Communication

~~USART (Universal Synchronous/Asynchronous Receiver/Transmitter)~~

Serial Communication

- Data transmission happens 1 bit at a time.
- Slower than parallel but efficient over longer dist.
- fewer wires
- cheap & simple
- USB, RS-232, SPI, UART
- Less Interference
- Better for long dist. communication

parallel Communication

- multiple bit at once (8, 16, etc.)
- Faster over short distances
- more wires
- more expensive & ~~costly~~ Complex
- Older printer ports, internal buses in computers, DDR RAM
- High data rate over short distances.

Synchronous & Asynchronous communication.

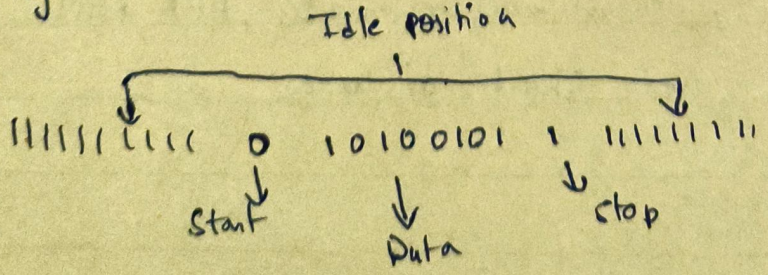
Synchronous

- Clock signals tells when to send
- Both sender & reciever are synchronized to the same time.
- eg. I2C, SPI, DDR RAM
- Faster & more reliable
- Less chance of misreading of data.
- extra wire for the clock
- complex hardware.

Asynchronous

- Data is sent with start and stop bit so that Reciever knows when to start & stop receiving.
- Receiver uses timer based ~~start~~ on agreed speed (baud Rate) to read the bits.
- eg. UART, RS 232, serial ports (like COM ports)
- Simpler, fewer wires
- works well for slower or occasional data
- Slightly ~~Sl~~ slower due to start & stop bits
- less accurate at higher speeds.

For eg. Data -> 10100101



USART

- Universal Synchronous Asynchronous Reciever Transmitter.
- Hardware communication module
- found in many microcontrollers like arduino, PIC/AVR, STM32, etc.
- Serial data communication.
- 2 modes : Synchronous & Asynchronous.

Synchronous.	Asynchronous
• Clock signal	• No clock
• Both device share same clock	• Start & stop bit to time data transfer.
• Faster & more precise	• More common in simple serial comm.
• 3 lines : data, clock, ground.	• Requires 2 lines Transmitter/reciever & Gnd.

Key features.

→ Dual mode.
→ send & receive signal at same time.
→ Configurable baudrate.
→ Buffering
→ Error detection.

USART	UART
• Clock optional	• No Clock
• Sync & Async	• Async.
• more versatile	• Simle & common.

Choice, not chance, determines human destiny.

Registers in AVR

UCSRnA	USART Control & Status Register A
UCSRnB	USART " " " " B
UCSRnC	" " " " " C
UBRRn	USART Baud Rate Register
UDRRn	" Data "

Inside UCSR0A (8bit)

Bit	Name	func.
7	RXC0	Set when a byte has been received & is ready to be read from UDR0
6	TXC0	Set when the transmit is complete
5	UDRE0	Set when the data register is empty, its ready to send new data
4	FE0	Frame error - set if there a bad stop bit (corrupt file)
3	DOR0	Data Overrun - Set if new data has arrived before the previous data was read.
2	UPE0	Parity error - set if parity check fails.
1	U2X0	Double transmission speed for faster baud rates.

0 MPCM0 Multiprocessor Comm. mode - Used for addressing in network.

UBRR n - USART Baud Rate Register.

used to set baud rate

Baud Rate = how fast the data is sent on received (bits/sec.)

UBRRn = (F_CPU / (16* Baud)) - 1 U2Xn = 0 (Normal speed)

UBRRn = (F_CPU / (8* Baud)) - 1 U2Xn = 1 (double speed)

F_CPU = CPU clock frequency (16 mHz on Arduino uno)

Baud = deside Baud Rate (eg. 9600)

UBRRn is a 16 bit architecture

∴ It is split into 2 parts.

