 18F8H, then the sequence of operations followed by the CPU is as follows :
- Task : Read data from memory location 18F8H

Step 1 : CPU sends out "18F8H" i.e. the address of the desired memory location on the 16-bit address bus.

Step 2 : CPU sends the memory read signal on the control bus.

Step 3 : The memory read signal enables the memory device to put the data stored in the location 18F8H onto the data bus. This data travels on the data bus from memory to CPU.

1.1.5 ALU (Arithmetic and Logic Unit)

- This unit is used to perform arithmetic operations (like add, subtract, multiply, divide etc) and logical operations like (AND, OR, EX-OR etc.).
- It consists of two inputs operands (4 bits in case of the ALU in Fig. 1.1.4) 'A' and 'B' and one output operand i.e. 'F'. 'S' are the select lines to select the operation (OPCODE). Besides these, there is also an input called as mode select for selecting Arithmetic or Logical operations.

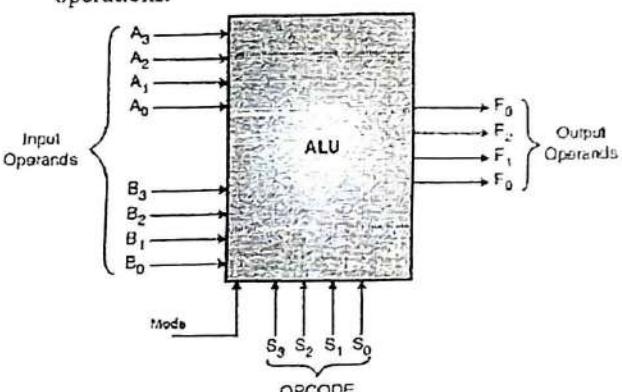


Fig. 1.1.4 : An ALU (Arithmetic logic unit)

- Now in the Fig. 1.1.4, to give an instruction to the system (ALU), the programmer has to give OPCODE and OPERAND i.e. the operation to be performed is to be given on the select lines while the data to be operated on is to be given on input operand lines.

- For example, if the operation is $5 + 7$ i.e. $(0\ 1\ 0\ 1)_2 + (0\ 1\ 1\ 1)_2$, the data lines of A are to be $(0\ 1\ 0\ 1)_2$, while that of B are to be $(0\ 1\ 1\ 1)_2$.
- A binary code for addition is to be given on select lines. Suppose $(0\ 0\ 1\ 1)_2$ refers to add operation for this ALU. So the connections are to be made by the programmer as shown in Fig. 1.1.5.

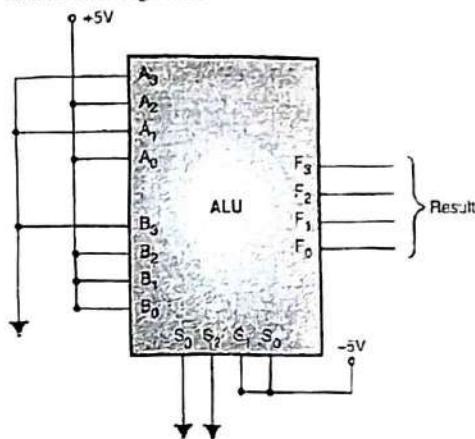


Fig. 1.1.5 : ALU connections to implement $5 + 7$

- To do these connections, the programmer may take few seconds, while the IC can perform the operation in microseconds.
- Hence, the time in which the ALU could have performed few million operations, it can perform only one operation.
- The reason for this delay is the slow speed of the programmer to enter the instruction.
- Hence if the instructions were stored in a memory and the processor accesses them, it would be much faster.
- Thus, one major additional aspect of a microprocessor over an ALU is that it can take instructions stored in memory, called as stored program concept.

1.1.6 Basic Important Terms

University Question

Q. What is an interrupt?

SPPU Dec. 12, 3 Marks

1. Interrupt

It is a mechanism by which an input/output device (Hardware interrupt) or an instruction (Software interrupt) can suspend the normal execution of the processor and get itself serviced.

2. Interrupt Service Routine (ISR)

A small program or a routine that when executed services, the corresponding interrupting source is called as an ISR.

3. Vectored/Non-vectored interrupt

If the ISR address or some information related to the ISR address of an interrupt is to be taken from the interrupting source itself, it is called as a non-vectored interrupt; else it is a vectored interrupt.

4. Maskable/Non-maskable interrupt

Interrupt that can be masked (disabled) and unmasked (enabled) by the programmer is called as maskable interrupt else it is a non maskable interrupt.

1.2 Introduction to Microcontroller

A microprocessor is a chip which is dependent on other chips for many functions and the most important of them to interface ports to the outside world data, while a microcontroller is as good as a single chip computer which has everything in-built.

1.2.1 Microprocessor

- It is a device that integrates a number of useful functions into a single IC package. Some functions are :
- Ability to execute a stored set of instructions to carry out user defined tasks.
- Ability to access external memory chips to read/write data from/to memory.
- Ability to interface with I/O devices.

1.2.2 Microcontroller

Microcontroller is a device which integrates a number of the components of a microprocessor system onto a single chip like CPU, Memory, I/O modules etc. It only needs to be supplied with power and clock source. Thus a microcontroller combines on the same chip :

- The CPU core
- I/O
- Memory
- The Fig. 1.2.1 shows the block diagram of a microcontroller.

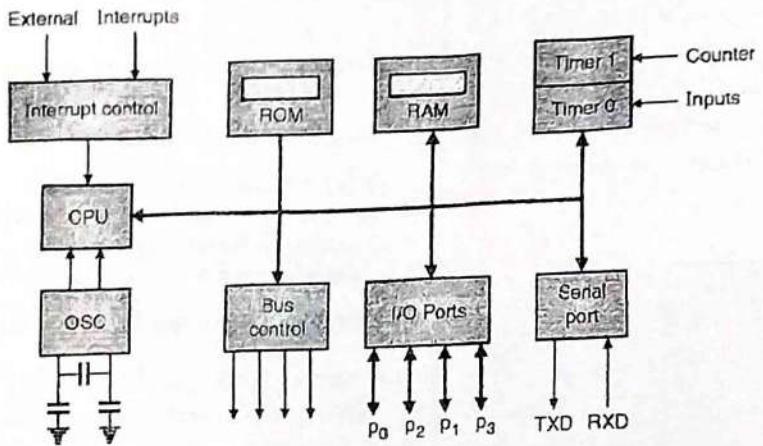


Fig. 1.2.1 : Block diagram of a microcontroller

1.2.3 Comparison of Microprocessors and Microcontrollers

University Questions

Q. Compare microprocessor and microcontroller.

SPPU - May 12, May 19, 6 Marks

Q. Explain clearly the differences between microcontroller and microprocessor.

SPPU - May 18, 6 Marks

Sr. No.	Microprocessor	Microcontroller
1.	Microprocessor is suited to processing information in computer systems.	Microcontroller is suited to control of I/O devices requiring a minimum component count.
2.	Microprocessor is processing intensive, has powerful addressing modes, instructions to perform complex operations and manipulate large volumes of data.	Processing capability of Microcontrollers never approaches those of Microprocessors. Microcontroller is to control of inputs and outputs. Hence instructions are to set/clear bits, Boolean operations (AND, OR, XOR, NOT, jump if a bit is set/cleared), etc. It has extremely compact instructions, many implemented in one byte.
3.	Microprocessor usually requires external circuitry to interface I/O devices.	Microcontroller has built-in I/O control module, event timing, counting etc.
4.	Microprocessor has very wide bus widths. It has large memory address spaces (>4 Gbytes) and lots of data (Data bus : 32, 64, 128 bits wide)	Microcontroller has narrow buses. It has relatively small memory address spaces (typically Kbytes) and less data (Data bus typically 8, 16 bits wide)
5.	Microprocessor are very fast (> 1 GHz)	Microcontroller are relatively slow (typically 10-20 MHz) since most I/O devices being controlled are relatively slow.
6.	Microprocessors are expensive.	Microcontrollers are cheap.

1.2.4 Advantages of Microcontrollers over Microprocessors

The advantages of microcontrollers like 8051 over microprocessors are :

1. Microcontrollers have on-chip memory i.e. ROM as well as RAM. ROM is used to store programs and RAM to store data.
2. Microcontrollers have on-chip I/O controllers. The I/O devices can be directly connected to these I/O pins. Devices like switches, displays, LEDs etc can be connected on I/O pins of microcontroller.
3. Microcontrollers have on-chip timer/counter. Hence to keep track of time or to count a particular event, no external chip is required.
4. Microcontrollers also have on-chip interrupt controller, and hence do not require any external interrupt controller. This makes the system compact as all the required operations are implemented on a single chip.

1.2.5 Disadvantages/Limitations of 8 bit Microcontrollers over Microprocessors

The disadvantages of microcontrollers like 8051 over microprocessors are :

1. Microcontrollers have slower speed compared to microprocessors.
2. Microcontrollers have less complex instructions compared to microprocessors. Hence to perform certain arithmetic operations, we need multiple instructions on a microcontroller rather than single instruction on a microprocessor.
3. The bus size of microcontrollers are also normally narrower compared to that of microprocessors.

1.2.6 Applications of Microcontrollers

The applications of microcontrollers are countless. Almost every electronic appliance today has a microcontroller in it to control the appliance. All electronic gadgets have microcontrollers in them to control. Let us list few of these applications :

1. Many home appliances like Microwave oven, washing machine, Television (TV), Air Conditioner (AC) etc. have microcontrollers in them to control their operation.

2. Most of the electronic gadgets like mobile phones, cameras, tablets, health band etc have microcontrollers.
3. Electronic weighing scale, digital thermometer and many such instruments have microcontrollers embedded in them.
4. It has a wide application in automation systems like home automation and home security system.
5. Microcontrollers also have their application in Robotics, automated car etc.
6. Medical instruments like ECG, lung function test, X-Ray machine etc. also use microcontrollers.

1.3 Introduction to PIC

- PIC microcontrollers are developed by Microchip Technology Inc.
- PIC 18FXXX belongs to a class of 8-bit microcontrollers of RISC (Reduced Instruction Set Computing) architecture.
- 8086 and 8051 have a CISC (Complex Instruction Set Computing) architecture RISC, 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. memory access or ALU operation, etc. For e.g. MOV AX,[BX] of 8086 is simple instruction
- Complex instructions are those that perform multiple operations i.e. one instruction accesses memory as well as performs ALU operation. For e.g. ADD AX,[BX] of 8086 is complex instruction.
- Since in RISC the numbers 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 micro programmed control unit.



1.4 Comparison of CISC and RISC Processors

University Question:

- Q. Compare RISC and CISC architectures.
- SPPU - May 15, 6 Marks; Dec. 15, 6 Marks; May 16, Oct. 16 (In Sem.), 4/6 Marks Dec. 16 Aug. 17 (In Sem.) 6/7 Marks, May 19, 6 Marks**

Sr. No.	Properties	RISC	CISC
1.	Number of Instructions	Less	More
2.	Addressing Modes	Less	More
3.	Instruction Formats	Less	More
4.	Instruction Size	Fixed	Variable
5.	Control Unit	Hardwired	Micro-programmed
6.	Number of Bus Cycles to execute an instruction	Single CPU cycle (for 80% Instructions)	Multiple CPU cycles
7.	Control Logic and Decoding Subsystem	Simple	Complex
8.	Pipelining	Large number of stages of Pipelining	Difficulty in efficient implementation
9.	Design time and Probability of Design Errors	Smaller time and less probable	Long time and significant probability
10.	Complexity of Compiler	Simpler	More complex and the results of "optimization" may not be most efficient and the fastest machine language code
11.	HLL Instructions	Supported	Not Supported

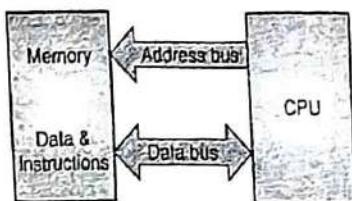
1.5 Harvard and Von Neumann Architectures

University Question:

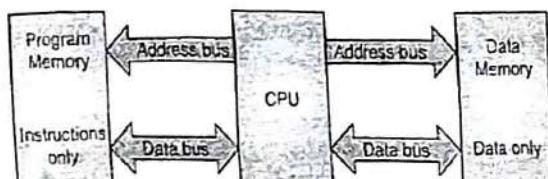
- Q. Compare Harvard and Von Neumann architecture.
- SPPU - Dec. 14, 7 Marks**

Sr. No.	Von-Neumann architecture	Harvard architecture
1.	Please refer Fig. 1.5.1(a)	Please refer Fig. 1.5.1(b)
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.
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.

- Von Neumann memory organization states that "Every memory location has a unique address, while the Harvard's memory organization states that "There can be multiple memory locations with same address but different function".
- One such example of Von Neumann and Harvard's memory organization is shown in the Fig. 1.5.1. Von Neumann's memory organization has all code, data and stack in a single memory, while the Harvard's memory organization has them in separate memories.
- Most of the microprocessors use Von Neumann's memory organization while microcontrollers use Harvard's memory organization.



(a) Von Neumann memory organization



(b) Harvard memory organization

Fig. 1.5.1

1.5.1 Harvard Architecture of PIC Microcontroller

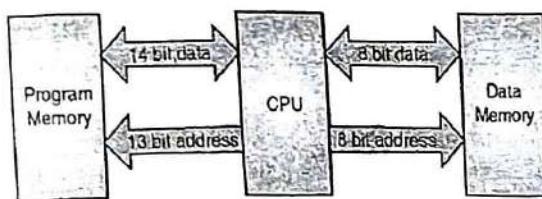


Fig. 1.5.2 : Block diagram for Harvard architecture of PIC microcontrollers

- Like 8051, PIC microcontrollers also 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 1.5.2 shows block diagram for Harvard architecture in PIC controllers.

- PIC uses 14 bits for instructions which allows for all instructions to be one word Instructions.
- Instructions are fetched from program memory using buses that are different from the ones that are used to access the data memory.
- 14-bit wide data bus for program memory fetches an entire instruction in a single access (as the size of all instructions in PIC is 14 bits each).

1.6 History of Microcontroller : Overview of PIC Microcontroller Family 12FXX, 16FXX and 18FXX

- The PIC16CXX is a family of low-cost, high-performance, CMOS, fully-static, 8-bit microcontrollers.
- All PIC16/17 series microcontrollers employ an advanced RISC architecture.
- The PIC16CXX microcontroller family has enhanced core features, eight-level deep stack, and multiple internal and external interrupt sources.
- The separate instruction and data buses of the Harvard architecture allow a 14-bit wide instruction word with separate 8-bit wide data.
- The two stage instruction pipeline allows all instructions to execute in a single cycle, except for program branches (which require two cycles).
- A total of 35 instructions (reduced instruction set) are available.
- Additionally, a large register set gives some of the architectural innovations used to achieve a very high performance.
- PIC16CXX microcontrollers typically achieve a 2:1 code compression and a 4:1 speed improvement over other 8-bit microcontrollers in their class.
- The PIC16C64/64A/R64 devices have 128 bytes of RAM and 33 I/O pins. In addition, several peripheral features are available, including: three timer/counters, one Capture/Compare/PWM module and one serial port.
- The Synchronous Serial Port can be configured as either a 3-wire Serial Peripheral Interface (SPI) or the two-wire Inter-Integrated Circuit (I^2C) bus. An 8-bit Parallel Slave Port is also provided.



- The PIC16C6X device family has special features to reduce external components, thus reducing cost, enhancing system reliability and reducing power consumption.
- There are four oscillator options, of which the single pin RC oscillator provides a low-cost solution, the LP oscillator minimizes power consumption, XT is a standard crystal, and the HS is for High Speed crystals.
- The SLEEP (power-down) mode offers a power saving mode. The user can wake the chip from SLEEP through several external and internal interrupts, and resets.
- A highly reliable Watchdog Timer with its own on-chip RC oscillator provides protection against software lockup.
- A UV erasable CERDIP packaged version is ideal for code development, while the cost-effective One-Time-Programmable (OTP) version is suitable for production in any volume.
- The PIC16C6X family fits perfectly in applications ranging from high-speed automotive and appliance control to low-power remote sensors, keyboards and telecom processors.
- The EPROM technology makes customization of application programs (transmitter codes, motor speeds, receiver frequencies, etc.) extremely fast and convenient.
- The small footprint packages make this microcontroller series perfect for all applications with space limitations. Low-cost, low-power, high performance, ease-of-use, and I/O flexibility make the PIC16C6X very versatile even in areas where no microcontroller use has been considered before (e.g. timer functions, serial communication, capture and compare, PWM functions, and co-processor applications).

1.6.1 PIC Microcontroller Features

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.
When the PIC microcontroller is operated at its maximum clock rate, each instruction can be executed with 0.2 μ s.
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 up to 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 4V 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. The PIC16C6X and 16C7X microcontroller operating range is 3 - 6 V. PIC 16C6X takes current less than 2 mA at 5 V at 4 MHz oscillator frequency. The PIC 16C7X takes 15 μ A at 3V and 32 KHz oscillator frequency.

1.7 Comparison of Features of PIC10, PIC12, PIC16, PIC18 Families

Microchip has introduced six different lines of 8-bit microcontrollers, namely PIC10XXX, PIC12XXX, PIC16C5X, PIC16CXX and PIC18XXXX. Each of these PIC microcontrollers have variations in number of instructions, instruction formats and different peripheral functions. This makes products designed with a different family of PIC microcontrollers incompatible. The Table 1.7.1 shows the comparison chart of these families of processors.

Table 1.7.1 : Comparison of PIC Family

Features	Baseline Architecture	Mid-Range Architecture	Enhanced Mid-Range Architecture	PIC 2B Architecture
Families	PIC10, PIC12, PIC16	PIC12, PIC16	PIC12FXXX PIC16F1XX	PIC18
Pin Count	6-40	8-64	8-64	18-100
Interrupts	No	Single interrupt capability	Single interrupt capability with hardware context save	Multiple interrupt capability with hardware context save
Performance	5 MIPS	5 MIPS	8 MIPS	Up to 16 MIPS
Instructions	33, 12-bit	35, 14-bit	49, 14-bit	83, 16-bit
Program Memory	Up to 3 KB	Up to 14 KB	Up to 28 KB	Up to 128 KB
Data Memory	Up to 138 Bytes	Up to 368Bytes	Up to 1.5 KB	Up to 4 KB
Hardware Stack	2 level	8 level	16 level	32 level
Key Features	<ul style="list-style-type: none"> • Comparator • 8-bit ADC • Data Memory • Internal Oscillator 	In addition to Baseline <ul style="list-style-type: none"> • SPI / I²CTM • UART • PWMs • LCD • 10-bit ADC • Op Amp 	In addition to Mid-Range <ul style="list-style-type: none"> • Multiple Communication Peripherals • Linear Programming Space • PWMs with Independent Time Base 	In addition to Enhanced Mid-Range <ul style="list-style-type: none"> • 8 × 8 Hardware Multiplier • CAN • CTMU • USB • Ethernet • 12-bit ADC
Highlights	Lowest cost in the smallest form factor	Optimal cost to performance ratio	Cost effective with more performance and memory	High performance, optimized for C programming advanced peripherals
Total Number of Devices	16	58	29	193



1.8 PIC18F458 Features

All the processors of PIC18 family have the same instruction set and the peripheral function design. It has some 16-bit instructions while some 32-bit. The features of 18F458 are listed below:

1. Linear program memory addressing up to 2 Mbytes.
2. Linear data memory addressing to 4 Kbytes.
3. Flash Memory of 32KB and 16K single word instructions.
4. 1536 bytes of SRAM while 256 bytes of EPROM for data storage.
5. Up to 10 MIPS operation.
6. DC - 40 MHz clock input.
7. 4 MHz-10 MHz oscillator/clock input with PLL active.
8. 16-bit wide instructions, 8-bit wide data path.
9. Priority levels for interrupts.
10. 8x8 Single-Cycle Hardware Multiplier.
11. High current sink/source 25 mA/25 mA.
12. Three external interrupt pins.
13. Four Timers
 - (i) **Timer0 module:** 8-bit/16-bit timer/counter with 8-bit programmable prescaler.
 - (ii) **Timer1 module:** 16 bit timer/counter
 - (iii) **Timer2 module:** 8-bit timer/counter with 8-bit period register (time base for PWM)
 - (iv) **Timer3 module:** 16-bit timer/counter
Secondary oscillator clock option - Timer1/Timer3
14. Capture/Compare/PWM (CCP) modules; CCP pins can be configured as:
 - (i) **Capture input:** 16-bit, max resolution 6.25 ns
 - (ii) **Compare:** 16-bit, max resolution 100 ns (T_{CY})
 - (iii) **PWM output:** PWM resolution is 1 to 10-bit
 - (iv) **Max. PWM freq. @:** 8-bit resolution = 156 kHz
10-bit resolution = 39 kHz

15. Enhanced CCP module which has all the features of the standard CCP module, but also has the following features for advanced motor control:
 - (i) 1, 2 or 4 PWM outputs
 - (ii) Selectable PWM polarity
 - (iii) Programmable PWM dead time
16. Master Synchronous Serial Port (MSSP) with two modes of operation:
 - o 3-wire SPI (Supports all 4 SPI modes)
 - o I²C Master and Slave mode
17. Addressable USART module:
Supports interrupt-on-address bit
18. 10-bit, up to 8-channel Analog-to-Digital Converter module (A/D) with:
 - o Conversion available during Sleep
 - o 8 channels available
19. Analog Comparator module:
Programmable input and output multiplexing
20. Comparator Voltage Reference module
21. Programmable Low-Voltage Detection (LVD) module:
Supports interrupt-on-Low-Voltage Detection
22. Programmable Brown-out Reset (BOR), Power-on Reset (POR), Power-up Timer (PWRT) and Oscillator Start-up Timer (OST)
23. Watchdog Timer (WDT) with its own on-chip RC oscillator
24. Programmable code protection
25. Power-saving sleep mode
26. Selectable oscillator options, including:
 - o 4x Phase Lock Loop (PLL) of primary oscillator
 - o Secondary Oscillator (32 kHz) clock input
27. In-Circuit Serial Programming (ICSP) via two pins
28. CAN 2.0B port complies with ISO CAN conformance test
29. Low-power, high-speed Enhanced Flash technology
30. Wide operating voltage range (2.0V to 5.5V)

The Table 1.8.1 gives a comparison of the features of the PIC18FXX8 family.

Table 1.8.1: PIC18FXX8 Device Features

Features		PIC18F248	PIC18F258	PIC18F448	PIC18F458
Operating Frequency		DC-40 MHz	DC-40 MHz	DC-40 MHz	DC-40 MHz
Internal Program Memory	Bytes	16K	32K	16K	32K
	# of Single-Word Instructions	8192	16384	8192	16384
Data Memory (Bytes)		768	1536	768	1536
Data EEPROM Memory (Bytes)		256	256	256	256
Interrupt Sources		17	17	21	21
I/O Ports		Ports A, B, C	Ports A, B, C	Ports A, B, C, D, E	Ports A, B, C, D, E
Timers		4	4	4	4
Capture/Compare/PWM Modules		1	1	1	1
Enhanced Capture / Compare/PWM Modules		-	-	1	1
Serial Communications		MSSP, CAN, Addressable USART	MSSP, CAN, Addressable USART	MSSP, CAN Addressable USART	MSSP, CAN Addressable USART
Parallel Communications (PSP)		No	No	Yes	Yes
10-bit Analog-to-Digital Converter		5 Input channels	5 Input channels	8 Input channels	8 Input channels
Analog Comparators		No	No	2	2
Analog Comparators V _{REF} Output		N/A	N/A	Yes	Yes
Resets (and Delays)		POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT/OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)	POR, BOR, RESET Instruction, Stack Full, Stack Underflow (PWRT, OST)
Programmable Low-Voltage Detect		Yes	Yes	Yes	Yes
Programmable Brown-out Reset		Yes	Yes	Yes	Yes
CAN Module		Yes	Yes	Yes	Yes
In-circuit Serial programming™ (ICSP™)		Yes	Yes	Yes	Yes
Instruction Set		75 Instructions	75 Instructions	75 Instructions	75 Instructions
Packages		28-pin SPDIP 28-pin SOIC	28-pins SPDIP 28-pin SOIC	40-pin PDIP 44-pin PLCC 44 pin TQFP	40-pin PDIP 44-pin PLCC 44 pin TQFP



1.9 Pin Diagram of PIC18F458

Fig. 1.9.1 shows the pin diagram of PIC18F458. The functions of the pins are related to the block to which they are connected. We will understand these functionalities in the next section.

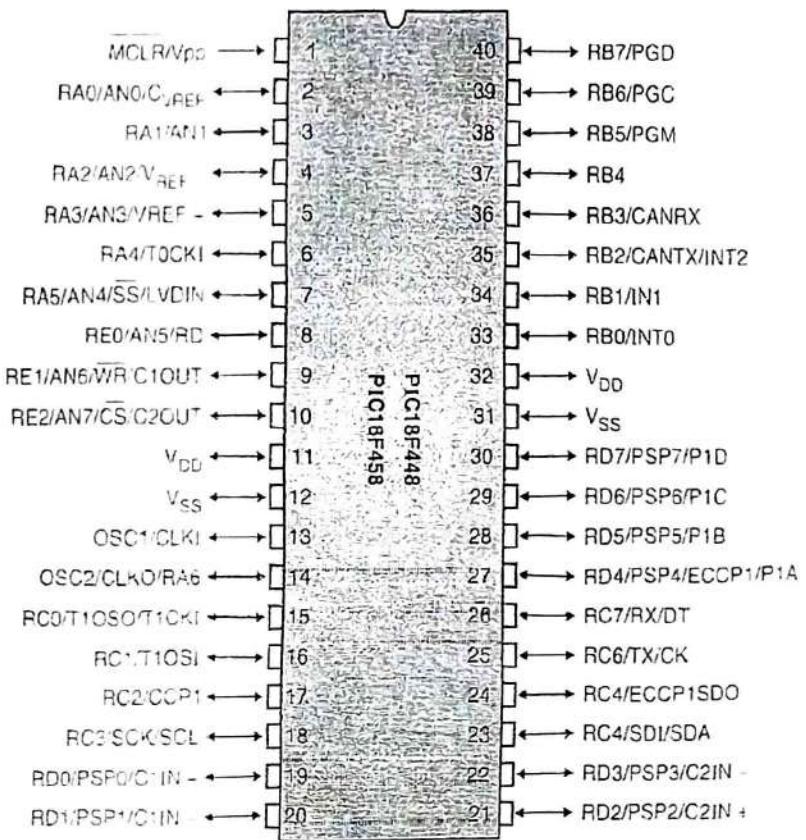


Fig. 1.9.1 : Pin Diagram of PIC18F458

1.10 Architecture of PIC18F458 (Block Diagram)

University Questions

Q. Explain status register of PIC18F458.

SPPU - Dec. 14, Aug. 15(In Sem.), Oct. 16(In Sem.), 5/6 Marks

Q. Explain the status register of PIC 18 microcontroller.

SPPU - Dec. 14, 6 Marks

Q. Explain the status register of PIC 18F458.

SPPU - May 16, 6 Marks

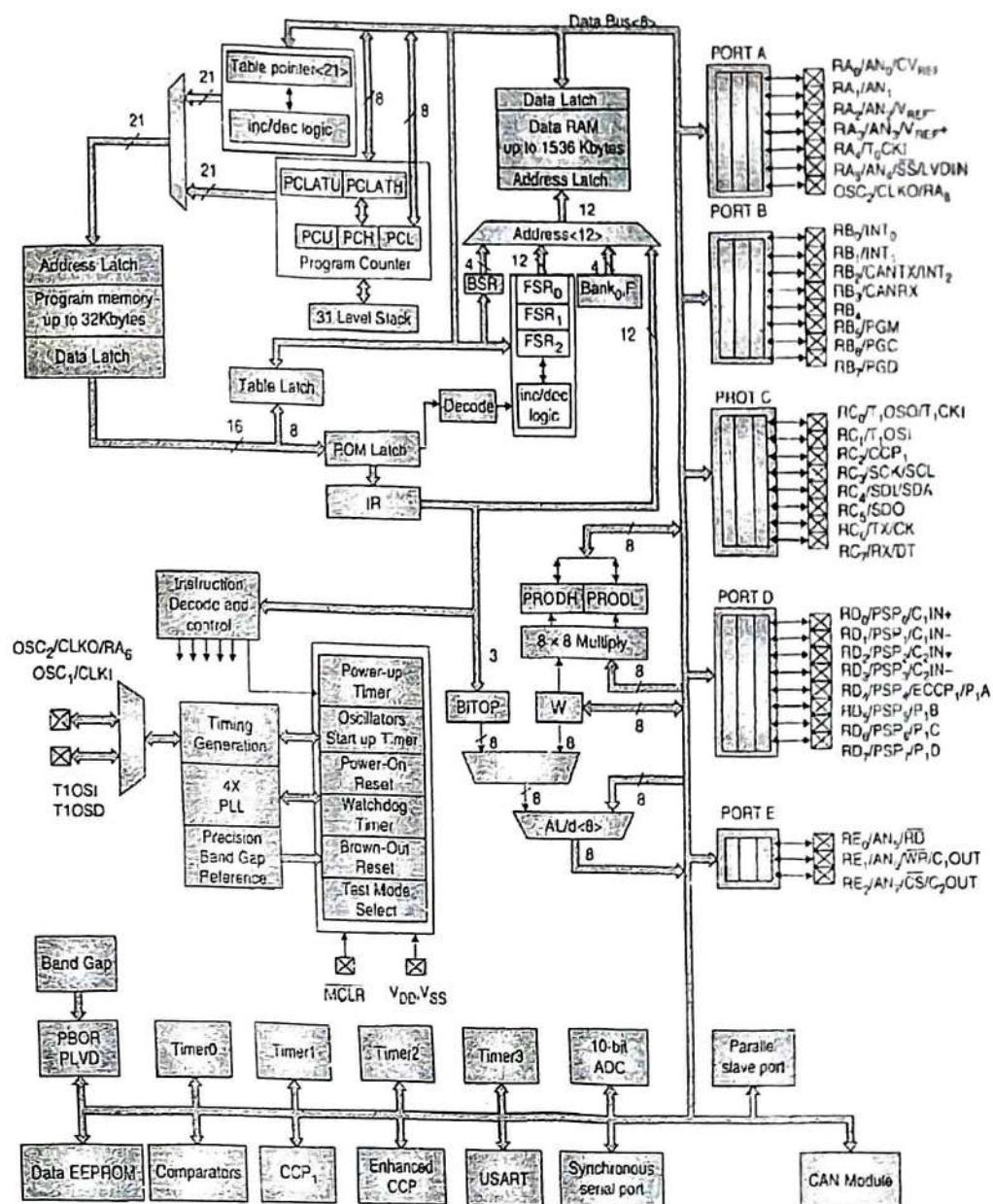


Fig. 1.10.1 : Block diagram of PIC18F458

- The block diagram of the PIC18 family member PIC18F458 is shown in Fig. 1.10.1.
- Fig. 1.10.1 shows the peripheral functions available in the PIC18F458. These will also be discussed in details in the subsequent sections and chapters.
- PIC18F458 has internal 8-bit data bus connecting all the peripherals as shown in the Fig. 1.10.1. Let us discuss each of these blocks along with their associated pins.

1.10.1 Program Memory

University Question

Q. Explain the program memory map for PIC18F458.

SPPU - Aug. 15 (In Sem.), 3 Marks

The program memory provides the instructions. This program memory is also connected to the instruction register through the ROM latch like system interface data bus. The address is again provided by the table pointer or the Program Counter latch. The table pointer also gets the address from the program counter.

Bit 2: Z (Zero bit) Indication of a zero result

This bit is set when the result of an executed arithmetic or logic operation is zero.

1 = Result equals zero

0 = Result does not equal zero

Bit 1: DC (Digit Carry) DC Transfer

This bit is affected by the operations like addition, subtraction. Unlike C bit, this bit represents transfer from the fourth resulting place. It is set in case of subtracting smaller from greater number and is reset in the other case.

1 = transfer occurred on the fourth bit according to the order of the result

0 = transfer did not occur

DC bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

Bit 0: C (Carry) Transfer

Bit that is affected by operations of addition, subtraction and shifting.

1 = transfer occurred from the highest resulting bit

0 = transfer did not occur

C bit is affected by ADDWF, ADDLW, SUBLW, SUBWF instructions.

- The status register contains arithmetic status of the ALU, the Reset status and bank select bits for data memory.
- It can be used as a destination register for any instruction just like any other register.

1.10.5(C) Program Counter and PCLATH**University Questions**

Q. Write short note on program counter.

SPPU - Dec. 14, 2 Marks

Q. Explain the function of Program counter.

SPPU - Aug. 17 (In Sem.), 4 Marks

- The Program Counter (PC) is a 21-bit register that contains the address of the instruction being executed.
- It is physically carried out as a combination of a 5-bit register PCLATU for the five higher bits of the address, and the 8-bit registers each PCLATH and PCL for the lower 16-bits of the address.
- By its incrementing or change (i.e. in case of jumps) microcontroller executes program instructions step-by-step.

- Register PCL is readable as well as writable directly.
- PCH is not readable. PCH register is updated through the PCLATH
- The five PCLATU bits are not readable. They are indirectly writable through the PCLATH register.
- The upper bits of PC are cleared upon reset.

1.10.5(D) Stack**University Questions**

Q. Explain Stack organization of PIC 18f458 microcontroller.

SPPU - Aug. 17 (In Sem.), 6 Marks

Q. Explain stack memory organisation and stack pointer in detail.

SPPU - Dec. 19, 6 Marks

- PIC18CXX8 has a 21-bit stack with 31 levels, or in other words, a group of 31 memory locations, 21 bits wide, with special purpose.
- Its basic role is to keep the value of program counter after a jump from the main program to an address of a subprogram.
- In order for a program to know how to go back to the point where it started from, it has to return the value of a program counter from a stack. When moving from a program to a subprogram, program counter is being pushed onto a stack (example of this is CALL instruction).
- When executing instructions such as RETURN, RETLW or RETFIE which were executed at the end of a subprogram, program counter will be taken from a stack so that program will continue where was stopped before it was interrupted. These operations of placing on and taking off from a program counter stack are called PUSH and POP, and are named according to similar instructions on some bigger microcontrollers.

1.10.5(E) INDF and FSR

- INDF (Indirect through FSR) register is not a physical register.
- Indirect addressing is possible with the help of this register.
- An instruction that uses the INDF register accesses the register pointed to by the file select register.
- Reading the INDF register will produce 00H.
- Writing to the INDF register results in no-operation. The status bits are affected.



1.10.6 8 × 8 Multiply Unit

There is a separate 8×8 Multiply unit and is associated with a PRODH (Product High) and PRODL (Product Low) registers to store the result. An 8×8 hardware multiplier is included in the ALU of the PIC18F458 device. By making the multiply a hardware operation, it completes this multiplication in a single instruction cycle. This is an unsigned multiply that gives a 16-bit result, stored in the 16-bit product register pair (PRODH:PRODL). The multiplier does not affect any flags in the ALU STATUS register. Making the 8×8 multiplier execute in a single cycle gives the following advantages:

1. Higher computational throughput
2. Reduces code size requirements for multiply algorithms

The performance increase allows the device to be used in applications previously reserved for Digital Signal Processors (DSP).

1.10.7 Five I/O (Ports A, B, C, D, E)

There are five I/O (Ports A, B, C, D, E) ports available on PIC18F458 device. Some of their pins are multiplexed with one or more alternate functions from the other peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin. Each port has three registers for its operation. These registers are:

1. TRIS register (data direction register)
2. PORT register (reads the levels on the pins of the device)
3. LAT register (output latch): The Data Latch (LAT register) is useful for read-modify-write operations on the value that the I/O pins are driving.

We will discuss about each of these ports later

1.10.8 Four Timers (Timer 0 to Timer 3)

There are four timers (Timer0 to Timer3) of 8-16 bits with different features in each of them like support for Compare, Capture and PWM (CCP). There are two such CCP ports namely CCP1 and ECCP.

The Timer0 module has the following features :

1. Software selectable as an 8-bit or 16-bit timer/counter.
2. Readable and writable.

3. Dedicated 8-bit software programmable prescaler (input clock pulse prescaling).
4. Clock source selectable to 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.

The Timer 1 module timer/counter has the following features:

1. 16-bit timer/counter (two 8-bit registers: TMR1H and TMR1L).
2. Readable and writable (both registers).
3. Internal or external clock select.
4. Interrupt-on-overflow from FFFF H to 0000 H.
5. Reset from CCP module special event trigger.

The Timer 2 module timer has the following features:

1. 8-bit timer (TMR2 register).
2. 8-bit period register (PR2).
3. Readable and writable (both registers).
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 optional use of TMR2 output to generate clock shift.

The Timer 3 module timer/counter has the following features:

1. 16-bit timer/counter (two 8-bit registers; TMR3H and TMR3L)
2. Readable and writable (both registers)
3. Internal or external clock select
4. Interrupt-on-overflow from FFFFH to 0000H
5. Reset from CCP module trigger.

1.10.8(A) Applications of Timer in PIC18F

Capture mode

In Capture mode, CCP1H:CCP1L captures the 16-bit value of the TMR1 or TMR3 registers when an event occurs on the pin RC2/CCP1.

An event is defined as either of the following:

1. Falling edge
2. Rising edge

3. 4th rising edge
4. 16th rising edge

Compare mode

In Compare mode, the 16-bit CCPR1 register value is constantly compared against either the TMR1 register pair value or the TMR3 register pair value. When a match occurs, the CCP1 pin :

- is driven High
- is driven Low
- toggles output (high-to-low or low-to-high) or
- remains unchanged

PWM mode

- In Pulse Width Modulation (PWM) mode, the CCP1 pin produces up to a 10-bit resolution PWM output. Since the CCP1 pin is multiplexed with the PORTC data latch, the TRISC<2> bit must be cleared to make the CCP1 pin an output. Let us understand what PWM is and how it works.
- A PWM output has a fixed period and a varying time for which the output stays high as shown in Fig. 1.10.4. The frequency of the PWM is the inverse of the period (1/period). The duty cycle is given by the formula :

$$\text{Duty cycle} = \frac{T_{on}}{T_{on} + T_{off}}$$

- The timer 2 or timer 4 value decides the T_{on} value. Hence the Timer 2 or Timer 4 value decides the duty cycle. Since the period is same, the change in duty cycle causes a change in the average DC value of the signal.
- This average value of the signal can be taken as the analog equivalent of the digital value in the timer 2 or 4. Hence many a times, PWM is also referred to as a DAC (Digital to Analog Convertor).

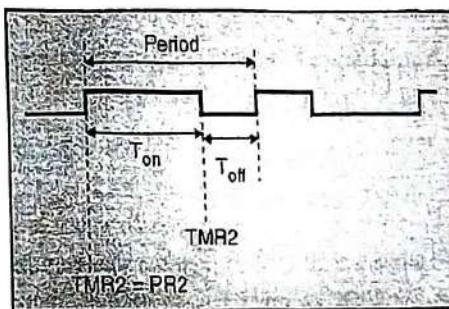


Fig. 1.10.4 : PWM output

1.10.9 Analog to Digital Convertor (ADC)

There is also a 10-bit Analog to Digital Convertor (ADC). Hence, the resolution of 1024 levels (since, 10-bit ADC) module allows conversion of an analog input signal to a corresponding 10-bit digital number. There are three control registers to control various operations of the ADC like selecting one of the 16 input channels, enabling or disabling the ADC, reference voltage source etc. There are two registers to hold the lower 8-bit and the upper 2-bit of the digital equivalent. The start of ADC can also be triggered by the CCP module discussed earlier.

1.10.10 Analog Comparator

- PIC18FXX8 also has two Analog Comparators. The comparator can compare two analog voltages and hence a decision can accordingly be taken.
- The comparator module contains two analog comparators. The inputs to the comparators are multiplexed with the pins RF1 to RF6. A single comparator is shown in Fig. 1.10.5, along with the relationship between the analog input levels and the digital output. When the analog input at V_{IN+} is less than the analog input V_{IN-} , the output of the comparator is a digital low level. When the analog input at V_{IN+} is greater than the analog input V_{IN-} , the output of the comparator is a digital high level! The shaded areas of the output of the comparator in Fig. 1.10.5 represent the uncertainty, due to input offsets and response time.

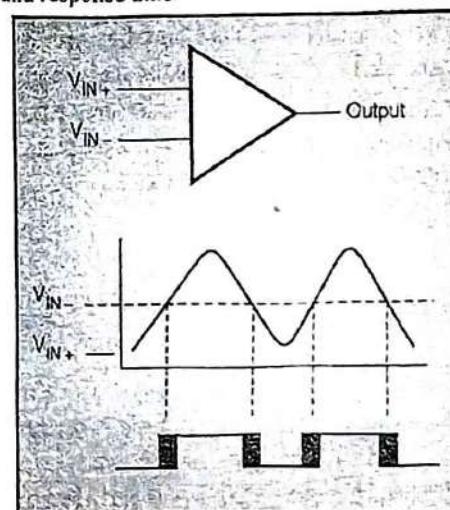


Fig. 1.10.5 : Single comparator



1.10.11 Master Synchronous Serial Port (MSSP)

The Master Synchronous Serial Port (MSSP) module is a serial interface, useful for communicating with other peripheral or microcontrollers. 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 (I^2C)
 - (a) Full Master (Master or Multi master) mode
 - (b) Slave mode (with general address call)

1.10.11(A) Serial Peripheral Interface (SPI) Mode

The SPI mode is a full duplex mode i.e. it allows 8 bits of data to be synchronously transmitted and received simultaneously. All four modes of SPI are supported. To accomplish communication, typically three pins are used:

1. Serial Data Out (SDO) - RC5/SDO
2. Serial Data In (SDI) - RC4/SDI/SDA
3. Serial Clock (SCK) - RC3/SCK/SCL

Additionally, a fourth pin may be used when in a Slave mode of operation: Slave Select (SS) - RF7/SS.

1.10.11(B) I^2C (Inter Integrated Circuit) Mode

This mode is mainly used for communication between the two microcontrollers and hence the name Inter IC communication. The MSSP module in I^2C mode fully implements all master and slave functions and provides interrupts on Start and Stop bits in hardware to determine a free bus (in case of 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:

1. Serial clock (SCL) - RC3/SCK/SCL
2. Serial data (SDA) - RC4/SDI/SDA

1.10.12 USART Ports

The Universal Synchronous Asynchronous Receiver Transmitter (USART) module (also known as a Serial Communications Interface or SCI) is one of the two types of serial I/O modules available on PIC18FXX8. It has two USARTs, which can be configured independently of each other.

Each can be configured as a full-duplex asynchronous system that can communicate with peripheral devices, such as CRT terminals and personal computers, or 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:

1. Asynchronous (full-duplex)
2. Synchronous - Master (half-duplex)
3. Synchronous - Slave (half-duplex)

The pins of USART1 and USART2 are multiplexed with the functions of PORTC (RC6/TX1/CK1 and RC7/RX1/DT1) and PORTG (RG1/TX2/CK2 and RG2/RX2/DT2), respectively.

1.10.13 Interrupt and its Handling in PIC18F

University Question

Q. Write short note on interrupt structure of PIC18F458 microcontroller.

SPPU - Aug. 15 (In Sem.), 4 Marks

- PIC18F has multiple interrupts and a special feature that allows different vector location allocation for different interrupts.
- The vector address of high priority interrupt is at 0008H, while that of low priority interrupt is at 0018 H. High priority interrupts events override any low priority interrupts.
- There are many registers that are involved in controlling interrupt operation. Each interrupt source has three different bits to control its operation, namely:
 1. Flag bit that indicates occurrence of an interrupt.
 2. Enable bit that indicates whether the interrupt is enabled or now. A '1' indicates that the interrupt is enabled.
 3. Priority bit to indicate whether the interrupt is in high priority block or low priority block.
- Once the interrupt priority is enabled, there are two bits which enable interrupts globally.
- If the interrupt flag, enable bit and appropriate global interrupt enable bit are set, and the interrupt occurs, then the interrupt will vector to address 0008H or 0018H, depending on the priority level.
- Interrupts can be disabled individually by their corresponding enable bits.

- Whenever any interrupt is in service, the Global Interrupt Enable bit is cleared to disable further interrupts.
- High priority interrupt sources can interrupt a low priority interrupt.
- Whenever an interrupt occurs, the return address is pushed onto the stack and the PC is loaded with the interrupt vector address (0008H or 0018H).
- When an interrupt occurs, in the ISR (Interrupt Service Routine), one can check all the flags meant for each interrupt and find out the interrupt that caused the low / high priority interrupt.
- Once detected, the corresponding interrupt flag must be cleared.
- The last instruction in the ISR is always RETFIE (return from interrupt) instruction that exits the interrupt service routine and sets the global interrupt enable bit for low and high priority interrupts and hence re-enables interrupts.
- There is a latency of one to four cycles depending on the interrupt source for the occurrence of interrupt to the execution of its ISR.

1.11 Oscillator Configurations

University Questions

Q. Write a note on oscillator modes of PIC18F458.

SPPU - May 15, 6 Marks

Q. Write a short note on oscillator support used in PIC18F458 microcontroller.

SPPU - Aug. 15(In Sem.), Oct. 16(In Sem.), 5/6 Marks

The oscillator of PIC 18 can operate in eight different modes. The programmer can program three configuration register bits viz. FOSC2, FOSC1 and FOSCO, to select one of these eight modes:

1. LP : Low-Power Crystal
2. XT : Crystal/Resonator
3. HS : High-Speed Crystal/Resonator
4. HS4 : PLL High-Speed Crystal/Resonator with PLL enabled
5. RC : External Resistor/Capacitor
6. RCIO : External Resistor/Capacitor with I/O pin enabled
7. EC : External Clock
8. ECIO : External Clock with I/O pin enabled

The Fig. 1.11.1 shows the oscillator block diagram of PIC18FXX8.

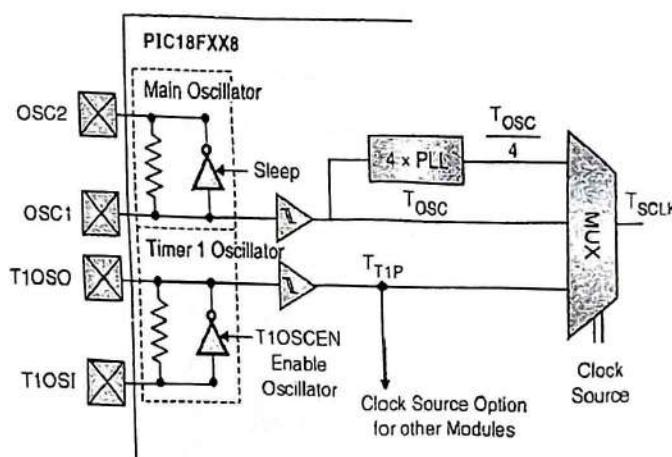


Fig. 1.11.1 : Oscillator Clock Sources

The MUX in the Fig. 1.11.1, allows the clock source to be switched from the main oscillator to a low frequency oscillator. The low frequency oscillator is the one used by Timer 1. This switching can be done by enabling the bit OSCSEN bit of configuration register. The OSCCON register also needs its SCS bit to be set for using the low frequency oscillator. Also the Timer1 oscillator should be enabled by the bit T1OSCEN. The structure of OSCCON and CONFIG registers are shown below.

OSCCON : OSCILLATOR CONTROL REGISTER

U-0	U-0	U-0	U-0	U-0	U-0	U-0	R/W-1
		-	-			-	SCS

bit 7-1 Unimplemented : Read as '0'

bit 0 SCS : System Clock Switch bit

When OSCSEN configuration bit = 0 and T1OSCEN bit is set :

1 = Switch to Timer1 oscillator/clock pin

0 = Use primary oscillator/clock input pin

When OSCSEN is clear or T1OSCEN is clear :

Bit is forced clear.

Fig. 1.11.2

CONFIG1H : CONFIGURATION REGISTER 1 HIGH

U-0	U-0	R/P - 1	U-0	U-0	R/P-1	R/P-1	R/P-1
0	-	OSCEN	-	-	FOSC2	FOSC1	FOSC0

bit 7 – 6 Unimplemented : Read as '0'

bit 5 OSCSEN : Oscillator System Clock Switch Enable bit

1 = Oscillator system clock switch option is disabled (main oscillator is source)

0 = Oscillator system clock switch option is enabled (oscillator switching is enabled)

bit 4-3 Unimplemented : Read as '0'

bit 2-0 **FOSC2 : FOSCO** : Oscillator Selection bits

111 = RC oscillator w/OSC2 configured as RA6

110 = HS oscillator with PLL enabled/clock frequency = (4 × FOSC)

101 = EC oscillator w/OSC2 configured as RA6

100 = EC oscillator W/OSC2 configured as divide -by -4 clock output

011 - RC oscillator

010 - HS oscillator

001 = XT oscillator

800 - LP oscillator

Fig. 1.11.3

1.11.1 HS, HS4, XT or LP Oscillator

University Question

Q. Write a note on oscillator modes of PIC18F458.

SPPU - May 15, 6 Marks

- In all these modes a crystal or ceramic resonator is connected between the pins OSC1 and OSC2 as shown in the Fig. 1.11.4. Crystal oscillator is kept in metal housing with two pins where you have the frequency written at which crystal oscillates. Two ceramic capacitor of different values connected on the terminal OSC1 and OSC2 respectively, whose other end is connected to the ground, are used for different frequencies of operation.
- Oscillator and capacitors can be packed in joint case with three pins. Such element is called ceramic resonator.
- Fig 1.11.4 shows ceramic resonators. A centre pin of the element is the ground, while end pins are connected with OSC1 and OSC2 pins on the microcontroller. A series resistor R_s is required to strip cut the crystal while the value of R_f varies with crystal chosen.

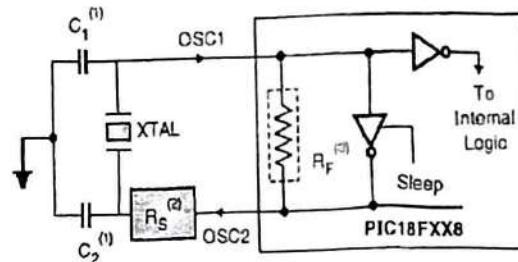


Fig. 1.11.4 : Ceramic resonators

1.11.2 RC Oscillator Mode

University Question

Q. Write a note on oscillator modes of PIC18F458.

SPPU - May 15, 6 Marks

- In applications where great time precision is not necessary, RC oscillator offers additional savings during purchase.
- The resonant frequency of RC oscillator depends on supply voltage rate, resistance R, capacitor C and working temperature. The resonant frequency is also influenced by normal variations in process parameters, by tolerance of external R and C components, etc. Fig. 1.11.5 shows how RC oscillator is connected with PIC18FXX8. The value of resistor R should be between 3 K Ω and 100 K Ω .

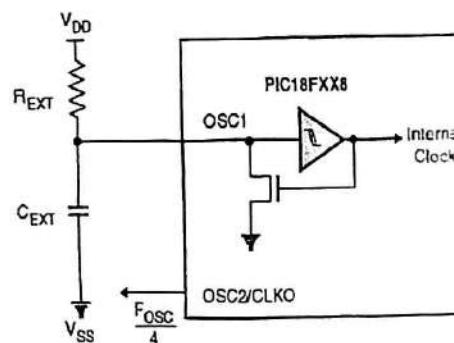


Fig. 1.11.5 : RC oscillator connected to PIC18FXX8

- Capacitor above 20pF should be used for noise and stability. No matter which oscillator is being used, in order to get a clock that microcontroller works upon; a clock of the oscillator must be divided by 4.

1.11.3 EC and ECIO Oscillator Modes

University Question

Q. Write a note on oscillator modes of PIC18F458.

SPPU - May 15, 6 Marks

- These modes require external clock source to be connected on OSC1 pin. These modes do not require any oscillator start-up time as others. Also they do not require any power-on reset circuit.

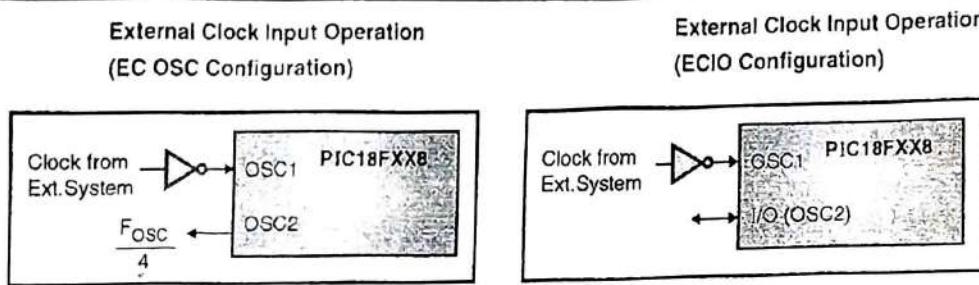


Fig. 1.11.6 : EC and ECIO Oscillator Modes

- Fig. 1.11.6 shows the connections in EC and ECIO mode. In EC mode, the oscillator frequency divided by 4 is available on OSC2 pin. In ECIO mode, OSC2 pin becomes an extra I/O pin as it has no functionality in oscillator operation.

1.11.4 Instruction Cycle

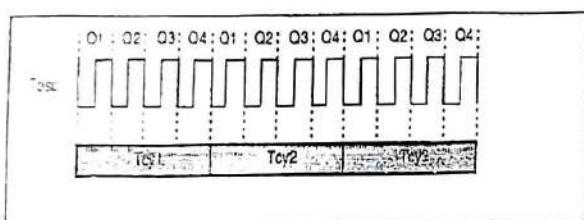


Fig. 1.11.7 : Relation between a clock and number of instruction cycle

- Oscillator clock divided by 4 can also be obtained on OSC2/CLKOUT pin, and can be used for testing or synchronizing other logical circuits.
- Following the supply voltage, oscillator starts oscillating. Oscillation at first has an unstable period and amplitude, but after some period of time it becomes stabilized.
- To prevent such inaccurate clock from influencing microcontroller's performance, we need to keep the microcontroller in reset state during stabilization of oscillator's clock.

1.12 PIC18 RESET

The PIC18 has multiple reset mechanisms. The PIC18FXX8 devices differentiate between various kinds of Reset:

- | | |
|---|--|
| (a) Power-on Reset (POR) | (b) MCLR Reset during normal operation |
| (c) MCLR Reset during Sleep | (d) Watchdog Timer (WDT) Reset (during normal operation) |
| (e) Programmable Brown-out Reset (PBOR) | (f) RESET Instruction |
| (g) Stack Full Reset | (h) Stack Underflow Reset |

A simplified block diagram of the On-Chip Reset Circuit is shown in Fig. 1.12.1.

The Enhanced MCU devices have a MCLR noise filter in the MCLR Reset path. The filter will detect and ignore small pulses. AWDT Reset does not drive MCLR pin low.

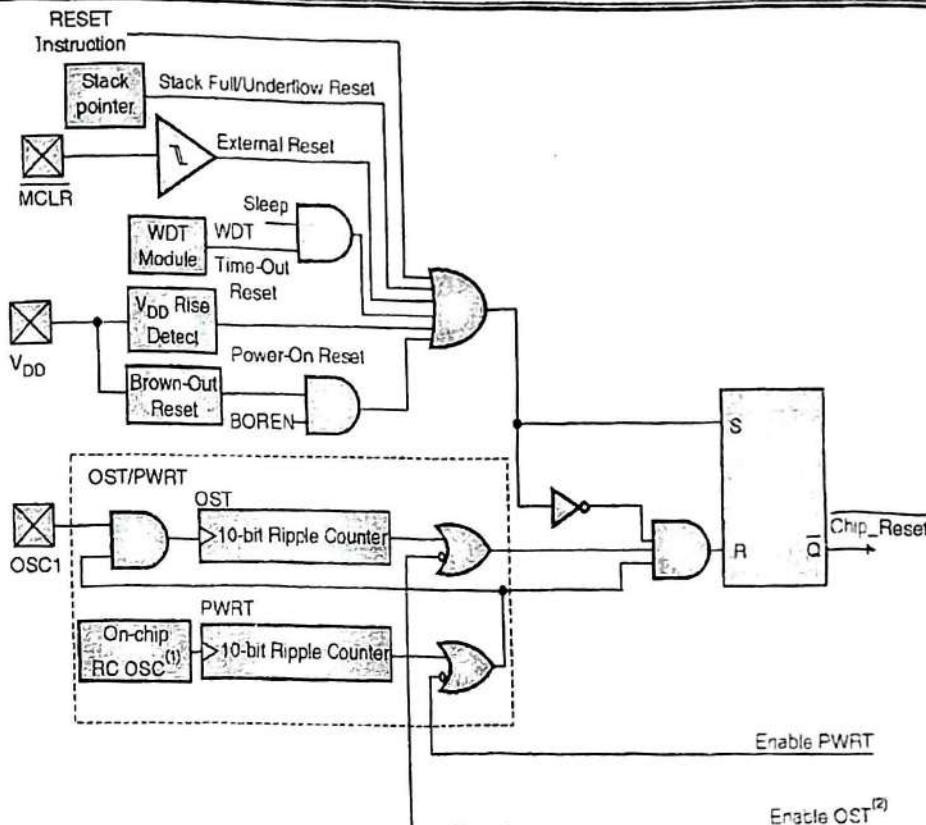


Fig. 1.12.1 : Simplified Block Diagram of ON-CHIP Reset Circuit

1.12.1 Power-on Reset (POR)

A Power-on Reset pulse is generated on-chip when a V_{DD} rise is detected. To take advantage of the POR circuitry, connect the MCLR pin directly (or through a resistor) to V_{DD}. This eliminates external RC components usually needed to create a Power-on Reset delay. A minimum rise rate for V_{DD} is specified (refer to parameter D004). For a slow rise time, see Fig. 1.12.2.

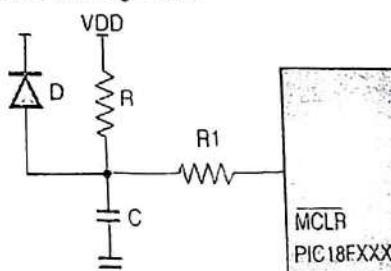


Fig. 1.12.2 External Power-on reset circuit
(for slow V_{DD} Power-up)

When the device starts normal operation (exits the Reset condition), device operating parameters (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. Brown-out Reset may be used to meet the voltage start-up condition.

1.12.2 MCLR

PIC18FXX8 devices have a noise filter in the MCLR Reset path. The filter will detect and ignore small pulses.

It should be noted that a WDT Reset does not drive MCLR pin low.

The behavior of the ESD protection on the MCLR pin differs from previous devices of this family. Voltages applied to the pin that exceed its specification can result in both Resets and current draws outside of device specification during the Reset event. For this reason, Microchip recommends that the MCLR pin no longer be tied directly to V_{DD}. The use of an RC network, as shown in Fig. 1.12.2, is suggested.

1.12.3 Power-up Timer (PWRT)

The Power-up Timer provides a fixed nominal time-out (parameter #33), 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 (PWREN in CONFIG2L register) is provided to enable/disable the PWRT.

The power-up time delay will vary from chip to chip due to VDD, temperature and process variation.

1.12.4 Oscillator Start-up Timer (OST)

The Oscillator Start-up Timer (OST) provides a 1024 oscillator cycle (from OSC1 input) delay after the PWRT delay is over. This additional delay ensures that the crystal oscillator or resonator has started and stabilized.

The OST time-out is invoked only for XT, LP, HS and HS4 modes and only on Power-on Reset or wake-up from Sleep.

1.12.5 PLL Lock Time-out

With the PLL enabled, the time out sequence following a Power-on Reset is different from other oscillator modes. A portion of the Power-up Timer is used to provide a fixed time-out that is sufficient for the PLL to lock to the main oscillator frequency. This PLL lock time-out (TPLL) is typically 2ms and follows the oscillator start-up time-out (OST).

1.12.6 Brown-out Reset (BOR)

A configuration bit, BOREN, can disable (if clear/programmed), or enable (if set), the Brown-out Reset circuitry. If VDD falls below parameter D005 for greater than parameter #35, the brown-out situation resets the chip. A Reset may not occur if VDD falls below parameter D005 for less than parameter #35. The chip will remain in Brown-out Reset until VDD rises above BVDD. The Power-up Timer will then be invoked and will keep the chip in Reset an additional time delay (parameter #33). If VDD drops below BVDD 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 BVDD, the Power-up Timer will execute the additional time delay.

1.12.7 Time-out Sequence

- On power-up, the time-out sequence is as follows: First, PWRT time-out is invoked after the POR time delay has expired, then OST is activated. The total time-out will vary based on oscillator configuration and the status of the PWRT. For example, in RC mode with the PWRT disabled, there will be no time-out at all.

- Microcontroller PIC16F458 knows several sources of resets:

- (a) Reset during switching the power on, POR (Power-On Reset)s
- (b) Reset during regular work by bringing logical zero to MCLR microcontroller's pin.
- (c) Reset during SLEEP regime.
- (d) Reset at Watchdog Timer (WDT) overflow.
- (e) Reset during at WDT overflow during SLEEP work regime.

- The most important reset sources are (a) and (b). The first one i.e. source (a) occurs each time a power supply is brought to the microcontroller and serves to bring all registers to their initial states. The second one i.e. (b) is a product of purposeful bringing in of a logical zero to MCLR pin during normal operation of the microcontroller. This second one is often used in program development.

- During a reset, RAM memory locations are not reset. They are unknown during a power up and are not changed at any reset. Unlike these, the SFR registers are reset to their initial values.

- The program counter is set to zero (0000 H) after reset. It enables the program to start executing from the first instruction that is written.

1.12.8 Reset at Supply Voltage Drop below the Permissible (Brown-out Reset)

- The impulse that is required for resetting during a voltage drop in the supply voltage V_{DD} (in a range from 1.2V to 1.8V), is generated by microcontroller itself.
- The impulse lasts 72 ms which is enough time for an oscillator to get stabilized. These 72 ms are provided by an internal PWRT timer which has its own RC oscillator.
- Microcontroller is in a reset mode as long as PWRT is active. However, as device is working, problem arises when supply doesn't drop to zero but falls below the limit that doesn't guarantee microcontroller's proper functioning. This is a likely case in practice, especially in industrial environment where disturbances and instability of supply are an everyday occurrence. To solve this problem we need to make sure that microcontroller is in a reset state each time supply falls below the approved limit. Fig. 1.12.3 shows example of drop in supply below proper level.



- If, according to electrical specification, the internal reset circuit of a microcontroller cannot satisfy the needs, special electronic components can be used which are capable of generating the desired reset signal.

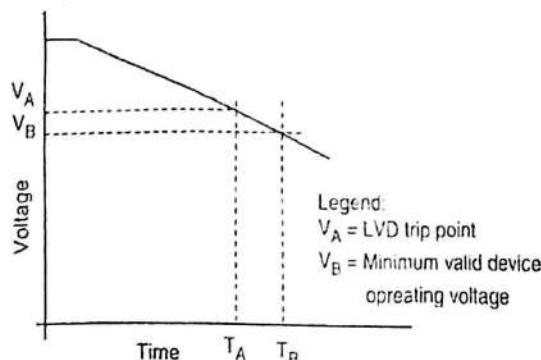


Fig. 1.12.3 : Example of voltage supply drop below the proper level

- The special components can also function in watching over supply voltage. If voltage drops below specified level, a logical zero would appear on MCLR pin which holds the microcontroller in reset state until voltage is not within limits that guarantee accurate performance.

1.12.9 Watchdog Timer

- The watchdog timer is a free running on-chip RC oscillator. It does not require any external components.
- It is used to ensure reliable operation of the system.
- It runs even if the clock on the OSC1/CLK IN and OSC2/CLKOUT pins of device has been stopped. E.g. execution of SLEEP instruction. Fig. 1.12.4 shows block diagram of watchdog timer.
- It can be permanently disabled by clearing the WDTE bit. If it is in SLEEP mode, watchdog timer time out causes device to wake up and continue normal operation. TO(Time-out) of watchdog timer generates a device reset.

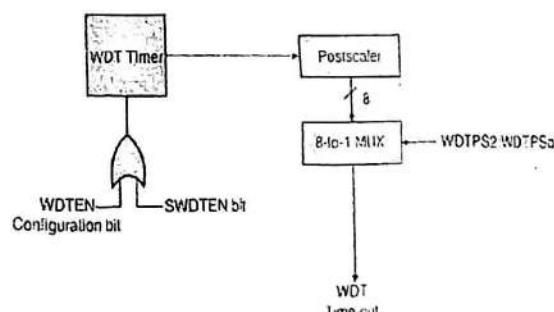


Fig. 1.12.4 : Watchdog timer block diagram

1.13 Program and Data Memory Organization

University Questions

- Q. Explain the program memory map for PIC18F458.
SPPU - Aug.15 (In Sem.), 3 Marks
- Q. Explain RAM organization of PIC18F458.
SPPU - Dec. 14, , Aug. 15 (In Sem.), Oct. 16 (In Sem.), 6 Marks
- Q. Explain the GP RAM for PIC microcontroller.
SPPU - Dec. 15, 3 Marks

- There are three sections of memory in PIC18 namely program memory, data RAM and data EEPROM. Since PIC also uses Harvard's memory organization, it has same address for multiple locations but different function. It also has separate bus for program and data and hence concurrent access of these memories is possible.
- The program memory address bus is 21-bit while the data memory address bus is 12-bit. Since most of the instructions are 16-bits, the data bus or instruction bus size of program memory is also 16-bit. The data bus size of data memory is 8-bit. The PIC18 memory space is shown in Fig 1.13.1. All these buses are separate as shown in Fig. 1.13.1.

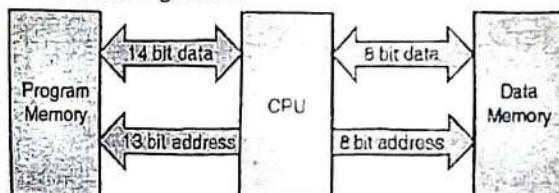


Fig. 1.13.1 : PIC18 program and data memory space

To provide the 21-bit program memory address, the PIC18 microcontroller has a 21-bit program counter that is divided into three registers: PCU, PCH, and PCL, only PCL being directly accessible to the user. PCH and PCL are eight bits, whereas PCU is five bits.

1.13.1 Data Memory

The data memory is implemented using SRAM (Static RAM). Each location in the data memory is also referred to as a register. The PIC18 microcontroller supports 4096 bytes of data memory locations. Hence, it requires 12 bits of address to select one of the data registers. The data memory map of the PIC18 microcontroller is shown in Fig. 1.13.2.



Eight bits of the PIC18 instruction are used to specify the register to be operated on. As a result only 256 registers can be accessed, i.e. the 4096 registers are divided into 16 banks. Only one bank of 256 registers is active at any time, selected by the 4-bit Bank Select Register (BSR).

The programmer has to change the contents of the BSR register in order to change the active bank. There are two sets of registers: General-Purpose Registers (GPRs) and Special-Function Registers (SFRs). GPRs are used to hold user random data (hence the name general purpose) when the PIC18 is executing a program. SFRs are control registers used for the CPU and on-chip peripheral devices.

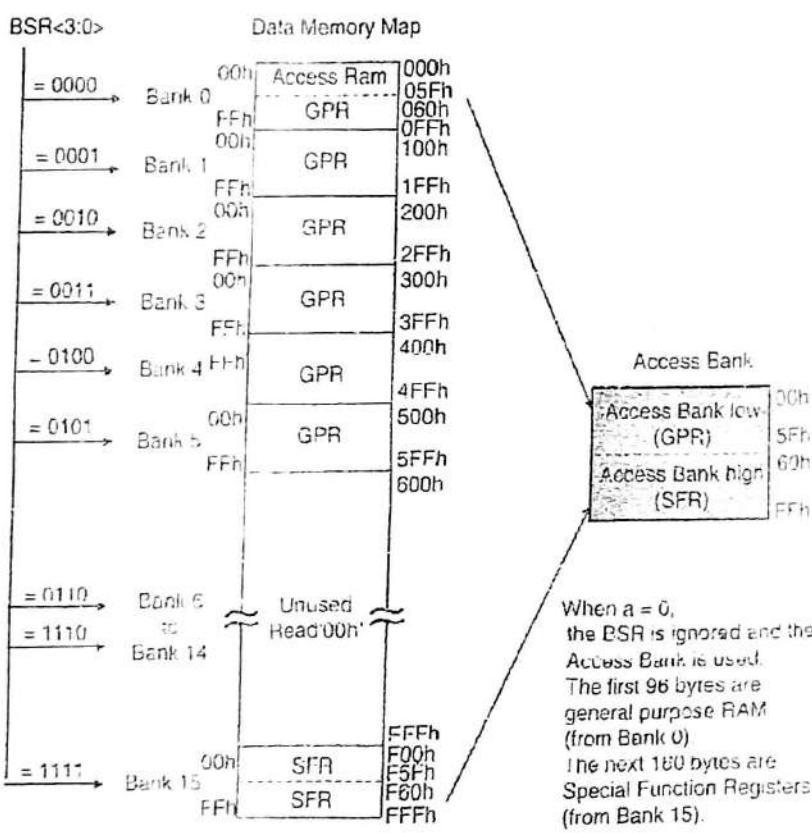


Fig. 1.13.2 : Data memory map of PIC18

The SFRs begin from the highest address (as shown in Fig. 1.13.2) and go downward, whereas GPRs must be used from address 0 and go upward. The first 96 bytes (i.e. in bank 0, 000H to 05FH) of the GPRs and the last 160 bytes (i.e. in bank 15, F60H to FFFF) of the SFRs are grouped into a special bank called access bank.

The PIC18 also has a 1KB of data EEPROM, which is readable and writable during normal operation with no extra power supply. The data EEPROM memory is not directly mapped in the 4096 bytes of register file space; instead, it is indirectly addressed through a special function register.

Note : BSR is the 4-bit bank select register.

1.13.1(A) Bank Select Register (BSR)

University Questions

- Q. Explain the function of Bank select register.
SPPU - Dec.16, Aug.17 (In sem.), Dec.17, 4 Marks
- Q. Explain the function of Bank select register. Write an instruction in assembly language which will select BANK 15.
SPPU - Oct. 19 (In sem.), 4 Marks

- The need for a large general purpose memory space dictates a RAM banking scheme. The data memory is partitioned into sixteen banks. When using direct addressing, the BSR should be configured for the desired bank.
- BSR<3:0> holds the upper 4 bits of the 12-bit RAM address. The BSR<7:4> bits will always read '0's and writes will have no effect.
- A MOVLB instruction has been provided in the instruction set to assist in selecting banks.
- If the currently selected bank is not implemented, any read will return all '0's and all writes are ignored.

- The Status register bits will be set/cleared as appropriate for the instruction performed.

1.13.1(B) Special Function Registers (SFR)

University Question

- Q. Explain the SFRs for PIC microcontroller.

SPPU - Dec. 15, Aug. 17 (In Sem.), 3 Marks

- The SFRs begin from the highest address (as shown in Fig. 1.13.2) and go downward, whereas GPRs must be used from address 0 and go upward. The first 96 bytes (i.e. in bank 0, 000H to 05FH) of the GPRs and the last 160 bytes (i.e. in bank 15, F60H to FFFFH) of the SFRs are grouped into a special bank called access bank.
- The special function registers are basically the control words of the various modules of a microcontroller. It includes modules like, reset control, timer control, serial port communication, parallel port communication etc.
- Fig. 1.13.3 shows the detailed structure of SFRs in PIC18F458 microcontroller

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2 ⁽²⁾	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2 ⁽²⁾	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2 ⁽²⁾	FBDh	CCP1CON	F9Dh	PIE 1
FFCh	STKPTR	FDCh	PREINC2 ⁽²⁾	FBCh	ECCPR1H ⁽⁵⁾	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2 ⁽²⁾	FBBh	ECCPR1L ⁽⁵⁾	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	ECCP1CON ⁽⁵⁾	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMROH	FB7h	ECCP1DEL ⁽⁵⁾	F97h	—
FF6h	TBLPTRL	FD6h	TMROL	FB6h	ECCPAS ⁽⁵⁾	F96h	TRISE ⁽⁵⁾
FF5h	TABLAT	FD5h	TOCON	FB5h	CVRCON ⁽⁵⁾	F95h	TRISD ⁽⁵⁾
FF4h	PRODH	FD4h	—	FB4h	CMCON ⁽⁵⁾	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	JNTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FDOh	RCON	FBOh	—	F90h	—
FEFh	INDFO ⁽²⁾	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTNCO ⁽²⁾	FCFh	TMR1L	FAEh	RCREG	F8Eh	—



Address	Name	Address	Name	Address	Name	Address	Name
FEDh	POSTDECO ⁽²⁾	FC0h	T1CON	FADh	TXREG	F8Dh	LATE ⁽⁵⁾
FECh	PREINCO ⁽²⁾	FCCh	TMR2	FACH	TXSTA	F8Ch	LATD ⁽⁵⁾
FEBh	PLUSWO ⁽²⁾	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSROH	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSROL	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1 ⁽²⁾	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POST INC 1 ⁽²⁾	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1 ⁽²⁾	FC5h	SSPCON2	FA5h	JPR3	F85h	—
FE4h	PREINC1 ⁽²⁾	FC4h	ADRESH	FA4h	PIR3	F84h	PORTE ⁽⁵⁾
FE3h	PLUSW1 ⁽²⁾	FC3h	ADRESL	FA3h	PIE3	F83h	PORTD ⁽⁵⁾
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PJE2	F80h	PORTA

Fig. 1.13.3 Special Function Register (SFR) map

1.13.2 Program Memory Organization

University Question

Q. Draw the stack pointer register.

SPPU - Aug. 17 (In Sem.), 6 Marks

- PIC18 microcontroller has a 21-bit program counter and hence is capable of addressing a 2MB program memory space.
- As discussed in the architecture of PIC18FXX8 microcontroller has a 31-entry return address stack (Processor stack); which is used to hold return addresses for subroutine call and interrupt processing.
- This return address stack is not a part of the program memory space. The program memory map is illustrated in Fig. 1.13.4
- As shown in Fig. 1.13.4, the address 0000 H is assigned to the reset vector, which is the starting address after reset. The address 0008 H onwards is the address of the high-priority interrupt service routine.
- Sixteen bytes are allocated to the high-priority interrupt service routine by default. The address 0018H onwards is allocated for low-priority interrupt service routine. The user program should follow the low-priority interrupt service routine.

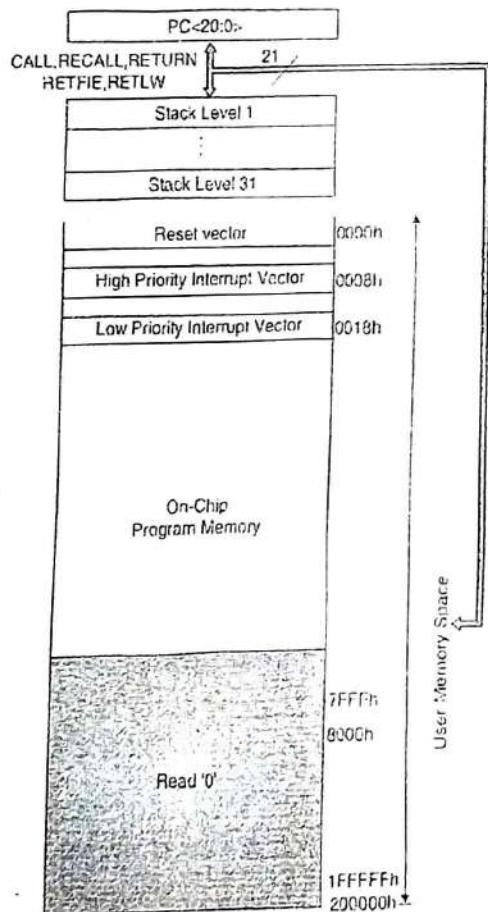


Fig 1.13.4 : Program memory space of PIC18

1.14 Pipelining

University Question

Q. Explain the concept of Pipelining used in PIC 18 microcontroller. **SPPU - May 16, 6 Marks**

- An Instruction cycle consists of cycles Q1, Q2, Q3 and Q4.
- The Program counter (PC) holds information about the address of the next instruction.
- The cycles of calling and executing instructions are connected in such a way that in order to make a call, one instruction cycle is needed, and one more is needed for decoding and execution. However, due to pipelining, each instruction is effectively executed in one cycle.
- If instruction causes a change on program counter, and PC does not point to the following but to some other address (which can be the case with jumps or with

calling subprograms), two cycles are needed for executing an instruction.

- This is so because instruction must be processed again, but this time from the right address.
- The calling cycle begins with Q1 clock, by writing into instruction register (IR). Decoding and executing begins with Q2, Q3 and Q4 clocks. Fig 1.14.2 shows Instruction pipelining in PIC microcontroller.
- In T_{CY0} i.e. in timing cycle 0 the PIC microcontroller reads in instruction MOVLW 55H (it does not matter to us what instruction was executed, because there is no rectangle pictured on the bottom).
- In T_{CY1} i.e. in timing cycle 1 the PIC microcontroller executes the instruction MOVLW 55H and simultaneously reads in MOVWF PORTB.
- In T_{CY2} i.e. in timing cycle 2 the PIC microcontroller executes the instruction MOVWF PORTB and simultaneously reads in CALL SUB_1

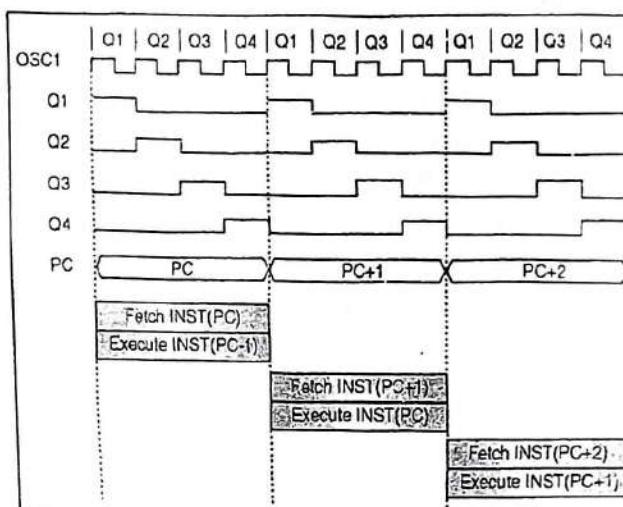
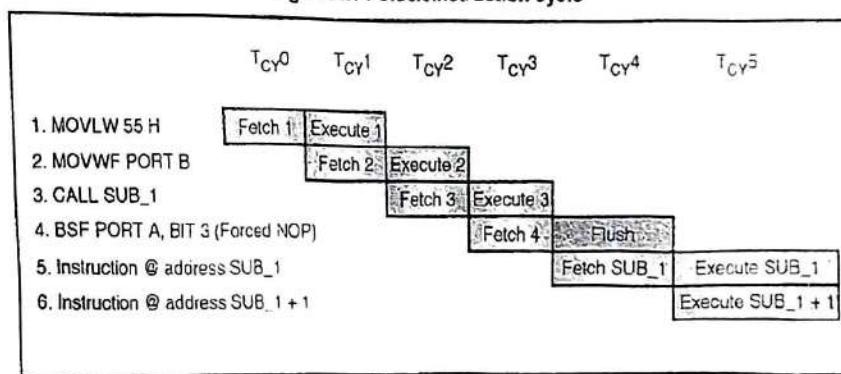


Fig. 1.14.1 : Clock/Instruction cycle



All instructions are single cycle except for any program branches. These take two cycles since the fetch instructions are "flushed" from the pipeline while the new instruction is being fetched and then executed.

Fig. 1.14.2 : Instruction pipelining in PIC microcontroller



- In **T_{CY3}** i.e. in timing cycle 3 the microcontroller executes the call of a subprogram CALL SUB_1, and simultaneously reads in instruction BSF PORTA, BIT3. As this instruction is not the one we need, or is not the first instruction of a subprogram SUB_1 whose execution is next in order, instruction must be read in again. This is a good example of an instruction needing more than one cycle.
- In **T_{CY4}** the instruction cycle is totally used up for reading in the first instruction from a subprogram at address SUB_1.
- in **T_{CY5}** i.e. timing cycle 5 the PIC microcontroller executes the first instruction from a subprogram SUB_1 and reads in the next one.

1.15 Power- Down Mode (Sleep)

- Power-down mode is entered by executing a SLEEP instruction. If enabled, the Watchdog Timer will be cleared but keeps running, the PD bit (RCON<2>) is cleared, the TO bit (RCON<3>) is set and the oscillator driver is turned off. The I/O ports maintain the status they had before the SLEEP instruction was executed (driving high, low or high-impedance).
- For lowest current consumption in this mode, place all I/O pins at either VDD or VSS, ensure no external circuitry is drawing current from the I/O pin, power-down the A/D and disable external clocks. Pull all I/O pins that are high-impedance inputs, high or low externally, to avoid switching currents caused by floating inputs. The TOCKI input should also be at VDD or VSS for lowest current consumption. The contribution from on-chip pull-ups on PORTB should be considered.
- The MCLR pin must be at a logic high level (VIHMC).

1.15.1 Wake-Up from Sleep

- The device can wake-up from Sleep through one of the following events:
 1. External Reset input on MCLR pin.
 2. Watchdog Timer wake-up (if WDT was enabled).
 3. Interrupt from INT pin, RB port change or a peripheral interrupt.

- The following peripheral interrupts can wake the device from Sleep:
 1. PSP read or write.
 2. TMR1 interrupt - Timer1 must be operating as an asynchronous counter.
 3. TMR3 interrupt - Timer3 must be operating as an asynchronous counter.
 4. CCP Capture mode interrupt.
 5. Special event trigger (Timer1 in Asynchronous mode using an external clock).
 6. MSSP (Start/Stop) bit detect interrupt.
 7. MSSP transmit or receive in Slave mode(SPI/I²C).
 8. USART RX or TX (Synchronous Slave mode).
 9. A/D conversion (when A/D clock source is RC).
 10. EEPROM write operation complete.

Other peripherals cannot generate interrupts, since during Sleep, no on-chip clocks are present.

1.16 Exam Pack (Review and University Questions)

- Q. Compare microprocessor and microcontroller.
(Refer Section 1.2.3) (May 12, May 19, 6 Marks)
- Q. What is an interrupt ?
(Refer Section 1.1.6) (Dec. 12, 3 Marks)
- Q. Compare Harvard and Von Neumann architecture.
(Refer Section 1.5) (Dec. 14, 7 Marks)
- Q. Explain status register of PIC18F458.
(Refer Section 1.10) (Dec. 14, Aug. 15 (In Sem.), Oct. 16 (In Sem.), 5/6 Marks)
- Q. Explain the status register of PIC 18 microcontroller.
(Refer Section 1.10) (Dec. 14, 6 Marks)
- Q. Write short note on program counter.
(Refer Section 1.10.5(C)) (Dec. 14, 2 Marks)
- Q. Explain RAM organization of PIC18F458.
(Refer Section 1.13)
(Dec. 14, , Aug. 15 (In Sem.), Oct. 16 (In Sem.), 6 Marks)

Q. Compare RISC and CISC architectures. (Refer Section 1.4) (May 15, 6 Marks, Dec.15, 6 Marks, May 16, Oct. 16 (In Sem.), 4/6 Marks Dec.16, Aug.17 (In sem.) 6/7 Marks, May 19, 6 Marks)	Q. Explain the function of Bank select register (Refer Section 1.13.1(A)) (Dec.16, Aug.17 (In sem.), Dec.17, 4 Marks)
Q. Write a note on oscillator modes of PIC18F458. (Refer Section 1.11) (May 15, 6 Marks)	Q. Explain the function of Program counter (Refer Section 1.10.5(C)) (Aug. 17 (In Sem.), 4 Marks)
Q. Write a note on oscillator modes of PIC18F458. (Refer Section 1.11.1) (May 15, 6 Marks)	Q. Explain Stack organization of PIC 18f458 microcontroller. (Refer Section 1.10.5(D)) (Aug. 17 (In Sem.), 6 Marks)
Q. Write a note on oscillator modes of PIC18F458. (Refer Section 1.11.2) (May 15, 6 Marks)	Q. Draw the stack pointer register. (Refer Section 1.13.2) (Aug. 17 (In Sem.), 6 Marks)
Q. Write a note on oscillator modes of PIC18F458. (Refer Section 1.11.3) (May 15, 6 Marks)	Q. Draw the status register for the PIC microcontroller and explain the function of Negative flag (Refer Section 1.10.5(B)) (Dec. 17, 4 Marks)
Q. Write short note on interrupt structure of PIC18F458 microcontroller. (Refer Section 1.10.13) (Aug. 15 (In Sem.), 4 Marks)	Q. Explain clearly the differences between microcontroller and microprocessor. (Refer Section 1.2.3) (May 18, 6 Marks)
Q. Write a short note on oscillator support used in PIC18F458 microcontroller. (Refer Section 1.11) (Aug. 15 (In Sem.), Oct. 16 (In Sem.), 5/6 Marks)	Q. Draw the status register for the PIC microcontroller and explain the function of Digit Carry flag (Refer Section 1.10.5(B)) (May 18, Dec. 18, 4 Marks)
Q. Explain the program memory map for PIC18F458. (Refer Section 1.13) (Aug.15 (In Sem.), 3 Marks)	Q. Draw the status register and explain the function of each flag. (Refer Section 1.10.5(B)) (Oct. 19 (In sem.), 6 Marks)
Q. Explain the program memory map for PIC18F458. (Refer Section 1.10.1) (Aug. 15 (In Sem.), 3 Marks)	Q. Explain the function of Bank select register. Write an instruction in assembly language which will select BANK 15. (Refer Section 1.13.1(A)) (Oct. 19 (In sem.), 4 Marks)
Q. Explain the GP RAM for PIC microcontroller. (Refer Section 1.13) (Dec. 15, 3 Marks)	Q. Explain stack memory organisation and stack pointer in detail. (Refer Section 1.10.2) (Dec. 19, 6 Marks)
Q. Explain the SFRs for PIC microcontroller. (Refer Section 1.13.1(B)) (Dec. 15, Aug. 17 (In Sem.), 3 Marks)	Q. Explain stack memory organisation and stack pointer in detail. (Refer Section 1.10.5(D)) (Dec. 19, 6 Marks)
Q. Explain the status register of PIC 18F458. (Refer Section 1.10) (May 16, 6 Marks)	
Q. Explain the concept of Pipe lining used in PIC 18 microcontroller. (Refer Section 1.14) (May 16, 6 Marks)	

2

Assembly Language Programming

Unit - I

2.1 Addressing Modes for PIC18 Microcontroller

University Questions

- Q. Explain any three addressing modes of PIC18 with one example each.
- Q. Explain with an example the addressing modes of PIC18F548.
- Q. Explain various addressing modes used in PIC18 microcontroller.
- Q. Explain Register indirect addressing mode and immediate addressing mode in detail, with suitable example.

SPPU - Dec. 14, 6 Marks

SPPU - Oct. 16 (In Sem.), 4 Marks

SPPU - Dec. 17, Oct. 19, 6 Marks

SPPU - Dec. 19, 4 Marks

Addressing modes refer to the different methods of selecting the operand for a instruction. The PIC18 microcontroller provides register direct, immediate, inherent, indirect, and bit-direct addressing modes for specifying instruction operands.

2.1.1 Register Direct

University Questions

- Q. Explain Register direct addressing mode of PIC18 microcontroller.
- Q. Explain the following in detail: i) direct addressing mode.

SPPU - Dec. 16, 3 Marks

SPPU - May 19, 3 Marks

The PIC18 device uses an 8-bit direct address (specified in the opcode itself) value to specify a data register as an operand. The register may be in the access bank or any other banks. In case if the register is in access bank, the 8-bit value is used to select a register in the access bank, and the bank value in the BSR register is ignored. If the access bank is not selected, then the access is done from the memory of the bank specified in the BSR register. The direct addressing mode address calculation is shown in Fig. 2.1.1.

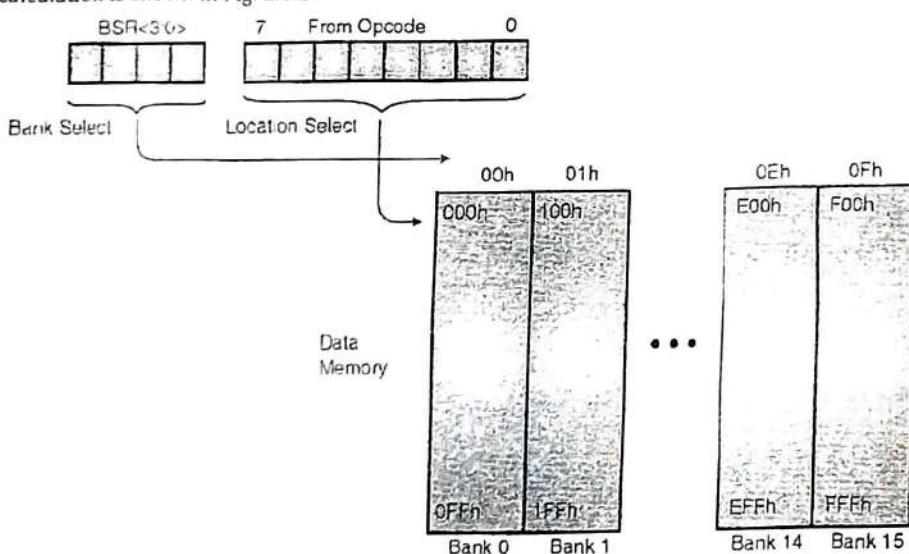


Fig. 2.1.1: Direct addressing mode



The following instructions illustrate the register direct addressing mode:

1. MOVWF 0x0C, BANKED

This instruction copies the contents of the WREG register to the data memory location 0x0C, bank selected by the BSR register. The word BANKED informs the assembler that the BSR register must be included in specifying the data register to be operated on.

2. MOVWF 0x1E, A

This instruction copies the contents of the WREG register to the memory location 0x1E in the access bank.

3. MOVFF REG1, REG2

This instruction copies the contents of the register reg1 to the register reg2. Both reg1 and reg2 are 12-bit values. The register BSR is ignored.

2.1.2 Immediate Mode

University Questions

Q. Explain Immediate addressing mode of PIC18 microcontroller. **SPPU - Déc. 16, 3 Marks**

Q. Explain the following in detail: Immediate addressing mode. **SPPU - May 19, 3 Marks**

In the immediate addressing mode, the operand is provided in the instruction itself. The following instructions illustrate the immediate addressing mode:

1. ADDLW 0x12

This instruction adds the hex value 12 to the WREG register and places the result in the WREG register.

2. MOVLW 0x1E

This instruction loads from the memory into the hex value 1E into the WREG register.

3. MOVLB 9

This instruction places the decimal value 9 in the lower four bits of the BSR register and hence changing the active bank i.e. the lower four bits become 1001; hence makes bank 9 the active bank. The value to be operated on directly is often called literal (immediate addressing mode in terms of Intel processor).

2.1.3 Inherent Mode

In the inherent (implied addressing mode in terms of Intel) mode, the operand is implied in the opcode field.

The instruction opcode does not provide the address of the implied operand. The following instructions illustrate the inherent mode:

1. MOVLW 0x4F

This instruction places the hex value 4F in the WREG register. In this addressing mode the hex number 4F is given in the instruction machine code itself. The destination WREG is implied in the opcode field. The address of the WREG register 0xFE8 is not specified, which is implied or inherent.

2. ANDLW 0x9A

This instruction performs an AND operation on the corresponding bits of the hex number 9A and the WREG register (i.e., bit i of WREG and with bit i of the value 0x9A; i = 0 . . . 7). In this example, only the immediate value 0x13 is specified in the instruction machine code. The address of the WREG register 0xFE8 is not specified, which is implied or inherent.

2.1.4 Register Indirect Mode

Indirect addressing is used in cases where the data memory address in the instruction is not fixed. A special function register FSR, is used as a pointer to the data memory location that is to be read or written. Since this pointer register is in SRAM, the contents can be modified by the program. This can be useful for look up tables in the data memory and for software stacks. Fig. 2.1.2 shows the operation of indirect addressing. This shows the moving of the value to the data memory location, specified by the value of the FSR register.

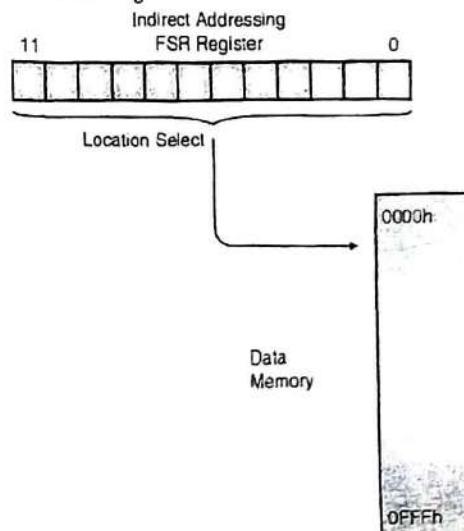


Fig. 2.1.2: Indirect addressing mode



There are three indirect addressing registers: FSR0, FSR1, and FSR2. To address the entire data memory space of 4096 bytes, 12 bits are required. To store the 12-bit address information, two 8-bit registers (also called as **indirect addressing registers**) are used namely:

1. **FSR0:** Composed of FSR0H and FSR0L.
2. **FSR1:** Composed of FSR1H and FSR1L
3. **FSR2:** Composed of FSR2H and FSR2L

Indirect addressing is possible by using one of the INDF registers. Any instruction using the INDF register actually accesses the register pointed to by the File Select Register (FSR). Reading the INDF register itself, indirectly (i.e. when FSR = 0), will read 00h. Writing to the INDF register indirectly, results in a no operation. The FSR register contains a 12-bit address, as shown in Fig. 2.1.2. The INDF registers are not physical registers. Addressing INDF register actually addresses the register whose address is contained in the FSR register (FSR is a pointer). If an instruction writes a value to INDF0, the value will be written to the data register with the address indicated by the register pair FSR0H:FSR0L. A read from INDF1 reads the data from the data register with the address indicated by the register pair FSR1H:FSR1L. INDFn can be used in a program anywhere an operand can be used.

Each FSR register has an INDF register associated with it plus four additional register addresses. Performing an operation on one of these five registers determines how the FSR will be modified during indirect addressing.

