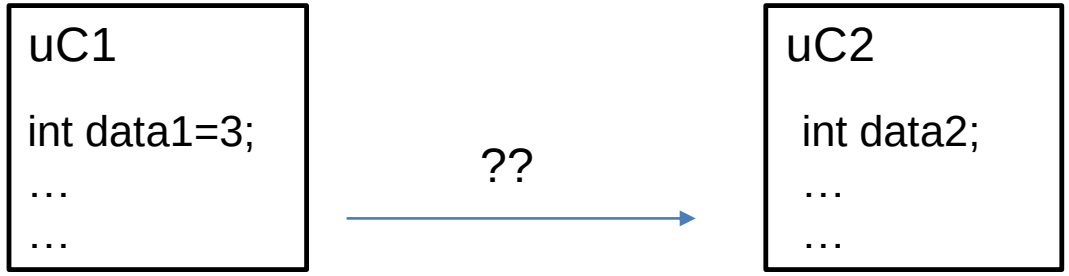
Computer interfacing (CI)

# 8. Communications.

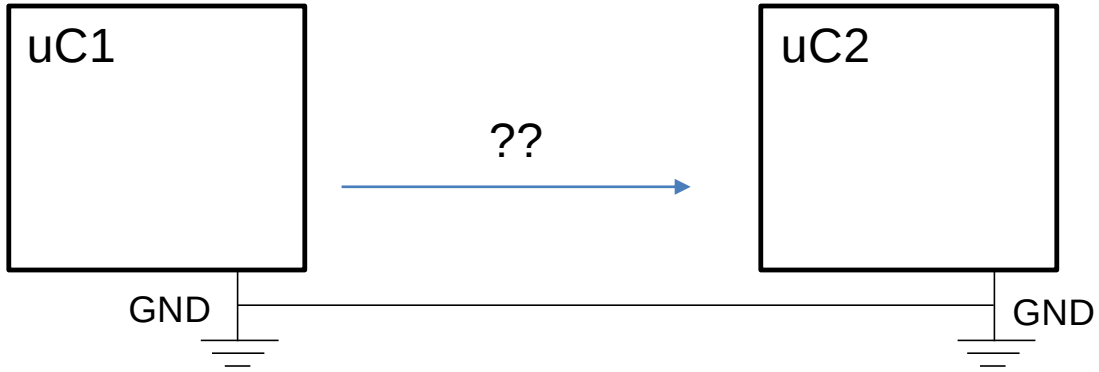
# 8.1 Need of communications

Two independent systems, uC1 and uC2, need to share some data (collaborate, networks...)




We'd like to do something like "data2=data1;" But the information is in separated devices RAM memories.

To solve it, some **wiring** will be necessary between the microcontrollers I/O.

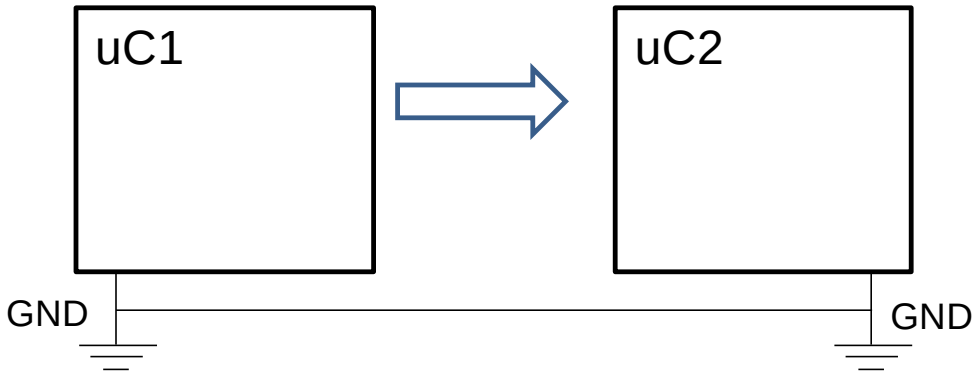



But! **voltage references** must be the same in both systems (remember $V_{IH}$, $V_{IL}$, $V_{OH}$, $V_{OL}$), that's why **GND** needs to be the **first** connection.

# 8.2 Basic concepts of communications

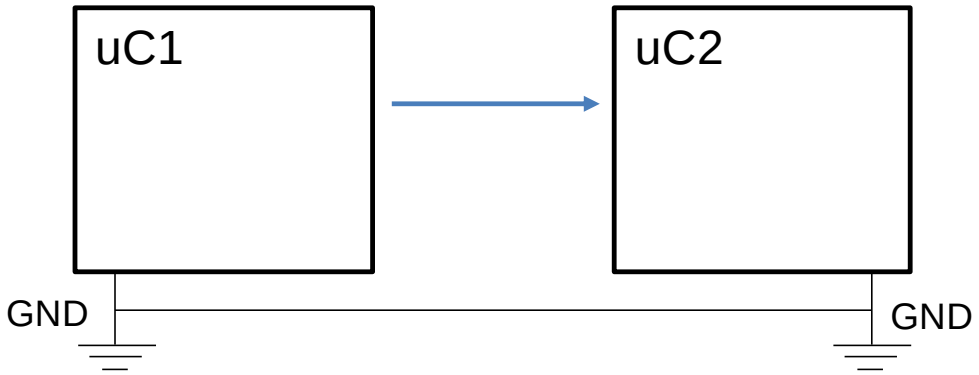
## Classification of wiring systems:

**Parallel:** n wires are sent per direction. It allows to do faster transfers:




PORTx= data1;  data2= PORTy;

**Serial:** only 1 wire is sent per direction. Slower transfers, but saving money.



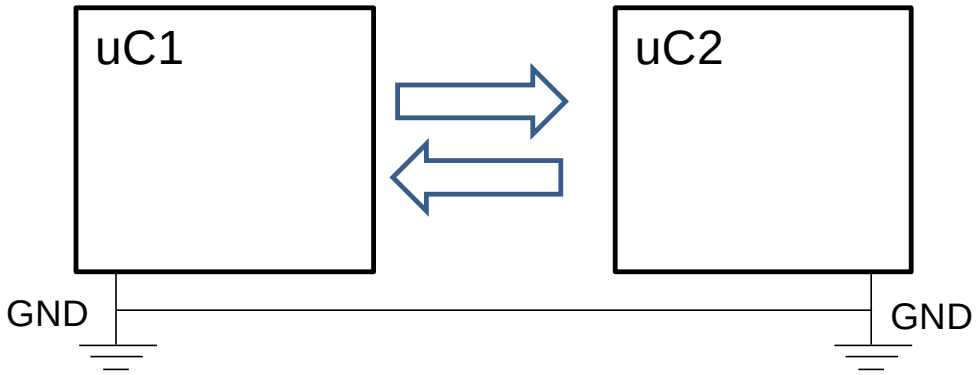

```
PORTxbits.Rx0= data1.bit0;    data2.bit0=PORTybits.Ry0;
PORTxbits.Rx0= data1.bit1;    data2.bit1=PORTybits.Ry0;
PORTxbits.Rx0= data1.bit2;    data2.bit2=PORTybits.Ry0;
…                             …
```

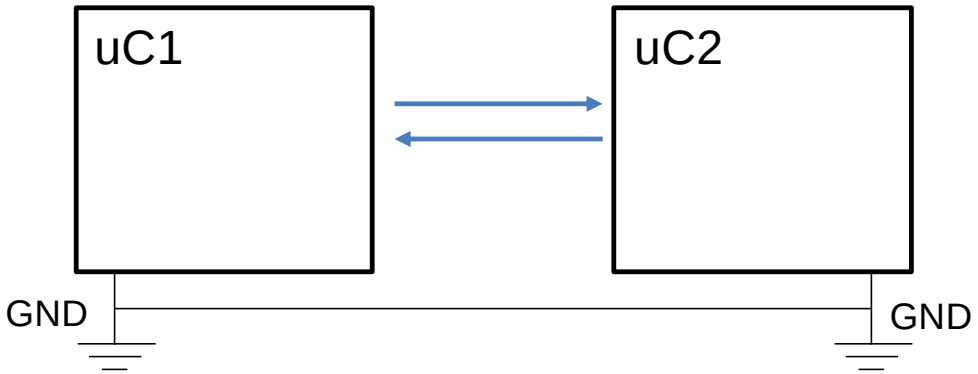
But…, when to start? When is done?
What if clock speeds are different?

# 8.2 Basic concepts of communications

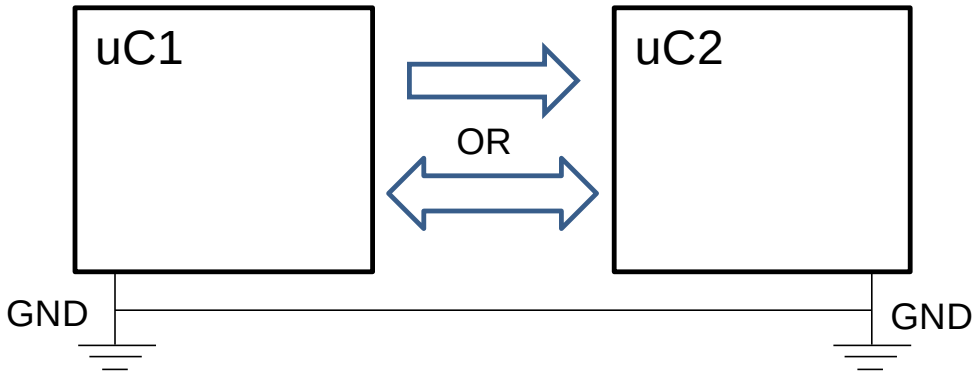
## Classification of wiring systems:

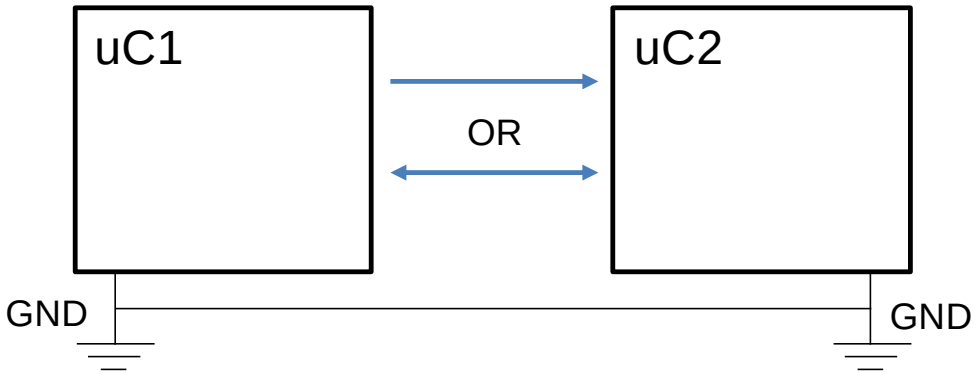
**Full-duplex:** Data can be sent simultaneously in both directions.




**Half-duplex:** Data is sent only in one direction (or one at a time)

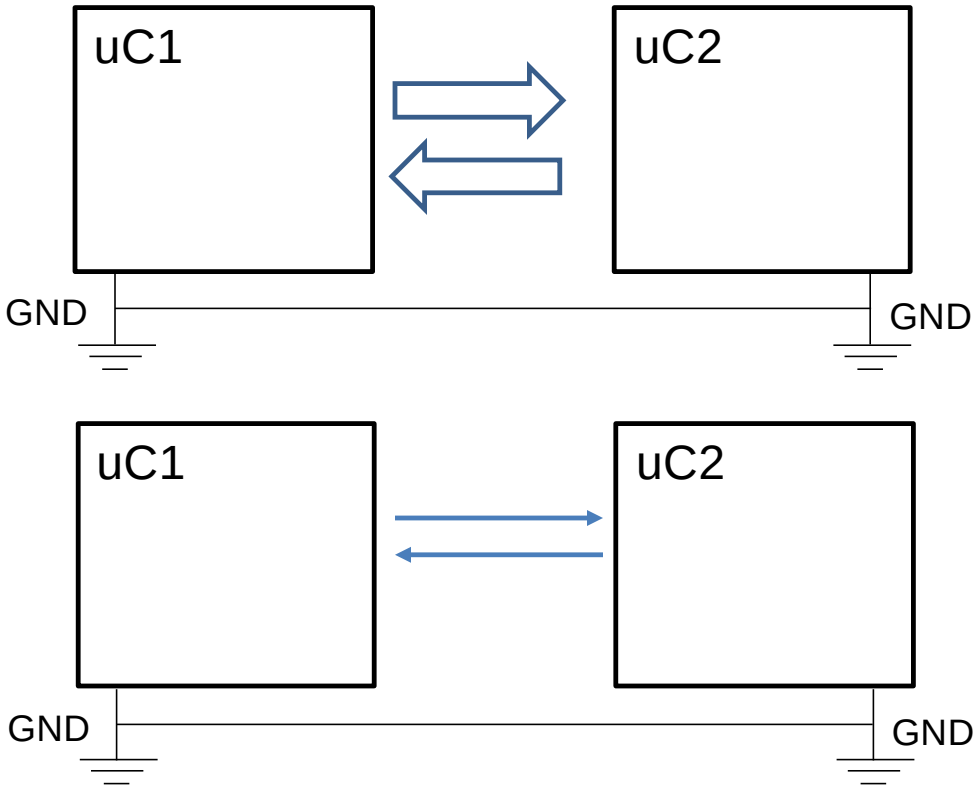


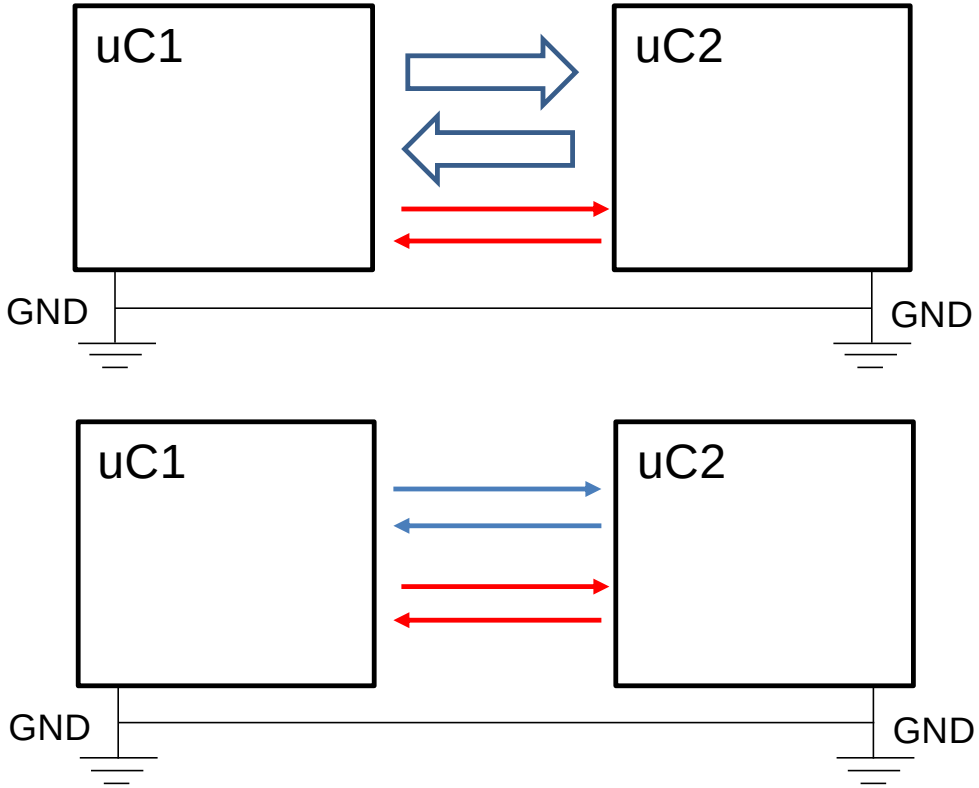
# 8.2 Basic concepts of communications

## Classification of wiring systems:

**Asynchronous:** No clock or extra signaling (handshaking) is added.



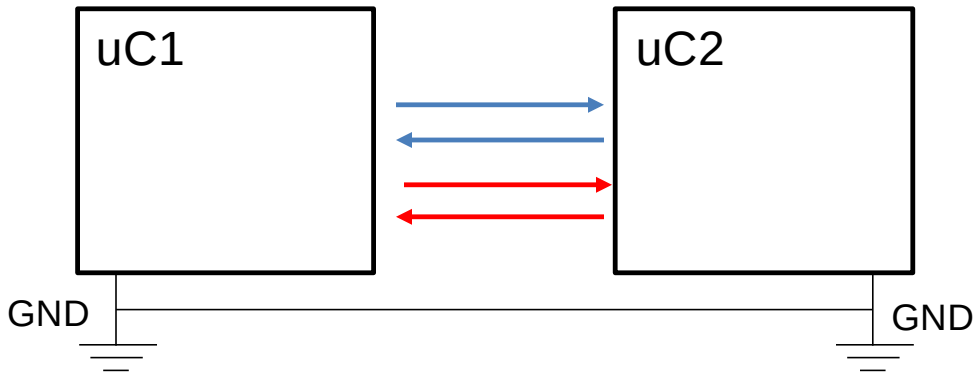

**Synchronous:** Clock and maybe other signals are added to the communications protocol.



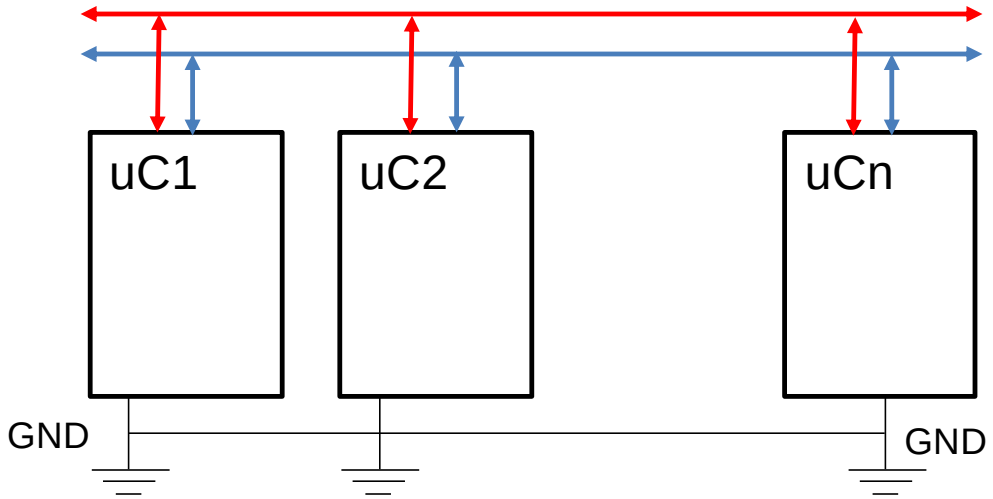
# 8.2 Basic concepts of communications

## Classification of wiring systems:

**Point to point:** There are only two devices involved in the communication process.



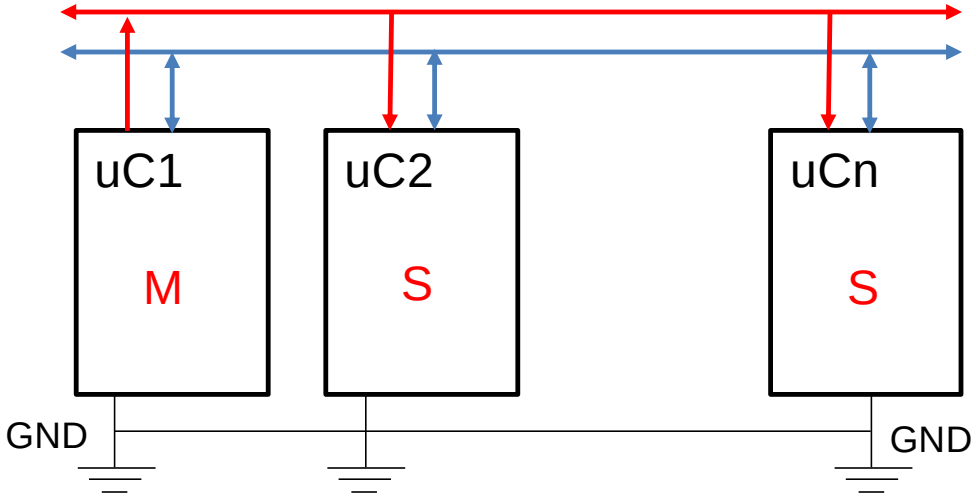

**Multipoint or network:** There will be a large number of devices connected (arbitration needed).



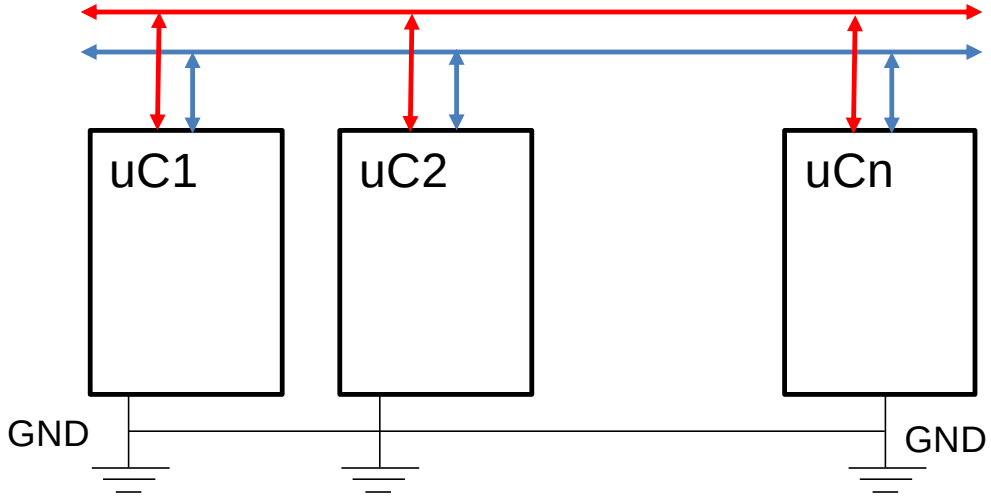
# 8.2 Basic concepts of communications

## Classification of wiring systems:

**Master/Slave:** Only one device can start the communication process.



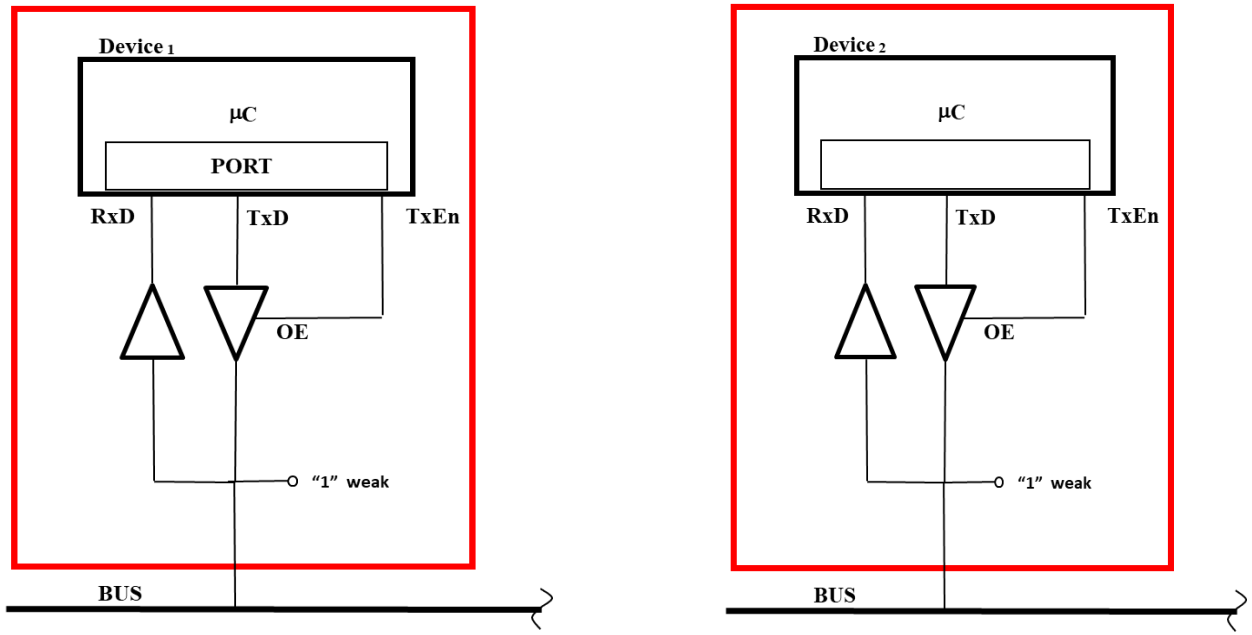

**Multimaster:** More than one device can start the communication process.




Typically, the master will generate the clock, create handshake signals, etc…

# 8.2 Basic concepts of communications

- As seen, existing communication systems can be labeled as serial or parallel, half-duplex or full-duplex, synchronous or asynchronous, point-to-point or multipoint, master-slave or multimaster. There are examples for almost every combination.
- Most interfaces can be implemented on simple microcontroller pins, some times we need additional hardware (e.g. bus drivers).
- Half-duplex multipoint systems, like Ethernet, use the following interface:



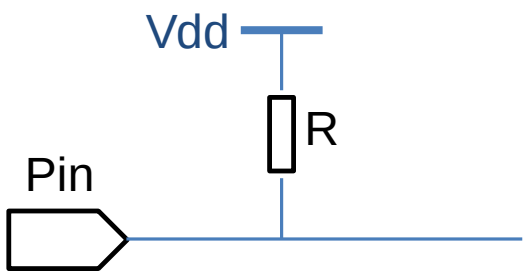
# 8.3 Serial line protocol

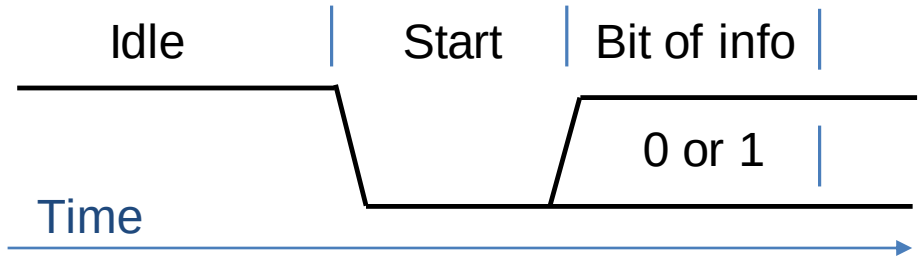
Is the simplest most widely used serial communications protocol.

