Communication interfaces

- I2C - Inter-Integrated Circuit
- SPI - Serial Peripheral Interface
- UART(RS232) - Universal Asynchronous Receiver and Transmitter
- JTAG - Joint Test Action Group
- USB - Universal Serial Bus
Communication interfaces

AN10216-01 I^2C Manual

Data Transfer Rate (Mbps)

- GTLP
- BTL
- ETL
- General Purpose Logic
- CML
- 1394.a
- LVDS = RS-644 ECL/PECL/LVPECL
- RS-422
- RS-485
- RS-232
- RS-423

Backplane Length (meters)

- 0.5
- 0
- 10
- 1
- 0.1

Cable Length (meters)

- 100
- 10
- 1
- 0.1
<table>
<thead>
<tr>
<th>Communication interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN (1 Wire)</td>
</tr>
<tr>
<td>I²C (‘Industrial’, and SMBus)</td>
</tr>
<tr>
<td>SPI</td>
</tr>
<tr>
<td>CAN (fault tolerant)</td>
</tr>
<tr>
<td>I²C</td>
</tr>
<tr>
<td>CAN (high speed)</td>
</tr>
<tr>
<td>I²C ‘High Speed mode’</td>
</tr>
<tr>
<td>USB (1.1)</td>
</tr>
<tr>
<td>SCSI (parallel bus)</td>
</tr>
<tr>
<td>Fast SCSI</td>
</tr>
<tr>
<td>Ultra SCSI-3</td>
</tr>
<tr>
<td>Firewire / IEEE1394</td>
</tr>
<tr>
<td>Hi-Speed USB (2.0)</td>
</tr>
</tbody>
</table>

AN10216-01 I²C Manual
## Communication interfaces

<table>
<thead>
<tr>
<th>Bus</th>
<th>Data rate (bits/sec)</th>
<th>Length (meters)</th>
<th>Length limiting factor</th>
<th>Nodes Typ. number</th>
<th>Node number limiting factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>I²C</td>
<td>400k</td>
<td>2</td>
<td>wiring capacitance</td>
<td>20</td>
<td>400pF max</td>
</tr>
<tr>
<td>I²C with buffer</td>
<td>400k</td>
<td>100</td>
<td>propagation delays</td>
<td>any</td>
<td>no limit</td>
</tr>
<tr>
<td>I²C high speed</td>
<td>3.4M</td>
<td>0.5</td>
<td>wiring capacitance</td>
<td>5</td>
<td>100pF max</td>
</tr>
<tr>
<td>CAN 1 wire</td>
<td>33k</td>
<td>100</td>
<td>total capacitance</td>
<td>32</td>
<td>load resistance and transceiver current drive</td>
</tr>
<tr>
<td>CAN differential</td>
<td>5k</td>
<td>10km</td>
<td>propagation delays</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>USB (low-speed, 1.1)</td>
<td>1.5M</td>
<td>3</td>
<td>cable specs</td>
<td>2</td>
<td>bus specs</td>
</tr>
<tr>
<td>USB (full-speed, 1.1)</td>
<td>1.5/2M</td>
<td>25</td>
<td>5 cables linking 6 nodes (5m cable node to node)</td>
<td>127</td>
<td>bus and hub specs</td>
</tr>
<tr>
<td>Hi-Speed USB (2.0)</td>
<td>480M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEEE-1394</td>
<td>100 to 400M+</td>
<td>72</td>
<td>16 hops, 4.5M each</td>
<td>63</td>
<td>6-bit address</td>
</tr>
</tbody>
</table>

### AN10216-01 I²C Manual
I²C Interface
Inter-Integrated Circuit (IIC or I²C)
• Inter-integrated circuit (IIC or I²C) was defined by Philips providing a simple way to talk between IC’s by using a minimum number of pins (only 2 wires).

• Originally, the I²C bus was designed to link a small number of devices on a single board, to manage the tuning of a car radio or TV.

• Compatibility of parts (ICs) from different manufacturers: (Simple Hardware standards, Simple Software protocol standard)

• No specific wiring or connectors - most often it’s just PCB tracks

• It is a recognized standard throughout our industry and is used now by ALL major IC manufacturers
I²C: Features

- Only two bus lines are required: a serial data line (SDA) and a serial clock line (SCL).

- Each device connected to the bus is software addressable by a unique address and simple master/slave relationships exist at all times; masters can operate as master-transmitters or as master-receivers.

- It’s a true multi-master bus including collision detection and arbitration to prevent data corruption if two or more masters simultaneously initiate data transfer.

- Serial, 8-bit oriented, bi-directional data transfers can be made at up to 100 kbit/s in the Standard-mode, up to 400 kbit/s in the Fast-mode (1992 year), or up to 3.4 Mbit/s in the High-speed mode (1998 year).

- On-chip filtering (50 ns) rejects spikes on the bus data line to preserve data integrity.

- The number of ICs that can be connected to the same bus segment is limited only by the maximum bus capacitive loading of 400 pF.

- A respectable communication distance which can be extended to longer distances with bus extenders.

- Compatible with a number of processors with integrated I²C ports (MPU 8,16,32 bits)

- Easily emulated in software by any microcontroller.
I²C: Bus Terminology

- **Transmitter** - the device that sends data to the bus. A transmitter can either be a device that puts data on the bus of its own accord (a ‘master-transmitter’), or in response to a request from data from another devices (a ‘slave-transmitter’).

- **Receiver** - the device that receives data from the bus.

- **Master** - the component that initializes a transfer, generates the clock signal, and terminates the transfer. A master can be either a transmitter or a receiver.

- **Slave** - the device addressed by the master. A slave can be either receiver or transmitter.

- **Multi-master** - the ability for more than one master to co-exist on the bus at the same time without collision or data loss.

- **Arbitration** - the prearranged procedure that authorizes only one master at a time to take control of the bus.

- **Synchronization** - the prearranged procedure that synchronizes the clock signals provided by two or more masters.

- **SDA** - data signal line (Serial DAta)

- **SCL** - clock signal line (Serial CLock)
I²C: open drain, open collector

- Pull-up resistors
  - Typical value 2 kΩ to 10 kΩ

- Open Drain structure (or Open Collector) for both SCL and SDA
I²C: Data change, Start & Stop condition

Data on the SDA line must be stable before the rising edge of SCL. Data on SDA can change only when SCL is LOW.

START (S): A HIGH to LOW transition on the SDA line while SCL is HIGH.

STOP (P): A LOW to HIGH transition on the SDA line while SCL is HIGH.
START and STOP conditions are always generated by the master.

The bus is considered to be busy after the START condition.

The bus is considered to be free again a certain time after the STOP condition.

The bus stays busy if a repeated START (Sr) is generated instead of a STOP condition. START (S) and repeated START (Sr) conditions are functionally identical, therefore, the S symbol is usually used as a generic term to represent both the START and repeated START conditions, unless Sr is particularly relevant.

Detection of START and STOP conditions by devices connected to the bus is easy if they have the necessary interfacing hardware. However, microcontrollers with no such interface have to sample the SDA line at least twice per clock period to sense the transition.
• Every byte put on the SDA line must be eight bits long. The number of bytes that can be transmitted per transfer is unrestricted.
• Data is transferred with the Most Significant Bit (MSB) first.
• Each byte must be followed by an Acknowledge bit.
• If a slave cannot receive or transmit another complete byte of data until it has performed some other function, for example servicing an internal interrupt, it can hold the clock line SCL LOW to force the master into a wait state. Data transfer continues when the slave is ready for another byte of data and releases clock line SCL.
- The acknowledge takes place after every byte. The acknowledge bit allows the receiver to signal the transmitter that the byte was successfully received and another byte may be sent. The master generates all clock pulses, including the acknowledge ninth clock pulse.

- The **Acknowledge (ACK)** signal is defined as follows: the transmitter releases the SDA line during the acknowledge clock pulse so the receiver can pull the SDA line LOW and it remains stable LOW during the HIGH period of this clock pulse. Set-up and hold times must also be taken into account.

- When **SDA remains HIGH during this ninth clock pulse**, this is defined as the **Not Acknowledge (NACK)** signal. The master can then generate either a STOP condition to abort the transfer, or a repeated START condition to start a new transfer.
I²C: Not Acknowledge (NACK)

There are five conditions that lead to the generation of a NACK:

1. No receiver is present on the bus with the transmitted address so there is no device to respond with an acknowledge.
2. The receiver is unable to receive or transmit because it is performing some real-time function and is not ready to start communication with the master.
3. During the transfer, the receiver gets data or commands that it does not understand.
4. During the transfer, the receiver cannot receive any more data bytes.
5. A master-receiver must signal the end of the transfer to the slave transmitter.
I²C: Acknowledge (ACK) and Not Acknowledge (NACK)
I²C: The slave address and R/W bit

After the START condition (S), a slave address is sent. This address is seven bits long followed by an eighth bit which is a data direction bit:

- R/W = L -> WRITE
- R/W = H -> READ
Each device connected to the bus is addressable by a unique address, which is the first byte after the START procedure.

### 7-bit address

![7-bit address diagram]

### 10-bit address

![10-bit address diagram]

10-bit addressing expands the number of possible addresses. Devices with 7-bit and 10-bit addresses can be connected to the same I2C-bus, and both 7-bit and 10-bit addressing can be used in all bus speed modes. Currently, 10-bit addressing is not being widely used.
Address allocation is coordinated by the I²C-bus committee. 112 different addresses of devices are available (others reserved).