1. Do nothing to FSRn after an indirect access. In this access the address is specified by using the register INDFn ($n = 0 \dots 2$).
2. Auto-decrement FSRn after an indirect access (post decrement). In this access the address is specified by using the register POSTDECr ($n = 0 \dots 2$).
3. Auto-increment FSRn after an indirect access (post increment). In this access the address is specified by using the register POSTINCn ($n = 0 \dots 2$).
4. Auto-increment FSRn before an indirect access (preincrement). In this access the address is specified by using the register PREINCr ($n = 0 \dots 2$).

5. Use the value in the WREG register as an offset to FSRn. In this access the address is the signed value in WREG is added to the value in FSR to form an address before performing an indirect access. Neither the WREG nor the FSRn is modified after the access. This access is specified by using the register PLUSWN ($n = 0 \dots 2$).

The following examples illustrate the usage of indirect addressing modes in the above cases.

1. MOVWF INDF0

This instruction copies the contents of the WREG register to the data memory location specified by the FSR0 register. After the execution of this instruction, the contents of the FSR0 register are not changed.

2. MOVWF POSTDECO

This instruction copies the contents of the WREG register to the data memory location specified by the FSR0 register and after the operation the contents of FSR0 are decremented by 1.

3. MOVWF PREINCO

This instruction first increments the FSR0 register by 1 and then copies the contents of the WREG register to the data memory location specified by the FSR0 register.

4. CLRF PLUSW0

This instruction clears the memory location at the address equal to the sum of the value in the WREG register and that in the FSR0 register.

In the previous examples, one does not need to specify whether the register is in the access bank because the complete 12-bit data register address is taken from one of the FSR registers. But for the direct addressing mode the address is taken from the BSR register also, to select the bank. This can be explained by Fig. 2.1.3. In this Figure a multiplexer is shown; that provides the address of the RAM location to be accessed. The multiplexer gets its input from the 12-bit address in FSR register for indirect addressing mode; while the address is given by the instruction (8-bit) and BSR (4-bit) register in case of direct addressing mode.

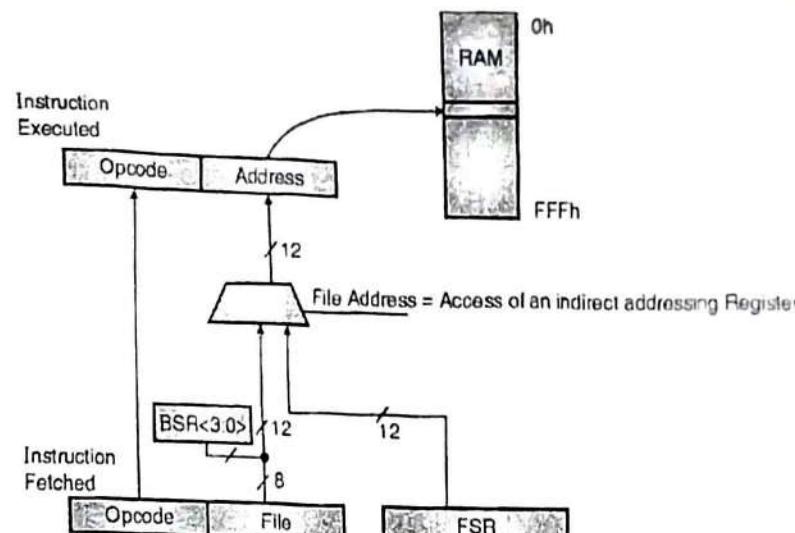


Fig. 2.1.3: Addressing mechanism of PIC18

2.1.5 Bit-Direct Addressing Mode

The PIC18F8720 microcontroller has five instructions that work with a single bit. These instructions use three bits of the opcode to specify the bit to be operated on.

For example:

1. **BCF PORTB, 2, A**

This instruction clears bit 2 of the data register PORTB, which will then pull the port B pin RB2 to low.

2. **BSF PORTA, 1, A**

This instruction sets the bit 1 of the data register PORTA, which will then pull the port A pin RA1 to high.

2.2 Instruction Set

- Each PIC16CXX instruction is a 14-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 PIC16CXX instruction set summary in Table 2.2.1 lists byte-oriented, bit-oriented, and literal and control operations. The Table 2.2.1 shows the opcode field descriptions.
- For byte-oriented instructions, 'f' represents a file register designator and 'd' represents a destination designator.
- The file register designator specifies which file register is to be used by the instruction.

- The destination designator specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the W register. If 'd' is one, the result is placed in the file register specified in the instruction.
- For bit-oriented instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the number of the file in which the bit is located.
- For literal and control operations, 'k' represents an eight or eleven bit constant or literal value.

Table 2.2.1: Opcode field descriptions

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 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.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1
label	Label name
TOS	Top of Stack
PC	Program Counter

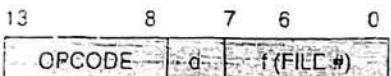


Field	Description
PCLATH	Program Counter High Latch
GIE	Global Interrupt Enable bit
WDT	Watchdog Timer/Counter
TO	Time-out bit
PD	Power-down bit
dest	Destination either the W register or the specified register file location
[]	Options
()	Contents
→	Assigned to
↔	Register bit field
ε	In the set of
italics	User defined term

The instruction set is highly orthogonal and is grouped into three basic categories :

1. Byte-oriented operations
2. Bit-oriented operations

Byte-oriented file register operations

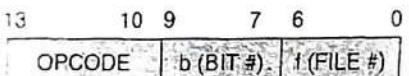


d = 0 for destination W

d = 1 for destination f

f = 7-bit file register address

Bit-oriented file register operations



b = 3-bit bit address

f = 7 bit file register address

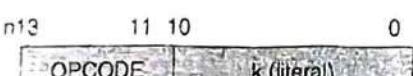
Literal and control operations

General



k = 8-bit immediate value

CALL and GOTO instructions only



k = 11-bit immediate value

3. Literal and control operations

All instructions are executed within one single instruction cycle, unless a conditional test is true or the program counter is changed as a result of an instruction.

In this case, the execution takes two instruction cycles with the second cycle executed as a NOP. One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 μ s. If a conditional test is true or the program counter is changed as a result of an instruction, the instruction execution time is 2 μ s.

Table 2.2.2 lists the instructions recognized by the MPASM assembler.

Fig. 2.2.1 shows the general formats that the instructions can have.

Note: To maintain upward compatibility with future PIC16CXX products, do not use the OPTION and TRIS instructions.

All examples use the following format to represent a hexadecimal number:

0xh

Where h signifies a hexadecimal digit.

Fig. 2.2.1: General format for instructions

Table 2.2.2: PIC16CXX Instruction Set

Mnemonic/ Operations	Description	Cycles	14-Bit Opcode		Status Affected	Notes
			MSB	LSB		
BYTE-ORIENTED FILE REGISTER OPERATIONS						
ADDWF f,d	Add W and f	1	00 0111	ffff ffff	C, DC, Z	1,2
ANDWF f,d	AND W with f	1	00 0101	ffff ffff	Z	1,2
CLRF f	Clear f	1	00 0001	ffff ffff	Z	2
CLRW -	Clear W	1	00 0001	0xxx xxxx	Z	
COMF f,d	Complement f	1	00 1001	ffff ffff	Z	1,2
DECFSZ f,d	Decrement f, Skip if 0	1(2)	00 1011	ffff ffff		1,2,3
INCF f,d	Increment f	1	00 1010	ffff ffff	Z	1,2
INCFSZ f,d	Increment f, Skip if 0	1(2)	00 1111	ffff ffff		1,2,3
IORWF f,d	Inclusive OR W with f	1	00 0100	ffff ffff	Z	1,2
MOVF f,d	Move f	1	00 1000	ffff ffff	Z	1,2
MOVWF f	Move W to f	1	00 0000	1fff ffff		
NOP -	No Operation	1	00 0000	0xx0 0000		
RLF f,d	Rotate Left f through Carry	1	00 1101	ffff ffff	C	1,2
RRF f,d	Rotate Right f through Carry	1	00 1100	ffff ffff	C	1,2
SUBWF f,d	Subtract W from f	1	00 0010	ffff ffff	C,DC,Z	1,2
SWAPF f,d	Swap nibbles in f	1	00 1110	ffff ffff		1,2
XORWF f,d	Exclusive OR W with f	1	00 0110	ffff ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS						
BCF f,b	Bit Clear f	1	01 00bb	bfff ffff		1,2
BSF f,b	Bit Set f	1	01 01bb	bfff ffff		1,2
BTFSZ f,b	Bit Test f, Skip if Clear	1(2)	01 10bb	bfff ffff		3
BTFSZ f,b	Bit Test f, Skip if Set	1(2)	01 11bb	bfff ffff		3
LITERAL AND CONTROL OPERATIONS						
ADDLW k	Add literal and W	1	11 111x	kkkk kkkk	C,DC,Z	
ANDLW k	AND literal with W	1	11 1001	kkkk kkkk	Z	
CALL k	Call subroutine	2	10 0kkk	kkkk kkkk		
CLRWDT -	Clear Watchdog Timer	1	00 0000	0110 0100	-- TO, PD	
GOTO k	Go to address	2	10 1kkk	kkkk kkkk		
IORLW K	Inclusive OR literal with W	1	11 1000	kkkk kkkk	Z	



Mnemonic, Operations	Description	Cycles	14-Bit Opcode		Status Affected	Notes
			MSB	LSB		
MOVLW k	Move literal to W	1	11 00xx	kkkk kkkk		
RETFIE -	Return from interrupt	2	00 0000	0000 1001		
RETLW K	Return with literal in W	2	11 01xx	kkkk kkkk		
RETURN -	Return from Subroutine	2	00 0000	0000 1000		
SLEEP -	Go into standby mode	1	00 0000	0110 0011	T0, PD	
SUBLW k	Subtract W from literal	1	11 110x	kkkk kkkk	C,DC,Z	
XORLW k	Exclusive OR literal with W	1	11 1010	kkkk kkkk	Z	

- Note :**
- When an I/O register is modified as a function of itself (e.g., MOVF PORTB, 1), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.
 - If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.
 - If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

2.3 Data Transfer Instructions

2.3.1 MOVF or MOVE MOVF f,d,a

University Questions

Q. Explain the instruction: MOVF f, d, a.

SPPU - Oct 16 (In Sem.), 2 Marks

Q. Explain the instruction: MOVF 0 x 04, 0, 1.

SPPU - May 18, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MOVF	f,d,a	f → dest	Z,N	1

Description	Move f The contents of register 'f' are moved to a destination dependent upon the status of 'd'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). Location 'l' can be anywhere in the 256-byte bank. If 'a' is '0', the Access Bank will be selected, overriding the RSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example MOVFRREG, 0, 0	Before Instruction REG = 0x11 W = 0x4D After Instruction REG = 0x11 W = 0x11

2.3.2 MOVFF

University Questions

Q. Explain following instruction with example: MOVFF

SPPU - May 15, 2 Marks

Q. Explain the instruction MOV fs, fd.

SPPU - Aug. 15 (In Sem.), Dec. 17, May 18, 2 Marks

Q. Explain the following instruction in detail: MOVFF.

SPPU - Dec. 16, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MOVFF	f _s , f _d	f _s → f _d	None	2
Description	Move f _s (source) to 1 st word f _d (destination) 2 nd word The contents of source register 'f _s ' are moved to destination register 'f _d '. Location of source 'f _s ' can be anywhere in the 4096-byte data space (000h to FFFh) and location of destination 'f _d ' can also be anywhere from 000h to FFFh. Either source or destination can be W (a useful special situation). MOVFF is particularly useful for transferring a data memory location to a peripheral register (such as the transmit buffer or an I/O port). The MOVFF instruction cannot use the PCL, TOSU, TOSH or TOSL as the destination register.			
Example MOVFFREG1,REG2	Before Instruction REG1 = 0x35 REG2 = 0x21 After Instruction REG1 = 0x35 REG2 = 0x35			

2.3.3 MOVLB

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MOVLB	K	k → BSR	None	1
Description	Move literal to BSR<3:0> The 8-bit literal 'k' is loaded into the Bank Select Register (BSR).			
Example MOVLB 5	Before Instruction BSR register = 0x02 After Execution BSR register = 0x05			

2.3.4 MOVLW K

University Questions

Q. Explain the instruction: MOVLW 0x04

SPPU - May 16, 2 Marks

Q. Explain the following instructions: MOVLW k

SPPU - Dec. 17, 2 Marks

Q. Explain the instruction : (iii) MOVLW 0XA0

SPPU - Oct. 19, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MOVLW	K	k → W	None	1
Description	Move literal to WREG The eight-bit literal 'k' is loaded into the W			
Example MOVLW 0xF5	After Instruction W = 0xF5			

2.3.5 MOVWF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MOVWF	f,a	W → f	None	1



Description	Move WREG to f Move data from W to register 'f'. Location 'f' can be anywhere in the 256-byte bank. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default)
Example MOVWF REG, 0	Before Instruction W = 0x4F REG = 0xFF After Instruction: W = 0x4F REG = 0x4F

2.3.6 SWAPP

University Question

Q. Explain following instruction in detail: SWAPP 0x56.

SPPU - Dec. 14, 1 Mark

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SWAPP	F,d,a	f<3:0> \rightarrow dest<7:4>, f<7:4> \rightarrow dest<3:0>	None	1

Description	SWAP nibbles in f The upper and lower nibbles of register 'f' are exchanged. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default)
Example SWAPFREG 1, 0	Before Instruction REG = 0x53 After Instruction REG = 0x35

2.3.7 LFSR

University Question

Q. Explain following instruction with flags they are affecting: LFSR

SPPU - Dec. 15, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
LFSR	F,k	k \rightarrow FSRF	None	2

Description	Move literal (12-bit) 2 nd word To FSRx1 st word The 12-bit literal 'k' is loaded into the File Select Register pointed to by 'f'.
Example LFSR 2, 0x3AB	After Instruction FSR2H = 0x03 FSR2L = 0xAB

2.3.8 PUSH

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
PUSH	None	PC + 2 \rightarrow TOS	None	1

Description	Push top of return stack (TOS) The PC + 2 is pushed onto the top of the return stack. The previous TOS value is pushed down on the stack. This instruction allows implementing a software stack by modifying TOS and then pushing it onto the return stack.
Example PUSH	Before Instruction TOS = 00345Ah PC = 000124h After Instruction PC = 000126h TOS = 000126h Stack (1 level down) = 00345Ah

2.3.9 POP

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
POP	None	TOS → bit bucket	None	1
Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
POP				1
Description	Pop top of return stack (TOS) The TOS value is pulled off the return stack and is discarded. The TOS value then becomes the previous value that was pushed onto the return stack. This instruction is provided to enable the user to properly manage the return stack to incorporate a software stack.			
Example POP GOTO NEW	Before Instruction TOS = 0031A2h Stack (1 level down) = 014332h After Instruction TOS = 014332h PC = NEW			

2.3.10 CLRF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
CLRF	f,a	000h → f 1 → Z	Z	1
Description	Clear f Clears the contents of the specified register. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).			
Example CLRFFLAG_REG,1	Before Instruction FLAG_REG = 0x5A After Instruction FLAG_REG = 0x00			

2.3.11 SETF

University Questions	SPPU - May 15, 2 Marks
Q. Explain following instruction with example: SETF.	



Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SETF	f,a	FFh → f	None	1

Description	Set f The contents of the specified register are set to FFh. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Example SETF REG, 1	Before Instruction REG = 0x5A After Instruction REG = 0xFF

2.4 Arithmetic Instructions

2.4.1 ADDLW

University Question

Q. Explain the following instruction in detail: ADDLW.

SPPU - Dec.16, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
ADDLW	K	W + k → W	C,DC,Z,OV,N	1

Description	Add literal and WREG The contents of the W are added to the 8-bit literal 'k' and the result is placed in W.
Example ADDLW 0x25	Before Instruction W = 0x10 After Instruction W = 0x35

2.4.2 ADDWF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
ADDWF	f,d,a	W + f → dest	C, DC, Z, OV, N	1

Description	Add WREG and f Add W to register 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR is used.
Example ADDWF REG, 0, 0	Before Instruction W = 0x17 REG = 0xC2 After Instruction W = 0xD9 REG = 0xC2

2.4.3 ADDWFC

University Questions

Q. Explain following instruction with example: ADDWFC

SPPU - May 15, 2 Marks

Q. Explain the following instruction: ADDWFC 0x20,0,0.

SPPU - Aug. 17 (In Sem.), 3 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
ADDWFC	F,d,a	W + f + C → dest	C, DC, Z, OV, N	1
Description	Add WREG and Carry bit to f Add W, the Carry flag and data memory location 'f'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed in data memory location 'f'. If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR will not be overridden.			
Example ADDWFC REG, 0, 1	Before Instruction Carry bit = 1 REG = 0x02 W = 0x4D After Instruction Carry bit = 0 REG = 0x02 W = 0x50			

2.4.4 DAW

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
DAW		If [W<3:0>>9] or [DC = 1] then W<3:0>+ 6 → W<3:0>; else W<3:0> → W<3:0>; If [W<7:4>>9] or [C = 1] then W<7:4>+ 6 → W<7:4>; else W<7:4> → W<7:4>; (unsigned comparison)	C	1
Description	Decimal Adjust WREG DAW adjusts the eight-bit value in W, resulting from the earlier addition of two variables (each in packed BCD format) and produces a correct packed BCD result.			
Example DAW DAW	Before Instruction W = 0xA5 C = 0 DC = 0 After Instruction W = 0x05 C = 1 DC = 0 Before Instruction W = 0xCE C = 0 DC = 0 After Instruction W = 0x34 C = 1 DC = 0			

2.4.5 SUBLW

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SUBLW	K	k - [W] → W	N, OV, C, DC, Z	1



Description	Subtract WREG from literal W is subtracted from the eight-bit literal 'k'. The result is placed in W.
Example 1. SUBLW 0x02	<p>Before Instruction W = 1 C = ?</p> <p>After Instruction W = 1 C = 1 ; result is positive Z = 0 N = 0</p>
Example 2. SUBLW 0x02	<p>Before Instruction W = 2 C = ?</p> <p>After Instruction W = 0 C = 1 ; result is zero Z = 1 N = 0</p>
Example 3. SUBLW 0x02	<p>Before Instruction W = 3 C = ?</p> <p>After Instruction W = FF; (2's complement) C = 0 ; result is negative Z = 0 N = 1</p>

2.4.6 SUBWF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SUBWF	F,d,a	f - W → dest	N, OV, C, DC, Z	1

Description	Subtract WREG from f Subtract W from register 'f' (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Example 1. SUBWF REG, 1, 0	<p>Before Instruction REG = 3 W = 2 C = ?</p> <p>After Instruction REG = 1 W = 2 C = 1 ; result is positive Z = 0 N = 0</p>
Example 2. SUBWF REG, 0, 0	<p>Before Instruction REG = 2 W = 2 C = ?</p> <p>After Instruction REG = 2 W = 0 C = 1 ; result is zero Z = 1 N = 0</p>
Example 3. SUBWF REG, 1, 0	<p>Before Instruction REG = 1 W = 2 C = ?</p> <p>After Instruction REG = FFh; (2's complement) W = 2 C = 0 ; result is negative Z = 0 N = 1</p>

2.4.7 SUBWFB

University Question

Q. Explain following instruction in detail: SUBWFB 0x53, W

SPPU - Dec. 14, 1 Mark

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SUBWFB	F,d,a	f - W - C → dest	N, OV, C, DC, Z	1

Description	Subtract WREG from f with borrow Subtract W and the Carry flag (borrow) from register 'f' (2's complement method). If 'd' is '0' the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1' then the bank will be selected as per the BSR value (default).			
Example 1. SUBWFB REG, 1, 0	<p>Before Instruction</p> <p>REG = 0x19 (0001 1001) W = 0x0D (0000 1101)</p> <p>C = 1</p> <p>After Instruction</p> <p>REG = 0x0C (0000 1011) W = 0x00 (0000 1101)</p> <p>C = 1 Z = 0</p> <p>N = 0 ; result is positive</p>			
Example 2. SUBWFB REG, 0, 0	<p>Before Instruction</p> <p>REG = 0x11 (0001 1011) W = 0x1A (0001 1010)</p> <p>C = 0</p> <p>After Instruction</p> <p>REG = 0x1B (0001 1011) W = 0x00</p> <p>C = 1 ; result is zero Z = 1</p> <p>N = 0</p>			
Example 3. SUBWFB REG, 1, 0	<p>Before Instruction</p> <p>REG = 0x03(0000 0011) W = 0x0E(0000 1101)</p> <p>C = 1</p> <p>After Instruction</p> <p>REG = 0xF5 (1111 0101) ;[2's comp]</p> <p>W = 0x0E ;(0000 1101) C = 0</p> <p>Z = 0 N = 1 ; result is negative</p>			

2.4.8 SUBFWB

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SUBFWB	F,d,a	W - f - C → dest	N, OV, C, DC, Z	1

Description	Subtract f from WREG with borrow Subtract register 'f' and Carry flag (borrow) from W (2's complement method). If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored in register 'f' (default). If 'a' is '0' the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).			
Example 1. SUBFWB REG, 1, 0	<p>Before Instruction</p> <p>REG = 3 W = 2 C = 1</p> <p>After Instruction</p> <p>REG = FF W = 2 C = 0</p> <p>Z = 0 N = 1; result is negative</p>			
Example 2. SUBFWB REG, 0, 0	<p>Before Instruction</p> <p>REG = 2 W = 5 C = 1</p>			



Example 3. SUBFWB REG, 1, 0	After Instruction REG = 2 W = 3 C = 1 Z = 0 N = 0; result is positive				
	Before Instruction	REG = 1	W = 2	C = 0	
	After Instruction	REG = 0	W = 2	C = 1	
		Z = 1; result is zero		N = 0	

2.4.9 INCF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
INCF	f,d,a	f + 1 → dest	C,DC,Z,OV,N	1

Description	Increment f The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example INCF CNT, 1, 0	Before Instruction CNT = 0xFF Z = 0 C = ? DC = ? After Instruction CNT = 0x00 Z = 1 C = 1 DC = 1

2.4.10 INCFSZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
INCFSZ	f,d,a	f + 1 → dest, skip if result = 0	None	1(2 or 3)

Description	Increment f, Skip if 0 The contents of register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example HERE INCFSZ CNT, 1, 0 NZERO : ZERO :	Before Instruction PC = Address (HERE) After Instruction CNT = CNT + 1 If CNT = 0; PC = Address (ZERO) If CNT ≠ 0; PC = Address (NZERO)

2.4.11 INCFSNZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
INCFSNZ	f,d,a	f + 1 → dest, skip if result ≠ 0	None	1(2 or 3)

Description	Increment f, Skip if not 0 The contents register 'f' are incremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is not '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the 'BSR value'. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example HERE INCFSNZ CNT, 1, 0 NZERO: ZERO :	Before Instruction PC = Address (Here) After Instruction REG = REEG + 1 If REG = 0; If Reg = Address (NZERO) PC = Address (NZERO) If REG = 0; PC = Address (zero)

2.4.12 DECF

University Question

Q. Explain following instruction with example: DECF.

SPPU - May 15, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
DECF	f,d,a	f - 1 → dest	C, DC, Z, OV, N	1

Description	Decrement f Decrement registers 'f'. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example DECF CNT, 1, 0	Before Instruction CNT = 0x01 Z = 0 After Instruction CNT = 0x00 Z = 1

2.4.13 DECFSZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
DECFSZ	f,d,a	f - 1 → dest skip if result = 0	None	1(2 or 3)

Description	Decrement f, Skip if 0 The contents of register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example HERE DECFSZ CNT, 1, 1 GOTO LOOP CONTINUE	Before Instruction PC = Address (HERE) After Instruction CNT = CNT - 1 If CNT = 0; PC = Address (CONTINUE) If CNT ≠ 0; PC = Address (HERE+2)

2.4.14 DECFSNZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
DECFSNZ	f,d,a	f - 1 → dest. skip if result ≠ 0	None	1(2 or 3)

Description	Decrement f, Skip if not 0 The contents register 'f' are decremented. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If the result is not '0', the next instruction which is already fetched is discarded and a NOP is executed instead, making it a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the "BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example Here DCFSNZ T Zero : NZERO :	Before instruction Temp = ? After instruction Temp = Temp - 1, If Temp = 0; PC = Address (Zero) If Temp ≠ 0; P = Address (Nzero)

2.4.15. MULLW

University Question

Q. Explain following instruction with flags they are affecting: MULW

SPPU - Dec. 15, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MULH.W	K	$W \times k \rightarrow PRODH:PRODL$	None	1

Description	<p>Multiply literal with WREG</p> <p>An unsigned multiplication is carried out between the contents of W and the 8-bit literal 'k'. The 16-bit result is placed in PRODH: PRODL register pair. PRODH contains the high byte. W is unchanged.</p> <p>None of the status flags are affected.</p> <p>Note that neither overflow nor carry is possible in this operation. A zero result is possible but not detected.</p>								
Example MULLW 0xC4	<p>Before Instruction</p> <table style="margin-left: 100px;"> <tr> <td>W = 0xE2</td> <td>PRODH = ?</td> </tr> <tr> <td>PRODL = ?</td> <td></td> </tr> </table> <p>After Execution</p> <table style="margin-left: 100px;"> <tr> <td>W = 0xE2</td> <td>PRODH = 0xAD</td> </tr> <tr> <td>PRODL = 0x08</td> <td></td> </tr> </table>	W = 0xE2	PRODH = ?	PRODL = ?		W = 0xE2	PRODH = 0xAD	PRODL = 0x08	
W = 0xE2	PRODH = ?								
PRODL = ?									
W = 0xE2	PRODH = 0xAD								
PRODL = 0x08									

2.4.16 MULWF

University Question

Q. Explain the instruction : (i) MULWF 0x20,0.

SPPU - Oct. 19, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
MULWF	f,a	$W \times f \rightarrow PRODH:PRODL$	None	1

Description	Multiply WREG with f An unsigned multiplication is carried out between the contents of W and the register file location 'f'. The 16-bit result is stored in the PRODH: PRODL register pair. PRODH contains the high byte. Both W and 'f' are unchanged. None of the status flags are affected. Note that neither overflow nor carry is possible in this operation. A zero result is possible but not detected. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a = 1, then the bank will be selected as per the BSR value (default).
Example MULWF REG, 1	Before Instruction W = 0xC4 REG = 0xB5 PRODH = ? PRODL = ? After Instruction W = 0xC4 REG = 0xB5 PRODH = 0x8A PRODL = 0x94

2.5 Logical Instructions

2.5.1 ANDLW

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
ANDLW	K	W.AND. k → W	Z,N	1

Description	AND literal with WREG The contents of W are ANDed with the 8-bit literal 'K'. The result is placed in W.
Example ANDLW 0x5F	Before Instruction W = 0xA3 After Instruction W = 0x03

2.5.2 ANDWF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
ANDWF	F,d,a	W.AND. f → dest	Z, N	1

Description	AND WREG with f The contents of W are ANDed with register f. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR will not be overridden (default).
Example ANDWF REG, 0, 0	Before Instruction W = 0x17 REG = 0xC2 After Instruction W = 0x02 REG = 0xC2

2.5.3 IORLW

University Questions

- Q. Explain following instruction in detail: IORLW 0 x 23.
Q. Explain following instruction with example: IORLW

SPPU - Dec. 14, 1 Mark

SPPU - May 15, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
IORLW	k	W.OR. k → W	Z,N	1



Description	Inclusive OR literal with WREG The contents of W are OR'ed with the eight-bit literal 'k'. The result is placed in W.			
Example IORLW 0x35	Before Instruction W = 0x9A After Instruction W = 0xBF			

2.5.4 IORWF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
IORWF	f,d,a	W.OR.f → dest	Z,N	1

Description	Inclusive OR WREG with f Inclusive OR W with register 'f'. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).			
Example IORWF RESULT, 0, 1	Before Instruction RESULT = 0x13 W = 0x91 After Instruction RESULT = 0x13 W = 0x93			

2.5.5 XORLW

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
XORLW	K	(W).XOR.k → (W)	Z,N	1

Description	Exclusive OR literal with WREG The contents of W are XOR'ed with the 8-bit literal 'k'. The result is placed in W.			
Example XORLW 0xAF	Before Instruction W = 0xB5 After Instruction W = 0x1A			

2.5.6 XORWF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
XORWF	F,d,a	W.XOR.f → dest	N,Z	1

Description	Exclusive OR WREG with f Exclusive OR the contents of W with register 'f'. If 'd' is '0' the result is stored in W. If 'd' is '1' the result is stored back in register 'f' (default). If 'a' is '0' the Access bank will be selected, overriding the BSR value. If 'a' is '1' then the bank will be selected as per the BSR value (default).			
Example XORWF REG, 1, 0	Before instruction REG = 0xaf W = 0xb5 After instruction REG = 0x1a W = 0xb5			

2.5.7 COMF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
COMF	f,d,a	f → dest	Z,N	1



Description	Complement f The contents of register 'f' are complemented. If 'd' is '0', the result is stored in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).		
Example COMF REG, 0,0	Before Instruction REG = 0x13	After Instruction REG = 0x13	W = 0xEC

2.5.8 NEGF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
NEGF	F,a	f + 1 → f	N, OV, C, DC, Z	1

Description	Negate f Location 'f' is negated using two's complement. The result is placed in the data memory location 'f'. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value.		
Example NEGF REG, 1	Before Instruction REG = 0011 1010 [0x3A]	After Instruction REG = 1100 0110 [0xC6]	

2.5.9 CPFSEQ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
CPFSEQ	f,a	f - W, Skip if f = W (unsigned comparison)	None	1(2 or 3)

Description	Compare f with WREG, skip = Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction. If 'f' = W, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).		
Example HERE CPFSEQ 0x25, A EQUAL: NEQUAL	Before Instruction PC Address = HERE Reg 0x25 = ? After Instruction If Reg 0x25 = W; If Reg 0x25 = W;	W = ? PC ≠ Address (EQUAL) PC = Address (NEQUAL)	

2.5.10 CPFSGT

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
CPFSGT	f,a	f - W Skip if f > W (unsigned comparison)	None	1(2 or 3)

Description	Compare f with WREG, skip > Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction. If the contents of 'f' are greater than the contents of WREG, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).		
-------------	--	--	--



Example	Before Instruction PC = Address (HERE) W = ?	
HERE		
CPFSGT REG, 0		After Instruction If REG > W; PC = Address (GREATER)
NGREATER :		If REG ≤ W, PC = Address (NGREATER)
GREATER:		

2.5.11 CPFSLT

University Question

Q. Explain following instruction with flags they are affecting: CPFSLT.

SPPU - Dec. 15, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
CPFSLT	f,a	(- W Skip if f < W (unsigned comparison))	None	1(2 or 3)

Description	Compare f with WREG, skip < Compares the contents of data memory location 'f' to the contents of W by performing an unsigned subtraction. If the contents of 'f' are less than the contents of W, then the fetched instruction is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected. If 'a' is '1', the BSR will not be overridden (default).
Example HERE CPFSLT REG, 1 NLESS : LESS :	Before Instruction PC = Address (HERE) W = ? After Instruction If REG < W; PC = Address (LESS) If REG ≥ W; PC = Address (NLESS)

2.5.12 TSTFSZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TSTFSZ	f,a	skip if f = 0	None	1(2 or 3)

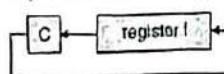
Description	Test f, Skip if 0 if 'f' = 0, the next instruction, fetched during the current instruction execution is discarded and a NOP is executed, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Example HERE TSTFSZ CNT, 1 NZERO: ZERO :	Before Instruction PC = Address (HERE) After Instruction If CNT = 0x00, PC = Address (ZERO) If CNT ≠ 0x00, PC = Address (NZERO)

2.6 Rotate Instructions

2.6.1 RLCF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RLCF	F,d,a	f<n>→dest<n+1>, f<7>→C, C→dest<0>	C, N, Z	1

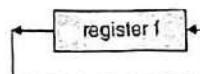
Description	Rotate Left f through Carry The contents of register 'f' are rotated one bit to the left through the Carry flag. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example RLCF REG, 0, 0	Before Instruction REG = 1110 0110 C = 0 After Instruction REG = 1110 0110 W = 1100 1100 C = 1



2.6.2 RLNCF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RLNCF	F,d,a	f<n>→dest<n+1>, [f<7>]→dest<0>	Z,N	1

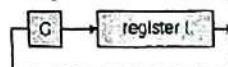
Description	Rotate Left f (No Carry) The contents of register 'f' are rotated one bit to the left. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is stored back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
Example RLNCFREG, 1, 0	Before Instruction REG = 1010 1011 After Instruction REG = 0101 0111



2.6.3 RRCF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RRCF	F,d,a	f<n>→dest<n-1>, f<0>→C, C→dest<7>	C,Z,N	1

Description	Rotate Right f through Carry The contents of register 'f' are rotated one bit to the right through the Carry flag. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).
-------------	--





Example
RRCF REG, 0, 0

Before Instruction

REG = 1110 0110 C = 0

After Instruction

REG = 1110 0110 W = 0111 0011 C = 0

2.6.4 RRNCF

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RRNCF	F,d,a	f<n>→dest<n-1>, f<0>→dest<7>	Z,N	1

Description

Rotate Right f (No Carry)

The contents of register 'f' are rotated one bit to the right. If 'd' is '0', the result is placed in W. If 'd' is '1', the result is placed back in register 'f' (default). If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' is '1', then the bank will be selected as per the BSR value (default).



Example

RRNCF REG, 1, 0

Before Instruction

REG = 1101 0111

After Instruction

REG = 1110 1011

2.7 Branch Instructions

2.7.1 BC

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BC	N	if Carry bit is '1' PC + 2 + 2n → PC	None	1(2)

Description

Branch if Carry

If the carry bit is '1' then the program will branch. The 2's complement number '2n' is added to the PC.

Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Example

HERE BC 5

Before Instruction

PC = address (HERE)

After Instruction

If Carry = 1;

PC = address (HERE + 12)

If Carry = 0;

PC = address (HERE + 2)

2.7.2 BN

University Question

Q. Explain the following instruction with suitable example: i) BN n

SPPU - May 19, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BN	n	if Negative bit is '1' PC + 2 + 2n → PC	None	1(2)

Description

Branch if Negative

If the Negative bit is '1' then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.

Example HERE BN JUMP	Before Instruction PC = address (HERE)
	After Instruction If Negative = 1; PC = address (JUMP) If Negative = 0; PC = address (HERE + 2)

2.7.3 BNC

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BNC	n	if Carry bit is '0' PC + 2 + 2n → PC	None	1(2)

Description	Branch if Not Carry If the carry bit is '0' then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.
Example HERE BNC n	Before Instruction PC = address (HERE) After Instruction If Carry = 0; PC = address (n) If Carry = 1; PC = address (HERE + 2)

2.7.4 BNN

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BNN	n	if Negative bit is '0' PC + 2 + 2n → PC	None	1(2)

Description	Branch if Not Negative If the Negative bit is '0' then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.
Example HERE BNN Jump	Before Instruction PC = address (HERE) After Instruction If Negative = 0; PC = address (Jump) If Negative = 1; PC = address (HERE + 2)

2.7.5 BNOV

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BNOV	N	if Overflow bit is '0' PC + 2 + 2n → PC	None	1(2)

Description	Branch if Not Overflow If the Overflow bit is '0' then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.
--------------------	--

Example HERE BNOV Jump	Before Instruction PC = address (HERE)
	After Instruction If Overflow = 0; PC = address (Jump) If Overflow = 1; PC = address (HERE - 2)

2.7.6 BNZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BNZ	N	jump to target address if Z=0	None	1(2)

Description	Branch if Not Zero
Example	LOC EQU 0x30 MOVF LOC, F BNZ L1

2.7.7 BOV

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BOV	N	If Overflow bit is '0' PC+ 2 + 2n → PC	None	1(2)

Description	Branch if Overflow If the Overflow bit is '0' then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is then a two-cycle instruction.
Example HERE BNOV Jump	Before Instruction PC = address (HERE) After Instruction If Overflow = 0; PC = address (Jump) If Overflow = 1; PC = address (HERE + 2)

2.7.8 BRA

University Question

Q. Explain following instruction in detail: BRA \$

SPPU - Dec 14, 1 Mark

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BRA	N	PC+ 2 + 2n → PC	None	2

Description	Branch Unconditionally Add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is a two-cycle instruction.
Example HERE BRA Jump	Before Instruction PC = address (HERE) After Instruction PC = address (Jump)



2.7.9 GOTO

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
GOTO	N	$k \rightarrow PC<20:1>$	None	2

Description	Go to address 1 st word, 2 nd word GOTO allows an unconditional branch anywhere within the entire 2-Mbyte memory range. The 20-bit value 'k' is loaded into PC<20:1> GOTO is always a two-cycle instruction.
Example GOTO THERE	After Instruction PC = Address (THERE)

2.7.10 BZ

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BZ	N	if Zero bit is '1' $PC + 2 + 2n \rightarrow PC$	None	1(2)

Description	Branch if Zero If the Zero bit is '1', then the program will branch. The 2's complement number '2n' is added to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be $PC + 2 + 2n$. This instruction is then a two-cycle instruction.
Example HERE BZ Jump	Before Instruction PC = address (HERE) After Instruction If Zero = 1; PC = address (Jump) If Zero = 0; PC = address (HERE+2)

2.8 CALL and RETURN Instructions

2.8.1 CALL

University Question

Q. Explain CALL instruction in PIC18.

SPPU - May 15, 3 Marks

Q. Explain the CALL and RETURN instructions of PIC 18 microcontroller.

SPPU - Oct. 19, 6 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
CALL	N,S	$PC + 4 \rightarrow TOS,$ $k \rightarrow PC<20:1>$, If s = 1 $(W) \rightarrow WS,$ $(STATUS) \rightarrow STATUS,$ $(BSR) \rightarrow BSRS$	None	2

Description	Call subroutine 1 st word, 2 nd word Subroutine call of entire 2-Mbyte memory range. First, return address (PC+4) is pushed onto the return stack. If s' = 1, the W, Status and BSR registers are also pushed into their respective shadow registers, WS, STATUS and BSRS. If s' = 0, no update occurs (default). Then, the 20-bit value 'k' is loaded into PC<20:1>. CALL is a two-cycle instruction.
-------------	---



Example HERE CALLTHERE,1	Before Instruction PC = address (HERE)
	After Instruction PC = address (THERE) TOS = address (HERE + 4) WS = W BSRS = BSR STATUSS = STATUS

2.8.2 RCALL

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RCALL	N	PC + 2 → TOS, PC + 2 + 2n → PC	None	2

Description	Relative Call Subroutine call with a jump up to 1K from the current location. First, return address (PC + 2) is pushed onto the stack. Then, add the 2's complement number '2n' to the PC. Since the PC will have incremented to fetch the next instruction, the new address will be PC+2+2n. This instruction is a two-cycle instruction.
Example HERE RCALL Jump	Before Instruction PC = Address (HERE) After Instruction PC = Address (Jump) TOS = Address (HERE+2)

2.8.3 RETURN

University Question	SPPU - May 15, 3 Marks
Q. Explain RETURN instruction in PIC18.	

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RETURN	S	(TOS) → PC, if S = 1 (WS) → W, STATUSS → STATUS, (BSRS) → BSR, PCLATU, PCLATH are unchanged	None	2

Description	Return from Subroutine Return from subroutine. The stack is popped and the top of the stack (TOS) is loaded into the program counter. If 'S' = 1, the contents of the shadow registers, WS, STATUSS and BSRS, are loaded into their corresponding registers, W, Status and BSR. If 'S' = 0, no update of these registers occurs (default).
Example RETURN	After Interrupt PC = TOS

2.8.4 RETFIE

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RETFIE	S	(TOS) → PC, 1 → GIE/GIEH or PEIE/GIE, If s = 1 (WS) → W, (STATUSS) → STATUS, (BSRS) → BSR, PCLATU, PCLATH Are unchanged	GIE/GIEH, PEIE/GIEL	2

Description	Return from interrupt enable Return from Interrupt. Stack is popped and Top-of-Stack (TOS) is loaded into the PC. Interrupts are enabled by setting either the high or low priority global interrupt enable bit, If's' = 1, the contents of the shadow registers, WS, STATUS and BSRS, are loaded into their corresponding registers, W, Status and BSR. If's' = 0, no update of these registers occurs (default).
Example RETFIE 1	After Interrupt PC = TOS W = WS BSR = BSRS STATUS = STATUS GIE/GIEH, PEIE/GIEL = 1

2.8.5 RETLW

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RETLW	K	k → W, (TOS) → PC, PCLATU, PCLATH are unchanged	None	2

Description	Return with literal in WREG W is loaded with the eight-bit literal 'k'. The program counter is loaded from the top of the stack (the return address). The high address latch (PCLATH) remains unchanged.
Example CALL TABLE; W contains table offset value W now has table value : TABLE ADDWF PCL ; W = offset RETLW k0 ; Begin table RETLW k1 ; : RETLW kn ; End of table	Before Instruction W = 0x07 After Instruction W = value of kn

2.9 Table Read/Write Instructions

2.9.1 TBLRD*

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLRD*	None	if TBLRD*, (Prog Mem (TBLPTR)) → TABLAT; TBLPTR - No Change;	None	2

Description	Table Read This instruction is used to read the contents of Program Memory (P.M.). To address the Program Memory, a pointer called Table Pointer (TBLPTR) is used. The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-Mbyte address range.
-------------	--



	<p>TBLPTR[0] = 0 Least Significant Byte of Program Memory Word TBLPTR[0] = 1 Most Significant Byte of Program Memory Word The TBLRD instruction can modify the Value of TBLPTR as follows</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
Example TBLRD *+ ; TBLRD *+ ;	<p>Before Instruction</p> <pre>TABLAT = 0x55 TBLPTR = 0x00A356 MEMORY(0x00A356) = 0x34</pre> <p>After Instruction</p> <pre>TABLAT = 0x34 TBLPTR = 0x00A357</pre> <p>Before Instruction</p> <pre>TABLAT = 0xAA TBLPTR = 0x01A357 MEMORY(0x01A357) = 0x12 MEMORY(0x01A358) = 0x34</pre> <p>After Instruction</p> <pre>TABLAT = 0x34 TBLPTR = 0x01A358</pre>

2.9.2 TBLRD*+

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLRD*+	None	if TBLRD *+, (Prog Mem (TBLPTR)) → TABLAT; (TBLPTR) + 1 → TBLPTR;	None	2

Description	<p>Table Read with post-increment</p> <p>This instruction is used to read the contents of Program Memory (P.M.). To address the Program Memory, a pointer called Table Pointer (TBLPTR) is used. The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-Mbyte address range.</p> <p>TBLPTR[0] = 0 : Least Significant Byte of Program Memory Word TBLPTR[0] = 1: Most Significant</p> <p>Byte of Program Memory Word</p> <p>The TBLRD instruction can modify the value of TBLPTR as follows:</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
Example TBLRD *+ ;	<p>Before Instruction</p> <pre>TABLAT = 0x55 TBLPTR = 0x00A356</pre>



	MEMORY[0x00A356] = 0x34 After Instruction TABLAT = 0x34 TBLPTR = 0x00A357
--	--

2.9.3 TBLRD*-

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLRD*-	None	if TBLRD *-, (Prog Mem (TBLPTR)) → TABLAT; (TBLPTR) - 1 → TBLPTR;	None	2

Description	<p>Table Read with post-decrement</p> <p>This instruction is used to read the contents of Program Memory (P.M.). To address the Program Memory, a pointer called Table Pointer (TBLPTR) is used. The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-Mbyte address range.</p> <p>TBLPTR[0] = 0: Least Significant Byte of Program Memory Word TBLPTR[0] = 1: Most Significant Byte of Program Memory Word</p> <p>The TBLRD instruction can modify the Value of TBLPTR as follows:</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
-------------	--

2.9.4 TBLRD+*

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLRD+*	None	if TBLRD +*, (TBLPTR) + 1 → TBLPTR; ProgMem(TBLPTR) → TABLAT;	None	2

Description	<p>Table Read with pre-increment</p> <p>This instruction is used to read the contents of Program Memory (P.M.). To address the Program Memory, a pointer called Table Pointer (TBLPTR) is used. The TBLPTR (a 21-bit pointer) points to each byte in the program memory. TBLPTR has a 2-Mbyte address range.</p> <p>TBLPTR[0] = 0: Least Significant Byte of Program Memory Word TBLPTR[0] = 1: Most Significant Byte of Program Memory Word</p> <p>The TBLRD instruction can modify the Value of TBLPTR as follows:</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
-------------	---

Example TBLRD +';	<p>Before Instruction</p> <pre> TABLAT = 0xAA TBLPTR = 0x01A357 MEMORY(0x01A357) = 0x12 MEMORY(0x01A358) = 0x34 </pre> <p>After Instruction</p> <pre> TABLAT = 0x34 TBLPTR = 0x01A358 </pre>
------------------------------------	--

2.9.5 TBLWT*

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLWT*	None	If TBLWT*, (TABLAT) → Holding Register; TBLPTR – No Change; Holding Register;	None	2(5)

Description	<p>Table Write</p> <p>This instruction uses the 3 LSBs of TBLPTR to determine which of the 8 holding registers the TABLAT is written to. The holding registers are used to program the contents of program Memory. The TBLPTR (a 21-bit pointer) points to each byte in the Program Memory. TBLPTR has a 2-mbyte address range. The LSB of the TBLPTR Selects which byte of the program memory location to access.</p> <p>TBLPTR[0] = 0: Least Significant Byte of Program Memory Word</p> <p>TBLPTR[0] = 1: Most Significant Byte of Program Memory Word</p> <p>The TBLWT instruction can modify the value of TBLPTR as follows: no change post-increment post-decrement pre-increment</p>
Example	<p>Example 1: TBLWT *+ ;</p> <p>Before Instruction</p> <pre> TABLAT = 0x55 TBLPTR = 0x00A356 HOLDING REGISTER (0x00A356) = 0xFF </pre> <p>After Instruction (table write completion)</p> <pre> TABLAT = 0x55 TBLPTR = 0x00A357 HOLDING REGISTER (0x00A356) = 0x55 </pre> <p>Example 2: TBLWT +* ;</p> <p>Before Instruction</p> <pre> TABLAT = 0x34 TBLPTR = 0x01389A HOLDING REGISTER </pre>

	$(0x01389A) = 0xFF$ HOLDING REGISTER $(0x01389B) = 0xFF$ After Instruction(table write completion) $TABLAT = 0x34$ $TBLPTR = 0x01389B$ HOLDING REGISTER $(0x01389A) = 0xFF$ HOLDING REGISTER $(0x01389B) = 0x34$
--	---

2.9.6 TBLWT*+

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLWT*+	None	if TBLWT *+, $(TABLAT) \rightarrow$ Holding Register; $(TBLPTR) + 1 \rightarrow TBLPTR;$	None	2(5)

Description	<p>Table Write with post-increment</p> <p>This instruction uses the 3 LSBs of TBLPTR to determine which of the 8 holding registers the TABLAT is written to. The holding registers are used to program the contents of program Memory. The TBLPTR (a 21-bit pointer) points to each byte in the Program Memory. TBLPTR has a 2-mbyte address range. The LSB of the TBLPTR Selects which byte of the program memory location to access.</p> <p>$TBLPTR[0] = 0$: Least Significant Byte of Program Memory Word</p> <p>$TBLPTR[0] = 1$: Most Significant Byte of Program Memory Word</p> <p>The TBLWT instruction can modify the value of TBLPTR as follows:</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
Example	<p>TBLWT*+;</p> <p>Before Instruction</p> $TABLAT = 0x55$ $TBLPTR = 0x00A356$ HOLDING REGISTER $(0x00A356) = 0xFF$ <p>After Instruction (table write completion)</p> $TABLAT = 0x55$ $TBLPTR = 0x00A357$ HOLDING REGISTER $(0x00A356) = 0x55$

2.9.7 TBLWT⁻

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLWT ⁻	None	if TBLWT ⁻ , (TABLAT) → Holding Register; (TBLPTR) - 1 → TBLPTR;	None	2(5)

Description	<p>Table Write with post-decrement</p> <p>This instruction uses the 3 LSBs of TBLPTR to determine which of the 8 holding registers the TABLAT is written to. The holding registers are used to program the contents of program Memory. The TBLPTR (a 21-bit pointer) points to each byte in the Program Memory. TBLPTR has a 2-mbyte address range. The LSB of the TBLPTR Selects which byte of the program memory location to access.</p> <p>TBLPTR[0] = 0: Least Significant Byte of Program Memory Word</p> <p>TBLPTR[0] = 1: Most Significant Byte of Program Memory Word</p> <p>The TBLWT instruction can modify the value of TBLPTR as follows:</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
-------------	--

2.9.8 TBLWT⁺

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
TBLWT ⁺	None	if TBLWT ⁺ , (TBLPTR) + 1 → TBLPTR; (TABLAT) → Holding Register;	None	2(5)

Description	<p>Table Write with pre-increment</p> <p>This instruction uses the 3 LSBs of TBLPTR to determine which of the 8 holding registers the TABLAT is written to. The holding registers are used to program the contents of program Memory. The TBLPTR (a 21-bit pointer) points to each byte in the Program Memory. TBLPTR has a 2-mbyte address range. The LSB of the TBLPTR Selects which byte of the program memory location to access.</p> <p>TBLPTR[0] = 0: Least Significant Byte of Program Memory Word</p> <p>TBLPTR[0] = 1: Most Significant Byte of Program Memory Word</p> <p>The TBLWT instruction can modify the value of TBLPTR as follows:</p> <ul style="list-style-type: none"> no change post-increment post-decrement per-increment
-------------	---

Example TBLWT +* ;	<p>Before Instruction</p> <p>TABLAT = 0x34 TBLPTR = 0x01389A HOLDING REGISTER (0x01389A) = 0xFF HOLDING REGISTER (0x01389B) = 0xFF</p> <p>After Instruction (table write completion)</p> <p>TABLAT = 0x34 TBLPTR = 0x01389B HOLDING REGISTER (0x01389A) = 0xFF HOLDING REGISTER (0x01389B)= 0x34</p>
-------------------------------------	--

2.10 Machine Control Instructions

2.10.1 CLRWDT

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
CLRWDT		000h → WDT, 000h → WDT postscaler 1 → $\overline{\text{TO}}$, 1 → $\overline{\text{PD}}$	$\overline{\text{TO}}$, $\overline{\text{PD}}$	1

Description	<p>Clear Watchdog Timer</p> <p>CLRWDT instruction resets the Watchdog Timer. It also resets the postscaler of the WDT. Status bits — TO and PD are set.</p>						
Example : CLRWDT	<p>Before Instruction</p> <p>WDT Counter = ?</p> <p>After Instruction</p> <table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">WDT Counter = 0x00</td> <td style="width: 50%;">WDT Postscaler = 0</td> </tr> <tr> <td>—</td> <td>—</td> </tr> <tr> <td>TO = 1</td> <td>PD = 1</td> </tr> </table>	WDT Counter = 0x00	WDT Postscaler = 0	—	—	TO = 1	PD = 1
WDT Counter = 0x00	WDT Postscaler = 0						
—	—						
TO = 1	PD = 1						

2.10.2 NOP

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
NOP	None	No operation	None	1
		Description	No operation	
		Example	None	



2.10.3 SLEEP

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
SLEEP	None	00h → WDT, 0 → WDT postscaler, 1 → TO, 0 → PD	TO, PD	1

Description	Go into Standby mode The power-down status bit, (PD) is cleared. The Time-out status bit (TO) is set. Watchdog Timer and its postscaler are cleared. The processor is put into Sleep mode with the oscillator stopped.		
Example SLEEP	Before Instruction TO = ? PD = ? After Instruction TO = 1 PD = 0 ; If WDT causes wake-up, this bit is cleared.		

2.10.4 RESET

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
RESET	None	Reset all registers and flags that are affected by a MCLR Reset.	All	1

Description	Software device Reset This instruction provides a way to execute a MCLR Reset in software.		
Example RESET	After Instruction Registers = Reset Value Flags * = Reset Value		

2.11 Bit Addressable Instructions

2.11.1 BCF

University Question

Q. Explain the following instruction with suitable example: BCF f,b,a.

SPPU - May 19, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BCF	F,d,a	0 → f	None	1

Description	Bit Clear f Bit 'b' in register 'f' is cleared. If 'a' is '0', the Acess Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).		
Example BCF FLAG_REG, 7, 0	Before Instruction FLAG_REG = 0xC7 After Instruction FLAG_REG = 0x47		



2.11.2 BSF

University Questions

- Q. Explain following instruction with example: BSF SPPU - May 15, 2 Marks
 Q. Explain instruction: BSF f, b, a SPPU - Oct. 16 (In Sem.), 2 Marks
 Q. Explain the following instruction. (i) BSF PORTD, 0, 0. SPPU - Aug. 17 (In Sem.), 3 Marks
 Q. Explain the following instructions: BSF PORTD, 0 SPPU - May 18, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BSF	F,b,a	$1 \rightarrow f$	None	1

Description	Bit Set f Bit 'b' in register 'f' is set. If 'a' is '0', Access Bank will be selected overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value.		
Example BSF FLAG_REG, 7, 1	Before Instruction FLAG_REG = 0x0A	After Instruction FLAG_REG = 0x8A	

2.11.3 BTFSC

University Questions

- Q. Explain following instruction in detail: BTFSC PORTB, 3 SPPU - Dec. 14, 1 Mark
 Q. Explain the instruction: BTFSC f, b, a SPPU - May 16, 2 Marks
 Q. Explain the instruction : (ii) BTFSC 0X20,1,0 SPPU - Oct. 19, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BTFSC	F,b,a	skip if $f = 0$	None	1 (2 or 3)

Description	Bit Test f. Skip if Clear If bit 'b' in register 'f' is '0', then the next instruction is skipped. If bit 'b' is '0', then the next instruction fetched during the current instruction execution is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).		
Example HERE BTFSC FLAG, 1, 0 FALSE: TRUE :	Before Instruction PC = address (HERE)	After Instruction If $\text{FLAG}<1> = 0$ PC = address (TRUE) If $\text{FLAG}<1> = 1$; PC = address (FALSE)	

2.11.4 BTFSS

University Question

- Q. Explain the instruction BTFSS f, d, a. SPPU - Aug. 15 (In Sem.), 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BTFSS	F,b,a	skip if $(f) = 1$	None	1 (2 or 3)



Description	Bit Test f, Skip if Set If bit 'b' in register 'T' is '1', then the next instruction is skipped. If bit 'b' is '1', then the next instruction fetched during the current instruction execution is discarded and a NOP is executed instead, making this a two-cycle instruction. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example HERE BTFS FLAG, 1, 0 FALSE: TRUE :	Before Instruction PC = Address (HERE) After Instruction If FLAG<1> = 0 PC = address (FALSE) If FLAG<1> = 1; PC = address (TRUE)

2.11.5 BTG

University Question

Q. Explain the following instructions: BTG f,b,a

SPPU - Dec. 17, 2 Marks

Mnemonic	Operands	Symbolic Operation	Flags affected	Machine Cycles
BTG	F,d,a	f → f	None	1

Description	Bit Toggle BIT 'b' in data memory location 'T' is inverted. If 'a' is '0', the Access Bank will be selected, overriding the BSR value. If 'a' = 1, then the bank will be selected as per the BSR value (default).
Example BTG 0x25, 7, A	Before Instruction Reg 0x25 = 0111 0101 [0x25] After Instruction Reg 0x25 = 1111 0101 [0xF5]

2.12 Assembly Language Programs

Assembly language programming is a method of writing programs using instructions that are the English language equivalent of machine code. The syntax of each instruction is structured to allow direct translation to machine code by the assembler.

Assembly Language Program Structure

A PIC18 assembly program consists of three types of statements:

- Assembler directive:** Assembler directives are assembler commands that are used to direct the assembler to do something. It may be used for its input, output, and data allocation. An assembly program must be terminated with an END directive. Any statement after the END directive will be ignored by the assembler.
- Assembly language instruction:** These instructions are PIC18 instructions. Some are defined with labels.

The PIC18 MCU allows us to use up to 27 different instructions as discussed earlier.

- Comments:** These are statements for the programmer to be referred later to understand the program flow and its functionality. It may also be used by some other programmer to interpret the program.

Each line of the assembly program source file may consist of up to four fields:

- Label:** Labels must start in column 1.
- Mnemonic:** Mnemonics may start in column 2 or beyond.
- Operand(s):** Operands follow the mnemonic.
- Comment:** Comments may follow the operands, mnemonics, or labels and can start in any column.

The maximum width of a column is 256 characters. The programmer must use a colon to separate the label and the mnemonic and use space to separate the mnemonic and the operand(s). Operands are to be separated by commas.

A label must begin with an alphabetic character or an underscore (_) and may contain alphanumeric characters, the underscore and the question mark (?). Labels can be up to 32 characters long. The mnemonic field can be either an assembly instruction mnemonic or an assembler directive. If there is a label on the same line, instructions must be separated from that label by a colon. If an operand field is present, it follows the mnemonic field. The operand field may contain operands for instructions and arguments in case of assembler directives. The comment field that begins with a semicolon (;) is optional and is added for documentation purpose. All characters followed by the semicolon are ignored through the end of the line.

For example, let us identify the four fields in the following source statement `too-low addlw0x02; increment WREG by 2.`

The four fields in the given source statements are as follows:

- (a) too-low is a label.
- (b) addlw is an instruction mnemonic.
- (c) 0x02 is an operand.
- (d) Increment WREG by 2 is a comment.

2.13 Development Tools for PIC Microcontrollers

The PIC16/17 microcontrollers are supported with a full range of hardware and software development tools.

- PICMASTER/PICMASTER CE Real-Time In-Circuit Emulator
- ICEPIC Low-Cost PIC16C5X and PIC16CXXX In-Circuit Emulator
- PRO MATE II Universal Programmer
- PICSTART Plus Entry-Level Prototype Programmer
- PICDEM-1 Low-Cost Demonstration Board
- PICDEM-2 Low-Cost Demonstration Board
- PICDEM-3 Low-Cost Demonstration Board
- MPASM Assembler
- MPLAB-SIM Software Simulator
- MPLAB-C (C Compiler)
- Fuzzy logic development system (fuzzyTECH-MP)

2.14 MPLAB Integrated Development Environment Software

The MPLAB IDE Software brings an ease of software development previously unseen in the 8 bit microcontroller market. MPLAB is a windows based application which contains:

- A full featured editor
- Three operating modes
 - o Editor
 - o Emulator
 - o Simulator
- A project manager
- Customizable tool bar and key mapping
- A status bar with project information
- Extensive on-line help
- MPLAB allows you to Edit your source files (either assembly or 'C')
- One touch assemble (or compile) and download to PIC16/17 tools (automatically updates all project information)
- Debug using:
 - o Source files
 - o Absolute listing file
- Transfer data dynamically via DDE (soon to be replaced by OLE)
- Run up to four emulators on the same PC.

The ability to use MPLAB with Microchip's simulator allows a consistent platform and the ability to easily switch from the low cost simulator to the full featured emulator with minimal retraining due to development tools.

2.14.1 Assembler (MPASM)

University Question

Q. Explain the use of assembler

SPPU - Oct 16 (In Sem.), 2 Marks

The MPASM Universal Macro Assembler is a PC hosted symbolic assembler. It supports all Microchip microcontroller series including the PIC12C5XX, PIC14000, PIC16C5X, PIC16CXXX, and PIC17CXX families.



MPASM offers full featured Macro capabilities, conditional assembly, and several source and listing formats. It generates various object code formats to support Microchip's development tools as well as third party programmers.

MPASM allows full symbolic debugging from PICMASTER, Microchip's Universal Emulator System.

MPASM has the following features to assist in developing software for specific use applications.

- Provides Translation of Assembler source code to object code for all Microchip microcontrollers.
- Macro assembly capability.
- Produces all the files (Object, Listing, Symbol, and special) required for symbolic debug with Microchip's emulator systems.
- Supports Hex (default), Decimal and Octal source and listing formats.

MPASM provides a rich directive language to support programming of the PIC16/17. Directives are helpful in making the development of your assemble source code shorter and more maintainable.

2.14.2 Software Simulator (MPLAB-SIM)

University Question

Q. Explain simulator.

SPPU - Aug. 14(In Sem.), 2 Marks

The MPLAB-SIM Software Simulator allows code development in a PC host environment. It allows the user to simulate the PIC16/17 series microcontrollers on an instruction level. On any given instruction, the user may examine or modify any of the data areas or provide external stimulus to any of the pins. The input/output radix can be set by the user and the execution can be performed in; single step, execute until break, or in a trace mode.

MPLAB-SIM fully supports symbolic debugging using MPLAB-C and MPASM. The Software Simulator offers the low cost flexibility to develop and debug code outside of the laboratory environment making it an excellent multi-project software development tool.

2.14.3 C Compiler (MPLAB-C)

University Questions

Q. Explain compiler.

SPPU - Aug.14(In Sem.), 2 Marks

Q. Explain the use of compiler.

SPPU - Oct. 16(In Sem.), 2 Marks

The MPLAB-C Code Development System is a complete 'C' compiler and integrated development environment for Microchip's PIC16/17 family of microcontrollers. The compiler provides powerful integration capabilities and ease of use not found with other compilers. For easier source level debugging, the compiler provides symbol information that is compatible with the MPLAB IDE memory display (PICMASTER emulator software versions 1.13 and later).

2.15 Introduction to Assembly Language Programming and its Structure

Assembly language programming is a method of writing programs using instructions that are the English language equivalent of machine code. The syntax of each instruction is structured to allow direct translation to machine code by the assembler.

2.15.1 Assembly Language Program Structure

A PIC18 assembly program consists of three types of statements:

1. **Assembler directives:** Assembler directives are assembler commands that are used to direct the assembler to do something. It may be used for its input, output, and data allocation. An assembly program must be terminated with an END directive. Any statement after the END directive will be ignored by the assembler.
2. **Assembly language instructions:** These instructions are PIC18 instructions. Some are defined with labels. The PIC18 MCU allows us to use up to 77 different instructions as discussed earlier.
3. **Comments:** These are statements for the programmer to be referred later to understand the program flow and its functionality. It may also be used by some other programmer to interpret the program.

Each line of the assembly program source file may consist of up to four fields:

- (i) **Label:** Labels must start in column 1.
- (ii) **Mnemonic:** Mnemonics may start in column 2 or beyond.
- (iii) **Operand(s):** Operands follow the mnemonic.



- (iv) **Comment:** Comments may follow the operands, mnemonics, or labels and can start in any column.

The maximum width of a column is 256 characters. The programmer must use a colon to separate the label and the mnemonic and use space to separate the mnemonic and the operand(s). Operands are to be separated by commas. A label must begin with an alphabetic character or an underscore (_) and may contain alphanumeric characters, the underscore and the question mark (?). Labels can be up to 32 characters long. The mnemonic field can be either an assembly instruction mnemonic or an assembler directive. If there is a label on the same line, instructions must be separated from that label by a colon. If an operand field is present, it follows the mnemonic field. The operand field may contain operands for instructions and arguments in case of assembler directives. The comment field that begins with a semicolon (;) is optional and is added for documentation purpose. All characters followed by the semicolon are ignored through the end of the line.

For example, let us identify the four fields in the following source statement `too-low addlw 0x02 ; increment WREG by 2.`

The four fields in the given source statements are as follows:

- too-low is a label.
- addlw is an instruction mnemonic.
- 0x02 is an operand.
- increment WREG by2 is a comment

2.16 Assembler Directives

Assembler directives are similar to instructions in an assembly language program. But assembler directives direct the assembler to do something other than creating the machine code for an instruction. Assembler directives provide the programmer a means to instruct the assembler how to process subsequent assembly language instructions. It also provides a way to define program constants and reserve space for dynamic variables. Each assembler has its own set of directives. We will discuss the directives provided by the assembler called as MPASM.

MPASM provides five types of directives:

- Control directives:** Control directives permit sections of conditionally assembled code like branch instructions. Table 2.16.1 lists the different control directives.

- Data directives:** Data directives are those that are used for the allocation of memory and provide a way to refer to data items symbolically. Table 2.16.2 lists the different data directives.
- Listing directives:** Listing directives are those directives that control the MPASM listing file format. They allow the specification of titles, pagination, and other listing control. Table 2.16.3 lists the different listing directives.
- Macro directives:** These directives control the operation related to macro. Table 2.16.4 lists the different macro directives.
- Object directives:** These directives are used only to create an object file. Table 2.16.5 lists the different object directives.

Table 2.16.1: MPASM control directives

Directive	Description	Syntax
CODE	Begin executable code section	[<name>] code [<address>]
#DEFINE	Define a text substitution label	#define <name> [<value>]
		#define <name> [<arg>,...<arg>] <value>
ELSE	Begin alternative assembly block to IF	else
END	End program block	end
ENDIF	End conditional assembly block	endif
ENDW	End a while loop	endw
IF	Begin conditionally assembled code block	if <expr>
IFDEF	Execute if symbol has been defined	ifdef<label>
IFNDEF	Execute if symbol has not been defined	ifndef<label>

Directive	Description	Syntax
#INCLUDE	Include additional source code	#include <<include_file>> "include_file"
RADIX	Specify default radix	radix <default radix>
#UNDEFINE	Delete a substitution label	#undefine<label>
WHILE	Perform loop while condition is true	while <expr>

Table 2.16.2 : MPASM* data directives

Directive	Description	Syntax
CBLOCK	Define a block of constant	cblock [<expr>]
COSTANT	Declare symbol constant	constant <label> [=<expr>,...,<label> [=<expr>]]
DA	Store strings in program memory	[<label>] da <expr>[,<expr>,...,<expr>]
DATA	Create numeric and text data	[<label>] data <expr>[,<expr>,...,<expr>] [<label>] data "<text string>"["<text string>"]
DB	Decrease data of one byte	[<label>] db<expr> [,<expr>,...,<expr>] [<label>] db "<text string>"["<text string>"]
DT	Define table	[<label>] dt<expr>[,<expr>,...,<expr>] [<Label>] dt "<text_string>" [,"<text string>"]
DW	Declare data of one word	[<label>] dw<expr> [,<expr>,...,<expr>] [<Label>] dw "<text_string>" [,"<text string>"]

Directive	Description	Syntax
ENDC	End an automatic constant block	endc
EQU	Define an assembly constant	<label> equ <expr>
FILL	Fill memory	[<label> fill <expr>, <count>]
RES	Reserve memory	[<label>] res <mem_units>
SET	Define an assembler variable	<label> set <expr>
VARIABLE	Declare symbol variable	variable <label>[=<expr>,...,<label> [=<expr>]]

Table 2.16.3: MPASM assembler listing directives

Directive	Description	Syntax
ERROR	Issue an error message	error "<text_string>"
ERRORLEVEL	Set error level	Errorlevel 0 1 2 <+ -> <message number>
LIST	Listing options	list [<list_option>,..., <list_option>]
MESSG	Create user defined message	messg "<message_text>"
NOLIST	Turn off listing options	nolist
PAGE	Insert listing page eject	page
SPACE	Insert blank listing lines	space <expr> ...
SUBTITLE	Specify program subtitle	subtitle "<sub_text>"
TITLE	Specify program title	title "<title_text>"

Table 2.16.4 : MPASM macro directives

Directive	Description	Syntax
ENDM	End of a macro definition	endm
EXITM	Exit from a macro	exitm
MACRO	Declare macro definition	<label> macro [<arg>...<arg>]</arg>
EXPAND	Expand macro listing	expand
LOCAL	Declare local macro variable	local <label>[,<label>]
NOEXPAND	Turn off macro expansion	noexpand

Table 2.16.5 : MPASM object file directives

Directive	Description	Syntax
BANKSEL	Generate RAM bank selecting code	bankset<label>
CODE	Begin executable code section	[<name> code [<address>]]
_CONFIG	Specify configuration bits	_config<expr> OR _config<addr>, <expr>
EXTERN	Declare an external label	extern <label>[,<label>]
GLOBAL	Export a defined label	global <label>[,<label>]
IDATA	Begin initialized data section	[<name>] idata [<address>]
ORG	Set program origin	<label> org <expr>
PROCESSOR	Set processor type	processor <processor_type>
UDATA	Begin uninitialized data section	[<name>] udata [<address>]
UDATA_SHR	Begin shared uninitialized data section	[<name>] udata_shr [<address>]

2.17 Writing Simple Programs

Let us see some simple programs based on data transfer or addition / subtraction instructions.

Program 2.17.1 : Write PIC18 instructions to transfer data from

- (a) WREG to data registers at location 0x40.
- (b) The data register at 0x50 to the data register at 0x30,
- (c) The data register at 0x3F to WREG, and
- (d) Load the value 0x100 into FSR0.

Soln. :

- (a) MOVWF 0x40,A ; force access bank
- (b) MOVFF 0x50,0x30 ;
- (c) MOVF 0x3F,W,A : force access bank and copy register 0x3F to WREG
- (d) LFSR FSR0,0x100 ; load the value 0x100 into FSR0

Program 2.17.2 : Write PIC18 instructions to perform the following operations:

- (a) Add the content of WREG and that of the data register at 0x40 (in access bank) and leave the sum in WREG.
- (b) Increment the WREG register by 8.
- (c) Add the WREG register, the register with the name of sum, and carry and leave the result in sum. The variable sum is in access bank.

Soln. :

- (a) ADDWF 0x40,W,A
- (b) ADDLW 8
- (c) ADDWFC sum, F, A

Program 2.17.3 : Write PIC18 program to increment the contents of three registers 0x30 – 0x32 by 2.

Soln.:

The procedure for incrementing the value of a register by 3 is as follows:

Step 1: Place the value 2 in the WREG register.

Step 2: Execute the addwf i, d, a instruction.

The following program will increment the specified three registers by 3:



Label	Instruction	Comment
	org 0x00	
	goto start	
start:		
	MOVLW 0x2	
	ADDWF 0x30, F, A	increment the register at 0x30 by 2
	ADDWF 0x31, F, A	increment the register at 0x31 by 2
	ADDWF 0x32, F, A	increment the register at 0x32 by 2
	end	

Program 2.17.4 : Write an instruction sequence to add the contents of three data registers located at 0x40-0x42 and store the sum at 0x50.

Soln. :

The required operation can be achieved by the following procedure:

- Step 1: Load the contents of the register at 0x40 into the WREG register.
- Step 2: Add the contents of the register at 0x41 into the WREG register.
- Step 3: Add the contents of the register at 0x42 into the WREG register.
- Step 4: Store the contents of the WREG register in the register at 0x50.

The following program will perform the desired operation:

Label	Instruction	Comment
	org 0x00	
	goto start	
start:		
	MOVF 0x40, W, A	WREG t [0x40]
	ADDWF 0x41, W, A	add the contents of the register at 0x41 to WREG
	ADDWF 0x42, W, A	add the contents of the register at 0x42 to WREG
	MOVWF 0x50, A	0x50 t [WREG]
	end	

Program 2.17.5 : Write an instruction sequence to add 10 to the data registers at 0x300-0x303 using the indirect post increment addressing mode.

Soln. :

The procedure for solving this problem is as follows:

- Step 1: Load the value 0x300 into the FSR0 register.
- Step 2: Load the value 10 into the WREG register.
- Step 3: Add the value in WREG to the data register pointed by FSR0 using the indirect post increment mode.
- Step 4: Repeat Step 3 three more times.

The program that carries the operations from Step 1 to Step 3 is as follows:

Label	Instruction	Comment
	org 0x00	
	goto start	
start:		
	MOVLW 0x0A	
	LFSR FSR0,0x300	place 0x300 in FSR0
	ADDWF POSTINCO,F	
	end	

Program 2.17.6 : Write a program to subtract 9 from the data registers located at 0x50 – 0x53.

Soln. : This operation can be implemented by placing 9 in the WREG register and then executing the subwf f, F, A instruction. The following program will implement the required operation :

Label	Instruction	Comment
	org 0x00	
	goto start	
start:		
	MOVLW 0x09	place 9 in the WREG register
	SUBWF 0x50,F,A	decrement the contents of the register at 0x50 by 9



	SUBWF 0x51,F,A	decrement the contents of the register at 0x51 by 9
	SUBWF 0x52,F,A	decrement the contents of the register at 0x52 by 9
	SUBWF 0x53,F,A	decrement the contents of the register at 0x53 by 9
	end	

Program 2.17.7 : Write an instruction to perform each of the following operations:

- Subtract the WREG register from the file register at 0x30 and leave the difference in the file register at 0x30.
- Subtract the file register at 0x30 and borrow from the WREG register.
- Subtract the WREG register from the file register at 0x50 and leave the difference in WREG.

Soln. :

The following instructions will perform the specified operations:

- SUBWF 0x30,F,A
- SUBFWB 0x30,W,A
- SUBWF 0x50,W,A

Program 2.17.8 : Write a assembly language program to find the smallest of two number which are stored at 0x50, 0x51 and place smaller number in file register location 0x52.

SPPU - Aug. 14(In Sem.), 4 Marks

Label	Instruction	Comment
	org 0x00	
	GOTO start	
start:		
	MOVF 0x50,W,A	copy the value of location at 0x50 to WREG
	CPFSLT 0x51	compare WREG with 0x51 and skip next instruction if W is smaller
	MOVF 0x51,W,A	copy the value of location at 0x51 to WREG
	MOVWF 0x52,A	save the smaller number at memory location 0x52
	end	

Program 2.17.9 : Write a program to copy data from memory location 202H to WREG. **SPPU - Dec. 16, 6 Marks**

Soln. :

Label	Instruction	Comment
	org 0x00	
	goto start	
start:		
	LFSR FSR0,0x202	Initialize pointer FSR0 to 0x202
	MOVF INDFO,W	Move from 0x202 to WREG
	end	

Program 2.17.10 : Write a assembly language program to copy an array of 100 elements starting from a location 0x010 to a memory location 0x200 onwards.

SPPU - Oct. 16(In Sem.), 6 Marks

Soln. :

Label	Instruction	Comment
	counter equ 0x64	Initialize counter to 100
	org 0x00	
	goto start	
start:	LFSR FR1,0x200	Initialize pointer FSR0 to 0x200
	LFSR FSR0,0x010	Initialize pointer FSR0 to 0x010
next:	MOVF POSTINC0,W	Move from memory to WREG and increment the pointer
	MOVWF POSTINC1	Store the value and increment the pointer
	DECf counter,f	Decrement counter
	BNZ next	Repeat if counter not zero
	end	

2.18 Writing Programs to Perform Arithmetic Computations

The PIC18 microcontroller has instructions for performing 8-bit addition, subtraction, and multiplication operations. The PIC18 microcontroller doesn't provide any instruction for division, and hence this operation must also be implemented by the use of subtraction instruction.



2.18.1 Perform Addition Operations

The PIC18 microcontroller has two ADD instructions with two operands and one ADD instruction with three operands. These ADD instructions are designed to perform 8-bit operations. The execution result of the ADD instruction will affect all flag bits of the STATUS register. The three-operand ADD instruction will be needed for performing multi byte addition operations (also called as multi-precision addition). When dealing with multi byte numbers, there is an issue regarding how the number is stored in memory. If the least significant byte of the number is stored at the lowest address, then the byte order is called little-endian. Otherwise, the byte order is called big-endian. MPLAB uses little-endian byte order hence we will follow the same.

Program 2.18.1: Write a program that adds the three numbers stored in data registers at 0x20, 0x30, and 0x40 and places the result in data register at 0x50.

Soln.:

The algorithm for adding three numbers is as follows:

- Step 1 :** Load the number stored at 0x20 into the WREG register
- Step 2 :** Add the number stored at 0x30 and the number in the WREG register and leave the sum in the WREG register.
- Step 3 :** Add the number stored at 0x40 and the number in the WREG register and leave the sum in the WREG register.
- Step 4 :** Store the contents of the WREG register in the memory location at 0x50.

The program that implements this algorithm is as follows:

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	higher priority interrupt vector
	RETFIE	
	org 0x18	lower priority interrupt vector
	RETFIE	
start:		
	MOVF 0x20,W,A	copy the contents of 0x20 to WREG
	ADDWF 0x30,W,A	add the value in 0x30 to that of WREG
	MOVWF 0x40,W,A	add the value in 0x40 to that of WREG
	end	save the sum in memory location 0x50

Label	Instruction	Comment
	ADDWF 0x30,W,A	add the value in 0x30 to that of WREG
	ADDWF 0x40,W,A	add the value in 0x40 to that of WREG
	MOVWF 0x50,A	save the sum in memory location 0x50
	end	

Program 2.18.2 : Write a program that adds the 24-bit integers stored at 0x10 to 0x12 and 0x13 to 0x15, respectively, and store the result at 0x20 to 0x22.

Soln. :

The addition starts from the least significant byte (at the lowest address for little-endian byte order). One of the operand must be loaded into the WREG register before addition can be performed. The program is as follows:

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:		
	MOVF 0x10,W,A	copy the value of location at 0x10 to WREG
	ADDWF 0x13,W,A	add and leave the sum in WREG
	MOVWF 0x20,A	save the sum at memory location 0x20
	MOVF 0x11,W,A	copy the value of location at 0x11 to WREG
	ADDWFC 0x14,W,A	add with carry and leave the sum in WREG
	MOVWF 0x21,A	save the sum at memory location 0x21

Label	Instruction	Comment
	MOVF 0x12,W,A	copy the value of location at 0x12 to WREG
	ADDWF 0x15,W,A	add with carry and leave the sum in WREG
	MOVWF 0x22,A	save the sum at memory location 0x22
	end	

Program 2.18.3 : Write a program to add two 16 bit numbers which are stored at memory location 0x32, 0x33 and 0x51, 0x52 least significant byte is stored at lower address. Store Result in 0x61, 0x62. **SPPU - Aug. 14 (In Sem.), 4 Marks**

Soln. :

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	
	RETIE	
	org 0x18	
start:		
	MOVF 0x32,W,A	copy the value of location at 0x32 to WREG
	ADDWF 0x51,W,A	add and leave the sum in WREG
	MOVWF 0x61,A	save the sum at memory location 0x61
	MOVF 0x33,W,A	copy the value of location at 0x33 to WREG
	ADDWF 0x52,W,A	add with carry and leave the sum in WREG
	MOVWF 0x62,A	save the sum at memory location 0x62
	end	

Program 2.18.4 : Write an assembly language program to add the constant AAH to the contents of file reg 0x36 and store the result in file reg 0x40. **SPPU - Dec. 14, 7 Marks**

Soln. :

Label	Instruction	Comment
	org 0x00	
	GOTO start	

Label	Instruction	Comment
start:		
	MOVFLW 0xAA	copy the value 0xAA to WREG
	ADDWF 0x36,W,A	add and leave the sum in WREG
	MOVWF 0x40,A	save the sum at memory location 0x40
	end	

Program 2.18.5 : Write a program to add 5 elements in an array starting from 0x20H. Store the results at 0x40H. **SPPU - Dec. 15, 7 Marks**

Soln. :

Label	Instruction	Comment
	org 0x00	
	goto start	
start:		
	MOVF 0x20,W,A	Move into WREG the contents of [0x20]
	ADDWF 0x21,W,A	add the contents of the register at 0x21 to WREG
	ADDWF 0x22,W,A	add the contents of the register at 0x22 to WREG
	ADDWF 0x23,W,A	add the contents of the register at 0x23 to WREG
	ADDWF 0x24,W,A	add the contents of the register at 0x24 to WREG
	MOVWF 0x40,A	Move to 0x40 from [WREG]
	end	

Program 2.18.6 : Write an instruction sequence in assembly language to add a number 0x05 to the contents of memory location 0x06 and store the result at the same location. **SPPU - May 16, 4 Marks**

Soln. :

Label	Instruction	Comment
	org 0x00	
	GOTO start	



Label	Instruction	Comment
start:		
	MOVFLW 0xA05	copy the value 0x05 to WREG
	ADDWF 0x06,W,A	add and leave the sum in WREG
	MOVWF 0x06,A	save the sum at memory location 0x06
	end	

Program 2.18.7 : Write an assembly language program to add 02H to each element of an array starting from location 0x100. The length of array is 10.

SPPU - Aug. 15 (In Sem.), 6 Marks

Soln.:

Label	Instruction	Comment
	counter equ 0x0A	Initialize counter to 10
	org 0x00	
	goto start	
start:		
	LFSR FSR0,0x100	Initialize pointer FSR0 to 0x100
next:	MOVF INDF0,W	Move from memory to WREG
	ADDLW 0x02	Add 0x02 to W reg
	MOVWF POSTINCO	Store the result and increment the pointer
	DECF counter,f	Decrement counter
	BNZ next	Repeat if counter not zero
	end	

Program 2.18.8 : Write an assembly language program to add the contents of file register 0x40 H to contents of file register 0x41H and store the result in file register 0x42H.

SPPU - Oct. 16 (In Sem.), 6 Marks

Soln.:

Label	Instruction	Comment
	org 0x00	
	goto start	
start:		

Label	Instruction	Comment
	MOVF 0x40,W,A	Move into WREG the contents of [0x40]
	ADDWF 0x41,W,A	add the contents of the register at 0x41 to WREG
	MOVWF 0x42,A	Move to 0x42 from [WREG]
	end	

2.18.2 Perform Subtraction Operations

The PIC18 microcontroller has two two-operand and two three-operand SUBTRACT instructions. SUBTRACT instructions will also affect all the flag bits of the STATUS register. Three-operand subtraction instructions are provided mainly to support the implementation of multibyte subtraction (also called as multiprecision subtraction). A multiprecision subtraction must be performed from the least significant byte toward the most significant byte.

Program 2.18.9 : Write a program to subtract 3 from memory locations 0x10 to 0x13.

Soln.:

The algorithm for this problem is as follows:

- Step 1: Place 3 in the WREG register.
- Step 2: Subtract WREG from the memory location 0x10 and leave the difference in the memory location 0x10.
- Step 3: Subtract WREG from the memory location 0x11 and leave the difference in the memory location 0x11.
- Step 4: Subtract WREG from the memory location 0x12 and leave the difference in the memory location 0x12
- Step 5: Subtract WREG from the memory location 0x13 and leave the difference in the memory location 0x13

The assembly program that implements this algorithm is as follows:

Label	Instruction	Comment
	org 0x00	
	GOTO start	

Label	Instruction	Comment
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:		
	MOVLW 0x05	place the value 5 in WREG
	SUBWF 0x10,F,A	subtract 5 from memory location 0x10
	SUBWF 0x11,F,A	subtract 5 from memory location 0x11
	SUBWF 0x12,F,A	subtract 5 from memory location 0x12
	SUBWF 0x13,F,A	subtract 5 from memory location 0x13
	end	

Program 2.18.10 : Write a program that subtracts the 32-bit number stored at 0x20 to 0x23 from the number stored at 0x10 to 0x13 and store the result at 0x30 to 0x33.

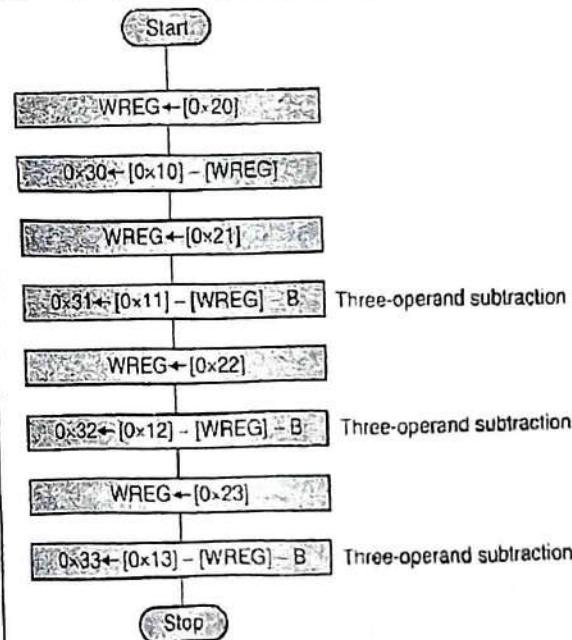
Soln. :

The logic flow of this problem is shown in Flowchart1

The program that implements the logic illustrated in Flowchart P. 2.18.10 is as follows

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:		
	MOVF 0x20, W, A	
	SUBWF 0x10, W, A	subtract the least significant byte

Label	Instruction	Comment
	MOVWF 0x30, A	
	MOVF 0x21, W, A	
	SUBWFB 0x11, W, A	subtract the second to least significant byte
	MOVWF 0x31, A	
	MOVF 0x22, W, A	
	SUBWFB 0x12, W, A	subtract the second to most significant byte
	MOVWF 0x32, A	
	MOVF 0x23, W, A	
	SUBWFB 0x13, W, A	subtract the most significant byte
	MOVWF 0x33, A	
	end	



Flowchart P. 2.18.10



2.18.3 Binary Coded Decimal (BCD) Addition

All processors perform arithmetic using binary arithmetic. However, input and output devices generally use decimal numbers because we are used to decimal numbers. Incorrect BCD sums can be adjusted by performing the following operations:

1. Add 0x6 to every sum digit greater than 9.
2. Add 0x6 to every sum digit that had a carry of 1 to the next higher digit.

The decimal adjust WREG (DAW) instruction adjusts the 8-bit value in the WREG register resulting from the earlier addition of two variables (each in packed BCD format) and produces a correctly packed BCD result. This instruction operates similar to DAA of Intel processors.

Program 2.18.11: Write a program that adds the decimal numbers stored at 0x23 & 0x24 and stores the sum in 0x25. The result must also be in BCD format.

Soln.: The instruction is as follows

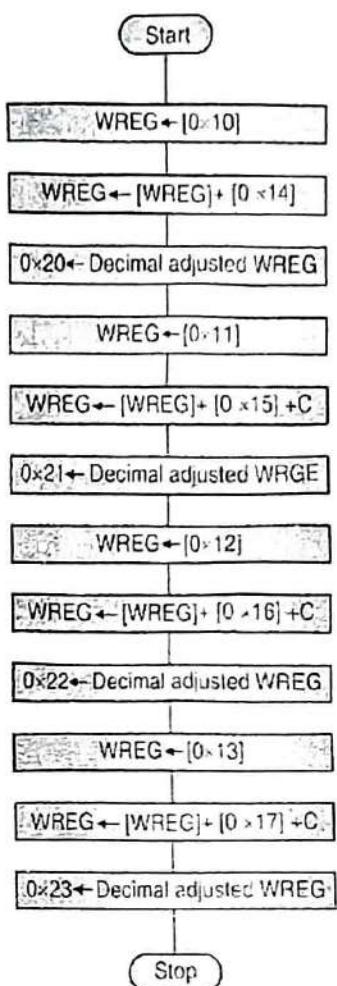
Label	Instruction
	org 0x00
	GOTO start
	org 0x08
	RETFIE
	org 0x18
	RETFIE
start:	
	MOVF 0x10,W,A
	ADDWF 0x14,W,A
	DAW
	MOVWF 0x20,A
	MOVF 0x11,W,A
	ADDWFC 0x15,W,A
	DAW
	MOVWF 0x21,A
	MOVF 0x12,W,A
	ADDWFC 0x16,W,A
	DAW
	MOVWF 0x22,A
	MOVF 0x13,W,A
	ADDWFC 0x17,W,A
	DAW
	MOVWF 0x23,A
	end

Program 2.18.12: Write an instruction sequence that adds the 4 digit decimal numbers stored at 0x10 to 0x13 and 0x14 to 0x17 and stores the decimal result in 0x20 to 0x23. All operands are in the access bank.

Soln. :

In order to make sure that the sum is also in decimal format, decimal adjustment must be done after the addition of each byte pair. The flow chart of this problem is shown in Flowchart P. 2.18.12. The program is as follows:

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:		
	MOVF 0x10,W,A	WREG $\leftarrow [0x10]$
	ADDWF 0x14,W,A	WREG $\leftarrow [0x10] + [0x14]$
	DAW	decimal adjust WREG
	MOVWF 0x20,A	save the least significant sum digit
	MOVF 0x11,W,A	WREG $\leftarrow [0x11]$
	ADDWFC 0x15,W,A	WREG $\leftarrow [0x11] + [0x15] + Cy$
	DAW	Decimal Adjust WREG
	MOVWF 0x21,A	WREG $\rightarrow [0x21]$
	MOVF 0x12,W,A	WREG $\leftarrow [0x12]$
	ADDWFC 0x16,W,A	WREG $\leftarrow [0x12] + [0x16] + Cy$
	DAW	Decimal Adjust WREG
	MOVWF 0x22,A	WREG $\rightarrow [0x22]$
	MOVF 0x13,W,A	WREG $\leftarrow [0x13]$
	ADDWFC 0x17,W,A	WREG $\leftarrow [0x13] + [0x17] + Cy$
	DAW	Decimal Adjust WREG
	MOVWF 0x23,A	save the most significant sum digit
	end	



Flowchart P. 2.18.12

2.18.4 Multiplication

The PIC18 microcontroller provides two unsigned multiply instructions.

1. The **MULLW k** instruction multiplies an 8-bit literal with the WREG register and places the 16-bit product in the register pair PRODH:PRODL.
2. The **MULWF f,a** instruction multiplies the contents of the WREG register with that of the specified file register and leaves the 16-bit product in the register pair PRODH:PRODL.

In each case, the upper byte of the product is placed in the PRODH register, whereas the lower byte of the product is placed in the PRODL register.

Program 2.18.13 : Write a program to multiply two 8-bit numbers stored in data memory locations 0x10 and 0x11, respectively, and place the product in data memory locations 0x20 and 0x21.

Soln. :

The program is as follows:

Label	Instruction
	org 0x00
	GOTO start
	org 0x08
	RETFIE
	org 0x18
	RETFIE
start:	
	MOVF 0x10, W,A
	MULWF 0x11,A
	MOVFF PRODH, 0x21
	MOVFF PRODL, 0x20
	end

The **unsigned multiply instructions** can also be used to perform multiprecision (multibyte) multiplications. In a multiprecision multiplication, the multiplier and the multiplicand must be broken down into 8-bit chunks, and multiple 8-bit by 8-bit multiplications must be performed. Assume that we want to multiply a 16-bit hex number P by another 16-bit hex number.

For illustrating the procedure, we will break P and Q down as follows:

$$P = P_H P_L$$

$$Q = Q_H Q_L$$

Where P_H and Q_H are the upper eight bits of P and Q, respectively, and P_L and Q_L are the lower eight bits. Four 8-bit by 8-bit multiplications are performed, and then the partial products are added together as shown in Fig. P. 2.18.13

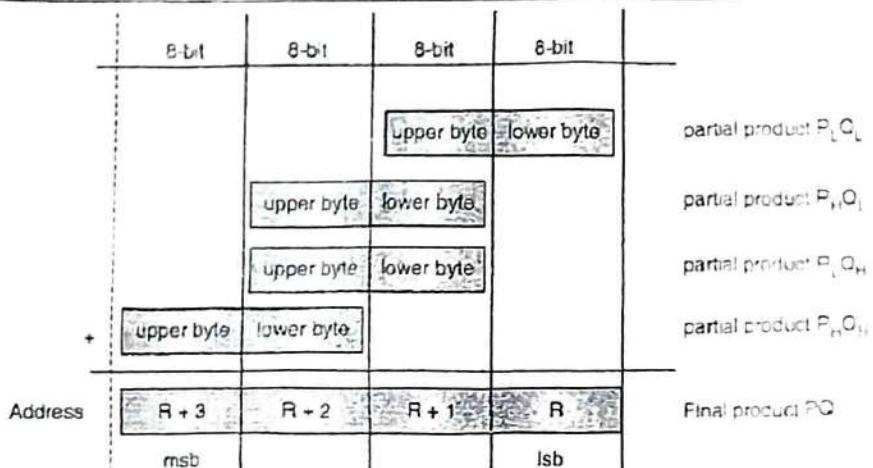


Fig. P. 2.18.13: 16-bit by 16-bit multiplication

Note: msb stands for most significant byte and lsb stands for least significant byte.

Program 2.18.14: Write a program to multiply two 16-bit unsigned integers assuming that the multiplier and multiplicand are stored in data memory locations M1 and M1 + 1 and N1 and N1 + 1, respectively. Store the product in data memory locations PR and PR + 3. The multiplier, the multiplicand, and the product are located in the access bank

Soln.: The algorithm for the unsigned 16-bit multiplication is as follows:

- Step 1: Compute the partial product $P_L Q_L$ and save it in locations PR and PR + 1.
- Step 2: Compute the partial product $P_L Q_H$ and save it in locations PR + 2 and PR + 3.
- Step 3: Compute the partial product $P_R Q_L$ and add it to memory locations PR + 1 and PR + 2. The C flag may be set to 1 after this addition.
- Step 4: Add the C flag to memory location PR + 3.
- Step 5: Compute the partial product $P_R Q_H$ and add it to memory locations PR + 1 and PR + 2. The C flag may be set to 1 after this addition.
- Step 6: Add the C flag to memory location PR + 3.

The assembly program to implement this logic is as follows:

Label	Instruction	Comment
	phequ 0x37	upper byte of the first number
	plequ 0x23	lower byte of the first number
	qhequ 0x66	upper byte of the second number
	qlequ 0x45	lower byte of the second number
M1 set 0x00		set the address to store multiplicand

Label	Instruction	Comment
N1 set 0x02		set the address to store multiplier
PR set 0x06		product
org 0x00		
GOTO start		
org 0x08		
RETFIE		
org 0x18		
RETFIE		
start:	MOVLW qh	set up test numbers
	MOVWF M1+1,A	
	MOVLW ql	
	MOVWF M1,A	
	MOVLW ph	
	MOVWF N1+1,A	
	MOVLW pl	
	MOVWF N1,A	

Label	Instruction	Comment
	MOVF M1+1,W,A	
	MULWF N1+1,A	compute pH xqH
	MOVFF PRODL, PR+2	
	MOVFF PRODH,PR+3	
	MOVF M1,W,A	compute pLxqL
	MULWF N1,A	
	MOVFF PRODL, PR	
	MOVFF PRODH,PR+1	
	MOVF M1,W,A	
	MULWF N1+1,A	compute pLxqH
	MOVF PRODL,W,A	add pLxqH to PR
	ADDWF PR+1,F,A	
	MOVF PRODH,W,A	
	ADDWFC PR+2,F,A	
	MOVLW 0	
	ADDWFC PR+3,F,A	add carry
	MOVF M1+1,W,A	
	MULWF N1,A	compute pH xqL
	MOVF PRODL,W,A	add pH xqL to PR
	ADDWF PR+1,F,A	
	MOVF PRODH,W,A	
	ADDWFC PR+2,F,A	
	MOVLW 0	
	ADDWFC PR+3,F,A	add carry
	NOP	
	end	

Similarly multiplication of further lengths can also be performed.

