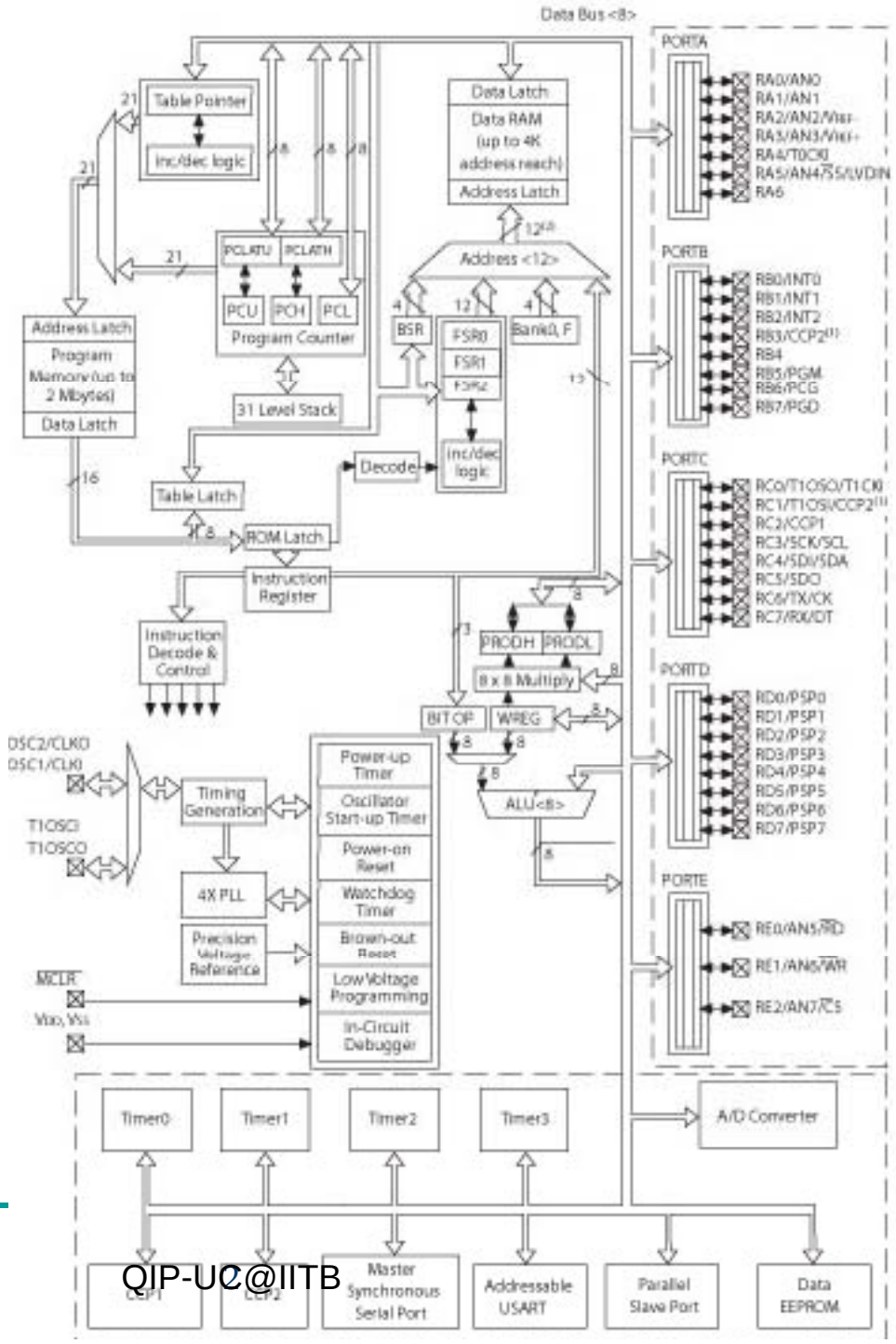
Interrupt & Serial Communication in PIC

Virendra Singh
Computer Architecture and Dependable Systems Lab
Department of Electrical Engineering
Indian Institute of Technology Bombay
http://www.ee.iitb.ac.in/~viren/
E-mail: viren@ee.iitb.ac.in

PIC18F4X2 Architecture Block Diagram



Interrupt

- Interrupts can be generated by various internal or external hardware events.
- Interrupts are internally or externally generated asynchronous hardware signals that force the processor to stop its current program and carry out the function called.
- The function called by the interrupt is often referred to as an interrupt service routine (ISR).
- The processor's current register contents and status must be saved and the current program address stored on the stack so that the background task can be resumed when the ISR has finished.

Interrupt – Priority

- If the program uses multiple interrupts, one ISR may be interrupted by another.
- The interrupts may need to be assigned an order of priority, so that a less important task does not interrupt a more important one.
- When the higher-priority ISR is being executed, the lower-priority interrupt can be disabled, or masked, until it is finished.
- In more complex programs, numerical levels of priority can be assigned, with higher priorities taking precedence. .
- Further, the different interrupt sources have to be identified explicitly (fully and clearly) by a user routine.

Interrupt

- A single microcontroller can serve several devices.
- There are two (2) methods by which devices receive service from the microcontroller:
  - Polling
  - Interrupts

Interrupt vs. Polling

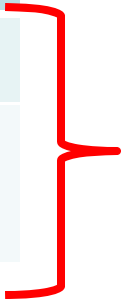
Interrupt Method

- Whenever any device needs the microcontroller's service, the device notifies it by sending an interrupt signal.
- Interrupt is the signal sent to the microprocessor to mark the event that requires immediate attention.
- Upon receiving an interrupt signal, the microcontroller stops current program and serve the device (execute ISR).
- The program associated with the interrupt is called ISR (interrupt service routine) or interrupt handler.
- Each device can get the attention of the microcontroller based on the priority assign to it.
- Can ignore a device request for service

Interrupt Service Routine (ISR)

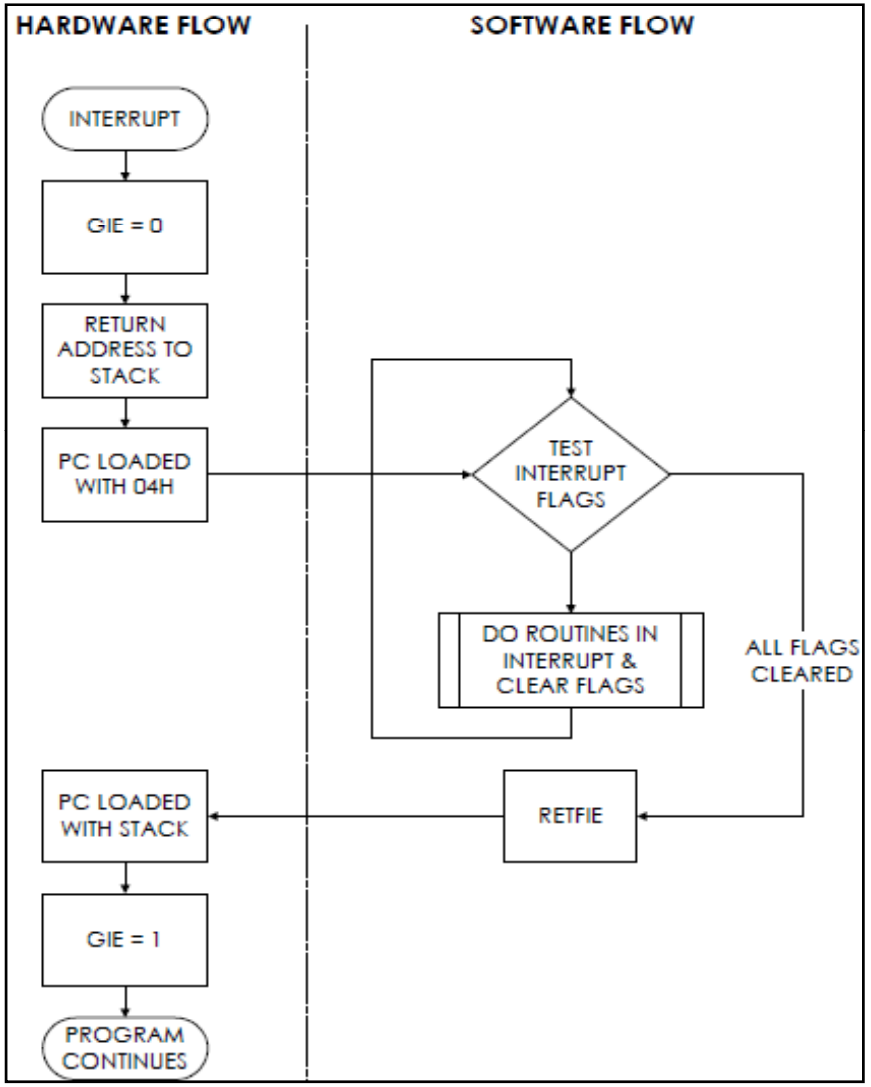
- For every interrupt, there must be an interrupt service routine (ISR) or interrupt handler.
- When an interrupt is invoked, the uC runs the ISR.
- Generally, in most microprocessors, for every interrupt there is a fixed location in memory that holds the address of its ISR.
- The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table (IVT).