<table>
<thead>
<tr>
<th>Slave address</th>
<th>R/W bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 000</td>
<td>0</td>
<td>general call address[1]</td>
</tr>
<tr>
<td>0000 000</td>
<td>1</td>
<td>START byte[2]</td>
</tr>
<tr>
<td>0000 001</td>
<td>X</td>
<td>CBUS address[3]</td>
</tr>
<tr>
<td>0000 010</td>
<td>X</td>
<td>reserved for different bus format[4]</td>
</tr>
<tr>
<td>0000 011</td>
<td>X</td>
<td>reserved for future purposes</td>
</tr>
<tr>
<td>0000 1XX</td>
<td>X</td>
<td>Hs-mode master code</td>
</tr>
<tr>
<td>1111 1XX</td>
<td>1</td>
<td>device ID</td>
</tr>
<tr>
<td>1111 0XX</td>
<td>X</td>
<td>10-bit slave addressing</td>
</tr>
</tbody>
</table>

See: I²C-bus specification and user manual Rev. 6 — 4 April 2014
1. Master-transmitter transmits to slave-receiver. The transfer direction is not changed. The slave receiver acknowledges each byte.

A master-transmitter addressing a slave receiver with a 7-bit address (the transfer direction is not changed)
2. Master reads slave immediately after first byte. At the moment of the first acknowledge, the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter. This first acknowledge is still generated by the slave. The master generates subsequent acknowledges. The STOP condition is generated by the master, which sends a not-acknowledge (!A) just before the STOP condition.
3. Combined format. During a change of direction within a transfer, the START condition and the slave address are both repeated, but with the R/W bit reversed. If a master-receiver sends a repeated START condition, it sends a not-acknowledge (!A) just before the repeated START condition.

Combined formats can be used, for example, to control a serial memory. The internal memory location must be written during the first data byte. After the START condition and slave address is repeated, data can be transferred.
When the device does not have hardware interface, it must constantly monitor the bus via software. Obviously, the more times the microcontroller monitors or polls the bus, the less time it can spend carrying out its intended function. There is therefore a speed difference between fast hardware devices and a relatively slow microcontroller which relies on software polling. In this case, data transfer can be preceded by a start procedure which is much longer than normal. The start procedure consists of:

- A START condition (S)
- A START byte (0000 0001)
- An acknowledge clock pulse (ACK)
- A repeated START condition (Sr).
I²C: START byte

After the START condition S has been transmitted by a master which requires bus access, the START byte (0000 0001) is transmitted. Another microcontroller can therefore sample the SDA line at a low sampling rate until one of the seven zeros in the START byte is detected. After detection of this LOW level on the SDA line, the microcontroller can switch to a higher sampling rate to find the repeated START condition Sr which is then used for synchronization. A hardware receiver resets upon receipt of the repeated START condition Sr and therefore ignores the START byte.

An acknowledge-related clock pulse is generated after the START byte. This is present only to conform with the byte handling format used on the bus. No device is allowed to acknowledge the START byte.
I^2C: Clock synchronization

Two masters can begin transmitting on a free bus at the same time and there must be a method for deciding which takes control of the bus and complete its transmission. This is done by clock synchronization and arbitration. In single master systems, clock synchronization and arbitration are not needed.

Clock synchronization is performed using the wired-AND connection of I^2C interfaces to the SCL line. This means that a HIGH to LOW transition on the SCL line causes the masters concerned to start counting off their LOW period and, once a master clock has gone LOW, it holds the SCL line in that state until the clock HIGH state is reached. However, if another clock is still within its LOW period, the LOW to HIGH transition of this clock may not change the state of the SCL line. The SCL line is therefore held LOW by the master with the longest LOW period. Masters with shorter LOW periods enter a HIGH wait-state during this time.
When all masters concerned have counted off their LOW period, the clock line is released and goes HIGH. There is then no difference between the master clocks and the state of the SCL line, and all the masters start counting their HIGH periods. The first master to complete its HIGH period pulls the SCL line LOW again. In this way, a synchronized SCL clock is generated with its LOW period determined by the master with the longest clock LOW period, and its HIGH period determined by the one with the shortest clock HIGH period.
Overview I2C Module: Atmel AVR Atmega32 (TWI- Two Wire Interface)
Overview I2C Module: Atmel AVR Atmega32 (TWI- Two Wire Interface)
Overview I2C Module:
Atmel AVR Atmega32 (TWI- Two Wire Interface)
Overview I2C Module: Atmel AVR Atmega32 (TWI- Two Wire Interface)
Overview I2C Module:
Atmel AVR Atmega32 (TWI- Two Wire Interface)
Overview I2C Module: STATUS REGISTER Atmel AVR Atmega32 - Status Codes for Master

### Transmitter Mode

<table>
<thead>
<tr>
<th>Status Code (TWSR)</th>
<th>Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>$08</td>
<td>A START condition has been transmitted</td>
</tr>
<tr>
<td>$10</td>
<td>A repeated START condition has been transmitted</td>
</tr>
<tr>
<td>$18</td>
<td>SLA+W has been transmitted; ACK has been received</td>
</tr>
<tr>
<td>$20</td>
<td>SLA+W has been transmitted; NOT ACK has been received</td>
</tr>
<tr>
<td>$28</td>
<td>Data byte has been transmitted; ACK has been received</td>
</tr>
<tr>
<td>$30</td>
<td>Data byte has been transmitted; NOT ACK has been received</td>
</tr>
<tr>
<td>$38</td>
<td>Arbitration lost in SLA+W or data bytes</td>
</tr>
</tbody>
</table>

### Receiver Mode

<table>
<thead>
<tr>
<th>Status Code (TWSR)</th>
<th>Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>$08</td>
<td>A START condition has been transmitted</td>
</tr>
<tr>
<td>$10</td>
<td>A repeated START condition has been transmitted</td>
</tr>
<tr>
<td>$28</td>
<td>Arbitration lost in SLA+W or NOT ACK bit</td>
</tr>
<tr>
<td>$40</td>
<td>SLA+R has been transmitted; ACK has been received</td>
</tr>
<tr>
<td>$48</td>
<td>SLA+R has been transmitted; NOT ACK has been received</td>
</tr>
<tr>
<td>$50</td>
<td>Data byte has been received; ACK has been returned</td>
</tr>
<tr>
<td>$58</td>
<td>Data byte has been received; NOT ACK has been returned</td>
</tr>
</tbody>
</table>
Overview I2C Module: STATUS REGISTER Atmel AVR Atmega32 - Status Codes for Slave

### Transmitter Mode

<table>
<thead>
<tr>
<th>Status Code (TWSR) Preselector Bits are 0</th>
<th>Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8A</td>
<td>Own SLA+R has been received; ACK has been returned</td>
</tr>
<tr>
<td>$B0</td>
<td>Arbitration lost in SLA+R/W as master; own SLA+R has been received; ACK has been returned</td>
</tr>
<tr>
<td>$B8</td>
<td>Data byte in TWDR has been transmitted; ACK has been received</td>
</tr>
<tr>
<td>$C0</td>
<td>Data byte in TWDR has been transmitted; NOT ACK has been received</td>
</tr>
<tr>
<td>$C8</td>
<td>Last data byte in TWDR has been transmitted (TWEA = &quot;0&quot;); ACK has been received</td>
</tr>
</tbody>
</table>

### Receiver Mode

<table>
<thead>
<tr>
<th>Status Code (TWSR) Preselector Bits are 0</th>
<th>Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>$60</td>
<td>Own SLA+W has been received; ACK has been returned</td>
</tr>
<tr>
<td>$68</td>
<td>Arbitration lost in SLA+R/W as master; own SLA+W has been received; ACK has been returned</td>
</tr>
<tr>
<td>$70</td>
<td>General Call address has been received; ACK has been returned</td>
</tr>
<tr>
<td>$78</td>
<td>Arbitration lost in SLA+R/W as master; General Call address has been received; ACK has been returned</td>
</tr>
<tr>
<td>$80</td>
<td>Previously addressed with own SLA+W; data has been received; ACK has been returned</td>
</tr>
<tr>
<td>$88</td>
<td>Previously addressed with own SLA+W; data has been received; NOT ACK has been returned</td>
</tr>
<tr>
<td>$90</td>
<td>Previously addressed with general call; data has been received; ACK has been returned</td>
</tr>
<tr>
<td>$98</td>
<td>Previously addressed with general call; data has been received; NOT ACK has been returned</td>
</tr>
<tr>
<td>$A0</td>
<td>A STOP condition or repeated START condition has been received while still addressed as slave</td>
</tr>
</tbody>
</table>

### Miscellaneous States

<table>
<thead>
<tr>
<th>Status Code (TWSR) Preselector Bits are 0</th>
<th>Status of the Two-wire Serial Bus and Two-wire Serial Interface Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F8</td>
<td>No relevant state information available; TWINT = &quot;0&quot;</td>
</tr>
<tr>
<td>$00</td>
<td>Bus error due to an illegal START or STOP condition</td>
</tr>
</tbody>
</table>
TWI in a Typical Transmission (Atmel AVR Atmega32)

C example:
TWCR = (1<<TWINT) | (1<<TWSTA) | (1<<TWEN); // Send START condition
TWI in a Typical Transmission (Atmel AVR Atmega32)

C example:

```c
while (!(TWCR & (1<<TWINT))); // Wait for TWINT Flag set. This indicates
//that the START condition has been
//transmitted
```

GUT - Intel 2015/16  Embedded Systems Development Technology 35
TWI in a Typical Transmission (Atmel AVR Atmega32)

