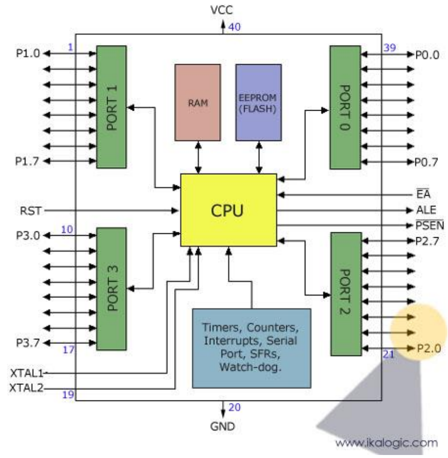
Embedded Systems Design, Spring 2025



Serial ports and peripherals (USART and I2C)

Outline

- Introduction to communication with peripherals
- Serial port overview
- Configuring the Atmega 328p for USART communication
- Serial port communication example
- Interrupts for serial communication

- I2C overview
- Atmega 328p for I2C master communication
- I2C Example with LM75 sensor

Peripherals and communication

- What is a peripheral?

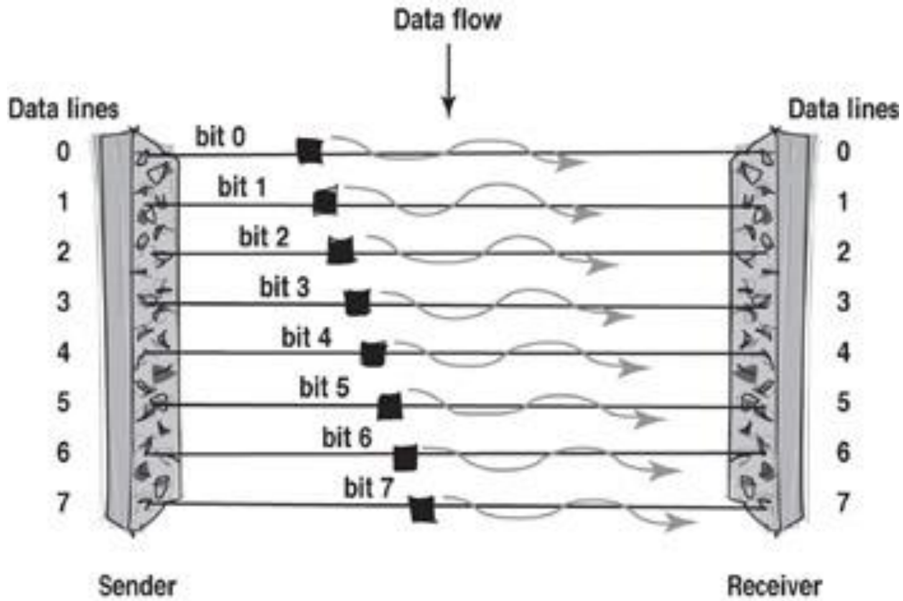
- What is a communication protocol?

Communication with peripherals

Protocol	Bit rate
RS – 232 / USART	between 110 bit/s and 115200 bit/s
RS - 485	35 Mbit/s
Parallel port (LPT)	12,000 Kbit/s
Serial Peripheral Interface (SPI)	up to 9 Mbit/s
Inter Integrated Circuit (I2C)/TWI	1 Mbit/s, 3.4 Mbit/s
USB	1.5 / 12 / 480 / 5,000 Mb/s
Ethernet	1 Gbit/s, 10 Gbit/s, 100 Gbit/s
SATA	1.5, 3.0, 6.0 Gbit/s
DVI	4.95 Gbit/s @ 165 MHz
HDMI	10.2 Gbit/s / 18 Gbit/s
PCI-Express	**v4.0**: 1 / 16 lanes: 1.8 / 31.51 GB/s