Interrupt	ROM Location (Hex)
Power-on-Reset	0000h
Interrupt Vector	0008h (Default upon power-on reset)



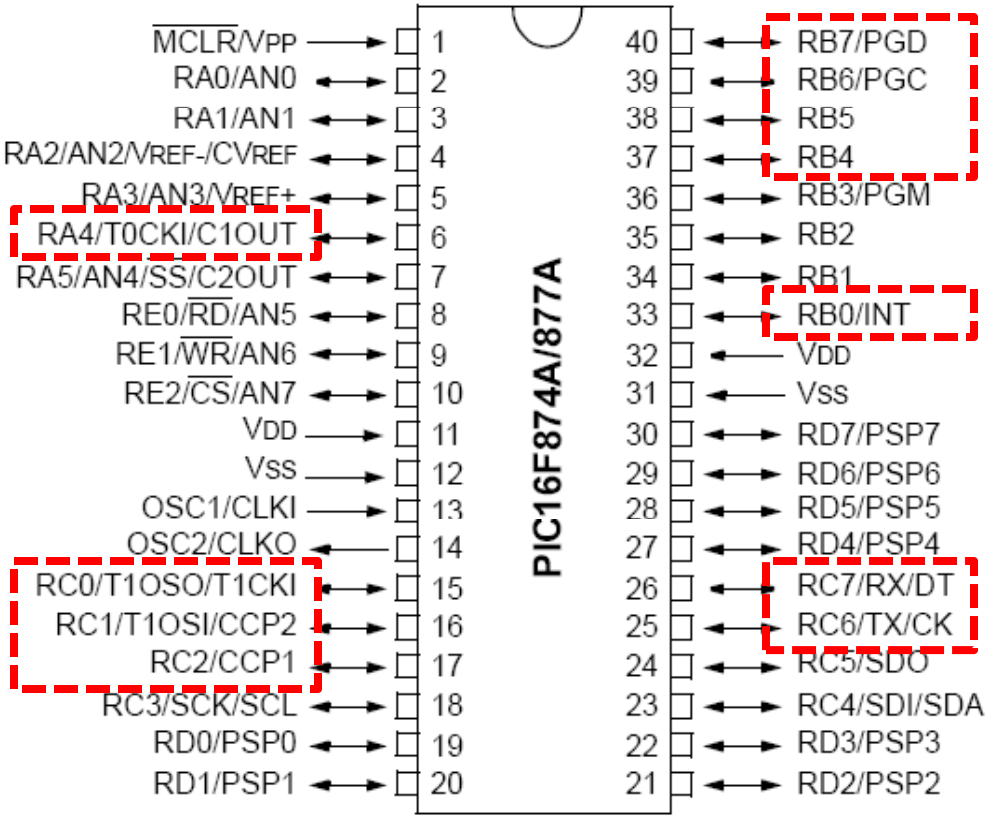
Steps in Executing Interrupt

1. Upon receives interrupt signal, uC finishes the instruction it is executing and saves the address of the next instruction (program counter) on the stack
2. It jumps to a fixed location in memory (interrupt vector table (IVT)). The IVT directs the microcontroller to the address of the ISR.
3. The microcontroller gets the address of the ISR from the IVT and jumps to it. It start to execute the interrupt service subroutine until it reaches the last instruction of the subroutine - RETFIE (Return from Interrupt Exit)
4. Upon executing the RETFIE instruction, the microcontroller returns to the place where it was interrupted



Sources of Interrupts

- Example of sources of interrupts:
  - Each Timers (Timer 0, 1, 2)
  - 1 interrupt for external hardware: Pin RB0 (INT)
  - The PORTB-Change interrupt: Pin RB7 - RB4
  - 2 interrupts for serial communication USART (Receive and Transmit) : Pin RC6 (TX) & RC7 (RX)
  - The ADC (Analog-to-Digital Converter)
  - The CCP (Compare Capture PWM)
  - And many more…



Enabling and Disabling Interrupt

- **INTCON** register records individual interrupt requests in flag bits. Also has individual and global interrupt enable bits.
- In general each interrupt source have following related bits.
  - **Enable Bit**
    - They are suffixed (attached) with xx**IE** (Interrupt Enable).
    - It can be used to enable/disable the related interrupt.
    - When set to '1' it enables the interrupt.
  - **Flag Bit**
    - The are suffixed with xx**IF** (Interrupt Flag).
    - It is set automatically by the related hardware when the interrupt condition occurs.
    - When it is set to '1' we know that interrupt has occurred.

Enabling and Disabling Interrupt

- Upon reset, all interrupts are disabled (masked). None of interrupt will be responded to uC if they are activated.
- Interrupts must be enabled (unmasked) by software in order for uC to respond
- The D7 bit of INTCON (Interrupt Control) register is responsible for enabling and disabling interrupts globally - GIE (Global Interrupt Enable
  - **GIE = INTCON<7>**

How to set the required registers to work with interrupts?

- The settings of the interrupts are done by using 3 interrupt control registers:
  - **INTCON Register**
    - contains peripheral interrupt enable (PEIE) bit
  - **PIE (PIE1, PIE2) Register**
    - contains corresponding peripheral interrupt enable bits
  - **PIR (PIR1, PIR2) Register**
    - contains peripheral interrupt flag bits

INTCON Register

- A readable and writable register which contains various enable and flag bits for TMR0 register overflow, RB port change and external RB0/INT pin interrupts.

Global Interrupt Enable: GIE (INTCON<7>)

- GIE = 1 : enabling all interrupts (unmask interrupt)
  - Interrupts are allow to happen. Each interrupt source is enabled by setting to HIGH the corresponding interrupt enable (IE) bit.
- GIE = 0 : disabling all interrupts globally (mask interrupt)
  - No interrupt is acknowledged, even if the corresponding interrupt enable (IE) bit is HIGH.
- Individual interrupts can be disabled through their corresponding enable bits in various registers.
- Individual interrupt flag bits are set regardless of status of their corresponding mask bit or GIE bit.

INTCON Register

- GIE bit is cleared on Reset.
- Upon activation of interrupt, GIE bit is cleared (GIE = 0) to make sure another interrupt cannot interrupt the uC while it is servicing the current one.
- At the end of ISR, GIE is set HIGH (GIE = 1) to allow another interrupt to come in.

PEIE (INTCON<6>)

- This bit, along with GIE, must be set HIGH (PEIE = 1) to enable any peripheral interrupt such as Timers and serial port.

TMR0IE (INTCON<5>) & INTE (INTCON<4>)

- These bits, along with GIE, must be set HIGH for an interrupt to be responded to

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	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF
bit 7							bit 0

INTCON register

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

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

bit 5 **TMR0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 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 **TMR0IF**: 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; a mismatch condition will continue to set the bit. Reading PORTB will end the mismatch condition and allow the bit to be cleared (must be cleared in software).
0 = None of the RB7:RB4 pins have changed state

PIE Register

- **PIE1**: Contains the individual enable bits of the peripheral interrupts.
- **PIE2**: Contains the individual enable bits for the CCP2 peripheral interrupt, SSP bus collision interrupt, EEPROM write operation interrupt and comparator interrupt.

Note:

- Along with GIE, bit PEIE (INTCON<6>) must be SET (PEIE = 1) to enable any peripheral interrupt

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

bit 7 **PSPIE:** Parallel Slave Port Read/Write Interrupt Enable bit(1)
1 = Enables the PSP read/write interrupt
0 = Disables the PSP read/write interrupt

Note 1: PSPIE is reserved on PIC16F873A/876A devices; always maintain this bit clear.

bit 6 **ADIE**: A/D Converter Interrupt Enable bit
1 = Enables the A/D converter interrupt
0 = Disables the A/D converter interrupt

bit 5 **RCIE**: USART Receive Interrupt Enable bit
1 = Enables the USART receive interrupt
0 = Disables the USART receive interrupt

bit 4 **TXIE**: USART Transmit Interrupt Enable bit
1 = Enables the USART transmit interrupt
0 = Disables the USART transmit interrupt

bit 3 **SSPIE**: Synchronous Serial Port Interrupt Enable bit
1 = Enables the SSP interrupt
0 = Disables the SSP interrupt

bit 2 **CCP1IE**: CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1 **TMR2IE**: TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

bit 0 **TMR1IE**: TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

PIE1 register

PIE2 register

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIE	—	EEIE	BCLIE	—	—	CCP2IE
bit 7							bit 0

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

bit 6 **CMIE**: Comparator Interrupt Enable bit
1 = Enables the comparator interrupt
0 = Disable the comparator interrupt

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

bit 4 **EEIE**: EEPROM Write Operation Interrupt Enable bit
1 = Enable EEPROM write interrupt
0 = Disable EEPROM write interrupt

bit 3 **BCLIE**: Bus Collision Interrupt Enable bit
1 = Enable bus collision interrupt
0 = Disable bus collision interrupt

bit 2-1 **Unimplemented:** Read as '0'

bit 0 **CCP2IE**: CCP2 Interrupt Enable bit
1 = Enables the CCP2 interrupt
0 = Disables the CCP2 interrupt

PIR Register

- **PIR1**: Contains the individual flag bits of the peripheral interrupts.
- **PIR2**: Contains the individual flag bits for the CCP2 peripheral interrupt, SSP bus collision interrupt, EEPROM write operation interrupt and comparator interrupt.

Note:

- Interrupt flag bits are SET when an interrupt condition occurs, regardless of state of its corresponding enable bit or GIE (INTCON<7>).
- User software should ensure the appropriate interrupt flag bits are CLEAR prior to enabling an interrupt.

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

bit 7 **PSPIF**: Parallel Slave Port Read/Write Interrupt Flag bit(1)
1 = A read or a write operation has taken place (must be cleared in software)
0 = No read or write has occurred
Note 1: PSPIF is reserved on PIC16F873A/876A devices; always maintain this bit clear.

bit 6 **ADIF**: A/D Converter Interrupt Flag bit
1 = An A/D conversion completed
0 = The A/D conversion is not complete

bit 5 **RCIF**: USART Receive Interrupt Flag bit
1 = The USART receive buffer is full
0 = The USART receive buffer is empty

bit 4 **TXIF**: USART Transmit Interrupt Flag bit
1 = The USART transmit buffer is empty
0 = The USART transmit buffer is full

bit 3 **SSPIF**: Synchronous Serial Port (SSP) Interrupt Flag bit
1 = The SSP interrupt condition has occurred and must be cleared in software before returning from the Interrupt Service Routine. The conditions that will set this bit are:
- SPI – A transmission/reception has taken place.
- I^2C Slave – A transmission/reception has taken place.
- I^2C Master
  - A transmission/reception has taken place.
  - The initiated Start condition was completed by the SSP module.
  - The initiated Stop condition was completed by the SSP module.
  - The initiated Restart condition was completed by the SSP module.
  - The initiated Acknowledge condition was completed by the SSP module.
  - A Start condition occurred while the SSP module was Idle (multi-master system).
  - A Stop condition occurred while the SSP module was Idle (multi-master system).