UBRRnH & UBRRnL

Input

TXCn → Datareg. → RXCn

Baud Rate

Baud Rate is the speed of data transmission.

Mode	Formula
Normal Async.	$UBRRn = \frac{F\_CPU}{16 \times Baud} - 1$
Double speed	$UBRRn = \frac{F\_CPU}{8 \times Baud} - 1$
Synchronous	$UBRRn = \frac{F\_CPU}{2 \times Baud} - 1$

Data transmission and Reception

Data Framing

Start bit → 0
Databit → 5-9 bits
Parity bit → optional error checking (even/odd)
Stop bit → end of frame (1 or 2 bits, always 1)

8N1 format (most common)
1 Startbit
8 Databit
0 paritybit
1 Stop bit.

Sending a byte

Setup

```
UBRR0 = 103;                    // baud rate = 9600
UCSR0B = (1 << TXEN0);          // Enable Transmitter
UCSR0C = (1<< UCSZ01) | (1<< UCSZ00); // set frames to 8 bit.
```

Sending

```
while (!(UCSR0A & (1<< UDRE0))); // waiting until TX buffer is empty
UDR0 = 'A';
```

Receiving Data

```
UCSR0B = (1 << RXEN0);          // enable Receiver
while (!(UCSR0A & (1<<RXC0)));  // wait for data
char data = UDR0;               // Read the byte.
```

Polling	Interrupt	DMA
• CPU checks a flag repeatedly • Simple to implement • CPU blocked while waiting	USART triggers an interrupt when ready • Non blocking, efficient • Needs interrupt setup	Direct memory access moves data (no CPU) • Fast, zero CPU load. • More complex, not in all MCUs

It isn't what you have in your pocket
that makes you thankful,
but what you have in your heart.

Parity error

It is an extra bit in the data

It counts the number of 1s present in the data and sends in the UDRn

the parity bit is 1 in odd parity and 0 in even parity for even number of 1s

The transmitter counts the number of 1s and sends it with the UDRn.

The receiver will be expecting the same parity but if there will be a change then there will be a parity error.

UPM 01	UPM 00	Parity Mode
0	0	Disable
0	1	Reversed
1	0	even parity
1	1	odd parity.

Framing Error

Occures when expected stop bit doesnot occur at correct time..

Causes
- → Mismatched baud rate b/w sender & receiver
- → Electrical noise
- → Incorrect frame format setting

Overrun Error

new data arrives before the old data is read.

What happens

→ overwriting of new data on the old data

→ the old data is lost forever.

I2C

I^2C or Inter integrated circuit is a synchronous, multimaster multi slave, serial communication protocol.

Why I^2C?

- Communication b/w 2 chips.
- Uses only 2 wires regardless of the number
- Efficient for short ranged, low speed comm. on a pcb

Master - Slave Architecture

- Master: Initiates comm.
- Slave: responds to master's commands.
- Multi master possible
- Slaves have unique

Two wire interface.

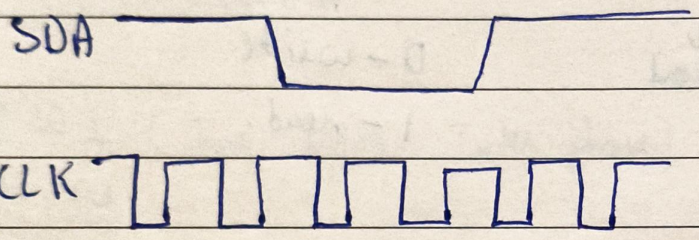
SDA	Serial Data	Carries Data
SCL	Serial Clock	carries clock

Both lines are open drains.

Open drains: you need to just pull it down but need not pull high. It automatically goes high when not pulled low

I^2C Speed Modes

Std.	100 kHz	} Most sensors work in this range.
Fast	400 kHz	
Fast +	1 MHz	
High Speed	3-4 MHz	} needs spl. hardware support.



Master SDA
pulls down to initiate the comm.
SDA pull down marks the start of address reading for the slave.
The slave starts to read next 7 address bits and 1 R/W bit when the clock signal is high.
Then the slave pulls down SDA sending the ACK signal.
Then the master releases the SDA.