2.19 Program Loops

By writing loops in a program we can perform repeated task with less memory space requirement for program. A loop may be executed for a finite or infinite number of times. A finite loop is a sequence of instructions that will be executed for a given finite number of times, while an endless loop is a sequence of instructions that will be repeated forever (mainly required in Embedded systems).

2.19.1 Changing the Program Counter

A normal program execution can be defined as one in which the CPU executes instructions sequentially starting from lower addresses towards higher addresses. The implementation of a program loop requires the capability of changing this sequential or normal program execution. In the normal program execution, the program counter value is incremented by 2 or 4 based on the instruction size. The PIC18 program counter is 21 bits wide, 8-bit lower byte PCL register, 8-bit higher byte PCH register and the 5-bit highest part of address is contained in register PCU. The PCL register is directly mapped into the data memory and is readable or writable as any other data register.

The PCH and PCU registers are not directly readable or writable. Updates to the PCH register can be performed through the PCLATH register while that to the PCU register can be performed through the PCLATU register. Fig.2.19.1 shows the interaction of the PCU, PCH, and PCL registers with the PCLATU and PCLATH registers.

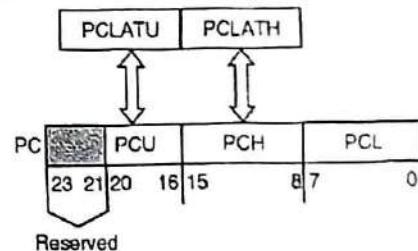


Fig. 2.19.1: Program counter structure

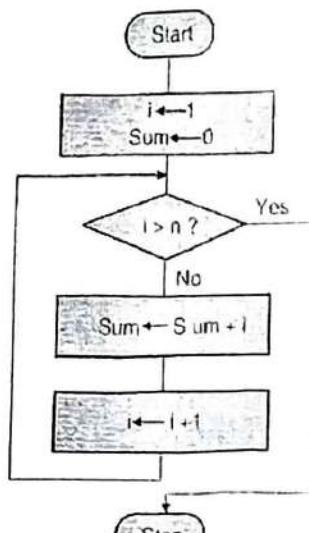
Whenever the PCL is read, the contents of PCH and PCU are transferred to PCLATH and PCLATU, respectively. Whenever PCL is written into, the contents of PCLATH and PCLATU are transferred to PCH and PCU, respectively. The resultant effect is a branch.

Program 2.19.1 : Write a program to compute $1 + 2 + 3 + \dots + n$ and save the sum at 0x00 and 0x01 assuming that the value of n is in a range such that the sum can be stored in two bytes.



Soln.: The logic flow for computing the desired sum is shown in Flowchart P. 2.19.1. This flowchart implements thefor $i = 1$ to n loop construct. The following program implements the algorithm.

Label	Instruction	Comment
	n equ 9	
	sum_hi set 0x01	high byte of sum
	sum_lo set 0x00	low byte of sum
	i set 0x02	loop index i
	org 0x00	reset vector
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:	CLRF sum_hi,A	initialize sum to 0
	CLRF sum_lo,A	
	CLRF i,A	initialize i to 0
	INCF i,F,A	i starts from 1
	sum_lp MOVLW n	place n in WREG
	CPFSGT i,A	compare i with n and skip if $i > n$
	BRA add_lp	perform addition when $i \leq n$
	BRA exit_sum	it is done when $i > 50$
	add_lp MOVF i,W,A	place i in WREG
	ADDWF sum_lo,F,A	add i to sum-lo
	MOVLW 0	
	ADDWFC sum_hi,F,A	add carry to sum-hi
	INCF i,F,A	increment loop index i by 1
	BRA sum_lp	
	exit_sum NOP	
	end	

Flowchart P. 2.19.1: For computing $1 + 2 + \dots + n$

Program 2.19.2: Write a program to find the largest number stored in the array that is stored in data memory locations from 0x10 to 0x5F.

Soln. : We use the indirect addressing mode to step through the given data array. The algorithm to find the largest element of the array is as follows:

Step 1 : Set the value of the data memory location at 0x10 as the max.

Step 2 : Compare the next data memory location with the max. If the new memory location is larger, then replace the max with the value of the current data memory location.

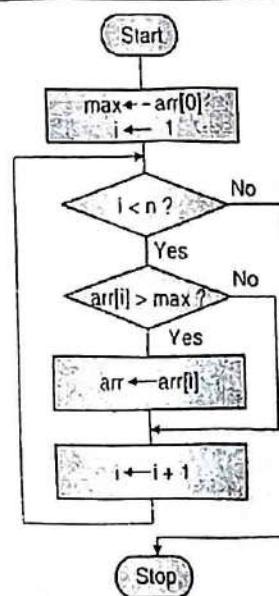
Step 3 : Repeat the same comparison until all the data memory locations have been checked

The flowchart of this algorithm is shown in Flowchart P. 2.19.2.

The PIC18 assembly program that implements this algorithm is as follows:

Label	Instruction	Comment
	max equ 0x00	
	iequ 0x01	
	n equ 0xA	the array count
	#include <p18F8720.inc>	
	org 0x00	

Label	Instruction	Comment	Label	Instruction	Comment
	GOTO start				following 7 instructions update the max in every iteration if required.
	org 0x08			MOVF POSTINC0,W	place arr[i] in WREG and increment array pointer
	RETFIE			CPFSGT max,A	is max > arr[i]?
	org 0x18			GOTO replace	if condition is no
	RETFIE			GOTO next	if condition is yes
start:	MOVF 0x10,W,A	set arr[0] as the initial max		MOVWF max,A	update the array max
	MOVWF max,A			INCF i,F,A	
	LFSR FSR0,0x11	place 0x11 (address of arr[1]) in FSR0		GOTO again	
	CLRF i,A	initialize loop counter i to 0		NOP	
again:	MOVLW n - 1	initialize the number of comparisons to be made		end	
	CPFSLT i,A	skip if i < n - 1			
	GOTO done	all comparisons have been done the			



Flowchart P. 2.19.2: For finding the maximum



2.20 Reading and Writing Data in Program Memory

The PIC18 program memory has 16-bit locations, whereas the data memory has 8-bit locations. To handle this mismatch of bus size between the program memory and data memory, the PIC18 microcontroller moves data between these two memory spaces through an 8-bit register (TABLAT). In order to read and write program memory, the PIC18 microcontroller provides two instructions that allow the processor to move bytes between the program memory and the data memory viz. Table read (TBLRD) and Table write (TBLWT). Fig. 2.20.1 shows the operation of a table read with program memory and data memory.

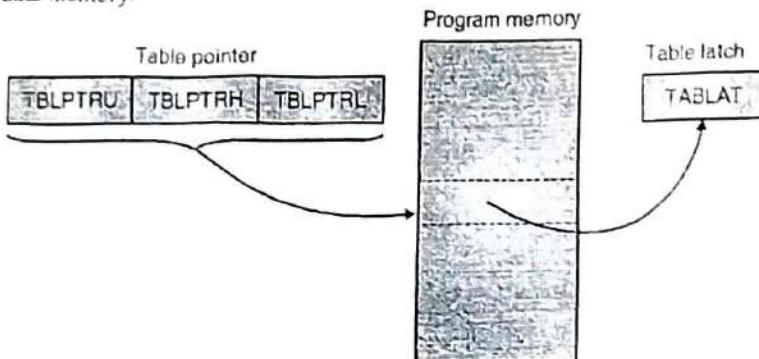


Fig. 2.20.1: Table read operation

Fig. 2.20.2 shows the operation of a table write with program memory and data memory

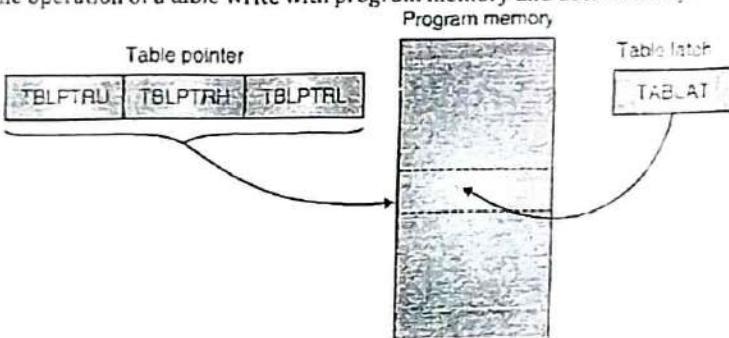


Fig. 2.20.2: Table write operation

Whenever a table-read instruction is executed, a byte will be transferred from program memory to the table latch (TABLAT). PIC18 microcontroller has a certain number of holding registers to hold data to be written into the program memory. Holding registers must be filled up before the program-memory-write operation can be started.

Program 2.20.1 : Write a program to read a byte from program memory location at 0x60 into TABLAT.

Soln.: One of the operations is to read the byte in program memory is to set up the table pointer. The following program will read the byte from the program memory:

Label	Instruction	Comment
	org 0x00	
	GOTO start	

Label	Instruction	Comment
	org 0x08	
	RETFIE	
	org 0x60	
	ldb 3	to store data in program memory
	org 0x18	
	RETFIE	
start:	CLRF TBLPTRU,A	set TBLPTR to point to data memory at 0x60
	CLRF TBLPTRH,A	
	MOVLW 0x60	
	MOVWF TBLPTRL,A	

Label	Instruction	Comment
	TBLRD*	read the byte into TABLAT
end		

Program 2.20.2 : Write an instruction sequence to do the following:

- (a) Set bits 5, 3, and 1 of the PORTA register to logic '1'
 - (b) Clear bits 7, 6, and 1 of the PORTB register to logic '0'
 - (c) Toggle bits 7, 3, and 0 of the PORTC register
- Soln. :** These requirements can be achieved as follows :
- (a) MOVLW B'00101010'
IORWF PORTA, F, A
 - (b) MOVLW B'00111101'
ANDWF PORTB, F, A
 - (c) MOVLW B'10001110'
XORWF PORTC

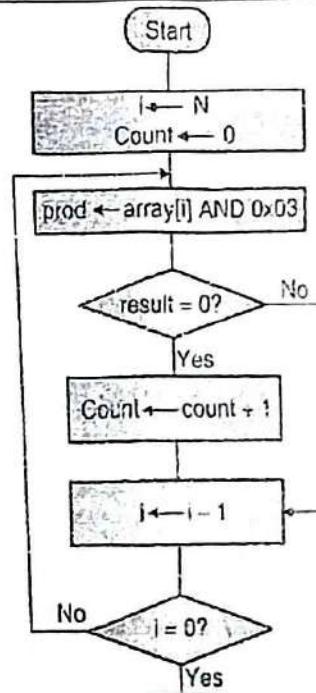
Program 2.20.3 : Write a program to find out the number of elements in an array of 8-bit elements that are multiple of 4. The array is in the program memory.

Soln. :

A number is a multiple of 4 if its least significant two bits are 00. This can be tested by ANDing the array element with B'0000 0011'; zero result indicates the least two bits are 00 and hence the number is divisible by 2. The flow chart shown in Flowchart P. 2.20.3 shows the implementation of this logic. The program is as follows:

Label	Instruction
	n equ 0x0A
	iset 0x00
	count set 0x01
	mask equ 0x03
	org 0x00
	GOTO start
	org 0x08
	RETFIE
	org 0x60
	Arraydb0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09
	org 0x18

Label	Instruction
	RETFIE
start:	CLRF count,A
	MOVLW n
	MOVWF i
	MOVLW upper array
	MOVWF TBLPTRU,A
	MOVLW high array
	MOVWF TBLPTRH,A
	MOVLW low array
	MOVWF TBLPTRL,A
	MOVLW mask
loop1:	TBLRD*+
	ANDWF TABLAT,F,A
	BNZ over
	INCF count
over:	DECFSZ i
	BRA loop1
	NOP
	end



Flowchart P. 2.20.3

Program 2.20.4 : Draw the flow chart and write an assembly language program for PIC18 controller to read and write five bytes from program memory to data memory. Assume the starting address of the program, data memory

Soln. :

Label	Instruction
	i set 0x00
	org 0x00
	goto start
	org 0x60
	array dB 0x00, 0x01, 0x02, 0x03, 0x04
start :	MOVLW 0x05
	MOVWF i
	MOVLW upper array
	MOVWF TBLPTRU, A
	MOVLW high array
	MOVWF TBLPTRH, A
	MOVLW low array
	MOVWF TBLPTRL, A
	LFSR FSR0,0x01
loop1:	TBLRD* +
	MOVF TABLAT,W, A
	MOVWF POSTINC0, A
	DECFSZ i
	BRA loop1
	end

2.21 Using Program Loop to Create Time Delays

A time delay can be created by repeating a set of instructions for certain no of times according to the required delay.

Program 2.21.1 : Write a program loop to create a time delay of 0.5ms. This program loop is to be run on a P18FF8720 clocked by a 40-MHz crystal oscillator

Soln.: Since each instruction cycle consists of four oscillator cycles, one instruction cycle lasts for $4 \times t = 4 \times 1/f = 4 \times 1/40M = 0.1\mu s = 100ns$

To create a delay of 0.5 ms, we will use a delay of 2 μs for 250 times. The following program loop will create a time delay of 0.5 ms:

Label	Instruction	Comment
	cnt set 0x00	
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIF	
start:	MOVLW D'250'	
	MOVWF cnt, A	
again:	NOP	17 instruction cycle, 1 for each NOP
	NOP	
	DECFSZ cnt,F,A	1 instruction cycle (2 when [cnt] = 0)
	BRA again	2 instruction cycle
	end	

This program tests the looping condition after 17 NO P instructions have been executed, and hence it implements this for 250 times. Longer delays can be created by adding another layer of loop.



Program 2.21.2 : Write a program loop to create a time delay of 100ms. This program loop is to be run on a PIC18F8720 clocked by a 40-MHz crystal oscillator.

Soln. : A 100-ms time delay can be created by repeating the program loop in Program 14.9.1 for 200 times. The program loop is as follows :

Label	Instruction	Comment
	cnt set 0x00	
	cnt1 set 0x01	
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:	MOVLW D'200'	
	MOVWF cnt1,A	
back:	MOVLW D'250'	
	MOVWF cnt, A	
again:	NOP	17 instruction cycle, 1 for each NOP
	NOP	
	DECFSZ cnt,F,A	1 instruction cycle (2 when [cnt] = 0)
	BRA again	2 instruction cycle
	DECFSZ cnt1,F,A	
	BRA back	
	end	

Program 2.21.3 : Write a program to calculate delay of 100 microsecond using PIC18F microcontroller (f_{req} = 40 MHz)

Soln.:

Since each instruction cycle consist of four oscillator cycles, one instruction cycle lasts for

$$4 \cdot t = 4 \cdot \frac{1}{f} = 4 \times \frac{1}{40M} = 0.1 \mu\text{sec} = 100 \text{ nsec}$$

To create a delay of 100 μsec we will use a delay of 2 μsec for 50 times. The following loop generates delay of 100 μsec .

Label	Instruction
	cnt set 0x00
	org 0x00
	goto start
	org 0x08
	RETFIE
	org 0x18
	RETFIE
start :	MOVLW D '50'
	MOVWF cnt, A
again :	NOP
	DECFSZ cnt, F, A
	BRA again
	end



2.22 Rotate Instructions based Programs

Rotate instructions can be used to manipulate bit fields and multiply or divide a number by a divisor of power of 2. The operation of multiplying by the power of 2 can be implemented by shifting the dividend operand to the left by n, where the multiplier is nth power of 2. Whereas dividing by the power of 2 can be implemented by shifting the operand to the right by n, where the divisor is nth power of 2.

Program 2.22.1 : Write a program to multiply the three-byte number located at 0x00 to 0x02 by 16.

Soln. :

Multiplying by 16 can be implemented by shifting to the left four places. The left shifting operation should be performed from the least significant byte towards the most significant byte. The following program will implement this logic.

Label	Instruction	Comment
	l set 0x00	
	u set 0x01	
	t set 0x02	
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:		
start:	MOVLW 0x03	set loop count to 3
loop:	BCF STATUS, C, A	clear the C flag
	RRCF 0x00, F, A	shift right one place
	RRCF 0x01, F, A	
	RRCF 0x02, F, A	
	DECFSZ WREG, W, A	
	BRA loop	not completed yet, continue
start:	MOVLW 0x04	set loop count to 4
loop:	BCF STATUS, C, A	clear the C flag
	RLCF 0x00, F, A	shift left one place
	RLCF 0x01, F, A	
	RLCF 0x02, F, A	
	DECFSZ WREG, W, A	
	GOTO loop	not completed yet, continue
	end	

Program 2.22.2 : Write an instruction sequence to divide the three-byte number stored at 0x00 to 0x02 by 8.

Soln. :

Dividing by 8 can be implemented by shifting the number to the right three positions. The right-shifting operation should be performed from the most significant byte towards the least significant byte. The following program will implement this logic.

Label	Instruction	Comment
	l set 0x00	
	u set 0x01	
	t set 0x02	
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:		
start:	MOVLW 0x03	set loop count to 3
loop:	BCF STATUS, C, A	clear the C flag
	RRCF 0x00, F, A	shift right one place
	RRCF 0x01, F, A	
	RRCF 0x02, F, A	
	DECFSZ WREG, W, A	
	BRA loop	not completed yet, continue

2.23 Exam Pack (Review and University Questions)

- Q. Explain any three addressing modes of PIC18 with one example each. (Refer Section 2.1) (Dec. 14, 6 Marks)
- Q. Explain with an example the addressing modes of PIC18F548. (Refer Section 2.1) (Oct. 16(In Sem.), 4 Marks)
- Q. Explain various addressing modes used in PIC 18 microcontroller. (Refer Section 2.1) (Dec. 17, Oct. 19(In Sem., 6 Marks)



- | | |
|--|---|
| Q. Explain Register direct addressing mode of PIC18 microcontroller.

(Refer Section 2.1.1) (Dec. 16, 3 Marks) | Q. Explain the following instruction: ADDWFC 0x20,0,0

(Refer Section 2.4.3) (Aug. 17 (In Sem.), 3 Marks) |
| Q. Explain the following in detail: i) direct addressing mode.

(Refer Section 2.1.1) (May 19, 3 Marks) | Q. Explain following instruction in detail: SUBWFB 0x53, W

(Refer Section 2.4.7) (Dec. 14, 1 Mark) |
| Q. Explain immediate addressing mode of PIC18 microcontroller.

(Refer Section 2.1.2) (Dec. 16, 3 Marks) | Q. Explain following instruction with example: DECF

(Refer Section 2.4.12) (May 15, 2 Marks) |
| Q. Explain the following in detail: Immediate addressing mode . (Refer Section 2.1.2) (May 19, 3 Marks) | Q. Explain following instruction with flags they are affecting: MULLW (Refer Section 2.4.15) (Dec. 15, 2 Marks) |
| Q. Explain the instruction: MOVF f, d, a.

(Refer Sections 2.3 and 2.3.1)

(Oct. 16 (In Sem.), 2 Marks) | Q. Explain following instruction in detail: IORLW 0 x 23. (Refer Section 2.5.3) (Dec. 14, 1 Mark) |
| Q. Explain the instruction: MOVF 0 x 04, 0, 1.

(Refer Sections 2.3 and 2.3.1) (May 18, 2 Marks) | Q. Explain following instruction with example: IORLW

(Refer Section 2.5.3) (May 15, 2 Marks) |
| Q. Explain following instruction with example: MOVFF

(Refer Section 2.3.2) (May 15, 2 Marks) | Q. Explain following instruction with flags they are affecting: CPFSLT (Refer Section 2.5.11) (Dec. 15, 2 Marks) |
| Q. Explain the instruction MOV fs, fd.

(Refer Section 2.3.2)

(Aug. 15 (In Sem.), Dec. 17, May 18, 2 Marks) | Q. Explain the following instruction with suitable example:
i) BN n (Refer Section 2.7.2) (May 19, 2 Marks) |
| Q. Explain the following instruction in detail: MOVFF.

(Refer Section 2.3.2) (Dec. 16, 2 Marks) | Q. Explain following instruction in detail: BRA S

(Refer Section 2.7.8) (Dec. 14, 1 Mark) |
| Q. Explain the instruction: MOVLW 0x04

(Refer Section 2.3.4) (May 16, 2 Marks) | Q. Explain CALL instruction in PIC18

(Refer Sections 2.8 and 2.8.1) (May 15, 3 Marks) |
| Q. Explain the following instructions: MOVLW k

(Refer Section 2.3.4) (Dec. 17, 2 Marks) | Q. Explain RETURN instruction in PIC18.

(Refer Section 2.8.3) (May 15, 3 Marks) |
| Q. Explain following instruction in detail: SWAPF 0x56.

(Refer Section 2.3.6) (Dec. 14, 1 Mark) | Q. Explain the following instruction with suitable example:
BCF f,b,a.

(Refer Sections 2.11 and 2.11.1) (May 19, 2 Marks) |
| Q. Explain following instruction with flags they are affecting:
LFSR (Refer Section 2.3.7) (Dec. 15, 2 Marks) | Q. Explain following instruction with example: BSF

(Refer Section 2.11.2) (May 15, 2 Marks) |
| Q. Explain following instruction with example: SETF

(Refer Section 2.3.11) (May 15, 2 Marks) | Q. Explain instruction: BSF f, b, a

(Refer Section 2.11.2) (Oct. 16 (In Sem.), 2 Marks) |
| Q. Explain the following instruction in detail: ADDLW.

(Refer Sections 2.4 and 2.4.1) (Dec. 16, 2 Marks) | Q. Explain the following instruction (i) BSF PORTD, 0, 0.

(Refer Section 2.11.2) (Aug. 17 (In Sem.), 3 Marks) |
| Q. Explain following instruction with example: ADDWFC

(Refer Section 2.4.3) (May 15, 2 Marks) | Q. Explain the following instructions: BSF PORTD, 0

(Refer Section 2.11.2) (May 18, 2 Marks) |
| | Q. Explain following instruction in detail: BTFSC PORTB, 3

(Refer Section 2.11.3) (Dec. 14, 1 Mark) |



- | | |
|---|--|
| Q. Explain the instruction: BTFSC f, b, a

(Refer Section 2.11.3) (May 16, 2 Marks) | Q. Write an assembly language program to add the constant AAH to the contents of file reg 0x36 and store the result in file reg 0x40.

(Refer Program 2.18.4) (Dec. 14, 7 Marks) |
| Q. Explain the instruction BTFSS f, d, a

(Refer Section 2.11.4) (Aug. 15(In Sem.), 2 Marks) | Q. Write a program to add 5 elements in an array starting from 0x20H. Store the results at 0x40H.

(Refer Program 2.18.5) (Dec. 15, 7 Marks) |
| Q. Explain the following instructions: BTG f,b,a

(Refer Section 2.11.5) (Dec. 17, 2 Marks) | Q. Write an instruction sequence in assembly language to add a number 0x05 to the contents of memory location 0x06 and store the result at the same location.

(Refer Program 2.18.6) (May 16, 4 Marks) |
| Q. Explain the use of assembler.

(Refer Section 2.14.1) (Oct. 16 (In Sem.), 2 Marks) | Q. Write an assembly language program to add 02H to each element of an array starting from location 0x100. The length of array is 10.

(Refer Program 2.18.7) (Aug. 15(In Sem.), 6 Marks) |
| Q. Explain simulator.

(Refer Section 2.14.2) (Aug. 14(In Sem.), 2 Marks) | Q. Write an assembly language program to add the contents of file register 0x40 H to contents of file register 0x41H and store the result in file register 0x42H.

(Refer Program 2.18.8) (Oct. 16(In Sem), 6 Marks) |
| Q. Explain compiler.

(Refer Section 2.14.3) (Aug.14(In Sem.), 2 Marks) | Q. Explain Register indirect addressing mode and Immediate addressing mode in detail, with suitable example. (Refer Section 2.1) (Dec. 19, 4 Marks) |
| Q. Explain the use of compiler.

(Refer Section 2.14.3) (Oct. 16(In Sem.), 2 Marks) | Q. Explain the instruction : (iii) MOVLW 0XA0

(Refer Section 2.3.4) (Oct. 19(In Sem), 2 Marks) |
| Q. Write a assembly language program to find the smallest of two number which are stored at 0x50, 0x51 and place smaller number in file register location 0x52.

(Refer Program 2.17.8) (Aug. 14(In Sem.), 4 Marks) | Q. Explain the instruction : (i) MULWF 0x20,0

(Refer Section 2.4.16) (Oct. 19(In Sem), 2 Marks) |
| Q. Write a program to copy data from memory location 202H to WREG.

(Refer Program 2.17.9) (Dec. 16, 6 Marks) | Q. Explain the CALL and RETURN instructions of PIC 18 microcontroller (Refer Section 2.8.1) (Oct. 19(In Sem), 6 Marks) |
| Q. Write a assembly language program to copy an array of 100 elements starting from a location 0x010 to a memory location 0x200 onwards.

(Refer Program 2.17.10) (Oct. 16(In Sem.), 6 Marks) | Q. Explain the instruction : (ii) BTFSC 0X20,1,0

(Refer Section 2.11.3) (Oct. 19(In Sem, 2 Marks) |
| Q. Write a program to add two 16 bit numbers which are stored at memory location 0x32, 0x33 and 0x51, 0x52 least significant byte is stored at lower address. Store Result in 0x61, 0x62.

(Refer Program 2.18.3) (Aug. 14(In Sem.), 4 Marks) | |

3

UNIT - II

Programming of PIC Microcontroller in C

3.1 Embedded C Concepts

- There are many disadvantages of assembly programming, due to which high level languages are used in programming the embedded systems.
- Fig. 3.1.1 shows the conversion process that can translate the program written in high level language to corresponding machine language.

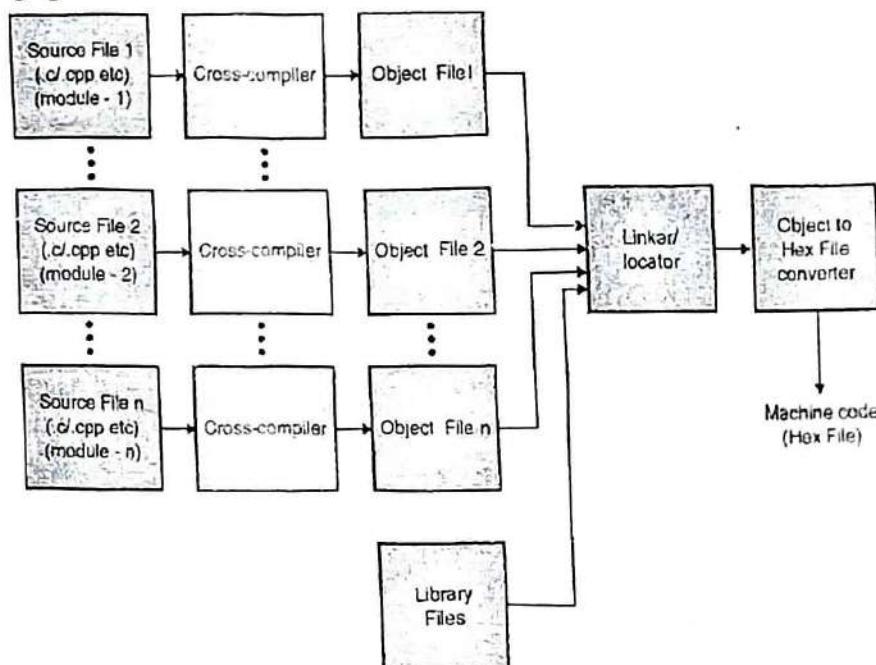


Fig. 3.1.1: High level language to machine language translation process

- The high level programs can be converted to the corresponding assembly program using various software tools like "Keil", which is a cross compiler. Linker/locater combines the object codes corresponding to the source codes and library object files into absolute single executable code. Object to Hex file converter converts the absolute executable code into Hex file that can be dumped into the program memory of target processor/controller.

3.1.1 Comparison between Object Oriented and Procedural Language

Sr. No.	Procedural Language	Object Oriented Language
1.	Emphasis is on procedure or method of doing things	Emphasis is on the data instead of the procedure
2.	In this case the programs are divided into small reusable functions	Program is made up of objects that have functions to work on it.
3.	The data is passed amongst the functions using global variables	The objects are bound with the functions that operate on it

Sr. No.	Procedural Language	Object Oriented Language
4.	Data can be transferred easily amongst the functions	The data of an object can be accessed only by the functions of the same class.
5.	It uses the top-down approach for the programmers	In this case the programmer has to use the bottom-up approach
6.	Example : Pascal, C	Example : C++, Java

3.1.2 Advantages of High Level based Embedded Software Development

1. Less development time

Since the high level programming languages are structured programming languages, it is fast and easy to develop them. This will shorten the time for the embedded software development.

2. Developer independency

High level language syntax is universal and a program written in the high level language can be easily understood by anyone who knows the syntax of the language.

3. Portability

The programs made in high level language is portable i.e. a program written for a microcontroller can be used on multiple microcontrollers.

3.2 Programming in Embedded C (From Embedded System and Real-time Perspective)

- The main concern in microcontroller programming is program memory space. The machine code generated by the assembly language program is much smaller than the machine code generated by the 'C' program.
- But programming in C, has following advantages :
 1. Programming in C is less tedious and less time consuming as compared to assembly.
 2. It is easier to modify and update.
 3. The program requires slight modifications to be written for some other microcontroller i.e. the entire program need not be changed.
 4. C also provides some library functions.
- When we write a program in 'C' we need a compiler to convert it into the machine language.

- Here we will be using Pentium processor based compiler to do this programming. The compiler has to make a code of 8051 processor from the C program given to it such a compiler is called as **cross compiler** i.e. it produces the machine code for another processes, and not as that used in the system. The widely used cross compiler for PIC18F458 and many other PIC microcontrollers is "MPLAB".
- Keil uvision has a header file for every microcontroller. This header file contains variable declaration, macro definitions and definitions for Special Function Registers (SFRs).
- The header file for PIC18F458 is "P18F458.h". The latest MPLAB IDE requires including the header file "xc.h" instead of "P18F458.h". This file has given names for all the SFRs and hence it becomes convenient for the programmer to write the program directly by using these names.
- The names given to the SFRs are as used by us in normal programming.
- Embedded software is usually developed using system called **Host system** such as desktop PC or laptop. The reason is that the embedded systems generally don't have the hardware such as keyboard, monitor, and high capacity memory for embedded software development. Using tool chain such as text editor, cross compiler, linker / locator, object code to Hex file convertor, embedded 'C' code can be converted into target specific processor / controller Hex file which then dumps into target processor / controller memory via serial link.

3.2.1 Use of Modifiers

- The C language allows us to modify the default characteristics of simple data types. Mainly, these data type modifiers alter the range of allowable values. Type modifiers apply to data only, not to functions.

- We can use them with variables, parameters and returned data from functions. Some type modifiers can be used with any variable, while others are used with a set of specific types.

1. Value constancy modifiers

- Const modifier is used to create variable with unchangeable values. Suppose we have embedded processor with floating point processing unit and in the program we want to use π , whose value is constant. `const float PI=3.1415926;`
- When the program is compiled, the locator allocates the ROM space for PI variable.
- Volatile variables are the variables whose value may change by external means such as occurrence of some event. For example, a variable that is "stored" at the location of a port data register will change as the port value changes.
- Using the volatile keyword informs the compiler that it cannot depend upon the value of a variable and should not perform any optimizations based on assigned values.

2. Allowable value modifiers

By default, integer data types include both positive values and negative values. The signed keyword forces the compiler to use MSB of variable as a sign bit. By default short, integer and long data types are signed. Unsigned keyword is a modifier for short, integer and long to permit only positive values.

3. Storage class modifiers

Storage class modifiers control memory allocations for declared identifiers. C supports four storage class modifiers that can be used in variable declaration : extern, static, register and auto. Only extern is used in function declarations.

4. The extern modifier

The extern modifier is used to indicate a particular data element or a function is externally defined or global member.

5. The static modifier

Variables used in the function block are restricted to function only. Modifier static changes local variable in the function block to global variable that can be accessible outside the function.

When the variable is declared with static modifier, the compiler reserves memory space for it. It does not save on the local parameter stack. If the variable is initialized in the program, then ROM is allocated by the locator. If variable is declared with static modifier without initialization, then RAM is allocated by the locator.

6. The register modifier

When variable is declared with the register modifier, this declaration tells the compiler to optimize access to the variable for speed. A CPU register is temporarily allocated when needed. There is no RAM or ROM allocation.

7. The auto modifier

When a variable is initialized in the program with modifier 'auto' or no modifier, locator allocates ROM for that variable else allocates RAM.

3.2.2 Interrupt Service Routines (ISR)

- At the end of each instruction cycle, the processor checks to see if any interrupts have been requested. Therefore whenever interrupt occurs, it won't be immediately checked by microprocessor. Microprocessor first completes execution of current instruction and then checks for an interrupt.
- The sequence of responding to the interrupt is normally as follow :
 1. First, microprocessor will complete execution of current instruction.
 2. It checks for any internal interrupt, suppose the same is not present.
 3. After knowing the interrupt, the next sequence will be executed which is common to all interrupt.
 - (a) Push flag register.
 - (b) Disable INTR input and single step function
 - (c) Push returns address.
 - (d) CALL INTERRUPT service routine.
This CALL is equivalent of an intersegment indirect called instruction.
 - (e) At the end of ISR, user will write IRET i.e. return from an interrupt.
 - (f) In response to IRET, microprocessor will pop the return address.



3.2.3 Macros

- When the repeated group of instructions is too short not appropriate to be written as a procedure, we use a macro. A macro is a group of instructions that perform a task. Each time we call the macro in our program the assembler will insert the group of defined instructions in place of "call".

The macros are useful for following purposes :

1. To simplify and reduce the amount of repetitive coding.
2. To reduce the errors caused by repetitive coding.
3. To make the assembly language program more readable.
4. A macro executes faster because, there is no need of call and return.

The basic format of a Macro is

```
macro name MACRO , Define macro
    ; Body of macro
ENDM ; End macro.
```

- The MACRO directive on the first line tells the assembler that the instructions that follow up to ENDM are a part of macro definition. The ENDM directive ends the macro definition. The instructions between MACRO and ENDM comprise the body of macro definition.

Example:

```
INIT MACRO      ; define MACRO.
MOV AX, @ Data
MOV DS, AX      } ; Body of MACRO
MOV ES, AX
ENDM           ; END MACRO.
```

The assembler places the macro instructions in the program when it is invoked, this is called as Macro expansion. 3.2.4 'C' Program Elements

- Before going for programming, let us see a very important thing required for programming i.e. the tokens of a programming language.

3.2.4 C Tokens

- The C programs are made up of different things termed as tokens.
- The different tokens of C are its,

1. Character set
2. Keywords
3. Identifiers
4. Constants and variables
5. Data types
6. Operators

- We will see these tokens in the subsequent sub-sections.

3.2.4(A) Character Set of C

- When we will be writing C programs, all these will be found in our program. Table 3.2.1 gives a list of C character set.

Table 3.2.1 : Character Set of C

Sr. No.	Characters	List included
1.	Alphabets (Upper case and lower case)	A, B, CZ a, b, c,z
2.	Digits (numbers)	0,1,2,...9
3.	Special symbols (all those seen on a keyboard, nothing besides that)	<> {} () [] , . ; : ! ? ' " / + * = % & # @ \ ~ ` \$ ^ _ - (The names used for these symbols are given in the Table 4.2.2)
4.	Other special characters	Blank Space, Tab, Carriage Return (Enter Key)

3.2.4(B) Keywords

- These are some special words that have a predefined meaning for the C compiler. Hence, these words cannot be used as identifiers (identifiers are discussed in Section 3.2.4(C)).
- These are a set of words which are reserved for the certain operations and hence are also sometimes referred as reserved words.



- All keywords are in lower case.
- The keywords used in C are as given as follow :

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

- These keywords will be used in different places in programming.
- These keywords include all those keywords also as declared by ANSI (American National Standards Institute) C. This institute has published the standards to be used in C programming language.

3.2.4(C) Identifiers

- Identifiers are names given to different user defined things like variables, constants, functions, classes, objects, structures, unions, etc. While making these identifiers we need to follow some rules. These rules are stated as follow :

 1. The identifier can consist of alphabets, digits and a special symbol i.e. '_' (underscore).

2. An identifier cannot start with a digit. It can start either with an alphabet or underscore.
3. It cannot contain any special symbol except underscore. Blank spaces are also not allowed.
4. It cannot be a keyword.
5. It is case sensitive i.e. an alphabet capital in one identifier with same name in another identifier with that alphabet small case will be considered different (for more details see examples in this section).
6. Earlier, there was a limit of the length of the Identifier to be 32 characters, but now this limit is removed. Hence, an Identifier can be as long as required and minimum of one character.

- A list of valid and invalid identifiers is given below with reasons wherever required.

1. simple interest: Valid
2. char: Invalid, because it is a keyword
3. 3friends : Invalid, because starts with a digit
4. _3friends : Valid
5. Simple interest : Invalid, because blank spaces are not allowed
6. #3friends : Invalid, because no special symbol except underscore is allowed.
7. void : Invalid, because keyword not allowed.
8. Void : Valid, case sensitive.

3.2.4(D) Scalar Data Types in C

University Questions

- Q. Write a short note on any two data types used in embedded C programming.
 Q. Write a short note on any two C data types for PIC 18 microcontroller.
 Q. Explain Integer and float data types in detail.

SPPU - Dec. 16, 4 Marks

SPPU - Oct. 19, 4 Marks

SPPU - Dec. 19, 6 Marks

- The data type decides the type of data and the memory locations required for storing that type of data in the memory.
- The data types of C can be divided into three types: Primitive, Derived and User defined data types. The different primitive types of data that can be used in C are integer, character, fraction type numbers, etc.
- Table 3.2.2 shows the different primitive data types and the memory space required for storing them.



Table 3.2.2 : Data types

Sr.No.	Data type	Type of data to be stored	Range	Space required in memory (bytes)
1.	char	1 character in ASCII form	- 127 to 128	1
2.	signed char	1 character in ASCII form	- 127 to 128	1
3.	unsigned char	1 character in ASCII form	0 to 255	1
4.	int	Integer nos.	- 32768 to 32767	2
5.	signed int	Integer nos.	- 32768 to 32767	2
6.	unsigned int	Integer nos.	0 to 65535	2
7.	short int	Integer nos.	- 32768 to 32767	2
8.	long int	Integer nos.	- 2147483648 to 2147483647	4
9.	signed short int	Integer nos.	- 32768 to 32767	2
10.	signed long int	Integer nos.	- 2147483648 to 2147483647	4
11.	unsigned short int	Integer nos.	0 to 65535	2
12.	unsigned long int	Integer nos.	0 to 4294967296	4
13.	float	Fraction nos.	$3.4e^{-38}$ to $3.4e^{38}$	4
14.	double	Fraction nos.	$1.7e^{-308}$ to $1.7e^{308}$	8
15.	long double	Fraction nos.	$3.4e^{-4932}$ to $1.1e^{4932}$	10

Note:- The arithmetic operations can be done even on char type of data. This is because the processor stores ASCII (American Standard Code for Information Interchange) to store the characters.

- The ASCII value for capital 'A' is 65, 'B' is 66, and 'C' is 67 and so on. While the ASCII values for small alphabets are 97 for 'a', 98 for 'b', and 99 for 'c' and so on.
- The user defined data types are structure, union, class and enumeration. The derived data types are array, function, pointer and reference.
- The derived and user defined data types will be studied in the subsequent chapters wherever it is needed.
- Constants are used to declare the values that remain constant, for e.g. value of pi.
- The use of constants in program will be discussed later wherever required.
- Variables are values given to identifiers that can change their values during the execution of the program.

3.3 Operators

Operators are used to indicate the operation to be performed. The data on which the operation is to be performed are called as operands. The operators are classified based on the number of operands required for the given operation. Let us see these operators in the following sub-sections.

3.2.4(E) Constants and Variables

- Constants are values given to the identifiers that do not change their values throughout the execution of the program.
- Constants can be defined either by writing the keyword const before the data type or by using #define.

3.3.1 Unary Operators

These are operators that require only one operand. Let us discuss these operators one by one.

1. Unary minus (-)

- The symbol shown in the bracket is used as unary minus operator.
- It returns the negative value of the variable to which it is preceded.
- For e.g. if $x = 3$ and $y = 6$ then $y = -x$; will make the value of y as -3 .

2. Casting operator (()) or type conversion

- It is many times required to convert one data type to another.
- The casting operator is used for type conversion i.e. it can be used to convert a data of one type to another data type.
- For e.g. if we have `int x=3;`
`float y=5.6;`
 then the statement, `x=(int) y;`
 will result in the value of x as 5.

- We will see the use of this operator in the programming.

3. Logical Not operator (!)

- The symbol shown in the brackets i.e. the exclamation mark is used as a logical not operator. It is used to check certain conditions in a condition statement. It performs the logical not of the result obtained from the expression on its right.
- For e.g. if $x=1$, then $y=\text{!}x$;
 will result with y having the value 0.

- We will see some more logical not operator based expressions and program followed by this section.

4. Address of operator (&)

- This operator returns the address of the variable associated with it. It will be studied more deeply in the chapter on Pointers.
- For e.g. if we write $y=\&x$;
 then the memory address allocated to the variable x will be copied into the variable y . For this the variable y must be a pointer variable.

5. Indirection operator or value of operator (*)

- This operator returns the value of the data at the given address.
- For e.g. $z=\ast y$;
 will give the value of the data stored at the address given by y .

6. Scope resolution operator (::)

- It resolves the scope of a variable to be inside the function or outside the function. This operator will also be studied in a later chapter.
- For e.g. if an external variable x is to be accessed then the same can be accessed using the scope resolution operator by the statement, $y=::x$;

7. sizeof operator

- This operator is used to know the size of a variable as required to store its value in memory. It can also be used to find the size of a data type.
- As seen in the Section 3.2.4(d), the space required to store different data type is different ranging from 1 byte to 10 bytes.
- For e.g. the statement `sizeof(int);`
- Will return the value 2, as int requires 2 bytes.
- Another e.g. `int x,y;`

```
char a;
int x=sizeof(a);
float y=sizeof(float);
then x will have 1, and y will have 4
```

8. Bitwise not operator (~)

- This operator is used to perform bitwise NOT operation. We have also seen some examples of NOT operations.
- The bitwise NOT operator can be used to perform binary NOT operation. This operation can be performed by using the operator given in brackets i.e. `~`.
- For e.g. If $x=3$,

`then y=~x;`
 will result in y having a value of -4



9. Increment Operator (++)

- It returns the value of the variable added with one and stores the result in the variable itself.

For e.g. if $x=5$,

then $x++;$

will make the value of x equal to 6.

- This " $x++;$ " (also called as post increment operator) can also be written as " $++x;$ " (also called as pre increment operator).
- It can also be used to store the result in another variable. But in this case the post increment and pre increment statements will have different behaviour as explained below with examples.
- In post increment case,
for e.g. if $x=5$, then $y=x++;$
will make the value of y equal to 5 and x equal to 6.
As the name says post increment, it first gives the previous value and then increments.
- In pre increment case, for e.g. if $x=5$, then $y=++x;$
will make the value of y equal to 6 and x equal to 6.
As the name says pre increment, it first increments and then gives the incremented value.
- More such examples will be seen with programs.

10. Decrement Operator (--)

- It returns the value of the variable subtracted with one and stores the result in the variable itself.
- For e.g. if $x=5$,
then $x--;$
will make the value of x equal to 4.
- This " $x--;$ " (also called as post decrement operator) can also be written as " $--x;$ " (also called as pre decrement operator).
- It can also be used to store the result in another variable. But in this case the post decrement and pre decrement statements will have different behaviour as explained below with examples.
- In post decrement case, for e.g. if $x=5$, then $y=x--;$
will make the value of y equal to 5 and x equal to 4.

As the name says post decrement, it first gives the previous value and then decrements.

- In pre decrement case, for e.g. if $x=5$, then $y=--x;$
will make the value of y equal to 4 and x equal to 4.
- As the name says pre decrement, it first decrements and then gives the decremented value.
- More such examples will be seen with programs.

3.3.2 Binary Operators

- Operators that require two operands are called as binary operators.
- These operators are further classified into various types namely the Arithmetic operators, Logical operators, Bitwise operators and Relational operators.
- We will see these operators one by one in this section.

1. Arithmetic operators

- This set includes the basic arithmetic operators to perform basic arithmetic operations like addition, subtraction, multiplication and division. There are five operators in this set. They are:
 1. * to find the product
 2. / to find the quotient after division
 3. % to find the remainder after division
 4. + to find the sum
 5. - to find the difference
- One important thing to be noted here is that the '/' operator returns only the quotient, while the '%' operator (also called as MOD operator) returns the remainder after division.
- MOD operator is possible only for int or char type of data. It doesn't work on float and double type of data.
- Example of each of these operators
 1. $2 * 2 = 4;$
 2. For int type of data, $5 / 3 = 1$;
For float type of data, $5 / 3 = 1.67$
 3. $5 \% 3 = 2;$
 4. $2 + 2 = 4;$
 5. $3 - 2 = 1;$

2. Bitwise operators

- These operators work bitwise on a data.
- They perform different operations on bits of a data like AND, OR, EXOR and NOT.
- The operators are listed below :
 1. \sim to perform bitwise NOT operation.
 2. $\&$ to perform bitwise AND operation.
 3. $|$ to perform bitwise OR operation.
 4. \wedge to perform bitwise EXOR operation.
 5. $<<$ to perform bitwise left shift operation.
 6. $>>$ to perform bitwise right shift operation.
- These operators are used to perform bitwise binary operations on the data.
- As we have addition, subtraction operations in decimal data for performing arithmetic operations; similarly AND, OR, NOT are basic bitwise operations on the binary data.
- The result of these operations can be understood from the following examples :

1. $5 \& 3 = 1$

$$\begin{array}{r} (5)_{10} = (0 \ 1 \ 0 \ 1)_2 \\ (3)_{10} = (0 \ 0 \ 1 \ 1)_2 \\ \hline (0 \ 0 \ 0 \ 1)_2 = (1)_{10} \end{array}$$

2. $12 | 9 = 13$

$$\begin{array}{r} (12)_{10} = (1 \ 1 \ 0 \ 0)_2 \\ (9)_{10} = (1 \ 0 \ 0 \ 1)_2 \\ \hline (1 \ 1 \ 0 \ 1)_2 = (13)_{10} \end{array}$$

3. $8 ^ 10 = 2$

$$\begin{array}{r} (8)_{10} = (1 \ 0 \ 0 \ 0)_2 \\ (10)_{10} = (1 \ 0 \ 1 \ 0)_2 \\ \hline (0 \ 0 \ 1 \ 0)_2 = (2)_{10} \end{array}$$

4. $\sim 7 = -8$

$$\begin{array}{r} (7)_{10} = (0 \ 1 \ 1 \ 1)_2 \\ (1 \ 0 \ 0 \ 0)_2 = (-8)_{10} \end{array}$$

(According to C, wherein it will take more no. of bits and hence -8 will be the result). Hence; in general the $\sim x$ is always equal to $-(x+1)$.

5. $10 << 2 = 40$

Assuming the data to be char i.e. 8 bit data,

$$(10)_{10} = (0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0)_2$$

After shifting left once $(0 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0)_2$

After shifting left for the second time

$$(0 \ 0 \ 1 \ 0 \ 0 \ 0 \ 0)_2 = (40)_{10}$$

Note : When shifting to left, each of the bit is shifted left. The first bit is lost and the last bit is inserted as 0.

6. $13 >> 3 = 1$

Assuming the data to be char i.e. 8 bit data

$$(13)_{10} = (0 \ 0 \ 0 \ 1 \ 0 \ 1 \ 1)_2$$

After shifting left once $(0 \ 0 \ 0 \ 0 \ 1 \ 0 \ 1)_2$

After shifting left for the second time
 $(0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0)_2$

After shifting for the third time
 $(0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1)_2 = (1)_{10}$

Note : When shifting to right, each of the bit is shifted right. The first bit is inserted as 0 and the last bit is lost.

3. Logical operators

- Logical operators follow the same truth table as for the bitwise operators but they are used to check conditions instead of performing operations on a data.
- The logical operators are AND and OR. The symbols used for these operators in C are $\&\&$ and $\|$ respectively.
- For example a statement will result in "true" i.e. 1 if the value of y is greater than 5 AND less than 10, else it will result in false i.e. 0.
 $y > 5 \&\& y < 10;$
- Another example a statement will result in "true" i.e. 1 if the value of y is greater than 5 OR equal to 2, else it will result in false i.e. 0.
 $y > 5 \| y == 2;$
- Logical operators will be understood in more details with the expressions and program examples followed by this section.

4. Relational operators

- The operators like '<' (less than) and '>' (greater than) used in the above examples are relational operators.



- We have seen the example also of the “==” (equality operator) in the above logical operators.
- A list of all the relational operators is given below :
 1. == used to check if the two things are equal.
 2. != used to check if the two things are not equal.
 3. < used to check if the first data is less than the second one.
 4. > used to check if the first data is greater than the second one.
 5. <= used to check if the first data is less than or equal to the second one.
 6. >= used to check if the first data is greater than or equal to the second one.
- The examples of these relational operators are as shown with the logical operators above.

3.3.3 Ternary Operators

- An operator that requires three operands is called as a ternary operator.
- There is only one ternary operator in C. This operator is used to check a condition and accordingly do one of the two things based on the condition being true or false.
- The syntax (way of writing) of this operator is as given below:

(condition) ? <value if condition is true> : <value if condition is false>;

- Hence; as shown in the syntax, first the condition is to be written in brackets followed by a question mark (?). Then the operation that is to be performed if condition is true and then a colon (:) followed by the operation to be performed if the condition is false.
- For example :

$z=(x>y)?x:y;$

This statement will put the value of x into z if the given condition i.e. $x > y$ is true. Else; the value of y will be put into z. Hence, the value of the greater variable will be put into z.

- A slightly complicated use of this operator is to find the greatest of three numbers as shown in the example,

$g=(x>y)?((x>z)?x:z):((y>z)?y:z);$

This statement will give the largest of x, y and z into the variable g.

3.3.4 Assignment Operators and Statements

- These operators are used to assign the value of the expression or variable on the right of the assignment operator to the variable on its left.
- The simple assignment operator is '='. But there are some more assignment operators called as composite assignment operators.
- The different assignment operators are as listed below :
 1. = : This operator assigns the value of the expression or variable on its right to the variable on its left. For e.g. $y=x+2;$
 2. += : This operator adds the variable on its left and right and the result is put into the variable on its left. For e.g. $y+=x;$ is same as $y=y+x;$
 3. -= : This operator subtracts the variable on its right from the variable on its left and the result is put into the variable on its left. For e.g. $y-=x;$ is same as $y=y-x;$
 4. *= : This operator multiplies the variable on its left and right and the result is put into the variable on its left. For e.g. $y*=x;$ is same as $y=y*x;$
 5. /= : This operator divides the variable on its right from the variable on its left and the result is put into the variable on its left. For e.g. $y/=x;$ is same as $y=y/x;$
 6. %= : This operator finds the remainder by dividing the variable on its right from the variable on its left and the result is put into the variable on its left. For e.g. $y\%=x;$ is same as $y=y \% x;$
 7. &=: This operator ANDs the variables on its left and right and the result is put into the variable on its left. For e.g. $y\&=x;$ is same as $y=y \& x;$
 8. |= : This operator ORs the variables on its left and right and the result is put into the variable on its left. For e.g. $y|=x;$ is same as $y=y|x;$

9. $\wedge=$: This operator EXORS the variables on its left and right and the result is put into the variable on its left. For e.g. $y \wedge= x$; is same as $y = y \wedge x$;

10. $<<=$: This operator shifts in left direction the variable on its left for the number of times indicated by the variable or value on right and the result is put into the variable on its left. For e.g. $y <<= x$; is same as $y = y << x$;

11. $>>=$: This operator shifts in right direction the variable on its left for the number of times indicated by the variable or value on right and the result is put into the variable on its left. For e.g. $y >>= x$; is same as $y = y >> x$;

3.3.5 Selection Operators

- These operators are used to select certain element of a set of elements. Here, we will make a list of these operators and see the detailed use of these operators when we study the corresponding topic where these operators are required.
- The different operators in this set are listed below:
 1. [] : This operator is used to select an element of Array.
 2. . : This is called as period operator and is used to select an element of a structure or union.
 3. -> : This is used to select an element of a structure or union pointed by a pointer.
 4. () : This is called as a function call operator and is used to call or select a function.
 5. , : The comma (,) operator is used to separate the different values etc.

3.3.6 Decision Making and Branching Statement

3.3.6(A) if-else Selective Statement

University Question

- Q. Explain the following control structure used in embedded C
 (i) If then else construct

SPPU - May-16, 4 Marks

- This is a very important statement used to check the condition and accordingly execute a set of statements, based on whether the condition is true or false. Hence it is called as selective statement. It selectively executes some statements and doesn't execute some.

- The syntax of the if-else condition is as follow.

if(condition)

{

statements1;

}

else

{

statements2;

}

}

- The set of statements named as statements1 in the above syntax are executed if the condition given with the if statement is true. The set of statements 2 are not executed in this case.
- If the condition specified in the if statement is false then the statements named as statements2 in the above syntax are executed. The set of statements 1 are not executed in this case.

Note: The else part is optional i.e. we can have a if statement without the else statement. As seen in the above given syntax, only the first half i.e. we can have if statement as shown below.

if (condition)

{

Statements

}

will be there

3.3.6(B) switch-case Statement

University Question

- Q. Explain the following control structure used in embedded C
 (i) Switch construct

SPPU - May 16, 2 Marks

- In the previous section we have seen the if-else ladder. A better solution of this is switch-case. Using switch-case the if-else ladder can be implemented in a much better way.
- The syntax of switch-case is given below:

```
switch(expression / variable)
{
    case label1: statements;
        break;
    case label2: statements;
        break;
    case label3: statements;
        break;
    .
    .
    .
    case label n: statements;
        break;
    default : statements;
}
```

Note : Break statement transfers the control outside the current loop. We will see some more cases of break statement along with the for / while / do-while statements.

- The expression or variable can be given in the brackets associated with the switch statement. The values of this expression/ variable are the labels associated with the cases inside the switch statement.
- The value of the expression/ variable is first compared with the label1. If they are equal, then the statements followed by the corresponding case are executed. The break statement followed by these statements, transfers the control after the switch statement.
- If the expression/ variable are not equal to label1, then it is directly compared to label2 without executing the statements followed by the label1.

- Hence the statements of only that case are executed with the correct label value of the expression/ variable.

Note :

1. Break statement is not compulsory. But if the break statement is not written everything will be executed after the case statements.
2. Default is not necessary. But in case if default is not written and some case occurs which is not considered then there will be no operation performed and the user will not be able to realize the problem in the program.
3. The label can only be value; it cannot be a condition or an expression.
4. The brackets associated with the switch statement can have either the variable or expression and no condition.

3.3.7 Looping Statements

- In programming we need to sometimes control the flow of operation other than just the sequential statements. In this case we need the control statements.
- Control statements are classified into two types viz. the iterative statements and conditional statements.
 1. Iterative statements are used to perform certain operation repetitively for certain number of times. In C we have three iterative statements viz. for loop, while loop and do-while loop. Iterative statements are also called as repetitive statements as they repeat a set of statements for a given number of times.
 2. Conditional statements are used to perform the operations based on a particular condition i.e. if a condition is true perform one task else perform another task. In C we have two conditional statements namely if-else statements and switch-case statements. Conditional statements are also called as selective statements i.e. these statements select a particular statement to be executed based on the condition.

In the subsequent sections we will see all the iterative statements one by one in detail. We will be writing only the Pseudo code type algorithms from here onwards. For some programs the stepwise algorithm will be given for ease of students to understand the program.

3.4 C-Control Structures for Iteration

3.4.1 for Loop

- For is a iterative statement. It is used to repeat a set of statements number of times. The syntax (method of writing) for statement is given below.

for(initialization; condition; increment/decrement / updating)

statements

}

- The sequence of execution of the for loop is such that the initialization statements are executed first. These initialization statements are executed only once. They are used to initialize the values of the variables.
- The second step is checking the condition specified. There can be only one condition. If more than one conditions are required they can be combined using the logical AND, OR operators. If the condition is false the execution of the loop will be terminated i.e. the execution will go outside the braces of for loop, if the condition is not true.
- The third step is to execute all the statements inside the curly braces. These statements will be executed sequentially. The number of statements can be of any count. There can be another control statement if required inside one control statement.
- The fourth step is the increment / decrement or updating operations. These operations are not necessarily increment or decrement operations, but mostly these are increment decrement and hence called so. We can update the variables even here before starting the next iteration of the iterative statements.
- Finally the control goes back to the second step. As said the first step is executed only once, the steps that are repeated continuously are the second, third and fourth steps. After the fourth step the condition is checked again. If the condition is true the execution continues, else the control goes outside the for loop i.e. the curly braces.

3.4.1(4) while and do-while loops

University Question

- Q. Explain the following Control structures using Pseudo code
 (i) While construct

SPPU - May 18, 2 Marks

- While and do-while loops are also the repetitive operations.
- The operations are slightly different than the for loop but the same operations can be implemented by for while or do-while loops. Although there is one major difference in a do-while loop wherein one particular operation cannot be implemented. This will be discussed later in this section.
- The syntax of while loop is given below:

while(condition)

{

statements

}

while(condition)

{

- One major point to be noted is that in case of do-while loop, the statements are executed at least once even if the condition is not true for the first statement. On the other hand, in case of for loop and while loop, even for the first time the statements are executed only if the condition is true.
 - These statements like "break", "continue" and "goto" transfer the control to a different place in the program. We will see the operation of each of these. We have already seen the use of break statement in switch-case. Let us see the use of continue and break in other loop statements.
 - The break statement neglects the statements after it in the loop and transfers the control outside the loop as shown in Fig. 3.4.1. The Fig. 3.4.1 shows the operation of break statement in each of the loop statements i.e. for, while and do-while statements.

```
for(initialization; condition; inc / dec)
{
    -
    -
    -
    -
    break;
    -
    -
    -
}
}
```

(a) Operation of break statement in a for loop

```

while(condition)
{
    -
    -
    -
    -
    break;
}

do
{
    -
    -
    -
    -
    break;
} while(condition);

```

(b) Operation of break statement in a while loop

(c) Operation of break statement in a do-while loop

Fig. 3.4.1

- The `continue` statement also neglects the statements after it in the loop and transfers the control back to the starting of the loop for next iteration. The Fig. 3.4.2 shows the operation of `continue` statement in each of the loop statements i.e. for, while and do-while statements.

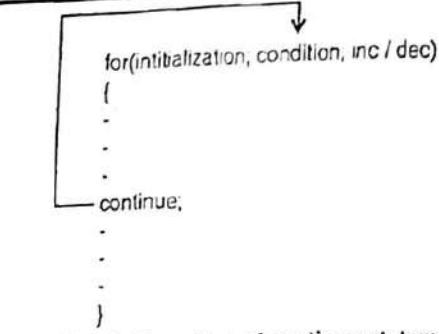
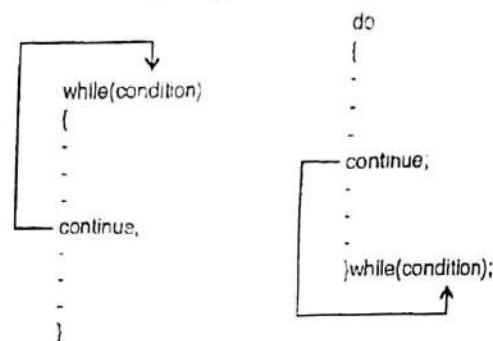


Fig. 3.4.2 (a) : Operation of continue statement in a for loop



(b) Operation of continue statement in a while loop

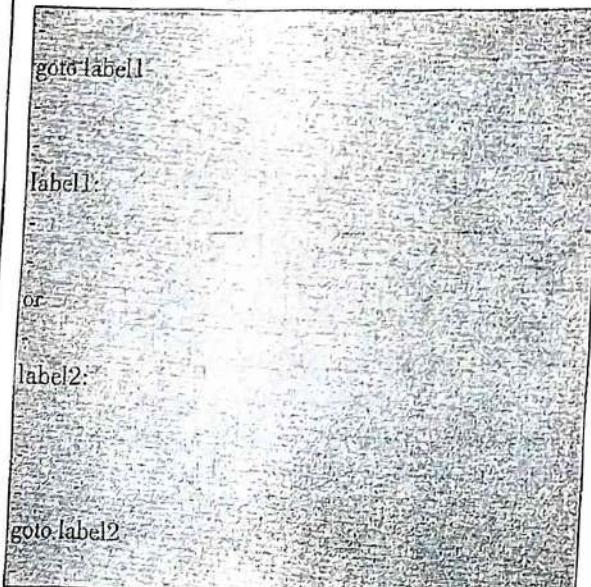
(c) Operation of continue statement in a do-while loop

Fig.3.4.2

- The `goto` statement transfers the control to the label specified with the `goto` statement. A label is any name given to a particular part in the program. The label is to be followed with a colon (:).
 - The syntax of a `goto` statement is as shown below:

- The syntax of a goto statement is as shown below:

The by-mail *get-a-statement* is as shown below:



- Thus it shows that the label can be before or after the goto statement and hence the control can be transferred to earlier or next statements using a goto statement.

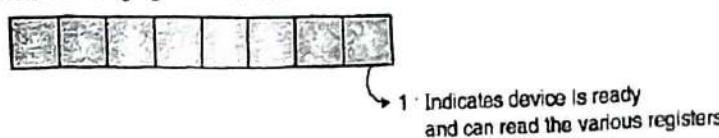


Fig. 3.4.3: Format of STATUS register

We can read the various registers of device using various loop statement.

```
// using while loop
while (STATUS != 0x01); // Wait till STATUS register = 0x01

// using do while loop
do
{
} while (STATUS != 0x01); // Loop till STATUS register = 0x01

// using for loop
for (; STATUS != 0x01;); // Repeat until STATUS register = 0x01
```

3.4.2 Arrays and Pointers

- An array is a group of related data items. Arrays are usually declared with data type of array, name of the array and the number of related data items (elements) to be placed in the array. For example, the following array declaration.

```
char array1[5];
```

declares a character array with name 'array1' and reserves memory for 5 character elements as shown in Fig. 3.4.4.

a[0]	1
a[1]	2
a[2]	3
a[3]	4
a[4]	5

Fig. 3.4.4: Memory allocation for an array 'a' of 5 character elements

1. Pointers

- A pointer is a variable that contains the address of another variable in the memory. Pointer is a memory pointing based technique for variable access and modification. Pointers are very useful in,
 - (i) Accessing and modifying variables that are defined outside the function.
 - (ii) Improves the speed of execution.

- (iii) Accessing contents within a block of memory.
- (iv) Reducing the length and complexity of a program.
- (v) Dynamic memory allocation.

- To understand the pointer concept, let us look at the data memory organisation of microcontroller AT 89C51. AT 89C51 has 128 bytes of internal data memory, the memory is organised as,

Consider the following statement,

```
char input = 0x20;
```

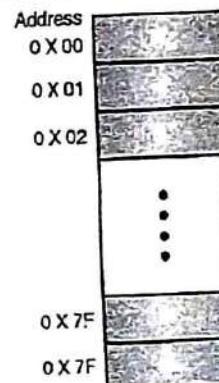


Fig. 3.4.5: Data memory organization for 8051

- Compiler will assign a memory location to the variable anywhere within the internal memory 0x00 to 0x7F. Let us assume that compiler has chosen the address location 0x55 for input Fig. 3.4.6 shows the representation of variable.

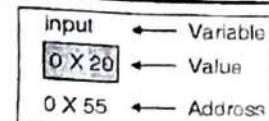


Fig. 3.4.6: Representation of a variable

- We can access to the value 0×20 using either name of the variable or the address 0×55 .
- Suppose we assign the address of input variable to a variable p. The Link between the variable p and input shown in Fig. 3.4.7. The address of p is 0×00 .

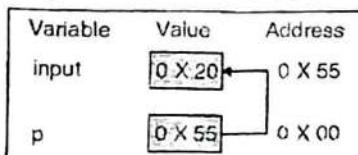


Fig. 3.4.7: Pointer as variable

- Address of variable can be accessed with the help of operator '&'.
- The statement `p = &input;` will assign address of variable input to variable p. Since the pointers are the variables that must declared before usage. The declaration of a pointer variable takes the following form.

```
data type * ptr_name;
```

- This tells the compiler three things about variable `ptr_name`.
 1. The asterisk (*) tells that the variable `ptr_name` is a pointer name.
 2. `ptr_name` needs a memory location.
 3. `ptr_name` points to a variable of type `data type`.

In our example, statement

```
char * p;
```

- Declares the variables p as a pointer variable that points to a character data type.
- Using the above pointer declaration and initialisation we can access the data memory of AT 89C51 microcontroller as follows :

```
char input = 0 X 20; // Declaring input as
                     // character variable with initial value 0 X 20
char * p; // Declaring a character pointer p
p = &input; // Assign the address of input to p
char a; // Define variable a
a = * p; // indirection operator returns the
```

```
// value of variable where address specified
// in variable p. This statement will
// assign 0 X 20 to variable a.
```

3.4.3 Function in C

- Function is a self-contained block of statements that performs a given task. The programs we were writing until now had only one function i.e. the main function. In this chapter most of our programs will have two or more functions.
- The syntax of a function is given below :

```
return_type function_name (argument_list)
```

statements:

- Let us see the meaning of each of these terms used in the syntax,

1. "return_type" is the data type of the data to be returned by the function. The main function we have always been writing has return type of void i.e. no data is returned. The return type of the data can be any of the data types like int, char, float etc.
2. The "function_name" can be anything as far as the rules of identifiers seen earlier are considered. For example "main" is the name of function we have been writing until now.
3. In the brackets we write the "argument_list". Argument list is a set of data types along with the identifiers to accept a set of data passed to the function. Until now we were not passing any data to the main function, and hence the brackets have always been empty. In the programs we will be writing in this chapter we will be having functions that accept the data from another functions.
4. Statements inside the function are the statements to be executed when that function is called. We can call a function using the function call operator. We will see about all these in the subsequent programs.

Declaration and usage of a function pointer with `typedef` is given as follow :

```
typedef int(* fncptr) () ;
fncptr fptr1 ;
```

The following program illustrates the declaration, definition and the usage of function pointer.

```
#include <stdio.h>
void cube (int x) ;
void main ()
{
    void (*fptr1) (int) ; // Declare a function pointer
    fptr1 = cube; // Assign the address of
    function cube to // function pointer variable
    fptr1 (2); // Invoke the function through
    function pointer
}

// Function for printing the
// cube of a number.

void cube (int x)
```

```
{ printf("cube of %d = %d \n", x, x * x * x);
}
```

- Function pointer can be useful in late binding. Based on the need arises in the application, we can invoke the required function by the binding the function with the right function pointer. This less use of 'if' and 'switch-case' statements with function names. Function pointers are useful for handling the situations which require passing of a function as argument to another function.
- Function pointers are frequently used within the functions where the function should be able to work with a number of functions whose name are known only at the run time.

- The following sample code snippet shows the usage of function pointer as parameter to a function.

```
#include <stdio.h> // Function prototype declaration
void square (int a) ;
void cube (int a) ;
void compute (void (*fptr1) (int), int a) ;
void main ()
{
    void (*fptr2) (int) ; // Declare a function pointer
    //Assign the function address //to the function pointer
    fptr2 = square; // Invoke the function 'square' //through function pointer fptr2
    compute (fptr2, 3); // Assign the function 'cube' address to the function pointer
    fptr2 = cube; // Invoke the function 'cube' through function pointer fptr2
    compute (fptr2, 3); // Function implementations
}

void compute (void (* fptr2) (int), int a)
{
    fptr2 (a) ;
}

void square (int a)
{
    printf("square of %d = %d \n", a, a * a) ;
}

void cube (int a)
{
    printf("cube of %d = %d \n", a, a * a * a) ;
}
```

Array of function pointers declaration using 'typedef' qualifier.



For example

1.

```
typedef int (*fncptr) () ;  
fncptr fptr [5] () ;
```

- The above statement declares an array of pointer to functions, which returns int and takes no parameters, using `typedef` function pointer declaration.
- We can do declaration and initialisation of array of pointers to functions at the same time

2.

```
typedef int (*fncptr1) () ;  
fncptr1 fptrarr [] () = { /* * initialisation */ } ;
```

- Following statement declare and initialise an array of pointers to functions to NULL, which return int and takes no parameters using `typedef` function pointer declaration.

```
typedef int (*fncptr1) () ;  
fncptr1 fptrarr [4] () = {NULL} ;
```

The following code snippet shows the usage of function pointer arrays :

```
#include <stdio.h>  
// Function declaration  
void square (int a) ;  
void cube (int a) ;  
void main ()  
{  
    // Declare a function pointer array of size 2 and initialises to "NULL"  
  
    void (*fptr) [2] (int) = {NULL} ;  
    // Assign the address of function // "square" to fptr [0]  
    fptr [0] = square ;  
    // Invoke the function square //using first element of array of  
    // function pointer  
    fptr [0] (3) ;  
    // Assign the address of function // "cube" to fptr [1]  
    fptr [1] = cube ;  
    // Invoke the function cube //through function pointer fptr [1](3)  
    fptr [1] (3) ;  
}  
void square (int a)  
{  
    printf("square of %d = %d \n", a, a * a) ;  
}  
void cube (int a)  
{  
    printf("cube of %d = %d \n", a, a * a * a) ;  
}
```

3.5 Exam Pack (Review and University Questions)

- Q. Write a short note on any two data types used in embedded C programming.
(Refer Section 3.2.4(D)) **(Dec. 16, 4 Marks)**
- Q. Explain the following control structure used in embedded C : (i) if then else construct
(Refer Sections 3.3.6 and 3.3.6(A)) **(May 16, 4 Marks)**
- Q. Explain the following control structure used in embedded C : (i) Switch constructs
(Refer Section 3.3.6(B)) **(May 16, 2 Marks)**
- Q. Explain the following control structure used in embedded C : (i) While construct
(Refer Section 3.4.1(A)) **(May 16, 2 Marks)**
- Q. Write a short note on any two C data types for PIC 18 microcontroller. **(Refer Section 3.2.4(D))** **(Oct. 19, 4 Marks)**
- Q. Explain Integer and float data types in detail. **(Refer Section 3.2.4(D))** **(Dec 19, 6 Marks)**
-

□□□

4

UNIT - II

I/O Port Programming

4.1 I/O Port Programming

University Question

Q. Explain different I/O ports and associated SFRs of PIC18F458. **SPPU - Dec. 14, 7 Marks**

- All port pins can be designated as input or output, according to the needs of a device that's being developed.
- In order to define a pin as input or output pin, the right combination of zeros and ones must be written in TRIS register. If the appropriate bit of TRIS register contains logical "1", then that pin is an input pin, and if the opposite is true, it's an output pin.
- Every port has its TRIS register. Thus, port A has TRIS A, and port B has TRIS B. Pin direction can be changed during the course of work which is particularly fitting for one-line communication where data flow constantly changes direction.
- PORT A, PORT B, PORT C, PORT D and PORT E state registers are located in bank 0, while TRIS A, TRIS B, TRIS C, TRIS D and TRIS E pin direction registers are located in bank 1.

particular bit of TRIS A is cleared to 0, it will make the corresponding PORT A pin an output pin.

- On a Power-on Reset, these pins are configured as inputs and read as 0
- Reading the PORT A register reads the status of the pins, whereas writing to it will write to the port latch. Read-modify-write operations read from the LATA register and write to the latch of PORTA.
- The RA4 pin is multiplexed with the Timer0 module clock input to become the RA4/TOCKI pin. The RA4/TOCKI pin is a Schmitt Trigger input and an open-drain output. All other RA port pins have TTL input levels and full CMOS output drivers.
- The other PORT A pins are multiplexed with analog inputs and the analog V_{REF+} and V_{REF-} inputs. The operation of each pin is selected by clearing/setting the control bits in the ADCON1 register (A/D Control Register 1). On a Power-on Reset, these pins are configured as analog inputs and read as 0.

Note : On a Power-on Reset, RA5 and RA3, RA0 are configured as analog inputs and read as '0'. RA6 and RA4 are configured as digital inputs.

- The TRIS A register controls the direction of the RA pins, even when they are being used as analog inputs. The user must ensure the bits in the TRIS A register are maintained set, when using them as analog inputs.

4.2 Port A

University Questions

Q. Explain I/O port pins and multiplexed function of port A. **SPPU - Aug. 14 (In Sem.), 4 Marks**

Q. Explain port A of PIC18F458. **SPPU - Dec. 14, 2 Marks**

- PORT A is a 7-bit wide, bidirectional port. The corresponding data direction register is TRIS A.
- If a particular bit of TRIS A is set to 1, it will make the corresponding PORT A pin an input pin while if a

Program 4.2.1 : Port A

Soln.:

CLRF PORTA	; Initialize PORT A by clearing output data latches
CLRF LATA	; Alternate method to clear output data latches
MOVLW 07h	; configured A/D
MOVWF ADCON1	; for digital inputs
MOVLW 0CFh	; Value used to initialize

```
; data direction
MOVWF TRISA      ; Set RA3:RA0 as inputs
; and RA5:RA4 as outputs
```

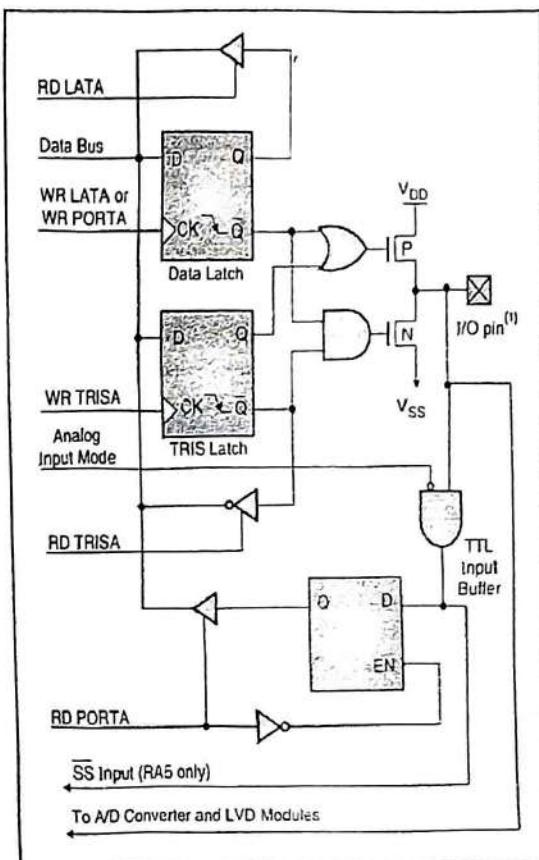


Fig. 4.2.1: RA3 : RA0 and RA5 pins block diagram

- As seen in the Fig. 4.2.1, the TRIS Latch is used to enable the 'P' or 'N' FET and thereby connecting the pin for input or output operation. Also the alternate functions of the pin are implemented by enabling a certain path through the gate.
- In case if the pin is configured as output pin, the "data latch" controls the output of the pin. In case if the pin is configured to be input pin, the same is connected to the latch that stores the contents of the port pin.

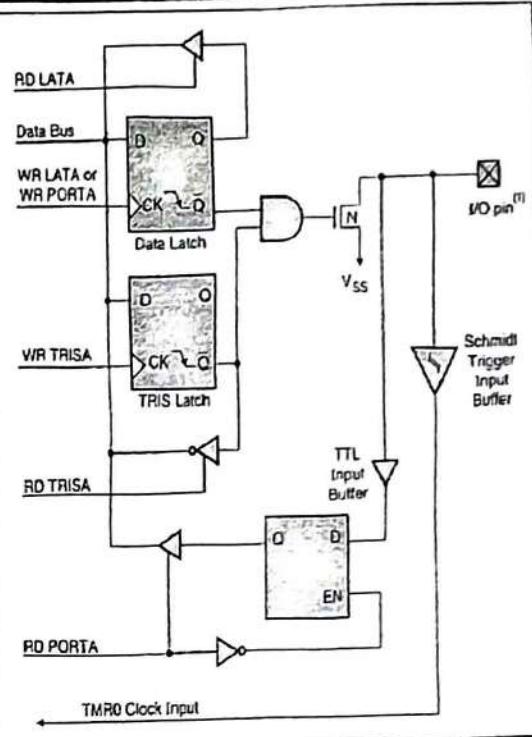


Fig. 4.2.2: RA4/T0CKI pin block diagram

4.3 Port B

University Questions

Q. Explain port B of PIC18F458

SPPU - Dec. 14, 2 Marks

Q. Mention alternate function of Port B.

SPPU - May 19, 4 Marks

- PORT B is an 8-bitwide, bidirectional port. The corresponding data direction register is TRIS B. If a particular bit of TRIS B is set to 1, it will make the corresponding PORT B pin an input pin while if a particular bit of TRISB is cleared to 0, it will make the corresponding PORTB pin an output pin.
- Reading the PORT B register reads the status of the pins, whereas writing to it will write to the port latch. Read-modify-write operations read from the LATB register and write to the latch of PORTB.

Program 4.3.1: Initializing PORT B

Soln.:

```
CLRF LATB    ; Initialize PORT B by clearing
; output data latches
MOVLW 0CFh  ; Value used to initialize data
; direction
```



```
MOVWF TRISB ; Set RB3 : RB0 as inputs
              ; RB5 : RB4 as outputs
              ; RB7 : RB6 as inputs
```

Program 4.3.2 : Write a assembly language program to read data from pin 3 port B (RB3) and transfer it to pin 7 of port C (RC7)

SPPU - Aug. 14 (In Sem.), 6 Marks

Soln.:

Label	Instruction	Comment
	org 0x00	
	GOTO start	
start:	BCF TRISC,7	Set port C pin 7 as output
	BSFTRISB,3	Set port B pin 3 as input
next:	BTFSC PORTB,3	Test port B pin 3
	GOTÖ over	If set, goto over
	BCF PORTC,7	Clear port C pin 7
	GOTO next	Repeat
over:	BSF PORTC,7	Set port C pin 7
	GOTO next	repeat
	end	

Program 4.3.3 : Write an assembly language program to copy the status of PORTB.0 to PORTD.0 without affecting the other pins of both the ports.

SPPU - Aug. 15 (In Sem.), 6 Marks

OR Write a program in assembly language to configure bits RD0 and RB0 as input bits.

SPPU - Oct. 19 (In Sem.), 4 Marks

Soln.:

Label	Instruction	Comment
	org 0x00	
	GOTO start	
start:	BCF TRISD,7	Set port D pin 0 as output
	BSFTRISB,0	Set port B pin 0 as input
next:	BTFSC PORTB,0	Test port B pin 0
	GOTÖ over	If set, goto over
	BCF PORTD,0	Clear port D pin 0
	GOTO next	Repeat
over:	BSF PORTD,0	Set port D pin 0
	GOTO next	repeat
	end	

- Each of the PORT B pins has a weak internal pull-up. A single control bit can turn on all the pull-ups. This is performed by clearing bit RBPU (INTCON2 register).

- The weak pull-up is automatically turned off when the port pin is configured as an output. The pull-ups are disabled on a Power-on Reset.
- Four of the PORT B pins i.e. RB7 to RB4 have an interrupt-on-change feature. Only those pins that are configured as inputs can cause this interrupt to occur while if a pin out of RB7 to RB4 is configured as an output pin, then that pin is excluded from the interrupt-on-change comparison.
- The input pins of RB7 to RB4 are compared with the old value latched on the last read of PORT B. The "mismatch" outputs of RB7:RB4 are ORed together to generate the RB Port Change Interrupt with Flag bit RBIF (INTCON register).
- This interrupt can wake the device from Sleep. The user, in the Interrupt Service Routine, can clear the interrupt in the following manner :
 - (a) Any read or write of PORT B (except with the MOVFF instruction). This will end the mismatch condition.
 - (b) Clear flag bit RBIF
- A mismatch condition will continue to set flag bit RBIE. Reading PORT B will end them is match condition and allow flag bit RBIF to be cleared.
- The interrupt-on-change feature is recommended for wake-up on key depression operation and operations where PORT B is only used for the interrupt-on-change feature. Polling of PORT B is not recommended while using the interrupt-on-change feature.
- The reading and writing of a port pin of port B is similar to that of port A as seen in Section 4.2.

4.4 Port C

University Question

Q. Explain port C of PIC18F458.

SPPU - Dec. 14, 2 Marks

- PORT C is an 8-bit wide, bidirectional port. The corresponding data direction register is TRIS C. If a particular bit of TRISC is set to 1, it will make the corresponding PORT C pin an input pin while if a particular bit of TRIS C is cleared to 0, it will make the corresponding PORTC pin an output pin.
- Reading the PORT C register reads the status of the pins, whereas writing to it will write to the port latch. Read-modify-write operations read from the LATC register and write to the latch of PORTC.



- PORT C is multiplexed with several peripheral functions listed in Table 4.4.1. PORT C pins have Schmitt Trigger input buffers.

Table 4.4.1 : PORT C pins have Schmitt Trigger input buffers

Name	Function
RC0/T1OS0/T1CKI	Input / output port pin, timer 1 oscillator output or timer 1 / timer 3 clock input.
RC1/T1OS1	Input / output port pin or timer 1 oscillator input.
RC2/CCP1	Input / output port pin or capture 1 input / compare 1 output / PWM1 output.
RC3/SCK/SCL	Input / output port pin or synchronous serial clock for SPI™/I ² C™.
RC4/SDI/SDA	Input / output port pin or SPI data in (SPI mode) or data I/O (I ² C mode).
RC5/SDO	Input / output port pin or synchronous serial port data output.
RC6/TX/CK	Input / output port pin, addressable USART asynchronous transmit or addressable USART synchronous clock.
RC7/RX/DT	Input / output port pin, addressable USART a synchronous receive or addressable USART synchronous data.

- When enabling peripheral functions, care should be taken in defining TRIS bits for each PORT C pin. Some peripheral override the TRIS bit to make a pin an output, while other peripherals override the TRIS bit to make a pin an input.
- The pin override value is not loaded into the TRIS register. This allows read-modify-write of the TRIS register, without concern due to peripheral overrides.

Program 4.4.1 : Initializing PORT C

Soln.:

```
CLRF PORTC ; Initialize PORT C by
              ; clearing output
              ; data latches
CLRF LATC ; Alternate method
              ; to clear output
              ; data latches
MOVLW 0CFh ; Value used to
              ; initialize data
              ; direction
MOVWF TRISC ; Set RC3 : RC0 as inputs
              ; RC5 : RC4 as outputs
              ; RC7 : RC6 as inputs
```

The reading and writing of a port pin of port C is similar to that of port A as seen in Section 4.2.

4.5 Port D

University Question

Q. Explain port D of PIC18F458.

SPPU - Dec. 14, 2 Marks

- PORT D is an 8-bit wide, bidirectional port. The corresponding data direction register for the port is TRIS D. If a particular bit of TRIS D is set to 1, it will make the corresponding PORTD pin an input pin while if a particular bit of TRIS D is cleared to 0, it will make the corresponding PORTD pin an output pin. Reading the PORTD register reads the status of the pins, whereas writing to it will write to the port latch. Read-modify-write operations read from the LATD register and write to the latch of PORTD.
- PORTD uses Schmitt Trigger input buffers. Each pin is individually configurable as an input or output.
- PORTD can be configured as an 8-bitwide, microprocessor port (Parallel Slave Port or PSP) by setting the control bit PSP MODE (TRIS E<4>). In this mode, the input buffers are TTL.
- PORTD is also multiplexed with the analog comparator module and the ECCP module.

**Program 4.5.1 : Initializing PORT D****Soln.:**

```

CLRF PORTD ; Initialize PORT D by
             ; clearing output
             ; data latches
CLRF LATD  ; Alternate method
             ; to clear output
             ; data latches
MOVLW 07h  ; comparator off MOVWF CMCON
MOVLW 0CFh  ; Value used to
             ; initialize data
             ; direction
MOVWF TRISD ; Set RD3 : RD0 as inputs
             ; RD5 : RD4 as outputs
             ; RD7 : RD6 as input

```

- The reading and writing of a port pin of port D is similar to that of port A as seen in Section 4.2.

4.6 Port E**University Question**

Q. Explain port E of PIC18F458.

SPPU - Dec. 14, 2 Marks

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.

- The corresponding data direction register for the port is TRIS E. If a particular bit of TRIS E is set to 1, it will make the corresponding PORT E pin an input pin while if a particular bit of TRIS E is cleared to 0, it will make the corresponding PORT E pin an output pin. Reading the PORT E register reads the status of the pins, whereas writing to it will write to the port latch. Read-modify-write operations read from the LATE register and write to the latch of PORTE.
- The TRIS E register also controls the operation of the Parallel Slave Port through the control bits in the upper half of the register.
- PORT E pins are real so multiplexed with inputs for the A/D converter and outputs for the analog comparators. When selected as an analog input, these pins will read as '0's.

- Direction bits TRIS E <2:0> control the direction of the RE pins, even when they are being used as analog inputs. The user must make sure to keep the pins configured as inputs when using them as analog inputs.

Program 4.6.1 : Initializing PORT E**Soln.:**

```

CLRF PORTE ; Initialize PORT E by
             ; clearing output
             ; data latches
CLRF LATE  ; Alternate method
             ; to clear output
             ; data latches
MOVLW 03h  ; Value used to
             ; initialize data
             ; direction
MOVWF TRISE ; Set RE1 : RE0 as inputs
             ; RE2 as an output
             ; (RE4 = 0- PSP MODE OFF)

```

- The reading and writing of a port pin of port E is similar to that of port A as seen in Section 4.2.

4.7 Simple Programs for I/O Ports**Program 4.7.1 : Write a C program to generate a square wave on all pins of Port A.****Soln.:**

```

#include <P18F458.h>
void main(void)
{
    TRISA = 0X0000 ; //Port A pins initialized as
                     //output port
    while(1)
    {
        PORTA=0x00 ; //Make all pins of port A as '0'
        PORTA=0xFF ; //Make all pins of port A as '1'
    }
}

```

Program 4.7.2 : Write a C program to toggle all pins of Port C.



Soln.:

```
#include <P18F458.h>
void main(void)
{
    TRISC = 0X0000 ; //Port C pins initialized as
                     //output port
    while(1)
    {
        PORTC=0x00; //Make all pins of port C as '0'
        PORTC=0xFF; //Make all pins of port C as '1'
    }
}
```

Program 4.7.3 : Write a C program to toggle all pins of Port A for 100 times

Soln.:

```
#include <P18F458.h>
void main(void)
{
    int i;
    TRISA = 0X0000 ; //Port A pins initialised as
                     //output port
    for(i=1;i<=100;i++)
    {
        PORTA=0x00; //Make all pins of port A as '0'
        PORTA=0xFF; //Make all pins of port A as '1'
    }
    while(1)
    {
    }
}
```

Program 4.7.4 : 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.

Soln.:

```
#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 4.7.5 : 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 ;
    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++) j
}
```

Program 4.7.6: Write a program in C language to get a byte from port B and send it to port C.

SPPU - Oct. 16, 6 Marks.

Soln.:

```
#include<P18F4580.h>
void main(void)
{
    TRISB=0xFF; //Set port B as input
    TRISC=0x00; //Set port C as output port
    while(1)
    {
        PORTC=PORTB;
    }
}
```

Program 4.7.7: Write a program in C to configure Port B as input port and the most significant 4 bits of Port D as input bits and the least significant 4 bits of the same port as output bits.

SPPU - May 16, 4 Marks.

Soln.:

```
#include<P18F4580.h>
void main(void)
{
    TRISB=0xFF; //Set port B as input
    TRISD=0xF0; //Set port D lower 4 pins as
    output port
                                // and upper 4 pins as input pins
    while(1);
}
```

4.8 Exam Pack (Review and University Questions)

- Q. Explain different I/O ports and associated SFRs of PIC18F458. (Refer Section 4.1) (Dec. 14, 7 Marks)
- Q. Explain I/O port pins and multiplexed function of port A. (Refer Section 4.2) (Aug. 14 (In Sem.), 4 Marks)
- Q. Explain port A of PIC18F458. (Refer Section 4.2) (Dec. 14, 2 Marks)
- Q. Explain port B of PIC18F458. (Refer Section 4.3) (Dec. 14, 2 Marks)
- Q. Mention alternate function of Port B. (Refer Section 4.3) (May 19, 4 Marks)
- Q. Write a assembly language program to read data from pin 3 port B (RB3) and transfer it to pin 7 of port C (RC7). (Refer Program 4.3.2) (Aug. 14 (In Sem.), 6 Marks, Lab Assignment)
- Q. Write an assembly language program to copy the status of PORTB.0 to PORTD.0 without affecting the other pins of both the ports. (Refer Program 4.3.3) (Aug. 15 (In Sem.), 6 Marks)
- Q. Explain port C of PIC18F458. (Refer Section 4.4) (Dec. 14, 2 Marks)
- Q. Explain port D of PIC18F458. (Refer Section 4.5) (Dec. 14, 2 Marks)
- Q. Explain port E of PIC18F458. (Refer Section 4.6) (Dec. 14, 2 Marks)
- Q. Write a program in C language to get a byte from port B and send it to port C. (Refer Program 4.7.6) (Oct. 16, 6 Marks)
- Q. Write a program in C to configure Port B as input port and the most significant 4 bits of Port D as input bits and the least significant 4 bits of the same port as output bits. (Refer Program 4.7.7) (May 16, 4 Marks)
- Q. Write a program in assembly language to configure bits RD0 and RB0 as input bits. (Refer Program 4.3.3) (Oct. 19 (In Sem.), 6 Marks)

5

UNIT - II

Timer/Counter Programming

5.1 Timers / Counters

- The same circuit is used as timer as well as counter. As a timer the register is incremented at a regular interval. The rate of counting can be varied by different prescaling of oscillator frequency.
- As a counter the register is incremented in response to a 1 to 0 transition at its corresponding external input pin. Thus it can count the number of times an event occurs. The count here can also prescaling factor.

- Dedicated 8-bit software programmable prescaler.
- Clock source selectable to be external or internal.
- Interrupt-on-overflow from FFh to 00h in 8-bit mode and FFFFh to 0000h in 16-bit mode.
- Edge select for external clock.
- Fig. 5.3.3 shows the Timer 0 Control register (TOCON).
- Fig. 5.3.1 shows a simplified block diagram of the Timer 0 module in 8-bit mode and Fig. 5.3.2 shows a simplified block diagram of the Timer 0 module in 16-bit mode.

5.2 Prescaling of PIC Timers

University Questions

Q. Explain the use of Pre-scalar.

SPPU - Aug. 17 (In sem.), 2 Marks

Q. With reference timers explain the terms pre scalar and post scalar.

SPPU - May 18, 4 Marks

- An 8-bit counter is available as a prescaler for the Timer 0 module. The prescaler is not readable or writable.
- The PSA and TOPS2: TOPS0 bits determine the prescaler assignment and prescale ratio.
- Clearing bit PSA will assign the prescaler to the Timer 0 module. When the prescaler is assigned to the Timer 0 module, prescale values of 1:2, 1:4...1:256 are selectable.
- When assigned to the Timer 0 module, all instructions writing to the TMRO register (e.g., CLRFTMRO, MOVWF TMRO, BSFTMRO, x...etc.) will clear the prescaler count.

Note : Writing to TMRO when the prescaler is assigned to Timer 0 will clear the prescaler count but will not change the prescaler assignment.

5.3 Timer 0

The Timer 0 module has the following features:

- Software selectable as an 8-bit or 16-bit timer/counter
Readable and writable.

5.3.1 Timer 0 Block Diagram

- Timer 0 can operate as a timer or as a counter.
- Timer mode is selected by clearing the TOCS bit. In Timer mode, the Timer 0 module will increment every instruction cycle (without prescaler). If the TMROL register is written, the increment is inhibited for the following two instruction cycles. The user can work around this by writing an adjusted value to the TMROL register.
- Counter mode is selected by setting the TOCS bit. In Counter mode, Timer 0 will increment either one very rising or falling edge of pin RA4/TOCKI. The incrementing edge is determined by the Timer 0 Source Edge Select bit (TOSE). Clearing the TOSE bit selects the rising edge. When an external clock input is used for Timer 0, it must meet certain requirements. The requirements ensure the external clock can be synchronized with the internal phase clock (TOCS). Also, there is a delay in the actual incrementing of Timer 0 after synchronization.
- The Fig. 5.3.1 and 5.3.2 shows the block diagram of Timer 0 in 8-bit as well as in 16-bit mode.

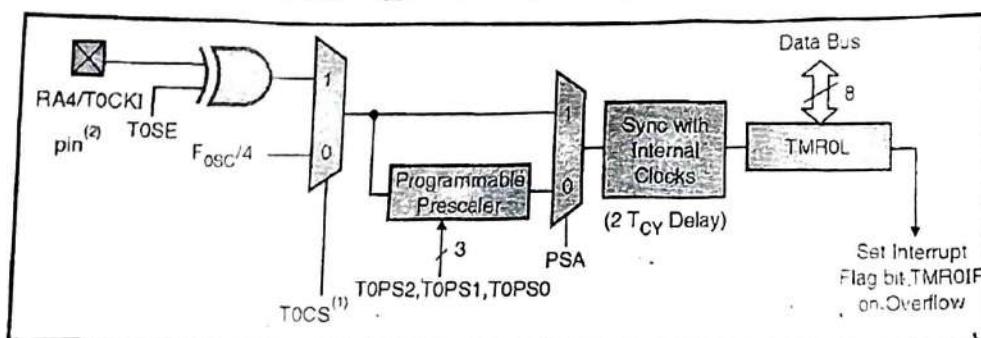


Fig. 5.3.1 : Timer 0 block diagram in 8-bit mode

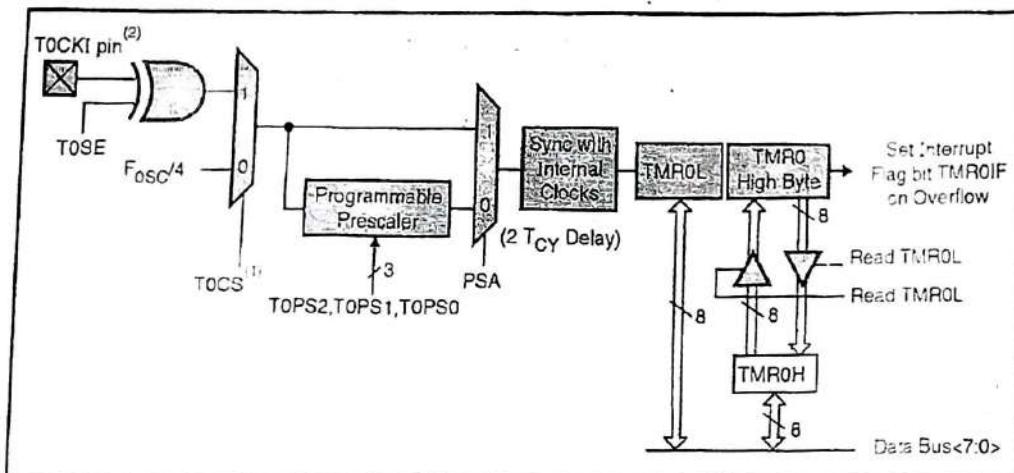


Fig. 5.3.2 :Timer 0 block diagram in 16-bit mode

16-bit mode timer reads and writes

- Timer 0 can be set in 16-bit mode by clearing the T08BIT in TOCON. Registers TMROH and TMROL are used to access the 16-bit timer value.
- TMROH is not the high byte of the timer/counter in 16-bit mode, but is actually a buffered version of the high byte of Timer 0 (refer to Fig. 5.32). The high byte of the Timer 0 timer/counter is neither directly readable nor writable. TMROH is updated with the contents of the high byte of Timer 0 during a read of TMROL. This provides the ability to read all 16bits 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.