0 = No SSP interrupt condition has occurred

bit 2 **CCP1IF**: CCP1 Interrupt Flag bit
Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred
Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred
PWM mode:
Unused in this mode.

bit 1 **TMR2IF**: TMR2 to PR2 Match Interrupt Flag bit
1 = TMR2 to PR2 match occurred (must be cleared in software)
0 = No TMR2 to PR2 match occurred

bit 0 **TMR1IF**: TMR1 Overflow Interrupt Flag bit
1 = TMR1 register overflowed (must be cleared in software)
0 = TMR1 register did not overflow

PIR1 register

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
—	CMIF	—	EEIF	BCLIF	—	—	CCP2IF
bit 7							bit 0

PIR2 register

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

bit 6 **CMIF**: Comparator Interrupt Flag bit
1 = The comparator input has changed (must be cleared in software)
0 = The comparator input has not changed

bit 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 **BCLIF**: Bus Collision Interrupt Flag bit
1 = A bus collision has occurred in the SSP when configured for I^2C Master mode
0 = No bus collision has occurred

bit 2-1 **Unimplemented:** Read as '0'

bit 0 **CCP2IF**: CCP2 Interrupt Flag bit

Capture mode:
1 = A TMR1 register capture occurred (must be cleared in software)
0 = No TMR1 register capture occurred

Compare mode:
1 = A TMR1 register compare match occurred (must be cleared in software)
0 = No TMR1 register compare match occurred

PWM mode:
Unused

Programming Timer Interrupt

Polling Method

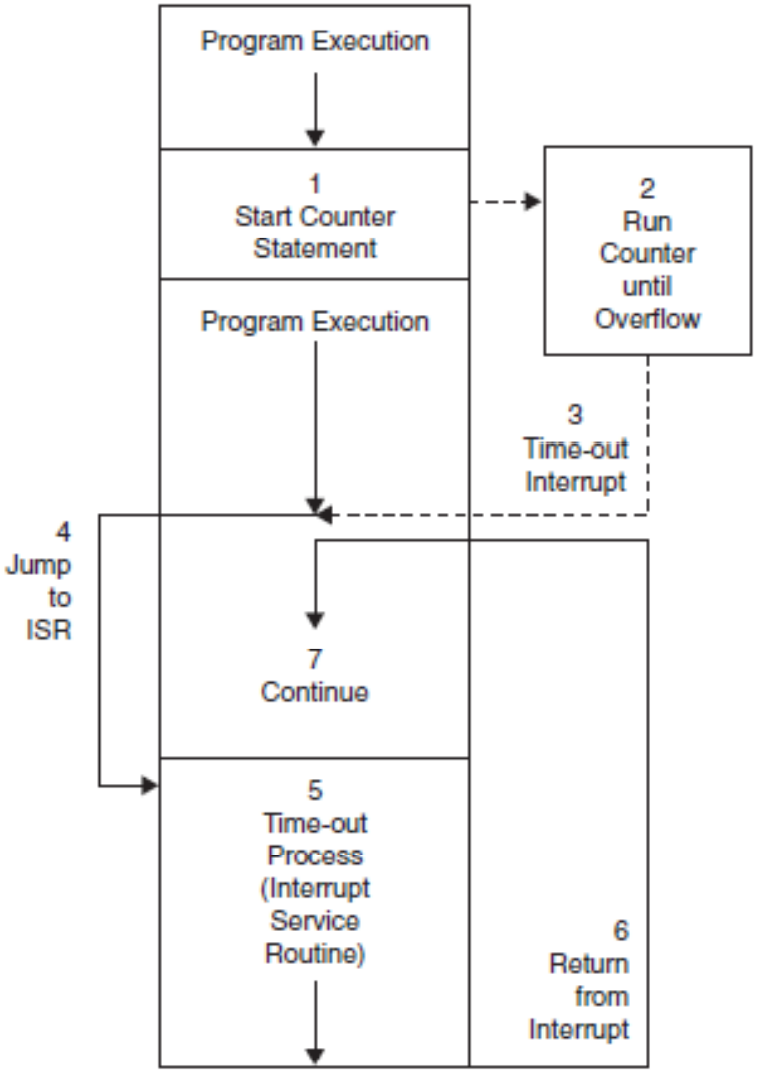
- In polling, we have to wait until the timer flag, TMR0IF (INTCON<2>) or TMR1IF (PIR1<0>) is raised when timer rolls over.
- Drawback: uC is tied down waiting for timer flag to be raised, and cannot do anything else.

Interrupt Method

- The timer interrupt in the interrupt register is enabled, TMR0IF or TMR1IF is raised whenever the timer rolls over, and uC jumps to the interrupt vector table to service ISR.
- Advantage: uC can do other things until it is notified that the timer has rolled over.

Programming Timer Interrupt

- An interrupt routine (ISR) has been written and assigned to the timer interrupt.
- The timer is set up during program initialization and started by preloading or clearing it.
- The main program and timer count then proceed concurrently, until a time-out occurs and the interrupt is generated.
- The main program is suspended and the ISR executed.
- When finished, the main program is resumed at the original point.
- If the ISR contains a statement to toggle an output bit, a square wave could be obtained with a period of twice the timer delay.

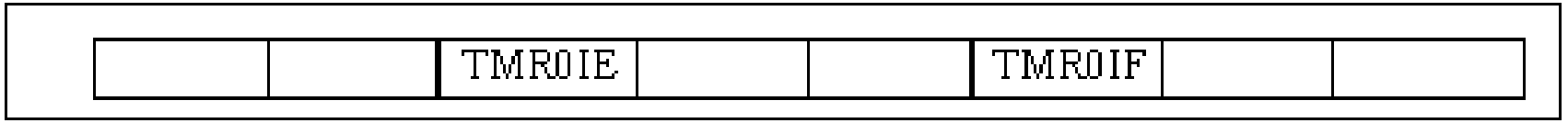


Programming Timer Interrupt

- To use an interrupt, we must **enable the interrupt** because all interrupts are masked upon power-on reset.
- **TMRxIE** bit: enables interrupt for Timer x

Interrupt	Flag Bit	Register	Enable Bit	Register
Timer0	TMR0IF	INTCON	TMR0IE	INTCON
Timer1	TMR1IF	PIR1	TMR1IE	PIE1
Timer2	TMR2IF	PIR1	TMR2IE	PIE1

Timer Interrupt Flag Bits and Associated Registers



INTCON Register with Timer0 Interrupt Enable and Interrupt Flag

Timer Interrupt

Timer0 Interrupt

- An overflow (FFH → 00H) in the TMR0 register will set flag bit, TMR0IF (INTCON<2>).
- The interrupt can be enabled/disabled by setting/clearing enable bit, TMR0IE (INTCON<5>).

INTCON	GIE	PEIE	TMR0IE	INTE	RBIE	TMR0IF	INTF	RBIF

Timer1 Interrupt

- An overflow (FFFFH → 0000H) in the TMR1 register (TMR1H:TMR1L) will set flag bit, TMR1IF (PIR1<0>).
- The interrupt can be enabled/disabled by setting/clearing enable bit, TMR1IE (PIE1<0>).

PIR1	PSPIF(1)	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
PIE1	PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE

Steps in Programming Timer Interrupt

1. Initialize PIC.
2. Off the Timer0 or Timer1 by loading the value into the T1CON register.
3. Load reg. TMR0 or TMR1 with initial value.
4. Clear the TMR0IF flag or TMR1IF flag.

   INTCONbits.TMR0IF=0; **// Timer0**

   PIR1bits.TMR1IF=0; **// Timer1**
5. Enable the Timer Interrupt TMR0IE or TMR1IE

   INTCONbits.TMR0IE=1; **// Timer0**

   PIE1bits.TMR1IE=1; **// Timer1**
6. Enable peripheral interrupts **INTCONbits.PEIE=1;**
7. Enable all interrupts **INTCONbits.GIE=1;**

Steps in Programming Timer Interrupt

8. Start the Timer0 or Timer1.
9. Write Main Function. Do main function while waiting for interrupt.
10. When interrupt raised (**TMR0IF=1 or TMR1IF =1**), jump to ISR: **void interrupt *ISR_label* (void)**
11. In ISR:
    1. Stop the Timer0 or Timer1.
    2. Write ISR function.
    3. Reload TMR0 or TMR1 register.
    4. Clear the TMR0IF flag or TMR1IF flag.
    5. Re-enable interrupts.
    6. Start Timer0 or Timer1 (if required in ISR).

