Embedded Notes

By : Asma Hakouz

#بأفكارنا_نُبدع

Full material

Getting Started with Embedded Systems

Chapter 1
Sections 1-6

Dr. Iyad Jafar

Outline

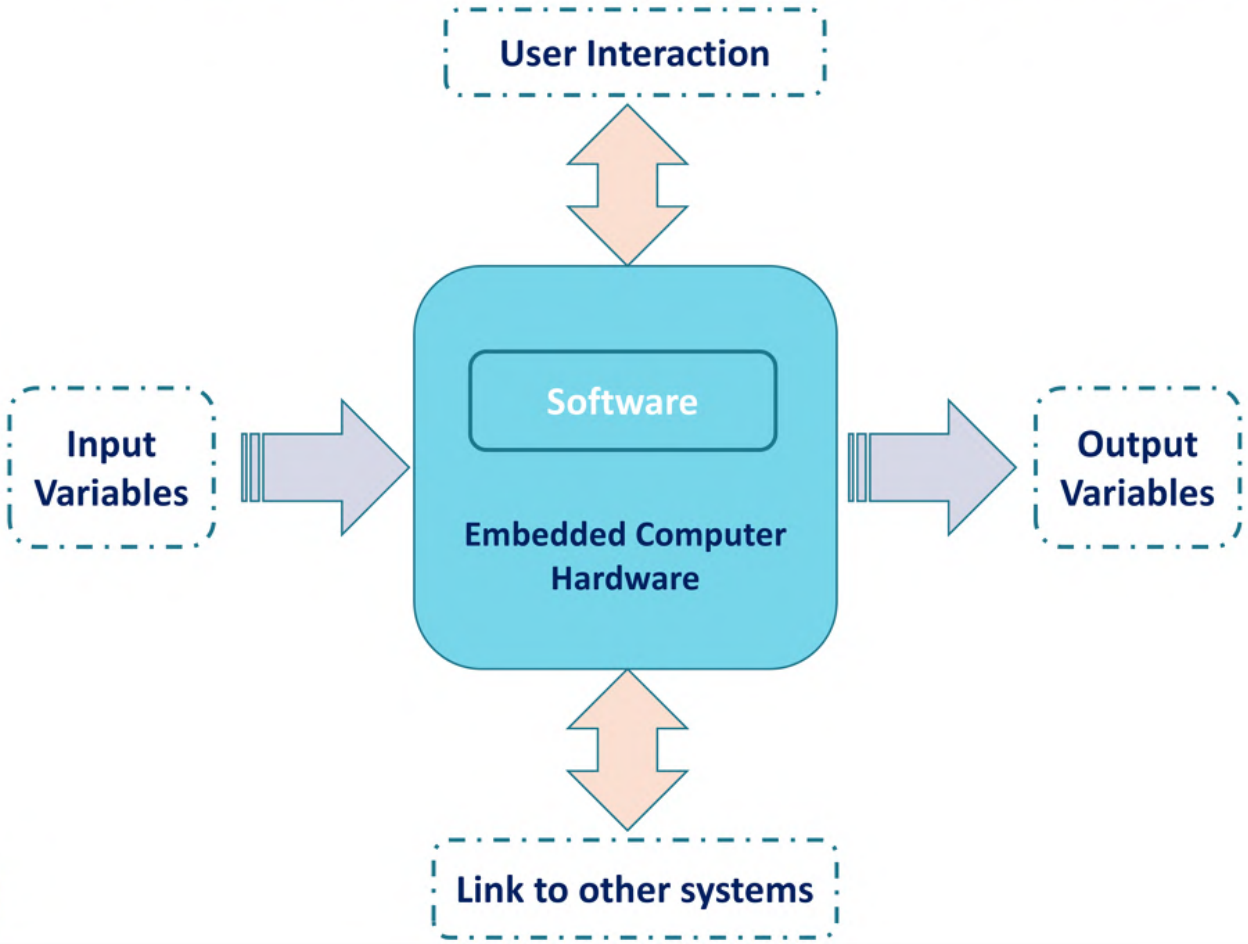
- What is an Embedded System?
- The Essence of Embedded Systems
- Embedded systems examples
- Some Computer Essentials
- Microprocessors vs. Microcontrollers
- The PIC Microcontroller
- The PIC 12 Series as an Example

What is an Embedded System?

- **An *embedded system* is a computer system that is**
  - designed to perform one or a few dedicated functions often with real-time computing constraints
  - *embedded as part of a complete* device often including hardware and mechanical parts.

- By contrast, a general-purpose computer, such as a personal computer, is designed to be flexible and to meet a wide range of end-user needs.

The Essence of Embedded Systems



The Essence of Embedded Systems

- **Characteristics**
  - *Microcontroller or DSP based*
  - *Software driven*
  - *Reliable*
  - *Real-time control system*
  - *Autonomous / human interactive / network interactive*
  - *Operate on diverse input variables and in diverse environments*

Examples

- Automotive
- Avionics/Aerospace/Defence
- Industrial Automation
- Telecommunications
- Consumer Electronics & Intelligent Homes & Retail (Thin Clients/POS)
- Scientific & Medical Equipment
- Computer peripherals

Examples









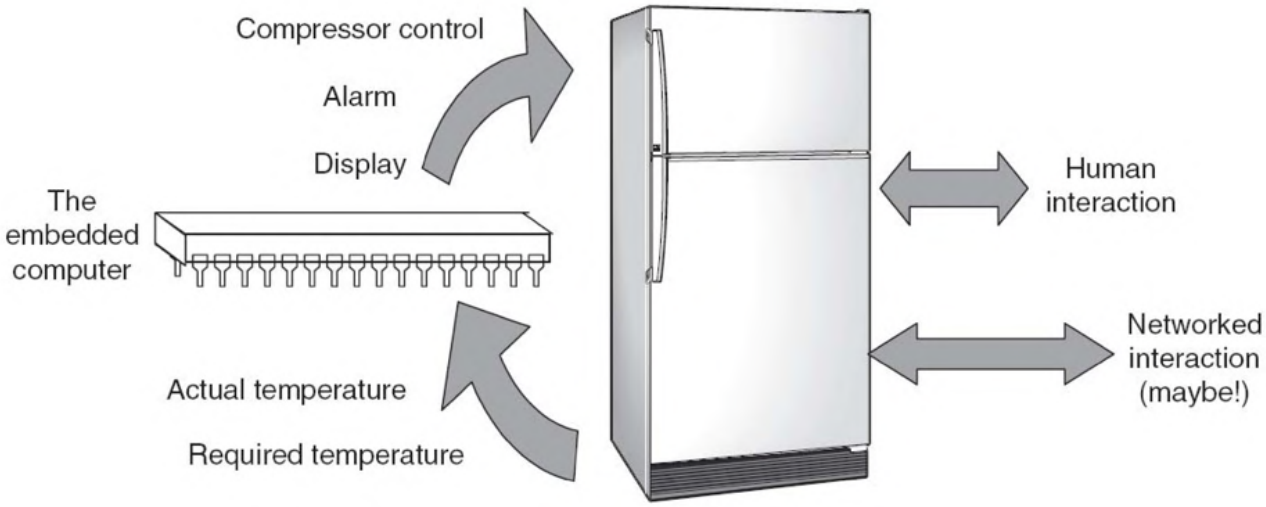






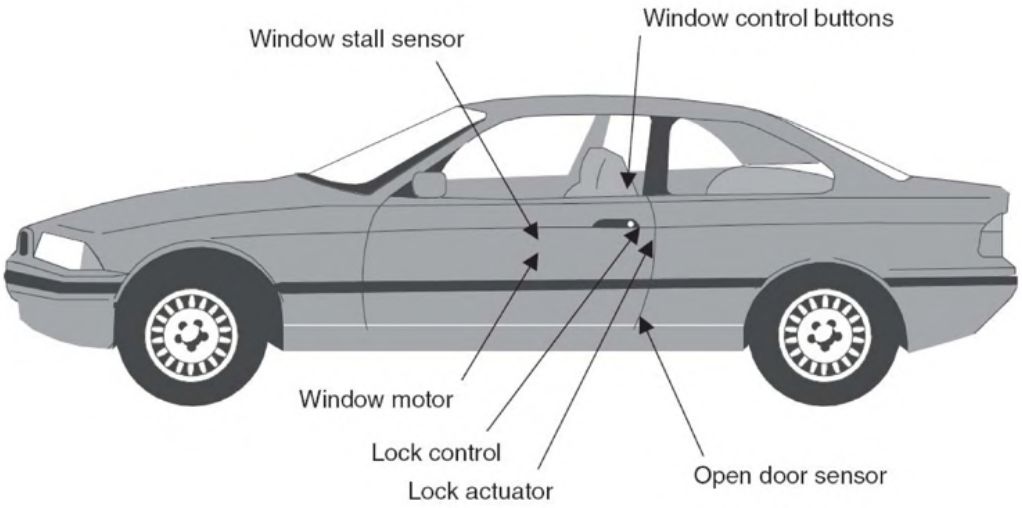


Examples



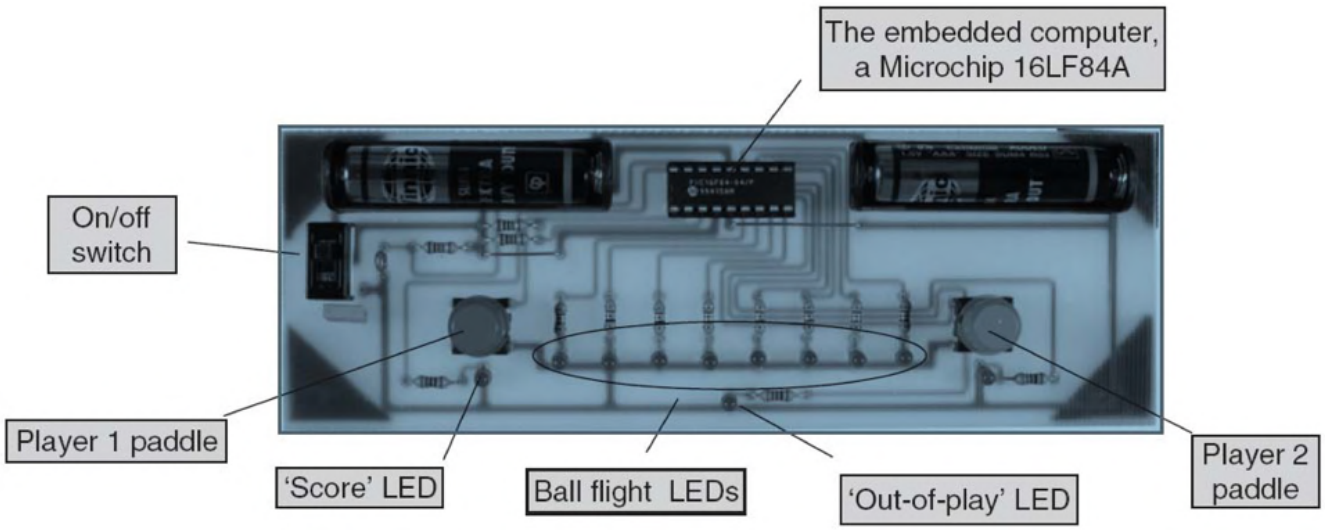
- The refrigerator is required to maintain low temperature by reading the current value and controlling the compressor accordingly

Examples



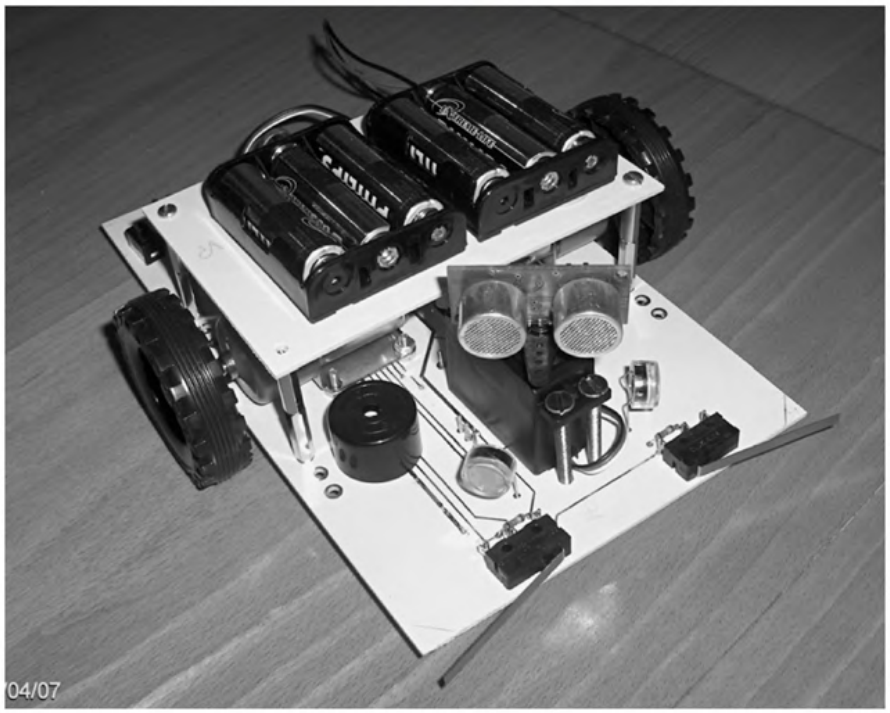
- Different sensors in the car door produce signals that are of great importance when integrated with the rest of the car functionality

Examples



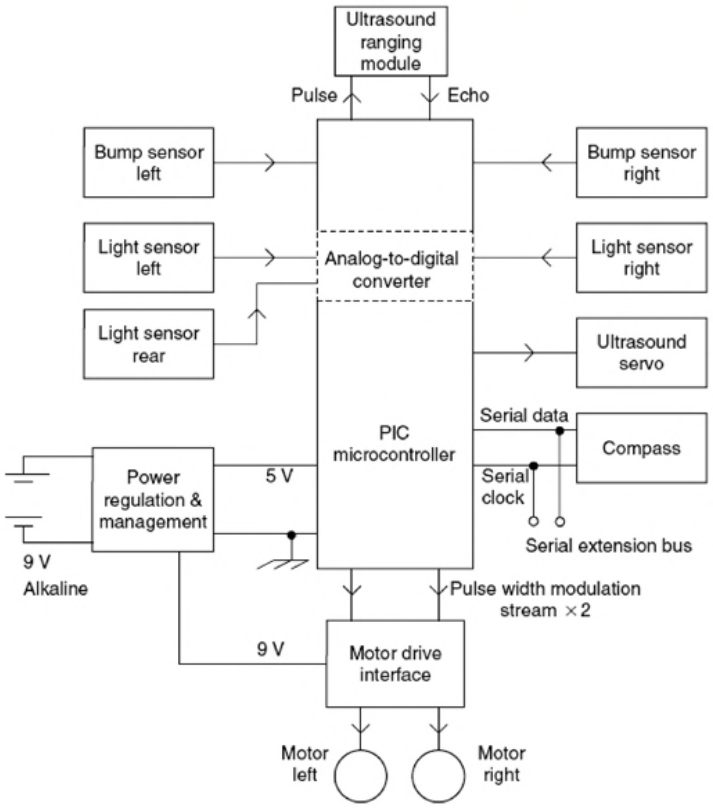
- The Electronic 'ping-pong'

Examples



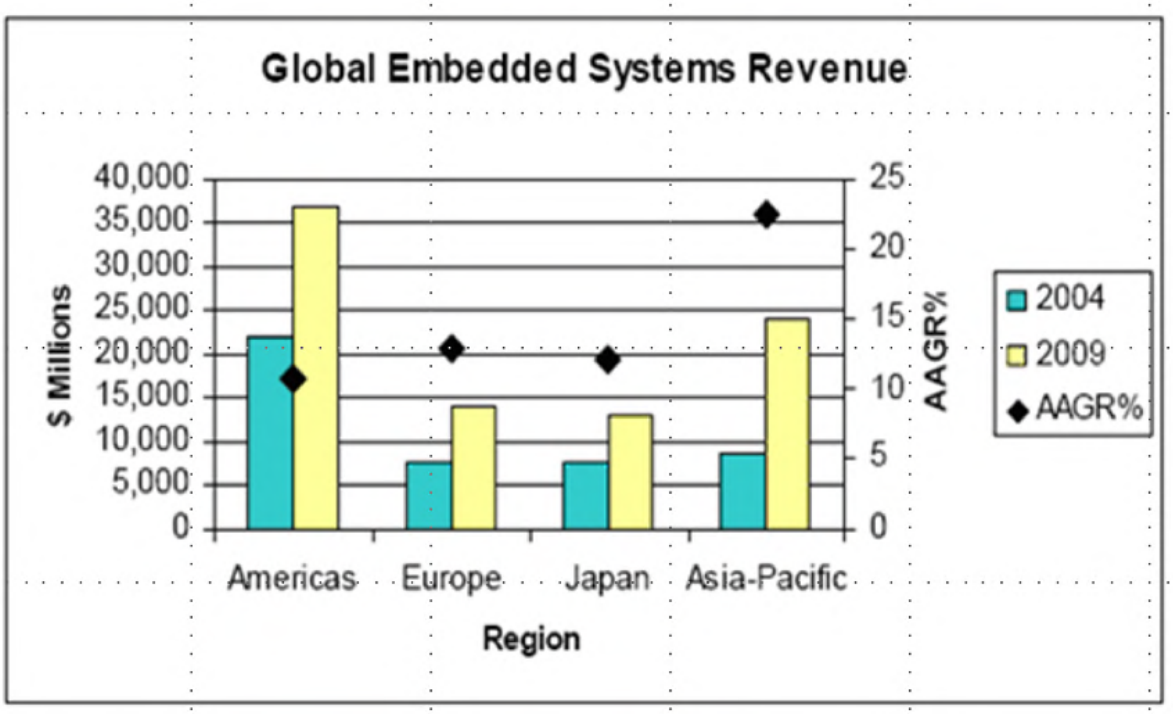
- The Derbot Autonomous Guided Vehicle
- More sensors and powerful microcontroller

Examples



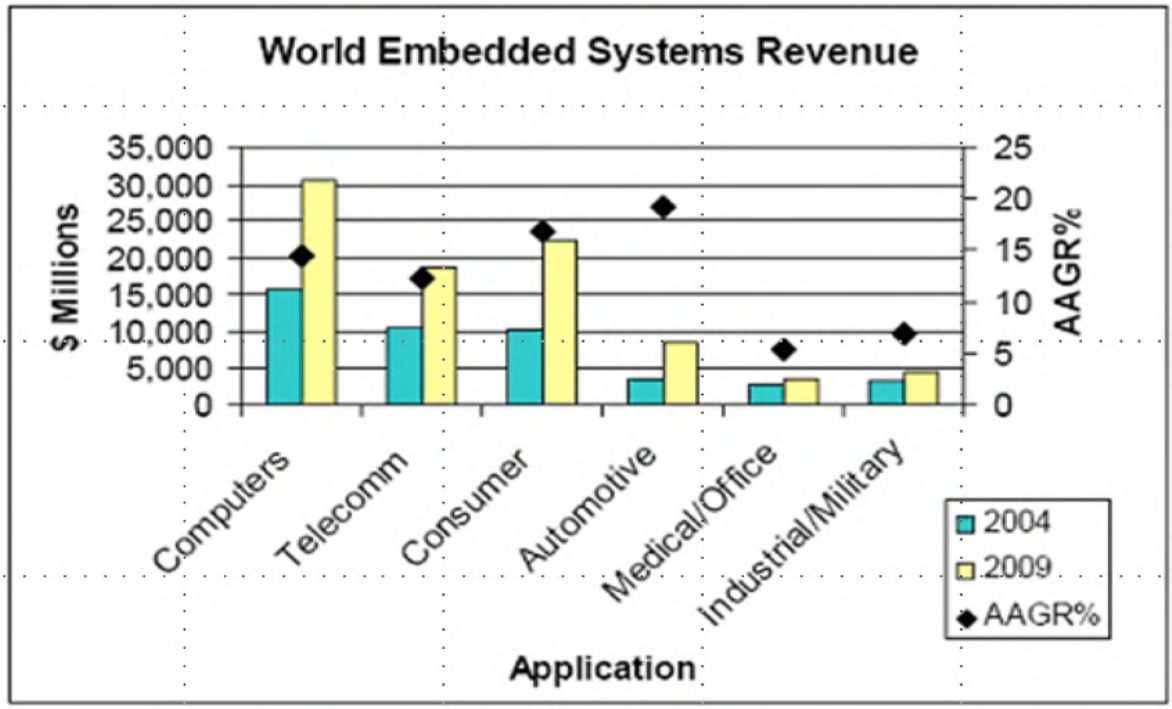
- The Derbot Autonomous Guided Vehicle

Embedded Systems Market



Source: BCC research http://www.bccresearch.com

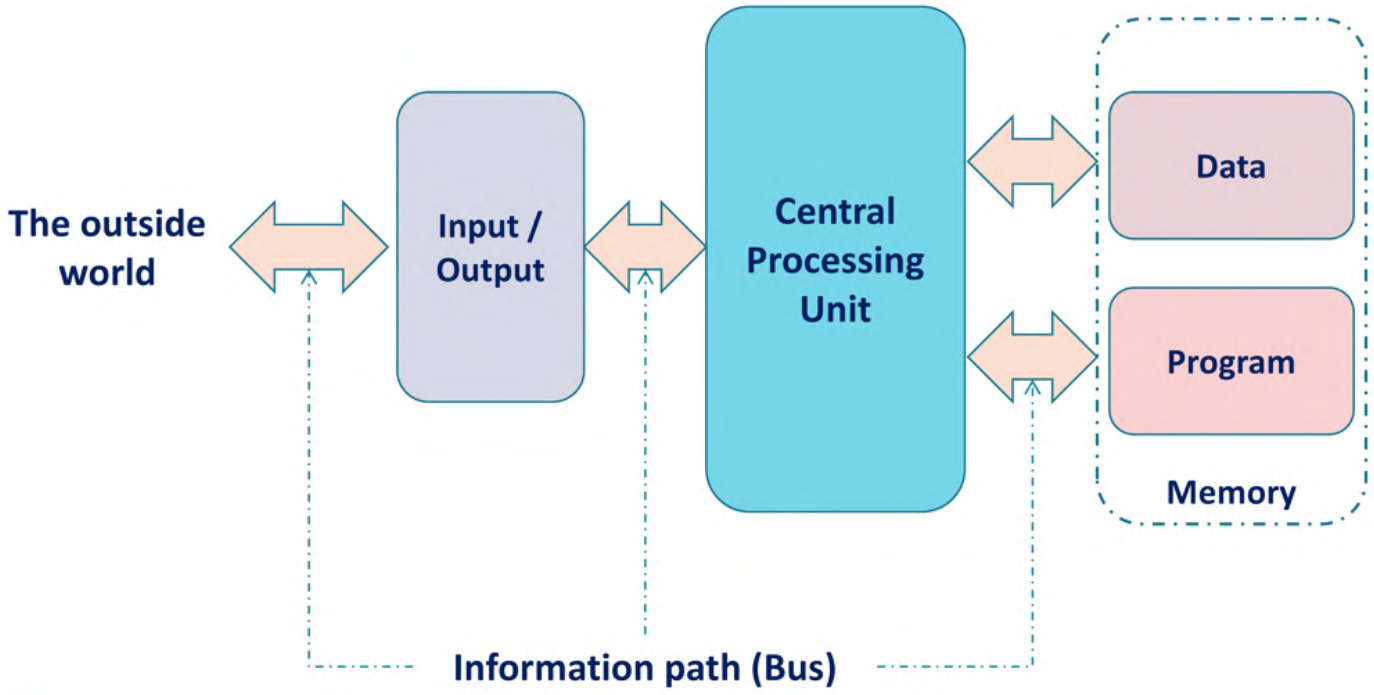
Embedded Systems Market



Source: BCC research http://www.bccresearch.com

Some Computer Essentials

- **Elements of a Computer**

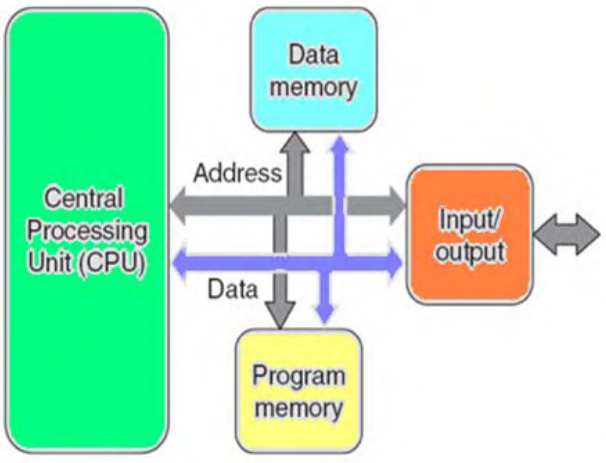


Some Computer Essentials

Memory Organization

The Von Neumann Architecture

- One address bus and one data bus
- I/O may be also connected to these busses
- Simple and logical architecture, however
  - Same memory width for instruction and data ?!
  - Shared busses ?!

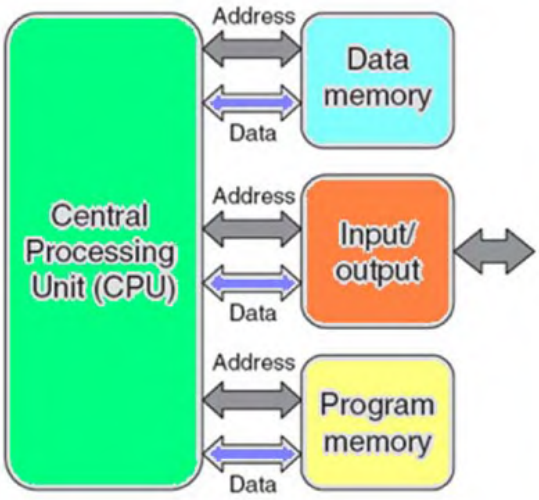


Some Computer Essentials

Memory Organization

The Harvard Architecture

- Separate address and data bus for program memory and data memory
- More flexibility;
  - Different memory width
  - Simultaneous access of data and program memories
- Complex ?!



Some Computer Essentials

Instruction Sets

- Every CPU has a set of instructions that it can recognize and execute
- There are different approaches in designing instructions for the CPU in attempt to speed up program execution
  - **CISC (Complex Instruction Set Computers)**
    - Many instructions and addressing modes
    - Instructions have different levels of complexity (different size and execution time)
    - Relatively slow
    - Shorter programs
  - **RISC (Reduced Instruction Set Computers)**
    - Few instructions and addressing modes
    - Simple instructions of fixed size
    - Relatively fast
    - Longer programs

Some Computer Essentials

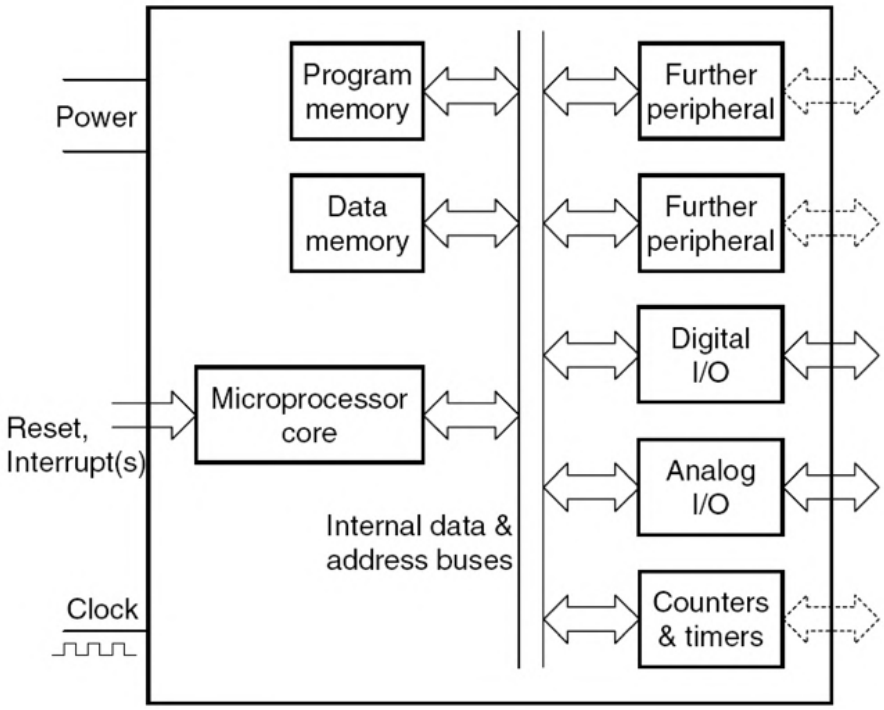
- **Memory Types**
  - **Volatile**
    - Holds its contents as long as power is ON
    - Used as temporary *storage to hold data*
    - Easy to write
    - RAM
  - **Non-volatile**
    - Retains its values on power out
    - More difficult to write in terms of time and power
    - In embedded systems, it is *usually used to store programs*
    - ROM

Microprocessors and Microcontrollers

- First microprocessors in the 1970s
  - The computer CPU on a single chip
  - Initially, memory and I/O interfacing outside the CPU
  - As technology evolved, the microprocessor became more self-contained, powerful, and faster
- A special category of microprocessors emerged
  - Microcontrollers
  - Intended for control purposes
  - *No high computational power, huge memories, or high speed is required*
  - *Has excellent I/O capabilities*
  - Small, low cost, and self contained

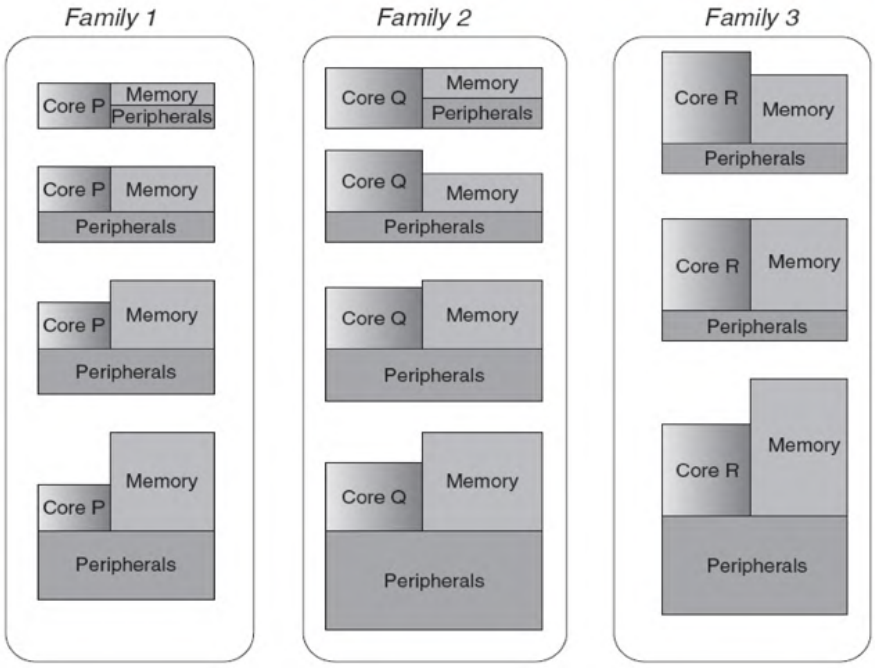
Microprocessors and Microcontrollers

- A generic microcontroller



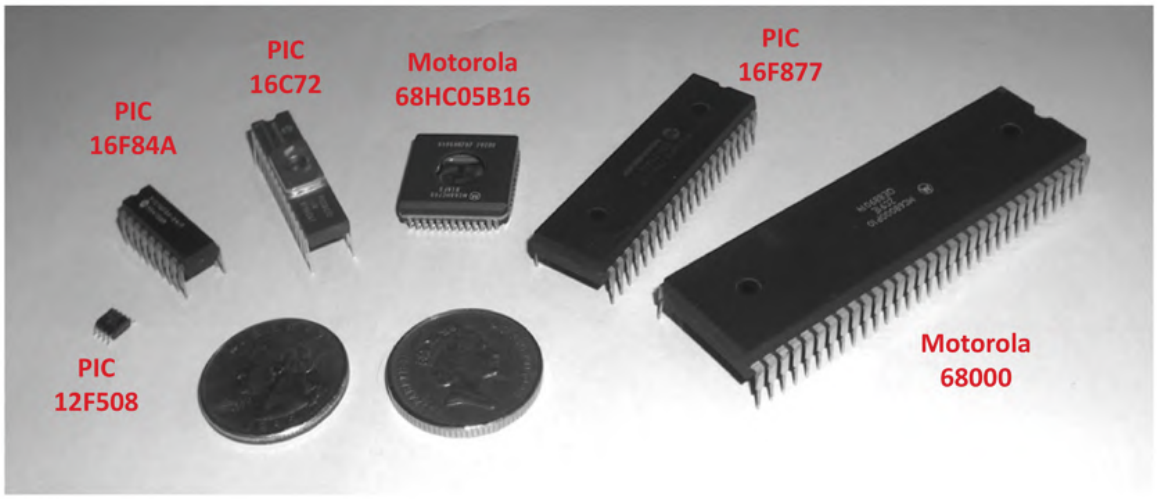
Microprocessors and Microcontrollers

- Microcontroller Families
  - Different families with *each family built around the same core*
  - Family members differ in *memory size* and *peripheral capabilities*



Microprocessors and Microcontrollers

- Microcontroller Packaging
  - Plastic packaging
  - Pins for I/O, clock, communication, and Power.
  - The number of pins usually determines the size of the chip



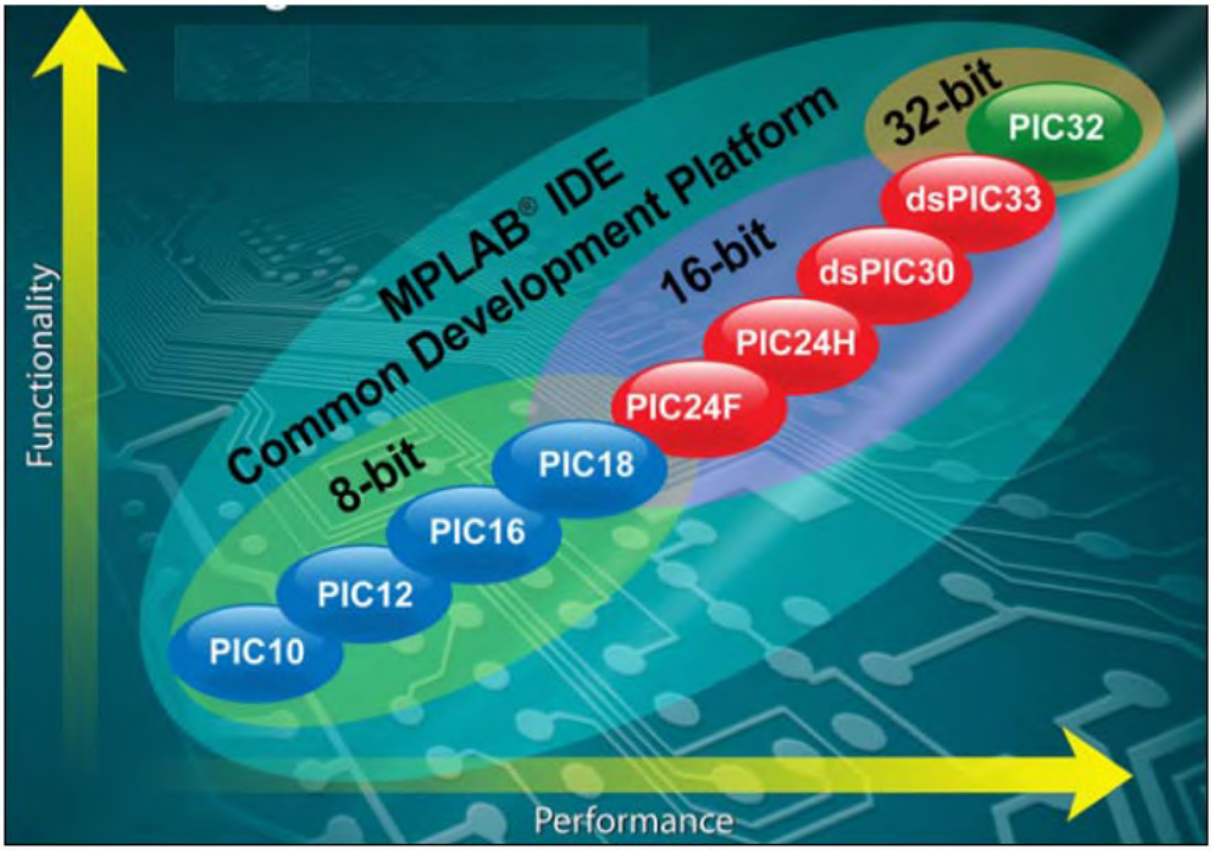
Microchip and the PIC Microcontrollers

- Peripheral Interface Controller (PIC) was originally a design by General Instruments intended for simple control applications

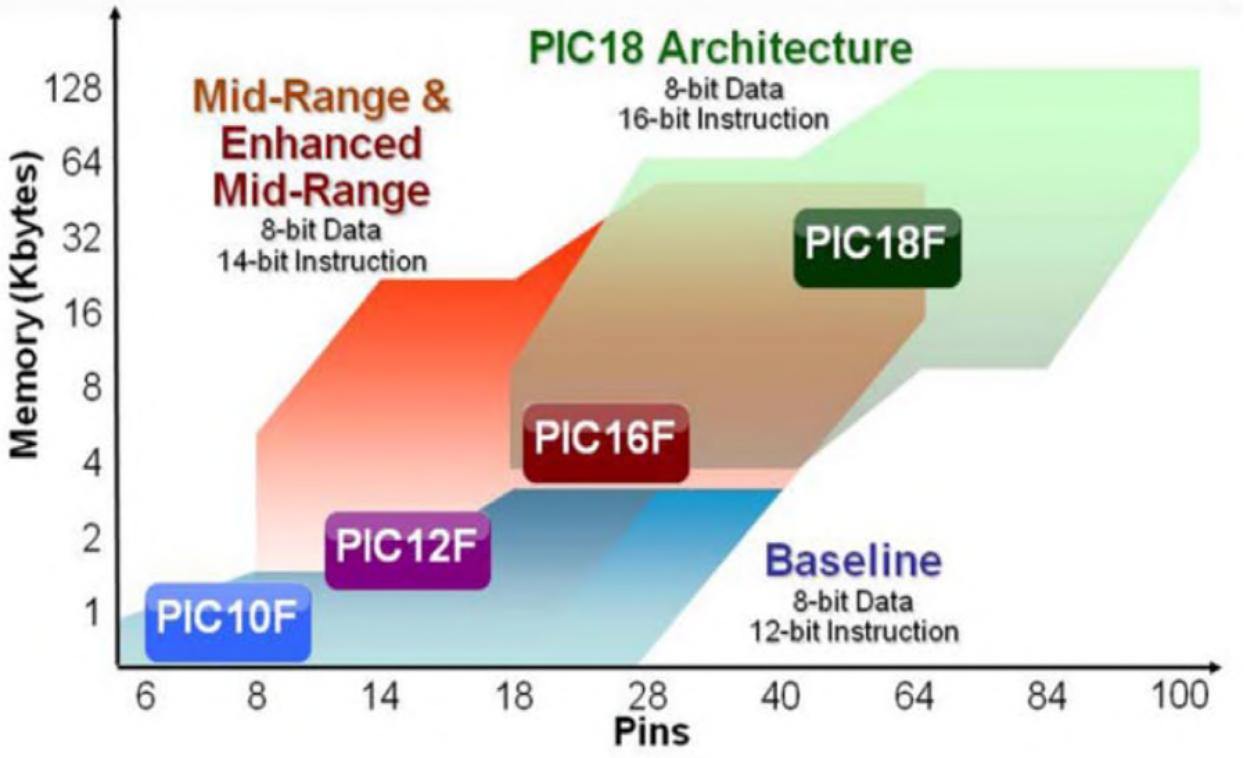
- In the late 1970s, GI introduced PIC® 1650 and 1655
  - Standalone design
  - RISC with 30 instructions
  - Single working register (accumulator)
  - Many attractive features

- PIC was sold to Microchip

Microchip and the PIC Microcontrollers



Microchip and the PIC Microcontrollers



Microchip and the PIC Microcontrollers

- **PIC Families**

PIC Family	Stack Size (words)	Instruction Word Size	No. of Instructions	Interrupt Vectors
12CX/12FX	2	12- or 14-bit	33	None
16C5X/16F5X	2	12-bit	33	None
16CX/16FX	8	14-bit	35	1
17CX	16	16-bit	58	4
18CX/18FX	32	16-bit	75	2

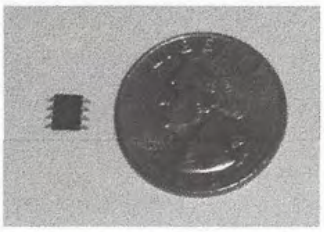
  - Example: the 16**C**84 was the first of its kind built using CMOS technology. It was later reissued as 16**F**84**A** incorporating flash memory and other technological features

Microchip and the PIC Microcontrollers

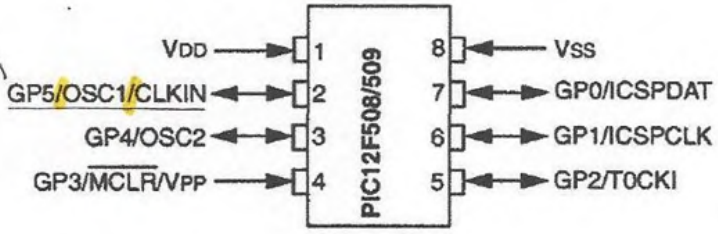
- **PIC Characteristics**
  - Low-cost
  - Self-contained
  - 8-bit
  - Harvard architecture
  - RISC
  - Pipelined
  - Single accumulator (the working or W register)
  - Fixed reset and interrupt vectors

The PIC 12 Series

- PIC 12F508/509 F: Flash memory
- The smallest and simplest PIC



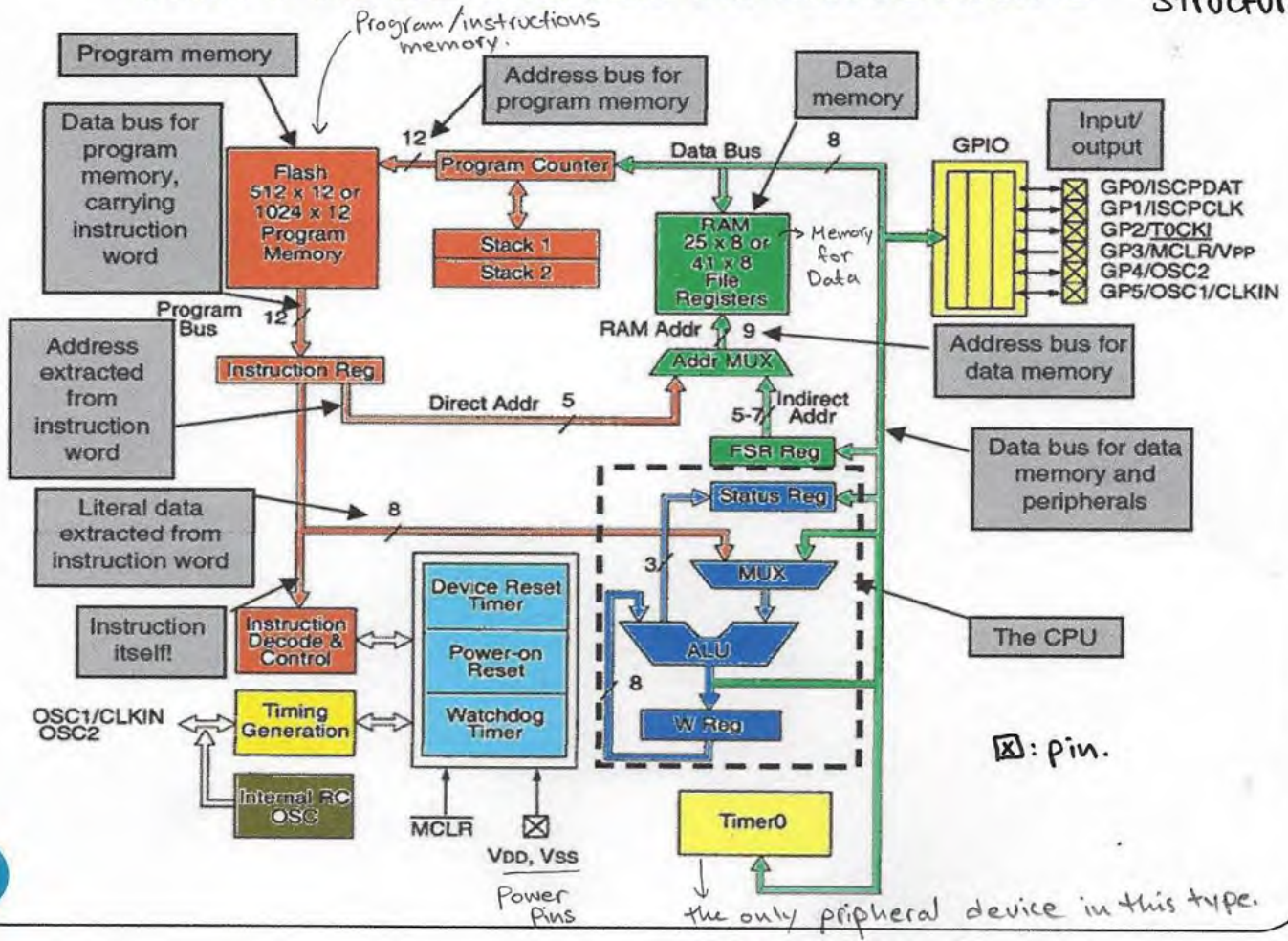
it's a multiplexed & configurable pin; can be used for different purposes but only one at a time.



Key

V_{DD}:	Power supply	V_{SS}:	Ground
V_{PP}:	Programming voltage input	MCLR:	Master clear
OSC1, OSC2:	Oscillator pins	CLKIN:	External clock input
GP0 to GP5:	General-Purpose input/output pins (bidirectional except GP3)		
CSPDAT:	In-Circuit Serial Programming™ data pin.		
CSPCLK:	In-Circuit Serial Programming™ clock pin.		

The PIC 12 Series Architecture (harvard structure).



Slide 30:

* RAM: 25x8 ⇒ 25: # of memory locations (File Registers).
8: # of bits in each location (width)

→ 25x8 is for PIC12F508 , 41x8 is for PIC12F509

* Flash: 512 x 12
↳ 12-bits is the length of 1 instruction.

** Pripheral devices in PIC microcontrollers are memory-mapped, so they are accessed as memory locations. (That's why there's no bus dedicated for them).

** There are 2 address buses & 2 data buses ; 1 for program (Flash) memory & the other for data memory & the pripherals.

** n-bits address bus can address 2^n memory locations.

Summary

- *An embedded system has one or more computers embedded within it that perform control operations*
- *A microcontroller is at the heart of embedded systems. It is basically a microprocessor with extended I/O capabilities*
- *Microchip is one of the popular vendors for a large variety of microcontrollers with different features*

Introducing the PIC 16 Series and the 16F84A

Chapter 2
Sections 1-8

Dr. Iyad Jafar

Outline

- Overview of the PIC 16 Series
- An Architecture Overview of the 16F84A
- The 16F84A Memory Organization
- Memory Addressing
- Some Issues of Timing
- Power-up and Reset
- The 16F84A On-chip Reset Circuit

Overview of the PIC 16 Series

- The PIC 16 series is classified as a mid range microcontroller
- The series has different members all built around the same core and instruction set, but with different memory, I/O features, and package size

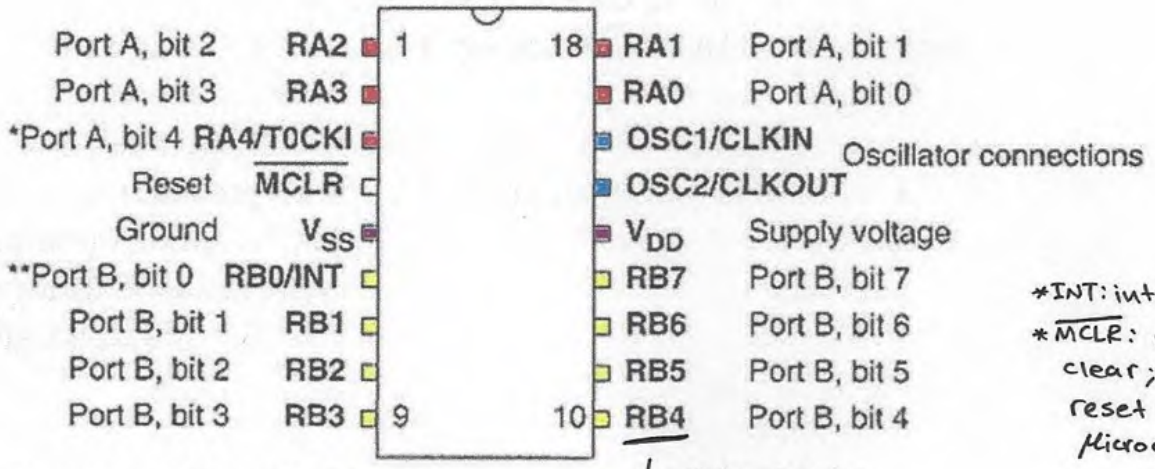
Some members of the PIC 16 Series family

Device number	No. of pins*	Clock speed	Memory (K = Kbytes, i.e. 1024 bytes)	Peripherals/special features
16F84A	18	DC to 20 MHz	1K program memory, 68 bytes RAM, 64 bytes EEPROM	1 8-bit timer 1 5-bit parallel port 1 8-bit parallel port
16LF84A	As above	As above	As above	As above, with extended supply voltage range
16F84A-04	As above	DC to 4 MHz	As above	As above
16F873A	28	DC to 20 MHz	4K program memory 192 bytes RAM, 128 bytes EEPROM	3 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 5 10-bit ADC channels, 2 analog comparators
16F874A	40	DC to 20 MHz	4K program memory 192 bytes RAM, 128 bytes EEPROM	5 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 8 10-bit ADC channels, 2 analog comparators
16F876A	28	DC to 20 MHz	8K program memory 368 bytes RAM, 256 bytes EEPROM	3 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 5 10-bit ADC channels, 2 analog comparators
16F877A	40	DC to 20 MHz	8K program memory 368 bytes RAM, 256 bytes EEPROM	5 parallel ports, 3 counter/timers, 2 capture/compare/PWM modules, 2 serial communication modules, 8 10-bit ADC channels, 2 analog comparators

*For DIP package only.

The general pin-out for PIC16F84A:

An Architecture Overview of the 16F84A



*also counter/timer clock input

**also external interrupt input

*INT: interrupt.

*$\overline{MCLR}$: master clear; used to reset the Microcontroller.

↳ I/O ports

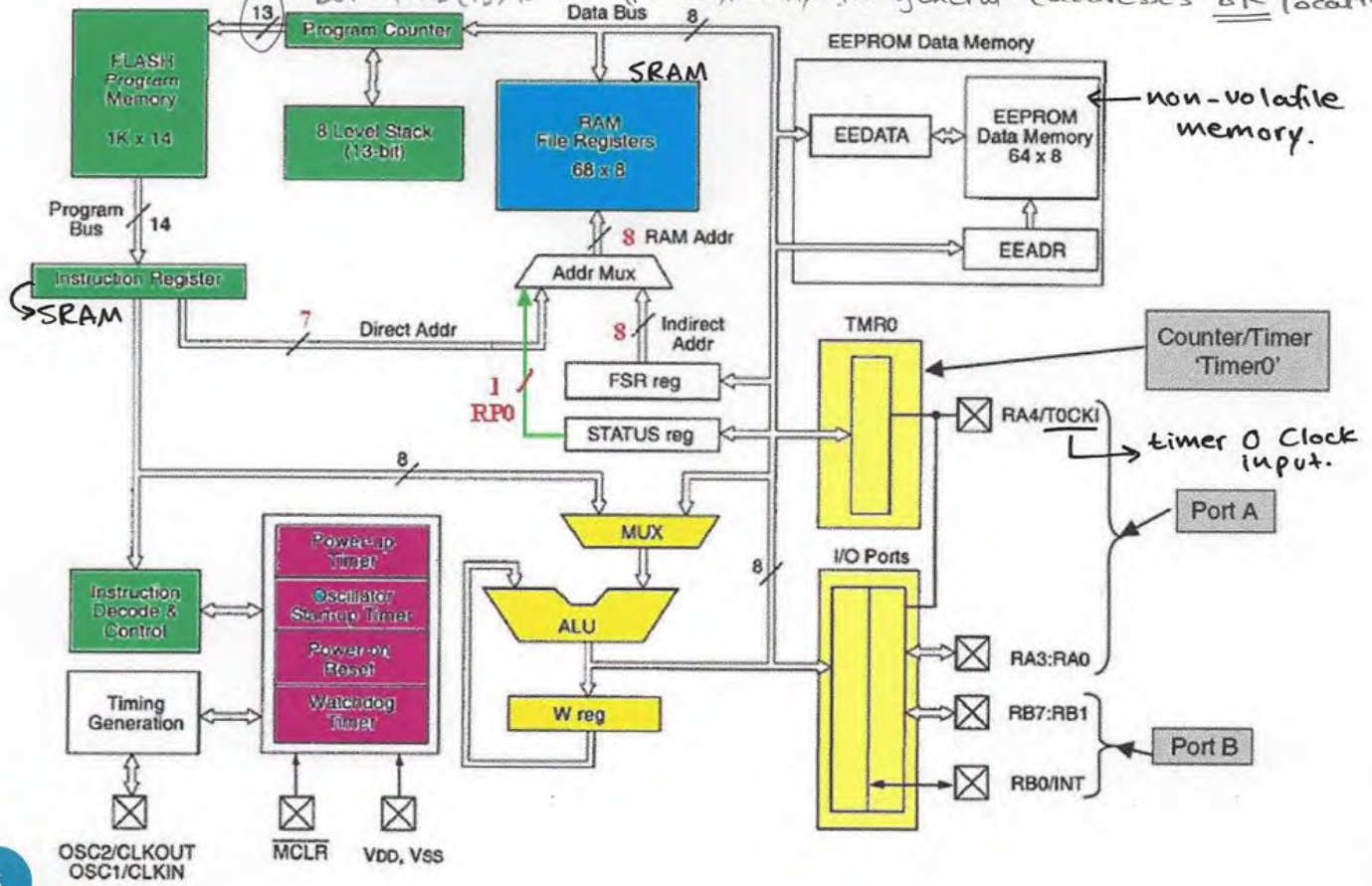
- 18 Pins / DC to 20MHz / 1K program Memory/ 68 Bytes of RAM / 64 Bytes of EEPROM / 1 8-bit Timer / 1 5-bit Parallel Port / 1 8-bit Parallel Port

↳ Electrically Erasable Programmable ROM: used to store data permenantly (even when we turn off the system)

5

An Architecture Overview of the 16F84A

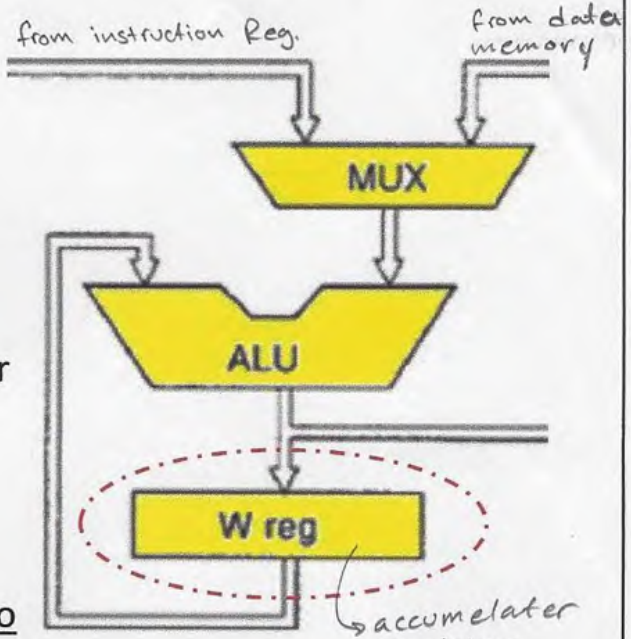
for PIC16F84A we only need 10-bits to address all mem. locations (1K) but this (13) is for (PIC F16) Family in general (addresses 8K locations)



6

The PIC 16F84A ALU and Working Register

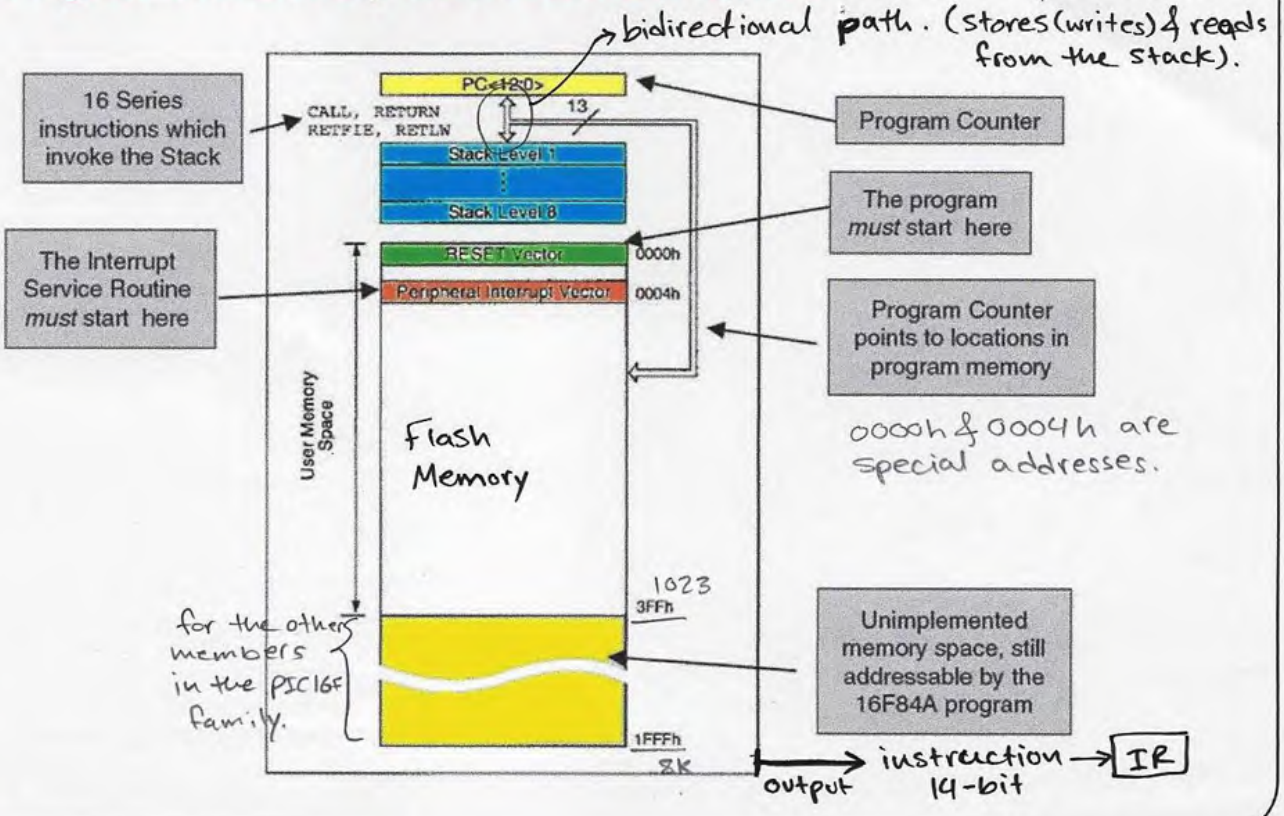
- ***Arithmetic & Logic Unit***
  - 8-bit ALU
  - Supports 35 simple instructions
  - Input operands are
    - The working register
    - Content of some file register or a literal
  - The result is stored in Working register or in a File register
- ***The Working Register***
  - Inside the CPU
  - For many instructions, it can be chosen to hold the result of the last instruction executed by the CPU



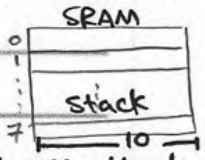
The PIC 16F84A Memory Organization

- *Program Memory and Related Units*

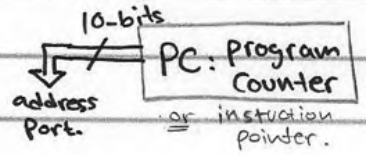
1K x 14 ≡ (1024 x 14) bits



* Program counter: is a register that holds the address of the next instruction.
* the (8 level stack) is used to store addresses not data.
  & it's an SRAM (volatile) memory.

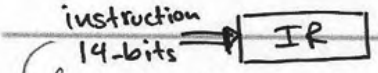


* My program must start at 0000 H ; since this is the address that the PIC will first consult upon power-up.



; at least 10-bits width. (for PIC16F84A)

* IR (Instruction Register): stores the instruction being executed.



from the program memory.

The PIC 16F84A Memory Organization

- **Program Memory**
  - *1K x 14 Bits*
  - *Address range 0000H – 03FFH*
  - *Flash (nonvolatile)*
  - *10000 erase/write cycles*
  - *Location 0000H is reserved for the reset vector*
  - *Location 0004H is reserved for the Interrupt Vector*
- **Program Counter**
  - *Holds the address of the instruction to be executed (next instruction)*
- **Stack**
  - *8 levels (each is 13 bits)*
  - *SRAM (volatile)*
  - *Used to store/load the return address with instruction like CALL, RETURN,RETFIE, and RETLW (interrupts and subroutines)*
- **Instruction Register**
  - *Holds the instruction being executed*

The PIC 16F84A Memory Organization

14-bits

- **The Configuration Word** (used to store some data to configure the Microcontroller.)
  - A special part of the program memory (beyond 8k) ↳ 0x2007
  - Allows the user to configure different features of the microcontroller at the time of program download and is not accessible within the program or while it is running

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	FOSC1	FOSC0
bit13													bit0

bit 13-4 **CP**: Code Protection bit
1 = Code protection disabled
0 = All program memory is code protected → to prevent anyone from accessing your code

bit 3 **$\overline{\text{PWRTE}}$**: Power-up Timer Enable bit
1 = Power-up Timer is disabled
0 = Power-up Timer is enabled

bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0 **FOSC1:FOSC0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

* R/P-u: (after the (-) is the default value)
- R: can be read.
- P: you can write on it only while programming.
- u: its value at start-up is user-defined

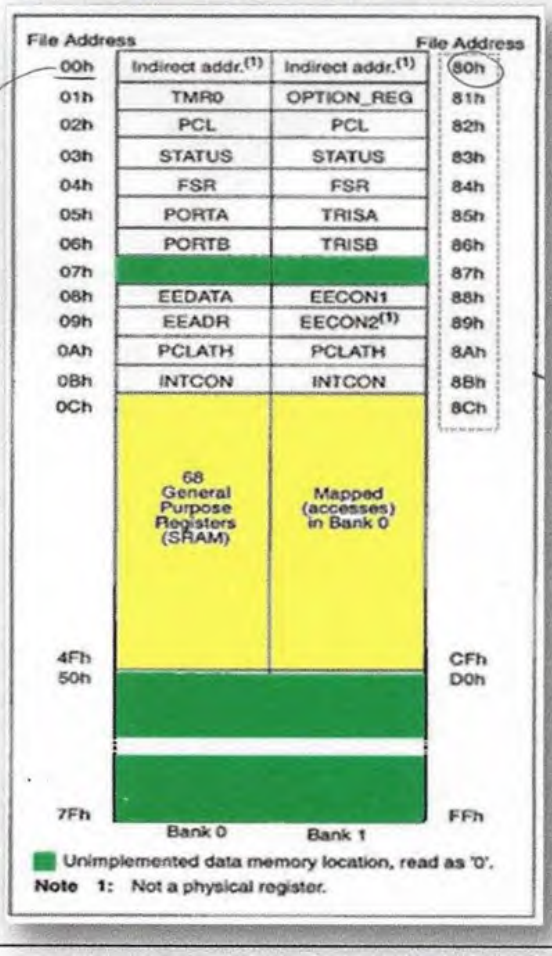
The PIC 16F84A Memory Organization

Data Memory and Special Function Registers (SFRs)

- SRAM (volatile)
- Banked addressing

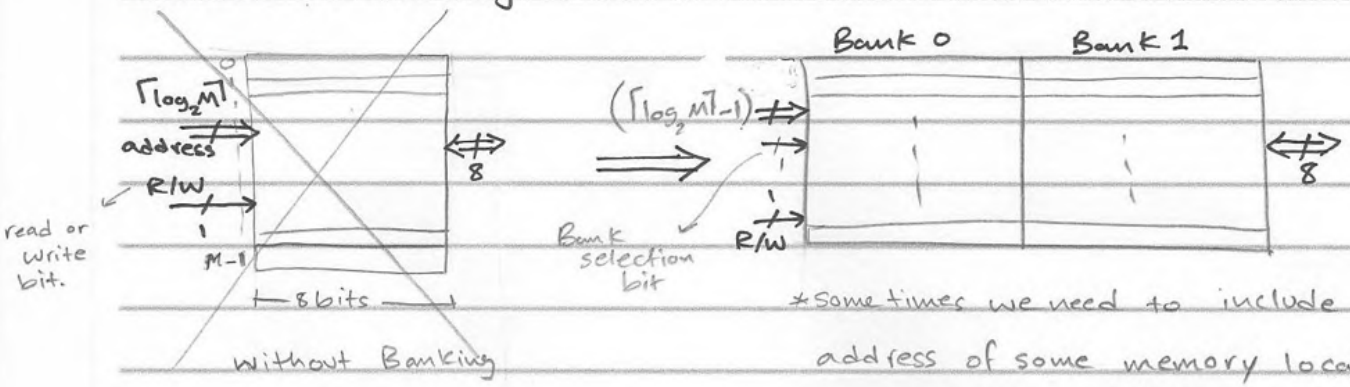
00h → 00000000 = Bank selection
80h → 1000 0000 = Bank1

- **Special Function Registers SFRs**
  - Locations 01H-0BH in bank 0 and 81H-8BH in bank 1
  - Used to communicate with I/O and control the microcontroller operation
  - Some of them hold I/O data
- **General Purpose Registers**
  - Addresses 0CH – 4FH (68 Bytes)
  - Used for storing general data



Slide 11:

Banked addressing:



* Sometimes we need to include an address of some memory location inside the instructio which is 14 bits only, so in order to reduce * of bits used to specify the addres, Banked addressing was used.

* Where we need $(\lceil \log_2 M \rceil - 1)$ bits in the instruction to specify the row. & Bank selection bit(s) specified by the user. (not part of the instruction.

* Bank selection bit (s) is the most significant bit of the address.

The PIC 16F84A Memory Organization

Special Function Registers (SFRs)

Address	Bank 0	Bank 1	Address
00h	INDF	←	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL ←	82h
03h	STATUS	←	83h
04h	FSR	←	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	Unimplemented	←	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2	89h
0Ah	PCLATH	←	8Ah
0Bh	INTCON	←	8Bh
0Ch - 4Fh	GPR	←	8Ch - CFh

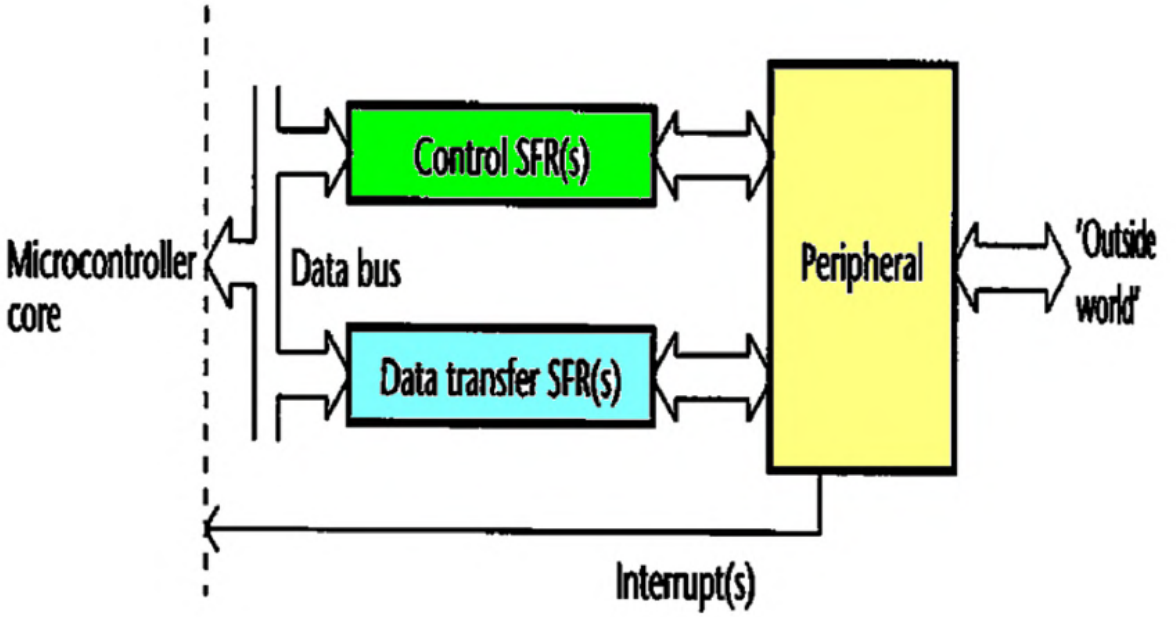
frequently used Reg.

accessing both 02h & 82h is the same since both are PCL (when you access 82h you'll be redirected to 02h). thus, we don't need a bank selection bit here.

INDF : Data memory contents by indirect addressing
TMR0 : Timer counter
PCL : Low order 8 bits of program counter
STATUS : Flag of calculation result
FSR : Indirect data memory address pointer
PORTA : PORTA DATA I/O
PORTB : PORTB DATA I/O
EEDATA : Dtata for EEPROM
EEADR : Address for EEPROM
PCLATH : Write buffer for upper 5 bits of the program counter
INTCON : Interruption control
OPTIN_REG : Mode set
TRISA : Mode set for PORTA
TRISB : Mode set for PORTB
EECON1 : Control Register for EEPROM
EECON2 : Write protection Register for EEPROM

The PIC 16F84A Memory Organization

- ***Special Function Registers (SFRs) interacting with peripherals***



File Registers

GPR
General Purpose Reg.

SFR
Special Function Reg.

M-C related
(INTCON)
(STATUS)

I/O & peripherals

data

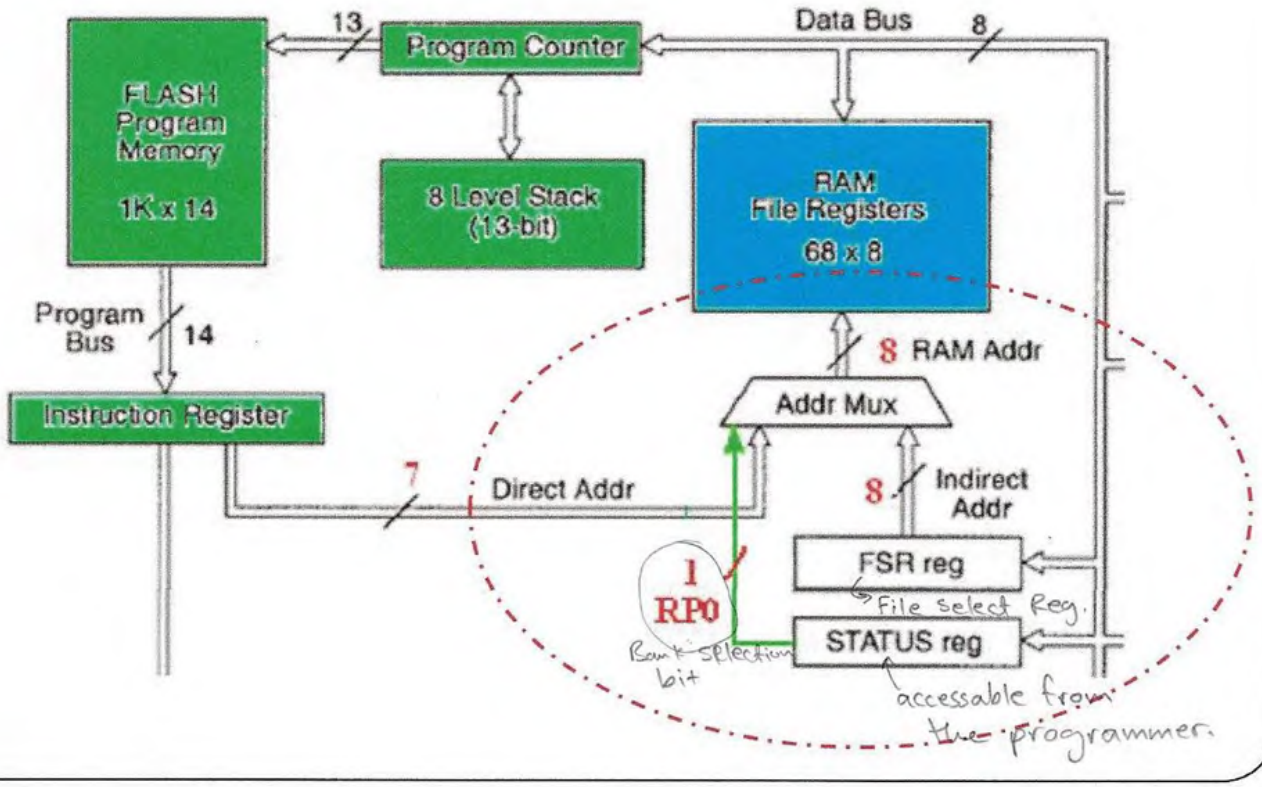
control

The PIC 16F84A Memory Organization

- ***Data Memory Addressing***
  - For PIC 16F84A, the address of any memory location (File Register is 8 bits
    - One bit is used to select the bank
    - Seven bits to select a location in the bank
  - Bank selection is done through using bits 5 and 6 of the STATUS registers (RP0 and RP1)
  - For the 16F84A, only RP0 is needed since we have two banks
  - In general, two forms to address the RAM (File Registers)
    - **Direct addressing –** the 7-bit address is part of the instruction + bank selection bit(s) from status Reg.
    - **Indirect addressing**
      - the 7-bit address is loaded in lower 7 bits of the *File Select Register* (FSR , 04H)
      - Bank selection is done using the most significant bit of FSR and the IRP bit in the STATUS register
        - we use this bit if more than 2 banks are used.

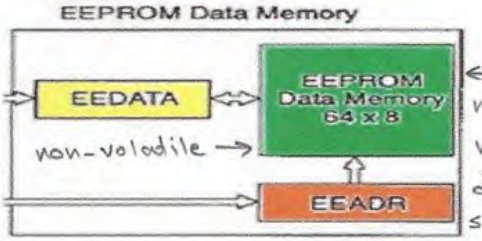
The PIC 16F84A Memory Organization

- ***Data Memory Addressing***



The PIC 16F84A Memory Organization

- ***Data Related***
  - ***EEPROM Data Memory***
    - *64 bytes Non-volatile*
    - *10 000 000 erase/write cycles* (advantage.)
    - *Used to store data that is likely to be needed for long term*
    - *Operation is controlled through EEDATA (08H), EEADR (09H), EECON1 (88H), and EECON2 (89H) SFRs*
    - ***To read a location***
      - *store the address in EEADR and set the RD bit in EECON1*
      - *data is copied to EEDATA register*
    - ***To write to a location***
      - *data and address are placed in EEDATA and EEADR, respectively*
      - *enable writing by setting the WREN bit in EECON1 SFR*
      - *store 55H then AAH in EECON2*
      - *commit writing by enabling the WR bit*
      - *Once the write is done, the EEIF flag is set in EECON1.*



it is a memory mapped device. so, in order to communicate with it we need:-
① address
② Data
③ R/W bit

long procedure for protection to prevent writing by mistake.

The PIC 16F84A Memory Organization

- ***The EECON1 Register (88H)*** (bank 1)

set by Programmer.

U-0	U-0	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
—	—	—	EEIF	WRERR	WREN	WR	RD
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **EEIF:** EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation is not complete or has not been started

bit 3 **WRERR:** EEPROM Error Flag bit
1 = A write operation is prematurely terminated (any MCLR Reset or any WDT Reset during normal operation)
0 = The write operation completed

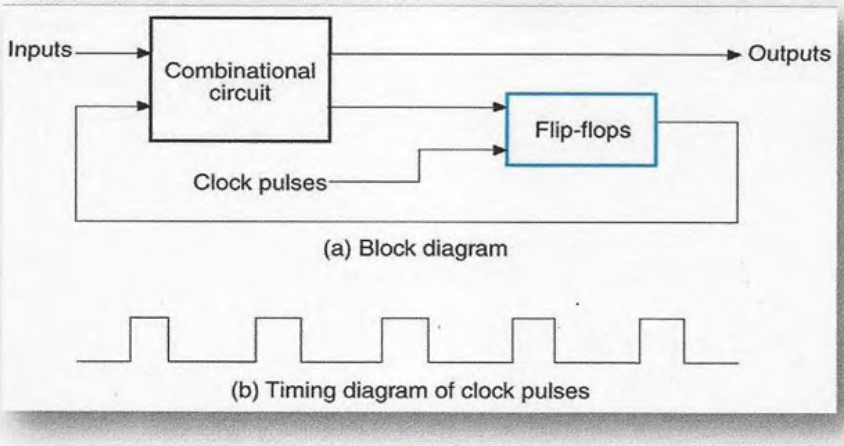
bit 2 **WREN:** EEPROM Write Enable bit
1 = Allows write cycles
0 = Inhibits write to the EEPROM

bit 1 **WR:** Write Control bit
1 = Initiates a write cycle. The bit is cleared by hardware once write is complete. The WR bit can only be set (not cleared) in software.
0 = Write cycle to the EEPROM is complete

bit 0 **RD:** Read Control bit
1 = Initiates an EEPROM read RD is cleared in hardware. The RD bit can only be set (not cleared) in software.
0 = Does not initiate an EEPROM read

Some Issues of Timing

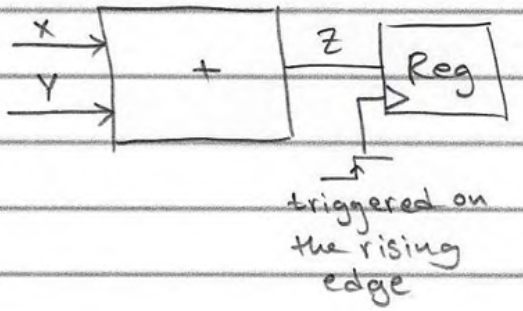
- **The Clock**
  - The microcontroller is made up of combinational and sequential logic. Thus, it requires a clock !
  - Clock – a continuously running fixed frequency logic square wave
  - Timers, counters, serial communication functions are also dependent on the clock
  - Operating frequency has direct impact on power consumption
  - Every microcontroller has a range for its clock → why? (next slide).