Why not SCL always high?
- SCL tells when to read & write
- Syncs the timing b/w sender and receiver.
- Slave wouldnt know which bit to read when

Why I^2C uses SCL?

- Allows precise bit timing.
- Supports multiple devices with different speeds.
- Ensure reliable and synchronised comm.

7bit VS 10bit

7bit

7 address bit + R/W bit

AAAAAAA R

Address — Read

0 - write

1 - read.

10101110

address — write

Very common.

10bit

1st byte	Indicates 10bit mode + upper 2 bit + R/W
2nd byte	Lower 8 bits of address
Data	Actual data to read/write
control	Uses ACK/NACK

Data Frame Format

- Start Condition
- Address + R/W bit
- ACK from Slave
- Data byte from master / Slave
- ACK from receiver
- NACK from reciver end to stop
- Stop Condition

TWI (Two wire Inter phase)

Registers

TWBR	TWI Bit Rak Register	Sets Clock frequency.
TWCR	TWI Control Register	Enable TWI, starts transmission, enables ACK, etc.
TWSR	TWI Status Register.	Shows current state of TWI operation.
TWDR	TWI Data Register	Holds data to be sent or recrived.
TWAR	TWI Address register	Sets own slave address.

$$SCL\ freq = \frac{CPU\_freq}{(16 + 2 \times TWBR \times 4^{TWPS})}$$

Wisdom consists not so much in knowing what to do in the long run as in knowing what to fo next.

TWPS = TWI Prescaler bit.

TWPS0 & TWPS1 are 2 bits in TWSR

TWPS 1	TWPS 0	TWPS	4^{TWPS}
0	0	0	1
0	1	1	4
1	0	2	16
1	1	3	64

TWBR is just an 8 bit value b/w 0–255 which directly affects the clock rate.

TWCR → Two wire Control register.

- Enabling / disabling TWI
- Starting / stopping Comm.
- Acknowledging data
- Handling interrupts.

P.T.O

Structure of TWCR 8 bit reg.

Bit	Name	Purpose.
7	TWINT	TWI Interrupt - set when an I^2C event is done.
6	TWENA	TWI Enable Acknowledge - send Ack after a byte is received.
5	TWSTA	TWI start condition - initiate start condition.
4	TWSTO	TWI Stop condition - send a stop condition
3	TWWC	TWI write collision - sent if write occurs during an active write.
2	TW EN	TWI Enable - must be set to use TWI hardware.
1	reserved	pretended to not exist. can be used for future uses.
0	TWIE	TWI Interrupt enable - For interrupt driven I^2C

TWSR = TWI Status Register.

has 2 type of bits → status code
→ prescaler bits

Structure of TWSR

Bits	Name	Purpose
7-3	TWS7-TWS3	Status code bits (5 bits)
2	Unused	Always reads 0
1	TWPS1	prescaler bit 1 } Used for SCL freq.
0	TWPS0	prescaler bit 0

Status bit (TWS7 - TWS3)

These hold a status code that tells you what happened in I^2C interface. Helps you track process.

It checks state of comm. like whether:

-> Start condition was sent
-> An address was acknowledged
-> Data was received
-> A slave responded
-> Something went wrong.

Common status codes:

0x08 - Start condition
0x10 - repeated start condition
0x18 - SLA+W transmitted & ACK received
0x20 - SLA+W transmitted but NACK
0x28 - Data byte transmitted ACK received
0x38 - You tried sending data but other master took control

TWDR – TWI Data register
It is 8 bit register which can store 1 byte of data. no control or flag bits.

Sending data.

```
TWDR = 0x42; //loading the byte to send.
TWCR = (1 << TWINT) | (1 << TWEN); //start transmission
while (!(TWCR & (1<<TWINT)));
```

Receiving data.

```
TWCR = (1 << TWINT)|(1 << TWEN) | (1<< TWEA)
                                        //expecting ACK
while (!(TWCR &(1 << TWINT))); //wait until data is received.
uint8_t  received = TWDR;
```

TWAR – Two wire Address Register.
mainly 2 things → slave address
→ enable general call.

Structure of TWAR

Bit	Name	Purpose
7-1	TWA6-TWA0	7 bits slave address
0	TWGCE	General call recognition Enable (0 = disable, 1 = enabled)

Single Master, Single Slave

→ One master controls the bus
→ One slave listens and responds

working
- Master sends start → Address of slave → Data
- slave responds with ACK → Receives/sends data
- Master sends stop when done.