"interrupt" is a reserved keyword

Example

Write a program to generate a square wave with a period of 2 ms on pin RB5. Use Timer0 with 1:256 prescaler.

```
#include <htc.h>
__CONFIG (FOSC_HS & WDTE_OFF & PWRTE_OFF & BOREN_OFF & LVP_OFF);
#define _XTAL_FREQ      20000000
#define LED5 RB5        // LED for Timer1

void interrupt chk_isr(void) ;          // declare interrupt check ISR function
void Timer0_ISR(void);                  // declare Timer0 ISR function

void main(void)
{
    TRISB=0b00000000;                   // RB5 as output
    PORTB=0b00000000;                               // LED RB5 off
    OPTION_REG=0b00100111;              //Off Timer0, Timer0 module, 1:256
    TMR0=0xEC;                          // load Timer0 register with initial value
    INTCONbits.TMR0IF=0;                // clear Timer0 interrupt flag bit TMR0IF (INTCON<2>); INTCON=0b00000000;
    INTCONbits.TMR0IE=1;                // enable Timer0 interrupt; INTCON=0b11100000
    INTCONbits.PEIE=1;
    INTCONbits.GIE=1;
```

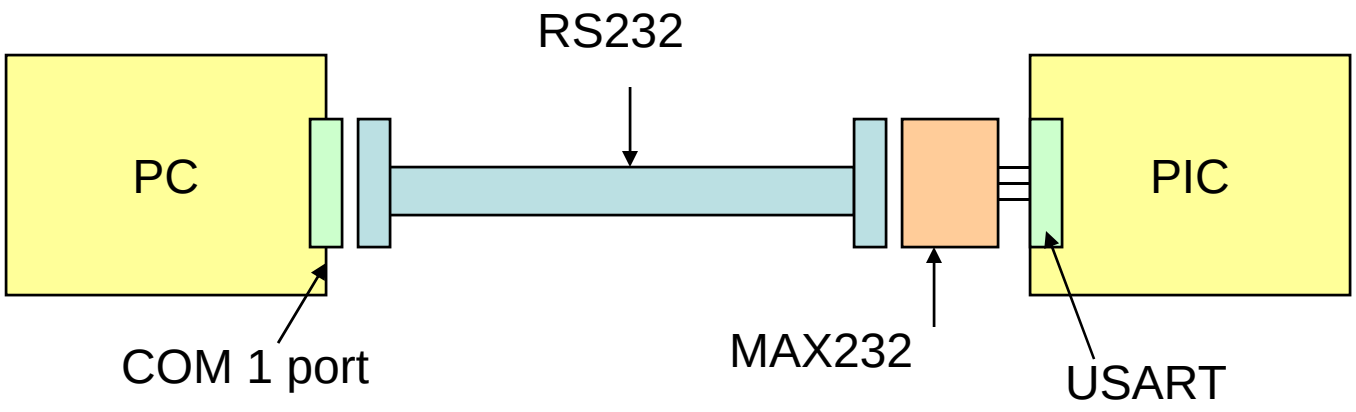
```
        while(1)                        // keep looping until interrupt comes, jumps to interrupt function
        {
                OPTION_REG=0b00000111;          // turn on Timer0 T0CS (OPTION_REG<5>)
        }
}

void interrupt chk_isr(void)            // interrupt  function
{
        if(INTCONbits.TMR0IF==1)        // Timer0 causes interrupt?
    {
        Timer0_ISR();                   // Yes. Execute Timer0 ISR  function
        }
}

void Timer0_ISR(void)                   // Timer0 ISR function
{
        OPTION_REG=0b00100111;          // OFF Timer0
        LED5 = !LED5;                   // toggle RB5
        TMR0=0xEC;                      // re-load Timer0 register
        INTCONbits.TMR0IF=0;            // clear Timer0 interrupt flag bit TMR0IF (INTCON<2>)
        INTCONbits.TMR0IE=1;            // enable Timer0 interrupt; INTCON=0b11100000
        INTCONbits.PEIE=1;
        INTCONbits.GIE=1;
}
```

"interrupt" is a reserved keyword

Data Communication

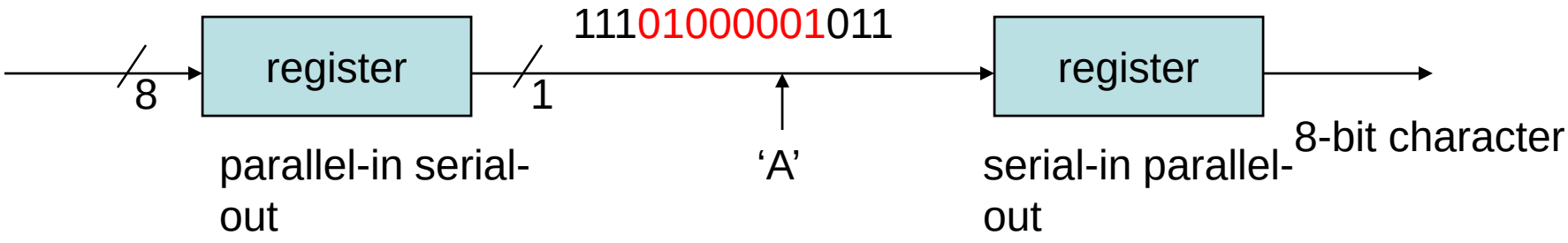
PIC and PC

- The 8051 module connects to PC by using RS232.
- RS232 is a protocol which supports half-duplex, synchronous/asynchronous, serial communication.



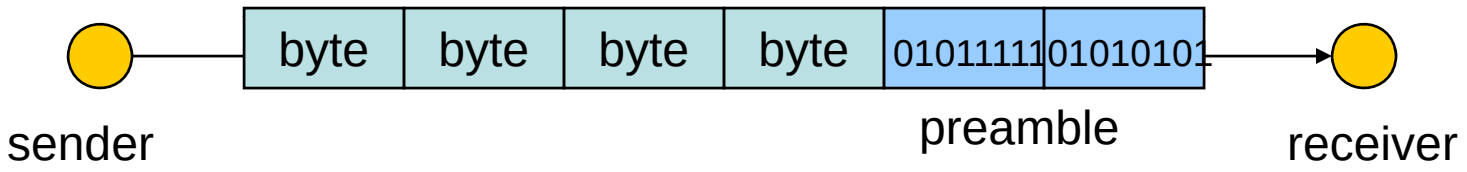
Serial Communication

- How to transfer data?
  - Sender:
    - The byte of data must be converted to serial bits using a parallel-in-serial-out shift register.
    - The bit is transmitted over a single data line.
  - Receiver
    - The receiver must be a serial-in-parallel-out shift register to receive the serial data and pack them into a byte.

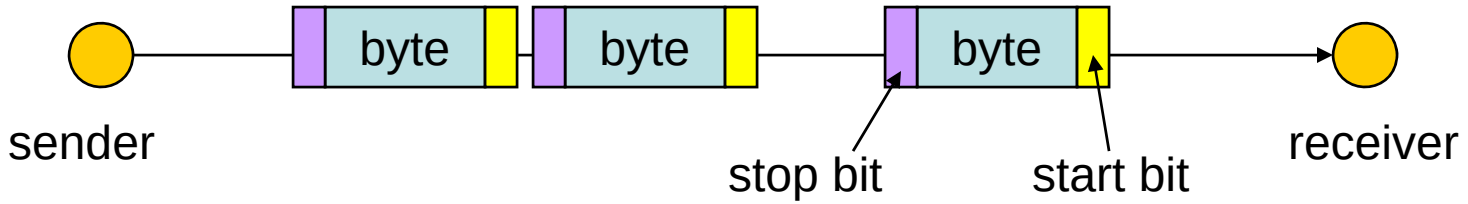


Asynchronous vs. Synchronous

- Serial communication uses two methods:
  - In synchronous communication, data is sent in blocks of bytes.



  - In asynchronous communication, data is sent in bytes.



UART and USART

- It is possible to write software to use both methods, but the programs can be tedious and long.
- Special IC chips are made for serial communication:
  - USART (universal synchronous-asynchronous receiver-transmitter)
  - UART (universal asynchronous receiver-transmitter)
- The PIC chip has a built-in USART.

Serial Communication

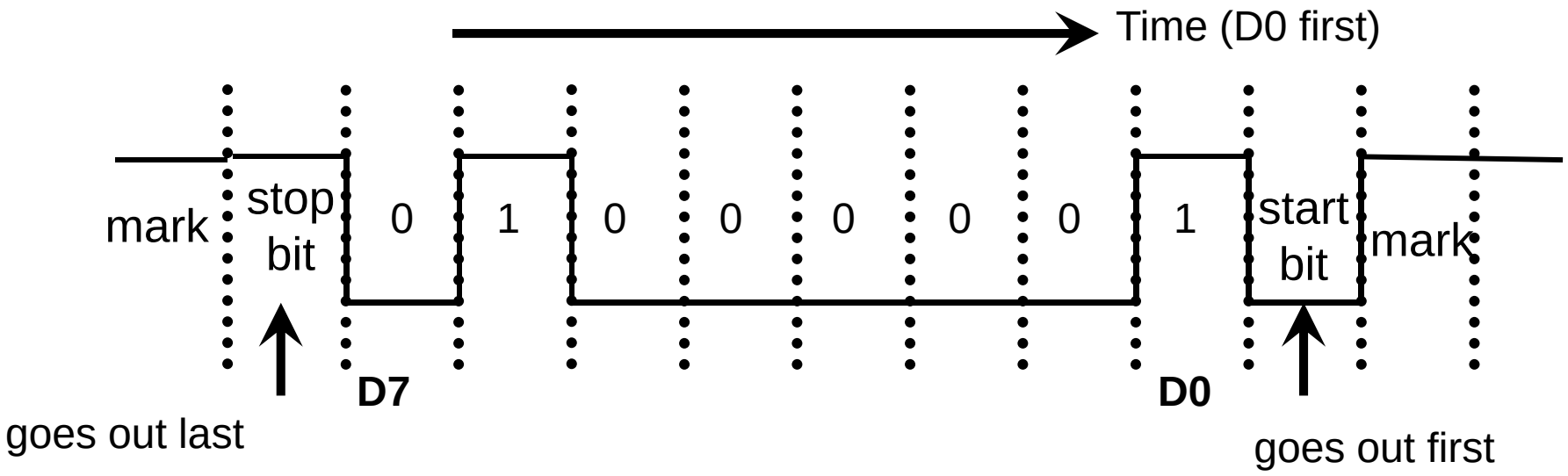
PIC Serial Communication

- The PIC has serial communication capability built into it.
  - Synchronous & Asynchronous mode

- How to detect that a character is sent via the line in the asynchronous mode?
  - Answer: Data framing!