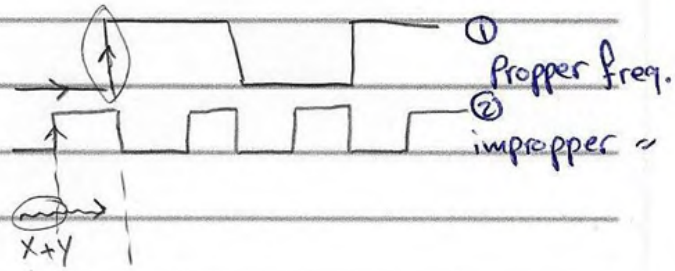


*Clock rate limitations:-

1) higher frequency means higher power consumption, which results in heat.

2) delay:-





in case ② the adder will write wrong results, since you didn't give it enough time to do the addition before the rising edge.

Some Issues of Timing

- **Instruction Cycle**
  - The main clock is divided by a fixed value (4 in the 16 series) into a lower-frequency signal
  - The cycle time of this signal is called the ***instruction cycle***
  - The primary unit of time in the action of processor

Clock frequency	Instruction cycle	
	Frequency	Period
20 MHz	5 MHz	200 ns
4 MHz	1 MHz	1 µs
1 MHz	250 kHz	4 µs
32.768 kHz	8.192 kHz	122.07 µs

if $F_{osc} = 20\,MHz$: freq. of the oscilator

$\rightarrow T_{osc} = \frac{1}{F_{osc}} = 0.5\,\mu sec.$

but in data sheet instruction's execution rate is $\frac{20}{4}\,M$

since in PIC microcontrollers one common clock, some devices use a clock derivated from the common clk such as CPU which needs clk rate $= \left(\frac{\text{Input clk}}{4}\right)$

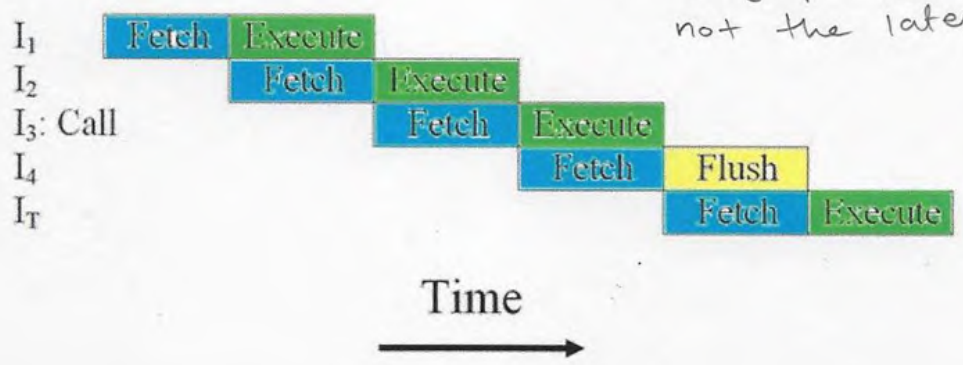
(this is called instructions freq.

$T_{cy} = \frac{4}{F_{osc}}$; instruction cycle

Some Issues of Timing

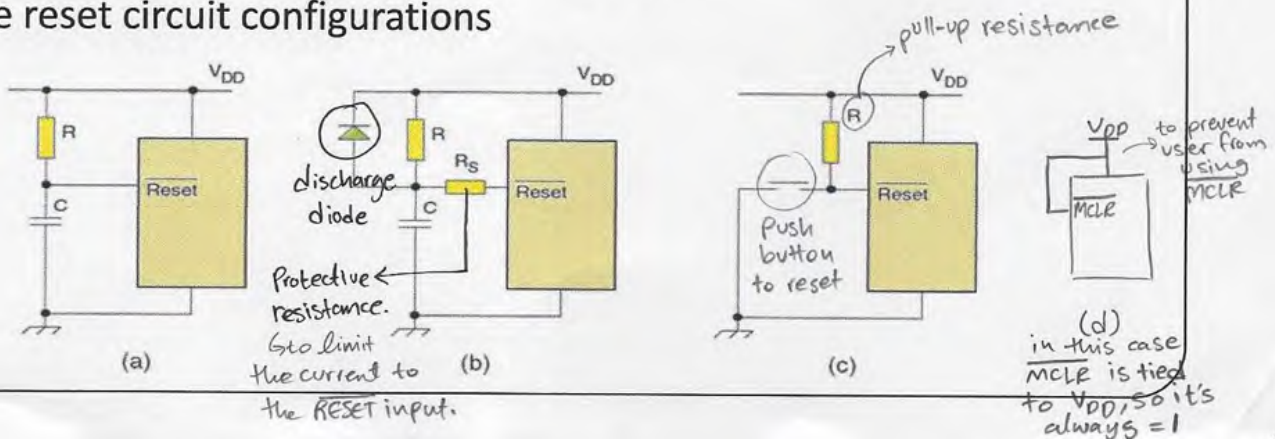
- **Pipelining**
  - Every instruction in the computer has to be fetched from memory and then executed. These steps are usually performed one after another
  - The CPU can be designed to fetch the next instruction while executing the current instruction. This improves performance significantly!
  - This is called ***Pipelining***
  - All PIC microcontrollers implement pipelining (RISC+Harvard make it easy)

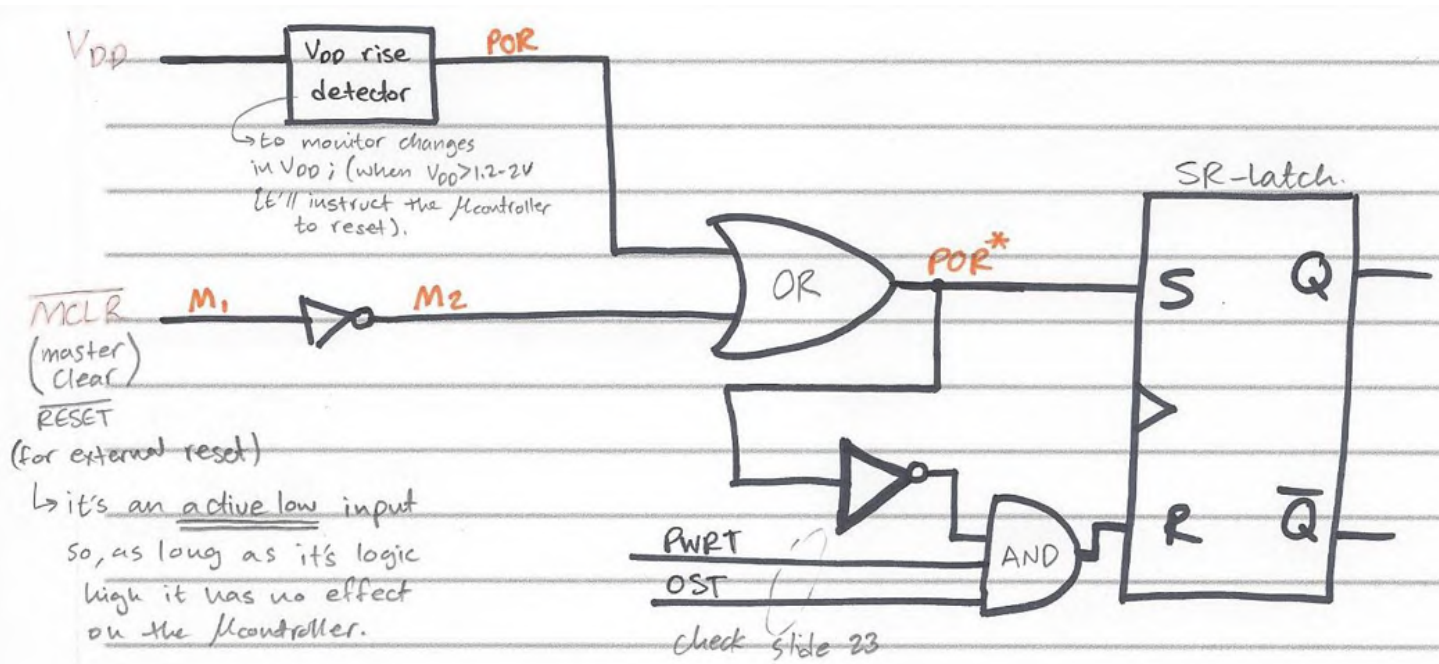
by this we improved the throughput (# of inst./time) not the latency (time/instr.)



Power-up and Reset

- On power-up, the microcontroller must start to execute the program stored in the program memory from its beginning (address 0000H)
- A specialized circuit inside the microcontroller detects this and is responsible for putting the microcontroller in the ***reset state***:
  - the program counter is set to zero
  - the SFRs are set such that the peripherals are in safe and disabled
- Another way to put the microcontroller in the reset state is to apply logic zero to the Master Clear input (MCLR)
- Some reset circuit configurations





remember:-

SR-latch's truth table:

S	R	Q	Q̄	
0	0	No change		
1	0	1	0	(set)
0	1	0	1	(reset)
1	1	invalid		

* In order to reliably start program execution on PIC μc., it's necessary to hold the device in a reset condition until the power supply has reached a consistently high enough voltage.

* An extended reset interval can be achieved by:

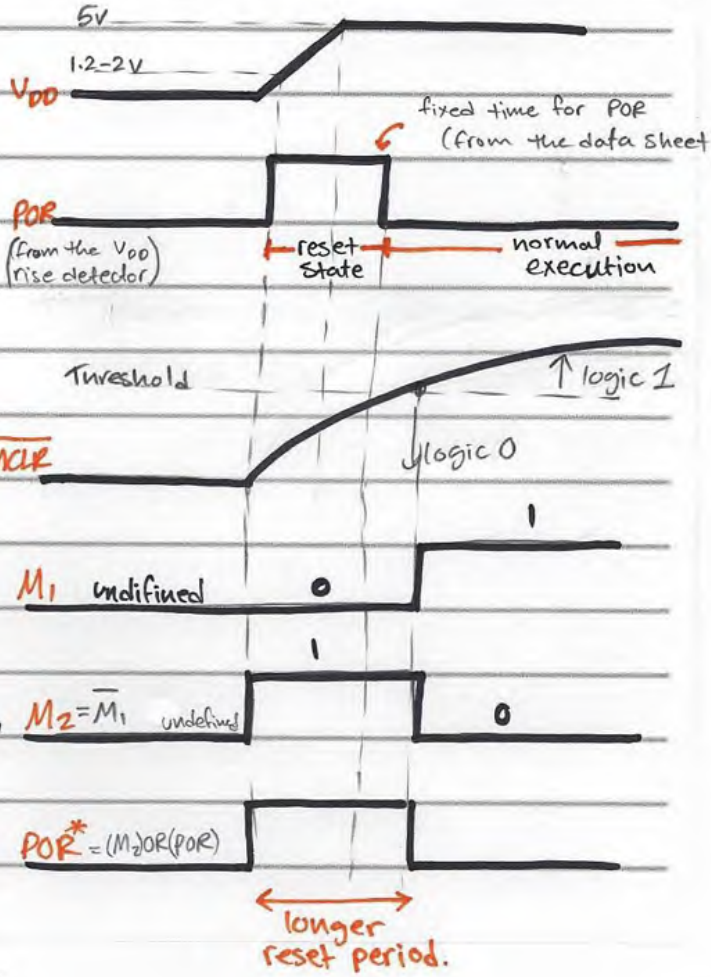
1) an external RC CKT attached to the ($\overline{MCLR}$) Pin. (as in Fig(a) slide 22)

(As long as the $\overline{MCLR}$ pin is held low, the μcontroller is held in reset. When it's taken high, the program execution starts.

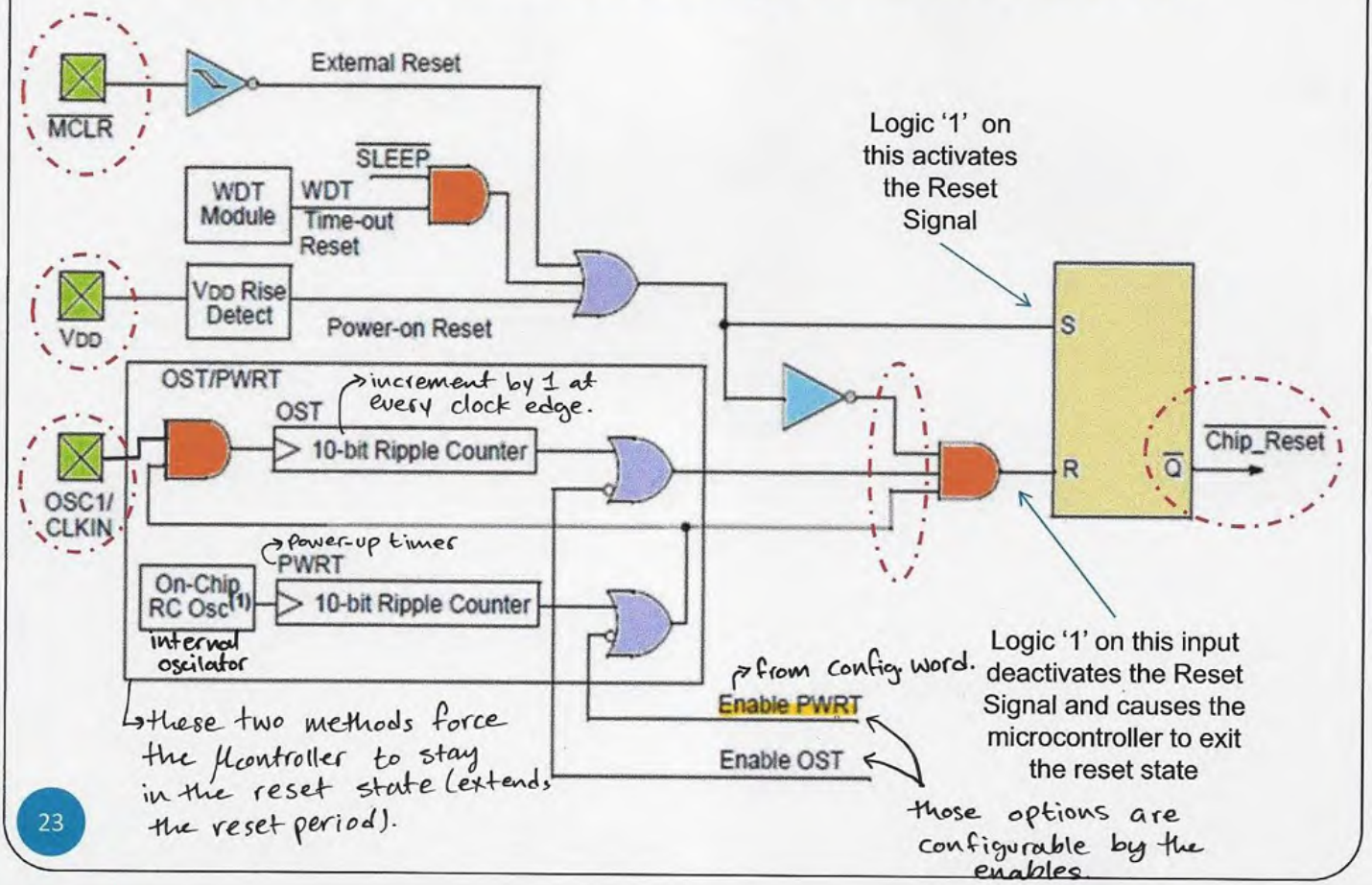
* When do we need longer reset period?

→ when $\frac{\Delta V_{DD}}{\Delta t} < 0.05\,V/sec.$

original reset time won't be sufficient for the power supply to give stable voltage.



The 16F84A on-Chip Reset Circuit



WDT Module : interupt timer (Watch Dog timer)

- once it reaches its maximum, it resets the µcontroller (overflow)
- it's configurable by the programmer (from the configuration word).

* µcontroller can enter reset based on other factors than V_{DD} & $\overline{MCLR}$
  - WDT; internal reset source.
  - V_{DD} & $\overline{MCLR}$; external reset src.

  } absence of those will get the µcontroller out of the reset.

* Where can we adjust OST? (enable)

it's only activated when a crystal oscilator is connected.

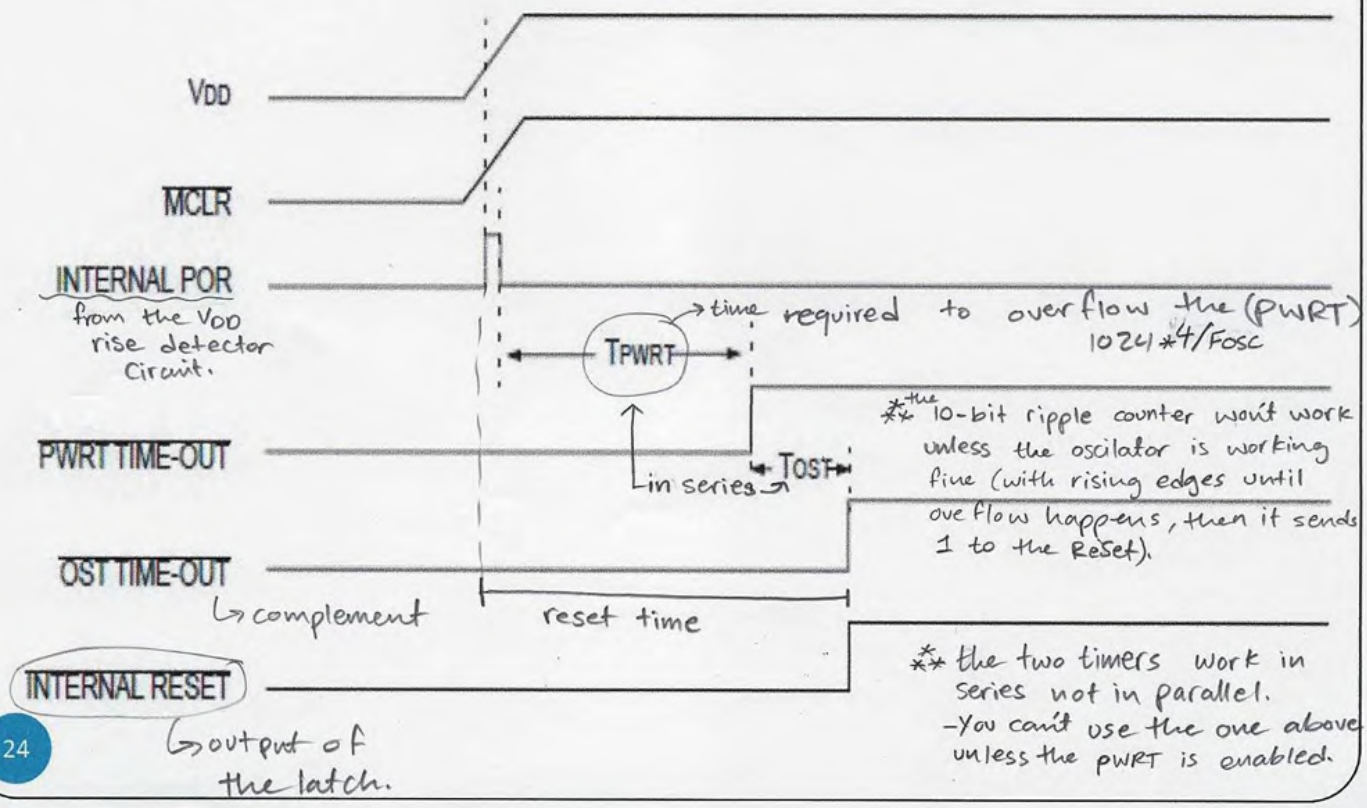
→ the type of the oscilator is defined from Fosc0 & Fosc1 bits in the config. word

Fosc0	Fosc1	OSTE
0	0	1
0	1	1
1	0	1
1	1	0

$OSTE = \overline{F_1 \cdot F_2}$ (AND)

The 16F84A on-Chip Reset Circuit

Example on reset timing when MCLR is disabled is connected to VDD



Summary

- The PIC 16F84A series is a diverse and cost effective family of microcontrollers
- The PIC 16F84A is pipelined RISC processor with Harvard architecture
- The PIC 16F84A has three different memory types
- An important memory area is the Special Function Register area which act as link between the CPU and peripherals
- Reset operation must be understood for proper operation of the microcontroller

Starting to Program

Chapter 4
Sections 1-4 , 10

Dr. Iyad Jafar

Outline

- Introduction
- Program Development Process
- The PIC 16F84A Instruction Set
- Examples
- The PIC 16F84A Instruction Encoding
- Assembler Details
- Sample Programs

Introduction

- Every computer can recognize and execute a group of instructions called the *Instruction Set*
- These instruction are represented in binary (*machine code*)
- *A program* is a sequence of instructions drawn from the instruction set and combined to perform specific operation
- To run the program:
  - It is loaded in binary format in the system memory
  - The computer steps through every instruction and execute it
  - Execution continues unless something stops it like the end of program or an interrupt

How to Write Programs

- **Machine code**
  - Use the binary equivalent of the instructions
  - Slow, tedious, and error-prone

```
00 0111 0001 0101
```

- **Assembly**
  - Each instruction is given a mnemonic (= lookup table)
  - A program called *Assembler* converts to machine code
  - Rather slow and inefficient for large and complex programs

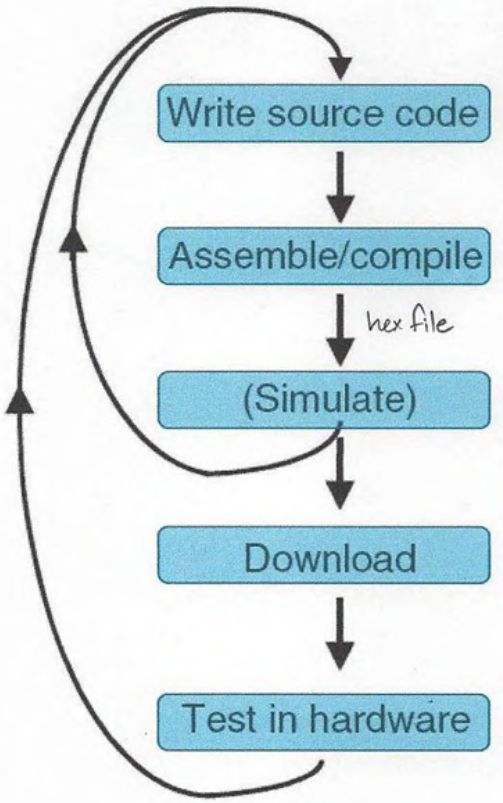
```
addw NUM, w
```

- **High-level language**
  - Use English-like commands to program
  - A program called *Compiler* converts to machine code
  - Easy !! The program could be inefficient !

```
for (i=0; i<10; i++) sum += a[i];
```

compiler: high-level —step ①→ Assembly —step ②→ machine lang.

Program Development Process

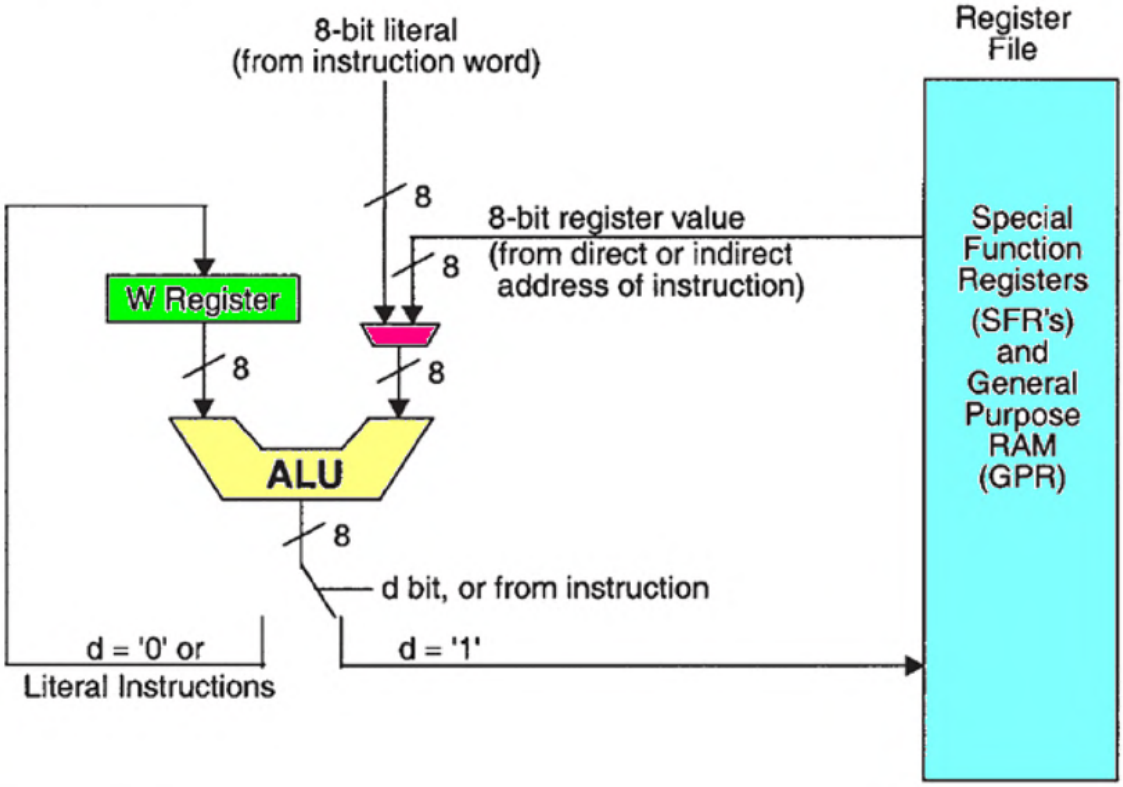


e.g: using MPLAB

optional
↳ (used to reduce # of write/Erase cycles of the Microcontroller.)

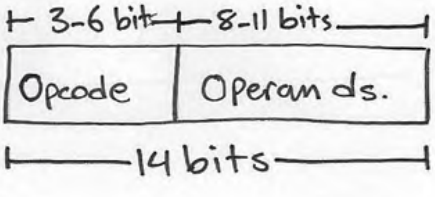
The PIC 16 Series Instruction Set

- **The PIC 16 Series ALU**



The PIC 16 Series Instruction Set

- 35 instructions !!! RISC (check Appendix 1 for the instruction set of PIC 16 series.)
- The binary code of the instruction itself is called the *Opcode*
- Most of these instruction operate/use on values called *Operands (ranging from no operands to two)*
- Three categories of instructions
  1. Byte-oriented file register operations
  2. Bit-oriented file register operations
  3. Literal and control operations
- Type of operations
  1. Arithmetic
  2. Logic
  3. Data movement
  4. Control
  5. Misc



The PIC 16 Series instruction set

TABLE A1.1 PIC 16 Series Instruction Set Summary

Mnemonic, operands		Description	Cycles	14 Bit opcode MSb			LSb	Status affected	Notes
BYTE ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f,d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f,d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW		Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f,d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f,d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f,d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f,d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f,d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f,d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f,d	Movef	1	00	1000	dfff	ffff	Z	1,2
MOVW	f	Move W to f	1	00	0000	lfff	ffff		
NOP		No Operation	1	00	0000	0xx0	0000		
RLF	f,d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f,d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f,d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f,d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f,d	Exclusive OR W with f	1	00	0110	dfff	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	0lbb	bfff	ffff		1,2
BTFSC	f,b	Bit Test f, Skip if Clear	1(2)	01	l0bb	bfff	ffff		3
BTFSS	f,b	Bit Test f, Skip if Set	1(2)	01	llbb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	lllx	kkk	kkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkk	kkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkk	kkk		
CLRWDT		Clear Watchdog Timer	1	00	0000	0110	0100	TO.PD	
GOTO	k	Go to address	2	10	lkkk	kkk	kkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkk	kkk	Z	

Table A1.1 PIC 16 Series Instruction Set Summary—Cont'd

Mnemonic, operands		Description	Cycles	14 Bit opcode MSb			LSb	Status affected	Notes
MOVL	k	Move literal to W	1	11	00xx	kkk	kkk		
RETFIE		Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	0lxx	kkk	kkk		
RETURN		Return from Subroutine	2	00	0000	0000	1000		
SLEEP		Go into standby mode	1	00	0000	0110	0011	TO.PD	
SUBLW	k	Subtract W from literal	1	11	110x	kkk	kkk	C.DC.Z	
XORLW	k	Exclusive OR literal with W	**1**	11	1010	kkk	kkk	Z	

Note 1: 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'.
2: If this instruction is executed on the TMR0 register (and, where applicable, d 1), the prescaler will be cleared if assigned to the Timer 0 module.
3: 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) Bit-Oriented file reg. instructions: (4 instr.)

they access file reg. & modify or test a single bit.

ex: BCF 0x10 , 3
(op-1 = 0x10, op-2 = 3)

BCF: clear a bit in file reg.
0x10 → 7-bits
3 → clears bit # 3; index starts from 0
(3-bit operand): to address 8 numbers. (0→7)

3) literal (constant) & control instructions: (13 instr.)

they deal with constants.

ex: ADDLW D'15'

ADDLW: adds literal to the working reg.
D'15': D means Decimal.
8-bit operand called (k)

; (you have no choice of saving at file reg. (since we'll need ffd = 8 extra bits to do it).

GOTO 0x01F

GOTO: control
11-bit operand also called (k)
→ this is considered as literal instr. since we need to specify an address (constant)

1) Byte-oriented File register instructions: (18 inst.)

they process bytes in file registers.

ex: ADDWF 0x20 , 0

op code operand 1 operand 2

↳ ** the opcode is not case sensitive. ADDWF = addwf

op-1. 0x20 is a 7-bit address operand called (f): file reg. address operand.

in this case:-

0x20 ≡ 0xA0 because 0xA0 has the same row as 0x20 but with different bank (bank 1). so, taking 7-bits only (neglecting the bank selection bit) they'll look the same.

op-2. 0 ; is a 1-bit operand called (d)

direction / distination.

↳ 0 : save in working reg.

↳ 1 : save in file reg. with address f (we can't specify another address since max. # of operands is 2).

1+7+7=15 > 14 !

The PIC 16 Series Instruction Set

- **Introduction to PIC 16 ISA**
  - Types of operands
    - A 7-bit address for a memory location in RAM (Register File) denoted by **f**
    - A 3-bit to specify a bit location within an the 8-bit data denoted by **b**
    - A 1-bit to determination the destination of the result denoted by **d**
    - A 8-bit number for literal data or 11-bit number for literal address denoted by **k**

The PIC 16 Series Instruction Set

- Examples
  - **clrw**
    - Clears the working register W
  - **clrf f**
    - Clears the memory location specified by the 7-bit address f
  - **addwf f , d**
    - Adds the contents of the working register W to the memory location with 7-bit address in f. the result is saved in *W if d =0, or in f if d = 1*
  - **bcf f , b**
    - Clears the bit in position specified by b in memory location specified by 7-bit address f
  - **addlw k**
    - Adds the content of W to the 8-bit value specified by k. The result is stored back in W

The PIC 16 Series Instruction Set

Byte-oriented File Register Operations

- Format: **op f, d**
  - **op**: operation
  - **f**: address of file or register
  - **d**: destination (0: working register, 1: file register)

- Example: address label (this is case sensitive Porta ≠ PORTA)

```
addwf   PORTA, 0
```

Adds the contents of the working register and register PORTA, puts the result in the working register.

The PIC 16 Series Instruction Set

Bit-oriented File Register Operations

- Format: **op f, b**
  - **op**: operation
  - **f**: address of file or register
  - **b**: bit number, 0 through 7

- Example:

```
bsf STATUS, 5
```

Sets to 1 Bit 5 of register STATUS. → at 03H & 83H

The PIC 16 Series Instruction Set

Literal and Control Operations

- Format: **op k**
  - **op**: operation
  - **k**: literal, an 8-bit if data or 11-bit if address

- Examples:

```
addlw   5
```

Adds to the working register the value 5.

```
call    9
```

Calls the subroutine at address 9.

Summary :-

* Byte-oriented instr.

Op f, d → ADDWF / SUBWF / ANDWF

Op f → CLRF

op → CLRW

* Bit-oriented instr.

Op f, b → BTFSC / BSF

* literal & control

Op k (8-bits) → ADDLW / ANDLW

Op → RETURN / RETFIE

op k (11-bits) → GOTO / CALL

The PIC 16 Series Instruction Set

Arithmetic Instructions

Mnemonic	Operands	Description	Cycles	Status Affected
ADDWF	f, d	Add W and f	1	C,DC,Z
COMF	f, d	Complement f (one's complement)	1	Z
DECF	f, d	Decrement f	1	Z
INCF	f, d	Increment f	1	Z
SUBWF	f, d	Subtract W from f	1	C,DC,Z
ADDLW	k	Add literal and W	1	C,DC,Z
SUBLW	k	Subtract W from literal	1	C,DC,Z

* Carry isn't affected

F - W_reg ←

L - W_reg ←

W is always the subtracter (after -)

d = 0 , result is stored in W

d = 1 , result is stored in F

→ to achieve 2's comp.:
① 1's comp.
② INCF (add 1)

e.g.:-
00101011
↓ 1's comp.
11010100
↓ +1
11010101
= 2's comp.

Ex:- ① ADDWF 0x42, 1 , let W_{reg} = 0xFC & Mem[0x42] = 0x17

** in PIC 16 all the GPRs are in Bank 0, so we don't care about the bank selection bit.

Sol:-

```
  1  1
 0xFC
 0x17  +
------
0x113
```

overflow (we only have 8 bits.)

⟹ after executing the instr.:

W_{reg} = 0xFC

Mem[0x42] = 0x13

Carry & Digital Carry bits in Status reg = 1

↳ the carry from the 4th bit

Zero flag = 0

② SUBWF 0x10, 0 , let W_{reg} = 0x10 & Mem[0x10] = 0x13

↳ we deal with (borrow) instead of (carry)

Sol:

```
 0x13
 0x10  -
------
 0x03
```

⇒ No borrow (file reg. > W_{reg})

∴ carry bit = 1 ; this indicates that there's no borrow

*if W_{reg} = 0x20

```
 0x13
 0x20  -
```

= 2's complement of (0x20)

```
        0x13  +
```

* Carry = 0

③ let STATUS = 0x0A , Mem[0x31] = 0xFF , W_{reg} = 0x00

STATUS = 0x0A = 0000 1 0 1 0 (Z DC C)

INCF 0x31, 0

0xFF + 1 = 0x00 (with overflow)

after execution:-

W_{reg} = 0x00

Mem[0x31] = 0xFF

STATUS ⟶ carry & DC aren't affected.

Zero bit = 1

∴ STATUS = 0x0E

The PIC 16 Series Instruction Set

Logic Instructions

Mnemonic	Operands	Description	Cycles	Status Affected
ANDWF	f, d	AND W with f	1	Z
IORWF	f, d	Inclusive OR W with f	1	Z
XORWF	f, d	Exclusive OR W with f	1	Z
ANDLW	k	AND literal with W	1	Z
IORLW	k	Inclusive OR literal with W	1	Z
XORLW	k	Exclusive OR literal with W	1	Z

bit-wise

Period = $\frac{4}{Fosc}$ (time / instr. cycle)

d = 0 , result is stored in W
d = 1 , result is stored in F

The PIC 16 Series Instruction Set

Data Movement Instructions

Mnemonic	Operands	Description	Cycles	Status Affected
MOVF	f, d	Move f	1	Z
MOVWF	f	Move W to f	1	
SWAPF	f, d	Swap nibbles in f	1	
MOVLW	k	Move literal to W	1	

why (d) is used? it only copies f to f? (next slide).

Nibble Nibble
4-bit 4-bit

it copies the reg. content

$W_{reg(new)} = F = W_{reg(old)}$

* MOVFF f_1, f_2 ×
7+7+3-6 > 14 ×
≡ { MOVF f_2, 0
MOVWF f_1

d = 0 , result is stored in W
d = 1 , result is stored in F

* In MOVF instr. we use (d), so that when we put d=1 it copies f to f & the zero flag will equal one if f contents are 0x00, thus, this is a way to check if F.reg. is empty, another way to check this are:-

e.g: to check if Mem[0x33] = 0x00 ?

[1] CLRW or MOVLW 0x00
SUBWF 0x33,0
if z==1 ? ∴ Freg is empty

[2] MOVLW 0xFF
ANDWF 0x33,0
z==1 ? ✓ ∴ empty

[3] MOVLW 0x00
IORWF 0x33

[4] INCF 0x33, 0 or 1
DECF 0x33, 0 or 1 → if z==1 ⇒ empty

* So, why did they add (d) & made it to affect Z while there are many ways to do it otherwise?
→ because it's a frequently used operation

The PIC 16 Series Instruction Set

Control Instructions

Mnemonic	Operands	Description	Cycles	Status Affected
DECFSZ	f, d	Decrement f, Skip if 0	1 (2)	
INCFSZ	f, d	Increment f, Skip if 0	1 (2)	
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	
CALL	k	Call subroutine	2	
GOTO	k	Go to address	2	
RETFIE	-	Return from interrupt	2	
RETLW	k	Return with literal in W	2	
RETURN	-	Return from Subroutine	2	

Skips the next instr.

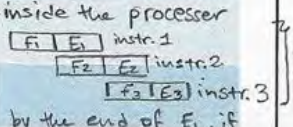
usually used in for loops.

Conditional flow control instructions.

overflow → eg: FF+1

? this is determined based on the test results (condition)
- False → 1 cycle
- True → you need to skip the next instr. thus, a cycle will be lost due to pipelining
e.g:- 1) DECFSZ 0x10, 1
2) CLRW
3) CLRF 0x20

↳ inside the processer



Pipelined Processer with a depth of 2 (FAE)

by the end of E_1, if the result is to skip, the processer will skip instr.2 which has been already ~~Fetched~~, so a cycle will be lost.

d = 0 , result is stored in W , d = 1 , result is stored in F

The PIC 16 Series Instruction Set

Miscellaneous Instructions

Mnemonic	Operands	Description	Cycles	Status Affected
CLRF	f	Clear f	1	Z
CLRW	-	Clear W	1	Z
NOP	-	No Operation	1	
RLF	f, d	Rotate Left f through Carry	1	C
RRF	f, d	Rotate Right f through Carry	1	C
BCF	f, b	Bit Clear f	1	
BSF	f, b	Bit Set f	1	
CLRWDT	-	Clear Watchdog Timer	1	TO',PD'
SLEEP	-	Go into standby mode	1	TO',PD'

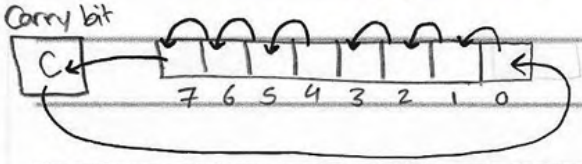
used to waste time! it affects nothing (used to write software delays)

in order not to affect the Z flag, you can use MOVLW 0x0 instead.

used to prevent the timers overflow which resets the μcont.

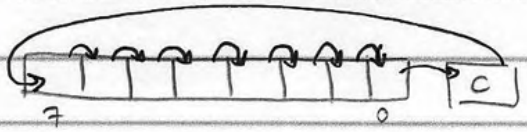
d = 0 , result is stored in W , d = 1 , result is stored in F

RLF : rotate ↑ (left) through carry



**try to write a code to rotate without going through the carry!

RRF



what about a shift instruction?

SLF f,d X ≡ { BCF 0x03,0 ; clear the carry
RLF f,0

SRLF f,d X ≡ { BCF 0x03,0
RRF f,0

Logical Shift (add zero)

SRAF f,d X ≡ { think about it !

Arithmetic Shift (sign ↓ →)

The PIC 16 Series Instruction Set

Examples

Instruction	Operation	Flags Affected
bcf 0x31, 3	clear bit 3 in location 0x31	None
bsf 0x04 , 0	set bit 0 location 0x04	None
bsf STATUS , 5	set bit 5 in STATUS register to select bank 1 in memory	None
bcf STATUS , C	clear the carry bit in the status register	None
addlw 4	Adds 4 to working register W and store the result in back in W	C, DC, Z
addwf 0x0C, 1	Add the content of location 0x0C to W and store the result in 0CH (d =1)	C, DC , Z
sublw 10	Subtract W from 10 and put the result in W	C, DC, Z
subwf 0x3C, 0	Subtract W from contents of location 0x3C and store the result in W	C, DC, Z

label, must be defined by user.

The PIC 16 Series Instruction Set

Examples

Instruction	Operation	Flags Affected
incf 0x06, 0	Increment location 0x06 by 1 and store result in W	Z
decf TEMP, 1	Decrement location TEMP by 1 and store in TEMP	Z
compf 0x10, 1	Complement the value in location 10H and store in 0x10	Z
andlw B'11110110'	AND literal value 11110110 with W and store result in W	Z
andwf 0x33, 1	AND location 0x33 with W and store result in 0x33	Z
iorlw B'00001111'	Inclusive-or W with 00001111	Z
iorwf X1 , 0	Inclusive-or W with location X1 and store result in W	Z
xorlw B'01010101'	Exclusive-or W with 01010101	Z
xorwf 0x2A , 0	Exclusive-or W with location 0x2A and store result in W	

The PIC 16 Series Instruction Set

Examples

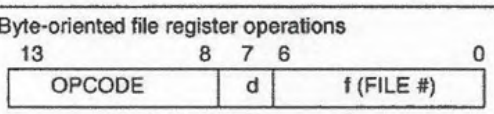
Instruction	Operation	Flags Affected
clrw	Clear W	Z
clrf 0x01	Clear location 0x01	Z
movlw 18	Move literal value 18 into W	NONE
movwf 0x40	Move contents of W to location 0x40	NONE
movf 0x21 , 0	Move contents of location 0x21 to W	Z
movf 0x21 , 0x33	Incorrect syntax	--
movwf 0x1B , 1	Incorrect syntax	--
swapf T1 , 1	Swap 4-bit nibbles of location T1	NONE
swapf DATA , 0	Move DATA to W, swap nibbles, no change on DATA	NONE
rlf TEMP , 1	Rotate contents of location TEMP to left by one bit position through the C flag	C
rlf 0x25 , 0	Copy contents of 0x25 to W and rotate to left by one bit position through the C flag	C

The PIC 16 Series Instruction Set Encoding

the processer identifies the type of the instr. by the last 2 bits (Most significant 2 bits) [check appendix 1]

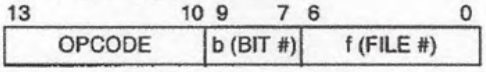
MSB
Byte-oriented 00
bit- " 01
Control & literal (mixed)

Byte-oriented file register operations



d = 0 for destination W
d = 1 for destination f
f = 7-bit file register address

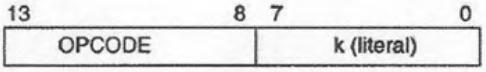
Bit-oriented file register operations



b = 3-bit bit address
f = 7-bit file register address

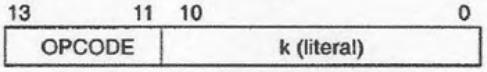
Literal and control operations

General



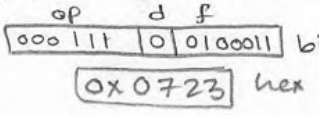
k = 8-bit immediate value

CALL and GOTO instructions only



k = 11-bit immediate value

ex:-
ADDWF 0xA3, 0



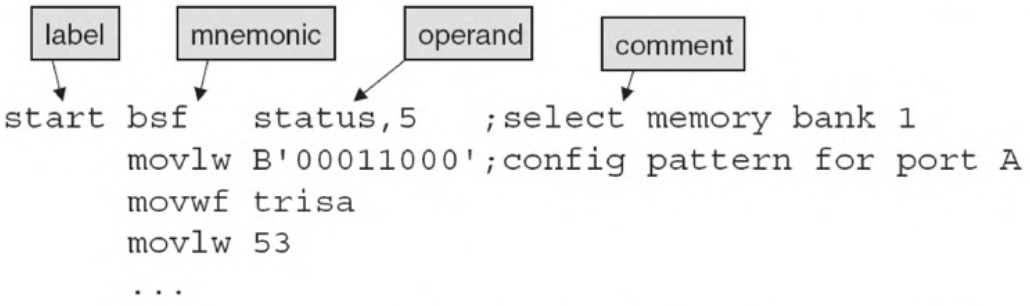
binary

0x0723 hex

Check Appendix A for opecode binary codes

Assembler Details

- Any assembler line may have up to four different elements



- We can specify values in different bases in assembler programs

Radix	Example
Decimal	D'255'
Hexadecimal	H'8d' or 0x8d
Octal	O'574'
Binary	B'01011100'
ASCII	'G' or A'G'

Assembler Details

- **Assembler directives**
  - These are assembler-specific commands to aid the processing of assembly programs

Assembler directive	Summary of action
#include	Include additional source file
org (originate).	Set program origin
equ	Define an assembly constant; this allows us to assign a value to a label
cblock and endc	Define a block of variables
end	End program block

group of equate statements.

e.g:-
STATUS EQU 0x03
C EQU 0
:
BCF STATUS,C instead of BCF 0x03,0.

→instead of defining SFRs. Labels.

ORG 0x0000
MOVLW D'10' ← this instr.'s address will be 0x0000
:
ORG 0x0110
MOVF 0x20,0 ← addr. = 0x0110

→this is for the assembler to terminate (end) assembling process not to end program execution!

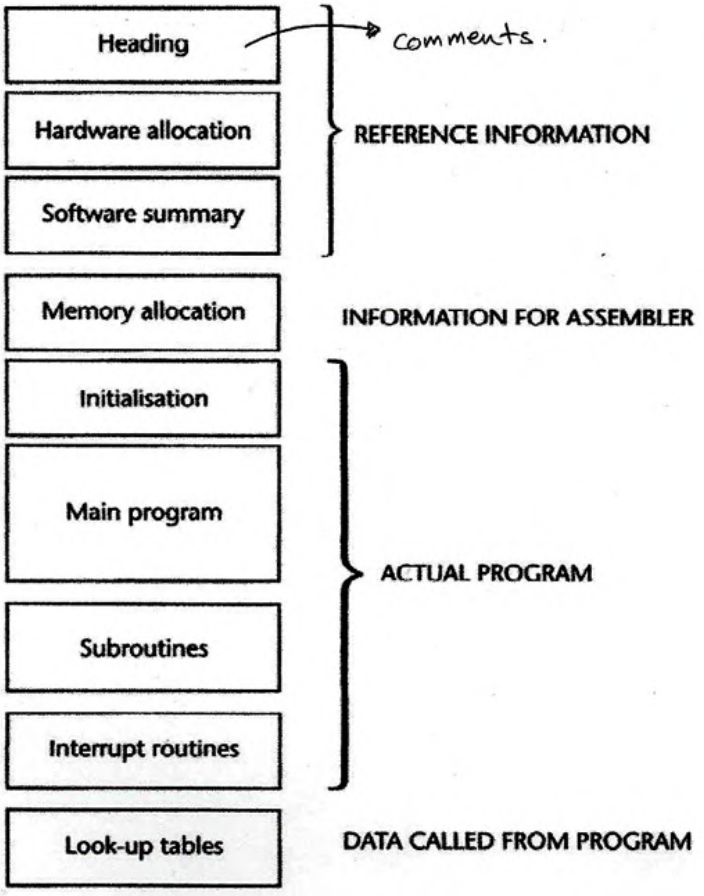
"P16F84.inc" Equates:

```
;==============================
;       Register Definitions
;==============================

W               EQU     H'0000'
F               EQU     H'0001'

;----- Register Files-----------

INDF            EQU     H'0000'
TMR0            EQU     H'0001'
PCL             EQU     H'0002'
STATUS          EQU     H'0003'
FSR             EQU     H'0004'
PORTA           EQU     H'0005'
PORTB           EQU     H'0006'
EEDATA          EQU     H'0008'
EEADR           EQU     H'0009'
PCLATH          EQU     H'000A'
INTCON          EQU     H'000B'

OPTION_REG      EQU     H'0081'
TRISA           EQU     H'0085'
TRISB           EQU     H'0086'
EECON1          EQU     H'0088'
EECON2          EQU     H'0089'
```

```
;----- STATUS Bits -------------
IRP             EQU     H'0007'
RP1             EQU     H'0006'
RP0             EQU     H'0005'
NOT_TO          EQU     H'0004'
NOT_PD          EQU     H'0003'
Z               EQU     H'0002'
DC              EQU     H'0001'
C               EQU     H'0000'
;----- INTCON Bits -------------
GIE             EQU     H'0007'
EEIE            EQU     H'0006'
T0IE            EQU     H'0005'
INTE            EQU     H'0004'
RBIE            EQU     H'0003'
T0IF            EQU     H'0002'
INTF            EQU     H'0001'
RBIF            EQU     H'0000'
;----- OPTION Bits -------------
NOT_RBPU        EQU     H'0007'
INTEDG          EQU     H'0006'
T0CS            EQU     H'0005'
T0SE            EQU     H'0004'
PSA             EQU     H'0003'
PS2             EQU     H'0002'
PS1             EQU     H'0001'
PS0             EQU     H'0000'
```

```
;----- EECON1 Bits -------------
EEIF            EQU     H'0004'
WRERR           EQU     H'0003'
WREN            EQU     H'0002'
WR              EQU     H'0001'
RD              EQU     H'0000'

;==============================
;       Configuration Bits
;==============================

_CP_ON          EQU     H'000F'
_CP_OFF         EQU     H'3FFF'
_PWRTE_ON       EQU     H'3FF7'
_PWRTE_OFF      EQU     H'3FFF'
_WDT_ON         EQU     H'3FFF'
_WDT_OFF        EQU     H'3FFB'
_LP_OSC         EQU     H'3FFC'
_XT_OSC         EQU     H'3FFD'
_HS_OSC         EQU     H'3FFE'
_RC_OSC         EQU     H'3FFF
```

Program Structure



* Remember that directives are not executed in the program, they are only used in assembling process.
(they dont have addr. in the program memory).

Sample Program 1

- Write a program to add the numbers stored in locations 31H, 45H, and 47H and store the result in location 22H

Sample Program 1

```
;************************ EQUATES ********************************
STATUS      equ     0x03            ; define SFRs
RP0         equ     5
;************************ VECTORS ********************************
            org     0x0000          ; reset vector
            goto    START
            org     0x0004 ; to skip the interrupt vector.
INVEC       goto    INVEC           ; interrupt vector
;************************ MAIN PROGRAM ***************************
START       bcf     STATUS , RP0    ; select bank 0
            movf    0x31 , 0        ; put first number in W
            addwf   0x45 , 0        ; add second number
            addwf   0x47 , 0        ; add third number
            movwf   0x22            ; save result in 0x22
DONE        goto    DONE            ; endless loop ; to stop program
                                                       execution.
            end
```

addr. 0x00

0x04

addr. 0x05, 0x06, ...

no need to do it; since all GPRs are located at bank 0, & even if RP0=1 it'll be mapped to the corresponding GPR at bank 0.

it's not necessary in this program to do this since we didn't use it

Sample Program 2

- Write a program to swap the contents of location 0x33 with location 0x11

Sample Program 2

```
; ************************ EQUATES ********************************
STATUS          equ     0x03            ; define SFRs
RP0             equ     5
;************************ VECTORS ********************************
;
                org     0x0000          ; reset vector
                goto    START
                org     0x0004
INVEC           goto    INVEC           ; interrupt vector
;************************ MAIN PROGRAM ***************************
;
START           bcf     STATUS , RP0    ; select bank 0
                movf    0x33 , 0        ; put first number in W
                movwf   0x22            ; store the 1st number temporarily
                movf    0x11 ,  0       ; get 2nd number
                movwf   0x33            ; store 2nd in place of 1st
                movf    0x22 , 0        ; get 1st number from 0x22
                movwf   0x11            ; store 1st  in place of 2nd
DONE            goto    DONE            ; endless loop
                end
```

Summary

- The PIC 16F84A has 35 instructions to perform different computational and control operations
- Programs can be written using different levels of abstraction
- Using assemblers simplifies the program development process
- There exist many IDE to aid writing programs and simulate their behavior before putting them into hardware

Building Assembler Programs

Chapter 5
Sections 1-6

Dr. Iyad Jafar

Outline

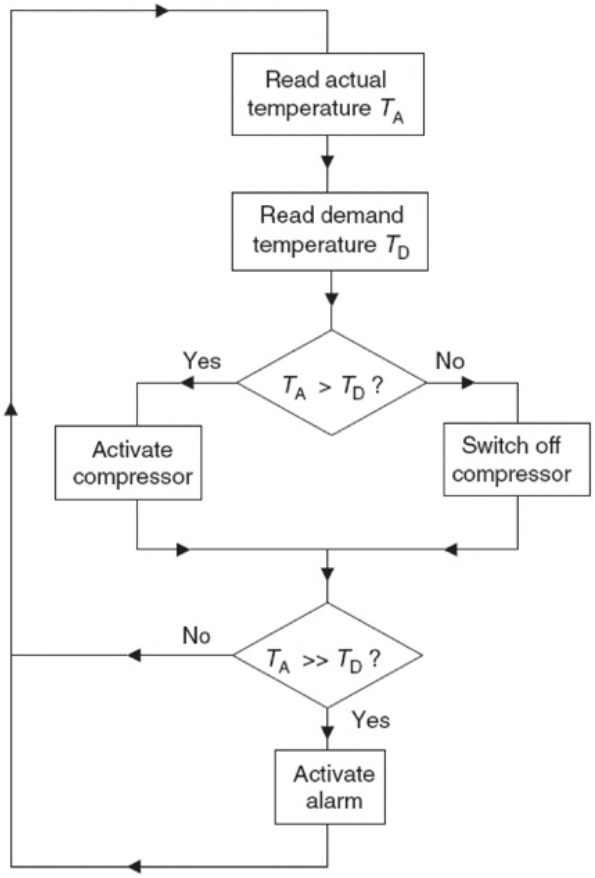
- Building Structured Programs
- Conditional Branching
- Subroutines
- Generating Time Delays
- Dealing with Data
- Example Programs

Building Structured Programs

- Writing programs is not an easy task; especially with large and complex programs
- It is essential to put down a design for the program before writing the first line of code
- This involves documenting the programs flow charts and state diagrams

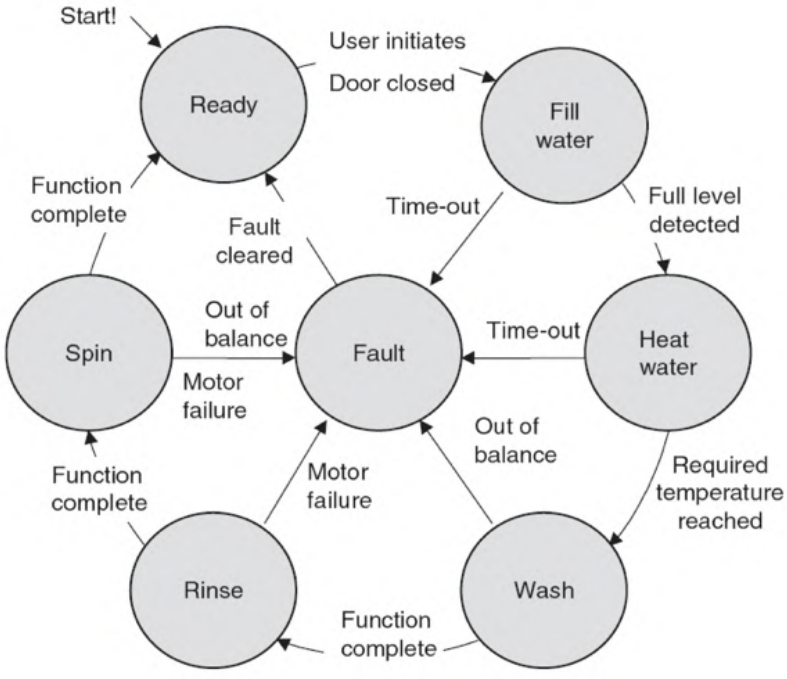
Building Structured Programs

- **Flowcharts**
  - Rectangle for process
  - Diamond for decision



Building Structured Programs

- **State Diagrams**
  - Circle for state
  - Arrow for state transition labeled with condition(s) that causes the transition

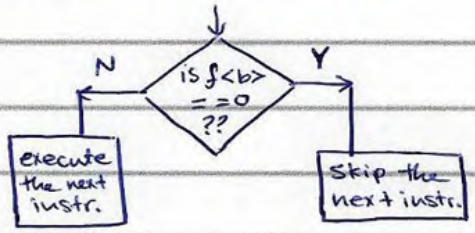


Conditional Branching

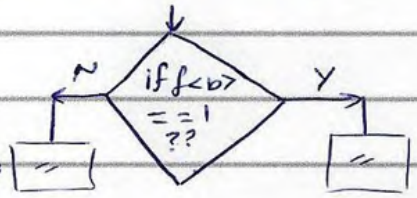
- Microprocessors and microcontroller should be able to make decisions
- This enables them to behave according to the state of logical variables
- The PIC 16 series is not an exception ! They have *four conditional skip* instructions
- These instructions *test for a certain condition and skip the following instruction* if the tested condition is true !

Conditional instructions:-

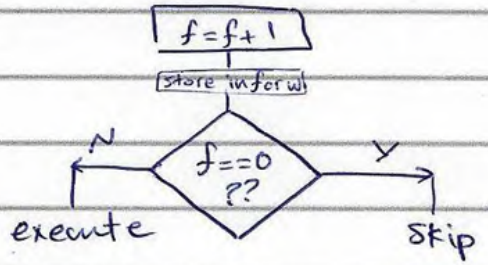
① BTFSC f, b



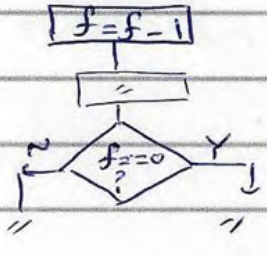
② BTFSS f, d



③ INCFS2 f, d



④ DECFSZ f, d



Conditional Branching

Instruction	Operation	Example
btfsc f , b	Test bit at position b in register f. skip next instruction if the bit is clear '0'	btfsc STATUS , 5
btfss f , b	Test bit at position b in register f. skip next instruction if the bit is set '1'	btfss 0x21 , 1

Instruction	Operation	Example
decfsz f , d	Decrement the contents of register f by 1 and place the result in W if d = 0 or in f if d = 1. Skip next instruction if the decremented result is zero	decfsz 0x44, 0
incfsz f , d	Increment the contents of register f by 1 and place the result in W if d = 0 or in f if d = 1. Skip next instruction if the incremented result is zero	incfsz 0xd1 , 1

Conditional Branching

- **Example1:** a program to add two numbers in locations 0x11 and 0x22. If there is ***no*** carry, store the result in location 0x33, else store the result in location 0x44

- Reminder: the STATUS Register

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

Conditional Branching
Example 1

```
STATUS    equ    0x03          ; define SFRs
          org    0x0000        ; reset vector
          goto   START
          org    0x0006
START     movf   0x11 , 0      ; get first number to W
          addwf  0x22 , 0      ; add second number
          btfsc  STATUS , 0    ; check if carry is clear
          goto   C_SET         ;  go to label C_Set if C==1
          movwf  0x33          ; store result in 0x33
          goto   DONE
C_SET     movwf  0x44
DONE      goto   DONE          ; endless loop
          end
```


EXAMPLE 1.5: Write a program that multiplies Location LOC by 10:

```
*include "P16F84A.inc"
 LOC     equ    0x20
 Counter equ    0x21      ; to store the loop index.
         ORG    0x0000
         MOVLW  0x09      ;to initialize the counter.
         MOVWF  Counter,
         MOVF   LOC, w
ADD      ADDWF  LOC,w
         DECFSZ Counter,f ; if w instead of f we'll enter an
                            endless loop.
         GOTO   ADD
         MOVWF  LOC
DONE     GOTO   DONE
         END
```

** if we want to use INCFSZ we must initialize the counter to 255-9=246

Conditional Branching

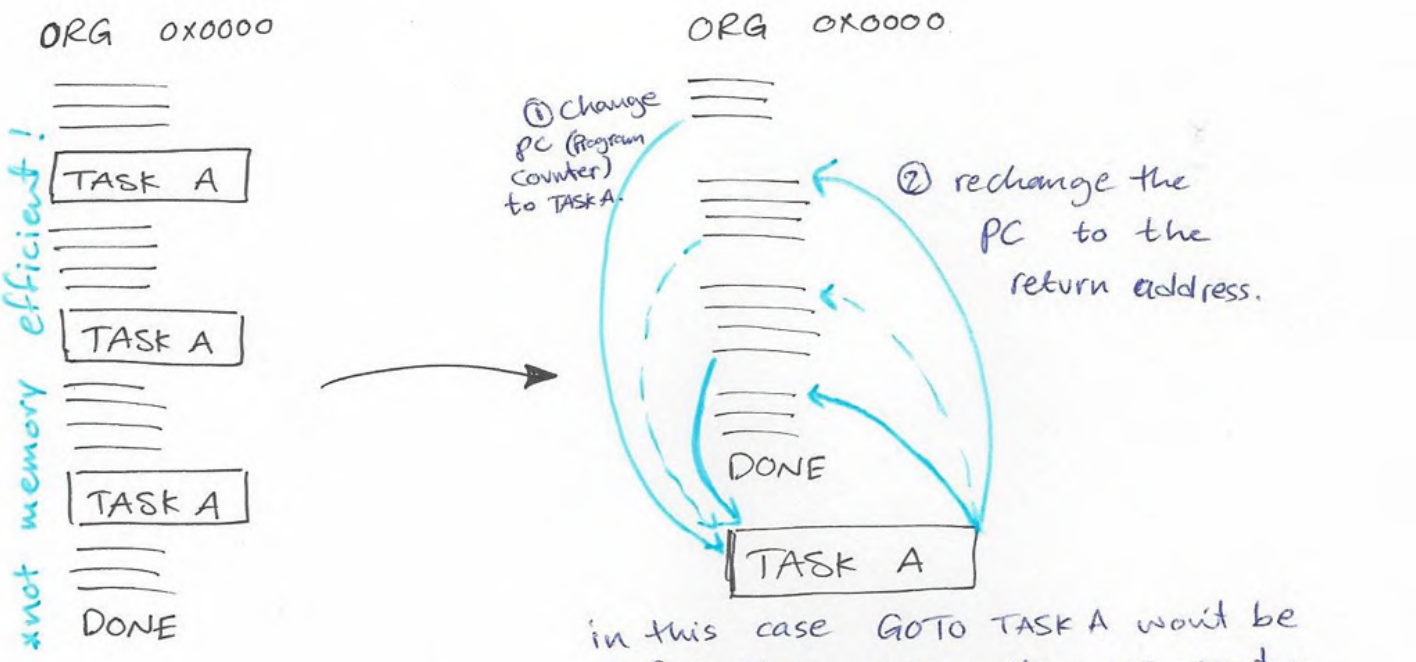
- **Example2:** assume a 16-bit counter with upper byte in location COUNTH and lower byte in location COUNTL. Write the code to decrement the counter until it is zero. "decrementing the counter is allowed if the counter is initially non-zero"

Conditional Branching
Example 2

```
COUNTL          equ     0x10            ; lower byte of counter in 0x10
COUNTH          equ     0x11            ; upper byte of counter in 0x11
                #include "P16F84A.INC"
                org     0x0000
START           movf    COUNTL , F      ; check if the both locations are zeros
                btfss   STATUS ,Z       ; if so, then finish
                goto    DEC_COUNTL      ; if COUNTL is not zero, decrement it
                movf    COUNTH , F      ; if it is zero check COUNTH
                btfsc   STATUS ,Z
                goto    DONE            ; if both are zeros, then DONE
                decf    COUNTH, F
DEC_COUNTL      decf    COUNTL,F
                goto    START
DONE            goto    DONE            ; program gets here if both are zeros
                end
```

Subroutines

- In many cases, we need to use a block of code in a program in different places
- Instead of writing the code wherever it is needed, we can use *subroutines/functions/procedures*
  - Blocks of code saved in memory and can be called/used from anywhere in the program
  - When a subroutine is called, execution moves to place where the subroutine is stored
  - Once the subroutine is executed, execution resumes from where it was before calling the subroutine



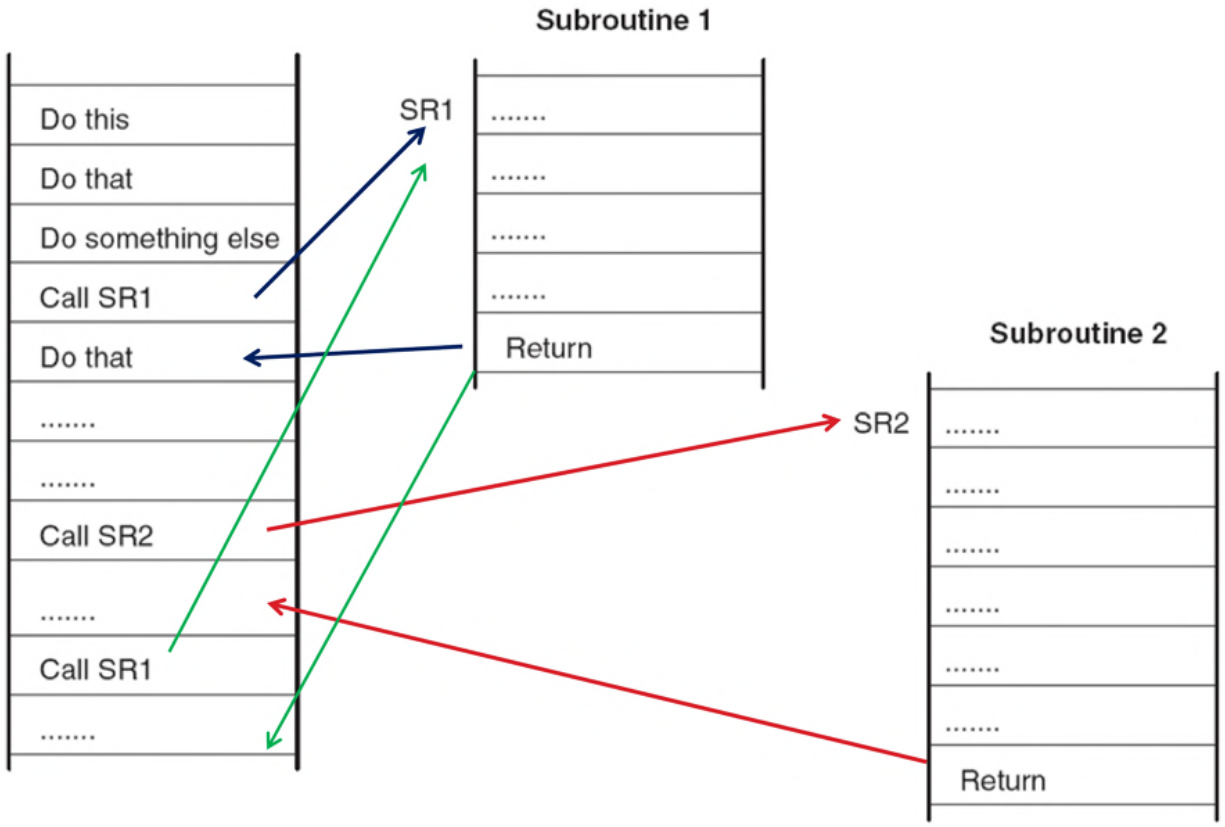
in this case GOTO TASK A won't be useful since every time we want a different return address so to get back to the main program won't work using GOTO instr.

Thus, we use CALL & RETURN

→ CALL instruction saves the address of the next instr. (the return addr.) in the Stack memory.

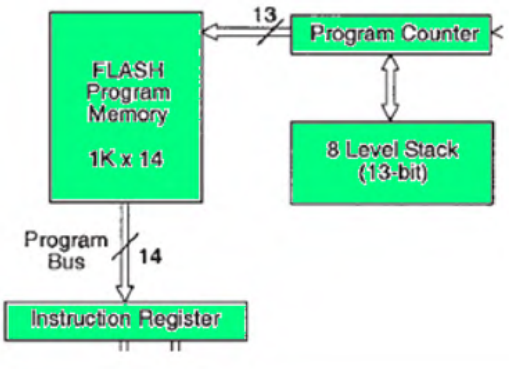
↳ Last in first out structure.

Subroutines



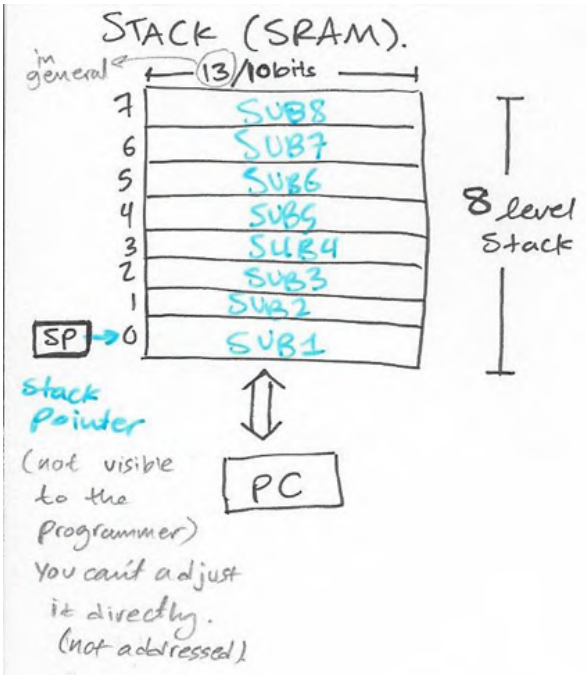
Subroutines

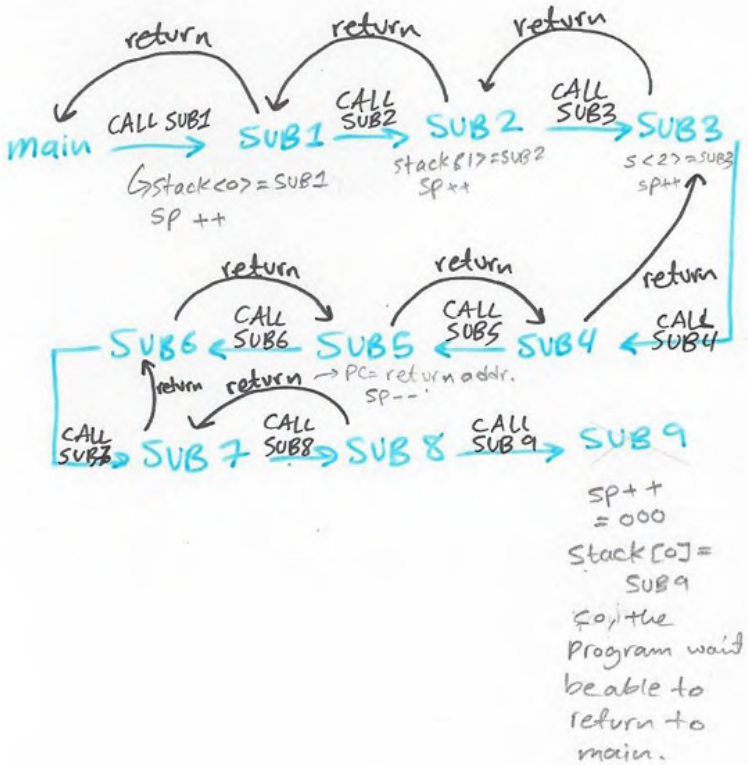
- The ***program counter*** holds the address of the instruction to be executed
- In order to call a subroutine, the program counter has to be loaded with the address of the subroutine
- Before that, the current value of the PC is saved in ***stack*** to assure that the main program can continue execution from the following instruction



Subroutines

- In PIC, to invoke a subroutine we use the ***CALL*** instruction followed by the address of the subroutine
- The address is usually *specified by a symbolic label in the program*
- To exit a subroutine and return to the main program, we use the ***RETURN*** or ***RETLW*** instructions





Subroutines - Example

```
; A subroutine to perform multiplication between locations 0x30 and 0x31. the result is
    returned in the working register.
STATUS      equ     0x03            ; define SFRs
            org     0x0000          ; reset vector
            goto    START
            org     0x0005
START       ........
            movlw   0x15            ; pass the first number
            movwf   0x30
            movlw   0x09            ; pass the second number
            movwf   0x31
            call    multiply        ; call the subroutine
            ........
             movlw  0x05            ; pass the first number
            movwf   0x30
            movlw   0x04            ; pass the second number
            movwf   0x31
            call    multiply        ; call the subroutine
            ........
DONE        goto    DONE            ; endless loop
```


Example - Continued

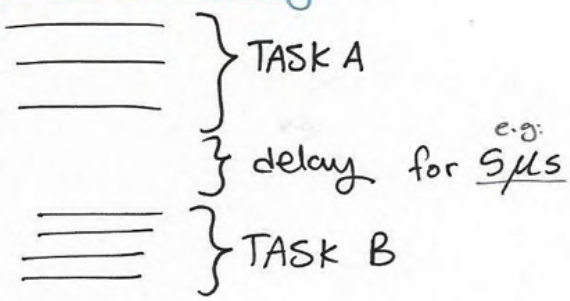
```
multiply    clrw
Repeat      addwf  0x30, 0   ; repeated  addition
            decfsz 0x31, 1   ; counter
            goto   repeat
            return

            end
```

Generating Time Delays

- In many applications, it is required to delay the execution of some block of code; i.e. a time delay!
- In most microcontrollers this can be done by
  - Software
  - Hardware (Timers)
- To generate time delay using software, *let the microcontroller execute non useful instructions* for certain number of times!
- If we know the clock frequency and the cycles to execute each instruction we can generate different delays

$$Delay = \#cycles \text{ x } clock\ cycle\ time$$
$$= \#\text{cycles x } 4 / F_{osc}$$

Software delay:



e.g:

Time = # of cycles × (cycle time)
= # of (NOP) × (4 / Fosc)

e.g:- let Fosc = 4 MHz
∴ Tcycle = 4/4M = 1 μsec.

∴ # of Nop = 5μ / 1μ = 5 NOP instructions!

* What if the delay = 500μs ?

500 μsec —(Fosc=4M)→ 500 NOP → inefficient!!

→ Solution: Use loops

e.g:

# of cycles			
1	MOVLW	N	} overhead (these are executed one time only)
1	MOVWF	COUNTER	
1(2)	DELAY DECFSZ	COUNTER, F	} doesn't affect the STATUS Register. (safe to use).
2 or (0) if skipped (at the last cycle).	GOTO	DELAY	

when Skip happens

N = ??

Delay = (# cycles) * Tcy
= (#iterations × # cycles / iteration + overhead) × Tcy

for the previous program:-

$Delay = (\underbrace{overhead}_{\text{exec. 1 time only}} + \underbrace{(N-1)\times 3}_{\text{2 form(GOTO) \& 1 from DECFSZ}} + \underbrace{2}_{\text{due to the last loop}}) \times T_{cy}$

since the skip doesn't happen until the last loop (for (N-1) loops).

$500\mu = (2 + (N-1)\times 3 + 2) \times 1\mu$

$N = 166.3\bar{3}$! can't store non-integer value

$\therefore N = 166$ or 167

$Delay = (2 + 166\times 3 + 2)\times 1\mu = 502\,\mu$ sec. for $N = 167$ $\quad > 500\mu$

or $Delay = (2 + 165\times 3 + 2)\times 1\mu = 499\,\mu$ sec. for $N = 166$ $\quad < 500\mu$

* if the delay = 1000 μsec. ?

$1000\mu = (2 + [N-1)\times 3 + 2] \times 1\mu$

$N = 333$ but counter is an 8-bit register! (can store up to 255 only)!

Solution: Do a delay within a delay !:D

→ cont.

```
    MOVLW  N
    MOVWF  COUNTER
Delay NOP
    NOP
or  DECFSZ COUNTER,F
    GOTO   Delay
```

delay = (overhead + # ite × # cycles/ite) * Tcy

$= (2 + \underbrace{5\times(N-1) + 4}_{N \text{ cycles}}) \times 1\mu\text{sec}$

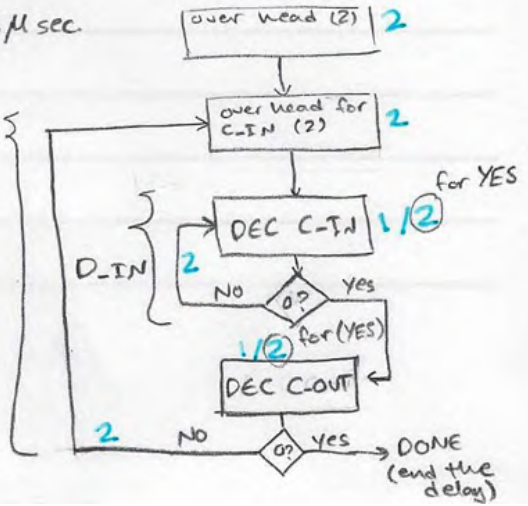
N cycles
→ 5 cycles/ite for (N-1) cycles
↳ 4 cycles in the (last) cycle

* For large delays:

```
    MOVLW  D'255'   ] overhead (for the outer loop).
    MOVWF  C_OUT
D_OUT MOVLW D'255'  ] overhead for the inner loop.
    MOVWF  C_IN
or D_IN DECFSZ C_IN
    GOTO   D_IN
or  DECFSZ C_OUT
    GOTO   D_OUT
```

⇒ if f = 0.5 MHz, find the delay?