Is a serial, asynchronous, full-duplex, point-to-point, multimaster communication protocol. Developed in the 1960's. Still in use.

Main ideas:

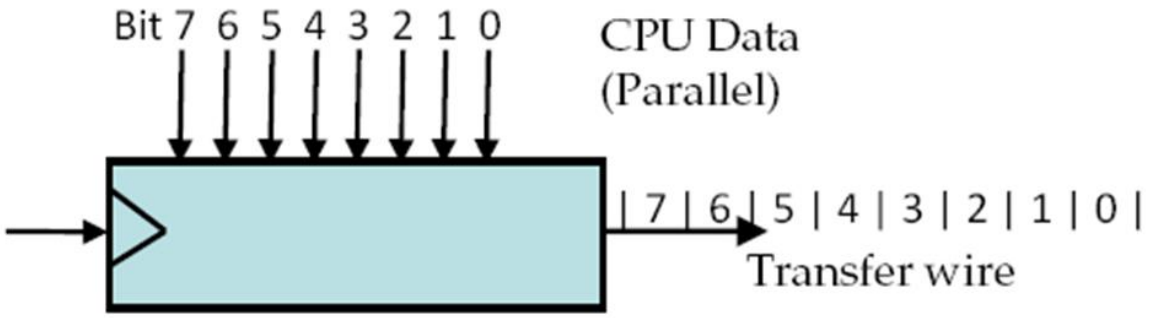
- There is a default (idle) state in the bus (typically a logical one).
- A break or start signal indicates that communication starts (typically a zero).
- After that, bits are sent one by one, occupying a fixed, preset, amount of time.



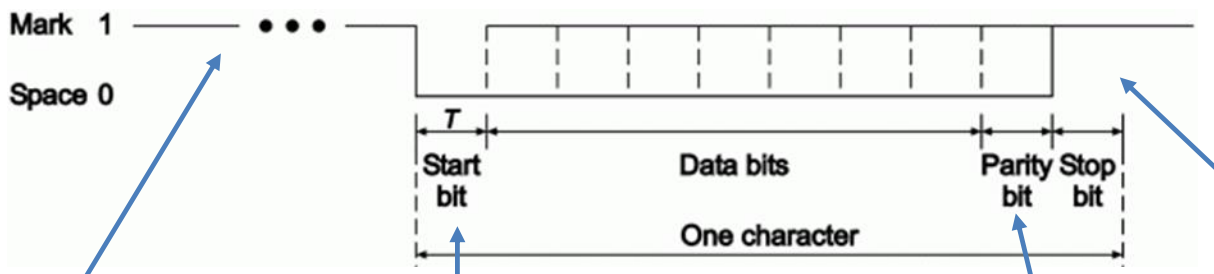



# 8.3 Serial line protocol

Key element: Shift register, as a **parallel to serial** converter.




Function chronogram:




Default value 'Idle' is a 1

Start bit is a 0, breaks the idle state

Parity bit (optional)

Stop bit is a 1

Total bit transmitted = 1 bit (start) + n bit (Data) + 0 or 1 bit (parity) + 1 to 2 bit (stop)

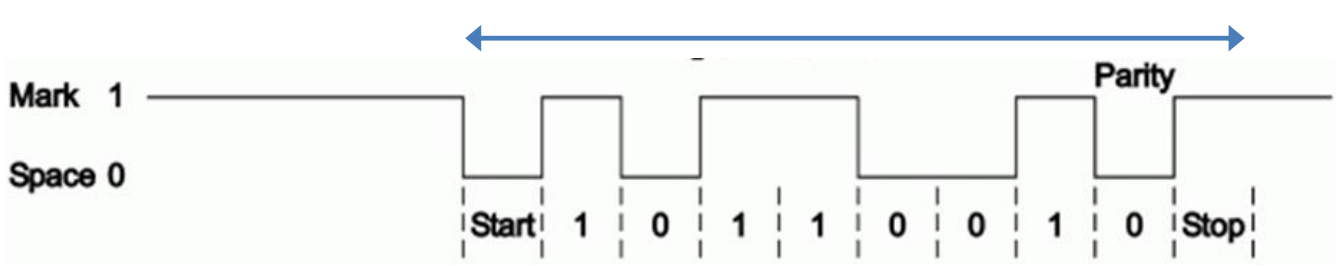
# 8.3 Serial line protocol

There are 4 parameters to be defined *a priori*:

- **Transmission speed**: 1200, 2400, 4800, 9600, … , 115200 bps (bits per second or baud). *Note:* the inverse of transmission speed is the duration of a bit.
- **Bits per character:** bits used to codify the information 5, 6, 7 or 8 (standard), used to save time if less symbols are to be sent.
- **Parity bit:** can be,
  - even (count the number of 1's in the character, if even, parity=0, if odd, parity=1)
  - odd (count the number of 1's in the character, if odd, parity=0, if even, parity=1)
  - none, not calculated and not sent (transmission is one bit shorter).
- **Stop bit:** specifies 1, $1_{1/2}$ or 2 bit times, allows the receptor to collect data and save it to a buffer.

# 8.3 Serial line protocol

## Transmission example:




Total time = 1 + 7 + 1 + 1 =
10 bit / 9600 bps = 1,04 ms

Example: send a Letter 'M' = ASCII '4D' = $1001101_2$ codified 7 bit, with even parity at 9600 bps.

Related terms:

- RS232 (RS: Recommended Specification 232) specifies voltages, pins, connectors…
- USART (Universal Synchronous Asynchronous Receiver Transmitter) a hardware peripheral present in the microcontrollers to implement the protocol.
- RS422, RS423, RS485 evolutions of the protocol to implement differential signaling, multipoint connection, etc.

# 8.3 Serial line protocol

## Typical serial line communication errors:

1. Framing error
   - May occur due to clock synchronization problems (not the same speed between the transmitter and the receiver, or not enough clock precision).
   - Can be detected by the missing stop bit (a 0 is detected in the stop bit time).
2. Receiver overrun
   - May occur when the CPU did not read the received data for a while, or if it receives more data than the data that can be processed.
   - Can be solved by extending the stop bit time.
3. Parity errors
   - May occur due to bit change values, typically by electromagnetic noise. When the receiver checks the bits, the parity bit is not consistent with data bits.

# 8.3 Serial line protocol

## Implementations:

Serial line and other communication buses can be implemented in two ways:

- Bit-banging. Some standard GPIO pins are dedicated to create the communication bus. All control is made by software (set bits, clear bits, delays, etc.).

- Hardware peripheral. Microcontrollers have specialized hardware units (allowing access to pins, implementing the state machines, setting timers, etc.) to create the communication bus. Some control registers are to be set. See the chip pin-out to discover which pins are connected to the peripherals.

# 8.3 Serial line protocol

## Bit-banging example, transmission.

```
// Input vars, data:  8 bits to be sent (starting by the Lsb), bps: (bits per second) transmission speed.

void Transmit_Byte( BYTE data,  float bps)
{
        float Tbit = 1.0 / bps;
        TrisBit = 0;                    // Set the pin as output (suppose TrisBit is defined)
        DataBit = 0;                    // Put the start bit to zero in the DataBit (suppose is defined)
        Delay(Tbit);                    // Waiting for the time of a bit.
        for ( int i = 0; i < 8; i++)
        {
                DataBit = data&01;      // Bit to bit transmission (information).
                Delay(Tbit);            // Waiting for the time of a bit.
                data = data>>1;         // Right shift to get the next bit.
        }
        DataBit = 1;                    // End of the communication (stop bit)
        Delay(Tbit);                    // Waiting for the time of a bit.
        TrisBit = 1;                    // Release the pin (setting as an input).
}
```

# 8.3 Serial line protocol

## Bit-banging example, reception.

```
// Function to receive a byte; bps: (bits per second) transmission speed

BYTE Receive_Byte(float bps)            // Assume the pin is an input.
{
        float Tbit = 1.0 / bps;
        BYTE data = 0;                  // Variable to store the 8-bits of data.
        BYTE state = DataBit;           // Variable to check the state of the communication bus.
        while (state != 0)              // Wait for a start bit. It is a good idea to add a time-out condition!
        {
                state = DataBit;
        }
        Delay(1.5 * Tbit);              // Wait the time of 1.5 bit (start + half of a bit) to get the stable data.
        for ( int i = 0; i < 8; i++)    // Collect 8 bit of data.
        {
                data  = data | (DataBit<<i);    // Reception, shift and or to the collection.
                Delay(Tbit);                    // Waiting for the time of a bit.
        }
        return(data);
}
```

# 8.3 Serial line protocol

## The PIC18F peripheral for serial communications.

PIC18F has **two** USART interfaces. USART stands for Universal Synchronous Asynchronous Receiver Transmitter.

**USART-Related Pins**

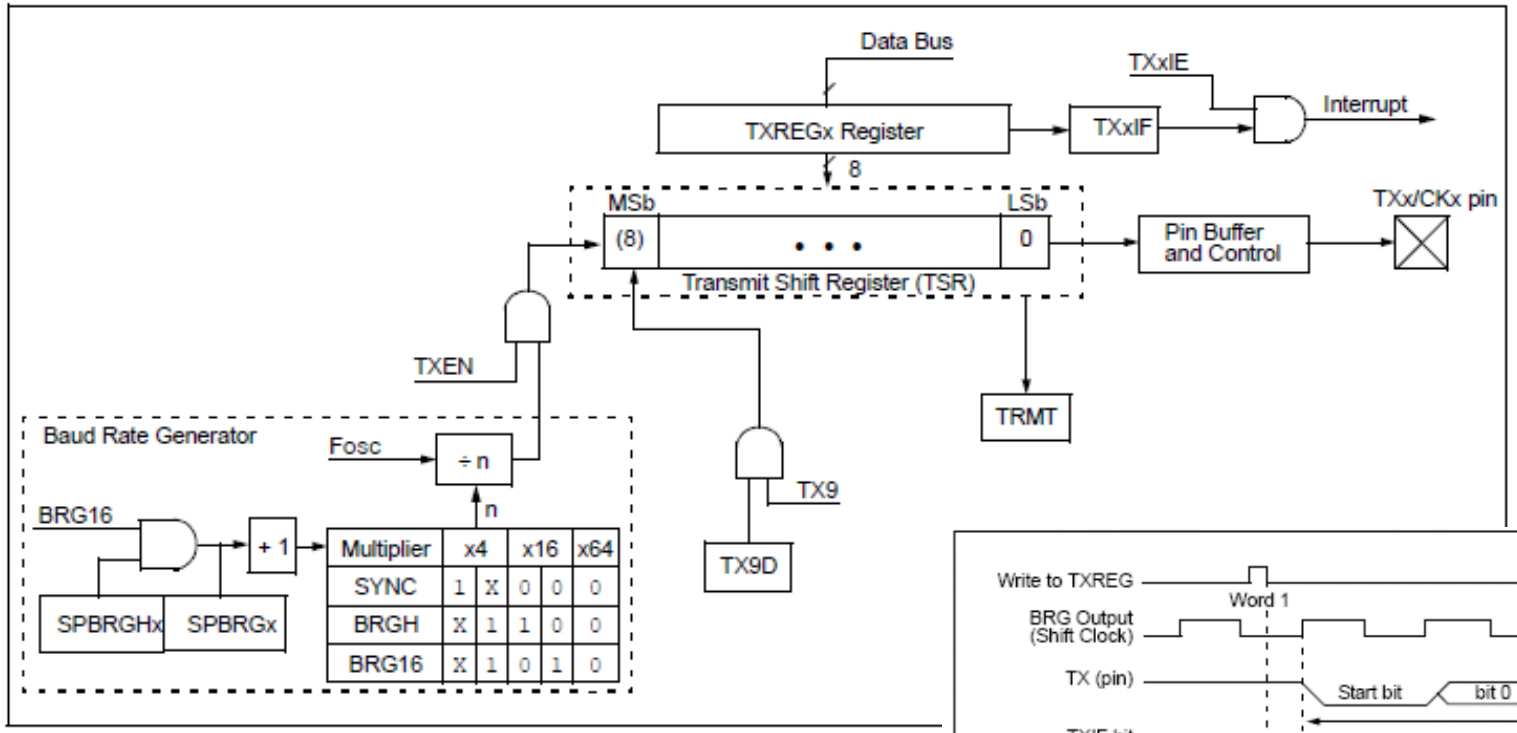
- RC6/TX1/CK1 and RC7/RX1/DT1 (USART1)
- RD6/TX2/CK2 and RD7/RX2/DT2 (USART2)

**USART-Related Registers**

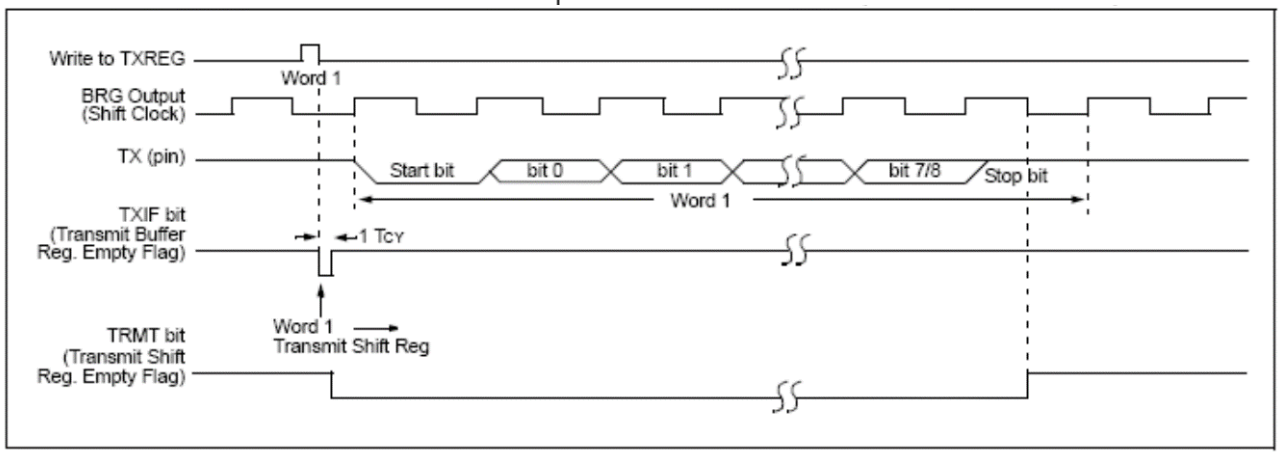
- Transmit status register (TXSTA) - Transmit register (TXREG)
- Receive status register (RCSTA) - Receive register (RCREG)
- Baud rate Control register (BAUDCON)
- Baud rate generator register (SPBRG)

# 8.3 Serial line protocol

## USART: Transmission hardware Block Diagram.

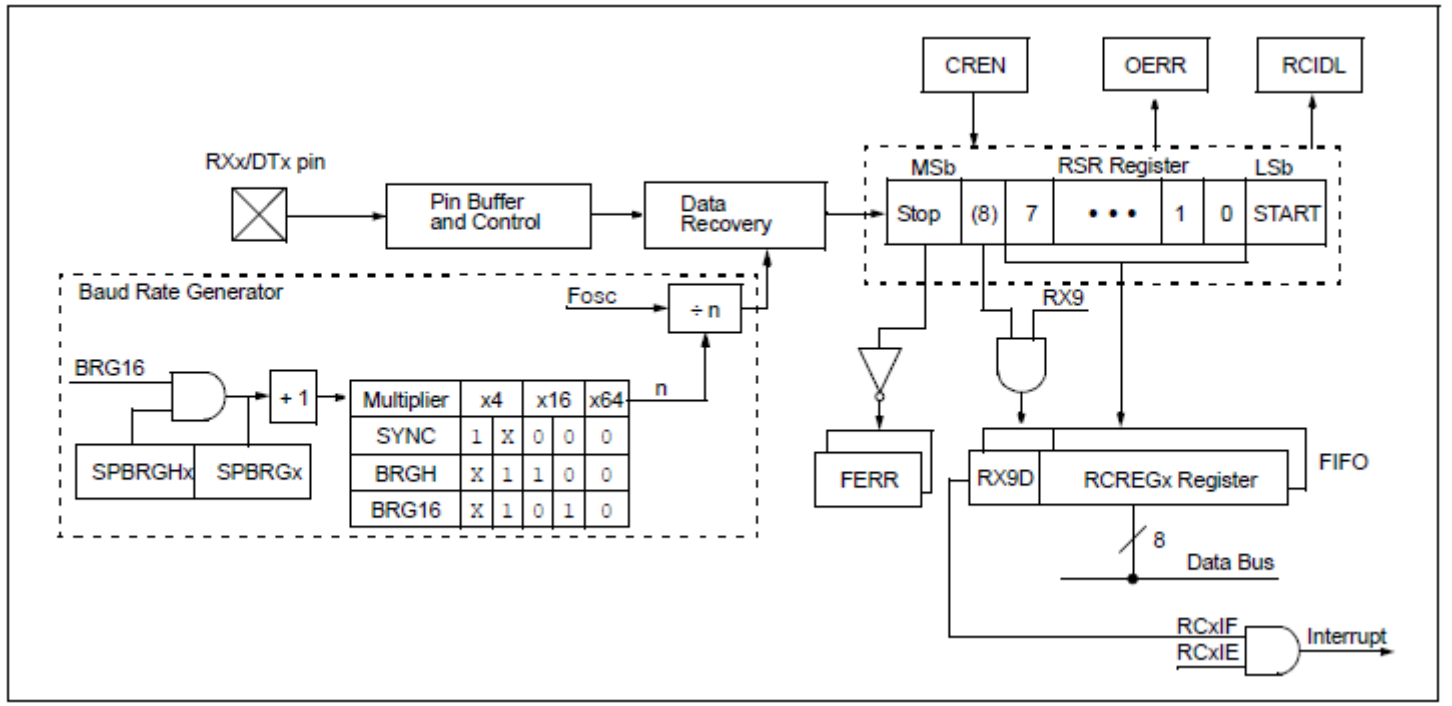



Transmission example.