Framing

- Framing: Each character is placed in between start and stop bits
  - Framing ASCII “A” (41H)



Framing

- We have a total of 10 bits for each character:
  - 8-bits for the ASCII code
  - 2-bits for the start and stop bits
  - 25% overhead
- In some systems in order to maintain data integrity, the parity bit is included in the data frame.
  - In an odd-parity bit system the total number of bits, including the parity bit, is odd.
  - UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

Data Transfer Rate

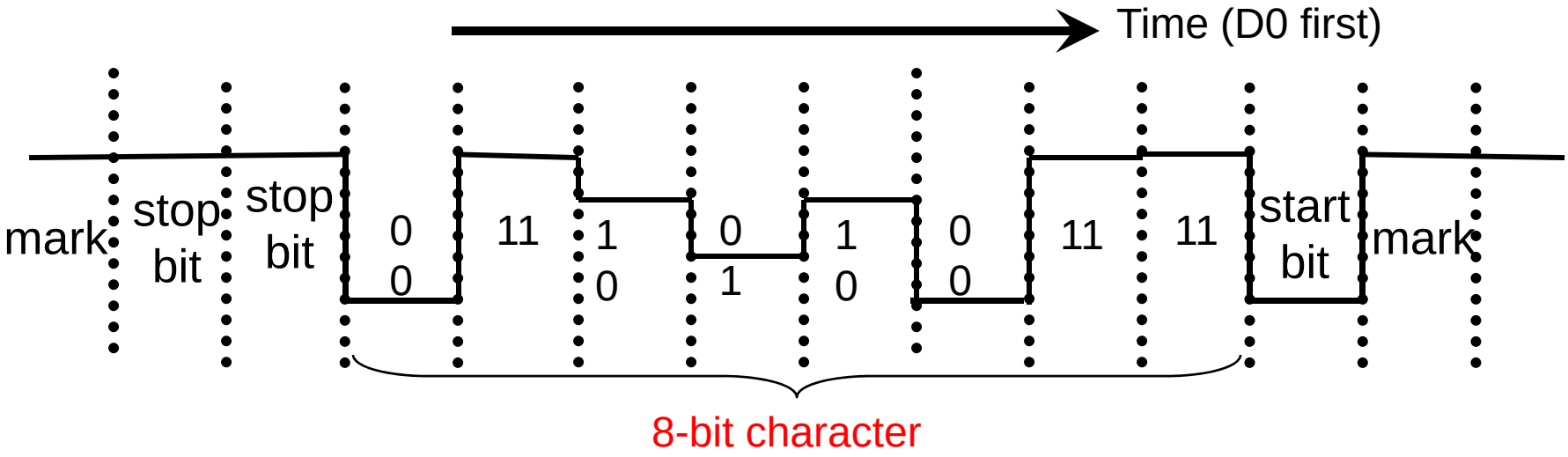
- How fast is the data transferred?
- Three methods to describe the speed:
  - Baud rate is defined as the number of signal changes per sec.
    - The rate of data transfer is stated in *Hz* (used in modem).
  - Data rate is defined as the number of bits transferred per sec.
    - Each signal has several voltage levels.
    - The rate of data transfer is stated in *bps* (bits per second).
  - Effective data rate is defined as the number of actual data bits transferred per second.
    - Redundant bits must be removed

Data Transfer Rate

- The data transfer rate depends on communication ports incorporated into that system.
  - Ex: 100-9600 bps in the early IBM PC/XT
  - Ex: 56 kbps in Pentium-based PC
  - The baud rate is generally limited to 100kHz.

Example of Data Transfer Rate

- Data is sent in the following asynchronous mode:
  - 2400 baud rate
  - each signal has 4 voltage levels (-5V, -3V, 3V, 5V)
  - one start bit, 8-bit data, 2 stop bits



Example of Data Transfer Rate

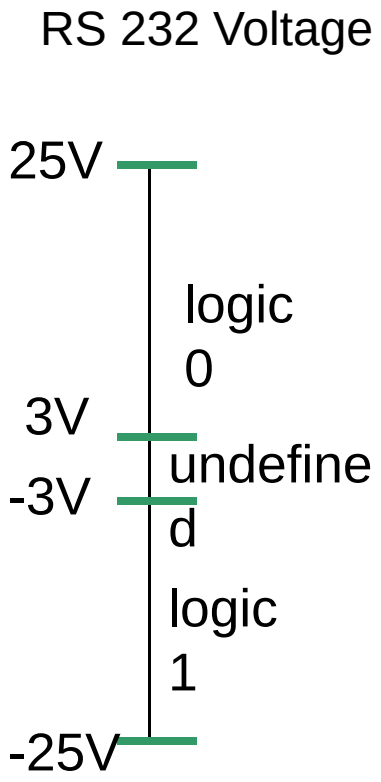
- 2400 baud = 2400 signals per second =2400 Hz
- 4 voltage level: $Log_2 4=2$
  - 2 bits is sent in every signal change
- Data rate = 2 * 2400 Hz = 4800 bps
- Effective ratio = 8 / (1+8+2) =8/11
- Effective data rate = data rate * effective ratio = 4800 * 8 /11=3490.9

RS232 Standard

- RS232 is an interfacing standard which is set by the Electronics Industries Association (EIA) in 1960.
  - RS232 is the most widely used serial I/O interfacing standard.
  - RS232A (1963), RS232B (1965) and RS232C (1969), now is RS232E
- Define the voltage level, pin functionality, baud rate, signal meaning, communication distance.

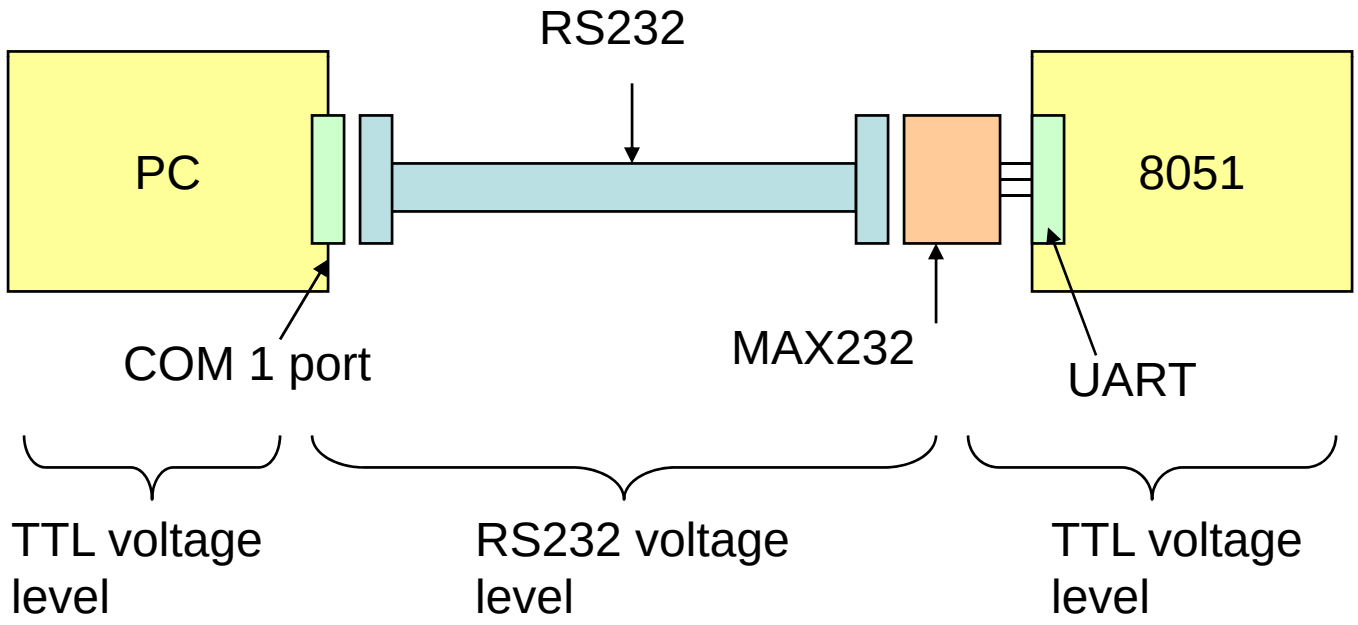
RS232 Voltage Level

- The input and output voltage of RS232 is not of the TTL compatible.
  - RS232 is older than TTL.
- We must use voltage converter (also referred to as line driver) such as MAX232 to convert the TTL logic levels to the RS232 voltage level, and vice versa.
  - MAX232, TSC232, ICL232

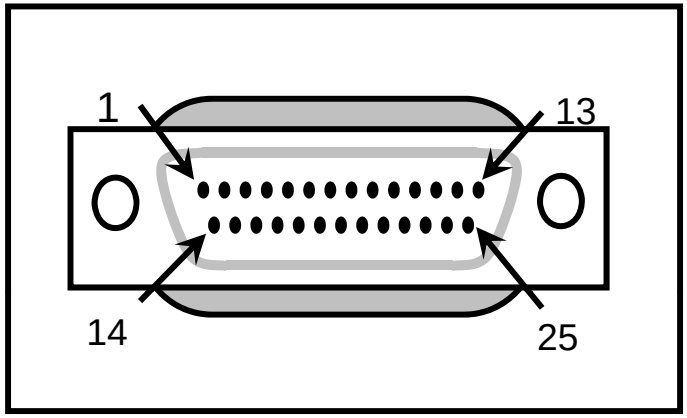


MAX232

- MAX232 IC chips are commonly referred to as line drivers.