C example:

```c
if (((TWSR & 0xF8) != START) ERROR(); // Check value of TWI Status Register.
    // Mask prescaler bits. If status is different
    // from START go to ERROR

TWDR = SLA_W;
TWCR = (1<<TWINT) | (1<<TWEN); // Load SLA_W into TWDR Register.
    // Clear TWINT bit in TWCR to start
    // transmission of address
```
TWI in a Typical Transmission (Atmel AVR Atmega32)

C example:

```c
while (!(TWCR & (1<<TWINT))); // Wait for TWINT Flag set. This indicates that the SLA+W has been transmitted, and ACK/NACK has been received.
```
TWI in a Typical Transmission (Atmel AVR Atmega32)

**C example:**

```c
if (((TWSR & 0xF8) != MT_SLA_ACK) )ERROR();  //Check value of TWI Status Register.
   //Mask prescaler bits. If status is different
   //from MT_SLA_ACK go to ERROR
TWDR = DATA;  //Load DATA into TWDR Register.
TWCR = (1<<TWINT) | (1<<TWEN);  //Clear TWINT bit in TWCR to start
   //transmission of data
```
C example:

```c
while (!(TWCR & (1<<TWINT))); // Wait for TWINT Flag set. This indicates
// that the DATA has been transmitted,
// and ACK/NACK has been received.
```
TWI in a Typical Transmission (Atmel AVR Atmega32)

**C example:**

```c
if (((TWSR & 0xF8) != MT_DATA_ACK) ERROR(); //Check value of TWI Status Register.
    //Mask prescaler bits. If status is different
    //from MT_DATA_ACK go to ERROR

    TWCR = (1<<TWINT)|(1<<TWEN)|(1<<TWSTO); //Transmit STOP condition
```

GUT - Intel 2015/16

Embedded Systems Development Technology 40
Example of I2C-bus applications
Example of clock and calendar with 240 x 8-bit RAM (PCF8583)
Digital Temperature Sensor and Thermal Watchdog with 2-Wire Interface (LM75)
Example of I²C 8-bit I/O expander (PCF8574)

PCF8574:
- 100 kHz I2C-bus interface (Standard-mode I2C-bus)
- Operating supply voltage 2.5 V to 6 V
- 8-bit remote I/O pins that default to inputs at power-up
- Latched outputs directly drive LEDs
- Total package sink capability of 80 mA
- Active LOW open-drain interrupt output
- Eight programmable slave addresses using three address pins
- Low standby current (2.5 μA typical)
- ESD protection exceeds 2000 V HBM per JESD22-A114 and 1000 V CDM per JESD22-C101
Example of 8-bit I/O expander (PCF8574)

<table>
<thead>
<tr>
<th>SYMBOL</th>
<th>PIN</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0</td>
<td>1</td>
<td>address input 0</td>
</tr>
<tr>
<td>A1</td>
<td>2</td>
<td>address input 1</td>
</tr>
<tr>
<td>A2</td>
<td>3</td>
<td>address input 2</td>
</tr>
<tr>
<td>P0</td>
<td>4</td>
<td>quasi-bidirectional I/O 0</td>
</tr>
<tr>
<td>P1</td>
<td>5</td>
<td>quasi-bidirectional I/O 1</td>
</tr>
<tr>
<td>P2</td>
<td>6</td>
<td>quasi-bidirectional I/O 2</td>
</tr>
<tr>
<td>P3</td>
<td>7</td>
<td>quasi-bidirectional I/O 3</td>
</tr>
<tr>
<td>VSS</td>
<td>8</td>
<td>supply ground</td>
</tr>
<tr>
<td>P4</td>
<td>9</td>
<td>quasi-bidirectional I/O 4</td>
</tr>
<tr>
<td>P5</td>
<td>10</td>
<td>quasi-bidirectional I/O 5</td>
</tr>
<tr>
<td>P6</td>
<td>11</td>
<td>quasi-bidirectional I/O 6</td>
</tr>
<tr>
<td>P7</td>
<td>12</td>
<td>quasi-bidirectional I/O 7</td>
</tr>
<tr>
<td>INT</td>
<td>13</td>
<td>interrupt output (active LOW)</td>
</tr>
<tr>
<td>SCL</td>
<td>14</td>
<td>serial clock line</td>
</tr>
<tr>
<td>SDA</td>
<td>15</td>
<td>serial data line</td>
</tr>
<tr>
<td>VDD</td>
<td>16</td>
<td>supply voltage</td>
</tr>
</tbody>
</table>
Example of I²C 8-bit I/O expander (PCF8574)

Simplified schematic diagram of each I/O.
A quasi-bidirectional I/O is an input or output port without using a direction control register. Whenever the master reads the register, the value returned to master depends on the actual voltage or status of the pin.

At power on, all the ports are HIGH with a weak 100uA internal pull-up to VDD, but can be driven LOW by an internal transistor, or an external signal. The I/O ports are entirely independent of each other, but each I/O octal is controlled by the same read or write data byte.

Advantages of the quasi-bidirectional I/O over totem pole I/O include:

- Better for driving LEDs since the p-channel (transistor to VDD) is small, which saves die size and therefore cost. LED drive only requires an internal transistor to ground, while the LED is connected to VDD through a current-limiting resistor. Totem pole I/O have both n-channel and p-channel transistors, which allow solid HIGH and LOW output levels without a pull-up resistor — good for logic levels.

- Simpler architecture — only a single register and the I/O can be both input and output at the same time. Totem pole I/O have a direction register that specifies the port pin direction and it is always in that configuration unless the direction is explicitly changed.

- Does not require a command byte. The simplicity of one register (no need for the pointer register or, technically, the command byte) is an advantage in some embedded systems where every byte counts because of memory or bandwidth limitations.”
Example of I²C 8-bit I/O expander (PCF8574)

There is only one register to control four possibilities of the port pin:

- **Input HIGH**: The master needs to write 1 to the register to set the port as an input mode if the device is not in the default power-on condition. The master reads the register to check the input status. If the external source pulls the port pin up to VDD or drives logic 1, then the master will read the value of 1.
- **Input LOW**: The master needs to write 1 to the register to set the port to input mode if the device is not in the default power-on condition. The master reads the register to check the input status. If the external source pulls the port pin down to VSS or drives logic 0, which sinks the weak 100 uA current source, then the master will read the value of 0.
Example of I²C 8-bit I/O expander (PCF8574)

There is only one register to control four possibilities of the port pin:

- **Output HIGH**: The master writes 1 to the register. There is an additional ‘accelerator’ or strong pull-up current when the master sets the port HIGH. The additional strong pull-up is only active during the HIGH time of the acknowledge clock cycle. This accelerator current helps the port’s 100 μA current source make a faster rising edge into a heavily loaded output, but only at the start of the acknowledge clock cycle to avoid bus contention if an external signal is pulling the port LOW to VSS/driving the port with logic 0 at the same time. After the half clock cycle there is only the 100 μA current source to hold the port HIGH.
- **Output LOW**: The master writes 0 to the register. There is a strong current sink transistor that holds the port pin LOW. A large current may flow into the port, which could potentially damage the part if the master writes a 0 to the register and an external source is pulling the port HIGH at the same time.
Example of I²C 8-bit I/O expander (PCF8574)

Application of multiple PCF8574s with interrupt:

An interrupt is generated by any rising or falling edge of the port inputs in the input mode. After time $t_{iv}$ the signal INT is valid.
Example of I²C 8-bit I/O expander (PCF8574)

PCF 8574: WRITE mode (output):
Example of I²C 8-bit I/O expander (PCF8574)

PCF 8574: READ mode (input).
Example of I²C 8-bit I/O expander (PCF8574)

Typical Application
Example of I²C 8-bit I/O expander (PCF8574)

Arduino example 1:

```c
#include <PCF8574.h>
#include <Wire.h>

PCF8574 expander;

void setup()
{
    expander.begin(0x20);
    expander.pinMode(4, OUTPUT);
}

void loop()
{
    expander.digitalWrite(4, LOW);
    delay(1000);
    expander.digitalWrite(4, HIGH);
    delay(1000);
}
```
Example of I²C 8-bit I/O expander (PCF8574)

Arduino example 2:

```cpp
#include <PCF8574.h>
#include <Wire.h>

PCF8574 expander;

void setup()
{
  Serial.begin(9600);
  expander.begin(0x20);
  expander.pinMode(0, INPUT);
}

void loop()
{
  byte value = expander.digitalRead(0);
  Serial.println(value, DEC);
  delay(100);

```
Example of I²C 8-bit I/O expander (PCF8574)

Arduino example 3:

```cpp
#include <PCF8574.h>
#include <Wire.h>

PCF8574 expander;

void setup()
{
    Serial.begin(9600);

    expander.begin(0x20);
    expander.pinMode(4, INPUT);
    expander.pullUp(4);
}

void loop()
{
    byte value = expander.digitalRead(4);

    Serial.println(value, DEC);
    delay(100);
}
```
SPI- Serial Peripheral Interface
SPI: Overview

- Serial Peripheral Interface (SPI) is a 4-wire full-duplex synchronous serial data link developed by Motorola.
- Used for connecting peripherals to each other and to microprocessors.
- Usually “3 + n” wire interface (with n = number of devices) are needed.
- Only one master active at a time, which generates a clock signal.
- Various Speed transfers (function of the system clock).
- Serial clock with programmable polarity and phase.
- SPI provides support for a high bandwidth (1 mega baud) network connection amongst CPUs and other devices supporting the SPI.
• MOSI - This pin is used to transmit data out of the SPI module when it is configured as a Master and receive data when it is configured as Slave.