# 8.3 Serial line protocol

## USART: Reception hardware Block Diagram.



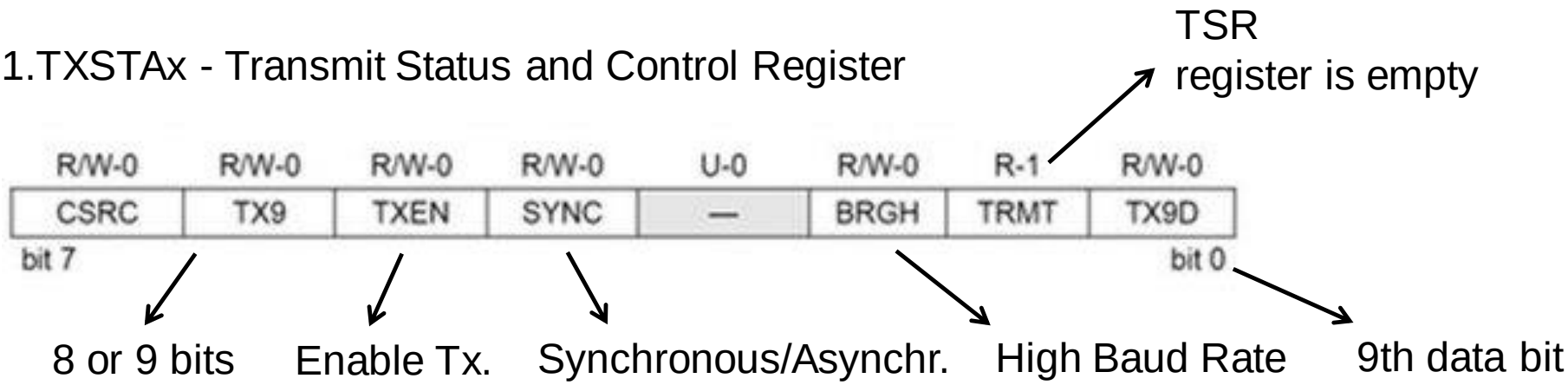

Reception starts upon detection of a START bit

- Transmission integrity is checked testing STOP bit level
- Received data in Shift register is stored in FIFO (One single address, RCREG)
- If three characters are received in a row, FIFO overflows Overrun error
- RCIF is set HIGH until FIFO is empty

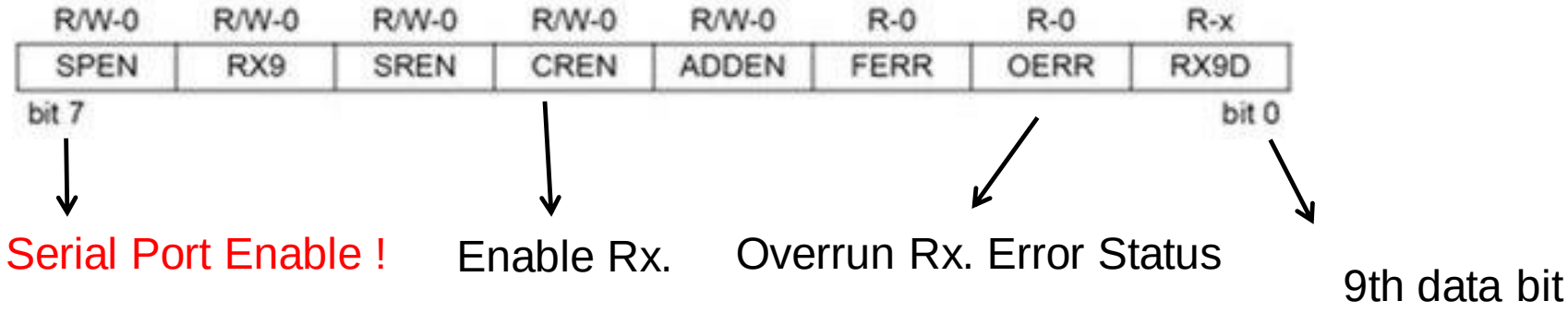
# 8.3 Serial line protocol

## USART: Transmission and Reception control registers.

1.TXSTAx - Transmit Status and Control Register




2. RCSTAx - Receive Status and Control Register



# 8.3 Serial line protocol

## Baud rate generator. The quest for precision.

Registers associated with the Baud Rate Generator System (shaded cells not involved) and resulting formulas.

| Name | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Reset Values on page |
|---|---|---|---|---|---|---|---|---|---|
| TXSTA | CSRC | TX9 | TXEN | SYNC | SENDB | BRGH | TRMT | TX9D | 55 |
| RCSTA | SPEN | RX9 | SREN | CREN | ADDEN | FERR | OERR | RX9D | 55 |
| BAUDCON | ABDOVF | RCIDL | RXDTP | TXCKP | BRG16 | — | WUE | ABDEN | 55 |
| SPBRGH | EUSART Baud Rate Generator Register High Byte | | | | | | | | 55 |
| SPBRG | EUSART Baud Rate Generator Register Low Byte | | | | | | | | 55 |

| Configuration Bits | | | BRG/EUSART Mode | Baud Rate Formula |
|---|---|---|---|---|
| SYNC | BRG16 | BRGH | | |
| 0 | 0 | 0 | 8-bit/Asynchronous | Fosc/[64 (n + 1)] |
| 0 | 0 | 1 | 8-bit/Asynchronous | Fosc/[16 (n + 1)] |
| 0 | 1 | 0 | 16-bit/Asynchronous | |
| 0 | 1 | 1 | 16-bit/Asynchronous | Fosc/[4 (n + 1)] |
| 1 | 0 | x | 8-bit/Synchronous | |
| 1 | 1 | x | 16-bit/Synchronous | |

**Legend:** x = Don't care, n = value of SPBRGH:SPBRG register pair

We try to approximate a number (e.g. 9600) with dividers of Fosc.

# 8.3 Serial line protocol

## Baud rate generator. The quest for precision.

Precalculated values for standard baud-rates, provided by the manufacturer.

Errors accumulate bit by bit, so lower errors are preferable to avoid issues with the framing of the data.

| BAUD RATE (K) | SYNC = 0, BRGH = 0, BRG16 = 0 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Fosc = 40.000 MHz | | | Fosc = 20.000 MHz | | | Fosc = 10.000 MHz | | | Fosc = 8.000 MHz | | |
| | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) |
| 0.3 | — | — | — | — | — | — | — | — | — | — | — | — |
| 1.2 | — | — | — | 1.221 | 1.73 | 255 | 1.202 | 0.16 | 129 | 1.201 | -0.16 | 103 |
| 2.4 | 2.441 | 1.73 | 255 | 2.404 | 0.16 | 129 | 2.404 | 0.16 | 64 | 2.403 | -0.16 | 51 |
| 9.6 | 9.615 | 0.16 | 64 | 9.766 | 1.73 | 31 | 9.766 | 1.73 | 15 | 9.615 | -0.16 | 12 |
| 19.2 | 19.531 | 1.73 | 31 | 19.531 | 1.73 | 15 | 19.531 | 1.73 | 7 | — | — | — |
| 57.6 | 56.818 | -1.36 | 10 | 62.500 | 8.51 | 4 | 52.083 | -9.58 | 2 | — | — | — |
| 115.2 | 125.000 | 8.51 | 4 | 104.167 | -9.58 | 2 | 78.125 | -32.18 | 1 | — | — | — |

| BAUD RATE (K) | SYNC = 0, BRGH = 1, BRG16 = 0 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Fosc = 40.000 MHz | | | Fosc = 20.000 MHz | | | Fosc = 10.000 MHz | | | Fosc = 8.000 MHz | | |
| | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) | Actual Rate (K) | % Error | SPBRG value (decimal) |
| 0.3 | — | — | — | — | — | — | — | — | — | — | — | — |
| 1.2 | — | — | — | — | — | — | — | — | — | — | — | — |
| 2.4 | — | — | — | — | — | — | 2.441 | 1.73 | 255 | 2.403 | -0.16 | 207 |
| 9.6 | 9.766 | 1.73 | 255 | 9.615 | 0.16 | 129 | 9.615 | 0.16 | 64 | 9.615 | -0.16 | 51 |
| 19.2 | 19.231 | 0.16 | 129 | 19.231 | 0.16 | 64 | 19.531 | 1.73 | 31 | 19.230 | -0.16 | 25 |
| 57.6 | 58.140 | 0.94 | 42 | 56.818 | -1.36 | 21 | 56.818 | -1.36 | 10 | 55.555 | 3.55 | 8 |
| 115.2 | 113.636 | -1.36 | 21 | 113.636 | -1.36 | 10 | 125.000 | 8.51 | 4 | — | — | — |

# 8.3 Serial line protocol

Example. Baud rate calculation for a $F_{osc}$=20MHz, high speed asynchronous mode, desired speed 9600bps.

If BRGH = 1, the value to be written into the SPBRG register is:

$$SPBRG = 20 \times 10^6 \div (16 \times 9600) - 1 = 130 - 1 = 129$$

The actual baud rate: $20{,}000{,}000 \div (16 \times 130) = 9615.4$

So, the resultant error rate is $(9615.4 - 9600) \div 9600 \times 100\% = 0.16\%$.

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

$$SPBRG = 20{,}000{,}000 \div (64 \times 9600) - 1 = 31$$

The actual baud rate is: $20000000 \div (64 \times 32) = 9765.6$

And the resultant error rate is $(9765.6 - 9600) \div 9600 \times 100\% = 1.7\%$.

# 8.3 Serial line protocol

## Code snippets

/* Configure USART1 to transmit and receive data in asynchronous mode using 8-bit data format, disable interrupt, set baud rate to 9600. Assume the frequency of the crystal oscillator is 16 MHz */

```
void usart1_Init(void)
{
        TXSTA1          = 0x24;   /* USART Configuration Register */
        RCSTA1          = 0x10;   /* Enable reception */
        SPBRG1  = 103;    /* Set de Baud rate */
        TRISCbits.RC7   = 1;      /* configure RX1 pin for input */
        TRISCbits.RC6   = 1;      /* configure TX1 pin for output */
        PIE1bits.TXIE   = 0;      /* disable transmit interrupt */
        RCSTA1bits.SPEN = 1;      /* enable USART port */
}
```

# 8.3 Serial line protocol

## Code snippets

```
/* Transmit a char using the USART1. Wait the transmission to be idle */

void putc_usart1 (char xc);
{
    while (! TXSTA1.TRMT);
    TXREG1 = xc;
}

/* Transmit a null-terminated string using the previous function */

void puts_usart1 (unsigned rom char *cptr)
{
        while(*cptr)
                putc_usart1 (*cptr++);
}
```

# 8.3 Serial line protocol

## Code snippets

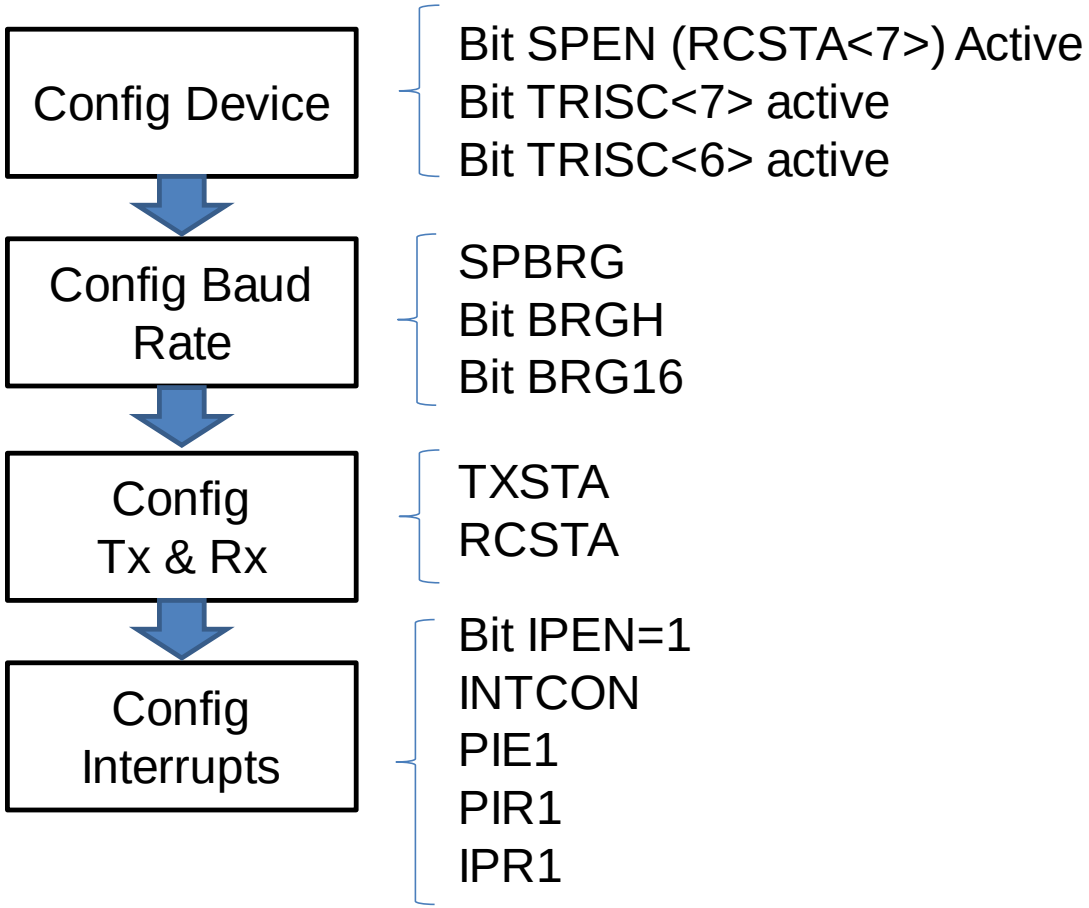
/* Receive a char using the USART1. Wait for the IF to be set.
Receive a String Terminated with the CR (Carry Return) character (0x3d) */

```
unsigned char getc_usart1 (void)
{
        while (! PIR1bits.RCIF);
        return RCREG1;                  /*  RCIF clears automatically  */
}

void gets_usart1 (char *ptr)
{
        while (1)
        {
                char xx = getc_usart1( );       /* read a character */
                if (xx == 0x3D ) {              /* is it a carriage return? */
                        *ptr = '\0';            /* terminate the string with a NULL */
                        return; }
                ptr++ = xx;                     /* store the received character in the buffer */
        }
}
```

# 8.3 Serial line protocol

## USART: Recommended initialization procedure.



# 8.3 Serial line protocol

## Flow Control of USART in Asynchronous Mode.

- In some circumstances, the software cannot read the received data and needs to inform the transmitter to stop.
- In some other situation, the transmitter may need to be told to suspend transmission because the receiver is too busy to read data.
- Both situations are handled by **flow control**.

- There are two flow control methods: **hardware** and **XON/XOFF characters**.
- XON and XOFF are two standard ASCII characters.
- The ASCII code for XON and XOFF are 0x11 and 0x13, respectively.
- Whenever a microcontroller cannot handle the incoming data, it sends the XOFF to the transmitter.
- When the microcontroller can handle incoming characters, it sends out XON character.

# 8.4 The RS232 Standard

## The EIA232 Standard for communications (specifies hardware connectors and electronics)

Developed in 1960, RS-232 (Recommended Standard 232) is a standard for serial binary single-ended data and control signals connecting between a DTE (Data Terminal Equipment) and a DCE (Data Circuit-terminating Equipment or Modem)

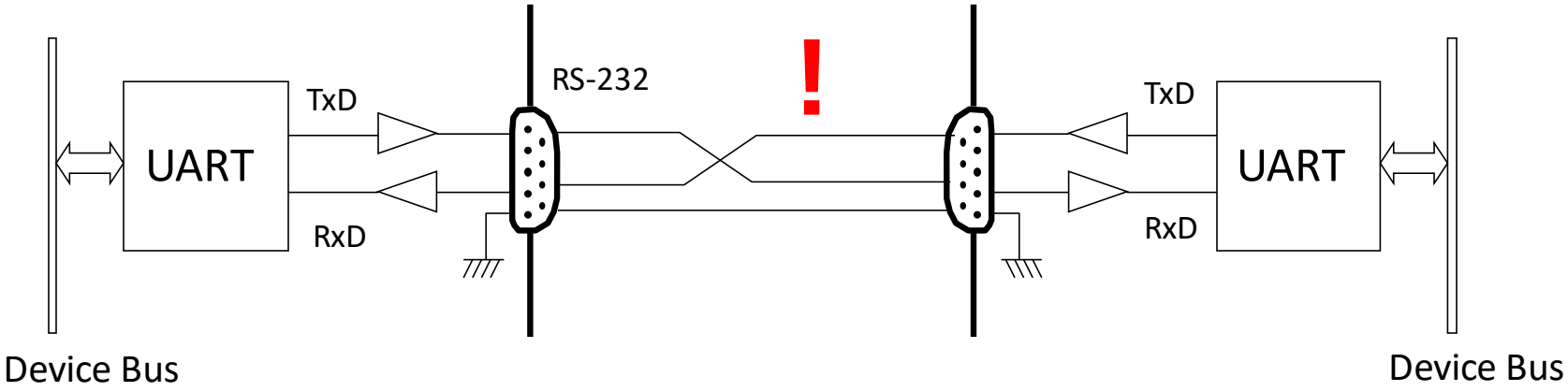
The standard requires the transmitter to use +15 V and −15 V, but requires the receiver to distinguish voltages as low as +3 V and -3 V (it supports wires really long).

It defines common asynchronous speeds: 200, 2.400, 4.800, 9.600, 19.200, 57.600, 115.200 bauds (for a binary two-level signal transmissions, one baud is equal to one bit per second).

It defines A male DB-9 connector for a serial port, including Data Pins (Receive and Transmit), Ground, and **Flow Control** and **Handshaking** pins.

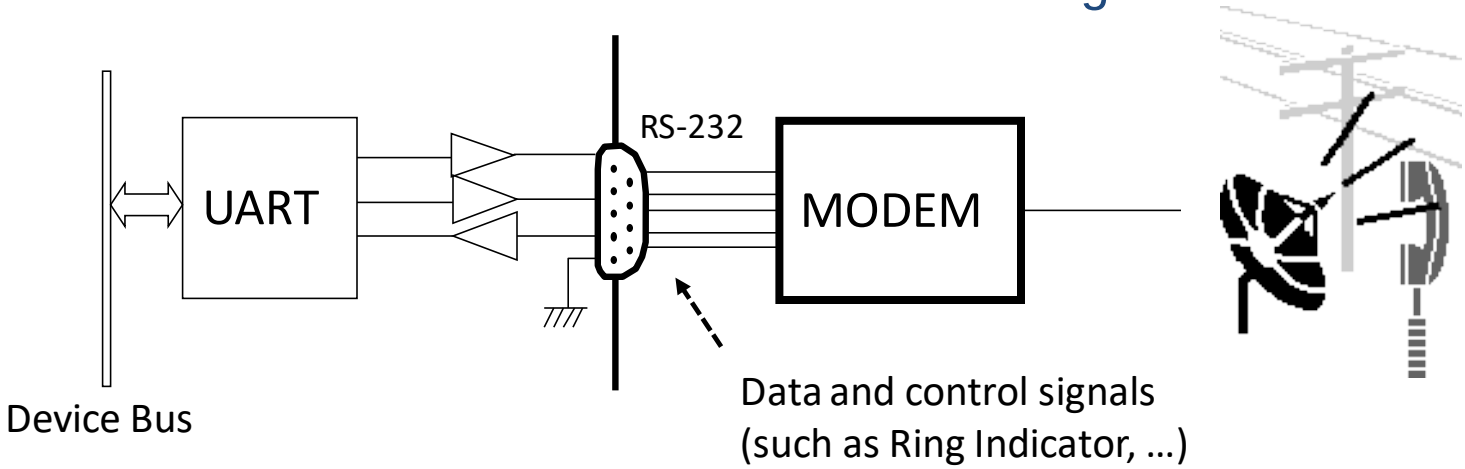
# 8.4 The RS232 Standard

## Microcontroller to Microcontroller connection using RS232.




! Null-Modem Connection. TxD and RxD are cross connected!!

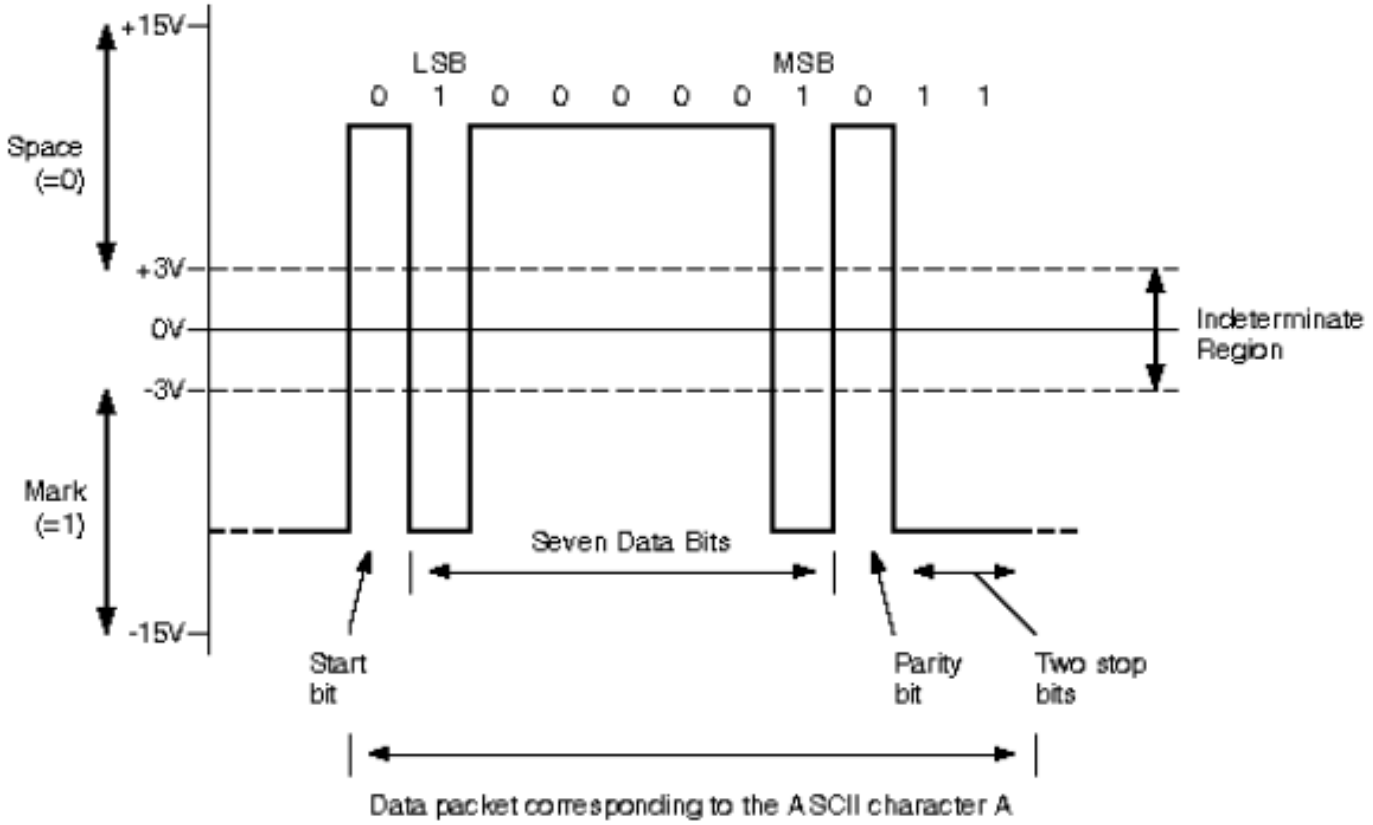
## Microcontroller to Modem connection using RS232.




Modem system Connection. Some specialized systems (antennas, GPS, satellite) use this configuration.