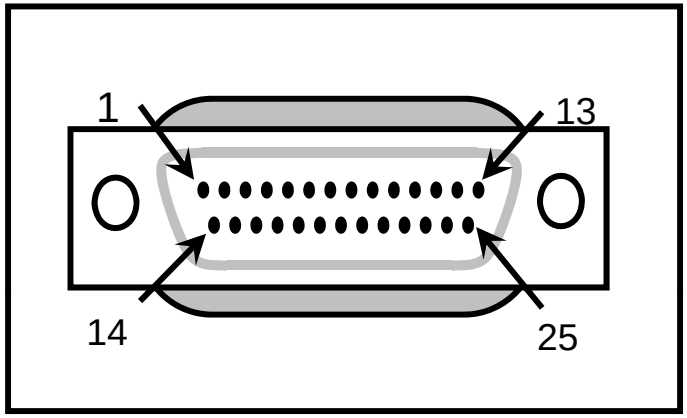


RS232 Pins for DTE



Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send (RTS)
5	Clear to send (CTS)
6	Data set ready (DSR)
7	Signal ground (GND)
8	Data carrier detect (DCD)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send

RS232 Pins for DTE

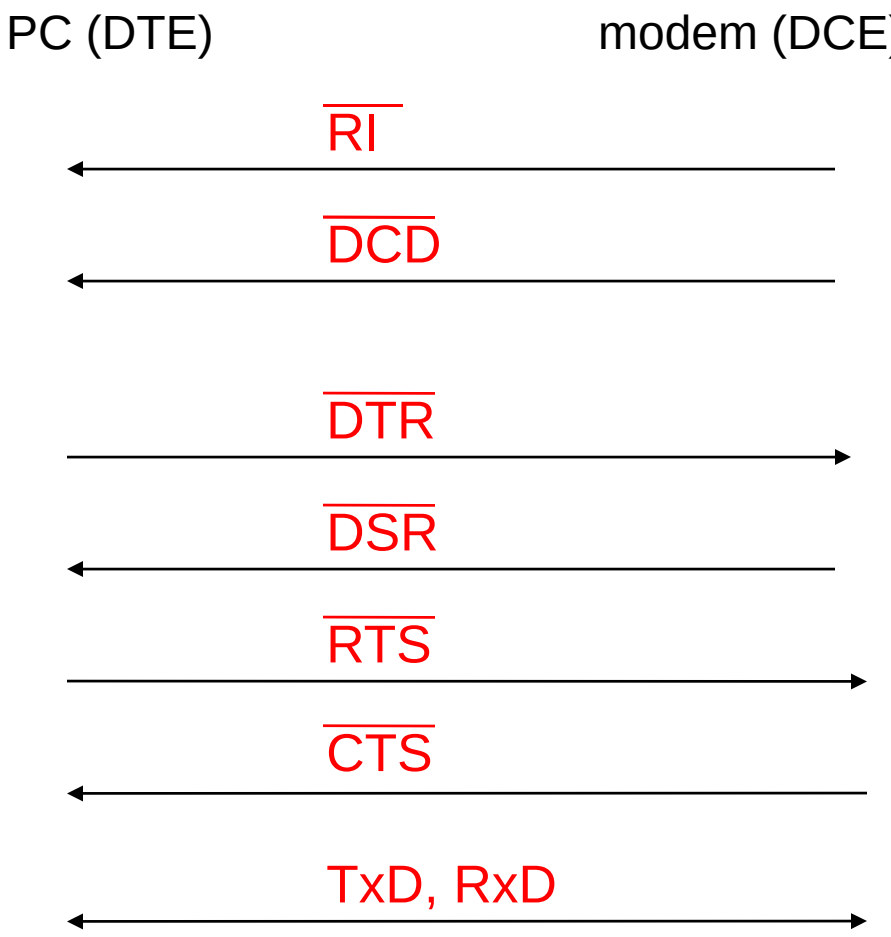


Pin	Description
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to sent
20	Data terminal ready (DTR)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

RS232 Handshaking Signals

- Many of the pins of the RS232 connector are used for handshaking signals.
  - DTR (data terminal ready)
  - DSR (data set ready)
  - RTS (request to send)
  - CTS (clear to send)
  - RTS and CTS are hardware control flow signals.
  - DCD (carrier detect, or data carrier detect)
  - RI (ring indicator)
- They are not supported by the 8051 UART chips.

Communication Flow



Telephone is ringing

Connection between two modems is set

PC is ready

modem is ready

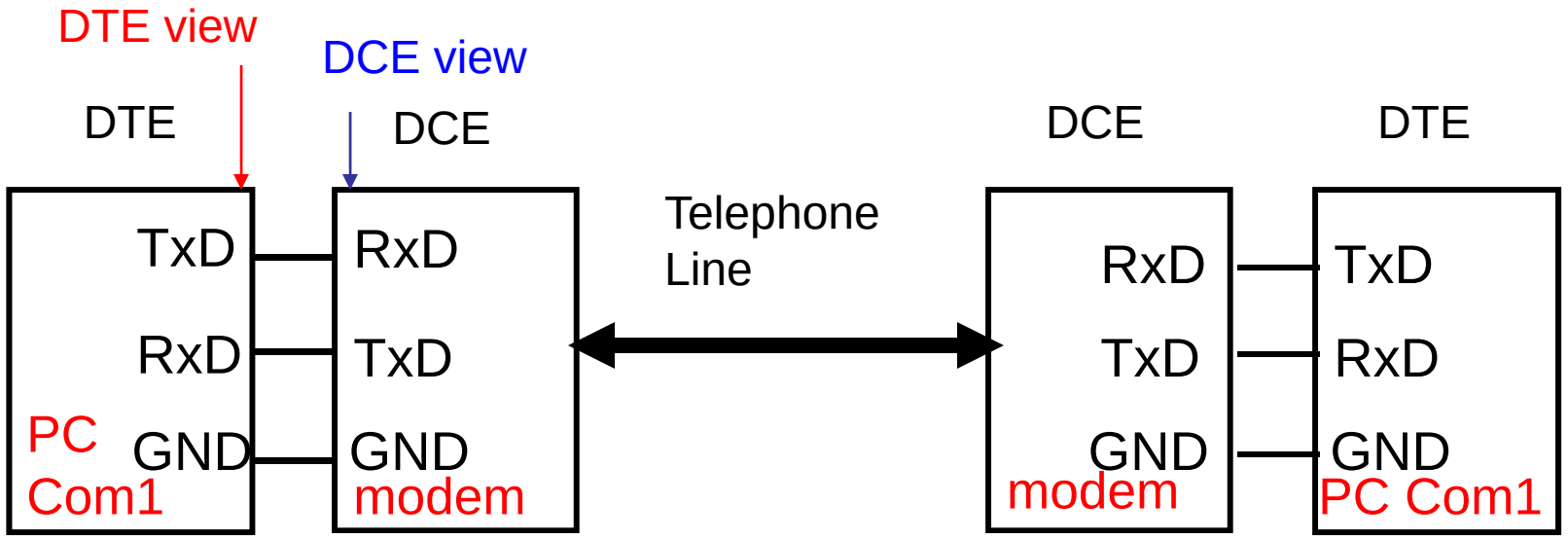
PC wants to sent data

modem is ready to receive

transmit data

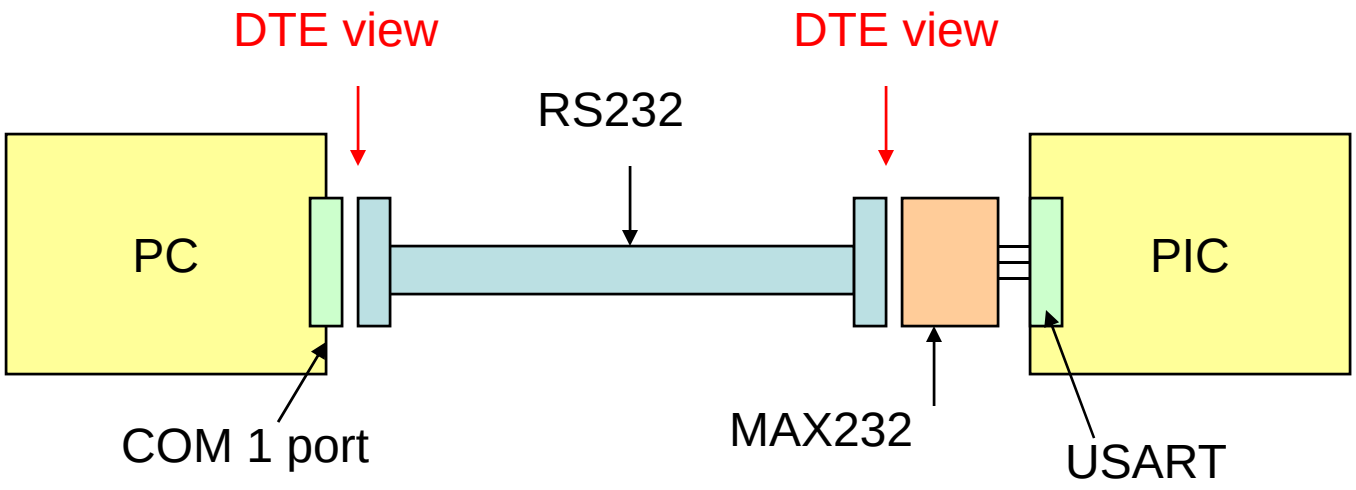
DTE and DCE

- Communication Equipments are classified as
  - DTE (data terminal equipment)
    - Terminals and computers that send and receive data
  - DCE (data communication equipment)
    - Communication equipment (only for transfer data), modem