- A write to the high byte of Timer 0 must also take place through the TMROH Buffer register. Timer 0 high byte is updated with the contents of the buffered value of TMROH when a write occurs to TMROL. This allows all 16 bits of Timer 0 to be updated at once.

Timer 0 interrupt

- The TMRO interrupt is generated when the TMRO register overflow from FFh to 00h in 8-bit mode or FFFFh to 0000h in 16-bit mode. This overflow sets the TMROIF bit in the Interrupt control (INTCON) register. The interrupt can be masked by clearing the TMROIE bit. The TMROIF bit must be cleared in software by the Timer 0 module Interrupt Service Routine before re-enabling this interrupt. The TMRO interrupt cannot awaken the processor from Sleep since the timer is shut-off during sleep.

5.3.2 Timer 0 Registers

University Questions:

- Q. Explain TOCON register. SPPU - Aug. 15 (In Sem.), Oct. 16 (In Sem.), 4 Marks
 Q. Explain Timer 0 (TOCON) control register in detail. SPPU - Dec. 16, 6 Marks
 Q. Draw the TOCON register. SPPU - Aug. 17 (In sem.), 4 Marks
 Q. Draw TOCON register and explain function of individual bits of TOCON register. SPPU - May 18, Dec. 18, 6 Marks

The Table 5.3.1 shows the registers associated with the Timer 0.

Table 5.3.1: Registers associated with Timer 0.

Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Value on POR,BOR	Value on all other Resets
Timer 0 Module Low Byte Register										
Timer 0 Module High Byte Register										
INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	INTOIF	RBIF	0000000x	0000000u
TOCON	TMROON	T08BIT	TOCS	TOSE	PSA	TOPS2	TOPS1	TOPSO	11111111	11111111
TRISA					PORTA Data Direction Register ⁽¹⁾				-11111111	-11111111

Legend: x= unknown, u=unchanged, -=unimplemented locations read as '0'.

Shaded cells are not used by Timer 0.

Note1: Bit 6 of PORTA, LATA and TRISA is enabled in ECIO and RCIO oscillator modes only. In all other oscillator modes, it is disabled and reads as '0'.

- The TOCON register is a readable and writable register that controls all the aspects of Timer 0, including the prescale selection.

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
TMROON	T08BIT	TOCS	TOSE	PSA	TOPS2	TOPS1	TOPSO
bit 7							bit 0

Fig. 5.3.3 : TOCON : Timer 0 control register

bit 7 [TMROON] : Timer 0 On/Off Control bit 1=Enables Timer 0 0=Stops Timer 0	bit 3 [PSA] : Timer 0 Prescaler Assignment bit 1=Timer 0 prescaler is not assigned. Timer 0 clock input by passes prescaler. 0=Timer 0 prescaler is assigned. Timer 0 clock input comes from prescaler output.
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 an 16-bit timer/counter	bit 2-0 [TOPS2:TOPSO] : Timer 0 Prescaler Select bits 111=1:256 Prescale value 110=1:128 Prescale value 101=1:64 Prescale value 100=1:32 Prescale value 011=1:16 Prescale value 010=1:8 Prescale value 001=1:4 Prescale value 000=1:2 Prescale value
bit 5 [TOCS] : Timer 0 Clock Source Select bit 1=Transition on TOCKI pin 0=Internal instruction cycle clock (CLK0)	
bit 4 [TOSE] : Timer 0 Source Edge Select bit 1=Increment on high-to-low transition on TOCKI pin 0=Increment on low-to-high transition on TOCKI pin	

Legend

R = Readable bit

W = Writable bit; **U** = Unimplemented bit, read as '0'

-n = Value at POR 'T' = Bit is set '0' = Bit is clear edx = Bit is unknown

5.4 Timer 1

University Question

Q. Explain timer 1 and its applications of PIC18XX in detail **SPPU - Dec. 14, May 15, 2 Marks**

SPPU - Dec. 14, May 15; 2 Marks

The Timer 1 module timer/counter has the following features:

- 16-bit timer/counter (two 8-bit registers:TMR1H and TMR1L)
 - Readable and writable (both registers)
 - Internal or external clock select
 - Interrupt-on-overflow fromFFFFh to 0000h
 - Reset from CCP module special event trigger
 - Fig. 5.4.3 shows the Timer 1 Control register.
 - This register controls the operating mode of the Timer 1 module, as well as contains the Timer 1 oscillator Enable bit (T1OSCEN). Timer 1 can be enabled/disabled by setting/clearing control bit, TMR1ON (T1CONregister).

Note : Timer 1 is disabled on POR.

5.4.1 Timer 1 Block Diagram

Timer 1 Operation

Timer 1 can operate in one of these modes:

- As a timer
 - As a synchronous counter
 - As an a synchronous counter
 - The operating mode is determined by the clock select bit, TMR1CS (T1CON register).
 - When TMR1CS is clear, Timer 1 increments every instruction cycle. When TMR1CS is set, Timer 1 increments on every rising edge of the external clock input or the Timer1 oscillator, if enabled.
 - When the Timer 1 oscillator is enabled (T1OSCEN is set), the RC1/T1OSI and RC0/T1OSO/T1CKI pins become inputs. That is, the TRISC<1:0> value is ignored.
 - Timer 1 also has an internal "Reset input". This Reset can be generated by the CCP module.
 - Fig. 5.4.1 is a simplified block diagram of the Timer 1 module.

Timer 1 Oscillator

- A crystal oscillator circuit is built in between pins T1OSI (input) and T1OSO (amplifier output). It is enabled by setting control bit T1OSCEN (T1CON register). The oscillator is a low-power oscillator rated upto 50 kHz. It will continue to run during Sleep. It is primarily intended for a 32 kHz crystal. Table 5.4.1 shows the capacitor selection for the Timer 1 oscillator.
 - The user must provide a software time delay to ensure proper start-up of the Timer 1 oscillator.

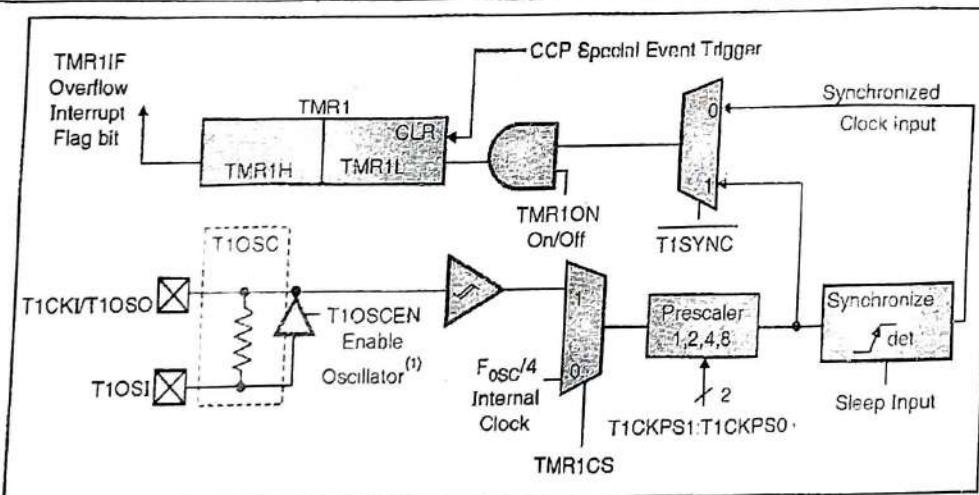


Fig. 5.4.1 : Timer 1 block diagram

Table 5.4.1 : Capacitor selection for the alternate oscillator

Osc. Type	Frequency	C1	C2
LP	32 kHz	TBD ⁽¹⁾	TBD ⁽¹⁾
Crystal to be Tested:			
32.768 kHz	Epson C-001R32.768K-A	±20 PPM	

- Note 1:** Microchip suggests 33 pF as a starting point invalidating the oscillator circuit.
2. Higher capacitance increases the stability of the oscillator, but also increases the start-up time.
 3. Since each resonator/crystal has its own characteristics, the user should consult the resonator/crystal manufacturer for appropriate values of external components.
 4. Capacitor values are for design guidance only.

Timer 1 Interrupt

- The TMR1 register pair (TMR1H:TMR1L) increments from 0000h to FFFFh and rolls over to 0000h. The TMR1 interrupt, if enabled, is generated on overflow which is latched in interrupt flag bit, TMR1IF (PIR registers). This interrupt can be enabled/disabled by setting/clearing TMR1 Interrupt Enable bit, TMR1IE (PIE registers).

Resetting Timer 1 Using a CCP Trigger Output

- If the CCP module is configured in compare mode to generate a "special event trigger" (CCP1M3:CCP1M0=1011), this signal will reset Timer 1 and start an A/D conversion (if the A/D module is enabled).

Note : The special event triggers from the CCP1 module will not set interrupt flag bit, TMR1IF(PIR registers).

- Timer 1 must be configured for either Timer or Synchronized Counter mode to take advantage of this feature. If Timer 1 is running in Asynchronous Counter mode, this Reset operation may not work.
- In the event that a write to Timer 1 coincides with a special event trigger from CCP1, the write will take precedence.
- In this mode of operation, the CCP1H:CCP1L register pair effectively becomes the period register for Timer 1.

Timer 1, 16-Bit Read/Write Mode

- Timer 1 can be configured for 16-bit reads and writes (Refer Fig. 5.4.2). When the RD16 control bit (T1CON register) is set, the address for TMR1H is mapped to a buffer register for the high byte of Timer 1.
- A read from TMR1L will load the contents of the high byte of Timer 1 into the Timer 1 High Byte Buffer register. This provides the user with the ability to accurately read all 16 bits of Timer 1 without having to determine whether a read of the high byte, followed by a read of the low byte, is valid due to a rollover between reads.
- A write to the high byte of Timer 1 must also take place through the TMR1H Buffer register. Timer 1 high byte is updated with the contents of TMR1H when a write occurs to TMR1L. This allows a user to write all 16 bits to both the high and low bytes of Timer 1 at once.
- The high byte of Timer 1 is not directly readable or writable in this mode. All reads and writes must take place through the Timer 1 High Byte Buffer register. Writes to TMR1H do not clear the Timer 1 prescaler. The prescaler is only cleared on writes to TMR1L.

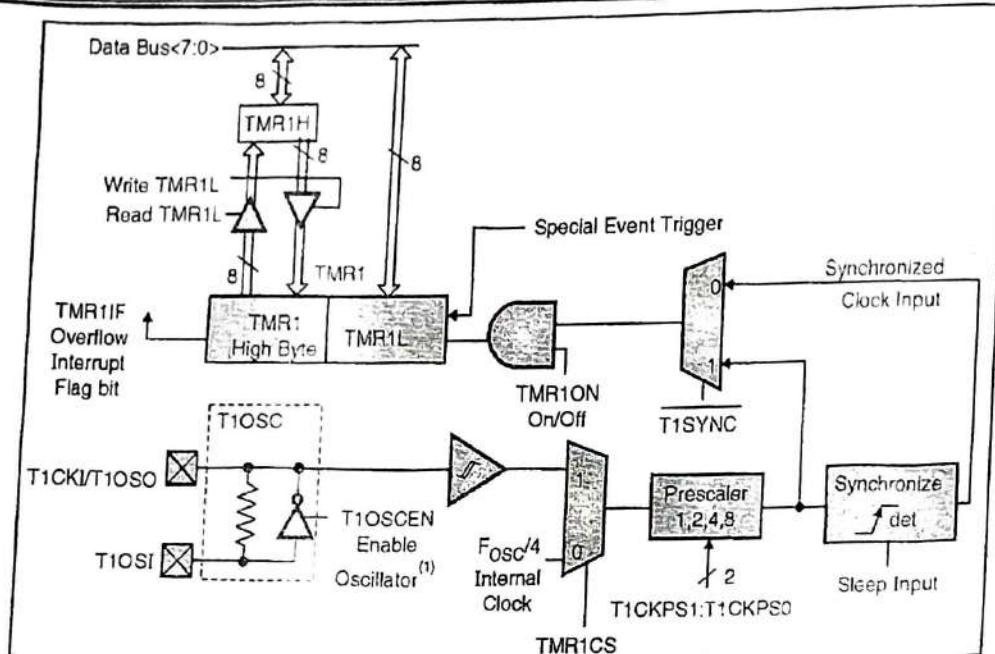


Fig. 5.4.2 : Timer 1 block diagram: 16-bit read/write mode

5.4.2 Timer 1 Registers

Table 5.4.2 : Registers associated with Timer 1 as a timer/counter

Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on all other Resets
INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	INTOF	RBIF	00000000x	00000000u
PIR1	PSP1IF ⁽¹⁾	ADIP	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	00000000	00000000
PIE1	PSP1IF ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	00000000	00000000
IPR1	PSP1IP ⁽¹⁾	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	11111111	11111111
TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxxxxx	uuuuuuuu
TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxxxxx	uuuuuuuu
T1CON	RD16	—	T1CKPS1	T1CKPS0	T1OSCEN	—	TMR1CS	TMR1ON	0-000000	u-uuuuuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer 1 module.

Note: These registers or register bits are not implemented on the PIC18F248 and PIC18F258 and read as '0's.

5.4.3 Timer 1 Control Register (T1CON)

R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	—	T1CKPS1	T1CKPS0	T1OSCEN	—	TMR1CS	TMR1ON

bit 7

bit 0

Fig. 5.4.3 : T1CON: Timer 1 control register

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	[Unimplemented] : Read as '0'
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 shut-off The oscillator inverter and feedback resistor are turned off to eliminate power drain.
bit 2	[T1SYNC] : Timer 1 External Clock Input Synchronization Select bit When TMR1CS= 1: 1=Do not synchronize external clock input 0=Synchronize external clock input When TMR1CS= 0: This bit is ignored. Timer 1 uses the internal clock when 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 ($F_{osc}/4$)
bit 0	[TMR1ON] : Timer 1 On bit 1=Enables Timer 1 0=Stops Timer 1

Legend:

R = Readable bit W = Writable bit, U = Unimplemented bit, read as '0'
 - n = Value at POR, '1' = Bit is set '0' = Bit is cleared, x = Bit is unknown

5.5 Timer 2

The Timer 2 module timer has the following features :

- 8-bit timer(TMR2 register)
- 8-bit period register (PR2)
- Readable and writable (both registers)
- Software programmable prescaler (1:1,1:4, 1:16)
- Software programmable post scaler (1:1 to 1:16)

- Interrupt on TMR2 match of PR2
- SSP module optional use of TMR2 output to generate clock shift
- Fig. 5.5.2 shows the Timer 2 Control register. Timer2 can be shut-off by clearing control bit TMR2ON (T2CON register) to minimize power consumption. The prescaler and postscaler selection of Timer 2 are controlled by this register.

5.5.1 Timer 2 Block Diagram

1. Timer 2 Operation

Timer2 can be used as the PWM time base for the PWM mode of the CCP module. The TMR2 register is readable and writable and is cleared on any device Reset. The input clock($F_{osc}/4$) has a prescale option of 1:1,1:4 or 1:16, selected by control bits T2CKPS1:T2CKPS0 (T2CON register). The match output of TMR2 goes through a 4-bit post scaler (which gives a 1:1 to 1:16 scaling inclusive) to generate a TMR2 interrupt (latched in flag bit TMR2IE, PIR registers).

The prescaler and postscaler counters are cleared when any of the following occurs :

- A write to the TMR2 register
- A write to the T2CON register
- Any device Reset (Power-on Reset, MCLR Reset, Watch dog Timer Reset or Brown-out Reset)
- TMR2 is not cleared when T2CON is written.

Fig. 5.5.1 is a simplified block diagram of the Timer 2 module.

Note:- Timer 2 is disabled on POR.

2. Timer 2 Interrupt

The Timer2 module has an 8-bit period register, PR2. A Timer2 increment from 00h until it matches PR2 and then resets to 00h on the next increment cycle. PR2 is a readable and writable register. The PR2 register is initialized to FFh upon Reset.

3. Output of TMR2

The output of TMR2 (before the postscaler) is a clock input to the Synchronous Serial Port module which optionally uses it to generate the shift clock.

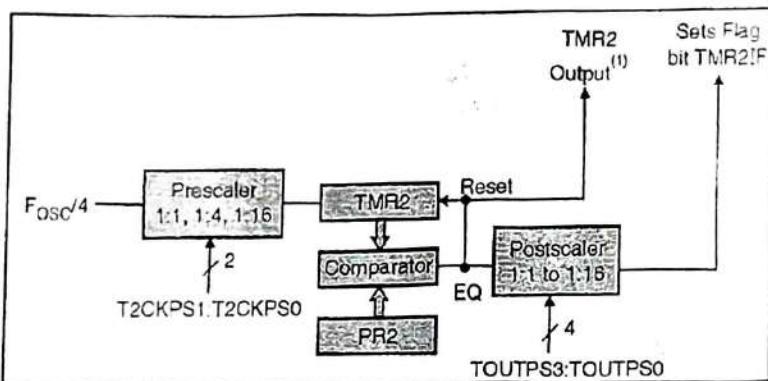


Fig. 5.5.1 : Timer 2 block diagram

5.5.2 Timer2 Registers and TMR2IF Flag

Table 5.5.1 : Registers associated with timer 2 as a timer/counter

Name	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Value on POR.BOR	Value on all other Resets
INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	[INT0IF]	RBIF	0000000x	0000000u
PIR1	PSPIF ⁽¹⁾	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	00000000	00000000
PIE1	PSPIE ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	00000000	00000000
IPR1	PSPIP ⁽¹⁾	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	11111111	11111111
TMR2	Timer2 Module Register								00000000	00000000
T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	0000000	-0000000
PR2	Timer2PeriodRegister								11111111	11111111

Legend : x=unknown, u=unchanged, -=unimplemented, read as '0'. Shaded cells are not used by the Timer 2 module.

Note : These registers or register bits are not implemented on the PIC18F248 and PIC18F258 and read as '0's.

5.5.3 Timer 2 Control Register (T2CON)

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	

bit 7

bit 0

Fig. 5.5.2:T2CON:Timer2controlregister

bit7 [Unimplemented] : Read as '0'

bit6-3 [TOUTPS3:TOUTPS0] : Timer 2 Output Postscale Select bits

0000=1:1 Postscale

0001=1:2 Postscale

1111=1:16 Postscale bit 2

TMR2ON : Timer 2 On bit

1=Timer 2 is on

0=Timer2 is off

bit 1-0 [T2CKPS1:T2CKPS0] : Timer2 Clock Prescale Select bits

00=Prescaler is 1

01=Prescaler is 4

1x=Prescaler is 16

Legend :

R=Readable bit

W=Writable bit

U=Unimplemented bit, reads '0'

'n'=Value at POR

'1'=Bit is set

'0'=Bit is cleared

'x'=Bit is unknown

5.6 Timer 3

The Timer 3 module timer/counter has the following features :

- 16-bit timer/counter(two 8-bit registers:TMR3H and TMR3L)
- Readable and writable (both registers)
- Internal or external clock select
- Interrupt-on-overflow from FFFFh to 0000h
- Reset from CCP1/ECCP1module trigger

Note:- Timer 3 is disabled on POR

5.6.1 Timer 3 Block Diagram

Timer 1 Oscillator

The Timer 1 oscillator may be used as the clock source for Timer 3. The Timer 1 oscillator is enabled by setting the T1OSCEN bit (T1CON register). The oscillator is a low-power oscillator rated upto 50 kHz. Refer to Section 5.4 "Timer 1 Module" for Timer 1 oscillator details.

Timer 3 Interrupt

The TMR3 register pair (TMR3H:TMR3L) increments from 0000h to OFFFFh and rolls over to 0000h. The TMR3 interrupt, if enabled, is generated on overflow which is latched in interrupt flag bit TMR3IF (PIR registers). This interrupt can be enabled/disabled by setting/ clearing TMR3 Interrupt Enable bit, TMR3IE (PIE registers).

Resetting Timer 3 Using a CCP Trigger Output

If the CCP module is configured in compare mode to generate a "special event trigger" (CCP1M3:CCP1M0=1011), this signal will reset Timer 3.

Note:- The special event triggers from the CCP module will not set interrupt flag bit TMR3IF (PIR registers).

- Timer 3 must be configured for either Timer or Synchronized Counter mode to take advantage of this feature. If Timer 3 is running in Asynchronous Counter mode, this Reset operation may not work. In the event that a write to Timer 3 coincides with a special event trigger from CCP1, the write will take precedence. In this mode of operation, the CCP1H:CCP1L register pair becomes the period register for Timer 3.
- Fig. 5.6.1 is a simplified block diagram of the Timer 3 module.

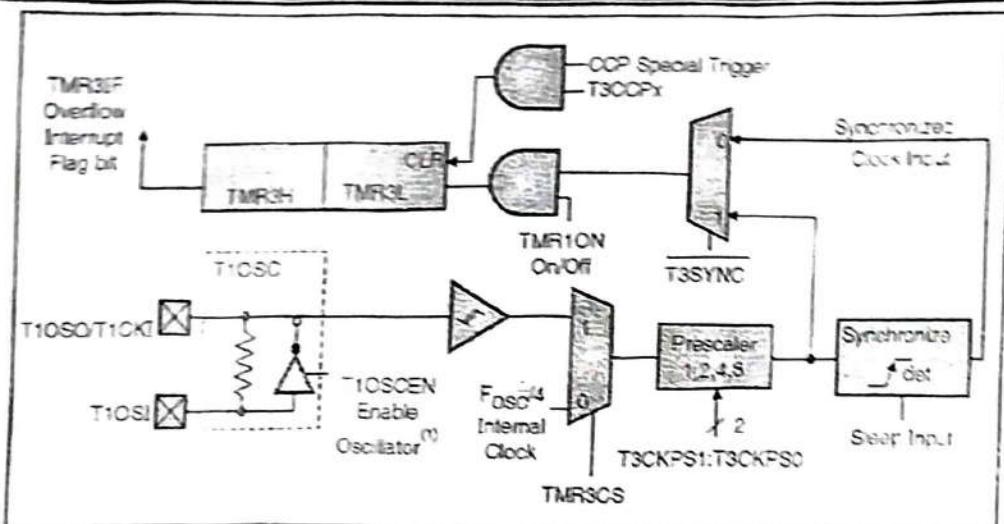


Fig. 5.6.1 : Timer 3 block diagram

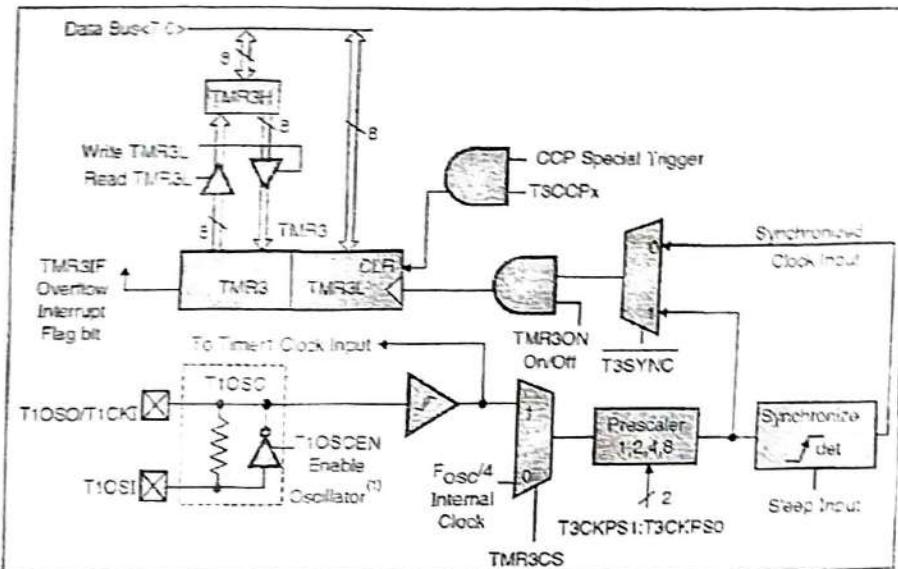


Fig. 5.6.2 : Timer 3 block diagram configured in 16-bit read/write mode

5.6.2 Timer 3 Control Register (T3CON)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
RD16	T3ECCP1	T3CKPS1	T3CKPS0	T3CCP1	-----	T3SYNC	TMR3CS

bit 7

bit 0

Fig. 5.6.3: T3CON: Timer 3 control register

bit7 [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 [T3EOCP1:T3CCP1]: Timer 3 and Timer 1 to CCP1/ECCP1 Enable bits

1x=Timer 3 is the clock source for compare/capture CCP1 and ECCP1 modules

01=Timer 3 is the clock source for compare/capture of ECCP1, Timer 1 is the clock source for compare/capture of CCP1

00=Timer 1 is the clock source for compare/capture CCP1 and ECCP1 modules

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 [TSSYNC]: 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 or T1CKI (on the rising edge after the first falling edge)

0=Internal clock($F_{osc}/4$)

bit 0 [TMR3ON]: Timer 3 On bit

1=Enables Timer 3

0=Stops Timer 3

Legend :

R=Readable bit

W=Writable bit

U=Unimplemented bit, reads '0'

-n=Value at POR

'1'=Bit is set

'0'=Bit is cleared

'x'=Bit is unknown

5.7 Programming the PIC18 Timers

Program 5.7.1 : Write a C18 program to toggle all the bits of PORT B continuously with some delay. Use Timer 0, 16-bit mode, and no prescaler options to generate the delay.

OR Write programming steps for generation of time delay using timer.

Soln. :

```
#include <P18F4580.h>
void T0Delay(void);
void main(void)
{
    TRISB=0x00; //PORTB output port
```

```
while(1) //repeat forever
{
    PORTB=0x55; //toggle all bits of PortB
    T0Delay(); //delay size unknown
    PORTB=0xAA; //toggle all bits of PortB
    T0Delay();
}

void T0Delay()
{
    T0CON=0x03; //Timer 0, 16-bit mode no prescaler
    TMROH=0x35; //load TH0
    TMROL=0x00; //load TL0
```



```

TOCONbits.TMROON = 1; //turn on T0
while (INTCONbits.TMROIF == 0); //wait for TFO to
                                //rollover
TOCONbits.TMROON = 0; //turn off T0
INTCONbits.TMROIF = 0; //clear TFO
}

```

Program 5.7.2 : Write a C18 program to toggle only the PORT B, 4 bit continuously every 50 mS. Use Timer 0, 16-bit mode, the 1:4 prescaler to create the delay. Assume XTAL = 10MHz.

OR Write C program to generate delay of .50msec using Timer0. Assume crystal frequency of 10MHz.

SPPU - Dec. 19, 5 Marks

Soln. :

```

#include <P18F4580.h>
void TODelay(void)
{
#define mybit PORTBbits.RB4
void main(void)
{
    TRISBbits.TRISB4 = 0;
    while(1)
    {
        mybit ^= 1; //togglePORTB.4
        TODelay(); //Timer 0, mode 1(16-bit)
    }
}
void TODelay()
{
    TOCON = 0x01; //Timer 0, 16-bit mode, 1:4 prescaler
    TMROH = 0x85; //loadTH0
    TMROL = 0xEE; //loadTL0
    TOCONbits.TMROON = 1; //turn on Timer 0
    while (INTCONbits.TMROIF == 0); //wait for TFO to
                                    //roll over
    TOCONbits.TMROON = 0; //turn off Timer 0
    INTCONbits.TMROIF = 0; //clear TFO
}

```

Program 5.7.2(A) : Assuming XTAL = 10 MHz, write a program to generate a square wave of 2 KHz on port C.5.

SPPU - Aug. 15, 6 Marks

OR

Assuming crystal frequency = 10MHz, write a program in C language to generate square wave form with a frequency of 2KHz on PORTC.5. Use timer 0 in 16bit mode without a prescalar.

Soln. :

Frequency of square wave = 2 KHz

$$\therefore \text{Time period} = \frac{1}{2 \text{ KHz}} = 500 \mu\text{s}$$

Hence, the square wave will be logic '1' for 250 μs and it will be logic '0' for 250 μs .

$$\text{XTAL} = 10 \text{ MHz}$$

$$\therefore \text{Timer clock frequency} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5 \text{ MHz}} = 0.4 \mu\text{s}$$

$$\therefore \text{Number of counts required} = \frac{250 \mu\text{s}}{0.4 \mu\text{s}} = 625$$

The values to be loaded into TMROH and TMROL are :

$$65536 - 625 = (64911)_{10} = FD8FH$$

$$\therefore \text{TMROH} = FDH$$

$$\text{TMROL} = 8FH$$

C18 program:

```

#include <P18F4581.h>
#define square PORTCbits.RC5
void delay()
{
    TOCON = 0x08; // Timer 0, 16 bit mode, no prescaler
    TMROL = 0x8F; // Load TMROL
    TMROH = 0xFD; // Load TMROH
    TOCONbits.TMROON = 1; // Start Timer 0
    while (INTCONbits.TMROIF == 0); // wait for TMROIF to overflow
    TOCONbits.TMROON = 0; // stop Timer 0
    INTCONbits.TMROIF = 0; // Clear TMROIF flag
}

void main(void)
{
    TRISCbits.TRISC5 = 0; // program RC5 as output bit
    while (1)
    {
        square = ~square; // toggle bit RC5
        delay();
    }
}

```

Program 5.7.3 : Write a C18 program to generate a frequency of 2 Hz only on pin PORT B 5. Use Timer 0, 8-bit mode to create the delay.

Soln.:

```
#include <P18F4580.h>
void TOM8Delay(void);
#define mybit PORTBbits.RB5
void main(void)
{
    unsigned char x, y;
    TRISBbits.TRISB5 = 0;
    while (1)
    {
        mybit^=1; //toggle PORTB.5
        for(x=0;x<250;x++)//due to for loop overhead
        for(y=0;y<35;y++)//we put 35 and not 39
            TOM8Delay();
    }
}
void TOM8Delay()
{
    TOCON=0x45; //Timer 0, 16-bits mode,
    //prescaler//1:64
    TMROL=-1; //Load TLO
    TOCONbits.TMROON=1; //Turn on T0
    while(INTCONbits.TMROIF==0);
    //Wait for TFO to roll over
    TOCONbits.TMROON=0; //Turn off T0
    INTCONbits.TMROIF=0; //Clear TFO
}
```

Program 5.7.4 : Write a C18 program to generate a frequency of 250 Hz on all bits of PORT C. Use Timer 0, 16-bit mode, and no prescaler to create the frequency. Assume XTAL = 10 MHz.

Soln.:

```
#include <P18F4580.h>
void TODelay(void);
void main(void)
{
    unsigned char x;
    TRISC=0; //PORTC output port
```

```
PORTC=0x55;
while (1)
{
    PORTC=~PORTC;//toggle all bits of port C
    for (x = 0;x<20; x++)
        TODelay();
}
void TODelay()
{
    TOCON=0x0; //Timer 0, 16 bit mode, no prescaler
    TMROH=0xFF; //load TH0
    TMROL=0x06; //load TL0
    TOCONbits.TMROON=1; //turn on T0
    while (INTCONbits.TMROIF==0);
    //wait for TFO to roll over
    TOCONbits.TMROON=0; //turn off T0
    INTCONbits.TMROIF=0; //clear TFO
}
```

Program 5.7.5 : A switch is connected to pin PORTB.7. Write a C18 program to monitor SW and create the following frequencies on pin PORTB.0:

SW = 0 : 500 Hz

SW = 1 : 750 Hz

Use Timer 0 with prescaler for both of them.

Soln. :

```
#include <P18F4580.h>
#define mybit PORTBbits.RB0
#define SW PORTBbits.RB7
void TOPSDelay(unsigned char);
void main(void)
{
    TRISBbits.TRISB7=1; //make PB.7 an input
    TRISBbits.TRISB0=0; //make PB.0 an output
    SW=1;
    while (1)
    {
        mybit^=1; //toggle PB.0
        if(SW==0) //check switch
```



```

T0PSDelay(0),
else
T0PSDelay(1),
}

void T0PSDelay(unsigned char c)
{
    TOCON = 0x05;           //Timer 0, 16-bits mode,
                           //prescaler1 : 64
    if(c == 0)
    {
        TMROH=0xFF;         //load TH0
        TMROL=0xD9;         //load TL0
    }
    else
    {
        TMROH=0xFF;         //load TH0
        TMROL=0xE6;         //load TL0
    }
    TOCONbits.TMR0ON=1;      //turn on T0
    while(INTCONbits.TMROIF==0);
    //wait for TF0 to roll over
    TOCONbits.TMR0ON=0;      //turn off T0
    INTCONbits.TMROIF=0;     //clear TF0
}

```

Program 5.7.6 : Write a C18 program to create a frequency of 2500 Hz on pin PORTB.1. Use Timer 1 to create the delay.

Soln.:

```

#include <P18F4580.h>
void T1Delay(void);
#define mybit PORTBbits.RB1
void main(void)
{
    TRISBbits.TRISB1=0;
    while(1)
    {
        mybit^=1;           //toggle PB.1
        T1Delay();
    }
}
void T1Delay()

```

```

T1CON=0x00;           //Timer 1, 16-bit mode, no
                      //prescaler
TMR1H=0xFE;           //load TH1
TMR1L=0x0C;           //load TL1
T1CONbits.TMR1ON=1;   //turn on T1
while(PIR1bits.TMRLIF==0);
//wait for TF1 to roll over
T1CONbits.TMR1ON=0;   //turn off T1
PIR1bits.TMRLIF=0;    //clear TF1
}

```

Program 5.7.7 : Assume that a 1-Hz external clock is being fed into pin TOCKI (RA4). Write a C18 program for counter 0 in 8-bit mode to count up and display the state of the TMROL count on PORTB. Start the count at 0H.

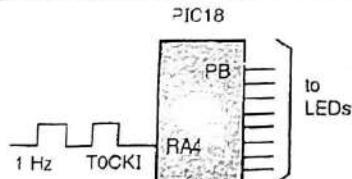
Soln.:

```

#include <P18F4580.h>
void main (void)
{
    TRISAbits.TRISA4=1; //make RA4/TOCKI an input
    TRISB=0;
    TOCON=0x63; //counter 0, 8-bit mode, no prescaler
    TMROL=0;           //set count to 0
    while(1)           //repeat forever
    {
        do
        {
            TOCONbits.TMR0ON=1; //turn on T0
            PORTB=TMROL;     //place value on pins
        }
        while(INTCONbits.TMROIF==0);
        //wait for TF0 to roll over
        TOCONbits.TMR0ON=0; //turn off T0
        INTCONbits.TMROIF=0; //clear TF0
    }
}

```

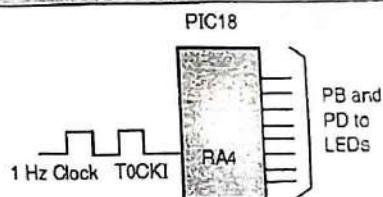
PORTB is connected to 8 LEDs. TOCKI (RA4) is connected to a 1-Hz external clocks.



Program 5.7.8 : Assume that a 1-Hz external clock is being fed into pin TOCKI (RA4). Write a C program for counter 0 in mode 1 (16-bits) to count the pulses and display the TMROL and TMR0H registers on PORT D and PORT B, respectively.

Soln.:

```
#include <P18F4580.h>
void main (void)
{
    TRISAbits.TRISA4=1; //make RA4 an input for T0CKI
    TRISB=0;           //PORTB output port
    TRISD=0;           //PORTD output port
    TOCON=0x25;        //Timer 0, 16-bit mode,
                       //prescaler 1:64
    TMROH=0;          //set count to 0
    TMROL=0;          //set count to 0
    while (1)          //repeat forever
    {
        do
        {
            TOCONbits.TMROON=1; //turn on T0
            PORTB=TMROL;
            PORTD=TMROH;      //place value on pins
        }
        while (INTCONbits.TMROIF==0);
        //wait for rollover
        TOCONbits.TMROON=0; //turn off T0
        INTCONbits.TMROIF=0; //clear TF0
    }
}
```



Program 5.7.9: Assume that a 64-Hz external clock is being fed into pin T0CKI (RA4). Write a C program for counter 0 in 8-bit mode to display the count in ASCII. The 8-bit binary count must be converted to ASCII. Display the ASCII digits (in binary) on PORTB, PORTC, and PORTD, where PORTB has the least significant digit. Set the initial value of TMROL to 0.

Soln.:

To display the TMROL count, we must convert 8-bit binary data to ASCII. The ASCII values will be shown in binary. For example, '9' will show as 00111001 on the ports.

```
#include <P18F4580.h>
void BinToASCII (unsigned char);
void main()
```

```
unsigned char value;
TRISAbits.TRISA4=1; //make RA4 an input
TRISB=0;           //make PORTB an output
TRISC=0;           //make PORTC an output
TRISD=0;           //make PORTD an output
TMROL=0;
TOCON=0x65;        //counter 0, 8 bit mode, prescaler 1:64
while (1)
{
    do
    {
        TOCONbits.TMROON=1; //turn on T0
        value=TMROL;
        BinToASCII (value);
    }
    while (INTCONbits.TMROIF==0);
    //wait for TFO to roll over
    TOCONbits.TMROON=0; //turn off T0
    INTCONbits.TMROIF=0 //clear TF0
}
```

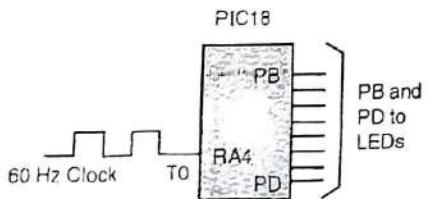
void BinToASCII (unsigned char value)

```
{
    unsigned char x, d1, d2, d3;
    x=value/10;
    d1=value%10;
    d2=x%10;
    d3=x/10;
    PORTB=0x30|d1;
    PORTC=0x30|d2;
    PORTD=0x30|d3;
}
```

Program 5.7.10 : Assume that a 60-Hz external clock is being fed into pin T0CKI(RA4). Write a C program for Counter 0 in 8-bit mode to display the seconds and minutes on PORT B and PORT D, respectively.

Soln.:

```
#include <P18F4580.h>
void ToTime(unsignedchar);
void main()
{
    unsigned char sec;
    TRISB=TRISD=0; //PORT B, D outputs
    T0CON=0x68; //Timer 0, no prescaler
    TMROL=-60; //sec=60 pulses
    while(1)
    {
        do
        {
            T0CONbits.TMR0ON=1; //turn on T0
            sec=TMROL;
            To Time(sec);
        }
        while (INTCONbits.TMR0IF==0);
        //wait for TFO to rollover
        T0CONbits.TMR0ON=0; //turnoff T0
        INTCONbits.TMR0IF=0; //clear TFO
    }
}
void ToTime(unsignedchar value)
{
    unsigned char sec, min;
    min=value/60;
    sec=value%60;
    PORTB=sec;
    PORTD=min;
}
```



Program 5.7.11: Assuming that XTAL=10 MHz, write a C18 program to turn on pin PORT B4 when TMR2 reaches value 100.

Soln.:

```
#include <P18F4580.h>
#define mybit PORT B bits.RB4
void main(void)
```

```
TRISBbits.TRISB1=0; //PORTB4asoutput
T2CON=0x0; //Timer2,noprescaler postscaler
TMR2=0x00; //TMR2=0
mybit=0; //PB.4=0
PR2=100; //load period register2
T2CONbits.TMR2ON=1; //turnonT0
while(PIR1bits.TMR2IF==0);
//wait for TMR2IF to be raised
mybit=1; //PB.4=0
T2CONbits.TMR2ON=0; //turnoffT2
PIR1bits.TMR2IF=0; //clearTFO
while(1); //stay here
}
```

Program 5.7.12 : Using the prescaler and postscaler, find the longest time delay that we can create using Timer 2. Assume that XTAL = 10MHz

Soln.:

```
#include <P18F4580.h>
#define mybit PORTBbits.RB4
void main (void)
{
    TRISBbits.TRISB4=0;
    T2CON=0x7B; //Timer2,prescaler=16,
    //postscaler = 16,
    TMR2=0x00; //TMR2=0
    while(1)
    {
        PR2=255; //load period register2
        T2CONbits.TMR2ON=1; //turnonT2
        while(PIR1bits.TMR2IF==0);
        //wait for TMR2IF to be raised
        mybit=~mybit; //toggle PORTB4
        T2CONbits.TMR2ON=0; //turnoff T2
        PIR1bits.TMR2IF=0; //clear TFO
    }
}
```

Because XTAL= 10MHz, TMR2 counts up every $0.4\mu\text{s}$. Therefore, when $\text{TMR2H} = \text{PR2} = 255$, PORT B4 will be turned on and off every 52ms because $255 \times 0.4\mu\text{s} \times 16 \times 16 = 26.112\text{mS}$.

Program 5.7.13 : Write a C18 program to create a frequency of 2500 Hz on pin PORT B.1. Use Timer 3 to create the delay.

Soln. :

```
#include <P18F4580.h>
void T3Delay(void);
#define mybit PORTBbits.RB1
void main(void)
{
    TRISBbits.TRISB1=0; //PB1 as an output
    T3CON=0x00; //Timer 3,16 bit mode, no prescaler
    while(1)
    {
        mybit=~mybit; //toggle PB.1
        T3Delay();
    }
}
void T3Delay()
{
    TMR3H=0xFE; //load TH3
    TMR3L=0x0C; //load TL3
    T3CONbits.TMR3ON=1; //turn on T3
    while(PIR2bits.TMR3IF==0),
        //wait for TF3 to roll over
    T3CONbits.TMR3ON=0; //turn off T3
    PIR2bits.TMR3IF=0; //clear TF3
}
```

Program 5.7.14 : Write a program to generate 100 msec delays using Timer 1. What are the values to be loaded in T1CON, TMR1H, TMR1L? Assume that XTAL = 8 MHz?

Soln. :

XTAL = 8 MHz

Prescaler : 1 : 4

$$\text{Timer clock frequency} = \frac{1}{16} \times 8 \text{ MHz} = 0.5 \text{ MHz}$$

$$\text{Time for one clock pulse} = \frac{1}{0.5 \text{ MHz}} = 2 \mu\text{sec}$$

Delay required = 100 msec

$$\text{Clock pulses to be counted} = \frac{100 \text{ msec}}{2 \mu\text{sec}} = 50000$$

$$\text{Count} = 65536 - 50000 = (15536)_{10} = (3CB0)_{16}$$

```
#include <P18F4580.h>
void T1Delay()
```

```
T1CON=0x20; //Timer 1, 16-bit mode & 1:4 prescaler
TMR1H=0x3C; //set higher byte to 3CH
TMR1L=0xB0; //set lower byte to BOH
T1CONbits.TMR1ON=1 //Switch on timer 1
while(INTCONbits.TMR1IF==0);
//wait for timer overflow
T1CONbits.TMR1ON=0; //Switch off the timer
INTCONbits.TMR1IF=0; //Clear timer overflow flag
}
void main (void)
{
    TRISB=0; //Set port B pins as output
    while(1)
    {
        PORTB=0x00; //Clear all pins of Port B
        T1Delay(); //Delay of 100msec
        PORTB=0xFF; //Set all port B pins
        T1Delay();
    }
}
```

Program 5.7.15 : 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.

Soln. :

XTAL = 10 MHz

Prescaler : 1 : 256

$$\text{Timer clock frequency} = \frac{1}{512} \times 10 \text{ MHz} = 0.01953125 \text{ MHz}$$

$$\text{Time for one clock pulse} = \frac{1}{0.01953125 \text{ MHz}} = 51.2 \mu\text{sec}$$

Delay required = 250 msec

$$\text{Clock pulses to be counted} = \frac{250 \text{ msec}}{51.2 \mu\text{sec}} = 4883$$

$$\text{Count} = 65536 - 4883 = (60653)_{10} = (\text{ECED})_{16}$$

```
#include <P18F4580.h>
void T0Delay()
{
    T0CON=0x07;
    //Timer 0, 16-bit mode, and 1:256 prescaler
    TMROH=0xEC; //set higher byte to ECH
    TMROL=0xED; //set lower byte to EDH
    T0CONbits.TMROON=1 //Switch on timer 0
}
```



```

while(INTCONbits.TMROIF==0);
    //wait for timer overflow
    TOCONbits.TMROON=0; //Switch off the timer
    INTCONbits.TMROIF=0; //Clear timer overflow flag
}

void main(void)
{
    TRISA=0; //Set port A pins as output
    while(1)
    {
        PORTA=0x00; //Clear all pins of Port A
        T1Delay(); //Delay of 100msec
        PORTA=0xFF; //Set all port A pins
        T0Delay();
    }
}

```

Program 5.7.16 : Write a C program to toggle all bits of Ports B, C, D continuously with a 250 ms delay. Assume XTAL = 10 MHz.

Soln. :

XTAL = 10 MHz

Prescaler : 1 : 256

$$\text{Timer clock frequency} = \frac{1}{512} \times 10 \text{ MHz} = 0.01953125 \text{ MHz}$$

$$\text{Time for one clock pulse} = \frac{1}{0.01953125 \text{ MHz}} = 51.2 \mu\text{sec}$$

Delay required = 250 msec

$$\text{Clock pulses to be counted} = \frac{250 \text{ msec}}{51.2 \mu\text{sec}} = 4883$$

$$\text{Count} = 65536 - 4883 = (60653)_{10} = (\text{ECED})_{16}$$

```

#include <P18F4580.h>
void T0Delay()
{
    TOCON=0x07; //Timer 0, 16-bit mode & 1:256
    //prescaler
    TMROH=0xEC; //set higher byte to ECH
    TMROL=0xED; //set lower byte to EDH
    TOCONbits.TMROON=1 //Switch on timer 0
    while(INTCONbits.TMROIF==0);
    //wait for timer overflow
    TOCONbits.TMROON=0; //Switch off the timer
    INTCONbits.TMROIF=0; //Clear timer overflow flag
}

```

```

void main(void)
{
    TRISB=0; //Set port B pins as output
    TRISC=0; //Set port C pins as output
    TRISD=0; //Set port D pins as output
    while(1)
    {
        PORTB=0x00; //Clear all pins of Port B
        PORTC=0x00; //Clear all pins of Port C
        PORTD=0x00; //Clear all pins of Port D
        T1Delay(); //Delay of 100msec
        PORTB=0xFF; //Set all port B pins
        PORTC=0xFF; //Set all port C pins
        PORTD=0xFF; //Set all port D pins
        T0Delay();
    }
}

```

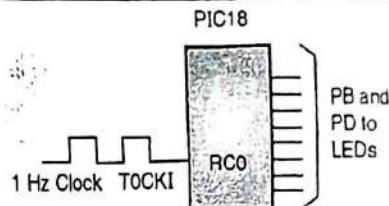
Program 5.7.17 : Assume that a 1-Hz external clock is being fed into pin T3(RC0). Write a C18 program for Timer 3 to be used as a counter. It should count the pulses and display the TMR3H and TMR3L registers on PORT D and PORT B, respectively.

Soln. :

```

#include <P18F4580.h>
void main(void)
{
    TRISCBits.TRISCO=1; //make RC0 an input for TICK1
    TRISRB=0; //make PORTB an output
    TRISD=0; //make PORTD an output
    T3CON=0x02; //Timer 3L, 16-bit mode, no prescaler
    TMR3H=0; //set count to 0
    TMR3L=0; //set count to 0
    while(1) //repeat forever
    {
        do
        {
            T3CONbits.TMR3ON=1; //turn on T3
            PORTB=TMR3L; //place value on pins
            PORTD=TMR3H;
        }
        while(PIR2bits.TMR3IF==0);
        //wait for TF3 to rollover
        T3CONbits.TMR3ON=0; //turnoff T3
        PIR2bits.TMR3IF=0; //clear TF3
    }
}

```



Program 5.7.18 : Write a C18 program to toggle all bits of Port B continuously with delay of 10 ms using Timer 0, 16 bit and no prescaler XTAL=10 MHz.

Soln. :

XTAL = 10 MHz

Prescaler : 1 : 256

$$\text{Timer clock frequency} = \frac{1}{512} \times 10 \text{ MHz} = 0.01953125 \text{ MHz}$$

$$\text{Time for one clock pulse} = \frac{1}{0.01953125 \text{ MHz}} = 51.2 \mu\text{sec}$$

Delay required = 10 msec

$$\text{Clock pulses to be counted} = \frac{250 \text{ msec}}{51.2 \mu\text{sec}} = 195$$

$$\text{Count} = 65536 - 195 = (65341)_{10} = (\text{FF3D})_{16}$$

```
#include <P18F4580.h>
void TODelay()
{
    TOCON=0x07; //Timer 0, 16-bit mode, and 1:256 prescaler
    TMROH=0xFF; //set higher byte to FFH
    TMROL=0x3D; //set lower byte to 3DH
    TOCONbits.TMROON=1 //Switch on timer 0
    while(INTCONbits.TMROIF==0); //wait for timer overflow
    TOCONbits.TMROON=0; //Switch off the timer
    INTCONbits.TMROIF=0; //Clear timer overflow flag
}
void main(void)
{
    TRISB=0; //Set port B pins as output
    while(1)
    {
        PORTB=0x00; //Clear all pins of Port B
    }
}
```

```
T1Delay(); //Delay of 100msec
PORTB=0xFF; //Set all port B pins
TODelay();
}
```

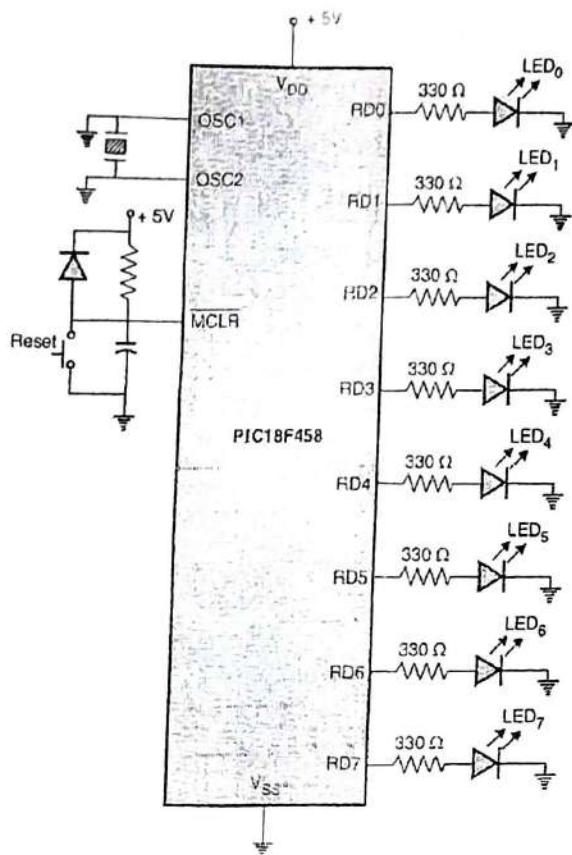
Program 5.7.19: Write a C18 program to generate square wave of 50 Hz continuously using Timer 0, 16 bit and no prescaler. XTAL = 10 MHz.

Soln. :

```
#include <P18F4580.h>
void TODelay(void);
void main(void)
{
    unsigned char x;
    TRISC=0; //PORTC output port
    PORTC=0x55;
    while(1)
    {
        PORTC=~PORTC; //toggle all bits of port C
        for(x = 0;x<20; x++)
            TODelay();
    }
}
void TODelay()
{
    TOCON=0x0; //timer 0, 16 bit mode, no
    //prescaler
    TMROH=0x55; //load TH0
    TMROL=0x02; //load TL0
    TOCONbits.TMROON=1; //turn on T0
    while (INTCONbits.TMROIF==0);
    //wait for TFO to rollover
    TOCONbits.TMROON=0; //turn off T0
    INTCONbits.TMROIF=0; //clear TFO
}
```

Program 5.7.20 : Interface LEDs to PIC 18FXX controller. Write embedded C program to flash LEDs after every 1 sec.

Soln.:



```
#include <P18F458.h>
void Delay (unsigned int x)
{
    unsigned int i;
    unsigned char j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++);
}

void main (void)
{
    TRISD = 0; //Port D is output port
    while (1)
    {
        PORTD = 1; // turn on RD0
        Delay (1000);
        PORTD = 2; //turn on RD1
        Delay (1000);
        PORTD = 4; // turn on RD2
        Delay (1000),
    }
}
```

```
PORTD = 8; //turn on RD3
Delay (1000);
PORTD = 16; // turn on RD4
Delay (1000);
PORTD = 32; // turn on RD5
Delay (1000);
PORTD = 64; //turn on RD6
Delay (1000);
PORTD = 128; // turn on RD7
Delay (1000);
```

Program 8.7.21 : Write assembly language program to toggle all the bits of port B continuously with some delay. Use timer 0 in 16 bit mode and no prescaler to generate maximum delay. XTAL = 10 MHz. **SPPU - Dec. 14, 6 Marks**

Label	Instruction	Comments
	CLRF TRISB	Make port B an output port
	MOVLW 0x08	Load TCON register for Timer 0, 16 bit mode, no prescaler
	MOVWF TCON	
repeat:	MOVLW 0x00	Load Timer 0, high byte
	MOVWF TMROH	
	MOVLW 0x00	Load Timer 0, low byte
	MOVWF TMROL	
	BCF INTCON, TMROIF	Clear TMROIF
	MOVLW 0x00	Clear all pins of port B
	MOVWF PORTB	
again:	BSF TCON, TMROON	Start Timer 0
wait:	BTFSS INTCON, TMROIF	Wait for timer to overflow
	BRA wait	
	BCF TCON, TMROON	Stop Timer 0
	MOVLW 0xFF	Set all pins of port B
	MOVWF PORTB	
	BRA again	
	GOTO repeat	

Program 5.7.22: Assuming that clock pulses are fed into pin T0CK1. Write a program for counter 0 in 8 bit mode to count the pulses and display the state of the TMROL count on PORTB.

SPPU - May 15, 6 Marks

Soln. :

Label	Instruction	Comments
	BSF TRISA, 4	Set pin 4 of port A as input pin
	CLRF TRISB	Program PORTB is output port
	MOVLW 0x68	Load T0CON for Timer 0, 8 bit, Counter, no prescaler
	MOVWF T0CON	
repeat:	MOVLW 0x00	Load TMROL to 00H
	MOVWF TMROL	
	BCF INTCON, TMROIF	Clear Timer 0 interrupt flag
	BSF T0CON, TMROON	Start Timer 0
wait:	MOVFF TMROL, PORTB	Display count on PORTB
	BTFS S INTCON, TMROIF	Check if counter overflow?
	BRA wait	
	BCF T0CON, TMROON	Stop Timer 0
	BRA repeat	

Program 5.7.23: Find the value to be loaded in T0CON for following configuration: Timer 0 in 16 bit mode, prescaler of 128 and internal clock.

SPPU - May 15, 4 Marks

Soln. : The value to be loaded in TOCON for Timer 0 in 16 bit mode, prescaler of 128 and internal clock.

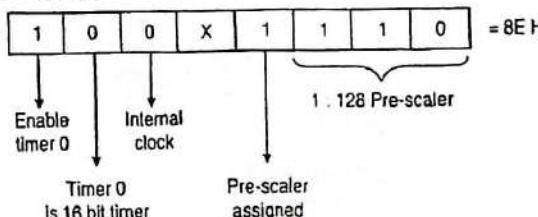


Fig. P. 5.7.17 : T0CON

Program 5.7.24: Write an assembly program using timer 1 to blink LED connected to PORTB at every 500 microsecond. Crystal frequency is 16MHz.

SPPU - Dec. 15, 7 Marks

Soln. :

$$\text{XTAL frequency} = 16 \text{ MHz}$$

$$\therefore \text{Timer clock frequency} = \frac{16 \text{ MHz}}{4} = 4 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{4 \text{ MHz}} = 0.25 \mu\text{s}$$

$$\therefore \text{Number of counts required} = \frac{500 \mu\text{s}}{0.25 \mu\text{s}} = 2000.$$

The values to be loaded into TMR1H and TMR1L are :

$$65536 - 2000 = (63536)_{10} = F830H$$

$$\therefore \text{TMR1H} = F8H$$

$$\text{TMR1L} = 30H$$

Assembly program

Label	Instruction	Comments
	CLRF TRISB	Make PORTB = output
	MOVLW 0x00	Load T1CON for Timer1, 16 bit, internal clock, no prescaler
	MOVWF T1CON	
again:	MOVLW 0xF8H	Load TMR1H with F8H
	MOVWF TMR1H	
	MOVLW 0x30H	Load TMR1L with 30H
	MOVWF TMR1L	
	BCF PIR1, TMR1IF	Clear Timer1 interrupt flag
	COMF PORTB,F	Toggle all pins of PORTB
	BSF T1CON, TMR1ON	Start Timer1
wait:	BTFS PIR1, TMR1IF	Wait for timer to overflow
	BRA wait	
	BCF PIR1, TMR1ON	Stop Timer1
	BRA again	

Program 5.7.25

Using Timer 0 in 16-bit mode, write a C-language program to obtain a time delay of 1 ms. Assume 8-MHz crystal, leading edge clock, and a prescale value of 1: 128.

SPPU - May 16, 6 Marks

Soln. :

$$\text{XTAL frequency} = 8 \text{ MHz}$$

$$\therefore \text{Timer clock frequency} = \frac{f_{\text{osc}}}{4} \times \frac{1}{128} = 0.015625 \text{ MHz}$$

$$\text{Timer clock period} = 64 \mu\text{s}$$

$$\text{Number of pulses to be counted} = \frac{1 \text{ ms}}{64 \mu\text{s}} = 15.625$$

$$\therefore \text{Count} = 65536 - 15 \\ = (65521)_{10} = (\text{FFF1})_H$$

$$\therefore \text{TMR0H} = FF H$$

$$\therefore \text{TMR0L} = F1 H$$

**C program**

```
#include <PIC18F4580.h>
#define LED PORTBbits.RB5

void delay() //delay of 1ms
{
    TMROL = 0xF1H; //Load TMROL with 0xF1H
    TMROH = 0xFFH; //Load TMROH with 0xFFH
    TOCON = 0x06; //Program Timer0 in 16 bit mode, with 1:128 prescaler
    TOCONbits.TMR0ON = 1; //start Timer0
    while (INTCONbits.TMR0IF==0); //wait for TMROIF to overflow
    TOCON bits.TMR0ON = 0; //stop timer0
    INTCONbits.TMR0IF = 0; //clear TMROIF flag
}

void main (void)
{
    TRISBbits.TRISB5 = 0; // Program RB5 as output pin
    while (1)
    {
        LED = ~LED; // toggle bit LED
        delay(); //Call delay
    }
}
```

Program 5.7.26 : Write an assembly program using the timer 1 interrupt to create a square wave of 3 KHz on pin RB7.
Assume XTAL = 10 MHz.

SPPU - Dec. 14, 7 Marks

Soln. :

Let XTAL frequency = 10 MHz.

$$\therefore \text{Timer clock frequency} = \frac{f_{\text{osc}}}{4} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5 \text{ MHz}}$$

= 0.4 μ s i.e. counter will increment every 0.4 μ s.

Assuming frequency of square wave required is 3KHz

$$\text{Time period of square wave} = \frac{1}{3 \text{ KHz}} = 333 \mu\text{s}$$

Hence, the square wave will be logic '1' for 167 μ s and logic '0' for 167 μ s.

$$\therefore \text{Number of counts} = \frac{167 \mu\text{s}}{0.4 \mu\text{s}} = 418$$

$$\therefore \text{Count} = 65536 - 418 = (65118)_{10} = \text{FE5EH}$$

$$\therefore \text{TMROH} = \text{FEH}$$

$$\therefore \text{TMROL} = \text{5EH}$$



Assembly program

Label	Instruction	Comments
	ORG 0000H	
	GOTO START	Jump to start label
	ORG 0008 H	Interrupt vector table
	BTFS S INTCON, TMROIF	Is the interrupt due to Timer 0 interrupt ?
	RETFIE	If not return to start
	GOTO Timer_ISR	If yes, goto Timer 0 ISR
	ORG 0100H	
START :	BCF TRISB, 7	Program RB7 an output pin
	MOVWF TOCON	Load TOCON register
	MOVLW 0x08	Program Timer 0 as 16-bit timer with internal clock and no prescaler
	MOVLW 0xFE	Initialize TMROH
	MOVWF TMROH	
	MOVLW 0x5E	Initialize TMROL
	MOVWF TMROL	
	BCF INTCON, TMROIF	Clear TMROIF flag
	BSF TOCON, TMROON	Start Timer 0
	BSF INTCON, PEIE	Enable all peripheral interrupts
	BSF INTCON, GIE	Globally enable interrupts
here:	BSF INTCON, TMROIE	Enable Timer 0 interrupt
	BRA here	
Timer_ISR	ORG 0200H	
	MOVLW 0xFE	Initialize TMROH
	MOVWF TMROH	
	MOVLW 0x5E	Initialize TMROL
	MOVWF TMROL	
	BTC PORTB, 7	Toggle bit RB7
	BCF INTCON, TMROIF	Clear Timer 0 Interrupt Flag
	RETFIE	Return from ISR
	END	



Program 5.7.27 : Write assembly language program by using timer 0 interrupt to generate square wave on pin RB1.

SPPU - Dec. 14, 6 Marks

Soln.:

Let XTAL frequency = 10 MHz.

$$\therefore \text{Timer clock frequency} = \frac{f_{\text{xtal}}}{4} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5 \text{ MHz}}$$

= 0.4 μ s i.e. counter will increment every 0.4 μ s.

Assuming frequency of square wave required is 3KHz

$$\text{Time period of square wave} = \frac{1}{3 \text{ KHz}} = 333 \mu\text{s}$$

Hence, the square wave will be logic '1' for 167 μ s and logic '0' for 167 μ s.

$$\therefore \text{Number of counts} = \frac{167 \mu\text{s}}{0.4 \mu\text{s}} = 418$$

$$\therefore \text{Count} = 65536 - 418 = (65118)_{10} = \text{FE5EH}$$

$$\therefore \text{TMR0H} = \text{FEH}$$

$$\therefore \text{TMR0L} = \text{5EH}$$

Assembly program:

Label	Instruction	Comments
	ORG 0000H	
	GOTO START	Jump to start label
	ORG 0008 H	Interrupt vector table
	BTFS INTCON, TMROIF	Is the interrupt due to Timer 0 interrupt?
	RETFIE	If not return to start
	GOTO Timer_ISR	If yes, goto Timer 0 ISR
	ORG 0100H	
START :	BCF TRISB, 1	Program RB1 an output pin
	MOVWF TOCON	Load TOCON register
	MOVLW 0x03	Program Timer 0 as 16-bit timer with internal clock and no prescaler
	MOVLW 0xFE	Initialize TMR0H
	MOVLW 0x5E	
	MOVWF TMROH	
	MOVLW 0x5E	Initialize TMR0L
	MOVWF TMROL	
	BCF INTCON, TMROIF	Clear TMROIF flag
	BSF TOCON, TMROON	Start Timer 0

Label	Instruction	Comments
	BSF INTCON, PEIE	Enable all peripheral interrupts
	BSF INTCON, GIE	Globally enable interrupts
	BSF INTCON, TMROIE	Enable Timer 0 interrupt
here:	BRA here	
Timer_ISR	ORG 0200H	
	MOVLW 0xFE	Initialize TMROH
	MOVWF TMROH	
	MOVLW 0x5E	Initialize TMROL
	MOVWF TMROL	
	BTG PORTB, 1	Toggle bit RB1
	BCF INTCON, TMROIF	Clear Timer 0 Interrupt Flag
	RETFIE	Return from ISR
	END	

Program 5.7.28 : Write assembly language program by using timer 0 interrupt to generate square wave on pin RB1.

SPPU - Dec. 14; 6 Marks

Soln.:

Let XTAL frequency = 10 MHz.

$$\therefore \text{Timer clock frequency} = \frac{f_{\text{ext}}}{4} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5 \text{ MHz}}$$

= 0.4 μ s i.e. counter will increment every 0.4 μ s.

Assuming frequency of square wave required is 3KHz

$$\text{Time period of square wave} = \frac{1}{3 \text{ KHz}} = 333 \mu\text{s}$$

Hence, the square wave will be logic '1' for 167 μ s and logic '0' for 167 μ s.

$$\therefore \text{Number of counts} = \frac{167 \mu\text{s}}{0.4 \mu\text{s}} = 418$$

$$\therefore \text{Count} = 65536 - 418 = (65118)_{10} = \text{FESEH}$$

$$\therefore \text{TMROH} = \text{FEH}$$

$$\therefore \text{TMROL} = \text{SEH}$$

Assembly program

Label	Instruction	Comments
	ORG 0000H	
	GOTO START	Jump to start label
	ORG 0008 H	Interrupt vector table

Label	Instruction	Comments
	BTFSS INTCON.TMROIF	Is the interrupt due to Timer 0 interrupt?
	RETFIE	If not return to start
	GOTO Timer_ISR	If yes, goto Timer 0 ISR
	ORG 0100H	
START :	BCF TRISB, 1	Program RB1 an output pin
	MOVWF TOCON	Load TOCON register
	MOVLW 0x08	Program Timer 0 as 16-bit timer with internal clock and no prescaler
	MOVLW 0xFE	Initialize TMROH
	MOVWF TMROH	
	MOVLW 0x5E	Initialize TMROL
	MOVWF TMROL	
	BCF INTCON.TMROIF	Clear TMROIF flag
	BSF TOCON.TMROON	Start Timer 0
	BSF INTCON.PEIE	Enable all peripheral interrupts
	BSF INTCON.GIE	Globally enable interrupts
	BSF INTCON.TMROIE	Enable Timer 0 interrupt
here:	BRA here	
Timer_ISR	ORG 0200H	
	MOVLW 0xFE	Initialize TMROH
	MOVWF TMROH	
	MOVLW 0x5E	Initialize TMROL
	MOVWF TMROL	
	BTG PORTB, 1	Toggle bit RB1
	BCF INTCON.TMROIF	Clear Timer 0 Interrupt Flag
	RETFIE	Return from ISR
	END	

5.8 Exam Pack (Review and University Questions)

- Q. Explain the use of Pre-scalar. (Refer Section 5.2) (Aug. 17 (In sem.), 2 Marks)
- Q. With reference timers explain the terms pre scalar and post scalar. (Refer Section 5.2) (May 18, 4 Marks)
- Q. Explain T0CON register. (Refer Section 5.3.2) (Aug. 15 (In Sem.), Oct. 16 (In Sem.), 4 Marks)
- Q. Explain Timer 0 (T0CON) control register in detail. (Refer Section 5.3.2) (Dec. 16, 6 Marks)

- Q. Draw the T0CON register (Refer Section 5.3.2) (Aug. 17 (In sem.), 4 Marks)
- Q. Draw T0CON register and explain function of individual bits of T0CON register.
(Refer Section 5.3.2) (May 18, Dec. 18, 6 Marks)
- Q. Explain timer 1 and its applications of PIC18XX in detail (Refer Section 5.4) (Dec. 14, May 15, 2 Marks)
- Q. Assuming XTAL = 10 MHz, write a program to generate a square wave of 2 KHz on port C.5
(Refer Program 5.7.2(A)) (Aug. 15, 6 Marks)
- Q. Write assembly language program to toggle all the bits of port B continuously with some delay. Use timer 0 in 16 bit mode and no prescaler to generate maximum delay. XTAL = 10 MHz. (Refer Program 5.7.21) (Dec. 14, 6 Marks)
- Q. Assuming that clock pulses are fed into pin T0CK1, Write a program for counter 0 in 8 bit mode to count the pulses and display the state of the TMR0L count on PORTB. (Refer Program 5.7.22) (May 15, 6 Marks)
- Q. Find the value to be loaded in T0CON for following configuration : Timer 0 in 16 bit mode, prescaler of 128 and internal clock. (Refer Program 5.7.23) (May 15, 4 Marks)
- Q. Write an assembly program using timer 1 to blink LED connected to PORTB at every 500 microsecond. Crystal frequency is 16MHz. (Refer Program 5.7.24) (Dec. 15, 7 Marks)
- Q. Using Timer 0 in 16-bit mode, write a C language program to obtain a time delay of 1 ms. Assume 8-MHz crystal, leading edge clock, and a prescale value of 1: 128. (Refer Program 5.7.25) (May 16, 6 Marks)
- Q. Write an assembly program using the timer 1 interrupt to create a square wave of 3 KHz on pin RB7. Assume XTAL = 10 MHz. (Refer Program 5.7.26) (Dec. 14, 7 Marks)
- Q. Write assembly language program by using timer 0 interrupt to generate square wave on pin RB1.
(Refer Program 5.7.27) (Dec. 14, 6 Marks)
- Q. Write assembly language program by using timer 0 interrupt to generate square wave on pin RB1.
(Refer Program 5.7.28) (Dec. 14, 6 Marks)
- Q. Write C program to generate delay of 50 msec. using Timer 0. Assume crystal frequency of 10MHz.
(Refer Program 5.7.2) (Dec. 19, 5 Marks)

□□□

Note

Model Question Paper (In Sem.)

Processor Architecture (214451)

Semester IV - Information Technology (Savitribai Phule Pune University)

Time : 1 Hour

Maximum Marks : 30

Instructions to the candidates :

1. Answer Q.1 or Q.2, Q.3 or Q.4
2. Neat diagrams must be drawn wherever necessary.
3. Figure to the right indicate full marks.
4. Make suitable assumptions, if necessary.

Q. 1 (a) Draw the interrupt handling in PIC18F458. (Refer Section 1.10.13) (5 Marks)

(b) Compare Microprocessor and Microcontroller. (Refer Section 1.2.3) (5 Marks)

(c) List the features of PIC 18F458. (Refer Section 1.8) (5 Marks)

OR

Q. 2 (a) Explain Program Memory organization of PIC18F458. (Refer Section 1.13.2) (5 Marks)

(b) Write an instruction sequence to add 10 to the data registers at 0x300-0x303 using the indirect post increment addressing mode. (Refer Program 2.17.5) (5 Marks)

(c) Explain Pipelining in PIC18F458. (Refer Section 1.14) (5 Marks)

Q. 3 (a) Write a program to add 5 elements in an array starting from 0x20H. Store the results at 0 x 40 H.

(Refer Program 2.18.5) (5 Marks)

(b) Explain the port structure of PIC18F458 and hence write a program to generate square wave on all pins of Port A. (Refer Section 4.2 and Program 4.7.1) (5 Marks)

(c) Draw the block diagram of PIC 18F458 (Refer Section 1.10) (5 Marks)

OR



- Q. 4**
- (a) Write a program in C to configure Port B as input port and the most significant 4 bits of Port D as input bits and the least significant 4 bits of the same port as outputbits. (Refer Program 4.7.7) (3 Marks)
 - (b) Explain the timer0 modes and registers of PIC18F458. (Refer Sections 5.3.1 and 5.3.2) (7 Marks)
 - (c) Explain working register and status register of PIC18F458.
(Refer Sections 1.10.5(A) and 1.10.5(B)) (5 Marks)
-

□□□

6

UNIT - III

Interrupt Programming

6.1 Interrupts Vs Polling

- There are two methods to achieve this synchronization, polling method and interrupt-driven method. When a data is to be read from an input device the Interface chip has to latch the data from the input device.
- While reading from the input devices using these methods are explained as follows:
 1. **The polling method:** In this case the interface chip uses a status flag to indicate whether it has read a data from the I/O device and is ready for the processor to access it. Whenever the processor wants to access this device, it checks this flag to decide if new data has been received from the input device. The microprocessor may check this status flag continuously or periodically.
 2. **Interrupt-driven method:** In this case, the interface chip generates an interrupt signal to the microprocessor whenever it has received new data from the input device. The interrupt service routine is then executed which makes the microprocessor to read the input data.
- Similarly while writing data to the output device, the microprocessor has to make sure that the output device is not busy. The two synchronization methods implement this as explained follows:
 1. **The polling method :** As already discussed this type of interface device uses a status flag to indicate whether the previous data has been sent to the output device and is it ready to accept new data. The microprocessor may check this status flag continuously or periodically.
 2. **Interrupt-driven method :** In this case the interface chip generates an interrupt signal to the microprocessor whenever previous data has been sent to the output device and it is ready to new data.

This causes the microprocessor to execute the interrupt service routine which sends the data to the interface chip.

- The PIC18 parallel ports do not support either of these methods directly in the hardware. If the user wants he can add an external parallel interface chip, such as an Intel 8255, to the PIC18 microcontroller, to implement either of the two methods.

6.2 Interrupt Service Routine (ISR)

- When an interrupt occurs, the processor services the same by executing a small program called as Interrupt Service Routine (ISR). This ISR is stored in the memory. The address where the ISR is stored is called as an ISR.
- The interrupt vector addresses for PIC microcontroller are as given in the Table 6.2.1.

Table 6.2.1 : Interrupt vector table (IVT) for the PIC18

Interrupt	ROM Location (Hex)
Power-on Reset	0000
High priority interrupt	0008 (Default upon power-on reset)
Low priority interrupt	0018

Interrupt sources of PIC 18Fxx8 microcontroller are listed as follows:

1. One interrupt for each of the timers.
2. External hardware interrupts RB0 (Port B.0), RB1 (Port B.1) and RB2 (Port B.2) pins named as INT0, INT1 and INT2.
3. Two interrupts for serial communication i.e. one for reception complete indication and one for transmission complete indication.
4. One interrupt for the ADC complete indication.
5. Also one interrupt for each channel of CCP indicating compare equal and captured in the modes Compare and Capture respectively.



- Besides these, there are many other interrupt sources in PIC. The gate diagram Fig. 6.2.1 resembles the interrupt signalling in PIC. Each of these interrupts can be programmed as higher priority and lower priority interrupts.

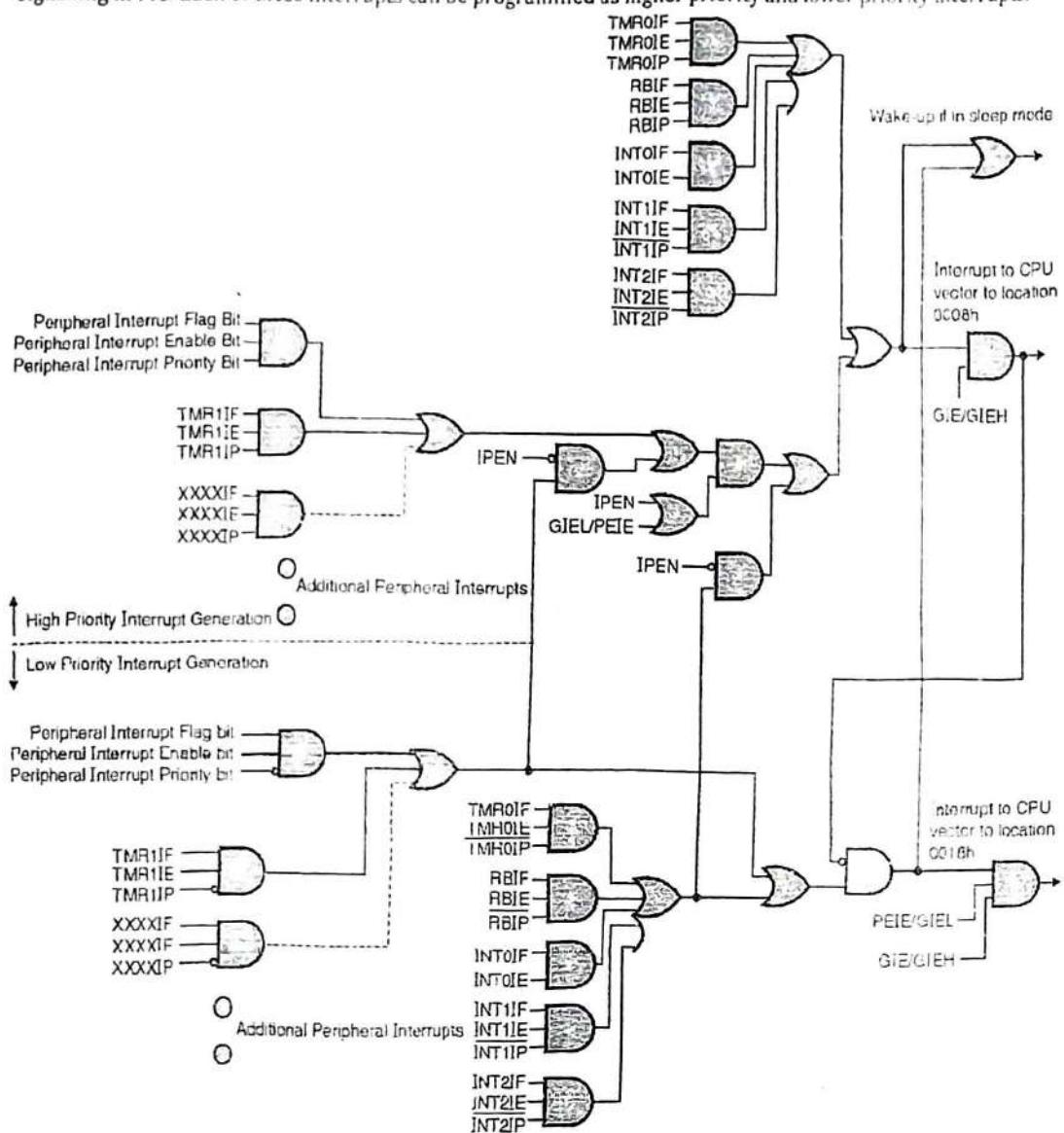


Fig. 6.2.1

6.3 Steps in Executing an Interrupt

University Question

Q. Discuss steps in executing an interrupt in PIC18F458.

SPPU - Dec. 19, 8 Marks

The following steps take place when an interrupt occurs :

- Step 1 : The current instruction execution is completed.
- Step 2 : The return address is pushed onto the stack.
- Step 3 : The control transfers to the interrupt vector table (IVT) that has the ISR address.
- Step 4 : The ISR address is copied into the Program Counter (PC) and hence the control is transferred to the new location.
- Step 5 : The last instruction of the ISR i.e. RETFIE when encountered the microcontroller gets back (POPs) the return address from the stack and hence returns to the location from where branching had occurred.

6.4 PIC18 Interrupt Structure

University Questions

- Q. Write a short note on interrupt structure of PIC18 microcontroller. **SPPU - Dec. 17, 9 Marks**
- Q. Explain the interrupt structure of PIC18 microcontroller. **SPPU - Dec. 18, 8 Marks**

There are 13 registers that are used to control interrupt operation. These registers are:

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

6.4.1 Timer Flag Interrupts

6.4.1(A) PIR Registers

The Peripheral Interrupt Request (PIR) registers contain the individual flag bits for the peripheral interrupts (Fig. 6.4.1 through Fig. 6.4.3). Due to the number of peripheral interrupt sources, there are three Peripheral Interrupt Request (Flag) registers (PIR1, PIR2, PIR3).

Note :

1. Interrupt flag bits are set when an interrupt condition occurs regardless of the state of its corresponding enable bit or the Global Interrupt Enable bit, GIE (INTCON register).
2. User software should ensure the appropriate interrupt flag bits are cleared prior to enabling an interrupt and after servicing that interrupt.

6.4.1(B) Interrupt Control Register

There are three interrupt control registers. Their structures and significance of each bit is shown in the Fig 6.4.1.

R/W-0	R/W-0	R/W-0	R/W-	R/W-	R/W-0	R/W-0	R/W-	R/W-
0	0	0			0	x		
GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	INTOIF	RBIF	bit0

Fig. 6.4.1 : INCON Interrupt Control Register

bit7 [GIEI/GIEH]: Global Interrupt Enable bit

When IPEN (RCON<7>) = 0:

- 1 = Enables all unmasked interrupts
- 0 = Disables all interrupts

Interrupt enables

When IPEN (RCON<7>) = 1:

1 = Enables all high priority interrupts

0 = Disables all priority interrupts

bit6 [PEIEI/GIEL]: Peripheral Interrupt Enable bit

When IPEN (RCON<7>) = 0:

1 = Enables all unmasked peripheral interrupts

0 = Disables all peripheral interrupts

When IPEN (RCON<7>) = 1:

1 = Enables allow priority peripherals interrupts

0 = Disables allow priority peripherals interrupts

bit5 [TMROIE] : TMRO Overflow Interrupt Enable bit

1 = Enables the TMRO overflow interrupt

0 = Disables the TMRO overflow interrupt

bit4 [INT0IE] : INT0 External Interrupt Enable bit

1 = Enables the INT0 external interrupt

0 = Disables the INT0 external interrupt

bit3 [RBIE] : RB Port change interrupt Enable bit

1 = Enables the RB port change interrupt

2 = Disables the RB port Change Interrupt

bit2 [TMROIF] : TMRO Overflow Interrupt Flag bit

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

0 = TMRO register did not overflow

bit1 [INT0IF] : INT0 External Interrupt Flag bit

1 = The INT0 external interrupt occurred (must be Cleared in software by reading PORT B)

0 = The INT0 external interrupt did not occur

bit0 [RBIF] : RB Port change Interrupt Flag bit

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

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

R/W-1 R/W-1 R/W-1 U-0 U-0 R/W-1 U-0 R/W-1

RBPU	INTEDGO	INTEDG1	-	-	TMROIP	-	RBIP
------	---------	---------	---	---	--------	---	------

Fig. 6.4.2 : INTCON2 - Interrupt Control Register 2

bit 7 [RBPU] : PORT B Pull-up Enable bit

1 = All PORT B pull-ups are disabled



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

bit 6 [INTEDG0] : External Interrupt 0 Edge Select bit

1 = Interrupt on rising edge

0 = Interrupt on falling edge

bit 5 [INTEDG1] : External interrupt1 Edge Select bit

1 = Interrupt on rising edge

0 = Interrupt on falling edge

bit 4-3 [Unimplemented] : Read as '0'

bit 2 [TMR0IP] : TMRO Overflow Interrupt Priority bit

1 = High priority

0 = Low priority

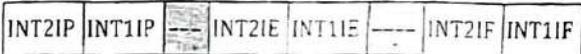
bit 1 [Unimplemented] : Read as '0'

bit 0 [RBIP] : RB Port Change interrupt Priority bit

1 = High priority

0 = Low priority

R/W-1 R/W-1 U-0 R/W-0 R/W-0 I-0 R/W-0 R/W-0



bit 7

bit 0

Fig. 6.4.3 : INTCON3 : interrupt control register 3

bit7 [INT2IP] : INT2 External Interrupt Priority bit

1 = High priority

0 = Low priority

bit6 [INT1IP] : INT1 External Interrupt Priority bit

1 = High priority

0 = Low priority

bit5 [Unimplemented] : Read as '0'

bit4 [INT2IE] : INT2 External Interrupt Enable bit

1 = Enables the INT2 external interrupt

0 = Disables the INT2 external interrupt

bit3 [INT1IE] : INT1 External Interrupt Enable bit

1 = Enables the INT1 external interrupt

0 = Disables the INT1 external interrupt

bit2 [Unimplemented] : Read as '0'

bit1 [INT2IF] : INT2 External Interrupt Flag bit

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

0 = The INT2 external interrupt did not occur

bit0 [INT1IF] : INT1 External interrupt Flag bit

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

0 = The INT1 external interrupt did not occur

Program 6.4.1 : Show the instructions to (a) enable (unmask) the Timer 0 interrupt and external hardware interrupt 0 (INT0), and (b) disable (mask) the Timer0 interrupt . then (c) show how to disable (mask) all the interrupts with a single instruction.

Soln.:

(a) BSF INTCON, TMROIE ; enable Timer 0 interrupt

BSP INTCON,INT0IE ; enable external interrupt 1 (INT0)

BSF INTCON, GIE ; allow all interrupts to come in

We perform the above action with the following two instructions:

MOVW B'10110000' ; GIE = 1, TMROIF = 1, INTIFO = 1

MOVWE INTCON ; Load (Disable) Timer0 interrupt

(b) BCF INTCON, TMROIE ; mask (disable) Timer0 interrupt

(c) BCF INTCON, GIE ; mask all interrupts globally



6.5 PIC18 Interrupt Programming in C Using C18 Compiler

Program 6.5.1 : Write a program for a system with 8 switches on port C and 8 LEDs on port D. Use timer 0 to generate a square wave on Port B pin 5.

```
ORG 0000H
GOTO MAIN          ; Jump to main function at 0100H
ORG 0008H          ; interrupt vector table
BTFS S INTCON, TMROIF ; Is it a Timer0 interrupt?
RETFIE            ; if no, then return to main
GOTO TO_ISR        ; if yes, then go Timer
                   ; 0 ISR

;--main program for initialization and keeping CPU busy
ORG 00100H;
MAIN BCF TRISB, 5      ; Set PB5 as an output pin
SETFTRISC           ; Set PORTC as output port
CLRF TRISD           ; Set PORTD as output port
MOVLW 0x08            ; Timer0 configured as 16-bit, noprescale, internal clk
MOVWF TOCON           ; load TOCON reg
MOVLW 0xF2            ; Load W reg with the low byte
MOVWF TMROL           ; load Timer0 low byte
MOVLW 0xFF            ; Load W reg with the high byte
MOVWF TMROH           ; load Timer0 high byte
BCFINTCON, TMROIF     ; clear timer interrupt flag bit
BSF TOCON, TMROON     ; start Timer0
BSF INTCON, TMROIE    ; enable Timer 0 interrupt
BSF INTCON, GIE        ; enable interrupts globally
;-- Make the processor to wait for interrupt
OVERMOVFF PORTC, PORTD ; send data from to
PORTC
PORTD BRA OVER         ; stay in this loop forever
TO_ISR                ; ISR for Timer 0
ORG 200H
MOVLW 0xFF            ; Load W reg with the high byte
MOVWF TMROH           ; load Timer0 high byte
MOVLW 0xF2            ; Load W reg with the low byte
MOVWF TMROL           ; Load Timer 0 low byte
BTGPOR TB, 5           ; toggle RB5
BCFINTCON, TMROIF     ; clear timer interrupt flag bit
EXIT RETFIE           ; return from interrupt service routine
END
```

6.5.1 Difference between the RETURN and RETFIE

Both the instructions i.e. RETURN and RETFIE perform the same actions of popping off the top bytes from the stack top into the program counter, and thereby the processor to return to where it left off. However, RETFIE also performs the additional task of clearing the GIE flag, thus the indication that the current interrupt servicing is done and now can process a new interrupt.

Hence, we cannot use RETURN instead of RETFIE, else the interrupts will not be enabled.

Sr.No.	RETURN	RETFIE
1	It performs the task of popping off the top bytes from the stack top into the program counter	It performs the task of popping off the top bytes from the stack top into the program counter
2	It doesn't clear the GIE flag bit	It also performs a task of clearing the GIE flag, thus the indication that the current interrupt servicing is done and now can process a new interrupt
3.	It must be used after the functions or procedures	It must be used after Interrupt service routines

Program 6.5.2: Write a program using Timer 0 and Timer 1 interrupts to generate square waves on pins RB1 and RB7 while the data is transferred from port C to port D.

OR Write assembly language program by using timer 0 interrupt to generate square wave on pin PB1.

SPPU - Dec. 14, 6 Marks

Soln.:

```

ORG 0000H
GOTO MAIN ; Jump to main function at 0100H
ORG 0008H ; interrupt vector table
GOTO CHK INT ; Jump to location 0040H
; check to see the source of interrupt
ORG 0040H
CHK INT:
    BTFSC INTCON, TMROIF ; Is it a Timer0
    ; interrupt?
    BRATO_ISR ; If yes, then branch to T0_ISR
    BTFSC PIR1, TMRIIF ; Is it Timer1 interrupt?
    BRAT1_ISR ; If yes, then branch to T1_ISR
    RETFIE ; If no, then return to main
;--main program for initialization and keeping CPU busy
ORG 0100H
MAIN BCF TRISB, 1 ; SetPB1 as an output pin
SETF TRISC ; Set PORTC as input port
CLRF TRISD ; Set PORTD as output port
BCF TRISD, 7 ; SetPB7 as an output pin
MOVLW 0x08 ; Configure Timer0 as 16-bit, noprescale, internal clk

```



```

MOVWF TOCON ; Initialize TOCON reg
MOVLW 0xF2 ; Load W reg. with F2H, the lower byte
MOVWF TMROL ; load lower byte into TMROL
MOVLW 0xFF ; Load W reg. with FFH, the higher byte
MOVWF TMROH ; load higher byte into TMROH
BCF INTCON, TMROIF ; clear Timer0 interrupt flag bit
MOVLW 0x0 ; Program Timer1 to 16-bit, noprescale mode and internal clk
MOVWF T1CON ; load T1CON reg
MOVLW 0xF2 ; load W reg. with F2H, the lower byte
MOVWF TMR1L ; load lower byte into TMR1L
MOVLW 0xFF ; load W reg. with FFH, the higher byte
MOVWF TMR1H ; load higher byte into TMR1H
BCF PIR1,TMR1IF ; clear Timer1 interrupt flag bit
BSF PIE1, TMR1IE ; enable Timer1 interrupt
BSF INTCON, TMROIE ; enable Timer0 interrupt
BSF INTCON, PEIE ; enable peripheral interrupts
BSF INTCON, GIE ; enable global interrupts
BSF T1CON, TMR1ON ; start Timer1
BSF TOCON, TMROON ; start Timer0
-----keep the processor busy waiting for interrupt
OVER MOVFFPORTC, PORD ; send data from
ORTC ; to PORTD
BRAOVER ; stay in this loop forever
; ---ISR for Timer 0
TO_ISR
ORG 200H
MOVLW 0xFF ; Load W reg. with FFH, the higher byte
MOVWF TMROH ; load higher byte into TMROH
MOVLW 0xF2 ; Load W reg. with F2H, the lower byte
MOVWF TMROL ; load lower byte into TMROL
BTG PORTB, 1 ; toggle PB1
BCF INTCON, TMROIF ; clear timer interrupt flag bit
GOTO CHK_INT ; -----ISR for Timer1
T1_ISR
ORG 300H
MOVLW 0xF2 ; load lower byte into TMR1L
MOVWF TMR1L ; load lower byte into TMR1L
MOVLW 0xFF ; Load W reg. with FFH, the higher byte
MOVWF TMR1H ; load higher byte into TMR1H
BTGPORTE, 7 ; Toggle port B pin 7
BCF PIR1, TMR1IF ; clear Timer1 interrupt flag bit
GOTOCHK_INT
END

```



Program 6.5.2(A): Write a program in C language which will copy a byte from Port C to Port D on occurrence of an INT0 interrupt.

SPPU - Oct. 16, 6 Marks

Soln.:

```
#include <P18F4580.h>
#pragma code hi_priori = 0x0008 // high priority interrupt location
void interrupt0_ISR (void)
{
    PORTD = PORTC; // Copy from Port C to PORTD
    INTCONbits.INT0IF = 0; // Clear external interrupt 0
}
void hi_priori(void)
{
    if (INTCONbits.INT0IF == 1) // if interrupt is there on external interrupt 0
        interrupt0_ISR();
}
void main (void)
{
    TRISD = 0; // make port D an output port
    TRISC = 0x FF; // Make port C as input port
    INTCONbits.INT0IF = 0; // Clear External interrupt 0
    INTCONbits.INT0IE = 1; // Enable external interrupt 0
    INTCONbits.GIE = 1; // Enable global interrupt
}
```

Program 6.5.2(B): Write a program in C language which will complement the contents of Port D on occurrence of an INT0 interrupt.

SPPU - Aug. 15 (In Sem.), 6 Marks

Soln.:

```
#include <P18F4580.h>
#pragma code hi_priori = 0x0008 // high priority interrupt location
void interrupt0_ISR (void)
{
    PORTD = ~ PORTD; // Complement PORTD
    INTCONbits.INT0IF = 0; // Clear external interrupt 0
}
void hi_priori(void)
{
    if (INTCONbits.INT0IF == 1) // if interrupt is there on external interrupt 0
        interrupt0_ISR();
}
void main (void)
{
    TRISD = 0; // make port D an output port
```

```

INTCONbits.INT0IF = 0; // Clear External interrupt 0
INTCONbits.INT0IE = 1; // Enable external interrupt 0
INTCONbits.CIE = 1; // Enable global interrupt
}

```

Program 6.5.2(C): Write an assembly program using the timer 0 interrupt to create a square wave of 3 KHz on pin RB7.
Assume XTAL = 10 MHz.