# 8.4 The RS232 Standard

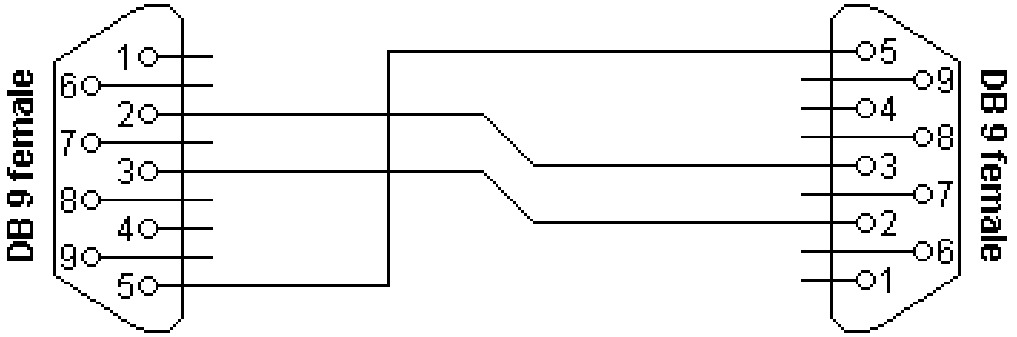
## Electrical and Logical Specifications




| Name | Direction | Function | Comment |
|---|---|---|---|
| TD | OUT | Transmitted Data | Data Pair |
| RD | IN | Received Data | |
| RTS | OUT | Request to Send | Handshake |
| CTS | IN | Clear to Send | |
| DTR | OUT | Data Term Ready | Handshake |
| DSR | IN | Data Set Ready | |
| DCD | IN | Carrier Detect | Inform DTE |
| RI | IN | Ring Indicator | |

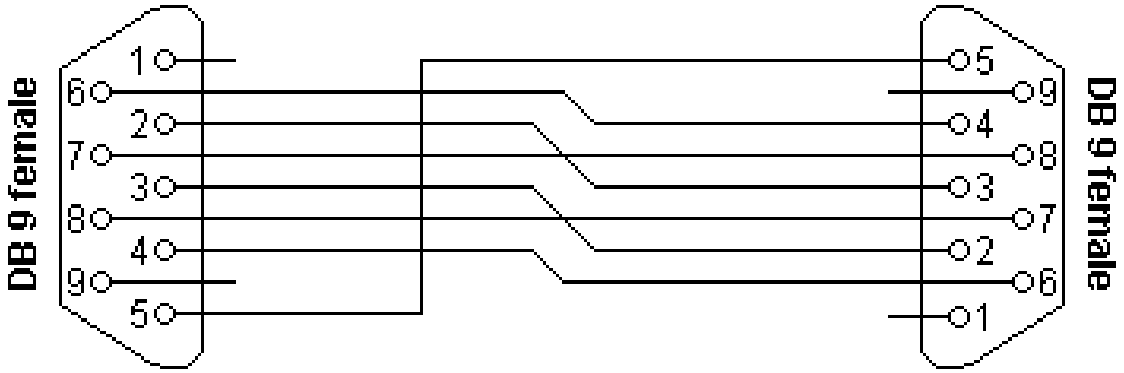
# 8.4 The RS232 Standard

## Example connections




Null modem connection without Handshaking

| Connector 1 | Connector 2 | Function | |
|---|---|---|---|
| 2 | 3 | Rx | Tx |
| 3 | 2 | Tx | Rx |
| 5 | 5 | Signal Ground | |




Null modem connection with full Handshaking

| Connector 1 | Connector 2 | Function | |
|---|---|---|---|
| 2 | 3 | Rx | Tx |
| 3 | 2 | Tx | Rx |
| 4 | 6 | DTR | DSR |
| 5 | 5 | Signal ground | |
| 6 | 4 | DSR | DTR |
| 7 | 8 | RTS | CTS |
| 8 | 7 | CTS | RTS |

# 8.5 SPI and I2C buses

SPI stands for Serial Peripheral Interface. I2C stands for Inter-integrated Circuit.

- Both buses are designed to connect devices that lay on the same PCB (Printed Circuit Board) or are really near (centimeters). A difference with Serial Line + RS232 is that Serial Line can be used to send/receive data up to 15 meters.
- SPI and I2C are commonly used to connect a High Performance Processor (e.g. Smartphones) to all its peripherals (accelerometers, touchscreen, sensors…).

- Both buses are able to connect several devices. SPI is a Master-Slave bus, I2C allows some multi-master modes (Improved I2C = I3C implementations).
- Both buses are synchronous, with a clock line provided by the master. SPI is full-duplex and I2C is half duplex.
- Microcontrollers like PIC18F have only one peripheral able to manage both buses.

- SPI uses physical lines in order to select the slave devices to communicate. I2C uses an addressing system and the address of the selected device is sent through the data lines.

# 8.5 SPI and I2C buses

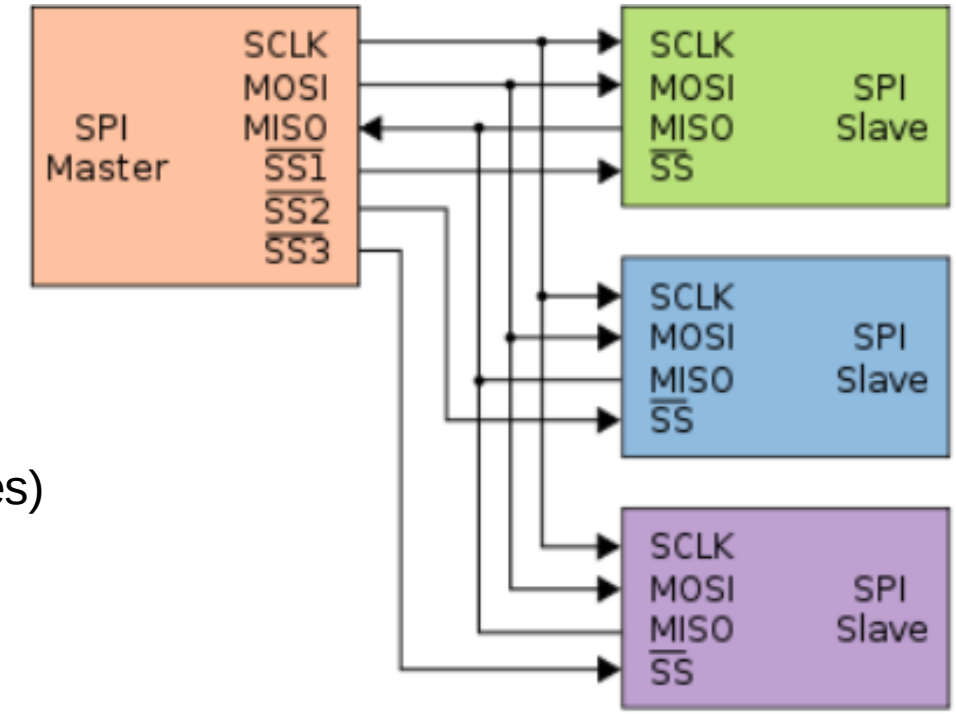
## SPI Bus.

The **Serial Peripheral Interface Bus** or **SPI** bus is a synchronous serial data link standard that operates in full duplex mode (there are two dedicated data wires). Devices communicate in master/slave mode where the master device initiates the data frame. Multiple slave devices are allowed with individual slave select (chip select) lines.

SPI connections:

- SCLK: bus clock provided by the master.
- MOSI (sometimes DO): Master Out Slave In
- MISO (sometimes DI): Master in Slave Out
- SS: Slave Select (one per slave).

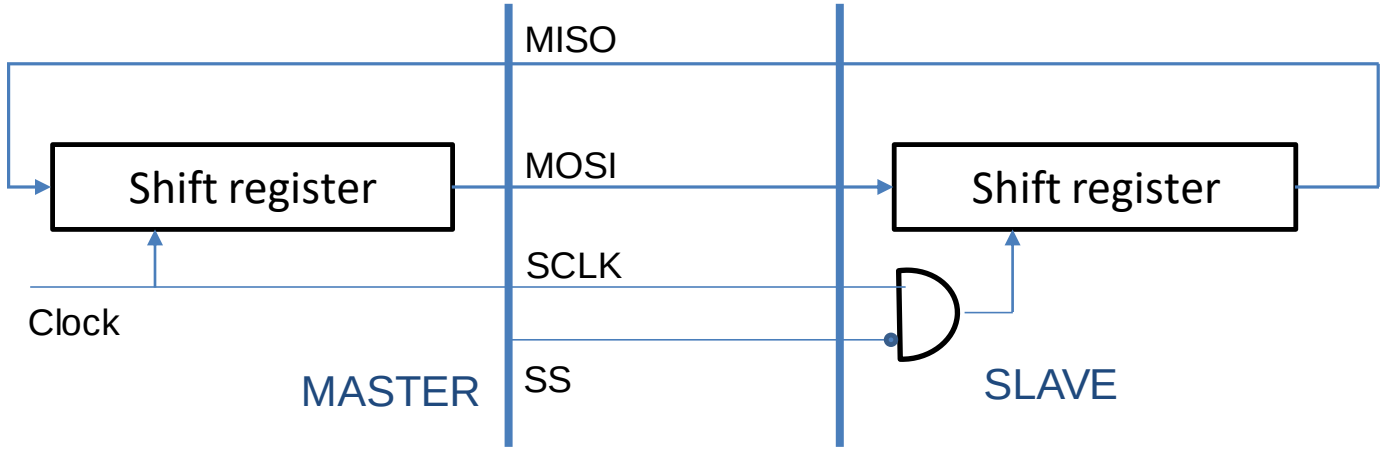
(Ground is assumed to be the same for all devices)



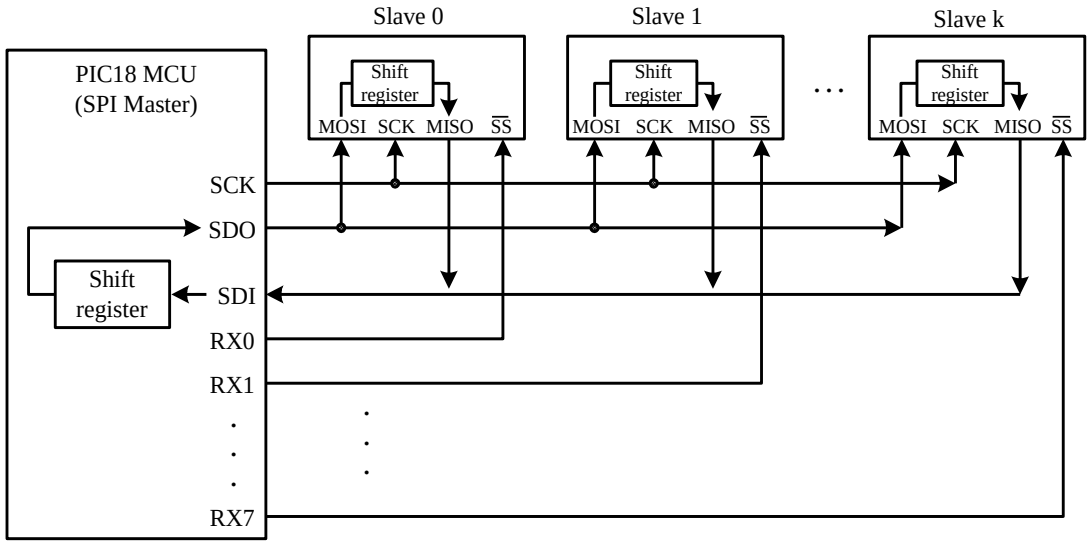
# 8.5 SPI and I2C buses

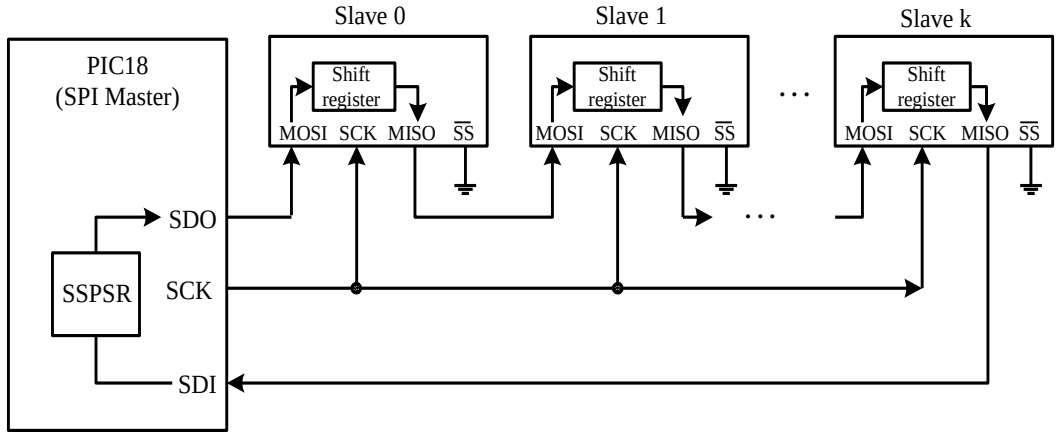
## SPI Bus.

SPI is based in the interconnection of two shift registers. At least (if only one register is to be read), 8 clocks will be needed to exchange data. Master rules all the process.




## SPI Topologies. Bus-star and Ring.



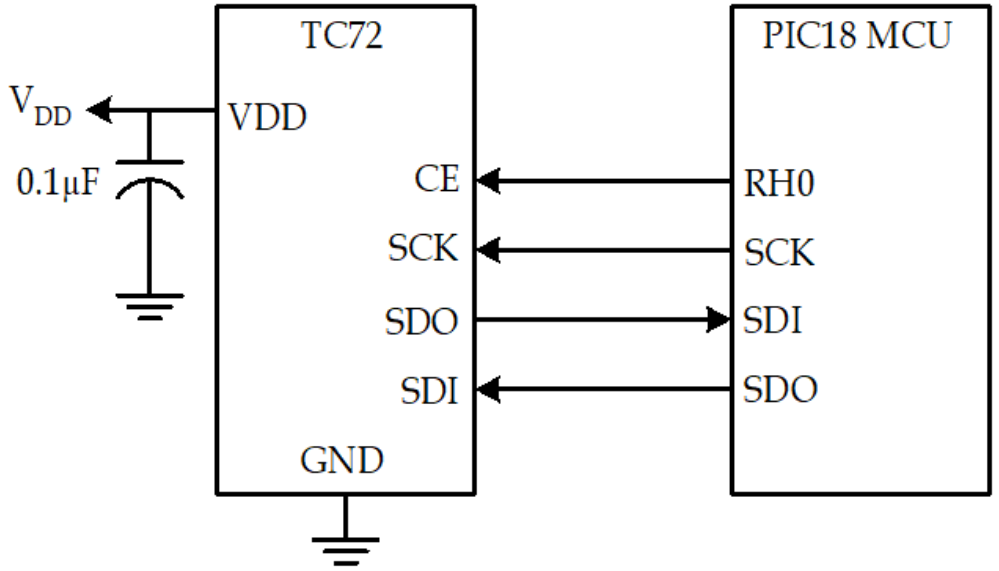



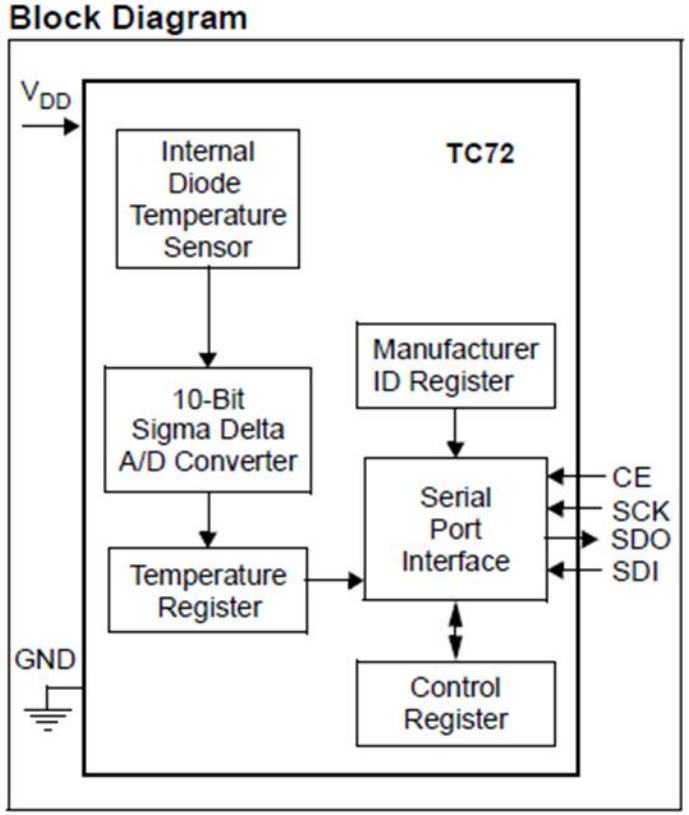

k wires are saved. Time is multiplied by (k-1).
Risk of failure if one Slave fails.

# 8.5 SPI and I2C buses

## SPI Bus example 1.

Connection of a digital temperature sensor through SPI (Microchip TC72).



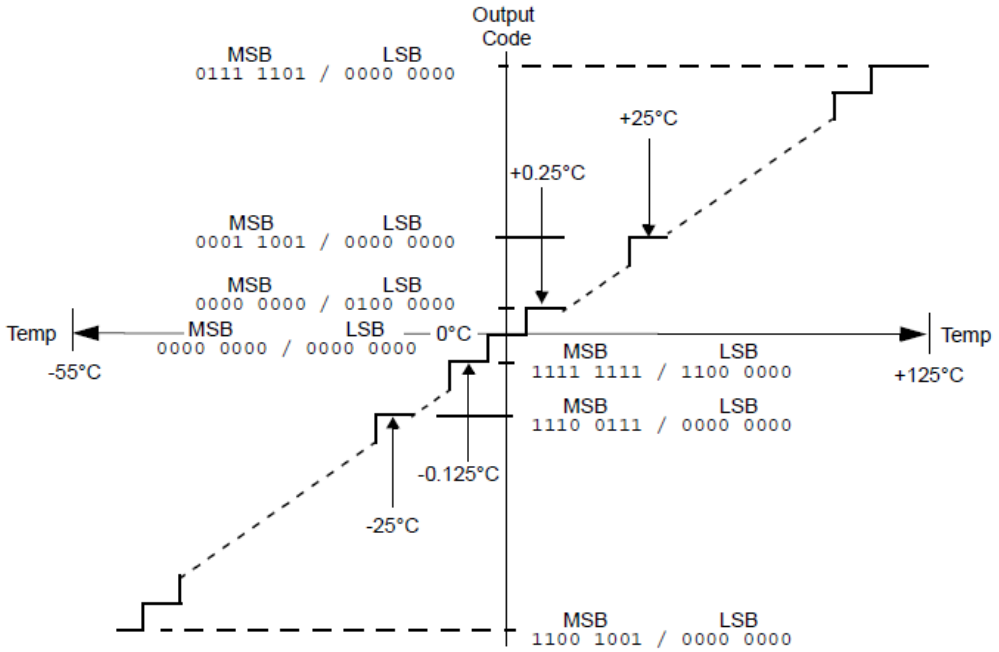




This device has internal registers (ID, Control and Temp) that can be read through SPI.

# 8.5 SPI and I2C buses

## SPI Bus example 1.

Connection of a digital temperature sensor through SPI (Microchip TC72).




**Note:** The ADC converter is scaled from -128°C to -127°C, but the operating range of the TC72 is specified from -55°C to +125°C.

### Typical Applications

- Personal Computers and Servers
- Hard Disk Drives and Other PC Peripherals
- Entertainment Systems
- Office Equipment
- Datacom Equipment
- Mobile Phones
- General Purpose Temperature Monitoring

**TABLE 3-2: TEMPERATURE REGISTER**

| D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | Address/ Register |
|---|---|---|---|---|---|---|---|---|
| Sign | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^3$ | $2^1$ | $2^0$ | 02H Temp. MSB |
| $2^{-1}$ | $2^{-2}$ | 0 | 0 | 0 | 0 | 0 | 0 | 01H Temp. LSB |

**Example:**

Temperature = 41.5°C

MSB Temperature Register= 00101001b
$= 2^5 + 2^3 + 2^0$
$= 32 + 8 + 1 = 41$

LSB Temperature Register = 10000000b = $2^{-1} = 0.5$

**TABLE 3-1: TC72 TEMPERATURE OUTPUT DATA**

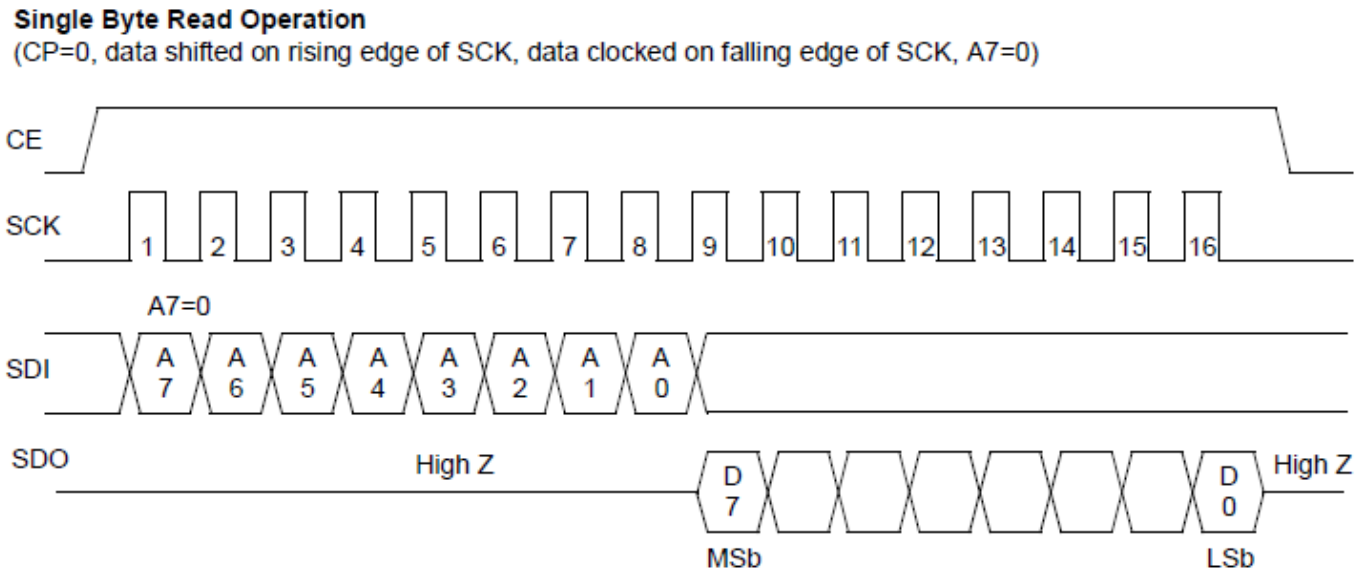
| Temperature | Binary MSB / LSB | Hex |
|---|---|---|
| +125°C | 0111 1101/0000 0000 | 7D00 |
| +25°C | 0001 1001/0000 0000 | 1900 |
| +0.5°C | 0000 0000/1000 0000 | 0080 |
| +0.25°C | 0000 0000/0100 0000 | 0040 |
| 0°C | 0000 0000/0000 0000 | 0000 |
| -0.25°C | 1111 1111/1100 0000 | FFC0 |
| -25°C | 1110 0111/0000 0000 | E700 |
| -55°C | 1100 1001/0000 0000 | C900 |

Temperature will be read in two bytes (MSB and LSB), 10-bit 2-Complement.

# 8.5 SPI and I2C buses

## SPI Bus example 1.

Connection of a digital temperature sensor through SPI (Microchip TC72).

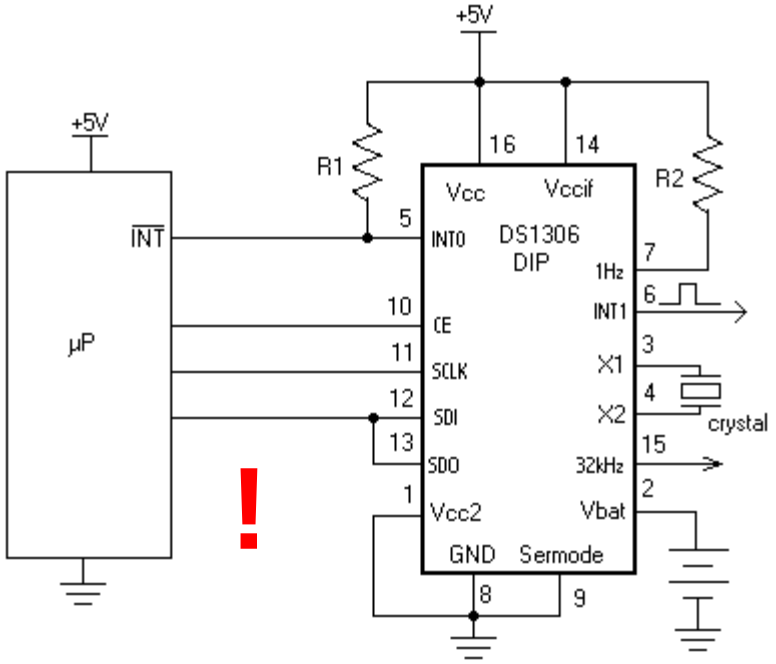



Example of reading of one of the internal registers. Masters set the SS and generates 16 clocks. In the first 8 the address of the register is sent, in the last 8, the data is received.