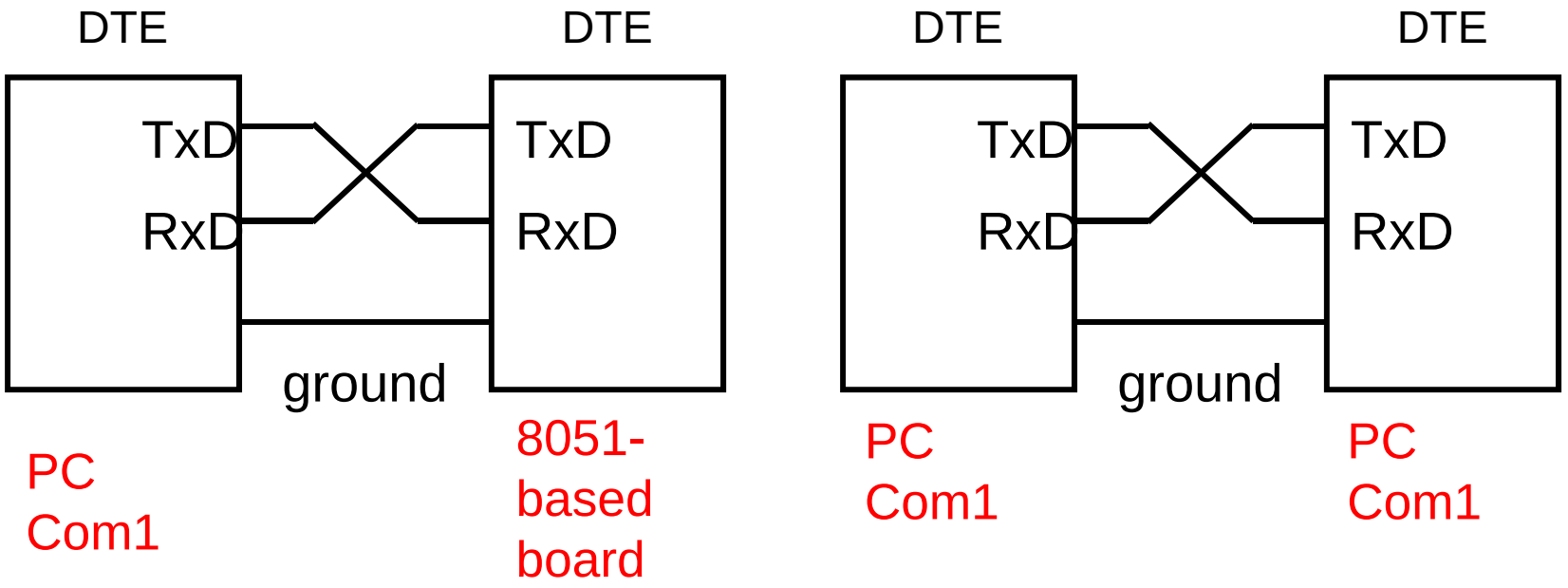
IBM PC/compatible COM ports

- IBM PC had 2 COM ports.
  - Both COM ports have RS232-type connectors.
  - For mouse, modem



Null Modem Connection

- The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and GND.
  - null modem connection



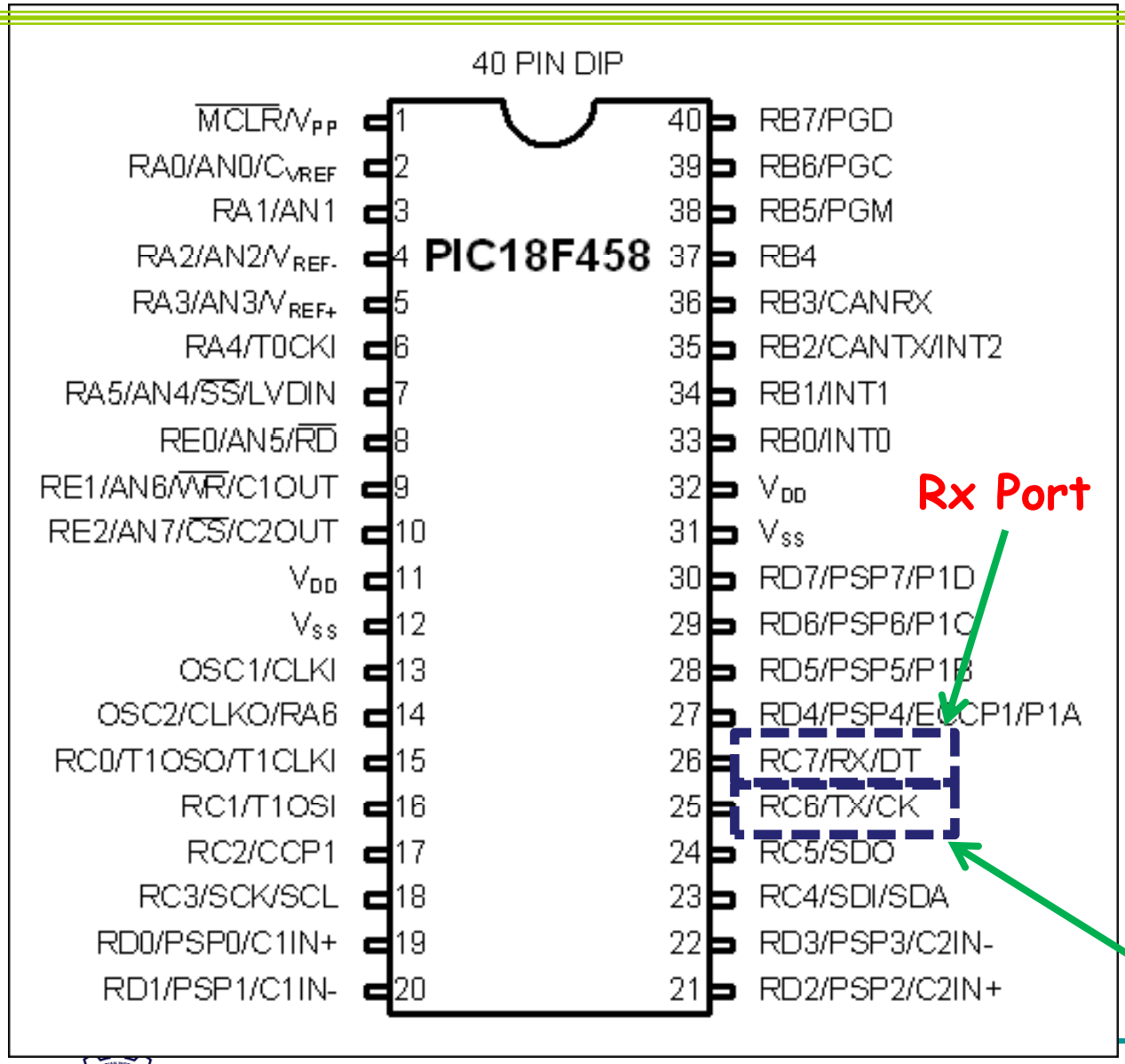
RS422 & RS485

- By using RS232, the limit distance between two PCs is about 15m.
- It works well even the distance=30m.
- If you want to transfer data with long distance (ex: 300m), you can use RS422 or RS485.

TxD and RxD pins in the PIC

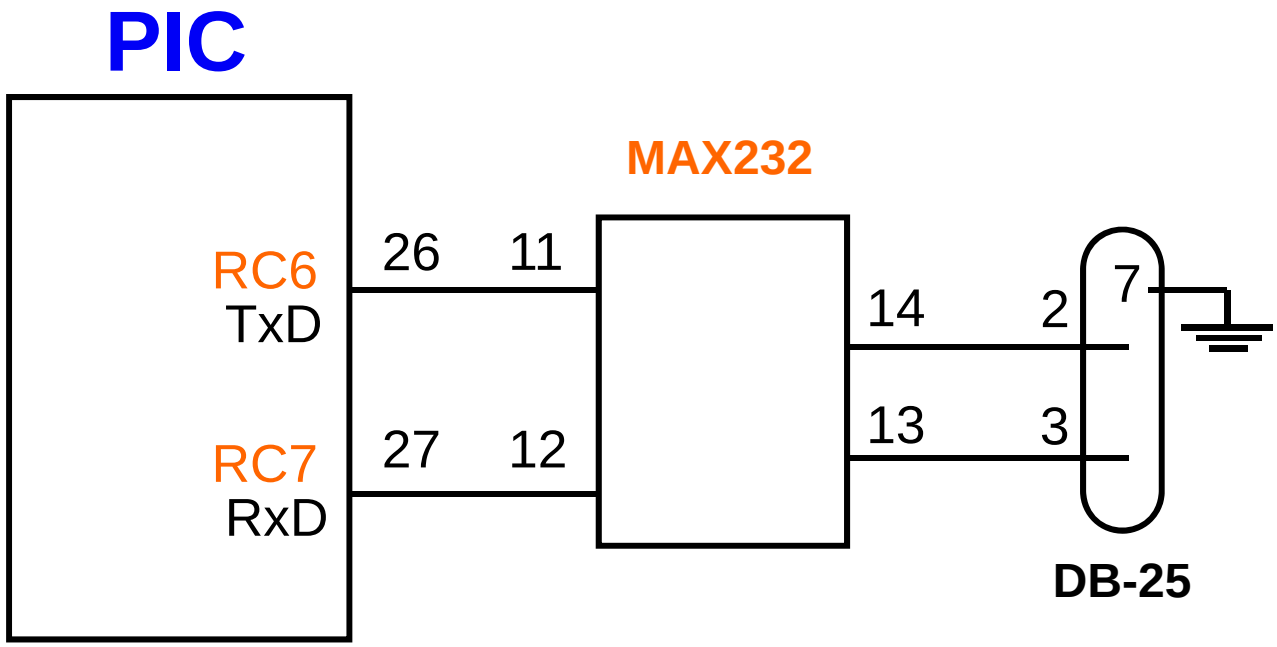
- In PIC, the data is received from or transmitted to
  - Rx: received data
  - Tx: transmitted data
- TxD and RxD of the PIC are TTL compatible.
- The PIC requires a line driver to make them RS232 compatible.
  - One such line driver is the MAX232 chip.

PIC18 Serial Port



MAX232

- MAX232 chip converts from RS232 voltage levels to TTL voltage levels, and vice versa.
  - MAX232 uses a +5V power source which is the same as the source voltage for the 8051.