* $T_{cy} = \frac{4}{0.5M} = 8\mu$ sec.



$$Delay = \Big[\underbrace{1+1}_{\text{overhead of the outer loop}} + \Big[\underbrace{3\times(255-1)}_{N_{IN}} + \underbrace{2(1)}_{\text{one cycle only when C-IN=0}} + \underbrace{1+1}_{\text{over head of the inner loop.}} + \underbrace{1+2}_{\text{GOTO}\ \&\ \text{DECFSZ C-OUT}}\Big] \times \underbrace{(255-1)}_{N_{OUT}} + \cdots$$

(1) decfsz + (2) GOTO] for $\frac{N-1}{254}$ cycles. / 3

(2) decfsz + (0) GOTO (skipped).

since the inner loop is repeated 255 times one of them will end a skip.

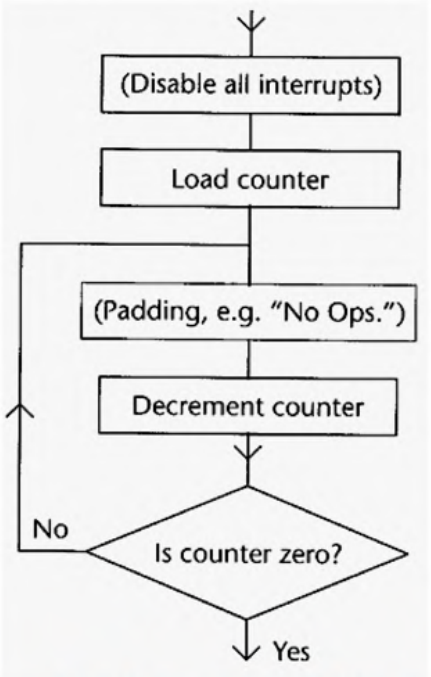
$$\rightarrow + \Big[3\times(\underbrace{255-1}_{N_{IN}}) + 2(1) + 1+1+2+\underbrace{0}_{\text{GOTO (skipped)}}\Big] \times \underbrace{(1)}_{\text{the last cycle of the outer loop.}} \Big] * T_{cycle}$$

$$= \Big[\underbrace{2}_{\text{over head}} + \underbrace{(3\times254+2+5)\times254}_{(N-1)\text{ loops with the condition = false}} + \underbrace{(3\times254+2+4)\times1}_{\text{last loop with condition = true}}\Big] \times 8\mu$$

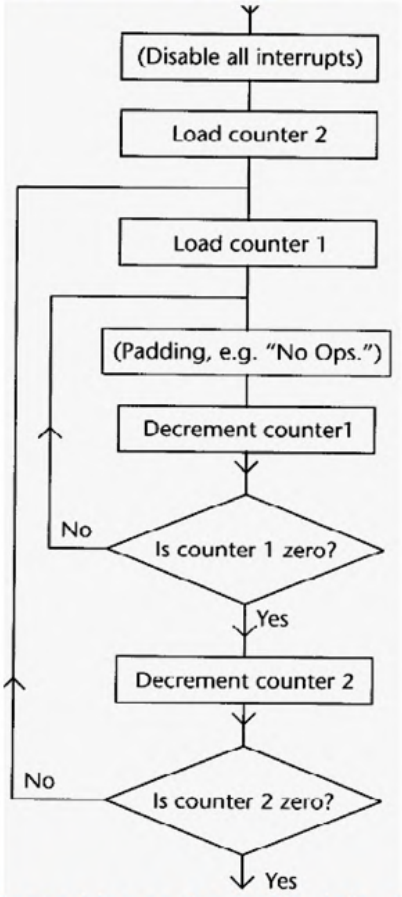
$$= 1568768\,\mu sec. \cong 1.5688\ sec.$$

Generating Time Delays

- Structure of Delay Loops



One loop for small delays



Nested loops for large delays

Generating Time Delays

- Example1: Determine the time required to execute the following code. Assume the clock frequency is 800KHz.

```
        1   movlw   D'200' ; initialize counter
        1   movwf   COUNTER
del     1   nop             ; main loop for delay
        1   nop
      1/2   decfsz  COUNTER, F
        2   goto    del
```

overhead

- What if this code to be used as a subroutine??!!

we add +(2+2)*Tcy
↳ for call & Return.

Generating Time Delays

- Example2: analyze the following subroutine and show how it can be used to generate a delay of 10 ms exactly including the call instruction. Assume 4 MHz clock frequency

```
TenMs   nop             ; beginning of subroutine
        movlw   D'13'
        movwf   COUNTH
        movlw   D'250'
        movwf   COUNTL
Ten1    decfsz  COUNTL , F   ; inner loop
        goto    Ten1
        decfsz  COUNTH , F   ; outer loop
        goto    Ten1
        return
```

this instr. was added to achieve exactly 10 msec delay

- executed 13 times
- first one COUNTL = 250 initially.
- the (11) time with COUNTL = 0 initially (without skipping the 2nd GOTO)
- the last instr. is the same as the last 11 but with skipping GOTO.

Example 1: $F_{osc} = 800\,KHz \rightarrow T_{cy} = \frac{4}{800k} = 5\mu sec.$

delay = # cycles × T_{cy}

= (over head + # iter × # cycles/iter) × T_{cy}.

= $[2 + (1+1+1+2)\times 199 + (1+1+2+0)\times 1] \times 5\mu$

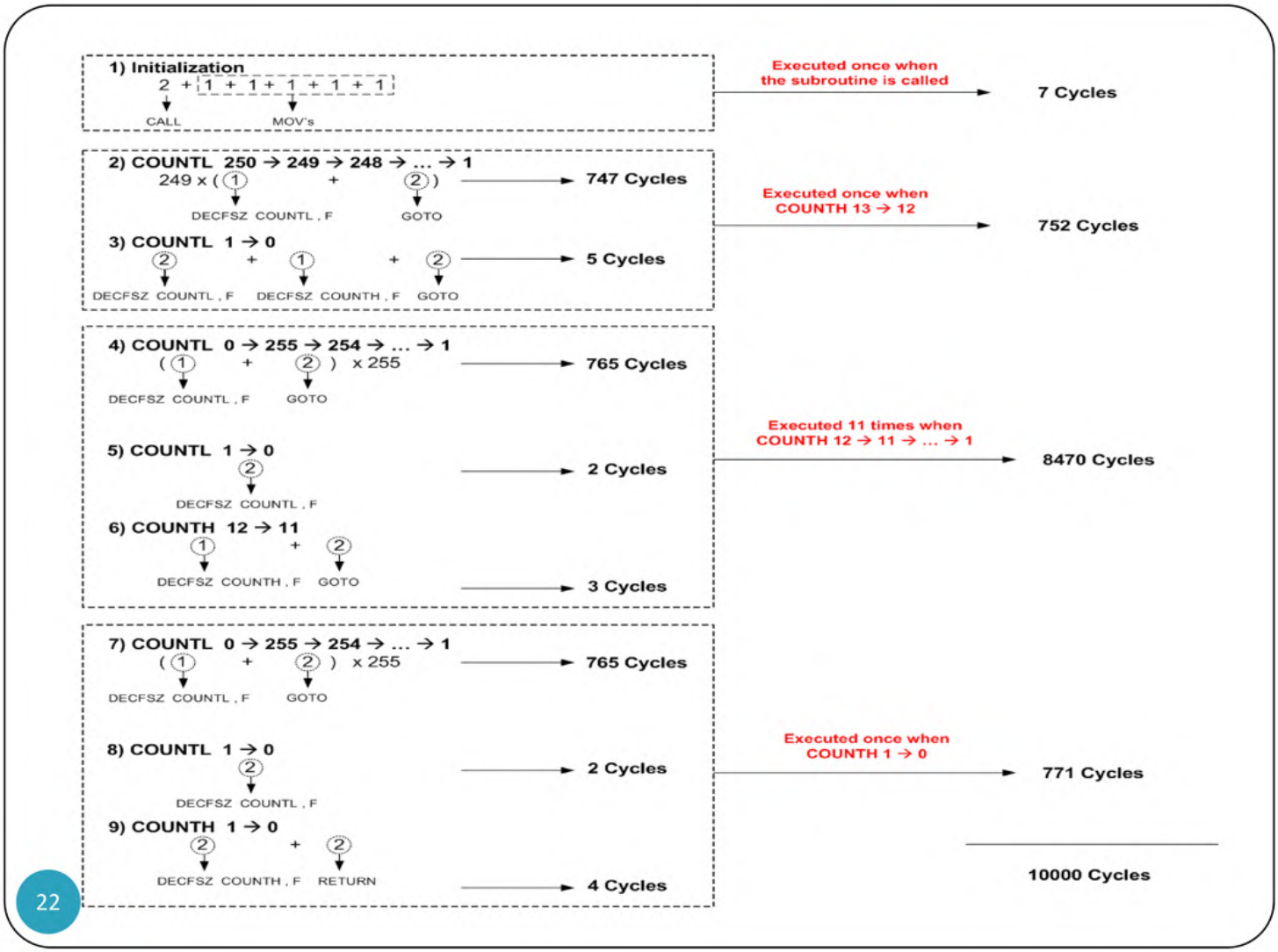
= 5.005 msec.

Example 2:

delay = $\left(\underbrace{2}_{\text{CALL}} + \underbrace{2}_{\text{RETURN}} + \underbrace{5}_{\text{over head}} + \underbrace{(3\times 249 + 5\times 1)\times 1}_{\text{the cycle with COUNTL = 250 initially.}} + \underbrace{(3\times 255 + 5\times 1)\times 11}_{\text{12 cycles with COUNTL = 0 initially.}} + (3\times 255 + 4\times 1)\times 1\right) \times 1\mu$

(13) — outer cycle (1+11+1)

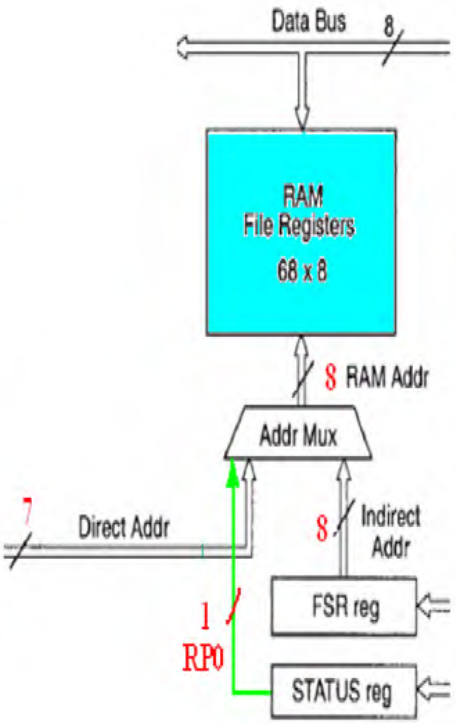
= 10000 μsec.



Working with Data

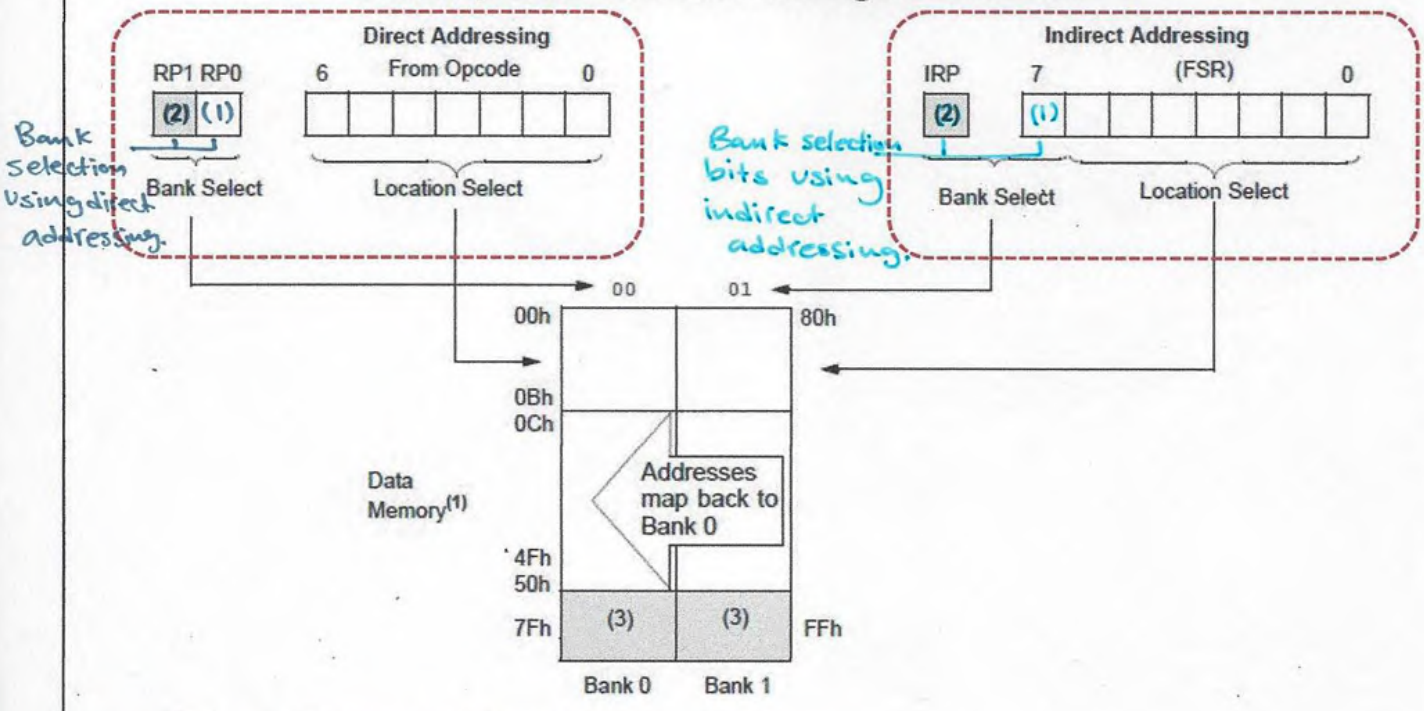
Indirect Addressing

- Direct addressing is capable of accessing single bytes of data
- Working with list of values using direct addressing is inconvenient since the address is part of the instruction
- Instead, we can use indirect addressing where
  - The File Select Register FSR register acts as a pointer to data location.
  - The FSR can be incremented or decremented to change the address
- The value stored in FSR is used to address the memory whenever the INDF (0x00) register is accessed in an instruction
- This forces the CPU to use the FSR register to address memory



Working with Data

Direct/Indirect Addressing in 16F84A



Note 1: For memory map detail, see Figure 2-2.
2: Maintain as clear for upward compatibility with future products.
3: Not implemented.

Why do we use indirect addressing?

e.g: write the code to clear the locations 0x11 through 0x20

~~CLRF 0x11~~
~~CLRF 0x12~~
⋮
~~CLRF 0x20~~
not efficient

⟹

```
     MOVLW   D'16'
     MOVWF   COUNTER
Loop CLRF    [ ?? ]   → 0x10  → we need a dynamic address!
     DECFSZ  COUNTER
     GOTO    Loop
```

→ Thus, we use the special function register "file Select Register" & modify the commands as follows:

```
     MOVLW   D'16'
     MOVWF   COUNTER
     MOVLW   0x11
     MOVWF   FSR        ← in the "*.inc" file.
Loop CLRF    INDF       → because CLRF FSR will clear the FSR Reg. itself
     INCF    FSR,F
     DECFSZ  COUNTER,F
     GOTO    Loop
```

INDF:- indirect file reg. ; address = 0x00 or 0x80 (in both banks). it has no physical location

فقط رقم مجوز حتى يفهّم الcontroller إنه ياخذ الaddress من الFSR بدل الinstruction.

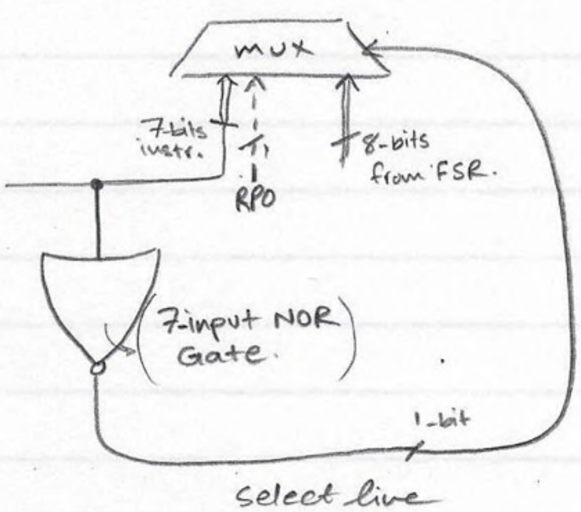
*INCF INDF,F will increase the file reg. with the same address as the one stored in FSR.

- **Hardware approach:** (select line of the mux).

how to identify the type of addressing?

based on the lower 7-bits of the instruction

|—————|— f —| (bits 6 … 0)



Select line

1: indirect → when all the bits are zeros (since INDF's address is 0x00)
0: direct

Working with Data

- **Example:** a program to add the values found locations 0x10 through 0x1F and store the result in 0x20

#include "P16F84A.inc"

```
STATUS      equ     0x03        ; define SFRs
FSR         equ     0x04
INDF        equ     0x00
RESULT      equ     0x20
N           equ     D'15'       Since the first step will be moving f=0x10 (addr.) to the working register.
COUNTER     equ     0x21
            org     0x0000      ; reset vector
            goto    START
            org     0x0005
START       movlw   N           ; initialize counter
            movwf   COUNTER
            movlw   0x11        ; initialize FSR as a pointer
            movwf   FSR
            movf    0x10 , W    ; get 1st number in W
LOOP        addwf   INDF , W    ; add using indirect addressing
            incf    FSR , F     ; point to next location
            decfsz  COUNTER , F ; decrement counter
            goto    LOOP
            movwf   RESULT
DONE        goto    DONE
            end
```


Look-up tables:

e.g:

0x10 → 0	0
1	1
2	4
3	9
4	16
5	25

→ this look-up table is used as an alternative to a subroutine that can be used to compute the square of numbers (0-5)

→ but how to load & access this table?

→ The naive way:

Loading the table:

```
MOVLW   D'0'
MOVWF   0x10
MOVLW   D'1'
MOVWF   0x11
MOVLW   D'4'
MOVWF   0x12
MOVLW   D'9'
MOVWF   0x13
:       :
MOVLW   D'25'
MOVWF   0x15        → = 12 instructions
```

accessing the table:

```
MOVLW   0x10        ; → address of the 1st item
MOVWF   FSR         ; in the look-up table
MOVLW   D'2'        ; the input value = 2 ↳ # of entry.
ADDWF   FSR,F
MOVF    INDF,W      → = 5 instructions.
```

∴ for n-instr. (entry) table, we need:

[(2n) + 5] instructions (memory inefficient.)

↳ to load → to access.

* we don't use this approach!

* In PIC16' series the previous method is inefficient since we have limited data memory & program memory.

Thus, look-up tables in PIC 16 are defined as subroutines!

Load the table

```
TABLE   ADDWF   PCL (F)      <- this has to be F!
                             -> program counter (Lower 8 bits).
        RETLW   D'0'         (program counter will point to the 1st instr.
        RETLW   D'1'          automatically after calling the subroutine,
        RETLW   D'4'          thus it must be modified with respect to
        RETLW   D'9'          the entry we want to access.)
        RETLW   D'16'
        RETLW   D'25'   => (n+1) instr. = 6 instr.
```

** PCL isn't an actual (physical) register; it a reserved place (0x02), which whenever used it will notify the Microcontroller that the user wants to adjust the program counter.

access the table

```
        MOVLW   D'2'
        CALL    TABLE   => 2 instr.
```

total # of instr. = (n+1) + 2
(Load, access)

→ another advantage of this method is that the table is stored in the program memory

** When ever you modify the PCL, it will take 2 cycles (Thus any instr. that changes program flow will take 2 cycles).

Working with Data

Look-up Tables

- A look-up table is a block of data held in the program memory that the program accesses and uses
- The **movlw** instruction allows us to embed one byte within the instruction and use ! How about a look-up table ?
- In PIC, look-up tables are defined as a subroutine inside which is a group of **retlw** instructions
- The **retlw** instruction is similar to the return instruction; however, it has one operand which is an 8-bit literal that is placed in **W** after the subroutine returns
- In order to choose one of the **retlw** instructions in the look-up table, the program counter is modified to point to the desired instruction by changing the value in the **PCL** register **(0x02)**
- The **PCL** register holds the lower 8 bits of the program counter

Working with Data

- **Example:** a subroutine to implement a look-up table for the squares for number 0 through 5. To compute the square, place the number in W before calling the subroutine SQR_TABLE

```
SQR_TABLE   addwf   PCL , 1 ; modify the PCL to point the
                            ; required instruction
            retlw   D'0'    ;  square value of 0
            retlw   D'1'    ;  square value of 1
            retlw   D'4'    ;  square value of 2
            retlw   D'9'    ;  square value of 3
            retlw   D'16'   ;  square value of 4
            retlw   D'25'   ;  square value of 5

; Remember that the PC always points to the instruction to be
                            executed
```


Summary

- Building complex programs requires putting down it requirements and design
- Programs tends to execute instructions sequentially unless branching or subroutines are used
- A subroutine is piece of code that can be called from anywhere inside the program
- A simple way to generate time delays is to use delay loops

Working with Time: Interrupts, Counters, and Timers

Chapter 6

Dr. Iyad Jafar

Outline

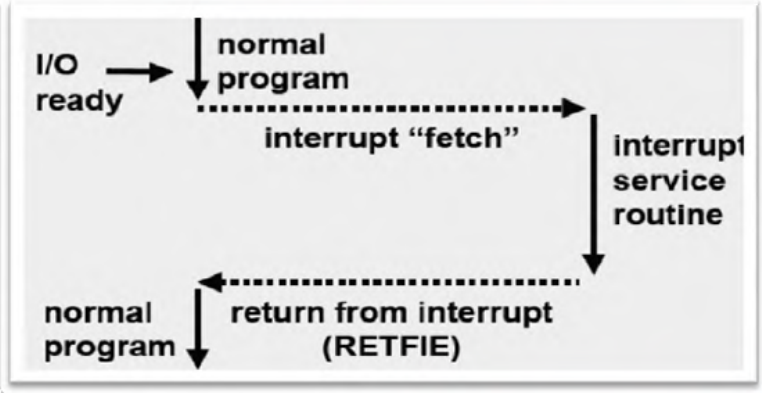
- Introduction
- Interrupts
- Timer/Counter
- Watchdog Timer
- Sleep Mode
- Summary

Introduction

- Microcontroller should be able to deal with time
  - Respond in a timely manner to external events
  - Measure time between events
  - Generate time-based activity
- For this purpose, microcontrollers are usually provided with timers and support interrupts

Interrupts

- An interrupt is an event that causes the microcontroller to halt the normal flow of the program and execute another program called the interrupt service routine



- Interrupts can be thought of as *hardware-initiated subroutine calls*
- Usually, interrupts are generated by I/O devices such as timers or external devices

Interrupts vs Polling

- Advantages
  - Immediate response to I/O service request
  - Normal execution continues until it is known that I/O service is needed
- Disadvantages
  - Coding complexity for interrupt service routines
  - Extra hardware needed
  - Processor's interrupt system I/O device must generate an interrupt request

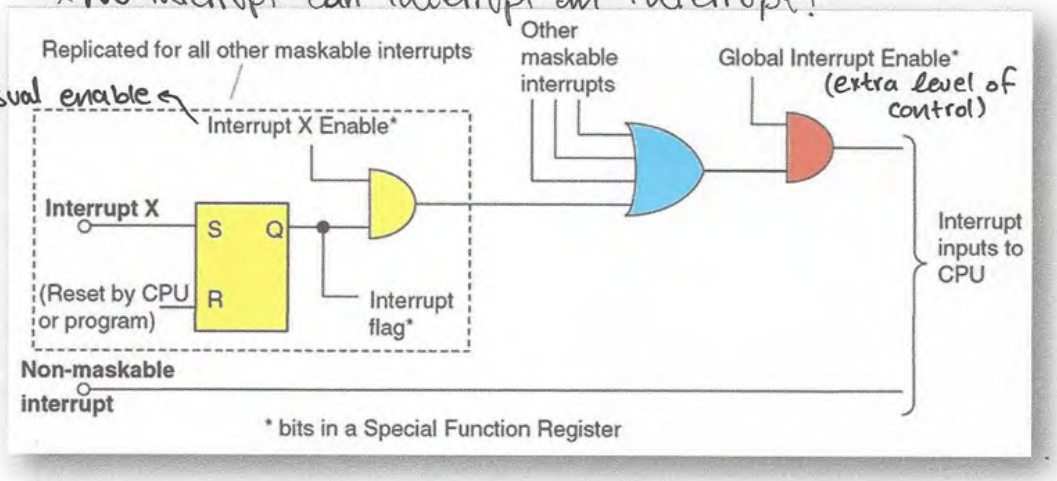
General Hardware Structure for Interrupts

- Interrupts sources can be *external and internal*
- Two types of interrupts : *maskable* and *non-maskable*
  - Maskable can be enabled/disabled by setting/clearing some bits
  - Non-maskable interrupts can not be disabled and they always interrupt the CPU

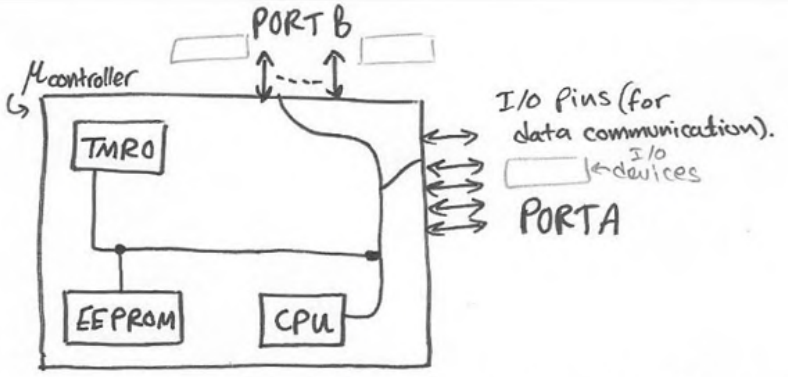
* In PIC16F84A all interrupts are maskable.

- Usually, each interrupt has a flag (a bit) that is set whenever the interrupt occurs

* NO interrupt can interrupt an interrupt!



*when an ISR is executed, other Interrupts are disabled, but must be stored so that they won't be lost. (that's why we use an SR-Latch)



I/O devices communicate with CPU Asynchronously. So, how to inform the CPU that a device wants to send or recieve data ?

1) Polling.
2) Interrupts.

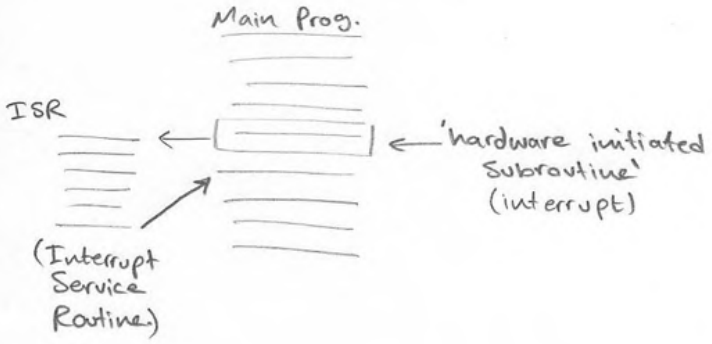
1. **Polling:** to allocate a time slot within the program to check wether a device is asking to use CPU resources or not.

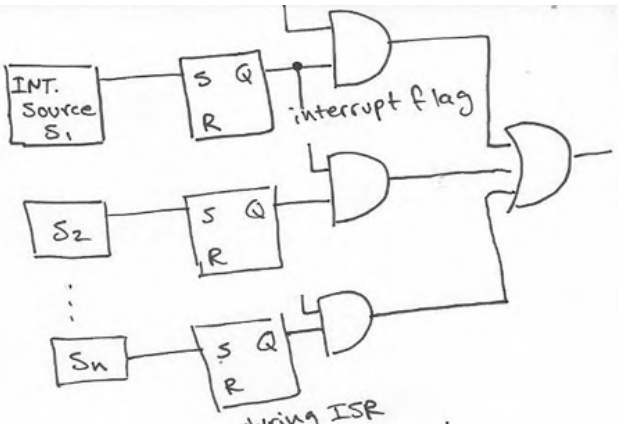
* Polling disadvantages:
  1. it wastes CPU resources.
  2. it causes a delay in service; because the device must wait untill the CPU reaches the polling interval to be served.

2. **Interrupts:**

it's a way to allert the CPU that some external device wants to use its resources.

* disadvantages:
  1. you need to add aditional hardware.
  2. you need to design your program & CPU operation to accomodate this process. (coding complexity).





*When an interrupt occurs, the CPU finishes the execution of the current instruction & then starts the execution of the ISR.

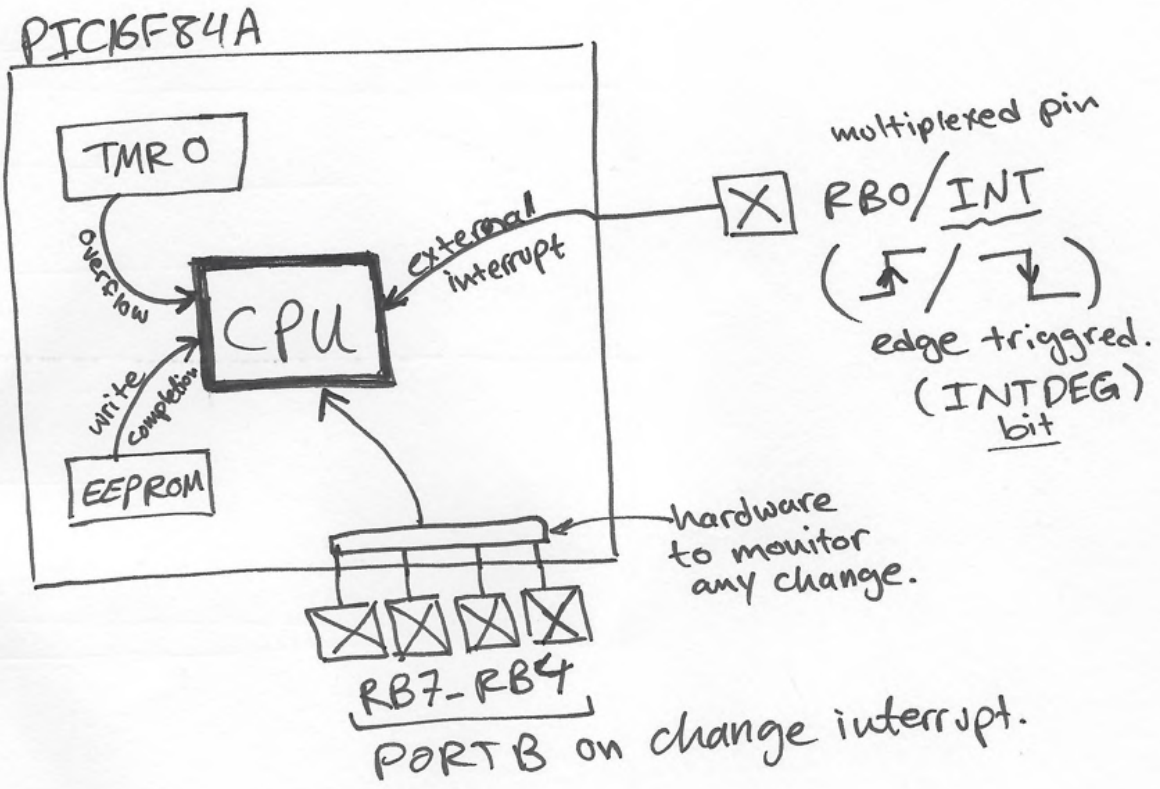
* When the CPU has responded to an interrupt it's necessary to clear the interrupt flag.

if S=1, R=0 : set interrupt flag = 1
but during ISR the GIE bit is disabled ∴ the request will be stored even if the source of interrupt has gone (S=0, R=0 : no change) will store the request.

The 16F84A Interrupt Structure

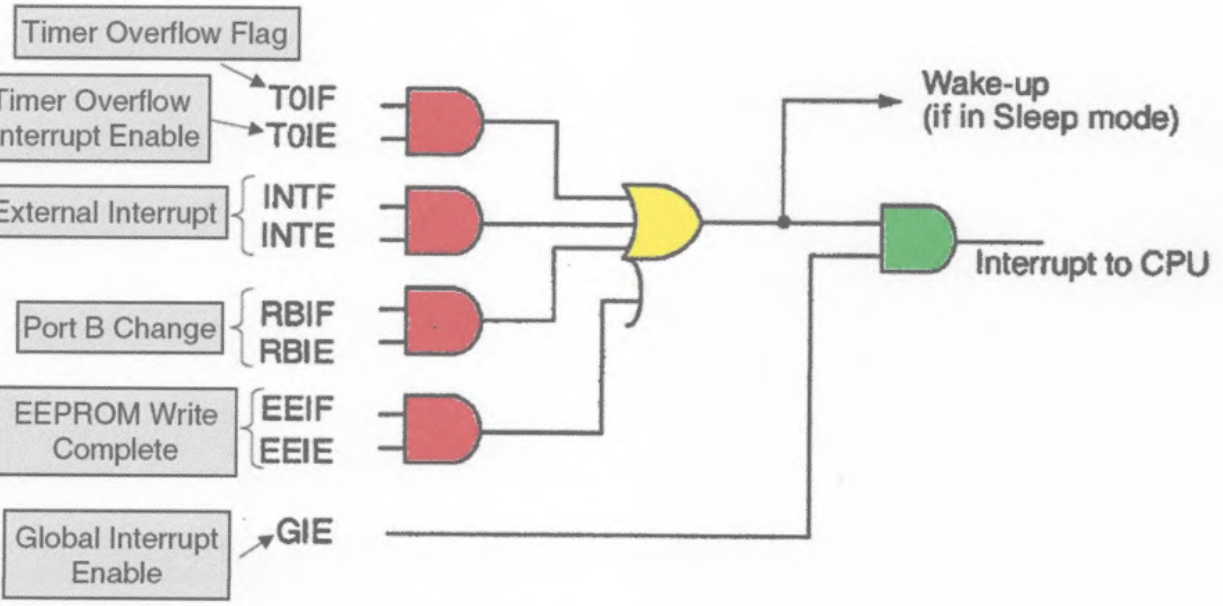
- **Sources of interrupts** (All of them are maskable).
  - ***External interrupt*** (enable: INTE , flag: INTF)
    - The only external interrupt input
    - The input is multiplexed with RB0 pin of port B
    - It is edge triggered (controled using INTEDG bit in the option register.)
  - ***Timer overflow interrupt*** (TOIE, TOIF)
    - It is an internal interrupt that occurs when the 8-bit timer overflows
  - ***Port B interrupt change*** (RBIE, RBIF)
    - An interrupt occurs when a change is detected on any of the upper 4 bits of port B
  - ***EEPROM write complete interrupt*** (EEIE, EEIF)

Sources of interrupt:



The 16F84A Interrupt Structure

- **Interrupt Hardware Structure**



writing on EEPROM takes time (>1 cycle). So CPU completes the execution of the program while the writing process is in progress.

No non-maskable interrupts in 16F84A

The 16F84A Interrupt Structure

- **The INTCON Register**

at CPU's power up All interrupts are initially disabled.

INTCON REGISTER (ADDRESS 0Bh, 8Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

why don't care (x)?

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

bit 6 **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

*EEIF : in EECON register

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

The 16F84A Interrupt Structure

- **The Option Register (81H) – interrupt related bit**

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

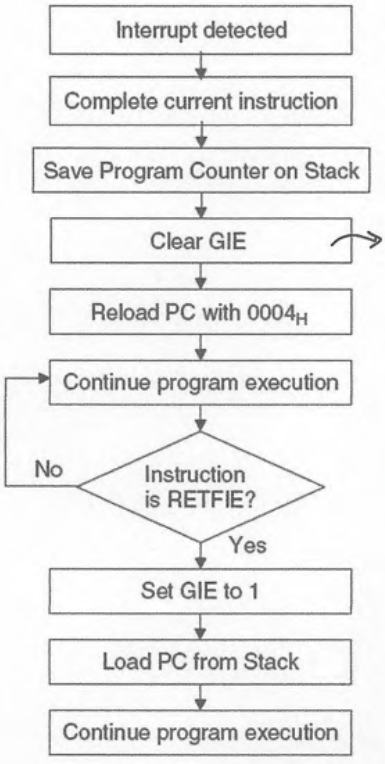
bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Select the transition type on input RB0/INT that will cause an interrupt

The 16F84A Interrupt Structure

- **Interrupt Operation**



so that no interrupt can interrupt this current interrupt.

* RETFIE ≡ RETURN + BSF INTCON, GIE
even though it's equivelant to these 2 instr., they can't be used as an alteranative, why?
1) if BSF was executed before RETURN, an int might occur & we will have an interrupt within an interrupt!
2) if we put BSF after RETURN →

The 16F84A Interrupt Structure

- **How to use interrupts ?**

1. Start the interrupt service routine at 0x0004
2. Clear the flag of the used interrupt in the INTCON register (if it is not cleared on reset, e.g. RBIF)
   =(don't care) on reset
3. Enable the corresponding interrupt by setting its bit in INTCON register
4. Enable global interrupts by setting the GIE bit
5. End the interrupt subroutine with **RETFIE** instruction to resume program execution

4.5 (ip) *Interrupt flags are not cleared automatically, so it's the programmer's responsibility to clear them within the ISR.

The 16F84A Interrupt Structure

- **Example**

Write a PIC16F84 program that continuously adds the content of memory location 0x0A until an external interrupt is observed on RB0. In this case the result is stored in location 0x10 and the working register is cleared. The interrupt should be configured on the arrival of rising edge.

The 16F84A Interrupt Structure

- **Example**

```
                #include   p16F84A.inc          ; include the definition file for 16F84A
                org        0x0000              ; reset vector
                goto       START
                org        0x0004              ; define the ISR
                goto       ISR
                org        0x0006               ; Program starts here
START           bsf        STATUS , RP0        ; select bank 1 (OPTION register).
                bsf        INTCON , INTE       ; enable external interrupt on RB0/INT
                bsf        OPTION_REG , INTEDG ; select to interrupt on rising edge
                bsf        INTCON , GIE        ; enable global interrupts
                bcf        STATUS , RP0        ; select bank 0
                molw       0x00                ; clear W
ADD             addwf      0x0A , 0            ; add the contents of 0x0A to W
                goto       ADD                 ; keep adding until an interrupt occurs

                org        0x00BC              ; location of ISR
ISR             movwf      0x10                ; on interrupt store the accumulated result
                clrw                           ; clear working register
                bcf        INTCON , INTF       ; clear the interrupt flag
                retfie                         ; return from the ISR
                end
```

no need to do it, it's 1 by default on reset.

Context Saving

(this is also applicable to other subroutines)

- **What if the main program is to preserve the W register and interrupt uses it?**
  - **Save it temporarily in memory at the beginning of the ISR**

    MOVWF TEMP ; push
  - **Restore the value at the end of ISR**

    MOVF TEMP, W ; pop

- **What if we want to preserve some memory location such as the STATUS register on interrupt?**
  - **Save it temporarily in memory at the beginning of the ISR**

    SWAPF STATUS,0 ; push ← W

    MOVWF TEMP
  - **Restore the value at the end of ISR**

    SWAP TEMP, 0 ; pop ← W

    MOVWF STATUS

move the content (after swapping the nibbles) without affecting STATUS reg.

The 16F84A Interrupt Structure

- **Multiple Interrupts**
  - Note that there is only one interrupt vector for all types of interrupts
  - In other words, regardless of the interrupt type, the microcontroller will start executing from location 0x0004 on any interrupt
  - How to determine the source of interrupt ?
  - Check the interrupt flag bits in the INTCON register at the beginning of the interrupt service routine to determine what source of the interrupt !

extreme case scenario (4 int. are enabled)

```
Interrupt_SR   btfsc intcon,0   ;test RBIF
               goto  portb_int  ;Port B Change routine
               btfsc intcon,1   ;test INTF
               goto  ext_int    ;external interrupt routine
               btfsc intcon,2   ;test T0IF
               goto  timer_int  ;timer overflow routine
               btfsc eecon1,4   ;test EEPROM write complete flag
               goto  eeprom_int ;EEPROM write complete routine
```

check next page

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

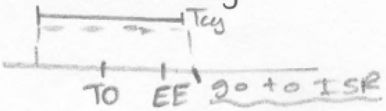
→ cont.

Interrupt_SR

check the flags to determine the source of the interrupt.

(the order of these bit tests determines the priority of execution when multiple events occur simultanuously.

e.g:



Port B_int

BCF flag

RETFIE better the GOTO label (2cy) then RETFIE + (2cy)

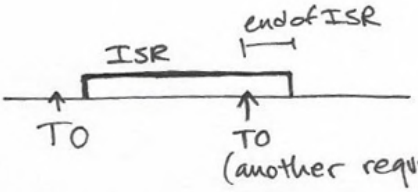
eeprom_int

(saving the Wreg & File reg. the int. contents, ...

BCF

RETFIE

scenario #2: (multiple requests from the same source).



* if we clear the flag at the end of ISR, the 2nd request won't be served!

Solution:- clear the flag at the beginning

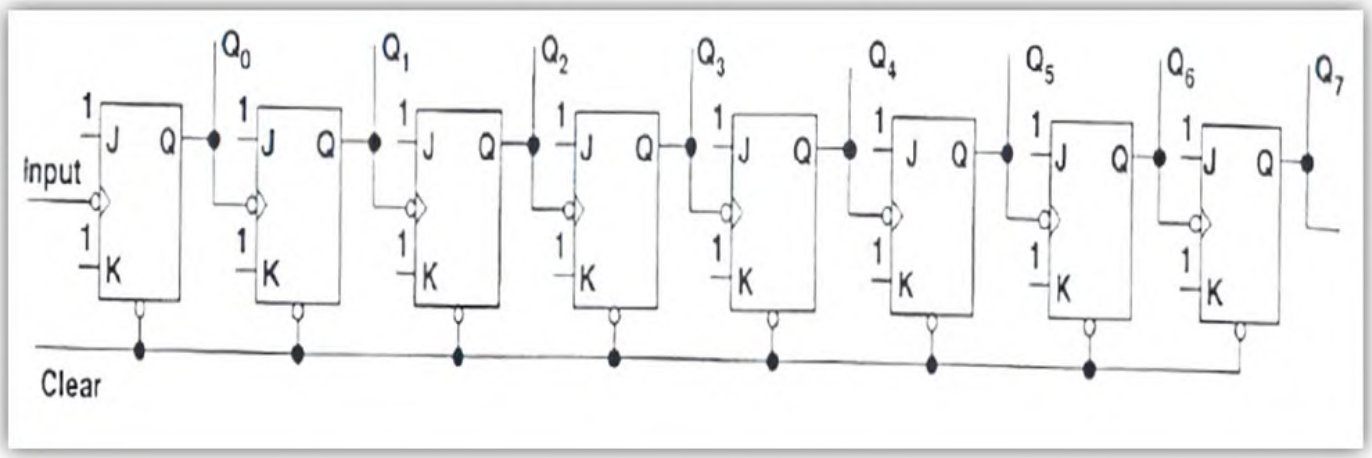
→ if a 3rd request occurs it will be lost.

↳ one solution might be optimizing the ISR so that it takes less time to be executed.

or (hardware sol.): use shift register. (to store multiple requests).

Counters and Timers

- Digital counters can be built with flip-flops. They can count up or down, reset, or loaded with initial value
- When the most significant bit changes from 1 to 0 , this indicates an overflow. This signal can be used to interrupt the microcontroller
- *If the counter operates using a clock with known frequency we can use it as a timer*



Counters and Timers

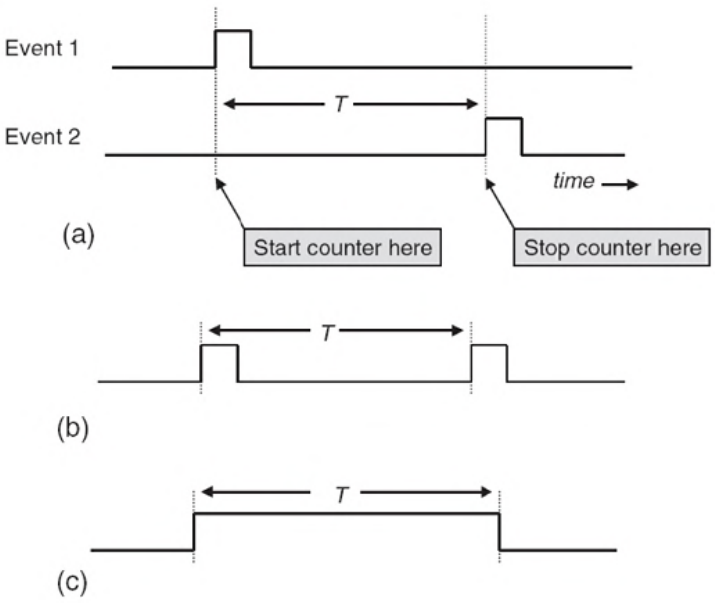
- **Timer applications**

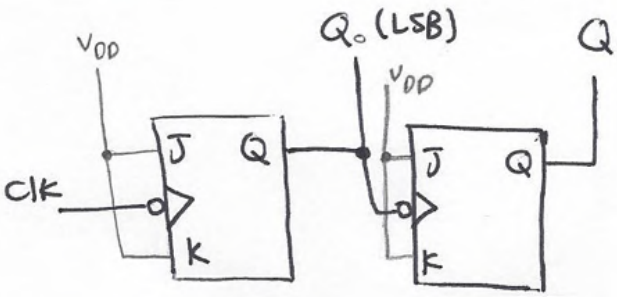
(a) Measure the time between two events

(b) Measure the time between two pulses

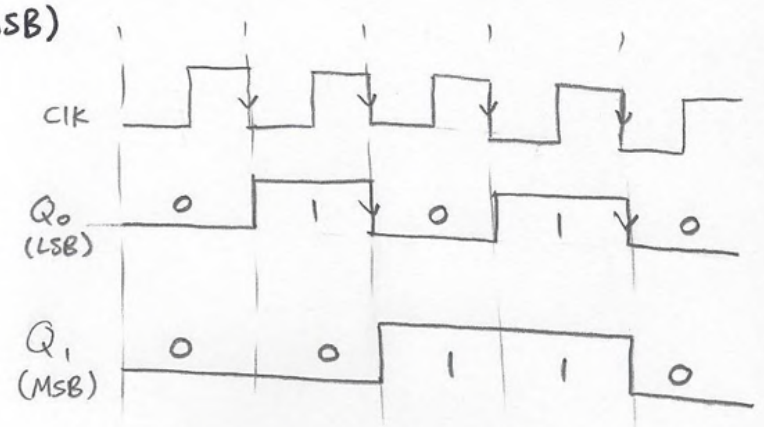
(c) Measure a pulse duration

Use **polling** or **interrupts**





2-bit Asynchronous counter
↳(since they don't have the same clk)



(toggle at each falling edge).

* Q_0 acts as a clk to Q_1.

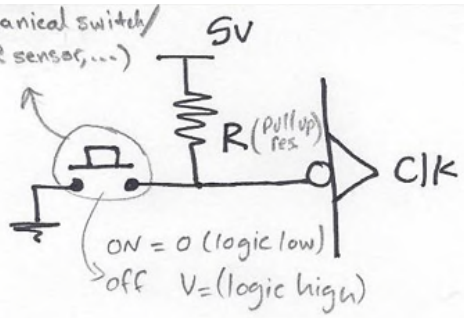
"Modulo-4 counter"
- upward counter

$00 \rightarrow 01 \rightarrow 10 \rightarrow 11 \rightarrow 00 \rightarrow \ldots$
(overflow)

* (clk) signal sources:-

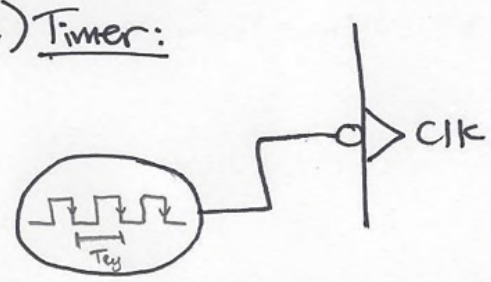
1) counter (counting some event).
2) Timer (counts time).

1) (mechanical switch/ IR sensor, ...)



→ it counts the number of times the switch is closed.

2) Timer:

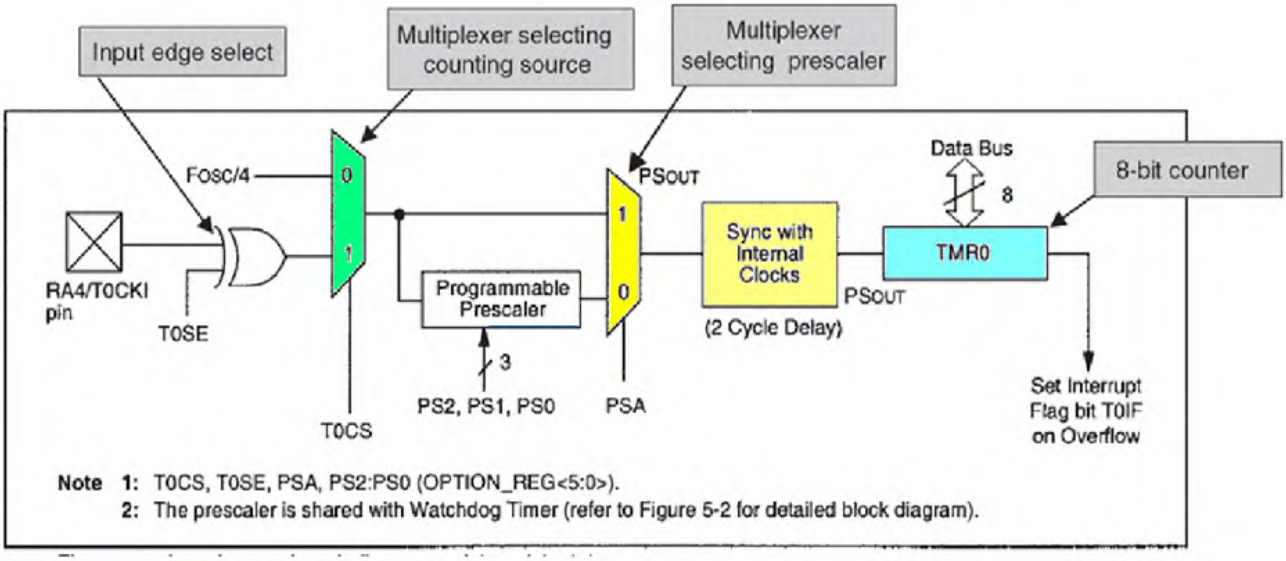


* to count time we count the # of cycles of a known Tcy.

→ Time = #of cycles * Tcy
(count value).

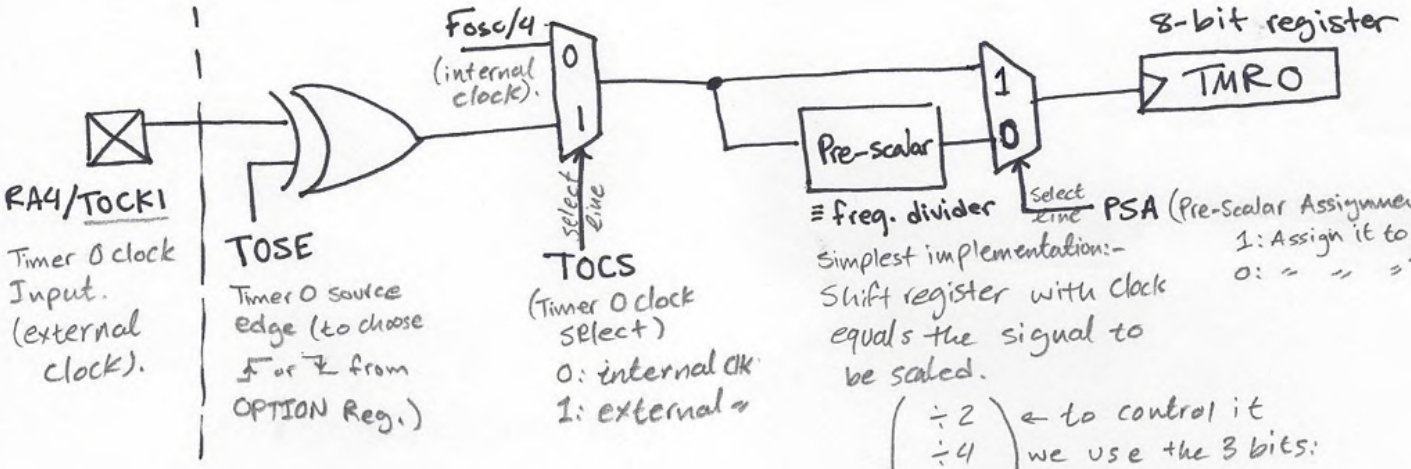
* How do CPU communicate with counters?
by polling or interrupts.

The 16F84A Timer 0 Module



- 8-bit counter , memory address 0x01
- Configurable counter using the OPTION register (0x81)
- Two sources for the timer clock : instruction cycle clock (Fosc/4) or RA4/T0CKI
- The programmable prescaler is shared with the Watchdog Timer WDT
- The value of frequency division is determined by PS2, PS1, and PS0 bits in the OPTION register

TMR0 module:



→ For an 8-bit counter & (Fosc/4) as a clock signal :

$$\text{Time} = \frac{4}{\text{Fosc}} * \text{count}$$

$$= T_{cy} * 256$$

let Fosc = 4MHz

∴ Time = 256 μsec. !

if we want to extend this time you have 2 choices

1. change 'count' × (TMR0 is an 8-bit reg)
2. change Fosc × constant
3. multiply it by a constant (freq. division)

$$\text{Time}' = \frac{4}{\text{Fosc}} * \text{count} * P$$

P: Pre-scalar value.

The 16F84A Timer 0 Module

- **The Option Register – Timer related bits**

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

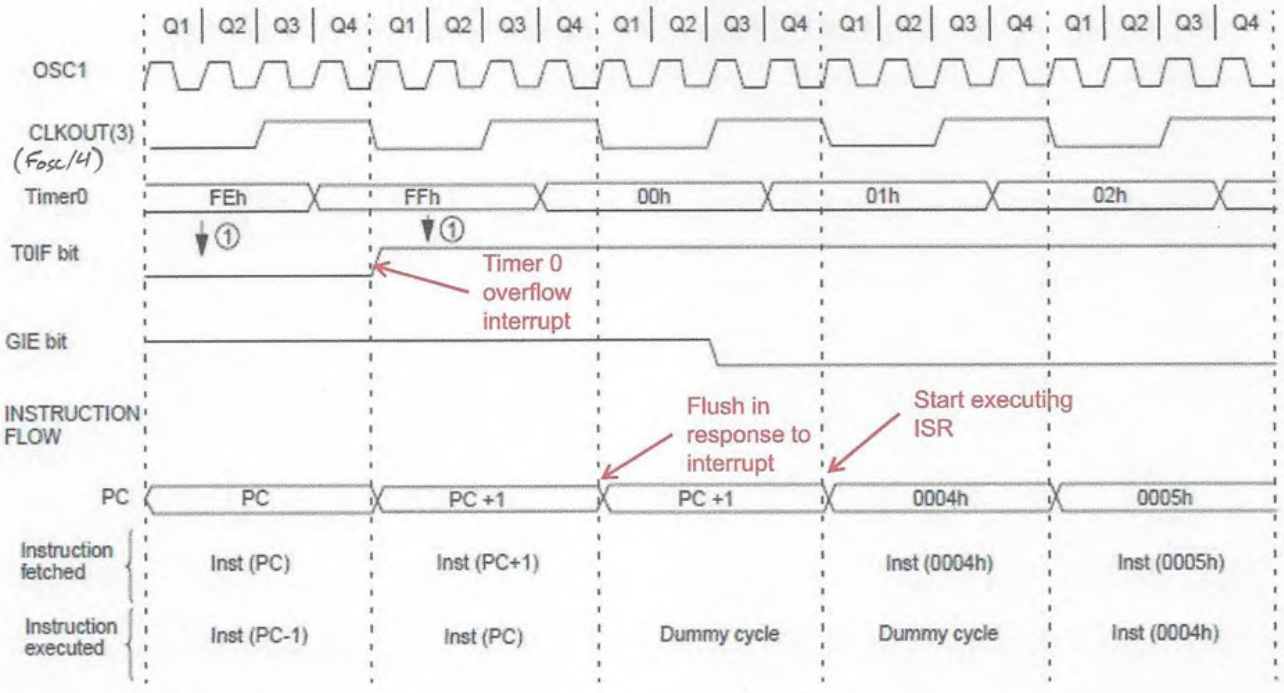
bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Since P=1 is satisfied when PSA = 1 (assigned to WDT).

The 16F84A Timer 0 Module

- **Timer Timing** Pre-scalar here = 1



Note 1: Interrupt flag bit T0IF is sampled here (every Q1).
2: Interrupt latency = 4TCY where TCY = instruction cycle time.
3: CLKOUT is available only in RC oscillator mode.

The 16F84A Timer 0 Module

- **Example: Write a program that generates a 5 ms delay using the TMR0 module without using interrupts. Assume the clock frequency is 800 KHz.**
  - Fosc = 800 KHz → the timer internal clock = Fosc/4 = 200 KHz → instruction cycle = 5 us → timer increment every 5 us
  - For these settings, the timer generates an interrupt after 256 * 5 us = 1280 us only ?!
  - How about changing the prescale factor ?
    - 256 x prescale x 5 us = 5 ms → prescale = 3.9 ~= 4
    - This will generate a delay of 4 x 256 x 5 us = 5.12 ms
  - What if we need more accurate delay !! We can play around with the count value (we don't have to start from 0 always)
    - N x prescale x 5 us = 5 ms --> N x prescale = 1000 → we can select the prescale 8 and the count N to be 125
    - We have to load TMR0 with 256 – 125 = 131 as initial value

22

* advantages of hardware delays against software delays:

1) Hardware delays are more accurate.
2) Using a software delay will force the Micro controller to put all its resources into processing some kind of loop (delay loop) which will block the execution of other instr.. while HW delays will allow CPU to execute usfel instr. while counting (until it overflows & interrupt CPU's execution).

example:-

Time = T_{cy} * count * Pre-scalar

5ms = 5μsec. * 256 * prescalar

Pre-scalar = 3.9 ≈ 4

actual time = 5μ x 256 x 4 = 5.12 msec > 5 msec.

how to make it = 5msec exactly ?

cont. example.

Time = 5μ (fixed) x (256) x (pres) ? (? (8 خيارات) التغيير محدود)

5 msec. = 5μ x (N x P)

N x P = 1000

if P = 2 → N = 500 X > 256!

if P = 4 → N = 250 < 256 ✓ → this represents # of counts TMR0 has to do.
TMR0 initial value = 256 - 250 = 6

P = 8 → N = 125 ✓ TMR0 init. = 256 - 125 = 131

P = 16 → N = 62.5 X N has to be integer.

∴ for P = 8

config. →

D'131								
	1	0	0	0	0	0	1	1	TMR0

X	X	0	0	0	0	1	1	0	OPTION reg. 0x02

T0CS (internal clk) Fosc/4; T0SE (-↓-); PSA (assigned to TMR0); PSA2:PSA0 [P = 8]

* without interrupt ∴ we have to use polling; keep checking if T0IF == 1 continuously.

The 16F84A Timer 0 Module

- **Example – cont'd**

→ TMR0 counts 5 msec.
This whole delay ≠ 5msec exactly
for exact time → [You have to consider the overhead.

**NOTE
* Whenever you write on TMR0 reg. Prescalar bits are cleared, so make sure you write on TMR0 before adjusting prescalar bits.

```
        #include p16f84A.inc
        org     0x0000
        goto    start
        org     0x0010
start   . . . .
        call    delay5
        . . . .
delay5  movlw   D'131'  ;         preload T0, it overflows after 125 counts
        movwf   TMR0
        bsf     STATUS, RP0       ;select memory bank 1
        movlw   B'00000010'       ;set up T0 for internal input, prescale by 8
        movwf   OPTION_REG
        bcf     STATUS, RP0       ;select bank 0
del1    btfss   intcon,T0IF       ;test for Timer Overflow flag
        goto    del1              ;loop if not set (no timer overflow)
        bcf     intcon,T0IF       ;clear Timer Overflow flag
        return
```

(settings config. — delay5 through bcf STATUS, RP0; Polling — btfss/goto; clear the flag then return)

(T0SE = 1 on reset (external clk), when it's changed to 0 (internal clk) it immediately starts counting)

clear Timer Overflow flag. this isn't ISR so why do I need to clear the flag? so that if I need to use the subroutine another time T0IF must be 0 initially.

$$Time = \frac{4}{F_{osc}} * P * count$$

$$= \frac{4}{4M} * \underbrace{128}_{P_{max}} * \underbrace{256}_{N_{max}} << 1\ sec!$$

how to create delay = 1sec?

* First, choose a basic time unit (e.g: 0.05 sec.) & configure TMR0 module to count 0.05 sec.
  → 0.05 sec. = 1μsec. × P × N
* Each time an overflow occurs increment some counter (inside the ISR).
* When counter value = 20 this means the delay = 1 sec.
  $\equiv (20 \times 0.05)$
  $=$
  # of interrupts (overflows) ↖ basic time unit

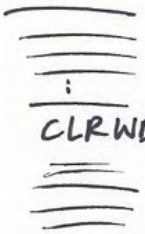
Watchdog Timer

- Special timer internal to the microcontroller that is continually counting up
- Can be used to reset the Microcontroller if a program fails or gets stuck
- If enabled and it overflows, the microcontroller is reset
- Properties
  - The WDT timer is enabled/disabled by the *WDTE* bit in the configuration word
  - It has its own internal *RC oscillator*
  - The nominal time-out period is *18 ms*
  - It can be extended through the prescaler bits in the *OPTION* register (up to 128x18 ms= 2.3 sec)
  - The WDT timer can be cleared by software using the *CLRWDT* instruction
- **How does the watchdog timer know if the program is stuck ???!!!**

* To prevent WDT from reseting Your program You have to clear it regularly (prevent overflow)

e.g: clear WDT every 18msec (without pre-scalar).

Main

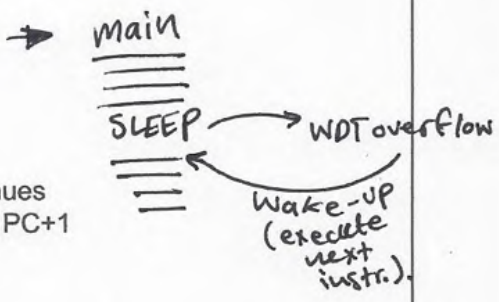


CLRWDT → at 18msec
↳ this is a nominal value
(it may change due to temp.)

When the CPU get stuck, it resets due to WDT overflow since CLRWDT won't be executed. While WDT will continue running because it has its own oscillator.

Sleep Mode

- An important way to save power!
- The microcontroller can be put in sleep mode by using the ***SLEEP*** instruction
- **Once in sleep mode, the microcontroller operation is almost suspended**
  - The oscillator is switched off
  - The WDT is cleared. If the WDT is enabled, it continues running
  - Program execution is suspended
  - All ports retain their current settings
  - $\overline{PD}$ and $\overline{TO}$ bits are cleared and set respectively
  - Power consumption falls to a negligible amount
- **To exit the sleep mode**
  - Interrupt occurs (even if GIE = 0) (↳(back to slide 8).) } Program continues execution from PC+1
  - WDT wake-up
  - External reset the MCLR pin } MC is reset !



Summary

- Microcontrollers can deal with time by using timers and interrupts
- Interrupts saves the microcontrollers computational power as they require its attention when they occur only
- Most interrupts are configurable
- Hardware timer can be used as a counter or a timer and it is very useful in measuring time

Parallel Ports, Power Supply, and the Clock Oscillator

Chapter 3

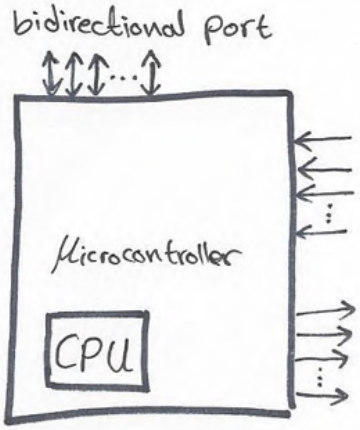
Dr. Iyad Jafar

Outline

- Why Do We Need Parallel Ports?
- Hardware Realization of Parallel Ports
- Interfacing to Parallel Ports
- The PIC 16F84A Parallel Ports
- The Power Supply
- The Clock Oscillator

Why Do We Need Parallel Ports?

- Almost any microcontroller needs to transfer digital data from/to external devices and for different purposes
  - Direct user interface – switches, LEDs, keypads, displays
  - Input measurement - from sensors, possibly through ADC
  - Output control information – control motors and actuators
  - Bulk data transfer – to other systems/subsystems
- Transfer could be serial or parallel !



* In general these data ports can be either digital or analog.

* In PIC16F84A, there are 2 data ports

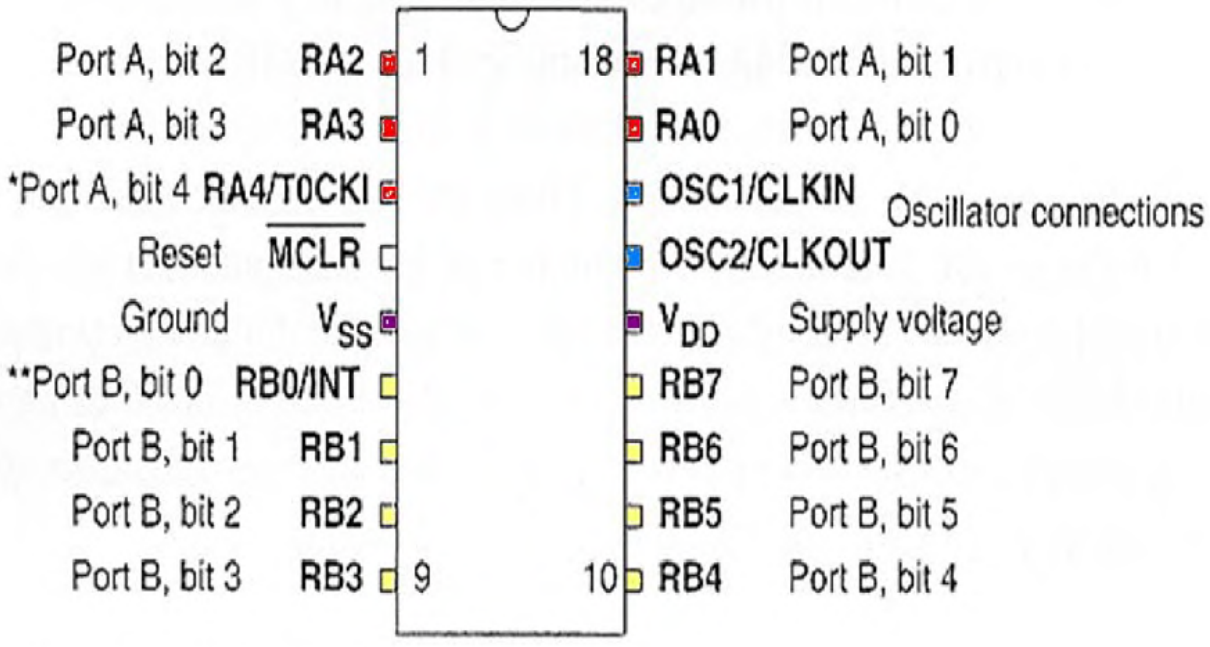
PORT A (5 bits)] both are digital & bidirectional.
PORT B (8 bits)]

* These ports are memory-mapped; they can be accessed as memory locations.

PORT A ⟶ at 0x05
PORT B ⟶ at 0x06

* TRISA (at 0x85) & TRISB (at 0x86) are SFRs to configure PORT A & B Pins as input or output.

The PIC 16F84 Parallel Ports



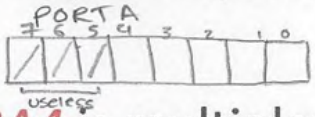
*also counter/timer clock input

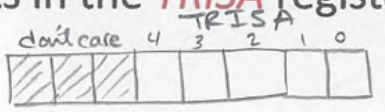
**also external interrupt input

The PIC 16F84 Parallel Ports

PORT A

- *5-bit general-purpose* bidirectional digital port (RA0-RA4)
- **Related registers**
  - Data from/to this port is stored in *PORTA* register *(0x05)*
  - Pins can be configured for input or output by setting or clearing corresponding bits in the *TRISA* register *(0x85)*





(we can configure each pin (bit) indivisually).
↳0: output
↳1: Input

- Pin *RA4* is multiplexed and can be used as the clock for the TIMER0 module

The PIC 16F84 Parallel Ports

PORT B

- *8-bit general-purpose* bidirectional digital port
- **Related registers**
  - Data from/to this port is stored in *PORTB* register *(0x06)*
  - Pins can be configured for input or output by setting or clearing corresponding bits in the *TRISB* register *(0x86)*
- **Other features**
  - Pin *RB0* is multiplexed with the external interrupt INT and has Schmitt trigger interface
  - Pins *RB4 – RB7* have a useful 'interrupt on change' facility

*Always remember to change RP0 before accessing TRISA or TRISB. (both in Bank 1)

The PIC 16F84 Parallel Ports

- **Example –** *configuring port B such that pins 0 to 2 are inputs, pins 3 to 4 outputs, and pins 5 to 7 are inputs*

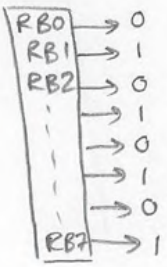
```
bsf     STATUS , RP0    ; select bank1
movlw   0xE7            ;  7 6 5 4 3 2 1 0
                        ;  1 1 1 0 0 1 1 1
movwf   TRISB           ; PORTB<7:5> input,
                        ; PORTB<4:3> output
                        ; PORTB<2:0> input
```


The PIC 16F84 Parallel Ports

TRIS B = 00000000 ← 10101010

- **Example –** *configuring PORTB as output and output value 0xAA*

```
bsf     STATUS , RP0    ; select bank1
clrf    TRISB           ; PORTB is output
movlw   0xAA
* bcf   STATUS , RP0    ; select bank0
movwf   PORTB           ; output data
```



- **Example –** *configuring PORTA as input, read it and store the value in 0x0D*

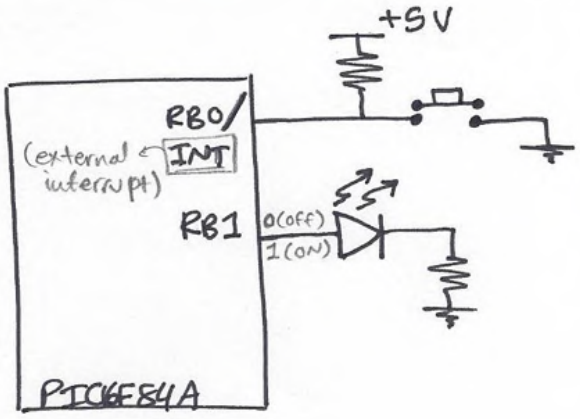
```
bsf     STATUS , RP0    ; select bank1
movlw   0xFF
movwf   TRISA           ; PORTA is input
bcf     STATUS , RP0    ; select bank0
movf    PORTA, W        ; read data
movwf   0x0D            ; save data
```

Example on Using I/O Ports

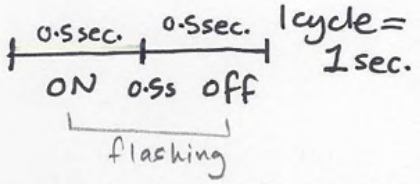
- **Example –** *on external interrupt (rising edge ON RB0), start flashing a LED connected to RB1 every 1 second approximately. If another interrupt occurs, stop flashing, and so on … assume 4MHz clock.*
- ***Requirements:***

1) *Configure RB0 as input and RB1 as output*
2) *Enable external interrupt (INTE) and global interrupts (GIE)*
3) *Write a 0.5 second delay routine*
4) *Keep track of the current status of flashing (on/off)*

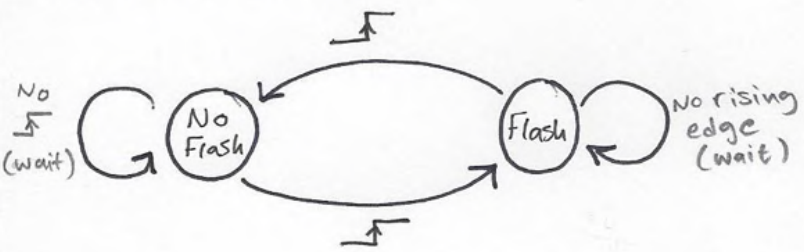
Slide 9:



* 1st, configure pins:
  RB0 → input
  RB1 → output.
* 2nd, enable external interrupt & GIE bit.
  INTE = 1
  GIE = 1
* write a software delay = 0.5 sec.



* State machine!



* 4th, you need 1 bit to save the state of the machine (out of two states {flash/no flash}).

Example on Using I/O Ports

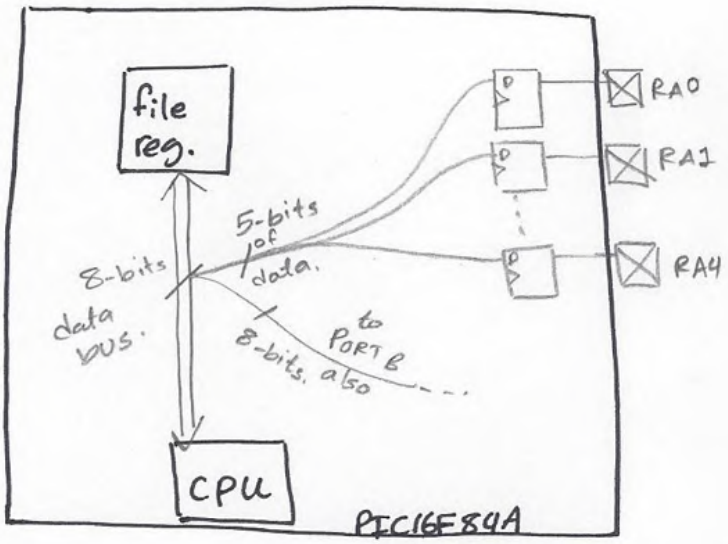
```
#include    "P16F84A.INC"
FLASH       EQU     0X20                ; STORE THE STATE OF FLASHING
COUNT1      EQU     0X21                ; COUNTER FOR DELAY LOOP
COUNT2      EQU     0X22                ; COUNTER FOR DELAY LOOP
            ORG     0X0000
            GOTO    START
            ORG     0X0004
            GOTO    ISR
;-------------------------------- MAIN PROGRAM ----------------------------------------
START       CLRF    FLASH               ; CLEAR FLASHING STATUS
            BSF     STATUS,RP0          ; SELECT BANK 1
            MOVLW   B'00000001'         ; CONFIGURE RB0 AS INPUT AND RB1 AS OUPUT
            MOVWF   TRISB
            BSF     OPTION_REG, INTEDG  ; SELECT RISING EDGE FOR EXTERNAL INTERRUPT
            BSF     INTCON , INTE       ; ENABLE EXTERNAL INTERRUPT
            BSF     INTCON , GIE        ; ENABLE GLOBAL INTERRUPT
            BCF     STATUS,RP0          ; SELECT BANK 0
            CLRF    PORTB               ; CLEAR PORTB; TURN OFF LED
WAIT        BTFSS   FLASH , 0           ; IF BIT 0 OF FLASH IS CLEAR THEN NO FLASHING
            GOTO    WAIT                ; WAIT UNTIL BIT 0 IS SET
            MOVLW   B'00000010'
            XORWF   PORTB , 1           ; COMPLEMENT RB1 TO FLASH
            CALL    DEL_p5sec
            GOTO    WAIT
```

Why XOR? this way i'll only modify bit #1. since (0)XOR(bn) = bn

Example on Using I/O Ports

```
;---------------- DELAY ROUTINE -----------------------
DEL_p5sec
            MOVLW   D'0'
            MOVWF   COUNT1
            MOVLW   D'244'
            MOVWF   COUNT2
LOOP        NOP
            NOP
            NOP
            NOP
            NOP
            DECFSZ  COUNT1 , 1
            GOTO    LOOP
            DECFSZ  COUNT2 , 1
            GOTO    LOOP                ; delay 0.500207 seconds
            RETURN

 ---------------- INTERRUPT SERVICE ROUTINE ----------------
ISR         MOVLW   0x01
            XORWF   FLASH , 1           ; COMPLEMENT THE STATUS
            BCF     INTCON , INTF       ; CLEAR THE INTF FLAG
            RETFIE

            END
```

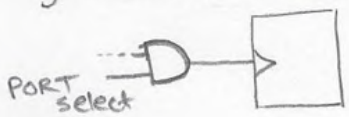
* PORTA, PORTB, TRISA & TRISB have addresses within the memory space, but they are not located (physically) inside the data memory.

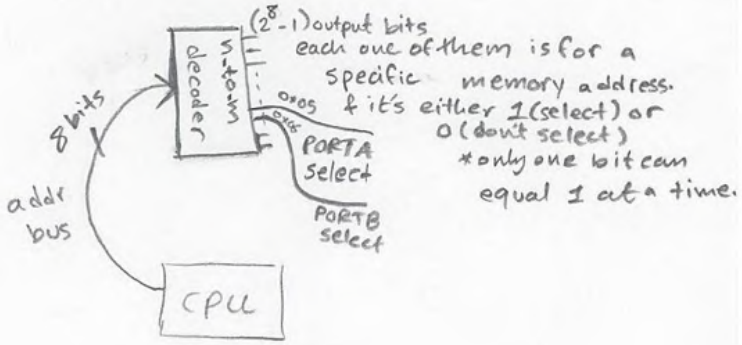


*how to choose this reg?