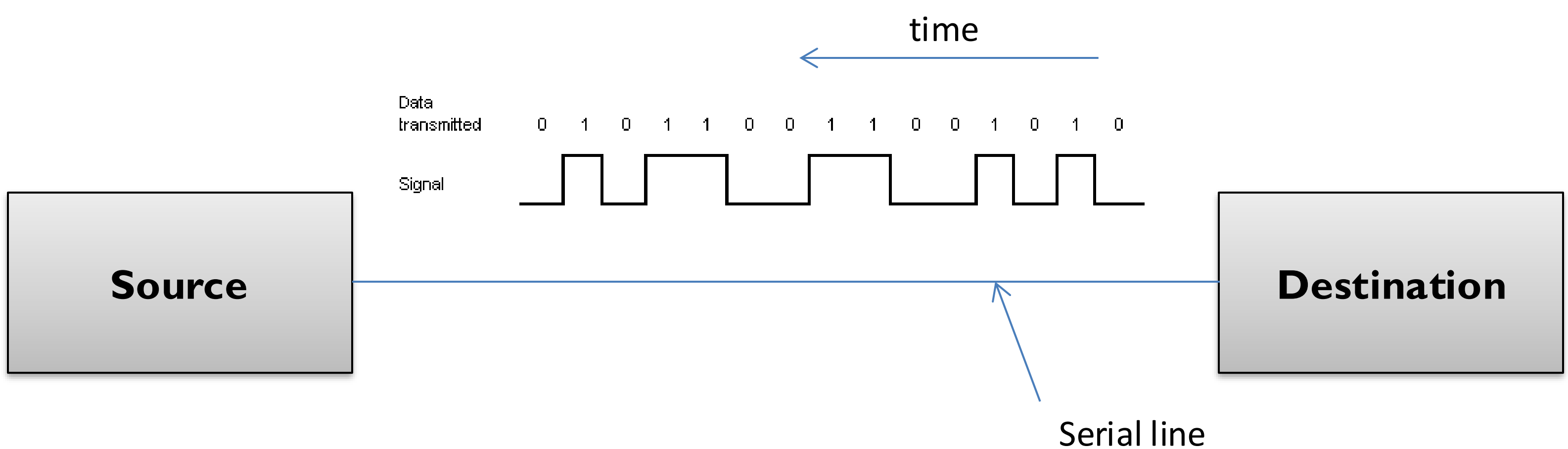
Parallel communication

- Parallel communication implies sending a whole byte (or more) of data over multiple parallel wires
- Control bits are used to determine the timing for reading/writing the data



Serial Communication

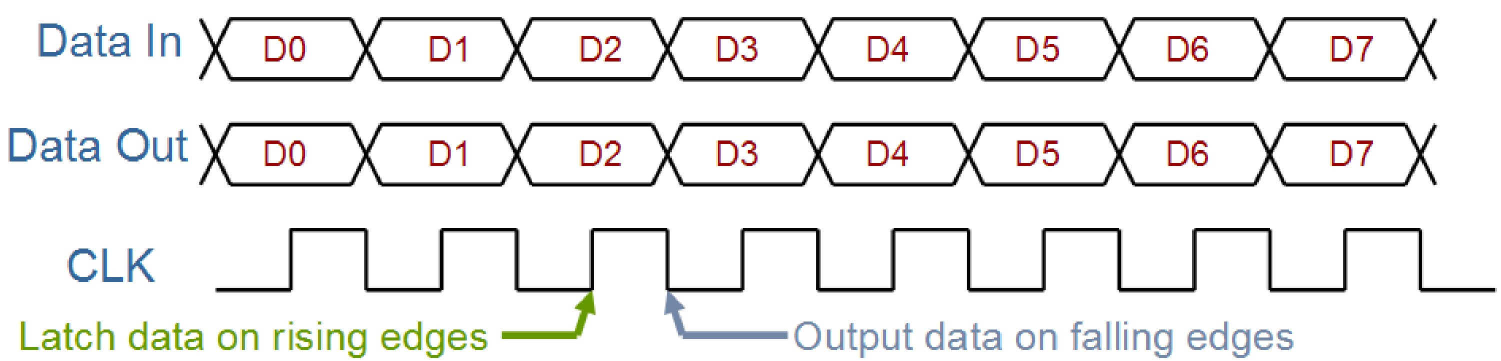
- Bit serial transmission



- Synchronization problems. Transitions occur with respect to a transmitter clock.
- Two modes of operation:
  - **Synchronous** (both systems have synchronized clocks)
  - **Asynchronous** (each system uses its own independent clock)