SPPU - Dec. 14, 7 Marks

Soln. :

$$\text{The time period of square wave} = \frac{1}{3 \text{ KHz}} = 333 \mu\text{s}$$

Let us assume the duty cycle of square wave is 50%. Hence, the pin will be high for 166μs and low for 166μs.

$$\text{Let XTAL} = 10 \text{ MHz.}$$

$$\therefore \text{Timer clock frequency} = \frac{f_{osc}}{4} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\text{Timer clock period} = \frac{1}{2.5 \text{ MHz}}$$

= 0.4 μs i.e. counter will increment every 0.4 μs.

$$\therefore \text{Number of pulses to be counted} = \frac{166 \mu\text{s}}{0.4 \mu\text{s}} = 416$$

$$\therefore \text{Counter value} = 65536 - 416 = (65120)_{10} = \text{FE60H}$$

$$\therefore \text{TMROH} = \text{FEH}$$

$$\therefore \text{TMROL} = 60H$$

Assembly program

Label	Instruction	Comments
	ORG 0000H	
	GOTO START	Jump to start
	ORG 0008 H	Interrupt vector table
	BTFSS INTCON, TMROIF	Is it because of Timer 0 interrupt?
	RETFIE	If not return to start
	GOTO Timer_ISR	If yes goto Timer 0 ISR
	ORG 0100H	
START :	BCF TRISB, 5	Set RB5 as an output pin
	MOVLW 0x08	Program Timer 0 as internal clock, 16 bit mode and no prescaler
	MOVWF T0CON	
	MOVLW 0xFE	Load TMROH
	MOVWF TMROH	
	MOVLW 0x60	Load TMROL
	MOVWF TMROL	



Label	Instruction	Comments
	BCF INTCON, TMROIF	Clear TMROIF flag
	BSF TCON, TMR0ON	Start Timer 0
	BSF INTCON, TMROIE	Enable Timer 0 interrupt
	BSF INTCON, GIE	Globally enable interrupts
	BSF INTCON, PEIE	Enable all peripheral interrupts
here:	BRA here	Wait for timer interrupt
Timer_ISR	ORG 0200H	
	MOVLW 0xFE	Load TMR0H
	MOVWF TMR0H	
	MOVLW 0x60	Load TMR0L
	MOVWF TMR0L	
	BTG PORTB, 5	Toggle bit RB5
	BCF INTCON, TMROIF	Clear Timer 0 Interrupt Flag
	RETIE	
	END	

Program 6.5.3 : Use timer 0 and timer 1 interrupts to generate square waves on pins RB1 and RB7, while data is being transferred from PORTC to PORTD.

Soln.:

```
#include <P18F458.h>
#define myPB1bit    PORTBbits.RB1
#define myPB7bit    PORTBbits.RB7
void T0_ISR (void);
void T1_ISR (void);
#pragma interrupt chk_isr           //used for high priority interrupt only
void chk_isr (void)
{
    if (INTCONbits.TMR0IF == 1) //Timer0 causes interrupt?
        T0_ISR ();           //Yes. Execute Timer0 ISR
    if (PIR1bits.TMR1IF == 1)   //Or was it Timer1?
        T1_ISR ();           //Yes. Execute Timer1 ISR
}
#pragma code My_HiPrio_Int = 0x08//high priority interrupt
void My_HiPrio_Int (void)
{
```

```

asm
GOTOchk_isr
_endasm
}

#pragma code
void main (void)
{
    TRISBbits. TRISB1 = 0; // RB1 = OUTPUT
    TRISRbits. TRISB7 = 0; // RB7 = OUTPUT
    TRISC = 255;           // PORTC = INPUT
    TRISD = 0;             // PORTD = OUTPUT
    TOCON = 0x0;           // T1CON, 16-bit mode, no prescaler
    TMROH = 0x35;          // load TH0
    TMROL = 0x00;          // load TL0
    T1CON = 0x88;          // Timer1, 16-bit mode, no prescaler
    TMR1H = 0x35;          // load TH1
    TMR1L = 0x00;          // load TL1
    INTCONbits. TMROIF = 0; // clear TF0
    PIR1bits. TMR1IF = 0; // clear TF1
    INTCONbits. TMROIE = 1; // enable Timer0 interrupt
    INTCONbits. TMROIE = 1; // enable Timer1 interrupt
    TOCONbits. TMROON = 1; // turn on Timer0
    T1CONbits. TMR1ON = 1; // turn on Timer1
    INTCONbits. PEIE = 1; // enable all peripheral interrupts
    INTCONbits. GIE = 1; // enable all interrupt globally
    while (1)             // keep looping until interrupt comes
    {
        PORTD = PORTC; // send data from PORTC to PORTD
    }
}

void T0_ISR (void)
{
    my PB1bit= ~my PB1bit; //toggle PORTB. 1
    TMROH = 0x35;          //load TH0
    TMROL = 0x00;           //load TL0
    INTCONbits. TMROIF = 0; //clear TF0
}

void T1_ISR (void)
{
    my PB7bit = ~my PB7bit; //toggle PORTB. 7
    TMR1H = 0x35;           //load TH0
    TMR1L = 0x00;           //load TL0
    PIR1bits. TMR1IF = 0; //clear TF1
}

```

6.6 Hardware Interrupts

University Question

Q. Explain programming external hardware interrupts in PIC18 microcontroller.

SPPU - Dec. 14, 4 Marks

The three hardware interrupts are already discussed in the earlier sections. Let us see the bits relevant to the hardware interrupts.

Table 6.6.1 : Hardware interrupt flag bits and associated registers

Interrupt (pin)	Flag bit	Register	Enable bit	Register
INT0 (RB0)	INT0IF	INTCON	INT0IE	INTCON
INT1 (RB1)	INT1IF	INTCON3	INT1IE	INTCON3
INT2 (RB2)	INT2IF	INTCON3	INT2IE	INTCON3

Program 6.6.1: Write a program which toggles the LED connected on port B pin 7 every time an interrupt occurs on INT0 and also transfers data from port C to port D.

SPPU - Dec. 17, 9 Marks

Soln.:

```
#include<P18F4580.h>
#define mybit PORT B bits.RB7
void chk_isr(void);
void INT0_ISR(void); // pragma interrupt chk_isr; // used for high-priority int
void chk_isr(void)
{
    if(INTCONbits.INT0IF==1) //INT0 caused interrupt?
        INT0_ISR(); //Yes. Execute INT0 program
}

#pragma code My_HiPrio_Int= 0x08 //high-priority interrupt location
void My_HiPrio_Int(void)
{
    asm
    GOTO chk_isr
    endasm
}

#pragma code
void main(void)
{
    TRISBbits.TRISB7=0; //RB7=OUTPUT
    TRISBbits.TRISB0=1; //INT0 = INPUT
    TRISC=0xFF; //PORTC = INPUT
    TRISD=0; //PORTD = OUTPUT
    INTCONbits.INT0IF=0; //clear TF1
    INTCONbits.INT0IE=1; //enable Timer0 interrupt
}
```

```

INTCONbits.CIE=1;           //enable all interrupts
while(1)                   //keep looping until interrupt comes
{
    PORTD=PORTC;
}
void INT0_ISR(void)
{
    mybit=~mybit;
    INTCONbits.INT0IE=0;   //clear INT0 flag
}

```

6.7 Serial Port Interrupts

- Interrupts are required for serial communication to indicate the end of transfer and also to indicate data received on the serial port.
- TXIF is the transfer interrupt indicating that the data in TXREG is transferred and can transfer next byte. RCIF is the Receive Interrupt indicating that the data is received on the serial port.
- It indicates that the data received must be read, else if another data is received, this data will be overwritten with the new data.
- The bits and registers relevant to the serial interrupts are listed in Table 6.7.1 :

Table 6.7.1 : Serial port interrupt flag bits and their associated registers

Interrupt	Flag bit	Register	Enable bit	Register
TXIF (Transmit)	TXIF	PIR1	TXIE	PIE1
RCIF (Receive)	RCIF	PIR1	RCIE	PIE1

6.8 Setting the Interrupt Priority

In the PIC18 microcontroller, there are only two levels of interrupt priority :

(a) low level, and (b) high level.

- While address 0008 is assigned to high-priority interrupts, the low-priority interrupts are directed to address 00018 in the interrupt vector table. Upon power-on-reset, all interrupts are automatically designated as high priority and will go to address 00008H.

- This is done to make the PIC18 compatible with the earlier generation of PIC microcontrollers such as PIC16xxx.
- We can make the PIC18 a two level priority system by way of programming IPEN (interrupt priority enable) bit in the RCON register.
- Upon power-on-reset, the IPEN bit contains 0, making the PIC18 a single priority level chip, just like the PIC16xxx. To make the PIC18 a two-level priority system, we must first set the IPEN bit to HIGH.
- It is only after making IPEN=1 that we can assign a low priority to any of the interrupts by programming the bits called IP (interrupt priority).
- IPR1 (interrupt priority register) has the IP bits for TXIP, RCIP, TMR1IP, and TMR2IP. If IPEN=1, then the IP bit will take effect and will assign a given interrupt a low priority.
- As a result of assigning a low priority to a given interrupt, it will land at the address 0018 instead of 0008 in the interrupt vector table.
- The IP (interrupt priority) bit along with the IF (interrupt flag) and IE (interrupt enable) bits will complete all the flags needed to program the interrupts for the PIC18.
- The INT0 has only one priority and that is high priority. That means all the PIC18 interrupts can be assigned a low or high priority level, except the external hardware interrupt to INT0.
- By making IPEN=1, we enable the interrupt priority feature.



- Now we must also enable two bits to enable the interrupts:
 - (a) We must set GIEH=1. The GIEH bit is part of the INTCON register (Fig. 6.4.1) and is the same as GIE, which we have used in previous sections. In this regard there is no difference between the priority and no-priority systems.
 - (b) The second bit we must set high is GIEL (part of INTCON). Making GIEL=1 will enable all the interrupts whose IP=0. That means all the interrupts that have been given the low priority will be forced to vector location 00018H.

Table 6.8.1 : Interrupt vector table for the PIC18

Interrupt	ROM Location (Hex)
Power-on-Reset	0000
High-priority Interrupt	0008 [Default upon power-on reset]
Low-priority interrupt	0018 (selected with IP bit)

6.9 Interrupt Inside an Interrupt / Nested Interrupt

- A high-priority interrupt can interrupt a low-priority interrupt. This is an interrupt inside an interrupt or nested interrupt
- In PIC18 microcontroller, a higher priority interrupt can suspend the execution of a lower priority interrupt and get itself serviced.
- But a same priority interrupt cannot suspend the execution of the ISR in execution.
- The GIEH and GIEL bits play an important role in the process of nested Interrupts. Whenever the processor starts executing an interrupt the GIEH and the GIE bits are automatically cleared, thereby disabling all interrupts. In case, we want higher priority interrupt to be nested, then we need to set the GIE bit.

- This will allow higher priority interrupts to suspend current interrupt service routine and get them serviced. The RETFIE instruction at the end of an ISR automatically sets the GIEH and GIE bits, thereby enabling the interrupts on return.

6.10 Exam Pack (Review and University Questions)

- Q. Write a short note on interrupt structure of PIC18 microcontroller.
(Refer Section 6.4) (Dec. 17, 9 Marks)
- Q. Explain the interrupt structure of PIC18 microcontroller.
(Refer Section 6.4) (Dec. 18, 8 Marks)
- Q. Write assembly language program by using timer 0 interrupt to generate square wave on pin RB1.
(Refer Program 6.5.2) (Dec. 14, 6 Marks)
- Q. Write a program in C language which will copy a byte from Port C to Port D on occurrence of an INT0 interrupt. (Refer Program 6.5.2(A))
(Oct. 16, 6 Marks)
- Q. Write a program in C language which will complement the contents of Port D on occurrence of an INT0 interrupt.
(Refer Program 6.5.2(B)) (Aug. 15 (In Sem.), 6 Marks)
- Q. Write an assembly program using the timer 1 interrupt to create a square wave of 3 KHz on pin RB7. Assume XTAL = 10 MHz.
(Refer Program 6.5.2(C)) (Dec. 14, 7 Marks)
- Q. Explain programming of external hardware interrupts in PIC18 microcontroller.
(Refer Section 6.6) (Dec. 14, 4 Marks)
- Q. Write a program which toggles the LED connected on port B pin 7 every time an interrupt occurs on INT0 and also transfers data from port C to port D.
(Refer Program 6.6.1) (Dec. 17, 9 Marks)
- Q. Discuss steps in executing an interrupt in PIC18F458. (Refer Program 6.3) (Dec. 19, 8 Marks)

7

UNIT - III

PIC Interfacing - I

7.1 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.

7.1.1 Key Switch Mechanism

- 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. 7.1.1 shows the general operation of a keyboard.

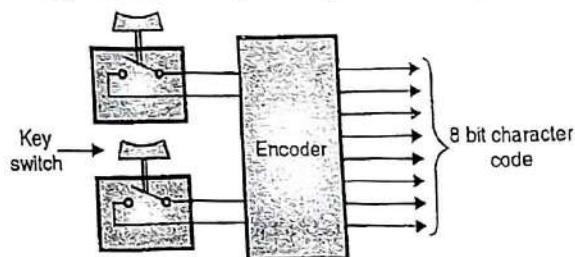


Fig. 7.1.1 : General operation of a keyboard

- 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.

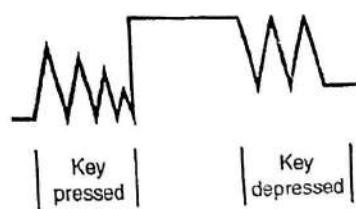
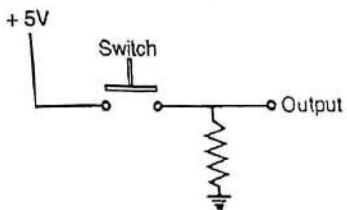


Fig. 7.1.2 : Bouncing of key switch

- 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. The key debouncing is done through hardware and software.
- Fig. 7.1.2 shows the bouncing of key switch.

7.1.2 Hardware Key Debouncing

- It is implemented by using flip-flop or latch. Fig. 7.1.3 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.

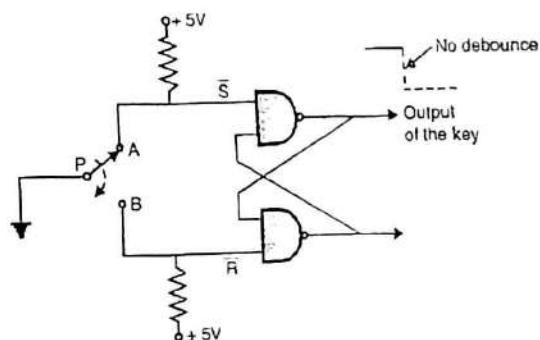


Fig. 7.1.3 : Hardware key debouncing

7.1.3 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. 7.1.4.

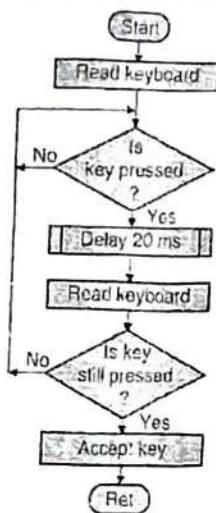


Fig. 7.1.4 : Software key debouncing

7.2 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.

7.2.1 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. 7.2.1 shows the interfacing of octal non-matrix type keyboard.

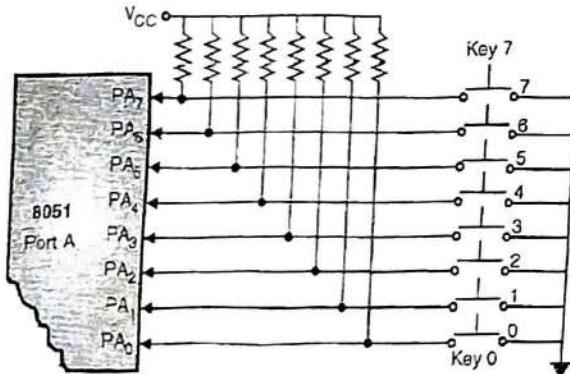


Fig. 7.2.1 : Non matrix type keyboard

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. 7.2.2

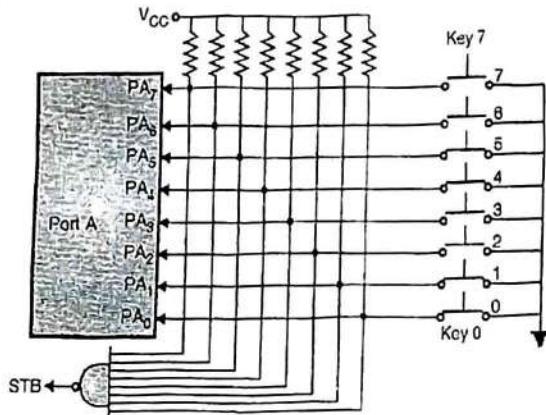


Fig. 7.2.2 : Hardware technique of identification

When all keys are open, the output of NAND gate (STB) goes low. When one of the keys is pressed, the output of NAND gate (STB) goes high. The STB is used to identify that the key is pressed. This STB signal can be used to interrupt the microcontroller.

7.2.2 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 is reduced.

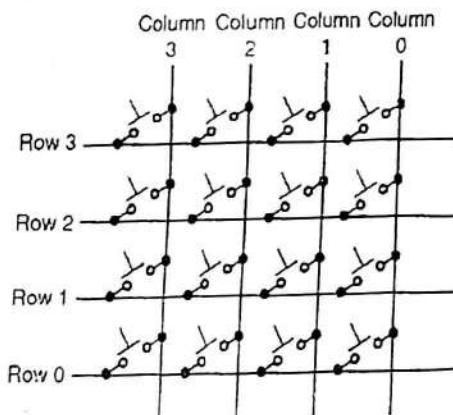


Fig. 7.2.3

- Fig. 7.2.3 shows 16 keys arranged in 4 rows and 4 columns. No connection 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.

- If non matrix type connection is used then 16 lines will be required. So using method reduces the number of connections.
- Fig. 7.2.4 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.

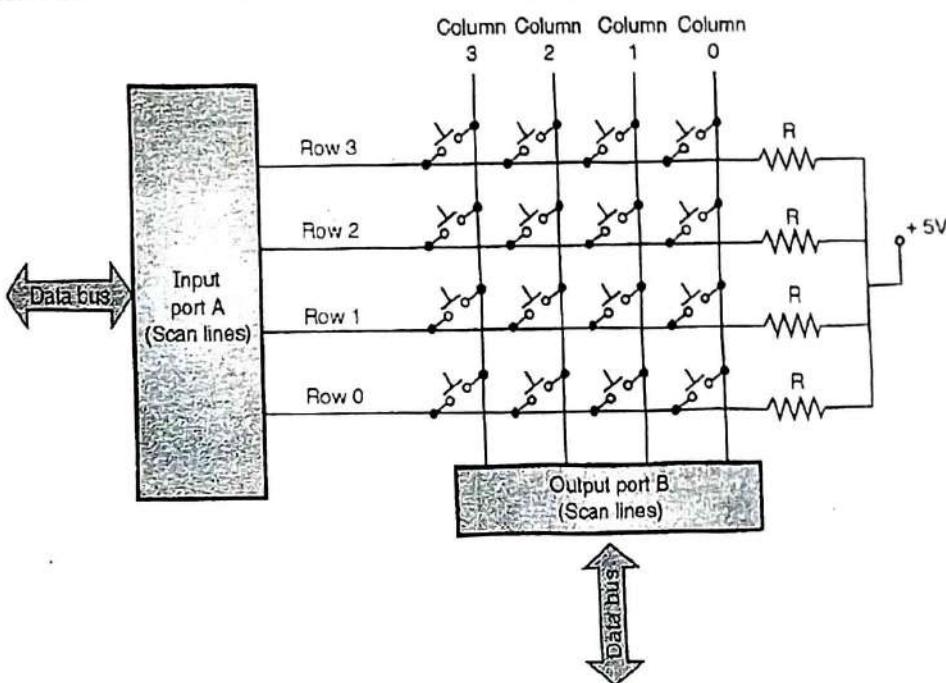


Fig. 7.2.4 : Matrix key board connections

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,

- To identify if any key is pressed or not.
 - 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.
 - Read the status of rows i.e. returns 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.
- Debouncing the key (Using software debouncing as explained earlier.)
- Identifying the pressed key.
 - Activate the keys from one column by making one column line zero.

- Read the status of return lines. The zero on any return line indicates that key is pressed.
- Activate the keys from next column and repeat steps (b) and (c) for all the columns.

Ex. 7.2.1: Interface a 4×4 matrix keyboard to PIC18F458. Display key pressed 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.

SPPU - May 15, Dec. 15, May 16, Dec. 16, 8 Marks

Soln.:

Fig. P. 7.2.1 shows the interfacing of a 4×4 matrix keyboard to PIC18F458.

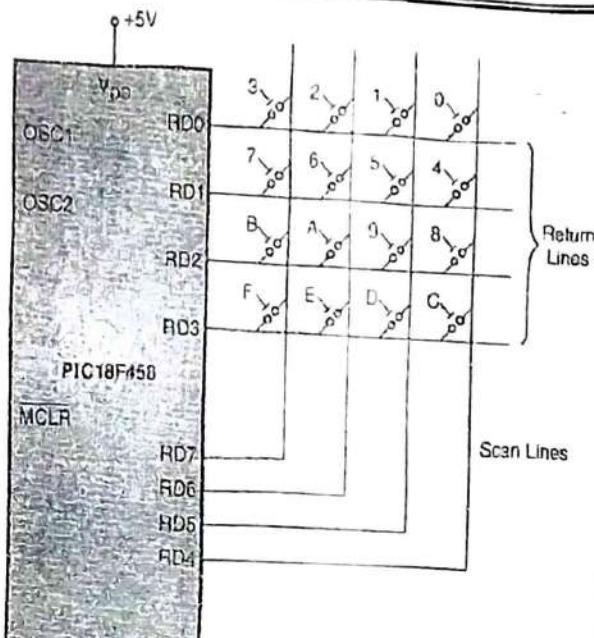


Fig. P. 7.2.1 : Interfacing a 4×4 matrix keyboard to PIC18F458

The 4×4 matrix keyboard is connected to the port D of PIC microcontroller. RD0-RD3 are return lines while pins RD7- RD4are return lines. For detecting the key presses two methods can be used. They are,

- Interrupt method
- Scanning method

In this program we use the interrupt method for detecting the key pressed.

```
#include <P18F458.h>
unsigned char keypad[4][4] = {{'0', '1', '2', '3'},{'4', '5', '6', '7'},{'8', '9', 'A', 'B'}, {'C', 'D', 'E', 'F'}};

void serialTX(unsigned char p)// send character
{
    while (PIR1bits.TXIF != 1); // wait till ready
    TXREG = p; // send key pressed to
    //serial port - hyper terminal
}

void delay(unsigned int msec)
{
    unsigned int xl, yl;
    for (xl = 0; xl < msec; xl++)
        for (yl = 0; yl < 165; yl++)
}
}
```

```
void IF_ISR(void) //Identify the key pressed
{
    unsigned char xl, C = 0, R = 4;
    delay(15);
    xl = PORTD; //get column into variable xl
    xl = xl ^ 0xF0; //toggle the high nibble
    if (!xl) return;
    while (xl << 1) C++; //determine the column
    PORTD = 0xFF; //select row 0
    if (PORTD != 0xFF) // is there a nibble
        //change yes then row = 0
    R = 0;
    else
    {
        PORTD = 0xFD; //select row 1
        if (PORTD != 0xFD) // is there a nibble
            //change
        R = 1; //yes then row = 1
        else
        {
            PORTD = 0xFB; //select row 2
            if (PORTD != 0xFB) //is there a nibble
                //change
            R = 2; //yes then row = 2
            else
            PORTD = 0xF7; //select row 3
            if (PORTD != 0xF7) // is there nibble
                //change ?
            R = 3; //yes then row = 3
        }
    }
    if (R < 4) // is valid row found ?
    {
        PORTD = key[R][C]; //Display on LEDs
        serialTX(keypad[R][C]); // then send
        //char
        while (PORTD != 0xF0)
        PORTD = 0xF0; //Wait for release
        INTCONbits.RDIF = 0; //Reset flag
    }
}
```

```

}

#pragma code hi_Priority_int = 0x0008
//high priority IVT
void hi_Priority_int (void)
{
    _asm
        Gotolow_Priority_isr
    _endasm
}

#pragma code
#pragma interrupt low_Priority_isr
//low priority IVT
void low_Priority_isr (void)
{
    if (INTCONbits.RDIF == 1) // Is RDIF = 1 ?
        IF_ISR (); //If yes, call the ISR
}

#pragma code
void main ()
{
    TRISB = 0; // Set Port B an output port
    INTCON2bits.RDPU = 0;
    //Enable PORTD pull up resistors
    TRISD = 0xF0;
    PORTD = 0xF0; //Set return and scan lines
    while(1)
    {
        while (PORTD != 0xF0);
        // Wait for key to be pressed.
        TXSTA = 0x20;
        RCSTAbits.SPEN = 1;
        //Enable serial port
        SPBRG = 15;
        // Set baud rate at 9600
        TXSTAbits.TXEN = 1;
        //Transmission Enable
        INTCONbits.RDIE = 1;
        //Enable PORTD interrupt on change
        INTCONbits.GIE = 1; //enable interrupts
        //globally
    }.
}

```

Ex. 7.2.2: Explain step wise procedure and design methodology of PIC test board.

Soln.:

A PIC test board can be made using simple components like LED, switches, relays etc.

The board is to be made and the PIC microcontroller with a sample program can be connected to the test board. If the function is as per the program then the PIC microcontroller can be confirmed to be in good condition.

A sample program with test board connections is shown in the following example. In this case LED, keypad, buzzer and relay are connected to the PIC microcontroller. A program is written to test the microcontroller. The program is for switching on the LED if switch s1 is pressed and switching on the buzzer if switch s2 is pressed.

```

#include <P18F458.h>

#define s1 PORTDbits.RD4
#define s2 PORTDbits.RD5
#define BUZZER PORTBbits.RB1
#define LED PORTBbits.RB0

void main(void)
{
    TRISB = 0X00; //Port A pins initialised as output
    //port
    TRISD = 0xFF; //Port D pins initialised as input
    //port
    while(1)
    {
        if(s1==0) LED=0 else LED=1;
        if(s2==1) BUZZER=1 else BUZZER=0;
    }
}

```

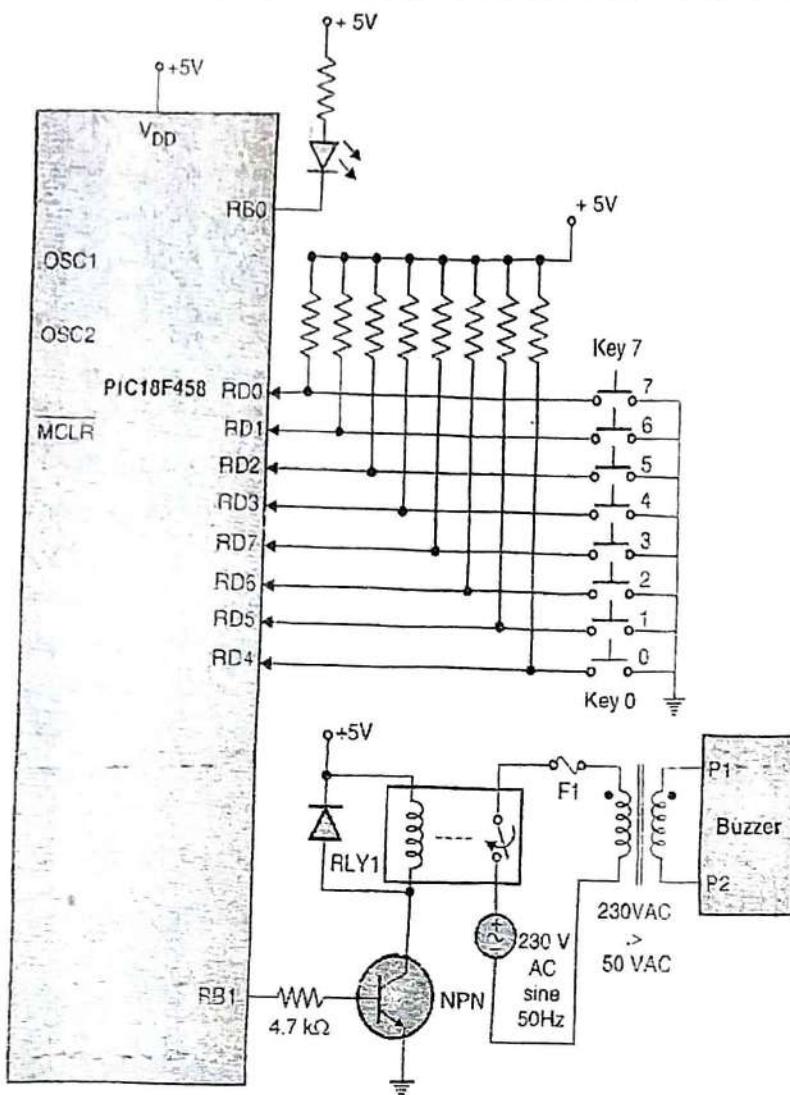


Fig. P. 7.2.2

7.3 Interfacing of Switches

- DIP (Dual In Package) switches is an IC package with usually four or eight switches. Fig. 7.3.1 shows an eight-input DIP package connected to an input port (Port B in this case). The port B has to be configured as input port in order to make the circuit to work.

When a switch is open, the corresponding pin is pulled up to V_{cc} internally and hence will be read as 1. When a switch is closed, the corresponding pin will be pulled to low by the external connection to ground as shown in Fig. 7.3.1 and hence be read as 0.

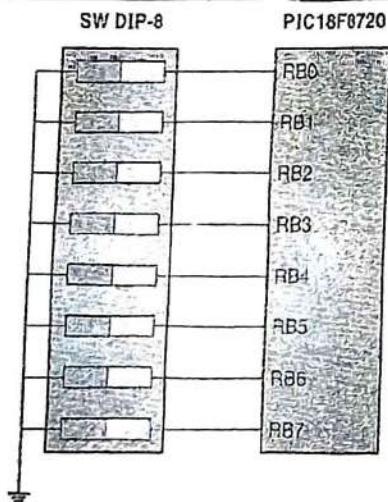


Fig. 7.3.1 : Connecting a set of eight DIP switches to port B of the PIC18F8720



7.4 Display

- 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.

7.4.1 LED Displays

University Question

Q. Draw a neat diagram of interfacing an LED with PIC microcontroller. **SPPU - Dec. 16, 4 Marks**

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 PIC18F458, through a resistor as shown in Fig. 7.4.1.
- This method requires PIC18F458 to source a huge amount of current required by the LED i.e. around 20 mA. But PIC18F458 is not capable of sourcing a current more than 2 mA. This will make the LED glow very dim.

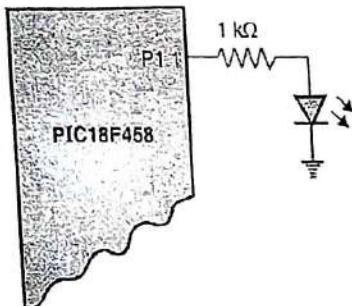


Fig. 7.4.1 : LED driven by PIC18F458 port pin (Method 1)

Method 2:

Connect the anode of LED to V_{cc} through resistor. Connect the cathode of LED to the port pin of PIC18F458 as shown in the Fig. 7.4.2.

This method requires PIC18F458 to sink a huge current required by LED i.e. 20 mA. PIC18F458 can sink huge currents and hence it makes LED glow brighter. Hence we will always use method 2, to interface LED.

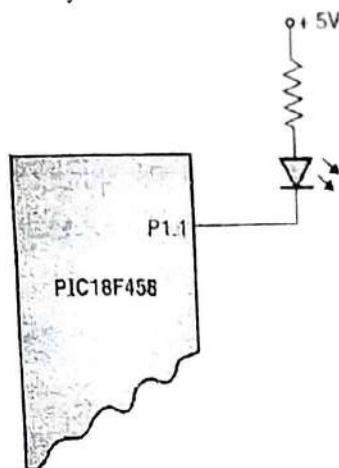


Fig. 7.4.2 : LED driven by PIC18F458 port pin (Method 2)

Note: Source current is current to be sourced i.e. given or provided. Sink current is the current to be sunk 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 simplest output device that can be interfaced to a microcontroller is an LED. The current required to light an LED may range from 10 mA to 20 mA.
- The voltage drop across the LED when it is forward biased can range from about 0.7 V to 2.2 V. A PIC18 pin can drive an LED directly.
- A resistor of 330Ω to 470Ω is required to limit the current flow. A typical circuit connection is shown in Fig. 7.4.3.

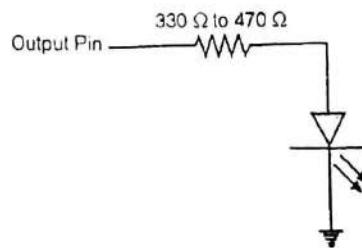


Fig. 7.4.3 : PIC18 output port pin driving an LED

Program 7.4.1 : Write a PIC18 assembly program to generate a square wave of 1 kHz.

Soln. :

To implement a square wave of 1 kHz, we need a delay of 500 μ s. We will toggle the port pin after every 500 μ s, hence the total clock period will be 1ms.

This program generates the square wave on all the pins of port B.

Label	Instruction	Comment
	cnt set 0x00	
	org 0x00	
	GOTO start	
	org 0x08	
	RETFIE	
	org 0x18	
	RETFIE	
start:	CLRF TRISB,A	
	MOVLW 0x00	
	MOVWF PORTB	
back:	MOVLW 0xFF	
	XORWF PORTB,I	
	MOVLW D'250'	
	MOVWF cnt, A	
again:	NOP	17 instruction cycle, 1 for each NOP
	NOP	

Label	Instruction	Comment
	NOP	
	DECFSZ cnt,F,A	1 instruction cycle (2 when [ent] = 0)
	BRA again	2 instruction cycle
	BRA back	
	End	

Ex. 7.4.1(A) Eight LED are connected to port A. Write a program which will connect to port A. Assume delay subroutine written at 0x30H. **SPPU - Dec. 15, 8 Marks**

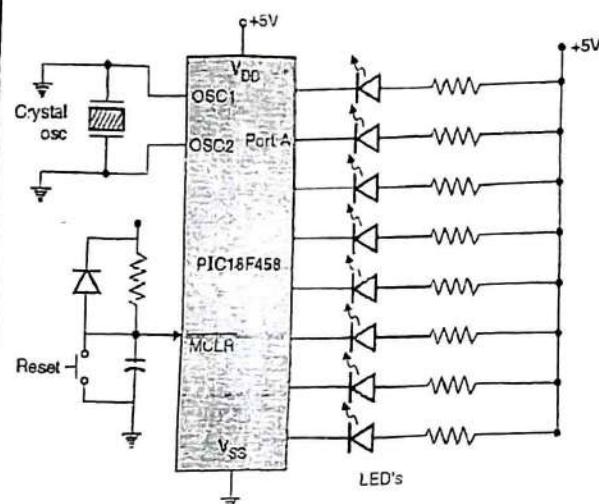
Soln. :

Fig. P.7.4.1(A)

Assembly program

Label	Instruction	Comments
	MOVLW 00H	WREG = 00H
	MOVWF TRISA	Set PORT A an output port
again:	MOVLW 0xFFH	Turn on LED
	MOVWF PORTA	
	CALL DELAY	Delay
	MOVLW 0x00H	Turn off LED (blink LED)



Label	Instruction	Comments
	MOVWF PORTA	
	CALL DELAY	Delay
	GOTO again	Repeat

Program 7.4.2: Write a PIC18 assembly program to blink a LED.

SPPU - Dec. 16, 4 Marks

Soln.: The same program as in Program 7.4.1, when replaced with a delay of 500ms instead of 500μs, will work to blink an LED connected on any pin of port B.

Label	Instruction	Comments
	cnt Set 0 x 00	
	ent 1dry 0 x 01	
	cnt2 set 0 x 00	
	org 0 x 00	
	GOTO start	
	org 0 x 08	
	RETFIE	
	org 0 x 18	
	RETFIE	
Start :	CLRF TRISB, A	
	MOVLW 0 x 00	
	MOVWF PORTB	
back :	MOVLW 0 x FF	
	XORWF PORTB, I	
	MOVLWD '250'	
	MOVWF cnt1, A	
again1:	MOVLWD '4'	
	MOVWF cnt2, A	
again2:	MOVLWD '250'	
	MOVWF cnt, A	

Label	Instruction	Comment
Again :	NOP	17 instruction cycle, 1 for each NOP
	NOP	
	BRA again	2 instruction cycle
	DECFSZ ent2, F, A	1 instruction cycle (2 when [cnt] = 0)
	BRA again2	
	DECFSZ ent1, F, A	
	BRA again1	
	BRA back	
	end	

Program 7.4.3 : Write an assembly language program for an interface shown in Fig. P.7.4.3. Read an 8-bits data through DIP-switch connected on PORTB and output the data to bit-0 of PORTC where an LED is connected. The clock frequency of the controller is 40 MHz and the data transfer rate at PORTC is 100 milliseconds.

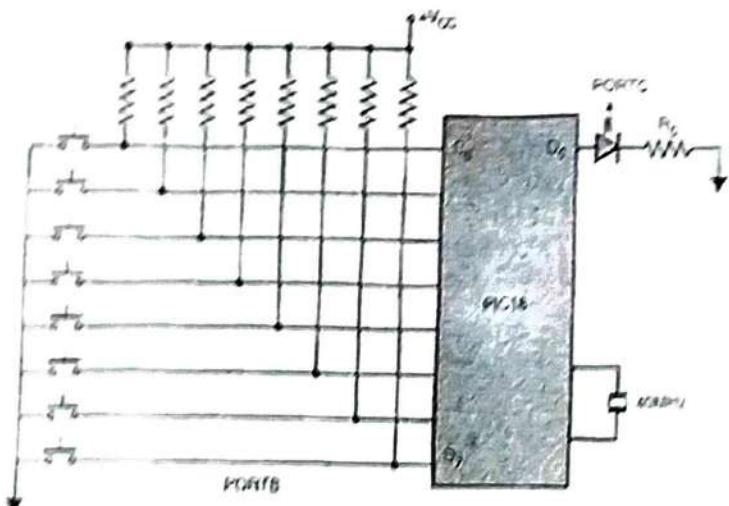


Fig. P. 7.4.3

Soln. :

Label	Instruction	Label	Instruction
dat : set 0 x 00		nxt :	MOVWF cnt2, A
cnt : set 0 x 01			MOVLW D'200'
cnt1 : set 0 x 02			MOVWF cnt1, A
cnt2 : set 0 x 03		back :	MOVLW D'250'
org 0 x 00			MOVWF cnt, A
goto start		again :	NOP
start :	BSE INTCON2, RBP2, A		NOP
	MOVLW 0 x FF		NOP
	MOVWF TRISB, A		NOP
	CLRF TRISC, A		NOP
repeat :	MOVE portB, W, A		NOP
	MOVWF portC, A		NOP
	MOVWF dat		NOP
	MOVLW D'8'		NOP

Label	Instruction
	NOP
	DECFSZ cnt2, F, A
	BRA next
	DECFSZ cnt1, F, A
	BRA back
	RRNCF dat, w, a
	MOVWF data
	MOVWF portC, A
	DECFSZ cnt2, F, A
	BRA next
	BRA repeat
	end

Program 7.4.4 : Write an assembly language program for an interface shown in Fig. P.7.4.4. The interface is to implement a 3 x 3 bit calculator. The calculator has a two bit command interpreter as given in Table P.7.4.4 to select different arithmetic operations. Display the results on LEDs connected on PORTC.

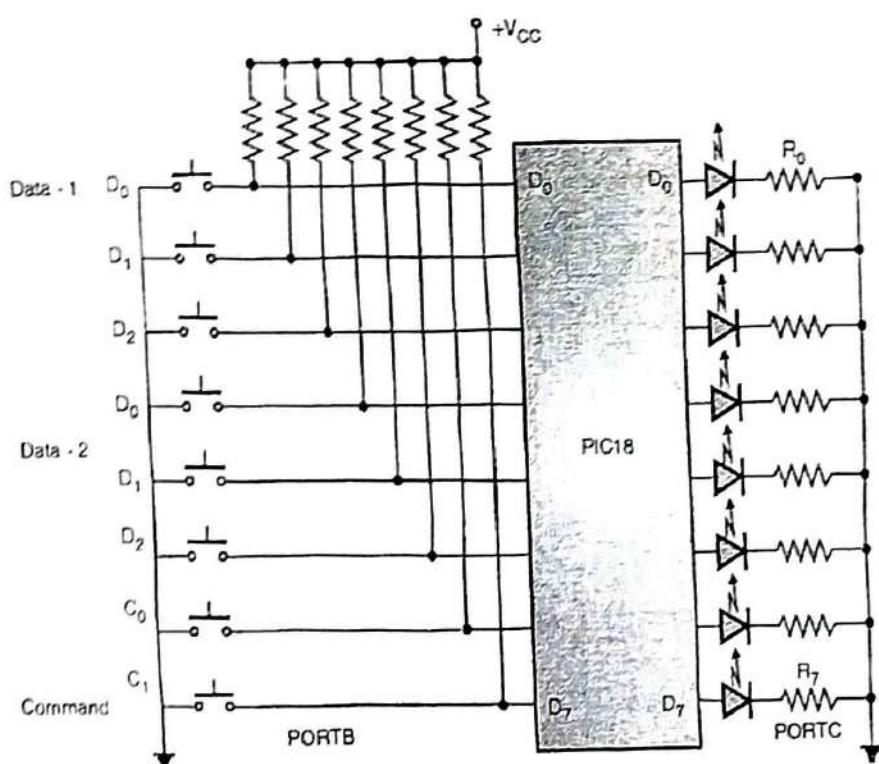


Fig. P.7.4.4



Table P.7.4.4

C0	C1	Case-1
0	0	Addition
0	1	Subtraction
1	0	Multiply
1	1	Divide

Soln.:

Label	Instruction
	n01 set 0x00
	n02 set 0x01
	opn set 0x02
	mask1 equ 0x07
	mask2 equ 0x38
	mask3 equ 0xC0
	res set 0x03
	res1 set 0x04
start :	BSF INTCON2, RBP2, A
	MOVLW 0xFF
	MOVWF TRISB, A
	CLRF TRISC, A
repeat :	MOVF portB, W, A
	MOVWF opn, A
	MOVLW mask1
	ANDWF opn,W, A
	MOVWF n01, A
	MOVLW mask2
	ANDWF opn,W, A
	MOVWF n02, A
	MOVLW mask3
	ANDWF opn,F, A
	MOVF opn,W, A
	CPFSEQ 0x00
	BRA next
	MOVF n01,W, A
	ADDWF n02 ,W, A

Label	Instruction
	MOVWF res,A
	BRA repeat
next :	CPSEQ 0x01
	BRA next1
	MOVF n01,W, A
	SUBWF n02,W, A
	MOVWF res, A
	BRA repeat
next1 :	CPSEQ 0x02
	BRA next2
	MOVF n01,W, A
	MULWF n02, a
	MOVFF PRODH, res
	MOVFF PRODL, res l
	BRA repeat
next2 :	MOVLW 0x00
	MOVWF res, a
	MOVF n01,W, A
again :	CPFSLT n02, a
	BRA over
	SUBWF n02,W, A
	INCF res, f, a
	BRA again
over :	MOVWF res1, a
	BRA repeat.
	end

Program 7.4.5: 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.

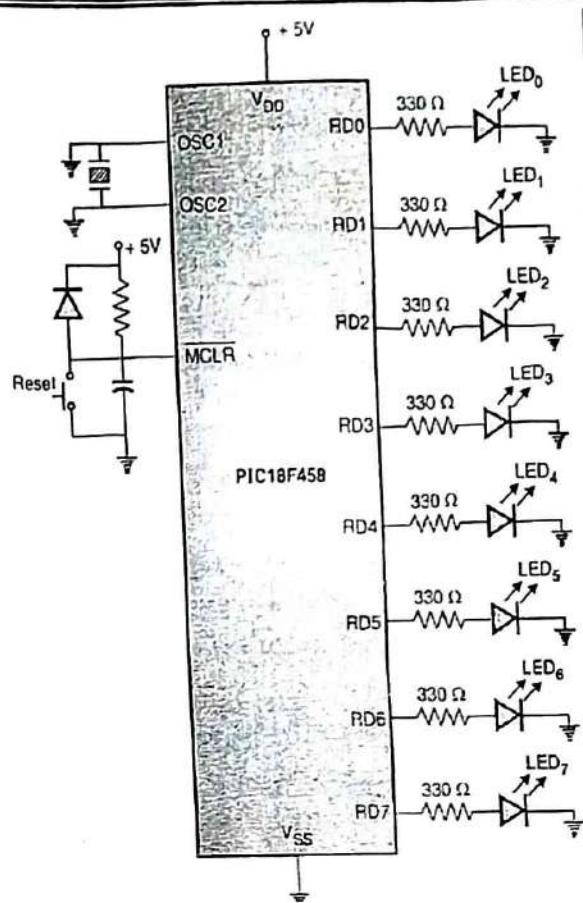
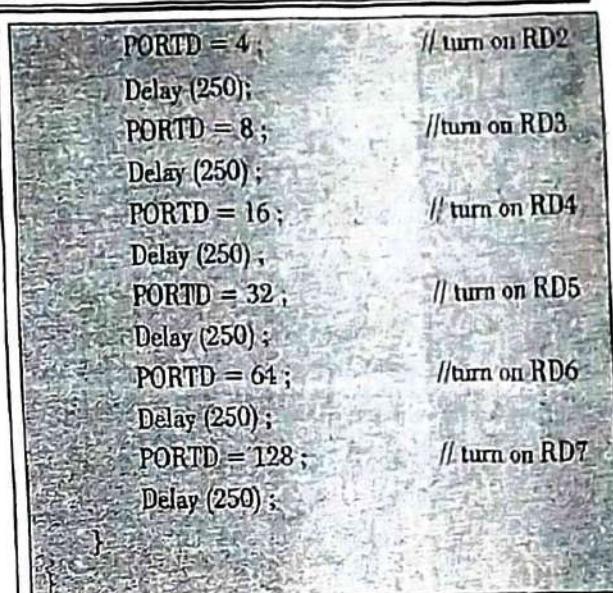


Fig. P. 7.4.5

Soln.:

```
# include < P18F458.h>
void Delay (unsigned int x)
{
    unsigned int i;
    unsigned char j ;
    for(i = 0 ; i < x ; i++)
        for(j = 0 ; j < 165 ; j++);
}

void main (void)
{
    TRISD = 0; //Port D is output port
    while(1)
    {
        PORTD = 1; // turn on RD0
        Delay (250);
        PORTD = 2; //turn on RD1
        Delay (250);
    }
}
```



Program 7.4.6: Draw an interfacing diagram of LED connected to Port B of PIC18F and write program embedded C program for Ring Counter.

Soln.:

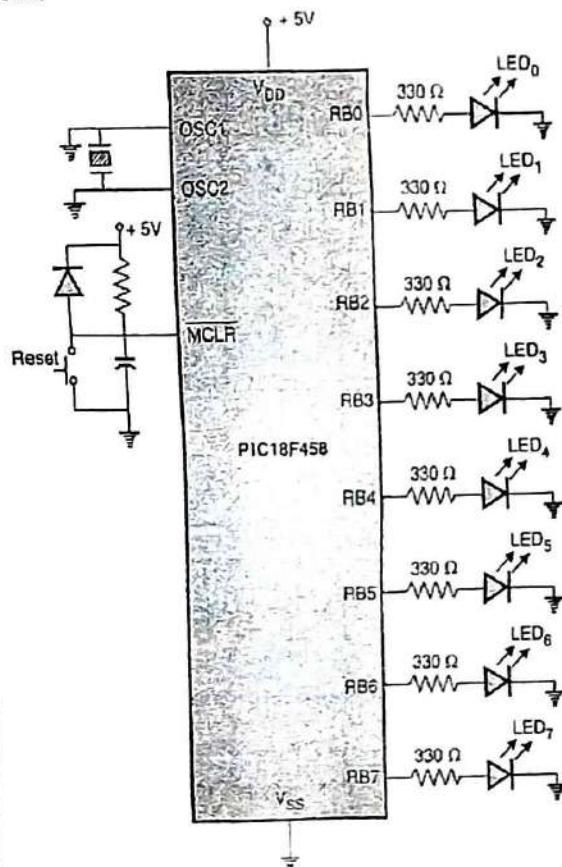


Fig. P. 7.4.6



```

#include < P18F458.h >

void Delay (unsigned int x)
{
    unsigned int i ;
    unsigned char j ;
    for (i = 0 ; i < x ; i++)
        for (j = 0 ; j < 165 ; j++) ;

}

void main (void)
{
    TRISB = 0; //Port B is output port
    while (1)
    {
        PORTB = 1; //turn on RB0
        Delay (1000);
        PORTB = 2; //turn on RB1
        Delay (1000);
        PORTB = 4; // turn on RB2
        Delay (1000);
        PORTB = 8; //turn on RB3
        Delay (1000);
        PORTB = 16; //turn on RB4
        Delay (1000);
        PORTB = 32; // turn on RB5
        Delay (1000);
    }
}

```

```

PORTB = 64; //turn on RB6
Delay (1000);
PORTB = 128; // turn on RB7
Delay (1000);
}

```

7.4.2 Seven Segment Display (SSD)

- Seven-segment displays are often used when the embedded product needs to display few digits (upto 8 or 16) and few letters. Although a PIC18F8720 microcontroller has enough current to drive a seven-segment display, a pull up resistor is advisable as there may be some more I/O devices in the system. In Fig. 7.4.4 port D drives a common-cathode seven-segment display through the buffer chip 74HC244.

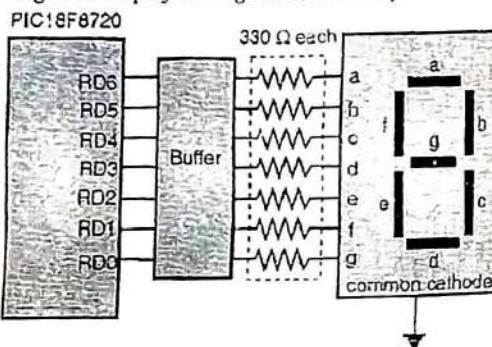


Fig. 7.4.4 : Driving a signal seven-segment display

- When an application needs to display multiple BCD digits, the time-multiplexing technique is used. In this method, each display glows for some time and all others are off i.e. only one display is on at any given instant. But because of persistence of vision, a human feels as if all the displays are glowing together. Hence it is necessary that the off time should be much less than the persistence of vision time.
- Fig. 7.4.5 shows the common cathode type of seven-segment display connected to the collector of the NPN transistor. When a pin of port B is logic high, the connected NPN transistor will be driven into the saturation region, allowing the display to be lighted.
- A 2N2222 power transistor can sink from 100 mA to 300 mA of current, while the maximum current that flows into the common cathode is about 70 mA (7 segments \times 10 mA).

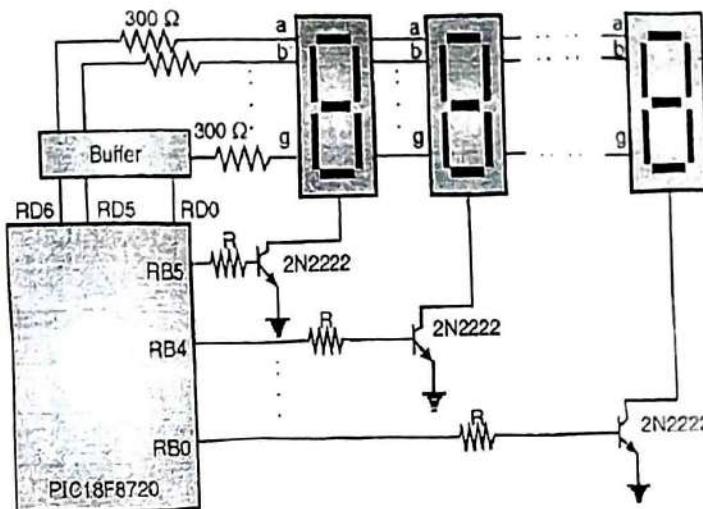


Fig. 7.4.5 : Port B and Port D together drive six seven-segment displays

Program 7.4.7 : Write an instruction sequence to display 5 on

- (a) The seven-segment display shown in Fig. 7.4.4.
- (b) On LSB the seven-segment display shown in Fig. 7.4.5.

Soln.:

- (a) To display 5 on the seven segment display (shown in Fig. 7.4.4) the following things need to be done
 1. Configure Port D as output port
 2. Output the hex value for displaying 5 (0x5B) on port D

The following program implements this algorithm:

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	
	RETIE	
	org 0x18	
	RETIE	
start:	CLRF TRISD,A	configure Port D for output
	CLRF TRISB,A	configure Port B for output
	MOVLW 0x5B	
	MOVWF PORTD	output the segment pattern of 5
	MOVLW 0x01	
	MOVWF PORTB	select the LSB seven segment display
	end	

- (b) To display the value 5 on the LSB seven-segment display (shown in Fig. 7.4.5) the following things need to be done:

1. Configure both port B and port D for output
2. Drive the RBO pin to high

3. Drive the RB4 ... RBO pins to low
4. Output the hex value 0x5B to port D

The following program implements this algorithm

Label	Instruction	Comment
	org 0x00	
	GOTO start	
	org 0x08	
	RETIE	
	org 0x18	
	RETIE	
start:	CLRF TRISD,A	configure Port D for output
	CLRF TRISB,A	configure Port B for output
	MOVLW 0x5B	
	MOVWF PORTD	output the segment pattern of 5
	MOVLW 0x01	
	MOVWF PORTB	select the LSB seven segment display
	end	

Program 7.4.8 : Interface two common-anode seven segment LEDs to PORTB and PORTC of 18F microcontroller and write instructions to design an up-counter counting from 00 to 59 at the interval of 100 ms and display the count at two seven-segment LEDs.



Soln.:

Label	Instruction
	count set 0x00
	cnt1 set 0x01
	cnt set 0x02
	org 0x00
	GOTO start
	org 0x08
	RETFIE
	org 0x18
	RETFIE
	org 0x50
	array dB 0x03, 0x9F, 0x25, 0x0D, 0x99, 0x49, 0x41, 0x1F, 0x01, 0x09
	org 0x100
start:	CLRF TRISB, A
	CLRF TRISC, A
	MOVLW upper array
	MOVWF TBLPTRU, A
	MOVLW high array
	MOVWF TBLPTRH, A
	MOVLW low array
	MOVWF TBLPTRL, A
	CLRF count, A
next:	MOVF count, W, A
	ANDLW 0x0F
	ADDWF TBLPTRL, f, A
	DECFSZ count, F, A
	MOVF TABLAT, W, A
	MOVWF PORTC, A
	MOVLW low array
	MOVWF TBLPTRL, A
	SWAPF count, W, A
	ANDLW 0x0F
	ADDWF TBLPTRL, f, A
	DEC TBLPTRL

Label	Instruction
	MOVF TABLAT, W, A
	MOVWF PORTB, A
	MOVLW low array
	MOVWF TBLPTRL, A
	MOVLW D'200'
	MOVWF cnt1, A
back:	MOVLW D '250'
	MOVWF cnt, A
again:	NOP
	DECFSZ cnt, F, A
	BRA again
	DECFSZ cnt1, F, A
	BRA back
	INCF count
	MOVLW D '59'
	CPFSEQ count, a
	BRA next
	end

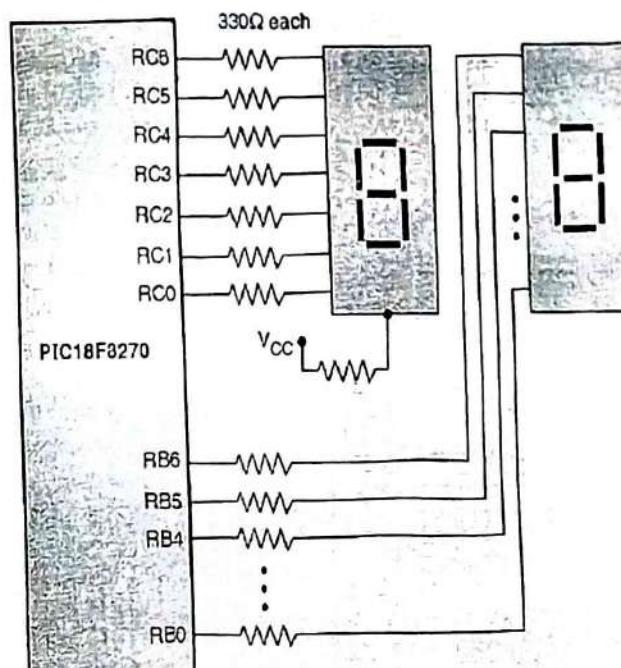


Fig. P.7.4.7

~~7.5~~ **Interfacing of PIC18F458 8 bit Model LCD (16 x 2)**

University Questions:

Q. Explain the functions of pins associated with LCD (16 x 2). **SPPU - Dec. 14, Dec. 15, 8 Marks**

Q. Explain the function of following pin associated with 16 X 2 LCD controllers: RS. **SPPU - May 15, May 16, 2 Marks**

Q. Explain the function of following pin associated with 16 X 2 LCD controller: R/W. **SPPU - May 15, May 16, 2 Marks**

Q. Explain the function of following pin associated with 16 X 2 LCD controllers: E. **SPPU - May 15, 2 Marks**

Q. Explain the function of EN pin in detail. **SPPU - May 16, 2 Marks**

Q. Explain the function of following pin associated with 16 X 2 LCD controllers: DB0-DB7. **SPPU - May 15, 2 Marks**

Q. Explain the functions of following pins of LCD (16x2). **SPPU - Dec. 17, 8 Marks**

- (i) Register select (RS)
- (ii) Read/Write (R/W)
- (iii) Enable (E)
- (iv) VEE

Q. List the steps for reading Busy flag and explain following pins of LCD (16 x 2).

- (i) Register select (RS)
- (ii) Enable (E)

SPPU - May 18, 8 Marks

Q. Draw an interfacing diagram of LCD (16 x 2) with PIC18 microcontroller and explain the functions of various pins of LCD. **SPPU - Dec. 18, 8 Marks**

Q. Explain 8-bit mode of LCD Interfacing with PIC18F458 with the help of suitable diagram. **SPPU - Dec. 19, 8 Marks**

⊖ Seven-segment displays are bulky and can display less number of characters. When many characters are to be displayed the persistence of vision used to perform time multiplexing may not work.

⊖ Liquid crystal displays (LCDs) are small and can display huge number of characters.

⊖ It also has some more advantages like low power consumption, small size and ability to display both characters and graphics.

- The basic construction of an LCD is shown in Fig. 7.5.1. The LCD's allow light to pass through it when activated. A segment is activated by the application of a low-frequency bipolar signal in the range of 30 Hz to 1000 Hz.

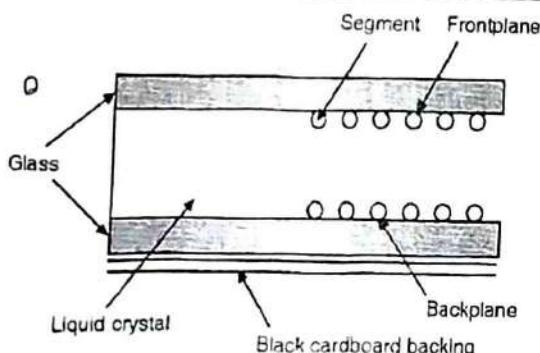


Fig. 7.5.1: A liquid crystal display (LCD)

- The Hitachi HD44780 is one of the most popular LCD display controllers. The Fig. 7.5.2 explains the interfacing of this LCD with the PIC18F8720.

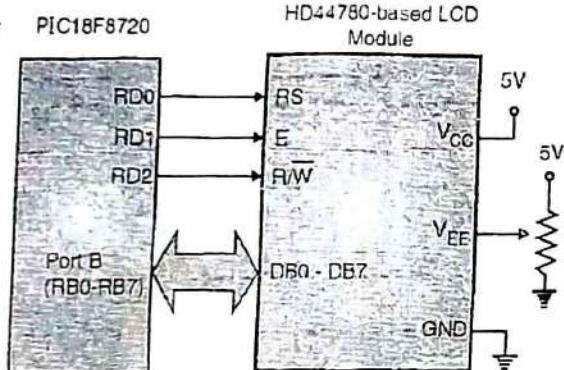


Fig. 7.5.2: LCD interface example (8-bit bus)

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. 7.5.3.

- 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 debugged.

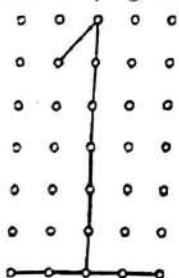
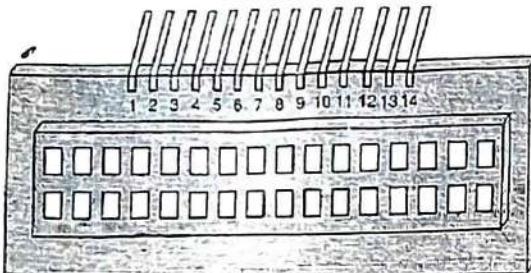


Fig. 7.5.3: Displaying 1 on LCD display of a 5x7 dot matrix

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.

A typical 16 by 2 (i.e. 16 characters and 2 such lines) LCD looks as shown in Fig. 7.5.4.

Fig. 7.5.4: Structures of 16×2 LCD

Some LCD have their pins on left or on bottom. The functions of the pins of LCD are listed in the Table 7.5.1.

Table 7.5.1: Pin description of LCD

Pins	Symbol	Functions...
1	V_{ss}	Ground
2	$V_{ddor}(V_{cc})$	+ 5 V supply
3	V_0	Contrast voltage
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)

- There are two registers of LCD viz. Instruction command code register and data register.

- The RS pin is used to select one of these register.
- R/W pin is used to read or write to LCD.
- 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.
- The list of commands that can be given to the LCD are as listed in the Table 7.5.2.

Table 7.5.2: 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 1 st line
0xC0	Move cursor to beginning of 2 nd line
0x38	Initialize 2 line display of 5x7 matrix

7.5.1 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/W = '0', to indicate write.
- Make E = '1' To give a high-to-low pulse on E pin so as
- Wait for 120 μ sec.
- 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/W = '0', to indicate write
- Make E = '1' To give a high-to-low pulse on E pin so as
- Wait for 120 μ sec.
- 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 PIC18F458 as input port.
- Make RS = '0' to indicate instruction.
- Make R/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.



7.5.2 Interfacing LCD Module with PIC18F458

University Questions

Q. Draw a neat diagram of interfacing of 16x2 LCD with PIC18F458 microcontroller in 8 bit mode. Assume suitable port pins for interfacing.

SPPU - May 16, 4 Marks

Q. Draw and explain LCD interfacing with pic18f458.

SPPU - May 19, 9 Marks

- Fig. 7.5.5 shows the interfacing of a 16 character \times 2 line LCD modules with the microcontroller PIC18F458. The data lines are connected to Port 1 of PIC18F458. The control lines RS, R/W and E are driven by P 3.2, P 3.3 and P 3.4. The voltage at V_{EE} pin is adjusted by potentiometer to adjust contrast of LCD.
- The display can be controlled by giving appropriate command codes to the LCD module. The command codes can be used to display or force the cursor to home position or blink cursor.

Table 7.5.3 shows the command codes.

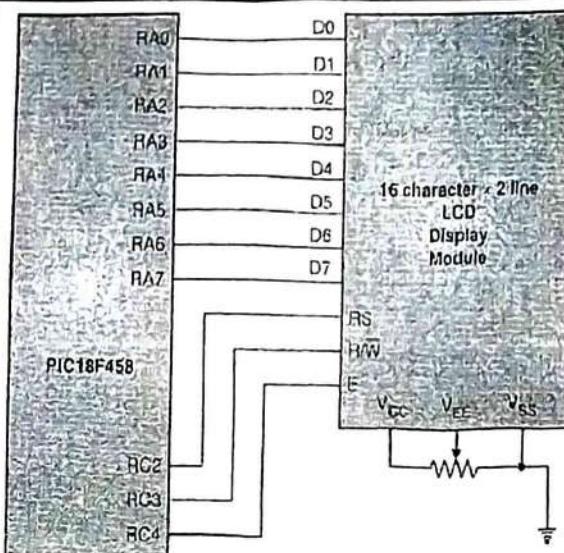


Fig. 7.5.5: Interfacing 16 \times 2 LCD to PIC18

Table 7.5.3

Commands	RS R/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 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 1/D = 1 increment 1/D = 0 decrement S = 1 accompanies display shift	It sets the direction for moving the cursor and specifies the shift of display. These operations are done during data read and data write.
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

Commands	RS R/W DB ₇ DB ₆ DB ₅ DB ₄ DB ₃ DB ₂ DB ₁ DB ₀	Description
Function Set	0 0 0 0 1 DL N F - - DL = 1, 8 bits F = 0:5 x 7 dots DL = 0, 4 bits F = 1:5 x 10 dots N = 1 2 lines N = 0 1 line	It sets interface data length (DL), number of data lines (L) and character font (F).
Set CG RAM address (character generator RAM)	0 0 0 1 ACG (ACG : CG RAM address)	Sets the character generator RAM address. The character generator RAM data is sent and received once this setting is done.
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.
Read Busy Flag and Address	0 1 BF AC AC : address counter for CG and DD RAM address BF = 0 can accept command or instruction BF = 1 busy in Internal operation	Reads busy flag indicating if internal operation is being done and reads address counter contents.
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.

- To display a message on the LCD module, it is essential to initialize the LCD module by writing a series of command codes in the command register in the appropriate sequence. The initialization includes command codes for clearing display, returning home and shifting cursor automatically after a character is written.
- Once the initialization is completed we can read/write data to/from CG RAM or DD RAM. The DD RAM stores characters in ASCII code whereas CG RAM stores characters in internally generated character code.

Ex. 7.5.1.: Interface 2-line, 16 character LCD display to PIC18F458. Use only one port. Write assembly language program to display message on line 2 of LCD.

OR

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.

SPPU - Dec.14, 4 Marks, Dec. 15, 8 Marks

OR

Draw and explain the interfacing of LCD with Port D and Port E of PIC18F xxx microcontroller. Write C code to display "WELCOME".

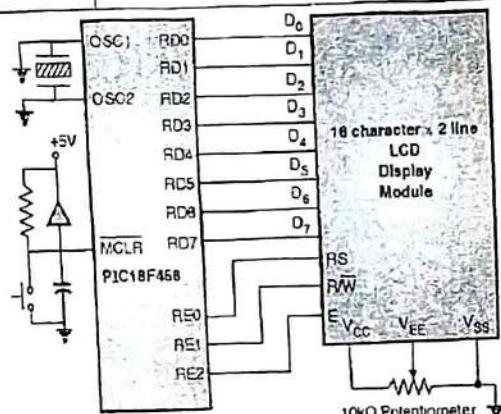


Fig. P. 7.5.1: Interfacing 16 x 2 LCD to PIC18

Soln. : C Program

```
# include <P18F458.h>
# define RS PORTEbits.RE0 //RS = RE0
# define RW PORTEbits.RE1 //R/W = RE1
# define EN PORTEbits.RE2 //EN = RE2
```



```

char msg[9] = "WELCOME";
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};
void MSDelay(unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++);
}

void CMD(unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay(250);
    EN = 0;
}

void DAT(unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    EN = 1;
    MSDelay(250);
    EN = 0;
}

void main(void)
{
    int i;
    TRISE = 0; // Make port E an output port
    TRISD = 0; // Make port D an output port
    EN = 0;
    Delay(250);
    for (i = 0; i <= 4; i++)
    {
        CMD(cmds[i]),
}

```

```

        MSDelay(250);
    }
    for (i = 0; i <= 6; i++)
    {
        DAT(msg[i]);
        MSDelay(250);
    }
    while (1);
}

```

Ex. 7.5.2: 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'

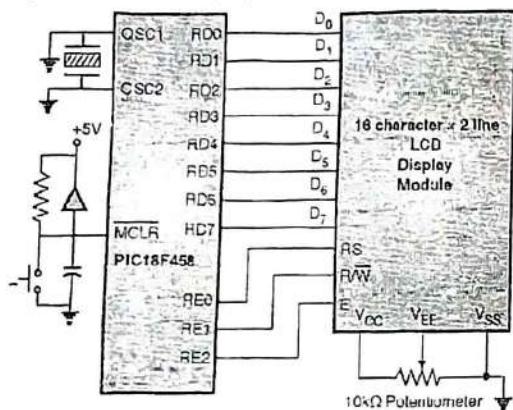


Fig. P. 7.5.2: Interfacing 16 x 2 LCD to PIC18

Soln.:

C Program

```

#include <P18F458.h>
#define RS PORTEbits.RE0 //RS = RE0
#define RW PORTEbits.RE1 //R/W = RE1
#define EN PORTEbits.RE2 //EN = RE2
char msg[13] = "S.P.P.U Pune";
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};
void MSDelay(unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
}

```

```

for (j = 0; j < 165; j++) {
}

void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}

void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}

void main (void)
{
    int i;
    TRISE = 0; // Make port E an output port
    TRISD = 0; // Make port D an output port
    EN = 0;
    Delay (250);
    for(i=0;i<=4;i++)
    {
        CMD (cmds[i]);
        MSDelay (250);
    }
    for(i=0;i<=11;i++)
    {
        DAT (msg[i]);
        MSDelay (250);
    }
    while(1);
}

```

Ex. 7.5.3 : 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.

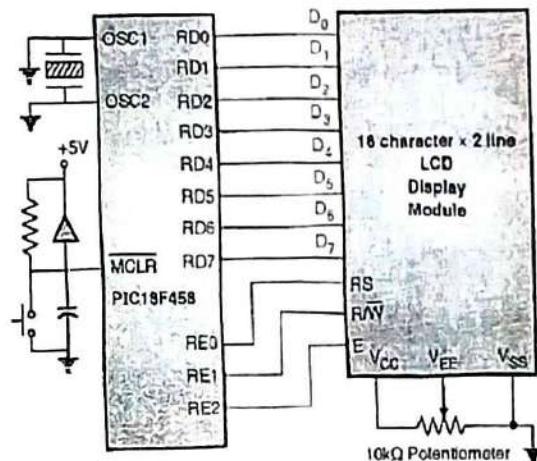


Fig. P. 7.5.3: Interfacing 16×2 LCD to PIC18

Soln.:

C Program

```

#include <P18F458.h>
#define RS PORTEbits.RE0 //RS = RE0
#define RW PORTEbits.RE1 //R/W = RE1
#define EN PORTEbits.RE2 //EN = RE2
char msg[11] = "My College";
char cmds[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};

void MSDelay (unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++);
}

void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}

```



```

EN = 0;
}

void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}

void main (void)
{
    init();
    TRISE = 0; // Make port E an output port
    TRISD = 0; // Make port D an output port
    EN = 0;
    Delay (250);
    for(i=0;i<4;i++)
    {
        CMD (cmd[i]);
        MSDelay (250);
    }
    for(i=0;i<=9;i++)
    {
        DAT (msg[i]);
        MSDelay (250);
    }
    while(1);
}

```

Ex. 7.5.4: Design a frequency counter for counting number of pulses and display same on LCD.

OR

Design frequency counter for the range from DC to 5 MHz frequency using PIC18FXXX. Design and draw interfacing circuit. Also explain required flowchart

Soln.:

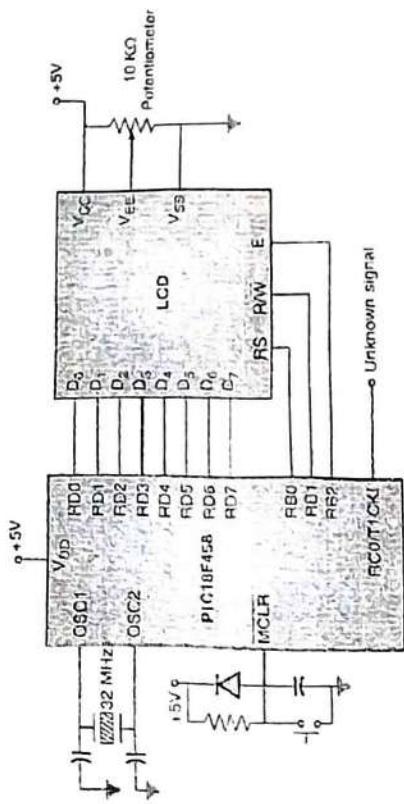


Fig. P. 7.5.4: Interfacing diagram for frequency counter with display on LCD

A 1 second delay can be created by executing a delay of 100 msec, 10 times.

Program:

```

#include <P18F458.h>
unsigned int i = 0, toverflow cnt;
unsigned long int freq;
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define EN PORTBbits.RB2
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++)
{
    TMROH = 0x3C; // Put 15535 in Timer 0
    //high byte and low byte so that it will roll over
    //in 50000 clock cycles
    TMR0L = 0xAF
    INTCONbits.TMROIF = 0; // Clear TMROIF
    //flag
    while (! (INTCONbits.TMROIF));
    // wait till Timer 0 rolls over
}
}

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
    }
}

void low_ISR (void) // ISR of low priority interrupt
{
    _asm
    retfie 0          //Return if not Timer 0 interrupt
    _endasm
}

# pragma code hi_Priority_int = 0x0008//high priority
IVT

void hi_Priority_int (void)
{
    _asm
    GOTO high_isr
    _endasm
}

# pragma code

```

```

#pragma code low_vector = 0x1811//low priority IVT

{
    _asm
    GOTO low_ISR
    _endasm
}

#pragma code
void MSDelay (unsigned int x)
{
    unsigned int i, j;
    for (i = 0 ; i < x ; i++)
    for (j = 0; j < 165 ; j++);
}

void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    E = 1;
    MSDelay (250);
    E = 0;
}

void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    E = 1;
    MSDelay (250);
    E = 0;
}

void main (void)
{
    char t0_count, temp, templ;
    unsigned char xl, y, z;
    int j;
    for (j = 0 ; j < 5000 ; j++);
    //wait for sometime for LCD to stabilize on power up
    for(j=0;j<=4;j++)

```

```

CMD (cmds[i]) ;
MSDelay (250) ;
}

toverflow_cnt = 0 ;
// initialize Timer 1 overflow count to 0.

freq = 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 T1CK1 and prescaler 1
INTCON = 0xC0 ; //Enable global and
//peripheral interrupts
delay (10) ; // Create one second delay and
//wait for interrupt
INTCONbits.CIE = 0 ; // Disable global
//interrupts.

temp = TMR1L ; // save frequency low byte
frequency = toverflow_count * 65536 +
TMR1H * 256 + temp ; // compute frequency
CMD(0x80);

temp1 = TMR1H ;
x1 = temp1 & 0x0F ; // mask upper 4 bits
z = x1 | 0x30 ; // make it ASCII
PORTD = z ;
DAT (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
DAT (z) ; // display the hundreds digit
temp = TMR1L;
x1 = temp & 0x0F ; // mask upper 4 bits
z = x1 | 0x30 ; // Make it ASCII
DAT (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

```

```

    DAT(z);           // display the units digit
    DAT(" ");         // display single space
    DAT("Hz");        // display unit of frequency Hz
    DAT("Z");
}

while(1);
}

```

Ex. 7.5.5: Design of DAS system for pressure monitoring system (use any suitable sensor)

Soln.:

ASCX30AN is a 0-30psi pressure transducer.

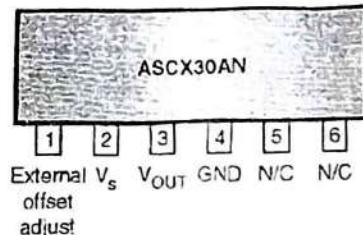


Fig. P. 7.5.5: ASCX30AN pin diagram

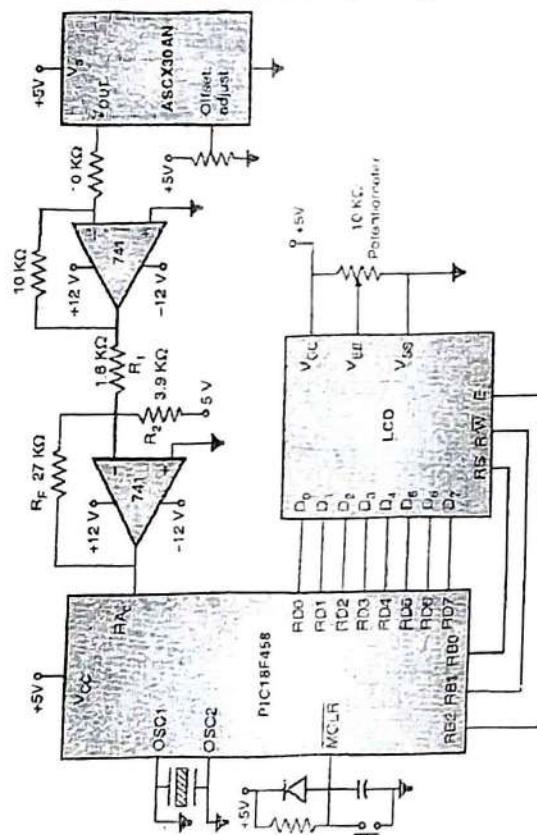


Fig. P. 7.5.5(a): Interfacing

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. 7.5.5(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. In order to find the barometric pressure we need to divide A/D result by 7.53V.

$$\text{Barometric pressure} = 948 + \frac{\text{A/D result}}{7.53}$$

$$= 48 + \frac{\text{A/D result} \times 100}{753}$$

Program

```
#include <P18F458.h>
unsigned int i, j;
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};
char unit[1] = "mbar";
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define EN PORTBbits.RB2
#define temp PORTAbits.RA0
#define Vref PORTAbits.RA3
#define busy PORTDbits.RD7
void MSDelay (unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++);
}
void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}
void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
```

```
EN = 1;
MSDelay (250);
EN = 0;
}
void main()
{
    unsigned char ADC_output, Lo_byte, Hi_byte,
    output;
    for(i=0;i<=4;i++)
    {
        CMD (cmds[i]);
        MSDelay (250);
    }
    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,  $\frac{f_{osc}}{64}$ 
    ADCON1 = 0xC5; //  $\frac{f_{osc}}{64}$ , 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 | Highbyte;
        output = 948 + (ADC_output * 100) / 7.53 ;
        PORTD = output ;
        DAT (output/100);
        DAT ((output % 100)/10);
        // Display pressure
```

```

DAT (output % 10) ;
for(j=0;j<=3;j++)
{
    DAT (unit[j]) ;
    MSDelay(250)
}
}

```

Ex. 7.5.6: 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 $+150^{\circ}\text{C}$ with an output of 10 mV for each degree centigrade. E.g. for 80°C temperature the output will be $80 \times 10 = 800$ mV.



Fig. P. 7.5.6: 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,

$$V_{out} = 10 \text{ mV} \times 1024 \text{ steps} = 10240 \text{ mV}$$

$$= 10.24 \text{ V}$$
 However, this much voltage is not acceptable.
 - If a step size of 2.5 mV is selected,

$$V_{out} = 2.5 \text{ mV} \times 1024$$

$$V_{out} = 2560 \text{ mV} = 2.56 \text{ V}$$
 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 $V_{ref} = 2.56 \text{ V}$.
 - The real temperature can be got by dividing the A/D output by 4.
 - Fig. P. 7.5.6(a) shows the interfacing diagram of PIC18F458 with LM35 temperature sensor.

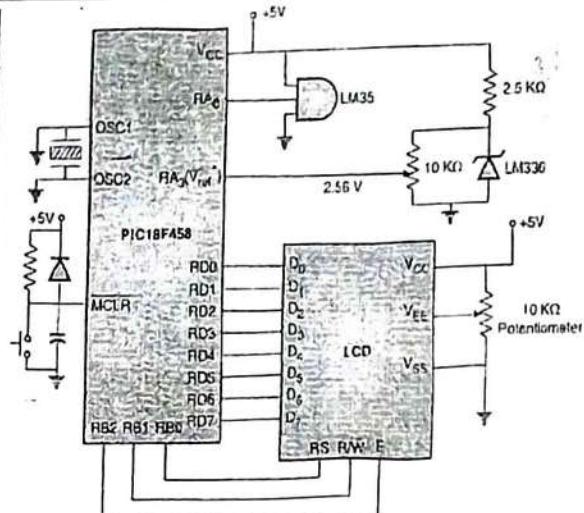


Fig. P. 7.5.6(a): Interfacing diagram of PIC18F458 based data acquisition system for measurement of temperature

- Table P. 7.5.6(a) gives the 10 bit output values for ADC and the temperature with $V_{ref} = 2.56$ V.

Table P. 7.5.6

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

Program

```
#include <P18F458.h>
unsignedinti = 0 ;
charcmds[5]={0x38,0x0E,0x01,0x06,0xC0};
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define EN PORTBbits.RB2
#define temp PORTAbits.RA0
#define Vref PORTAbits.RA3
#define busy PORTDbits.RD7
void MSDelay (unsigned int x)
{
    unsigned int i,j;
    for (i = 0 ; i<x ; i++)
        for (j = 0 ; j < 165 ; j++)
}
void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    E = 1;
    MSDelay (250);
    E = 0;
}
void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    E = 1;
    MSDelay (250);
    E = 0;
}
void main ()
{
    unsigned char ADC_output, Lo_byte, Hi_byte ;
    for(i=0;i<=4;i++)
    {
        CMD (cmds[i]);
        MSDelay (250);
    }
    TRISD = 0; //make port D an output port
    TRISAbits.TRISA0 = 1; //Make RA0 an input
    //pin
    TRISAbits.TRISA3 = 1; // RA3 = 1 for Vref
    //input
}
```

$\text{ADCON0} = 0x81$; //Channel 0, ADC on, $\frac{\text{fosc}}{64}$
 $\text{ADCON1} = 0xC5$; // $\frac{\text{fosc}}{64}$, 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|Highbyte ;
 PORTD = ADC_output ;
 DAT (ADC_output/100); //Display the
//temperature
 DAT ((ADC_output%100)/10);
 DAT(ADC_output % 10);
 DAT ("°");
 DAT ('C'); //unit of temperature is
// in °C
 CMD (0x80) ;
}

Ex. 7.5.7 : Draw an interfacing diagram of LCD 16x2 with PIC18FXXXX and write an embedded C program to display 'GST' on line one and 'INDIA' at 5th position on second line.

Soln.:

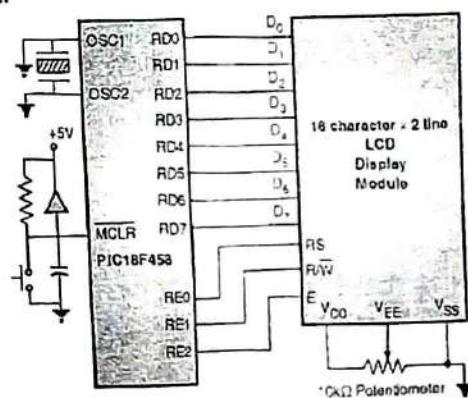


Fig. P. 7.5.7: Interfacing 16x2 LCD to PIC18

C Program

```

# include <P18F458.h>
#define RS PORTEbits.RE0 //RS = RE0
#define RW PORTEbits.RE1 //R/W = RE1
#define EN PORTEbits.RE2 //EN = RE2
char msg[4] = "GST";
char msg1[6] = "INDIA";
char cmd[5] = {0x38,0xE,0x01,0x06,0x80};
void MSDelay (unsigned int x)
{
    unsigned int i, j ;
    for (i = 0 ; i < x ; i++)
        for (j = 0; j < 165 ; j++) ;
}
void CMD (unsigned char cmd) //command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0 ;
    EN = 1;
    MSDelay (250) ;
    EN = 0 ;
}
void DAT (unsigned char dat) //display function
{
    PORTD = dat ;
    RS = 1 ;
    RW = 0 ;
    EN = 1;
    MSDelay (250) ;
    EN = 0 ;
}
void main (void)
{
    int i;
    TRISE = 0; // Make port E an output port
    TRISD = 0; // Make port D an output port
    EN = 0;
    Delay (250);
    for(i=0;i<=4;i++)
    {
        CMD (cmd[i]) ;
        MSDelay (250) ;
    }
    for(i=0;i<=2;i++)
    {

```

```

        DAT (msg[i]) ;
        MSDelay (250) ;
    }
    CMD(0xC5) ;
    for(i=0;i<=4;i++)
    {
        DAT (msg[i]) ;
        MSDelay (250) ;
    }
    while(1);
}

```

Ex. 7.5.8 : Draw interfacing diagram of LCD with PIC 18FXXXX, and write an C program to display 'SPPU' on first line with offset of 6.

Soln.:

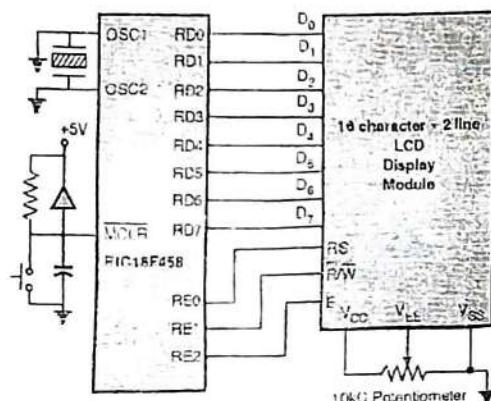


Fig. P. 7.5.8 : Interfacing 16x2 LCD to PIC18

C Program

```

# include <P18F458.h>
# define RS PORTEbits.RE0 //RS = RE0
# define RW PORTEbits.RE1 //R/W = RE1
# define EN PORTEbits.RE2 //EN = RE2
char msg[13] = "SSPU";
char cmd[5] = {0x38, 0xE0, 0x01, 0x06, 0x86};
void MSDelay (unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++);
}
void CMD (unsigned char cmd) //command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
}

```

```

MSDelay(250);
EN = 0;
}

void DAT(unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    EN = 1;
    MSDelay(250);
    EN = 0;
}

void main(void)
{
    int i;
    TRISE = 0; // Make port E an output port
    TRISD = 0; // Make port D an output port
    EN = 0;
    Delay(250);
    for(i=0;i<=4;i++)
    {
        CMD(cmds[i]);
        MSDelay(250);
    }
    for(i=0;i<=4;i++)
    {
        DAT(msg[i]);
        MSDelay(250);
    }
    while(1);
}

```

Ex. 7.5.9 : 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.

Soln.:

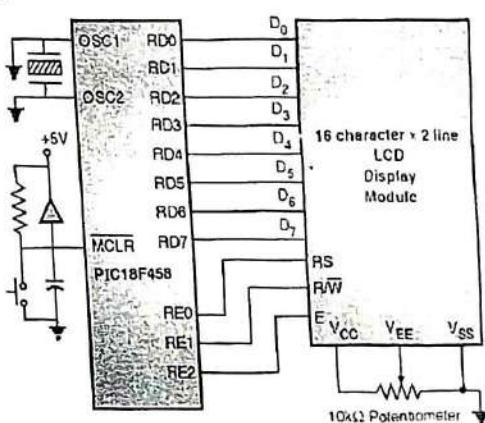


Fig. P. 7.5.9 : Interfacing 16 x 2 LCD to PIC18

C Program

```

#include <P18F458.h>
#define RS PORTEbits.RE0 //RS = RE0
#define RW PORTEbits.RE1 // R/W = RE1
#define EN PORTEbits.RE2 //EN = RE2
char msg[4] = "SPPU";
char msg1[11] = "UNIVERSITY";
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0x84};

void MSDelay(unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++)
}

void CMD(unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay(250);
    EN = 0;
}

void DAT(unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    EN = 1;
    MSDelay(250);
    EN = 0;
}

```

```
Delay (250);  
for(i=0;i<=4;i++)  
{  
    CMD (&mds[i]);  
    MSDelay (250);  
}  
for(i=0;i<=3;i++)  
{  
    DAT (msg[i]);  
    MSDelay (250);  
}  
CMD(0xC5);  
for(i=0;i<=9;i++)  
{  
    DAT (msg[i]);  
    MSDelay (250);  
}  
while(1);  
}
```

7.6 Interfacing Relays

University Questions

Q. With the help of a neat interfacing diagram explain how an electromagnetic relay can be controlled through PIC 18 microcontroller. **SPPU - Dec.17, 8 Marks**

Q. Explain interfacing of Relay and opto-Isolator to PIC18F458 with the help of suitable diagram.

SPPU - Dec. 19, 8 Marks

- 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. 7.6.1 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.

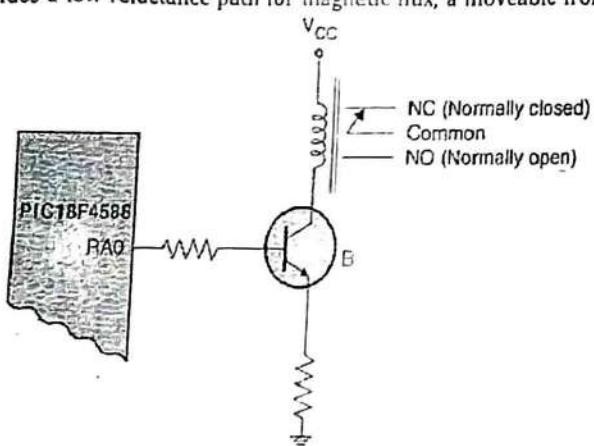


Fig. 7.6.1: Interfacing relay with 8051 diagram

Ex. 7.6.1 : Design a PIC test board test board using LED, keypad, buzzer and relays connected to ports with control using keys and write a C program for testing with S1 pressed LED ON and S2 pressed relay and buzzer ON.

Soln.:

```

#include <PI18F458.h>
#define s1 PORTDbits.RD4
#define s2 PORTDbits.RD5
#define BUZZER PORTBbits.RB1
#define LED PORTBbits.RB0

void main(void)
{
    TRISB = 0X00; //Port A pins initialized as output port
    TRISD = 0XFF; //Port D pins initialized as input port
    while(1)
    {
        if(s1==0) LED=0 else LED=1;
        if(s2==1) BUZZER=1 else BUZZER=0;
    }
}

```

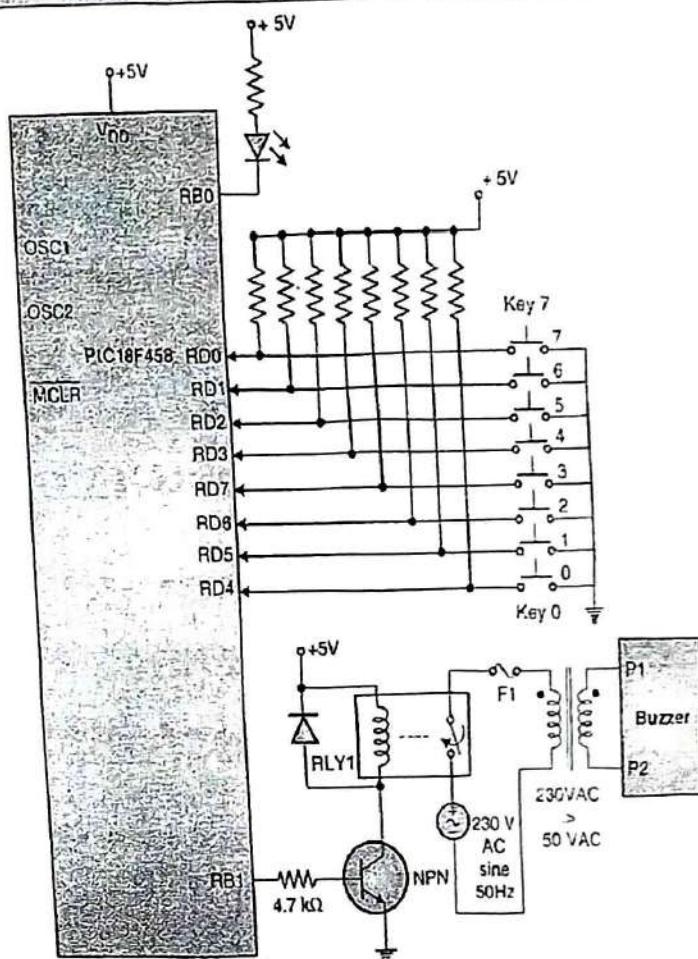


Fig. P. 7.5.8

7.7 Exam Pack (Review and University Questions)

- Q. Draw an interfacing diagram for 4x4 matrix key board and display the Key pressed on LED. Write a code.
(Refer Ex. 7.2.1) (May 15, Dec. 15, May 16, Dec. 16, 8 Marks)
- Q. Draw a neat diagram of interfacing an LED with PIC microcontroller. **(Refer Section 7.4.1)** (Dec. 16, 4 Marks)
- Q. Eight LED are connected to port A. Write a program which will connect to port A. Assume delay subroutine written at 0x30H. **(Refer Ex. 7.4.1(A))** (Dec. 15, 8 Marks)
- Q. Write a PIC18 assembly program to blink a LED. **(Refer Program 7.4.2)** (Dec. 16, 4 Marks)
- Q. Explain the functions of pins associated with LCD (16 x 2). **(Refer Section 7.5)** (Dec. 14, Dec. 15, 8 Marks)
- Q. Explain the function of following pin associated with 16 X 2 LCD controllers: RS.
(Refer Section 7.5) (May 15, May 16, 2 Marks)
- Q. Explain the function of following pin associated with 16 X 2 LCD controller: R/W .
(Refer Section 7.5) (May 15, May 16, 2 Marks)
- Q. Explain the function of following pin associated with 16 X 2 LCD controllers: E.
(Refer Section 7.5) (May 15, 2 Marks)
- Q. Explain the function of EN pin in detail. **(Refer Section 7.5)** (May 16, 2 Marks)
- Q. Explain the function of following pin associated with 16 X 2 LCD controllers: DB0-DB7.
(Refer Section 7.5) (May 15, 2 Marks)
- Q. Explain the functions of following pins of LCD (16x2) : (i) Register select (RS) (ii) Read/Write (R/W) (iii) Enable (E) (iv) VEE **(Refer Section 7.5)** (Dec. 17, 8 Marks)
- Q. List the steps for reading Busy flag and explain following pins of LCD (16 x 2) : (i) Register select (RS) (ii) Enable (E)
(Refer Section 7.5) (May 18, 8 Marks)
- Q. Draw an interfacing diagram of LCD (16 x 2) with PIC18 microcontroller and explain the functions of various pins of LCD.
(Refer Section 7.5) (Dec. 18, 8 Marks)
- Q. Draw a neat diagram of interfacing of 16x2 LCD with PIC18F458 microcontroller in 8 bit mode. Assume suitable port pins for interfacing **(Refer Section 7.5.2)** (May 16, 4 Marks)
- Q. Interface 2 line, 16 character LCD display to PIC18458. Use only one port. Write assembly language program to display message on line 2 of LCD.
- OR** 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. (Dec. 14, 4 Marks, Dec. 15, 8 Marks)
- OR** Draw and explain the interfacing of LCD with Port D and Port E of PIC18F xxx microcontroller. Write C code to display 'WELCOME'. **(Refer Ex. 7.5.1)**
- Q. With the help of a neat interfacing diagram explain how an electromagnetic relay can be controlled through PIC 18 microcontroller. **(Refer Section 7.6)** (Dec. 17, 8 Marks)
- Q. Draw and explain LCD interfacing with pic18f458. **(Refer Section 7.5.2)** (May 19, 9 Marks)
- Q. Explain 8 bit mode of LCD interfacing with PIC18F458 with the help of suitable diagram.
(Refer Section 7.5) (Dec. 19, 8 Marks)
- Q. Explain interfacing of Relay and opto-isolator to PIC18F458 with the help of suitable diagram.
(Refer Section 7.6) (Dec. 19, 8 Marks)

□□□

8

CCP Programming & PIC Interfacing-II

UNIT - IV

8.1 CCP Module in PIC18 Microcontroller

The CCP (Capture/Compare/PWM) module contains a 16-bit register that can operate as a 16-bit capture register, as a 16-bit compare register or as a PWM duty cycle register.

