Arduino serial communication

Arduino supports serial communications. Though there are many types of serial communication ways, In this part, I am going to discuss only on UART, SPI and I2C serial communication for Arduino devices.

It's important to know that, UART transfers information in a asynchronous way while other two (SPI and I2C) work in a synchronous way.

UART (Universal asynchronous Reception & transmission)

• Uses 2 wires to send data (pin 1-TX) & receive data (pin 0-RX).

• Common ground between the Arduino's.

• Arduino has UART communication via USB that can be connected to pc.

• No clocks required.

Characteristics

To transmit data, UART requires three main characteristics: a start bit, a stop bit and baud rate. To be able to transfer a message UART needs to synchronize data between the two devices. This is achieved by grouping the message that is going to be send in packages of known dimensions. The transmitter indicates to the receptor the moment that a package will start transmitting, and from that moment the clocks of both Arduinos work in parallel. After sending the package the devices go back to the resting state and a new package can be send .

Start bit

The resting state in UART is a continuous high voltage or a one (1), due to this the best way to check that the connection is working. If the initial state is a low voltage or a zero (0), it would be no difference between resting state and no signal for an error in the transmission. The transmitter sends the starting bit, indicating the receptor that the message is starting. This bit is always a zero (0) .

Data Transmission

After receiving the start bit the receiver knows that a message is starting and it has to start reading the data from next period. The measurement is taken in the middle part of each bit to avoid mistakes related to the voltage changing at the beginning or the end of the bit. To achieve this data transmission has to be known for both transmitter and receiver.

Parity bit

Parity bit is an error detection added at the end of the data transmitted. The parity bit will be equal to the sum of all the bits. This means that if the number of high bits transmitted is even, the parity bit will be a zero (0), and if the amount of high bits transmitted is odd, the parity bit will be a one (1). We can make example of parity for a 8-bit message: 01100110, in this case the amount of high bits is four (4). In this case, the parity bit would be a zero (0). In this example the full data sent by the transmitter would be: 011001100, including the parity bit. If for any reason the message received is 011000100, the receiver will add the first 8 bits and will compare the result with the parity bit. In this example the sum of the 8 bits would be one (1), and after comparing with the received parity bit, the receiver would know that there is an error in the data received. However, this is not the most efficient way to corroborate the data transmission, since in case two bits are gotten wrong, the sum to the parity bit will be correct. We can consider our example with two errors in it, e.g., 011000110. In this case the receiver will add 0 and after compare with the parity bit will detect that the data received is correct even though there are two incorrect bits.

Stop bit

Once the data package has been sent, the transmitter sends the final or stop bit. The stop bit is always a one (1), and this will the receiver to know that the message has finished.

Baud rate

To achieve a proper data transmission, the speed of the bit transmission must be known by the transmitter and the receiver. This speed is known as Baud rate, and it is measured in bauds per second. In the basic UART system, this is equivalent to bits per second. The bigger the baud rate, the less time that a bit needs to be transmitted. With this idea, the best option would be to use the highest possible baud rate, but the higher the speed Page 2 Microcontrollers transmission, the more possibilities to have transmission errors. The baud rate has to be slower that the processor speed.

SPI

The SPI is a serial communication based in four wires. Two wires; the MOSI, that connects the output of the master with the input of the slave; and the MISO, that connects the input of the master with the output of the slave. Besides this, SPI allow a single master to connect with several different slaves through an address. For this purpose we use a third line SS Slave Select and finally, to can synchronize the master device with the slaves, we use a SCK, serial clock signal. In the Arduino world we can communicate with SPI in the following way PIN D10 - SS (Slave Select) PIN D11 - MOSI (Master out, slave in) PIN D12 - MISO (Master in, slave out) PIN D13 - SCK (Serial clock)

12C

The I2C is a serial communication protocol based in two wires, a data line SDA, and a clock line SCL. This communication protocol allows us to connect different Slaves with a single Master with only one data line. for this purpose, it is needed to send the information of the Slave that is required to access into the same message, the server will check if it is the one who has been selected and will proceed to receive and send information, if the address is not the one that corresponds to it, it will ignore the call. To connect this protocol in the Arduino world the following PINs are used: PIN A4 or SDA PIN A5 or SCL.

Implementations:

UART: https://www.tinkercad.com/things/9uzpKeoRpaS-uart

SPI: https://www.tinkercad.com/things/8HLktYa1ruB-spi

12c: https://www.tinkercad.com/things/a7pQxeEBNDz-12c