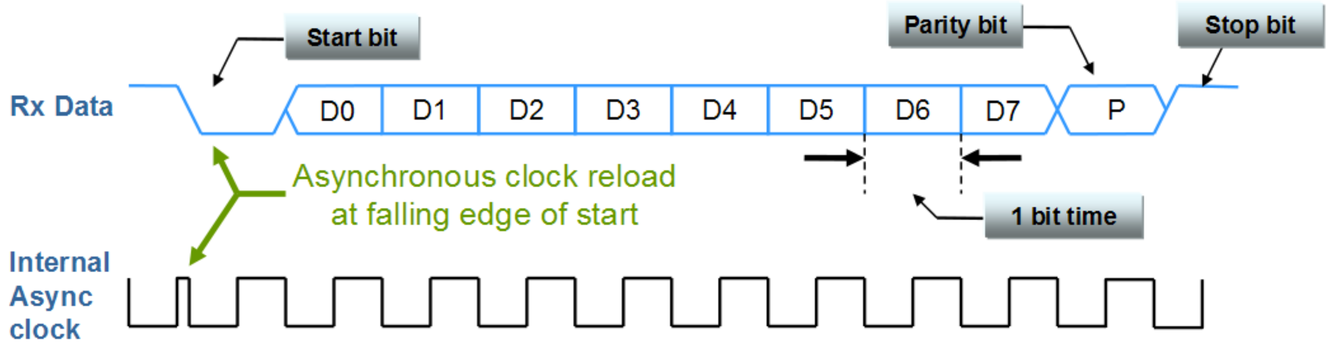
Synchronous Serial Communication

- Transmitter and receiver clocks are synchronized
- Reduces synchronization problems at bit level or byte level
- Data rate for the link must be the same at receiver and transmitter
- It requires to have an extra wire to transmit the clock



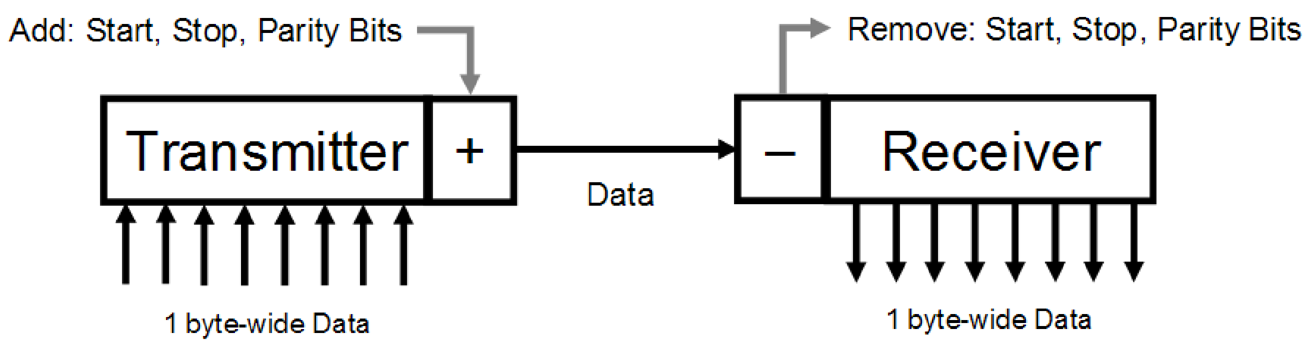
Asynchronous Serial Communication

- Removes the clock line, but it requires additional bits for synchronization and control (2-3 extra bits)
- 1 start bit (indicates beginning of the byte)
- 1 parity bit (error detection – optional)
- 1 stop bit (indicates the ending of the byte)