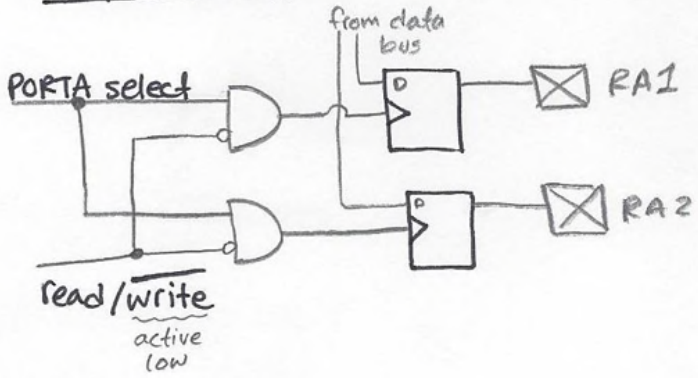
"كيف بدّي أفهم الـ mc. إنه كيف بدّه العينة من الـ data bus لـ PORT؟"

by adjusting the clk of the D-flip flop

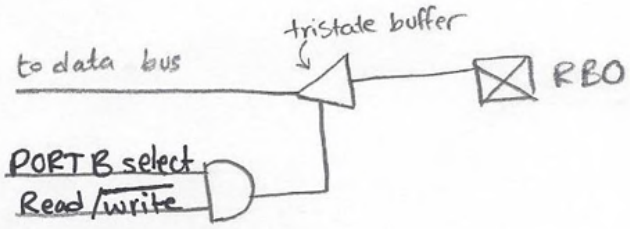




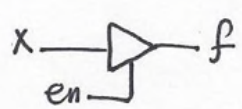
- Output PORTs: (e.g: RA1 & RA2)



- Input Ports: (e.g: RB0).

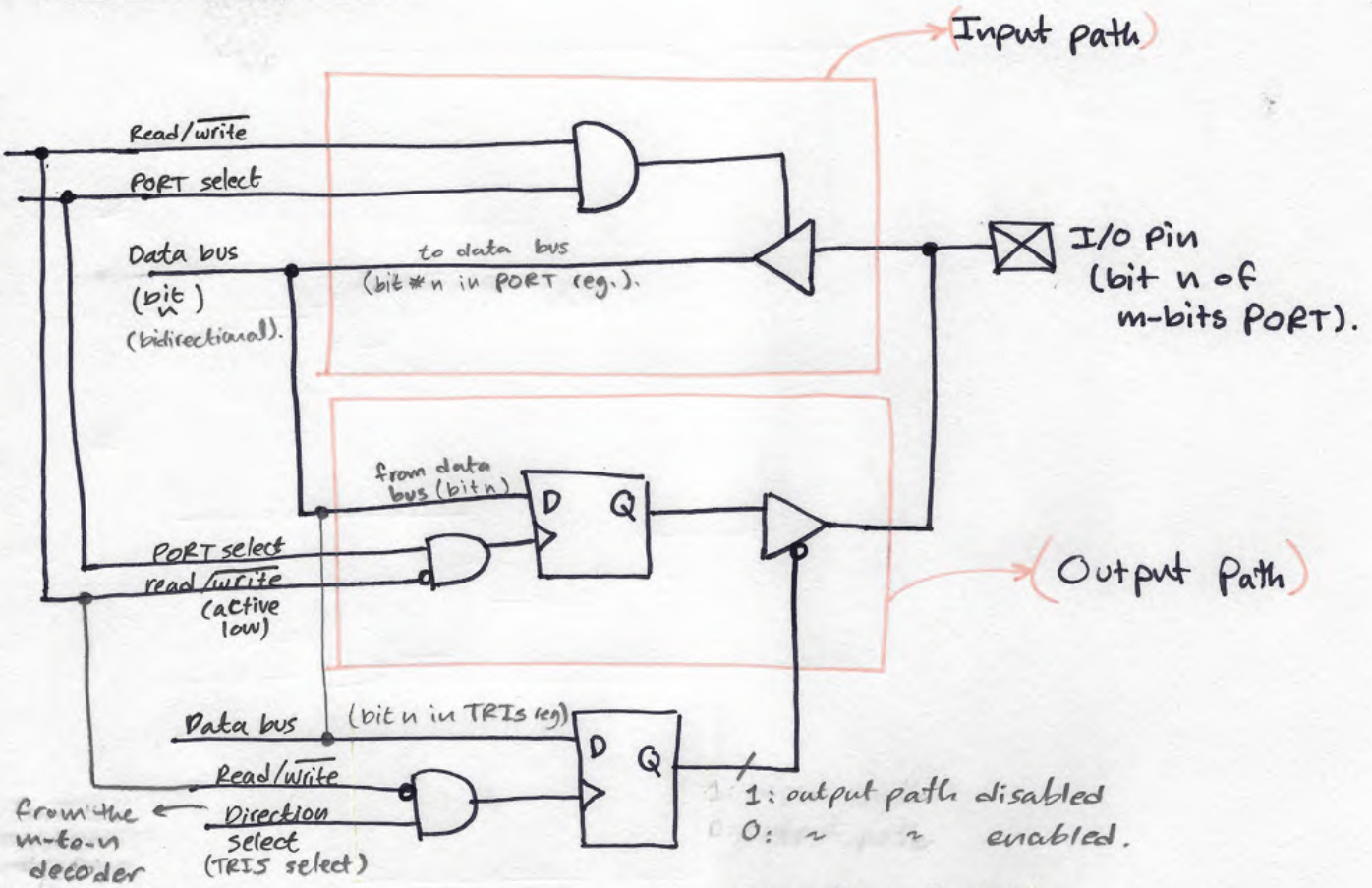


Remember: (TRI-state buffer).



en	X	f
0	0	high impedance (open CKT).
0	1	
1	0	0 } = X (as a buffer).
1	1	1

• Bidirectional PORT:

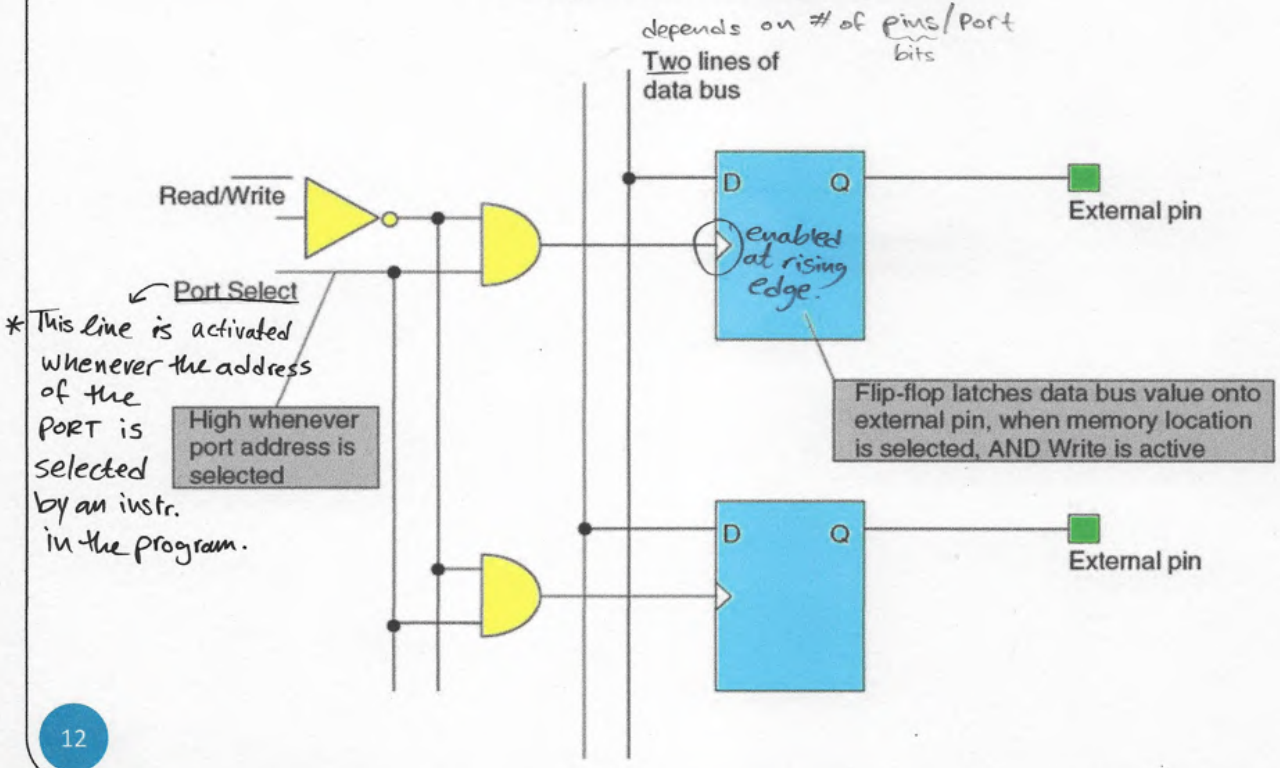


*Note that input (Read) path is always enabled because when the user outputs some value, he must be able to read it too. (we used this in flash-not flash example).

Direction (path) select.

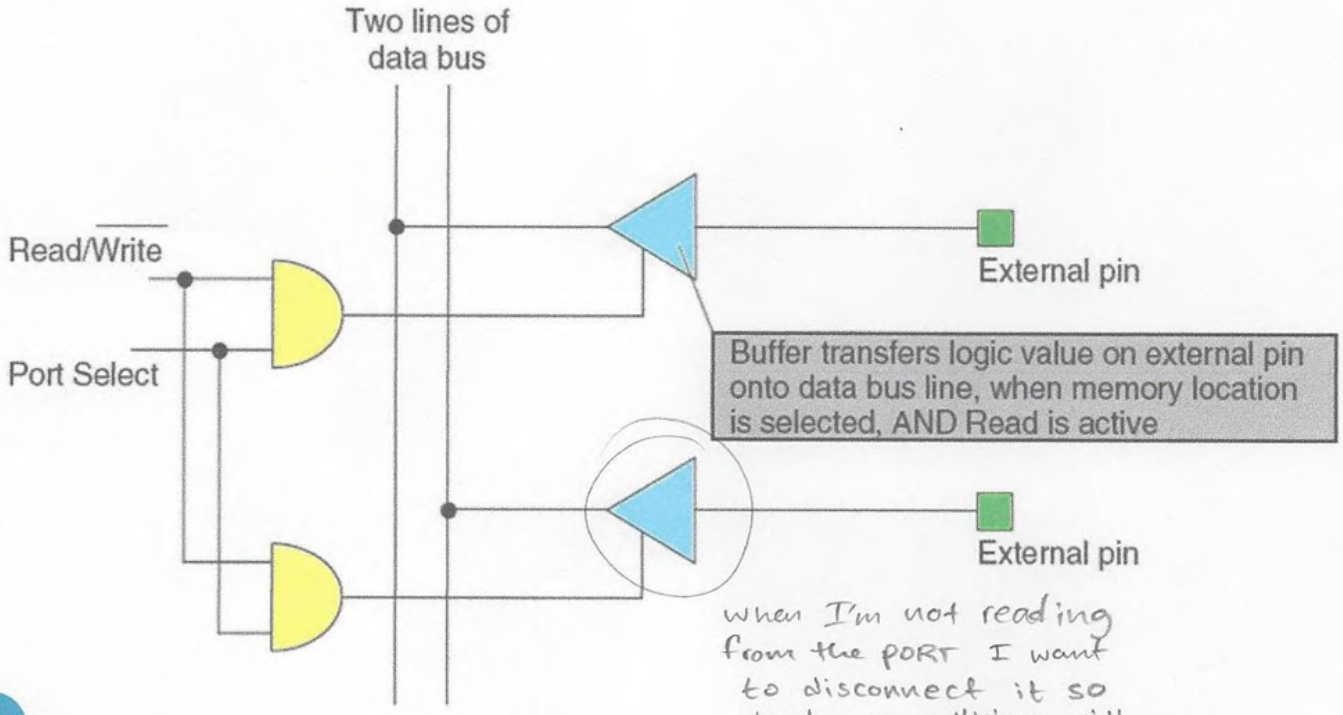
Hardware Realization of Parallel Ports

Output Parallel Port



Hardware Realization of Parallel Ports

Input Parallel Port

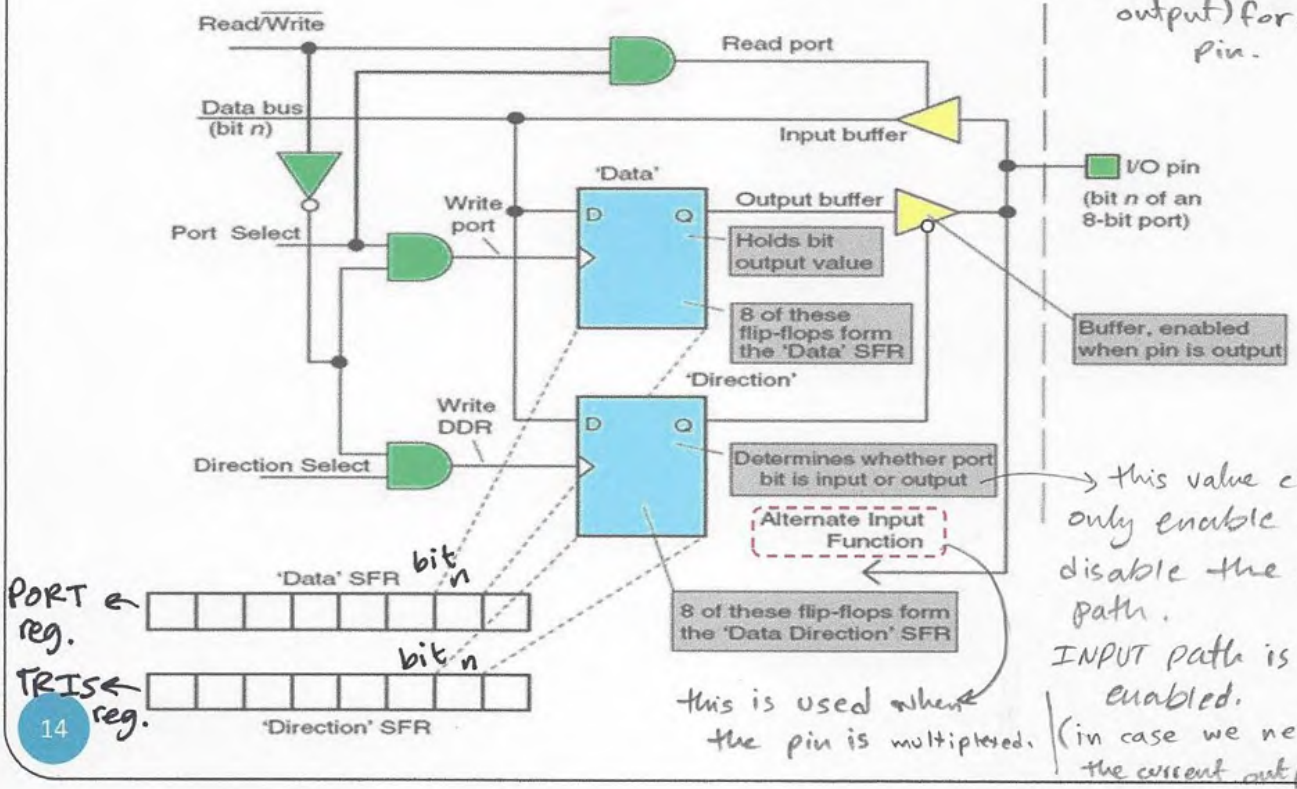


when I'm not reading from the PORT I want to disconnect it so that no collision will occur at the data bus.

Hardware Realization of Parallel Ports

Bidirectional Parallel Port

TRIS reg. bits are used to determine the direction (input/output) for each pin.



this value can only enable or disable the output path.

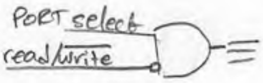
INPUT path is always enabled. (in case we need to read the current output value).

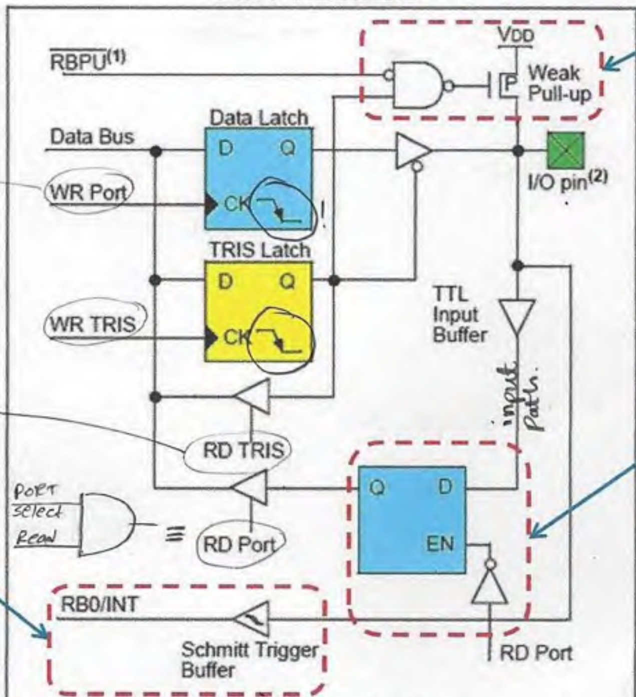
this is used when the pin is multiplexed.

Hardware Realization of Parallel Ports

PORT B

PINS RB3:RB0



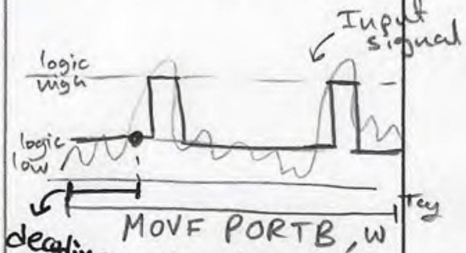


Configurable pull-up resistors using RBPU bit in the OPTION register

enabled when I want to read TRIS B value.

Multiplexed input

Latches input data whenever the port is read



وظيفة الـ latch كانه تغيير

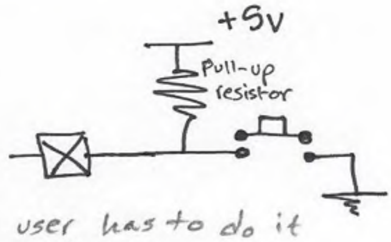
This latch will keep changing the input value (enabled) as long as you're not reading the input, when you activate (read) bit the latch will be disconnected & will store the last input value.

Note 1: TRISB = '1' enables weak pull-up (if $\overline{RBPU}$ = '0' in the OPTION_REG register).

2: I/O pins have diode protection to VDD and Vss.

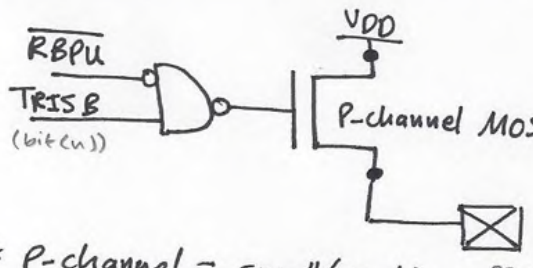
Pull up: it's an option available for PORT B pins.

Without internal Pullup:



the user has to do it externally.

internal pull-up: (PIC is self-contained :D)



*0 at P-channel ≡ small (weak) resistance

*1 at P-channel = high resistance (open CKT)

TRIS B	$\overline{RBPU}$	STATE = ($\overline{\text{TRISB and RBPU}}$)
0	0	1 (OFF)
0	1	1 (off)
1	0	0 (ON) active.
1	1	1 (off)

PIN is config. as input

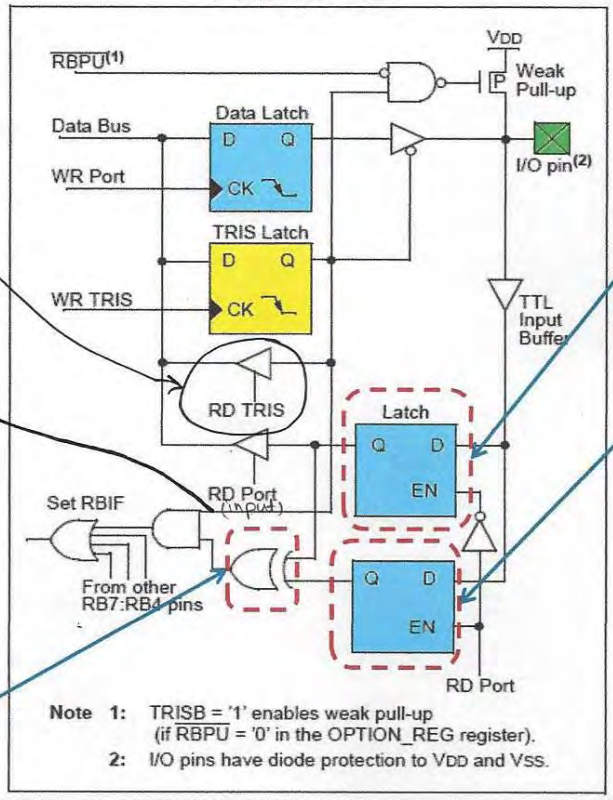
∴ RBPU = 1 (enabled)

* note that if you enable the weak pull-up resistor it will be activate it for all input pins. (can't be config. indivisually) since $\overline{RBPU}$ is only 1 bit.

Hardware Realization of Parallel Ports

PORT B

PINS RB7:RB4



This path is used when the user wants to read the TRIS reg.

This represents TRIS value; the pin has to be configured as input to be able to generate an interrupt.

Clearing the RBIF bit ? (next)

Lathes data on port read

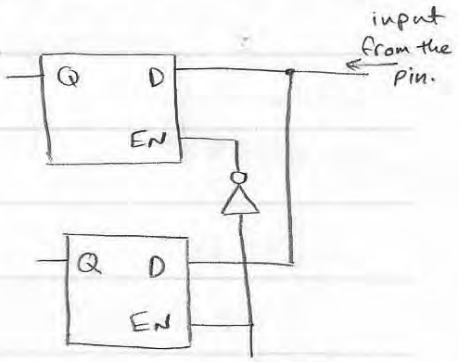
Holds previous latched data (for PORT B change interrupt).

→ This value is continuously compared with the current pin value (XOR) if any change happens an interrupt will occur.

Compares previous and present port input values

Enable signals for the 2 latches are complement (only one is enabled at a time).

RD.PORT	Latch 1 EN	Latch 2 EN	
1 (while reading the PORT)	0 (disabled)	1	→ Latch 1 holds the last input value
0	1 (enabled)	0	→ Latch 2 holds the last read value



→ ∴ to change the content of Latch 2, you have to read the PORT! (as long as you're not reading the Port Latch 2 won't change its value RD.PORT = 0, latch is disabled)

∴ use MOVF PORTB, 1 (or 0) to initialize the 2 latches with equal values. ↳ RD = 1, Latch 2 is enabled

remember that RBIF is (don't care) at startup so it has to be cleared automatically.

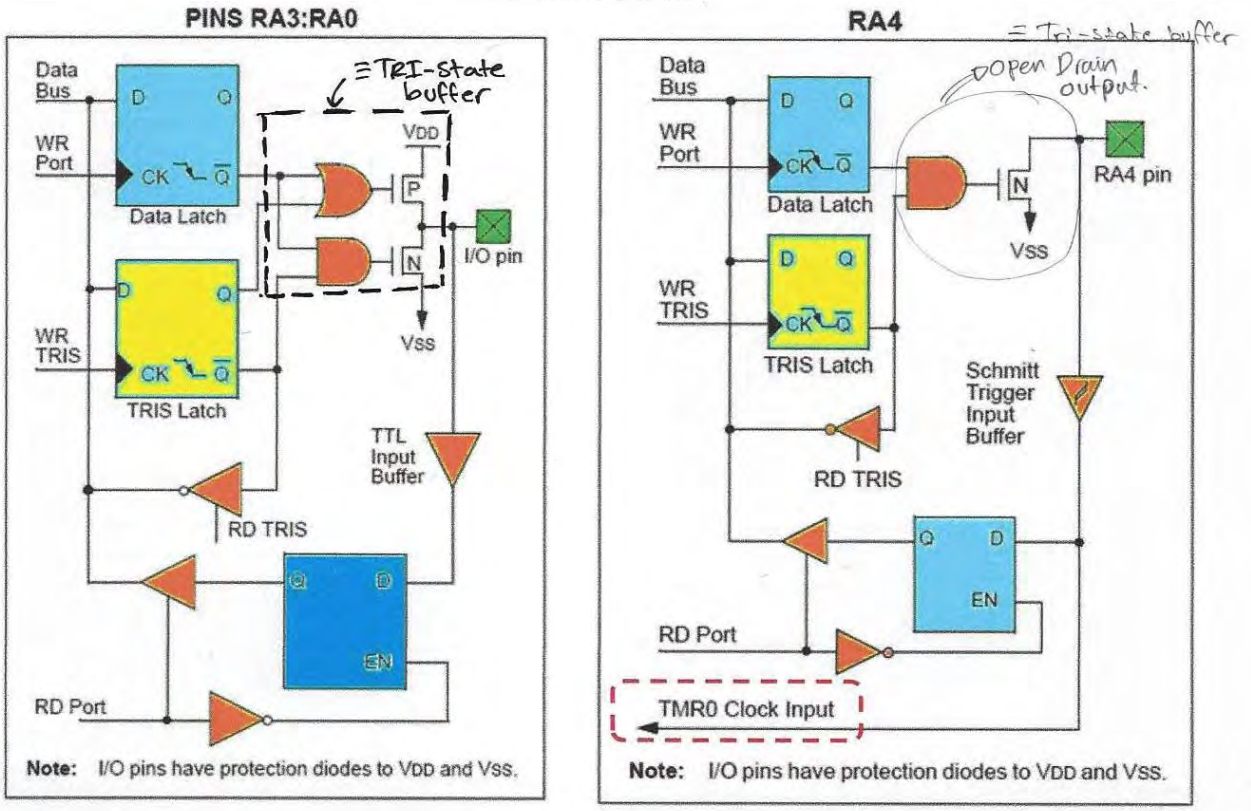
* Thus, to clear the flag RBIF at startup or during ISR you need 2 steps (you have to clear the cause of the interrupt before clearing the flag):

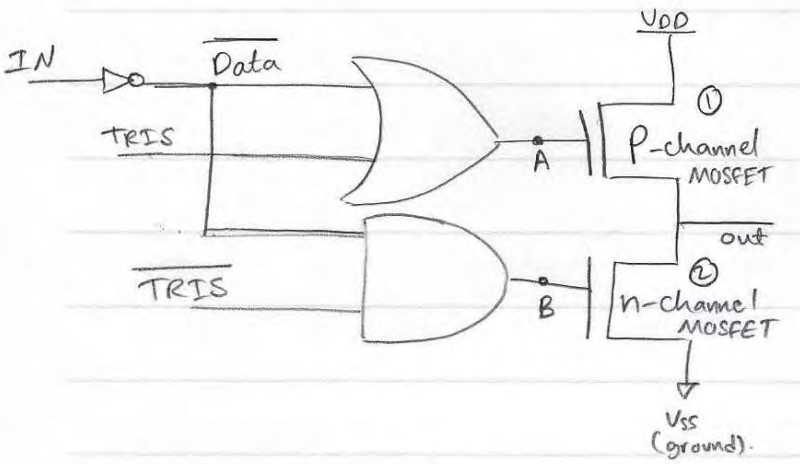
MOVF PORTB, 1
BCF INTCON, RBIF ← if we use this instr. only, we'll have multiple false interrupts for the same request

or we will have 1 more false int.

Hardware Realization of Parallel Ports

PORT A

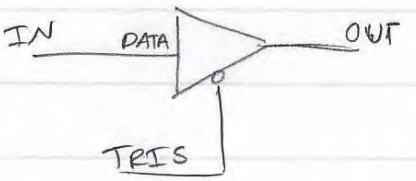




* remember

	IN	OUT
p-ch →	1	open CKT
	0	ON (acts as small R)
n-ch →	0	open CKT
	1	ON (~ ~ ~ ~)

≡



TRIS	OUT
0	Data
1	high impedance

TRIS	A	B	out
0 (output)	$\overline{Data}$	$\overline{Data}$	data
1 (Input)	1	0 → high impedance (open CKT).	

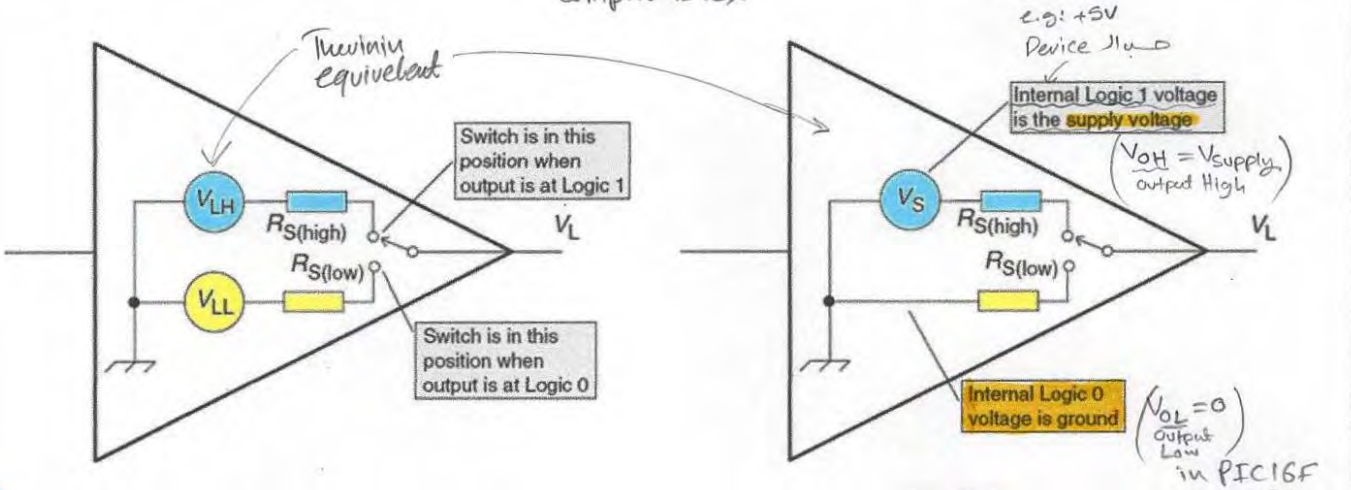
if Data = 1 → $\overline{Data}$ = 0 → n-ch: ② is off & p-ch: ① is on → output will be connected to VDD through small Resistor → ∴ out = Logic 1

Data = 0 → $\overline{Data}$ = 1 → ② is ON & ① is off → output will be connected to ground ∴ out = Logic 0

Hardware Realization of Parallel Ports

Electrical Characteristics

- Logic gates are designed to interface easily with each other, especially when connecting gates from the same family
- The concern arises when connecting logic gates to non-logic devices such as switches and LEDs (You have to make sure that they are electrically compatible).

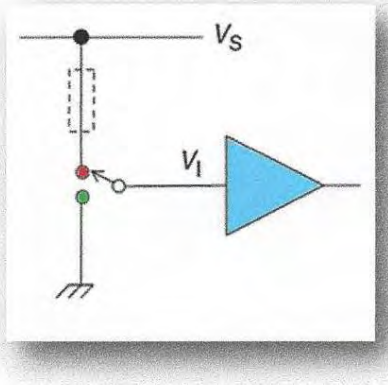


Generalized model (↳TTL)

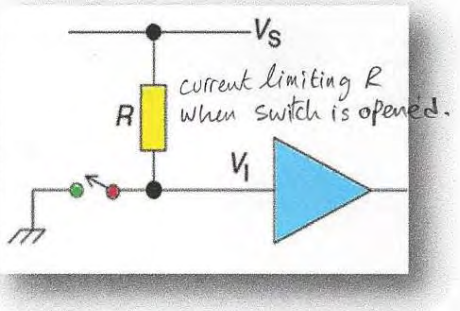
CMOS model ← we use this model in PIC16F

Interfacing to Parallel Ports

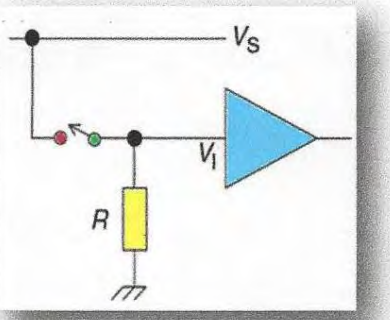
Switches



Interfacing to SPDT switch. A current limiting resistor might be needed

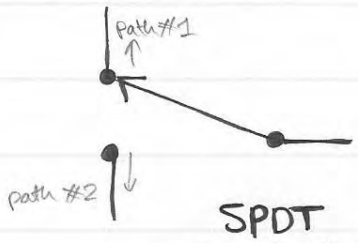


Interfacing to SPST switch. To reduce wasted current, the pull-up resistor R should be high (10-100KOhms)



Interfacing to SPST switch using a pull-down resistor

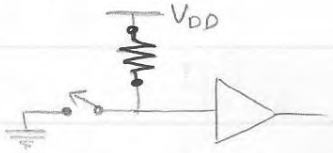
- Switches Types:



SPDT
Single Pole Double Throw Switch.

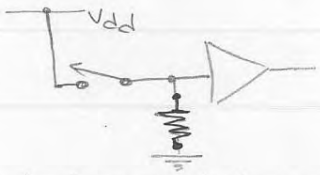
SPST
Single Pole Single Throw.

- Pull-up resistor:



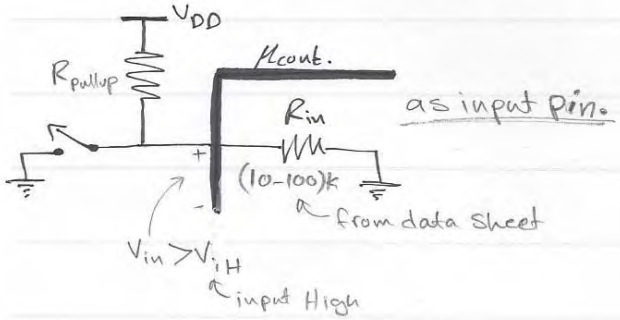
Pull-up resistor "pulls" the voltage of the wire it's connected to towards (V_{dd}) level [logic high] when the other active devices are disconnected.

- Pull-down resistor:



Pull-down resistance holds the signal level to ground (logic low) when other devices are disconnected.

* how to compute $R_{pull-up}$?



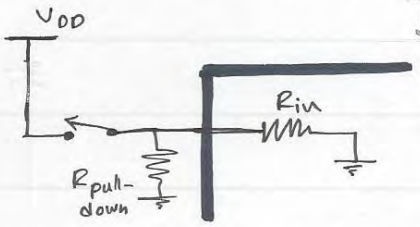
Voltage division!

$$V_{in} = V_{DD} * \frac{R_{in}}{R_{pull-up} + R_{in}}$$

choose: $V_{in} > V_{iH}$

* how to compute $R_{pull-down}$? $R_{pull-down} // R_{in}$

! logic is 1



++ $R_{Pull-down}$ has to be chosen to ensure that the µcontroller reads the line as logic low when the switch is off since (due to data sheet) there is Input leakage current = 1µA

so $\left(I_{leakage} * R_{pull\,down} < V_{iL}\right)$

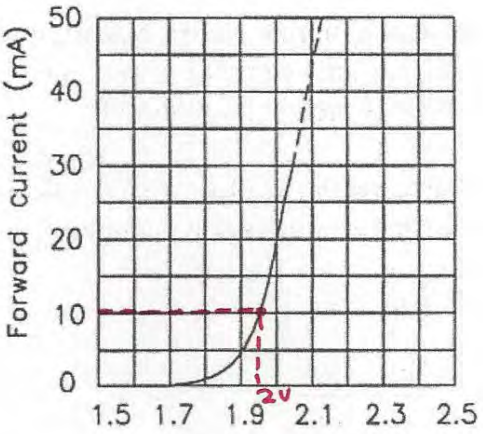
V_{iL}: logic low threshold ≈ 0.8V (PS3 in Data Sheet)

also, when switch is ON, make sure that the current entering the PIC < I_{max} sunk by Pin.

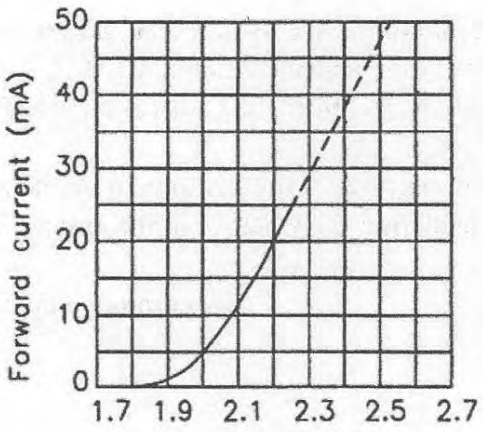
Interfacing to Parallel Ports

Light Emitting Diodes (LEDs) (Output device)

- A special type of diodes made of semiconductor material that can emit light when forward biased

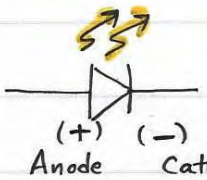


Type number: L-441D
Wavelength = 627 nm
15mcd typ.@ 10 mA
15 milli candle



Type number: L-44GD
Wavelength = 565 nm
12mcd typ.@ 10 mA

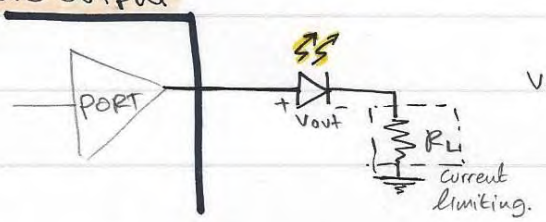
LED



cutoff voltage ($V_D > 0.7$) $\approx 1.7 \sim 2.1$ (V_D for the ordinary diode)

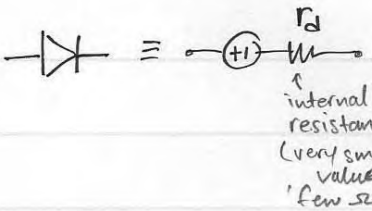
→ to connect LED to µcont.:
(Source)

1) as output



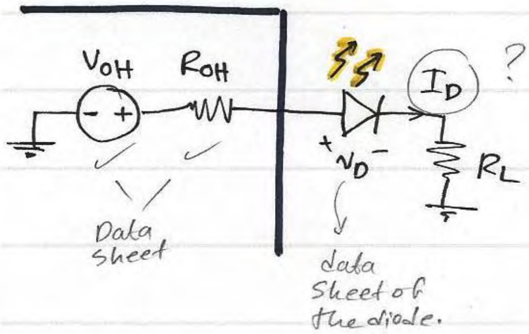
$V_{out} \geq V_D$ ON

without R_L ∫ → OUTPUT: $V_{out} = +5$ ON; $V_{out} = 0$ Off



very small resistance will pull high current → ∴ consumes more power. This is why we add (R_L)

how to compute R_L? → cont.



? from I_d curves (slide 20)

① حدد شدة الإضاءة
② بناء عليه حدد قيمة التيار، + V_D من
الـ curves .

e.g:-
15m candle @ 10 mA
$\therefore V_D \cong 2$ Volts

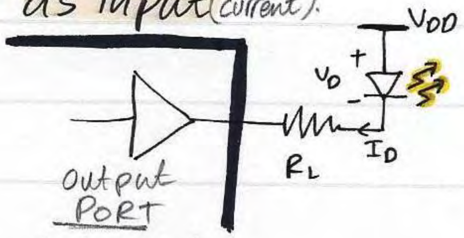
RL?

$$-V_{OH} + I_D(R_{OH} + R_L) + V_D = 0$$

(find. R_L)

2) as input (current). (Sink)

(w.r.t current)



Output PORT
but current is entering the Pin.

but output & input are defined interms of Voltages not currents.

* to light the LED you have to output logic low.

* why do I need this method?

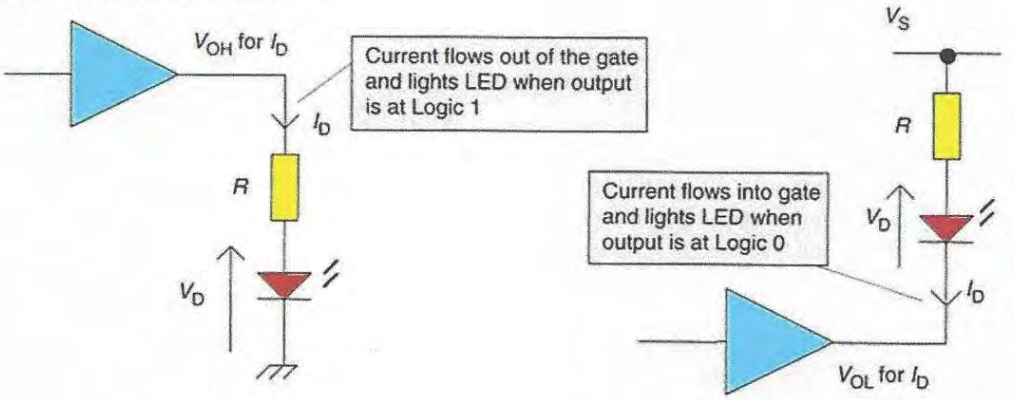
① So that when I want to use devices that require high Voltage such as motors which might need 12V. while $V_{DD} = 5$. So, I have to use external source.

② There is I_{max} to be sourced from the PIN < I_{max} sunk by PIN
(so, sometimes the pin can't supply the required amount of current).
espicially in TTL CKTS.
where sink current > source current.

Interfacing to Parallel Ports

Light Emitting Diodes (LEDs)

- LEDs can be driven from a logic output as long as the current requirements are met
- Interfacing of LEDs depending on the logic type and their capability to source and sink current



V_{OH} Logic gate output high voltage

For current source: $V_{OH} = RI_D + V_D$

$$R = \frac{V_{OH} - V_D}{I_D}$$

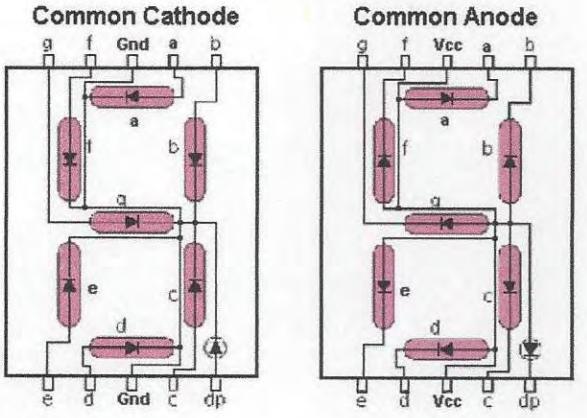
V_{OL} Logic gate output low voltage

For current sink: $V_S = V_{OL} + RI_D + V_D$ (general case (for TTL also))

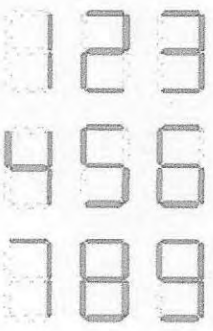
$$R = \frac{V_S - V_D - V_{OL}}{I_D}$$

Interfacing to Parallel Ports

7-Segment Display



check example in the website.



Digit Shown	Illuminated Segment (1 = illumination) a	b	c	d	e	f	g
0	1	1	1	1	1	1	0
1	0	1	0	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

from the dataSheet:- P49

I_{max} by any I/O PIN → Source 25mA, Sink 25mA] equal! — خلافاً لـ CMOS gates!

I_{max} by PORTA → Source 50mA note that $50 \neq 5 \times 25$!
→ Sink 80mA 〃 〃 $80 \neq 5 \times 25$!

لو بدي اشبك ال output ≤ 5 pins ما بقدر اسحب 25mA من كل واحد ! مجموع I ≥ * Source 50, * Sink 80

I_{max} by PORT B → Source 100mA
→ Sink 150mA

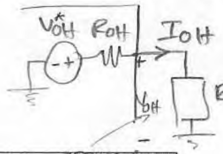
I_{max} into V_{DD} = 100mA

〃 out of V_{SS} (GND) = 150 mA

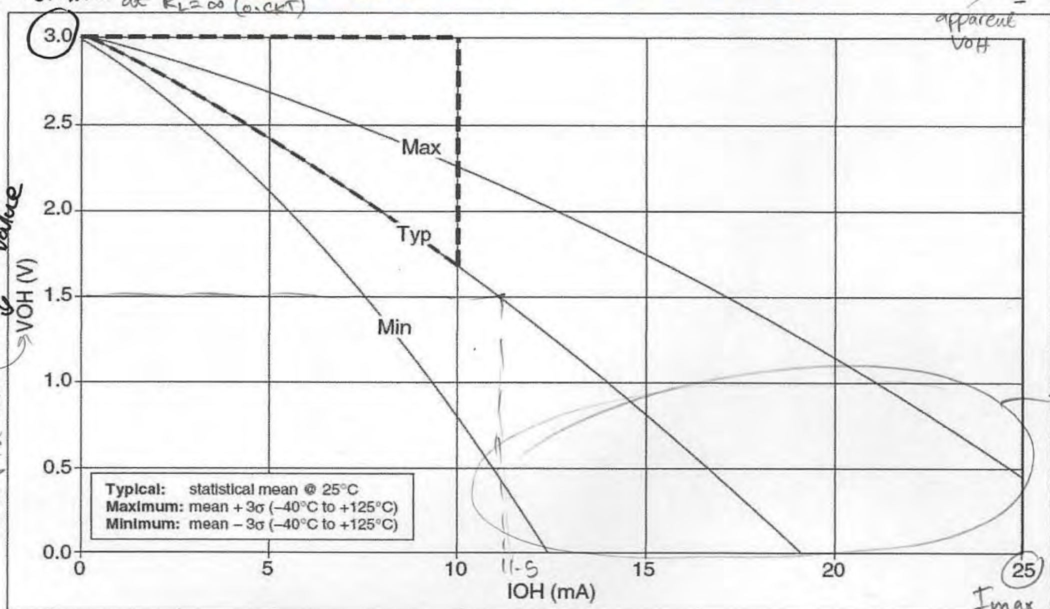
The PIC 16F84 Parallel Ports

Port Output Characteristics

(sourcing current)



V_{OH} vs. I_{OH} (VDD = 3V, −40 to 125°C)



here V_{OH} might become less than logic low threshold (in this CKT we don't care because we're dealing with resistance R_L. But if it was replaced by logic gates we must be careful)!

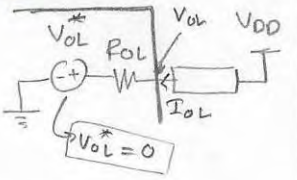
$R_{OH} = \frac{V_{OH}^* - V_{OH}}{I_{OH}} = \frac{3-1.5}{11.5} = 131\Omega$ ← R_{OH} = 130 Ω

The PIC 16F84 Parallel Ports

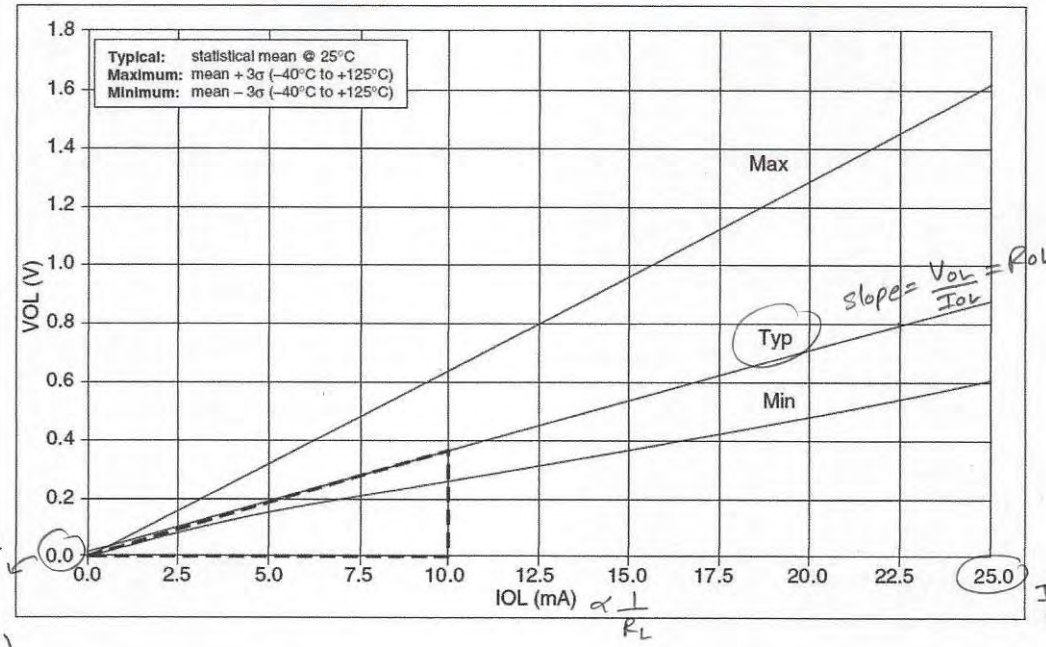
*check datasheet for more curves.

Port Output Characteristics

(Sinking current)



V_{OL} vs. I_{OL}(VDD = 3V, −40 to 125°C)



slope = $\frac{V_{OL}}{I_{OL}} = R_{OL} = \frac{0.8}{22.5m} = 35.7 \approx 36$

$I_{OL} \propto \frac{1}{R_L}$

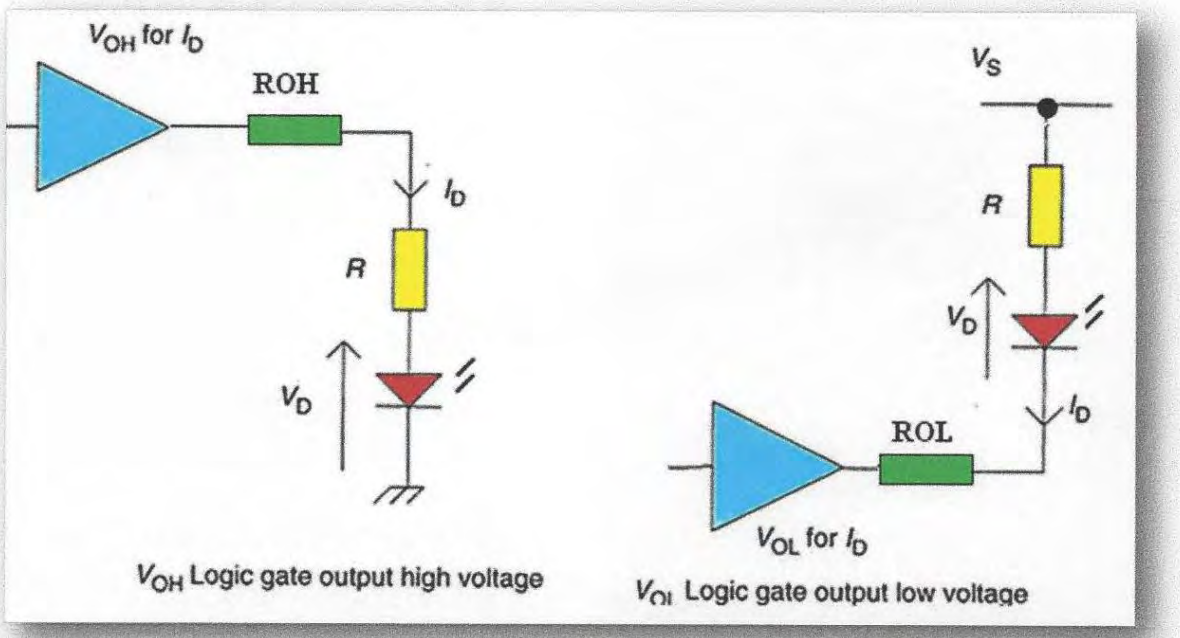
Imax Sink.

at open CKT ($R_L = \infty$)

R_{OL} = 36 Ω =

The PIC 16F84 Parallel Ports

Port Output Characteristics

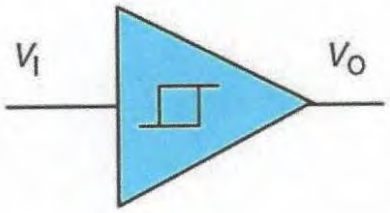


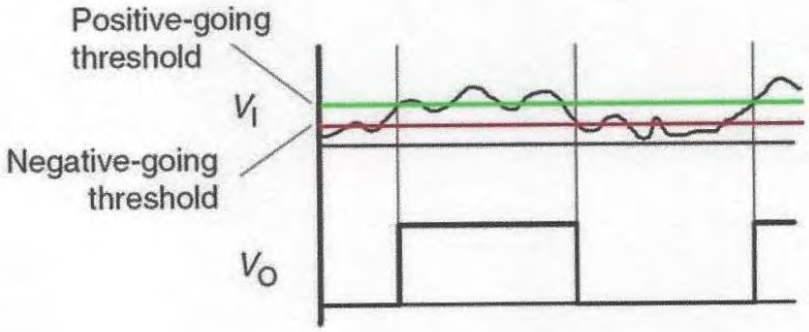
Computation of limiting resistors when internal resistance of the port pin is considered

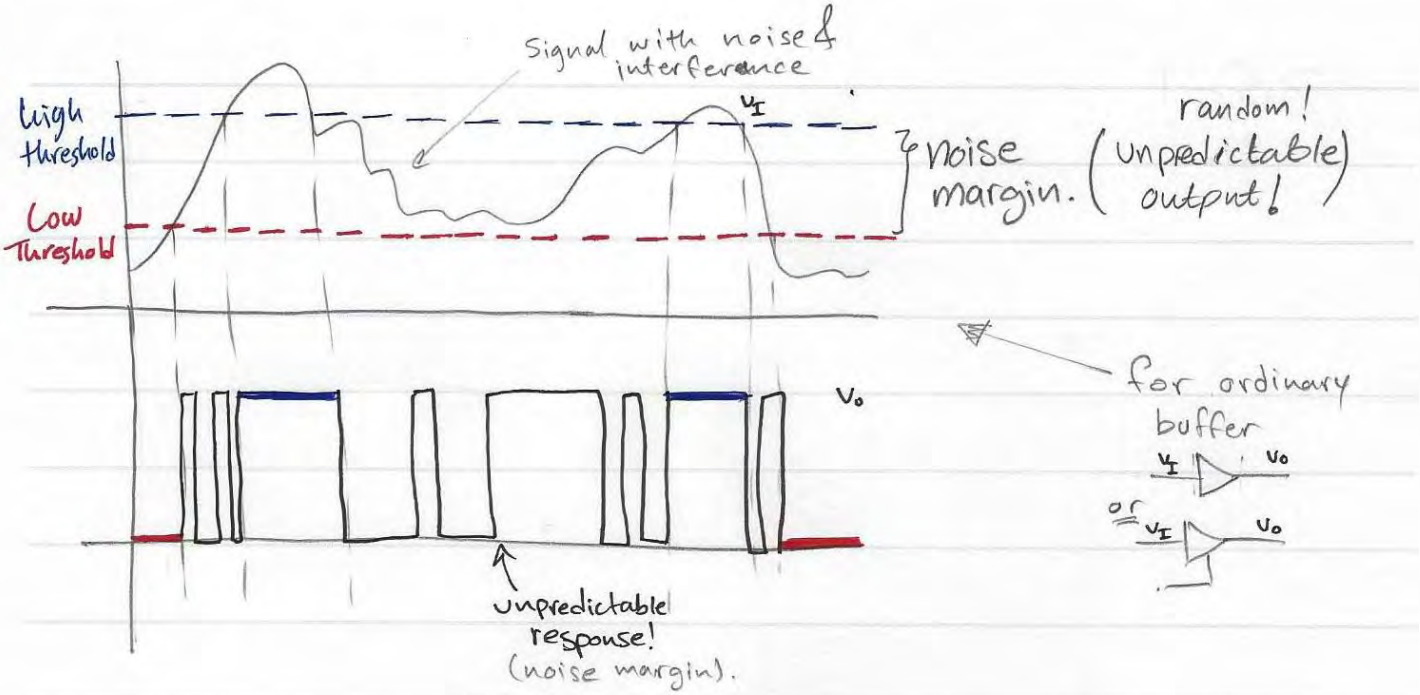
Hardware Realization of Parallel Ports

Electrical Characteristics

- **Schmitt Trigger Input**
  - A special type of gate with two thresholds
  - Remove fluctuations and corruptions in the input signal

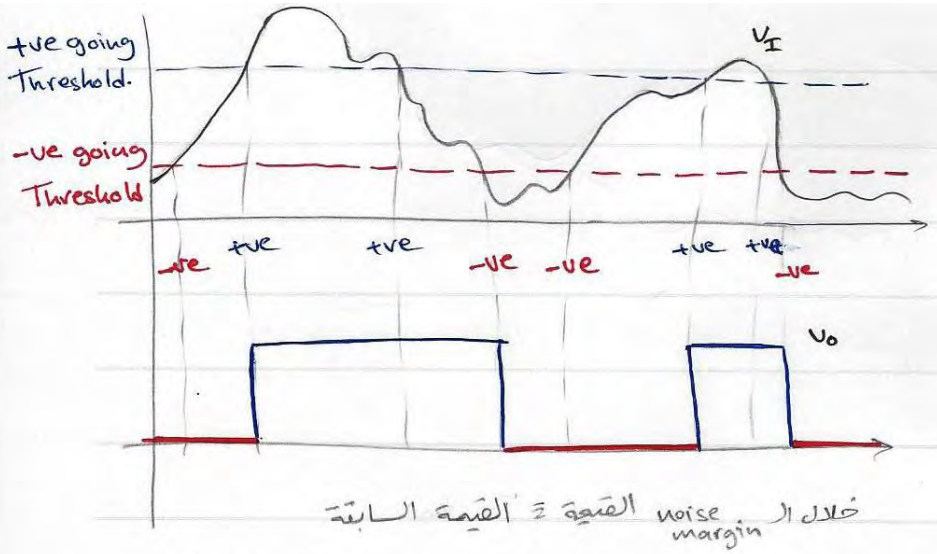




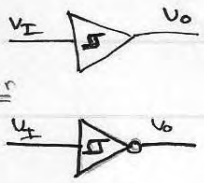


thus, we use another type of buffers called schmitt triger in some case (e.g: TMR0 INput Clk signal (with RA4) & external interrupt (with RB0))

*note that these applications (functions) are edge triggered (depends on falling/rising edges for their operations). too many (false) edges might be caused due to noise margin's random response.



Schmitt trigger buffer



or

noise margin خلال ال القيمة ≠ القيمة السابقة

changes the value requires crossing two thresholds.

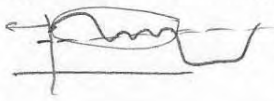
(i.e: to change from Logic low to Logic high, the signal has to cross -ve threshold first, enter the noise margin then cross +ve threshold).

(will stay logic low during it.)

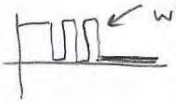
Why didn't we consider 1 threshold (if > threshold = 1 else = 0)?

take the following scenario.

Supposed to=Logic 1



⇒



will see it as 3 Pulses instead of 1. (due to nois & interferance).

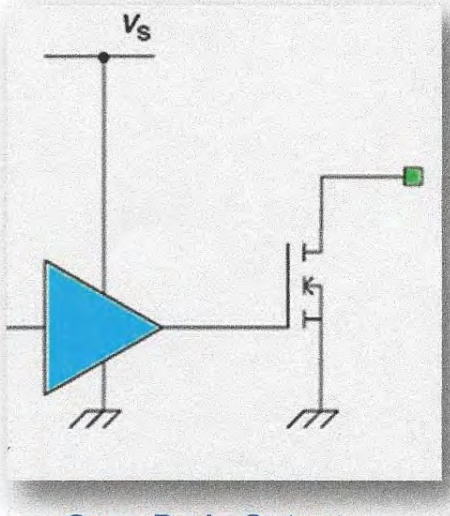
⟶ So, we put the noise margin to allow little fluctuation to occur without affecting output signal. (& use 2 thresholds)

Hardware Realization of Parallel Ports

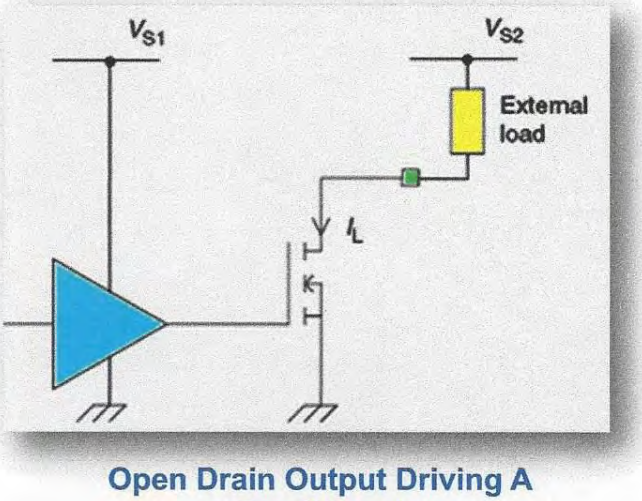
Electrical Characteristics

ما وع كيفيها ... وع كيف عنها !"

- **Open Drain Output**
  - Flexible style of output that can be adapted as a standard logic output or a direct drive for small loads



Open Drain Output

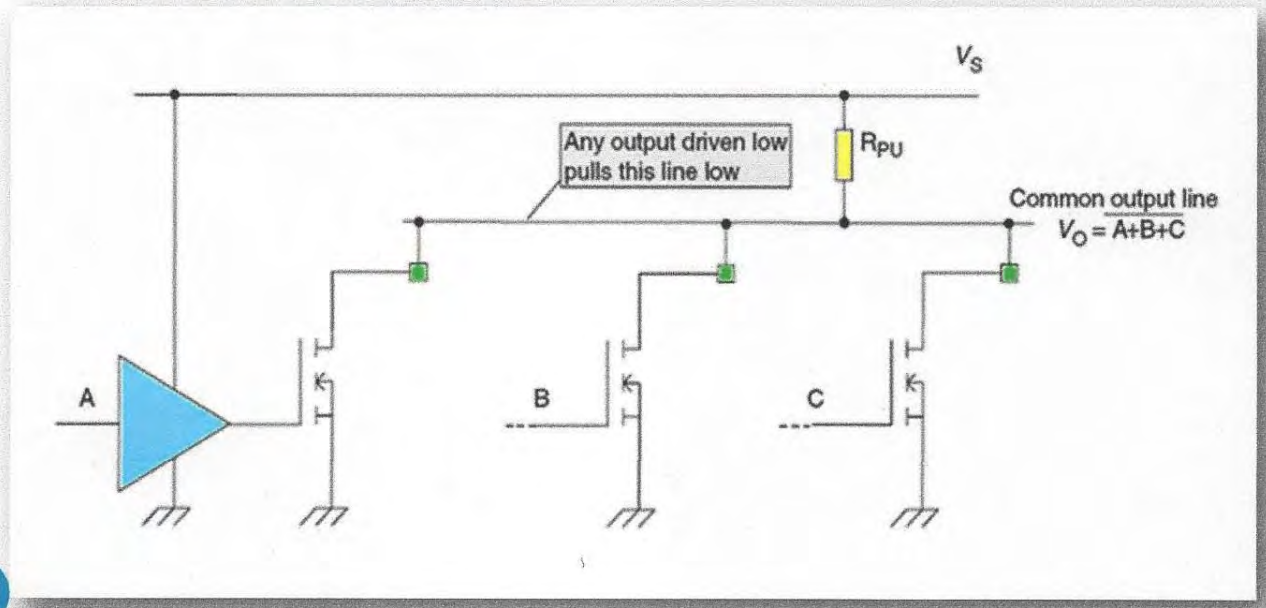


Open Drain Output Driving A Small Load

Hardware Realization of Parallel Ports

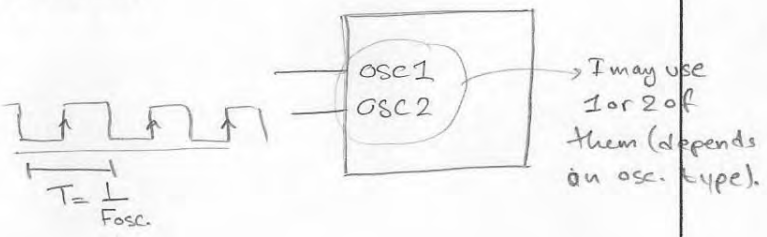
Electrical Characteristics

- **Open Drain Output**
  - Can be used as a wired-OR

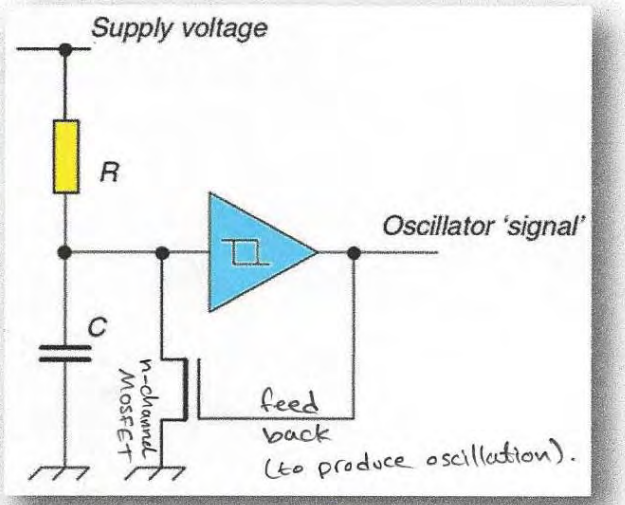


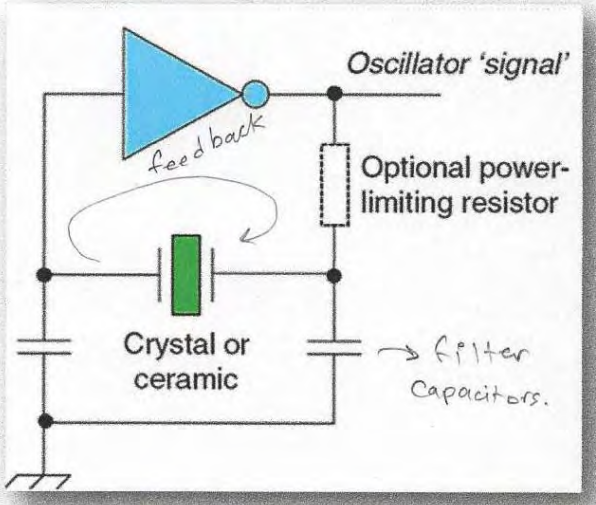
The Oscillator

- The choice of clock determines the operating characteristics for the microcontroller
- *Faster clock gives* faster execution, *but more power consumption*
- Accurate and stable operation of the microcontroller requires accurate and stable clock



The Oscillator
Oscillator types

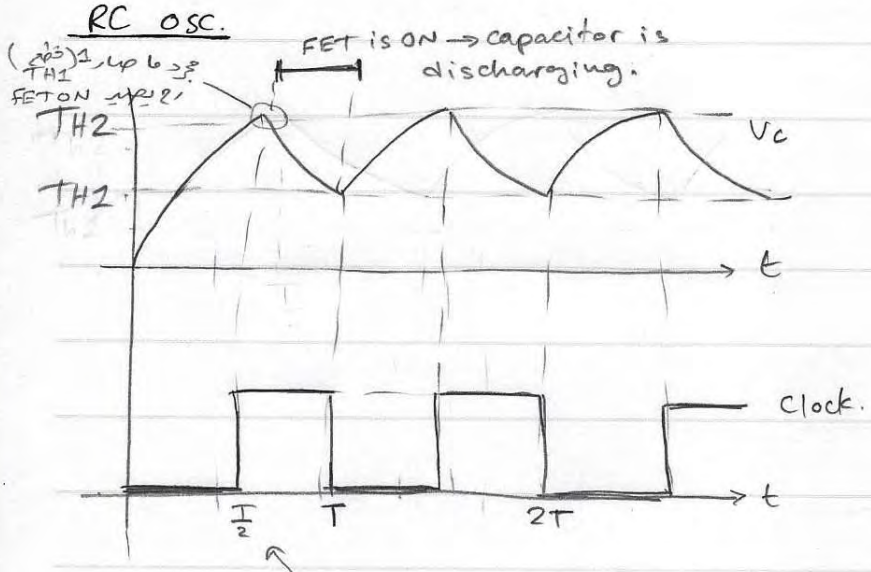




Resistor–capacitor (RC).
- *low cost*
- *not precise* (sensitive to temperature).

Crystal or ceramic
- *expensive*
- *stable and precise*
- *mechanically fragile*

RC osc.



in this case duty cycle = 0.5

ON : off
0.5 : 0.5] this is nt always the case

ON → discharging → $T = RC$ ← resistance of the MOSFET (small R)

off → charging → $T = RC$; $R = R_S$ if $R_S \simeq R_{n\text{-ch.}}$ → Dutycycle $\simeq \frac{1}{2}$

The PIC 16F84A Oscillator

- The 16F84A can be configured to operate in four different oscillator modes using the F0SC1 and F0SC0 in the configuration word

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	PWRTE	WDTE	F0SC1	F0SC0
bit13													bit0

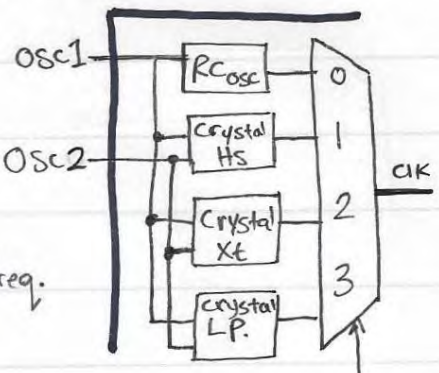
F0SC1	F0SC0	Mode
0	0	LP oscillator – *intended for low frequency (<200 KHz) crystal application to reduce power consumption*
0	1	XT oscillator – *standard crystal configuration (1-4 MHz)*
1	0	HS oscillator – *high speed (>= 4MHz)*
1	1	RC oscillator - *requires external resistor an capacitor*

31

Why do we need to tell the μcontroller what type of osc. is connected?

→ because part of osc. CKT components are embedded inside the μcontroller

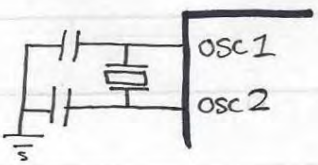
* Osc. type is selected by setting bits 0 & 1 in the configuration word (Fosc1 : Fosc2)



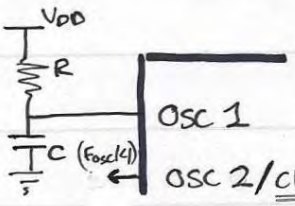
→ 1, 2 & 3 differ in freq.

00 : LP
01 : XT
10 : HS (high freq., consumes the most current)
11 : RC

Crystal osc. external CKT

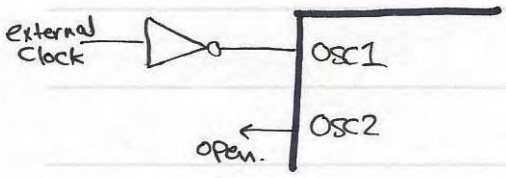


RC osc. external CKT



if the config. is "RC osc." this pin outputs Fosc/4

What about using an external clock? how to tell the µcontroller?

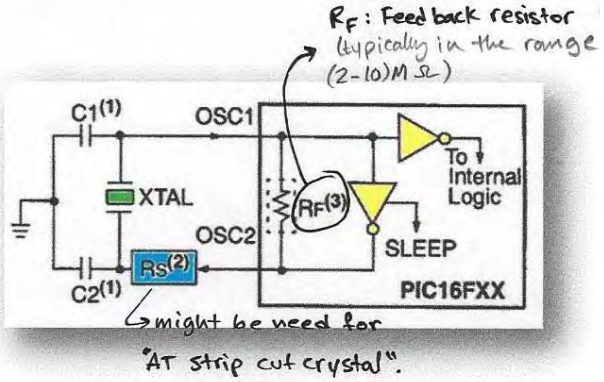


configure the osc. bits (Fosc1:0) one of the ↑ as ↑ crystal modes!

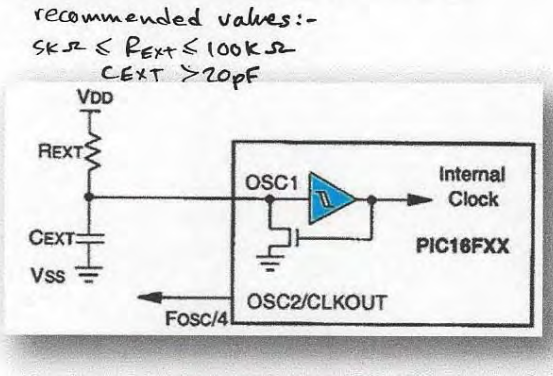
Since all crystal modes uses both osc1 & osc2 pins, while external clock uses only osc1 the µcontroller will know that the user is inputing an ext. osc.

The PIC 16F84A Oscillator

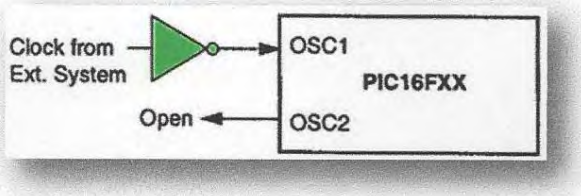
- The 16F84A has two oscillator pins ; OSC1 and OSC2.



XT configuration
(crystal)



RC configuration

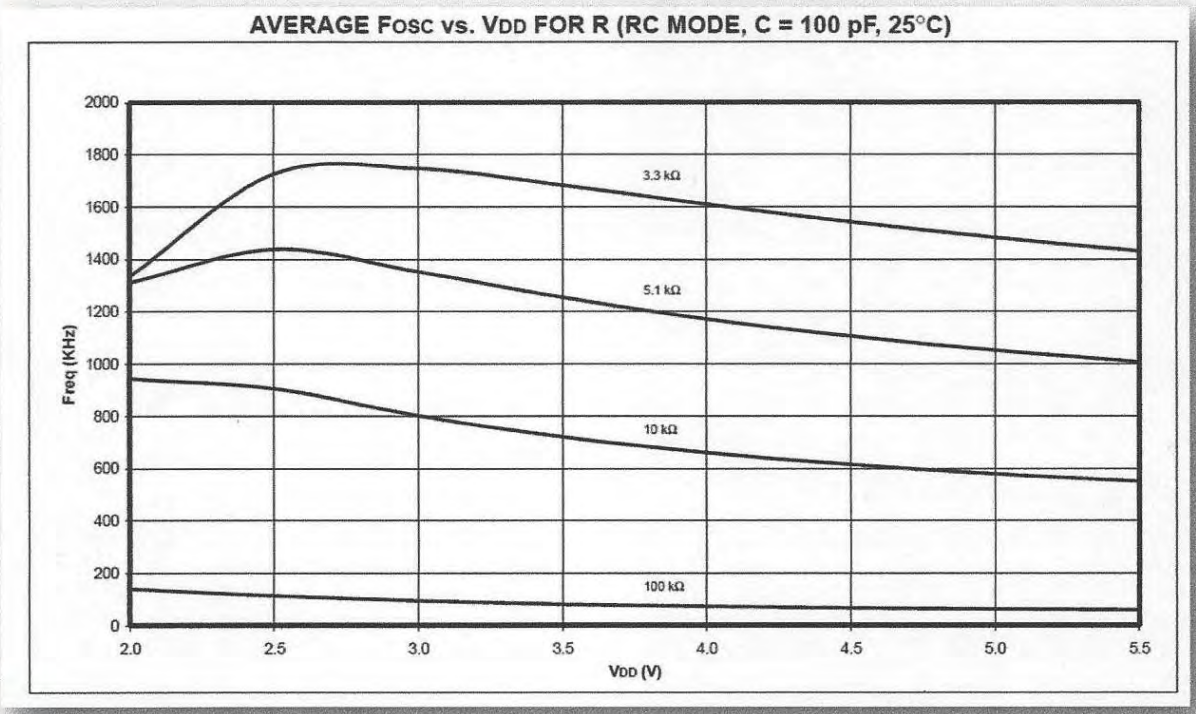


External Clock

The PIC 16F84A Oscillator

RC osc. freq. is a function of R_{EXT}, C_{EXT} & V_{DD}.