# 8.5 SPI and I2C buses

## SPI Bus example 2.

Connection of a Real-Time Clock to a microcontroller.





DS1306 RTC

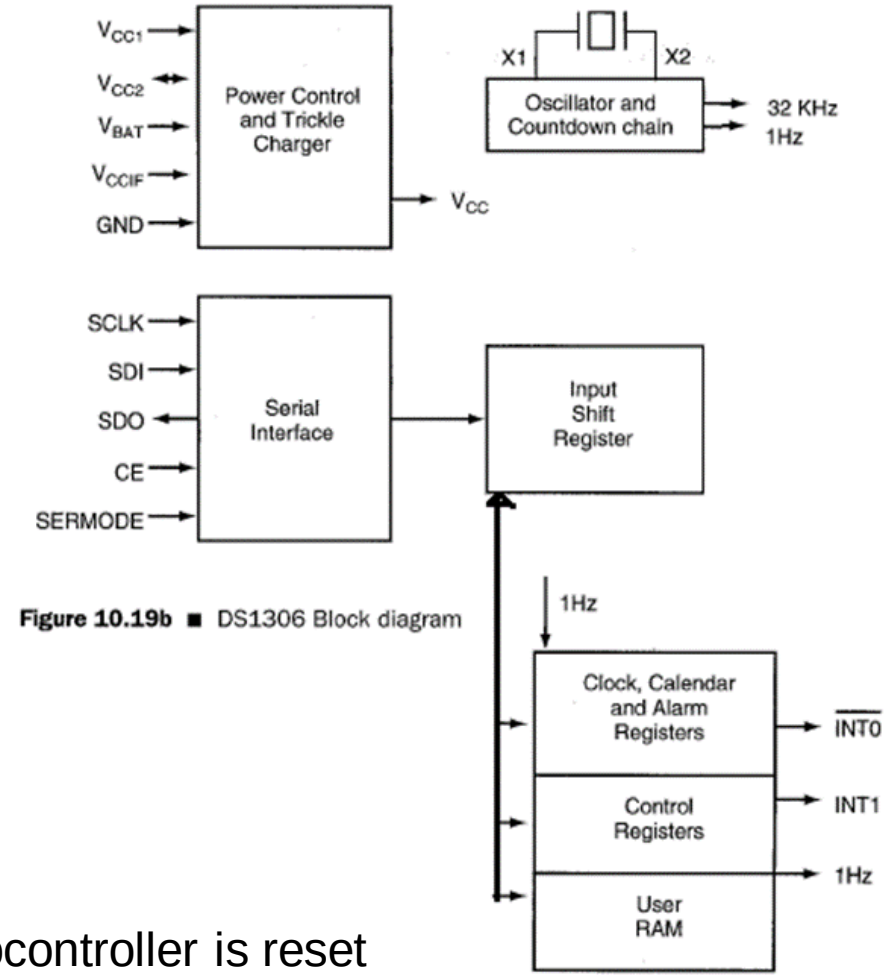



**Figure 10.19b** ■ DS1306 Block diagram

Real-Time clocks have a battery to keep the time when the microcontroller is reset or unpowered. They offer a good precision for time references.

# 8.5 SPI and I2C buses

## SPI Bus example 2.

Connection of a Real-Time Clock to a microcontroller (DS1306 RTC)

Register map of the RTC. It includes Y/M/D, H/M/S registers and two programable alarms. Note there are different addresses for Reading and Writing the registers.

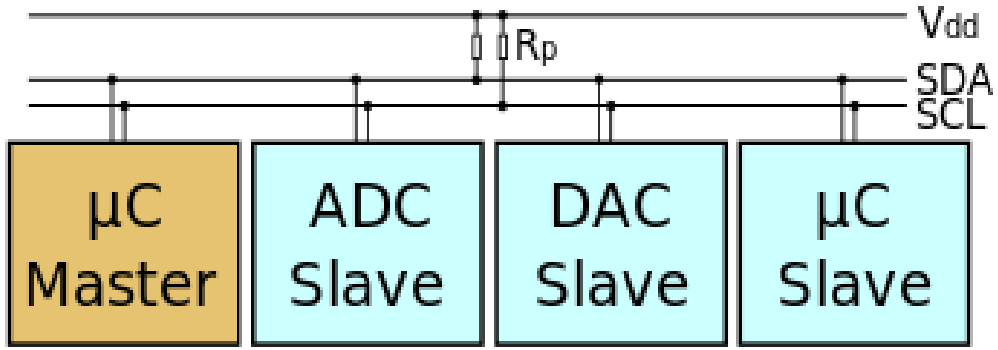
Alarms will be sent through the INT signal to the microcontroller.

| Hex address | | Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 | Range |
|---|---|---|---|---|---|---|---|---|---|---|
| Read | Write | | | | | | | | | |
| 0x00 | 0x80 | 0 | 10 Sec | | | Sec | | | | 00-59 |
| 0x01 | 0x81 | 0 | 10 Min | | | Min | | | | 00-59 |
| 0x02 | 0x82 | 0 | 12 | P | 10-HR | Hours | | | | 01-12 + P/A |
| | | | | A | | | | | | |
| | | | 24 | 10 | | | | | | 00-23 |
| 0x03 | 0x83 | 0 | 0 | 0 | 0 | 0 | Day | | | 01-07 |
| 0x04 | 0x84 | 0 | 0 | 10-Date | | Date | | | | 1-31 |
| 0x05 | 0x85 | 0 | 0 | 10-Month | | Month | | | | 01-12 |
| 0x06 | 0x86 | 10-Year | | | | Year | | | | 00-99 |
| 0x07 | 0x87 | M | 10-Sec Alarm 0 | | | Sec Alarm 0 | | | | 00-59 |
| 0x08 | 0x88 | M | 10-Min Alarm 0 | | | Min Alarm 0 | | | | 00-59 |
| 0x09 | 0x89 | M | 12 | P | 10-HR | Hour Alarm 0 | | | | 01-12 + P/A |
| | | | | A | | | | | | |
| | | | 24 | 10 | | | | | | 00-23 |
| 0x0A | 0x8A | M | 0 | 0 | 0 | 0 | Day Alarm 0 | | | 01-07 |
| 0x0B | 0x8B | M | 10-Sec Alarm 1 | | | Sec Alarm 1 | | | | 00-59 |
| 0x0C | 0x8C | M | 10-Min Alarm 1 | | | Min Alarm 1 | | | | 00-59 |
| 0x0D | 0x8D | M | 12 | P | 10-HR | Hour Alarm 1 | | | | 01-12 + P/A |
| | | | | A | | | | | | |
| | | | 24 | 10 | | | | | | 00-23 |
| 0x0E | 0x8E | M | 0 | 0 | 0 | 0 | Day Alarm 1 | | | 01-07 |
| — | | | | | | | | | | |
| 0x0F | 0x8F | Control Register | | | | | | | | — |
| 0x10 | 0x90 | Status Register | | | | | | | | — |
| 0x11 | 0x91 | Trickle Charger Register | | | | | | | | — |
| 0x12-1F | 0x92-9F | Reserve | | | | | | | | — |
| 0x20-7F | 0xA0-FF | 96-Bytes User RAM | | | | | | | | — |

# 8.5 SPI and I2C buses

## I2C Bus.

I2C, I²C *Inter-Integrated Circuit Bus* is a serial synchronous half-duplex bus (1992)



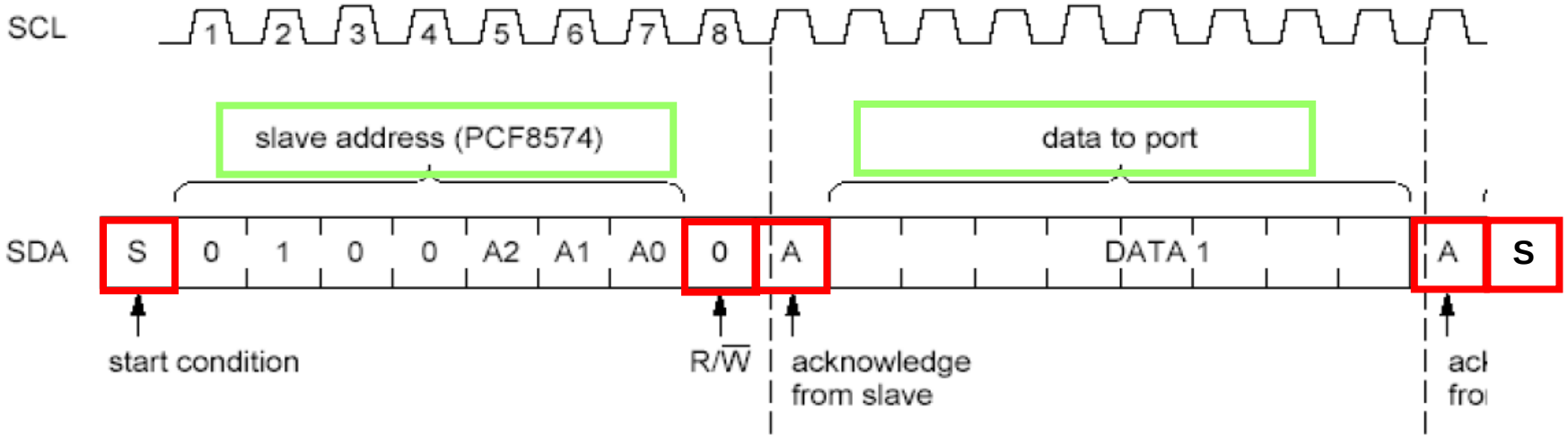

Only two signal lines SDA and SCL plus supply voltage and ground are required to be connected. SDA (Signal Data-Address) and SCL (Signal Clock) are connected to pull-ups to keep a default '1'.

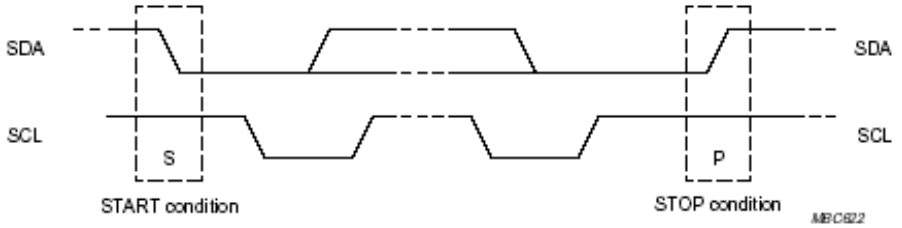
Common I²C bus speeds are the 400kbit/s fast mode, the100 kbit/s *standard mode* and the 10 kbit/s *low-speed mode*, but arbitrarily low clock frequencies are also allowed.

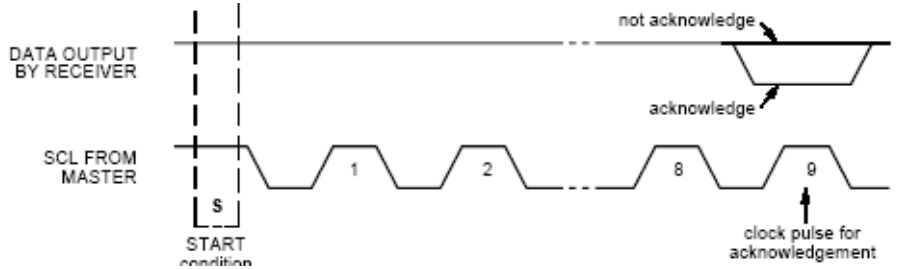
# 8.5 SPI and I2C buses

## I2C Bus.

I2C operates in several steps. (1) A start bit is sent. (2) The address of the device (7 bit) is sent. (3) the desired operation (R/W) is set. (4) An acknowledge from the device (must be a zero) is expected. If not received, operation is aborted. (5) Data (8 bits) is sent or received. (6) Acknowledge for Data is expected. (7) a Stop bit is set.



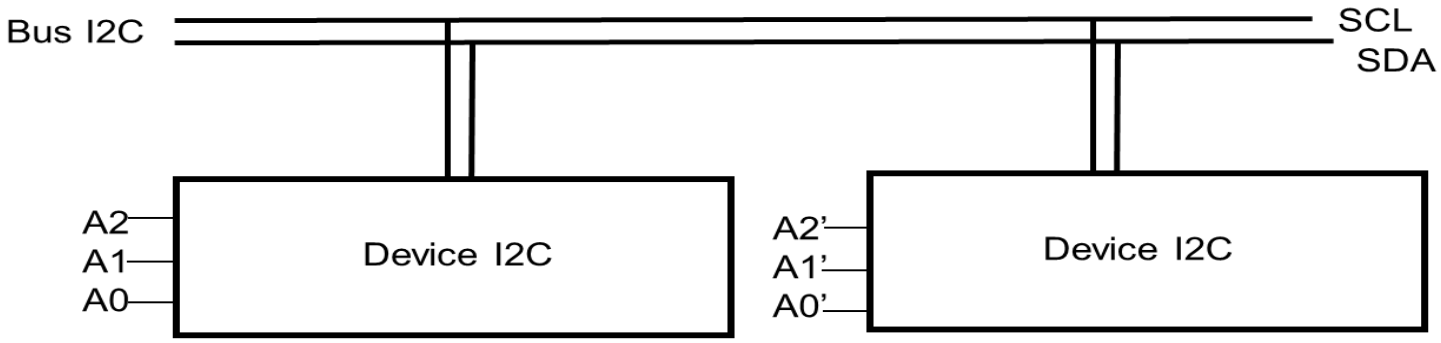



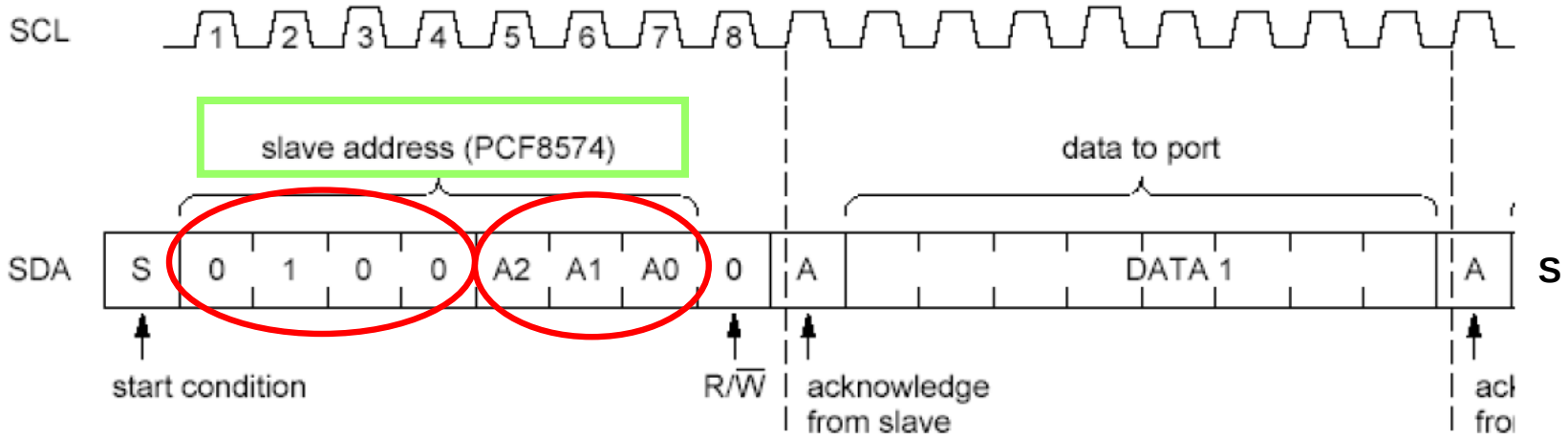
# 8.5 SPI and I2C buses

## I2C Bus.

Devices can have some pins available in order to hardwire the last bits of the address.



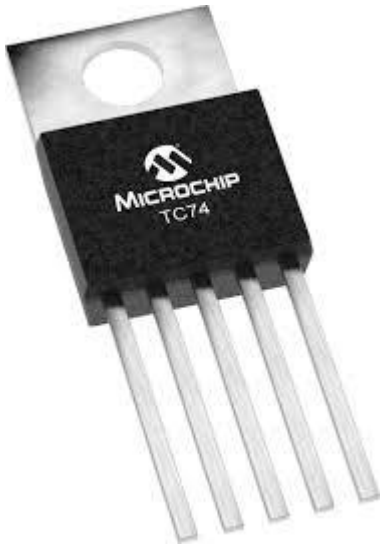

This allows us to connect different instances of the same device.

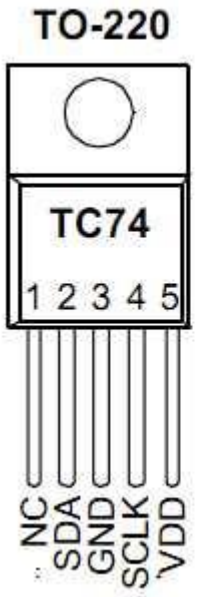


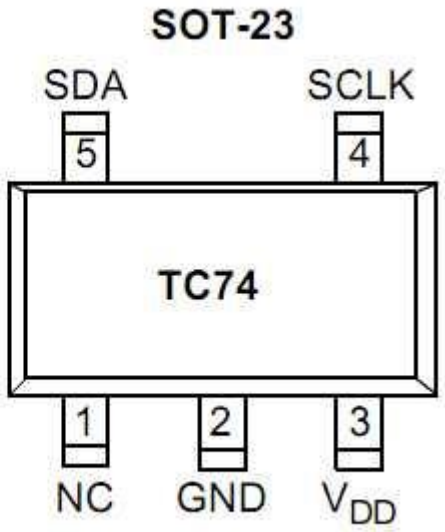
# 8.5 SPI and I2C buses

## I2C Example.

Using a microchip TC74 temperature sensor.




### 3.1.2 SMBUS/I²C SLAVE ADDRESS

The TC74 is internally programmed to have a default SMBus/I²C address value of `1001 101b`. Seven other addresses are available by custom order (contact Microchip Technology Inc.

# 8.5 SPI and I2C buses

## I2C Example.

Using a microchip TC74 temperature sensor.

**Write Byte Format**

| S | Address | WR | ACK | Command | ACK | Data | ACK | P |
|---|---|---|---|---|---|---|---|---|
| | 7 Bits | | | 8 Bits | | 8 Bits | | |

Slave Address

Command Byte: selects which register you are writing to.

Data Byte: data goes into the register set by the command byte.

**Read Byte Format**

| S | Address | WR | ACK | Command | ACK | S | Address | RD | ACK | Data | NACK | P |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 7 Bits | | | 8 Bits | | | 7 Bits | | | 8 Bits | | |

Slave Address

Command Byte: selects which register you are reading from.

Slave Address: repeated due to change in data-flow direction.

Data Byte: reads from the register set by the command byte.

**Receive Byte Format**

| S | Address | RD | ACK | Data | NACK | P |
|---|---|---|---|---|---|---|
| | 7 Bits | | | 8 Bits | | |

S = START Condition
P = STOP Condition
Shaded = Slave Transmission

Data Byte: reads data from the register commanded by the last Read Byte or Write Byte transmission.

# 8.5 SPI and I2C buses

## I2C Example.

Using a microchip TC74 temperature sensor.

**TABLE 4-1: COMMAND BYTE DESCRIPTION (SMBUS/I$^2$C READ_BYTE AND WRITE_BYTE)**

| Command | Code | Function |
|---|---|---|
| RTR | 00h | Read Temperature (TEMP) |
| RWCR | 01h | Read/Write Configuration (CONFIG) |

**TABLE 4-2: CONFIGURATION REGISTER (CONFIG); 8 BITS, READ/ WRITE)**

| Bit | POR | Function | Type | Operation |
|---|---|---|---|---|
| D[7] | 0 | STANDBY Switch | Read/ Write | 1 = standby, 0 = normal |
| D[6] | 0 | Data Ready * | Read Only | 1 = ready 0 = not ready |
| D[5]- D[0] | 0 | Reserved - Always returns zero when read | N/A | N/A |

**Note 1:** *DATA_RDY bit RESET at power-up and SHDN enable.

# 8.5 SPI and I2C buses

## I2C Example.

Using a microchip TC74 temperature sensor.

**TABLE 4-1: COMMAND BYTE DESCRIPTION (SMBUS/I$^2$C READ_BYTE AND WRITE_BYTE)**

| Command | Code | Function |
|---|---|---|
| RTR | 00h | Read Temperature (TEMP) |
| RWCR | 01h | Read/Write Configuration (CONFIG) |

**TABLE 4-2: CONFIGURATION REGISTER (CONFIG); 8 BITS, READ/ WRITE)**

| Bit | POR | Function | Type | Operation |
|---|---|---|---|---|
| D[7] | 0 | STANDBY Switch | Read/ Write | 1 = standby, 0 = normal |
| D[6] | 0 | Data Ready * | Read Only | 1 = ready 0 = not ready |
| D[5]- D[0] | 0 | Reserved - Always returns zero when read | N/A | N/A |

**Note 1:** *DATA_RDY bit RESET at power-up and SHDN enable.

# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

MSSP peripheral has two modes of operation:

1. Serial peripheral interface (SPI)
2. Inter-integrated circuit ($I^2C$)

Can be used to interface with serial EEPROM, shift registers, display drivers, A/D converters, D/A converters, digital temperature sensors, time-of-day chips, etc.
Devices are divided into the master and slaves in a system that uses either the SPI or $I^2C$ protocol to exchange data.