→ Simple and reliable
→ minimal chance of address collision or bus conflict.

Single Master, Multiple slave

→ one master
→ Multiple slave with unique addresses.
→ widely used in real systems
→ Only 2 wires needed regardless of number of slaves.
→ Max. number of slaves is limited by address spaces & bus capacitances
→ 7 bit addressing is most common → 127 devices (1 reserved)

Working
- Master addresses only 1 slave at a time
- Each slave listens to the bus but responds only when address matches
- slave doesnot transmit unless requested.

Multi Master Configuration

→ more than one master can initiate communication.
→ All devices share the same SDA & SCL Lines.
→ Bus Arbitration: If more than one masters are trying to send the data at the same time then this happens. Consider the below table

Bit position	M1	M2	SDA	Outcome
1	1	1	1	both continue
2	0	0	0	both continue
3	1	0	0	M1 backs off.

M2 wins and M1 tries again later.
→ Clock Synchronisation: Masters may stretch the clock to avoid timing issues.
→ Rare in most embedded systems.
→ Useful in system with redundant controller with multiple data source

Common I^2C issues and their causes.
→ No ACK from Slave
Symptom: After sending the slave address receive no ACK
Causes:
- wrong slave address
- Slave not powered or not initialized
- SDA/SCL Lines not pulled up properly
- slave not connected properly.

A man wrapped up in himself makes a very small bundle.

→ Bus Hang (SDA or SCL stuck Low)

Symptom: Comm. halts, Line remains low

Possible causes:

- A device held the line low
- No stop condition sent
- Noise during transfer
- Fault pull-up resistor.

→ Data corruption or misinterpretation.

Symptom: Received bytes are incorrect or out of order.

Possible cause:

- Mismatched clockspeed
- poor signal
- incorrect use of ACK, NACK

→ Slave not responding W/R

Possible cause:

- device requires specific register address before reading
- Not enough delay before comm. start.
- Wrong comm. mode

Debuggers

- Logic analyzer
- Oscilosope
- Bus scanner software.

SPI

Serial Peripheral Interphase
→ fast and full duplex comm. protocol.
→ 1 master and 1 or more Slave

Features

Speed	Very fast
full duplex	Data is sent and received simultaneously
Simple hardware	easy to implement with shift register.
No addressing	Slave is selected using seperate lines
point to point	Best for few devices over short distances.

4 Wire interphase.

MOSI	Master Out Slave In	M → S	Data M → S
MISO	Master in Slave Out	S → M	Data S → M
SCK	Serial Clock	M → S	clock generated by master
SS	Slave select	M → S	Used to select which slave is active

CPOL - Clock polarity
CPHA - Clock phase

4 mode - 0, 1, 2, 3

mode	CPOL	CPHA	Clock idle	Data capture On
0	0	0	Low	Rising edge
1	0	1	Low	falling edge
2	1	0	High	falling edge
3	1	1	High	Rising edge

CPOL = 0 Clock Low when idle

CPOL = 1 " high " "

CPHA = 0 Data sample on first edge

CPHA = 1 " " " second "

Why modes?

→ master must agree on SPI modes.

→ If they mismatch data gets corrupted

SPI Data frame size

→ Most common : 1 byte (8 bits) per transfer.

8 bits	Default for most SPI comm.
16 bits	Higher resolution sensor, DACs, ADCs
24 bits	Some digital audio codecs
32 bits	Custom protocols, wide data words

You can send either MSB first or LSB first.

Multiple Slaves: CS Lines or Daisy Chaining

CS lines
→ each slave has its own CS lines
→ Master sets 'CS low to speak to that particular slave.
→ Very common
→ Easy to control multiple slaves.
→ No dependency b/w slaves
→ Needs more GPIOs

Daisy Chaining
→ one CS line for ~~each~~ all slave
→ Data moves ~~through~~ each slaves like a shift register
→ Used in ~~dot~~ devices like Shift register & DACs
→ Fewer CS lines
→ Neat & tidy
→ Slower
→ More complex software logic

SPI has 3 key Registers
SPCR – SPI Control register
SPSR – SPI status register
SPDR – SPI Data register.