The operation of the CCP module is identical to that of the ECCP module with two exceptions. The CCP module has a capture special event trigger that can be used as a message received time-stamp for the CAN module which the ECCP module does not. The ECCP module, on the other hand, has enhanced PWM functionality and auto-shutdown capability. Aside from these, the operation of the module described in this section is the same as the ECCP.

8.2 Timers Required for CCP Applications

Capture/Compare/PWM Register1 (CCPR1) comprises of two 8-bit registers: CCPR1L (low byte) and CCPR1H (high byte).

It also has a control register named as CCP1CON that controls the operation of CCP1. All the registers are readable and writable.

The timers are required for using CCP module. The different timers for the different modes are listed in the Table 8.2.1.

Table 8.2.1 : CCP1 mode - timer resource

CCP1 mode	Timer resource
Capture	Timer1 or Timer3
Compare	Timer1 or Timer3
PWM	Timer2

8.3 CCP Registers

The CCPR1L and CCPR1H are the registers that hold the count value. These count values have different functions based on the mode i.e. compare, capture or PWM. We will see the uses of these registers later in this chapter. Let us now see the control registers required for the operation of CCP module.

8.3.1 CCP1CON Control Register

University Questions

- Q. Explain SFR CCP1CON register in detail.
- Q. Draw CCP1CON register.
- Q. Explain CCP1CON register in detail and also give its count, if we want to toggle CCP1 pin upon match.

SPPU - Dec. 14, May 15, Dec. 15, Dec. 16, 4 Marks

SPPU - Dec. 17, May 18, Dec. 18, 4 Marks

SPPU - May 19, 8 Marks

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DC1B1	DC1B0	CCP1M3	CCP1M2	CCP1M1	CCP1M0

bit 7

bit 0

Fig. 8.3.1: CCP1CON:CCP1 control register

bit 7-6 [Unimplemented] : Read as '0'

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

For Capture mode: Unused.

For Compare mode: Unused.

For PWM mode: These bits are the two LSbs (bit 1 and bit 0) of the 10-bit PWM duty cycle. The upper eight bits (DCx9:DCx2) of the duty cycle are found in CCPRxL.

- bit 3-0 [CCPxM3:CCPxM0] : CCPxMode 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

Legend:

R=Readable bit

W=Writable bit

U=Unimplemented bit, reads as '0'

-n=Value at POR

'1'=Bit is set

'0'=Bit is cleared

x=Bit is unknown

8.4 Generating of Waveform using Compare Mode of CCP Module

University Question

Q. Explain compare mode of operation of PIC18 in detail.

SPPU - Dec.14, Dec.16, 4 Marks

In compare mode, the 16-bit CCPR1 and ECCPR1 register value is constantly compared against either the TMR1 register pair value or the TMR3 register pair value. When a match occurs, the CCP1 pin can have one of the following actions :

- Driven high
- Driven low
- Toggle output (high-to-low or low-to-high)
- Remains unchanged

The action on the pin is based on the value of control bits CCP1M3:CCP1M0. At the same time, interrupt flag bit CCP1IF is set. The user must configure the CCP1 pin as an output by clearing the appropriate TRISC bit.

Note: Cleaning the CCP1CON register will force the CCP1 compare output latch to the default low level. This is not the data latch.

Fig. 8.4.1 shows the block diagram of compare mode configuration of CCP module.

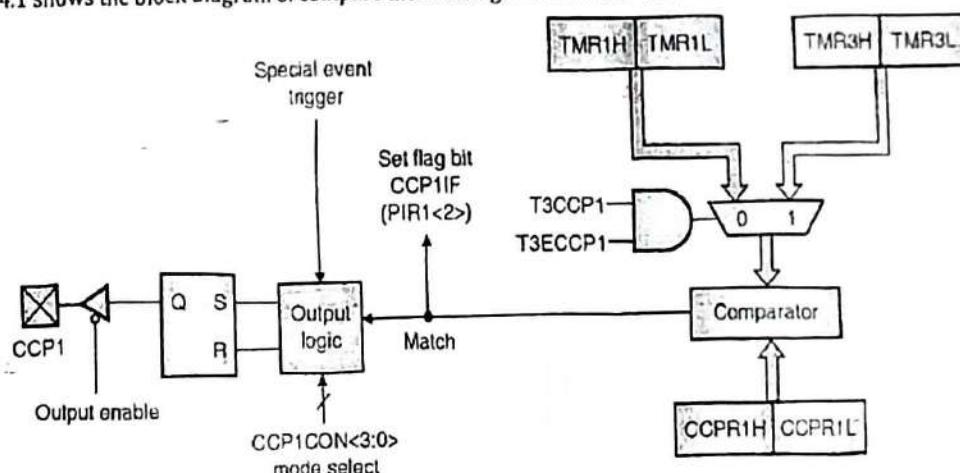


Fig. 8.4.1 : Block diagram of compare mode configuration of CCP module

8.4.1 Programming the CCP Module in Compare Mode

University Questions

- Q. List the steps involved in programming PIC microcontroller in Compare mode.
SPPU - Dec. 17, May 18, Dec. 18, 4 Marks
- Q. Mention the steps of programming Capture mode.
SPPU - May 19, 8 Marks
- Q. Explain steps for programming compare mode of CCP1 module in PIC18f458.
SPPU - Dec. 19, 8 Marks

The following steps are taken in programming the compare mode :

1. Initialize the CCP1CON register for the compare option.
2. Initialize the T3CON register for Timer 1 (or Timer 3)
3. Initialize the CCPR1H : CCPR1L registers.
4. Make the CCP1 pin an output pin.
5. Initialize Timer 1 (or Timer3) register values.
6. Start Timer1 (or Timer3).
7. Monitor the CCP1IF flag (or use an interrupt).

Program 8.4.1: Write a program to generate a square wave of 1Hz to blink LED.

Soln.:

```
MOVlw 0x02
MOVwf CCP1CON      ;Compare mode toggle
                     ;upon match
MOVlw 0x42
MOVwf T3CON        ;Timer3 for compare, 1:1 prescaler
BCF   TRISC, CCP1    ;CCP1 pin as output
BSF   TRISC, T1CKI   ;T3CLK pin as input pin
```

```
MOVlw D10
MOVwf CCPR1L      ;CCPR1L = 10
MOVlw 0x0          ;CCPR1H = 0
OVER CLRF TMR3H    ;clear TMR3H
CLRF TMR3L        ;clear TMR3L
BCF PIR1, CCP1IF   ;clear CCP1IF
BSF T3CON, TMR3ON  ;start timer3
B1  BTFS  PIR1, CCP1IF
BRA B1
;----- CCP toggle CCP pin upon match
B2  BCF T3CON, TMR3ON  ;stop Timer3
GOTO OVER          ;keep doing it
```

C program

```
#include < P18F458.h>
void main(void)
{
    CCP1CON=0x02;
    //Compare mode, toggle upon match
    T3CON=0x42;
    //Timer3 for compare, 1:1 prescaler
    TRISCbits.TRISC2=0; //CCP1 pin an output
    TRISCbits.TRISC0=1; //T3CLK pin an input
    CCPR1L=10;          //load CCPR1L
    CCPR1H = 0;          //load CCPR1H
    while (1)
{
```



```

TMR3H=0;
TMR3L=0;
PIR1bits.CCP1IF=0; //clear CCP1IF flag
T3CONbits.TMR3ON=1; //turn on Timer3
while (PIR1bits.CCP1IF==0); //wait for CCP1IF
//CCP toggles CCP pin upon match
T1CONbits.TMR3ON=0; //stop Timer3
}
}

```

Program 8.4.2: Write a program to create a square wave of 2.5 kHz period and 50% duty cycle

OR 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.: Assume oscillator frequency = 10 MHz

Frequency = 2.5 kHz

$$\text{Time Period} = \frac{1}{2.5 \text{ kHz}} = 0.4 \text{ msec}$$

$$\text{So count} = \frac{0.4 \text{ msec}}{0.4 \mu\text{sec}} = (1000)_{10} = (03E8)_{16}$$

MOVlw 0x02	
MOVWF CCP1CON	;Compare mode, toggle upon match
MOVlw 0x01	
MOVWF T3CON	;use Timer3 for compare mode
MOVlw 0x0...	
MOVWF T3CON	;Timer3, internal GJ,K, 1:1 prescale
BCLF TRISC, CCP1	;CCP1 pin as output
MOVlw 0x03	
MOVWF CCP1H	;CCPR1H = 0x03
MOVlw 0xE8	
MOVWF CCP1L	;CCPR1L = 0xE8
OVER CLRFTMR3H	;clear TMR3H
CLRFTMR3L	;clear TMR3L
BCLF PIR1, CCP1IF	;clear CCP1IF
BSFT3CON, TMR3ON	;start Timer3
B1 BTESSPIR1, CCP1IF	
BRA B1	;CCP toggles CCP1 pin upon match
BSFT3CON, TMR3ON	;stop Timer3
GOTO OVER	;repeat again

Program 8.4.2(A) : Using compare mode, write the assembly language program to toggle the LED every 10 pulses. Use Timer 1 as counter **SPPU - Dec. 14, 8 Marks**

Soln.:

Label	Instruction	Comment
	BSF TRISC,0	Program RC0 i.e. T1CKI pin as input pin
	BCF TRISC, 2	Program RC2 as output pin
	MOVLW 0x00	Load T3CON for Timer 3 as timer
	MOVWF T3CON	
	MOVLW 0x02	Load T1CON for Timer 1 as counter
	MOVWF T1CON	
	MOVLW 0x02	Initialize CCP1CON for compare mode
	MOVWF CCP1CON	
	MOVLW D'10'	Load CCPRL register
	MOVWF CCPRL	
again:	MOVLW 0x00	Load CCPRH register
	MOVWF CCPRH	
	CLRF TMR1L	Clear contents of TMR1L
	CLRF TMR1H	Clear contents of TMR1H
	BCF PIR1, CCP1IF	Clear CCP1IF flag
wait:	BSF T1CON, TMR1ON	Start Timer 1
	BTFS S PIR1, CCP1IF	Wait for pulse on timer 1
	BRA wait	
	BCF T1CON, TMR1ON	STOP Timer 1
	GOTO again	repeat

8.5 Capture Mode

University Questions

- Q. Explain capture mode of operation of PIC18 in detail. **SPPU - May 15, 8 Marks**
- Q. List the steps involved in programming PIC microcontroller in capture mode. **SPPU - May 18, Dec. 18, 4 Marks**
- Q. Explain steps for programming in capture mode. **SPPU - Dec. 15, 4 Marks**
- Q. Explain the steps for programming the capture mode of CCP module in PIC18 microcontroller for measuring period of pulse. **SPPU - Dec. 16, 8 Marks**

In capture mode, CCP1H:CCP1L captures the 16-bit value of the TMR1 or TMR3 register when an event occurs on pin RC2/CCP1. An event is defined as:

- Every falling edge
- Every rising edge
- Every 4th rising edge
- Every 16th rising edge

An event is selected by control bits CCP1M3:CCP1M0 (CCP1CON<3:0>). When a capture is made, the interrupt request flag bit, CCP1IF (PI R registers), is set. It must be cleared in software. If another capture occurs before the value in register CCP1 is read, the old captured value will be lost.

In capture mode, the RC2/CCP1 pin should be configured as an input by setting the TRISC<2> bit.

Note : If the RC2/CCP1 is configured as an output, a write to the port can cause a capture condition.

The timers used with the capture feature (either Timer 1 and/or Timer 3) must be running in timer mode or synchronized counter mode. In a synchronous counter mode, the capture operation may not work. The timer used with each CCP module is selected in the T3CON register.

The Fig. 8.5.1 shows the configuration of the CCP module in capture mode.

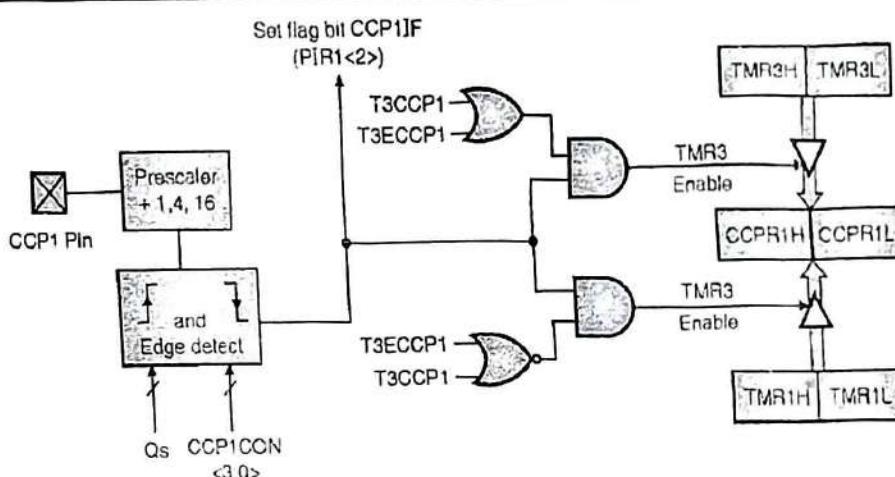


Fig. 8.5.1 : Configuration of the CCP module in capture mode

Steps for programming capture mode

The following steps are used in programming capture mode for measuring the period of a pulse:

1. Initialize the CCP1CON register for capture.
2. Make the CCP1 pin an input pin.
3. Initialize the T3CON register to select Timer3.
4. Read the Timer1 (or Timer3) register value on the first rising edge and save it.
5. Read the Timer1 (or Timer3) register value on the second rising edge and save it.
6. Subtract the value in step 4 from the value in step 5.

Program 8.5.1: Write a program to generate a square wave with frequency 2.5 KHz and 50% duty cycle on the CCP1 pin using Timer 3.

SPPU - Dec. 15, 8 Marks

Soln.:

Let XTAL frequency = 10 MHz

$$\therefore \text{Timer clock frequency} = \frac{f_{\text{osc}}}{4} = \frac{10 \text{ MHz}}{4} = 2.5 \text{ MHz}$$

$$\therefore \text{Timer clock period} = \frac{1}{2.5 \text{ MHz}} = 0.4 \mu\text{s}$$



Frequency required = 2500 Hz

$$\therefore \text{Time Period for one count} = \frac{1}{2500} = 400 \mu\text{s}$$

Assuming 50% duty cycle, delay required = 200 μs

$$\therefore \text{Pulses to be counted} = \frac{200 \mu\text{s}}{0.4 \mu\text{s}} = 500$$

$$\begin{aligned}\therefore \text{Count} &= 65536 - 500 \\ &= (65036)_{10} = \text{FE0CH}\end{aligned}$$

CCPR1H = FEH

CCPR1L = 0CH

C18 program:

```
#include < P18F458.h>
void main(void)
{
    T3CON = 0x42;           // Initialize T3CON for Timer 3 operation
    CCPR1H = FE;            // Initialize CCPR1H
    CCPR1L = 0C;            // Initialize CCPR1L
    CCP1CON = 0x02;          // Initialize CCP1CON register for compare mode
    TRISCBits.TRISC2 = 0;   // program CCP1 pin as output pin
    TRISCBits.TRISO = 1;    // make TICK pin as input pin
    while(1)
    {
        TMR3L = 0;           // Initialize timer 3 lower byte
        TMR3H = 0;           // Initialize timer 3 higher byte
        PIR1bits.CCP1IF = 0; // clear CCP1IF flag
        T3CONbits.TMR3ON = 1; // start timer 3
        while (PIR1bits.CCP1IF == 0); // wait FOR CCP1IF i.e. CCP interrupt
        T3CONbits.TMR3ON = 0; // stop Timer 3
    }
}
```

Program 8.5.1(A): Write a program to measure the pulse width of signal given on CCP1 pin and give output on Ports B and D.

OR 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.

SPPU - Dec. 15, May 16, Dec. 16, 9 Marks

OR Assume a pulse is being fed to the CCP1 pin. Using capture mode write assembly language program to measure the period of the pulse and puts the results on PORTB and PORTD. Use Timer 3 for Capture mode.

SPPU - May 15, 8 Marks

OR Write C program to measure time period of a square wave applied at CCP1 pin using capture mode of PIC18F458. Use timer3 and crystal frequency of 10 MHz.

SPPU - Dec. 19, 8 Marks

Soln.:

```
MOVlw 0x05
MOVWF CCP1CON;Capture mode rising edge
MOVLW 0x0
MOVWF T3CON ;ProgramTimer1, internal
;CLK, 1:1 prescale
CLRF TRISB ;set PORTB as output port
CLRF TRISD ;set PORTD as output port
BSF TRISC, CCP1 ;set CCP1 pin as input pin
MOVLW 0x0
MOVWF CCPR1H ;CCPR1H = 0
MOVWF CCPR1L ;CCPR1L = 0
OVER CLRF TMR1H ;clear TMR1H
CLRF TMR1L ;clear TMR1L
BCF PIR1, CCP1IF ;clear CCP1IF
RE_1 BTFS SPIR1, CCP1IF
BRA RE_1 ;stay here for 1 rising
;edge
RSF TICON,TMR1ON ;start Timer1
BCF PIR1, CCP1IF ;clear CCP1IF for next
RE_2 BTFS SPIR1, CCP1IF
BRA RE_2 ;stay here for 2 rising
;edge
BCF TICON,TMR1ON ;stop Timer1
MOVEFF TMR1L, PORTB ;put low byte on
;PORTB
MOVEFF TMR1H, PORTD ;put high byte
;on PORTD
GOTO OVER ;repeat entire process
```

C program

```
#include <p18f458.h>
void main()
{
    CCP1CON = 0x05; //capture mode on every rising edge
    T3CON = 0x0; //Timer1 for capture
    TICON = 0x0; //Timer1 for internal clk, 1:1
    //prescaler
    TRISB = 0; //make PORTB output port
    TRISD = 0; //make PORTD output port
```

```

TRISChits.TRISC2=1;//make CCP1 pin an output
CCPR1L=0;           //CCPR1L=0
CCPR1H=0;           //CCPR1H=0
while (1)
{
    TMR1H=0;          //clear Timer1
    TMR1L=0;
    PIR1bits.CCP1IF=0; //clear CCP1IF flag
    while (PIR1bits.CCP1IF==0); //wait for 1st rising edge
    TICONbits.TMR1ON=1; //start Timer1
    PIR1bits.CCP1IF=0; //clear CCP1IF for next
    //edge
    while (PIR1bits.CCP1IF==0); //wait for 2nd rising
    //edge
    TICONbits.TMR1ON=0; //stop Timer1
    PORTB=CCPR1L;      //display the clock count
    PORTD=CCPR1H;
}
}

```

8.6 PWM Mode

University Questions

- Q. Write a short note on PWM control DC motor using CCP mode. **SPPU - May 16, 8 Marks**
- Q. List the steps involved in programming PIC microcontroller in PWM mode. **SPPU - Dec. 17, 4 Marks**

In Pulse Width Modulation (PWM) mode, the CCP1 pin produces up to a 10-bit resolution PWM output. Since the CCP1 pin is multiplexed with the PORTC data latch, the TRISC<2> bit must be cleared to make the CCP1 pin an output. Let us understand what PWM is and how it works.

A PWM output has a fixed period and a varying time for which the output stays high as shown in Fig. 8.6.1. The frequency of the PWM is the inverse of the period (1/period). The duty cycle is given by the formula :

$$\text{Duty cycle} = \frac{T_{on}}{T_{on} + T_{off}}$$

The timer 2 or timer 4 value decides the T_{on} value. Hence the Timer 2 or Timer 4 value decides the duty cycle. Since the period is same, the change in duty cycle causes a change in the average DC value of the signal. This average value of the signal can be taken as the analog equivalent of the digital value in the timer 2 or 4.

Hence many a times, PWM is also referred to as a DAC (Digital to Analog Converter).

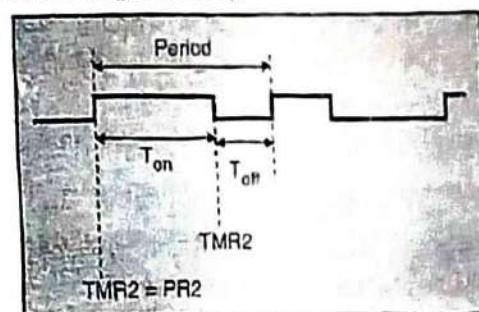


Fig. 8.6.1 : PWM OUTPUT

In Pulse-Width Modulation (PWM) mode, the CCP1 pin produces up to a 10-bit resolution PWM output. Since the CCP1 pin is multiplexed with the PORTC data latch, the TRISC<2> bit must be cleared to make the CCP1 pin an output.

Note : Clearing the CCP1CON register will force the CCP1PWM output-latch to the default low level. This is not the PORTC/I/O data latch.

Fig. 8.6.2 shows a simplified block diagram of the CCP module in PWM mode.

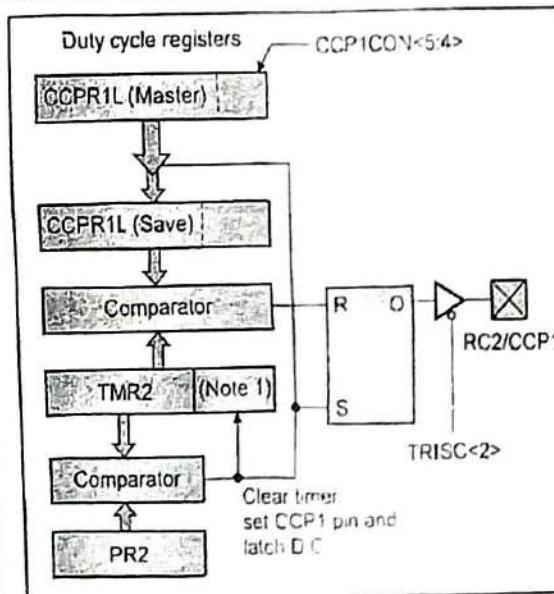


Fig. 8.6.2 : Simplified PWM block diagram

Note : 8-bit timer is concatenated with 2-bit internal Q clock, or 2 bits of the prescaler, to create 10-bit time base.

A PWM output (Fig. 8.6.3) has a time base (period) and a time that the output stays high (duty cycle). The frequency of the PWM is the inverse of the period (1/period).



8.6.1 Period of PWM

The PWM period is specified by writing to the PR2 register. The PWM period can be calculated using the following formula.

$$\text{PWM period} = [(PR2 + 1) \cdot 4 \cdot T_{OSC}] (\text{TMR2 Pre-scale value})$$

PWM frequency is defined as $1 / [\text{PWM period}]$.

When TMR2 is equal to PR2, the following three events occur on the next increment cycle.

- TMR2 is cleared
- The CCP1pin is set (exception: if PWM duty cycle=0%, the CCP1 pin will not be set)
- The PWM duty cycle is latched from CCPRL into CCPRH.

Note : The Timer 2 postscaler is not used in the determination of the PWM frequency. The postscaler could be used to have a servo update rate at a different frequency than the PWM output.

Ex. 8.6.1: Find the PR2 value for the following PWM frequencies. Assume XTAL = 10 MHz and prescaler = 1.

$$(a) 10 \text{ kHz}, \quad (b) 25 \text{ kHz}$$

Soln.:

$$\begin{aligned} (a) \text{ PR2 value} &= [(10 \text{ MHz} / (4 \times 10 \text{ kHz} \times 1)) - 1] \\ &= 250 - 1 = 249 \quad \dots \text{Ans.} \\ (b) \text{ PR2 value} &= [(10 \text{ MHz} / 4 \times 25 \text{ kHz} \times 1) - 1] \\ &= 100 - 1 = 99 \quad \dots \text{Ans.} \end{aligned}$$

Ex. 8.6.2: Find the minimum and maximum FPWM frequency allowed for XTAL = 10 MHz. State the PR2 and prescaler values for the minimum and maximum FPWM.

Soln.:

We get the minimum FPWM by making PR2 = 255 and prescaler = 16, which gives us $10 \text{ MHz} / (4 \times 16 \times 256) = 610 \text{ Hz}$Ans.

We get the maximum FPWM by making PR2 = 1 and prescaler = 1, which gives us $10 \text{ MHz} / (4 \times 1 \times 1) = 2.5 \text{ MHz}$Ans.

8.6.2 Duty Cycle of PWM

- The PWM duty cycle is specified by writing to the CCPRL register and to the CCP1CON<5:4>bits. Up to 10-bit resolution is available.

- The CCPRL contains the eight MSbs and the CCP1CON<5:4>contains the two LSbs. This 10-bit value is represented by CCPRL:CCP1CON<5:4>. The following equation is used to calculate the PWM duty cycle in time.

$$\text{PWM duty cycle} = (CCPRL:CCP1CON<5:4> \cdot T_{osc}) / (\text{TMR2 Pre-scale value})$$

- CCPRL and CCP1CON <5:4> can be written to at any time, but the duty cycle value is not latched into CCPRH until after a match between PR2 and TMR2 occurs (i.e., the period is complete). In PWM mode, CCPRH is a read-only register.
- The CCPRH register and a 2-bit internal latch are used to double-buffer the PWM duty cycle. This double-buffering is essential for glitchless PWM operation.
- When the CCPRH and 2-bit latch match TMR2, concatenated with an internal 2-bit Q clock or 2 bits of the TMR2 prescaler, the CCP1 pin is cleared.
- The maximum PWM resolution (bits) for a given PWM frequency is given by the following equation.

$$\text{PWM Resolution (max)} = \frac{F_{osc}}{\log_2(FPWM)} \text{ bits}$$

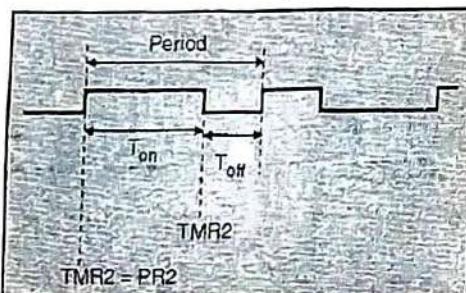


Fig. 8.6.3 : A PWM output(period)

Note: If the PWM duty cycle value is longer than the PWM period, the CCP1 pin will not be cleared.

Ex. 8.6.3: Find the values of registers PR2, CCPRL, and DC1B2:DC1B1 for the following PWM frequencies if we want a 75% duty cycle. Assume XTAL = 10 MHz.

$$(a) 1 \text{ kHz} \quad (b) 2.5 \text{ kHz}$$

Soln. :

(a) Using the $PR2 = F_{osc} / (4 \times FPWM \times N)$ equation, we must set $N = 16$ for prescale. Therefore, we have

$$\begin{aligned} PR2 &= [(10 \text{ MHz} / (4 \times 1 \text{ kHz} \times 16)) - 1] = 156 - 1 \\ &= 155 \text{ and because } 155 \times 75\% = 116.25 \text{ we have CCPRL} \\ &= 116 \text{ and DC1B2:DC1B1} = 01 \text{ for the 0.25 portion.} \end{aligned}$$

- (b) Using the $PR2 = F_{osc}/(4 \times FPWM \times N)$ equation, we can set $N = 4$ for prescale. Therefore, we have

$$PR2 = 10 \text{ MHz} / (4 \times 2.5 \text{ kHz} \times 4) - 1 = 250 - 1 = 249 \text{ and because } 249 \times 75\% = 186.75 \text{ we have CCPR1L} = 186 \text{ and DC1B2:DC1B1} = 11 \text{ for the 0.75 portion.}$$

8.6.3 Steps for Programming the CCP Module for PWM Generation

University Questions

- Q. Explain steps for programming the CCP module for PWM generation. **SPPU - Dec. 15, 4 Marks**
- Q. Explain the steps involved in PWM programming using CCP module in PIC18F458 microcontroller. **SPPU - May 16, 8 Marks**

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 CCP1CON 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.

Program 8.6.1: Write a program to generate PWM 1 kHz frequency with a 50% duty cycle. Let $N = 16$.

OR Write a program to create a 1 kHz PWM frequency with a 50% duty cycle on CCP1 pin. Assume XTAL = 10 MHz. Let $N = 16$. **SPPU - Dec. 15, 4 Marks**

Soln. :

Using the $PR2 = F_{osc}/(4 \times FPWM \times N)$ equation, we must set $N = 16$ for prescale.

Therefore, we have

$$PR2 = [(10 \text{ MHz} / (4 \times 1 \text{ kHz} \times 16)) - 1 = 156 - 1 = 155 \text{ and because } 155 \times 50\% = 77.5 \text{ we have CCPR1L} = 77 \text{ and DC1B2:DC1B1} = 10 \text{ for the 0.5 portion.}]$$

```
CLRF CCP1CON ;clear CCP1CON reg
MOVLW D'155'
MOVWF PR2
```

```
MOVLW D'77' ;50% duty cycle
MOVWF CCPR1L
BCF TRISC,CCP1 ;make PWM pin an output
MOVLW 0x01 ;Timer2, 4 prescale, no postscaler
MOVWF T2CON
MOVLW 0x1C ;PWM mode, 10 for DC1B1:B0
MOVWF CCP1CON
CLRF TMR2 ;clear Timer2
BSF T2CON,TMR2ON ;turn on Timer 2
AGAIN BCF PIR1,TMR2IF ;clear Timer2 flag
OVER BTFS S PIR1,TMR2IF ;wait for end of period
BRA OVER
GOTO AGAIN ;continue
```

Program 8.6.1(A): Create a 1.8 KHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL = 10 MHz.

SPPU - May 15, 8 Marks

Soln. :

Using the $PR2 = F_{osc}/(4 \times FPWM \times N)$ equation, we must set $N = 16$ for prescale.

Therefore, we have

$$PR2 = [(10 \text{ MHz} / (4 \times 1.8 \text{ kHz} \times 16)) - 1 = 87 - 1 = 86 \text{ and because } 86 \times 25\% = 21.5 \text{ we have CCPR1L} = 21 \text{ and DC1B2:DC1B1} = 10 \text{ for the 0.5 portion.}]$$

```
CLRF CCP1CON ;clear CCP1CON reg
MOVLW D'86'
MOVWF PR2
MOVLW D'21' ;25% duty cycle
MOVWF CCPR1L
BCF TRISC,CCP1 ;make PWM pin an output
MOVLW 0x01 ;Timer2, 4 prescale, no postscaler
MOVWF T2CON
MOVLW 0x2C ;PWM mode, 10 for DC1B1:B0
MOVWF CCP1CON
CLRF TMR2 ;clear Timer2
```



```

BSF T2CON,TMR2ON ;turn on Timer 2
AGAIN: BCF PIR1,TMR2IF      ;clear Timer2 flag
OVER: BTFSS PIR1,TMR2IF     ;wait for end of period
    BRA OVER
    GOTO AGAIN             ;continue

```

Program 8.6.2 : Write a program to generate PWM 1 kHz frequency with a 10% duty cycle.

OR Write a program for 1 kHz 10% duty cycle PWM waveform.

Soln.:

Using the $PR2 = \frac{F_{osc}}{(4 \times F_{PWM} \times N)}$ equation, we must set $N = 16$ for prescale.

Therefore, we have

$PR2 = [(10 \text{ MHz} / (4 \times 1 \text{ kHz} \times 16))] - 1 = 156 - 1 = 155$
and because $155 \times 10\% = 15.5$ we have CCPR1L = 15 and DC1B2:DC1B1 = 10 for the 0.5 portion

```

CLRF CCPICON      ;clear CCPICON reg
MOVLW D'155'
MOVWF PR2
MOVLW D'15'        ;10% duty cycle
MOVWF CCPRL
BCF TRISC,CCPI    ;make PWM pin an output
MOVLW D'0x01       ;Timer2, 4 prescale, no postscaler
MOVWF T2CON
MOVLW D'1C          ;PWM mode, 10 for DC1B1:BO
MOVWF CCPICON
CLRF TMR2          ;clear Timer2
BSF T2CON,TMR2ON  ;turn on Timer 2
AGAIN: BCF PIR1,TMR2IF ;clear Timer2 flag
OVER: BTFSS PIR1,TMR2IF ;wait for end of period
    BRA OVER
    GOTO AGAIN           ;continue

```

Program 8.6.3: Create a 2 kHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL = 10 MHz.

SPPU - Dec. 14, 4 Marks

Soln.:

Using the $PR2 = \frac{F_{osc}}{(4 \times F_{PWM} \times N)}$ equation, we must set $N = 16$ for prescale.

Therefore, we have

$PR2 = [(10 \text{ MHz} / (4 \times 2 \text{ kHz} \times 16))] - 1 = 78 - 1 = 77$ and because $77 \times 25\% = 19.25$ we have CCPR1L = 19 and DC1B2:DC1B1 = 01 for the 0.25 portion.

```

CLRF CCPICON      ;clear CCPICON reg
MOVLW D'77'
MOVWF PR2
MOVLW D'19'        ;25% duty cycle
MOVWF CCPRL
BCF TRISC,CCPI    ;make PWM pin an output
MOVLW D'0x01       ;Timer2, 4 prescale, no postscaler
MOVWF T2CON
MOVLW D'1C          ;PWM mode, 01 for DC1B1:BO
MOVWF CCPICON
CLRF TMR2          ;clear Timer2
BSF T2CON,TMR2ON  ;turn on Timer 2
AGAIN: BCF PIR1,TMR2IF ;clear Timer2 flag
OVER: BTFSS PIR1,TMR2IF ;wait for end of period
    BRA OVER
    GOTO AGAIN           ;continue

```

Program 8.6.3 (A) : Create a 1.8 kHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL=10 MHz.

SPPU - May 15, 4 Marks

Soln. :

$$PR2 = \left[\frac{F_{osc}}{F_{PWM} \times 4 \times N} \right] - 1$$

$$PR2 = \left[\frac{10 \times 10^6}{1.8 \times 10^3 \times 4 \times 16} \right] - 1 = 85$$

For 50% duty cycle

$$\frac{85 \times 25}{100} = 21.25$$

\therefore CCPR1L = 21 and DC1B2 : DC1B1 = 01 for 0.25

Assembly program

Label	Instruction	Comments
	CLRF CCP1CON	Initialize CCP1CON to 00
	MOVLW D '85'	Load PR2 to decimal 85
	MOVWF PR2	
	MOVLW D '21'	Initialize CCPR1L to 21 decimal
	MOVWF CCPR1L	
	BCF TRISC, 2	Program CCP1 pin as output pin
	MOVLW 0x03	Program Timer2 as 16 prescaler,
	MOVWF T2CON	
	MOVLW 0x1C	PWM mode, DC1B2 : DC1B1 = 01 for 0.25
	MOVWF CCP1CON	
	CLRF TMR2	Initialize TMR2 to 0
	BSF T2CON, TMR2ON	Start Timer 2
again:	BCF PIR1, TMR2IF	Clear Timer 2 interrupt flag
here:	BTFS S PIR1, TMR2IF	Wait for Timer 1 to overflow
	BRA here	
	GOTO again	Jump to again

Program 8.6.4 : Write a program for 2.5 kHz and 75 % duty cycle PWM generation with N=4.

Soln.: Using the $PR2 = F_{osc} / (4 \times FPWM \times N)$ equation, we must set N = 4 for prescale.

Therefore, we have

$PR2 = [(10 \text{ MHz} / (4 \times 2.5 \text{ kHz} \times 4))] - 1 = 250 - 1 = 249$ and because $249 \times 75\% = 186.75$ we have CCPR1L = 186 and DC1B2:DC1B1 = 11 for the 0.75 portion.

```

CLRF CCP1CON           ;clear CCP1CON reg
MOVLW D'249'
MOVWF PR2
MOVLW D'186'           ;75% duty cycle
MOVWF CCPR1L
BCF TRISC, CCP1        ;make PWM pin an output
MOVLW 0x01              ;Timer2, 4 prescale, no post scaler
MOVWF T2CON
MOVLW 0x3C              ;PWM mode, 11 for DC1B1:B0
MOVWF CCP1CON
CLRF TMR2               ;clear Timer2
BSF T2CON, TMR2ON       ;turn on Timer 2
AGAIN: BCF PIR1, TMR2IF ;clear Timer2 flag
OVER: BTFS S PIR1, TMR2IF ;wait for end of period
BRA OVER
GOTO AGAIN              ;continue

```

8.7 Speed Control of DC Motor using PWM Mode of CCP Module

University Question

Q. Explain DC motor control using PWM of CCP1 module with the help of suitable diagram

SPPU - Dec. 19, 8 Marks

- A DC motor works by converting electric power into mechanical work. This is accomplished by forcing current through a coil and producing a magnetic field that spins the motor.

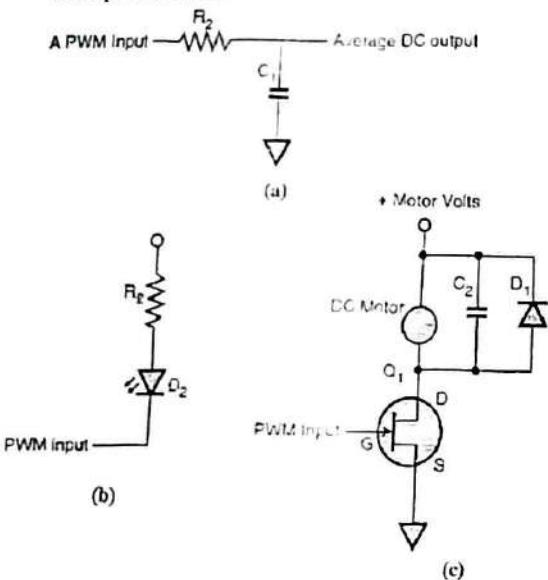


Fig. 8.7.1 : Use of PWM output

- The speed of the motor is dependent on the voltage given to it.
- Hence by varying the voltage to DC motor we can control the speed of DC motor.
- This is mainly implemented by PWM (Pulse Width Modulation)
- In PWM, a variation in the duty cycle generates variation in the average DC voltage.
- Pulse Width Modulation (PWM) is one of the methods to provide Digital to Analog Conversion, by controlling the duty cycle of the square wave provided at the output the average voltage varies and hence a DC voltage at the output of the filter available is dependent on the pulse width (counter digital value).

- A RC circuit provides filtering and gives the average DC voltage proportional to the count as shown in Fig. 8.7.1(a).
- The Fig. 8.7.1(b) shows how it can be used to turn on a LED and then to control its intensity by varying the voltage (inversely proportional) across it. The Fig. 8.7.1(c) shows how it can be used to turn on a DC motor and then to control its speed by varying the voltage across it.

8.7.1 Interfacing DC Motor using L293 H-Bridge and Opto-isolator

Q. Explain interfacing of Relay and opto-isolator to PIC18F455 with the help of suitable diagram.

SPPU - Dec. 19, 8 Marks

- 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 optoisolator at its input as shown the Fig. 8.7.2. The LED at the input triggers the transistor, optically and hence no electrical path exists between the LED and transistor.
- Since the motor requires high current, it cannot be driven from the pins of 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 direction is possible using H-bridge.
- Fig. 8.7.2(a) shows the pin diagram while Fig. 8.7.2(b) shows the block 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.

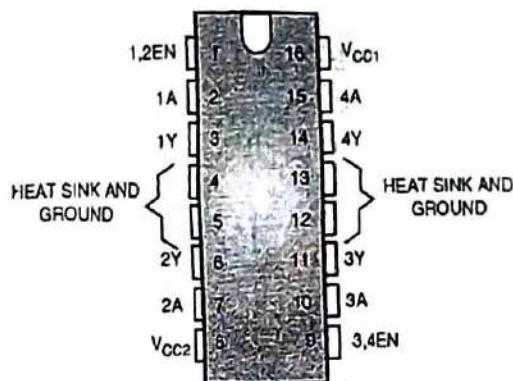


Fig. 8.7.2(a) : Pin Diagram

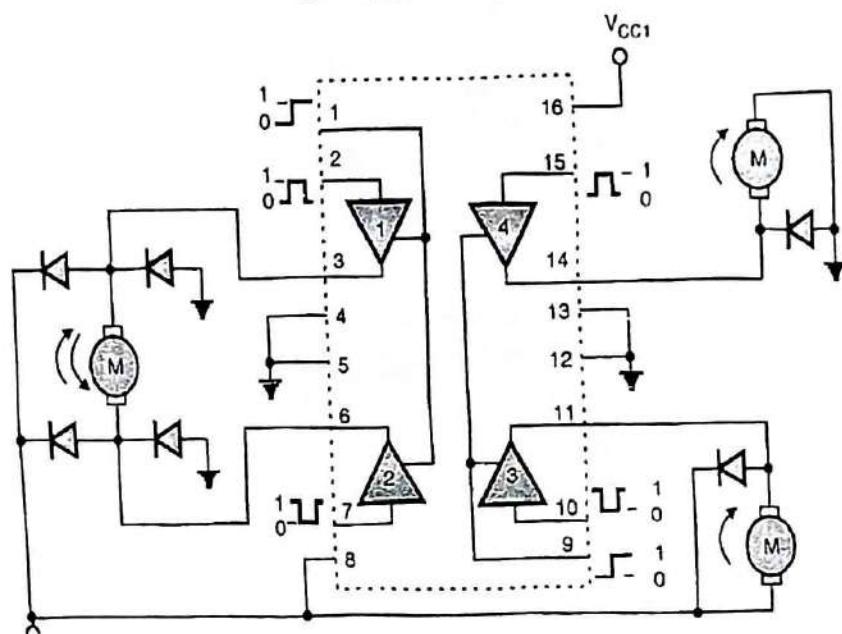


Fig. 8.7.2(b) : Circuit diagram

- 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.
- The Table 8.7.1 shows how an output can be enabled.

Table 8.7.1 : Function Table (each driver)

Inputs		Output
A	EN	Y
H	H	H
L	H	L
X	L	Z

H = high level, L = low level, X = irrelevant, Z = high impedance (off)

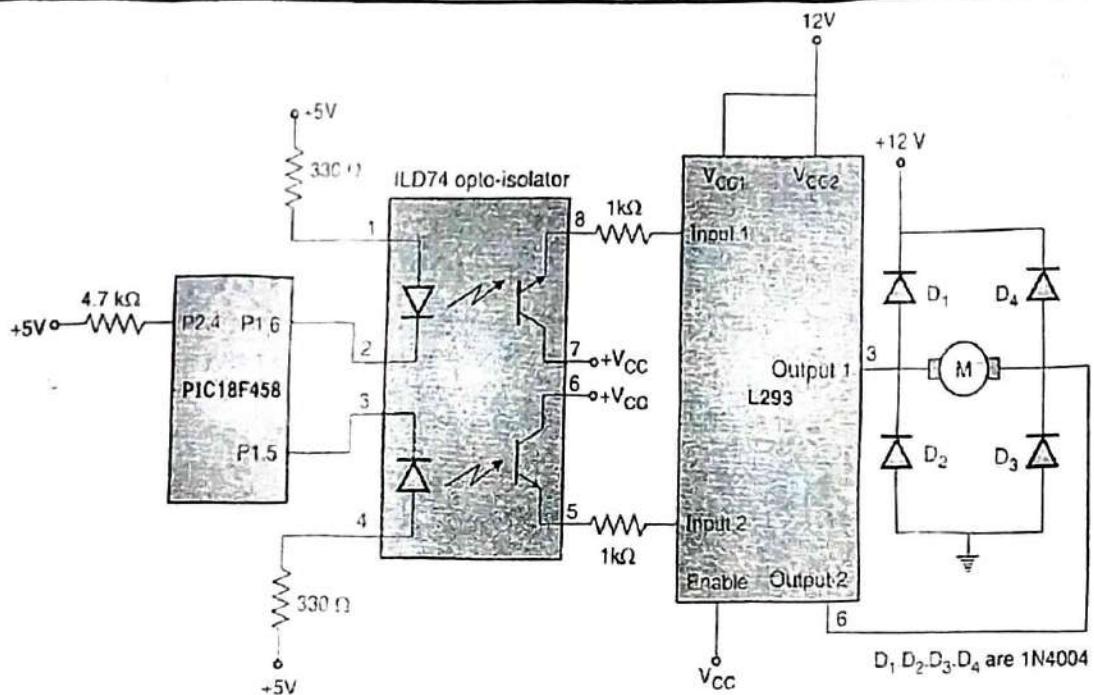


Fig. 8.7.3 : Interfacing DC motor with opto-isolator, L293 and PIC18F458 to control direction of the motor

Ex. 8.7.1: Draw an interfacing diagram and write an algorithm for DC Motor speed controller using PIC16XXX.

OR Draw an interfacing diagram to interface the DC motor with PIC 18FXXX for speed control using PWM with 5 kHz, 40% Duty cycle, N=4, Also write an embedded C program

OR Design a PIC based system to interface DC motor to monitor the status of switch connected to pin RC2 and do the following :

- If SW = 0, the DC motor moves with 50% duty cycle pulse
- If SW = 1, the DC motor moves with 25% duty cycle pulse

OR Explain speed control of DC motor using Compare mode.

SPPU - May 15, 4 Marks

Soln. : Interfacing Diagram

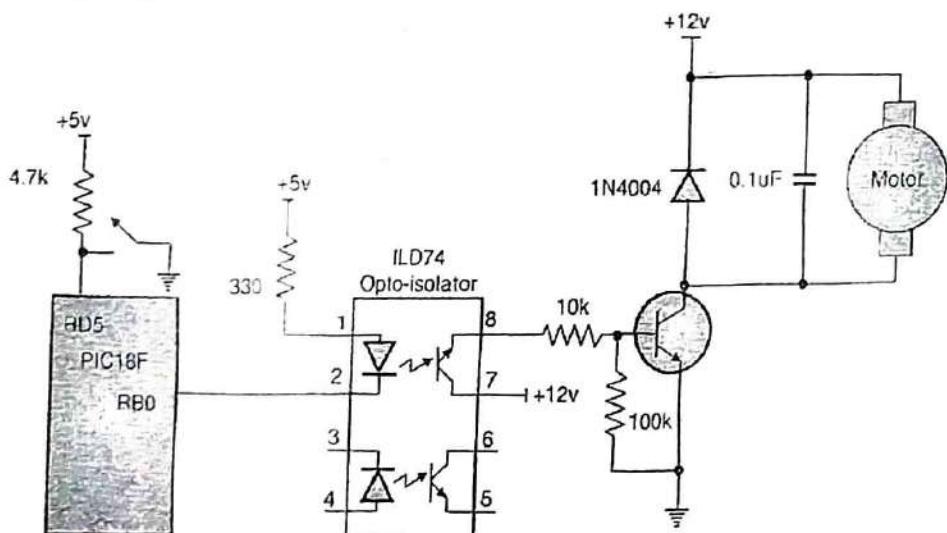


Fig. P. 8.7.1

C program

```
#include<p18f458.h>
#define SW1 PORTDbits.RD5
#define MTR PORTBbits.RB0
void MSDelay(unsigned int x)
{
    unsigned char i,j;
    for(i=0;i<1275;i++)
        for(j=0;j<x;j++);
}

void main()
{
    TRISD=0x20;
    TRISB=0xFE;
    while(1)
    {
        if(SW1==1)
        {
            MTR=1;
            MSDelay(25);
            MTR=0;
            MSDelay(75);
        }
        else
        {
            MTR=1;
            MSDelay(50);
            MTR=0;
            MSDelay(50);
        }
    }
}
```

8.8 Exam Pack (Review and University Questions)

- Q. Explain SFR CCP1CON register in detail. (Refer Section 8.3.1)
 (Dec. 14, May 15, Dec. 15, Dec. 16, 4 Marks)
- Q. Draw CCP1CON register. (Refer Section 8.3.1)
 (Dec. 17, May 18, Dec. 18, 4 Marks)
- Q. Explain CCP1CON register in detail and also give its count, if we want to toggle CCP1 pin upon match.
 (Refer Section 8.3.1) (May 19, 8 Marks)

- Q. Explain compare mode of operation of PIC18 in detail. (Refer Section 8.4) (Dec. 14, Dec. 16, 4 Marks)
- Q. List the steps involved in programming PIC microcontroller in Compare mode. (Refer Section 8.4.1) (Dec. 17, May 18, Dec. 18, 4 Marks)
- Q. Mention the steps of programming Capture mode. (Refer Section 8.4.1) (May 19, 8 Marks)
- Q. Explain steps for programming compare mode of CCP1 module in PIC18f458. (Refer Section 8.4.1) (Dec. 19, 8 Marks)
- Q. Using compare mode, write the assembly language program to toggle the LED every 10 pulses. Use Timer 1 as counter. (Refer Program 8.4.2(A)) (Dec. 14, 8 Marks)
- Q. Explain capture mode of operation of PIC18 in detail. (Refer Section 8.5) (May 15, 8 Marks)
- Q. List the steps involved in programming PIC microcontroller in capture mode. (Refer Section 8.5) (May 18, Dec. 18, 4 Marks)
- Q. Explain steps for programming in capture mode. (Refer Section 8.5) (Dec. 15, 4 Marks)
- Q. Explain the steps for programming the capture mode of CCP module in PIC18 microcontroller for measuring period of pulse. (Refer Section 8.5) (Dec. 16, 8 Marks)
- Q. Write a program to generate a square wave with frequency 2.5 KHz and 50% duty cycle on the CCP1 pin using timer 3. (Refer Program 8.5.1) (Dec. 15, 8 Marks)
- Q. Write a program to measure the pulse width of signal given on CCP1 pin and give output on Ports B and D.
- OR** 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
 (Dec. 15, May 16, Dec. 16, 9 Marks)
- OR** Assume a pulse is being fed to the CCP1 pin. Using capture mode write assembly language program to measure the period of the pulse and puts the results on PORTB and PORTD. Use Timer 3 for Capture mode.
 (May 15, 8 Marks)



- OR** Write C program to measure time period of a square wave applied at CCP1 pin using capture mode of PIC18F458. Use timer3 and crystal frequency of 10 MHz. (Refer Program 8.5.1(A)) (Dec. 19, 8 Marks)
- Q.** Write a short note on PWM control DC motor using CCP mode. (Refer Section 8.6) (May 16, 8 Marks)
- Q.** List the steps involved in programming PIC microcontroller in PWM mode.
(Refer Section 8.6) (Dec. 17, 4 Marks)
- Q.** Explain steps for programming the CCP module for PWM generation.
(Refer Section 8.6.3) (Dec. 15, 4 Marks)
- Q.** Explain the steps involved in PWM programming using CCP module in PIC18F458 microcontroller
(Refer Section 8.6.3) (May 16, 8 Marks)
- Q.** Write a program to generate PWM 1 kHz frequency with a 50% duty cycle. Let N = 16.
OR 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. (Refer Program 8.6.1) (Dec. 15, 4 Marks)
- Q.** Create a 1.8 kHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL = 10 MHz
(Refer Program 8.6.1(A)) (May 15, 8 Marks)
- Q.** Create a 2 kHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL = 10 MHz.
(Refer Program 8.6.3) (Dec. 14, 4 Marks)
- Q.** Create a 1.8 KHz PWM frequency with 25% duty cycle on the CCP1 pin. Assume XTAL=10 MHz.
(Refer Program 8.6.3(A)) (May 15, 4 Marks)
- Q.** Draw an interfacing diagram and write an algorithm for DC Motor speed controller using PIC18XXX.
- OR** Draw an interfacing diagram to interface the DC motor with PIC 18FXXX for speed control using PWM with 5 kHz, 40% Duty cycle, N=4, Also write an embedded C program
- 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
- OR** Explain speed control of DC motor using Compare mode. (Refer Ex. 8.7.1) (May 15, 4 Marks)
- Q.** Explain DC motor control using PWM of CCP1 module with the help of suitable diagram
(Refer Section 8.7) (Dec. 19, 8 Marks)
- Q.** Explain interfacing of Relay and opto-isolator to PIC18F458 with the help of suitable diagram.
(Refer Section 8.7.1) (Dec. 19, 8 Marks)

□□□

9

UNIT - IV

Serial Port Programming

9.1 Introduction to Serial Communication

9.1.1 Parallel vs Serial Interface

- 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.
- Now let's compare the specified 2 methods of data transfer.

Sr. No.	Parallel	Serial
1.	Parallel lines of 8/16/32 bits. Hence 8/16/32 bits can be transmitted simultaneously.	Only 1 bit is transmitted at a time.
2.	The data transfer is comparatively faster.	The data transfer is comparatively slower.
3.	Due to so many parallel paths 'crosstalk' among different bits is possible.	No 'crosstalk' possible.
4.	This cannot be used for distant communication.	This can be used for distant communication.
5.	More parallel hardware is required.	Less parallel hardware required.
6.	It is comparatively costlier.	It is comparatively cheaper.

- In these two methods the cost of connecting two distant points, is the main factor.

- So though the parallel data transfer is faster, it is preferred for small distances only. But for long distances, serial data transfer is preferred.
- In serial data transfer the 8 bits of data is converted into serial 8 bits; using shift register (parallel in serial out mode). These serial bits are transferred on single line using serial I/O data transfer
- To transfer 8 bits of data, it will require 8 clock pulses, on the other side exactly opposite process is done. These serial 8 bits are accepted and converted to parallel form to get 8 bits of data. This process is shown in Fig. 9.1.1.

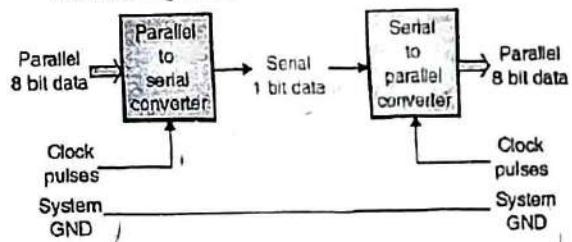


Fig. 9.1.1 : Serial I/O

9.1.2 Types of Communication Systems

The communication systems are classified on the basis of transmission :

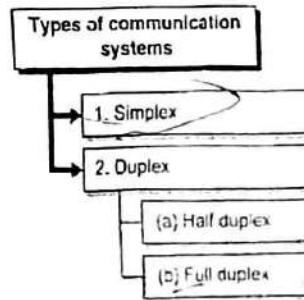


Fig. 9.1.2 : Types of communication systems

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 transmissions. It is further divided in 2 groups:

(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.

9.1.3 Baud Rate

The baud rate is the bit rate for serial communication. It is very important for serial communication that the transmitter and receiver are synchronised. For this, both the devices are programmed to the same bit rate or baud rate.

9.2 RS 232 Standard Signals used in RS 232

- In the early 1960s, a standards committee, today known as the Electronic Industries Association (EIA), developed a common interface standard for data communications equipment.
- The DTE and DCE are connected this is shown in the Fig. 9.2.1 moments to implement the RS 232 protocol.
- The DTE has to follow a particular sequence of steps before beginning with the serial transmission on TxD/RxD pins. These steps are shown in a flowchart form in Fig. 9.2.2.

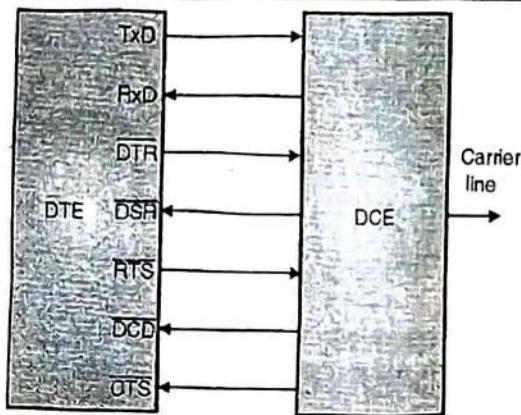


Fig. 9.2.1 : Connection between DTE and DCE

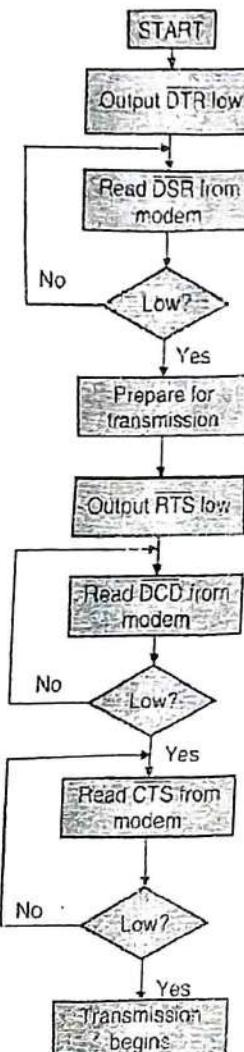


Fig. 9.2.2 : Flowchart for sequence of events in RS 232 protocol

9.3 PIC18 Connection to RS232

For connecting PIC microcontroller with a RS232 serial compatible device, we need only two pins of PIC microcontroller. These pins are the TxD (PortC.6 or RC6) pin and the RxD (PortC.7 or RC7) pin. The two pins must be connected to the MAX232 as shown in the Fig. 9.3.1.

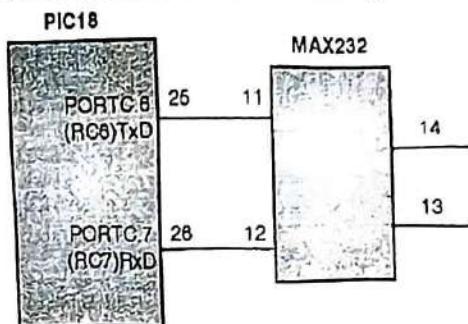


Fig. 9.3.1 : Inside MAX232 connection to the PIC18

9.4 Interfacing Serial Port and USART

- PIC microcontroller can also be directly used as USART (Universal Synchronous or Asynchronous Receiver Transmitter). The different modes permit PIC microcontroller to be used in either of the modes.
- We need to understand the different registers before interfacing serial port of PIC microcontroller.

9.4.1 SPBRG Register

- The BRG supports both the Asynchronous and Synchronous modes of the USART. 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 register) also controls the baud rate, in Synchronous mode, bit BRGH is ignored.
- The Table 9.4.1 shows the formula for computation of the baud rate for different USART modes which only apply in Master Mode (internal clock).
- Given the desired baud rate and Fosc, the nearest integer value for the SPBRG register can be calculated using the formula in Table 9.4.1.

- From this, the error in baud rate can be determined. Ex. 9.4.1 shows the calculation of the baud rate error for the following conditions:

$$F_{osc} = 16 \text{ MHz}$$

$$\text{Desired baud rate} = 9600$$

$$\text{BRGH} = 0$$

$$\text{SYNC} = 0$$

- It may be advantageous to use the high baud rate (BRGH = 1) even for slower baud clocks. This is because the $F_{osc}/(16(X + 1))$ 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.

9.4.2 Sampling

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.

Ex. 9.4.1: Calculate baud rate error.

Soln.:

$$\text{Desired baud rate} = F_{osc}/(64(X - 1))$$

Solving for X:

$$X = ((F_{osc}/\text{Desired baud rate})/64) - 1$$

$$X = ((16000000/9600)/64) - 1$$

$$X = [25.042] = 25$$

$$\begin{aligned} \text{Calculated baud rate} &= 16000000/(64(25+1)) \\ &= 9615 \end{aligned}$$

$$\begin{aligned} \text{Error} &= \frac{\text{Calculated baud rate} - \text{Desired baud rate}}{\text{Desired baud rate}} \\ &= (9615 - 9600)/9600 \\ &= 0.16\% \end{aligned}$$

...Ans.

Ex. 9.4.2: Write a program to read only numbers from input UART string.

Soln.:

```
#define CR 0x0D //define carriage return character
void gets_usart(char*ptr)
```



```

{
    char i;
    while (1)
    {
        i = gets_usart(); // read number
        if (i == CR)
            *ptr = '0';
    }
}

```

Table 9.4.1 : Baud rate formula

SYNC	BRGH = 0(Low speed)	BRGH=1(High speed)
0	(Asynchronous)Baud rate= $F_{OSC}/(64(X+1))$	Baud Rate= $F_{OSC}/(16(X+1))$
1	(Synchronous)Baud rate= $F_{OSC}/(4(X+1))$	NA

Legend : X=Value in SPBRG(0 to 255)

Table 9.4.2 : Registers associated with baud rate generator

Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Value on POR, BOR	Value on all other resets
TXSTA	CSRC	TX9	TXEN	SYNC		BRGH	TRMTS	RX9D	0000 - 010	0000 - 010
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000u
SPBRG Baud rate generator register									0000 0000	0000 0000
	R/W-0	R/W-0	R/W-0	R/W-0		U-0	R/W-0	R-1	R/W-0	
	CSRC	TX9	TXEN	SYNC		BRGH	TRMT	TX9D		
bit7										bit0

Fig. 9.4.1 : TXSTA: Transmit status and control register

bit 7 [CSRC] : Clock Source Select bit

Asynchronous mode :

Don't care.

Synchronous mode :

1 = Master mode (clock generated internally from BRG)

0 = Slave mode (clock from external source)

bit 6 [TX9] : 9-bit Transmit Enable bit

1 = Selects 9-bit transmission

0 = Selects 8-bit transmission

bit 5 [TXEN] : Transmit Enable bit

1 = Transmit enabled

0 = Transmit disabled

bit 4 [SYNC] : USART Mode Select bit

1 = Synchronous mode

0 = Asynchronous mode

bit 3 [Unimplemented] : Read as '0'

bit 2 [BRGH] : High Baud Rate Select bit

Asynchronous mode :

1 = High speed

0 = Low speed

Synchronous mode :

Unused in this mode.

bit1 [iTRMT] : Transmit Shift Register Status bit

1 = TSR empty

0 = TSR full

bit 0 [TX9D]: 9th bit of Transmit Data

Can be address/data bit or a parity bit

Table 9.4.3 : Register associated with asynchronous transmission

Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Value on POR, BOR	Value on all other resets
INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	INTOIF	RBIF	0000 000x	0000 000u
PIR1	PSP1F ⁽¹⁾	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
PIE1	PSP1E ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
IPR1	PSP1P ⁽¹⁾	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	1111 1111	1111 1111
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000u
TXREG	USART transmit register							0000 0000	0000 0000	0000 0000
TXSTA	CSRC	TX9	TXEN	SYNC	--	BRGH	TMRT	TX9D	0000 - 010	0000 - 010
SPBRG	Baud rate generator register							0000 0000	0000 0000	0000 0000

9.4.3 RCSTA (Receive Status and Control Register)

The RCSTA register is an 8-bit register used to enable the serial port to receive data, among other things. Fig. 9.4.2 describes various bits of the RCSTA register. In this section we use the 8-bit data frame.

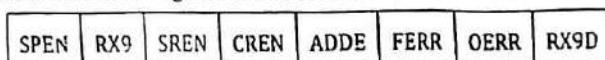


Fig 9.4.2 : RCSTA: Receive status and control register

[SPEN] D7 : Serial port enable bit

1 = Serial port enabled, which makes TX and RX pins as serial port pins

0 = Serial port disabled

[RX9] D6 : 9-bit receive enable bit

1 = Select 9-bit reception

0 = Select 8-bit reception (We use this option; therefore, D6 = 0)



- [SREN]** D5 : Single receive enable bit (not used in asynchronous mode D5 = 0)
- [CREN]** D4 : Continuous receive enable bit
1 = Enable continuous receive (in asynchronous mode)
0 = Disable continuous receive (in asynchronous mode)
- [ADDEN]** D3 : Address delete enable bit (Because used with the 9-bit data frame D3 = 0)
- [FERR]** D2 : Framing error bit
1 = Framing error
0 = No framing error
- [OERR]** D1 : Overrun error bit
1 = Overrun error
0 = No overrun error
- [TXD9]** D0 : 9th bit of receive data (Because we use the 8-bit option, we make D0 = 0)
Can be used as an address/data or a parity bit in some applications.

Table 9.4.4 : Registers associated with asynchronous reception

Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Value on POR, BOR	Value on all other resets
INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INTOIE	RBIE	TMROIF	INTOIF	RBIF	0000 000x	0000 000u
PIR1	PSPJF ⁽¹⁾	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
PIE1	PSPIE ⁽¹⁾	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
IPR1	PSPIP ⁽¹⁾	ADIP	RCIP	TXIP	SSPIP	CCP1IP	TMR2IP	TMR1IP	1111 1111	1111 1111
RCSTA	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D	0000 000x	0000 000u
RCREG	USART receive register								0000 000	0000 000
TXSTA	CSRC	TX9	TXEN	SYNC	--	BRGH	TRMT	TX9D	0000 - 010	0000 - 010
SPBRG	Baud rate generator register								0000 0000	0000 0000

9.5 USART Asynchronous Mode

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

- The Baud Rate Generator produces a clock, either x16 or x64 of the bit shift rate, depending on the BRGH bit (TXSTA register). Parity is not supported by the hardware but can be implemented in software (and stored as the ninth data bit).
- Asynchronous mode is stopped during Sleep. Asynchronous mode is selected by clearing the SYNC bit (TXSTA register).
- The USART Asynchronous module consists of the following important elements :
 - Baud rate generator
 - Sampling circuit
 - Asynchronous transmitter
 - Asynchronous receiver

9.6 USART Asynchronous Transmitter

- The USART transmitter block diagram is shown in Fig. 9.6.1. The heart of the transmitter is the Transmit (Serial) Shift Register (TSR). The TSR register obtains its data from the Read/Write Transmit Buffer register (TXREG). 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.
- The TSR 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 T_{CY}), the TXREG register is empty and flag bit TXIF (PIR1 register) is set.

- This interrupt can be enabled/disabled by setting/clearing enable bit TXIE (PIE1 register). Flag bit TXIF will be set regardless of the state of enable bit TXIE and cannot be cleared in software.
- It will reset only when new data is loaded into the TXREG register. While flag bit, TXIF, indicated the status of the TXREG register, another bit, TRMT (TXSTA register), shows the status of the TSR register. Status bit 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.

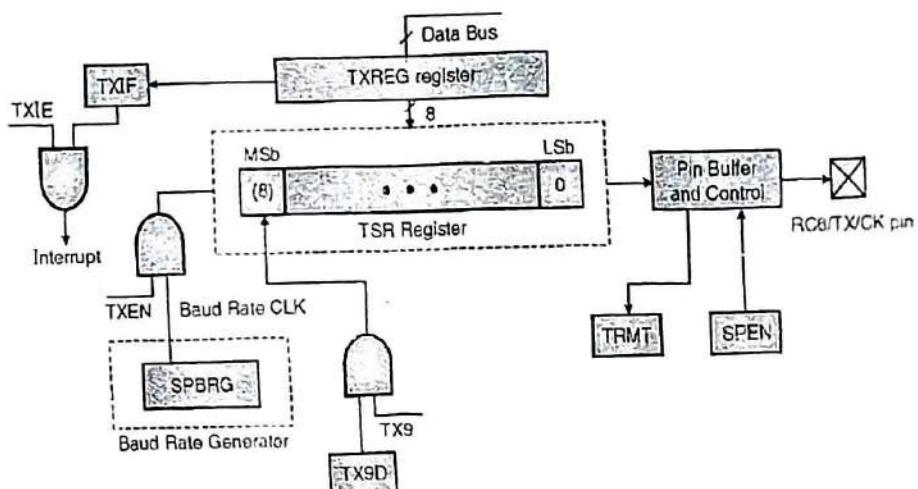


Fig. 9.6.1 : USART transmit block diagram

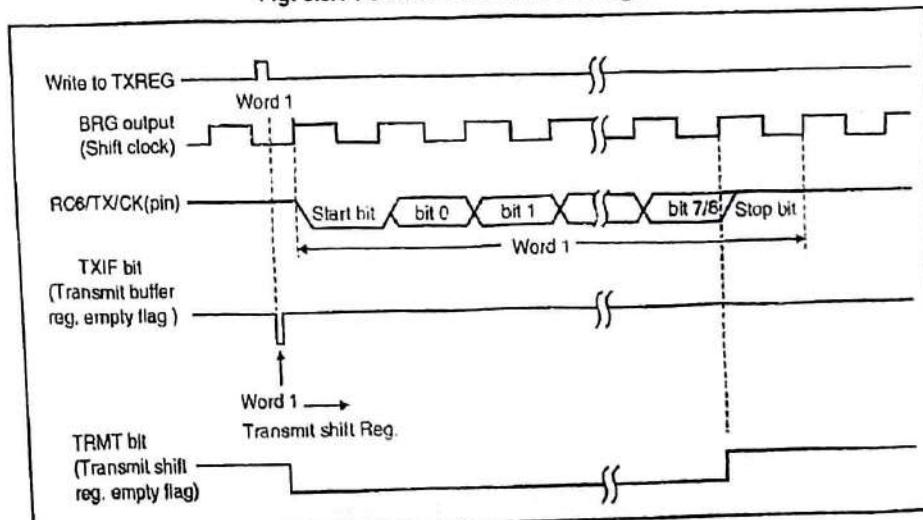


Fig. 9.6.2 : Asynchronous transmission

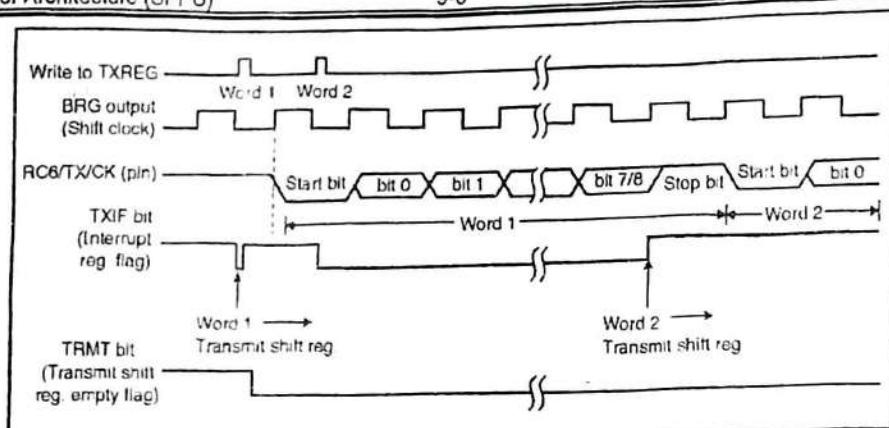


Fig. 9.6.3 : Asynchronous transmission (back to back)

9.6.1 Setting up Asynchronous Transmission

Steps to follow when setting up an asynchronous transmission :

1. Initialize the SPBRG register for the appropriate baud rate. If a high-speed baud rate is desired, set bit BRGH.
2. Enable the asynchronous serial port by clearing bit SYNC and setting bit SPEN.
3. If interrupts are desired, set enable bit TXIE.
4. If 9-bit transmission is desired, set transmit bit TX9. Can be used as address/data bit.
5. Enable the transmission by setting bit TXEN which will also set bit TXIF.
6. If 9-bit transmission is selected, the ninth bit should be loaded in bit TX9D.
7. Load data to the TXREG register (starts transmission).

9.7 USART Asynchronous Receiver

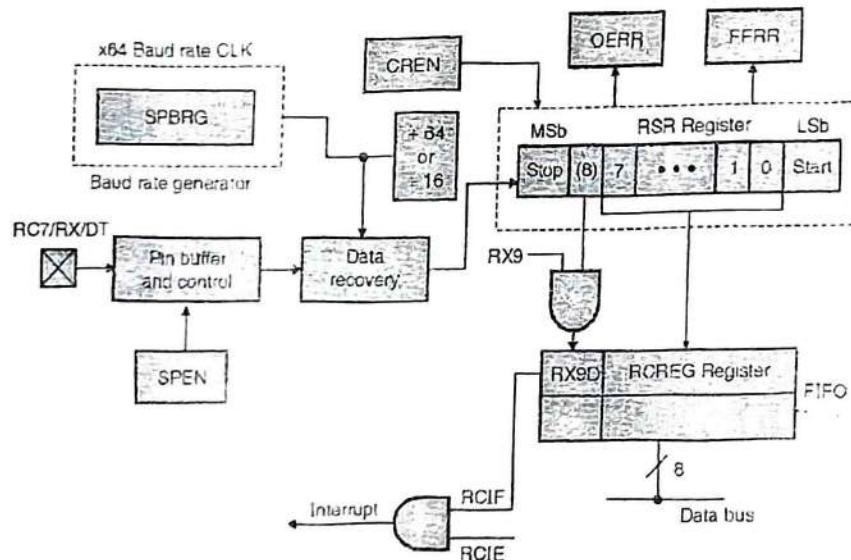


Fig. 9.7.1 : USART receive block diagram

The receiver block diagram is shown in Fig. 9.7.1. The data is received on the RC7/RX/DT pin and drives the data recovery block. The data recovery block is actually a high-speed shifter, operating at $x16$ times the baud rate, where as the main receive serial shifter operates at the bit rate or at F_{osc} . This mode would typically be used in RS-232 systems.

9.7.1 Setting up Asynchronous Reception without Address Detect Mode

Steps to follow when setting up an asynchronous reception :

1. Initialize the SPBRG register for the appropriate baud rate. If a high-speed baud rate is desired, set bit BRGH.
2. Enable the asynchronous serial port by clearing bit SYNC and setting bit SPEN.
3. If interrupts are desired, set enable bit RCIE.
4. If 9-bit reception is desired, set bit RX9.
5. Enable the reception by setting bit CREN.
6. Flag bit RCIF will be set when reception is complete and an interrupt will be generated if enable bit RCIE was set.
7. Read the RCSTA register to get the ninth bit (if enabled) and determine if any error occurred during reception.
8. Read the 8-bit received data by reading the RCREG register.
9. If any error occurred, clear the error by clearing enable bit CREN.

9.7.2 Setting up Asynchronous Reception with Address Detect

This mode would typically be used in RS-485 systems. Steps to follow when setting up an asynchronous reception with address detect enable :

1. Initialize the SPBRG register for the appropriate baud rate. If a high-speed baud rate is required, set the BRGH bit.
2. Enable the asynchronous serial port by clearing the SYNC bit and setting the SPEN bit.
3. If interrupts are required, set the RCEN bit and select the desired priority level with the RCIP bit.
4. Set the RX9 bit to enable 9-bit reception.
5. Set the ADDEN bit to enable address detect.
6. Enable reception by setting the CREN bit.

7. The RCIF bit will be set when reception is complete. The interrupt will be acknowledged if the RCIE and GIE bits are set.
8. Read the RCSTA register to determine if any error occurred during reception, as well as read bit 9 of data (if applicable).
9. Read RCREG to determine if the device is being addressed.
10. If any error occurred, clear the CREN bit.
11. If the device has been addressed, clear the ADDEN bit to allow all received data into the receive buffer and interrupt the CPU.

9.8 Programming the PIC18 to Transfer Data Serially

University Question

Q: List the steps that must be taken in programming PIC-18 microcontroller to transfer character bytes serially.

SPPU - Dec. 17, 8 Marks

To program PIC18 serial port to transfer serial data, the following steps are to be carried out :

1. TX pin (RC6) should be programmed as output pin
2. SPEN bit of RCSTA register should be set to enable serial port
3. SBREG must be loaded with the count for required baud rate.
4. TXSTA should be loaded with 20H for asynchronous mode, low baud rate and transmission enable.
5. TXREG should be loaded with the character to be transferred.
6. Before transmitting next character, we need to wait for TXIF to go high.
7. If another character is to be transmitted, goto step 5.

Program 9.8.1: Write a program to transfer the character 'H' serially at a baud rate of 9600. Assume crystal frequency of 10 MHz

SPPU - May 19, 9 Marks

Soln.:

<code>BCF TRISC,TX</code>	; Set TX pin as output pin
<code>BCF TRISC,TX</code>	; Set serial port enable
<code>MOVLW D'15'</code>	; Load the value of SPBRG in



W reg for 9600 baud rate
MOVWF SPBRG ; Load into SPBRG
MOVLW 'H' ; Load the value in W reg.
MOVWF TXSTA ; Load into TXSTA
Here:BRA Here ; wait

Program 9.8.2: Write a PIC18 program to transfer the letter 'A' serially at 9600 baud continuously. Let XTAL = 10 MHz.

SPPU - Dec. 14, May 16; 8 Marks

Soln.:

BCF TRISC,TX ; Set TX pin as output pin
BCF TRISC,TX ; Set serial port enable
MOVLW D'15' ; Load the value of SPBRG in
 ; W reg for 9600 baud rate
MOVWF SPBRG ; Load into SPBRG
MOVLW 'A' ; Load the value in W reg.
MOVWF TXSTA ; Load into TXSTA

Here:BRA Here ; wait

Program 9.8.3: Write a program for PIC18 microcontroller to transfer a letter 'T' serially and continuously at a baud rate of 9600. Use BRGH = 0. Assume crystal frequency = 10 MHz.

SPPU - Dec. 16, 8 Marks

Soln.:

BCF TRISC,TX ; Set TX pin as output pin
BCF TRISC,TX ; Set serial port enable
MOVLW D'15' ; Load the value of SPBRG in
 ; W reg for 9600 baud rate
MOVWF SPBRG ; Load into SPBRG
MOVLW 'T' ; Load the value in W reg.
MOVWF TXSTA ; Load into TXSTA

Here:BRA Here ; wait

Program 9.8.4: Write a program to transfer a letter 'P' serially and continuously at a baud rate of 4800. Assume Crystal frequency of 10 MHz.

SPPU - May 15, Dec. 15, 8 Marks

Soln.:

BCF TRISC,TX ; Set TX pin as output pin
BCF TRISC,TX ; Set serial port enable
MOVLW D'15' ; Load the value of SPBRG in
 ; W reg for 9600 baud rate
MOVWF SPBRG ; Load into SPBRG
MOVLW 'P' ; Load the value in W reg.
MOVWF TXSTA ; Load into TXSTA
Here:BRA Here ; wait

9.9 Programming the PIC18 to Receive Data Serially

To program PIC18 serial port to receive serial data, the following steps are to be carried out :

1. RX pin (RC7) should be programmed as input pin.
2. RCSTA register should be set to 90H to enable serial reception.
3. SBREG must be loaded with the count for required baud rate.
4. TXSTA should be loaded with 00H for low baud rate.
5. Wait for RCIF bit to go high to check if a byte is received.
6. When RCIF bit is set, copy the contents of RCREG
7. If another character is to be received, goto step 5.

Program 9.9.1: Write a program to receive a character and display it on LEDs on port D. Assume crystal frequency of 10 MHz and 9600 baud rate required.

SPPU - May 15; May 19, 8 Marks

Soln.:

CLRF TRISD ; Set Port D as output port
BSF TRISC,RX ; Set RX pin as input pin
MOVLW 0x90 ; Load the value of RCSTA in
 ; W reg
MOVWF RCSTA ; Load into RCSTA
MOVLW D'15' ; Load the value of SPBRG in
 ; W reg
MOVWF SPBRG ; Load into SPBRG
MOVLW 0x00 ; Load the value of TXSTA in
 ; W reg.

```

Agn :BTFSS PIR1:RCIF ; Wait for a serial
                           character to be read.

BRA Agn

MOVFF RCREG,PORTD ; Display on Port D

BRA Agn

```

9.10 SPI (Serial Peripheral Interface)

University Question

Q:- Write a short note on SPI protocol.

SPPU - Dec. 14, May 15, May 16, Dec. 16, 8 Marks

The SPI bus was developed by Motorola and is a,

1. Synchronous bus
 2. Full duplex, Bidirectional bus
 3. Four wire bus i.e. it uses 4 wires.
- It has a single master with may be single or multi slave system which is commonly used for high speed serial communication between microcontroller and peripheral devices, such as EEPROMs, data convertors and display drivers.
 - Although we may also have a system with multiple masters in the SPI system, but one of these works as a master at any given time. As discussed earlier, there are SPI uses four signal lines for communication. They are :
 1. **Master Out Slave In (MOSI)** : As the name says this signal carries the data from the master to the slave. In some cases this pin is also referred to as Slave Input/Slave Data In (SI/SDI).
 2. **Master In Slave Out (MISO)** : Similarly this pin is used to carry data from the slave to the master. In some cases this pin is also referred to as Slave Output/Slave Data Out (SO/SDO).
 3. **Serial Clock (SCLK)** : This is the synchronizing pulse or the clock signal for synchronizing.
 4. **Slave Select (SS)** : It is an active low signal used to select the slave device.

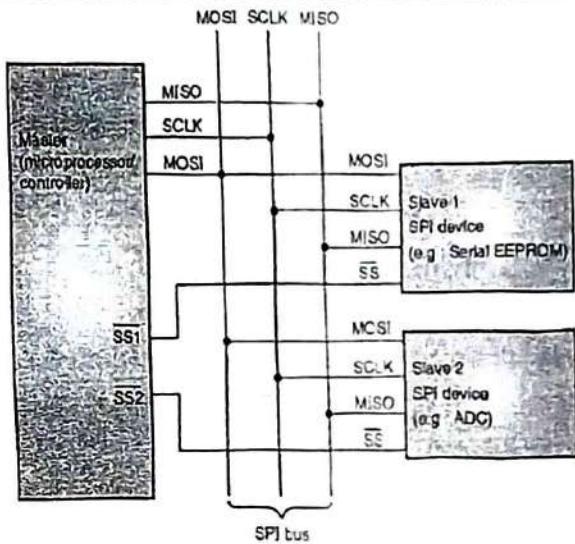


Fig. 9.10.1 : SPI bus interfacing

- The clock is supplied by the master to the slave. The master also selects the slave device by enabling its slave select pin. For those slave devices that are not selected i.e. the slave select pin is active high, the MISO pin remains tri-stated.
- The SPI devices have configuration registers to configure the devices. A register called as serial peripheral control register has fields for various parameters like master/slave selection for the device, baud rate selection for communication, clock signal selection etc.
- The principle of operation used in SPI is a shift register concept. In this, the data is transmitted from one of the devices (master or slave) to the other device (slave or master) with both of them having this operation done using shift register.
- Once a data byte is transmitted, the same is copied in the receiver and also transmitted back through the shift register to the transmitter. The transmitter checks for the data send earlier.
- If the data matches, then no issue else it sends a message indicating the data received is corrupted. Also the same data is then sent again to the receiver. Hence the two pins MOSI and MISO.
- For application that requires transfer of data in streams, SPI bus is most suitable. SPI has a limitation of not supporting the acknowledgement mechanism.

9.10.1 PIC18F SPI Bus

The SPI mode allows 8 bits of data to be synchronously transmitted and received simultaneously. All four modes of SPI are supported. To accomplish communication, typically three pins are used:

- o Serial Data Out (SDO) = RC5/SDO
- o Serial Data In (SDI) = RC4/SDI/SDA
- o Serial Clock (SCK) = RC3/SCK/SCL
- Additionally, a fourth pin may be used when in a slave mode of operation:
- o Slave Select (SS) = RA5/AN4/M/LVDIN
- Fig. 9.10.2 shows the block diagram of the MSSP module when operating in SPI mode.

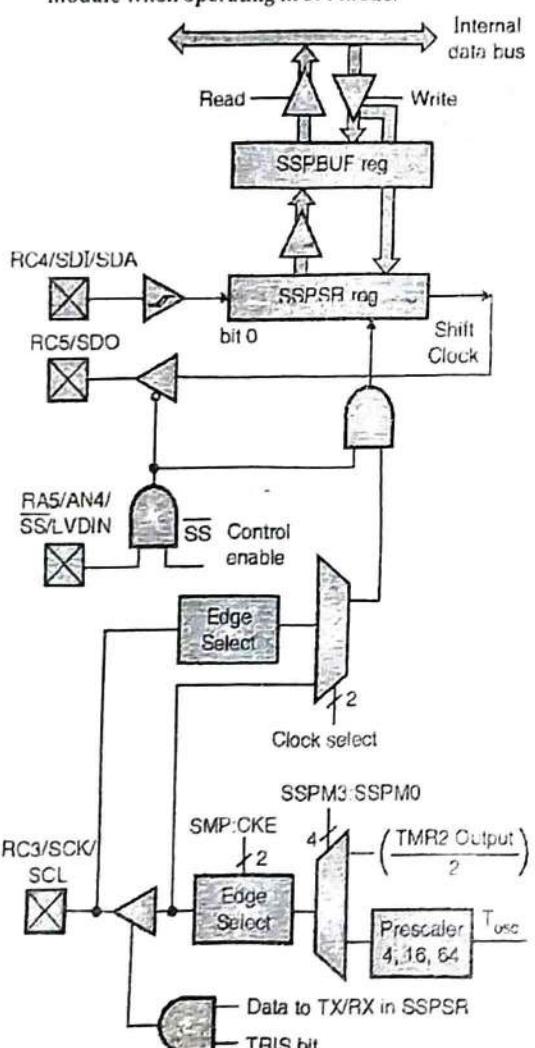


Fig. 9.10.2 : MSSP block diagram (SPITM mode)

9.10.2 Registers

- The MSSP module has four registers for SPI mode operation. These are:
 1. MSSP Control Register 1 (SSPCON1)
 2. MSSP Status Register (SSPSTAT)
 3. Serial Receive/Transmit Buffer (SSPBUF)
 4. MSSP Shift Register (SSPSR) — Not directly accessible
- SSPCON1 and SSPSTAT are the control and status registers in SPI mode operation. The SSPCON1 register is 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. 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.

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/A	P	S	R/W	UA	BF
bit 7							bit 0

Fig. 9.10.3 : SSPSTAT : MSSP status register (SPI mode)

bit 7 [SMP] : Sample bit

SPI Master mode:

- 1 = Input data sampled at end of data output time
- 0 = Input data sampled at middle of data output time

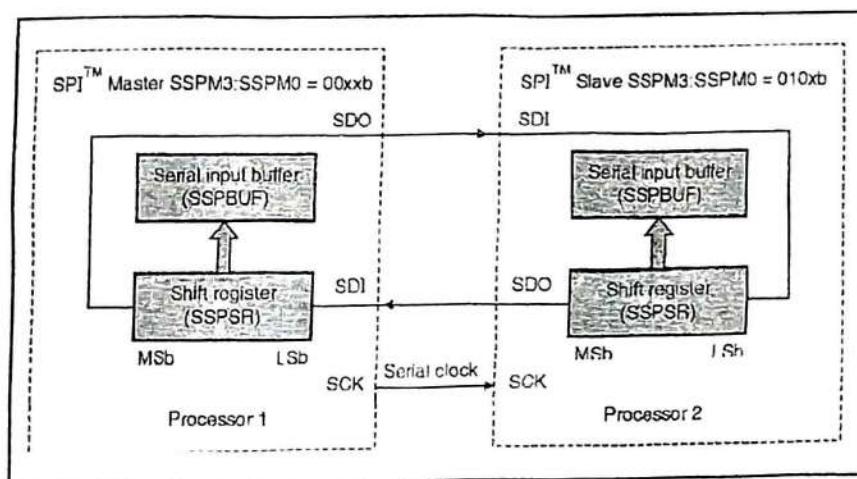
SPI Slave mode :

SMP must be cleared when SPI is used in Slave mode.

bit 6 [CKE] : SPI Clock Edge Select bit

- 1 = Transmit occurs on transition from active to Idle clock state

- 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. 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.
- This double-buffering of the received data (SSPBUF) allows the next byte to start reception before reading the data that was just received.
- 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. User software must clear the WCOL bit so that it can be determined if the following write(s) to the SSPBUF register completed successfully.
- When the application software is expecting to receive valid data, the SSPBUF should be read before the next byte of data to transfer is written to the SSPBUF.
- Buffer Full bit, BF (SSPSTAT<0>), indicates when SSPBUF has been loaded with the received data (transmission is complete).



9.10.5 : Programming SPI port

- When the SSPBUF is read, the BF bit is cleared. This data may be irrelevant if the SPI is only a transmitter.
- Generally, the MSSP interrupt is used to determine when the transmission/reception has completed.
- The SSPBUF must be read and/or written. If the interrupt method is not going to be used, then software polling can be done to ensure that a write collision does not occur.
- Fig. 9.10.5 shows the loading of the SSPBUF (SSPSR) for data transmission.
- The SSPSR is not directly readable or writable and can only be accessed by addressing the SSPBUF register. Additionally, the MSSP Status register (SSPSTAT) indicates the various status conditions.
- Fig. 9.10.5 shows a typical connection between two microcontrollers. The master controller (Processor 1) initiates the data transfer by sending the SCK signal. Data is shifted out of both shift registers on their programmed clock edge and latched on the opposite edge of the clock.
- Both processors should be programmed to the same Clock Polarity (CKP), then both controllers would send and receive data at the same time. Whether the data is meaningful (or dummy data) depends on the application software.

- This leads to three scenarios for data transmission:
 - o Master sends data : Slave sends dummy data
 - o Master sends data : Slave sends data
 - o Master sends dummy data : Slave sends data

9.11 I²C Bus

- The I²C (Inter-IC Communication) bus is a bidirectional serial communication protocol used for communication between the integrated circuits (ICs).
- It is developed by Phillips and is a synchronous protocol for serial communication. The specialty of this bus is that it requires only one line for bidirectional data and another for clock signal.
- There can be one master and 127 other devices or slaves, thus requiring 7-bit address to select a device.
- The communication has a packet of 20 bits, wherein each packet comprises of various fields explained below:
 1. One Start bit indicating the start of the packet.
 2. Seven address bits indicating the address of the device for which the data packet is meant.
 3. One bit indicating read or write cycle
 4. One bit indicating if the data packet is an acknowledgement of previously transferred data or is a data packet containing data.
 5. Eight bits of data.
 6. One bit indicating acknowledgement required or not for the given data. This bit is also called as NACK as it is negative logic bit
 7. Finally, one stop bit indicating end of packet.

9.12 I²C Mode

The MSSP module in I²C mode fully implements all master and slave functions (including general call support) 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:

1. Serial clock (SCL) - RC3/SCK/SCL
2. Serial data (SDA) - RC4/SDI/SDA

The user must configure these pins as inputs or outputs through the TRISC<4:3> bits.

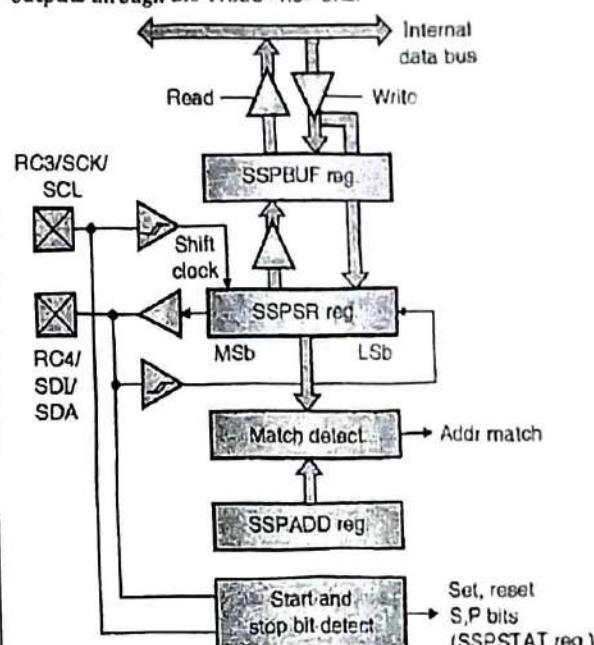


Fig. 9.12.1 : MSSP block diagram (I²C mode)

9.12.1 Registers

The MSSP module has six registers for I²C operation. These are:

1. MSSP Control Register 1 (SSPCON1)
 2. MSSP Control Register 2 (SSPCON2)
 3. MSSP Status Register (SSPSTAT)
 4. Serial Receive/Transmit Buffer (SSPBUF)
 5. MSSP Shift Register (SSPSR) — Not directly accessible
 6. 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.

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/Ā	P	S	R/Ā	UA	BF
bit 7				bit 0			

Fig. 9.12.2 : SSPSTAT: MSSP status register
(I²C mode)

bit 7 [SMP] : Slew Rate Control bit

In Master or Slave mode :

- 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)

bit 6 [CKE] : SMBus Select bit

In Master or Slave mode :

- 1 = Enable SMBus specific inputs
0 = Disable SMBus specific inputs

bit 5 [D/Ā] : Data/Address bit

In Master mode:

Reserved.

