Communication Protocols

Shrawan

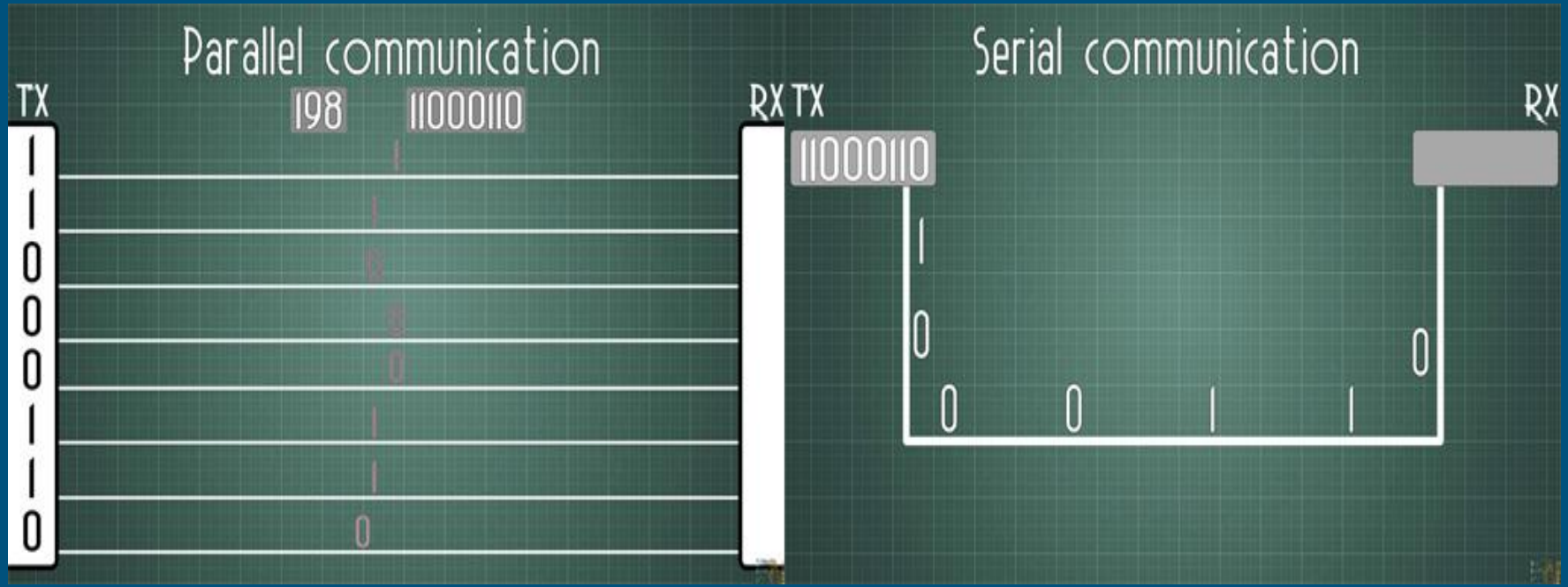
Communication

- Embedded electronics is all about interlinking circuits (processors or other integrated circuits) to create a symbiotic system.
- In order for those individual circuits to swap their information, they must share a common communication protocol.
- Hundreds of communication protocols have been defined to achieve this data exchange, and, in general, each can be separated into one of two categories: **Parallel or Serial.**

Parallel Vs Serial Communication

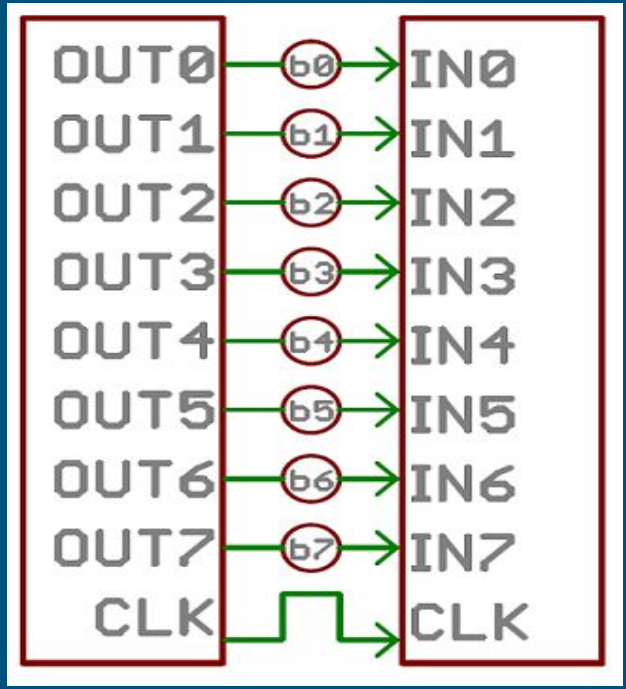
- Parallel interfaces transfer multiple bits at the same time.
- They usually require **buses** of data - transmitting across eight, sixteen, or more wires.
- Data is transferred in huge, crashing waves of **1's and 0's**.