• MISO - This pin is used to transmit data out of the SPI module when it is configured as a Slave and receive data when it is configured as Master.

• /SS - This pin is used to output the select signal from the SPI module to another peripheral with which a data transfer is to take place when its configured as a Master and its used as an input to receive the slave select signal when the SPI is configured as Slave.

• SCLK - This pin is used to output the clock with respect to which the SPI transfers data or receive clock in case of Slave.
SPI: External Signal Alternative naming

- MOSI - SIMO, SDO (for master devices), SDI (for slave devices), DI, DIN, SI, MTST.
- MISO - SOMI, SDO (for slave devices), SDI (for master devices), DO, DOUT, SO, MRSR.
- /SS - nCS, CS, CSB, CSN, EN, nSS, STE, SYNC.
- SCLK - SCK, CLK.
SPI: Configurations of bus connection:

- Single master and single slave

- Master and n-independent slaves (n=3)

- Daisy-chained SPI bus with master and cooperative slaves
SPI: An example of transmission
Overview SPI Module: Atmel AVR Atmega32

• Full-duplex, Three-wire Synchronous Data Transfer
• Master or Slave Operation
• LSB First or MSB First Data Transfer
• Seven Programmable Bit Rates
• End of Transmission Interrupt Flag
• Write Collision Flag Protection
• Wake-up from Idle Mode
• Double Speed (CK/2) Master SPI Mode
Overview SPI Module: Atmel AVR Atmega32

**Master Mode:**

- When configured as a Master, the SPI interface has no automatic control of the #SS line. This must be handled by user software before communication can start.
- Writing a byte to the SPI Data Register starts the SPI clock generator, and the hardware shifts the eight bits into the Slave.
- After shifting one byte, the SPI clock generator stops, setting the end of Transmission Flag (SPIF).
- If the SPI Interrupt Enable bit (SPIE) in the SPCR Register is set, an interrupt is requested.
- The Master may continue to shift the next byte by writing it into SPDR, or signal the end of packet by pulling high the Slave Select, SS line. The last incoming byte will be kept in the Buffer Register for later use.
Slave Mode:

- When configured as a Slave, the SPI interface will remain sleeping with MISO tri-stated as long as the #SS pin is driven high.
- Software may update the contents of the SPI Data Register, SPDR, but the data will not be shifted out by incoming clock pulses on the SCK pin until the SS pin is driven low.
- As one byte has been completely shifted, the end of Transmission Flag, SPIF is set.
- If the SPI Interrupt Enable bit, SPIE, in the SPCR Register is set, an interrupt is requested.
- The Slave may continue to place new data to be sent into SPDR before reading the incoming data. The last incoming byte will be kept in the Buffer Register for later use.

- In SPI Slave mode, the control logic will sample the incoming signal of the SCK pin. To ensure correct sampling of the clock signal, the minimum low and high periods should be longer than 2 CPU clock cycles.
Overview SPI Module: Atmel AVR Atmega32 as a Master

**C example:**

```c
void SPI_MasterInit(void)
{
    // Set MOSI , SCK output, all others inputs
    DDR_SPI = (1<<DD_MOSI)|(1<<DD_SCK);

    // Enable SPI, Master, set clock rate fck/16
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0);
}

void SPI_MasterTransmit(char cData)
{
    // Start transmission
    SPDR = cData;

    //Wait for transmission complete
    while(!(SPSR & (1<<SPIF)));
}
```
Overview SPI Module: Atmel AVR Atmega32 as a Slave

C example:

```c
void SPI_SlaveInit(void)
{
    // Set MISO output, all others input
    DDR_SPI = (1<<DD_MISO);
    // Enable SPI
    SPCR = (1<<SPE);
}

char SPI_SlaveReceive(void)
{
    // Wait for reception complete
    while(!((SPSR & (1<<SPIF))));
    // Return data register
    return SPDR;
}
```
SS Pin Functionality:

- When configured as a Slave:
  - The Slave Select (SS) pin is always input. When SS is held low, the SPI is activated, and MISO becomes an output if configured so by the user. All other pins are inputs. When SS is driven high, all pins are inputs except MISO which can be user configured as an output, and the SPI is passive, which means that it will not receive incoming data.
  - The SS pin is useful for packet/byte synchronization to keep the slave bit counter synchronous with the master clock generator. When the SS pin is driven high, the SPI Slave will immediately reset the send and receive logic, and drop any partially received data in the Shift Register.

- When the SPI is configured as a Master (MSTR in SPCR is set):
  - The user can determine the direction of the SS pin. If SS is configured as an output, the pin is a general output pin which does not affect the SPI system. Typically, the pin will be driving the SS pin of the SPI Slave.
  - If SS is configured as an input, it must be held high to ensure Master SPI operation. If the SS pin is driven low by peripheral circuitry when the SPI is configured as a Master with the SS pin defined as an input, the SPI system interprets this as another master selecting the SPI as a slave and starting to send data to it.
Overview SPI Module: Atmel AVR Atmega32- SPI Control Register – SPCR

Bit 7 – SPIE: SPI Interrupt Enable
Bit 6 – SPE: SPI Enable
Bit 5 – DORD: Data Order
Bit 4 – MSTR: Master/Slave Select

Bits 1, 0 – SPR1, SPR0: SPI Clock Rate Select 1 and 0
Overview SPI Module:
Atmel AVR Atmega32- SPI Control Register – SPCR
Data Mode

Bit 3 – CPOL: Clock Polarity

Bit 2 – CPHA: Clock Phase

<table>
<thead>
<tr>
<th>Leading Edge</th>
<th>Trailing Edge</th>
<th>SPI Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPOL = 0, CPHA = 0</td>
<td>Sample (Rising)</td>
<td>Setup (Falling)</td>
</tr>
<tr>
<td>CPOL = 0, CPHA = 1</td>
<td>Setup (Rising)</td>
<td>Sample (Falling)</td>
</tr>
<tr>
<td>CPOL = 1, CPHA = 0</td>
<td>Sample (Falling)</td>
<td>Setup (Rising)</td>
</tr>
<tr>
<td>CPOL = 1, CPHA = 1</td>
<td>Setup (Falling)</td>
<td>Sample (Rising)</td>
</tr>
</tbody>
</table>
Overview SPI Module:
Atmel AVR Atmega32- SPI
Control Register – SPCR
Data Mode
Bit 7 – SPIF: SPI Interrupt Flag is set When a serial transfer is complete

Bit 6 – WCOL: Write COLLision Flag is set if the SPI Data Register (SPDR) is written during a data transfer

Bit 5..1 – Reserved Bits

Bit 0 – SPI2X: Double SPI Speed Bit: When this bit is written logic one the SPI speed (SCK Frequency) will be doubled when the SPI is in Master mode
Example of SPI Serial EEPROM (AT25010B)

Features:
- Serial Peripheral Interface (SPI) Compatible
- Supports SPI Modes 0 (0,0) and 3 (1,1)
- Low-voltage and Standard-voltage Operation VCC = 1.8V to 5.5V
- 20MHz Clock Rate (5V)
- 8-byte Page Mode
- Block Write Protection - Protect 1/4, 1/2, or Entire Array
- Write Protect (WP) Pin and Write Disable Instructions for Both Hardware and Software Data Protection
- High Reliability Endurance: 1,000,000 Write Cycle, Data Retention: 100 Years
Example of SPI Serial EEPROM (AT25010B)
Features:
- 8-bit remote bidirectional I/O port (default input)
- High-speed SPI interface (MCP23S08) - 10MHz
- Hardware address pins - Two for the MCP23S08 to allow up to four devices using the same chip-select
- Configurable interrupt output pin - Configurable as active-high, active-low or open-drain
- Configurable interrupt source - Interrupt-on-change from configured defaults or pin change
- Polarity Inversion register to configure the polarity of the input port data
- External reset input
- Low standby current: 1 µA (max.)
- Operating voltage:
  - 1.8V to 5.5V SPI @ 5 MHz
  - 2.7V to 5.5V SPI @ 10 MHz
RS232(UART)
RS232 (1962) - is a standard for serial communication transmission of data between DTE (data terminal equipment) and a DCE (data communication equipment).