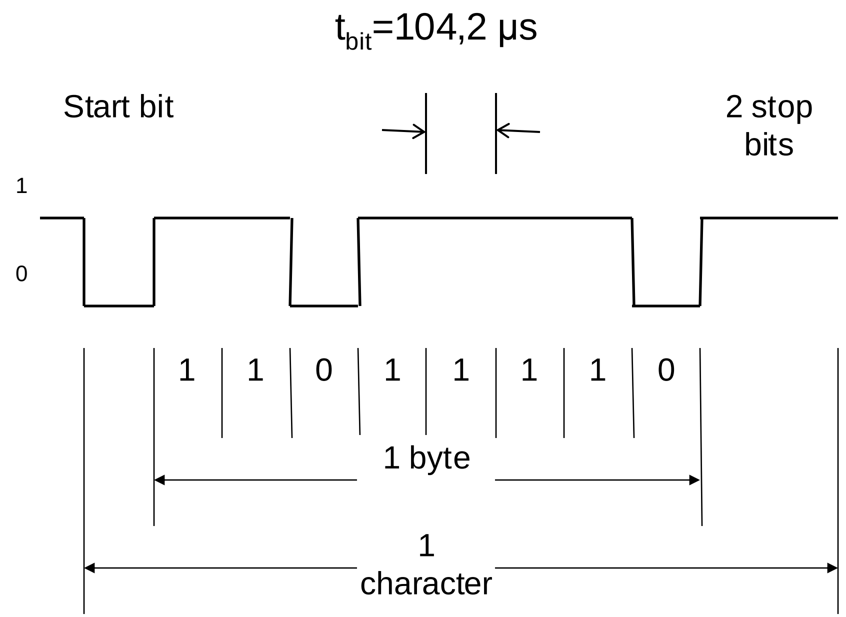


Asynchronous Serial Communication

- Baud rate: needs to be agreed by both parties before the communication starts.
  - Typical values: 2400, 4800, **9600**, 19200, 38400, **57600**,115200 bits/second.
- The number of stop bits + parity bits also need to be agreed upon.
- Usage of one single wire, typically used for short connections, low throughput



Example of one character

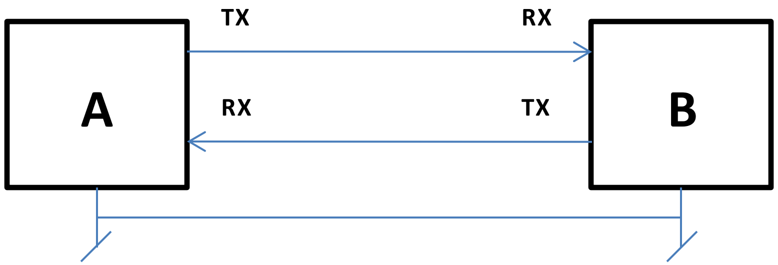


This figure shows the bit pattern when sending the byte 0x7B, at 9600 bit/s and with 2 stop bits.

Notice: the data bits are sent using the least significant bit first (LSB).

RS-232

- The RS232 standard includes details of:
  - The protocol to be used for data transmission
  - The voltages to be used on the signal lines
  - The connectors to be used to link equipment together.
- RS-232 is a peer-to-peer communication standard, intended to link only two devices together **(for long distances = over 1 m, less than 20 m)**
- The link is able to carry data from **A** to **B**, and from **B** to **A** at the same time. This is called *Full-duplex*.