The SPI and $I^2C$ module share the same signal pins and cannot to be active at the same time.
Three pins are used by this module:

1. Serial data out (SDO)—RC5/SDO
2. Serial data in (SDI)—RC4/SDI/SDA
3. Serial clock (SCK)—RC3/SCK/SCL

A fourth signal pin, RA5/SS (Slave Select), may be used in slave mode

# 8.5 SPI and I2C buses

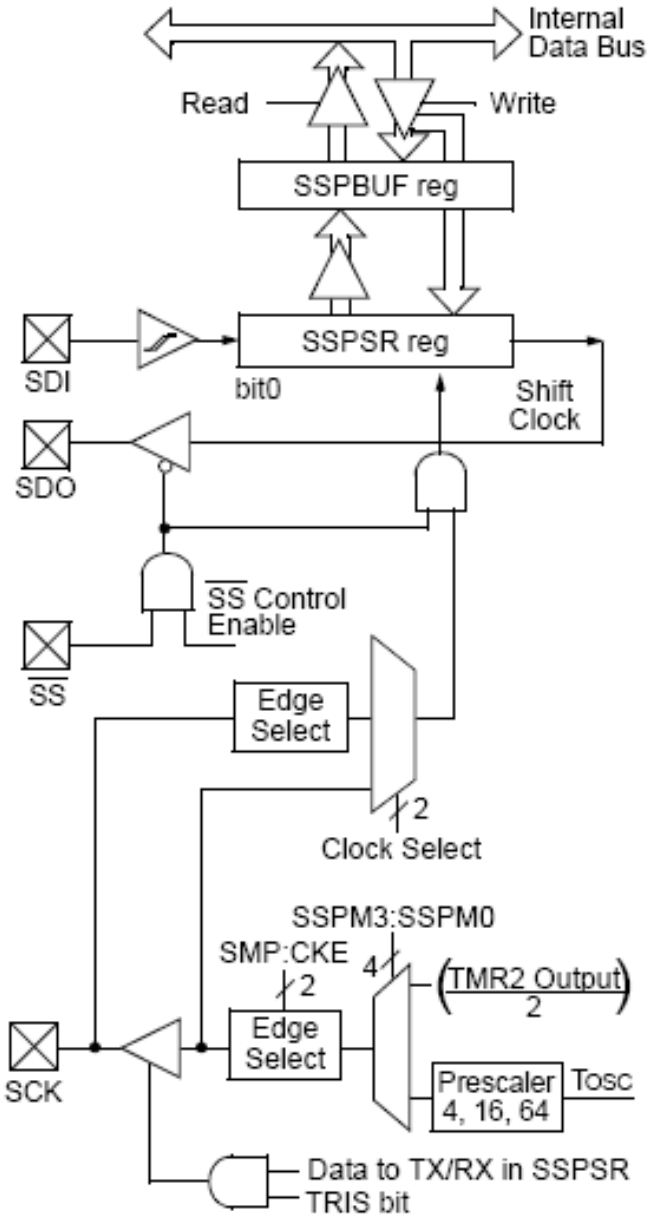
## The PIC18 MSSP Module

**SPI Mode** (Master or Slave) diagram block.

- Eight bits of data are exchanged synchronously in one operation.
- In slave mode, all four signals are used.
- In master mode, the SS pin is not needed (other GPIO can be programmed to select Slaves).
- Registers for SPI mode operation:

  1. MSSP control register 1 (SSPCON1)
  2. MSSP status register (SSPSTAT)
  3. Serial receive/transmit buffer (SSPBUF)
  4. MSSP shift register (SSPSR) -not directly accessible by the user-

- A write to SSPBUF will also write into the SSPSR register



# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

### **SPI Mode** Status and Control Registers.

**REGISTER 19-1: SSPSTAT: MSSP STATUS REGISTER (SPI MODE)**

| R/W-0 | R/W-0 | R-0 | R-0 | R-0 | R-0 | R-0 | R-0 |
|---|---|---|---|---|---|---|---|
| SMP | CKE(1) | D/$\overline{A}$ | P | S | R/$\overline{W}$ | UA | BF |
| bit 7 | | | | | | | bit 0 |

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

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

bit 5 **D/$\overline{A}$:** Data/$\overline{Address}$ bit
Used in I²C mode only.

bit 4 **P:** Stop bit
Used in I²C mode only. This bit is cleared when the MSSP module is disabled, SSPEN is cleared.

bit 3 **S:** Start bit
Used in I²C mode only.

bit 2 **R/$\overline{W}$:** Read/$\overline{Write}$ Information bit
Used in I²C mode only.

bit 1 **UA:** Update Address bit
Used in I²C mode only.

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

**Note 1:** Polarity of clock state is set by the CKP bit (SSPCON1<4>).

**REGISTER 19-2: SSPCON1: MSSP CONTROL REGISTER 1 (SPI MODE)**

| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
|---|---|---|---|---|---|---|---|
| WCOL | SSPOV(1) | SSPEN | CKP | SSPM3 | SSPM2 | SSPM1 | SSPM0 |
| bit 7 | | | | | | | bit 0 |

bit 7 **WCOL:** Write Collision Detect bit (Transmit mode only)
1 = The SSPBUF register is written while it is still transmitting the previous word (must be cleared in software)
0 = No collision

bit 6 **SSPOV:** Receive Overflow Indicator bit(1)
SPI Slave mode:
1 = A new byte is received while the SSPBUF register is still holding the previous data. In case of overflow, the data in SSPSR is lost. Overflow can only occur in Slave mode. The user must read the SSPBUF, even if only transmitting data, to avoid setting overflow (must be cleared in software).
0 = No overflow

bit 5 **SSPEN:** Master Synchronous Serial Port Enable bit
1 = Enables serial port and configures SCK, SDO, SDI and $\overline{SS}$ as serial port pins(2)
0 = Disables serial port and configures these pins as I/O port pins(2)

bit 4 **CKP:** Clock Polarity Select bit
1 = Idle state for clock is a high level
0 = Idle state for clock is a low level

bit 3-0 **SSPM3:SSPM0:** Master Synchronous Serial Port Mode Select bits
0101 = SPI Slave mode, clock = SCK pin, $\overline{SS}$ pin control disabled, $\overline{SS}$ can be used as I/O pin(3)
0100 = SPI Slave mode, clock = SCK pin, $\overline{SS}$ pin control enabled(3)
0011 = SPI Master mode, clock = TMR2 output/2(3,4)
0010 = SPI Master mode, clock = Fosc/64(3)
0001 = SPI Master mode, clock = Fosc/16(3)
0000 = SPI Master mode, clock = Fosc/4(3)

**Note 1:** In Master mode, the overflow bit is not set since each new reception (and transmission) is initiated by writing to the SSPBUF register.
**2:** When enabled, these pins must be properly configured as input or output.
**3:** Bit combinations not specifically listed here are either reserved or implemented in I²C™ mode only.
**4:** PR2 = 0x00 is not supported when running the SPI module in TMR2 Output/2 mode.

# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

**SPI Mode** Operation.

- The SDO pin of the master is connected to the SDI pin of the slave.
- The SDI pin of the master is connected to the SDO pin of the slave.
- To send data to the slave, the master writes data to the SSPBUF register, after which, eight clock pulses are triggered and data is shifted to the slave (SSPIF and BF bits are set).
- To read data from the slave, the master makes (possibly a dummy) write into the SSPBUF register to trigger eight clock pulses to shift in data, following a SSPBUF read.

- Data rate is set by the lowest bits of the SSPCON1 register.

1. FOSC/4 (or FCY)
2. FOSC/16 (or FCY/4)
3. FOSC/64 (or FCY/16)
4. Timer2 output/2

The highest data rate is 10 Mbps for 40 MHz crystal oscillator

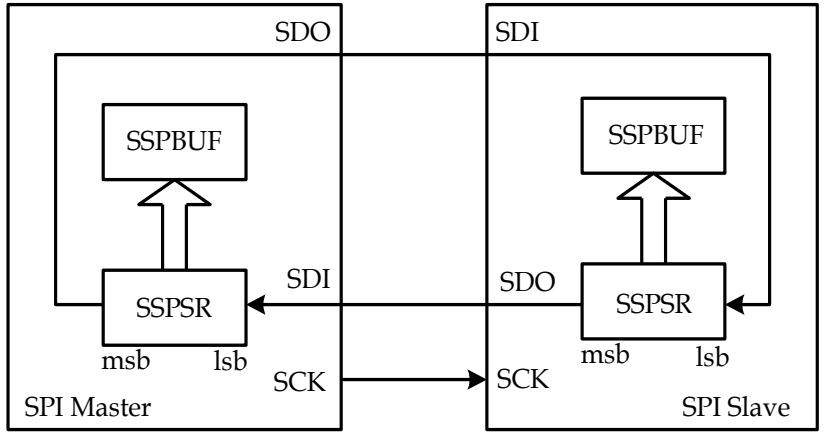



Figure 10.3 Connection between an SPI master and an SPI slave

# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

**SPI Mode.** Clock edge selection for shifting data.

- When the SPI module is not transmitting data, it is referred to as **idle**.
- One can set the SCK signal to be idle low or idle high.
- Setting the **CKP** bit of the SSPCON1 register to 1, makes the SCK signal idle high.
- The **CKE** bit of the SSPSTAT register and the **CKP** bit of the SSPCON1 register together select the edge of the SCK signal for shifting the data:

| CKP | CKE | SCK idle state | SCK edge for data transmission |
|---|---|---|---|
| 0 | 0 | low | rising |
| 0 | 1 | low | falling |
| 1 | 0 | high | falling |
| 1 | 1 | high | rising |

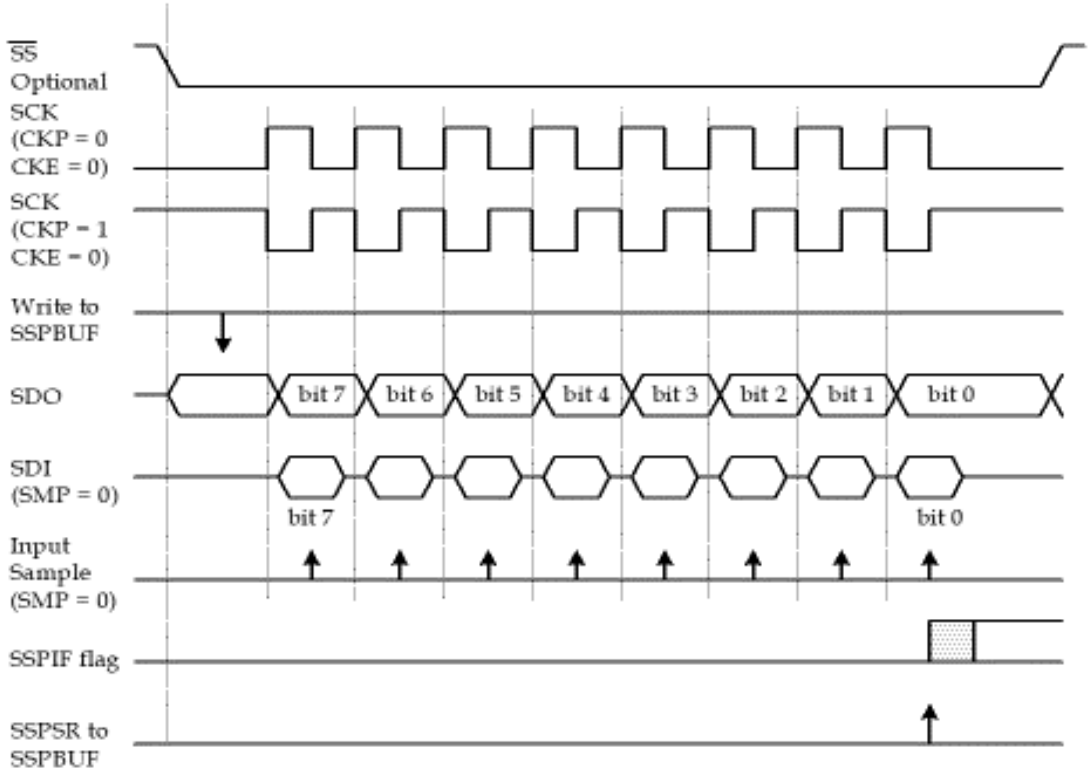
PIC18 offers a variety of clocking and edging options for compatibility with all SPI modes, master and slave.

- One can choose to use the middle or the end of a bit time to sample the incoming data.
- When the **SMP** bit of the SSPSTAT register is 1, incoming data is sampled at the end of the bit time. Otherwise, incoming data is sampled at the middle of a bit time.

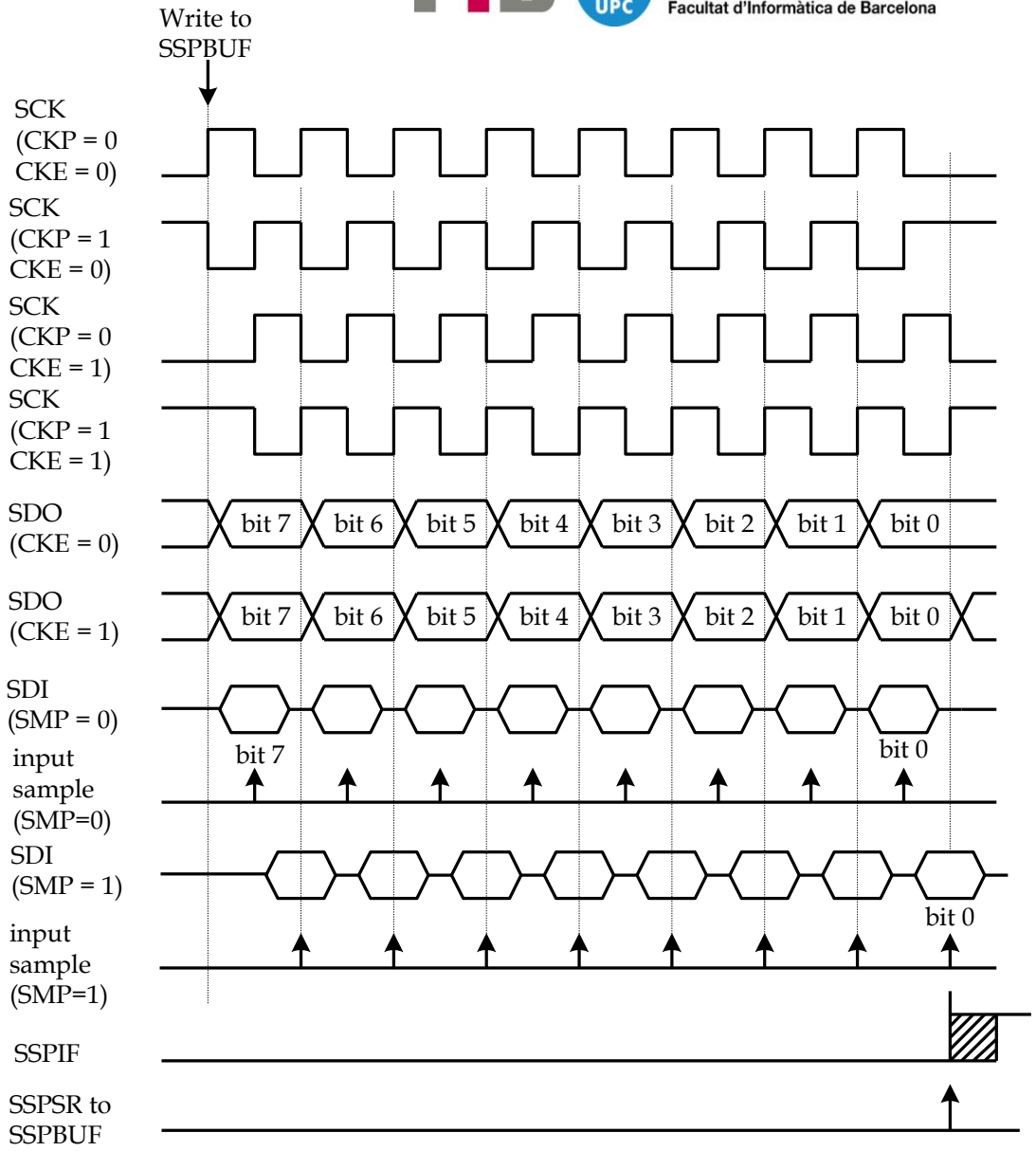
# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

**SPI Mode.** Clocking examples.




Slave mode, with CKE=0




SPI Master mode.

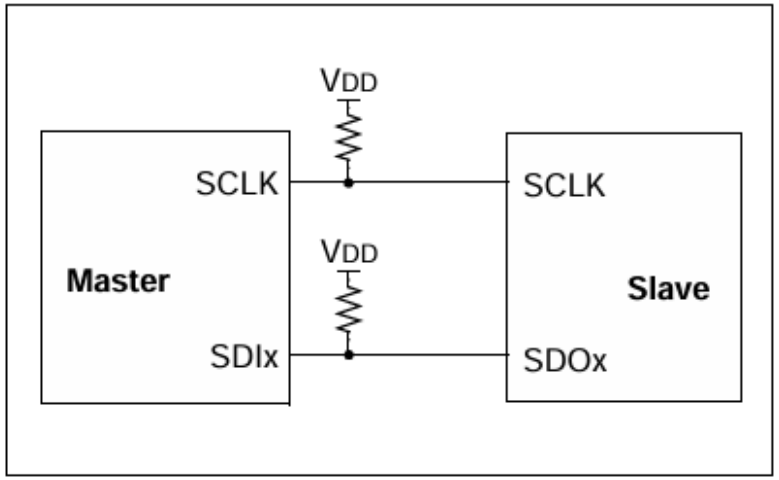
# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

**I2C Mode** (Master or Slave) functions.

There are four potential modes of operation for a given device:

- Master Transmit mode (master is transmitting data to a slave)
- Master Receive mode (master is receiving data from a slave)
- Slave Transmit mode (slave is transmitting data to a master)
- Slave Receive mode (slave is receiving data from the master)




Register configuration allow PIC18 to operate any of them.

Registers involved in the MSSP in $I^2C$ mode:

- SSPCON1
- SSPCON2
- SSPSTAT
- SSPBUF
- SSPADD (slave add. or Master Clk rate)

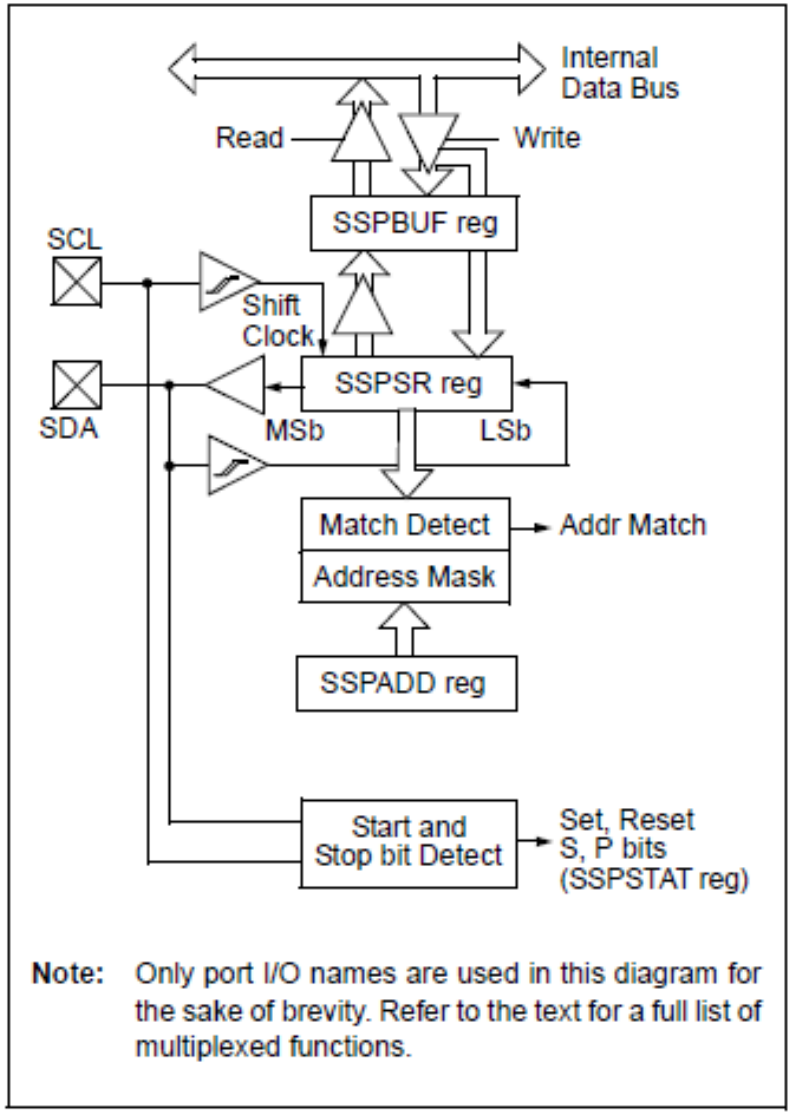
# 8.5 SPI and I2C buses

## The PIC18 MSSP Module

**I2C Mode.** Operations supported.

The I2C interface supports the following modes and features:
- Master mode
- Slave mode
- Byte NACKing (Slave mode)
- Limited Multi-master support
- 7-bit and 10-bit addressing
- Start and Stop interrupts
- Interrupt masking
- Clock stretching
- Bus collision detection
- General call address matching
- Address masking
- Address Hold and Data Hold modes
- Selectable SDAx hold times

**FIGURE 19-7: MSSP BLOCK DIAGRAM ($I^2C$™ MODE)**



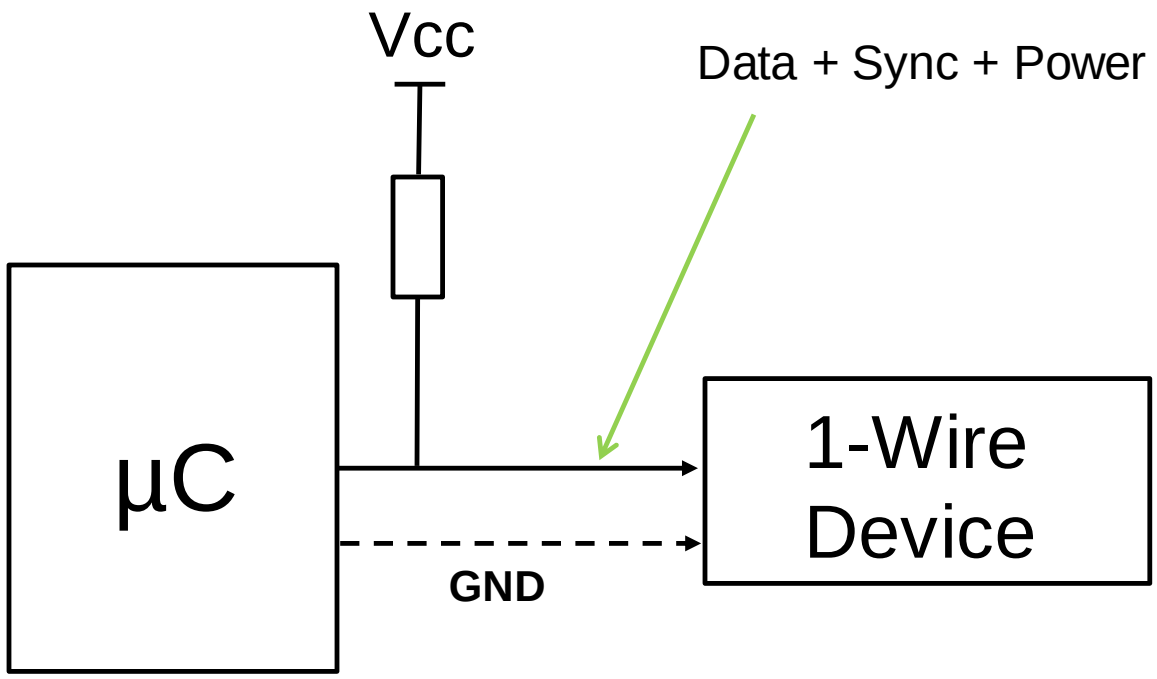

**Note:** Only port I/O names are used in this diagram for the sake of brevity. Refer to the text for a full list of multiplexed functions.

# 8.6 1-wire serial interface

## 1-wire bus

Is a Bidirectional, half-duplex, serial communication bus, that powers over a single connection and ground return. Two serial communication speeds 15Kbps or 125kbps. The idea is to put everything (power, data, clock) on a single line.