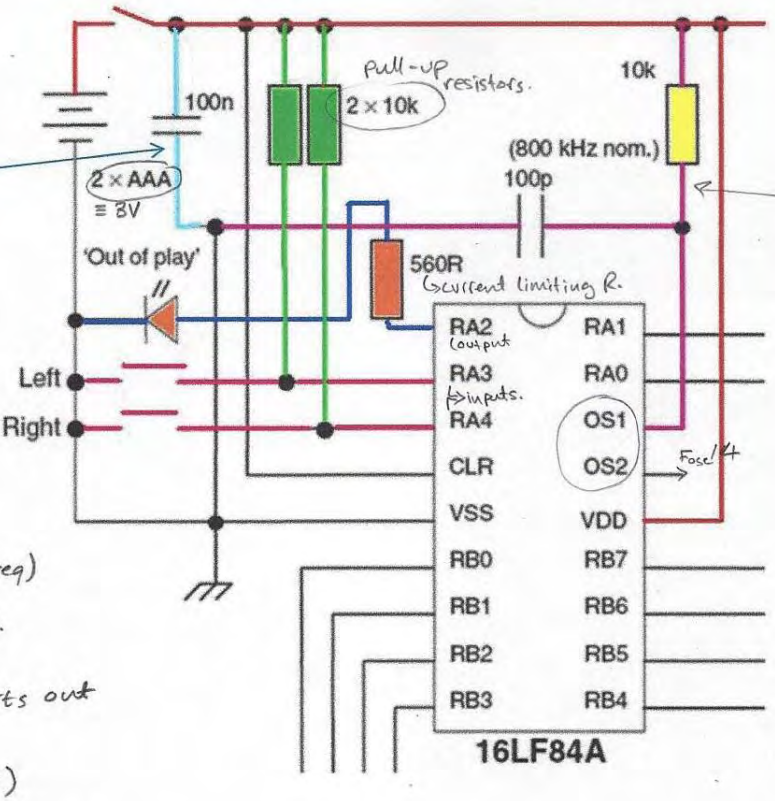
- RC oscillator frequency dependence on power supply



The Power Supply

PIC16F84A-04 (Commercial, Industrial, Extended) PIC16F84A-20 (Commercial, Industrial, Extended)			Standard Operating Conditions (unless otherwise stated) Operating temperature 0°C ≤ TA ≤ +70°C (commercial) -40°C ≤ TA ≤ +85°C (industrial) -40°C ≤ TA ≤ +125°C (extended)				
Param No.	**Symbol**	**Characteristic**	**Min**	**Typ†** (at 5V, 25°C)	**Max**	**Units**	**Conditions**
	VDD	**Supply Voltage** (extended supply range)					
D001		16LF84A	2.0	—	5.5	V	XT, RC, and LP osc configuration
D001		16F84A	4.0	—	5.5	V	XT, RC and LP osc configuration (4V)
D001A			4.5	—	5.5	V	HS osc configuration (4.5) ↑freq ↑VDD
D002	VDR	**RAM Data Retention Voltage (Note 1)**	1.5	—	—	V	Device in SLEEP mode (the limit to which VDD can be lowered without losing RAM data)
D003	VPOR	**VDD Start Voltage** to ensure internal Power-on Reset signal	—	Vss	—	V	See section on Power-on Reset for details
D004	SVDD	**VDD Rise Rate** to ensure internal Power-on Reset signal	0.05	—	—	V/ms	if $\frac{\Delta V_{DD}}{\Delta t} < 0.05$ V/ms we need external MCLR (RC CKT) or PWRT or osc timers
	IDD	**Supply Current (Note 2)**					
D010		16LF84A	—	1	4	mA	RC and XT osc configuration **(Note 4)** FOSC = 2.0 MHz, VDD = 5.5V
D010		16F84A	—	1.8	4.5	mA	RC and XT osc configuration **(Note 4)** FOSC = 4.0 MHz, VDD = 5.5V
D010A			—	3	10	mA	RC and XT osc configuration **(Note 4)** FOSC = 4.0 MHz, VDD = 5.5V (During FLASH programming)
D013			—	10	20	mA	HS osc configuration (PIC16F84A-20) FOSC = 20 MHz, VDD = 5.5V
D014		16LF84A	—	15	45	µA	LP osc configuration FOSC = 32 kHz, VDD = 2.0V, WDT disabled

The Power Supply



100 nF decoupling capacitor

↳ decouples the AC. (high freq). to ensure that almost DC voltage is passing.

C↓ very small

Z = open CKT (low freq)

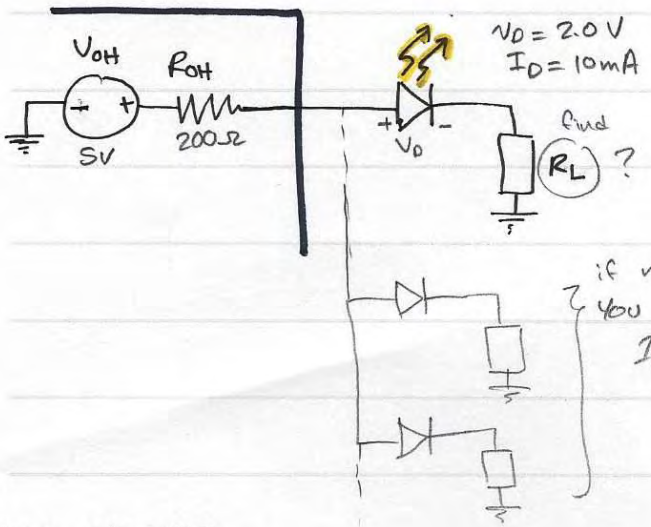
Z = 0 (high freq). short CKT

↳ ∴ this cap shorts out high freq. (works as LPF)

RC oscillator.

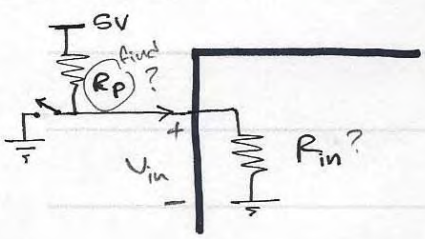
Fosc/4

example #1:



if multiple pins were used, you have to consider I_{max} (source).

example #2:



V_{IH} = 2.0 V (input high)

$V_{in} \geq 2$ (logic high)

$I_{IH} = 2\mu A$

$\rightarrow \frac{5 \times R_{in}}{R_{in}+R_p} \geq 2$

$I_{IH} R_{in} \geq V_{IH} \Rightarrow R_{in} = \frac{2}{2\mu} = 1M\Omega$

then find R_p.

Summary

- Parallel ports allow the exchange of data between the outside world and the CPU
- It is essential to understand the electrical characteristics and internal circuitry of ports
- PIC 16F84A has two parallel ports
- All microcontrollers need a clock. The clock speed determine the power consumption
- Active elements of the oscillator are usually built inside the microcontroller and the designer selects the type and configure it
- It is a must to understand the power requirements of the microcontroller

Starting with Serial

Chapter 10
Sections 1,2,9,10

Dr. Iyad Jafar

Outline

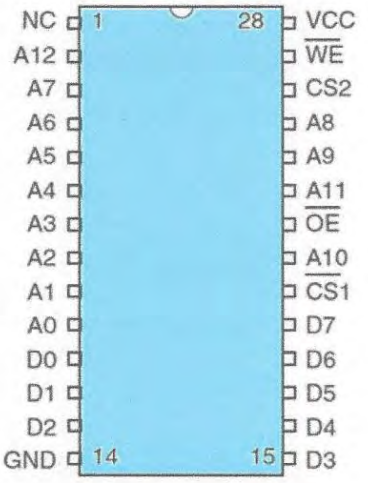
- Introduction
- Synchronous Serial Communication
- Asynchronous Serial Communication
- Physical Limitations
- Overview of PIC 16 Series
- The 16F87xA USART
- Summary

Introduction

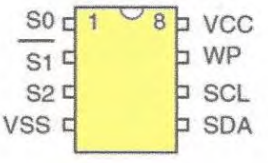
- Microcontrollers need to move data to and from external devices
- In general, two approaches
  - ***Parallel***
    - Data word bits are transferred at the same time
    - A wire is dedicated for each bit
    - Simple and fast but expensive
    - Short distances
  - ***Serial***
    - Bits are transferred one after another over the same link/wire
    - Requires complex hardware to transmit and receive
    - Slow
    - Short and long distances

Introduction

- Two memories of the same size. However, one uses parallel transfer while the other uses serial

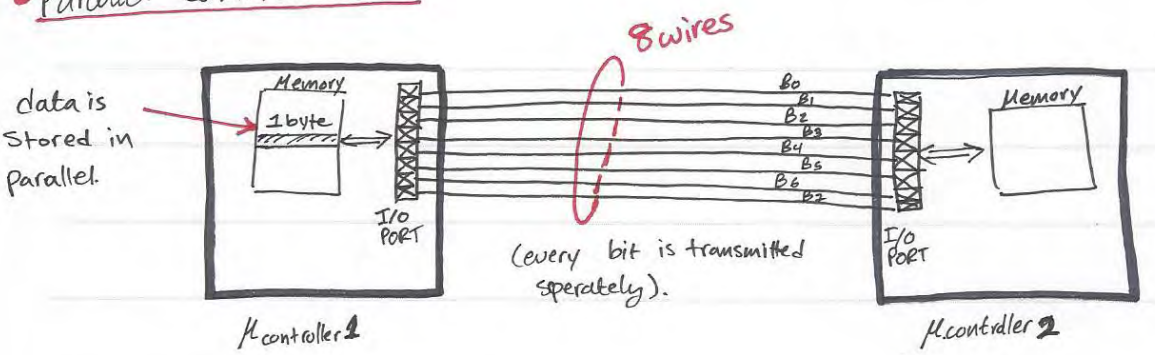


parallel (needs more pins).



serial.

• Parallel communication:



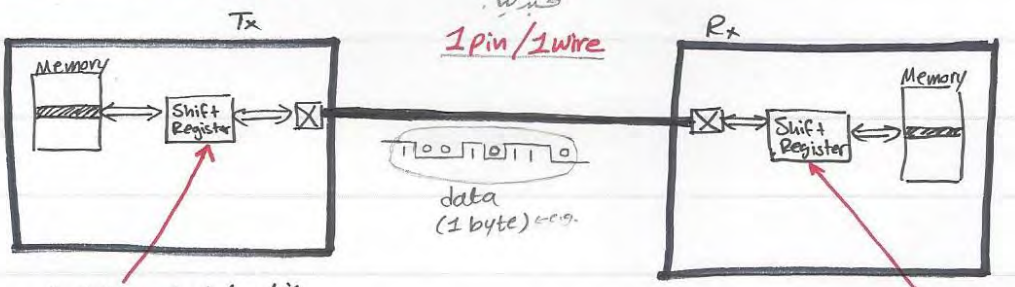
→ Advantages:

- very simple ; where parallel communication procedure matches parallel data storage.
- Fast.

→ Disadvantages:

- Expensive ; you need more # of wires & I/O pins.
- Can't run for long distances, due to:
  1) cost (↑ wire length ↑ power consumption).
  2) interferance (↑ length ↑ interferance).

• Serial Communication:



→ Advantages:

- cheap. (less # of wires & pins)
- Can be used for long distances (↓ interferance (self interferance))

→ disadvantages:

- additional hardware (for synchronization).

*what about the clock? (Rx & Tx must be synchronized).

Serial Communication

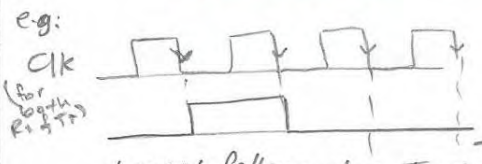
- Bits are transferred one after another on the same wire !!!
- Challenges
  - How to distinguish the start and end of the bit ?
  - How to determine the start and end of a word ?
- Two approaches
  - Synchronous serial communication
    - A separate clock signal is sent in parallel with the data
    - Each clock cycle represents one bit duration
  - Asynchronous serial communication
    - No clock signal !
    - Timing is derived from the data itself

Serial Communication

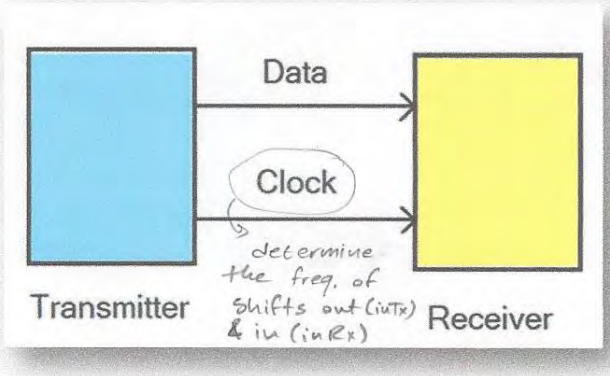
(we send data with the clock signal)

Synchronous

e.g:



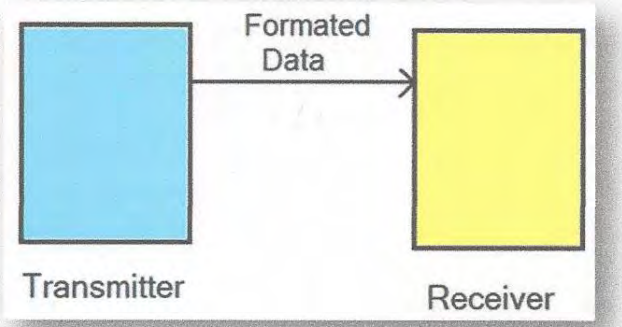
at every falling edge Tx shifts out a bit of the data. & Rx shifts in a bit of the data.



-ve:
1 more wire/channel (or)
∴ ↑cost ↑interference
∴ can't be used for long distances. only (1 - 2) meters

+ve:
faster than asynchronous. (without control bits).

Asynchronous



+ve:
* no clock is sent.
* 1 wire only
* can be used for long distances.

Asynchronous Serial Communication:-

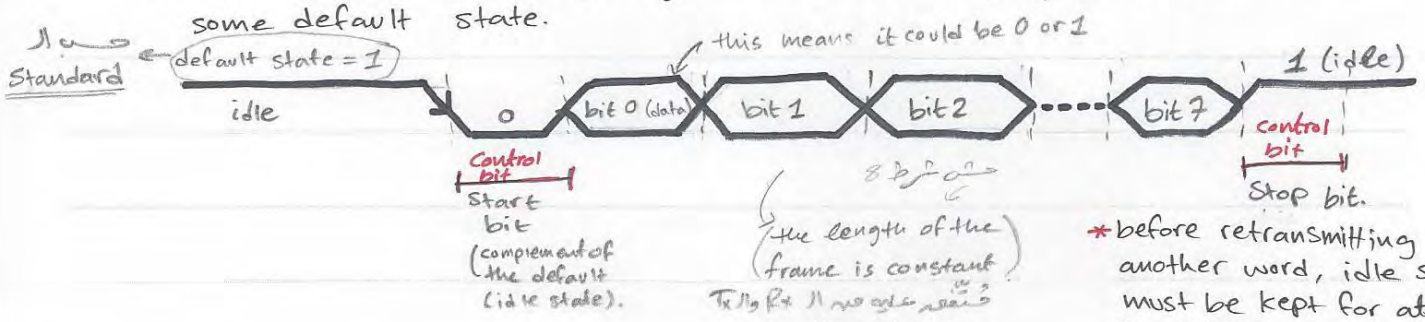
* Tx doesn't provide Rx with clock signal. They both generate their own clock signals.

but it's assumed that both clock rates are the same. (∴ we don't need extra wire for Clk).

but still there's a problem of phase shift (= delay)! how does the Rx know when are the start & end bits of a word? (→ to start shifting the data at Rx!)

Solution: add control bits (header) to the sent data.

→ when Tx isn't sending any data (idle state) put the line in some default state.



* before retransmitting another word, idle state must be kept for at least (1) T_b (bit period). this depends on the standard you're using.

Example: 1 MHz, 8 bits of data, (a) Synch. (b) asynch. (c) Parallel

a) Synch:

bit time = $\frac{1}{1\,MHz}$ = $1\,\mu Hz$ (= 1 M bps < bit per second.)

Time = $1\mu \times 8 = \boxed{8\,\mu sec.}$

b) asynch. (8 data bits + 2 control bits) overhead

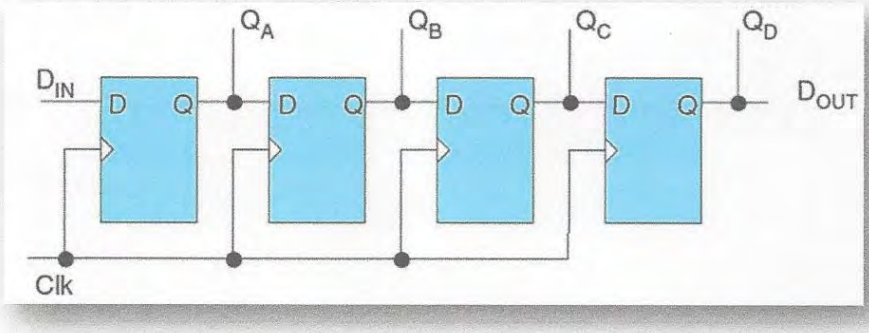
time = $10 \times 1\mu = \boxed{10\,\mu sec}$

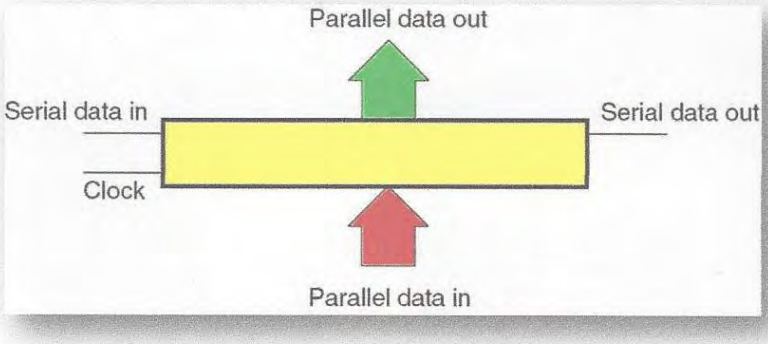
or = $\boxed{11\,\mu sec.}$ if with parity

c) Parallel → time = $\boxed{1\,\mu sec.}$

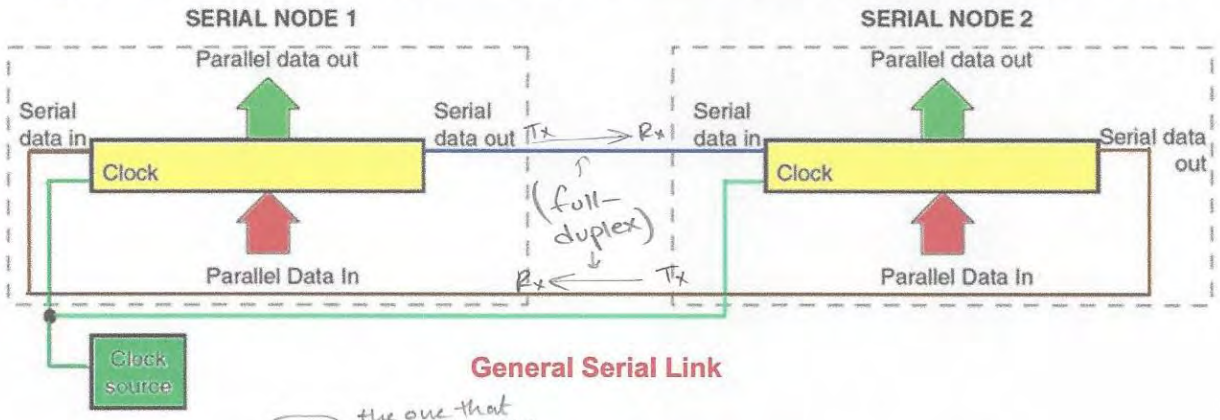
Serial Communication

- Data inside the memory and microprocessor is formatted in parallel. How to transmit it serially?
- Shift registers

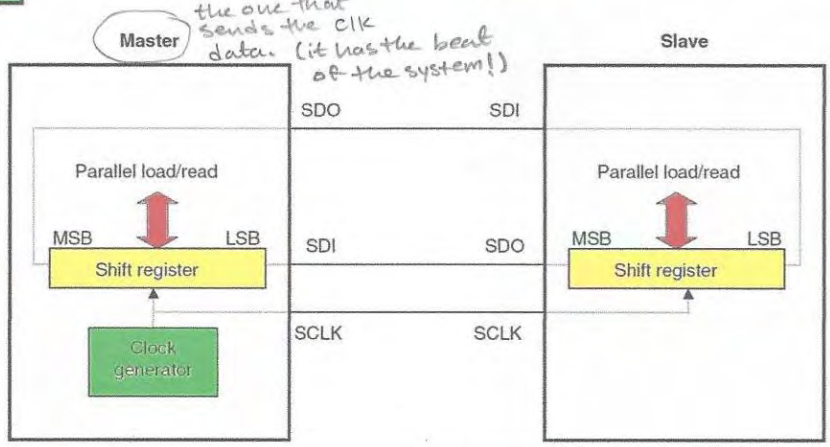




Synchronous Serial Communication

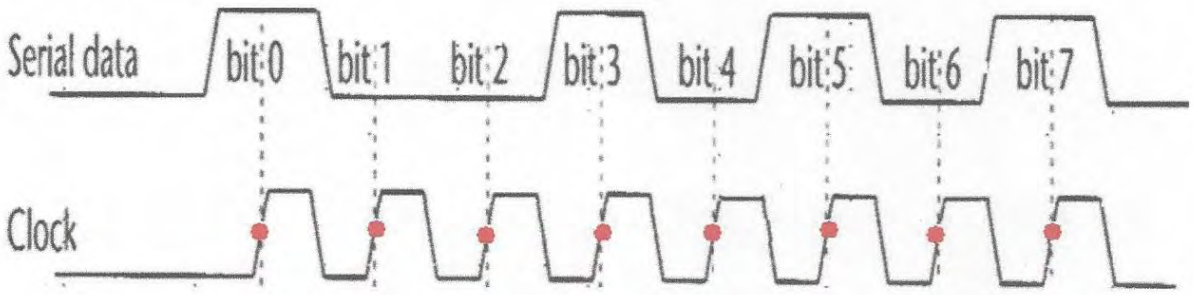


General Serial Link



Synchronous link implemented using a microcontroller

Synchronous Serial Communication



Advantages

- Simple hardware
- Efficient
- High speed

Disadvantages

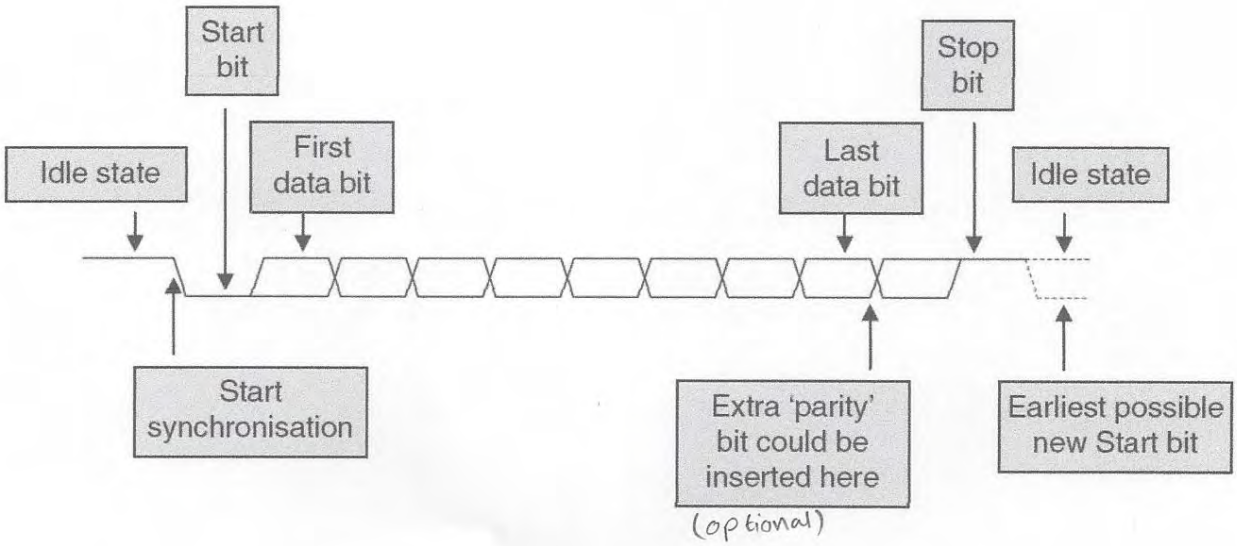
- Extra line for the clock
- The bandwidth needed for the clock is twice the data bandwidth
- Data and clock may lose synchronization over long distance

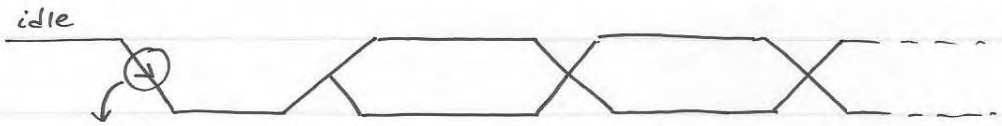
Asynchronous Serial Communication

- No clock signal !
- The transmitter and receiver should operate a clock at the same rate
- To synchronize the clocks of the transmitter and receiver, data is framed with a start and stop bits

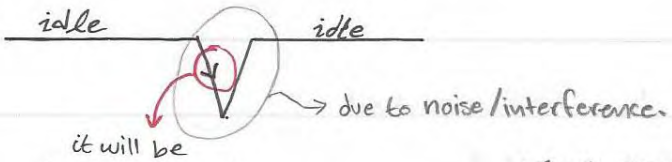
Asynchronous Serial Communication

- **Framing**





let's consider the following scenario:



it will be considered as the Start of the (start bit) & will issue a false reception operation.

*المشكلة في الإرسال على لحظة زمنية واحدة لتحديد بداية الإرسال و الاستقبال!

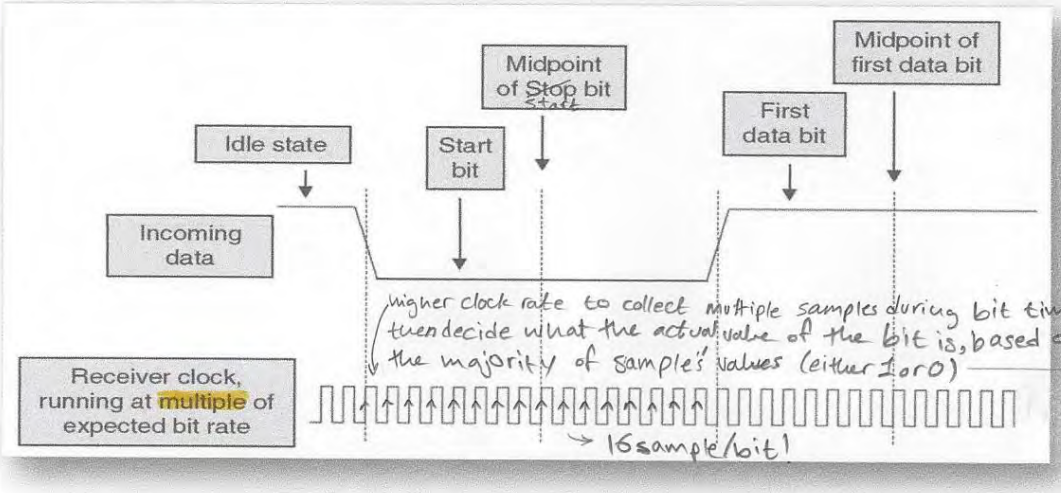
Solution: read more than 1 sample during bit time.

* This isn't done only for the start bit, it's done for all bits (to reduce the effect of noise).

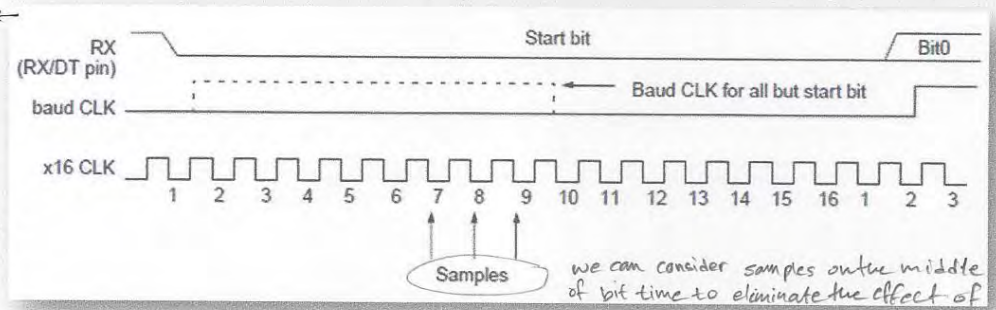
↑ # of samples → ↑ Storage Space → ↑ HW complexity → ↑ cost

Asynchronous Serial Communication

- **Synchronization**



in pic 16F87xA data is sampled 3 times by a majority detect circuit to determine if a high or a low level is present at the R_x Pin.

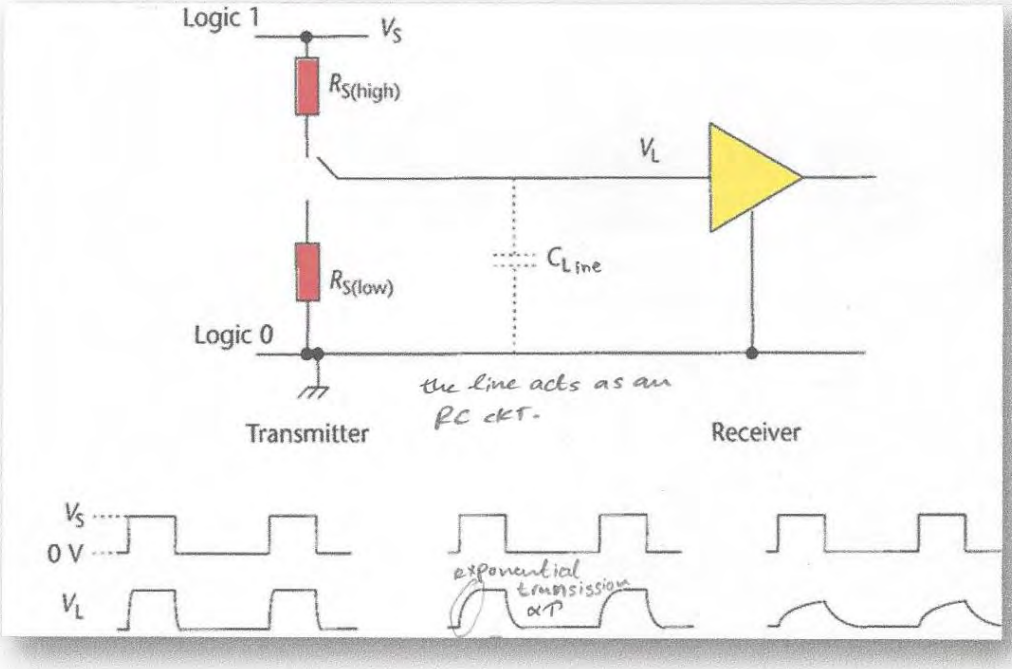


Physical Limitations

- **Time Constant effect**

$\tau \propto$ Bandwidth.

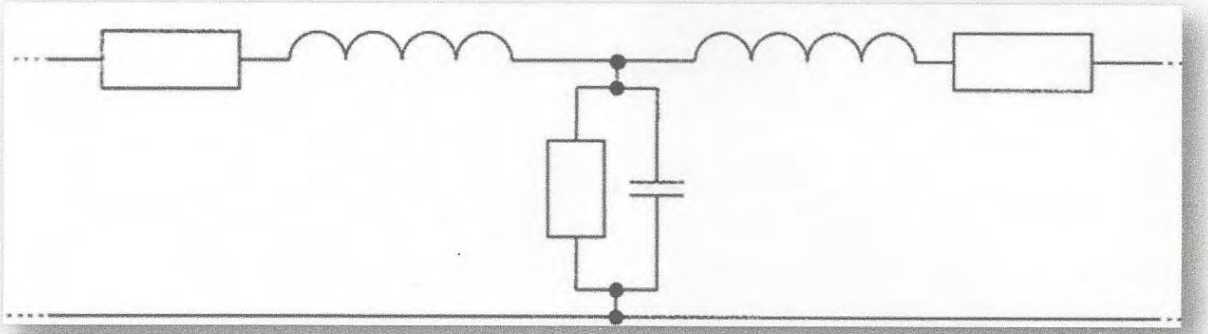
*To avoid this effect: You mus do some calculations to obtain τ_{line}. Thus, you'll know the limit for data rate (# of transissions per unit time).



Physical Limitations

- Transmission Line Effects
  - Characteristic impedance and reflections
  - Lines should be terminated properly

Solve by impedence matching.

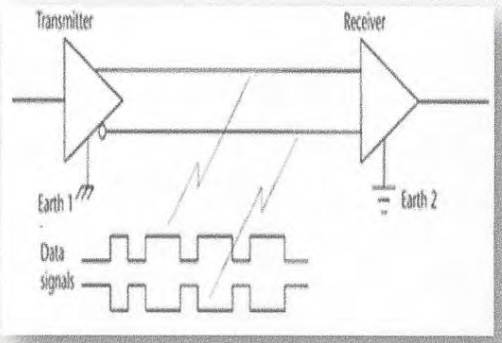


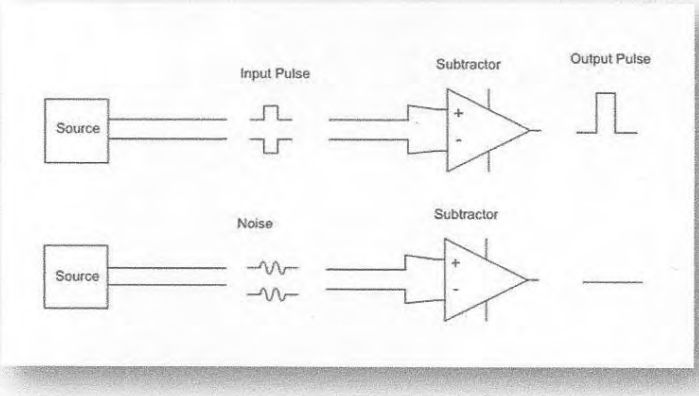
Physical Limitations

- **Electromagnetic Interference**
  - Generated due to high voltage rates of change.
  - How to minimize:
    - At source:
      - reduce voltage rate of change.
    - In communication link:
      - large separation from source of interference.
      - Increase data voltage.
      - Screening
      - Use optical links
    - At receiver:
      - Use filtering techniques

Physical Limitations

- **Ground Differentials**
  - With longer wires, ground potential at one point might not be the same at another point.
  - Solutions:
    - Differential transmission.
    - Electrical isolation
    - Use optical communication links





16

* Can we use PIC16F84A to send data serially?

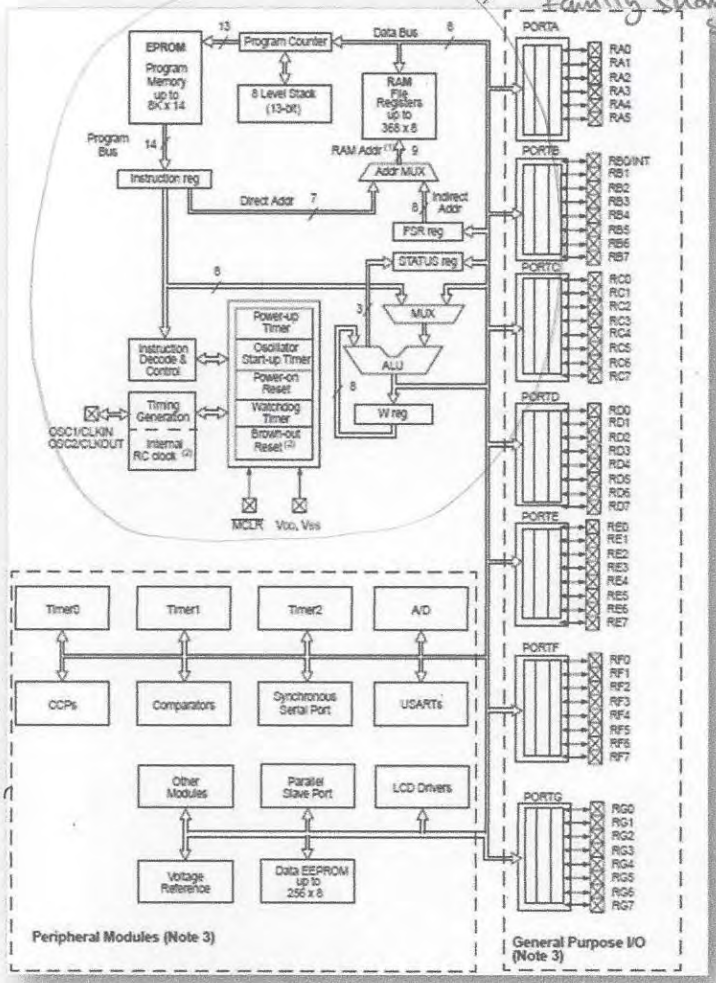
→ to send data serially (HW approach.) we need shift registers, with clock that shifts out data to a specific pin. but PIC16F84A doesn't have Hardware support for serial comm. However, this can be done by software. (Project).

↳ this will hold CPU's resources untill the end of transmission. Thus, an upgrade was introduced in other PIC devices. (by adding specialized hardware module(s) to do this job).

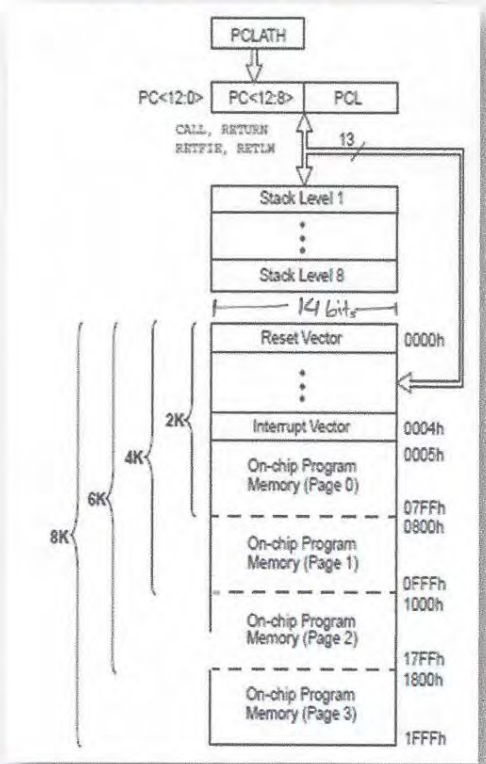
Overview of the PIC 16 Series

- We have already seen the PIC 16F84A
- Other members in the series have more features:
  - Additional I/O ports
  - More HW timers
  - A/D converters
  - LCD Drivers
  - USARTs (can be configured as synch. or asynch. Tx or Rx).
  - Synchronous Serial
  - Comparators
  - ....

members of the same family share the same core



Overview of the PIC 16 Series



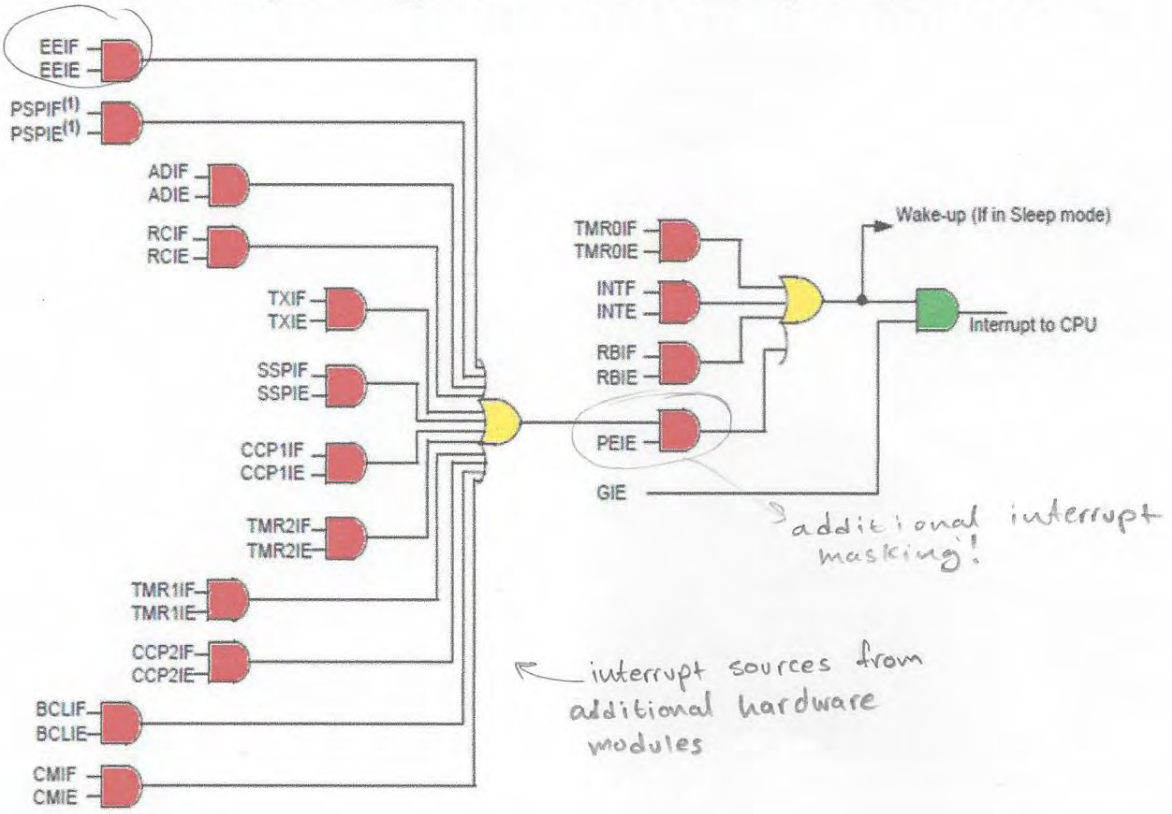
4 banks!

for interrupt enable & flag signals

	File Address		File Address		File Address		File Address
INDF	00h	INDF	80h	INDF	100h	INDF	180h
TMR0	01h	OPTION_REG	81h	TMR0	101h	OPTION_REG	181h
PCL	02h	PCL	82h	PCL	102h	PCL	182h
STATUS	03h	STATUS	83h	STATUS	103h	STATUS	183h
FSR	04h	FSR	84h	FSR	104h	FSR	184h
PORTA	05h	TRISA	85h		105h		185h
PORTB	06h	TRISB	86h	PORTB	106h	TRISB	186h
PORTC	07h	TRISC	87h	PORTF	107h	TRISF	187h
PORTD	08h	TRISD	88h	PORTG	108h	TRISG	188h
PORTE	09h	TRISE	89h		109h		189h
PCLATH	0Ah	PCLATH	8Ah	PCLATH	10Ah	PCLATH	18Ah
INTCON	0Bh	INTCON	8Bh	INTCON	10Bh	INTCON	18Bh
PIR1	0Ch	PIE1	8Ch		10Ch		18Ch
PIR2	0Dh	PIE2	8Dh		10Dh		18Dh
TMR1L	0Eh	PCON	8Eh		10Eh		18Eh
TMR1H	0Fh	OSCCAL	8Fh		10Fh		18Fh
T1CON	10h		90h		110h		190h
TMR2	11h		91h		111h		191h
T2CON	12h	PR2	92h		112h		192h
SSPBUF	13h	SSPADD	93h		113h		193h
SSPCON	14h	SSPSTAT	94h		114h		194h
CCPR1L	15h		95h		115h		195h
CCPR1H	16h		96h		116h		196h
CCP1CON	17h		97h		117h		197h
RCSTA	18h	TXSTA	98h		118h		198h
TXREG	19h	SPBRG	99h		119h		199h
RCREG	1Ah		9Ah		11Ah		19Ah
CCPR2L	1Bh		9Bh		11Bh		19Bh
CCPR2H	1Ch		9Ch		11Ch		19Ch
CCP2CON	1Dh		9Dh		11Dh		19Dh
ADRES	1Eh		9Eh		11Eh		19Eh
ADCON0	1Fh	ADCON1	9Fh		11Fh		19Fh
General Purpose Registers (2)	20h – 7Fh	General Purpose Registers (3)	A0h – EFh	General Purpose Registers (3)	120h – 16Fh	General Purpose Registers (3)	1A0h – 1EFh
		Mapped in Bank0 70h - 7Fh (4)	F0h – FFh	Mapped in Bank0 70h - 7Fh (4)	170h – 17Fh	Mapped in Bank0 70h - 7Fh (4)	1F0h – 1FFh
Bank0		Bank1		Bank2 (5)		Bank3 (5)	

Overview of the PIC 16 Series

Interrupt Logic for 16F874A/16F877A



Overview of the PIC 16 Series

Device	Pins	Features
16F873A 16F876A	28	3 parallel ports, 3 counter/timers, 2 capture/compare/PWM, **2 serial,** 5 10-bit ADC, 2 comparators
16F874A 16F877A	**40**	**5** parallel ports, 3 counter/timers, 2 capture/compare/PWM, **2 serial,** **8** 10-bit ADC, 2 comparators

The 16F87xA USART

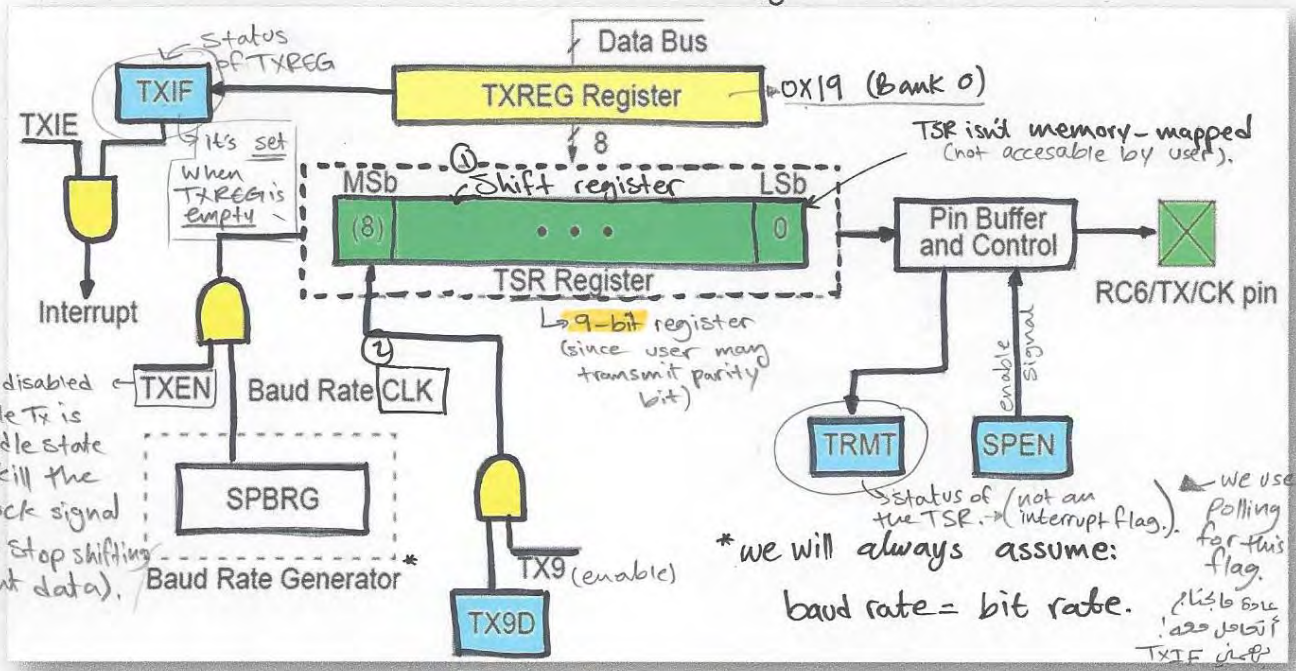
- The 16F87XA family has a Universal Synchronous Asynchronous Receiver Transmitter (USART)
  - Configurable
  - Half duplex synchronous master or slave
  - Full-duplex asynchronous transmitter and receiver
- The USART shares pins with PORTC (multiplexed pins).
  - pin 7 being the receive line RC7/Rx
  - pin 6 being the transmit line RC6/Tx
- Operation involves the following registers

TXSTA (0x98) TXREG (0x19) RCSTA (0x18)
RCREG (0x1A) SPBRG (0x99) PIE1 (0x8C)
PIR1 (0x0C) INTCON (0x0B, 0x8B,0x10B,0x18B)
TRISC (0x87)

The 16F87xA USART

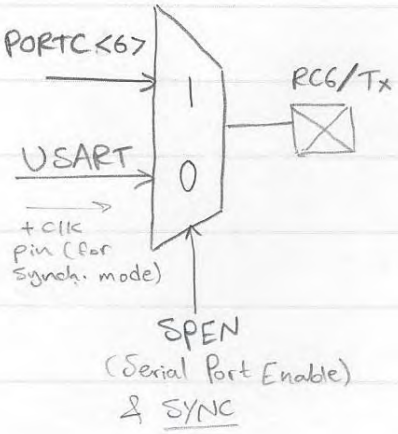
- **Asynchronous USART Transmitter Block Diagram**

basic elements of a transmitter: ①Shift register. ②clock.



* Parity bit must be computed by the programmer (by software) ← it's not done automatically.

* Start and Stop bits are generated automatically by the module, you only have to send the data (& parity)



* Operation of the USART is controlled by 2 registers
  → the port is enabled by the SPEN bit in RCSTA
  → the selection of synchronous / asynch. modes is done by the SYNC bit of the TXSTA reg.

The 16F87xA USART

- **Asynchronous USART Transmitter Operation Notes**
  - Data is transmitted LSB first on RC6 pin
  - The shift register TSR is buffered by the TXREG (19H) and is not accessible as a memory location (why?) to speed up the process; You can write on TXREG while the TSR reg. is shifting out data. (2 words: 1 at TSR being shifted out & the other in TXREG waiting to be buffered to TSR)
  - Transmission is controlled by the TXEN bit which enables the clock to start the transmission
  - To enable serial transmission on RC6, bit SPEN in RCSTA register has to be set (Serial Port ENable)
  - To transmit data, it must be loaded in the TXREG. It is transferred to TSR immediately if no transmission or after the stop bit from previous transmission is sent out
  - Transmission status is provided by two bits:
    - TXIF flag in PIR1 register indicates the status of TXREG. It is set when data is transferred to TSR. It is cleared on writing to TXREG. (TXIF is cleared by hardware and it is read-only). (& you don't have to clear it manually during ISR.)
    - TRMT flag in TXSTA it is set when the shift register is empty
  - Parity bit can be sent out by using ~~TXD9~~ TX9D bit and TX9 in TXSTA

↳ parity bit should be set up before its associated data word is written to TXREG. If this isn't done, then a transfer (write) to TXREG will start serial transmition before the 9th bit is in place.

The 16F87xA USART

TXSTA (98H)

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
bit 7							bit 0

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode:
Don't care.
Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9:** 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN:** Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled
Note: SREN/CREN overrides TXEN in Sync mode.

bit 4 **SYNC:** USART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

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

bit 2 **BRGH:** High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode.

bit 1 **TRMT:** Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of Transmit Data, can be Parity bit

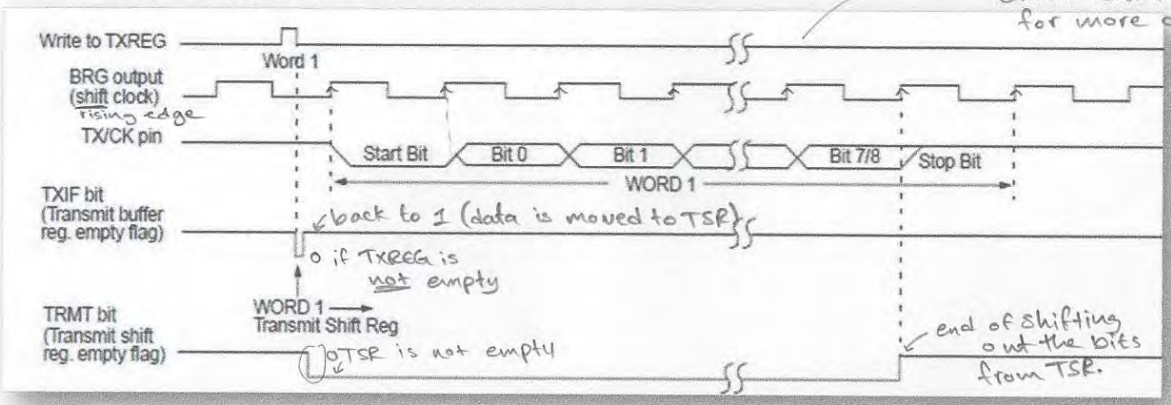
The 16F87xA USART

- **Steps for Using the asynchronous transmitter**

1. Clear TRISC<6> bit to configure RC6 as output
2. Set the SPBRG (0x99) register and BRGH (TXSTA<2>) bit to choose the appropriate baud rate (more on this later)
3. Enable asynchronous serial port by clearing the SYNC (TXSTA<4>) bit and setting the SPEN bit (RCTSA<7>)
4. If interrupts are desired, set the TXIE (PIE1<4>), GIE (INTCON<7>), and PEIE (INTCON<6>) bits
5. If 9-bit transmission is desired, set the TX9 (TXSTA<6>) bit
6. Enable transmission by setting the TXEN (TXSTA<5>), which will set the TXIF (PIR1<4>) bit
7. If 9-bit transmission is selected, then the ninth bit should be loaded in TX9D (TXSTA<0>)
8. Load data in TXREG (0x19) to start the transmission

The 16F87xA USART

- **Timing of asynchronous transmission**

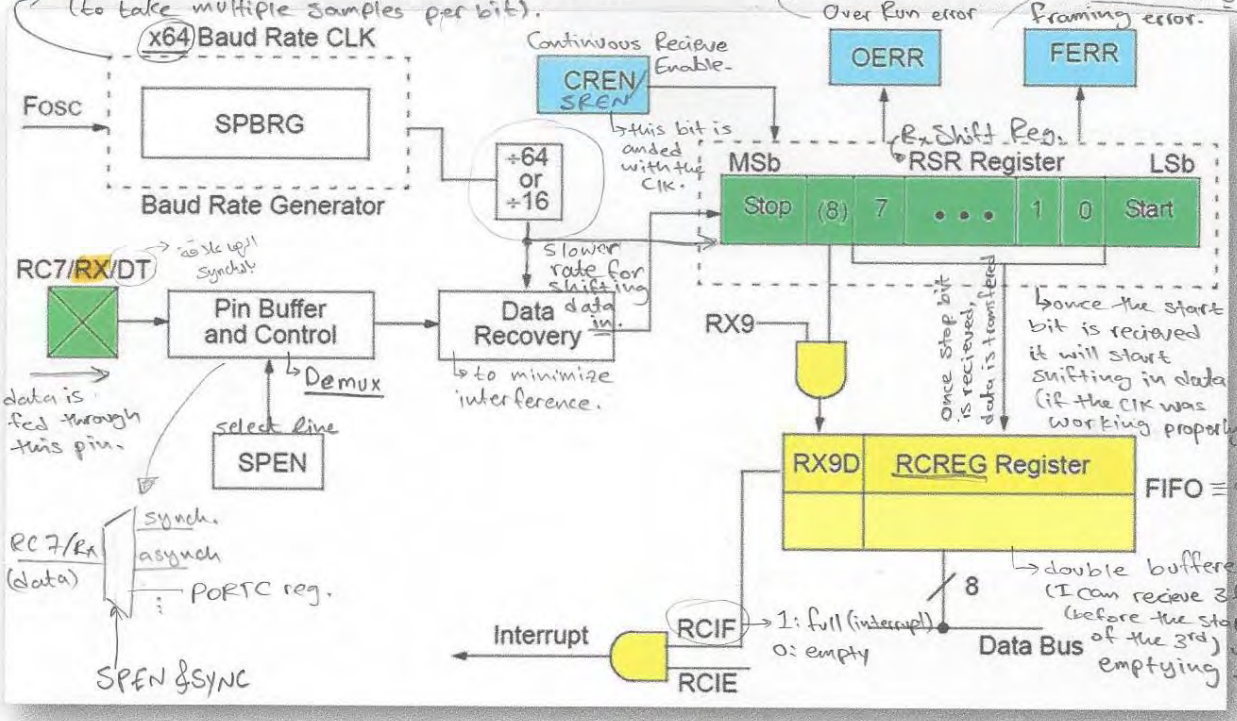


- **Registers involved in asynchronous transmission**

Address	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
0Bh, 8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	R0IF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D	0000 -00x	0000 -00x
19h	TXREG	USART Transmit Register								0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000

The 16F87xA USART

- **Asynchronous Receiver** (a mirror image of Tx!)



* RCREG is a buffer to the RSR reg.
* in Tx → TSR was a buffer for TXREG

The 16F87xA USART

Stop bit = 1.

- **Asynchronous USART Receiver Operation Notes**
  - Data is received LSB first on RC7 pin
  - Reception is enabled by the CREN bit
  - At the heart of the block is the RSR register. Once a stop bit is detected, data is transferred to RCREG register, if it is empty, and the RCIF flag is set. (RCIF is cleared by hardware and it is read-only). On-receive interrupt can be enabled by RCIE bit
  - The RCREG is FIFO double buffered register
    - can be used to receive bytes while reception continues in RSR
    - It can be read twice to read the received two bytes
    - If a stop bit (of the 3rd word) is detected in RSR and the RCREG is still full, an overrun error occurs and is indicated in OERR bit (The word is RSR is lost)
    - If OERR bit is set, shifting stops in RSR and transfers to the RCREG ~~is~~ are inhibited !
    - To clear the overrun error, clear the CREN bit.
  - If the stop bit is received as clear in RSR a framing error occurs and is indicated by the FERR bit.
  - The 9th bit of data RX9D and FERR are also double buffered. It is essential to read the RCSTA register before the RCREG to avoid losing the corresponding values of RX9D and FERR

((parity must be checked by user (software)).

The 16F87xA USART

RCSTA (18H)

Read-only. because parity is determined by Tx.

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R-0	R-0	R-0
SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (Configures RX/DT and TX/CK pins as serial port pins)
0 = Serial port disabled

bit 6 **RX9:** 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN:** Single Receive Enable bit
Asynchronous mode
Don't care
Synchronous mode - master
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode - slave
Unused in this mode

bit 4 **CREN:** Continuous Receive Enable bit
Asynchronous mode
1 = Enables continuous receive
0 = Disables continuous receive
Synchronous mode
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

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

bit 2 **FERR:** Framing Error bit
1 = Framing error (Can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR:** Overrun Error bit
1 = Overrun error (Can be cleared by clearing bit CREN)
0 = No overrun error

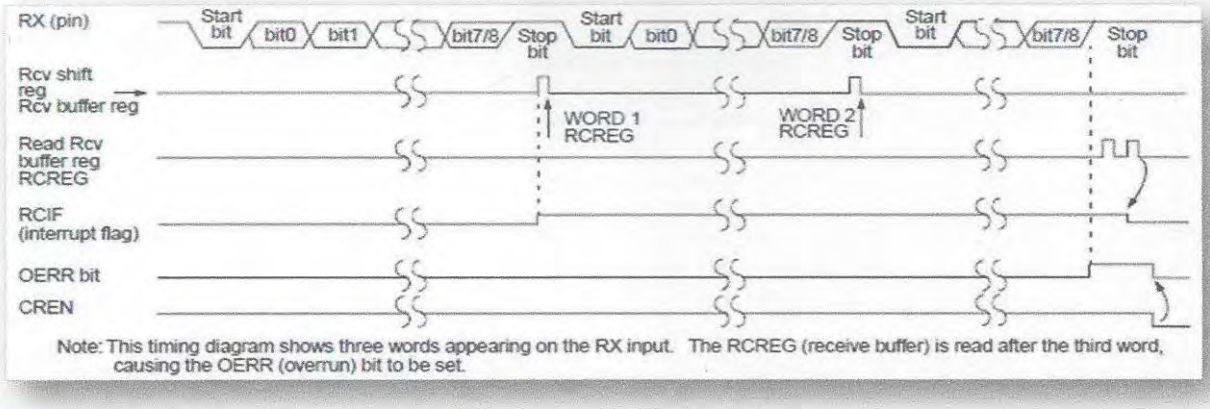
bit 0 **RX9D:** 9th bit of received data, can be parity bit.

The 16F87xA USART

- **Steps for Using the asynchronous receiver**
  1. Set the SPBRG (0x99) register and BRGH (TXSTA<2>) bit to choose the appropriate baud rate
  2. Enable asynchronous serial port by clearing the SYNC (TXSTA<4>) bit and setting the SPEN bit (RCTSA<7>)
  3. If interrupts are desired, set the RCIE (PIE1<5>), GIE (INTCON<7>), and PEIE (INTCON<6>) bits
  4. If 9-bit reception is desired, set the RX9 (RCSTA<6>) bit
  5. Enable the reception by setting bit CREN (RCSTA<4>)
  6. The RCIF (PIR1<5>) will be set when reception of one word is complete and an interrupt will be generated if RCIE is set
  7. Read the RCSTA (0x18) to get the 9th bit and determine if any error occurred (OERR, FERR)
  8. Read the 8-bit received data by reading RCREG (0x1A)
  9. If any error occurred, clear the error by clearing the CREN

The 16F87xA USART

- **Timing of asynchronous reception**

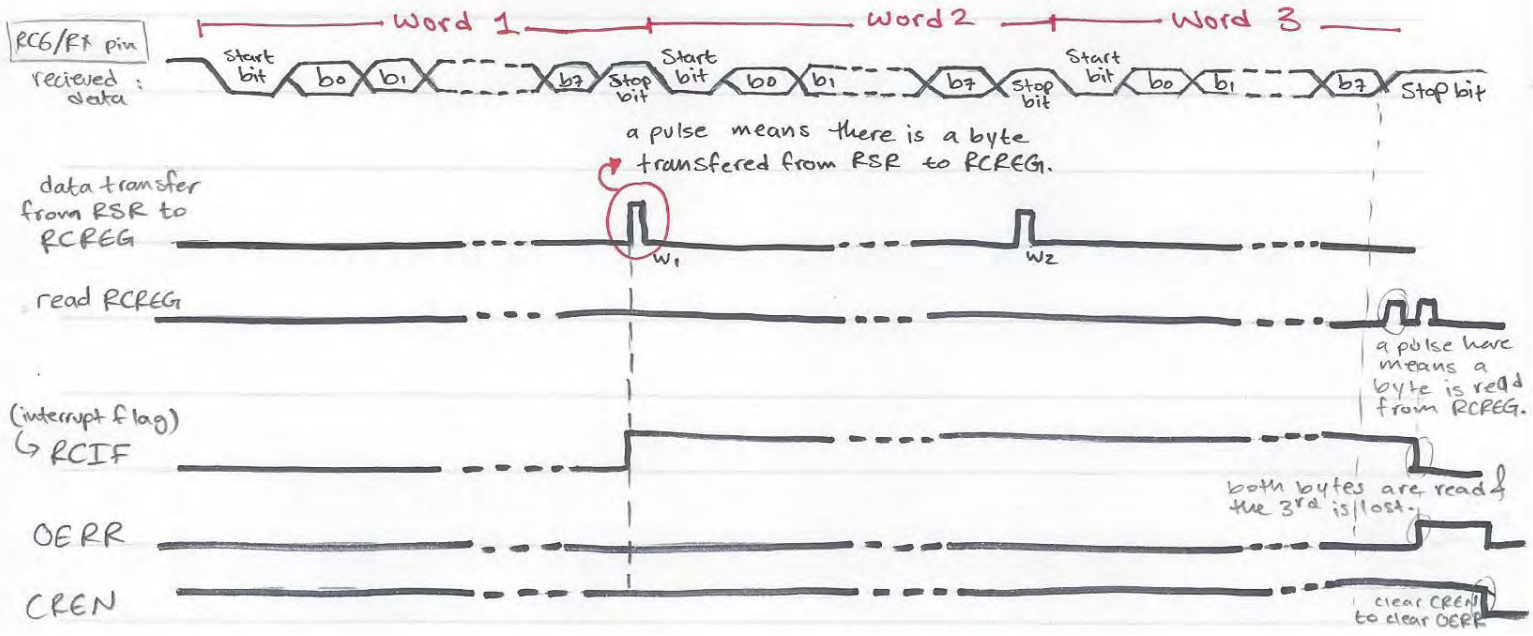


Note: This timing diagram shows three words appearing on the RX input. The RCREG (receive buffer) is read after the third word, causing the OERR (overrun) bit to be set.

- **Registers involved in asynchronous reception**

Address	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
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	R0IF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D	0000 -00x	0000 -00x
1Ah	RCREG	USART Receive Register								0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
98h	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99h	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000

* if an over run error occurs, The word in the RSR is lost. RCREG can be read twice to retrive the two bytes in the FIFO-reg. Over run bit OERR has to be cleared in software (it's a read-only bit, remember) & this is done by resetting the recieve enable (clear CREN bit, then set it again)



The 16F87xA USART

- **The BAUD Rate Generator**
  - The BAUD rate for USART is controlled by the value in the SPREG (99H), the SYNC and the BRGH bits in the TXSTA (19H)

Baude Rate Generator High.

SYNC	BRGH = 0	BRGH = 1
0 (asynchronous)	$\frac{F_{osc}}{64(SPBRG+1)}$	$\frac{F_{osc}}{16(SPBRG+1)}$
1 (synchronous)	$\frac{F_{osc}}{4(SPBRG+1)}$	

From data Sheet

TABLE 10-3: BAUD RATES FOR ASYNCHRONOUS MODE (BRGH = 0)

BAUD RATE (K)	Fosc = 20 MHz			Fosc = 16 MHz			Fosc = 10 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	-	-	-	-	-	-	-	-	-
1.2	1.221	1.75	255	1.202	0.17	207	1.202	0.17	129
2.4	2.404	0.17	129	2.404	0.17	103	2.404	0.17	64
9.6	9.766	1.73	31	9.615	0.16	25	9.766	1.73	15
19.2	19.531	1.72	15	19.231	0.16	12	19.531	1.72	7
28.8	31.250	8.51	9	27.778	3.55	8	31.250	8.51	4
33.6	34.722	3.34	8	35.714	6.29	6	31.250	6.99	4
57.6	62.500	8.51	4	62.500	8.51	3	52.083	9.58	2
HIGH	1.221	-	255	0.977	-	255	0.610	-	255
LOW	312.500	-	0	250.000	-	0	156.250	-	0

BAUD RATE (K)	Fosc = 4 MHz			Fosc = 3.6864 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	0.300	0	207	0.3	0	191
1.2	1.202	0.17	51	1.2	0	47
2.4	2.404	0.17	25	2.4	0	23
9.6	8.929	6.99	6	9.6	0	5
19.2	20.833	8.51	2	19.2	0	2
28.8	31.250	8.51	1	28.8	0	1
33.6	-	-	-	-	-	-
57.6	62.500	8.51	0	57.6	0	0
HIGH	0.244	-	255	0.225	-	255
LOW	62.500	-	0	57.6	-	0

TABLE 10-4: BAUD RATES FOR ASYNCHRONOUS MODE (BRGH = 1)

BAUD RATE (K)	Fosc = 20 MHz			Fosc = 16 MHz			Fosc = 10 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	-	-	-	-	-	-	-	-	-
1.2	-	-	-	-	-	-	-	-	-
2.4	-	-	-	-	-	-	2.441	1.71	255
9.6	9.615	0.16	129	9.615	0.16	103	9.615	0.16	64
19.2	19.231	0.16	64	19.231	0.16	51	19.531	1.72	31
28.8	29.070	0.94	42	29.412	2.13	33	28.409	1.36	21
33.6	33.784	0.55	36	33.333	0.79	29	32.895	2.10	18
57.6	59.524	3.34	20	58.824	2.13	16	56.818	1.36	10
HIGH	4.883	-	255	3.906	-	255	2.441	-	255
LOW	1250.000	-	0	1000.000		0	625.000	-	0

BAUD RATE (K)	Fosc = 4 MHz			Fosc = 3.6864 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	-	-	-	-	-	-
1.2	1.202	0.17	207	1.2	0	191
2.4	2.404	0.17	103	2.4	0	95
9.6	9.615	0.16	25	9.6	0	23
19.2	19.231	0.16	12	19.2	0	11
28.8	27.798	3.55	8	28.8	0	7
33.6	35.714	6.29	6	32.9	2.04	6
57.6	62.500	8.51	3	57.6	0	3
HIGH	0.977	-	255	0.9	-	255
LOW	250.000	-	0	230.4	-	0

Example

* check example in the website

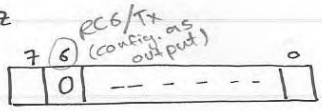
A program to transmit 3 bytes stored in locations 0x40, 0x41, and 0x42 serially with no parity at a rate of 9.6 Kbps. Assume PIC 16F877A with oscillator frequency of 20 MHz

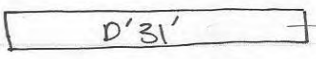
Requirements

1. setup the serial port for transmission
2. choose the appropriate value of SPBRG and BRGH to produce the required rate

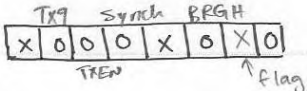
Example: (configuration)

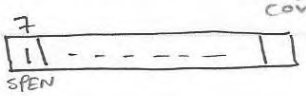
No parity
9600 bps
Fosc = 20MHz

TRISC 

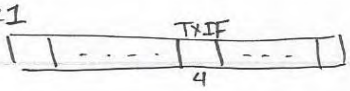
SPBRG 

→ for 9600 bps (with BRGH=0)

TXSTA 

RCSTA 

; enable serial transmition

we care about the TXIF flag (we'll use polling). PIR1 

INTCON x PIE x (we don't want an interrupt).

TXSTA is 0 by default ✓ ready.

Example

- **Example**

```
                #include  p16F877A.inc     ; include the definition file for 16F77A
                org       0x0000          ; reset vector
                goto      START
                org       0x0004          ; define the ISR
ISR             goto      ISR
                org       0x0006           ; Program starts here
START           bsf       STATUS , RP0
                bcf       STATUS , RP1    ; select bank 1
                bcf       TRISC, 6        ; set RC6 as output
                movlw     D'31'
                movwf     SPBRG           ; set the SPBRG value
                bsf       TXSTA, TXEN
                bcf       STATUS, RP0     ; select bank0
                bsf       RCSTA, SPEN     ; enable serial transmission
                movlw     0x40
                mowf      FSR             ; FSR has the address of the first element
```


Example

```
TX              movf      INDF, W     ; read byte to transmit
                movwf     TXREG       ; store in the transmission register
                incf      FSR, F      ; increment FSR to point to next address
WAIT            btfss     PIR1, TXIF  ; check if the TXREG is empty
                goto      WAIT
                movf      FSR,W
                sublw     0x43
                btfss     STATUS, Z   ; check if all values were transmitted
                goto      TX

DONE            goto      DONE
                end
```

Summary

- Serial communication transmits bits one after another in two modes: synchronous and asynchronous
- Stable and accurate clocking plays an important role in serial communication
- It is cheaper to use serial communication over long distances
- Some members of the 16 series are equipped with synchronous and asynchronous communication ports
- These ports can be configured to operated in different modes and rates

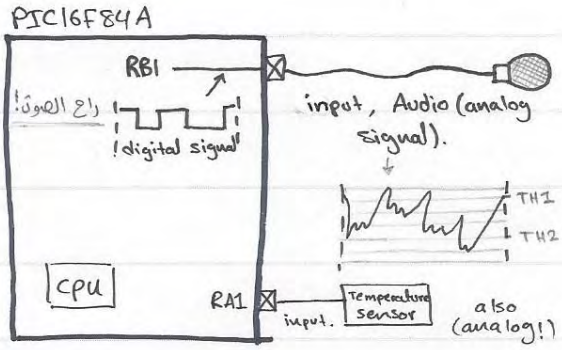
Data Acquisition and Manipulation

Chapter 11
Sections 1 - 3

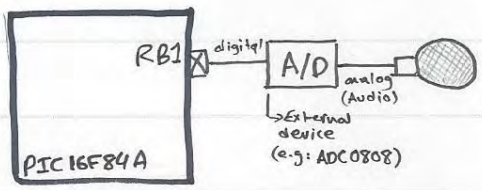
Dr. Iyad Jafar

Outline

- Analog and Digital Quantities
- The Analog to Digital Converter
- Features of Analog to Digital Converter
- The Data Acquisition System
- The 16F873 ADC
- Summary



→ PIC16F84A has only digital I/O pins.
→ Digital I/O pins can be used to interface with analog devices.
* Solution: use an external A/D converter.



Problem? we need to buy external A/D IC & interface it properly with the Micro controller.
* Solution: upgrade your μcontroller to another one that has an internal A/D, such as PIC16F778.

Analog and Digital Quantities

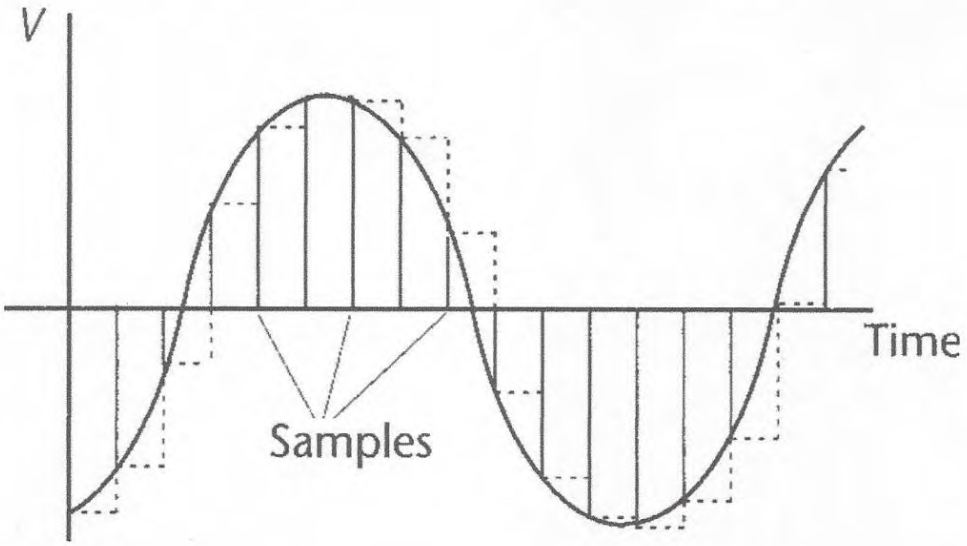
- Most signals produced by transducers are analog; continuously variable in time and can take infinite range of values
- Digital signals are *discrete representation* for the analog signals *in time and value*
- Digital signals perform better and are easier to work with
- Analog signals have to be converted into digital form in order to be processed by the microcontroller
- The device that performs this conversion is called Analog to Digital Converter (ADC)

Analog and Digital Quantities

Property	Analog	Digital
Representation	Continuous voltage or current	Binary Number
Precision	Infinite range of values	Only fixed number of digits combination are available
Resistance to Degradation	Suffers from drift, attenuation, distortion, interference. Recovery is hard	Tolerant to most forms of signal degradation. Error checking can be included for complete recovery
Processing	Processing using op amps and other sophisticated circuits. Limited, complex, and suffers from distortion	Powerful computer-based techniques
Storage	Analog storage for any length of time is almost impossible	All semiconductor memory techniques are digital

The Analog to Digital Converter

- Conversion to digital form requires two steps
  - Sampling
  - Quantization



→ Why do we need to convert analog signals to digital?

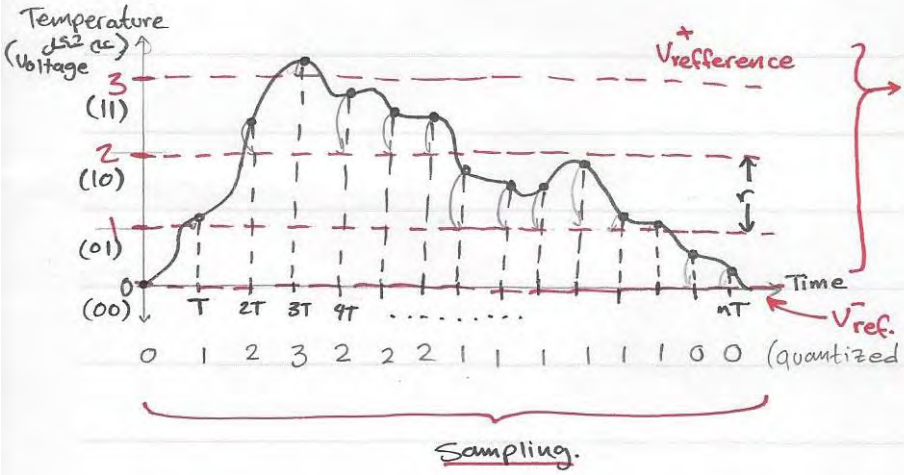
because devices used to process these signals have limited processing capabilities & limited storage (limited # of data words to be stored & limited length of each word). While analog quantities have infinite # of points (samples) in time & infinite possible values in magnitude (e.g: temperature)

↳ solved by Sampling.

↳ solved by Quantization

analog → Sampling → Quantization → encoding → digital

A/D conversion.



Quantization & truncation of values.
(divide input range into Quantum levels).

0 1 2 3 2 2 2 1 1 1 1 1 0 0 (quantized voltage values).

Sampling.

(Sampling rate $f_s = \frac{1}{T_s}$, T_s: Sample time).