- When no data present on the ’transmit’ line, Logic 1 level. The voltage levels are:

Sender:	Receiver:
Logic 1: -5 to -15 V	**Logic 1**: below -3 V
Logic 0: +5 - +15 V	**Logic 0:** above +3 V

USART

- The Universal Asynchronous Receiver/Transmitter (USART) takes bytes of data and transmits the individual bits in a sequential fashion

- Typical use:
- 1. start and configure the USART module
- 2. create a function to send a char (or byte)
- 3. create a function to receive a char (or byte)
- 4. write a simple program that uses these functions

Baud Rate Generator

- Set the desired baud rate according to the formula:

Asynchronous Normal mode (U2Xn = 0)	$BAUD = \frac{f_{OSC}}{16(UBRRn + 1)}$

- We choose BAUD, we know F_OSC and calculate the value for UBRR (UsartBaudRateRegister).
- UBRR is a 16 bit register (it has a low byte register and high byte register)

```
#define BAUDRATE 9600        //The baudrate that we want to use
#define BAUD_PRESCALER (((F_CPU / (BAUDRATE * 16UL))) - 1)
//This calculates the BAUD_Prescaler for us
```

- Now we can set both high and low registers:

```
UBRR0H = (uint8_t)(BAUD_PRESCALER>>8);
UBRR0L = (uint8_t)(BAUD_PRESCALER);
```

Other settings

- We need to enable RX and TX pins, in the **UCSR0B** (USART Control and Status Register B)

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

```
UCSR0B = (1<<RXEN0)|(1<<TXEN0); //enable RX and TX
```

- We configure the start/stop bits and data length in **UCSR0C**

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

parity Stop bits Data length

```
UCSR0C = ((1<<UCSZ00)|(1<<UCSZ01));//1 stop bit, 8 data, no parity
```

More settings

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- **Bit 7 – RXCn: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data). If the receiver is disabled, the receive buffer will be flushed and consequently the RXCn bit will become zero. The RXCn flag can be used to generate a receive complete interrupt (see description of the RXCIEn bit).

- **Bit 6 – TXCn: USART Transmit Complete**

This flag bit is set when the entire frame in the transmit shift register has been shifted out and there are no new data currently present in the transmit buffer (UDRn). The TXCn flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn flag can generate a transmit complete interrupt (see description of the TXCIEn bit).

- **Bit 5 – UDREn: USART Data Register Empty**

The UDREn flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written. The UDREn flag can generate a data register empty interrupt (see description of the UDRIEn bit). UDREn is set after a reset to indicate that the transmitter is ready.

Initialization function

```
void usart_init(void){
      UBRR0H = (uint8_t)(BAUD_PRESCALER>>8);
      UBRR0L = (uint8_t)(BAUD_PRESCALER);
      UCSR0B = (1<<RXEN0)|(1<<TXEN0);
      UCSR0C = ((1<<UCSZ00)|(1<<UCSZ01));
}
```

Sending data

- check if there is space in the sending buffer (by checking the register A)

• **Bit 5 – UDREn: USART Data Register Empty**

The UDREn Flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written.

- Then put data in the UDR0 (USART Data Register). It will be sent automatically

```
void usart_send( unsigned char data){
	while(!(UCSR0A & (1<<UDRE0))); //wait for transmit
buffer
	UDR0 = data; //data to be sent
}
```

Receiving data

- check if there is something newly received (by checking the register A)

• Bit 7 – RXCn: USART Receive Complete

UCSRnA

This flag bit is set when there is unread data in the receive buffer and cleared when the receive buffer is empty (i.e., does not contain any unread data).