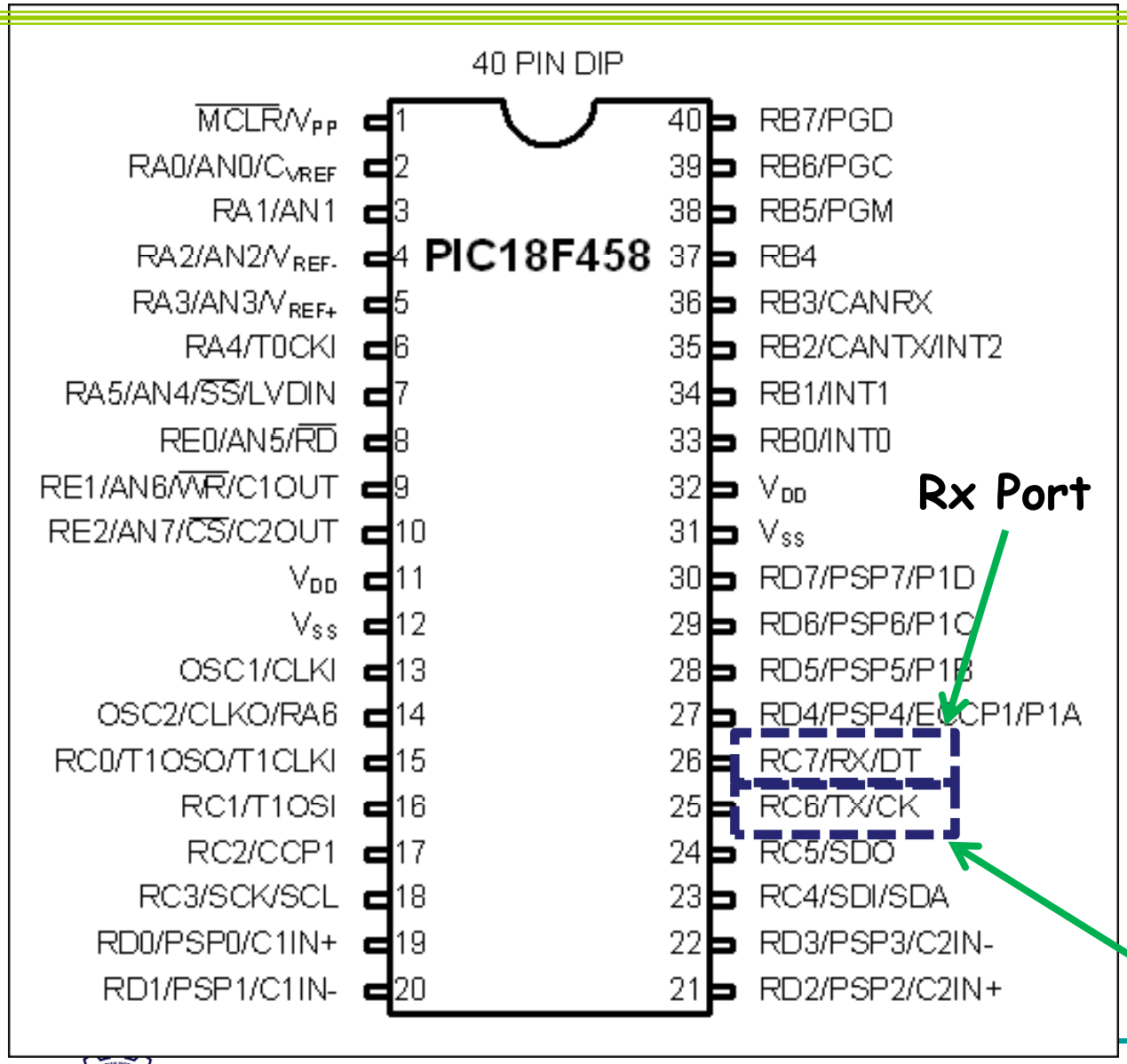


MAX 232

- MAX232 has two sets of line drivers.
  - MAX232 requires four capacitors ranging from 1 to 22 μF. The most widely used value for these capacitors is 22μF.

- MAX233 performs the same job as the MAX232 but eliminates the need for capacitors.
  - MAX233 and MAX232 are not pin compatible.

PIC18 Serial Port



- USART has both
  - Synchronous
  - Asynchronous
- 6 registers
  - SPBRG
  - TXREG
  - RCREG
  - TXSTA
  - RCSTA
  - PIR1

SPBRG Register and Baud Rate in the PIC18

- The baud rate in is programmable
- loaded into the SPBRG decides the baud rate
- Depend on crystal frequency

$$mBR = \frac{F_{osc}}{4*16*(X+1))}$$

Baud Rate	SPBRG (Hex Value)
38400	3
19200	7
9600	F
4800	20
2400	40
1200	81

*For XTAL = 10MHz

Baud rate Formula

If Fosc = 10MHz

X = (156250/Desired Baud Rate) - 1

Example:

Desired baud rate = 1200, Clock Frequency = 10MHz

X = (156250/1200) – 1

X = 129.21 = 129 = 81H

TXREG Register

- 8-bit register used for serial communication
- For a byte of data to be transferred via the Tx pin, it must be moved to the TXREG register first.
- The moment a byte is written into TXREG, it is fetched into a non-accessible register TSR

MOVFF PORTB, TXREG

- The frame contains 10 bits

RCREG Register

- 8-bit register used for serial communication
- When the bits are received serially via the Rx pin, the PIC18 deframes them by eliminating the START and STOP bit, making a byte out of data received and then placing it in the RCREG register

MOVFF RCREG, PORTB

TXSTA (Transmit Status and Control Register)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN(1)	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

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)

1 = Transmit enabled
0 = Transmit disabled

TXSTA . . .

bit 4	**SYNC:** EUSART Mode Select bit 1 = Synchronous mode 0 = Asynchronous mode
bit 3	**SENDB:** Send Break Character bit Asynchronous mode: 1 = Send Sync Break on next transmission (cleared by hardware upon completion) 0 = Sync Break transmission completed Synchronous mode: Don't care.
bit 2	**BRGH:** High Baud Rate Select bit Asynchronous mode: 1 = High speed 0 = Low speed Synchronous mode: Unused in this mode.
bit 1	**TRMT:** Transmit Shift Register Status bit 1 = TSR empty 0 = TSR full
bit 0	**TX9D:** 9th bit of Transmit Data Can be address/data bit or a parity bit.

Note 1: SREN/CREN overrides TXEN in Sync mode.

RCSTA (Receive Status and Control 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

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

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 (held in Reset)

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.

RCSTA . . .

bit 4 **CREN:** Continuous Receive Enable bit

Asynchronous mode:
1 = Enables receiver
0 = Disables receiver

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 loads 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

Asynchronous mode 9-bit (RX9 = 0):
Don't care.

bit 2 **FERR:** Framing Error bit

1 = Framing error (can be updated by reading RCREG register and receiving 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

This can be an address/data bit or a parity bit and must be calculated by user firmware.

PIR1 (Peripheral Interrupt Request Register 1)

---	--	RCIF	TXIF	--	---	--	

RCIF Receive interrupt flag bit
1 = The UART has received a byte of data and it is sitting in the RCREG register (receive buffer), waiting to be picked up.
Upon reading the RCREG register, the RCIF is cleared to allow the next byte to be received.
0 = The RCREG is empty.

TXIF Transmit interrupt flag bit
0 = The TXREG register is full.
1 = The TXREG (transmit buffer) register is empty.

The importance of TXIF: To transmit a byte of data, we write it into TXREG. Upon writing a byte into TXREG, the TXIF flag is cleared. When the entire byte is transmitted via the TX pin, the TXIF flag bit is raised to indicate that it is ready for the next byte. So, we must monitor this flag before we write a new byte into TXREG, otherwise, we wipe out the last byte before it is transmitted.

Programming the PIC18 to Transfer Data Serially

1. TXSTA register = 20H: Indicating asynchronous mode with 8-bit data frame, low baud rate and transmit enabled
2. Set Tx pin an output (RC6)
3. Loaded SPBRG for baud rate
4. Enabled the serial port (SPEN = 1 in RCSTA)
5. The character byte to transmit must be written into TXREG
6. Keep Monitor TXIF bit
7. To transmit next character, go to step 5

Example

Write a program for the PIC18 to transfer the letter ’B' serially at 9600 baud continuously. Assume XTAL = 10 MHz

```
        MOVLW     B'00100000'
        MOVWF     TXSTA
        MOVLW     D'15'; 9600 bps
        MOVWF     SPBRG
        BCF       TRISC,  TX
        BSF       RCSTA,  SPEN
OVER    MOVLW     A’B'
S1      BTFSS  PIR1,    TXIF
        BRA       S1
        MOVWF     TXREG
        BRA       OVER
```

TXSTA: Transmit Status and Control Register

The importance of the TSR register. To transfer a byte of data serially, we write it into TXREG. The TSR (transmit shift register) is an internal register whose job is to get the data from the TXREG, frame it with the start and stop bits, and send it out one bit at a time via the TX pin. When the last bit, which is the stop bit, is transmitted, the TRMT flag is raised to indicate that it is empty and ready for the next byte. When TSR fetches the data from TXREG, it clears the TRMT flag to indicate it is full. Notice that TSR is a parallel-in-serial-out shift register and is not accessible to the programmer. We can only write to TXREG. Whenever the TSR is empty, it gets its data from TXREG and clears the TXREG register immediately, so it does not send out the same data twice.

Programming the PIC18 to Receive Data Serially

1. RCSTA register = 90H: To enable the continuous receive in addition to the 8-bit data size option
2. The TXSTA register = 00H: To choose the low baud rate option
3. Loaded SPBRG for baud rate
4. Set Rx pin an input
5. Keep Monitor RCIF bit
6. Move RCREG into a safe place
7. To receive next character, go to step 5

Example

```
;Write a program for the PIC18 to receive data serially and
;put them on PORTB. Set the baud rate at 9600, 8-bit data
;and 1 stop bit

        MOVLW       B'10010000'
        MOVWF       RCSTA
        MOVLW       D'15'
        MOVWF       SPBRG
        BSF         TRISC,   RX
        CLRF        TRISB
R1      BTFSS   PIR1,   RCIF
        BRA         R1
        MOVFF       RCREG, PORTB
        BRA         R1
```

Increasing the Baud Rate

- Faster Crystal
  - May not be able to change crystal
- TXSTA.BRGH bit
  - Normally used low
  - Can be set high
  - Quadruples rate when set high

Thank You