→ (4) possible values need (2) bits to encode: (M, $\log_2 M$)

0	00
1	01
2	10
3	11

encoding

* In this case max. Quantization error

$Q_{e_{max}} = r$ (step size) $= 1$

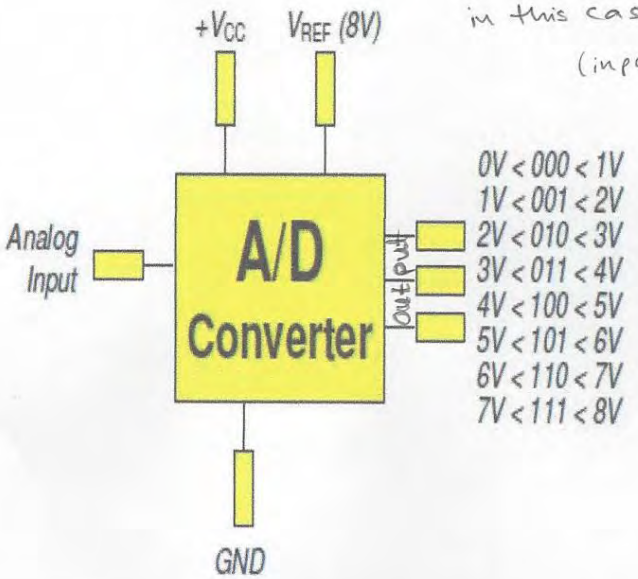
$\therefore \boxed{0 \le Q_e \le 1}$

anything outside this range will be clipped. ; Q_e = actual value − converted value.

$$\text{Stepsize } (r) = \frac{V^+_{ref} - V^-_{ref}}{M} = \frac{V^+ - V^-}{2^n} \equiv \frac{\text{input range}}{2^n}, \quad n: \text{\# of bits used.}$$

Features of Analog to Digital Converter

- **Conversion Characteristics**
  - The ADC accepts a voltage that is infinitely variable and converts it to one of a fixed number of output values

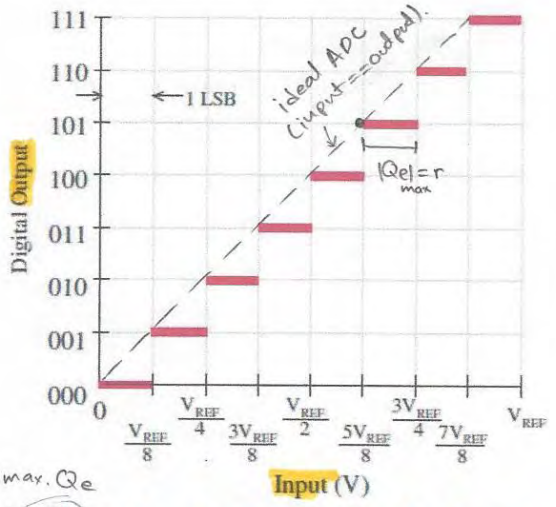


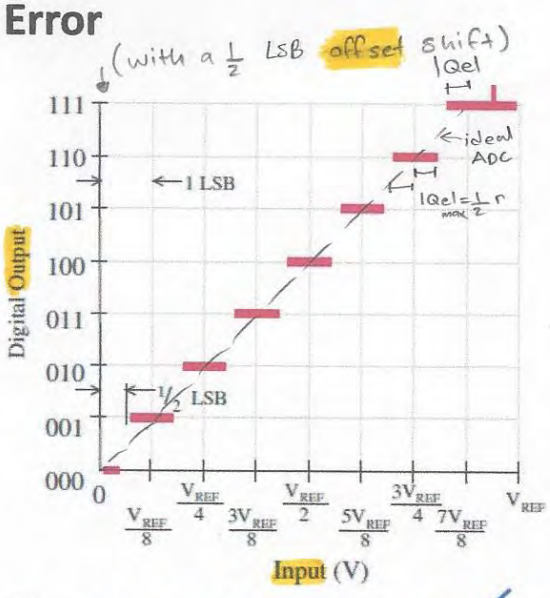
in this case $V^-_{ref} = 0$ (connected to GND.)
(input range = 8 − 0 = 8V)
2^3 = 8 levels)

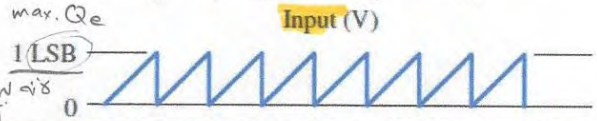
Features of Analog to Digital Converter

- **Conversion Characteristics**

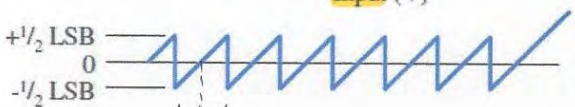
Quantization Error







The Magnitude of the Error Ranges from Zero to 1 LSB



∴ |Qel max| = ½ r

* In slide #6, if we used quantization with ½ LSB offset, output values will be as follow:

$0 \leq 000 < \frac{1}{2}V$

$\frac{1}{2}V \leq 001 < 1\,1/2\,V$

$1\,1/2\,V \leq 010 < 2\,1/2\,V$

$2\,1/2\,V \leq 011 < 3\,1/2\,V$

$3\,1/2\,V \leq 100 < 4\,1/2\,V$

⋮

∴ maximum Quantization Error

$|Qe_{max}| = \frac{1}{2}V$

→ for an n-bit ADC:

Resolution (R) ≡ Stepsize $= \frac{V^+ - V^-}{2^n}$

↓ R ↑ better performance

$Qe_{max} = R$ (without offset)

$Qe_{max} = \frac{R}{2} \equiv \left(\frac{V^+ - V^-}{2^{n+1}}\right)$ (with offset) in I/O curve.

Example: 10-bit ADC, $V^- = -2V$, $V^+ = 4V$

1. What is the input range? $V_r = V^+ - V^- = 4 - (-2) = 6V$

2. find R? $R = \frac{V_r}{2^n} = \frac{6}{2^{10}} = 5.86\,mV$

3. $|Q_e|_{max}$ without offset $= R = 5.86mV$
   " with " $= R/2 = 2.93mV$

4. if the digital value of some sample was $(0000\ 011110)_2$, then what's the value of the sample?

$(00\ 0001\ 1110)_2 \equiv (2+4+8+16)_{10} \equiv (30)_{10}$ ← this represents # of levels starting from -2V

$Sample\ Value = V^- + \#of\ levels * R$

$= -2 + 30 \times 5.86mV = (-1.824)V$

So, if you want to know the value, you have to do some math inside the μcontroller.

actual value = Sample value $- Q_e$

→ cont.

5. if the input value = 3.5 V, then what's the digital output?

$3.5V = -2 + \#of\ levels \times 5.86mV$

$\#of\ levels = \lfloor 939.5 \rfloor = 939$ (Q_e without offset).

↖ floor

$(939)_{10} = \boxed{(11\ 1010\ 1011)_2}$ ← digital output

#

* think about what will change if used a quantization with offset.
(R is the same).

↳ sample value won't change! only Q_e.

↳ actual Value $= \left(V_{ref.}^- + (\#of\ levels) \times R\right) - Q_e$ ← sample value

Features of Analog to Digital Converter

- **Reference voltages [V_{min},V_{max}]**
  - Determine the acceptable range of input analog voltage
  - Out of range input values are clipped
  - *Unipolar or bipolar*
  - Should be stable and accurate for proper operation
  - ***Input range*** $V_r = V_{max} - V_{min}$

- **Resolution**
  - The amount by which the input voltage has to change to go from one output value to another
  - The more the output bits the more the output steps and finer is the conversion
  - ***Resolution*** $= V_r / 2^n$
  - ***Quantization error Q = resolution / 2***

Features of Analog to Digital Converter

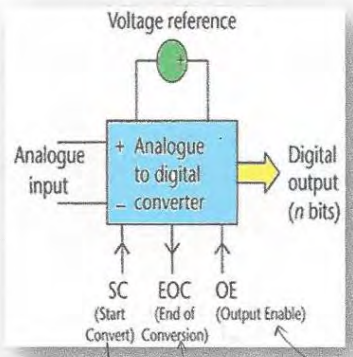
- **Conversion Characteristic**

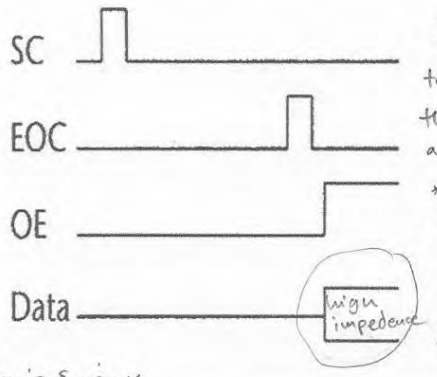
Quantization error as a function of ADC bits

n	No. of quantisation levels	Max. quantisation error as % of range	Quantisation error for range of 5 V
3	8	6.25	312.50 mV
4	16	3.13	156.25 mV
5	32	1.56	78.13 mV
6	64	0.781	39.06 mV
8	256	0.195	9.77 mV
10	1 024	0.0488	2.44 mV
12	4 096	0.0122	0.61 mV
16	65 536	0.00076	38.1 µV

Features of Analog to Digital Converter

- **Conversion Speed** $\propto \frac{1}{T_{conv}}$
  - Time for the ADC to do the conversion
  - Slow ADCs are used with low frequency signals
  - High accuracy ADCs take longer to complete conversion
- **Digital Interface**
  - Made up of control signals and data outputs
  - Data outputs – serial or parallel





* OE control signal is used to "three-state" the data bus (as a tri-state buffer)
* when OE is 0 (low): data from the ADC will be available on the data bus.
* when OE = 1 (high) output is forced to high-impedence state

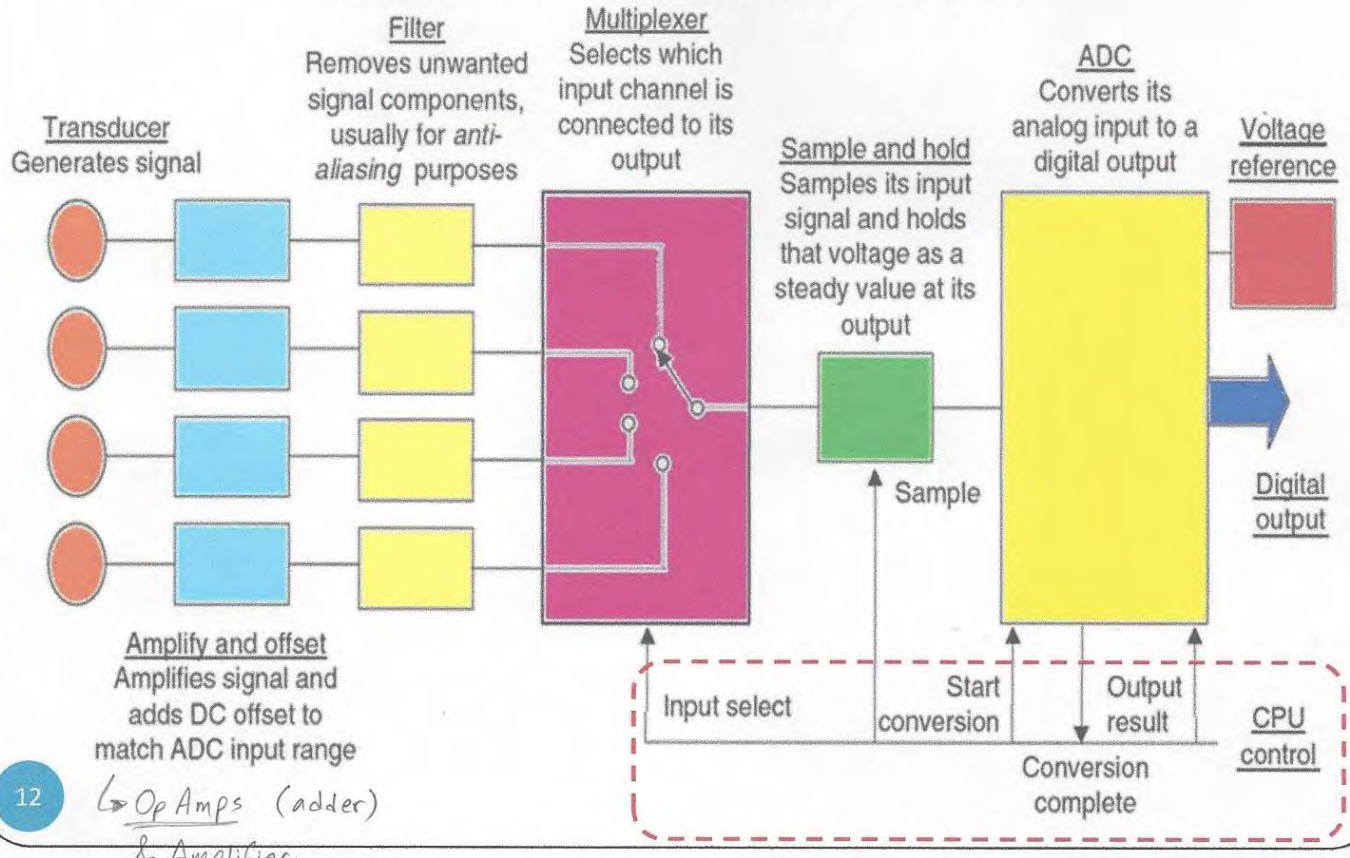
enable
flag
* مفيد في حال كان هناك اكثر من جهاز مشبوك على الـ data bus حتى نقرأ من جهاز ونفصل البقية.

The Analog to Digital Converter

- **ADC Types** (depending on the type of algorithms & implementation).
  - *Dual Ramp ADC*
    - Slow but with high accuracy
  - *Flash Converter ADC*
    - Fast but less accuracy
    - Used with high speed signals such as video and radar
  - *Successive Approximation ADC*
    - Medium speed and accuracy
    - Used in general-purpose industrial applications
    - *Commonly found in embedded systems*

The Data Acquisition System

Elements of data acquisition system



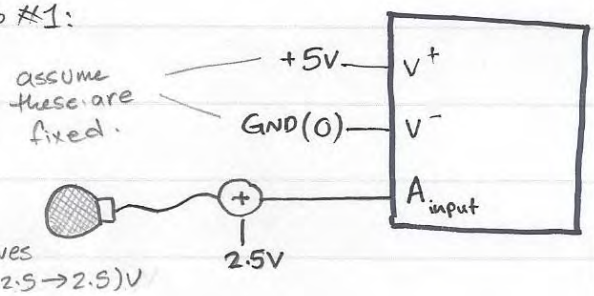
↳ Op Amps (adder) & Amplifier.

- Why do we need to offset the signal?

→ to match the input range of the ADC. ($V^+ \to V^-$) * since any value $< V^-$ or $> V^+$ will be clipped.

take the following scenario:

Scenario #1:



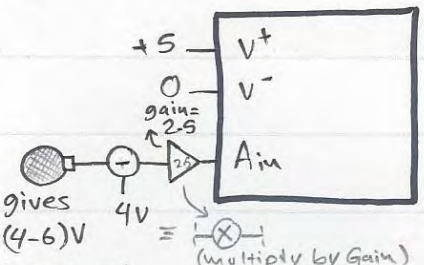
- problem?

-ve values will be clipped ($< V^-$)

→solution:

offset the signal by 2.5V

Scenario #2:



2problems!

1) 5→6 values will be clipped. ($> V^+$) → offset by -4

2) after the offset, the resulting range will be 0→2

So, I'm not exploiting the full range (0→5)

→ solution: amplify the signal buy (2.5)

solution

* Filtering is done for two major reasons:

1) to remove the noise.

2) to limit the signal's BandWidth. (remove some freq. component that are actually part of the signal) → to match the speed of the A/D (avoid to aliasing)

↳ Sampling theorem states that a signal can be reconstructed exactly from its samples if the samples were taken at a rate equals at least twice the BW of the signal.

$F_s \geq BW * 2$ → (nyquest rate: at $F_s = BW * 2$)

* if $BW_{signal} = 10KHz$ → you have to sample at at least $20KHz$

*What if $BW_s = 10KHz$ & A/D cannot run at speed higher than $5KHz$?
We use a filter to limit the BW of the signal & match the speed of the ADC.

* e.g: if the speed (rate) of ADC is $1KHz$, then the faster signal it can take is $0.5KHz$, & if we want to share the ADC with multiple devices (e.g: 2 devices) the faster signal is $0.25KHz$ for each. & so on...

The Data Acquisition System

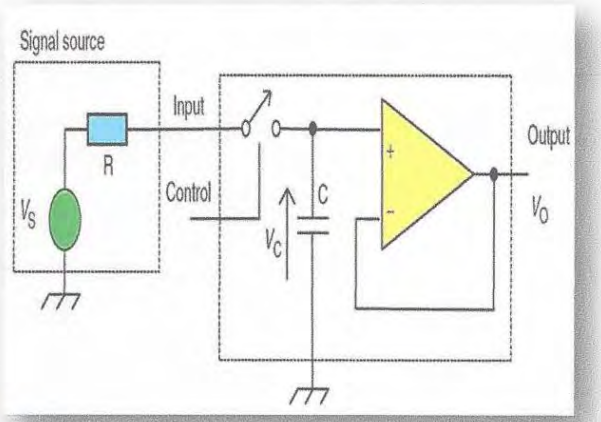
Elements of data acquisition system

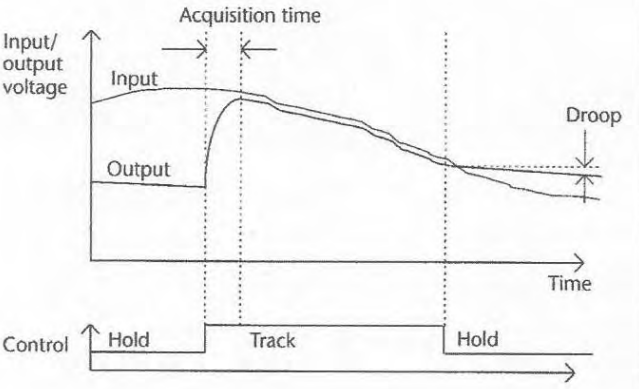
- **Amplification**
  - Most sensors produce low voltages
  - Need to amplify to exploit the input range of the ADC
  - Voltage level shifting might be needed for bipolar signals
- **Filtering**
  - Pick the actual signal and restrict its frequency content to the sampling rate of the ADC to avoid aliasing
  - Remove unwanted signals
- **Analog multiplexer** - [the select lines are digital, while input & output signals are analog.]
  - Used when working with multiple inputs instead of using multiple ADCs (multiple devices share the same ADC, [one at a time]). (which reduces the cost).
  - Semiconductor switches
    * disadvantage: you have to care about the speed

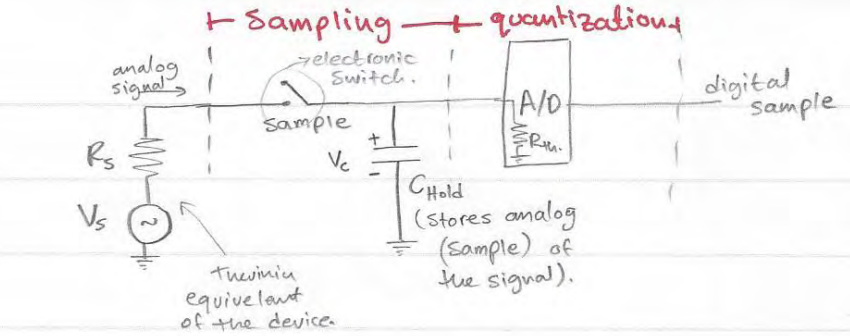
The Data Acquisition System

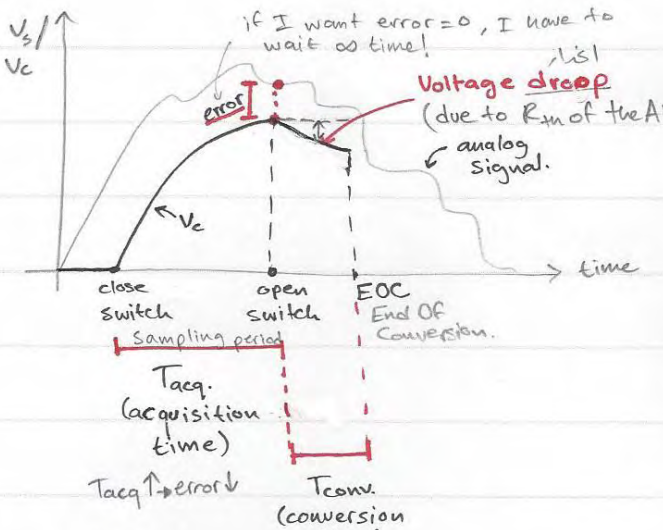
Elements of data acquisition system

- **Sample and Hold**
  - ADCs are unable to convert accurately a changing signal
  - We need to capture the sample value and hold it for the duration of the conversion process → we can't use latches since we're working with analog signals.
  - Acquisition time !









$T_{total} = T_{acq.} + T_{conv.}$

T_{acq} ↑ → error ↓

I choose the value of T_{acq}. base on the max. error I can tolerate.

example:-

if I want $V_c = 0.9 V_s$ ← Sampling Error.

$V_c = V_s (1 - e^{-t/RC})$

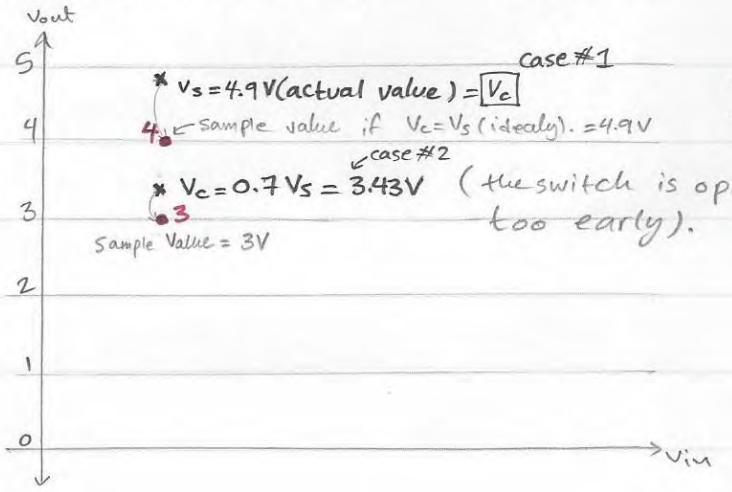
$0.9 V_s = V_s (1 - e^{-t/RC})$

$e^{-t/RC} = 0.1$

$t_{acq} = 2.3 R C_{Hold.}$

R: equivelant resistance of the device (e.g.: sensor...).

* We have two sources of error: 1. Quantization error. 2. Sampling error.



case1 → error = 0.9 V = Qe.

case2 → error = 1 + 0.9 = 1.9 V = $E_{acquisition} + Q_e$

* if the acquistion error is less than the step size it wont affect the total error.

So, we want $V_s - V_c \leq Q_e$ (hide the acquisition error whithin the quantization error).

take the extreme case: $V_s - V_c = Q_e$

$V_c = V_s - Q_e = V_s - \frac{V^+ - V^-}{2^{n+1}}$ → take the (curve with offset) case of quantization.

now, take the worst case: V_s is going from min to max. value during acq.

→ cont.

$$V_c = V_s\left(1 - \frac{1}{2^{n+1}}\right) = V_s\left(\frac{2^{n+1} - 1}{2^{n+1}}\right)$$

for n=10: $V_c = V_s\left(\frac{2047}{2048}\right) = \boxed{0.9995\, V_s}$

$0.9995 V_s = V_s\left(1 - e^{-t_{acq}/RC}\right)$ (V_c)

$T_{acq.} = \boxed{7.6\, R_s C_{Hold}}$ ← remember: this is for 10-bit ADC. only.

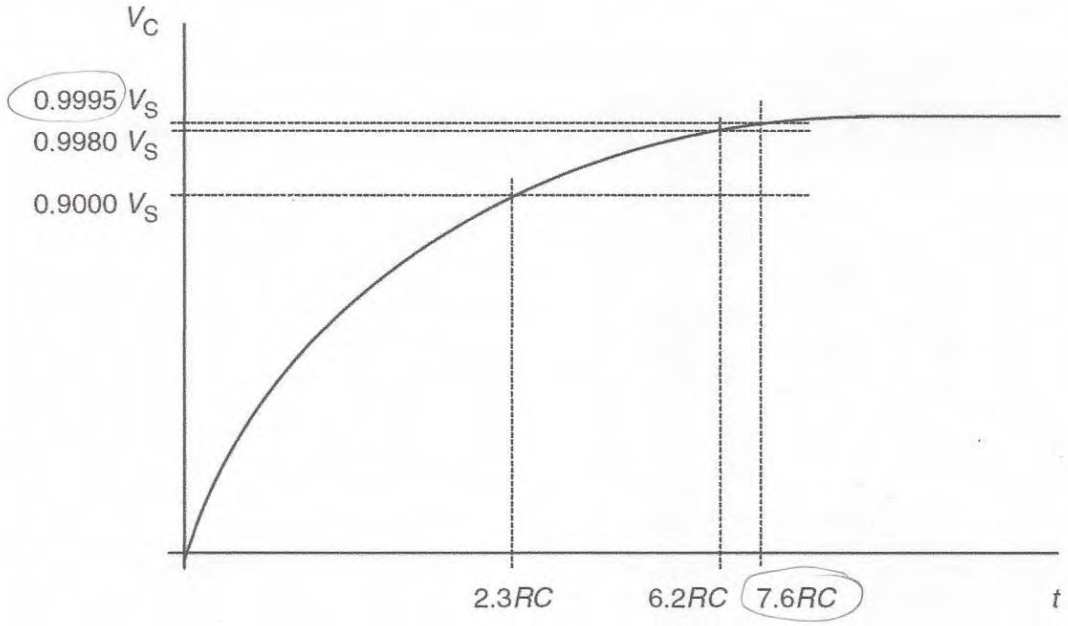
So, we have to wait $(7.6 R_s C_{Hold})$ sec. before opening the switch.

(after closing the switch.)

The Data Acquisition System

Elements of data acquisition system

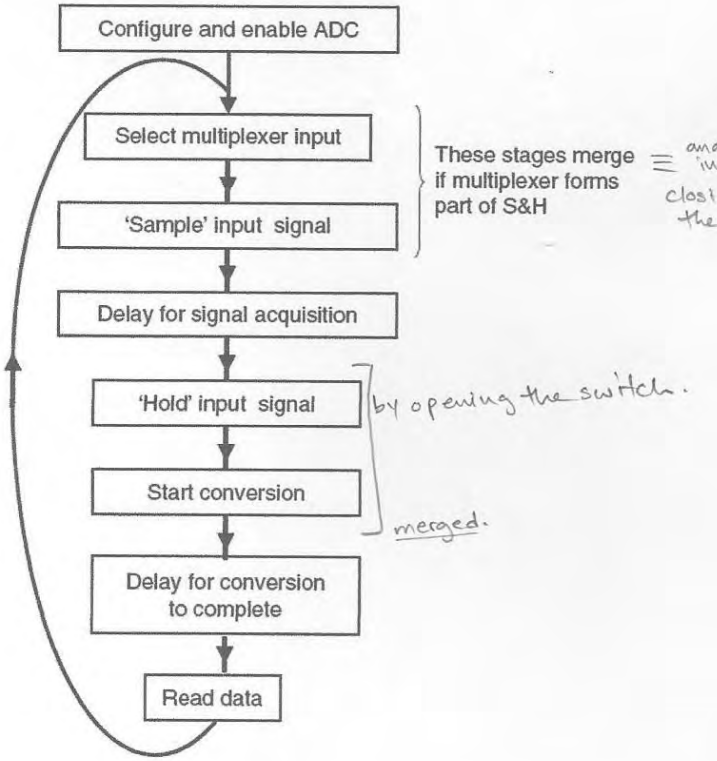
- **Sample and Hold**



Acquisition time increase as we increase the resolution of the ADC

The Data Acquisition System

Typical Timing Requirements for Analog to Digital Conversion



Data Acquisition in Microcontroller Environment

- Embedded systems need ADCs ; usually they are integrated within the MC as 8 or 10 bit ADCs
- Integration is not easy !
  - Proper operation of ADCs demands clean power supply and ground and freedom of interference
  - This is not easily available in digital devices
- Compromise accuracy of integrated ADCs !

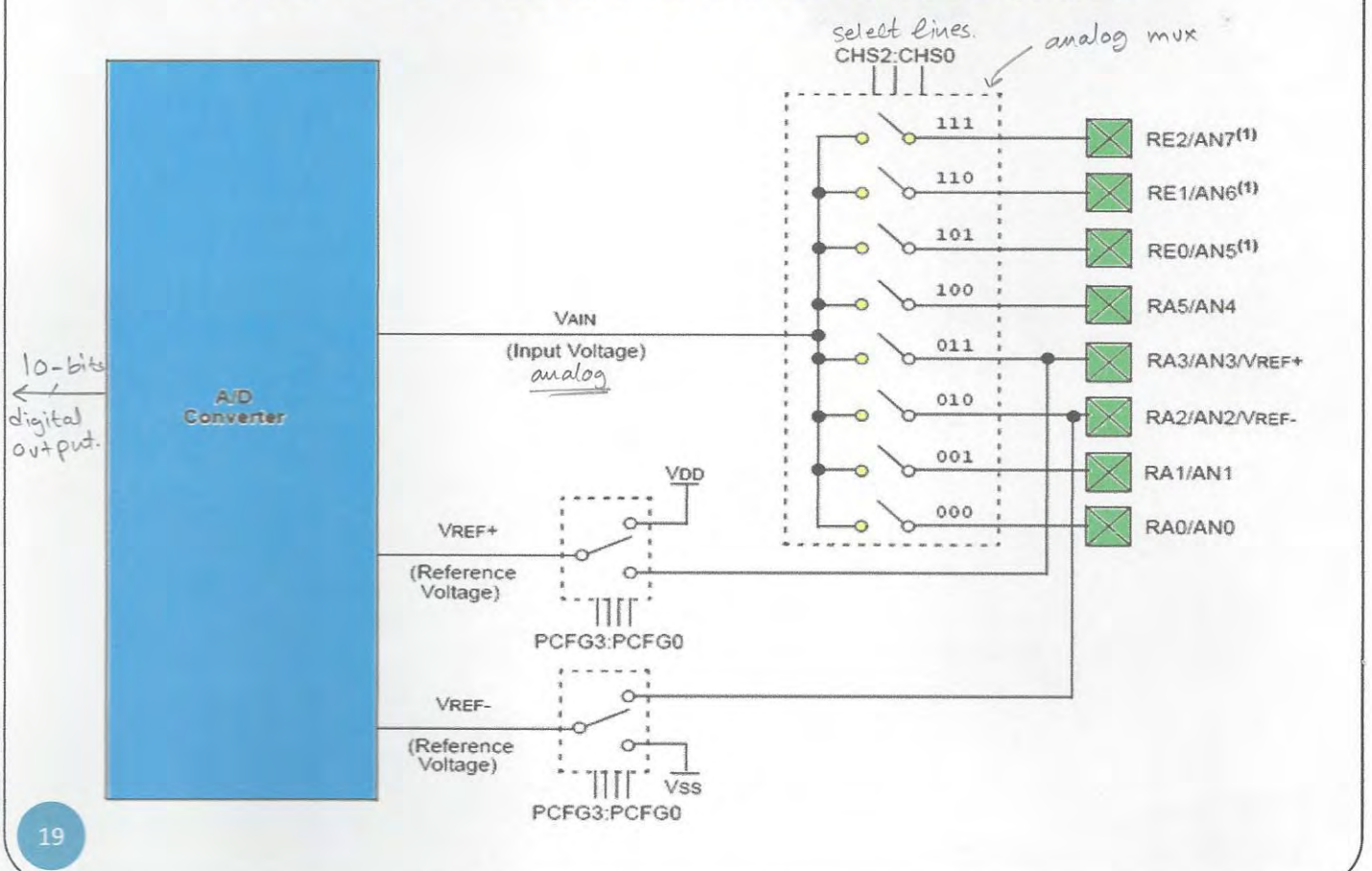
The PIC 16F87xA ADC Module

Device	Pins	Features
16F873A 16F876A	28	3 parallel ports, 3 counter/timers, 2 capture/compare/PWM, 2 serial, **5-10-bit ADC,** ≡ 5 channels 10-bit ADC ⇒ 5 choices for the analog mux. (5-to-1 max) 2 comparators
16F874A 16F877A	40	5 parallel ports, 3 counter/timers, 2 capture/compare/PWM, 2 serial, **8 10-bit ADC,** ≡ 8 channels 10-bit ADC. → (8-to-1 mux) 2 comparators

→ in PIC16F 873 / 876 → PORTA (5-bits) / AN0:AN4 (muxed with analog input)

→ in PIC16F 874 / 877 → PORTA (5-bits) / AN0:AN4
+ PORTE (3-bits) / AN5:AN7
RE0:RE2

The PIC 16F87xA ADC Module



The PIC 16F87xA ADC Module

Related Registers

- Operation is controlled by two SFRs
  - ADCON0 0x1F
  - ADCON1 0x9F
- Conversion result (10-bit) is placed in two SFRs
  - ADRESL 0x9E
  - ADRESH 0x1E
- ADC interrupts and flags are available in
  - PIE1 0x8C
  - PIR1 0x0C
- Related registers (I/O pins config.)
  - TRISA 0x85
  - TRISE 0x89 (in 40-pin devices)

The PIC 16F87xA ADC Module

Controlling the ADC

(1) Switching on

- The ADC is switched on/off by setting/clearing ADON bit (ADCON0<0>). (bit #0) (1: ON, 0: off)
- It is preferred to turn the ADC off when it is not needed as it offers some power saving.

(2) Setting Conversion Speed

- Operation of the ADC is governed by a clock with period T_{AD}. (the period of the clock that governs the conversion operation.)
- For correct conversions, T_{AD} must be 1.6 us at least.
- The ADC clock can be selected by software ($2T_{OSC}$, $4\ T_{OSC}$, $8\ T_{OSC}$, $16\ T_{OSC}$, $32\ T_{OSC}$, $64\ T_{OSC}$, or internal RC 2-4 us) (> 1.6 μs)
- Selection of ADC clock source is through ADCS2 (ADCON1<6>), ADCS1:ADCS0 (ADCON0<7:6>) ← 3-bits : 8 choices.
- If the system clock (Fosc) is fast (>500KHz), use it to derive the ADC clock. Otherwise, use the internal RC.

هاد واحد من الاستخدامات اللي بسببه حاطلينا ال main clk الها نفس ال freq لل CPU لأنها بدها freq أعلى $\left(\frac{F_{osc}}{2}\right)$ بينما لل CPU ← $\frac{F_{osc}}{4}$

* You have to optimize the speed of the clk with the speed of the signal for the minimum possible power consumption without loosing the data.

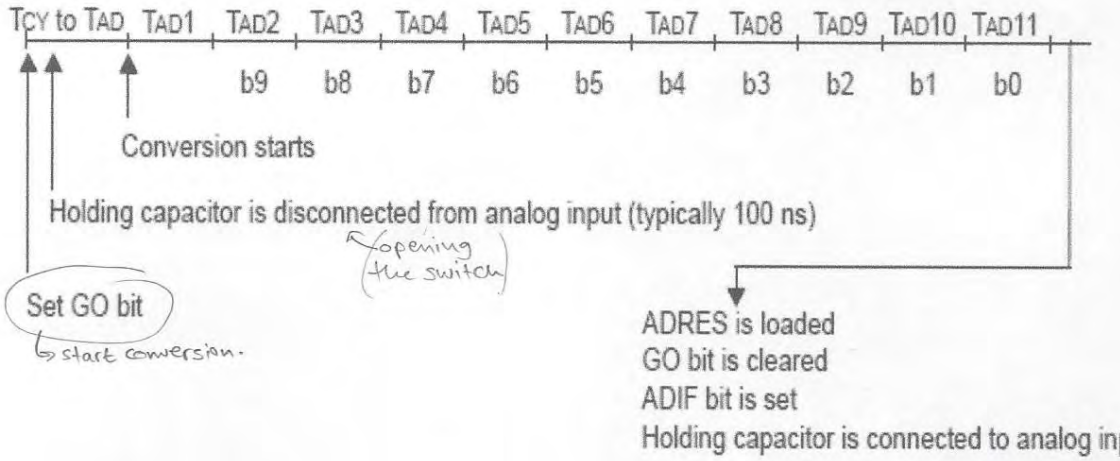
The PIC 16F87xA ADC Module

Controlling the ADC

Setting Conversion Speed

- A full 10-bit conversion requires 12 T_{AD}

$T_{conv.} =$



The PIC 16F87xA ADC Module

Controlling the ADC

(3) Configuring Inputs and Voltage Reference

- The ADCON1 and TRIS registers control the operation of the A/D port pins
- Inputs AN7 to AN0 can be configured as analog inputs or digital inputs.
- AN3 **(RA3)** and AN2 **(RA2)** can be used as the inputs for the external reference voltages separately
- Configuration is made through PCFG3:PCFG0 (ADCON1<3:0>)

(to configure the status of the pins [Analog / Digital / (V_{ref}^+, V_{ref}^- for RA2 & RA3)])

(4) Channel Selection

- We can select one out of five (or eight channels) as the analog input
- Use bits CHS2:CHS0 (ADCON0<5:3>)

(select lines for the analog mux.)

selection of the channel closes the sampling switch. (acquisition starts).

The PIC 16F87xA ADC Module

Controlling the ADC

(5) Starting Conversion and Flagging its End

- Conversion can be started by setting the GO/DONE' (ADCON0<2>) bit
- Once the conversion is complete, this bit is cleared to indicate the end of conversion
- *The GO/DONE' bit should not be set using the same instruction that turns on the A/D.* (Why?)

↳ 1. We have to wait for the A/D to startup properly. (transient time).
2. " " " " " the acquisition time.

* There 2 indicators for the end of conversion:-
1. DONE bit.
2. ADIF (interrupt flag).

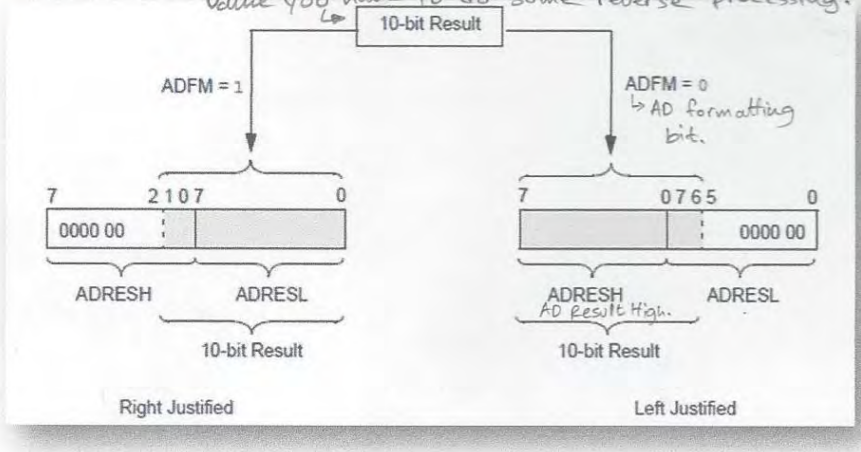
The PIC 16F87xA ADC Module

Controlling the ADC

(6) Formatting the result

- The ADC result is 10-bit data that is placed in ADRESH and ADCRESL (0x 1E and 0x9E respectively)
- The result can be left justified or right justified
- Selection of desired format is through the ADFM (ADCON1<7>) bit

this result represents the # of quantum levels starting from V⁻. So, to obtain the actual value you have to do some reverse processing.



بناءً على بقرر نوع الـ formatting اللي بدي ياه حسب الحسابات اللي عملتها على الـ result.

Be careful!

↳ ADRESH & ADRESL are in different Banks!

The PIC 16F87xA ADC Module

ADCON0 Register 0x1F

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/$\overline{\text{DONE}}$	—	ADON
bit 7							bit 0

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

ADCON1 <ADCS2>	ADCON0 <ADCS1:ADCS0>	Clock Conversion
0	00	Fosc/2
0	01	Fosc/8
0	10	Fosc/32
(0) x don't care	11	FRC (clock derived from the internal A/D RC oscillator) — the same
1	00	Fosc/4
1	01	Fosc/16
1	10	Fosc/64
1	11	FRC (clock derived from the internal A/D RC oscillator)

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits
- 000 = Channel 0 (AN0) RA0
- 001 = Channel 1 (AN1) RA1
- 010 = Channel 2 (AN2) ;
- 011 = Channel 3 (AN3) ;
- 100 = Channel 4 (AN4) RA4
- 101 = Channel 5 (AN5) RE0
- 110 = Channel 6 (AN6) RE1
- 111 = Channel 7 (AN7) RE2

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

When ADON = 1:
- 1 = A/D conversion in progress (setting this bit starts the A/D conversion which is automatically cleared by hardware when the A/D conversion is complete)
- 0 = A/D conversion not in progress

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

bit 0 **ADON:** A/D On bit
- 1 = A/D converter module is powered up
- 0 = A/D converter module is shut-off and consumes no operating current

The PIC 16F87xA ADC Module

ADCON1 Register 0x9F

by default 0000 all are analog with $V^+ = V_{DD}$ & $V^- = V_{SS}$ → So, take care when you want to use PORT A&E as digital I/O pins.

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

bit 7 **ADFM:** A/D Result Format Select bit
- 1 = Right justified. Six (6) Most Significant bits of ADRESH are read as '0'.
- 0 = Left justified. Six (6) Least Significant bits of ADRESL are read as '0'.

bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

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

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

Channels/Reference

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

The PIC 16F87xA ADC Module

Related Registers

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on MCLR, WDT
0Bh,8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
1Eh	ADRESH	A/D Result Register High Byte								xxxx xxxx	uuuu uuuu
9Eh	ADRESL	A/D Result Register Low Byte								xxxx xxxx	uuuu uuuu
1Fh	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/DONE	—	ADON	0000 00-0	0000 00-0
9Fh	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	00-- 0000
85h	TRISA	—	—	PORTA Data Direction Register						--11 1111	--11 1111
05h	PORTA	—	—	PORTA Data Latch when written: PORTA pins when read						--0x 0000	--0u 0000
89h(1)	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	0000 -111
09h(1)	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	---- -uuu

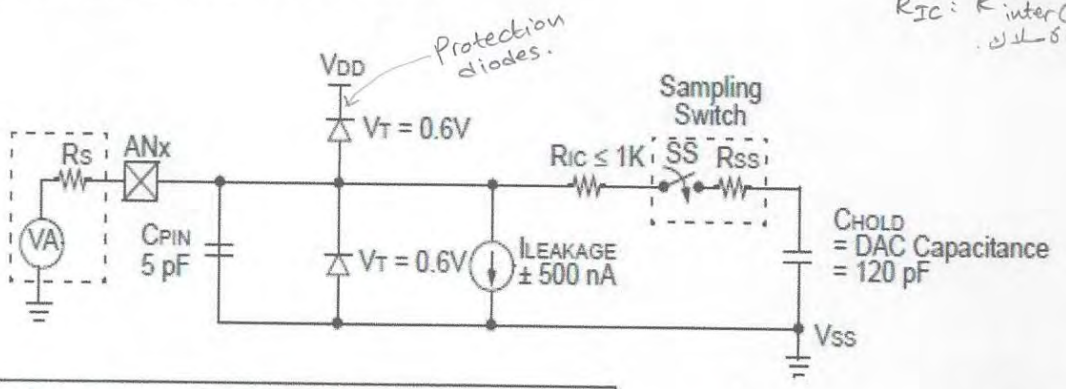
The PIC 16F87xA ADC Module

- **Steps for using the A/D module**
  1. Configure the A/D module
     a. Select analog pins/voltage reference and digital I/O (ADCON1)
     b. Select the A/D channel (ADCON0)
     c. Select the conversion clock (ADCON0)
     d. Turn the A/D module on (ADCON0)
  2. Configure interrupts (if desired) → (3 control levels: ADIE, PEIE & GIE.)
     the flag must be cleared by software
     1. Clear ADIF (PIR1<6>) and set ADIE (PIE1<6>)
     2. Set PEIE (INTCON<6>) then set GIE (INTCON<7>)
  3. Wait the required acquisition time → by hardware or software delays.
  4. Start conversion by setting the GO/DONE' bit
  5. Wait for conversion complete ← 1) by polling the interrupt flag ADIF or DONE.
     2) delay for Tconversion. then read the result
     3) wait for interrupt while doing sth. else.
  6. Read the A/D result register pair ADRESH:ADRESL

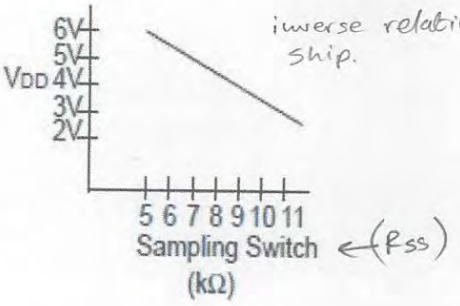
The PIC 16F87xA ADC Module

- **The analog input model**

R_{SS}: $R_{Sampling\ Switch}$.
R_{IC}: $R_{interConnect}$. مقاومة الوصلات

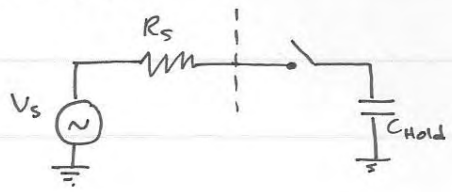


Legend:		
	CPIN	= input capacitance
	VT	= threshold voltage
	ILEAKAGE	= leakage current at the pin due to various junctions
	RIC	= interconnect resistance
	SS	= sampling switch
	CHOLD	= sample/hold capacitance (from DAC)



iwerse relation-ship.

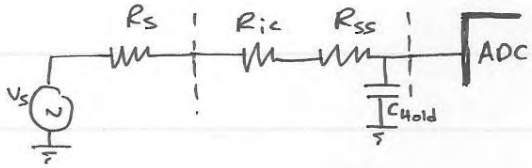
Previous CKT:



$T_{acq.} = 7.6\, R_S\, C_{Hold}$

(this is under the assumption that we want sampling error equal to Qe & n=10).

Actual analog CKT:



$T'_{acq.} = 7.6\,(R_S + R_{SS} + R_{IC})\,C_{Hold}$

$R_{IC} < 1K\Omega$ from data sheet.

R_{SS} is function of V_{DD} (the supply Voltage) (check slide 30)

also, from data sheet: you have to add other terms to accumulate for other things.

$T_{acq} = T_{AMP} + T_{Hold} + T_{temp.}$

$T'_{acq} = 7.6\, R_{eq}\, C_{Hold}$ $(R_S + R_{SS} + R_{IC})$

T_{AMP}: amplifier working as a buffer with very high impedence (to acquire the max. possible portion of the input signal).

$T_{temp.}$: temperature coeficient.

The PIC 16F87xA ADC Module

- **Calculating conversion speed (Qerror is ½ LSB)**

A/D Total Time = Acquisition Time + A/D Conversion time

$= T_{ACQ} + 12 * T_{AD}$

TACQ = Amplifier settling time

+ Hold capacitor charging time

+ Temperature coefficient

$TACQ = T_{AMP} + T_{HOLD} + T_{COFF}$

$THOLD = -(R_{IC}+R_{SS}+R_S) * C_{HOLD} * \ln(1/2^{(n+1)})$

$= -(R_{IC}+R_{SS}+R_S) * 120\ pF * \ln(1/2048)$

$= 7.6*R*C\ us$

Conversion Time $= 2\ \mu s + 7.6RC + (Temperature - 25°C)(0.05\ \mu s/°C) + 12\ T_{AD}$

The PIC 16F87xA ADC Module

- **Calculating conversion speed example**

$R_{SS} = 7k\Omega\ (V_{DD} = 5V)$, $R_{IC} = 1k\Omega$, $R_S = 0$,

Temp = 35 °C, $T_{AD} = 1.6\ \mu s$

$t_{ac} = 2\ \mu s$

$+ 7.6(7k\Omega + 1k\Omega + 0)(120pF)$

$+ (35 - 25)(0.05\ \mu s/°C)$

$= 2 + 7.3 + 0.5 = 9.8\ \mu s$

T_s

Total time $= t_{ac} + 12T_{AD} = 9.8 + 19.2\ \mu s = 29\ \mu s$

max. بما إنه اخترنا أقل error لأن ضنية

Maximum sampling rate ~= 34.5 KHz

∴ we can't sample a signal with BW > $\frac{34.5k}{2}$ = 17.25KHz

The PIC 16F87xA ADC Module

- **Repeated Conversions**
  - When a conversion is complete, the converter waits a period of 2*TAD before it is available to start a new conversion → to be accurate, this time must be added to Tconv. → Tconv = 14 TAD.
  - This time has to be added to the conversion time !
- **Trading off conversion speed and resolution** → for ex & more details. Check data sheet.
  - If resolution is not an issue, then we can start the conversion with correct clock then we switch it to higher clock
  - Consider only bits produced before switching the clock

The PIC 16F87xA ADC Module

- **Example: use the ADC in PIC 16F877A to obtain one sample of an analog signal connected RA0. Assume the ADC clock to be Fosc/8 and reference voltage to be internal. The PIC is operating with Fosc = 4 MHz, VDD = 5 v, and temperature 25 C. The result should be right justified.**

Setup:

1) set RA0 as analog input

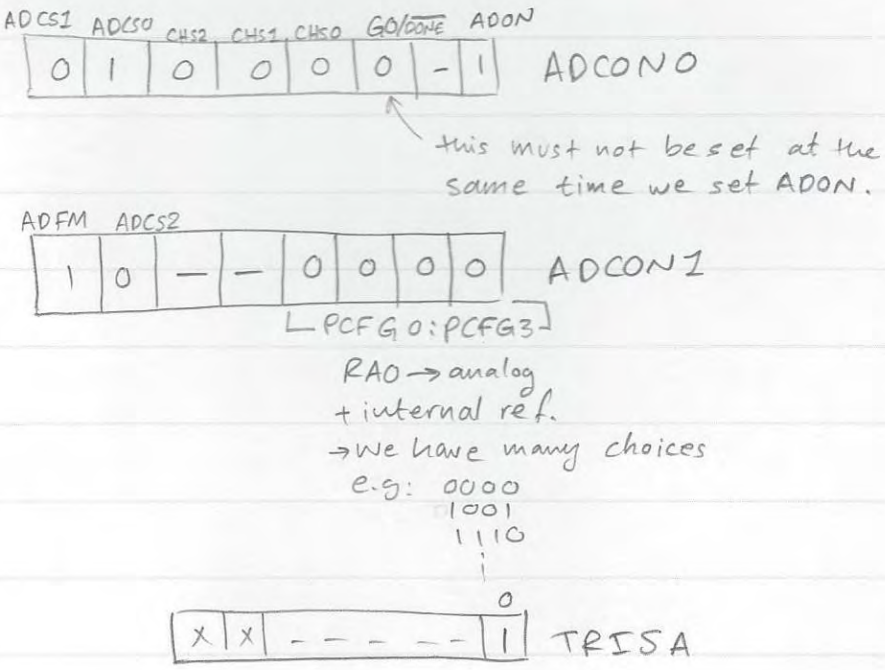
2) select the clock

3) generate appropriate delays (Tacq = 2+ 7.6 * (1K + 7K) * 120 pF = 9.3 us ~= 10 us)

Example

- RA0 (An0)
- $F_{AD} = Fosc/8$
- internal ref.
- $Fosc = 4\,MHz$
- $V_{DD} = 5V$
- temp. = 25°
- Result is right justified

- use polling (since we will convert only once).



Example

```
                #include  p16F877A.inc     ; include the definition file for 16F77A
                org       0x0000          ; reset vector
                goto      START
                org       0x0004          ; define the ISR
ISR             goto      ISR
                org       0x0006           ; Program starts here
START           bsf       STATUS, RP0     ; select bank 1
                movlw     B'00000001'
                movwf     TRISA           ; set RA0 as input
                movlw     B'10001110'     ; select RA0 as analog input, result right
                                          ; justified, and internal reference voltage
                movwf     ADCON1
                bcf       STATUS, RP0     ; select bank 0
                movlw     B'01000001'     ; turn on ADC, clock Fosc/8, select
                                          ; channel 0
                movwf     ADCON0
```

or 0000 or 1001 ...

Example

```
                ; start the conversion
                call    delay10us       ; acquisition time delay
                bsf     ADCON0, GO      ; start conversion
                btfsc   ADCON0, GO_DONE ; wait for conversion to complete
                goto    $-1
DONE            goto    DONE

delay10us       movlw   D'2'
                movwf   0x20            ; counter for delay loop
more            nop
                decfsz  0x20,1
                goto    more
                return

                end
```

Polling

ارجع للـ instr. الي قبل الـ instr السابقة

we can use BTFSS PIR1, ADIF

Summary

- Most signals produced by transducers are analog in nature, while all processing done by a microcontroller is digital.
- Analog signals can be converted to digital form using an analog-to-digital converter (ADC).
- The 16F873A has a 10-bit configurable ADC module
- Data values, once acquired, are likely to need further processing, including offsetting, scaling and code conversion.

The Human and Physical Interface

Chapter 8
Sections 1 - 9

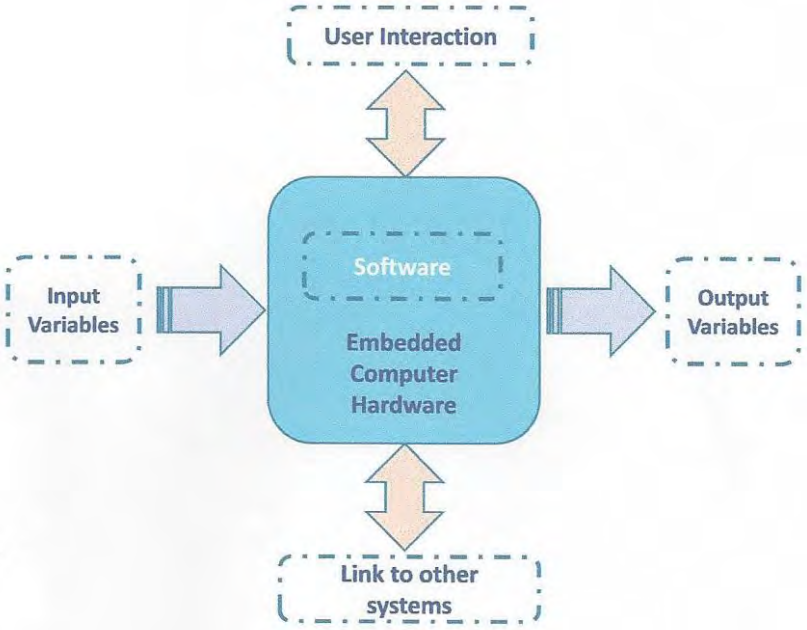
Dr. Iyad Jafar

Outline

- Introduction
- From Switches to Keypads
- LED Displays
- Simple Sensors
- Actuators
- Summary

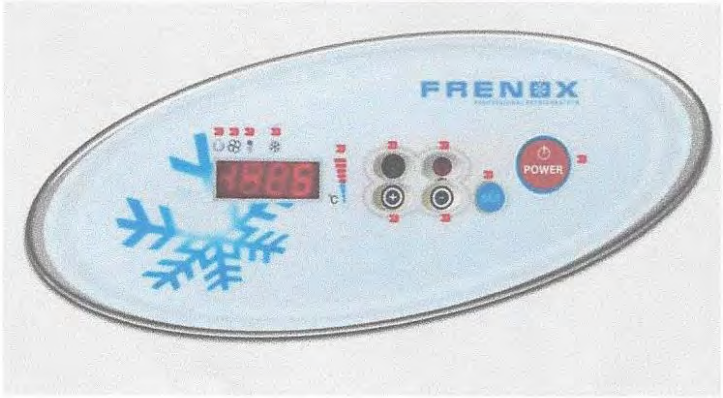
Introduction

- Humans need to interface with embedded systems ; input data and see response
- Input devices: switches, pushbuttons, keypads, sensors
- Output devices: LEDs, seven-segment displays, liquid crystal displays, motors, actuators

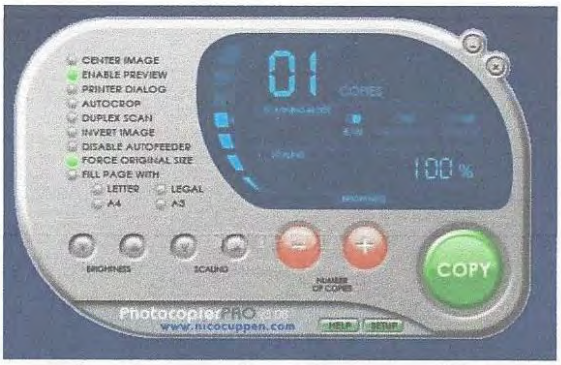


Introduction

- **Examples**



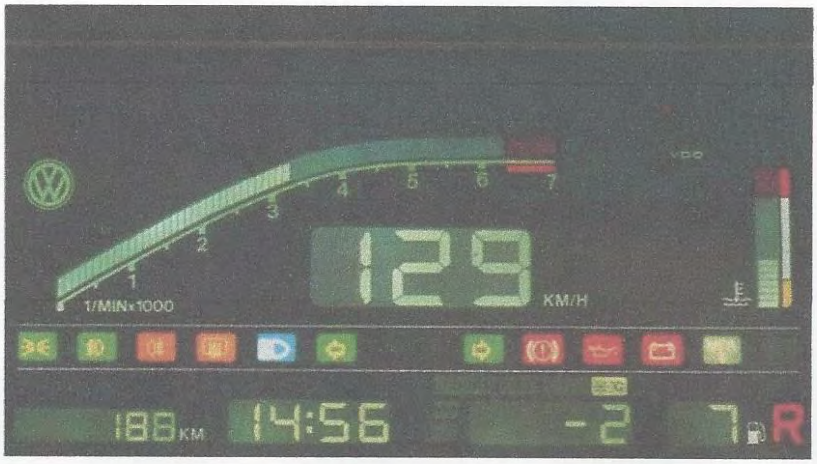
Fridge Control Panel



Photocopier Control Panel

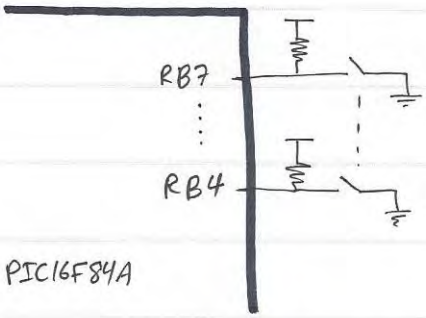
Introduction

- **Examples**



Car Dashboard

- If you want to input digital information to the Microcontroller, you can use switches. e.g: input numbers 0-9 (we need 4 pins).



→ Problems:

* We need too many pins to implement bigger numbers.
* Using switches is not user friendly. Also, not all users are familiar with binary numbers.

solution: Use keypads.

Moving From Switches to Keypads

- Switches are good for conveying information of digital nature
- They can be used in multiples; each connected to one port pin
- In complex systems, it might not be feasible to keep adding switches ?!
- Use keypads !
  - Can be used to convey alphanumeric values
  - A group of switches arranged in matrix form

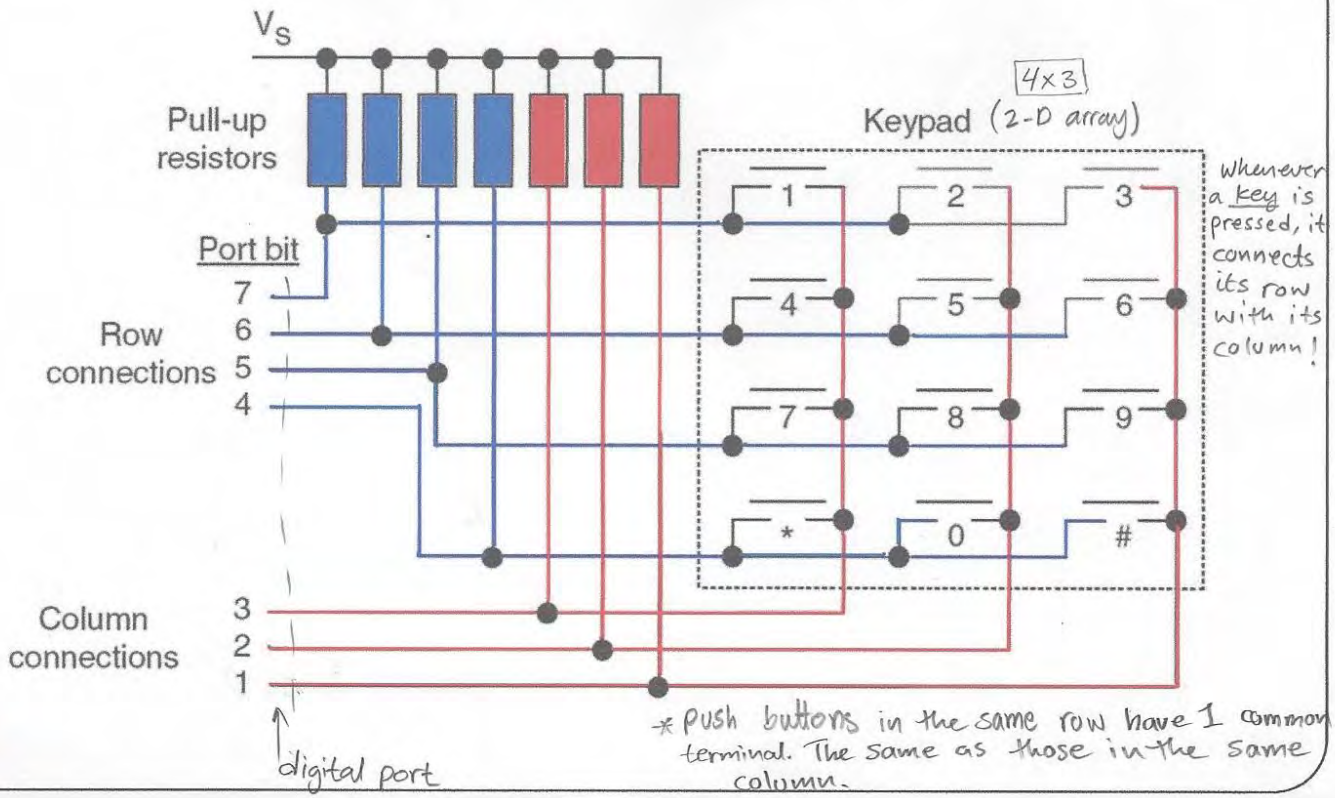
4x3 Keypad

4x4 Keypad.

Moving From Switches to Keypads

Interfacing keypad with Microcontroller...

Internal Structure of Keypad



* Assume all pins are configured as 'inputs':

→ If no button is pressed, you'll read [1111111]; since pins will be connected to V_s.

→ When you press a button (e.g: [1]), you'll also read [1111111] !

So, pressing the button didn't change the input value !!

* Solution: interfacing the keypad with the Microcontroller is actually done in 2 steps:

1. Configure 'row connections' (4:7) as inputs & 'column connections' (1:3) as outputs & output logic zero at column pins (1:3).

→ if no button is pressed you'll read [1111]. ← Row pins

→ if you press a button (e.g: 1), you'll read [0111]

this value can be used to know the row number

This zero indicates that the pressed button is in the first row. (either 1, 2 or 3)

* Save the resulting value, then go to step 2.

→ cont.

2. configure 'column connections' as inputs & 'Row connections' as outputs. (بقلب الدنيا (9: Logic

& output logic zero at row pins (4:7).

→ If no button is pressed, you'll read [111]; since pins are connected to Vs.

→ if a button is pressed (e.g: 1), you'll read [011] ← since pressing the button will connect the line to logic 0 (~GND).

this value can be used to know the column's number

Now, the button is fully identified by both its row & column numbers.

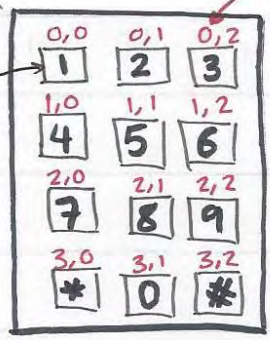
so, when Pressing button #1 we'll have: →

Step 1 (b_0 b_1 b_2 b_3)	Step 2 (b_0 b_1 b_2)	in this case the location of the (0) is the same as COL. or ROW #.
0 1 1 1	0 1 1	
7:4	3:1	
= ROW #0	COL #0	

* To be able to detect pressing of a button you can connect (row pins) to PORTB pins ($_{RB}(4:7)$) & activate PORT B on-change interrupt.

* The last step is to determine which label is assigned to the resulting Row # & column #.

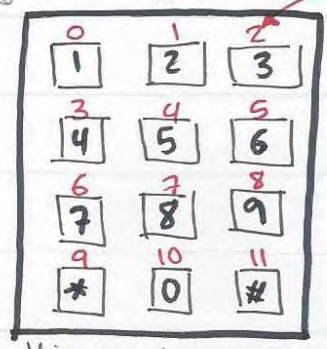
we need two indecies to represent a Label.



this method needs 2-dimensional array to connect each ROW & COL. #s to their assigned label.
* but 2-D arrays doesn't exist in pic Microcontroller, so you'll have to convert it to a 1-D array.

since memory is a linear arrangement of storage elements.

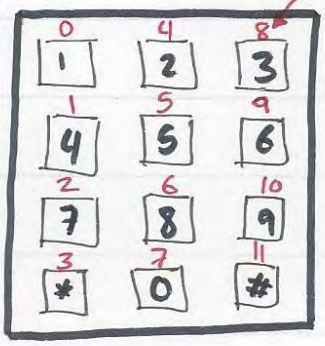
we need only 1 index to represent a Label. (linear index)



Button ID =
ROW_NO. * 3 + COL_NO.
total # of columns.

this can be represented as a look-up table.

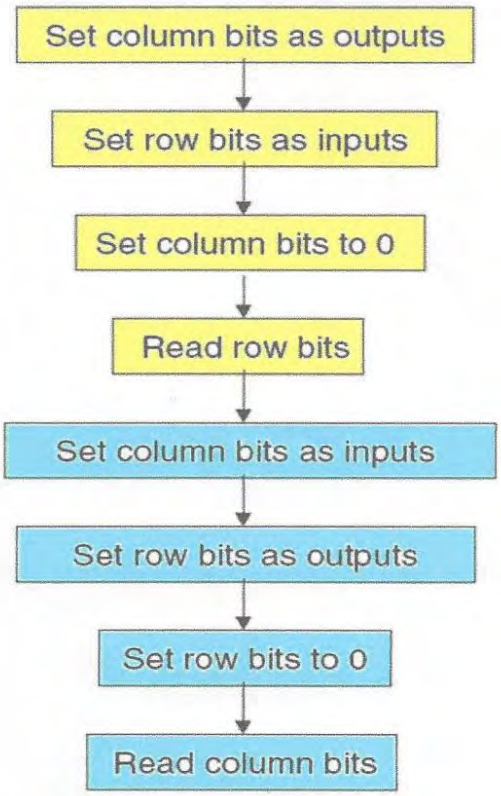
OR



Button ID =
COL_NO. * 4 + ROW_NO.
total # of rows.

Moving From Switches to Keypads

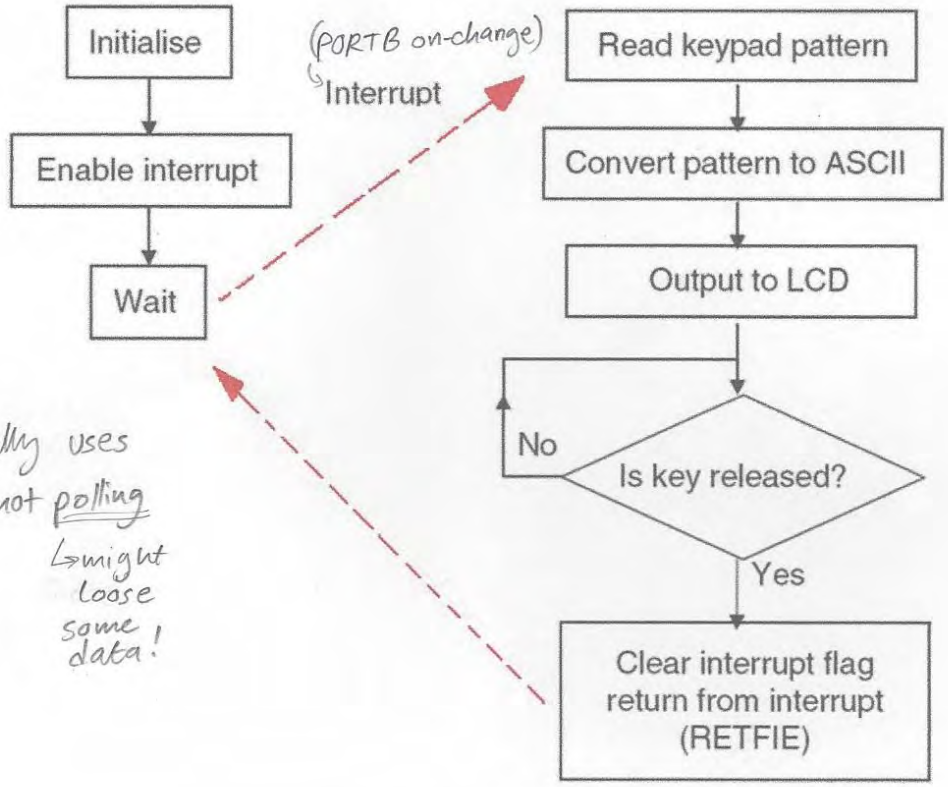
How to Determine the Pressed Key



Key	Value Read
1	0111 011X ← RB0 (don't care).
2	0111 101X
3	0111 110X
4	1011 011X
5	1011 101X
6	1011 110X
7	1101 011X
8	1101 101X
9	1101 110X
*	1110 011X
0	1110 101X
#	1110 110X

Moving From Switches to Keypads

Using Keypad in a Microcontroller



Keypads usually uses interrupts not polling
↳ might loose some data!

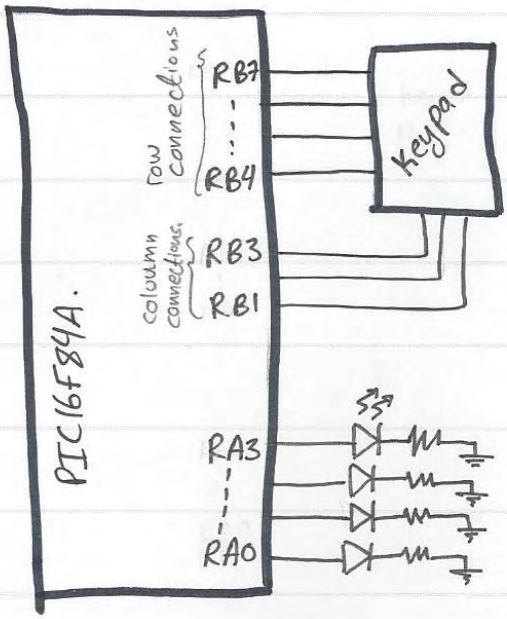
Moving From Switches to Keypads

Example

A program to read an input from a 4x3 keypad and displays the equivalent decimal number on 4 LEDs. If the pressed key is not a number, then the LEDs should be all on.

- The keypad will be connected to MC as follows
  - Rows 0 to 3 connected to RB7 to RB4, respectively.
  - Columns 0 to 2 connected to RB3 to RB1, respectively.
- Use PORTB on-change interrupt
- Connect the LEDs to RA0-RA3
- Based on the pressed key, convert the row and column values to binary using a lookup table

* hardware implementation:



Keypad Interfacing Example

```
                #include    P16F84A.INC
ROW_INDEX       EQU         0X20
COL_INDEX       EQU         0X21
                ORG         0X0000
                GOTO        START
                ORG         0X0004
                GOTO        ISR
START           BSF         STATUS, RP0
                MOVLW       B'11110000'
                MOVWF       TRISB       ; SET RB1-RB3 AS OUTPUT AND
                                        ; RB4-RB7 AS INPUT
                MOVLW       B'00000000'
                MOVWF       TRISA       ; SET RA0-RA3 AS OUTPUT
                BCF         STATUS, RP0
                CLRF        PORTB       ; INITIALIZE PORTB TO ZERO
                MOVF        PORTB,W     ; CLEAR RBIF FLAG
                BCF         INTCON, RBIF
                BSF         INTCON, RBIE
                BSF         INTCON, GIE ; ENABLE PORT b CHANGE INTERRUPT
LOOP            GOTO        LOOP        ; WAIT FOR PRESSED KEY
```

* input pins aren't affected

Keypad Interfacing Example

```
ISR             MOVF    PORTB, W    ; READ ROW NUMBER
                MOVWF   ROW_INDEX
                BSF     STATUS, RP0 ; READ COLUMN NUMBER
                MOVLW   B'00001110'
                MOVWF   TRISB
                BCF     STATUS, RP0
                CLRF    PORTB
                MOVF    PORTB, W
                MOVWF   COL_INDEX
                CALL    CONVERT     ; CONVER THE ROW AND COLUMN
RST_PB_DIRC     BSF     STATUS, RP0 ; PUT THE PORT BACK TO INITIAL SETTINGS
                MOVLW   B'11110000'
                MOVWF   TRISB       ; SET RB1-RB3 AS OUTPUT AND
                MOVLW   B'00000000' ; RB4-RB7 AS INPUT
                MOVWF   TRISA       ; SET RA0-RA3 AS OUTPUT
                BCF     STATUS, RP0
                CLRF    PORTB
                MOVF    PORTB, W    ; REQUIRED TO CLEAR RBIF FLAG
                BCF     INTCON, RBIF
                RETFIE
```

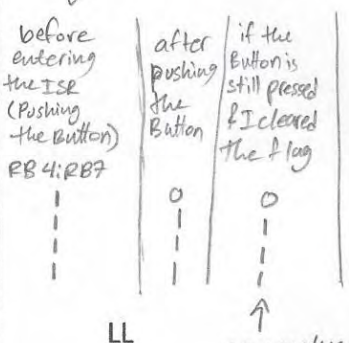

Keypad Interfacing Example

```
CONVERT         BTFSS   COL_INDEX,3 ; IF 1ST COLUMN, COL_INDEX=0
                MOVLW   0
                BTFSS   COL_INDEX,2 ; IF 2ND COLUMN, COL_INDEX=1
                MOVLW   1
                BTFSS   COL_INDEX,1 ; IF 3RD COLUMN, COL_INDEX=2
                MOVLW   2
                MOVWF   COL_INDEX   ; STORE THE COLUMN INDEX

FIND_ROW        BTFSS   ROW_INDEX,7 ; IF 1ST ROW, ROW_INDEX=0
                MOVLW   0
                BTFSS   ROW_INDEX,6 ; IF 2ND ROW, ROW_INDEX=1
                MOVLW   1
                BTFSS   ROW_INDEX,5 ; IF 3RD ROW, ROW_INDEX=2
                MOVLW   2
                BTFSS   ROW_INDEX,4 ; IF 4TH ROW, ROW_INDEX=3
                MOVLW   3
                MOVWF   ROW_INDEX

                ; CONTINUED ON NEXT PAGE
```

Checks the location of the 0 to determine COL_NO & ROW_NO.

Keypad Interfacing Example

```
COMPUTE_VALUE   MOVF    ROW_INDEX, W  ; KEY # = ROW_INDEX*3 + COL_INDEX
                ADDWF   ROW_INDEX, W
                ADDWF   ROW_INDEX, W
                ADDWF   COL_INDEX, W  ; THE VALUE IS IN W
; CHECK IF VALUE IS GREATER THAN 11. THIS HAPPENS WHEN THE BUTTON IS RELEASED
; LATER, AN INTERRUPT OCCURS WITH ALL SWITCHES OPEN, SO THE  MAPPED  VALUE IS ;
; ABOVE 11
                MOVWF   0X30          ; COPY THE BUTTON NUMBER
                MOVLW   0X0C          ≡ (11)10
                SUBWF   0X30,W
                BTFSC   STATUS, C     ; WILL NOT WORK CORRECTLY, OVERFLOW OCCURS
                GOTO    LL
                MOVF    0X30, W
                CALL    TABLE
                MOVWF   PORTA         ; DISPLAY THE NUMBER ON PORTA
LL              RETURN
```



new value for the latch that will be considered as the previous value where any change on it will issue an interrupt!

So, in this case when the button is released, the value will become [1111] & a false interrupt will occur! & it'll have an index (10) > 11. So, an error will happen. Thus we add a code to check for this case

Keypad Interfacing Example

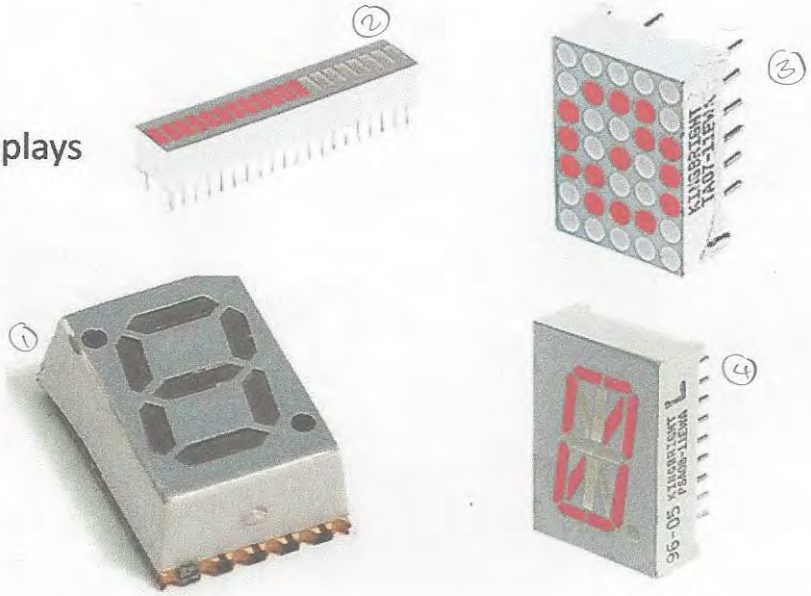
```
TABLE   ADDWF   PCL, F
        RETLW   0X01
        RETLW   0X02
        RETLW   0X03
        RETLW   0X04
        RETLW   0X05
        RETLW   0X06
        RETLW   0X07
        RETLW   0X08
        RETLW   0X09
        RETLW   0X0F    ; ERROR CODE   (*)
        RETLW   0X00
        RETLW   0X0F    ; ERROR CODE   (#)

        END
```

not numeric (all ones)

(Output devices)

LED Displays

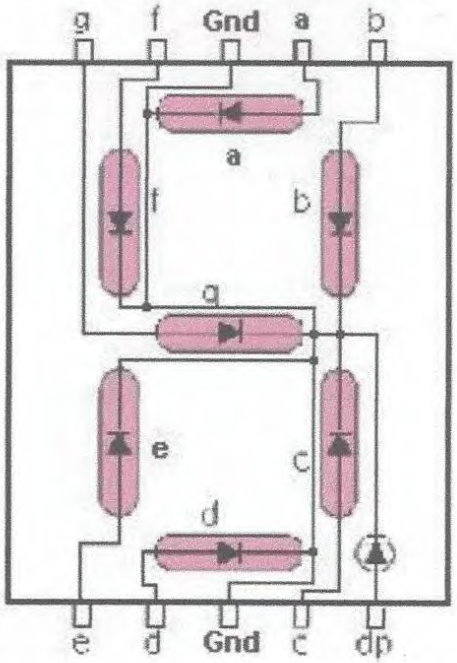
- Light emitting diodes are simple and effective in conveying information
- However, in complex systems it becomes hard to deal with individual LEDs
- Alternatives
  - Seven segment displays
  - Bargraph
  - Dot matrix
  - Star-burst



Seven Segment Display

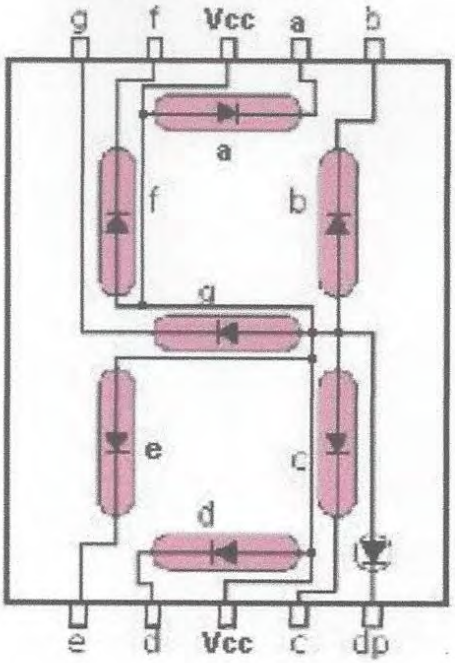
in this case, the Micro-controller will source the current

Common Cathode

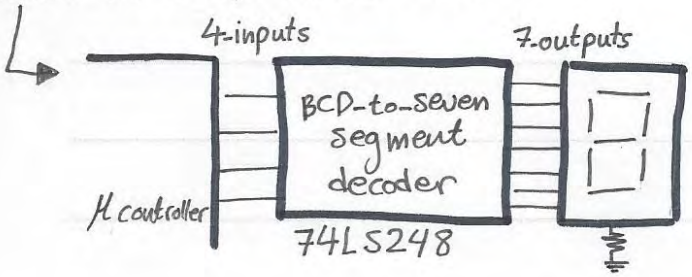


the Micro controller will sink the current

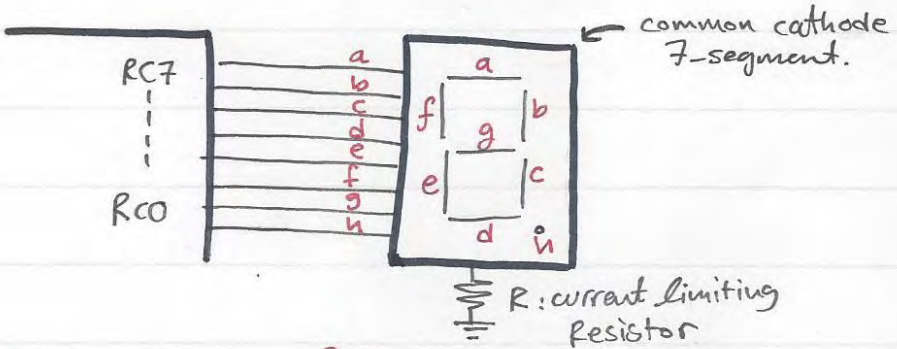
Common Anode



* Check the example on the website:



* How to interface 7-segment displays with PIC16F877?