In Slave mode:

- 1 = Indicates that the last byte received or transmitted was data
0 = Indicates that the last byte received or transmitted was address

bit 4 [P] : Stop bit

- 1 = Indicates that a Stop bit has been detected last
0 = Stop bit was not detected last

bit 3 [S] : Start bit

- 1 = Indicates that a Start bit has been detected last
0 = Start bit was not detected last

bit 2 [R/Ā] : Read/ Write Information bit
(I²C mode only)

In Slave mode :

- 1 = Read
0 = Write

In Master mode :

- 1 = Transmit is in progress
0 = Transmit is not in progress

bit 1 [UA] : Update Address bit (10-bit Slave mode only)

- 1 = Indicates that the user needs to update the address in the SSPADD register
0 = Address does not need to be updated

bit 0 [BF] : Buffer Full Status bit

In transmit mode :

- 1 = Receive complete, SSPBUF is full
0 = Receive not complete, SSPBUF is empty

In Receive mode:

- 1 = Data transmit in progress (does not include the ACK and Stop bits). SSPBUF is full
0 = Data transmit complete (does not include the ACK and Stop bits). SSPBUF is empty

| R/W-0 |
|-------|-------|-------|-------|-------|-------|-------|-------|
| WCOL | SSPOV | SSPEN | CKP | SSPM3 | SSPM2 | SSPM1 | SSPM0 |
| bit 7 | | | | bit 0 | | | |

Fig. 9.12.3 : SSPCON1: MSSP control register 1
(I²C mode)

bit 7 [WCOL] : Write Collision Detect bit

In Master Transmit mode:

- 1 = A write to the SSPBUF register was attempted while the I²C conditions were not valid for a transmission to be started (must be cleared in software)
0 = No collision

In Slave Transmit mode:

- 1 = The SSPBUF register is written while it is still transmitting the previous word (must be cleared in software)
0 = No collision

In receive mode (Master or slave mode)

This is a "don't care" bit

bit6 [SSPOV] : Receive Overflow Indicator bit**In Receive mode :**

1 = A byte is received while the SSPBUF register is still holding the previous byte (must be cleared in software)

0 = No overflow

In Transmit mode:

This is a "don't care" bit in Transmit mode.

bit5 [SSPEN] : Synchronous Serial Port Enable bit

1 = Enables the serial port and configures the SDA and SCL pins as the serial port pins

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

bit4 [CKP] : SCK Release Control bit**In Slave mode:**

1 = Release clock

0 = Holds clock low (clock stretch), used to ensure data setup time

In Master mode :

Unused in this mode.

bit3-0 [SSPM3:SSPM0] : Synchronous Serial Port Mode Select bits

1111 = I²C Slave mode, 10-bit address with Start and Stop bit interrupts enabled

1110 = I²C Save mode, 7-bit address with Start and Stop bit interrupts enabled

1011 = I²C Firmware Controlled Master mode (Slave Idle)

1000 = I²C Master mode, clock = F_{osc}/(4*(SSPADD+1))

0111 = I²C Slave mode, 10-bit address

0110 = I²C Slave mode, 7-bit address

R/W- R/W-0 R/W-0 R/W-0 R/W- R/W- R/W- R/W-0
0 0 0 0

GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
------	---------	-------	-------	------	-----	------	-----

bit 7

bit 0

Fig. 9.12.4 : SSPCON2: MSSP control register 2
(I²C mode)

bit 7 [GCEN] : General Call Enable bit (Slave mode only)

1 = Enable interrupt when a general call address (0000h) is received in the SSPSR

0 = General call address disabled

bit 6 [ACKSTAT] : Acknowledge Status bit (Master Transmit mode only)

1 = Acknowledge was not received from slave

0 = Acknowledge was received from slave

bit 5 ACKDT : Acknowledge Data bit (Master Receive mode only)

1 = Not Acknowledge

0 = Acknowledge

bit 4 [ACKEN] : Acknowledge Sequence Enable bit (Master Receive mode only)

1 = Initiate Acknowledge sequence on SDA and SCL pins and transmit ACKDT data bit. Automatically cleared by hardware.

0 = Acknowledge sequence Idle

bit 3 [RCEN] : Receive Enable bit (Master Mode only)

1 = Enables Receive mode for I²C

0 = Receive Idle

bit 2 [PEN] : Stop Condition Enable bit (Master mode only)

1 = Initiate Stop condition on SDA and SCL pins. Automatically cleared by hardware.

0 = Stop condition Idle

bit 1 [RSEN] : Repeated Start Condition Enable bit

(Master mode only)

1 = Initiate Repeated Start condition on SDA and SCL pins. Automatically cleared by hardware.

0 = Repeated Start condition Idle

bit 0 [SEN] : Start Condition Enable/Stretch Enable bit**In Master mode:**

1 = Initiate Start condition on SDA and SCL pins. Automatically cleared by hardware.

0 = Start condition Idle

In Slave mode :

1 = Clock stretching is enabled for both slave transmit and slave receive (stretch enabled)

0 = Clock stretching is enabled for slave transmit only (Legacy mode)



9.12.2 Operation

- 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:
 1. I²C Master mode, clock = F_{osc}/4 (SSPADD +1)
 2. I²C Slave mode (7-bit address)
 3. I²C Slave mode (10-bit address)
 4. I²C Slave mode (7-bit address) with Start and Stop bit interrupts enabled
 5. I²C Slave mode (10-bit address) with Start and Stop bit interrupts enabled
 6. 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 to inputs by setting the appropriate TRISC bits.
- To ensure proper operation of the module, pull-up resistors must be provided externally to the SCL and SDA pins.

9.12.3 Slave Mode

- 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. 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:
 - o The Buffer Full bit, BF (SSPSTAT<0>), was set before the transfer was received.
 - o 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 bit SSPIF (PIR1 <3>) is set. The BF bit is cleared by reading the SSPBUF register, while bit SSPOV is cleared through software.
- The SCL clock input must have a minimum high and low for proper operation. The high and low times of the I²C specification, as well as the requirement of the MSSP module, are shown in timing parameter #100 and parameter #101.

9.12.3(A) Addressing

- 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. MSSP Interrupt Flag bit, SSPIF (PIR1<3>), is set (interrupt is generated if enabled) on the falling edge of the ninth SCL pulse.
- In 10-bit Address mode, two address bytes need to be received by the slave.
- The five Most Significant bits (MSbs) of the first address byte specify if this is a 10-bit address. Bit R/W (SSPSTAT<2>) must specify a write so the slave device will receive the second address byte. For a 10-bit address, the first byte would equal 11110 A9 A8 0, where A9 and A8 are the two MSbs of the address.
- The sequence of events for 10-bit address is as follows, with steps 7 through 9 for the slave transmitter:

1. Receive first (high) byte of address (bits SSPIF, BF and bit UA (SSPSTAT<1>) are set).
2. Update the SSPADD register with second (low) byte of address (clears bit UA and releases the SCL line).
3. Read the SSPBUF register (clears bit BF) and clear flag bit SSPIF.
4. Receive second (low) byte of address (bits SSPIF, BF and UA are set).
5. Update the SSPADD register with the first (high) byte of address. If match releases SCL line, this will clear bit UA.
6. Read the SSPBUF register (clears bit BF) and clear flag bit SSPIF.
7. Receive Repeated Start condition.
8. Receive first (high) byte of address (bits SSPIF and BF are set).
9. Read the SSPBUF register (clears bit BF) and clear flag bit SSPIF.

9.12.3(B) 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).
- When the address byte overflow condition exists, then the no Acknowledge (ACK) pulse is given. An overflow condition is defined as either bit BF (SSPSTAT<0> is set or bit SSPOV (SSPCON1 <6>) is set).
- An MSSP interrupt is generated for each data transfer byte. Flag bit SSPIF (PIR1 <3>) must be cleared in software. The SSPSTAT register is used to determine the status of the byte.
- If SEN is enabled (SSPCON2<0> = 1), RC3/SCK/SCL will be held low (clock stretch) following each data transfer. The clock must be released by setting bit CKP (SSPCON1<4>).

9.12.3(C) 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 RC3/SCK/SCL is held low regardless of SEN. By stretching the clock, the master will be unable to assert another clock pulse until the slave is done preparing the transmit data.
- The transmit data must be loaded into the SSPBUF register, which also loads the SSPSR register. Then, pin RC3 / SCK/SCL should be enabled by setting bit CKP (SSPCON1 <4>). The eight data bits are shifted out on the falling edge of the SCL input. This ensures that the SDA signal is valid during the SCL high time.
- The ACK pulse from the master-receiver is latched on the rising edge of the ninth SCL input pulse. If the SDA line is high (not ACK), then the data transfer is complete. In this case, when the ACK is latched by the slave, the slave logic is reset (resets SSPSTAT register) and the slave monitors for another occurrence of the Start bit. If the SDA line was low (ACK), the next transmit data must be loaded into the SSPBUF register. Again, pin RC3/SCK/SCL must be enabled by setting bit CKP.
- An MSSP interrupt is generated for each data transfer byte. The SSPIF bit must be cleared in software and the SSPSTAT register is used to determine the status of the byte. The SSPIF bit is set on the falling edge of the ninth clock pulse.

9.12.4 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.

- Master mode of 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.

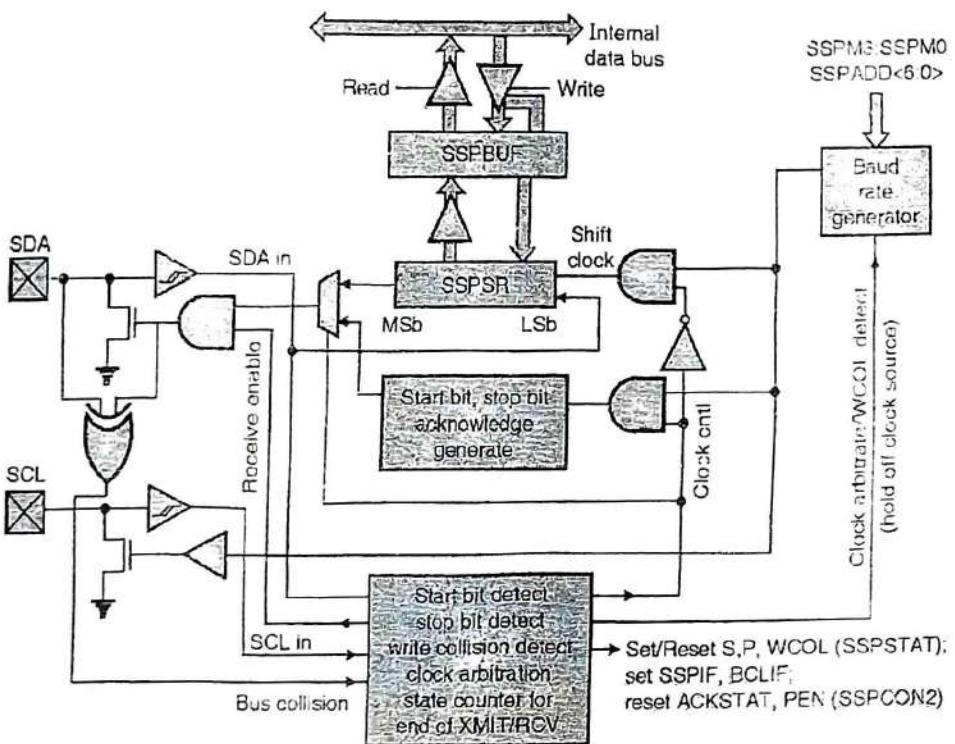


Fig. 9.12.5 : MSSP block diagram (I²CTM master mode)

A typical transmits 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 8 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 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.

9.12.5 Baud Rate Generator

- In I²C Master Mode, the Baud Rate Generator (BRG) reload value is placed in the lower 7 bits of the SSPADD register (Fig. 9.12.6).
- When a write occurs to SSPBUF, the Baud Rate Generator will automatically begin counting. The BRG counts down to 0 and stops until another reload has taken place. The BRG count is decremented twice per instruction cycle (T_{CY}) on the Q2 and Q4 clocks. In I²C Master Mode, the BRG is reloaded automatically.
- Once the given operation is complete (i.e., transmission of the last data bit is followed by (ACK), the internal clock will automatically stop counting and the SCL pin will remain in its last state.

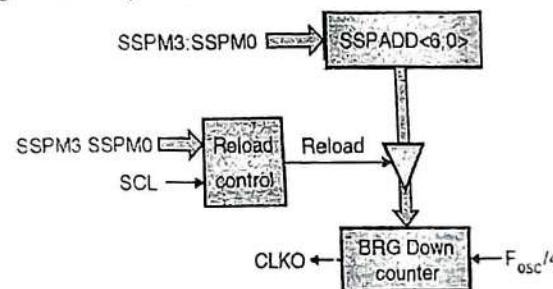


Fig. 9.12.6 : Baud rate generator block diagram

9.13 Comparison of I²C and SPI

SPI has a major advantage over I²C, that SPI is full duplex communication while I²C is half duplex. Hence SPI communication is preferred over I²C.

Table 9.13.1 : Comparison of I²C and SPI

Sr. No.	I ² C	SPI
1.	I ² C can be multi-master and multi-slave, which means there can be more than one master and slave attached to the I ² C bus	SPI can be multi-slave but does not support a multi-master serial protocol, that means there can be only one master attached to SPI bus.
2.	I ² C is half-duplex communication protocol.	SPI is a full duplex communication protocol.
3.	I ² C has the feature of clock stretching, that means if the slave cannot able to send fast data as fast enough then it suppresses the clock to stop the communication.	Clock stretching is not the feature of SPI.

9.14 Exam Pack (Review and University Questions)

- Q. List the steps that must be taken in programming PIC 18 microcontroller to transfer character bytes serially.
(Refer Section 9.8) **(Dec. 17, 8 Marks)**
- Q. Write a program to transfer the character 'H' serially at a baud rate of 9600. Assume crystal frequency of 10 MHz.
(Refer Program 9.8.1) **(May 19, 9 Marks)**
- Q. Write a PIC18 program to transfer the letter 'A' serially at 9600 baud continuously. Let XTAL = 10 MHz.
(Refer Program 9.8.2) **(Dec. 14, May 16, 8 Marks)**
- Q. Write a program for PIC18 microcontroller to transfer a letter 'T' serially and continuously at a baud rate of 9600. Use BRGH = 0. Assume crystal frequency = 10 MHz **(Refer Program 9.8.3)** **(Dec. 16, 8 Marks)**
- Q. Write a program to transfer a letter 'P' serially and continuously at a baud rate of 4800. Assume Crystal frequency of 10 MHz. **(Refer Program 9.8.4)** **(May 15, Dec. 15, 8 Marks)**
- Q. Write a program to receive a character and display it on LEDs on port D. Assume crystal frequency of 10 MHz and 9600 baud rate required. **(Refer Program 9.9.1)** **(May 15, May 19, 8 Marks)**
- Q. Write a short note on SPI protocol. **(Refer Section 9.10)** **(Dec. 14, May 15, May 16, Dec. 16, 8 Marks)**

□□□

6
X
S
V
u.s
X
S
23mW

10

UNIT V

PIC Interfacing - III

10.1 Interfacing DAC

PIC microcontroller doesn't have an internal DAC. Hence we need an external DAC or the same task of DAC can also be done using PWM. In this section we will see how to interface external DAC to PIC microcontroller.

10.1.1 Features of DAC 0808

Features of DAC 0808 are following:

- 1. Relative accuracy : $\pm 0.19\%$ error maximum.
- 2. Full scale current match : ± 1 LSB typical.
- 3. Fast settling time : 150 ns typical.
- 4. Non inverting digital inputs are TTL and CMOS compatible.
- 5. High speed multiplying input slew rate : 8 mA/ μ s.
- 6. Power supply voltage range : $\pm 4.5V$ to $\pm 18V$.
- 7. Low power consumption : 33 mW @ $\pm 5V$.

10.1.2 Pin Configuration and Functional Block Diagram

Reference amplifier drive and compensation

- The reference amplifier provides a voltage at pin 14 for converting the reference voltage to a current, and a turn-around circuit or current mirror for feeding the ladder.
- The reference amplifier input current, I_{14} , must always flow into pin 14, regardless of the set-up method or reference voltage polarity. The reference voltage source supplies the full current I_{14} .
- For bipolar reference signals, as in the multiplying mode, R15 can be tied to a negative voltage corresponding to the minimum input level.
- It is possible to eliminate R15 with only a small sacrifice in accuracy and temperature drift.

Digital to analog converter

- The compensation capacitor value must be increased with increases in R14 to maintain proper phase margin; for R14 values of 1, 2.5 and 5 k Ω , minimum capacitor values are 15, 37 and 75 pF. The capacitor may be tied to either V_{EE} or ground, but using V_{EE} increases negative supply rejection.
- A negative reference voltage may be used if R14 is grounded and the reference voltage is applied to R15. A high input impedance is the main advantage of this method. Compensation involves a capacitor to V_{EE} on pin 16, using the values of the previous paragraph.
- The negative reference voltage must be at least 4V above the V_{EE} supply. Bipolar input signals may be handled by connecting R14 to a positive reference voltage equal to the peak positive input level at pin 15. When a DC reference voltage is used, capacitive bypass to ground is recommended.
- The 5V logic supply is not recommended as a reference voltage. If a well regulated 5V supply which drives logic is to be used as the reference, R14 should be decoupled by connecting it to 5V through another resistor and bypassing the junction of the 2 resistors with 0.1 μ F to ground.
- For reference voltages greater than 5V, a clamp diode is recommended between pin 14 and ground. If pin 14 is driven by a high impedance such as a transistor current source, none of the above compensation methods apply and the amplifier must be heavily compensated, decreasing the overall bandwidth.

Output voltage range

- The voltage on pin 4 is restricted to a range of - 0.55 to 0.4V when $V_{EE} = - 5V$ due to the current switching methods employed in the DAC0808.
- The negative output voltage compliance of the DAC0808 is extended to - 5V where the negative supply voltage is more negative than - 10V.



- Using a full-scale current of 1.992 mA and load resistor of $2.5\text{ k}\Omega$ between pin 4 and ground will yield a voltage output of 256 levels between 0 and -4.980V .
- Floating pin 1 does not affect the converter speed or power dissipation. However, the value of the load resistor determines the switching time due to increased voltage swing. Values of R_L up to $500\text{ }\Omega$ do not significantly affect performance, but a $2.5\text{ k}\Omega$ load increases worst-case settling time to $1.2\text{ }\mu\text{s}$ (when all bits are switched ON).

Output current range

The output current maximum rating of 4.2 mA may be used only for negative supply voltages more negative than -8V , due to the increased voltage drop across the resistors in the reference current amplifier.

Accuracy

- Absolute accuracy is the measure of each output current level with respect to its intended value, and is dependent upon relative accuracy and full-scale current drift. Relative accuracy is the measure of each output current level as a fraction of the full-scale current.
- The relative accuracy of the DAC0808 is essentially constant with temperature due to the excellent temperature tracking of the monolithic resistor ladder. The reference current may drift with temperature, causing a change in the absolute accuracy of output current.

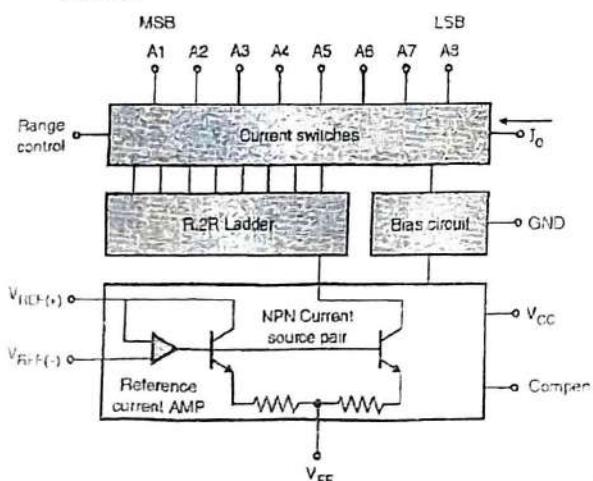


Fig. 10.1.1

- However, the DAC0808 has a very low full-scale current drift with temperature.

- The DAC0808 series is guaranteed accurate to within $\pm 1/2$ LSB at a full-scale output current of 1.992 mA.

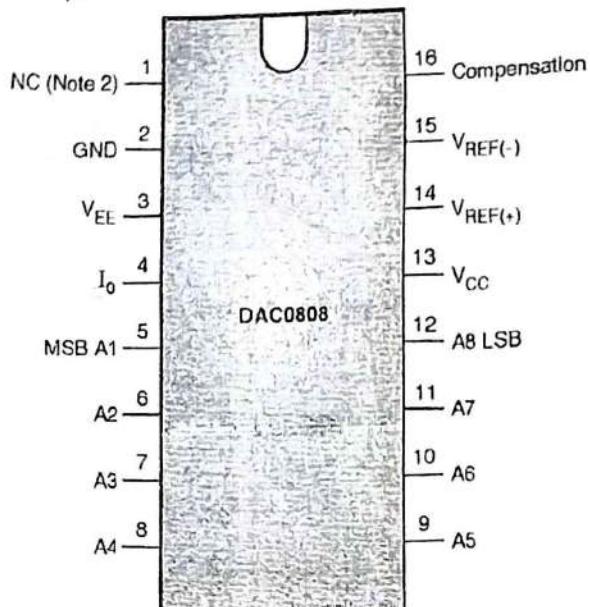


Fig. 10.1.2 : Dual-In-Line Package

- This corresponds to a reference amplifier output current drive to the ladder network of 2 mA, with the loss of 1 LSB ($8\text{ }\mu\text{A}$) which is the ladder remainder shunted to ground.
- The input current to pin 14 has a guaranteed value of between 1.9 and 2.1 mA, allowing some mismatch in the NPN current source pair. The 12-bit converter is calibrated for a full-scale output current of 1.992 mA.
- This is an optional step since the DAC0808 accuracy is essentially the same between 1.5 and 2.5 mA. Then the DAC0808 circuits' full-scale current is trimmed to the same value with R14 so that a zero value appears at the error amplifier output.
- The counter is activated and the error band may be displayed on an oscilloscope, detected by comparators, or stored in a peak detector.
- Two 8-bit D-to-A converters may not be used to construct a 16-bit accuracy D-to-A converter. 16-bit accuracy implies a total error of $\pm 1/2$ of one part in 65,536 or $\pm 0.00076\%$, which is much more accurate than the $\pm 0.019\%$ specification provided by the DAC0808.

Multiplying accuracy

- The DAC0808 may be used in the multiplying mode with 8-bit accuracy when the reference current is varied over a range of 256:1. If the reference current in the multiplying mode ranges from 16 μ A to 4 mA, the additional error contributions are less than 1.6 μ A. This is well within 8-bit accuracy when referred to full-scale.
- A monotonic converter is one which supplies an increase in current for each increment in the binary word. Typically, the DAC0808 is monotonic for all values of reference current above 0.5 mA. The recommended range for operation with a DC reference current is 0.5 to 4 mA.

Settling time

- The worst-case switching condition occurs when all bits are switched ON, which corresponds to a low-to-high transition for all bits.

- This time is typically 150 ns for settling to within $\pm 1/2$ LSB, for 8-bit accuracy, and 100 ns to $1/2$ LSB for 7 and 6-bit accuracy. The turn OFF is typically under 100 ns.
- These times apply when $R_L \leq 500\Omega$ and $C_o \leq 25 \text{ pF}$. Extra care must be taken in board layout since this is usually the dominant factor in satisfactory test results when measuring settling time. Short leads, 100 μ F supply bypassing for low frequencies, and minimum scope lead length are all mandatory.

10.1.3 Interfacing DAC with PIC18F458

University Question

Q. Explain with a neat diagram, interfacing of DAC 0808 with PIC microcontroller.

SPPU - Dec. 14, Dec. 16, 4 Marks

The interfacing of PIC microcontroller with DAC can be done as shown in the Fig. 10.1.3. The DAC parallel input can be connected to any 8-bit port of PIC microcontroller. The analog output of the DAC can be connected to a inverting amplifier and the output can be obtained at the V_o pin as shown in the Fig. 10.1.3.

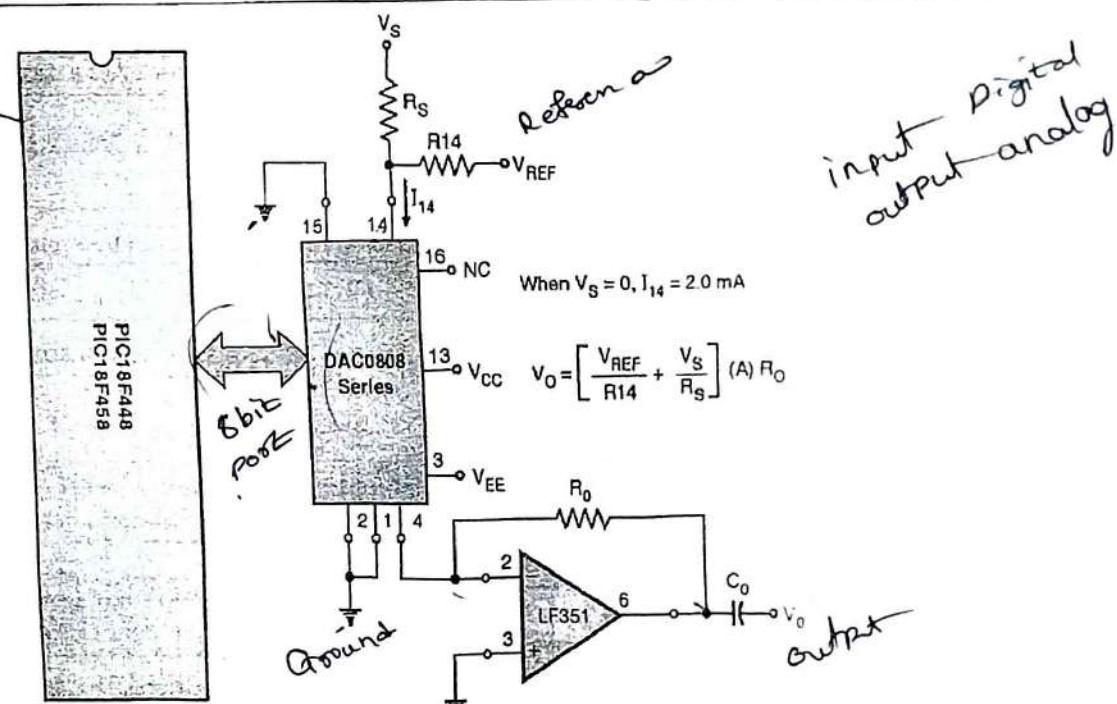


Fig. 10.1.3 : Interfacing DAC with PIC18F458

10.1.4 Programming PIC to Interface with DAC0808

Ex. 10.1.1: Design a PIC18 based system to interface DAC. Write the C and an assembly language programs to generate a sine wave.

Sol.: Fig. 10.1.3 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

Calculation of array elements

Sr. No.	Angle In degrees (θ)	$\sin (\theta)$	$\text{Count} = 2.5 + \frac{(2.5 * \sin \theta)}{5}$	
0	0	0	2.5	128
1	7.5	0.130578428	2.826446071	144
2	15	0.258920827	3.147302068	161
3	22.5	0.382829457	3.457073643	176
4	30	0.500182502	3.750456255	191
5	37.5	0.608970405	4.022426013	205
6	45	0.707330278	4.268325695	218
7	52.5	0.793577803	4.483944508	229
8	60	0.866236075	4.665590188	238
9	67.5	0.924060891	4.810152227	245
10	75	0.966052056	4.915155141	251
11	82.5	0.991520342	4.978800856	254
12	90	0.9999998	4.9999995	255
13	97.5	0.991355227	4.978388068	254
14	105	0.965734654	4.914336634	251
15	112.5	0.923576807	4.808942018	245
16	120	0.8656036	4.664008999	238
17	127.5	0.792807767	4.482019417	229
18	135	0.706435867	4.266089666	218
19	142.5	0.607966935	4.019917336	205

Sr. No.	Angle in degrees (θ)	$\sin (\theta)$	$\text{Count} = 2.5 + \frac{(2.5 * \sin \theta)}{5}$	count*255/5
20	150	0.499087156	3.74771789	191
21	157.5	0.381660992	3.45415248	176
22	165	0.257699252	3.144248131	160
23	172.5	0.129324662	2.823311654	144
24	180	-0.001264489	2.496838778	127
25	187.5	-0.131831986	2.170420034	111
26	195	-0.260141968	1.849645029	94
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	4.497E-06	0
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	0.98269377	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	111
48	360	0.002528976	2.50632244	128

C Program

```
# include<P18F458.h>

char array [] =
{128,144,161,176,191,205,218,229,238,245,251,
254,255,254,251,245,238,229,218,205,191,176,160,
144,127,111,94,79,64,50,37,26,17,10,4,1,0,1,4,10,
17,27,38,50,64,79,95,111,128};

// 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];
    }
}
```

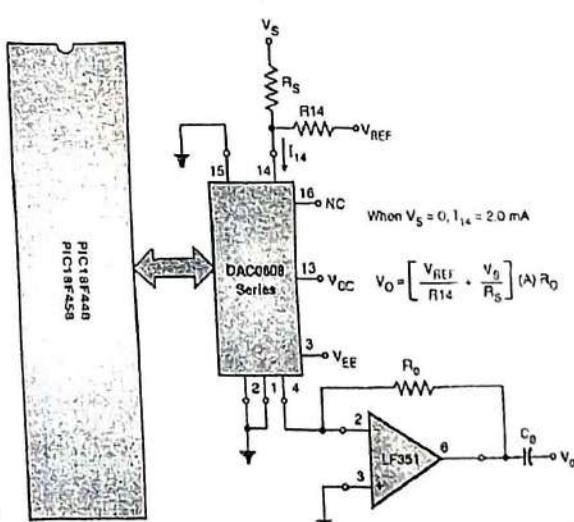
Assembly Program

Label	Instruction	Comments
	CLRF TRISD	Program PortD as output port
	MOVLW 0xFF	
again:	MOVWF PORTD	Move into PortD from WREG
	SUBLW 0x01	Decrement value from WREG
	BRA again	

C18 program

```
#include <P18F458.h>
void main (void)
{
    TRISD = 0; // Program Port D as output port
    unsigned char x;
    while (1) // infinite loop
    {
        for (x = 0xFF ; x > 0 ; x--)
            PORTD = x;
    }
}
```

Ex. 10.1.2 : Explain with a neat diagram, interfacing of DAC 0808 with PIC microcontroller and write a program for ramp waveform generation using DAC interfaced with PIC microcontroller through Port D. Assume the crystal frequency to be 10 MHz. **SPPU - Dec. 14, Dec. 16, Dec. 17, 5 Marks.**



Ex. 10.1.3: Write a program to generate a positive ramp (sawtooth) waveform for PIC18 microcontroller.

OR

Write a program for sawtooth waveform generation using DAC interfaced with PIC microcontroller through Port B. Assume the crystal frequency to be 10 MHz.

SPPU - Dec. 16, 5 Marks

Soln.:

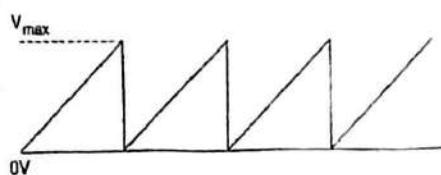
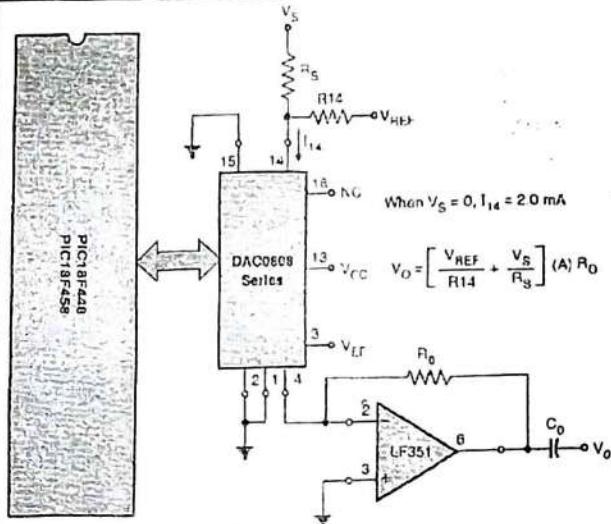
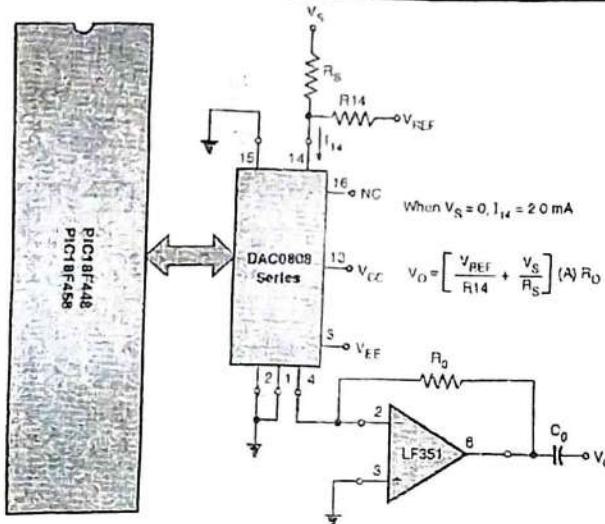


Fig. P. 10.1.3(a) : A positive ramp (sawtooth waveform)



Assembly Program

Label	Instruction	Comments
	CLRF TRISB	Program PortB as output port
	MOVLW 0x00	
again:	MOVWF PORTB	Move into PortB from WREG
	ADDLW 0x01	Increment value from WREG
	BRA again	

C18 program

```
#include <P18F458.h>
void main (void)
{
    TRISB = 0; // Program Port B as output port
    unsigned char x;
    while (1) // infinite loop
    {
        for (x = 0 ; x < FF , x++)
            PORTB = x;
    }
}
```

Ex. 10.1.4: Write a program to generate a negative ramp (sawtooth) waveform for PIC microcontroller.

OR

Write a program for sawtooth waveform generation using DAC interfaced with PIC microcontroller through port B. Assume the crystal frequency to be 10 MHz.

SPPU - May 15, Dec. 15, 8 Marks

Assembly Program

Label	Instruction	Comments
	CLRF TRISB	Program PortB as output port
	MOVLW 0xFF	
again:	MOVWF PORTB	Move into PortB from WREG
	SUBLW 0x01	Decrement value from WREG
	BRA again	

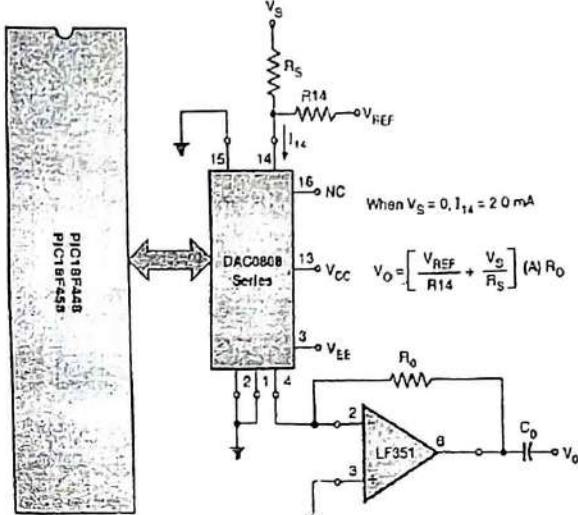
C18 program

```
#include <P18F458.h>
void main (void)
{
    TRISB = 0; // Program Port B as output port
    unsigned char x;
    while (1) // infinite loop
    {
        for (x = 0xFF ; x > 0 ; x--)
            PORTB = x;
    }
}
```



Ex. 10.1.5 : Write a program to generate a triangular waveform for PIC microcontroller.

SPPU - May 16, 9 Marks



C18 program:

```
#include <P18F458.h>
void main (void)
{
    TRISB = 0; // Program Port B as output port
    unsigned char x;
    while (1) // infinite loop
    {
        for (x = 0 ; x < 0xFF ; x++)
            PORTB = x;
        for (x = 0xFF ; x > 0 ; x--)
            PORTB = x;
    }
}
```

10.2 PIC18F458 ADC - Analog to Digital Converter

University Questions

Q. Explain features of on-board ADC.

SPPU - Dec. 14, 4 Marks

Q. Explain features of on-board ADC of PIC18F458.

SPPU - May 15, 4 Marks

Q. Explain in detail the functions of the following special function registers: ADRESH and ADRESL of PIC18 microcontroller.

SPPU - May 16, 4 Marks

Q. Explain with a neat diagram, interfacing of DAC 0808 with PIC microcontroller and write a program in C language for generation of Square waveform using DAC interfaced with PIC microcontroller through Port B. Use suitable delay. Assume the crystal frequency to be 10MHz.

SPPU - May 18, 9 Marks

The Analog-to-Digital (A/D) converter module has five inputs for the PIC18F2X8 devices and eight for the PIC18F4X8 devices. This module has the ADCON0 and ADCON1 register definitions that are compatible with the PIC micro® mid-range A/D module.

The A/D allows conversion of an analog input signal to a corresponding 10-bit digital number.

The A/D module has four registers. These registers are:

1. A/D Result High Register (ADRESH)
2. A/D Result Low Register (ADRESL)
3. A/D Control Register 0 (ADCON0)
4. A/D Control Register 1 (ADCON1)

The ADCON0 register, shown in Fig. 10.2.1, controls the operation of the A/D module.

The ADCON1 register, shown in Fig. 10.2.2, configures the functions of the port pins.

10.2.1 ADCON0 Register

University Questions

Q. Explain in detail ADCON 0.

SPPU - Dec. 14, 3 Marks

Q. Explain in detail the functions of the following special function register ADCON0 of PIC18 microcontroller. SPPU - May 16, Dec. 16, 4 Marks

Q. Explain in detail the functions of following flags related to onboard ADC of PIC microcontroller.

(iv) ADON

SPPU - Dec. 17, May 18, Dec. 18, 8 Marks

Q. Explain ADCON0 register in details.

SPPU - May 19, 4 Marks

read/write

R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 R/W-0 U-0 R/W-0

ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	-	ADON
bit7	6	5	4	3	2	1	bit0

Fig.10.2.1 : ADCON0 : A/D Control Register

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



ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	F _{Osc} /2
0	01	F _{Osc} /8
0	10	F _{Osc} /32
0	11	F _{RC} (Clock derived from the internal A/D RC oscillator)
1	00	F _{Osc} /4
1	01	F _{Osc} /16
1	10	F _{Osc} /64
1	11	F _{RC} (Clock derived from the 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/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.

Legend :

R=Readable bit W=Writable bit U=Unimplemented bit, read as '0'

-n=Value at POR '1'=bit is set '0'=bit is cleared x=bit is unknown

10.2.2 ADCON1 Register

University Questions

Q. Explain in detail ADCON 1.

SPPU - Dec. 14, 4 Marks

Q. Explain in detail the functions of the following special function register: ADCON1 of PIC18 microcontroller.

SPPU - May 16, Dec. 16, 5 Marks

Q. Explain in detail the functions of following flags related to onboard ADC of PIC microcontroller.
(iii) ADFM

SPPU - Dec. 17, May 18, Dec. 18, 8 Marks

Q. Explain ADCON1 register in details.

SPPU - May 19, 4 Marks

R/W-0 R/W-0 U-0 U-OR/W-0 R/W-0 R/W-0 R/W-0

ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

Fig. 10.2.2 : ADCON1 : A/D Control register

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 bits)

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

bit 5-4 Unimplemented : Read as '0'.

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



PCFG	AN7	AN6	AN5	AN4	AN3	AN2	AN1	ANO	V _{REF+}	V _{REF-}	C/R
0000	A	A	A	A	A	A	A	A	V _{DD}	V _{SS}	8/0
0001	A	A	A	A	V _{REF+}	A	A	A	AN3	V _{SS}	7/1
0010	D	D	D	A	A	A	A	A	V _{DD}	V _{SS}	5/0
0011	D	D	D	A	V _{REF+}	A	A	A	AN3	V _{SS}	4/1
0100	D	D	D	D	A	D	A	A	V _{DD}	V _{SS}	3/0
0101	D	D	D	D	V _{REF+}	D	A	A	AN3	V _{SS}	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	V _{REF+}	V _{REF-}	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	V _{DD}	V _{SS}	6/0
1010	D	D	A	A	V _{REF+}	A	A	A	AN3	V _{SS}	5/1
1011	D	D	A	A	V _{REF+}	V _{REF-}	A	A	AN3	AN2	4/2
1100	D	D	D	A	V _{REF+}	V _{REF-}	A	A	AN3	AN2	3/2
1101	D	D	D	D	V _{REF+}	V _{REF-}	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	V _{DD}	V _{SS}	1/0
1111	D	D	D	D	V _{REF+}	V _{REF-}	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O

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

Note : Shaded cells indicate channels available only on PIC18F4X8 devices.

Legend :

R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
- n = Value at POR '1' = bit is set	'0' = bit is cleared	x = bit is unknown

Note : On any device 'Reset', the port pins that are multiplexed with analog functions (ANx) are forced to be analog inputs.

10.2.3 A/D Conversion Time

- The analog reference voltage is software selectable to either the device's positive and negative supply voltage (V_{DD} and V_{SS}) or the voltage level on the RA3/AN3/ V_{REF+} pin and RA2/AN2/ V_{REF-} pin.
- 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.
- 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. The block diagram of the A/D module is shown in Fig. 10.2.3.
- Fig. 10.2.3 shows the operation of the A/D converter after the GO bit has been set. Clearing the GO/DONE bit during a conversion will abort the current conversion. The A/D result register pair will not be updated with the partially completed A/D conversion sample. That is, the ADRESH: ADRESL registers will continue to contain the value of the last completed conversion (or the last value written to the ADRESH: ADRESL registers). After the A/D conversion is aborted, a $2 T_{AD}$ wait is required before the next acquisition is started. After this $2 T_{AD}$ wait, acquisition on the selected channel is automatically started.

Read as 0
0 1 1 0
4 Least significant
most significant bit

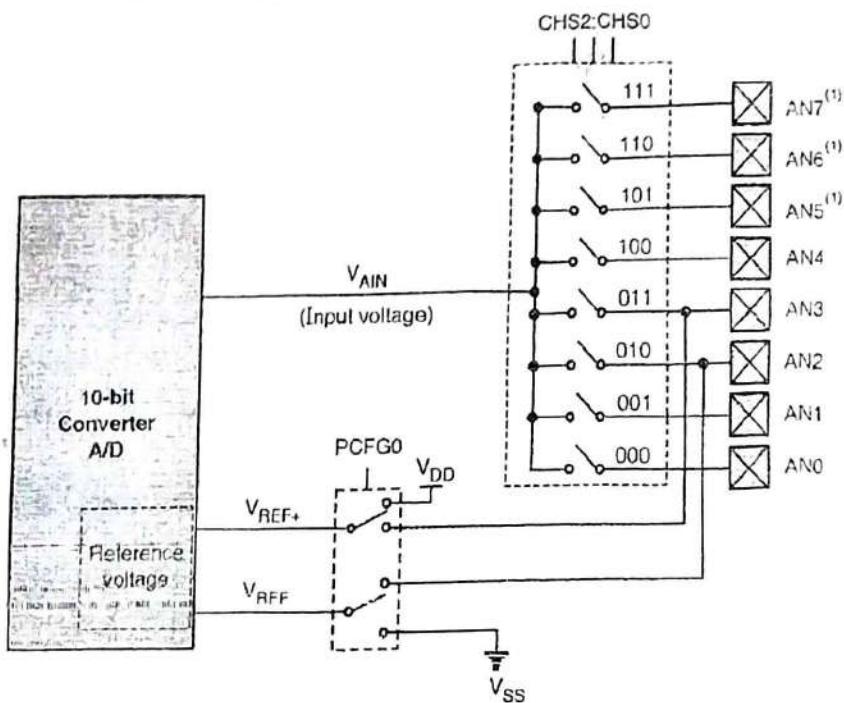


Fig. 10.2.3 : A/D Block diagram

Note: The GO/DONE bit should NOT be set in the same instruction that turns on the A/D.

- The value that is in the ADRESH/ADRESL registers is not modified for a Power-on Reset. The ADRESH/ADRESL registers will contain unknown data after a Power-on Reset.
- After the A/D module has been configured as desired, the selected channel acquired before the conversion is started. The analog input channels must have their corresponding TRIS bits selected as an input. After this acquisition time has elapsed, the A/D conversion can be started.

10.2.4 Steps for Programming ADC

University Question

- Q. Explain the steps involved in programming of A/D converter in PIC18F458 microcontroller using method of polling. **X8**

SPPU - May 16, Dec. 16, 9 Marks

- The following steps should be followed for doing an A/D conversion:

1. Configure the A/D module

- Configure analog pins, voltage reference and digital I/O (ADCON1)
- Select A/D input channel (ADCON0)
- Select A/D conversion clock (ADCON0)
- Turn on A/D module (ADCON0)

2. Configure A/D interrupt (if desired)

- Clear ADIF bit
- Set ADIE bit
- Set GIE bit

3. Wait the required acquisition time.

4. Start conversion

Set GO/DONE bit (ADCON0)

5. Wait for A/D conversion to complete, by either

Polling for the GO/DONE bit to be cleared OR
Waiting for the A/D interrupt

6. Read A/D Result registers (ADRESH/ADRESL)

clear bit ADIF if required.

For next conversion, goto step 1 or step 2 as required. The A/D conversion time per bit is defined as T_{AD} . A minimum wait of $2T_{AD}$ is required be for e next acquisition starts.

10.3 A/D Acquisition Requirements

- For the A/D converter to meet its specified accuracy, the charge holding capacitor (C_{Hold}) must be allowed to fully charge to the input channel voltage level. The analog input model is shown in Fig. 10.3.1.
- The source impedance (R_S) and the internal sampling switch (R_{SS}) impedance directly affect the time required to charge the capacitor C_{Hold} .
- The sampling switch (R_{SS}) impedance varies over the device voltage (V_{DD}). The source impedance affects the offset voltage at the analog input (due to pin leakage current).
- The maximum recommended impedance for analog Sources is $2.5\text{ k}\Omega$. After the analog input channel is selected (changed), this acquisition must be done before the conversion can be started.

Note : When the conversion is started, the holding capacitor is disconnected from the input pin.

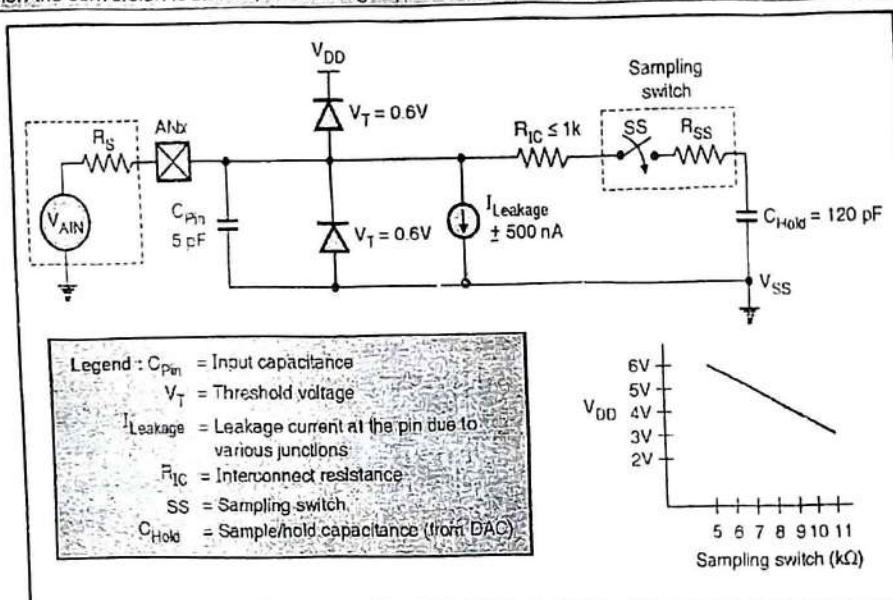


Fig.10.3.1 : Analog Input Model

Equation 1: Acquisition time

$$\begin{aligned} T_{ACQ} &= \text{Amplifier settling time} + \text{Holding capacitor} \\ &\quad \text{charging time} + \text{Temperature coefficient} \\ &= T_{AMP} + T_C + T_{COFF} \end{aligned}$$

Equation 2: A/D minimum charging time

$$V_{Hold} = (V_{REF} - (V_{REF}/2048)) \cdot (1 - e^{(-TC/C_{Hold}(R_{IC} + R_{SS} + R_S))})$$

$$\text{Or } T_C = -(120\text{ pF})(1\text{k}\Omega + R_{SS} + R_S)/n(1/2047)$$

Ex. 10.3.1: Calculating the minimum required acquisition time.

Soln.: To calculate the minimum acquisition time, Equation 1 may be used. This equation assumes that $\frac{1}{2}$ LSB error is used (1024 steps for the A/D). The $\frac{1}{2}$ LSB error is the maximum error allowed for the A/D to meet its specified resolution.

This example shows the calculation of the minimum required acquisition time T_{ACQ} . This calculation is based on the following application system assumptions :

1. $C_{Hold} = 120\text{ pF}$
2. $R_S = 2.5\text{ k}\Omega$
3. Conversion Error $\leq 1/2$ LSB
4. $V_{DD} = 5\text{V} \rightarrow R_{SS} = 7\text{k}\Omega$

5. Temperature = 50°C (system max.)

6. $V_{HOLD} = 0V$ at time = 0

$$T_{ACQ} = T_{AMP} + T_C + T_{COFF}$$

Temperature coefficient is only required for temperatures > 25°C.

$$T_{ACQ} = 2\mu s + T_C + [(Temp - 25^\circ C)(0.05\mu s/\text{ }^\circ C)]$$

$$T_C = -C_{HOLD}(R_{IC} + R_{SS} + R_S) \ln(1/2047)$$

$$= -120 \text{ pF} (1k\Omega + 7k\Omega + 2.5k\Omega) \ln(0.0004885)$$

$$= -120 \text{ pF} (10.5k\Omega) \ln(0.0004885)$$

$$= -1.26 \mu s (-7.6241) = 9.61\mu s$$

$$T_{ACQ} = 2\mu s + 9.61\mu s + [(50^\circ C - 25^\circ C)(0.05\mu s/\text{ }^\circ C)]$$

$$= 11.61\mu s + 1.25\mu s = 12.86\mu s$$

Note: When using external voltage references with the A/D converter, the source impedance of the external voltage references must be less than 20 Ω to obtain the specified A/D resolution. Higher reference source impedances will increase both offset and gain errors. Resistive voltage dividers will not provide a sufficiently low source impedance.

To maintain the best possible performance in A/D conversions, external V_{REF} inputs should be buffered with an operational amplifier or other low output impedance circuit.

- $16T_{OSC}$

- $32T_{OSC}$

- $64T_{OSC}$

- Internal RC oscillator.

- For correct A/D conversions, the A/D conversion clock (T_{AD}) must be selected to ensure minimum T_{AD} time of 1.6 μs.
- The Table 10.5.1 shows the resultant T_{AD} times derived from the device operating frequencies and the A/D clock source selected.

10.5 Configuring Analog Port Pins

- The ADCON1, TRISA and TRISE registers control the operation of the A/D port pins. The port pins that are desired as analog inputs must have their corresponding TRIS bits set(input). If the TRIS bit is cleared(output), the digital output level (V_{OH} or V_{OL}) will be converted.
- The A/D operation is independent of the state of the CHS2:CHS0 bits and the TRIS bits.

Note 1 : When reading the port register, all pins configured as analog input channels will read as cleared (a low level). Pins configured as digital inputs will convert an analog input. Analog levels on a digitally configured input will not affect the conversion accuracy.

Note 2 : Analog levels on any pin that is defined as a digital input (including the AN4:AN0 pins) may cause the input buffer to consume current that is out of the device's specification.

10.4 Selecting the A/D Conversion Clock

- The A/D conversion time per bit is defined as T_{AD} . The A/D conversion requires $12T_{AD}$ per 10-bit conversion. The source of the A/D conversion clock is software selectable. The seven possible options for T_{AD} are:

- $2T_{OSC}$
- $4T_{OSC}$
- $8T_{OSC}$

Table 10.5.1 : T_{AD} vs device operating frequencies

AD Clock Source(T_{AD})		Device Frequency			
Operation	ADCS2:ADCS0	20MHz	5MHz	1.25MHz	333.33kHz
$2T_{OSC}$	000	100ns ⁽²⁾	400ns ⁽²⁾	1.6μs	6μs
$4T_{OSC}$	100	200ns ⁽²⁾	800ns ⁽²⁾	3.2μs	12μs
$8T_{OSC}$	001	400ns ⁽²⁾	1.6μs	6.4μs	24μs ⁽³⁾
$16T_{OSC}$	101	800ns ⁽²⁾	3.2μs	12.8μs	48μs ⁽³⁾

AD Clock Source(T_{AD})		Device Frequency			
32 T_{OSC}	010	1.6 μs	6.4 μs	25.6 $\mu s^{(3)}$	96 $\mu s^{(3)}$
64 T_{OSC}	110	3.2 μs	12.8 μs	51.2 $\mu s^{(3)}$	192 $\mu s^{(3)}$
RC	011	2-6 $\mu s^{(1)}$	2-6 $\mu s^{(1)}$	2-6 $\mu s^{(1)}$	2-6 $\mu s^{(1)}$

Legend : Shaded cells are outside of recommended range.

Note : The RC source has a typical T_{AD} time of 4 μs .

These values violate the minimum required T_{AD} time.

For faster conversion times, the selection of another clock source is recommended.

Table 10.5.2 : T_{AD} vs device operating frequencies (For Extended, Lf Devices)

AD Clock Source (T_{AD})		Device Frequency			
Operation	ADC2:ADC0	4MHz	2MHz	1.25MHz	333.33kHz
2 T_{OSC}	000	500ns ⁽²⁾	1.0 $\mu s^{(2)}$	1.6 $\mu s^{(2)}$	6 μs
4 T_{OSC}	100	1.0 $\mu s^{(2)}$	2.0 $\mu s^{(2)}$	3.2 $\mu s^{(2)}$	12 μs
8 T_{OSC}	001	2.0 $\mu s^{(2)}$	4.0 μs	6.4 μs	24 $\mu s^{(3)}$
16 T_{OSC}	101	4.0 $\mu s^{(2)}$	8.0 μs	12.8 μs	48 $\mu s^{(3)}$
32 T_{OSC}	010	8.0 μs	16.0 μs	25.6 $\mu s^{(3)}$	96 $\mu s^{(3)}$
64 T_{OSC}	110	16.0 μs	32.0 μs	51.2 $\mu s^{(3)}$	192 $\mu s^{(3)}$
RC	011	3-9 $\mu s^{(1)}$	3-9 $\mu s^{(1)}$	3-9 $\mu s^{(1)}$	3-9 $\mu s^{(1)}$

Legend : Shaded cells are outside of recommended range.

Note : The RC source has a typical T_{AD} time of 6 μs .

These values violate the minimum required T_{AD} time.

For faster conversion times, the selection of another clock source is recommended.

10.6 Use of the ECCP Trigger

- An A/D conversion can be started by the "special event trigger" of the ECCP module. This requires that the ECCP1M3:ECCP1M0 bits (ECCP1CON<3:0>) be programmed as '1011' and that the A/D module is enabled (ADON bit is set). When the trigger occurs, the GO/ DONE bit will be set, starting the A/D conversion and the Timer1 (or Timer3) counter will be reset to zero.
- Timer1 (or Timer3) is reset to automatically repeat the A/D acquisition period with minimal software overhead (moving ADRESH/ADRESL to the desired location).
- The appropriate analog input channel must be selected and the minimum acquisition done before the "special event trigger" sets the GO/DONE bit (starts a conversion).



- If the A/D module is not enabled (ADON is cleared), the "special event trigger" will be ignored by the A/D module but will still reset the Timer1 (or Timer3) counter.

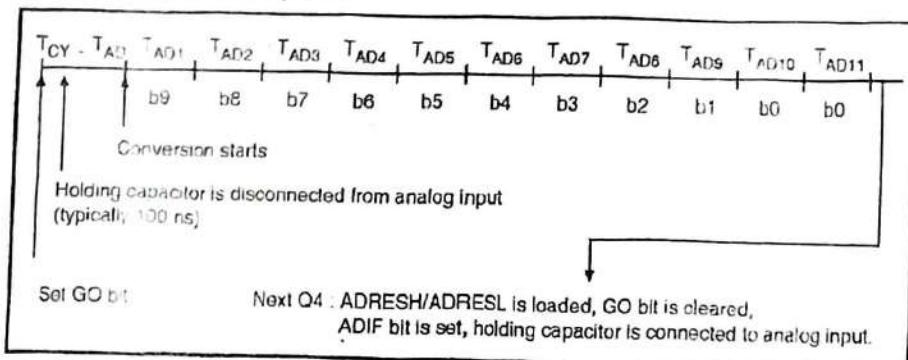
Fig. 10.6.1 : A/D conversion T_{AD} Cycles

Table 10.6.1 : Summary of A/D registers

Name	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Value on POR,BOR	Value on all other resets
INTCON	GIE/GIEH	PEIE/GIEL	TMROIE	INT0IE	RBIE	TMROIF	INT0IF	RBIF	0000000x	0000000u
PIR1	PSP1F ⁽¹⁾	ADIF	RC1F	TX1F	SSP1F	CCP1IF	TMR2IF	TMR1IF	00000000	00000000
PIE1	PSP1E ⁽¹⁾	ADIE	RC1E	TX1E	SSP1E	CCP1IE	TMR2IE	TMR1IE	00000000	00000000
IPR1	PSP1P ⁽¹⁾	ADIP	RC1P	TX1P	SSP1P	CCP1IP	TMR2IP	TMR1IP	11111111	11111111
PIR2	—	CM1F ⁽¹⁾	—	EE1F	BCL1F	LVD1F	TMR3IF	ECCP1IF ⁽¹⁾	-0-00000	-0-00000
PIE2	—	CM1E ⁽¹⁾	—	EE1E	BCL1E	LVD1E	TMR3IE	ECCP1IE ⁽¹⁾	-0-00000	-0-00000
IPR2	—	CM1P ⁽¹⁾	—	EE1P	BCL1P	LVD1P	TMR3IP	ECCP1IP ⁽¹⁾	-1-11111	-1-11111
ADRESH	A/D Result Register								XXXXXXX	uuuuuuuu
ADRESL	A/D Result Register								XXXXXXX	uuuuuuuu
ADCON0	ADC01	ADC00	CHS2	CHS1	CHS0	GO/DONE	—	ADON	000000-0	000000-0
ADCON1	ADFM	ADC02	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00--0000	00--0000
PORTA	—	RA6	RA5	RA4	RA3	RA2	RA1	RA0	-x0x0000	-u0u0000
TRISA	—	PORTA Data Direction Register							-1111111	-1111111
PORTE	—	—	—	—	—	RE2	RE1	RE0	----xxx	----000
LATE	—	—	—	—	—	LATE2	LATE1	LATE0	----xxx	----uuu
TRISE	IBF	OBF	IBOV	PSPMODE	—	TRISE2	TRISE1	TRISE0	0000-111	0000-111

10.6.1 A/D Result Registers

- 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.

- Fig. 10.6.2 shows 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.

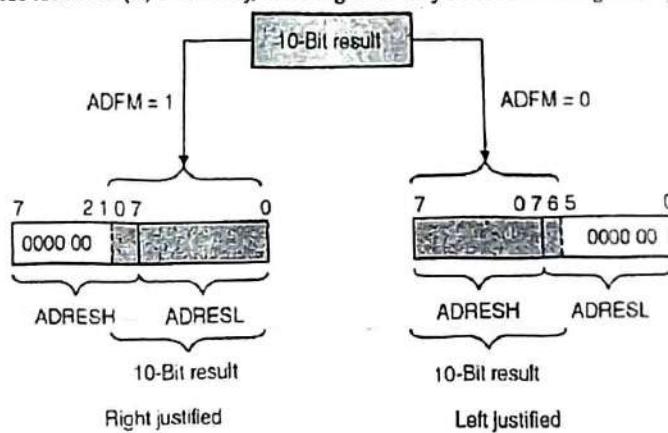


Fig.10.6.2 : A/D Result Justification

Ex. 10.6.1: 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.

SPPU - May 15, 8 Marks

Soln.:

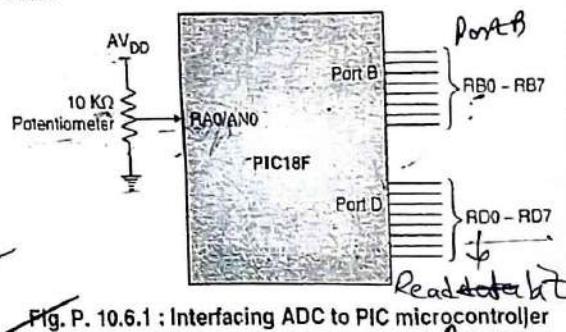


Fig. P. 10.6.1 : Interfacing ADC to PIC microcontroller
C Program

```
#include<P18F458.h>
void MSDelay (unsigned int x)
{
    unsigned int i;
    unsigned char j;
    for (i=0; i<x; i++)
        for (j=0; j<165; j++);
}
void main (void)
{
    TRISAbit.TRISA = 1; //Set RA0 pin as input
    //pin
    TRISD = 0; //Set Port D output port
    TRISB = 0; //Set port B output port
    TRISAbits.TRISA = 1; //RA0 = 1
```

```
ADCON1 = 0xCE;
// fosc / 64, AN0 input, right
// justified
```

```
ADCON0 = 0x81; // fosc / 64, channel 0, ADC on
```

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
    MSDelay (250); // Quarter second delay
}
```

Ex. 10.6.2: Write a program to read a data from ADC and store results from memory location 0x50H onwards.

SPPU - Dec. 15, 8 Marks

Soln.:

C Program

```
#include<P18F458.h>
static unsigned char *ucp = 0x50;
static unsigned char *lcp = 0x51;
void MSDelay (unsigned int x)
{
    unsigned int i;
```



```

unsigned char j;
for (i=0; i<x; i++)
    for (j=0; j<165; j++);

}

void main (void)
{
    TRISAbits.TRISA = 1; //Set RA0 pin as input
    //pin
    TRISAbits.TRISA = 1; //RA0 = 1

    ADCON1 = 0xCE; //fosc
    //64, A0 input, right
    //justified
    ADCON0 = 0x81; //fosc
    //64, channel 0, ADC on
    while (1)
    {
        ADCON0bits.GO = 1; // start conversion
        while (ADCON0bits.DONE == 1);
        //check end of conversion
        *lcp = ADRESL; //send low byte result
        //to 0x50
        *ucp = ADRESH; //Send high byte result
        //to 0x51
        MSDelay (250); // delay
    }
}

```

10.6.2 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. 10.6.3 : 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 hi_int (void)
{
    ISR(); //end high-priority interrupt
}

void ISR (void)
{
    if (PIR1bits.ADIF == 1) // Did ADC cause
    //interrupt ?
    {
        PORTB = ADRESL; //send low byte result
        //to Port B
        PORTD = ADRESH; //Send high byte
        //result to Port D
        MSdelay (250); // quarter second delay
        PIR1bits.ADIF = 0; // ADIF = 0
    }
}

void MSdelay (unsigned int x)
{
    unsigned int i,j;
    for (i=0; i<x; i++)
        for (j=0; j<165; j++);
}

void main (void)
{
    TRISAbits.TRISA = 1;
    //Set RA0 pin as input pin
    TRISD = 0; // Set Port D output port
    TRISB = 0; //Set Port B output port
    ADCON1 = 0xCE;
    ADCON0 = 0x81;
    PIR1bits.ADIF = 0; //ADIF = 0
    PIE1bits.ADIE = 1; //Activate ADIE interrupt
    INTCONbits.PEIE = 1; //Activate peripheral
    //interrupts
    INTCONbits.GIE = 1;
    //Activate global interrupts
    while (1)
}

```

```

    (
        MSDelay (250) ;
        ADCON0bits.GO = 1; //Start conversion
    }
}

```

Ex. 10.6.4 : 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.

SPPU - Dec. 15, 8 Marks

OR Draw the interfacing diagram of voltage measurement and also explain its interfacing procedure.

SPPU - May 19, 9 Marks

OR Interface analog voltage 0-5V to internal ADC and display value on LCD Ex. 10.6.3

SPPU - Dec. 14, May 15, 8 Marks

Soln.: Fig. P. 10.6.4 shows the interfacing diagram. The microcontroller ADC module converts the analog signal to 10 bit binary data. The analog input to PIC is given on pin RA0 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,

$$V_{out} = \text{Number of steps} \times \text{Step size.}$$

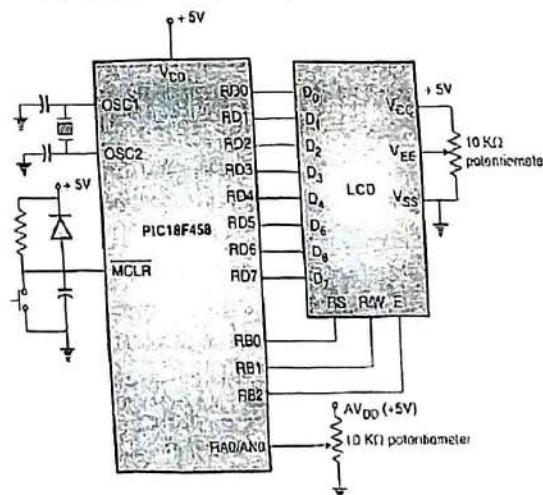


Fig. P. 10.6.4 : Interfacing PIC18F458 with analog voltage

$V_{out} = 1024 \times 5 \text{ mV} = 5.12 \text{ V}$ 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 RS PORTBbits.RB0 // RS = PORTB.0
#define RW PORTBbits.RB1 // RW = PORTB.1
#define E PORTBbits.RB2 // E = PORTB.2
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};
char abc[9] = {0xD0, 0x00, 0x80, 0x18, 0x08, 0x07, 0x18, 0x07, 0x20};

void MSDelay (unsigned int x)
{
    unsigned int i, j;
    for (i = 0; i < x; i++)
        for (j = 0; j < 165; j++);
}

void CMD (unsigned char cmd) //command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}

void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0;
}

#pragma code hi_int = 0x0008 //high priority
//interrupt
void hi_int (void)
{
    ISR();
}

```



```

#pragma code           //end high-priority interrupt
void ISR8 (void)
{
    unsigned long voltage;
    char digit;
    if (PIR1bits.ADIF == 1) // Did ADC cause
        //interrupt?
    // Did ADC cause interrupt?

    {
        voltage = ADRESH * 256 + ADRESL;
        voltage = voltage * 5000 / 1023
        // convert it to mV

        //Following logic for separating digits and display on.
        //LCD
        digit = (voltage / 1000) + 48;
        DAT (digit);
        DAT (".");
        digit = (voltage % 1000 / 10) + 48;
        DAT (digit);
        digit = ((voltage % 1000) % 100 / 10) + 48;
        DAT (digit);
        digit = (((voltage % 100) % 100) % 10) + 48;
        DAT (digit);
        DAT ("V");
        MSDelay (250); // quarter second delay
        PIR1bits.ADIF = 0; // ADIF = 0
    }
}

void main ()
{
    TRISD = 0;          // port D = output port
    TRISB = 0;          // port B = output port
    unsigned int i;
    for (i = 0; i <= 4; i++)
    {
        CMD (cmds[i]);
        MSDelay (250);
    }
    ADCON1 = 0xCE;
    ADCON0 = 0x81;
    PIR1bits.ADIF = 0; //ADIF = 0
    PIE1bits.ADIE = 1; //Activate ADIE interrupt
    INTCONbits.PEIE = 1; //Activate peripheral
                        //interrupts
}

```

```

INTCONbits.GIE = 1; //Activate global
                    //interrupts
while (1)
{
    MSDelay (250);
    ADCON0bits.GO = 1; //Start conversion
}

```

~~10.7 Temperature Sensor LM35~~

University Questions

- Q. With the help on Interfacing diagram and flowchart explain how PIC18 microcontroller can be used to measure temperature using LM35 sensor. **SPPU - May 18, 8 Marks**
- Q. With a neat interfacing diagram and explain temperature measurement using PIC 18 microcontroller. **SPPU - Dec.17, 9 Marks**
- Q. Explain interfacing of LM35 for Temperature measurement. **SPPU - May 19, 8 Marks**

- 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 industrial 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 $\pm 1/4^\circ\text{C}$ at room temperature and $\pm 3/4^\circ\text{C}$ over a full -55 to $+150^\circ\text{C}$ temperature range.

- The LM35 is rated to operate over a -55°C to $+150^\circ\text{C}$ temperature range.

- As shown in the Fig. 10.7.1 the connection for LM35 is very simple. Also the output scale is linear with a change of $10\text{mV}/^\circ\text{C}$.

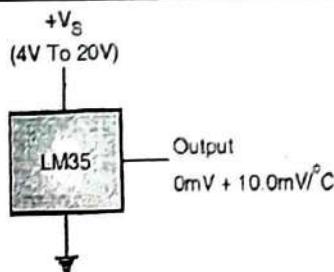


Fig. 10.7.1 : Circuit diagram for the LM35 basic temperature sensor (+2° C to +150° C)

Ex. 10.7.1: Draw an interfacing of temperature Sensor to PIC using Serial ADC and Indicate excess temperature when exceeds the set point by LED.

OR Design a PIC18F458 based system to interface LM35. Write the corresponding C and assembly program For reading and displaying temperature.

OR Explain the use of PIC-ADC module to interface the Temp sensor LM35 used for accepting the Temp and display on LED connected to port D, write an embedded C program.

SPPU - Dec. 16, 9 Marks

Soln.:

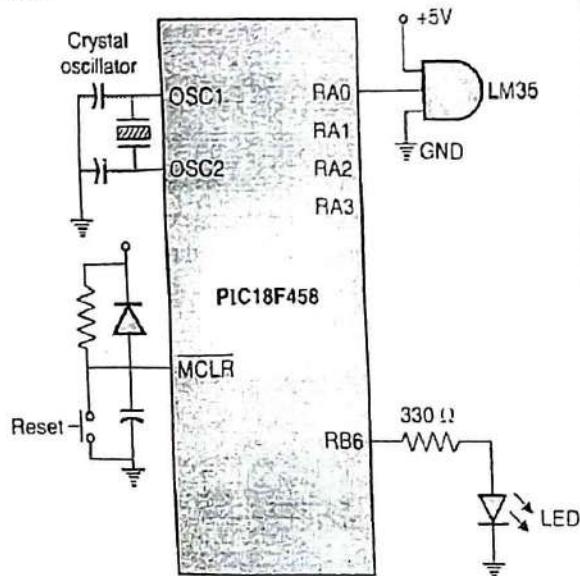


Fig. P.10.7.1

Program

```
# include <P18F458.h>
#define LED PORTBbits.RB6
void MSDDelay (unsigned int x)
{
    unsigned int i;
```

```
unsigned char j;
for (i=0; i<x; i++)
for (j=0; j<165; j++) ;

}

void main (void)
{
    unsigned char temp, set_value=35;
    TRISA bits.TRISA = 1; //Set RA0 pin as input pin
    TRISD = 0; //Set Port D output port
    TRISB = 0; //Set port B output port
    TRISA bits.TRISA = 1; //RA0 = 1

    ADCON1 = 0xCE;
    fosc //AN0 input, right justified
        / 64 , ADCON0 = 0x81; //fosc / 64 , channel 0, ADC on
    while (1)
    {
        ADCON0bits.GO = 1; // start conversion
        while (ADCON0bits.DONE == 1); //check end of conversion
        temp = ADRESH * 256 + ADRESL; //Read Temperature
        if(temp>set_value) LED=1; //LED ON if temperature above set value
        else LED=0;//LED OFF if
        //temperature not above set value
        MSDDelay (250); // Quarter second delay
    }
}
```

Ex. 10.7.1(A): Draw interfacing of LM35 with PIC 18F458. Write a program to measure the temperature and display the 10 bit digital equivalent value of the temperature on Port C and Port D.

SPPU - Dec. 14, May 16, 5 Marks

Soln.:

```
# include <PIC18F458.h>
# pragma code hi_int = 0x0008
//high priority interrupt
void hi_int (void)
{
```

```

ISRs () ;
}

#pragma code      //end high-priority interrupt
void ISRs (void)
{
    if (PIR1bits.ADIF == 1) // Did ADC cause
        //interrupt ?
    {
        PORTC = ADRESL; //send low byte result
                           //to Port C
        PORTD = ADRESH; //Send high byte
                           // result to Port D
        MSDelay (250);   // quarter second delay
        PIR1bits.ADIF = 0; // ADIF = 0
    }
}

void MSDelay (unsigned int x)
{
    unsigned int i,j;
    for (i=0; i<x; i++)
        for (j=0; j<165; j++);
}
}

void main (void)
{
    TRISAbits.TRISA = 1; //Set RA0 pin as input
    //pin
    TRISD = 0; // Set Port D output port
    TRISC = 0; // Set Port C output port
    ADCON1 = 0xCE;
    ADCON0 = 0x81;
    PIR1bits.ADIF = 0; //ADIF = 0
    PIE1bits.ADIE = 1; //Activate ADIE interrupt
    INTCONbits.PEIE = 1; //Activate peripheral
                           //interrupts
    INTCONbits.GIE = 1; //Activate global
    //interrupts
    while (1)
    {
        MSDelay (250);
        ADCON0bits.GO = 1; //Start conversion
    }
}
}

```

Ex. 10.7.2: Design a Home alarm system considering the parameters of door safety using sensors for detection of person and its movements. Display warning on LCD and LED, light the Lamp connected with Opto-oscillator. Draw the Flowchart with Initialization program.

SPPU: May 15, 9 Marks

Soln.: We will use a temperature sensor. In case if a person comes in the home, the temperature changes drastically. Hence by comparing the current and previous temperatures, if the value is more than the allowed variation, then we can say that there is some burglar in the home. The set value needs to be tested using trial and error method.

The following is the interface and program for the same

The LM35 operates over a temperature range of -55°C to $+150^{\circ}\text{C}$ with an output of 10 mV for each degree centigrade. e.g. for 80°C temperature the output will be $80 \times 10 = 800$ mV.

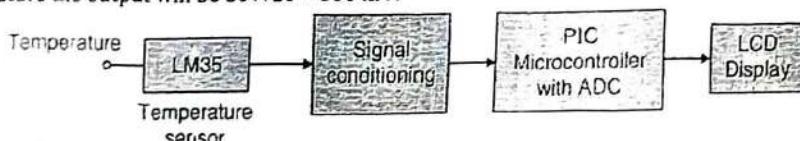


Fig. P. 10.7.2: Block diagram of data acquisition system used for temperature measurement

The A/D converter of PIC18F458 is of 10 bit resolution and maximum number of steps = $2^{10} = 1024$ Steps. For each $^{\circ}\text{C}$ LM35 gives 10 mV output.

For a step size of 10 mV,

$$\begin{aligned} V_{\text{out}} &= 10 \text{ mV} \times 1024 \text{ steps} = 10240 \text{ mV} \\ &= 10.24 \text{ V} \end{aligned}$$

However, this much voltage is not acceptable.

If a step size of 2.5 mV is selected,

$$V_{out} = 2.5 \text{ mV} \times 1024$$

$$V_{out} = 2560 \text{ mV} = 2.56 \text{ V}$$

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 $V_{ref} = 2.56 \text{ V}$.

The real temperature can be got by dividing the A/D output by 4.

Fig. P. 10.7.2(a) shows the interfacing diagram of PIC18F458 with LM35 temperature sensor.

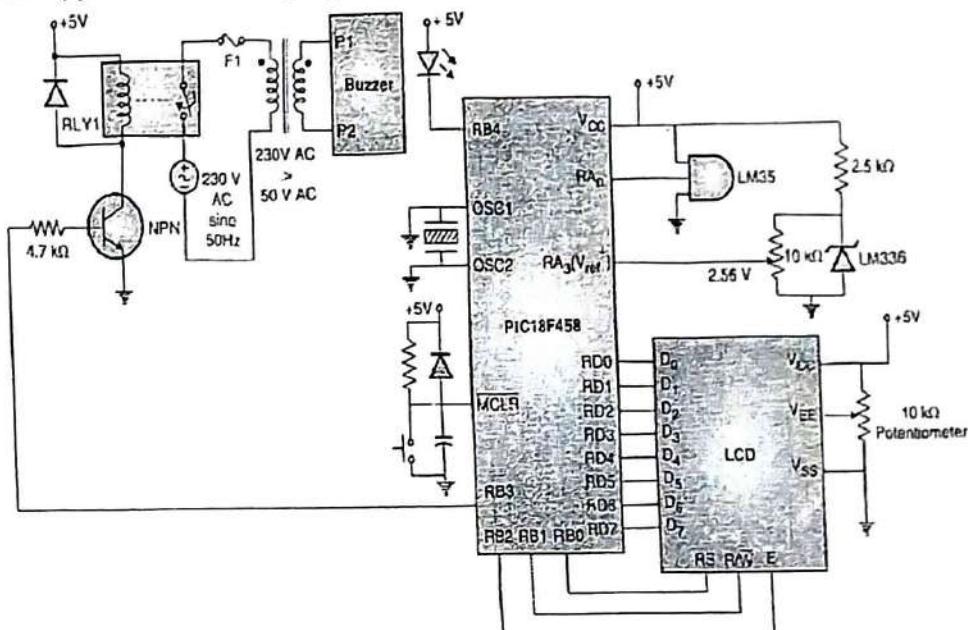


Fig. P. 10.7.2(a) : Interfacing diagram of PIC18F458 based data acquisition system for measurement of temperature

- Table P. 10.7.2 gives the 10 bit output values for ADC and the temperature with $V_{ref} = 2.56 \text{ V}$.

Table P. 10.7.2

Temperature °F	V_{in} (mV)	Number of V_{in} steps 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

Temperature °F	V_{in} (mV)	Number of V_{in} steps 2.5	10 bit A/D output	Real temperature in binary A/D output 4
120	1200	480	0111100000	0001111000
130	1300	520	1000001000	0010000010
140	1400	560	1000110000	0010001100
150	1500	600	1001011000	0010010110

Program:

```
#include <P18F458.h>
Unsigned int i = 0;
char cmd[5] = {0x38, 0x0E, 0x01, 0x06, 0xC0};
#define BUZZER PORTBbits.RB3
#define LED PORTBbits.RB4
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define EN PORTBbits.RB2
#define temp PORTAbits.RA0
#define Vref PORTAbits.RA3
#define busy PORTDbits.RD7
void MSDelay (unsigned int x)
{
```



```

unsigned int i, j;
for (i = 0; i < x; i++)
    for (j = 0; j < 105; j++);
}

void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    E = 1;
    MSDDelay (250);
    E = 0;
}

void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    E = 1;
    MSDDelay (250);
    E = 0;
}

void main ()
{
    unsigned char ADC_output, Lo_byte;
    Hi_byte, sel_value = 3, prev;
    for (i = 0; i <= 4; i++)
    {
        CMD (cmds[i]);
        MSDDelay (250);
    }

    TRISD = 0; //make port D an output port
    TRISAbits.TRISA0 = 1 //Make RA0 an input
    //pin
    TRISAbits.TRISA3 = 1; //RA3 = 1 for Vref
    //input
    fosc
    ADCON0 = 0x81; //Channel 0, ADC on, 64
    fosc
    ADCON1 = 0xC5; //64, right justified,
    //AN0 = analog input, AN3 = Vref
    while (1)
    {
        for (i = 0; i < 25; i++);
        //wait for sometime using software delay
    }
}

```

```

ADCON0 bits.GO = 1 //start conversion
while (ADCON0 bits.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 &= 0x0C0; // mask the lower 6 bits
ADC_output = Lo_byte High byte;
PORTD = ADC_output;
DAT (ADC_output/100); //Display the
//temperature
DAT ((ADC_output%100)/10);
DAT (ADC_output % 10);
DAT ("°");
DAT ('C'); //unit of temperature is in °C
CMD (0x80);
if (prev > ADC_output)
{
    if (prev - ADC_output > set_value)
    {
        BUZZER = 1;
        LED = 1;
    }
}
else
{
    if (ADC_output - prev > set_value)
    {
        BUZZER = 1;
        LED = 1;
    }
}
prev = ADC_output;
}
}

```

Ex. 10.7.3: Draw an interfacing diagram to PIC - ADC for measuring temperature of room and display on LCD, indicate over temperature by LED, buzzer and relay connected to port. Write an C program for testing.

SPPU-Dec. 16, 9 Marks

Soln.:

The LM35 operates over a temperature range of -55°C to $+150^{\circ}\text{C}$ with an output of 10 mV for each degree centigrade. E.g. for 80°C temperature the output will be $80 \times 10 = 800 \text{ mV}$.

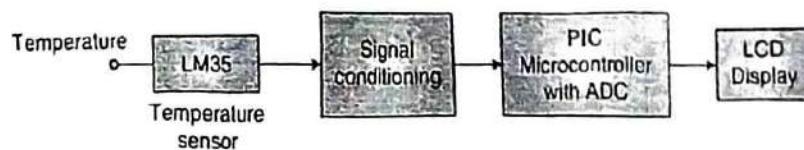


Fig. P. 10.7.3: Block diagram of data acquisition system used for temperature measurement

- The A/D converter of PIC18F458 is of 10 bit resolution and maximum number of steps = $2^{10} = 1024$ Steps. For each $^{\circ}\text{C}$ LM35 gives 10 mV output.

For a step size of 10 mV,

$$\begin{aligned} V_{\text{out}} &= 10 \text{ mV} \times 1024 \text{ steps} = 10240 \text{ mV} \\ &= 10.24 \text{ V} \end{aligned}$$

- However, this much voltage is not acceptable.
- If a step size of 2.5 mV is selected,

$$V_{\text{out}} = 2.5 \text{ mV} \times 1024$$

$$V_{\text{out}} = 2560 \text{ mV} = 2.56 \text{ V}$$

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 $V_{\text{ref}} = 2.56 \text{ V}$.

The real temperature can be got by dividing the A/D output by 4.

Fig. P. 10.7.3(a) shows the interfacing diagram of PIC18F458 with LM35 temperature sensor.

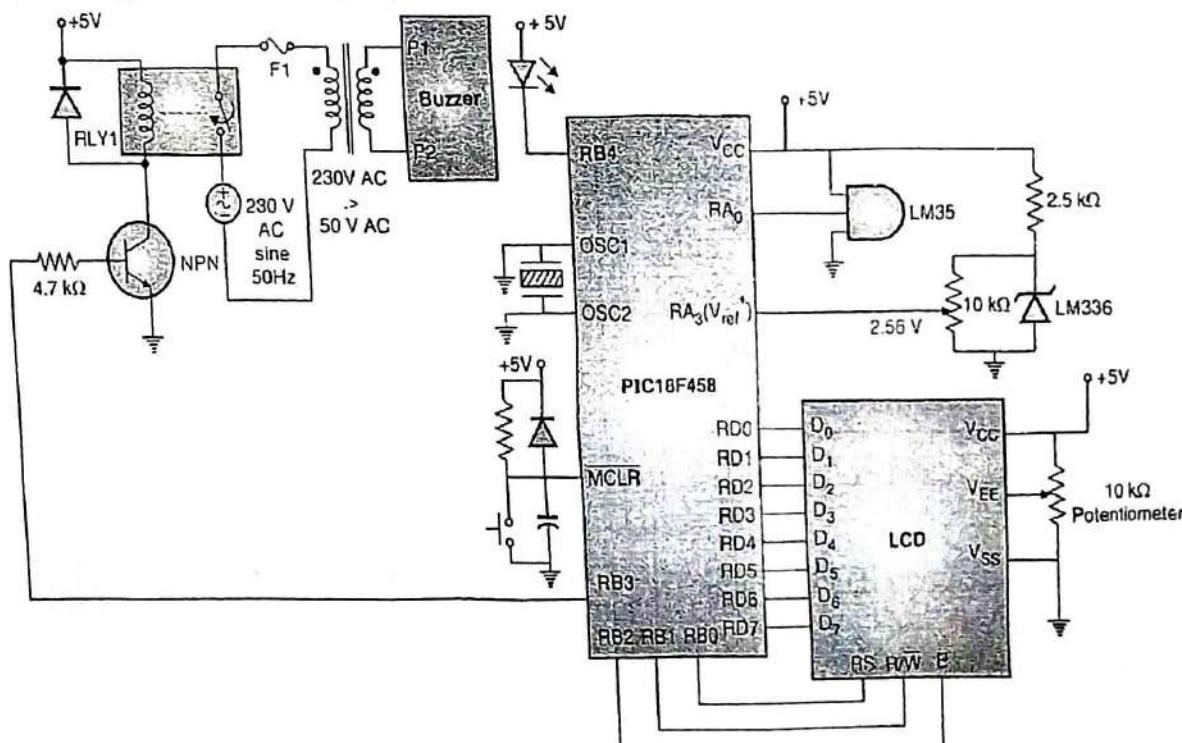


Fig. P. 10.7.3(a) : Interfacing diagram of PIC18F458 based data acquisition system for measurement of temperature



- Table P. 10.7.3 gives the 10 bit output values for ADC and the temperature with $V_{ref} = 2.56$ V.

Table P. 10.7.3

Temperature °F	V_r (mV)	Number of $\frac{V_{in}}{2.5}$ steps	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

Program:

```
#include <P18F458.h>
Unsigned int i = 0;
char cmd[5] = {0x38,0x0E,0x01,0x06,0xC0};
#define BUZZER PORTBbits.RB3
#define LED PORTBbits.RB4
#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
```

```
#define EN PORTBbits.RB2
#define temp PORTAbits.RA0
#define Vref PORTAbits.RA3
#define busy PORTDbits.RD7
void MSDelay (unsigned int x)
{
    unsigned int i, j ;
    for (i = 0 ; i < x ; i++)
        for (j = 0 ; j < 165 ; j++)
}
void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    E = 1;
    MSDelay (250);
    E = 0;
}
void DAT (unsigned char dat) //display function
{
    PORTD = dat;
    RS = 1;
    RW = 0;
    E = 1;
    MSDelay (250);
    E = 0;
}
void main ()
{
    unsigned char ADC_output, Lo_byte,
    Hi_byte, set_value=50;
    for(i=0;i<=4;i++)
    {
        CMD (cmd[i]);
        MSDelay (250);
    }
    TRISD = 0; //make port D an output port
    TRISAbits.TRISA0 = 1;//Make RA0 an input
    //pin
    TRISAbits.TRISA3 = 1; // RA3 = 1 for  $V_{ref}$ 
    //input
}
```

```

ADCON1 = 0xC5; //  $\frac{f_{osc}}{64}$ , 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;
    DAT(ADC_output/100); //Display the
    //temperature
    DAT((ADG_output%100)/10);
    DAT(ADC_output % 10);
    DAT("°");
    DAT('C'); //unit of temperature is in °C
    CMD(0x80);
    if(ADC_output > set_value)
    {
        BUZZER=1;
        LED=1;
    }
}

```

10.8 Features of DS1306

University Question

Q. State features of RTC.

SPPU - May 16, Dec. 17, May 18, Dec. 18, 10 Marks

Completely manages all time keeping functions

- Real-Time Clock (RTC) counts seconds, minutes, hours, date of the month, month, day of the week, and year with leap-year compensation valid up to 2100.
- 96-byte, battery-backed NV RAM for data storage.
- Two time-of-day alarms, programmable on combination of seconds, minutes, hours, and day of the week.
- 1 Hz and 32.768 kHz clock outputs.

Standard serial port interfaces with most microcontrollers

- Supports Motorola SPI (Serial Peripheral Interface) modes 1 and 3 or standard 3-wire interface.
- Burst mode for reading/writing successive addresses in Clock/RAM.

Multiple power supply pins ease adding battery for backup

- Dual-power supply pins for primary and backup power supplies.
- Optional ^{topre} trickle charge output to backup supply.

Optional industrial temperature range : -40°C to +85°C supports operation in a wide range of applications

- 20-pin TSSOP minimizes required space
- Underwriters Laboratory (UL®) recognized

10.9 Pin Functions of DS1306

Table 10.9.1 : Pin description

Pin		Name	Function
TSSOP	DIP		
1	1	V _{CC2}	Backup power supply : This is the secondary power supply pin. In systems using the trickle charger, the rechargeable energy source is connected to this pin.
2	2	V _{BAT}	Battery input for any standard +3V lithium cell or other energy source : If not used, V _{BAT} must be connected to ground. Diodes must not be placed in series between V _{BAT} and the battery charging, or improper operation will result. UL recognized to ensure against reverse charging current when used in conjunction with a lithium battery.



Pin		Name	Function
TSSOP	DIP		
3	3	X1	Connections for standard 32.768 kHz quartz crystal : The internal oscillator is designed for operation with a crystal having a specified load capacitance of 6pF. For more information on crystal selection and crystal layout considerations, refer to Application Note 58, "Crystal Considerations with Dallas Real-Time Clocks." The DS1306 can also be driven by an external 32.768kHz oscillator. In this configuration, the X1 pin is connected to the external oscillator signal and the X2 pin is floated.
5	4	X2	
7	5	INT0	Active-low interrupt 0 output : The INT0 pin is an active-low output of the DS1306 that can be used as an interrupt input to a processor. The INT0 pin can be programmed to be asserted by Alarm 0. The INT0 pin remains low as long as the status bit causing the interrupt is present and the corresponding interrupt enable bit is set. The INT0 pin operates when the DS1306 is powered by V _{CC1} , V _{CC2} or V _{BAT} . The INT0 pin is an open-drain output and requires an external pull up resistor.
8	6	INT1	Interrupt 1 output : The INT1 pin is an active-high output of the DS1306 that can be used as an interrupt input to a processor. The INT1 pin can be programmed to be asserted by Alarm 1. When an alarm condition is present, the INT1 pin generates a 62.5mS active-high pulse. The INT1 pin operates only when the DS1306 is powered by V _{CC2} or V _{BAT} . When active, the INT1 pin is internally pulled up to V _{CC2} or V _{BAT} . When inactive, the INT1 pin is internally pulled low.
9	7	1Hz	1Hz output : The 1Hz pin provides a 1Hz square wave output. This output is active when the 1Hz bit in the control register is a logic 1. Both INT0 and 1Hz pins are open-drain outputs. The interrupt, 1Hz signal, and the internal clock continue to run regardless of the level of V _{CC} (as long as a power source is present).
10	8	GND	Ground
11	9	SERMODE	Serial interface mode : The SERMODE pin offers the flexibility to choose between two serial interface modes. When connected to GND, standard 3-wire communication is selected. When connected to V _{CC} , SPI communication is selected.
12	10	CE	Chip enable : The chip enable signal must be asserted high during a read or a write for both 3-wire and SPI communication. This pin has an internal 55kΩ pull down resistor (typical).
14	11	SCLK	Serial clock : SCLK is used to synchronize data movement on the serial interface for either the SPI or 3-wire interface.
15	12	SDI	Serial data in : When SPI communication is selected, the SDI pin is the serial data input for the SPI bus. When 3-wire communication is selected, this pin must be tied to the SDO pin (the SDI and SDO pins function as a single I/O pin when tied together).
16	13	SDO	Serial data out : When SPI communication is selected, the SDO pin is the serial data output for the SPI bus. When 3-wire communication is selected, this pin must be tied to the SDI pin (the SDI and SDO pins function as a single I/O pin when tied together). V _{CCIF} provides the logic-high level.
17	14	V _{CCIF}	Interface logic power-supply input : The V _{CCIF} pin allows the DS1306 to drive SDO and 32kHz output pins to a level that is compatible with the interface logic, thus allowing an easy interface to 3V logic in mixed supply systems. This pin is physically connected to the source connection of the p-channel transistors in the output buffers of the SDO and 32kHz pins.
18	15	32kHz	32.768kHz output : The 32kHz pin provides a 32.768kHz output. This signal is always present. V _{CCIF} provides the logic-high level.
20	16	V _{CC1}	Primary power supply : DC power is provided to the device on this pin. V _{CC1} is the primary power supply.
4, 6, 13, 19	-	N.C	No connection

10.10 Control Registers and Operation of DS1306

- The DS1306 serial alarm Real-Time Clock (RTC) provides a full Binary Coded Decimal (BCD) clock calendar that is accessed by a simple serial interface.
- The clock/calendar provides seconds, minutes, hours, day, date, month, and year information. The end of the month date is automatically adjusted for months with 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.
- In addition, 96 bytes of NV RAM are provided for data storage. An interface logic power-supply input pin (V_{CCIF}) allows the DS1306 to drive SDO and 32 kHz pins to a level that is compatible with the interface logic. This allows an easy interface to 3V logic in mixed supply systems.
- The DS1306 offers dual-power supplies as well as a battery-input pin. The dual-power supplies support a programmable trickle charge circuit that allows a rechargeable energy source (such as a super cap or rechargeable battery) to be used for a backup supply. The V_{BAT} pin allows the device to be backed up by a non-rechargeable battery.
- The DS1306 is fully operational from 2.0V to 5.5V. Two programmable time-of-day alarms are provided by the DS1306. Each alarm can generate an interrupt on a programmable combination of seconds, minutes, hours, and day. "Don't care" states can be inserted into one or more fields if it is desired for them to be ignored for the alarm condition.
- A 1Hz and a 32kHz clock output are also available. The DS1306 supports a direct interface to SPI serial data ports or standard 3-wire interface. An easy-to-use address and data format is implemented in which data transfers can occur 1 byte at a time or in multiple-byte burst mode.

Table 10.10.1 : RTC registers and address map

Hex address		Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Range		
Read	Write											
00h	80h	0	10 Sec			Sec			00-59			
01h	81h	0	10 Min			Min			00-59			
02h	82h	0	12	P	10-HR	Hours			01 - 12 + P/A			
			A									
			24	10					00-23			
			0	0	0	0	Day		01-07			
04h	84h	0	0	10-Date		Date			01-31			
05h	85h	0	0	10-Month		Month			01-12			
06h	86h	10-Year			Year			00-99				
07h	87h	M	10-Sec alarm 0			Sec alarm 0			00-59			
08h	88h	M	10-Min alarm 0			Min alarm 0			00-59			
09h	89h	M	12	P	10-HR	Hour alarm 0			01 - 12 + P/A			
			A									
			24	10					00-23			
0Ah	8Ah	M	0	0	0	0	Day alarm 0		01-07			
0Bh	8Bh	M	10 Sec alarm 1			Sec alarm 1			00-59			



Hex address		Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0	Range	
Read	Write										
0Ch	8Ch	M	10 Min alarm 1				Min alarm 1				00-59
0Dh	8Dh	M	12	P	10-HR	Hour alarm 1				01 - 12 + P/A	
			A	24						00-23	
			10								
0Eh	8Fh	M	0	0	0	0	Day alarm 1			01-07	
-											
0Fh	8Fh	Control register							-		
10h	90h	Status register							-		
11h	91h	Trickle charger register							-		
12h-1Fh	92h-9Fh	Reserved							-		
20h-7Fh	A0h-FFh	96-Bytes user RAM							-		

- The DS1306 can be run in either 12-hour or 24-hour mode. Bit 6 of the hours register is defined as the 12- or 24-hour mode select bit. When high, the 12-hour mode is selected. In the 12-hour mode, bit 5 is the AM/PM bit with logic-high being PM. In the 24-hour mode, bit 5 is the second 10-hour bit (20 to 23 hours).
- The DS1306 contains two time-of-day alarms. Time-of-day alarm 0 can be set by writing to registers 87h to 8Ah. Time-of-day Alarm 1 can be set by writing to registers 8Bh to 8Eh. bit 7 of each of the time-of-day alarm registers are mask bits (Table 10.10.2).
- When all of the mask bits are logic 0, a time-of-day alarm only occurs once per week when the values stored in timekeeping registers 00h to 03h match the values stored in the time-of-day alarm registers. An alarm is generated every day when bit 7 of the day alarm register is set to a logic 1. An alarm is generated every hour when bit 7 of the day and hour alarm registers is set to a logic 1. Similarly, an alarm is generated every minute when bit 7 of the day, hour, and minute alarm registers is set to a logic 1. When bit 7 of the day, hour, minute, and seconds alarm registers is set to a logic 1, an alarm occurs every second.
- During each clock update, the RTC compares the Alarm 0 and Alarm 1 registers with the corresponding clock registers. When a match occurs, the corresponding alarm flag bit in the status register is set to a 1.

- If the corresponding alarm interrupt enable bit is enabled, an interrupt output is activated.

Table 10.10.2 : Time-of-day alarm mask bits

Alarm register mask bits (bit 7)				Function
Seconds	Minutes	Hours	Days	
1	1	1	1	Alarm once per second
0	1	1	1	Alarm when seconds match
0	0	1	1	Alarm when minutes and seconds match
0	0	0	1	Alarm hours, minutes, and seconds match
0	0	0	0	Alarm day, hours, minutes and seconds match

Special purpose register

The DS1306 has three additional registers (control register, status register, and trickle charger register) that control the real-time clock, interrupts and trickle charger.



Table 10.10.3 : Control register (Read 0Fh, Write 8Fh)

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	WP	0	0	0	1Hz	AIE1	AIE0

- WP (Write Protect)** : Before any write operation to the clock or RAM, this bit must be logic 0. When high, the write protect bit prevents a write operation to any register, including bits 0, 1, and 2 of the control register. Upon initial power-up, the state of the WP bit is undefined. Therefore, the WP bit should be cleared before attempting to write to the device. When WP is set, it must be cleared before any other control register bit can be written.
- 1Hz (1Hz Output Enable)** : This bit controls the 1Hz output. When this bit is a logic 1, the 1Hz output is enabled. When this bit is a logic 0, the 1Hz output is high-Z.
- AIE0 (Alarm Interrupt Enable 0)** : When set to a logic 1, this bit permits the interrupt 0 request flag (IRQFO) bit in the status register to assert INT0. When the AIE0 bit is set to logic 0, the IRQFO bit does not initiate the INT0 signal.
- AIE1 (Alarm Interrupt Enable 1)** : When set to a logic 1, this bit permits the interrupt 1 request flag (IRQF1) bit in the status register to assert INT1. When the AIE1 bit is set to logic 0, the IRQF1 bit does not initiate an interrupt signal, and the INT1 pin is set to a logic 0 state.

Table 10.10.4 : Status register (read 10H)

Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0	0	0	0	0	0	IRQF1	IRQFO

- IRQFO (Interrupt 0 Request Flag)** : A logic 1 in the interrupt request flag bit indicates that the current time has matched the Alarm 0 registers. If the AIE0 bit is also a logic 1, the INT0 pin goes low, IRQFO is cleared when the address pointer goes to any of the Alarm 0 registers during a read or write. IRQFO is activated when the device is powered by V_{CC1} , V_{CC2} or V_{BAT} .
- IRQF1 (Interrupt 1 Request Flag)** : A logic 1 in the interrupt request flag bit indicates that the current time has matched the Alarm 1 registers. If the AIE1 bit is also a logic 1, the INT1 pin generates a 62.5 ms active-high pulse. IRQF1 is cleared when the address pointer goes to any of the alarm 1 registers during a read or write. IRQF1 is activated only when the device is powered by V_{CC2} or V_{BAT} .

Serial interface

- The DS1306 offers the flexibility to choose between two serial interface modes. The DS1306 can communicate with the SPI interface or with a standard 3-wire interface. The interface method used is determined by the SERMODE pin. When this pin is connected to V_{CC} , SPI communication is selected.

~~Serial Peripheral Interface (SPI)~~

- The serial peripheral interface (SPI) is a synchronous bus for address and data transfer and is used when interfacing with the SPI bus on specific Motorola microcontrollers such as the 68HC05C4 and the 68HC11A8.
- The SPI mode of serial communication is selected by tying the SERMODE pin to V_{CC} . Four pins are used for the SPI. The four pins are the SDO (serial data out), SDI (serial data in), CE (chip enable), and SCLK (serial clock).
- The DS1306 is the slave device in an SPI application, with the microcontroller being the master. The SDI and SDO pins are the serial data input and output pins for the DS1306, respectively.
- The CE input is used to initiate and terminate a data transfer.



- The SCLK pin is used to synchronize data movement between the master (microcontroller) and the slave (DS1306) devices.
- The shift clock (SCLK), which is generated by the microcontroller, is active only during address and data transfer to any device on the SPI bus.
- The inactive clock polarity is programmable in some microcontrollers. The DS1306 determines on the clock polarity by sampling SCLK when CE becomes active. Therefore either SCLK polarity can be accommodated. Input data (SDI) is latched on the internal strobe edge and output data (SDO) is shifted out on the shift edge (Fig. 10.11.1).

There is one clock for each bit transferred. Address and data bits are transferred in groups of eight, MSB first.

~~10.11 Programming and Interfacing DS1306 with PIC Microcontroller~~

The interfacing of DS1306 with any microcontroller is as shown in the Fig. 10.11.1. The interface uses SPI method of interface. A crystal of 32.768 kHz is to be connected across the X1 and X2 pin.

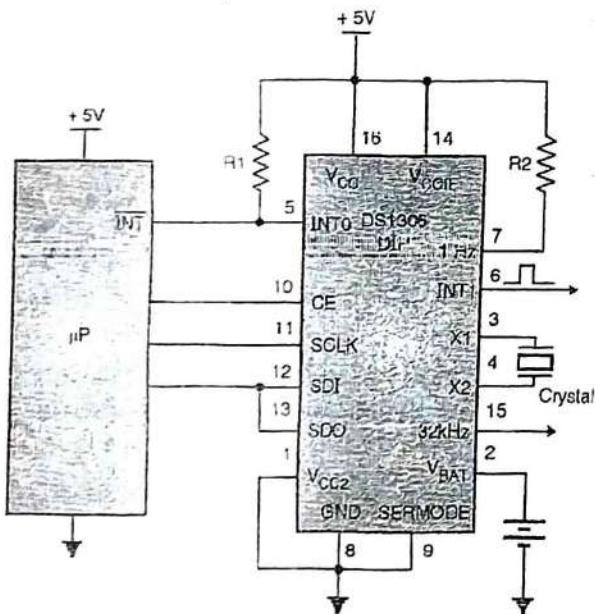


Fig.10.11.1

10.12 Interfacing DS1307 with PIC18

University Questions

Q. Interfacing DS1307 RTC chip using I²C and display date and time on LCD.

SPPU - Dec. 14, Dec. 16, 10 Marks

Q. Draw and explain interfacing of I²C based RTC with PIC18FXXX. Write a code in C.

SPPU - May 15, Dec. 15, 10 Marks

Q. Draw an interfacing diagram to interface RTC with PIC.

SPPU - May 16, Dec. 16, 4 Marks

Q. Draw an interfacing diagram with PIC 18FXXXX, write an initialization program.

SPPU - Dec. 17, May 18, Dec. 18 10 Marks

Fig. 10.12.1 shows the interfacing of DS1307 with PIC18 microcontroller.

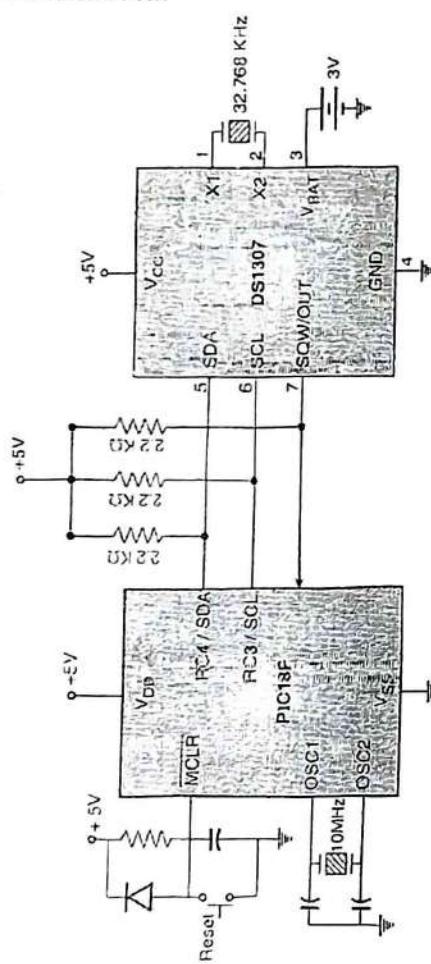


Fig. 10.12.1

The SDA and SCL are used for the serial communication using the I²C protocol. The SDA is the data pin while the SCL is the clock pin as studied in earlier chapters. A simple method can be employed to read the date and time from the RTC as shown in the Ex. 10.12.1.



Ex. 10.12.1 : Draw and explain interfacing of RTC with PIC18FXXX. Also write embedded C program to change the date.

OR : Draw an interfacing diagram with PIC 18FXXXX, write an initialization program.

OR Draw an interfacing diagram with PIC, write an initialization program

SPPU - Dec. 14, Dec. 16, Dec. 17,

May 18, Dec. 18, 10 Marks.

Soln. : Fig. P. 10.12.1 shows the interfacing of PIC18F458 with DS1307 RTC chip using I^C and LCD for reading the time and date and display it on the LCD.

Program