SPCR

Bit	Name	function
7	SPIE	SPI Interrupt enable
6	SPE	SPI Enable
5	DORD	Data Order
4	MSTR	Master / Slave select
3	CPOL	clock polarity
2	CPHA	clock phase
1,0	SPR 1:0	SPI clock rate (Prescaler

SPSR

Bit	Name	Description
7	SPIF	SPI interrupt flag
6	WCOL	write collision flag
5-1	reserved	not in use
0	SPI 2X	Double SPI speed bit

SPDR

write to SPDR to start transmission
Read from SPDR to get received data

Clock frequency setting

SPI clock freq = $\frac{F\_CPU}{prescaler}$

SPI2X	SPR1	SPR0	SPI Clock
0	0	0	4
0	0	1	16
0	1	0	64
0	1	1	128
1	0	0	2
1	0	1	8
1	1	0	32
1	1	1	64

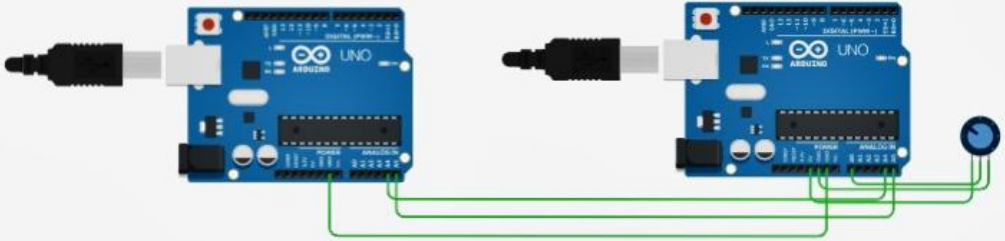
Debugging

- Logic Analyzer
- Oscilloscope
- Using flags.

Smashing Uusam-Waasa

All changes saved

Code | Start Simulation | Send To



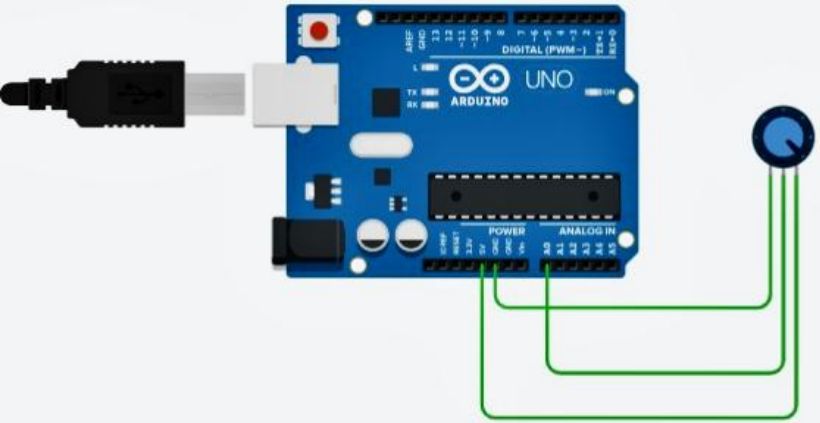
Text | 1 (Arduino Uno R3)

```
#include <Wire.h>

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

void loop() {
  Wire.requestFrom(8, 1);

  while (Wire.available()) {
    int receivedData = Wire.read();
    Serial.print("Received Data: ");
    Serial.println(receivedData);
  }

  delay(500);
}
```

Serial Monitor

```
Received Data: 255
Received Data: 255
Received Data: 255
Received Data: 122
Received Data: 122
Received Data: 168
Received Data: 179
```

Send | Clear



Text

1 (Arduino Uno R3)

```
1  void setup() {
2    Serial.begin(9600);
3  }
4
5  void loop() {
6    int sensorValue = analogRead(A0);
7    Serial.print("Sensor Value: ");
8    Serial.println(sensorValue);
9    delay(500);
10 }
11
```

Serial Monitor

```
Sensor Value: 225
Sensor Value: 491
Sensor Value: 921
Sensor Value: 1023
Sensor Value: 1023
Sensor Value: 1023
Sensor Value: 1023
```