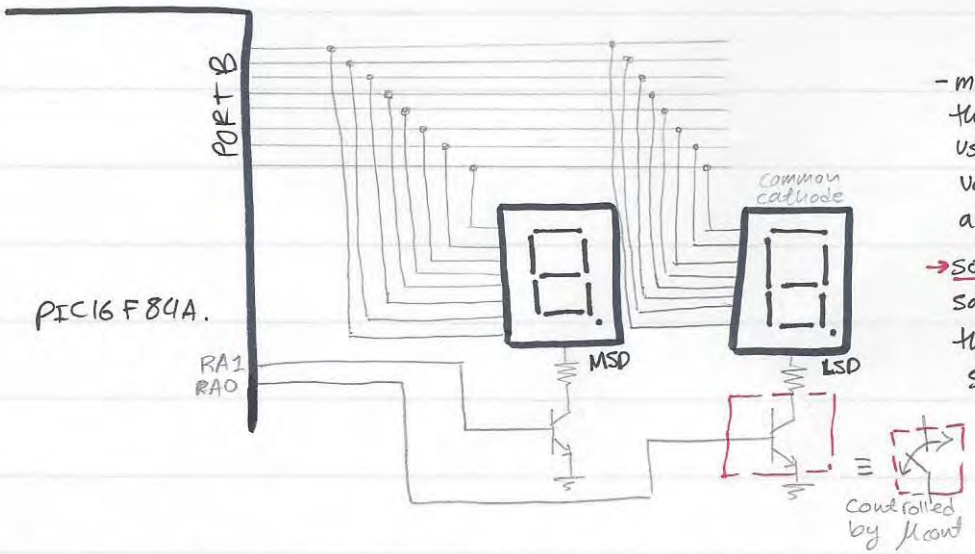


→ to display #3:  → (1111 0010) (don't care) →
```
MOVLW 0xF2
MOVWF PORTC
```

* what if we want to implement a 4-digits Clock? we'll need 7x4 = 28 pins!
↳ also, it will require excessive power supply requirements. → cont.

→ Solution: Apply "Digits multiplexing"

e.g: 2-digits counter:



- make the two displays share the same pins. So, whenever I use these pins to display some value on the 1st display, it'll also appear on the other one!

→ solution: let them share the same connections, but prevent them from using them at the same time. → Use switches. (electronic switch = transistors)

* when you want to display a value on one of the 7-segments close its switch.

* The digits are activated continuously in turn. (each one takes (5~20)ms in general).
If this was done at the right speed, human eye will be tricked into thinking that all digits are being continuously lit. (time sharing). → (5~20-15 خلال ثانية بحيث تنشيط الـ digits جوالي)

So, if we want to implement 4-digits clock, we need:

4x7 = 28 pins (without multiplexing)

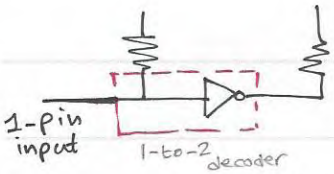
1x7 + (4) = 11 pins (with =)

↳ digits control bits.

cont. →

* Can I cut the cost more?

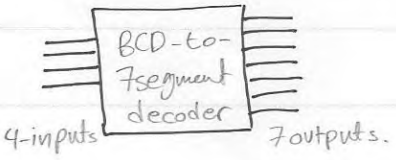
① Use decoders to control digits.
e.g: for 2 digits, use (1-to-2) decoder.



→ for the 4-digits case we'll need $1 \times 7 + \log_2 4 = 7 + 2 = 9$ pins

∴ to control (n) digits we'll need $(\log_2 n)$ pins instead of (n) pins.

② Use BCD-to-7segment decoder

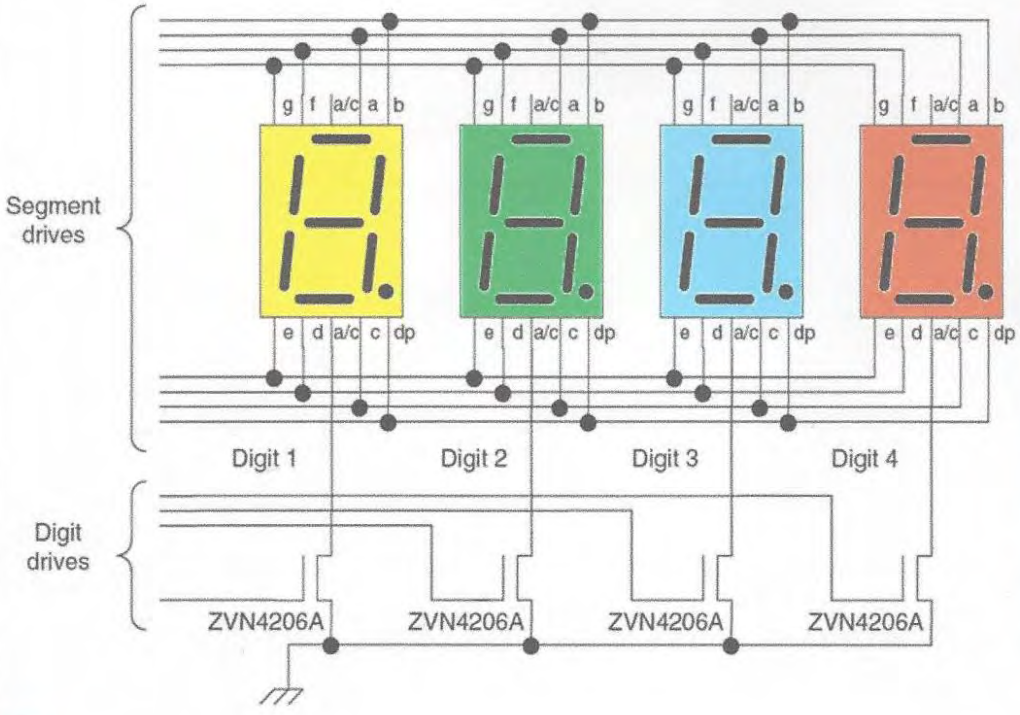


we'll need 4 pins insted of 7 to connect 7-segments.

∴ for the 4-digits case we'll need $1 \times 4 + 2 = 6$ pins only. ← min. cost (w.r.t # of pins).

Seven Segment Display

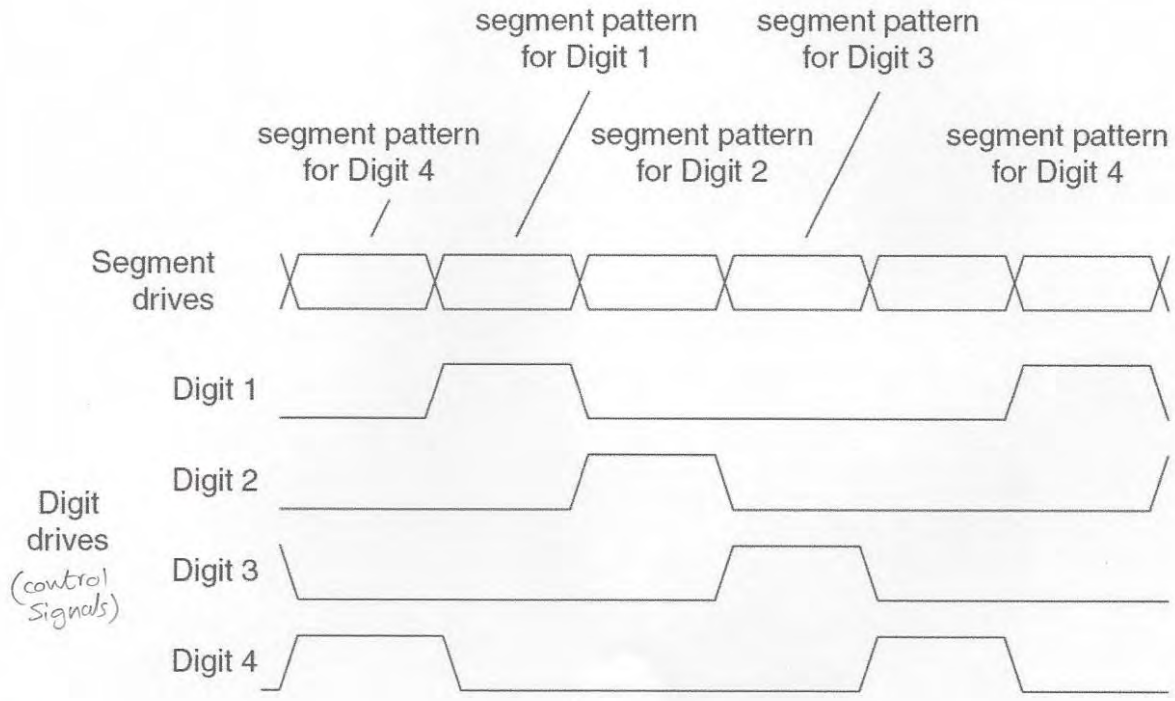
Multiplexing of seven segment digits



Connection	Port Bit
Segment a	RB7
Segment b	RB6
Segment c	RB5
Segment d	RB4
Segment e	RB3
Segment f	RB2
Segment g	RB1
Segment dp	RB0
Digit 1 drive	RA0
Digit 2 drive	RA1
Digit 3 drive	RA2
Digit 4 drive	RA3

Seven Segment Display

Multiplexing of seven segment digits



Seven Segment Display

Example

A program to count continuously the numbers 0 through 99 and display them on two seven segment displays. The count should be incremented every 1 sec. Oscillator frequency is 3 MHz.

the design is the same as the 2-digits example discussed before.

- connect the seven segment inputs **a** through **g** to RB0 through RB6 respectively
- connect the gates of the controlling transistors to RA0 (LSD) and RA1 (MSD)
- the main program will be responsible for display and multiplexing every 5 ms

* how to count 1 second?
① keep multiplexing continuously (with a delay = a base time unit (here = 5ms)) while counting (by software) until reaching base time unit * count = 1 sec (for the 2 digits) (in this example count must reach 100)
or (TMR1/TMR2)
② use timer0 (Hardware delay) to count for 1 sec. & interrupt CPU when it overflows.

Seven Segment Display Example

```
                #INCLUDE    PICF84A.INC
LOW_DIGIT       EQU         0X20
HIGH_DIGIT      EQU         0X21
COUNT           EQU         0X22
                ORG         0X0000
                GOTO        START
                ORG         0X0004
ISR             GOTO        ISR
START           BSF         STATUS, RP0
                MOVLW       B'00000000' ; set port B as output
                MOVWF       TRISB
                MOVWF       TRISA       ; SET RA0-RA1 AS OUTPUT
                BCF         STATUS, RP0
                CLRF        PORTB
                CLRF        PORTA
                CLRF        LOW_DIGIT   ; CLEAR THE COUNT VALUE
                CLRF        HIGH_DIGIT
                CLRF        COUNT
```


Seven Segment Display Example

```
DISPLAY         BSF         PORTA , 0
                BCF         PORTA,  1
                MOVF        LOW_DIGIT, W    ; DISPLAY LOWER DIGIT
                CALL        TABLE           ; GET THE SEVEN SEGMENT CODE
                MOVWF       PORTB
                CALL        DELAY_5MS       ; KEEP IT ON FOR 5 MS
                BCF         PORTA, 0
                BSF         PORTA, 1
                MOVF        HIGH_DIGIT, W   ; DISPLAY  HIGH DIGIT
                CALL        TABLE           ; GET THE SEVEN SEGMENT CODE\
                MOVWF       PORTB
                CALL        DELAY_5MS       ; KEEP IT ON FOR 5 MS
                ; CHECK IF  1 SEC ELAPSED
                INCF        COUNT,F         ; INCREMENT THE COUNT VALUE IF TRUE
                MOVF        COUNT, W
                SUBLW       D'100'
                BTFSS       STATUS, Z
                GOTO        DISPLAY         ; DISPLAY THE SAME COUNT
```

Seven Segment Display Example

```
; TIME TO INCREMENT THE COUNT
CLRF        COUNT
INCF        LOW_DIGIT, F  ; INCREMENT LOW DIGIT AND CHECK IF > 9
MOVF        LOW_DIGIT, W
SUBLW       0X0A
BTFSS       STATUS, Z
GOTO        DISPLAY
CLRF        LOW_DIGIT

INCF        HIGH_DIGIT, F ; INCREMENT HIGH DIGIT AND CHECK IF > 9
MOVF        HIGH_DIGIT, W
SUBLW       0X0A
BTFSS       STATUS, Z
GOTO        DISPLAY
CLRF        HIGH_DIGIT
GOTO        DISPLAY
```

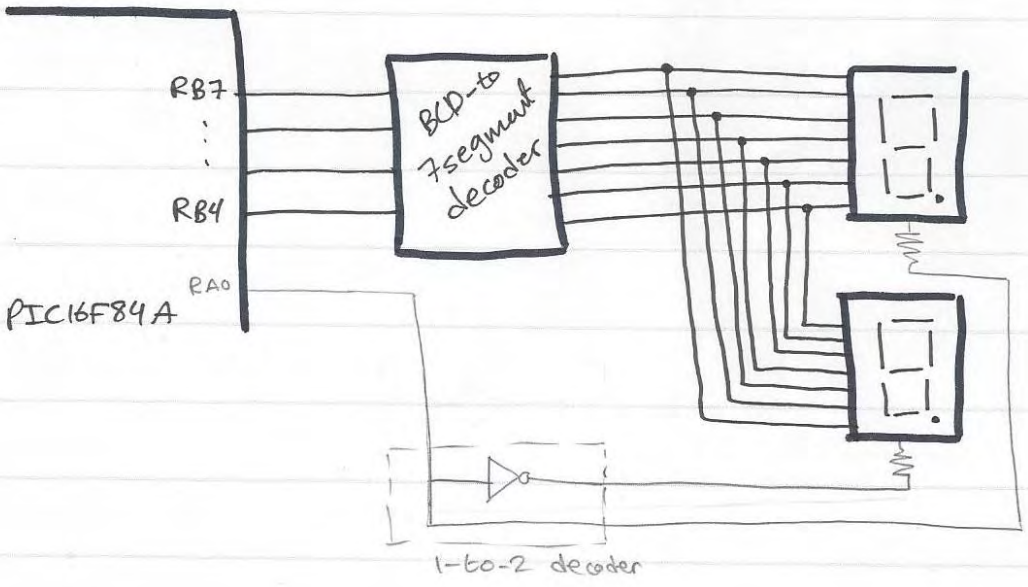

Seven Segment Display Example

```
DELAY_5MS   MOVLW       D'250'
            MOVWF       0X40
REPEAT      NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            DECFSZ      0X40,1
            GOTO        REPEAT
            RETURN
```

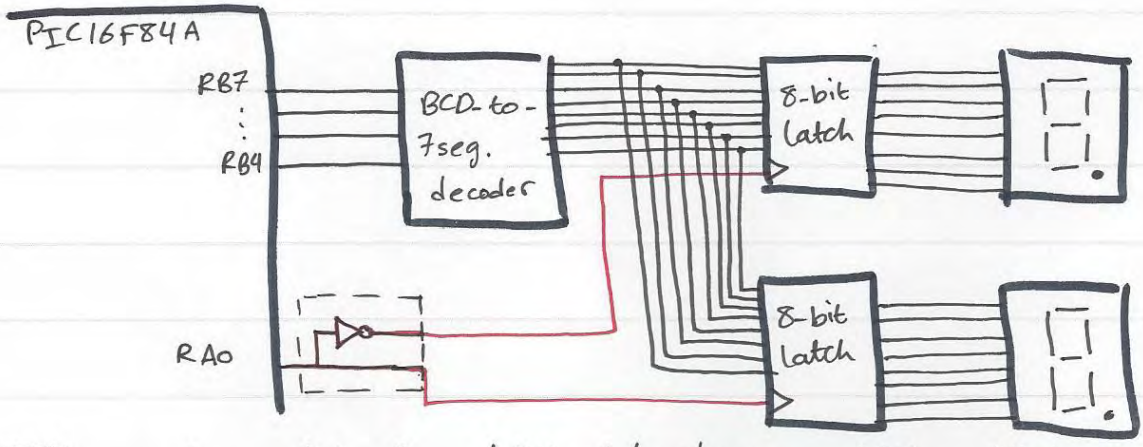
Seven Segment Display Example

```
TABLE       ADDWF  PCL,        1
            RETLW      B'00111111'          ;'0'
            RETLW      B'00000110'          ;'1'
            RETLW      B'01011011'          ;'2'
            RETLW      B'01001111'          ;'3'
            RETLW      B'01100110'          ;'4'
            RETLW      B'01101101'          ;'5'
            RETLW      B'01111101'          ;'6'
            RETLW      B'00000111'          ;'7'
            RETLW      B'01111111'          ;'8'
            RETLW      B'01101111'          ;'9'


            END
```


* If you don't want to use a look-up table, use a BCD-to-7segment decoder



* In order not to do multiplexing by software, you can add additional hardware to do the work



*check the example on the website.

* This needs a lot of additional hardware, which will increase the cost, space & complexity espicially for big # of digits.

Sensors

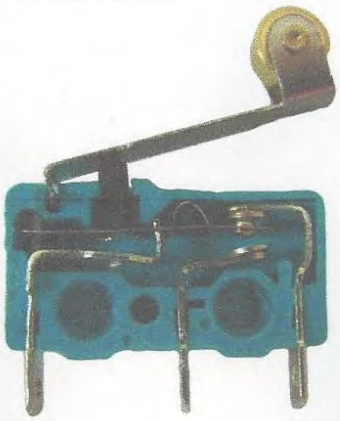
- Embedded systems need to interface with the physical world
- Must be able to detect the state of the physical variables and control them
- Input transducers or sensors are used to convert physical variables into electrical variables
  - Light sensors
  - Temperature sensors
- Output transducers convert electrical variables to physical; actuators

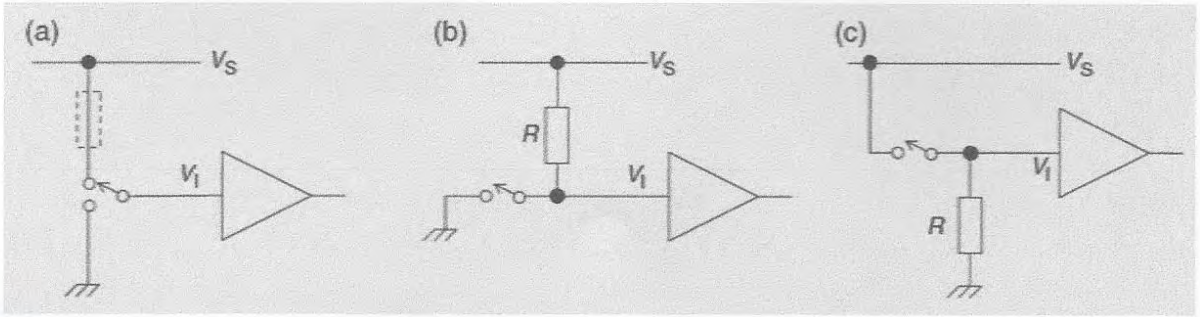
Sensors

→ to read & control environment variables.

The Microswitch

(sensitive switch).





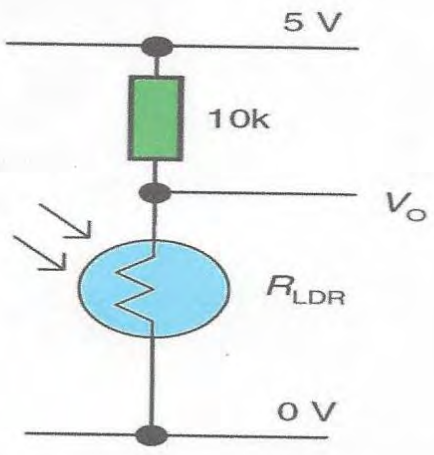
Sensors

Light-dependent Resistors

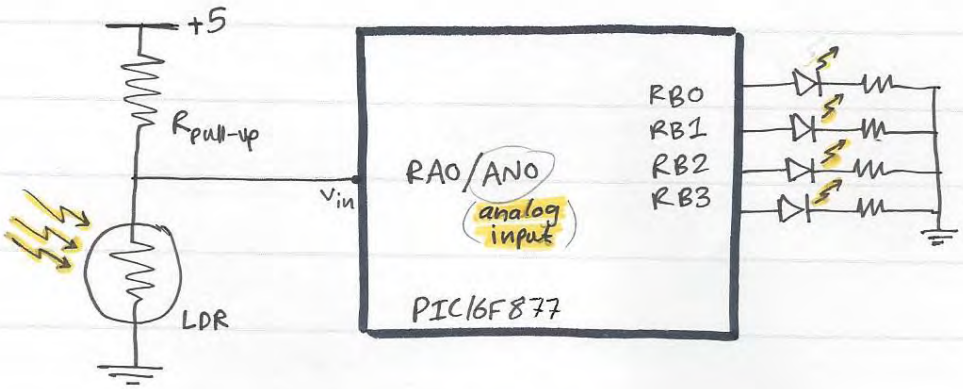
(passive sensor).

- A light-dependent resistor (LDR) is made from a piece of exposed semiconductor material
- When light falls on it, it creates hole–electron pairs in the material, which improves the conductivity.

Illumination (lux)	R_{LDR} (Ohms)	Vo
Dark	2M	5
10	9000	2.36
1000	400	0.19



Example: (Interfacing a light-dependent Resistor (LDR) with PIC16F877)

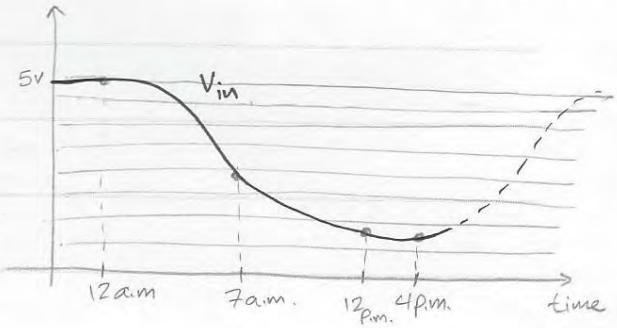


the μcontroller lights some, all or none of the LEDs based on light's intensity read by the LDR.

during complete darkness (no light) → conductivity is minimum → Resistivity maximum (open CKT) → high voltage (max.)

* ↑ light → ↑ conductivity → ↓ R_{LDR} → ↓ V_{in}.

(e.g.:) you can design a system with multiple thresholds, where crossing a threshold lights a specific # of LEDs ←

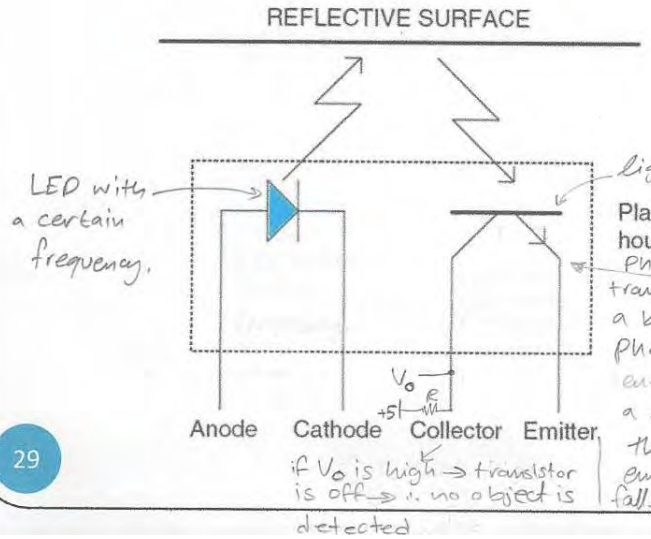


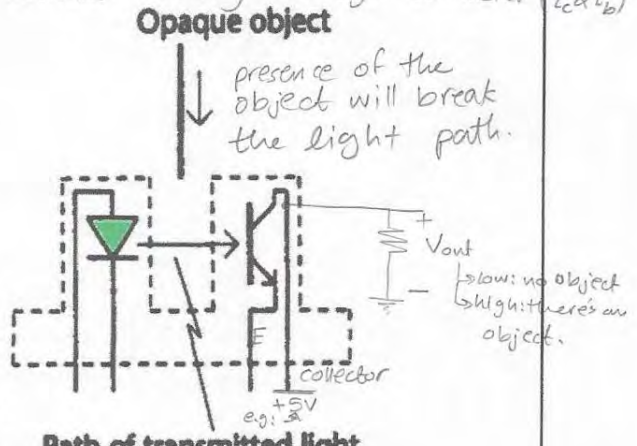
Sensors

Optical Object Sensing

(active sensors; they produce & measure energy) ← not accurate for distance calculations.

- Useful in sensing the presence or closeness of objects
- The presence of object can be detected
  - If it breaks the light beam
  - If it reflects the light beam → ↑ closeness of the object; ↑ light intensity of the reflected beam; ↑ current flowing through collector (i_c ≈ i_b)

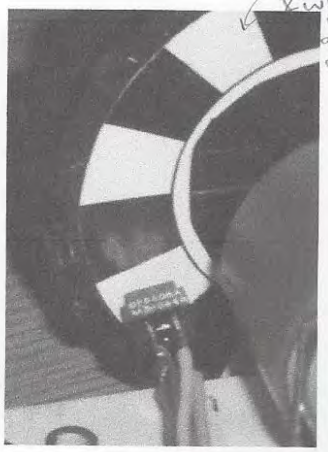




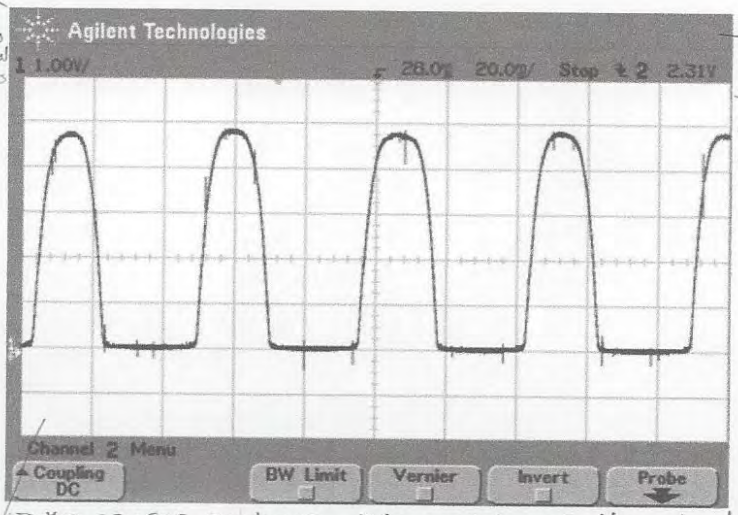
Sensors

Opto-sensor as a Shaft Encoder

- Useful in measuring distance and speed



a wheel with black & white marks at equal distances



if the pulses (peaks) are counted they can be used as the basis of distance measurem

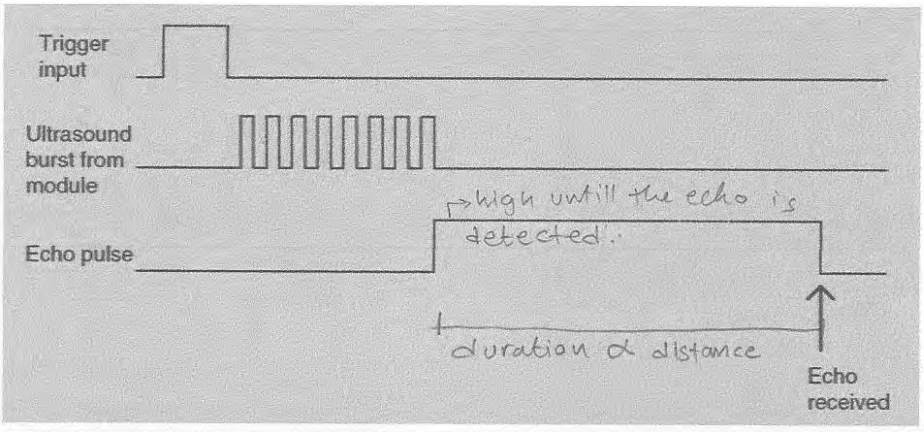
the sensor produces this wave as the wheel is rotating, a peak (high voltage → no light) is produced whenever a black section goes by since the black color absorbs most of the light. & when white section goes by the voltage becomes low (transistor is ON).

Sensors

Ultrasonic Object Sensor

same concept, but uses ultrasonic waves instead of light)

- Based on reflective principle of ultrasonic waves
- An ultrasonic transmitter sends out a burst of ultrasonic pulses and then the receiver detects the echo
- If the time-to-echo is measured, distance can be measured



now, what if we want to <u>control</u> the environment variables

e.g: controling Street lights. based on light intensity.

* the microcontroller will control turning ON and OFF the light bulb, it's not going to power it (since it needs 220V)!

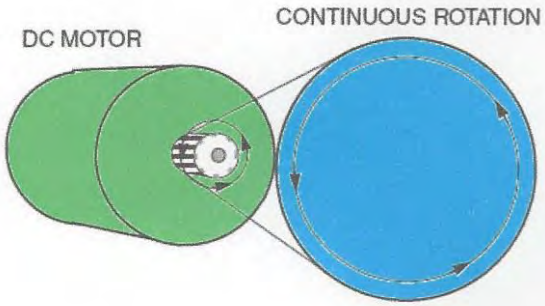
* Thus, we use an electromechanical switch (relay)
  - ↳ when we output logic 1 (~5V) from RB1, current will flow through the coil producing a magnetic field, which will attract the switch towards the coil, then the switch is opened. ∴ The light is turned OFF.
  - ↳ when we output logic low (0V), no current will flow, thus the magnetic field = 0 & the spring will pull the switch to <u>close</u>. Thus, the light will turn ON.

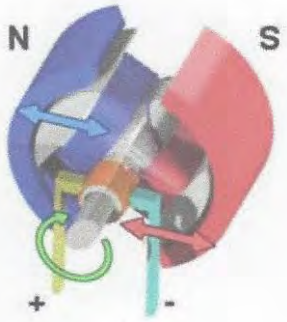
Actuators: motors and servos

- Embedded systems need to cause physical movement
- Linear or rotary motion
- Most actuators are electrical in nature
  - Solenoids (linear motion)
  - Servo motors (rotary motion)
  - DC or stepper motors (rotary motion)

DC Motors

- Range from the extremely powerful to the very small
- Wide speed range
- Controllable speed
- Good efficiency
- Can provide accurate angular positioning with angular shafts
- Only the armature winding needs to be driven

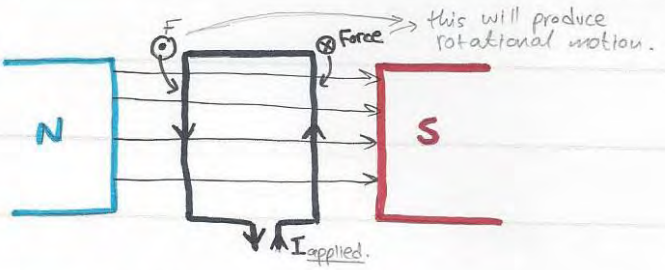




33

* DC Motor:

basic concept of operation:



*direction of rotation depends on the direction of the applied current.

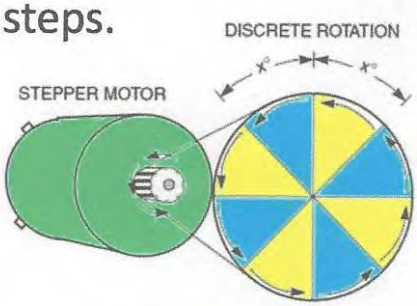
→as long as the DC motor is powered, it continuously rotates.

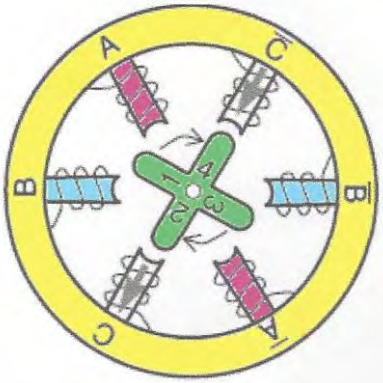
* Can we use it to control how far we go?
Yes, but it's hard (you need a feed back system)
e.g: optical shaft encoder.

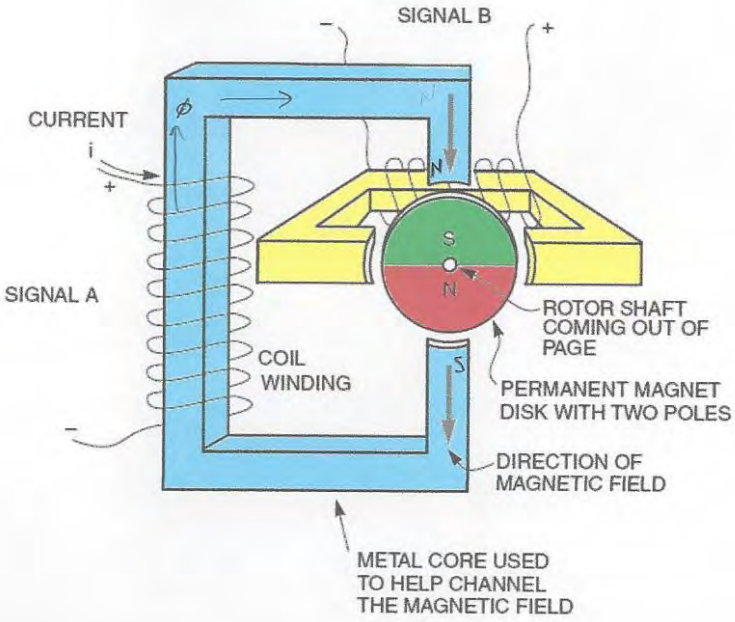
∴ DC motor is very easy to use when it's supposed to rotate continuously But when accurate positioning is needed it requires additional hardware. A good alternative in this case is to use stepper motor. (which can move in steps).

Stepper Motors

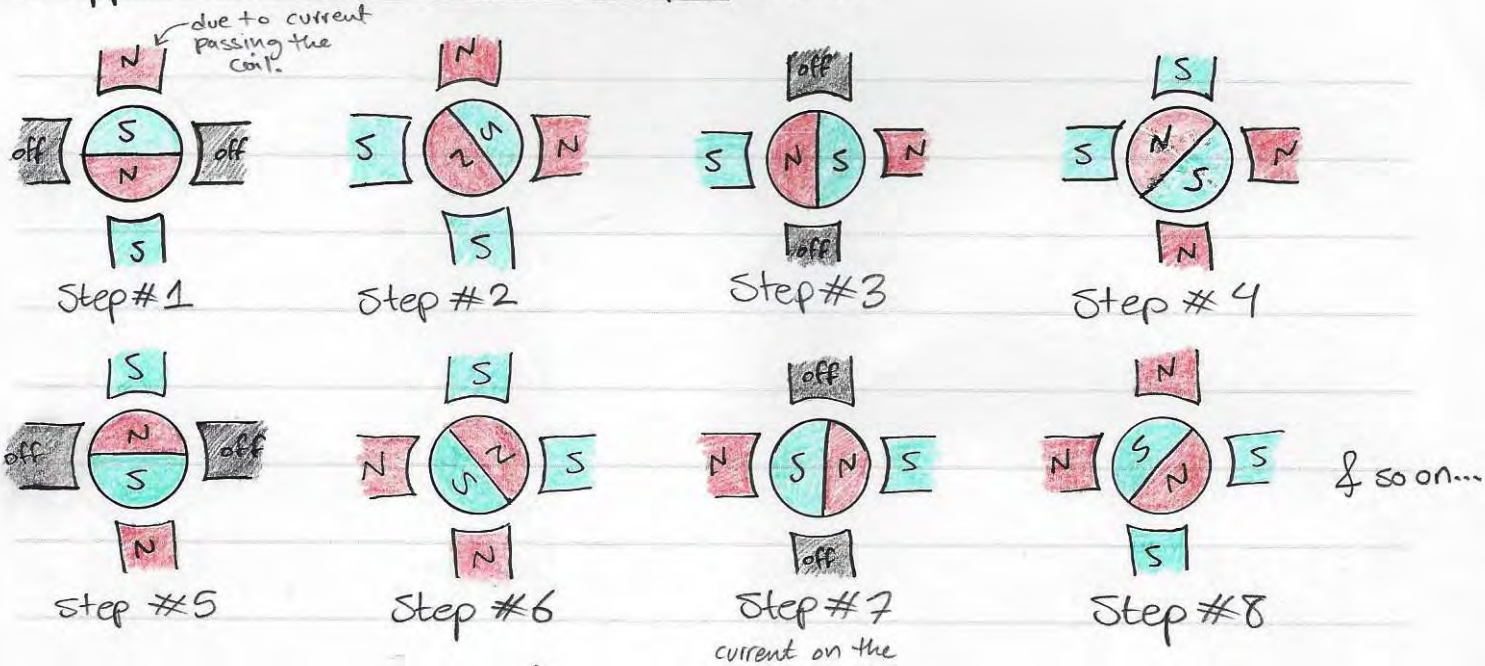
- A **stepper motor** (or **step motor**) is a synchronous electric motor that can divide a full rotation into a large number of steps.







Stepper Motor rotation (steps): (Example)



* if you keep switching ON & OFF the current on the coils with the previous sequence, you can get continuous motion but it won't be smooth as in DC motors. So, usually stepper motors aren't used for fast & continuous motion.
* To increase the # of steps, you can increase # of coils (or) # of magnetic poles (on the shaft).
* Problem? #of pins needed to interface stepper motor with µcontroller depends on # of coils which is usually big. → Solution: use servo motor. (only one pin required.)

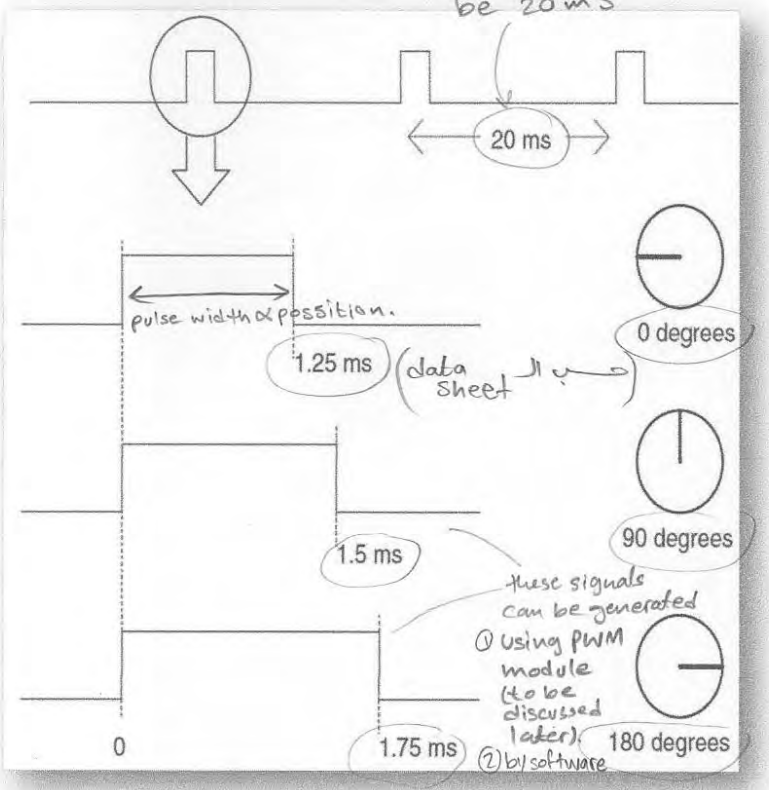
Stepper Motors

- **Features**
  - Simple interface with digital systems
  - Can control speed and position
  - More complex to drive
  - Awkward start-up characteristics
  - Lose torque at high speed
  - Limited top speed
  - Less efficient

Servo Motors

usually comercial servo motors expects the period of the cycle (ON-Off) to be 20 ms

- Allows precise angular motion
- The output is a shaft that can take an angular position over a range of 180°
- The input to the servo is a pulse stream whose width determines the angular position of the shaft



* most servo motors doesn't provide full rotation (360°), usually 180°-210°

Interfacing to Actuators

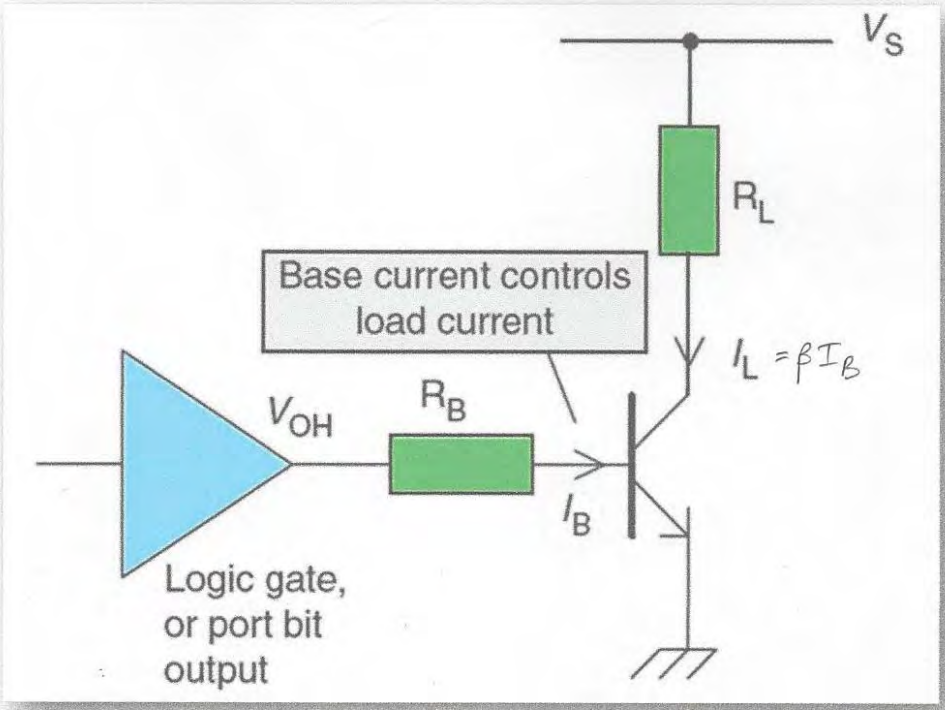
- Microcontrollers can drive loads with small electrical requirements
- Some devices, like actuators, require high currents or supply voltages
- Use switching devices
  - Simple DC switching using BJTs or MOSFETs
  - Reversible DC switching using H-bridge

*digital relays can be connected directly to µcontrollers, while other types might need high current to control the switch > Imax for the PORT.

Interfacing to Actuators

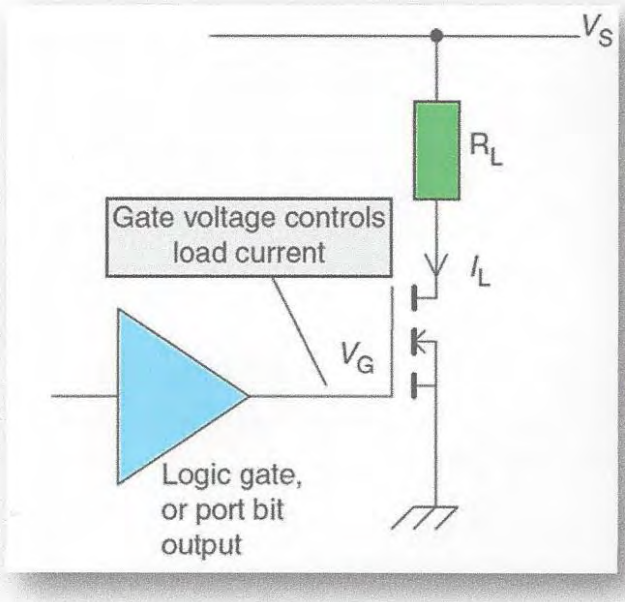
① interface directly when you have small voltage & current requirements.
② use an external source & put an electronic switch to control current flow (ON-OFF)

Simple DC interfacing

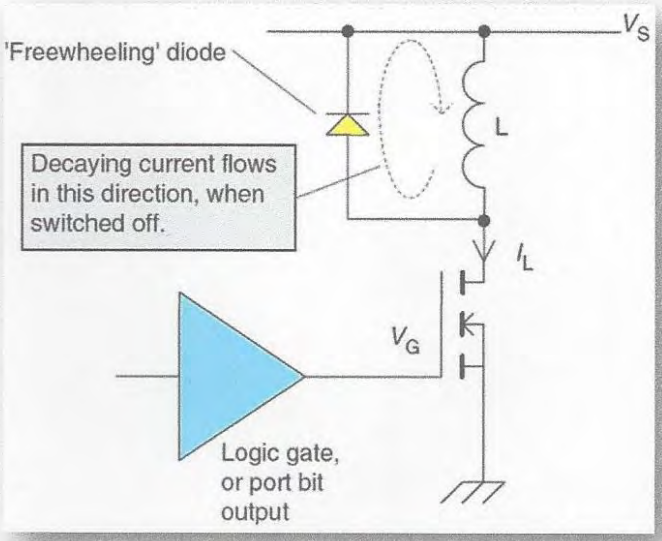


Interfacing to Actuators

Simple DC interfacing



Resistive load

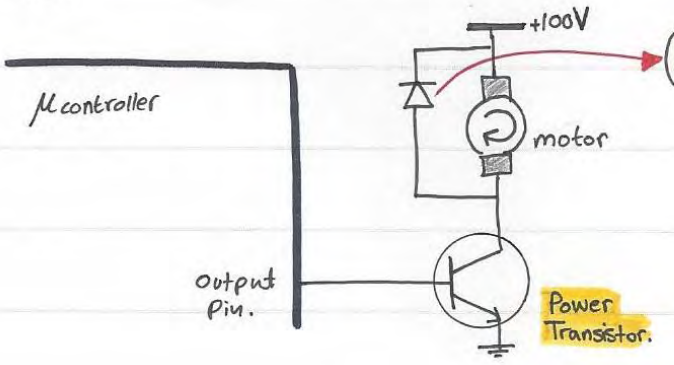


Inductive load

39

*This concept is general, not only for motors!

Example:



(Freewheeling diode): it's used to protect the switching device from being damaged by the reverse current of an inductive load. & it's placed so that it doesn't conduct when the current is being supplied to the load.

Interfacing to Actuators

Simple DC interfacing

Characteristics of two popular logic-compatible MOSFETs

Power Transistors:

Characteristic	ZVN4206A	ZVN4306A
Maximum drain-to-source voltage, V_{DS} (V)	60	60
Maximum gate-to-source threshold, $V_{GS(th)}$ (V)	3	3
Maximum drain-to-source resistance when 'on', $R_{DS(on)}$ (Ω)	1.5	0.33
Maximum continuous drain current, I_D	600 mA	1.1 A
Maximum power dissipation (W)	0.7	1.1
Input capacitance (pF)	100	350

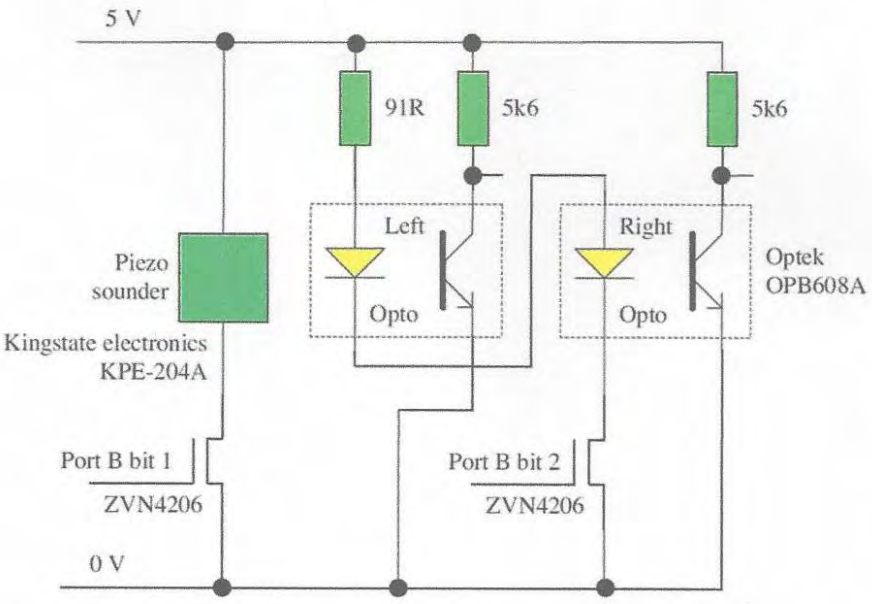
→ Why do we prefer MOSFET transistors over BJT?

→ we need almost zero current to control MOSFET transistor (what matters is the Gate voltage V_{GS}).

→ While in BJT needs base current that might reach few milli Amperes. to bias the transistor.

Interfacing to Actuators

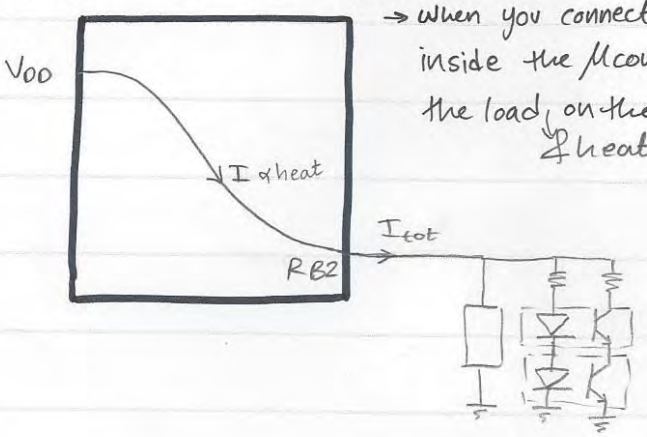
Driving Piezo Sounder and Opto-sensors



- Piezo sounder ratings: 9mA, 3-20 V
- The opto-sensor found to operate well with 91 Ohm resistor. The diode forward voltage is 1.7V. The required current is about 17.6 mA

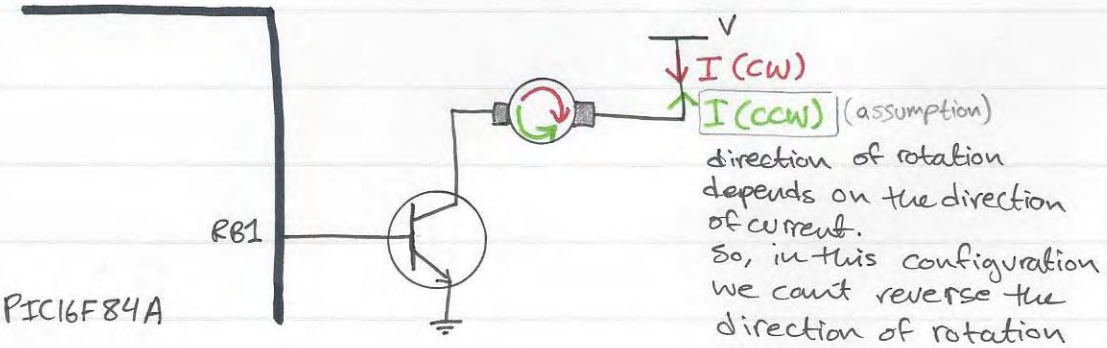
Slide(41): in this CKT we used external power supply = 5V, & total current required to drive the CKT was $I_{tot} = (17.6 + 9)m = 26.6\ mA$. So, since the required voltage = V_{DD} (internal μcontroller voltage) & $I_{tot} < I_{max}$ (that can be sourced by the PORT), why didn't we use the μcontroller to source the current?

① to reduce the load on μcontroller.



→ when you connect the CKT this way, the current passes inside the μcontroller to reach the port which will increase the load on the μcontroller. ↓ heat.

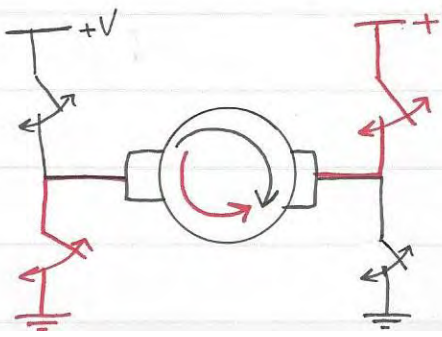
* let's take the following scenario:



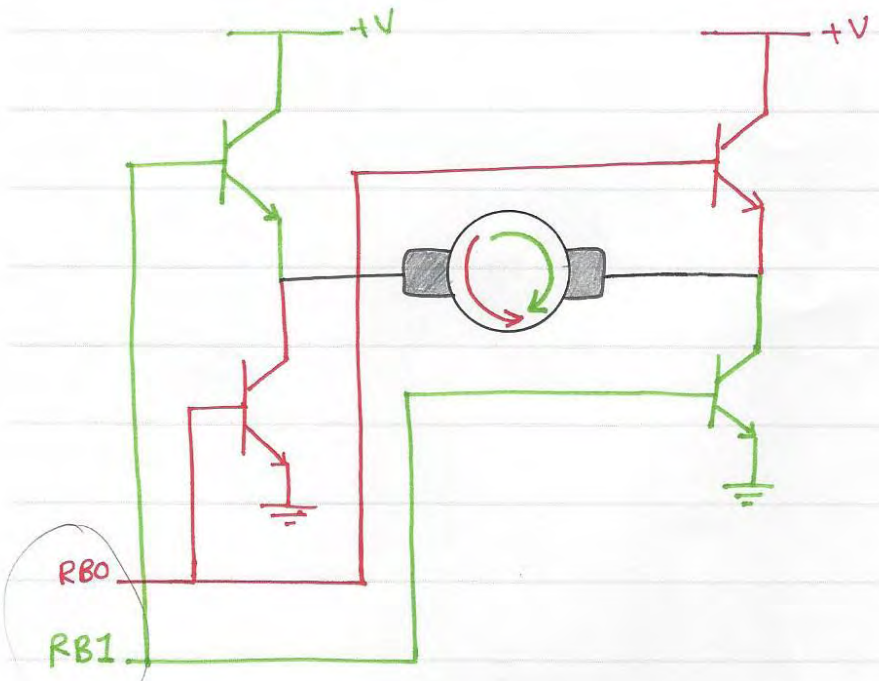
direction of rotation depends on the direction of current.
So, in this configuration we count reverse the direction of rotation since direction of the current is fixed.

Solution:

→ Basic idea: use switching.



problem? we don't want mechanical switches → thus we use transistors.



RB1	RB0	
0	0	; motor is OFF
0	1	; CCW rotation
1	0	; CW rotation

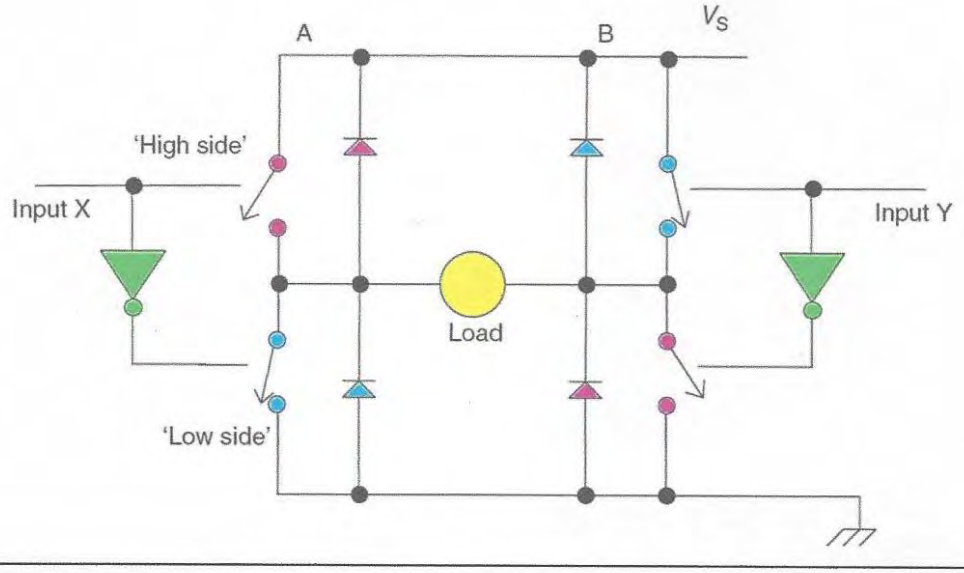
* instead of implementing this CKT, we can use an H-bridge IC [L2930] which contains 2 bridges.

↳ or we can use 1-to-2 decoder (this only applies when I want the motor to be always rotating, because when we use a decoder we can have either (01) or (10) but not (00)).

Interfacing to Actuators

Reversible DC Switching

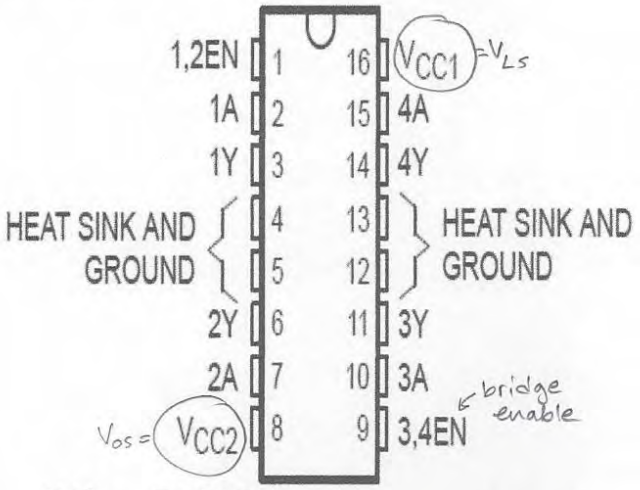
- DC switching allows driving loads with current flowing in one direction
- Some loads requires the applied voltage to be reversible; DC motors rotation depends on direction of current
- Use H-bridge !



*You can use it to connect up to 4 motors by dividing each of the 2 H-bridge into 2 half bridges & use them to connect 2 motors with fixed direction of rotation.

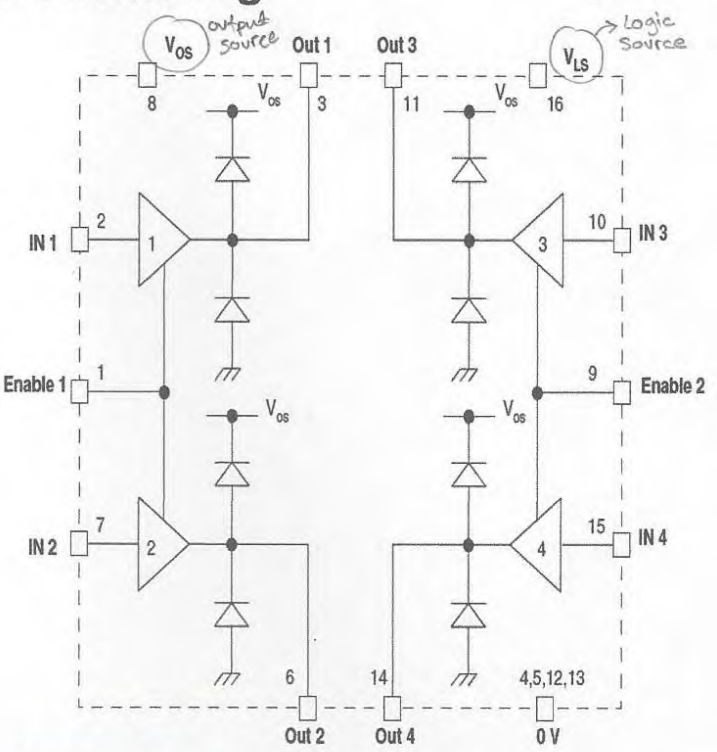
Interfacing to Actuators

Reversible DC Switching



2 Vcc sources:

1) one to drive the Logic (V_{LS})
2) one to drive the load (V_{OS})



L293D Dual H-bridge

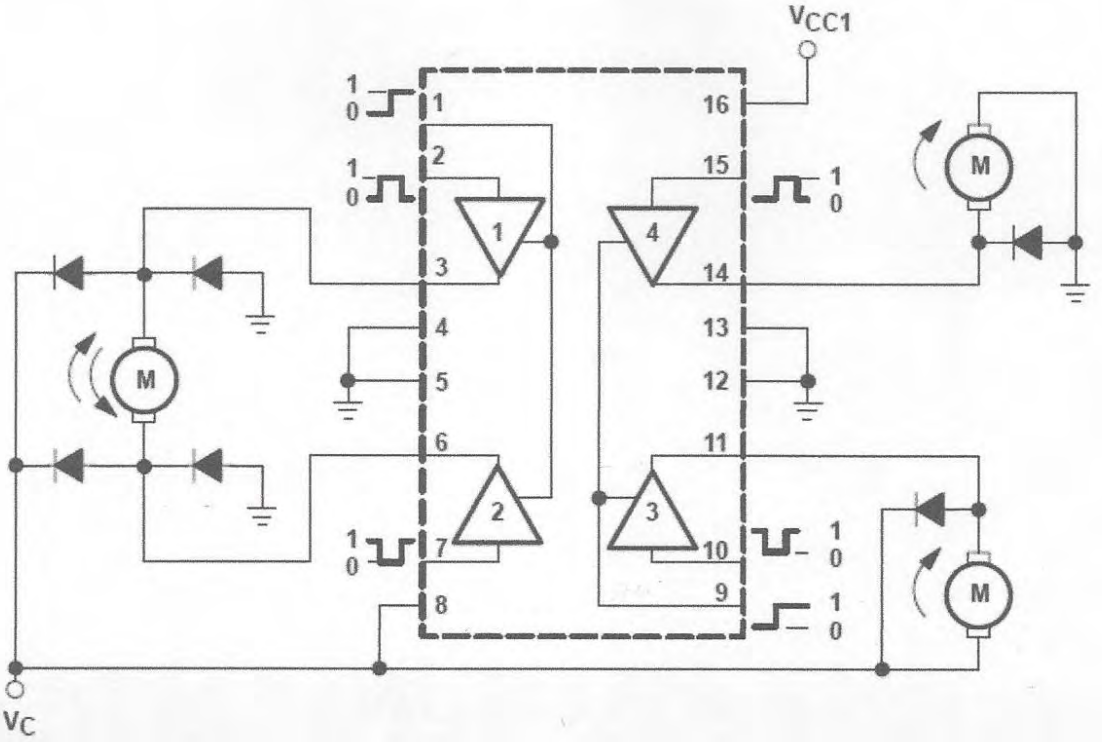
Peak output current 1.2 A per channel

thats why we connect it to a heat sink. pins 4:5 & 12:13

Interfacing to Actuators

Reversible DC Switching

Driving three motors using L293D

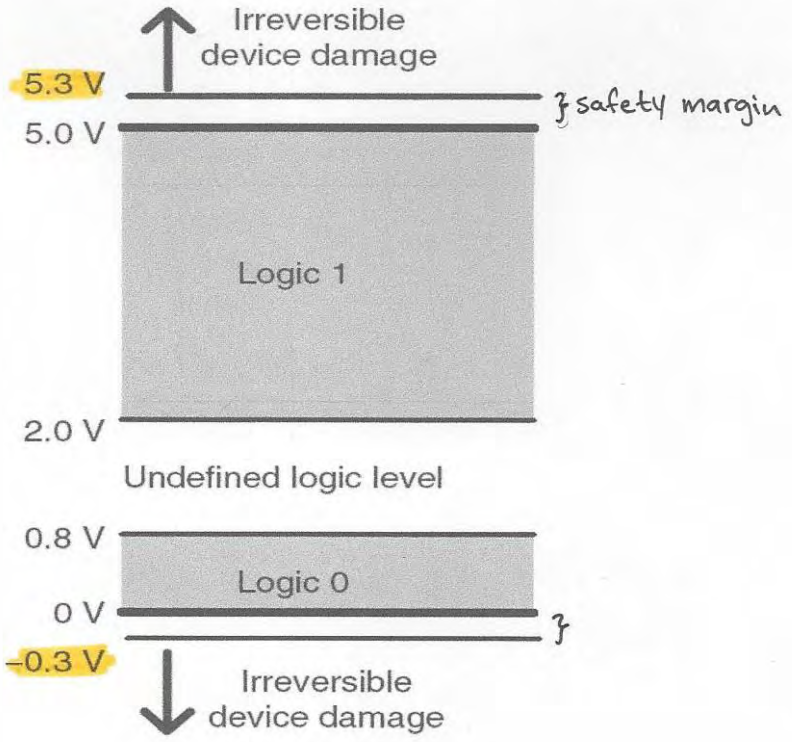


More on Digital Input

- When acquiring digital inputs into the microcontroller, it is essential that the input voltage is within the permissible and recognizable range of the MC
- Voltage range depends on the logic family; TTL, CMOS, ...
- Interfacing within the same family is safe
- What for the case
  - Interfacing to digital sensors
  - Signal corruption
  - Interference

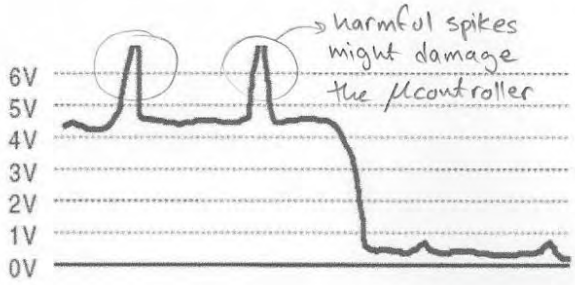
More on Digital Input

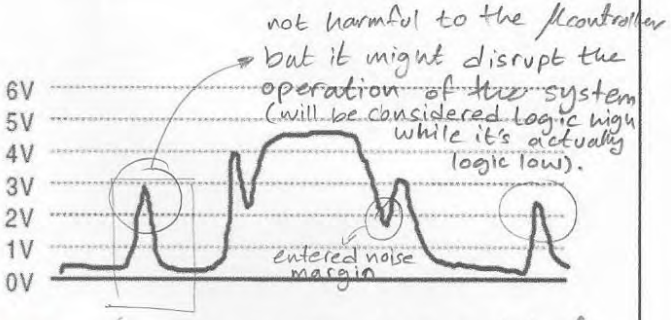
PIC16F873A Port Characteristics



More on Digital Input

Forms of Signal Corruption

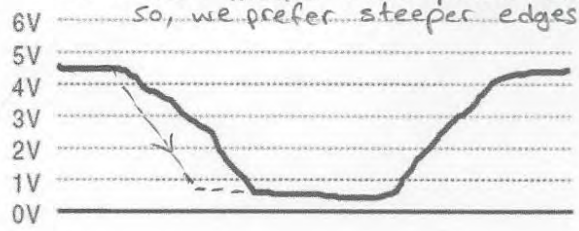




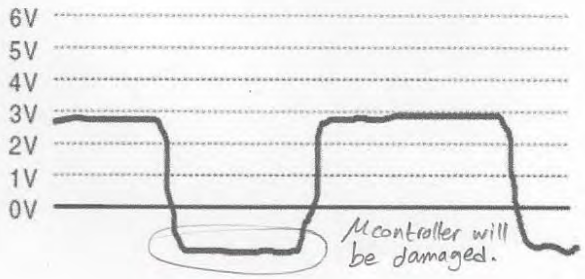
Spikes in the signal

↳ variations over a short period of time → high freq. components.
→ Solved by filtering (LPF)

Transition happens over a longer period of time, which means more power consumption. So, we prefer steeper edges



Slow edge

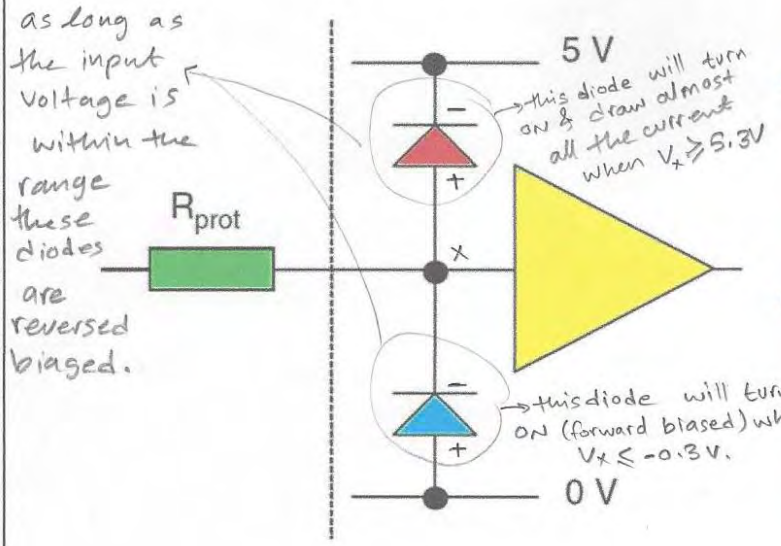


DC Offset in the signal

More on Digital Input

Clamping Voltage Spikes (for harmful spikes ↳ out of range)

as long as the input voltage is within the range these diodes are reversed biased.

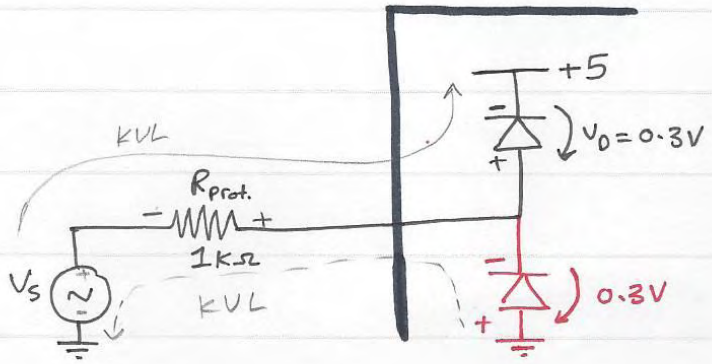


- All ports are usually protected by a pair of diodes
- An optional current limiting resistor can be added if high spikes are expected

- If R_{prot} = 1KΩ and the maximum diode current is 20 mA when Vd = 0.3v, then what is the maximum positive voltage spike that can be suppressed?

next

$I_{D_{max}} = 20mA$



$V_S - R_{prot.} * 20mA - 0.3 - 5 = 0$

$\boxed{V_{S_{max}} = 25.3\ V}$

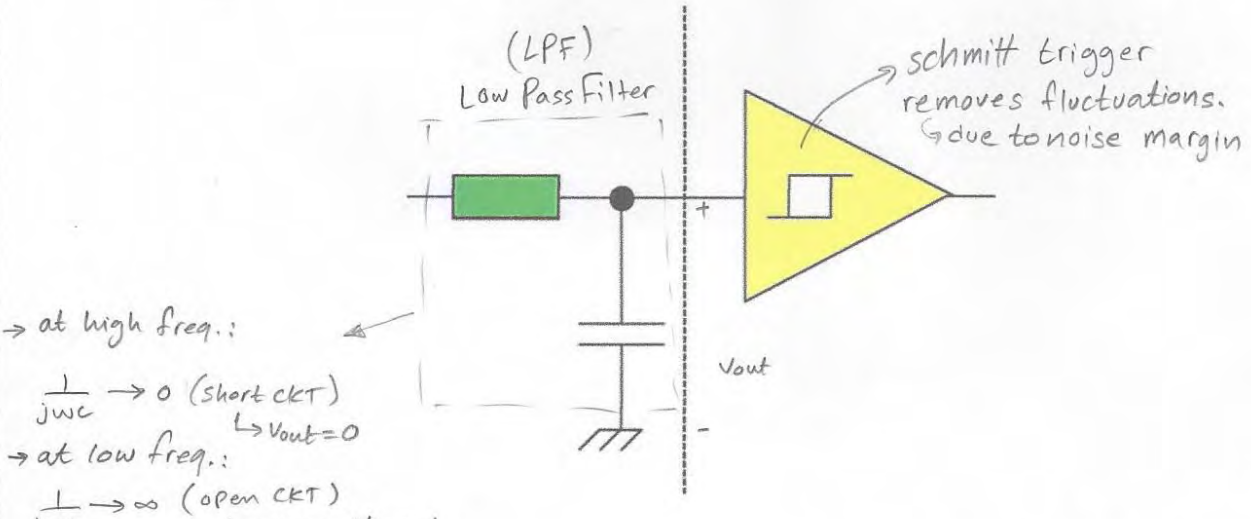
→ What is the minimum (max. negative) spike that can be suppressed?

$0.3 - R_{prot.} * 20mA - V_S = 0$

$V_S = -19.7\ V$

More on Digital Input

Analog Input Filtering

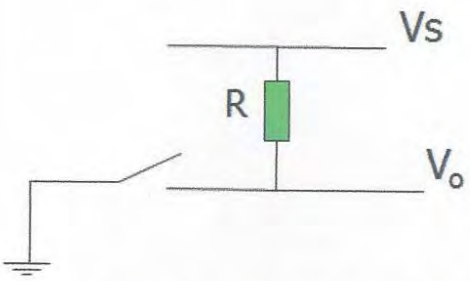


- Can use Schmitt trigger for speeding up slow logic edges.
- Schmitt trigger with RC filter can be used to filter voltage spikes.

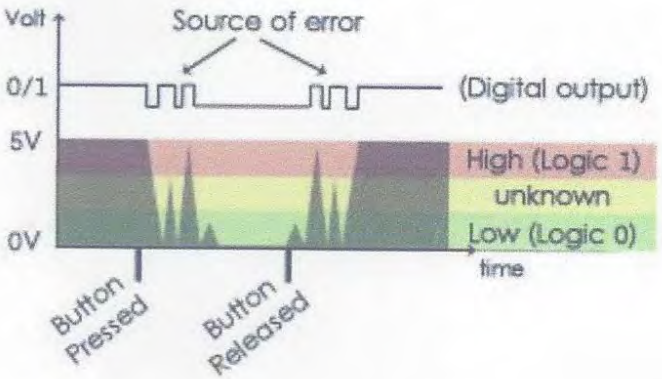
More on Digital Input

Switch Debouncing

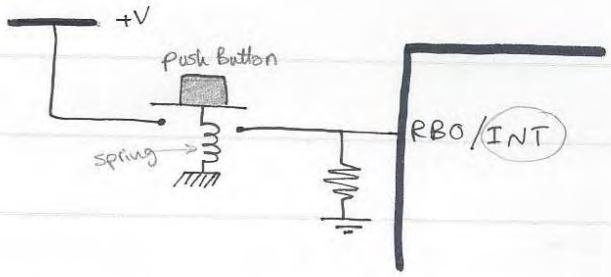
- Mechanical switches exhibit bouncing behavior
- The switch contact bounces between open and closed
- A serious problem for digital devices ?!

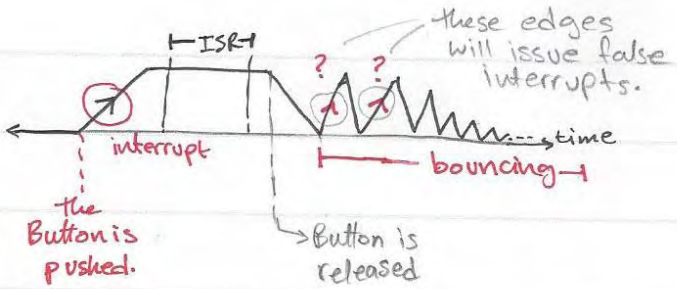


Simple switch interface



- Switch debouncing!! hardware and/or software techniques

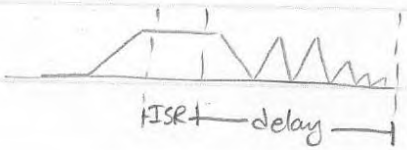




* to solve this problem:-

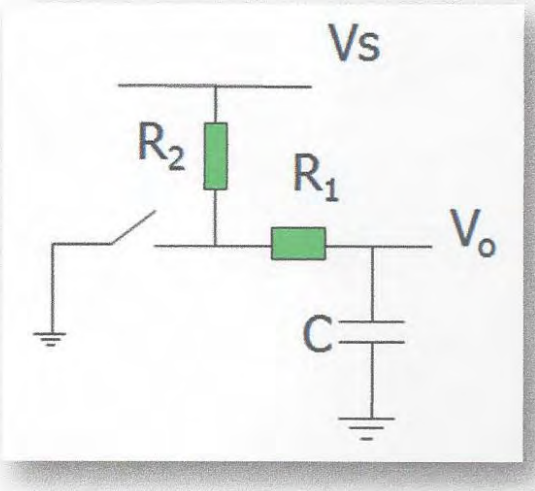
1- by hardware:
use schmitt trigger buffer (← to determine the logic level)
with LPF.
to suppress the spikes (high freq. components).