```
#include <PICF458.h>
#define CRYSTAL_FREQ 10000000
//XTAL = 10 MHz

#define RS PORTBbits.RB0
#define RW PORTBbits.RB1
#define EN PORTBbits.RB2
char cmd[5] = {0x38, 0xE, 0x01, 0x06, 0xC0};
char abc[9] = {0xD0, 0x00, 0x80, 0x18, 0x08, 0x07, 0x18,
0x07, 0x20};
void MSDelay (unsigned int x)
{
    unsigned int i, j ;
    for (i = 0 ; i < x ; i++)
        for (j = 0; j < 165 ; j++) ;
}

void CMD (unsigned char cmd)//command function
{
    PORTD = cmd;
    RS = 0;
    RW = 0;
    EN = 1;
    MSDelay (250);
    EN = 0 ;
}

void DAT (unsigned char dat) //display function
{
    PORTD = dat ;
    RS = 1 ;
    RW = 0 ;
    EN = 1;
    MSDelay (250) ;
    EN = 0 ;
}
```

```
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
    TRISChits.RC3 = 1; // configure SCL for
    // input
    TRISChits.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);
}

unsigned char rtc1307_read(unsigned char address)
//RTC DS1307 Read Function
{
```



```

unsigned char temp;
i2c_start();
i2c_send(0xD0);
i2c_send(address);
i2c_restart();
i2c_send(0xD1);
temp = i2c_read();
return temp;
}

unsigned char BCD2Upperch(unsigned char bcd)
{
    unsigned char temp;
    temp = bcd >> 4;
    temp = temp | 0x30;
    return (temp);
}

unsigned char BCD2Lowerch(unsigned char bcd)
{
    unsigned char temp;
    temp = bcd & 0x0F;
    temp = temp | 0x30;
    return (temp);
}

unsigned char sec,min,hour,date,month,year;
void main()
{
    char data[7], time[7] = "Time: ", da[6] = "Date: ";
    int i;
    for(i=0;i<=4;i++)
    {
        CMD(cmds[i]);
        MSDelay(250);
    }
    TRISB = 0x00; // Make port B an output port
    TRISC = 0xFF; // Make port C an input port
    TRISD = 0x00; // Make port D an output port
    i2c_init(); // To Generate the Clock of 100Khz
    i2c_start(); // start the I2C protocol
    for(i=0;i<=8;i++)
    {
        i2c_send(a[i]);
    }
}

```

```

i2c_stop(); //Stop the I2C Protocol
i2c_start(); //Start the Clock again
i2c_send(0xD0);
i2c_send(0x00);
i2c_send(0x00); //start Clock and set the
//second hand to Zero
i2c_stop();
while(1)
{
    sec = rtc1307_read(0x00);
    min = rtc1307_read(0x01);
    hour = rtc1307_read(0x02);
    date = rtc1307_read(0x04);
    month = rtc1307_read(0x05);
    year = rtc1307_read(0x06);
    for(i=0;i<=5;i++)
    {
        DAT(time[i]);
    }
    DAT(BCD2Upperch(hour));
    DAT(BCD2Lowerch(hour));
    DAT(':');
    DAT(BCD2Upperch(min));
    DAT(BCD2Lowerch(min));
    DAT(':');
    DAT(BCD2Upperch(sec));
    DAT(BCD2Lowerch(sec));
    for(i=0;i<=5;i++)
    {
        DAT(da[i]);
    }
    DAT(BCD2Upperch(date));
    DAT(BCD2Lowerch(date));
    DAT('/');
    DAT(BCD2Upperch(month));
    DAT(BCD2Lowerch(month));
    DAT('/');
    DAT(BCD2Upperch(year));
    DAT(BCD2Lowerch(year));
}

```

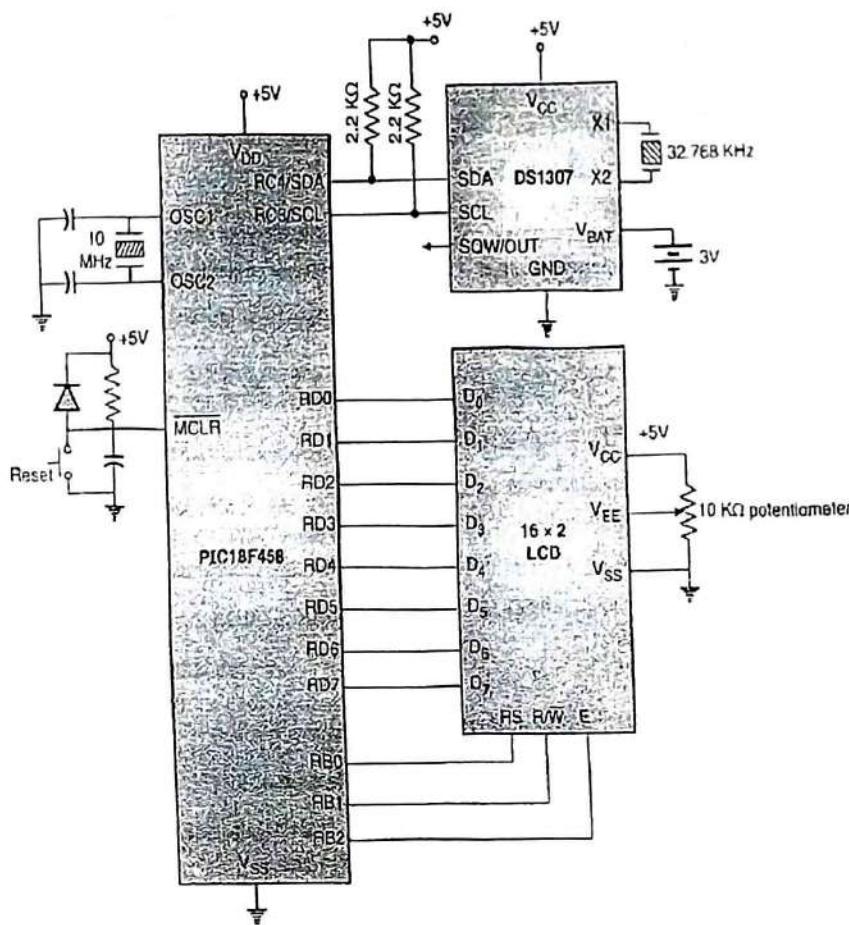


Fig. P. 10.12.1 : Interfacing DS1307 RTC chip using I²C to display date and time on LCD

10.13 Exam Pack (Review and University Questions)

Q. Explain with a neat diagram, interfacing of DAC 0808 with PIC microcontroller.

(Refer Section 10.1.3) (Dec. 14, Dec. 16, 4 Marks)

Q. Explain with a neat diagram, interfacing of DAC 0808 with PIC microcontroller and write a program for rampwaveform generation using DAC interfaced with PIC microcontroller through Port D. Assume the crystal frequency to be 10 MHz. (Refer Ex. 10.1.2)

(Dec. 14, Dec. 16, Dec. 17, 5 Marks)

Q. Write a program to generate a positive ramp (sawtooth) waveform for PIC18 microcontroller.

OR

OR Write a program for sawtooth waveform generation using DAC interfaced with PIC microcontroller through Port B. Assume the crystal frequency to be 10 MHz. (Refer Ex. 10.1.3) (Dec. 16, 5 Marks)

Q. Write a program to generate a negative ramp (sawtooth) waveform for PIC microcontroller.

OR

Write a program for sawtooth waveform generation using DAC interfaced with PIC microcontroller through port B. Assume the crystal frequency to be 10 MHz.

(Refer Ex. 10.1.4) (May 15, Dec. 15, 8 Marks)

Q. Write a program to generate a triangular waveform for PIC microcontroller. (Refer Ex. 10.1.5)

(May 16, 9 Marks)

Q. Explain features of on-board ADC.

(Refer Section 10.2) (Dec. 14, 4 Marks)

Q. Explain features of on-board ADC of PIC18F458.

(Refer Section 10.2) (May 15, 4 Marks)

Q. Explain in detail the functions of the following special function registers: ADRESH and ADRESL of PIC18 microcontroller. (Refer Section 10.2) (May 16, 4 Marks)	Q. 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. (Dec. 15, 8 Marks)
Q. Explain with a neat diagram, interfacing of DAC 0808 with PIC microcontroller and write a program in C language for generation of Square waveform using DAC interfaced with PIC microcontroller through Port B. Use suitable delay. Assume the crystal frequency to be 10MHz. (Refer Section 10.2) (May 18, 9 Marks)	OR Draw the interfacing diagram of voltage measurement and also explain its interfacing procedure. (May 19, 9 Marks)
Q. Explain in detail ADCON 0. (Refer Section 10.2.1) (Dec. 14, 3 Marks)	OR Interface analog voltage 0-5V to internal ADC and display value on LCD Ex. 10.6.3 (Refer Ex. 10.6.4) (Dec. 14, May 15, 8 Marks)
Q. Explain in detail the functions of the following special function register ADCON0 of PIC18 microcontroller. (Refer Section 10.2.1) (May 16, Dec. 16, 4 Marks)	Q. With the help on interfacing diagram and flowchart explain how PIC18 microcontroller can be used to measure temperature using LM35 sensor. (Refer Section 10.7) (May 18, 8 Marks)
Q. Explain in detail the functions of following flags related to onboard ADC of PIC microcontroller. (iv) ADON (Refer Section 10.2.1) (Dec. 17, May 18, Dec. 18, 8 Marks)	Q. With a neat interfacing diagram and explain temperature measurement using PIC 18 microcontroller. (Refer Section 10.7) (Dec. 17, 9 Marks)
Q. Explain ADCON0 register in details. (Refer Section 10.2.1) (May 19, 4 Marks)	Q. Explain interfacing of LM35 for Temperature measurement. (Refer Section 10.7) (May 19, 8 Marks)
Q. Explain in detail ADCON 1. (Refer Section 10.2.2) (Dec. 14, 4 Marks)	Q. Draw an Interfacing of temperature Sensor to PIC using Serial ADC and indicate excess temperature when exceeds the set point by LED.
Q. Explain in detail the functions of the following special function register ADCON1 of PIC18 microcontroller. (Refer Section 10.2.2) (May 16, Dec. 16, 5 Marks)	OR Design a PIC18F458 based system to interface LM35. Write the corresponding C and assembly program For reading and displaying temperature.
Q. Explain in detail the functions of following flags related to onboard ADC of PIC microcontroller. (iii) ADFM (Refer Section 10.2.2) (Dec. 17, May 18, Dec. 18, 9 Marks)	OR Explain the use of PIC-ADC module to interface the Temp sensor LM35 used for accepting the Temp and display on LED connected to port D, write an embedded C program. (Refer Ex. 10.7.1) (Dec. 16, 9 Marks)
Q. Explain ADCON1 register in details. (Refer Section 10.2.2) (May 19, 4 Marks)	Q. Draw interfacing of LM35 with PIC 18F458. Write a program to measure the temperature and display the 10 bit digital equivalent value of the temperature on Port C and Port D. (Refer Ex. 10.7.1(A)) (Dec. 14, May 16, 5 Marks)
Q. Explain the steps involved in programming of A/D converter in PIC18F458 microcontroller using method of polling. (Refer Section 10.2.4) (May 16, Dec. 16, 9 Marks)	Q. Design a Home alarm system considering the parameters of door safely using sensors for detection of person and its movements, Display warning on LCD and LED, light the Lamp connected with Opto-oscillator, Draw the Flowchart with initialization program. (Refer Ex. 10.7.2) (May 15, 9 Marks)
Q. 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. (Refer Ex. 10.6.1) (May 15, 8 Marks)	
Q. Write a program to read a data from ADC and store results from memory location 0x50H onwards. (Refer Ex. 10.6.2) (Dec. 15, 8 Marks)	



- | | |
|---|--|
| <p>Q. Draw an interfacing diagram to PIC - ADC for measuring temperature of room and display on LCD, indicate over temperature by LED, buzzer and relay connected to port, Write an C program for testing.
(Refer Ex. 10.7.3)</p> <p>Q. State features of RTC.
(Refer Section 10.8)</p> <p>Q. Interfacing DS1307 RTC chip using I²C and display date and time on LCD.
(Refer Section 10.12) (Dec. 14, Dec. 16, 10 Marks)</p> <p>Q. Draw and explain interfacing of I²C based RTC with PIC18FXXX. Write a code in C.
(Refer Section 10.12) (May 15, Dec. 15, 10 Marks)</p> | <p>Q. Draw an interfacing diagram to interface RTC with PIC.
(Refer Section 10.12) (May 16, Dec. 16, 4 Marks)</p> <p>Q. Draw an interfacing diagram with PIC 18FXXXX, write an initialization program.
(Refer Section 10.12)</p> <p>(Dec. 17, May 18, Dec. 18, 10 Marks)</p> <p>Ex. 10.12.1 : Draw and explain interfacing of RTC with PIC18FXXX. Also write embedded C program to change the date.</p> <p>OR Draw an interfacing diagram with PIC 18FXXXX, write an initialization program.</p> <p>OR Draw an interfacing diagram with PIC, write an initialization program (Refer Ex. 10.12.1)</p> <p>(Dec. 14, Dec. 16, Dec. 17, May 18, Dec. 18, 10 Marks)</p> |
|---|--|

□□□

11

UNIT - VI

Current Trends in Processor Architecture

11.1 The Acorn RISC Machine

- The first ARM processor was developed at Acorn Computers Limited, England, between 1983 and 1985.
- At that time ARM stood for Acorn RISC Machine, until the formation of Advanced RISC Machines Limited when it was renamed Advanced RISC Machine in 1990.
- The most 32-bit microcontrollers in the embedded industry are ARM microcontrollers, and it has a stake of 75% in this industry.
- With a common architecture, ARM has a very wide range of processor cores and delivers high performance together with low power consumption as well as system cost.
- The ARM processor are of a wide range with solutions for:
 1. Open platforms for wireless, consumer and imaging applications, that use complicate operating system.
 2. Mass storage, automotive, industrial and networking with real time embedded applications.
 3. Smart cards and SIMs requiring high security.

11.1.1 RISC Properties

A RISC system must satisfy the following properties :

1. Single-cycle execution of all (or at least 80 percent) instructions.
2. Single-word standard length of all instructions.
3. Small number of instructions (≤ 128).
4. Small number of instruction formats (≤ 4).
5. Small number of addressing modes (≤ 4).
6. Memory access possible by load and store instructions only.
7. All operations, except load and store, are register to register i.e. within the CPU.
8. It must have a hardwired control unit.
9. It must also have a relatively large (at least 32) general-purpose CPU register file.

11.1.2 Register Window

1. Since there are a huge number of registers in a RISC processor, it can be useful in saving the latency period during procedure call. Parameter passing in is possible by the register window. This policy also allows reasonable HLL support in RISC designs.
2. The register file is subdivided into groups of registers, called register windows.
3. A certain group of 'i' registers, suppose R_0 to $R_{(i-1)}$, are designated as global registers.
The global registers are accessible to all procedures running on the system at all the times.
4. On the other hand, each procedure is assigned a separate window within the register file, which is not accessible to other procedures.
5. The window base (first register within the window) is pointed to by a field called current window pointer (CWP) located in the CPU's status register (SR).

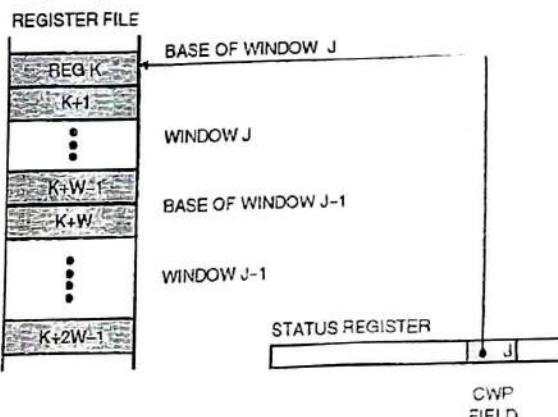


Fig. 11.1.1 : Simple non-overlapping register window

6. If the currently running procedure is assigned the register window J, hence taking up registers K, K+1, ..., K+W-1 (where W is the number of registers per window), the CWP contains the value J, hence pointing to the base of window J.

- If the next procedure to execute takes up window $J+1$, the value in the CWP field will be incremented accordingly to point to $J+1$.
7. Register windowing can work more efficiently for parameter passing between calling and called procedures by partial overlapping of the windows. The last N registers of window J will be the first N registers of window $J+1$.
8. If the procedure that is using window 'J', calls another procedure, then the new procedure will be assigned the next window i.e. ' $J+1$ ', and the former procedure can pass N parameters to the called procedure by placing their values into registers $(K+W-N)$ to $(K+W-1)$. Since the two windows are overlapping, the same registers will be available to the called procedure without actually moving the data.

11.1.3 Miscellaneous Features of RISC Systems

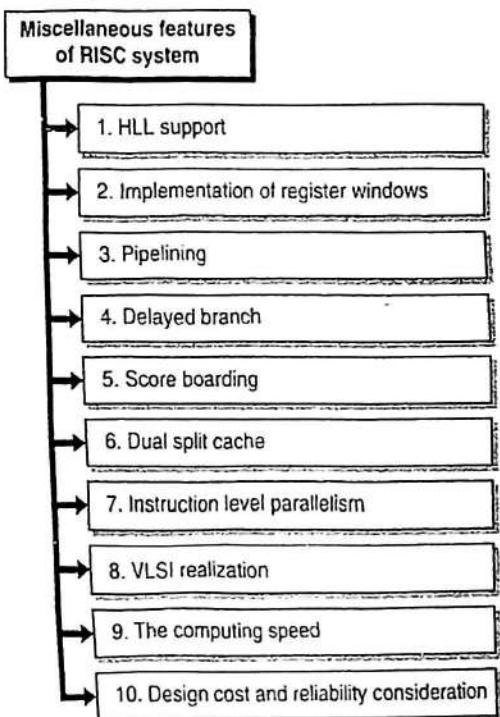


Fig. 11.1.2 : Miscellaneous Features of RISC

1. HLL support

- (a) The support for High Level Language (HLL) features is mandatory in the design of any computing system.

- (b) The procedure call-return and parameters passing is the most time-consuming operation in typical HLL programs.
- (c) HLL support is provided in RISC machines by supporting efficiently the handling of local variables, constants, and procedure calls, while leaving less frequent HLL operations to instructions sequences and subroutines.
- (d) One of the mechanisms supporting the handling of procedures, and their parameter passing in particular, is the feature of the register window.
2. Implementation of register windows
- (a) A CISC processor's control unit takes up a large percentage of the chip area, leaving very little space for other subsystems and basically not permitting a large register file, needed for an efficient implementation of windowing.
- (b) A RISC processor's control unit makes up a much smaller percentage of the chip area, yielding the necessary space for a large register file.
- (c) And hence implementation of register windowing (that requires huge number of registers) is possible in RISC processors
3. Pipelining
- (a) Pipelining was used on various CISC systems even before the RISC approach became popular.
- (b) But, a streamlined RISC can handle pipelines more efficiently.
4. Delayed branch
- (a) The problem occurs in a system where instructions are prefetched, right after a branch.
- (b) If the branch is conditional, and the condition is not satisfied, then the next instruction, which was prefetched, is executed, and since no branch is to be performed, no time is lost.
- (c) But, if the branch condition is satisfied, or the branch is unconditional, the next prefetched instruction is to be flushed and other instruction pointed to by the branch address is to be fetched in its place. The time required to prefetch the flushed instruction is wasted.
- (d) Such waste of time is solved by using the delayed branch approach.



- (e) In this approach, the instructions are reshuffled such that the operation does not change the result.
- (f) A successful branch is assumed and the execution of the branch is delayed until the already prefetched instructions are executed. Hence, no time is lost and there is no change in the intended program operation.
- (g) But the compiler has to take care that the instructions followed by the branch instructions are to be executed irrespective to the branch to be taken or not.

5. Scoreboarding

- (a) Another problem in instruction pipelines is that of data dependency.
- (b) The data in some register put by instruction 1 may be required by instruction 2; and before the value in the register is available, the instruction 2 may be ready for execution yielding a possibly incorrect result.
- (c) A method used to solve this problem is called as **scoreboarding**.
- (d) A special CPU control register i.e. the **scoreboard register**, is required for this purpose.
- (e) If there are 32 registers, a scoreboard register 32-bit long will be required; each of its bit represents one of the 32 CPU registers.
- (f) If a register window 'x' is involved for the execution of the task, then the corresponding bit in the scoreboard register is set indicating no other task can use this register window. Once the task execution is over, the corresponding bit is cleared.
- (g) This instruction will now be executed as soon as the execution of the instruction, which caused bit 'i' to be set, is completed.

6. Dual split cache

Another feature, implemented in not only a CISC but also RISC systems, is separated data and code caches, or split cache.

7. Instruction Level Parallelism (ILP)

Superscalar and super pipelined designs are also mostly implemented in a RISC design.

8. VLSI realization

- (a) The chip area, dedicated to the realization of the control unit, is considerably less. Therefore, on a RISC VLSI chip, there is more area available for other features (cache, FPU, part of the main memory, memory management unit, I/O ports, etc).
- (b) As a result of the considerable reduction of the control area, a large number of CPU registers can fit on-chip.
- (c) By reducing the control area on the VLSI chip and filling the area by numerous identical registers, the **regularization factor** (which is defined as, ratio of chip area utilized by other features to the chip space required by control unit) of the chip increases. The higher the regularization factor, the lower is the VLSI design cost.

9. The computing speed

- (a) A simpler and smaller control unit in RISC requires fewer gates. This results in shorter propagation paths for control unit signals, decreasing the delay time for control signals and hence yielding a faster operation.
- (b) A significantly reduced number of instructions, formats, and addressing modes results in a simpler and smaller decoding system, resulting in faster decoding operation.
- (c) A hardwire-controlled system can hence be implemented, with a reduced control unit that will in general be faster than a micro programmed control unit.
- (d) A relatively large CPU register file also reduces CPU-memory traffic to fetch and store data operands.
- (e) A large register set can also be used to store parameters to be passed from a calling to a called procedure, to store the information of a process that was interrupted by another.

10. Design cost and reliability considerations

- (a) It takes a shorter time to complete the design of a RISC control unit, because of the smaller instruction set, fixed instruction length and less instruction formats.



Thus contributing to the reduction in the overall design cost.

- (b) A simpler and smaller control unit will obviously have a reduced number of design errors and, therefore, a higher reliability.

11.1.4 ARM and RISC Design Philosophy

Q. What are the main features of ARM architecture?
(4 Marks)

The ARM architecture incorporated a number of features inherited from RISC design like :

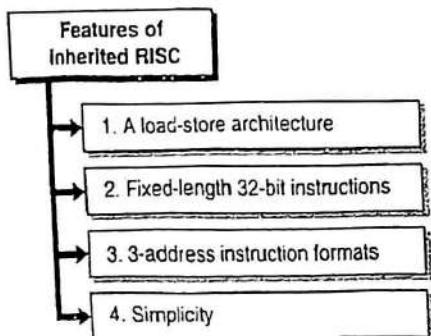


Fig. 11.1.3 : Architecture Inheritance

1. A load-store architecture

- A load-store architecture means that ALU instructions cannot access memory, they can operate only on registers.
- Data transfer between memory and processor is possible only by Load and Store instructions.
- LOAD instruction is used to load the data from memory into the processor's register.
- STORE instruction is used to store data in memory from the processor's register.
- The data in the registers can be processed by the ALU.

2. Fixed-length 32-bit instructions

- All the instructions are 32-bit in length.
- This fixed length instructions make the design of the control unit simpler.

- Fixed length instructions also make it possible to use hardwired control unit, which makes the operation of control unit faster.

3. 3-address instruction formats

- This is also referred to as 3-operand instructions
- The two source operands are different than the destination operand for e.g. in the add instruction given below

ADD d, S₁, S₂;

Operation of above instruction: d := S₁ + S₂

f bits	n bits	n bits	n bits
function	op 1 addr.	op 2 addr.	dest. addr.

Fig. 11.1.4 : A 3-address instruction format.

- Another feature of ARM is

4. Simplicity

Since the ARM has simple hardware and a strong instruction, it results in a better code density compared to general RISC. Although ARM is said to be a RISC processor, it has not implemented certain standard features of RISC processor. Here are some of them:

1. Register Windowing
2. Delayed Branching
3. Single cycle execution of all instructions.

11.2 ARM Family Core Architecture

Q. Draw and explain the ARM family core architecture.
(8 Marks)

- The ARM core is an engine within the system. It is responsible for fetching the instructions from the memory and executing them.
- The size of ARM core is very small; hence many additional components are added to the same chip.
- We will study the basic structure of the ARM core, as selection of an ARM core is the most critical decision while designing a system.



- Fig. 11.2.1 shows the ARM core data flow model. It supports Von-Neumann architecture.

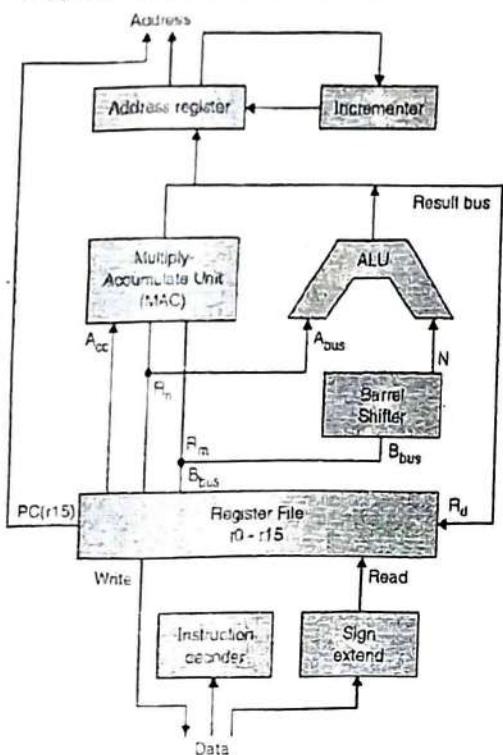


Fig. 11.2.1 : ARM core data flow model

- Now, let us see the different functional units :

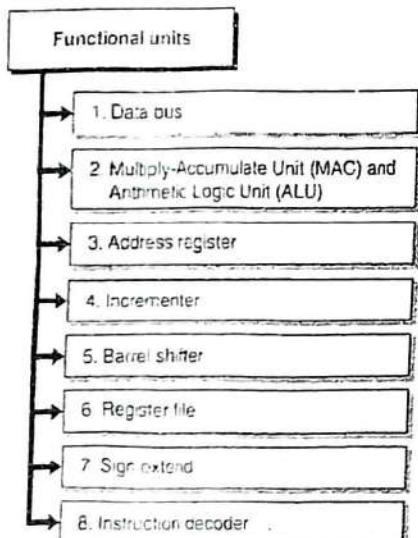


Fig. 11.2.2 : Functional Units

1. Data bus

- The data enters the ARM core through the data bus .
- The same data bus is used by all the instructions as the ARM core uses Von-Neumann architecture.

- There are two buses **A_{bus}** and **B_{bus}**. The data can be an instruction opcode.

2. Multiply-Accumulate Unit (MAC) and Arithmetic Logic Unit (ALU)

- Generally the ARM instructions are of three operands. In order to store the two input operand registers **R_m** and **R_n** are used.
- The ALU or MAC reads the two operand values from **R_m** and **R_n** registers using the **A_{bus}** and **B_{bus}**.
- Then the desired operation is done and the result is stored using the result bus or **C_{bus}** in the destination register **R_d**.

3. Address register

It is responsible for storing the address generated by the load and store instructions and placing it on the address bus.

4. Incrementer

- The incrementer is responsible for upgrading the address register contents before the core reads the next register value from memory location or writes the next register value to memory location.
- In case of exception / interrupt the execution of instructions is changed.

5. Barrel shifter

- The **R_m** register contents can be preprocessed in the barrel shifter before applying them to the ALU.
- A wide range of shift operations are possible using the barrel shifter and the ALU.

6. Register file

The register file is a bank of sixteen 32-bit registers. The registers are used for storing the data.

7. Sign extend

- Some of the instructions require signed values. Whenever the processor reads an 8 or 16 bit signed number from the memory, the sign extend hardware converts the numbers to 32 bit values and places the number in the register file.

8. Instruction decoder

- The instruction decoder decodes the instruction opcode read from the memory and then the instruction is executed.
- The ARM cores are of three types :



1. Application Core
 2. Embedded Core
 3. Secure Core
- Since the architecture of ARM based processor takes very less space and performs a lot of operations it is fit to be used in embedded systems.

11.3 Versions and Variants

- In the 1980s Apple Computer started working with a company called as Acorn. This work was such crucial that Acorn spun off the design team in 1990 into a new company called Advanced RISC Machines (ARM).
- Later, it became ARM Limited and it was floated on the London Stock Exchange and NASDAQ in 1998.
- The ARM was developed for the first time with its samples in 1985 and was called as ARM1, and the first "real" production systems was called as ARM2 developed in 1986.
- This ARM2 had a 32-bit data bus and 26 bit address bus with 32 bit registers.
- There were 16 such 32-bit registers with one of them serving as a program counter.
- Some of the bits of program counter were used as control and status flags.
- This ARM2, that used 32000 transistors, was almost the most simple and very useful 32-bit microprocessor in the world.
- The ARM2 was succeeded by ARM3, that had a cache memory of 4KB and hence improving the performance.
- Later ARM6 was developed and then the most successful implementation of ARM i.e. the ARM7TDMI which is present in hundreds of millions of cellular phones.
- Freescale (spun off of Motorola in 2004), IBM, Texas Instruments, Nintendo, Philips, VLSI, Atmel, Sharp, Samsung and STMicroelectronics have also licensed the basic ARM design for various uses.

- The ARM chip has become one of the most used CPU designs in the world, found in everything from hard drives, to mobile phones, to routers, to calculators, to children's toys.
- Today it accounts for over 75% of all 32-bit embedded CPU's.
- The ARM implementation has five versions of the instructions set that are defined till date.
- The versions are represented by version numbers 1 to 5.
- In order to specify collections of additional instructions included the instruction set, some of the versions are qualified with variant letters.
- M variant denotes addition of four instructions and T variant denotes additions of entire thumb instruction set.

The five versions are as follows

1. Version 1
 - This version was implemented on ARM 1 processor. It has 26 bit address space.
 - It supports branch instructions including a branch and link instruction designed for subroutine calls.
 - It supports a software interrupts instruction, for use in making operating system calls.
 - It supports byte, word and multi-word load-store instructions.
 - It supports the data processing instructions also.
 - Version 1 now a days has become obsolete.
2. Version 2
 - This version is a updated version of version 1. It also has 26 bit address space.
 - It supports coprocessor supported instructions.
 - It supports multiply and multiply-accumulate instructions.
 - It includes load and store instructions called SWP and SWPB in a variant called version 2.
 - It supports two more banked registers in fast interrupt mode.
 - Version 2 is also obsolete.



3. Version 3

- This version has 32 bit address space.
- In this version with a view to retain the contents of CPSR register in the case of an exception two new registers were added
- The program status information which was stored in register R15 is now stored in the CPSR (Current Program Status Register) and the SPSR (Saved Program Status Register).
- In version 3 two new instructions (MRS and MSR) were added to access the CPSR and SPSR registers.
- It supports two new processor modes. The new processor modes help to use Data Abort, undefined instruction exceptions in the code.

4. Version 4

- The version 4 is an extension of version 3.
- It includes additional features like :
 - (i) load / store halfword instructions.
 - (ii) instruction transfer to T state in T variants.
 - (iii) new privileged processor mode using the user mode register
- Version 4 made a clear distinction of the instructions that should cause the exception to be taken.
- Backwards-compatibility support for 26 bit architecture ceased to be obligatory in version 4.

5. Version 5

- The version 5 is an upward extension of version 4.
- It includes additional instructions that improve the efficiency of ARM interworking in T variants.
- Version 5 allows same code generation techniques to be used for T and non T variants.
- It supports an additional software breakpoint instruction
- Version 5 also supports a count leading zeros (CLZ) instruction that allows efficient integer divide and interrupt prioritization.
- Version 5 tightens the definition of how flags are set by multiply instructions.
- It supports co-processor instructions.

The variants are as follows

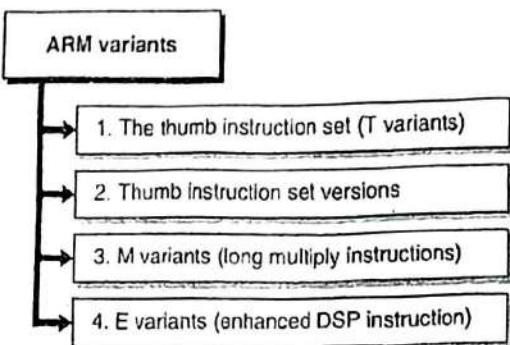


Fig. 11.3.1 : ARM Variants

1. The Thumb Instruction Set (T variants)

- It is a subset of the ARM instruction set. The thumb instruction set has 16 bit instructions. i.e. the Thumb instructions are half the size of ARM instructions.
- The Thumb instruction set has greater code density than the ARM instruction set.
- The drawbacks of the Thumb instruction set are that it requires more instructions to complete the same task.
- Hence the ARM instruction set is best for maximizing the performance.
- The Thumb instruction set does not include instructions required for exception handling.
- Due to these drawbacks the Thumb instruction set is used in conjunction with a suitable version of the ARM instruction set.
- The Thumb instruction set is not valid prior to version 4 of the ARM architecture. It is signified by letter T.

2. Thumb instruction set versions

- There are two versions of the Thumb instruction set i.e. version 1 and version 2.
- The version 1 of the Thumb instruction set is used in T variants of the ARM architecture version 4 and version 2 is used in T variants of the ARM architecture version 5.
- The Thumb instruction set version 2 modifies the instructions to improve the efficiency of ARM interworking.
- The Thumb instruction set tightens the definition of how flags are set by multiply instructions.
- The Thumb instruction set includes the breakpoint instruction.
- These modifications are related to the changes between ARM architecture versions 4 and 5.



3. M variants (Long Multiply Instructions)

- The ARM instruction set includes four additional multiply instructions for producing 64 bit multiply and multiply and accumulate results.
- These instructions imply the existence of a multiplier that is larger than the minimum. Its presence is denoted by variant letter M.
- The long multiply instructions were first included in the variant of version 3 and exists in all after version 3.

4. E variants (Enhanced DSP instructions)

- The E variants of the ARM instructions set consist of a number of ARM instructions that enhance the performance of an ARM processor for digital signal processing applications.
- The E variants support load, store, co-processor register transfer instructions that act on double word data.
- They include a cache preload instruction (PLD).
- They include several new multiply and multiply-accumulate instructions acting on 16 bit data.
- They were first included in the ARM architecture version 5T. They are not valid in non-T or non-M variants of the ARM architecture.

11.3.1 Terms Related to ARM Instruction Set

- Given below are some terms used in the ARM instruction set:
- {cond}: Condition field :
- We know that an ARM instruction is 32 bit long.
- The 4 MSBs of the instructions are called as the condition field. The condition field is of 4 bits, hence there are 16 conditions as listed in Table 11.3.1.
- Fig. 11.3.2 shows the condition field.



Fig. 11.3.2 : ARM condition field

Table 11.3.1

Condition field	{cond}	
Suffix	Description	Condition flag state
EQ	Equal	Z = 1
NE	Not Equal	Z = 0
CS	Unsigned higher or same	C = 1
CC	Unsigned lower	C = 0
MI	Negative	N = 1
PL	Positive 0 or zero	N = 0
VS	Overflow	V = 1
VC	No overflow	V = 0
HI	Unsigned higher	C = 0 and Z = 0
LS	Unsigned lower or same	C = 1 and Z = 1
GE	Signed greater than or equal	N = 1 and V = 1 or N = 0 and V = 0.
LT	Signed less than	N = 1 and V = 0 or N = 0 and V = 1.
GT	Signed greater than	Z = 0 and either N = 1 and V = 1 or N = 0 and V = 1.
LE	Signed less than or equal	Z = 1 or N = 1 and V = 0 or N = 0 and V = 1.
AL	Always (unconditional)	
NV	Reserved	

11.3.2 ARM 7, ARM 9 and ARM 11 Features

University Question

Q. List features of ARM7 processor. How it is different from pure RISC processor?

SPPU - Feb. 17, 6 Marks

The features of various ARM family of processors will be seen in this section. These features will be better understood in later sections, when we understand the working of ARM processors.

11.3.2(A) ARM 7 Features

The special features of ARM 7 are as listed below:

1. It is a 32-bit processor.
2. It has AHB bus interface.
3. It supports Thumb instruction set.
4. It is a RISC processor.
5. Later versions of ARM 7 have DSP instructions.



6. Later versions of ARM7 also have Jazelle instruction set support.
7. Its clock frequency is from 100 to 133 MHz.

11.3.2(B) ARM 9 Features

The features of ARM 9 are as listed below:

1. It is a 32-bit processor.
2. It has AHB bus interface.
3. It supportsThumb instruction set.
4. It is a RISC processor.
5. Later versions of ARM7 have DSP instructions.
6. Later versions of ARM7 also have Jazelle instruction set support.
7. Its clock frequency is from 180 to 210 MHz.
8. It has 4KB code cache memory.
9. It has 4KB data cache memory.
10. It has on-chip memory management unit.

11.3.2(C) ARM 11 Features

The features of ARM 11 are as listed below:

1. It is a 32-bit processor.
2. It has AHB bus interface.
3. It supportsThumb instruction set.
4. It is a RISC processor.
5. ARM 11 has 32 instructions.
6. ARM 11 also have Jazelle instruction set support.
7. Its clock frequency is from 800 MHz to 1 GHz.
8. It has 4 to 64 KB code cache memory.
9. It has 4 to 64 KB data cache memory.
10. It has on-chip memory management unit.
11. It has one to four-core SMP cluster.
12. It also supports SIMD.
13. It has 6 to 8 stages of instruction pipelining.
14. It has separate load-store and arithmetic pipeline.
15. It has branch prediction and return stack logic.
16. It has optional networks coupled with DMA for multi-media applications.

11.3.2(D) Comparison of ARM7, ARM9 and ARM11

University Questions

- Q. Compare ARM7, ARM9 and ARM11 series processors stating Features. (5 Marks)**
- Q. Compare various versions of ARM with respect to features, advantages, power dissipation. (5 Marks)**
- SPPU - Oct. 11, Feb. 12, Feb. 13, May 15, Dec. 15, Feb. 16, May 17, 5 Marks**
- Q. Compare the ARM7, ARM9 and ARM11 processors. (5 Marks)**
- SPPU - Mar. 19, 5 Marks**

Sr. No.	Feature	ARM 7	ARM 9	ARM 11
1.	Memory Organizations	von Neumann	Harvard	Harvard
2.	Pipeline stages	3	5	8
3.	Memory Management Unit	No	Yes	Yes
4.	Memory Protection Unit	No	Yes	Yes
5.	Performance	High	Higher	Highest
6.	Cache	No	Yes	Yes
7.	Write Buffers	No	Yes	Yes
8.	DSP Capability	No	Yes	Yes
9.	SIMD instructions	No	No	Yes
10.	Out-of-order Execution	No	No	Yes
11.	Dynamic Branch Prediction	No	No	Yes
12.	Load/Store and ALU parallelism	No	No	Yes

Applications of ARM 7

1. Networking - High Performance Applications Use Cortex A
2. Fiber to the Home Devices - Less demanding applications use Cortex R or M.
3. Routers



Applications of ARM 9 and 11

1. Storage Controllers
2. Toys
3. Gadgets
4. Industrial
5. Home Automation
6. Sensors, signal processing

11.3.2(E) Advantages and Suitability in Embedded Applications

The following advantages of ARM processors make it suitable to be used in Embedded Systems :

1. ARM has 32-bit architecture but supports 16 bit or 8 bit data types also.
2. ARM is programmable as little endian or big endian data alignment in memory.
3. ARM provides the advantage of using a CISC in terms of functionality along with the advantage of an RISC in terms of faster program implementation as well as reduced code lengths.
4. ARM processor has an RISC core for processing.
5. Combination of RISC and CISC features-ARM supports to a complex addressing modes based instruction set.
6. Due to the instant availability of the register word to the execution-unit.
7. Reduced code lengths-Most instructions use registers as operands.
8. Few bits in the instruction specify a register as operand. 5, 16 or 32 bits specify a memory address as operand and the displacement bits in the instruction.

11.4 The ARM Programmers Model

University Questions

- Q. Draw and explain the ARM programmers model. (6 Marks)
Q. Explain programmers model of ARM with the help of neat diagram. (5 Marks)
Q. Explain the term : Banked register in ARM.
SPPU - Oct 12, 4 Marks
- Q. Write significance of special registers R0, R1, R2, R3 in ARM. **SPPU - May 15, Dec 15, 3 Marks**
- Q. Explain programmer's model of ARM processor. **SPPU - May 16, 6 Marks**
- Q. Draw and explain the complete ARM register set with concept of changing modes on switch-on. **SPPU - May 18, 5 Marks**

- The ARM processor can be operated in seven different modes. The current mode of the processor decides the availability of the registers to the programmer.
- Fig 11.4.1 shows the programming model of the ARM processor. The ARM processor has 37 thirty two bit registers.
- The registers include of
 - (i) A dedicated program counter.
 - (ii) A dedicated current program status register.
 - (iii) 3 dedicated saved program status registers.
 - (iv) 30 general purpose registers.
- These registers are organized as banks and the registers accessible to the code depend on the mode of processor.
- The registers R0 to R12 are common i.e. the same physical registers are accessible in all the modes of the processor. R13, R14, R15, SPSR and the CPSR are available separately in each mode i.e. these registers are physically different in different modes of the processor.
- The remaining registers are different physically only in a fast interrupt mode.
- Fig. 11.4.1 shows the programming model of the ARM processor.

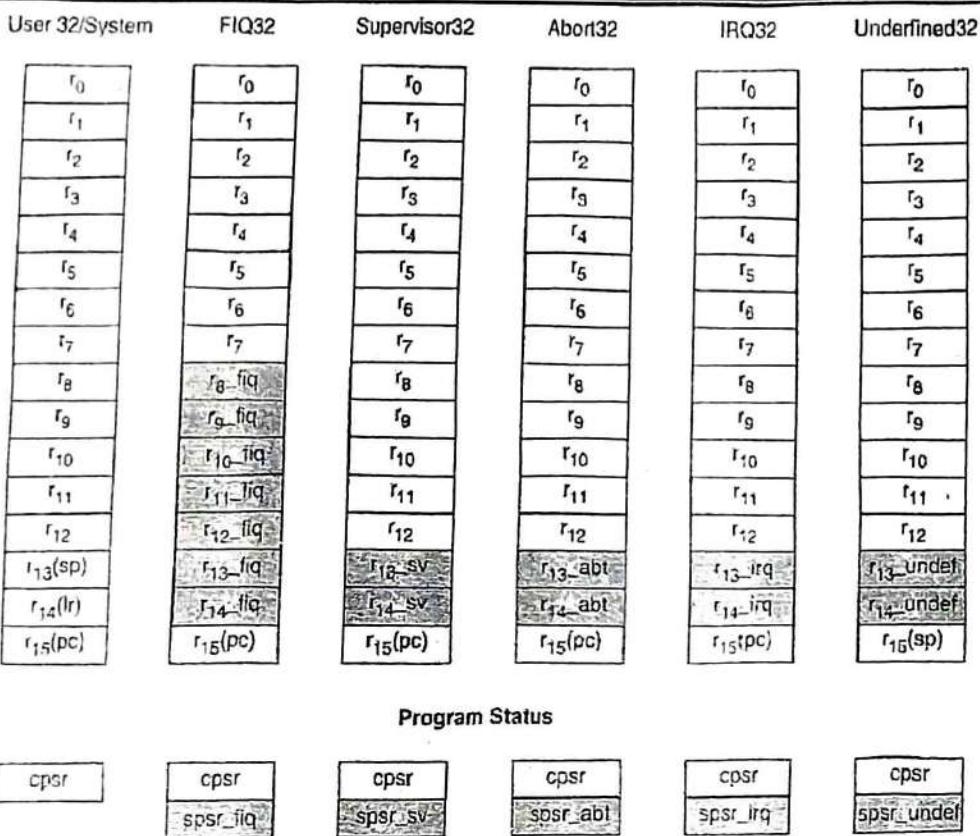


Fig. 11.4.1 : ARM programming model

1. General purpose registers
2. Stack Pointer (SP-R₁₃)
3. Link Register (LR-R₁₄)
4. Program Counter (PC-R₁₅)
5. Registers R₀-R₇
6. Registers R₈-R₁₄

1. General purpose registers

- The registers R₀ to R₁₂ are used as general purpose registers.
- They can be used for holding the data or addresses.
- Depending on the purpose, the registers R₁₃ to R₁₅ can also be used as general purpose registers.

2. Stack Pointer (SP-R₁₃)

- Register R₁₃ is a stack pointer as shown in Fig. 11.4.1.
- It stores the top of the stack in the current processor mode.

3. Link Register (LR-R₁₄)

Register R₁₄ is a link register. It is used for storing the return address in case of subroutine calls.

4. Program Counter (PC-R₁₅)

- Register R₁₅ is the program counter.
- It stores the address of the next instruction to be fetched from the memory by the processor.
- It can be used in place of registers R₀ to R₁₄. Hence it is considered as one of the general purpose registers.
- Its use is sometimes restricted depending on the type of instruction.
- Generally it is used as a pointer to the instruction which is two instructions after the instruction being executed.
- We know that all ARM instructions are 32 bit long, i.e. they are aligned on word boundary. Hence the PC value is stored in bits [31:2] with bits [1:0] equal to zero.

5. Registers R₀-R₇

- These registers are called as **unbanked registers**. They are general purpose registers and can be used wherever general purpose registers can be used.
- Each register is a 32 bit physical register in all the processor modes. The registers R₀-R₇ do not have any specific use.

6. Registers R₈-R₁₄

- These registers are called as **banked registers**. This is because the physical size of each of these registers depends on the current processor mode.
- Whenever a general purpose register is used, any of these registers can be accessed.
- Out of 37 registers, only 20 registers which are shown shaded in Fig. 11.4.1 are used as banked registers.
- Fig. 11.4.1 also indicates the mode of the bank registers. Banked registers of a particular mode are denoted by r number_mode e.g. abort mode has banked registers r_{13_abt}, r_{14_abt} and spsr_abt.

Note: Register R₈ to R₁₂ are subject to restrictions on the use of bank registers in systems where FIQ mode is never used. Hence, only one physical version of the register is always in use.

- Registers R₈ to R₁₂ do not have any specific applications. However, for interrupts that are simple enough to be processed using these registers, the existence of separate FIQ mode versions of register allows fast interrupt processing.
- Each of registers R₈ to R₁₂ have two banked physical registers such that one register is used in all processor modes except FIQ mode and the other is used in FIQ mode.
- Registers R₁₃ and R₁₄ have six banked physical registers, such that one register is used in user and system mode while the each of remaining five registers is used in one of the five exception modes.
- Wherever it is essential to be specific about the version used, use the names of form : r₁₃_mode e.g. r_{13_svc} (supervisor mode).
- Register R₁₅ is Stack Pointer (SP). In the ARM instruction set there are no specific instructions or functionality that use register R₁₅. However in the Thumb instruction set there are such instructions.

- Each exception mode has its own banked version of R₁₅ which should be initialized to point to a stack dedicated to that exception mode.
- The exception handler stores the values of other registers on the stack at the start. On return by reloading the values back to the registers will ensure that the exception handler will not corrupt the state of program which was in execution at the time when exception occurred.
- Register R₁₄ is a link register. It can be used as a general purpose registers. It has two special functions. They are,
 - (a) In each mode, the mode's own version of register R₁₄ is used to hold the subroutine return address. The return task is achieved by loading R₁₄ back to the program counter. This task can be obtained by methods given below :
 - (i) Execute either of these instructions :

```
MOV R15, R14
      or
MOV PC, LR
      OR
BX LR
```

 - (ii) While entering the subroutine, store the contents of register R₁₄ to the stack using instruction STMFD SP!, {<registers>, LR} and to return use instruction LDMFD SP! {<registers>, PC}.
- (b) In case of an exception, the exception mode's version of R₁₄ is set to the exception return address. The exception return is done processed in the same way as the subroutine return, but using different instructions to ensure full restoration of the program state.

11.5 Program Status Registers

University Questions

- Q. Draw and explain the structure of the program status register. (8 Marks)
- Q. Explain term CPSR register and processor modes. **SPPU - Oct. 12, 4 Marks**
- Q. Draw and explain CPSR register structure of LPC 2148. **SPPU - May 15, Dec. 15, May 17, 4 Marks**
- Q. Explain CPSR register in detail. What is the need of SPSR register ? **SPPU - Feb. 16, 6 Marks**



- Q.** Draw and explain the CPSR of ARM in detail.
SPPU - Mar. 18, 5 Marks
- Q.** Draw the structure of CPSR and explain the functions of each bit.
SPPU - Dec. 18, 6 Marks
- Q.** Describe CPSR and SPSR of ARM 7 in detail.
SPPU - Mar. 19, 5 Marks

- The ARM architecture supports two program status registers. They are the **current program status register (CPSR)** and the **saved program status register (SPSR)**.
- The CPSR is accessible in all the processor modes while the SPSR is accessible in privileged modes.
- The CPSR contains condition code flags, interrupt disable bits, current processor mode and other control and status information
- Each exception mode contains a saved program status register (SPSR). It is responsible for holding the value of CPSR incase the exception occurs. Fig. 11.5.1 shows the format of the CPSR and SPSR registers.

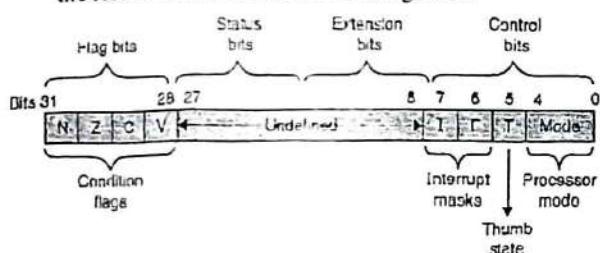


Fig. 11.5.1 : CPSR and SPSR register format

Now let us see description of each bit.

1. Control Bits (Bits 0-7)

- Bits (0-7) are the control bits of CPSR and SPSR register. These bits change incase an exception occurs.
- The control bits can be altered by the software only when the processor is in privilege mode.

2. Bits 0-4 (Processor Mode)

These bits determine the processor mode. The modes are listed in Table 11.5.1.

Table 11.5.1 : Processor modes

Processor mode	Bits				
	4	3	2	1	0
User	1	0	0	0	0
Fast Interrupt Request (FIQ)	1	0	0	0	1
Interrupt Request (IRQ)	1	0	0	1	0

Processor mode	Bits				
	4	3	2	1	0
Supervisor	1	0	0	1	1
Abort	1	0	1	1	1
Undefined	1	1	0	1	1
System	1	1	1	1	1

3. Bit 5 (Thumb Bit) (Architecture version 4T only)
 - The Thumb bit determines the state of the ARM core. The state of core determines the instruction set that is being executed.
 - There are 3 instruction sets : ARM, Thumb and Jazelle.
 - When the T bit is 1, the processor is in the Thumb state and executes the Thumb instructions.
 - Some ARM processor have additional bits to determine the state of the processor. e.g. J bit. The J bit is present in flags field is Jazelle enabled processors.
 - In such processors the J and T bits decide the state of the processor.
 - When both the T and J bits are zero, the processor is in the ARM state and executes ARM instructions. When the T bit is zero and J bit is 1, the processor is in Jazelle state and executes Jazelle instructions.
 - To change the state of processor specialized branch instructions are used.

Comparison of ARM and Thumb Instruction set

Sr. No.	Parameter	ARM instruction set	Thumb instruction set
1.	CPSR bit T	T = 0	T = 1
2.	Instruction size	32 bit	16 bit
3.	Core instructions	58	30
4.	Conditional execution	For most of the instructions	For branch instructions only.
5.	Program status register	Read-Write in privileged mode	No direct access.
6.	Register access	All 15 general purpose registers + PC	8 general purpose registers + PC + 7 high registers.

Sr. No.	Parameter	ARM instruction set	Thumbs instruction set
7.	Data processing Instructions	Access to Barrel shifter and ALU	Separate Barrel shifter and ALU instructions.
8.	Versions	Supported by all versions	Supported to all versions after version 4
9.	Exception handling instructions	Supported	Not supported
10.	Instructions required to do a task	Less	More than ARM instructions

4. Jazelle instruction set

- The Jazelle instructions are 8 bit in size. These instructions are designed with the view to increase the speed of execution of Java bytecodes.
- The Jazelle instructions are a hybrid mixture of software and hardware.
- Over 60% of the Java bytecodes are implemented in hardware while remaining are implemented in software.
- The Jazelle technology and a modified version of Java virtual machine are required to execute Java bytecodes.
- It is a closed instruction set. It is not openly available. To use Jazelle, a licensed software from ARM Limited and Sun Microsystems is required.

5. Bits 6, 7 (Interrupt Masks)

- The ARM core has two interrupts. They are fast interrupt request (FIQ) and interrupt request (IRQ).
- These interrupts are maskable. Bit 6 (F) controls the FIQ and Bit 7 (I) controls the IRQ. For their mask enable/disable.
- When bit is set i.e. F = 1 or I = 1 the respective interrupt is masked or disabled. When bit is clear, the interrupt is unmasked or available.

6. Condition code flags

- The flags are updated by the operations performed by the ALU. The flags in the CPSR can be tested by the instructions to determine whether the instruction is to be executed or not.

- The flags can be updated by execution of a comparison instruction or by execution of arithmetic, logical or move instruction where the destination register is not R₁₅.

7. Bit 27 (Saturation Flag : Q)

- It is present on the DSP extension of ARM processor core.
- This bit is set if an overflow or saturation occurs. The hardware can only set this flag. In order to clear this bit we need to write to CPSR directly.
- In case of SPSR this bit is used to hold and restore the CPSR Q flag in case of exceptions.

8. Bit 28 (Overflow Flag : V)

- This flag is set if an overflow occurs while addition or subtraction.
- For operations other than addition/subtraction overflow is unchanged.

9. Bit 29 (Carry Flag : C)

This flag is set if one of the following conditions exist.

- (i) For addition alongwith the comparison instruction CMP, if the addition produced a carry otherwise flag is clear.
 - (ii) For subtraction alongwith the comparison instruction CMN, if subtraction produced a borrow, then the carry flag is cleared otherwise it is set.
 - (iii) For operations not involved with addition/subtraction but include a shift operation, the carry flag is set to the last bit shifted out of the value by the shifter.
- For operations that do not include a shift, carry flag remains unaltered.

10. Bit 30 (Zero flag : Z)

If the result of comparison is zero i.e. equal numbers are present then the zero flag is set to 1. Z flag is otherwise cleared.

11. Bit 31 (Negative Flag : N)

- The negative flag is set if the result is negative and regarded as two's complement of a signed integer.
- Otherwise the flag is cleared if the result is positive or zero.



- Other methods to modify the N, Z, C and V flags are listed below :
 - (i) Execution of flag setting variants arithmetic and logical instructions such that the destination register is R15.
 - The variants also copy the SPSR to CPSR. They are mainly used while returning from exceptions.
 - (ii) Execution of MRC instruction such that the destination register is R15.
 - (iii) Execution of MSR instruction, as part of its function of writing a new value to CPSR or SPSR.
 - (iv) Execution of some variants of LDM instruction.

12. Reserved bits

- These bits are reserved for further expansion in future.
- The user should write his program in such manner that the bits are unmodified. If the user fails to do this it may lead to side-effects on future versions of the architecture.

11.6 Barrel Shifter

University Questions

- Q. What are the various functions of the barrel shifter of the ARM processor architecture ? (4 Marks)
- Q. What is the function of Barrel shifter in ARM data flow model. **SPPU - May 15, Dec. 15, 2 Marks**

- The ARM processor does not support actual shift instructions. But instead it supports a **Barrel shifter**.
- The Barrel shifter is responsible for carrying the shifts as a part of other instructions.
- The barrel shifter can be used to perform left shift operation by required amount i.e. it multiplies by the powers of 2.

e.g.: LSL #4 (multiply by 16).



Fig. 11.6.1 : Logical shift left operation

- The barrel shifter can also be used to perform logical and arithmetic right.
- In logical shift right the barrel shifter shifts the bits to the right by specified amount i.e. it divides by power of 2.

e.g. LSR #3 (divide by 8).

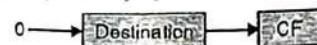


Fig. 11.6.2 : Logical shift right

- In arithmetic shift right the barrel shifter shifts the bits to the right by specified amount and preserves the sign bit, for 2's complement operations.

e.g. ASR #5 (divide by 32).

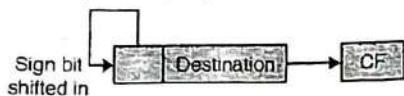


Fig. 11.6.3 : Arithmetic shift right

- The barrel shifter can also be used for performing rotations.
- In Rotate right (ROR) the barrel shifter performs operation similar to arithmetic shift right but wraps the bits around as they leave the LSB and appear as MSB.
- The last bit rotated is also used as carry out alongwith LSB.

e.g. ROR #2.



Fig. 11.6.4 : Rotate right

- For Rotate Right through carry or Rotate Right Extended (RRX) the barrel shifter uses the CPSR C flag as 33rd bit.
- In this operation the bits are rotated through the carry. The carry propagates to LSB and MSB to carry.

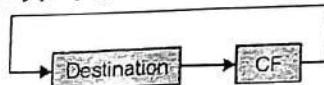


Fig. 11.6.5 : Rotate right through carry

- Using the barrel shifter alongwith ALU a wide range of addresses can be calculated.
- The contents of register R_m are preprocessed in the barrel shifter before applying them as input to the ALU.
- The register R_m can be a register, shifted register or immediate value.



11.7 Data Types

- The data types supported by the ARM processors are
Byte : 8 bits.
Half word : 16 bits. (halfwords should be aligned to two-byte boundaries)
Word : 32 bit (words must be aligned to four-byte boundaries).
- The ARM instructions are one word instructions and Thumb Instructions are half word instructions.
- All data operations are done on word quantities.
- All the three data types are supported in the ARM architecture version 4 and above.
- Prior to ARM architecture version 4 only bytes and words were supported.

11.8 Nomenclature

University Questions

- Q. What is meant by TDMI with respect to ARM7 core ? **SPPU - May 16, 4 Marks**
- Q. Explain the term ARM7TDMI. **SPPU - May 17, Feb. 17, 5 Marks**
- Q. What is TDMI, Compare the Thumb and ARM Instruction set features of ARM. **SPPU - Mar. 18, 5 Marks**
- Q. What is TDMI ? Draw and explain data flow model of ARM7 in detail. **SPPU - May 18, 5 Marks**
- Q. What is TDMI? **SPPU - Mar. 19, 5 Marks**

- The ARM processor uses a nomenclature that describes the various features implemented in the processor.
- The word "ARM" is followed by some letters and numbers that indicate features of the processor, these letters and numbers are as given below :

ARM {x}{y}{z}{T}{D}{M}{I}{E}{J}{F}{S}.

Such that

x : ARM family.

y : Memory management/protection unit.

z : Cache.

T : Thumb instruction set.

D : Debug via JTAG interface.

M : Long multiply instructions.

I : Embedded ICE macrocell.

E : Enhanced instructions (DSP applications).

J : Jazelle.

F : Vector Floating point unit.

S : Synthesizable version.

e.g.: ARM7TDMI

- It implies that processor core is based on ARM7 family.
- It supports the Thumb instruction set. It supports debug via JTAG interface.
- It supports the long multiply instructions.
- It supports hardware breakpoints and watchpoints via the Embedded ICE macrocell.
- The nomenclature does not consist any information regarding architecture revision.
- A group of processors of processor family have the same hardware characteristics e.g. : ARM920T, ARM940T share ARM 9 family characteristics.
- The Thumb instruction set is supported for all versions after version 4. The variant T is automatic for ARM v6 and above.
- The core supports debug via JTAG interface. JTAG interface is described by IEEE 1149.1 standard Test Access Port and boundary scan architecture.
- It is used to send and receive debug information between processor core and the test equipment. The variant D is automatic for ARM v5 and above.
- The ARM core supports long multiply instructions for version 3 and above.
- Embedded ICE macrocell is the debug hardware that is built in processor for setting breakpoints and watchpoints.
- Enhanced DSP instructions are included in version 5 and above of ARM architecture.
- Variant E is included for all versions of ARM v6 and above.
- The ARM core supports Jazelle Java acceleration architecture.
- It also supports vector floating point architecture.
- Synthesizable version implies that the processor core is supplied as source code that can be compiled into form easily used by EDA tools.



11.9 Processor Modes

University Questions

- Q. List and explain the various operating modes of the ARM processor. (8 Marks)
- Q. State and explain different operating modes of ARM7.

SPPU - Dec. 16, Feb. 17, Mar. 19, 6 Marks

- The ARM processor supports **seven** operating modes. The processor mode decides availability of the registers to the programmer and also access rights to the CPSR.

The processor modes are classified as :

1. privileged mode
2. non-privileged mode.

- Privileged mode supports full read and write access to the CPSR.
- The privileged modes supported by processor core are :

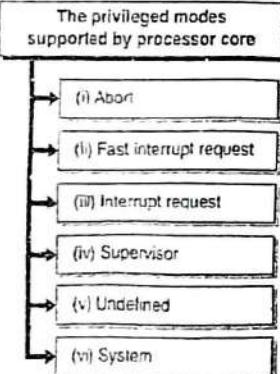


Fig. 11.9.1 : Privileged Modes

- Non-privileged mode allows only read access to the control field in the CPSR. However it allows read-write access to the condition flags. Non-privileged mode supported by processor core is :

1. User

1. User mode
2. Fast Interrupt Mode (FIQ mode)
3. Interrupt Mode (IRQ)
4. Supervisor mode (SVC)
5. Abort mode
6. Undefined mode (Udef)
7. System mode

1. User mode

- It is a non-privileged mode. Most of the tasks are executed in the user mode.
- Memory access is restricted in user mode. User cannot read directly from hardware device.

2. Fast Interrupt Mode (FIQ mode)

- This mode is entered whenever a high priority of interrupt is raised.
- FIQ mode is used to handle the peripherals that issue fast interrupts.
- This mode is privileged mode. The devices causing FIQs are floppy disc handling data, serial port etc.

3. Interrupt Mode (IRQ)

- This mode is a privileged mode.
- It is entered when a low priority interrupt is raised. e.g. keyboard, hard disk, floppy disc etc.

Difference between FIQ and IRQ

- An IRQ may be interrupted by an FIQ but an FIQ cannot be interrupted by IRQ.
- With FIQ we have to do processing fast. For achieving this, the processor has shadow registers.
- FIQs cannot call software interrupts (SWIs).
- If it becomes essential for an FIQ routine to re-enable interrupts, it will take longer time than it would take by an IRQ.
- FIQ should disable interrupts.

4. Supervisor mode (SVC)

- The ARM processor enters this mode on rest. This mode can also be entered if a software interrupt (SWI) instruction is executed.
- Supervisor mode has additional privileges that allow greater control of the system.
- An operating system kernel operates in this mode.
- In order to read from a I/O Module, programmer has to enter the supervisor mode.

5. Abort mode

The abort mode is entered whenever an attempt access memory fails .

6. Undefined mode (Undef)

- This mode is used to handle undefined instructions.

7. System mode

- This mode is a privileged mode. It is a special version of the user mode.
- It allows full read-write access to the CPSR.
- It is present in ARM architecture version 4 and above.
- Summary of ARM processor modes is given in Table 11.9.1.

Table 11.9.1 : ARM processor modes

Mode	Denotation	Privileged	Enter this mode in case of Exception	Mode select Bits in CPSR				
				B4	B3	B2	B1	B0
User mode	usr	No	No	1	0	0	0	0
Fast Interrupt request	fiq	Yes	Yes	1	0	0	0	1
Interrupt request	irq	Yes	Yes	1	0	0	1	0
Supervisor mode	svc	Yes	Yes	1	0	0	1	1
Abort	abt	Yes	Yes	1	0	1	1	1
Undefined	undef	Yes	Yes	1	1	0	1	1
System	sys	Yes	No	1	1	1	1	1

11.10 Pipeline

Q. Explain the three stage pipelining implemented in ARM processor. (8 Marks)

- The process of fetching the next instruction while the current instruction is being executed is called as "pipelining".
- Pipelining is supported by the processor to increase the speed of program execution. It also increases throughput.
- Several operations take place simultaneously, rather than serially in pipelining system.
- The pipeline has three stages fetch, decode and execute as shown in Fig. 11.10.1.

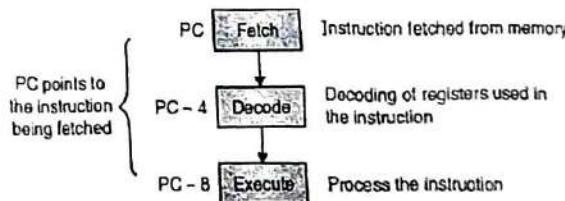
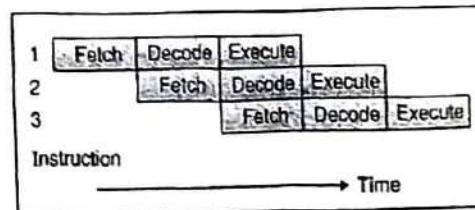


Fig. 11.10.1 : Three stage pipeline

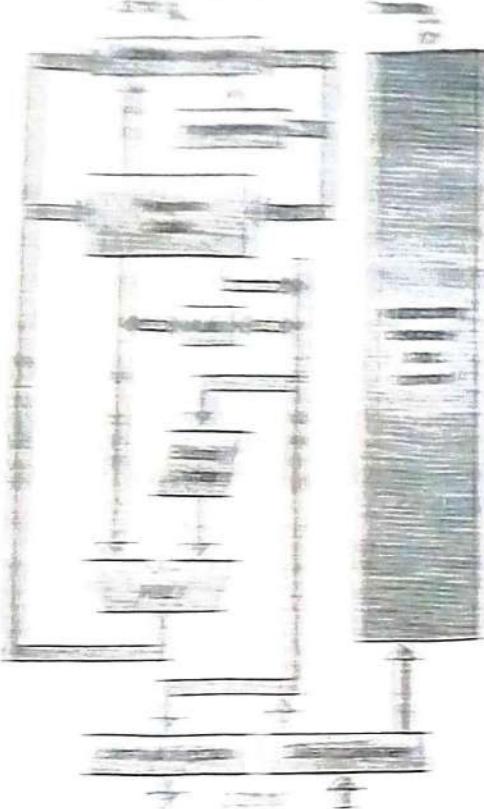
The three stages used in the pipeline are :

- (i) Fetch : In this stage the ARM processor fetches the instruction from the memory.
 - (ii) Decode : In this stage recognizes the instruction that is to be executed.
 - (iii) Execute : In this stage the processor processes the instruction and writes the result back to desired register.
- If these three stages of execution are overlapped, we will achieve higher speed of execution. Such pipeline exists in version 7 of ARM processor.
 - Once the pipeline is filled, each instruction requires one cycle to complete execution.
 - Fig. 11.10.2 shows a three stage pipelined instruction execution.



(a) Single cycle instruction execution for a 3-stage pipeline in ARM

Fig. 11.10.2 contd...



Journal of Clinical Endocrinology and Metabolism

- The first result is to increase processing capabilities and also to store additional values of computation. This is done with help of registers.
 - Thereafter the instruction counter is incremented by one step.
 - The amount of work time which stages can be reduced by increasing the number of stages in the pipeline.
 - To improve the performance of processor bus can be operated at higher operating frequency.
 - As more number of stages are required to fill the pipeline, the system latency also increases.
 - The data dependency between the stages can also be reduced as the stages of pipeline increase.
 - So the instructions need to be interleaved while writing to decrease data dependency.

- Registers:
 - 1. Program Counter: It contains the address of the next instruction to be executed.
 - 2. Register Transfer: The register transfer unit performs the transfer of data between memory and CPU.
 - 3. Buffer Register: We have also seen the buffer register that is used to do various operations asynchronization of the bus.
 - 4. The ALU: This unit performs the various arithmetic and logical operations.
 - 5. Address register and incrementer: The register stores the address and the incrementer increments the value so as to point to the next instruction.
 - 6. Data register: It is used as a buffer to store the data when written to the memory or read from the memory.
 - 7. Instruction Decoder: As the name says, it decodes the instruction and issues the control signals accordingly. Hence, the Instruction decoder is associated with the control logic that issues the control signals.

- The pipeline registers are used to store the results of intermediate calculations.
- The pipeline registers also store the values of the previous stages for the next stage to use them as inputs.
- The pipeline registers also store the address of memory to be accessed by the ALU, multiplier and divider units to fetch required data from the memory bank.

11.10.1 ARM9 Pipeline:

- Fig. 11.10.4: Five stage pipeline to be forwarded internally from one stage to next and of ARM9 TDMI processor**
- The speed of ARM processor is 1.6 GHz processor clock frequency.
 - The five stages of pipeline are: Fetch, decode and execute unit, memory unit and ALU, floating point unit and register file register.
 - The execution time taken by 1.6 GHz is equivalent to 1.25 clock cycles per instruction.
 - The clock frequency of ARM9 is greater than the 1.6 GHz processor.



Fig. 11.10.4: Five stage pipeline to be forwarded internally from one stage to next and of ARM9 TDMI processor

- In case of 5-stage pipelining ARM processor also uses data forwarding technology.
- This enables the data to be forwarded internally from one stage to next and hence reduce the chances of pipeline stalling.
- Fig. 11.10.5 shows the organization of the 5-stage pipeline organization in ARM9TDMI.

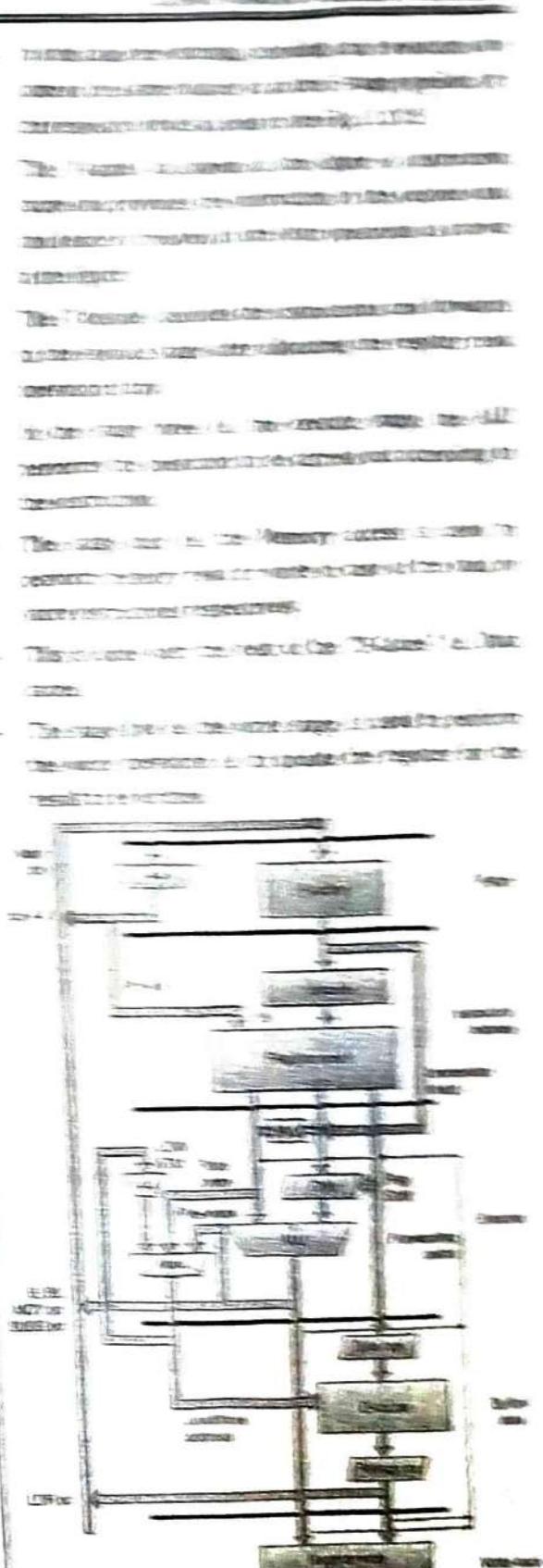


Fig. 11.10.5: Organization of 5-stage pipeline in ARM

11.10.2 ARM10 Pipeline

- The pipeline of ARM10 processor has six stages fetch, issue, decode and read register, execute shift and ALU, memory access and multiply, write register as shown in Fig. 11.10.6.
- The instruction throughput increases by around 34% more than ARM7 processor. The system latency also increases.

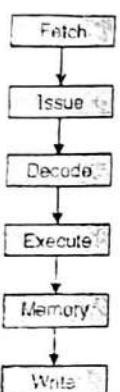


Fig. 11.10.6 : Six stage pipeline of ARM10 processor

11.11 Exceptions/Interrupts

Q. Explain the different exceptions in ARM processor. (8 Marks)

- The ARM processor supports five types of exception and privileged processing mode for each type.
- The types of exception are**
- (i) Normal interrupt.
 - (ii) Fast interrupt.
 - (iii) Memory aborts that are used to implement memory protection or virtual memory.
 - (iv) Attempt to execute an undefined instruction.
- Software interrupt instructions that are used to make a call to the operating system.
 - Whenever an exception occurs, the standard registers are replaced with registers that are specific to the exception mode.
 - The exception modes have replacement banked registers for R13 and R15.
 - For fast interrupt processing there are more registers as the processing has to be done fast.

- R14 contains the return address for exception processing. Once the exception is processed R14 returns to the address that caused the exception.
- The system mode uses the user mode registers in order to run the tasks that require privilege access to memory and/or coprocessors without limitations.
- R13 provides each exception handler an individual separate stack pointer.

Exception process

- When an exception occurs, the ARM processor stops the execution of current instruction and continues execution from one of fixed addresses referred as exception vector in memory. For each exception there is a separate vector location.
- Processor copies the CPSR register to the SPSR. Then appropriate CPSR bits are set. If the core implements ARM architecture 4T and is in the Thumb state, then the ARM state is entered. It stores the return address in R14.
- Maps the appropriate banked registers.
- To return from the exception handler the CPSR is restored from the SPSR and PC from R14.
- Exceptions are generated by internal and external sources that cause the processor to handle events like externally generated interrupt or attempt to execute an undefined instruction.
- At a time more than one exception can take place. Table 11.11.1 consists the types of exception and the processor mode used to process that exception.

Table 11.11.1 : Exception types

Exception type	Mode	Normal address	High vector address
Reset	Supervisor	0x00000000	0xFFFF0000
Undefined instructions	Undefined	0x00000004	0xFFFF0004
Software interrupt (SWI)	Supervisor	0x00000008	0xFFFF0008
Prefetch Abort (instruction fetch memory abort)	Abort	0x0000000C	0xFFFF000C

Exception type	Mode	Normal address	High vector address
Data Abort (data access memory abort)	Abort	0x00000010	0xFFFF0010
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C

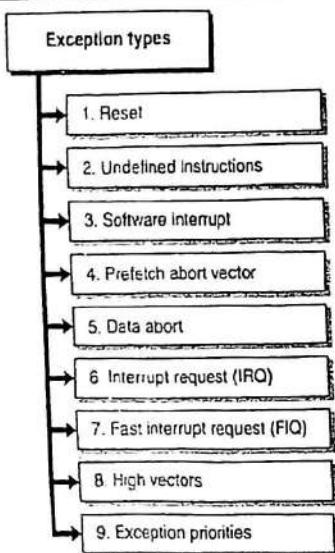


Fig. 11.11.1 : Exception types

1. Reset

When reset pin is activated, the current execution is stopped by the processor. After reset the processor begins execution at an address 0x00000000 or 0xFFFF0000 is the supervisor mode with interrupts disabled.

2. Undefined instructions

- Whenever an attempt is made to execute an undefined instruction, an undefined instruction exception occurs.
- If a coprocessor instruction is executed, the processor waits for any external coprocessor to acknowledge that it can execute the instruction.
- Incase no external processor acknowledges, an undefined instruction exception occurs.

3. Software interrupt

- Whenever a software interrupt instruction is executed a software interrupt exception is caused.
- The processor enters the supervisor mode inorder to request a particular supervisor function.

4. Prefetch abort vector

- This exception is generated if the processor tries to execute an invalid instruction. If the instruction is not executed then no prefetch abort exception occurs.
- This exception can also be generated as a result of executing a breakpoint instruction in ARM architecture version 5 and above.

5. Data abort

- A memory abort is signalled by the memory system. Activating an abort in response to data access (i.e. load or store) marks the data as invalid.

6. Interrupt request (IRQ)

- This exception is generated by asserting the IRQ input on the processor. When the I bit in CPSR is set, this interrupt is disabled.
- It has a lower priority than the FIQ.
- If the I bit in CPSR is clear then the processor checks for an IRQ at instruction boundaries.

7. Fast interrupt request (FIQ)

- If the FIQ input is asserted, an FIQ exception is generated.
- FIQ supports data transfer or channel process and has sufficient private registers to remove the need for register saving in such applications minimizing the overhead of context switching.
- Fast interrupts are disabled if the F bit in CPSR is set. If the bit is clear then the processor checks for an FIQ at instruction boundaries.

8. High vectors

- Some of the ARM implementations allow the exception vector locations to be shifted from their normal address 0x00000000-0x0000001C to address 0xFFFF0000-0xFFFF001C. These locations are called as high vectors.

9. Exception priorities

- Table 11.11.2 shows the exception priorities



Table 11.11.2

Priority	Exception
Highest 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Lowest 6	Undefined instruction, SWI

- The priority of a data abort exception is higher than FIQ. The priority of SWI and undefined instruction is same. But, both of them cannot take place at the same time.

11.12 Exam Pack (Review and University Questions)

- Q. What are the main features of ARM architecture?
 (Refer Section 11.1.4) (4 Marks)
- Q. Draw and explain the ARM family core architecture.
 (Refer Section 11.2) (8 Marks)
- Q. Compare ARM7, ARM9 and ARM11 series processors stating Features. (Refer Section 11.3.2(D)) (5 Marks)
- Q. Draw and explain the ARM programmers model.
 (Refer Section 11.4) (8 Marks)
- Q. Explain programmers model of ARM7 with the help of neat diagram. (Refer Section 11.4) (5 Marks)
- Q. Draw and explain the structure of the program status register. (Refer Section 11.5) (8 Marks)
- Q. What are the various functions of the barrel shifter of the ARM processor architecture ?
 (Refer Section 11.6) (4 Marks)
- Q. List and explain the various operating modes of the ARM processor. (Refer Section 11.9) (8 Marks)
- Q. Explain the three stage pipelining implemented in ARM processor. (Refer Section 11.10) (8 Marks)
- Q. Explain the five stage pipelining implemented in ARM 9 processor. (Refer Section 11.10.1) (8 Marks)
- Q. Explain the different exceptions in ARM processor.
 (Refer Section 11.11) (8 Marks)
- Q. List features of ARM7 processor. How it is different then pure RISC processor ?
 (Refer Section 11.3.2) (Feb. 17, 6 Marks)
- Q. Compare various versions of ARM with respect to features, advantages, power dissipation.
 (Refer Section 11.3.2(D))
 (Oct. 11, Feb. 12, Feb. 13, May 15, Dec. 15, Feb. 16, May 17, 5 Marks)
- Q. Compare the ARM7, ARM9 and ARM11 processors.
 (Refer Section 11.3.2(D)) (Mar. 19, 5 Marks)
- Q. Explain the term : Banked register in ARM
 (Refer Section 11.4) (Oct. 12, 4 Marks)
- Q. Write significance of special registers. r_{13}, r_{14}, r_{15} in ARM7. (Refer Section 11.4)
 (May 15, Dec. 15, 3 Marks)
- Q. Explain programmer's model of ARM processor
 (Refer Section 11.4) (May 16, 6 Marks)
- Q. Draw and explain the complete ARM register set with concept of changing mode on exception.
 (Refer Section 11.4) (May 18, 5 Marks)
- Q. Explain term : CPSR register and processor modes.
 (Refer Section 11.5) (Oct. 12, 4 Marks)
- Q. Draw and explain CPSR register structure of LPC 2148. (Refer Section 11.5)
 (May 15, Dec. 15, May 17, 4 Marks)

Q. Explain CPSR register in detail. What is the need of SPSR register ? (Refer Section 11.5)	(Feb. 16, 6 Marks)	Q. Explain the term ARM7TDMI (Refer Section 11.8) (May 17, Feb. 17, 5 Marks)
Q. Draw and explain the CPSR of ARM in detail. (Refer Section 11.5)	(Mar. 18, 5 Marks)	Q. What is TDMI, Compare the Thumb and ARM Instruction set features of ARM. (Refer Section 11.8) (Mar. 18, 5 Marks)
Q. Draw the structure of CPSR and explain the functions of each bit. (Refer Section 11.5)	(Dec. 18, 6 Marks)	Q. What is TDMI ? Draw and explain data flow model of ARM7 in detail. (Refer Section 11.8) (May 18, 5 Marks)
Q. Describe CPSR and SPSR of ARM 7 in detail. (Refer Section 11.5)	(Mar. 19, 5 Marks)	Q. What is TDMI, (Refer Section 11.8) (Mar. 19, 5 Marks)
Q. What is the function of Barrel shifter in ARM data flow model (Refer Section 11.6) (May 15, Dec. 15, 2 Marks)		Q. State and explain different operating modes of ARM7. (Refer Section 11.9) (Dec. 16, Feb. 17, Mar. 19, 6 Marks)
Q. What is meant by TDMI with respect to ARM7 core ? (Refer Section 11.8)	(May 16, 4 Marks)	

□□□

Note

Model Question Paper (End Sem.)

Processor Architecture (214451)

Semester IV – Information Technology (Savitribai Phule Pune University)

Time : $2\frac{1}{2}$ Hours

Maximum Marks : 70

Instructions to the candidates :

1. Answer Q.1 or Q.2, Q.3 or Q.4, Q.5 or Q.6, Q.7 or Q.8,
2. Neat diagrams must be drawn wherever necessary.
3. Figure to the right indicate full marks.
4. Make suitable assumptions, if necessary.

Q. 1 (a) Explain IVT and ISR of PIC18F458. (Refer Section 6.2) (6 Marks)

(b) With the help of a neat interfacing diagram explain how an electromagnetic relay can be controlled through PIC 18 microcontroller. (Refer Section 7.6) (7 Marks)

(c) Explain CCP1CON Control Register. (Refer Section 8.3.1) (5 Marks)

OR

Q. 2 (a) Explain the generation of waveform using compare mode in CCP module. (Refer Section 8.4) (6 Marks)

(b) Draw and explain the ARM family core architecture. (Refer Section 11.2) (4 Marks)

(c) Design a PIC test board test board using LED, keypad, buzzer and relays connected to ports with control using keys and write a C program for testing with S1 pressed LED ON and S2 pressed relay and buzzer ON.
(Refer Ex 7.6.1) (7 Marks)

Q. 3 (a) List the steps involved in programming PIC microcontroller in Compare mode.

(Refer Section 8.4.1) (6 Marks)

(b) List the steps involved in programming PIC microcontroller in PWM mode. (Refer Section 8.6) (6 Marks)

(c) Write a program to generate a positive ramp (sawtooth) waveform for PIC18 microcontroller.

(Refer Ex 10.1.3) (5 Marks)

OR

Q. 4 (a) Write a program to measure the pulse width of signal given on CCP1 pin and give output on Ports B and D.

(Refer Program 8.5.1(A)) (6 Marks)

(b) Using compare mode, write the assembly language program to toggle the LED every 10 pulses. Use Timer 1 as counter. (Refer Program 8.4.2(A)) (5 Marks)



- (c) Write a program for PIC18 microcontroller to transfer a letter 'T' serially and continuously at a baud rate of 9600. Use BRGH = 0. Assume crystal frequency = 10 MHz. (Refer Section 9.8.3) (6 Marks)

- Q. 5 (a) Draw and explain CPSR register structure of LPC 2148. (Refer Section 11.5) (5 Marks)
- (b) List the steps that must be taken in programming PIC 18 microcontroller to transfer character bytes serially. (Refer Section 9.8) (6 Marks)
- (c) Write a PIC18 assembly program to blink a LED. (Refer Ex 7.4.2) (7 Marks)

OR

- Q. 6 (a) Design a frequency counter for counting number of pulses and display same on LCD. (Refer Ex 7.5.4) (6 Marks)
- (b) Explain interfacing of LED with PIC18F458 with diagram and program example. (Refer Section 7.4.1) (5 Marks)
- (c) Write a program for 2.5 kHz and 75 % duty cycle PWM generation with N = 4. (Program 8.6.4) (6 Marks)

- Q. 7 (a) Draw a neat diagram of interfacing of 16x2 LCD with PIC18F458 microcontroller in 8 bit mode. Assume suitable port pins for interfacing. (Refer Section 7.5.2) (7 Marks)
- (b) Explain programming external hardware interrupts in PIC18 microcontroller. (Refer Section 6.6) (4 Marks)
- (c) Write assembly language program by using timer 0 interrupt to generate square wave on pin RB1. (Refer Program 6.5.2) (6 Marks)

OR

- Q. 8 (a) Explain steps for programming the CCP module for PWM generation. (Refer Section 8.6.3) (7 Marks)
- (b) Interface a 4 x 4 matrix keyboard to PIC18F458. Display key pressed on hyper terminal. (Refer Ex 7.2.1) (7 Marks)
- (c) Compare RET and RETFIE. (Refer Section 6.5.1) (3 Marks)

□□□

Needs
Sacrifice