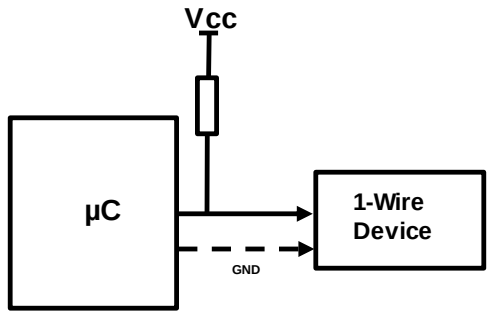
There is a unique Unalterable ID in every device !!!

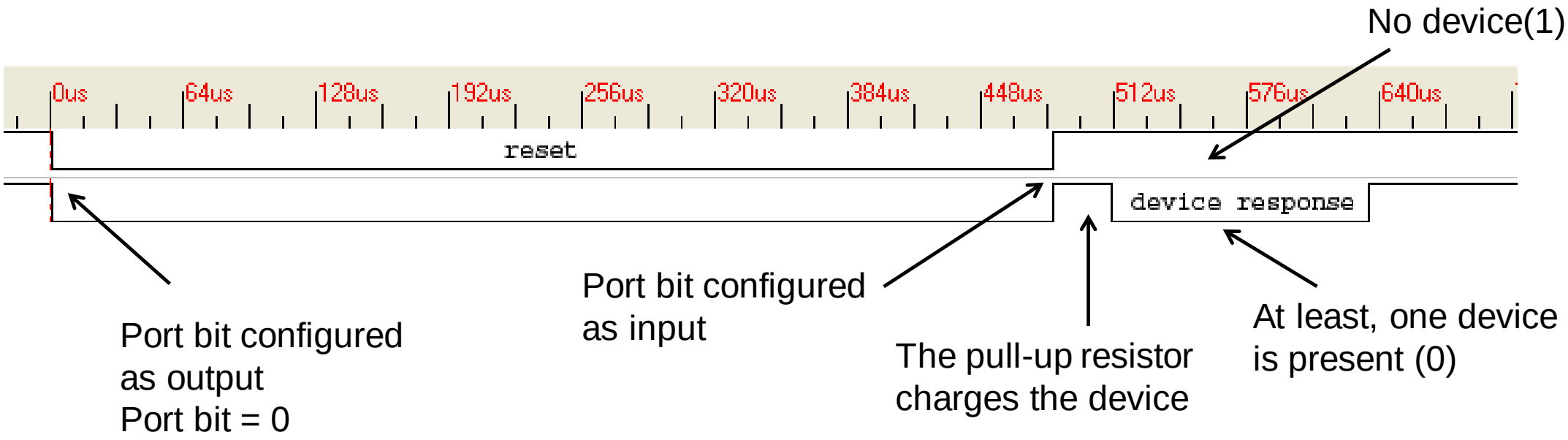



# 8.6 1-wire serial interface

## 1-wire bus operation.

The master starts a transmission with a "reset" pulse, which pulls the wire to 0 volts for 480 µs. This resets every slave device on the bus, probably by depriving them all of power. After that, any slave device, if present, shows that it exists with a "presence" pulse: it holds the wire to ground for at least 60 µs after the master releases the bus.



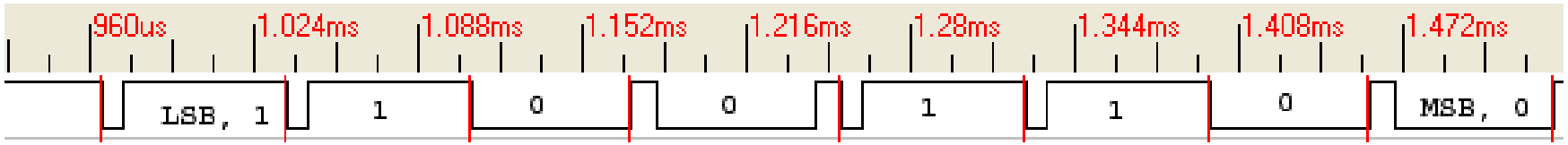
# 8.6 1-wire serial interface

## 1-wire bus operation.

1-wire protocol guarantees an edge (sync) and enough charge time (pull-up, power) per bit.

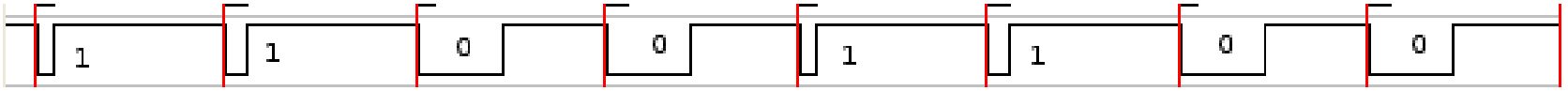
**1Wire. Sending bits…**

To send a "1", the master sends a very brief (1 - 15 µs) low pulse. To send a "0", the master sends a 60 µs low pulse.




**1Wire. Receiving bits…**

When receiving data, the master sends a 1-15 µs 0 volt pulse to start each bit. If the transmitting slave unit wants to send a "1", it does nothing, and the wire goes immediately up to the pulled-up voltage. If the transmitting slave wants to send a "0", it pulls the data line to ground for at least 15 µs.



# 8.6 1-wire serial interface

## 1-wire bus operation.

Bit banging example: detect if there is a 1-Wire device in RA0.

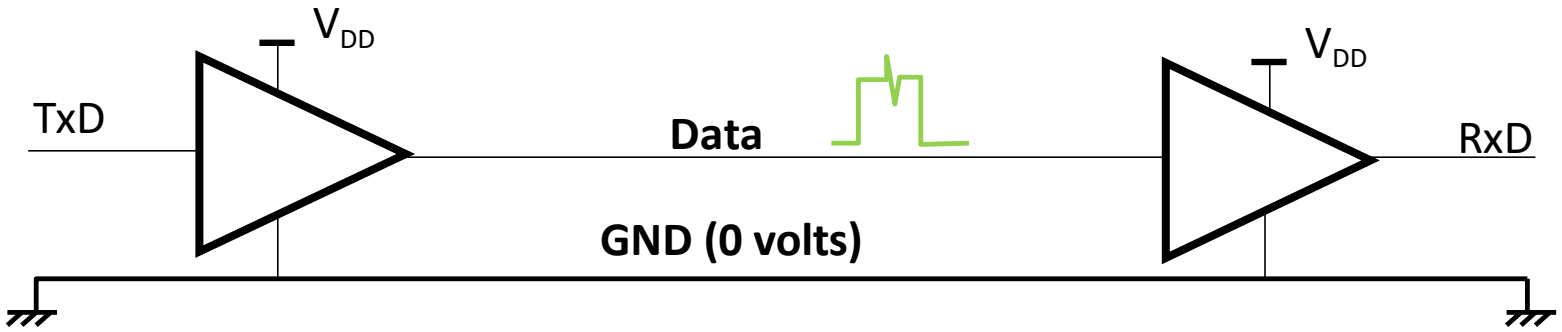
```
unsigned char 1WireDetect(void)
{
        unsigned char result;

        TRISAbits.RA0=0;          // Set pin as output
        PORTAbits.RA0=0;          // Bring to 0 to reset the (possible) devices
        Delay_us(480);            // drive low for minimum of 480 us
        TRISAbits.RA0=1;          // release line (input), let 4k7 pullup take the bus high
        Delay_us(60);             // 60uSec. within 15 to 60uSec, if there is a device, it will pull bus low
        result = PORTAbits.RA0;   // read the 1Wire bus line
        Delay_us(240);            // 240usec. ensure 'presence' pulse is complete
        return (~result);         // 0 = device found, 1 = device not found
}
```

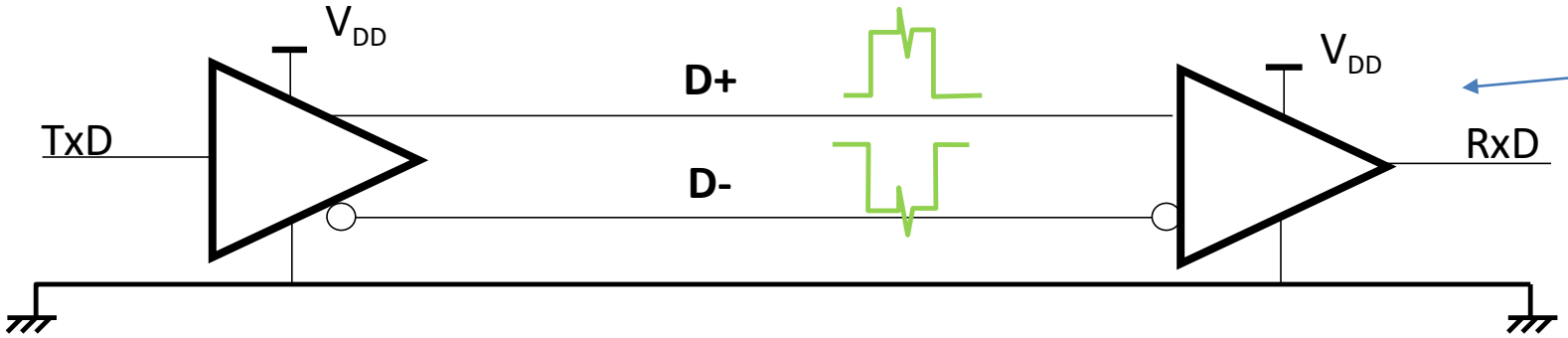
# 8.7 USB. Differential interfaces.

## Differential interfaces.

Single wire interfaces are very weak to noise (changes in logical levels)




**Differential** interfaces, where two wires are sent (D+ direct, D- negate) and then compared, are robust to electromagnetic noise allowing fast speeds and longer wires.

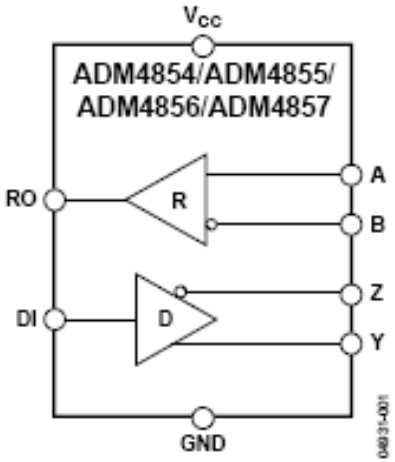



As noise degrades both signals (D+,D-) when are subtracted, noise is cancelled.

# 8.7 USB. Differential interfaces.

## Differential interfaces.

Differential interfaces are packed in standard components (Half and Full-Dupplex) and used by buses like RS-422, RS-485 (improved serial buses) and PC standard USB.

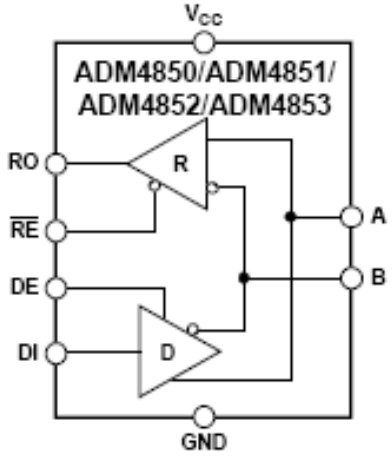



Full-duplex

**Differential transmission electrical interfaces (RS-422)**

Data reception enabling control

Data transmission enabling control



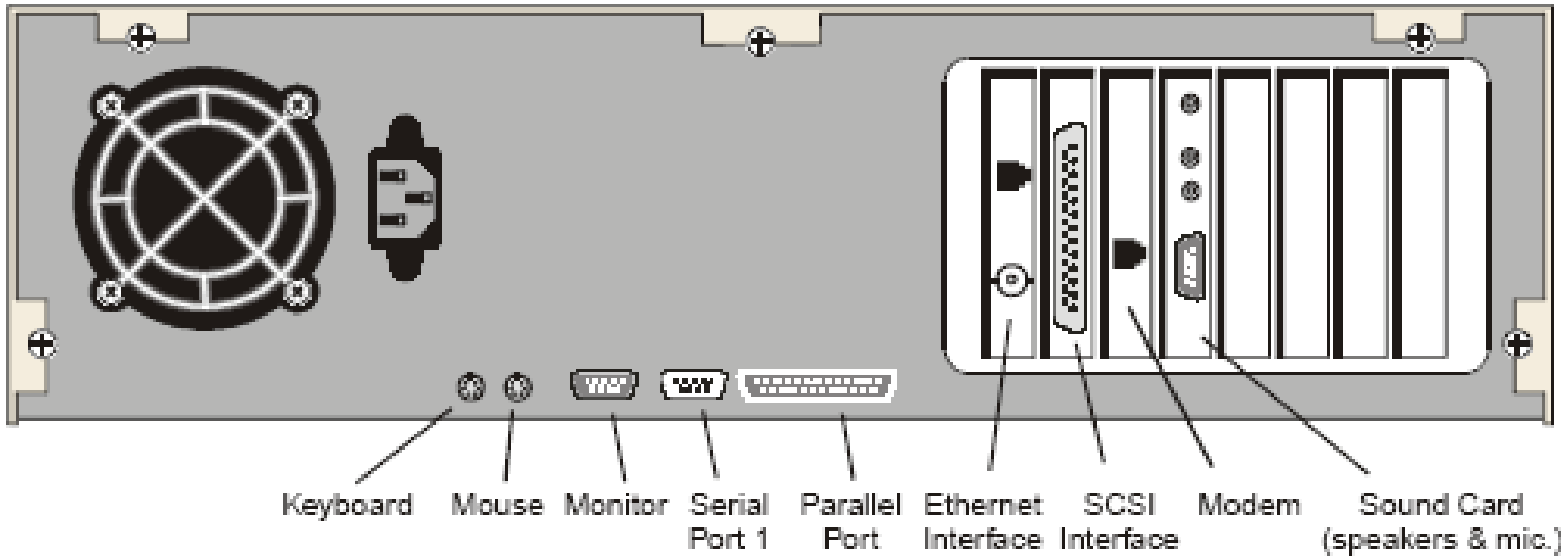

Half-duplex

**Differential transmission electrical interfaces (RS-485)**

# 8.7 USB. Differential interfaces.

## USB bus

**USB** is a serial, half-duplex (versions 1 and 2), full-duplex (version 3), master-slave, asynchronous and differential bus.
It was developed in mid-1990 by Compaq, Intel, Microsoft and NEC to standardize PC connectors. By the time, a standard PC had more than 10 different peripheral connectors.




# 8.7 USB. Differential interfaces.

## USB bus. Expected benefits.

- Low Cost interface (less wires, cheap connectors).
- Hot Pluggable (peripherals can be connected and configured without stopping the PC).
- Single Connector Type (not achieved: A, B, C, mini, micro, etc.…)
- Cable Power (majority of peripherals are powered by the bus (5V-500mA). Not suitable for printers or monitors).
- System Resources Requirements eliminated. There is protocol to identify and load a driver for every peripheral without previous configurations.
- Error detection and recovery. Some modes implement error check and corrections.
- Different data-transfer systems: depending on the peripheral, there are several data transfer protocols: interrupt, isochronous, bulk…

# 8.7 USB. Differential interfaces.

## USB bus. Specifications.

### USB 1 and 2 specifications.

Low Speed = 1.5 Mb/s (USB 1.1) Keyboard, Mouse
Full Speed = 12Mb/s (USB 1.1) Digital audio, Printers
High Speed = 480Mb/s (USB 2.0) Cameras, Disks

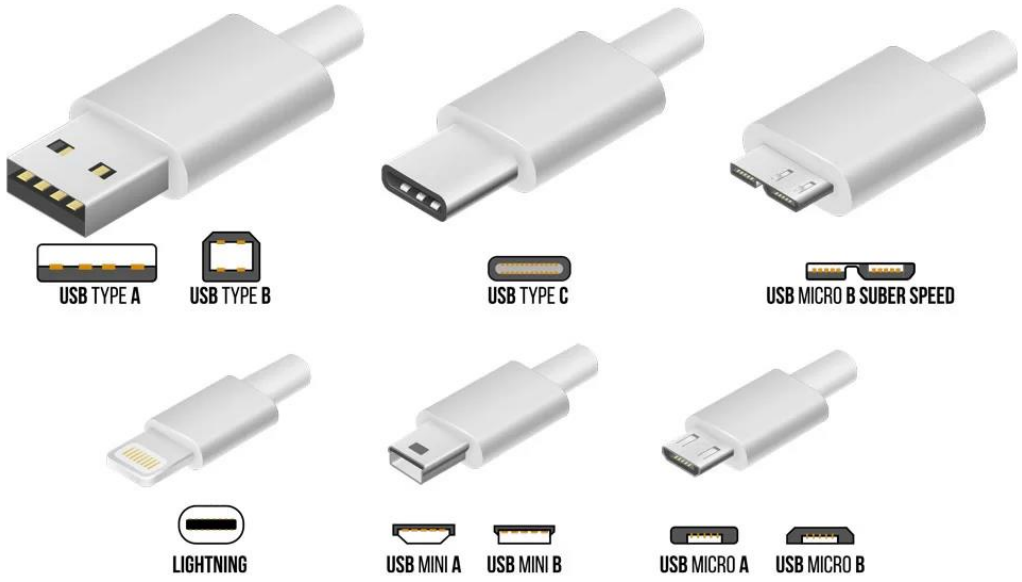
} Half-duplex system. 500mA. Several modes through endpoints: Control, Interrupt, Isochronous and Bulk.

### USB 3 (High bandwith transfers)

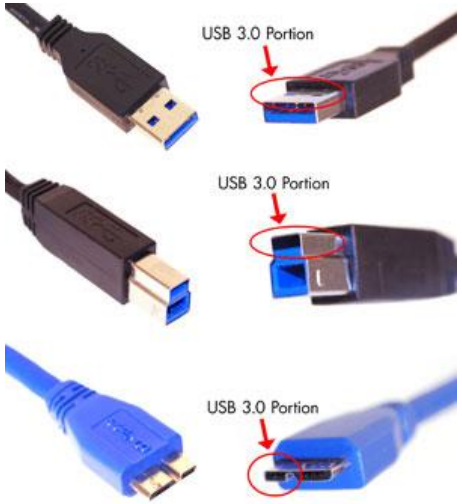
- Super Speed= 5Gb/s (USB 3.0) with back-Compatibility.
- Full-duplex with Transfer rates of 4.8 Gbps — 5 extra wires: SuperSpeed Transmit (SSTX+, SSTX–), SuperSpeed Receive (SSRX+, SSRX–), and an additional ground (GND).
- More power: up to 900mA.
- Point-to-Point Routing: Packets originated in the host contain a extra 20-bit "route string" field. Allows better power efficiency with less power for idle states.
- Improved bus utilization: A new feature was added (using packets NRDY and ERDY) to let a device asynchronously notify the host of its readiness.

# 8.7 USB. Differential interfaces.

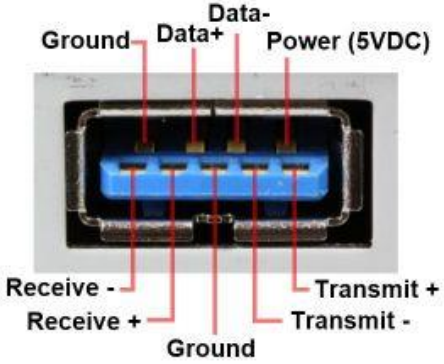
## USB bus. Connectors.



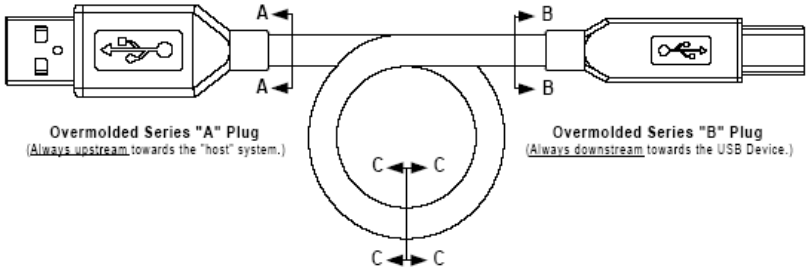

A, B (USB 1 and 2), and C connectors



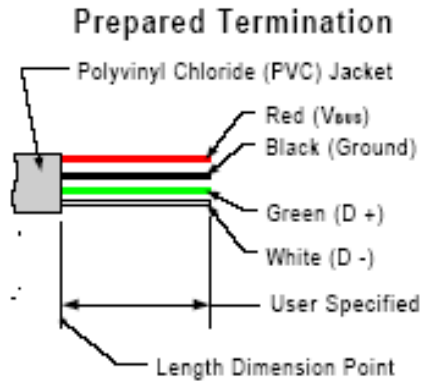

Hybrid connectors




USB 3 board connector




Wiring: 5 meters max.