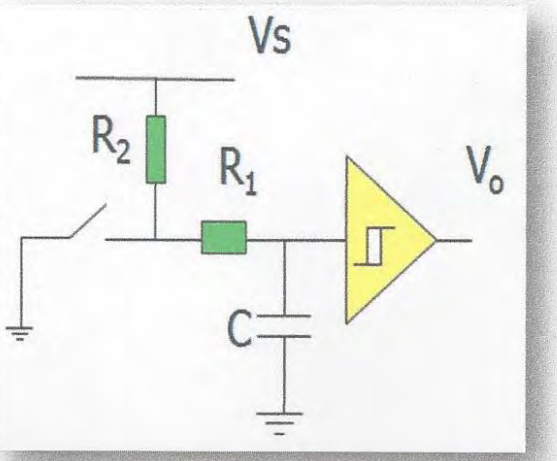
2- by software:
use software delays to make sure that ISR doesn't finish before releasing the Button & it settles down.

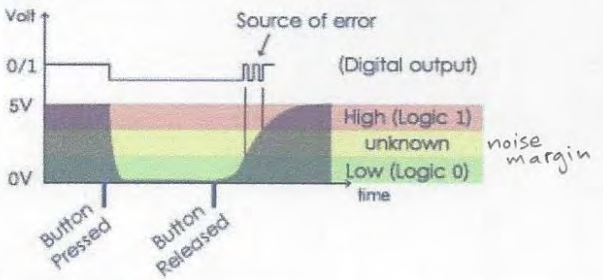


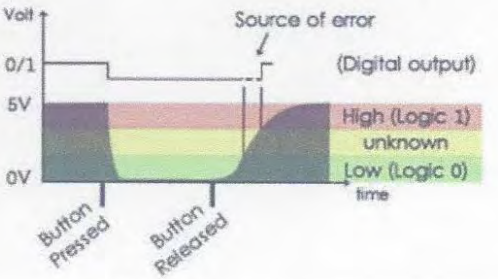
More on Digital Input

Switch Debouncing (HW solution)



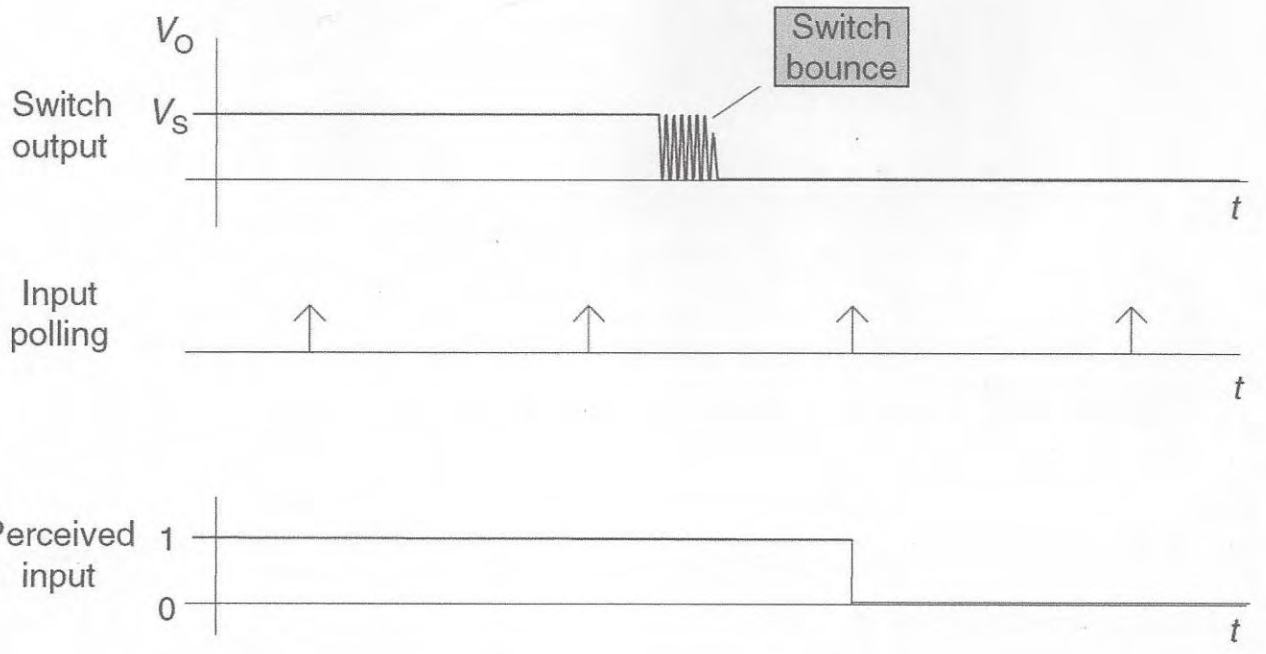






More on Digital Input

Switch Debouncing (SoftWare Solution).



Summary

- Microcontrollers must be able to interface with the physical world and possibly the human world
- Switches, keypads and displays represent typical examples for interfacing embedded systems with the humans
- Microcontrollers must be able to interface with a range of input and output transducers.
- Interfacing with sensors requires a reasonable knowledge of signal conditioning techniques
- Interfacing with actuators requires a reasonable knowledge of power switching techniques

Taking Timing Further

Chapter 9

Dr. Iyad Jafar

Outline

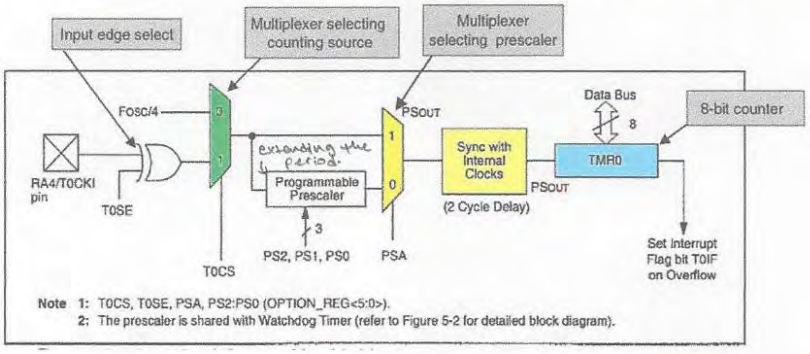
- Introduction
- Review of Timer 0 Module
- Timer 1 Module
- Timer 2 Module
- Capture/Compare/PWM (CCP)
- Digital-to-Analog Conversion
- Frequency Measurement
- Summary

not included in the final exam. X

Introduction

- **Why do we need timers ?**
  - Maintaining continuous counting functions
  - Recording ('capturing') in timer hardware the time an event occurs
  - Triggering events at particular times
  - Generating repetitive time-based events
  - Measuring frequency, e.g., motor speed

Review of Timer 0 Module



* prescalar can be used for both internal & external clocks. So, for example, you can increment the count once for each 4 push button presses.

File Address			File Address
00h	Indirect addr.(1)	Indirect addr.(1)	80h
01h	TMR0	OPTION_REG	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **RBPU**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

More Timer Modules

Device	Pins	Features
16F84A	18	**1 8-bit timer** 1 5-bit port 1 8-bit port
16F873A 16F876A	28	3 parallel ports, **3 counter/timers,** **2 capture/compare/PWM,** 2 serial, 5 10-bit ADC, 2 comparators
16F874A 16F877A	40	5 parallel ports, **3 counter/timers,** **2 capture/compare/PWM,** 2 serial, 8 10-bit ADC, 2 comparators

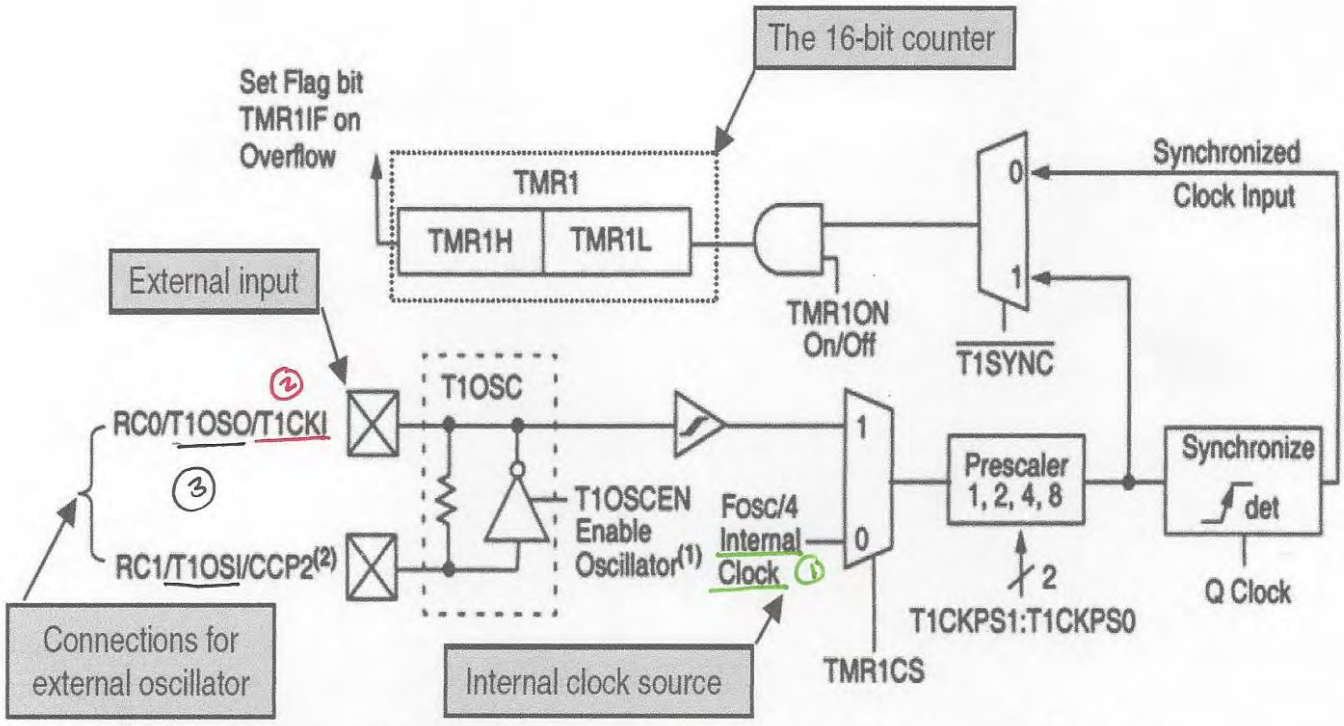
* counter → for some asynchronous event
* timer → counter for a square wave signal with known freq.

Timer 1 Module

- **Features**
  - 16-bit timer/counter (0000H – FFFFH)
  - Count value in TMR1H (0x0F) and TMR1L (0x0E)
  - TMR1 operation controlled by T1CON (0x10)
  - Three clock sources
    1. Internal clock Fosc/4
    2. External input (RC0/T1OSO/T1CKI) for counting purposes
       - Count on rising edge (after the first falling edge)
    3. External oscillator (RC1/T1OSI/CCP2)
       - Removes the dependency on the main oscillator
       - Intended for low frequency oscillation up to 200KHz (typically 32.768 KHz)
       - Counting continue in sleep mode

↳ to extend time

Timer 1 Module



Note 1: When the T1OSCEN bit is cleared, the inverter is turned off. This eliminates power drain.

Timer 1 Module

T1CON Register (0x10)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7:6 **Unimplemented:** Read as '0'

bit 5:4 **T1CKPS1:T1CKPS0:** Timer1 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:** Timer1 Oscillator Enable bit

1 = Oscillator is enabled
0 = Oscillator is shut off. The oscillator inverter and feedback resistor are turned off to eliminate power drain

bit 2 **T1SYNC:** Timer1 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. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS:** Timer1 Clock Source Select bit

1 = External clock from pin T1OSO/T1CKI (on the rising edge)
0 = Internal clock (Fosc/4)

bit 0 **TMR1ON:** Timer1 On bit

1 = Enables Timer1
0 = Stops Timer1

there's an explicit bit to turn ON & off the timer (while in TMR0 there wasn't. but it was connected to the external clock by default, so when the source of the clock is changed to internal or an external source is connected, TMR0 starts counting immediately.

Timer 2 Module

- **Features**
  - 8-bit counter/timer
  - Count value in TMR2 register (0x11)
  - TMR2 operation controlled by T2CON (0x12)
  - No external clock input
  - Has Capture and Compare register PR2 (0x92) and pulse width modulation capability

to be accurate it's a timer not a counter since it has no choice to connect an external clock as a source.

Timer 2 Module

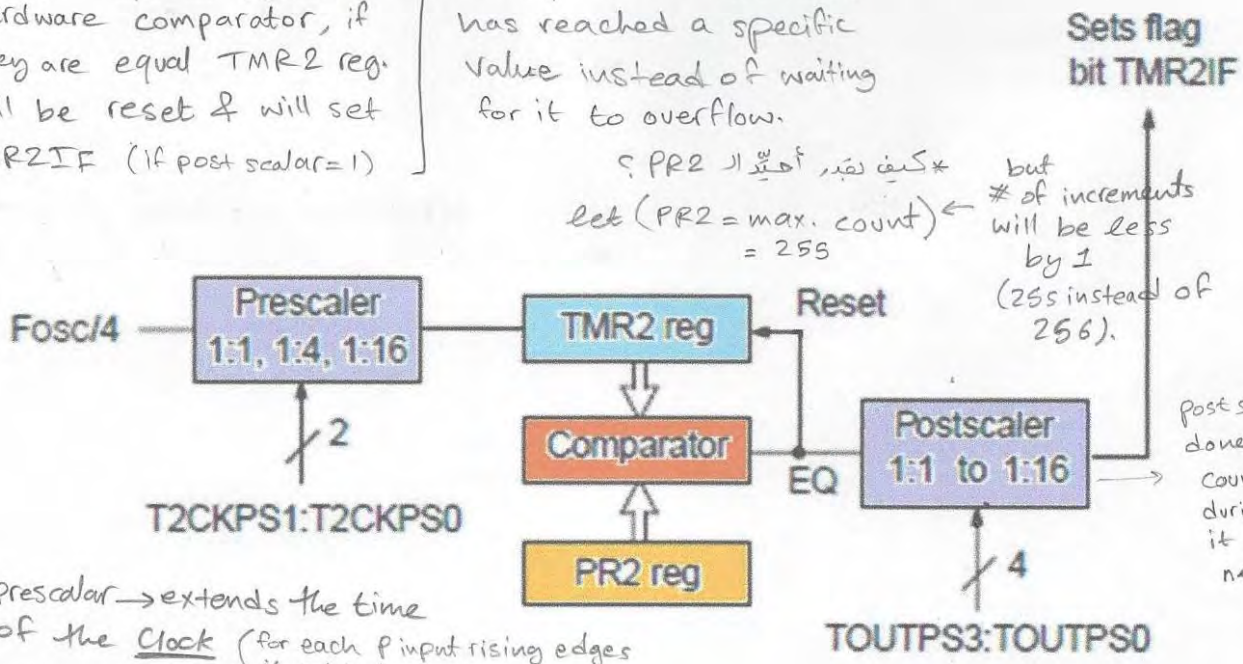
TMR2 reg. value is continuously compared with the value in PR2 reg. (TMR2 period register) using a special dedicated hardware comparator, if they are equal TMR2 reg. will be reset & will set TMR2IF (if post scalar=1)

→ this is used to issue an interrupt when the timer has reached a specific value instead of waiting for it to overflow.

* كيف نعيد أعلى PR2 ؟

let (PR2 = max. count) = 255

but # of increments will be less by 1 (255 instead of 256).



Post scaling can be done by software count # of occurrence during ISR untill it reaches the needed value.

* Prescalar → extends the time of the Clock (for each P input rising edges it outputs 1 2 2)

* Post scalar → determines the # of occurrence of the incident before issuing an interrupt. (for each P incidents, the flag will be set once).

further extension of time.

Timer 2 Module

T2CON Register (0x12)

U-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

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

bit 6:3 **TOUTPS3:TOUTPS0**: Timer2 Output Postscale Select bits

0000 = 1:1 Postscale
0001 = 1:2 Postscale
•
•
•
1111 = 1:16 Postscale

bit 2 **TMR2ON**: Timer2 On bit

1 = Timer2 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

Timer 2 Module

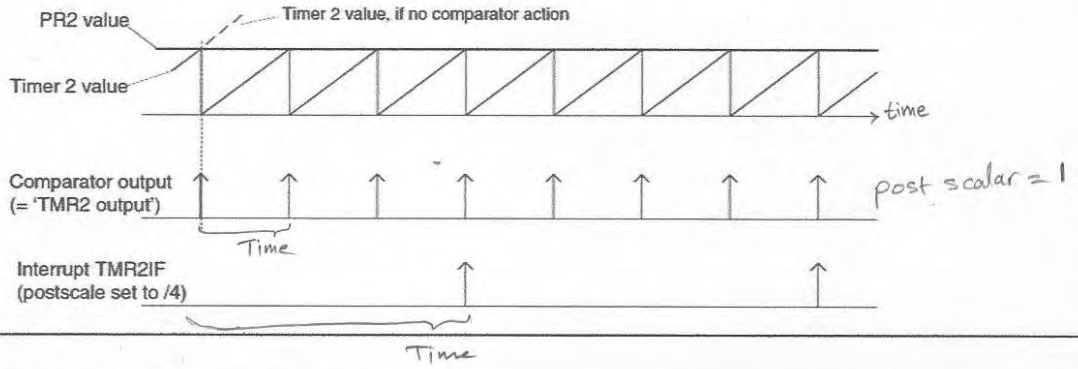
The PR2 register, comparator and prescaler

- Timer2 has a period register PR2 (0x92) that can be preset by the programmer
- The content of this register is continuously compared with the Timer2 when it is running
- When TMR2 equals PR2,
  - TMR2 is cleared
  - The comparator output (same as TMR2IF in PIR) is high which can be used as interrupt if TMR2IE (PIE) is set
  - The comparator output can be post-scaled by T2OUTPS3:T2OUTPS0 bits (T2CON)

1 cycle is needed to reset the timer.

$$Time = \frac{4}{Fosc} \times Prescalar * (PR2 - TMR2_0 + (1)) * Post$$

$TMR2_0$: TMR2 initial value



Capture/Compare/PWM Modules

G(record) G(Alarm)

- Embedded systems need to deal with time events such as setting an alarm or recording the time of an event
- This can be easily achieved by adding one or more registers to the timer/counter registers
  - A register that records the time. It is called the Capture register
  - A register that triggers an alarm. It is called the Compare register
- The PIC 16 series combine these functionalities in the Capture/Compare/PWM (CCP) modules which interact with Timer1 and Timer2 modules

CCP Mode	Timer Resource
Capture	Timer1
Compare	Timer1
PWM	Timer2

- **The PIC16F873A has two such modules**
  - Each has two 8-bit registers CCP1H (0x16) and CCP1L (0x15) for module CCP1 and CCP2H (0x1C) and CCP2L (0x1B) for module CCP2
  - These registers can be used to capture a value from the timer, store the value to compare with, or store the duty cycle of PWM stream
  - Mode of operation is controlled by CCP1CON (0x17) and CCP2CON (0x1D) registers

* in order to 'capture' time we need:

1) Timer 2) register to store the 'captured' time 3) inteface with the event (electrical signal, it might be edge triggered or level triggered.)

* in order to 'compare' time we need:

1) timer 2) register to store the value to be compared with timer 3) Action (to do when values are equal).

modules

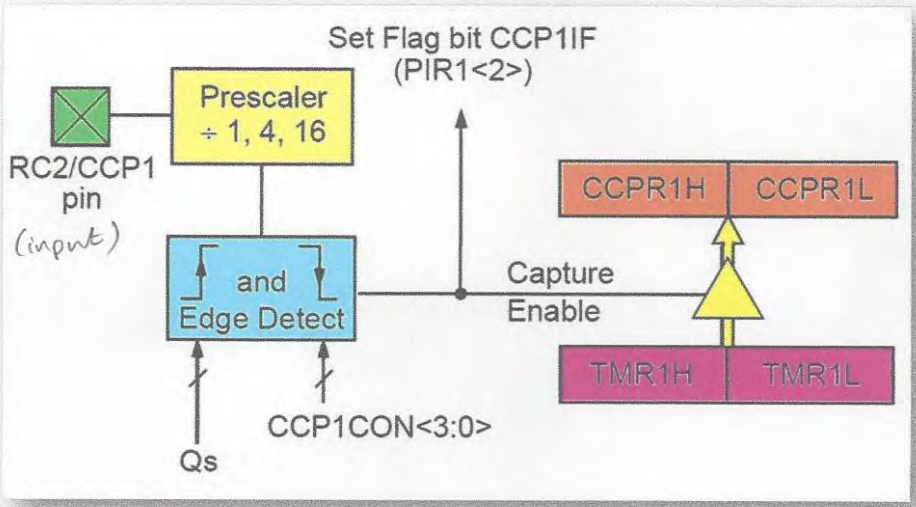
CCP1	CCP2	
CCP1H:CCP1L	CCP2H:CCP2L	← data registers
CCP1CON	CCP2CON	← configuration registers
RC2 pin/CCP1	RC1 pin/CCP2	

these registers are used to store the captured value of timer 1 based on some event. OR to store the set value to be compared to TMR1. OR to store the duty cycle of the PWM stream.

Capture/Compare/PWM Modules

Capture Mode

- The compare register operates like a stopwatch!
- Can record the value of the timer when an event occurs

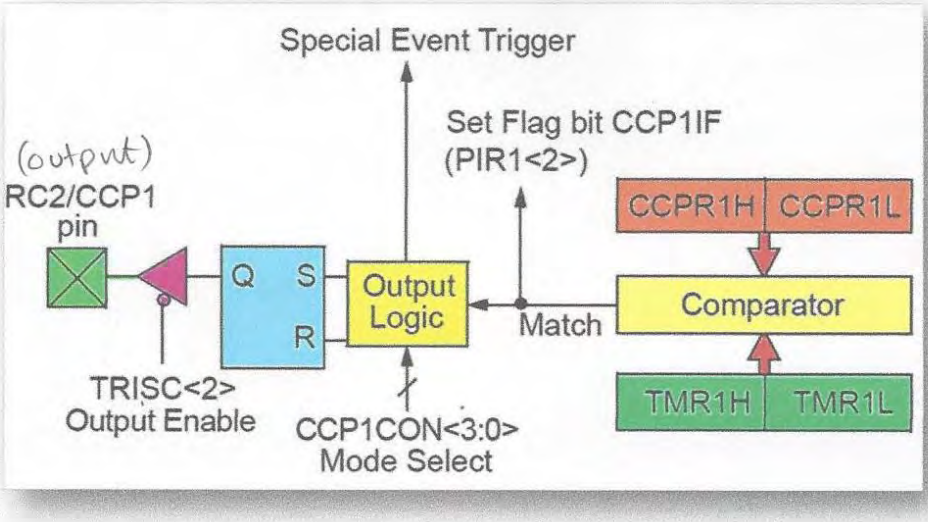


Block diagram of CCP1 module in capture mode

Capture/Compare/PWM Modules

Compare Mode

- The value stored in CCPR1H and CCPR1L is continuously compared to Timer1 registers
- The associated output pin can be set or cleared



Block diagram of CCP1 module in compare mode

Capture/Compare/PWM Modules

CCP Control Registers: CCP1CON and CCP2CON

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
bit 7							bit 0

bit 7:6 **Unimplemented:** Read as '0'

bit 5:4 **DCxB1:DCxB0**: PWM Duty Cycle bit1 and bit0

Capture Mode:
Unused

Compare Mode:
Unused

PWM Mode:

These bits are the two LSbs (bit1 and bit0) 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**: CCPx Mode Select bits

0000 = Capture/Compare/PWM off (resets CCPx module)
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,
Generate software interrupt on compare match
(CCPIF bit is set, CCP pin is unaffected)
1011 = Compare mode,
Trigger special event (CCPIF bit is set)
11xx = PWM mode

in falling edge mode ↑ you can't use prescalar

in this case, an internal Hardware trigger is generated which may be used to initiate an action.

8.2.4 SPECIAL EVENT TRIGGER

In this mode, an internal hardware trigger is generated which may be used to initiate an action.

The special event trigger output of CCP1 resets the TMR1 register pair. This allows the CCPR1 register to effectively be a 16-bit programmable period register for Timer1.

The special event trigger output of CCP2 resets the TMR1 register pair and starts an A/D conversion (if the A/D module is enabled).

Note:	The special event trigger from the CCP1 and CCP2 modules will not set interrupt flag bit TMR1IF (PIR1<0>).

INTERACTION OF CCP MODULES

Due to the modularity of the PIC16CXXX peripherals, future devices with two or more CCP modules on a device are possible. Each CCP module operates independently from the others, though their interaction with the timer resources must be taken into account.

When two or more CCP modules exist on a device, there can be an interaction between the CCP modules. This interaction is shown in Table 3. These interactions do NOT include any interaction (S/W) caused by the main program nor the interrupt service routines of the CCP sources.

Interaction of Two Capture Modes

When two CCP modules are in a Capture mode, Timer1 is the time-base for both captures. This means that they will have the same capture resolution, as determined by the Timer1 prescaler and frequency of the timer/counter clock. This clock can come from an external source (on the RC0/T1OSO/T1CKI pin), but must be synchronized to the device.

Interaction of One Capture Mode and One Compare Mode

When one CCP module is in a Capture mode and a second CCP module is in Compare mode, Timer1 is the time-base for both the captures and the compare. This means that the capture and the compare will have the same resolution, as determined by the Timer1 prescaler and frequency of the timer/counter clock. This clock can come from an external source (on the RC0/T1OSO/T1CKI pin), but must be synchronized to the processor clock. Also, care must be taken in that the compare can be configured to clear TMR1 register (when in special Trigger mode). Care must be taken in system design to ensure that this clearing of the TMR1 register does not have any negative impact on the capture function.

Interaction of Two Compare Modes

When two CCP modules are in a Compare mode, Timer1 is the time-base for both compares. This means that they will have the same compare resolution, as determined by the Timer1 prescaler and frequency of the timer/counter clock. This clock can come from an external source (on the RC0/T1OSO/T1CKI pin), but must be synchronized to the processor clock. Since the compare modules can be configured to clear TMR1 register (when in special Trigger mode), care must be taken in system design to ensure that this clearing of the TMR1 register does not have any negative impact on the compare function. If both compares are configured with a special trigger, which clears the TMR1 register, then the compare register that is closest to (but greater than) the TMR1 register value is the compare value that will reset the TMR1 register. Example 1 shows a possible case.

Interaction of Two PWM Modes

When two CCP modules are in a PWM mode, Timer2 is the time-base for both PWM outputs. This means that they will have the same PWM frequency and update rates, as determined by the Timer2 prescaler and frequency of the device. The resolution of the two PWMs may be different, since each CCP module has its own CCPxX:CCPxY bits for high resolution mode. These bits are found in the CCPxCON<5:4> register.

CONCLUSION

The Capture/Compare/PWM modules offer enormous flexibility in the use of the device timer resources. As with all resources, care must be taken to ensure that no adverse system complications can occur with the interaction between multiple CCP modules. The programs for simple operation of the various CCP modes should be a good foundation for modifications to suite your particular needs.

Table 14-3: Interaction of Two CCP Modules

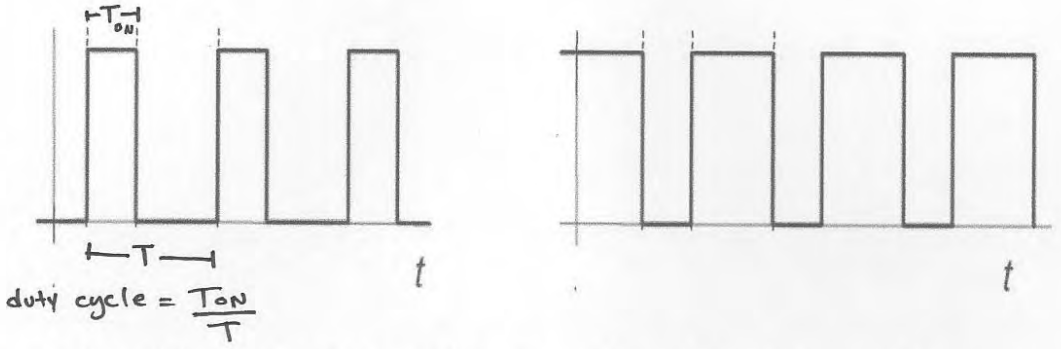
CCPx Mode	CCPy Mode	Interaction
Capture	Capture	Same TMR1 time-base.
Capture	Compare	The compare should be configured for the special event trigger, which clears TMR1.
Compare	Compare	The compare(s) should be configured for the special event trigger, which clears TMR1.
PWM	PWM	The PWMs will have the same frequency, and update rate (TMR2 interrupt).
PWM	Capture	None
PWM	Compare	None

Capture/Compare/PWM Modules

Pulse Width Modulation

e.g: servo motors.

- In many applications, it is required to have a stream of pulses with controllable width/duration

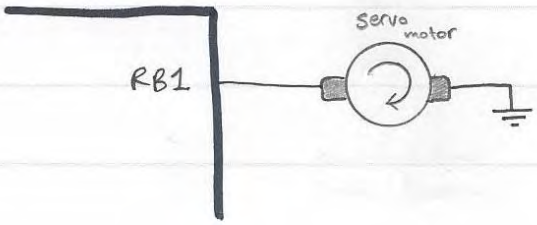


duty cycle = $\frac{T_{ON}}{T}$

- In embedded systems this can be done in software or hardware

How can we do it?

1) By Software:



→ 1st Configure RB1 as output

```
→ BSF   PORTB,1
  CALL  DELAY_ONTIME
  BCF   PORTB,1
  CALL  DELAY_OFFTIME
```

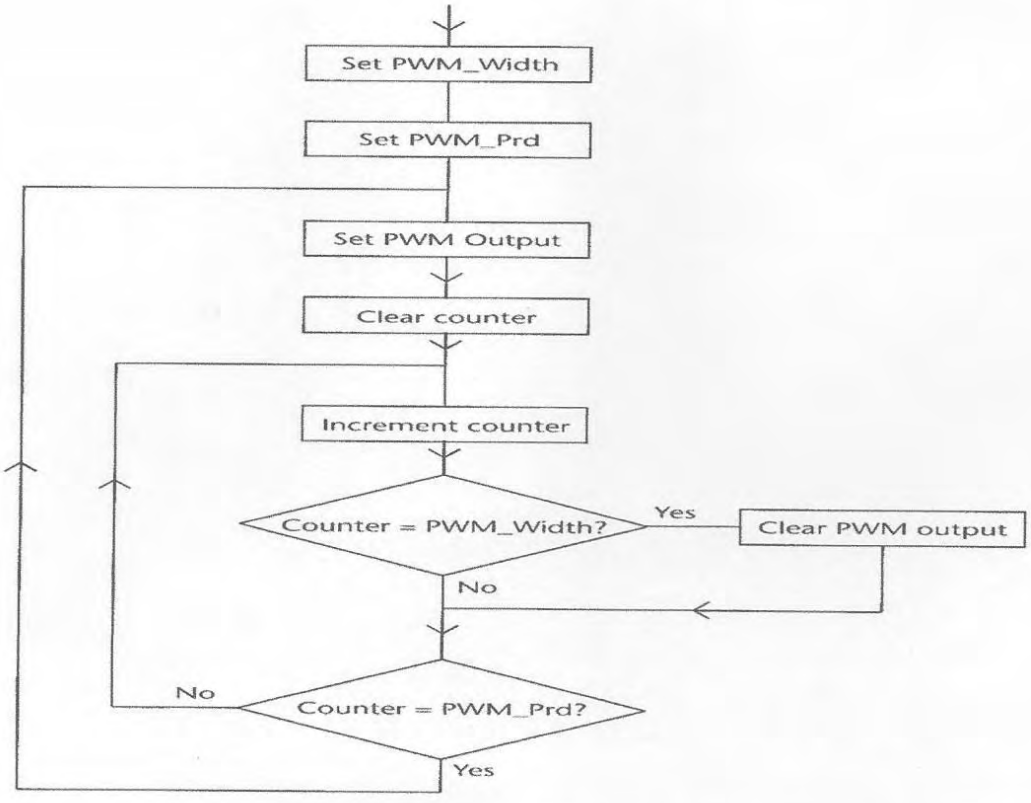
but in this case, the µcontroller is occupied & reserved to run this code.

better solution is to use hardware delays (e.g: TMR0) with interrupts. instead of software delay loops.

2) By hardware : (use ccp module).

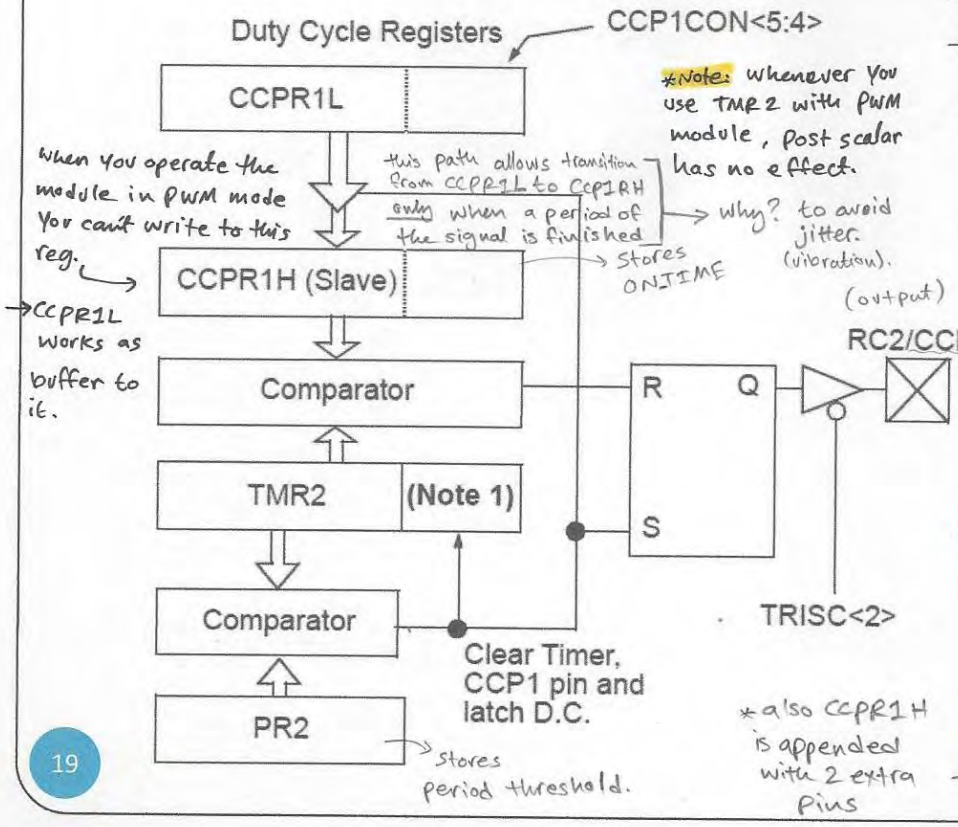
Capture/Compare/PWM Modules

Pulse Width Modulation (software)

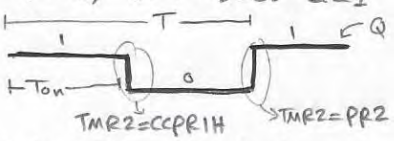


Capture/Compare/PWM Modules

Pulse Width Modulation (using CCP module)



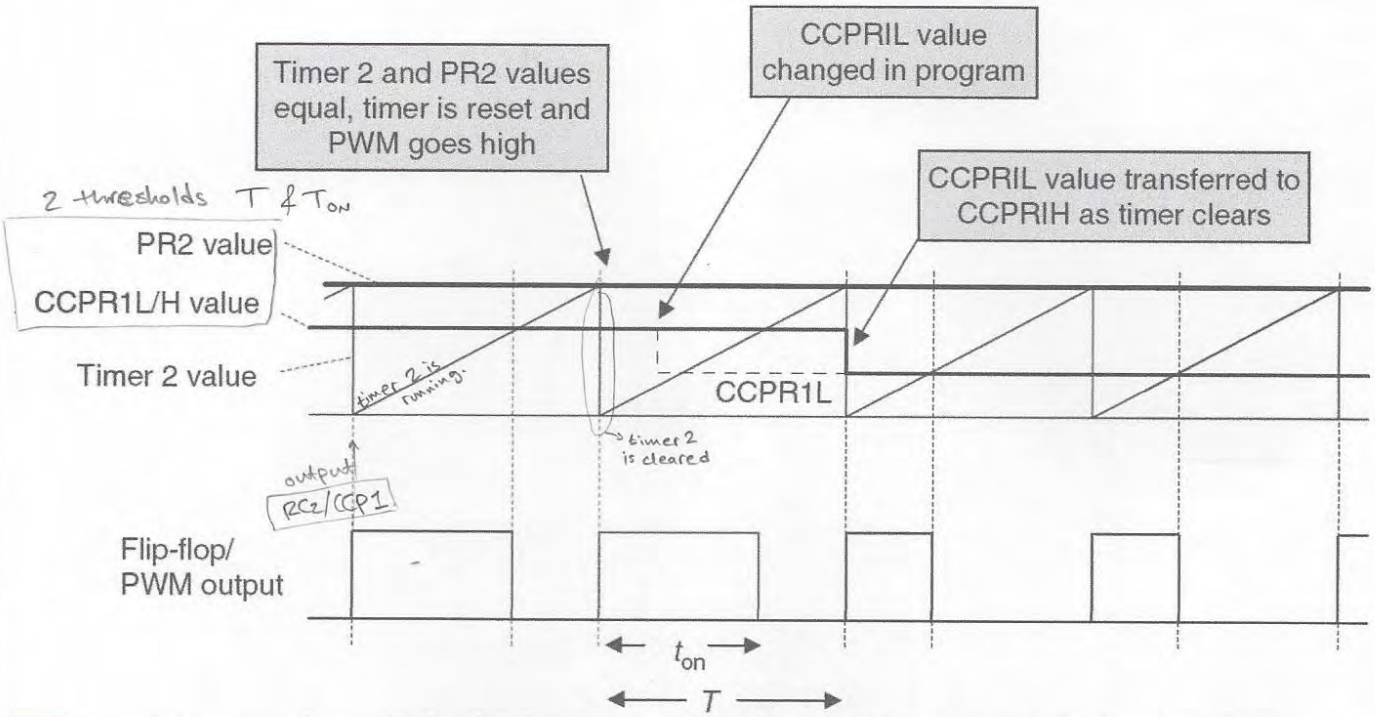
→ assume Q is initially 1
→ the 2 reg. are continuously compared with TMR2 value.
* when no equality occurs
R=0, S=0 → hold the previous value (no change). Q=1
* when TMR2 = CCPRH (ON time)
R=1, S=0 → reset → Q=0
* when TMR2 = PR2 (cycle time)
R=0, S=1 → set Q=1



- **Note 1:** The 8-bit timer is concatenated with 2-bit internal Q clock, or 2 bits of the prescaler, to create 10-bit time base.

Capture/Compare/PWM Modules

Pulse Width Modulation (using CCP module)



* PWM signals might also be used with a DC motor to control its speed
TON ↓ → Speed ↓ → ↓ Power consumption.

Capture/Compare/PWM Modules

Pulse Width Modulation (using CCP module)

Calculations

$T = (\mathbf{PR2} + 1) \times (\text{Timer2 input clock period})$

$\quad = (\mathbf{PR2} + 1) \times \{T_{osc} \times 4 \times (\text{Timer2 prescale value})\}$

عشان نضمن انه T>Ton
لانه Ton ← 10-bit
T ← 8-bit

$t_{on} = (\text{pulse width register}) \times (\text{PWM timer input clock period})$

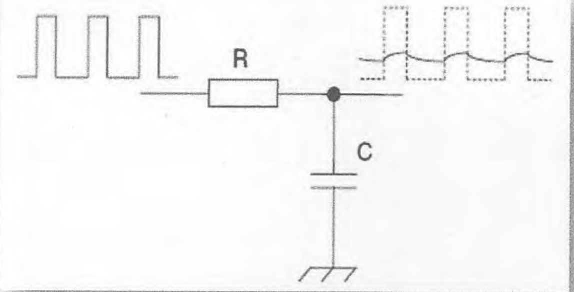
$\quad = (\text{pulse width register}) \times \{T_{osc} \times (\text{Timer2 prescale value})\}$

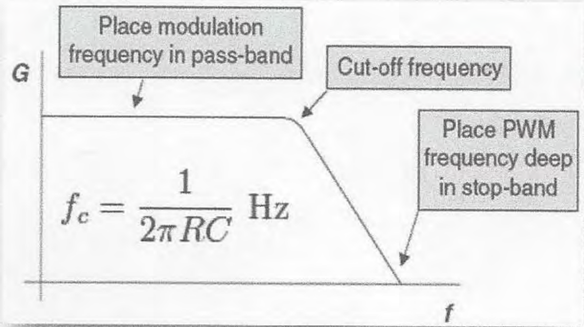
pulse width register = ***CCPR1L :: CCP1CON<5:4>***

PWM and Digital To Analog Conversion

not included

- PWM is perhaps primarily used for load control
- Can be used for simple and effective digital-to-analog conversion
  - Space-ratio is fixed
    - Low pass filter the PWM stream to obtain a DC signal with some ripple





  - Space-ratio is modulated
    - Varying output voltage is produced

PWM and Digital To Analog Conversion

- Generating a Sine Wave - change the on-time for the PWM signal so the output of the LPF will be different

```
        clrf   pointer
sin_loop
        movf   pointer,w
        call   sin_table   ;get most significant byte
        movwf  ccpr1l      ;move it to the PWM output
        incf   pointer,f   ;increment the pointer
        movf   pointer,w
        call   sin_table    ;get the MS byte
        andlw  B'11000000' ;we only use ms 2 bits
```


PWM and Digital To Analog Conversion

- Generating a Sine Wave

```
        movwf  temp
        bcf    status,c     ;adjust for CCP1CON
        rrf    temp,f
        rrf    temp,w
        iorlw  B'00001100' ;set some CCP1CON bits
        movwf  ccp1con
        incf   pointer,f
        movf   pointer,w
        …
        call   delay1
        goto   sin_loop
```

PWM and Digital To Analog Conversion

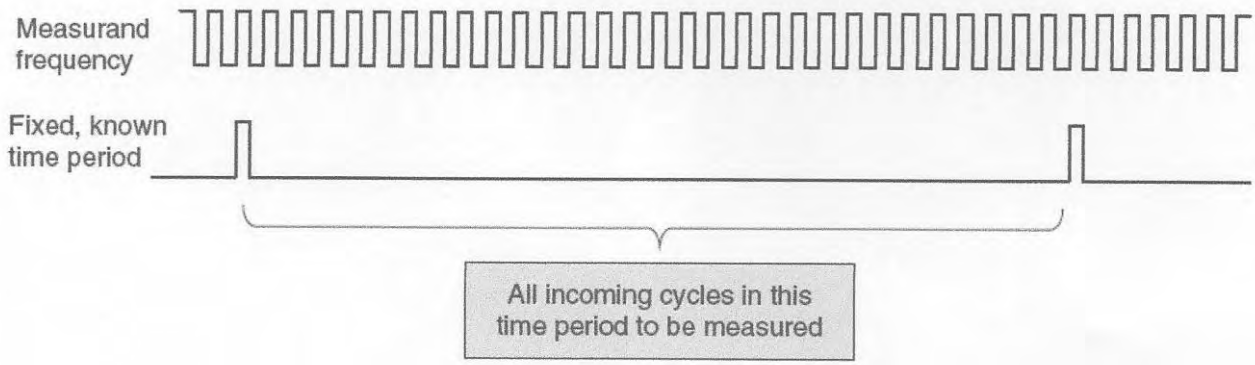
- Generating a Sine Wave

```
Sin_Table
        addwf pcl,1
        retlw 00      ;0 degrees, higher byte
        retlw 00      ;0 degrees, lower byte
        retlw 03      ;2 degrees, higher byte
        retlw 5A      ;2 degrees, lower byte
        retlw 06      ;4 degrees, higher byte
        retlw 0B2     ;4 degrees, lower byte
        ......
        ......
```


Frequency Measurement

not included

- Frequency measurement is a very important application of both counting and timing
- Both a counter and a timer are needed
  - The timer to measure the reference period of time
  - The counter to count the number of events within that time.



Example 1

Write a program to flash a LED that is connected to RA0 continuously such that it is ON for 3 seconds and OFF for 3 seconds. Use TIMER1 module to generate the delay and assume Fosc = 4MHz.

TABLE 6-2: REGISTERS ASSOCIATED WITH TIMER1 AS A TIMER/COUNTER

Address	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
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	--00 0000	--uu uuuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer1 module.

Example 1

- Maximum time that can be measured by TMR1 is

  Time = 2^16 * 4/Fosc * Prescaler

  = 65536*1usec*8 = 0.5243 s (max. time.)

- How about we configure TMR1 to measure 0.5 sec and use a software counter (post-scaler) to count six times

  0.5 = N * 1 usec * P → N = 62500, P = 8

  TMR1H:TMR1L = 2^{16} 65536 – 62500 = 3036 = 0x0BDC (initial value.)

  → TMR1H = 0x0B , TMR1L = 0xDC

- T1CON = 0x30

--	--	T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	T1ON
0	0	1	1	0	0	0	0

Example 1

```
COUNTER     EQU     0x20
            # include   "PIC16F877.INC"
            org     0x0000
            goto    START
            org     0x0004
ISR         goto    ISR
START       bcf     STATUS, RP1     ; select bank 1
            bsf     STATUS, RP0
            clrf    TRISA           ; set RA0 as output
            movlw   B'00000110'     ; configure RA0 as digital
            movwf   ADCON1          it's analog by default.
            bcf     STATUS, RP0
FLASH       movlw   0x06            ; initialize counter to 6
            movwf   COUNTER
WAIT_3sec   movlw   0x0B
            movwf   TMR1H           ; initialize TMR1H
```


Example 1

```
            movlw   0xDC
            movwf   TMR1L           ; initialize TMR1L
            movlw   0x30
            movwf   T1CON           ; initialize T1CON
            bsf     T1CON, TMR1ON   ; enable timer 1
WAIT_p5sec  btfss   PIR1, TMR1IF    ; wait for overflow
            goto    WAIT_p5sec
            bcf     T1CON, TMR1ON   ; stop timer
            bcf     PIR1, TMR1IF    ; clear interrupt flag
            decfsz  COUNTER, F
            goto    WAIT_3sec
            movlw   0xFF            ; change the state of RA0
            xorwf   PORTA, F
            goto    FLASH
            end
```

Example 2

Consider the contents of the following registers

TMR2 = D'44'

PR2 = D'100'

T2CON = 0x39

If the instruction *bsf T2CON, T2ON* is executed, then how long does it take to set the TMR2IF in the PIR1 register ? Assume Fosc = 8 MHz.

Example 2

T2CON

--	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	T2ON	T2CKPS1	T2CKPS0
0	0	1	1	1	0	0	1

- Initially, the timer is off
- Executing the instruction enables the timer
- The time required to set the TMR2IF is

  Time = (PR2+1) * prescaler * postscaler * 4 /Fosc

  if TMR2 is initialized to zero.
- However, TMR2 = 44. So the time is

  Time = [(PR2-TMR2+1) * 4 * 4/8MHz] + [(PR2+1) * 4 * 4/8MHz] * 7 = 1528 usec

  (first loop has initial value = 44; the last 7 loops has initial value = 0; *1)

1+7 = 8 (post scalar).

Example 3

Write a program that configures and uses the CCP1 module in PIC16F873A to generate a periodic square wave of frequency 50 Hz and 25% duty cycle. Assume that Fosc = 800 KHz.

Requirements

1) Configure RC2 as output
2) Configure TIMER2 module and compute the values to be placed in CCPR1L and PR2 registers which determine the duty cycle and the cycle time, respectively
3) Turn on the timer

Example 3

TABLE 8-5: REGISTERS ASSOCIATED WITH PWM AND TIMER2

Address	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
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
0Dh	PIR2	—	—	—	—	—	—	—	CCP2IF	---- ---0	---- ---0
8Ch	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
8Dh	PIE2	—	—	—	—	—	—	—	CCP2IE	---- ---0	---- ---0
87h	TRISC	PORTC Data Direction Register								1111 1111	1111 1111
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
92h	PR2	Timer2 Module's Period Register								1111 1111	1111 1111
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	uuuu uuuu
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	uuuu uuuu
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	uuuu uuuu
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	uuuu uuuu
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	--00 0000

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PWM and Timer2.

Example 3

- PWM signal specs
  - T = 1 /50 = 0.02 sec
  - Ton = 0.25 * T = 0.005 sec
- Need to configure the CCP1 and TIMER2
  - PR2 register
    T = (PR2+1) * 4 * Tosc * prescaler
    if we assume prescaler = 16, then PR2 = 249
  - Pulse-width register CCPR1L:CCP1CON<5:4>
    Ton = PWR * Tosc * prescaler
    already the prescaler is chosen to be 16 → PWR = 250 =0xFA
    → CCPR1L = B'00111110' and CCP1CON<5:4> = B'10'
  - T2CON = 0x06
  - CCP1CON = B'00101100'

Example 3

```
        # include   "PIC16F877.INC"
        org       0x0000
        goto      START
        org       0x0004
ISR     goto      ISR
START   bcf       STATUS, RP1       ; select bank 1
        bsf       STATUS, RP0
        bcf       TRISC, 2          ; set RC2 as output
        movlw     D'249'
        movwf     PR2               ; set the cycle time in PR2
        bcf       STATUS, RP0
        movlw     0x3E
        movwf     CCPR1L            ; set the ON time in CCPR1L
        bcf       CCP1CON, 4        ; specify the LSBs of the ON time
        bsf       CCP1CON, 5
```

Example 3

```
        bsf     CCP1CON, 3
        bsf     CCP1CON, 2  ; configure  CCP1 in PWM and
        movlw   0x06
        movwf   T2CON       ; configure timer 2 and enable it


DONE    goto    DONE
        end
```


Summary

- Timing is essential element of embedded systems design
- Wide range of timers is available in PIC microcontrollers with clever add-on features such as capture, compare, and pulse width modulation
- It is very occasional to have several timers running simultaneously in an embedded system

Registers involved in asynchronous transmission

Address	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
0Bh, 8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	R0IF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D	0000 -00x	0000 -00x
19h	TXREG	USART Transmit Register								0000 0000	0000 0000
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
98h (Bank1)	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99h (Bank1)	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000

Registers involved in asynchronous reception

Address	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
0Bh, 8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	R0IF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
18h	RCSTA	SPEN	RX9	SREN	CREN	—	FERR	OERR	RX9D	0000 -00x	0000 -00x
1Ah	RCREG	USART Receive Register								0000 0000	0000 0000
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
98h (Bank1)	TXSTA	CSRC	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D	0000 -010	0000 -010
99h (Bank1)	SPBRG	Baud Rate Generator Register								0000 0000	0000 0000

TXSTA register (Bank1 RP0=1, PR1=0)

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R-1	R/W-0
	TX9	TXEN	SYNC	—	BRGH	TRMT	TX9D
bit 7							bit 0

bit 7 **CSRC:** Clock Source Select bit
Asynchronous mode:
Don't care.
Synchronous mode:
1 = Master mode (clock generated internally from BRG)
0 = Slave mode (clock from external source)

bit 6 **TX9**: 9-bit Transmit Enable bit
1 = Selects 9-bit transmission
0 = Selects 8-bit transmission

bit 5 **TXEN**: Transmit Enable bit
1 = Transmit enabled
0 = Transmit disabled
Note: SREN/CREN overrides TXEN in Sync mode.

bit 4 **SYNC**: USART Mode Select bit
1 = Synchronous mode
0 = Asynchronous mode

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

bit 2 **BRGH**: High Baud Rate Select bit
Asynchronous mode:
1 = High speed
0 = Low speed
Synchronous mode:
Unused in this mode.

bit 1 **TRMT**: Transmit Shift Register Status bit
1 = TSR empty
0 = TSR full

bit 0 **TX9D:** 9th bit of Transmit Data, can be Parity bit

RCSTA register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit
1 = Serial port enabled (configures RC7/RX/DT and RC6/TX/CK pins as serial port pins)
0 = Serial port disabled

bit 6 **RX9**: 9-bit Receive Enable bit
1 = Selects 9-bit reception
0 = Selects 8-bit reception

bit 5 **SREN**: Single Receive Enable bit
Asynchronous mode:
Don't care.
Synchronous mode – Master:
1 = Enables single receive
0 = Disables single receive
This bit is cleared after reception is complete.
Synchronous mode – Slave:
Don't care.

bit 4 **CREN**: Continuous Receive Enable bit
Asynchronous mode:
1 = Enables continuous receive
0 = Disables continuous receive
Synchronous mode:
1 = Enables continuous receive until enable bit CREN is cleared (CREN overrides SREN)
0 = Disables continuous receive

bit 3 **ADDEN:** Address Detect Enable bit
Asynchronous mode 9-bit (RX9 = 1):
1 = Enables address detection, enables interrupt and load of the receive buffer when RSR<8> is set
0 = Disables address detection, all bytes are received and ninth bit can be used as parity bit

bit 2 **FERR**: Framing Error bit
1 = Framing error (can be updated by reading RCREG register and receive next valid byte)
0 = No framing error

bit 1 **OERR**: Overrun Error bit
1 = Overrun error (can be cleared by clearing bit CREN)
0 = No overrun error

bit 0 **RX9D:** 9th bit of Received Data (can be parity bit but must be calculated by user firmware)

Baud rate generator:

SYNC	BRGH = 0	BRGH = 1
0 (asynchronous)	$\frac{F_{osc}}{64(SPBRG+1)}$	$\frac{F_{osc}}{16(SPBRG+1)}$
1 (synchronous)	$\frac{F_{osc}}{4(SPBRG+1)}$	

TABLE 10-3: BAUD RATES FOR ASYNCHRONOUS MODE (BRGH = 0)

BAUD RATE (K)	Fosc = 20 MHz			Fosc = 16 MHz			Fosc = 10 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	-	-	-	-	-	-	-	-	-
1.2	1.221	1.75	255	1.202	0.17	207	1.202	0.17	129
2.4	2.404	0.17	129	2.404	0.17	103	2.404	0.17	64
9.6	9.766	1.73	31	9.615	0.16	25	9.766	1.73	15
19.2	19.531	1.72	15	19.231	0.16	12	19.531	1.72	7
28.8	31.250	8.51	9	27.778	3.55	8	31.250	8.51	4
33.6	34.722	3.34	8	35.714	6.29	6	31.250	6.99	4
57.6	62.500	8.51	4	62.500	8.51	3	52.083	9.58	2
HIGH	1.221	-	255	0.977	-	255	0.610	-	255
LOW	312.500	-	0	250.000	-	0	156.250	-	0

BAUD RATE (K)	Fosc = 4 MHz			Fosc = 3.6864 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	0.300	0	207	0.3	0	191
1.2	1.202	0.17	51	1.2	0	47
2.4	2.404	0.17	25	2.4	0	23
9.6	8.929	6.99	6	9.6	0	5
19.2	20.833	8.51	2	19.2	0	2
28.8	31.250	8.51	1	28.8	0	1
33.6	-	-	-	-	-	-
57.6	62.500	8.51	0	57.6	0	0
HIGH	0.244	-	255	0.225	-	255
LOW	62.500	-	0	57.6	-	0

TABLE 10-4: BAUD RATES FOR ASYNCHRONOUS MODE (BRGH = 1)

BAUD RATE (K)	Fosc = 20 MHz			Fosc = 16 MHz			Fosc = 10 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	-	-	-	-	-	-	-	-	-
1.2	-	-	-	-	-	-	-	-	-
2.4	-	-	-	-	-	-	2.441	1.71	255
9.6	9.615	0.16	129	9.615	0.16	103	9.615	0.16	64
19.2	19.231	0.16	64	19.231	0.16	51	19.531	1.72	31
28.8	29.070	0.94	42	29.412	2.13	33	28.409	1.36	21
33.6	33.784	0.55	36	33.333	0.79	29	32.895	2.10	18
57.6	59.524	3.34	20	58.824	2.13	16	56.818	1.36	10
HIGH	4.883	-	255	3.906	-	255	2.441	-	255
LOW	1250.000	-	0	1000.000		0	625.000	-	0

BAUD RATE (K)	Fosc = 4 MHz			Fosc = 3.6864 MHz		
	KBAUD	% ERROR	SPBRG value (decimal)	KBAUD	% ERROR	SPBRG value (decimal)
0.3	-	-	-	-	-	-
1.2	1.202	0.17	207	1.2	0	191
2.4	2.404	0.17	103	2.4	0	95
9.6	9.615	0.16	25	9.6	0	23
19.2	19.231	0.16	12	19.2	0	11
28.8	27.798	3.55	8	28.8	0	7
33.6	35.714	6.29	6	32.9	2.04	6
57.6	62.500	8.51	3	57.6	0	3
HIGH	0.977	-	255	0.9	-	255
LOW	250.000	-	0	230.4	-	0

TABLE 11-2: REGISTERS/BITS ASSOCIATED WITH A/D

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR, BOR	Value on $\overline{MCLR}$, WDT
0Bh,8Bh, 10Bh,18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
1Eh	ADRESH	A/D Result Register High Byte								xxxx xxxx	uuuu uuuu
9Eh (Bank1)	ADRESL	A/D Result Register Low Byte								xxxx xxxx	uuuu uuuu
1Fh	ADCON0	ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{DONE}$	—	ADON	0000 00-0	0000 00-0
9Fh (Bank1)	ADCON1	ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0	00-- 0000	00-- 0000
85h (Bank1)	TRISA	—	—	PORTA Data Direction Register						--11 1111	--11 1111
05h	PORTA	—	—	PORTA Data Latch when written: PORTA pins when read						--0x 0000	--0u 0000
89h(1) (Bank1)	TRISE	IBF	OBF	IBOV	PSPMODE	—	PORTE Data Direction bits			0000 -111	0000 -111
09h(1)	PORTE	—	—	—	—	—	RE2	RE1	RE0	---- -xxx	---- -uuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used for A/D conversion.

Note 1: These registers are not available on 28-pin devices.

ADCON0 Register 0x1F

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/$\overline{DONE}$	—	ADON
bit 7							bit 0

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

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

bit 5-3 **CHS2:CHS0:** Analog Channel Select bits
- 000 = Channel 0 (AN0)
- 001 = Channel 1 (AN1)
- 010 = Channel 2 (AN2)
- 011 = Channel 3 (AN3)
- 100 = Channel 4 (AN4)
- 101 = Channel 5 (AN5)
- 110 = Channel 6 (AN6)
- 111 = Channel 7 (AN7)

bit 2 **GO/$\overline{DONE}$:** 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

ADCON1 Register 0x9F

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM	ADCS2	—	—	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

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

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

bit 6 **ADCS2:** A/D Conversion Clock Select bit (ADCON1 bits in shaded area and in **bold**)

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

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

PCFG <3:0>	AN7	AN6	AN5	AN4	AN3	AN2	AN1	AN0	VREF+	VREF-	C/R
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	VREF+	A	A	A	AN3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	VREF+	A	A	A	AN3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	VREF+	D	A	A	AN3	VSS	2/1
011x	D	D	D	D	D	D	D	D	—	—	0/0
1000	A	A	A	A	VREF+	VREF-	A	A	AN3	AN2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	VREF+	A	A	A	AN3	VSS	5/1
1011	D	D	A	A	VREF+	VREF-	A	A	AN3	AN2	4/2
1100	D	D	D	A	VREF+	VREF-	A	A	AN3	AN2	3/2
1101	D	D	D	D	VREF+	VREF-	A	A	AN3	AN2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	VREF+	VREF-	D	A	AN3	AN2	1/2

A = Analog input D = Digital I/O
C/R = # of analog input channels/# of A/D voltage references

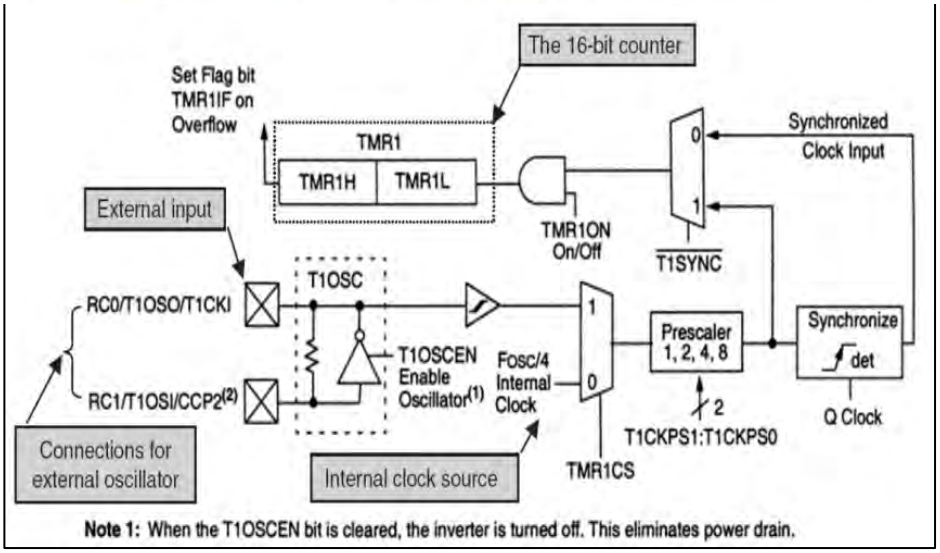
TABLE 6-2: REGISTERS ASSOCIATED WITH TIMER1 AS A TIMER/COUNTER

Address	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
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	--00 0000	--uu uuuu

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer1 module.

Note 1: Bits PSPIE and PSPIF are reserved on the 28-pin devices; always maintain these bits clear.

TMR1 MODULE:



T1CON Register (0x10)

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON
bit 7							bit 0

bit 7:6 **Unimplemented:** Read as '0'

bit 5:4 **T1CKPS1:T1CKPS0**: Timer1 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**: Timer1 Oscillator Enable bit

1 = Oscillator is enabled
0 = Oscillator is shut off. The oscillator inverter and feedback resistor are turned off to eliminate power drain

bit 2 **T1SYNC**: Timer1 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. Timer1 uses the internal clock when TMR1CS = 0.

bit 1 **TMR1CS**: Timer1 Clock Source Select bit

1 = External clock from pin T1OSO/T1CKI (on the rising edge)
0 = Internal clock (Fosc/4)

bit 0 **TMR1ON**: Timer1 On bit

1 = Enables Timer1
0 = Stops Timer1

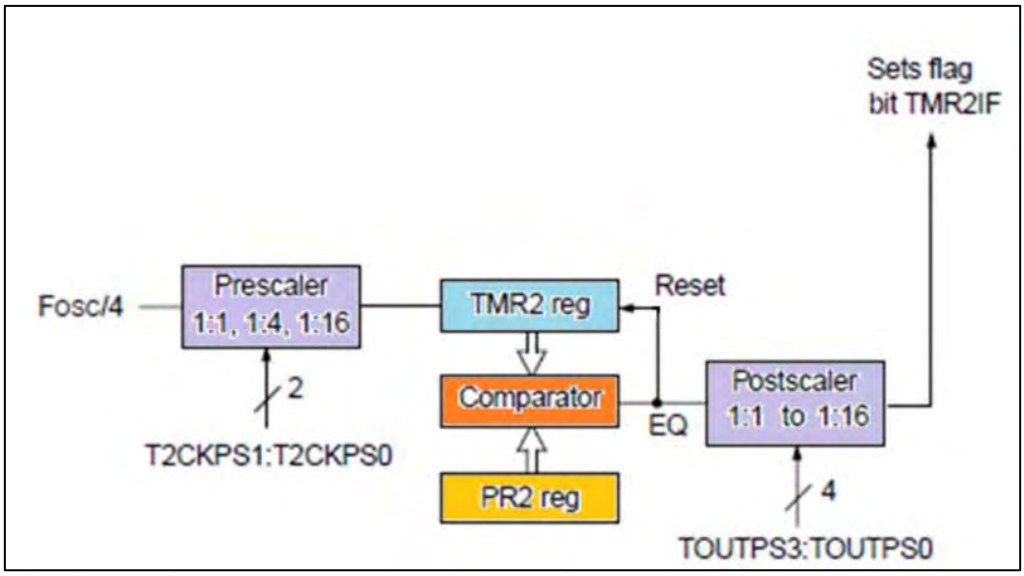
TABLE 7-1: REGISTERS ASSOCIATED WITH TIMER2 AS A TIMER/COUNTER

Address	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
0Bh, 8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
92h (Bank1)	PR2	Timer2 Period Register								1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by the Timer2 module.

Note 1: Bits PSPIE and PSPIF are reserved on 28-pin devices; always maintain these bits clear.

TMR2 MODULE:



T2CON Register (0x12)

U-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

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

bit 6:3 **TOUTPS3:TOUTPS0**: Timer2 Output Postscale Select bits

0000 = 1:1 Postscale
0001 = 1:2 Postscale
•
•
•
1111 = 1:16 Postscale

bit 2 **TMR2ON**: Timer2 On bit

1 = Timer2 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

TABLE 8-4: REGISTERS ASSOCIATED WITH CAPTURE, COMPARE AND TIMER1

Address	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
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
0Dh	PIR2	—	—	—	—	—	—	—	CCP2IF	---- ---0	---- ---0
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
8Dh (Bank1)	PIE2	—	—	—	—	—	—	—	CCP2IE	---- ---0	---- ---0
87h (Bank1)	TRISC	PORTC Data Direction Register								1111 1111	1111 1111
0Eh	TMR1L	Holding Register for the Least Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
0Fh	TMR1H	Holding Register for the Most Significant Byte of the 16-bit TMR1 Register								xxxx xxxx	uuuu uuuu
10h	T1CON	—	—	T1CKPS1	T1CKPS0	T1OSCEN	$\overline{\text{T1SYNC}}$	TMR1CS	TMR1ON	--00 0000	--uu uuuu
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	uuuu uuuu
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	uuuu uuuu
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	uuuu uuuu
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	uuuu uuuu
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	--00 0000

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by Capture and Timer1.

Note 1: The PSP is not implemented on 28-pin devices; always maintain these bits clear.

TABLE 8-5: REGISTERS ASSOCIATED WITH PWM AND TIMER2

Address	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
0Bh,8Bh, 10Bh, 18Bh	INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF	0000 000x	0000 000u
0Ch	PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	0000 0000	0000 0000
0Dh	PIR2	—	—	—	—	—	—	—	CCP2IF	---- ---0	---- ---0
8Ch (Bank1)	PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE	0000 0000	0000 0000
8Dh (Bank1)	PIE2	—	—	—	—	—	—	—	CCP2IE	---- ---0	---- ---0
87h (Bank1)	TRISC	PORTC Data Direction Register								1111 1111	1111 1111
11h	TMR2	Timer2 Module's Register								0000 0000	0000 0000
92h (Bank1)	PR2	Timer2 Module's Period Register								1111 1111	1111 1111
12h	T2CON	—	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0	-000 0000	-000 0000
15h	CCPR1L	Capture/Compare/PWM Register 1 (LSB)								xxxx xxxx	uuuu uuuu
16h	CCPR1H	Capture/Compare/PWM Register 1 (MSB)								xxxx xxxx	uuuu uuuu
17h	CCP1CON	—	—	CCP1X	CCP1Y	CCP1M3	CCP1M2	CCP1M1	CCP1M0	--00 0000	--00 0000
1Bh	CCPR2L	Capture/Compare/PWM Register 2 (LSB)								xxxx xxxx	uuuu uuuu
1Ch	CCPR2H	Capture/Compare/PWM Register 2 (MSB)								xxxx xxxx	uuuu uuuu
1Dh	CCP2CON	—	—	CCP2X	CCP2Y	CCP2M3	CCP2M2	CCP2M1	CCP2M0	--00 0000	--00 0000

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0'. Shaded cells are not used by PWM and Timer2.

Note 1: Bits PSPIE and PSPIF are reserved on 28-pin devices; always maintain these bits clear.

CCP Control Registers: CCP1CON and CCP2CON

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	DCxB1	DCxB0	CCPxM3	CCPxM2	CCPxM1	CCPxM0
bit 7							bit 0

bit 7:6 **Unimplemented:** Read as '0'

bit 5:4 **DCxB1:DCxB0**: PWM Duty Cycle bit1 and bit0

Capture Mode:

Unused

Compare Mode:

Unused

PWM Mode:

These bits are the two LSbs (bit1 and bit0) 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**: CCPx Mode Select bits

0000 = Capture/Compare/PWM off (resets CCPx module)
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,
Generate software interrupt on compare match
(CCPIF bit is set, CCP pin is unaffected)
1011 = Compare mode,
Trigger special event (CCPIF bit is set)
11xx = PWM mode

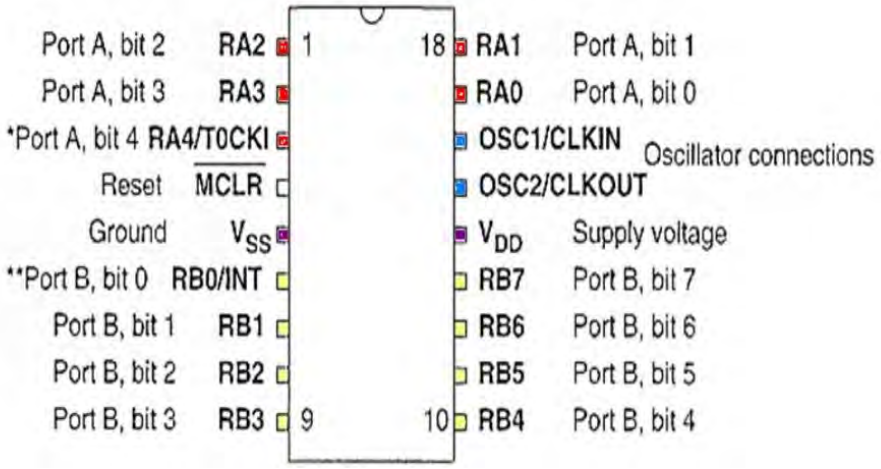


TABLE 6-4: RESET CONDITIONS FOR ALL REGISTERS

Register	Address	Power-on Reset	$\overline{MCLR}$ during: – normal operation – SLEEP WDT Reset during normal operation	Wake-up from SLEEP: – through interrupt – through WDT Time-out
W	—	xxxx xxxx	uuuu uuuu	uuuu uuuu
INDF	00h	---- ----	---- ----	---- ----
TMR0	01h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PCL	02h	0000 0000	0000 0000	PC + 1(2)
STATUS	03h	0001 1xxx	000q quuu(3)	uuuq quuu(3)
FSR	04h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PORTA(4)	05h	---x xxxx	---u uuuu	---u uuuu
PORTB(5)	06h	xxxx xxxx	uuuu uuuu	uuuu uuuu
EEDATA	08h	xxxx xxxx	uuuu uuuu	uuuu uuuu
EEADR	09h	xxxx xxxx	uuuu uuuu	uuuu uuuu
PCLATH	0Ah	---0 0000	---0 0000	---u uuuu
INTCON	0Bh	0000 000x	0000 000u	uuuu uuuu(1)
INDF	80h	---- ----	---- ----	---- ----
OPTION_REG	81h	1111 1111	1111 1111	uuuu uuuu
PCL	82h	0000 0000	0000 0000	PC + 1(2)
STATUS	83h	0001 1xxx	000q quuu(3)	uuuq quuu(3)
FSR	84h	xxxx xxxx	uuuu uuuu	uuuu uuuu
TRISA	85h	---1 1111	---1 1111	---u uuuu
TRISB	86h	1111 1111	1111 1111	uuuu uuuu
EECON1	88h	---0 x000	---0 q000	---0 uuuu
EECON2	89h	---- ----	---- ----	---- ----
PCLATH	8Ah	---0 0000	---0 0000	---u uuuu
INTCON	8Bh	0000 000x	0000 000u	uuuu uuuu(1)

Legend: u = unchanged, x = unknown, - = unimplemented bit, read as '0', q = value depends on condition

Note 1: One or more bits in INTCON will be affected (to cause wake-up).

2: When the wake-up is due to an interrupt and the GIE bit is set, the PC is loaded with the interrupt vector (0004h).

3: Table 6-3 lists the RESET value for each specific condition.

4: On any device RESET, these pins are configured as inputs.

5: This is the value that will be in the port output latch.

TABLE 2-1: SPECIAL FUNCTION REGISTER FILE SUMMARY

Addr	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on RESET	Details on page
Bank 0											
00h	INDF	Uses contents of FSR to address Data Memory (not a physical register)								---- ----	11
01h	TMR0	8-bit Real-Time Clock/Counter								xxxx xxxx	20
02h	PCL	Low Order 8 bits of the Program Counter (PC)								0000 0000	11
03h	STATUS(2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	8
04h	FSR	Indirect Data Memory Address Pointer 0								xxxx xxxx	11
05h	PORTA(4)	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	16
06h	PORTB(5)	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	18
07h	—	Unimplemented location, read as '0'								—	—
08h	EEDATA	EEPROM Data Register								xxxx xxxx	13,14
09h	EEADR	EEPROM Address Register								xxxx xxxx	13,14
0Ah	PCLATH	—	—	—	Write Buffer for upper 5 bits of the PC(1)					---0 0000	11
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	10
Bank 1											
80h	INDF	Uses Contents of FSR to address Data Memory (not a physical register)								---- ----	11
81h	OPTION_REG	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	9
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	11
83h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	8
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	11
85h	TRISA	—	—	—	PORTA Data Direction Register					---1 1111	16
86h	TRISB	PORTB Data Direction Register								1111 1111	18
87h	—	Unimplemented location, read as '0'								—	—
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	13
89h	EECON2	EEPROM Control Register 2 (not a physical register)								---- ----	14
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC(1)					---0 0000	11
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	10

Legend: x = unknown, u = unchanged, - = unimplemented, read as '0', q = value depends on condition

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> are never transferred to PCLATH.

2: The $\overline{TO}$ and $\overline{PD}$ status bits in the STATUS register are not affected by a $\overline{MCLR}$ Reset.

3: Other (non power-up) RESETS include: external RESET through $\overline{MCLR}$ and the Watchdog Timer Reset.

4: On any device RESET, these pins are configured as inputs.

5: This is the value that will be in the port output latch.

Configuration Word

R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u	R/P-u
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	$\overline{\text{PWRTE}}$	WDTE	F0SC1	F0SC0
bit13													bit0

bit 13-4 **CP:** Code Protection bit
1 = Code protection disabled
0 = All program memory is code protected

bit 3 **$\overline{\text{PWRTE}}$**: Power-up Timer Enable bit
1 = Power-up Timer is disabled
0 = Power-up Timer is enabled

bit 2 **WDTE**: Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0 **FOSC1:FOSC0**: Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

Status register

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{\text{TO}}$	$\overline{\text{PD}}$	Z	DC	C
bit 7							bit 0

bit 7-6 **Unimplemented:** Maintain as '0'

bit 5 **RP0**: Register Bank Select bits (used for direct addressing)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)

bit 4 **$\overline{\text{TO}}$**: Time-out bit
1 = After power-up, CLRWDT instruction, or SLEEP instruction
0 = A WDT time-out occurred

bit 3 **$\overline{\text{PD}}$**: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

bit 2 **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1 **DC**: Digit Carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{\text{borrow}}$, the polarity is reversed)
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0 **C**: Carry/$\overline{\text{borrow}}$ bit (ADDWF, ADDLW, SUBLW, SUBWF instructions) (for $\overline{\text{borrow}}$, the polarity is reversed)
1 = A carry-out from the Most Significant Bit of the result occurred
0 = No carry-out from the Most Significant Bit of the result occurred

Note: A subtraction is executed by adding the twos complement of the second operand. For rotate (RRF, RLF) instructions, this bit is loaded with either the high or low order bit of the source register.

EECON1 – address 88H (BANK 1)

U-0	U-0	U-0	R/W-0	R/W-x	R/W-0	R/S-0	R/S-0
—	—	—	EEIF	WRERR	WREN	WR	RD
bit 7							bit 0

bit 7-5 **Unimplemented:** Read as '0'

bit 4 **EEIF:** EEPROM Write Operation Interrupt Flag bit
1 = The write operation completed (must be cleared in software)
0 = The write operation is not complete or has not been started

bit 3 **WRERR:** EEPROM Error Flag bit
1 = A write operation is prematurely terminated (any $\overline{\text{MCLR}}$ Reset or any WDT Reset during normal operation)
0 = The write operation completed

bit 2 **WREN:** EEPROM Write Enable bit
1 = Allows write cycles
0 = Inhibits write to the EEPROM

bit 1 **WR:** Write Control bit
1 = Initiates a write cycle. The bit is cleared by hardware once write is complete. The WR bit can only be set (not cleared) in software.
0 = Write cycle to the EEPROM is complete

bit 0 **RD:** Read Control bit
1 = Initiates an EEPROM read RD is cleared in hardware. The RD bit can only be set (not cleared) in software.
0 = Does not initiate an EEPROM read

OPTION_REGISTER – address 81H (BANK 1)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$:** PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5 **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4 **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3 **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0 **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

INTCON register

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

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

bit 6 **EEIE**: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state