UART - Universal Asynchronous Receiver and Transmitter
## Communication Interfaces: RS232(UART)

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Dir</th>
<th>Notes/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>Protective/shielded ground</td>
</tr>
<tr>
<td>2</td>
<td>TD</td>
<td>OUT</td>
<td>Transmit Data (a.k.a TxD, Tx) (ASYNC)</td>
</tr>
<tr>
<td>3</td>
<td>RD</td>
<td>IN</td>
<td>Receive Data (a.k.a RxD, Rx) (ASYNC)</td>
</tr>
<tr>
<td>4</td>
<td>RTS</td>
<td>OUT</td>
<td>Request To Send (ASYNC)</td>
</tr>
<tr>
<td>5</td>
<td>CTS</td>
<td>IN</td>
<td>Clear To Send (ASYNC)</td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
<td>IN</td>
<td>Data Set Ready (ASYNC)</td>
</tr>
<tr>
<td>7</td>
<td>SGND</td>
<td>-</td>
<td>Signal Ground</td>
</tr>
<tr>
<td>8</td>
<td>CD</td>
<td>IN</td>
<td>Carrier Detect (a.k.a DCD).</td>
</tr>
<tr>
<td>9</td>
<td>-</td>
<td>-</td>
<td>Reserved for data set testing.</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>-</td>
<td>Reserved for data set testing.</td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>-</td>
<td>Unassigned</td>
</tr>
<tr>
<td>12</td>
<td>SDCD</td>
<td>IN</td>
<td>Secondary Carrier Detect. Only needed if second channel being used.</td>
</tr>
<tr>
<td>13</td>
<td>SCTS</td>
<td>IN</td>
<td>Secondary Clear to send. Only needed if second channel being used.</td>
</tr>
<tr>
<td>14</td>
<td>STD</td>
<td>OUT</td>
<td>Secondary Transmit Data. Only needed if second channel being used.</td>
</tr>
<tr>
<td>15</td>
<td>DB</td>
<td>OUT</td>
<td>Transmit Clock (a.k.a TCLK, TxCLK). Synchronous use only.</td>
</tr>
<tr>
<td>16</td>
<td>SRD</td>
<td>IN</td>
<td>Secondary Receive Data. Only needed if second channel being used.</td>
</tr>
<tr>
<td>17</td>
<td>DD</td>
<td>IN</td>
<td>Receive Clock (a.k.a. RCLK). Synchronous use only.</td>
</tr>
<tr>
<td>18</td>
<td>LL</td>
<td>-</td>
<td>Local Loopback</td>
</tr>
<tr>
<td>19</td>
<td>SRTS</td>
<td>OUT</td>
<td>Secondary Request to Send. Only needed if second channel being used.</td>
</tr>
<tr>
<td>20</td>
<td>DTR</td>
<td>OUT</td>
<td>Data Terminal Ready. (ASYNC)</td>
</tr>
<tr>
<td>21</td>
<td>RL/SQ</td>
<td>-</td>
<td>Signal Quality Detector/Remote loopback</td>
</tr>
<tr>
<td>22</td>
<td>RI</td>
<td>IN</td>
<td>Ring Indicator. DCE (Modem) raises when incoming call detected used for auto answer applications.</td>
</tr>
<tr>
<td>23</td>
<td>CH/CI</td>
<td>OUT</td>
<td>Signal Rate selector.</td>
</tr>
<tr>
<td>24</td>
<td>DA</td>
<td>-</td>
<td>Auxiliary Clock (a.k.a. ACLK). Secondary Channel only.</td>
</tr>
<tr>
<td>25</td>
<td>-</td>
<td>-</td>
<td>Unassigned</td>
</tr>
</tbody>
</table>
## Communication Interfaces: RS232(UART)

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Dir</th>
<th>Notes/Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DCD</td>
<td>IN</td>
<td>Data Carrier Detect. Raised by DCE when modem synchronized.</td>
</tr>
<tr>
<td>2</td>
<td>RD</td>
<td>IN</td>
<td>Receive Data (a.k.a RxD, Rx). Arriving data from DCE.</td>
</tr>
<tr>
<td>3</td>
<td>TD</td>
<td>OUT</td>
<td>Transmit Data (a.k.a TxD, Tx). Sending data from DTE.</td>
</tr>
<tr>
<td>4</td>
<td>DTR</td>
<td>OUT</td>
<td>Data Terminal Ready. Raised by DTE when powered on. In auto-answer mode raised only when RI arrives from DCE.</td>
</tr>
<tr>
<td>5</td>
<td>SGND</td>
<td>-</td>
<td>Ground</td>
</tr>
<tr>
<td>6</td>
<td>DSR</td>
<td>IN</td>
<td>Data Set Ready. Raised by DCE to indicate ready.</td>
</tr>
<tr>
<td>7</td>
<td>RTS</td>
<td>OUT</td>
<td>Request To Send. Raised by DTE when it wishes to send. Expects CTS from DCE.</td>
</tr>
<tr>
<td>8</td>
<td>CTS</td>
<td>IN</td>
<td>Clear To Send. Raised by DCE in response to RTS from DTE.</td>
</tr>
<tr>
<td>9</td>
<td>RI</td>
<td>IN</td>
<td>Ring Indicator. Set when incoming ring detected - used for auto-answer application. DTE raised DTR to answer.</td>
</tr>
</tbody>
</table>
## Communication Interfaces: RS232(UART)

<table>
<thead>
<tr>
<th>SPECIFICATIONS</th>
<th>RS232</th>
<th>RS423</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode of Operation</td>
<td>SINGLE-ENDED</td>
<td>SINGLE-ENDED</td>
</tr>
<tr>
<td>Total Number of Drivers and Receivers on One Line</td>
<td>1 DRIVER, 1 RECVR</td>
<td>1 DRIVER, 10 RECVR</td>
</tr>
<tr>
<td>Maximum Cable Length</td>
<td>50 FT. (15m)</td>
<td>4000 FT. (1200m)</td>
</tr>
<tr>
<td>Maximum Data Rate</td>
<td>20kb/s</td>
<td>100kb/s</td>
</tr>
<tr>
<td>Maximum Driver Output Voltage</td>
<td>+/-25V</td>
<td>+/-6V</td>
</tr>
<tr>
<td>Driver Output Signal Level (Loaded Min.)</td>
<td>+/-5V to +/-15V</td>
<td>+/-3.6V</td>
</tr>
<tr>
<td>Driver Output Signal Level (Unloaded Max)</td>
<td>+/-25V</td>
<td>+/-6V</td>
</tr>
<tr>
<td>Driver Load Impedance (Ohms)</td>
<td>3k to 7k</td>
<td>&gt;=450</td>
</tr>
<tr>
<td>Max. Driver Current in High Z State</td>
<td>Power On N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Max. Driver Current in High Z State</td>
<td>Power Off +/-6mA @ +/-2v</td>
<td>+/-100uA</td>
</tr>
<tr>
<td>Slew Rate (Max.)</td>
<td>30V/µS</td>
<td>Adjustable</td>
</tr>
<tr>
<td>Receiver Input Voltage Range</td>
<td>+/-15V</td>
<td>+/-12V</td>
</tr>
<tr>
<td>Receiver Input Sensitivity</td>
<td>+/-3V</td>
<td>+/-200mV</td>
</tr>
<tr>
<td>Receiver Input Resistance (Ohms)</td>
<td>3k to 7k</td>
<td>4k min.</td>
</tr>
</tbody>
</table>
Communication Interfaces: RS232(UART)

http://www.arcelect.com/rs232.htm
Overview USART Module: Atmel AVR Atmega32

- Full Duplex Operation (Independent Serial Receive and Transmit Registers)
- Asynchronous or Synchronous Operation
- Master or Slave Clocked Synchronous Operation
- High Resolution Baud Rate Generator
- Supports Serial Frames with 5, 6, 7, 8, or 9 Data Bits and 1 or 2 Stop Bits
- Odd or Even Parity Generation and Parity Check Supported by Hardware
- Data OverRun Detection
- Framing Error Detection
- Noise Filtering Includes False Start Bit Detection and Digital Low Pass Filter
- Three Separate Interrupts on TX Complete, TX Data Register Empty, and RX Complete
- Multi-processor Communication Mode
- Double Speed Asynchronous Communication Mode
The USART in Atmel AVR Atmega accepts all 30 combinations of frame formats:

- 1 start bit
- 5, 6, 7, 8, or 9 data bits
- no, even or odd parity bit
- 1 or 2 stop bits
Overview USART Module: Atmel AVR Atmega32

**USART initialization:**
- setting the baud rate,
- setting frame format and
- enabling the transmitter or the receiver
- For interrupt driven USART operation, the Global Interrupt Flag should be cleared (and interrupts globally disabled) when doing the initialization.

```
void USART_Init( unsigned int baud )
{
    /* Set baud rate */
    UBRRH = (unsigned char)(baud>>8);
    UBRRL = (unsigned char)baud;
    /* Enable receiver and transmitter */
    UCSRB = (1<<RXEN)|(1<<TXEN);
    /* Set frame format: 8data, 2stop bit */
    UCSRC = (1<<URSEL)|(1<<USBS)|(3<<UCSZ0);
}
```

**ARDUINO:**
- Serial.begin(speed)
- Serial.begin(speed, config)
Data Transmission (polling of the Data Register Empty (UDRE) Flag):
- Transmitter is enabled by setting the Transmit Enable (TXEN) bit in the UCSRB Register.
- When the Transmitter is enabled, the normal port operation of the TxD pin is overridden by the USART and given the function as the transmitter’s serial output.
- The baud rate, mode of operation and frame format must be set up once before doing any transmissions.
- A data transmission is initiated by loading the transmit buffer with the data to be transmitted.
- The buffered data in the transmit buffer will be moved to the Shift Register when the Shift Register is ready to send a new frame.

```c
void USART_Transmit( unsigned char data ) {
  /* Wait for empty transmit buffer */
  while ( !( UCSRA & (1<<UDRE)) ) ;
  /* Put data into buffer, sends the data */
  UDR = data;
}

ARDUINO:
Serial.write(val)
Serial.write(str)
Serial.write(buf, len)
Serial.print(val)
Serial.print(val, format)
```
Overview USART Module: Atmel AVR Atmega32

Data Reception polling of the Receive Complete (RXC) Flag:

- Receiver is enabled by writing the Receive Enable (RXEN) bit in the UCSRB Register to one,
- When the receiver is enabled, the normal pin operation of the RxD pin is overridden by the USART and given the function as the receiver’s serial input.
- The baud rate, mode of operation and frame format must be set up once before any serial reception can be done.
- The receiver starts data reception when it detects a valid start bit.
- Each bit that follows the start bit will be sampled at the baud rate (or XCK clock), and shifted into the receive Shift Register until the first stop bit of a frame is received. A second stop bit will be ignored by the receiver.

```c
unsigned char USART_Receive( void ) {
    /* Wait for data to be received */
    while ( !(UCSRA & (1<<RXC)) );
    /* Get and return received data from buffer */
    return UDR;
}
```

**ARDUINO:**
Serial.read()
Serial.readBytes(buffer, length)
Serial.readString()
Serial.readStringUntil(terminator)
The USART Receiver has three Error Flags:

- **Frame Error (FE):** Flag indicates the state of the first stop bit of the next readable frame stored in the receive buffer. The FE Flag is zero when the stop bit was correctly read (as one), and the FE Flag will be one when the stop bit was incorrect (zero). This flag can be used for detecting out-of-sync conditions, detecting break conditions and protocol handling.