- Then retrieve the new data from the UDR0 (USART Data Register).

```
unsigned char usart_receive(void){

      while(!(UCSR0A & (1<<RXC0))); //wait for new data
      return UDR0; //received data

}
```

Example program

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>
#define BAUDRATE 9600
#define BAUD_PRESCALER (((F_CPU / (BAUDRATE * 16UL))) - 1)
//Function prototypes
void usart_init(void);
unsigned char usart_receive(void);
void usart_send(unsigned char data);
int main(void){
usart_init();        //Call the USART initialization code

while(1){        //Infinite loop
       usart_send(50);    //send the value 50
       _delay_ms(5000);   //Delay for 5 seconds so it will re-
send the value every 5 seconds
 }

return 0;
} // DON'T FORGET TO ADD THE FUNCTIONS BODIES !!!
```

USART with interrupts

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – RXCIEn: RX Complete Interrupt Enable n**

Writing this bit to one enables interrupt on the RXCn flag. A USART receive complete interrupt will be generated only if the RXCIEn bit is written to one, the global interrupt flag in SREG is written to one and the RXCn bit in UCSRnA is set.

- **Bit 6 – TXCIEn: TX Complete Interrupt Enable n**

Writing this bit to one enables interrupt on the TXCn flag. A USART transmit complete interrupt will be generated only if the TXCIEn bit is written to one, the global interrupt flag in SREG is written to one and the TXCn bit in UCSRnA is set.

Example with interrupts

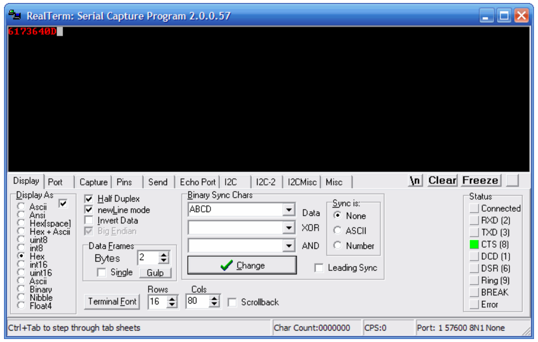
```
// Enable receiver to turn ON led 1 for receiving message with
value 1 and led 2 for receiving message with value 2
… // include necessary libraries, function prototypes,
int main(void){
…// configure leds as outputs
usart_init(); //Call the USART initialization code
UCSR0B|= (1<<RXCIE0); //enable interrupts for RXIE
sei(); //enable interrupts
while(1){         // empty infinite loop
}
return 0;
} // … don't forget to add function bodies

ISR(USART_RX_vect){
volatile unsigned char received_data = UDR0;
       if (received_data == 1) PORTD = 0b00010000; //led1 on
       if (received_data == 2) PORTD = 0b00100000; //led2 on
}
```

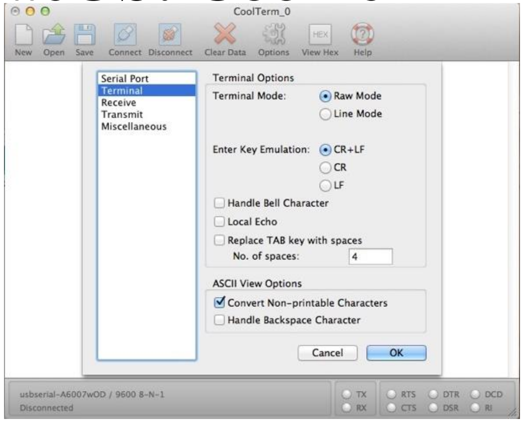
PlatformIO configuration

- Use the Serial Monitor of VS code to show received data in hexadecimal > Edit **platformio.ini**