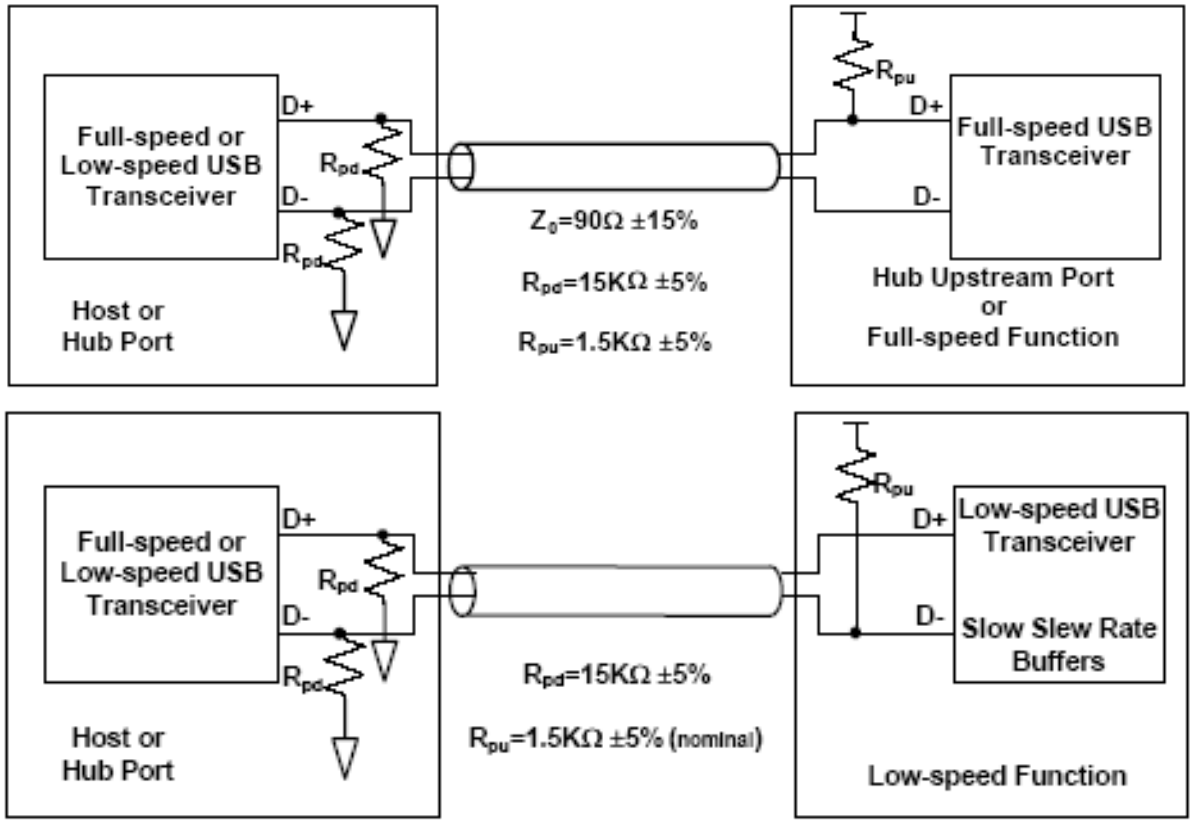


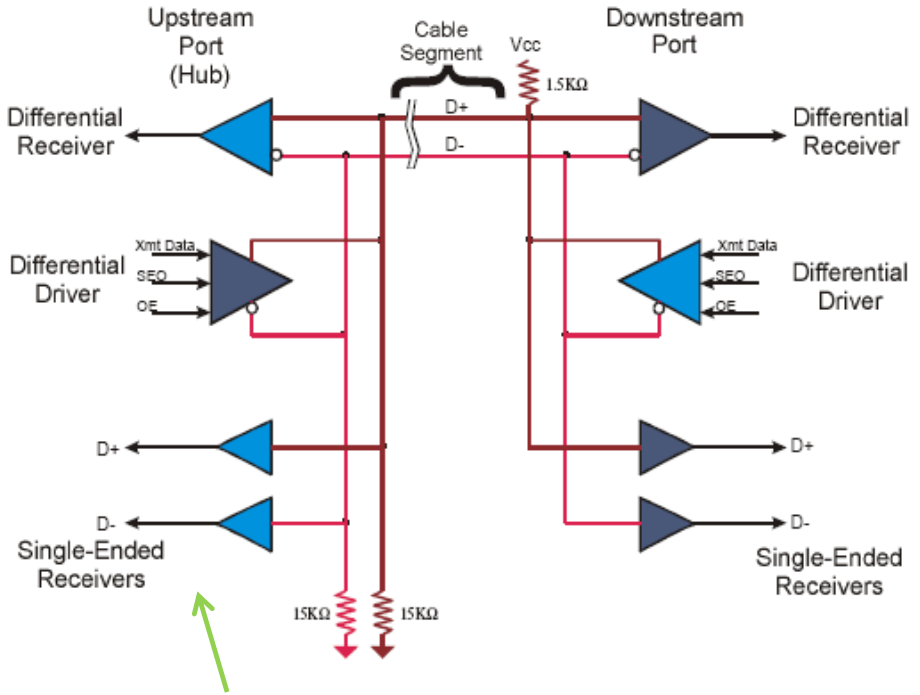

Wiring: USB 1 and 2.

# 8.7 USB. Differential interfaces.

## USB bus. Key operation concepts.

**Signaling**. The host can detect the connection of a new device by detecting a rising edge in one of the terminals. Devices have pull-ups on some of the lines, this can indicate things like expected speed.




Single-ended receivers allow the host/ master to detect changes on D+ or D-

# 8.7 USB. Differential interfaces.

## USB bus. Key operation concepts.

**Codification**. As an high-speed asynchronous bus, USB uses **NRZi** codification plus **stuffed** bits.

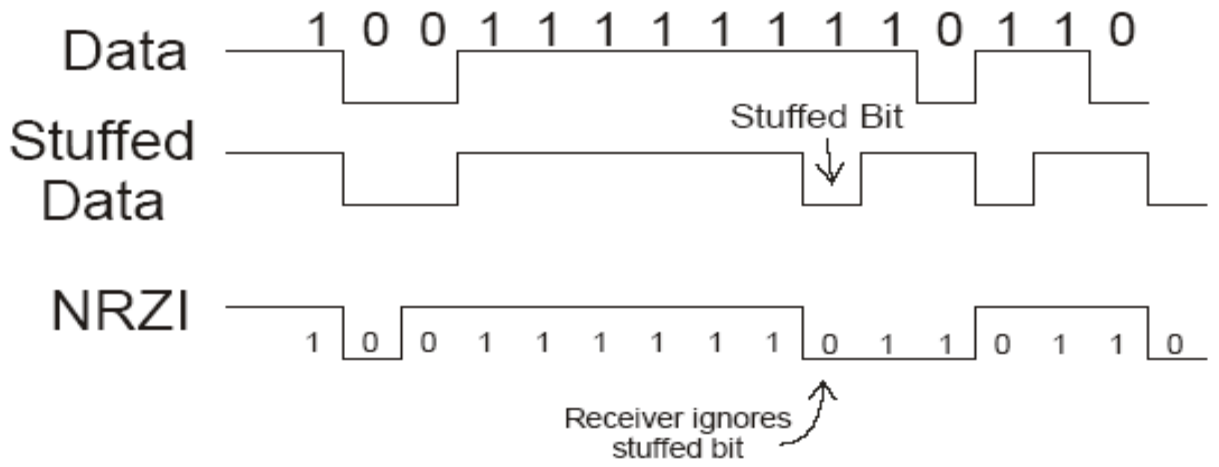
NRZi (non return to zero inverted) codifies: 1 - hold level. 0 – switch level

Stuffed bit (extra 0 after the 6th 1 bit) is used to avoid long series of '1'.

Changes are used to syncronize host and devices.

Example,

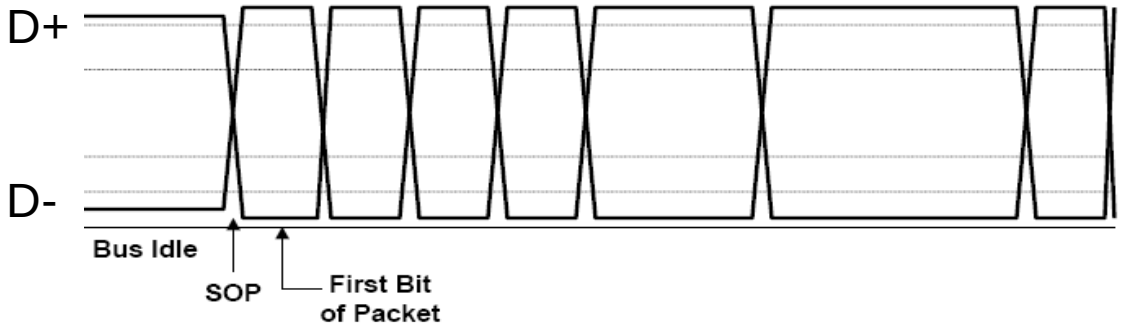
data to be sent = 100111111110110

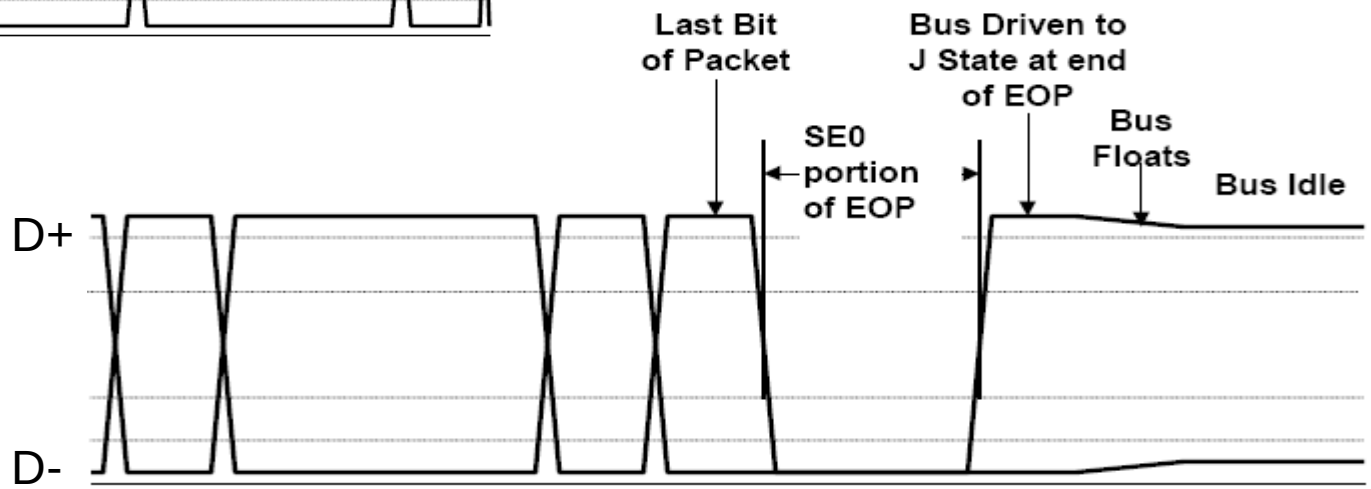


# 8.7 USB. Differential interfaces.

## USB bus. Key operation concepts.

**Codification**. **Start** is marked with a '0' in D+ (idle is 1). **Stop** is marked by simultaneously writing 00 in D+ and D-. This exception is detected by all the devices.

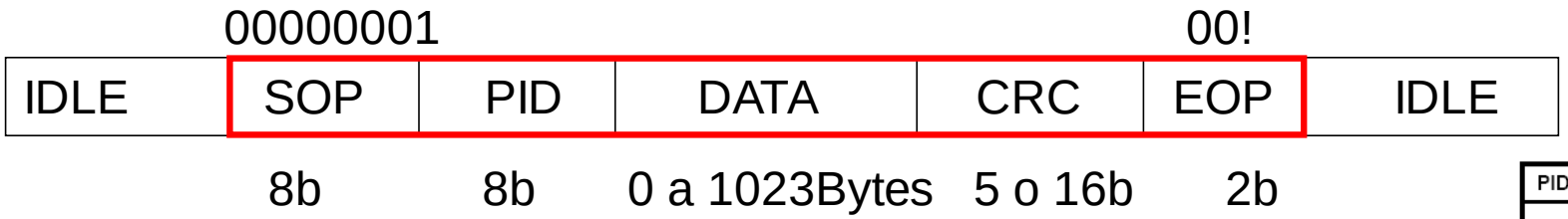


# 8.7 USB. Differential interfaces.

## USB bus. Key operation concepts.

**Packing**. Data is written in standard packets starting with a "Start of Packet" SOP and ending with an "End of packet" EOP (00 in both lines). Note that SOP has a bunch of zeros to synchronize the clocks.




PID (packet identifier), identifies the nature of the transmission:

- Tokens (IN, OUT, Start of Frame, Setup), are used to start the transmission of data (length, directions, type…)
- Handshake packets are used to notify errors or busy receiver.
- Data packets (0/1) are numbered to identify data if some packet is lost, there are errors in the communications, etc.

| PID Type | PID Name | PID[3:0]* | Description |
|---|---|---|---|
| Token | OUT | 0001B | Address + endpoint number in host-to-function transaction |
| | IN | 1001B | Address + endpoint number in function-to-host transaction |
| | SOF | 0101B | Start-of-Frame marker and frame number |
| | SETUP | 1101B | Address + endpoint number in host-to-function transaction for SETUP to a control pipe |
| Data | DATA0 | 0011B | Data packet PID even |
| | DATA1 | 1011B | Data packet PID odd |
| Handshake | ACK | 0010B | Receiver accepts error-free data packet |
| | NAK | 1010B | Rx device cannot accept data or Tx device cannot send data |
| | STALL | 1110B | Endpoint is halted or a control pipe request is not supported. |
| Special | PRE | 1100B | Host-issued preamble. Enables downstream bus traffic to low-speed devices. |

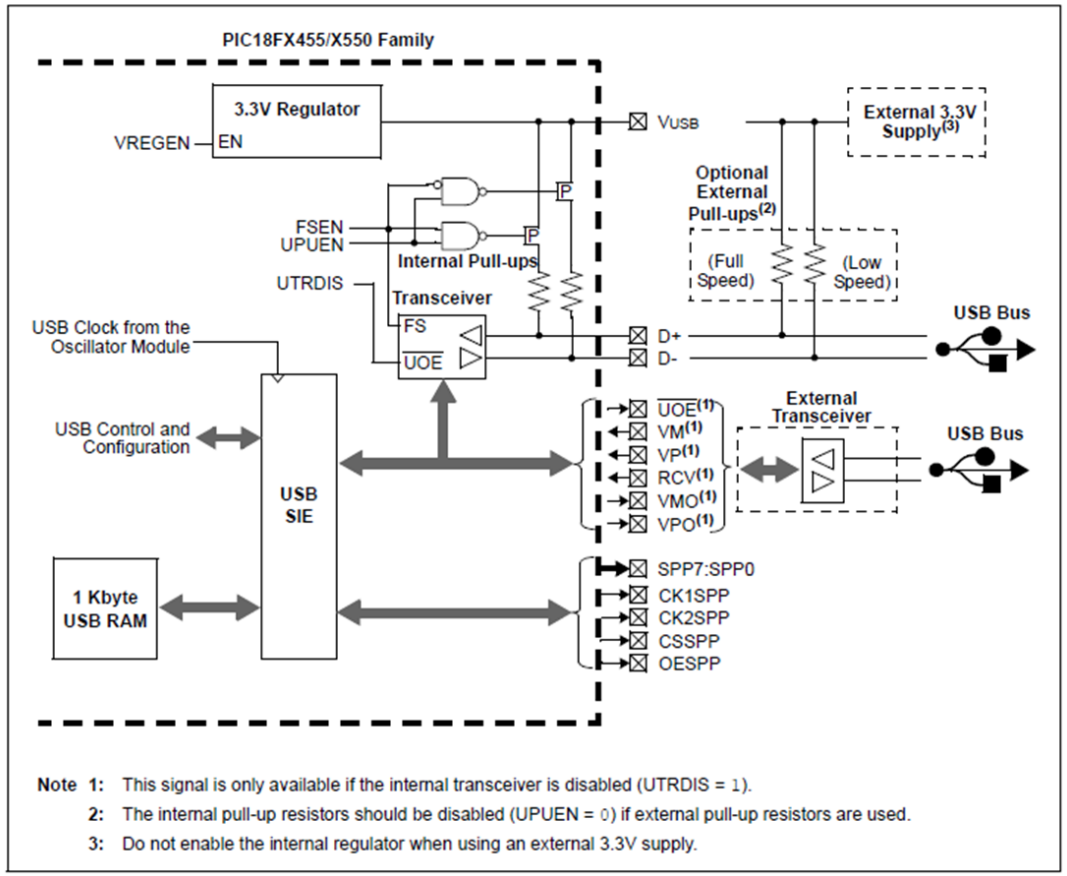
*Note: PID bits are shown in MSb order. When sent on the USB, the rightmost bit (bit 0) will be sent first.

# 8.7 USB. Differential interfaces.

## USB bus. Implementation on a PIC18F microcontroller.

Some 18F PIC micros, like 18F4550, have peripherals to work as USB device (not host).

**FIGURE 17-1: USB PERIPHERAL AND OPTIONS**




Note 1: This signal is only available if the internal transceiver is disabled (UTRDIS = 1).
2: The internal pull-up resistors should be disabled (UPUEN = 0) if external pull-up resistors are used.
3: Do not enable the internal regulator when using an external 3.3V supply.

# 8.7 USB. Differential interfaces.

## USB bus. Implementation on a PIC18F microcontroller.

At **hardware** level, several configurations for powering are available.

Bus power**:** The USB powers the microcomputer (5V, 0.5A) so the micro's software assumes that it is connected to the PC.

Self power**:** The microcomputer is self powered and the bus power is used to detect the connection between the two devices (attach sense)

Dual power**:** the microcomputer is powered either by the USB or by its own source of power.

FIGURE 17-10: BUS POWER ONLY

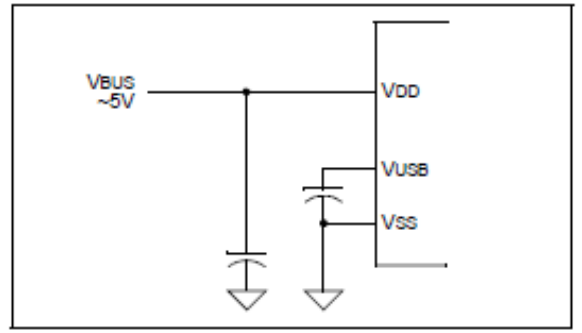



FIGURE 17-11: SELF-POWER ONLY

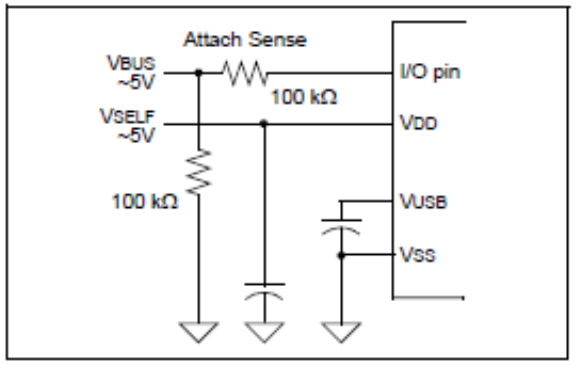
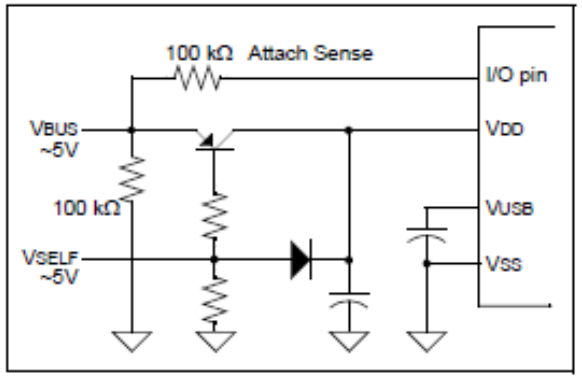



FIGURE 17-12: DUAL POWER EXAMPLE



# 8.7 USB. Differential interfaces.

## USB bus. Implementation on a PIC18F microcontroller.

At **software** level, a special code structure is necessary. If a USB request is not attended, the host will disconnect our microcontroller from the system. PIC can be programmed to work as a special keyboard, mouse or HID (Human Interface Device).

```
void main(void)
{
  Config_USB( );
  Config_IO( );
  while(1)
  {
    Attend_USB( );
    Attend_IO( );      // attend IO must be time-limited
  }
}
```

# 8.7 USB. Differential interfaces.

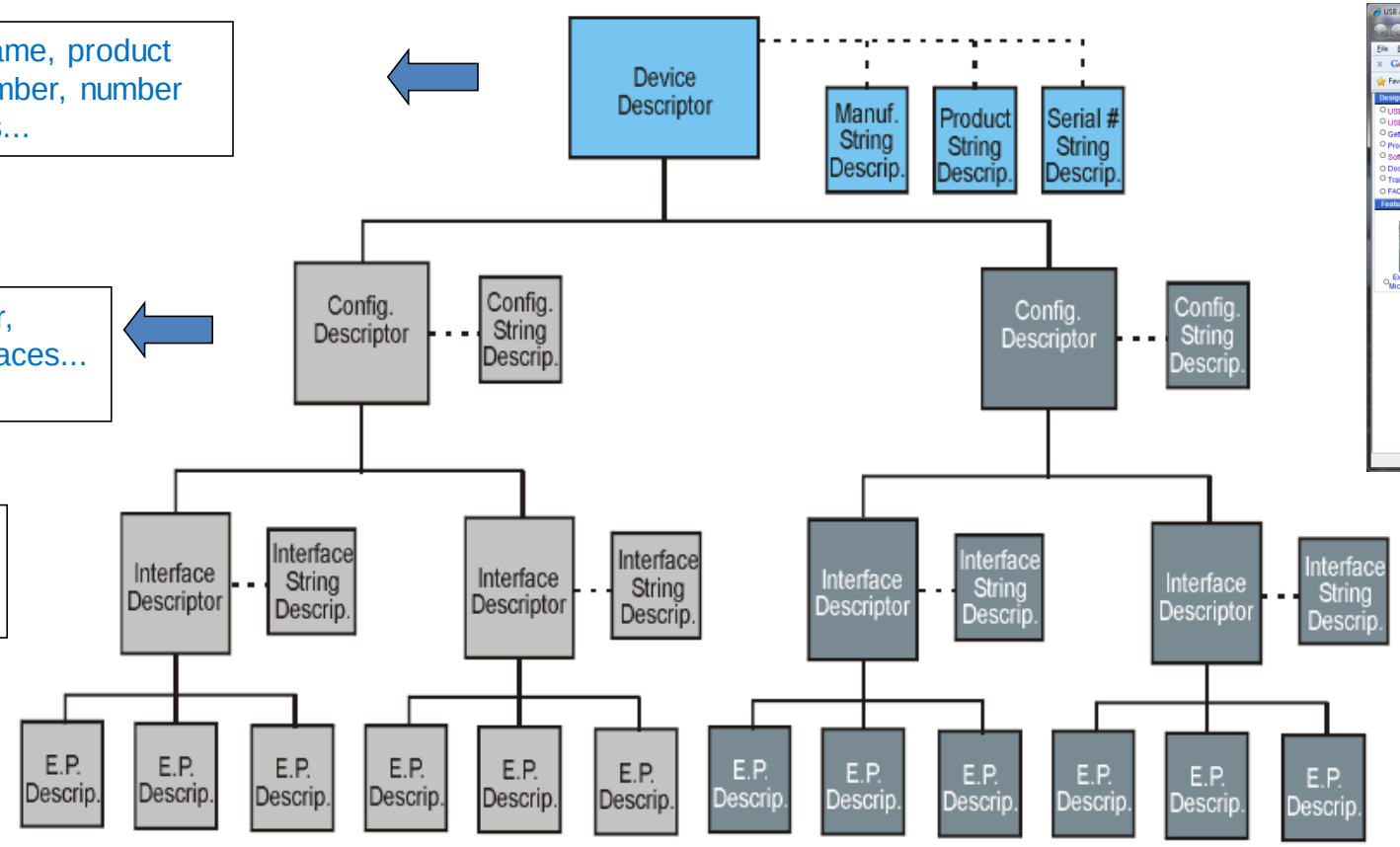
At **software** level a big set of libraries are needed to accomplish USB specification (every descriptor is coded).

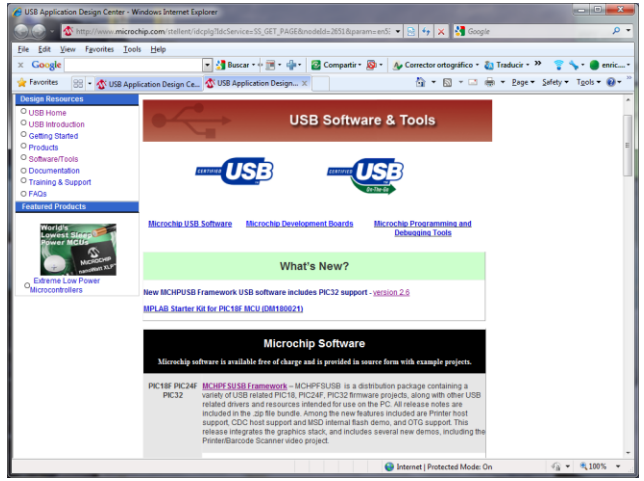
Manufacturer name, product name, serial number, number of configurations...

Maximum power, number of interfaces...

Device class, number of endpoints

*IN or OUT, Interrupt, bulk ...*




It is a good idea to use Microchip USB SDK.