- **Data OverRun (DOR):** Flag indicates data loss due to a receiver buffer full condition. A Data OverRun occurs when the receive buffer is full (two characters), it is a new character waiting in the receive Shift Register, and a new start bit is detected. If the DOR Flag is set there was one or more serial frame lost between the frame last read from UDR, and the next frame read from UDR.

- **Parity Error (PE):** Flag indicates that the next frame in the receive buffer had a parity error when received. If parity check is not enabled the PE bit will always be read zero.
Communication Interfaces: RS232(UART)

Communication Interfaces: RS232(UART)
Communication Interfaces: RS232(UART)

```cpp
// output, even if you don't use it:
pinMode(10, OUTPUT);

// see if the card is present and can be initialized:
if (!SD.begin(chipSelect)) {
    Serial.println("Card failed, or not present");
    // don't do anything more:
    return;
}
Serial.println("card initialized.");

void loop()
{
    // make a string for assembling the data to log:
    String dataString = "";

    // read three sensors and append to the string:
    for (int analogPin = 0; analogPin < 3; analogPin++) {
        int sensor = analogRead(analogPin);
        dataString += sensor;
    }

    // log the data string to the serial monitor:
    Serial.println(dataString);
}
```
Communication Interfaces: RS232(UART)/USB

UART port in Galileo Gen2

FTDI cable
Communication Interfaces: RS232(UART)/USB

FTDI 232R
Communication Interfaces: RS232(UART)/USB

FTDI 232R features:

- Single USB chip to asynchronous serial data transfer interface.
- Entire USB protocol handled on the chip. No USB specific firmware programming required.
- Fully integrated 1024 bit EEPROM storing device descriptors and CBUS I/O configuration.
- Fully integrated USB termination resistors.
- Fully integrated clock generation with no external crystal required plus optional clock output selection enabling a glue-less interface to external MCU or FPGA.
- Data transfer rates from 300 baud to 3 Mbaud (RS422, RS485, RS232) at TTL levels.
- Unique USB FTDIChip-ID™ feature.
- Configurable CBUS I/O pins.
- Transmit and receive LED drive signals.
- UART interface support for 7 or 8 data bits, 1 or 2 stop bits and odd / even / mark / space / no parity
- FIFO receive and transmit buffers for high data throughput.

- Synchronous and asynchronous bit bang interface options with RD# and WR# strobes.
- Device supplied pre-programmed with unique USB serial number.
- Integrated +3.3V level converter for USB I/O.
- Integrated level converter on UART and CBUS for interfacing to between +1.8V and +5V logic.
- True 5V/3.3V/2.8V/1.8V CMOS drive output and TTL input.
- Configurable I/O pin output drive strength.
- Integrated power-on-reset circuit.
- Fully integrated AVCC supply filtering - no external filtering required.
- UART signal inversion option.
- +3.3V (using external oscillator) to +5.25V (internal oscillator) Single Supply Operation.
- USB 2.0 Full Speed compatible.
- Available in compact Pb-free 28 Pin SSOP and QFN-32 packages (both RoHS compliant).
**Communication Interfaces: RS232(UART)/USB**

**FTDI 232R pinouts**

### Power and Ground Group

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>VCCIO</td>
<td>PWR</td>
<td>+1.8V to +5.25V supply to the UART Interface and CBUS group pins (1...3, 5, 6, 9...14, 22, 23). In USB bus powered designs connect this pin to 3V3OUT pin to drive out at +3.3V levels, or connect to VCC to drive out at 5V CMOS level. This pin can also be supplied with an external +1.8V to +2.8V supply in order to drive outputs at lower levels. It should be noted that in this case this supply should originate from the same source as the supply to VCC. This means that in bus powered designs a regulator which is supplied by the +5V on the USB bus should be used.</td>
</tr>
<tr>
<td>7, 18, 21</td>
<td>GND</td>
<td>PWR</td>
<td>Device ground supply pins</td>
</tr>
<tr>
<td>17</td>
<td>3V3OUT</td>
<td>Output</td>
<td>+3.3V output from integrated LDO regulator. This pin should be decoupled to ground using a 100nF capacitor. The main use of this pin is to provide the internal +3.3V supply to the USB transceiver cell and the internal 1.5kΩ pull up resistor on USBDP. Up to 50mA can be drawn from this pin to power external logic if required. This pin can also be used to supply the VCCIO pin.</td>
</tr>
<tr>
<td>20</td>
<td>VCC</td>
<td>PWR</td>
<td>+3.3V to +5.25V supply to the device core. (see Note 1)</td>
</tr>
<tr>
<td>25</td>
<td>AGND</td>
<td>PWR</td>
<td>Device analogue ground supply for internal clock multiplier</td>
</tr>
</tbody>
</table>

### USB Interface Group

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>USBDP</td>
<td>I/O</td>
<td>USB Data Signal Plus, incorporating internal series resistor and 1.5kΩ pull up resistor to 3.3V.</td>
</tr>
<tr>
<td>16</td>
<td>USBDM</td>
<td>I/O</td>
<td>USB Data Signal Minus, incorporating internal series resistor.</td>
</tr>
</tbody>
</table>

### Miscellaneous Signal Group

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>RESET#</td>
<td>Input</td>
<td>Active low reset pin. This can be used by an external device to reset the FT232R. If not required can be left unconnected, or pulled up to VCC.</td>
</tr>
<tr>
<td>26</td>
<td>TEST</td>
<td>Input</td>
<td>Puts the device into IC test mode. Must be tied to GND for normal operation, otherwise the device will appear to fail.</td>
</tr>
<tr>
<td>27</td>
<td>OSCI</td>
<td>Input</td>
<td>Input 12MHz Oscillator Cell. Optional – Can be left unconnected for normal operation. (see Note 2)</td>
</tr>
<tr>
<td>28</td>
<td>OSCO</td>
<td>Output</td>
<td>Output from 12MHz Oscillator Cell. Optional – Can be left unconnected for normal operation if internal Oscillator is used. (see Note 2)</td>
</tr>
</tbody>
</table>
Communication Interfaces: RS232(UART)/USB

FTDI 232R pinouts

<table>
<thead>
<tr>
<th>Pin No.</th>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TXD</td>
<td>Output</td>
<td>Transmit Asynchronous Data Output.</td>
</tr>
<tr>
<td>2</td>
<td>DTR#</td>
<td>Output</td>
<td>Data Terminal Ready Control Output / Handshake Signal.</td>
</tr>
<tr>
<td>3</td>
<td>RTS#</td>
<td>Output</td>
<td>Request to Send Control Output / Handshake Signal.</td>
</tr>
<tr>
<td>5</td>
<td>RXD</td>
<td>Input</td>
<td>Receiving Asynchronous Data Input.</td>
</tr>
<tr>
<td>6</td>
<td>RI#</td>
<td>Input</td>
<td>Ring Indicator Control Input. When remote wake up is enabled in the internal EEPROM taking RI# low (20ms active low pulse) can be used to resume the PC USB host controller from suspend.</td>
</tr>
<tr>
<td>9</td>
<td>DSR#</td>
<td>Input</td>
<td>Data Set Ready Control Input / Handshake Signal.</td>
</tr>
<tr>
<td>10</td>
<td>DCD#</td>
<td>Input</td>
<td>Data Carrier Detect Control Input.</td>
</tr>
<tr>
<td>11</td>
<td>CTS#</td>
<td>Input</td>
<td>Clear To Send Control Input / Handshake Signal.</td>
</tr>
<tr>
<td>12</td>
<td>CBUS4</td>
<td>I/O</td>
<td>Configurable CBUS output only Pin. Function of this pin is configured in the device internal EEPROM. Factory default configuration is SLEEP#. See CBUS Signal Options, Table 3.9.</td>
</tr>
<tr>
<td>13</td>
<td>CBUS2</td>
<td>I/O</td>
<td>Configurable CBUS I/O Pin. Function of this pin is configured in the device internal EEPROM. Factory default configuration is TXDEN. See CBUS Signal Options, Table 3.9.</td>
</tr>
<tr>
<td>14</td>
<td>CBUS3</td>
<td>I/O</td>
<td>Configurable CBUS I/O Pin. Function of this pin is configured in the device internal EEPROM. Factory default configuration is PWREN#. See CBUS Signal Options, Table 3.9. PWREN# should be used with a 10kΩ resistor pull up.</td>
</tr>
<tr>
<td>22</td>
<td>CBUS1</td>
<td>I/O</td>
<td>Configurable CBUS I/O Pin. Function of this pin is configured in the device internal EEPROM. Factory default configuration is RXLED#. See CBUS Signal Options, Table 3.9.</td>
</tr>
<tr>
<td>23</td>
<td>CBUS0</td>
<td>I/O</td>
<td>Configurable CBUS I/O Pin. Function of this pin is configured in the device internal EEPROM. Factory default configuration is TXLED#. See CBUS Signal Options, Table 3.9.</td>
</tr>
</tbody>
</table>