```
monitor_echo = yes
monitor_encoding = hexlify
```

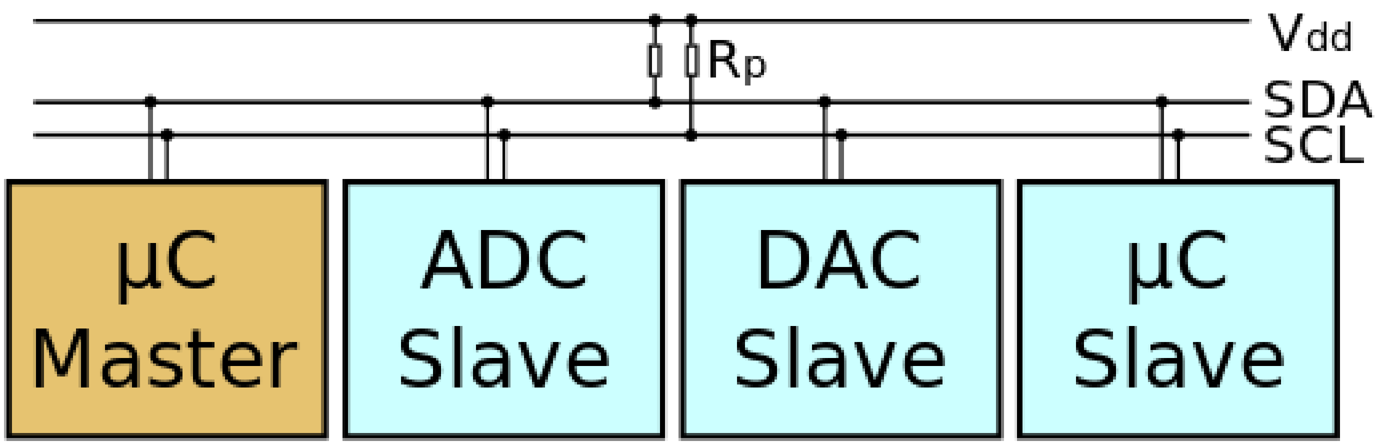
- **Windows : RealTerm / MacOS: CoolTerm**



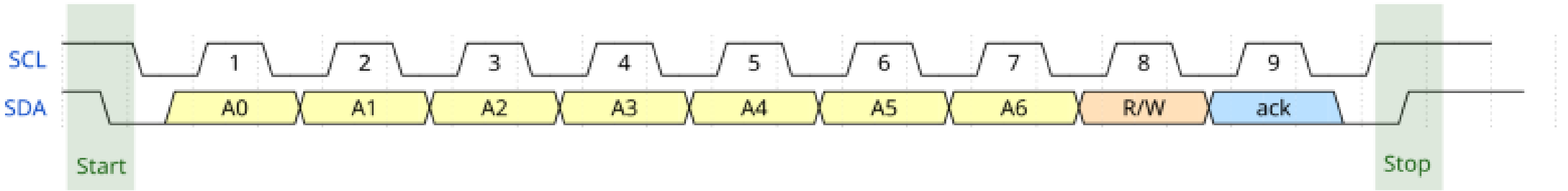


I2C

- You can connect multiple devices over the same bus
- Each device will have it’s own unique address (range: 8 – 119)
- You could then have multiple periphereals, at different addresses.
- Rp could as well be 4.7 kOhms



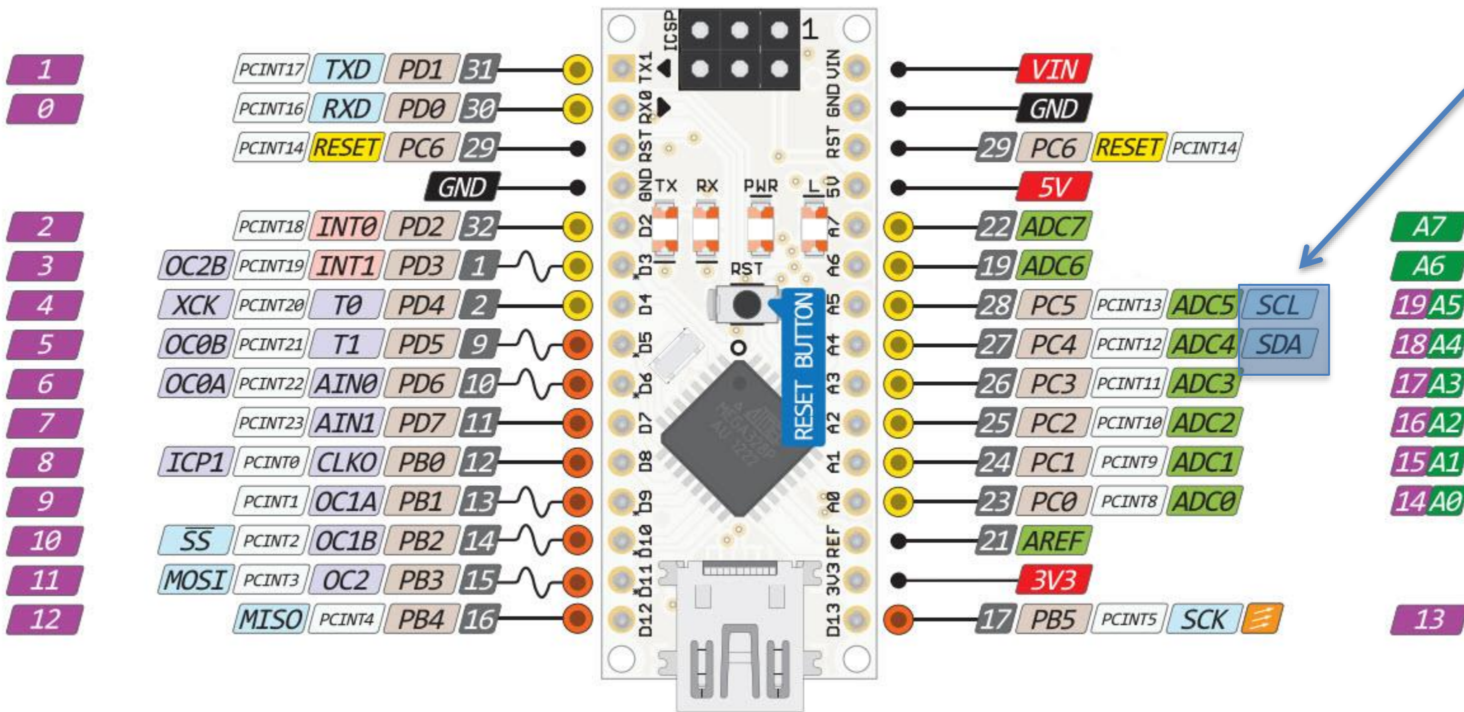
Timing diagram

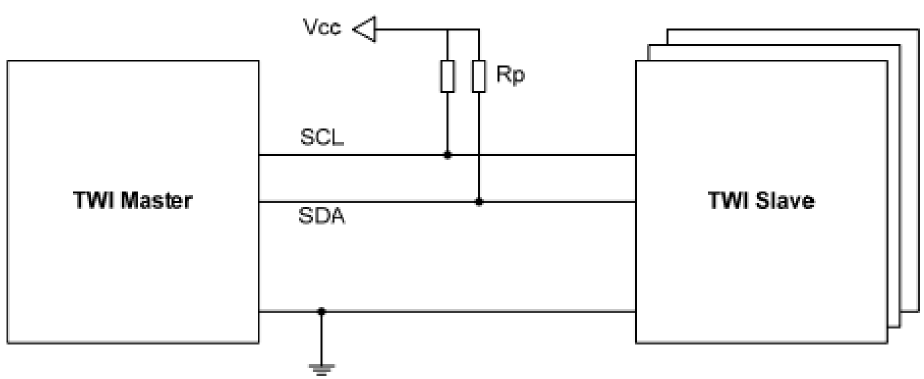


1. Data Transfer is initiated with a START bit signaled by SDA being pulled low while SCL stays high.
2. SDA sets the 1st data bit level while keeping SCL low
3. The data is sampled (received) when SCL rises for the first bit (A0).
4. This process repeats, SDA transitioning while SCL is low, and the data being read while SCL is high (A1, A2 etc.).
5. A STOP bit is signaled when SDA is pulled high while SCL is high.

Arduino Nano connection

TWI





Example

- Using the LM75 – temperature sensor