GUT - Intel 2015/16
Communication Interfaces: RS232(UART)/USB

FTDI 232R- USB to RS232 Converter
Communication Interfaces: RS232(UART)/USB

FTDI 232R- USB to MCU UART Interface
USB – Universal Serial Bus
## Communication Interfaces: USB

### USB assumptions:
- Easy to connect to PC peripherals;
- Enable adding new classes of devices that increase PC capabilities;
- Low cost of transmission speed to 12Mb/s (USB1.1) or 480Mb/s (USB 2.0);
- Full compliance to transmit data in real-time voice, audio, and compressed video sequences;
- Flexible protocol with mixed transmission type (data Isochronous, asynchronous messages);
- Providing a standard interface, suitable for high-speed implementation of new and existing products;
- Plug&Play, power supply providing

<table>
<thead>
<tr>
<th>Release name</th>
<th>Release date</th>
<th>Speed and max signalling rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB 0.8</td>
<td>December 1994</td>
<td></td>
</tr>
<tr>
<td>USB 0.9</td>
<td>April 1995</td>
<td></td>
</tr>
<tr>
<td>USB 0.99</td>
<td>August 1995</td>
<td>Low Speed (1.5 Mbit/s), Full Speed (12 Mbit/s)</td>
</tr>
<tr>
<td>USB 1.0</td>
<td>January 1996</td>
<td></td>
</tr>
<tr>
<td>USB 1.1</td>
<td>August 1998</td>
<td></td>
</tr>
<tr>
<td>USB 2.0</td>
<td>April 2000</td>
<td>High Speed (480 Mbit/s)</td>
</tr>
<tr>
<td>USB 3.0</td>
<td>November 2008</td>
<td>SuperSpeed (5 Gbit/s)</td>
</tr>
<tr>
<td>USB 3.1</td>
<td>July 2013</td>
<td>SuperSpeed+ (10 Gbit/s)</td>
</tr>
</tbody>
</table>
The structure of the USB interface:

- USB connection network system can be extended by creating a tree structure;
- Each of USB bus connection directly connects the two devices: two hubs or one hub with one USB device.
- The system can be expanded to a maximum of 7 levels (max 127 devices).
- The cable directly connecting two USB devices, consists of two signal lines to transmit data (D+ and D-) and two additional wires (VBus and GND), which are used to transmit power (<100mA or <500mA)
**Power Supply for devices:**
- Each new connected device can consume a maximum of 100 mA.
- The limit can be increased to 500 mA when the host accepts the request to provide full power.
- A longer cord causes larger voltage drop.
The structure of the USB interface:
- USB standard also defines the types of connectors

Communication Interfaces: USB
The communication protocol:

- Communication USB standard consists of sending packets, which are divided into:
  - **tokens** - they determine character of the information (control or data), marked the beginning of a frame and manage the bus.
  - **packet of data** - information is saved to the device or read from the device.
  - **Acknowledgment packed** - contains information about receives data or control instructions and the events that occur during transmission: confirmation positive and negative, internal errors.

- Sending and receiving data are placed in the so-called endpoints (endpoints) that are actually buffers in the form of a FIFO queue located in the memory.

- The data is written or read from the endpoints in a sequential manner. Due to the adopted way of addressing the device may have a maximum of 32 endpoints (16 inputs and 16 outputs).

- USB specification requires that each device was serviced Endpoint 0 performing control functions. It is the only one two-direction endpoint. Other endpoints are always input or output, wherein the output (Out) indicates data transfer from the host to the device.
The types of communication in USB interface:

- The communication process in USB system based on data frames which takes 1ms (125us). Communication can be implemented in four standard types of data:
  
  - **Isochronous transfers:** At some guaranteed data rate (often, but not necessarily, as fast as possible) but with possible data loss. There is no handshake and retransmission (in case of error) mechanism implemented (e.g. realtime audio or video).
  
  - **Bulk Transfer:** For devices which sporadic transfers large block of data using all remaining available bandwidth, but with no guarantees on bandwidth or latency. This transmission has handshake and retransmission mechanisms in case errors are detected. (e.g. file transfers).
  
  - **Interrupt transfers:** For devices that need guaranteed quick responses with interval defined in the descriptor. (e.g. pointing devices and keyboard).
  
  - **Control Transfer:** It is used to configure and send defined by the standard USB commands to the device.
Descriptors:

- The USB system can communicate with other devices with different types of communication. USB host must know communication properties of the devices. Therefore, in all USB devices, there is full of information on how to communicate with the device available during the enumeration process. This information is stored in the descriptors which are arrays with well-defined structure.
Device Descriptor:

It contains mainly the information necessary to load the appropriate driver for operating system (including parameters Product ID and Vendor ID) and determine the number of possible configurations.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>Size of the Descriptor in Bytes (18 bytes)</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>Device Descriptor (0x01)</td>
</tr>
<tr>
<td>2</td>
<td>bcdUSB</td>
<td>2</td>
<td>BCD</td>
<td>USB Specification Number which device complies too.</td>
</tr>
<tr>
<td>4</td>
<td>bDeviceClass</td>
<td>1</td>
<td>Class</td>
<td>Class Code (Assigned by USB Org) If equal to Zero, each interface specifies its own class code If equal to 0xFF, the class code is vendor specified. Otherwise field is valid Class Code.</td>
</tr>
<tr>
<td>5</td>
<td>bDeviceSubClass</td>
<td>1</td>
<td>SubClass</td>
<td>Subclass Code (Assigned by USB Org)</td>
</tr>
<tr>
<td>7</td>
<td>bMaxPacketSize</td>
<td>1</td>
<td>Number</td>
<td>Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64</td>
</tr>
<tr>
<td>8</td>
<td>idVendor</td>
<td>2</td>
<td>ID</td>
<td>Vendor ID (Assigned by USB Org)</td>
</tr>
<tr>
<td>10</td>
<td>idProduct</td>
<td>2</td>
<td>ID</td>
<td>Product ID (Assigned by Manufacturer)</td>
</tr>
<tr>
<td>12</td>
<td>bcdDevice</td>
<td>2</td>
<td>BCD</td>
<td>Device Release Number</td>
</tr>
<tr>
<td>14</td>
<td>iManufacturer</td>
<td>1</td>
<td>Index</td>
<td>Index of Manufacturer String Descriptor</td>
</tr>
<tr>
<td>15</td>
<td>iProduct</td>
<td>1</td>
<td>Index</td>
<td>Index of Product String Descriptor</td>
</tr>
<tr>
<td>16</td>
<td>iSerialNumber</td>
<td>1</td>
<td>Index</td>
<td>Index of Serial Number String Descriptor</td>
</tr>
<tr>
<td>17</td>
<td>bNumConfigurations</td>
<td>1</td>
<td>Integer</td>
<td>Number of Possible Configurations</td>
</tr>
</tbody>
</table>
Configuration Descriptor:

Each device must have a descriptor for each possible configuration (e.g., it may operate in different modes increased or decreased power then it should have descriptors for both configurations), and contain information about the number available interfaces for a given configuration.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>Size of Descriptor in Bytes</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>Configuration Descriptor (0x02)</td>
</tr>
<tr>
<td>2</td>
<td>wTotalLength</td>
<td>2</td>
<td>Number</td>
<td>Total length in bytes of data returned</td>
</tr>
<tr>
<td>4</td>
<td>bNumInterfaces</td>
<td>1</td>
<td>Number</td>
<td>Number of Interfaces</td>
</tr>
<tr>
<td>5</td>
<td>bConfigurationValue</td>
<td>1</td>
<td>Number</td>
<td>Value to use as an argument to select this configuration</td>
</tr>
<tr>
<td>6</td>
<td>iConfiguration</td>
<td>1</td>
<td>Index</td>
<td>Index of String Descriptor describing this configuration</td>
</tr>
<tr>
<td>7</td>
<td>bmAttributes</td>
<td>1</td>
<td>Bitmap</td>
<td>D7 Reserved, set to 1. (USB 1.0 Bus Powered)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D6 Self Powered</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D5 Remote Wakeup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D4..0 Reserved, set to 0.</td>
</tr>
<tr>
<td>8</td>
<td>bMaxPower</td>
<td>1</td>
<td>mA</td>
<td>Maximum Power Consumption in 2mA units</td>
</tr>
</tbody>
</table>
Interface Descriptor:

The configuration can have multiple interfaces and they are all active at the same time allowing access to various functions of the device by the various drivers. An example might be a CD-ROM using three controllers for controlling its functions, one to write data, the second to play music, and the third for rendering graphics.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>Size of Descriptor in Bytes (9 Bytes)</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>Interface Descriptor (0x04)</td>
</tr>
<tr>
<td>2</td>
<td>bInterfaceNumber</td>
<td>1</td>
<td>Number</td>
<td>Number of Interface</td>
</tr>
<tr>
<td>3</td>
<td>bAlternateSetting</td>
<td>1</td>
<td>Number</td>
<td>Value used to select alternative setting</td>
</tr>
<tr>
<td>4</td>
<td>bNumEndpoints</td>
<td>1</td>
<td>Number</td>
<td>Number of Endpoints used for this interface</td>
</tr>
<tr>
<td>5</td>
<td>bInterfaceClass</td>
<td>1</td>
<td>Class</td>
<td>Class Code (Assigned by USB Org)</td>
</tr>
<tr>
<td>6</td>
<td>bInterfaceSubClass</td>
<td>1</td>
<td>SubClass</td>
<td>Subclass Code (Assigned by USB Org)</td>
</tr>
<tr>
<td>7</td>
<td>bInterfaceProtocol</td>
<td>1</td>
<td>Protocol</td>
<td>Protocol Code (Assigned by USB Org)</td>
</tr>
<tr>
<td>8</td>
<td>iInterface</td>
<td>1</td>
<td>Index</td>
<td>Index of String Descriptor Describing this interface</td>
</tr>
</tbody>
</table>
Endpoint Descriptor:

It describes an element that is the source or data receiver in the device. Since USB is a serial interface, the endpoint is properly FIFO queue sequentially evacuable or filled with transmitted or received data bytes. The number of endpoints in the device may vary, however, can not be greater than 32 (16 inputs and 16 outputs), which results from the way they are addressing. The most important parameters for endpoints are transmission type and data buffer size.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>Size of Descriptor in Bytes (7 bytes)</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>Endpoint Descriptor (0x05)</td>
</tr>
<tr>
<td>2</td>
<td>bEndpointAddress</td>
<td>1</td>
<td>Endpoint</td>
<td>Endpoint Address Bits 0..3b Endpoint Number. Bits 4..6b Reserved. Set to Zero Bits 7 Direction 0 = Out, 1 = In (Ignored for Control Endpoints)</td>
</tr>
<tr>
<td>3</td>
<td>bmAttributes</td>
<td>1</td>
<td>Bitmap</td>
<td>Bits 0..1 Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 2..7 are reserved. If Isochronous endpoint, Bits 3..2 = Synchronisation Type (Iso Mode) 00 = No Synchronisation 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4 = Usage Type (Iso Mode) 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback Data Endpoint 11 = Reserved</td>
</tr>
<tr>
<td>4</td>
<td>wMaxPacketSize</td>
<td>2</td>
<td>Number</td>
<td>Maximum Packet Size this endpoint is capable of sending or receiving</td>
</tr>
<tr>
<td>6</td>
<td>bInterval</td>
<td>1</td>
<td>Number</td>
<td>Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk &amp; Control Endpoints. Isochronous must equal 1 and field may range from 1 to 255 for interrupt endpoints.</td>
</tr>
</tbody>
</table>
**String Descriptor:**

The device may have descriptive text containing a description of the device, configuration, manufacturer's name.

<table>
<thead>
<tr>
<th>Offset</th>
<th>Field</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bLength</td>
<td>1</td>
<td>Number</td>
<td>Size of Descriptor in Bytes</td>
</tr>
<tr>
<td>1</td>
<td>bDescriptorType</td>
<td>1</td>
<td>Constant</td>
<td>String Descriptor (0x03)</td>
</tr>
<tr>
<td>2</td>
<td>wLANGID[0]</td>
<td>2</td>
<td>Number</td>
<td>Supported Language Code Zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(e.g. 0x0409 English - United States)</td>
</tr>
<tr>
<td>4</td>
<td>wLANGID[1]</td>
<td>2</td>
<td>Number</td>
<td>Supported Language Code One</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(e.g. 0x0c09 English - Australian)</td>
</tr>
<tr>
<td>n</td>
<td>wLANGID[x]</td>
<td>2</td>
<td>Number</td>
<td>Supported Language Code x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(e.g. 0x0407 German - Standard)</td>
</tr>
</tbody>
</table>
Basic class USB devices:

**Communication Device Class:** The CDC class devices are visible to the operating system as standard serial ports (virtual COM ports). This is probably the easiest way to implement a USB port on your machine. Support from the computer remains practically unchanged, because the library supports serial ports are popular and easy to use. Some development environments (eg. Visual Studio) offer controls to operate the port.

**Mass Storage Device Class:** This class emulates drives or CD-ROM drives. This solution allows you to install the software without having to use the real storage media (eg. Discs). Examples of devices that use this solution GSM modems are equipped with a USB connector. They can have a virtual hard drive and built-in memory card reader. When such a modem, it will be visible in the system as an additional disk containing the necessary drivers. Class of mass storage devices is a good choice when an electronic device is capable of exchanging files with your computer. An example of such a solution are MP3 / MP4 players, memory card readers, cameras or flash memory (flash drive).

**Human Interface Device:** HID class device has been designed to operate user interface devices (keyboard, mouse, joystick, etc.). The most common systems operating in this configuration does not require any drivers, since defaults are included with the operating system to ensure the operation of primary controls devices.
Source of power in USB devices:

- All USB devices can be powered from a hub port to which they are plugged in or with its own power supply.
- Each USB device informs the USB host about the type of power supply by appropriate descriptors in the configuration procedure.
- Power supply voltage supplied from the bus has a limited output current.
- Power management is done via hubs (hubs).
- Devices requiring higher current than is specified in specifications will not be configured by the host.
- Immediately after connecting (prior to configuration), the device cannot consume current higher than 100 mA.
- After completing the setup procedures, the devices may consume no more than 500 mA.
USB Specification version 2.0 provides for the transfer of data from one of three speeds:

- Low-Speed 1.5Mb/s
- Full-Speed 12Mb/s
- High-Speed 480Mb/s
Traditional USB devices work with the host, which is usually the computer. It concentrates on enumeration hardware, bandwidth control and support for transmission. Sometimes you want to connect two peripherals without a computer (e.g. The camera to a printer to print photos). The omission in this regard computer greatly simplifies the process and makes it more convenient and faster.

USB 2.0 specification, however, does not provide for such an eventuality. For this reason, the USB-IF organization developed a supplement introducing such a possibility - USB On-The-Go (USB OTG). USB OTG changes include:

- the possibility of cooperation peripherals in point-to-point mode
- new, smaller connector,
- the ability to work as a system peripheral (USB device) or host (USB HOST),
- mechanisms for reducing power consumption to extend battery life.
Communication Interfaces: USB OTG
JTAG (Joint Test Action Group)
JTAG (Joint Test Action Group) is a common name of the IEEE 1149.1 standard defines the protocol used for testing circuits on printed circuit boards, also used for startup and programming of programmable devices and microprocessor systems.

http://www.xjtag.com/support-jtag/what-is-jtag.php
IEEE 1149.1 standard signals:

- TDI (Test Data In)
- TDO (Test Data Out)
- TCK (Test Clock)
- TMS (Test Mode Select)
- TRST (Test Reset) optional.
Communication Interfaces: JTAG

Interconnection Test Example

http://www.corelis.com/education/Boundary-Scan_Tutorial.htm
Communication Interfaces: JTAG

Typical Board with Boundary-Scan Components

http://www.corelis.com/education/Boundary-Scan_Tutorial.htm
Communication Interfaces: JTAG ATmega32

Features:
- JTAG (IEEE std. 1149.1 Compliant) Interface
- Boundary-scan Capabilities According to the IEEE std. 1149.1 (JTAG) Standard
- Debugger Access to:
  - All Internal Peripheral Units
  - Internal and External RAM
  - Internal Register File
  - Program Counter
  - EEPROM and Flash Memories
  - Extensive On-chip Debug Support for Break Conditions, Including AVR Break Instruction
  - Break on Change of Program Memory Flow
  - Single Step Break
  - Program Memory Breakpoints on Single Address or Address Range
  - Data Memory Breakpoints on Single Address or Address Range
- Programming of Flash, EEPROM, Fuses, and Lock Bits through the JTAG Interface
- On-chip Debugging Supported by AVR Studio®
Communication Interfaces: JTAG ATmega32

**JTAG port in ATmega32:**

- TDI: Test Data Input,
- TDO: Test Data Output,
- TCK: Test Clock,
- TMS: Test Mode Select,
- TRST: Test Reset,
- VCC: power supply
- GND: ground
- VREFF: power supply for output buffers of JTAG programmer
- NSRST to be connected to the RESET pin microcontroller
Communication Interfaces: AVRDRAGON

Features:

- Supports up to 3 hardware program breakpoints or 1 maskable data breakpoint (depending on the OCD module on the AVR device),
- Supports up to 32 software breakpoints,
- On-board 128kB SRAM for fast statement-level stepping,
- Robust level converters support 1.8V to 5.5V target operation,
- Uploads 256Kb code in ~60 seconds (XMEGA using JTAG interface),
- Full-speed USB 2.0 compliant host interface (12 MB/s),
- Firmware upgradeable for supporting future AVR devices
- Supports NanoTrace (depending on the OCD module on the AVR device; uses target device's memory)
- USB powered, and capable of sourcing power to an external target
Features:

• First on market three-in-one USB JTAG debugger - offers JTAG + RS232 (full modem signals supported) port + power supply all in one compact device

• Adds virtual RS232 port to your computer with all modem signals like: DTR, DSR, DCD, RTS, CTS, Rx, Tx

• Debugs all ARM microcontrollers with JTAG interface supported by OpenOCD

• Uses ARM's standard JTAG connector (2 rows × 10 pins at 0.1" step)

• Supports ARM targets working in voltage range 2.0 – 5.0 V DC

• Supported by the open-source community and OpenOCD debugger software
Communication Interfaces: Olimex ARM-USB-OCD

ARM Standard JTAG Connector (20-pins, 0.10"")
Communication Interfaces: Serial Wire Debug (SWD)

ARM Standard JTAG Connector (20-pins, 0.10") with SWD