Parallel Vs Serial Communication



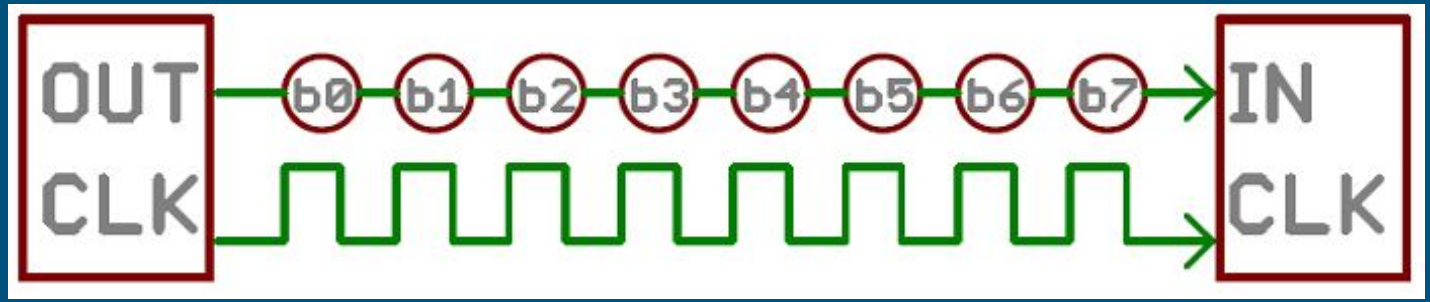
Parallel Communication

An 8-bit data bus, controlled by a clock, transmitting a byte every clock pulse. 9 wires are used.



Serial Communication

- Serial interfaces stream their data, one single bit at a time. These interfaces can operate on as little as one wire, usually never more than four.
- Example of a serial interface, transmitting one bit every clock pulse. Just 2 wires required!



Parallel Vs Serial Communication

Parallel communication certainly has its benefits. It's fast, straightforward, and relatively easy to implement.

But it requires many more input/output (I/O) lines. If you've ever had to move a project from a basic **Arduino Uno to a Mega**, you know that the I/O lines on a microprocessor can be precious and few.

So, we often opt for serial communication, sacrificing potential speed for pin real estate.

Asynchronous Serial

- A synchronous serial interface always pairs its data line(s) with a clock signal, so all devices on a synchronous serial bus share a common clock. This makes for a more straightforward, often faster serial transfer, but it also requires at least one extra wire between communicating devices.
- Examples of synchronous interfaces include SPI, and I2C.
- Asynchronous means that data is transferred **without support from an external clock signal**. This transmission method is perfect for minimizing the required wires and I/O pins, but it does mean we need to put some extra effort into reliably transferring and receiving data.

Baud Rate

- The baud rate specifies **how fast** data is sent over a serial line. It's usually expressed in units of bits-per-second (bps).
- Baud rates can be just about any value within reason. The only requirement is that both devices operate at the same rate. One of the more common baud rates, especially for simple stuff where speed isn't critical, is **9600 bps**. Other "standard" baud are 1200, 2400, 4800, 19200, 38400, 57600, and 115200.
- The higher a baud rate goes, the faster data is sent/received, but there are limits to how fast data can be transferred. You usually won't see speeds exceeding 115200 - that's fast for most microcontrollers. Get too high, and you'll begin to see errors on the receiving end, as clocks and sampling periods just can't keep up.

Framing the Data

Each block (usually a byte) of data transmitted is actually sent in a packet or frame of bits. Frames are created by appending synchronization and parity bits to our data.

A serial frame. Some symbols in the frame have configurable bit sizes.

Synchronization bits

The synchronization bits are two or three special bits transferred with each chunk of data. They are the **start bit** and the **stop bit(s)**.

The start bit is always indicated by an idle data line going from 1 to 0, while the stop bit(s) will transition back to the idle state by holding the line at 1.

Parity bits

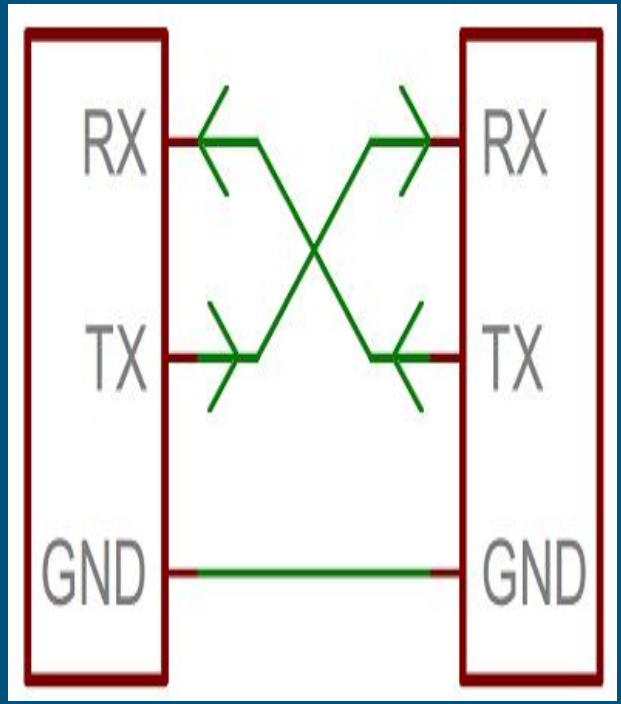
Parity is a form of very simple, **low-level error checking**. It comes in two flavors: odd or even.

To produce the parity bit, all 5-9 bits of the data byte are added up, and the evenness of the sum decides whether the bit is set or not.

For example, Assuming parity is set to even and was being added to a data byte like `0b01011101`, which has an odd number of `1`'s (5), the parity bit would be set to `1`. Conversely, if the parity mode was set to odd, the parity bit would be `0`.

Wiring and Hardware

- A serial bus consists of just two wires - one for sending data and another for receiving. As such, serial devices should have two serial pins: the receiver, **RX**, and the transmitter, **TX**.
- It's important to note that those *RX* and *TX* labels are with respect to the device itself. So the RX from one device should go to the TX of the other, and vice-versa. It's weird if you're used to hooking up VCC to VCC, GND to GND, MOSI to MOSI, etc., but it makes sense if you think about it. The transmitter should be talking to the receiver, not to another transmitter.
- A serial interface where both devices may send and receive data is either **full-duplex** or **half-duplex**. Full-duplex means both devices can send and receive simultaneously. Half-duplex communication means serial devices must take turns sending and receiving.



Serial Communication

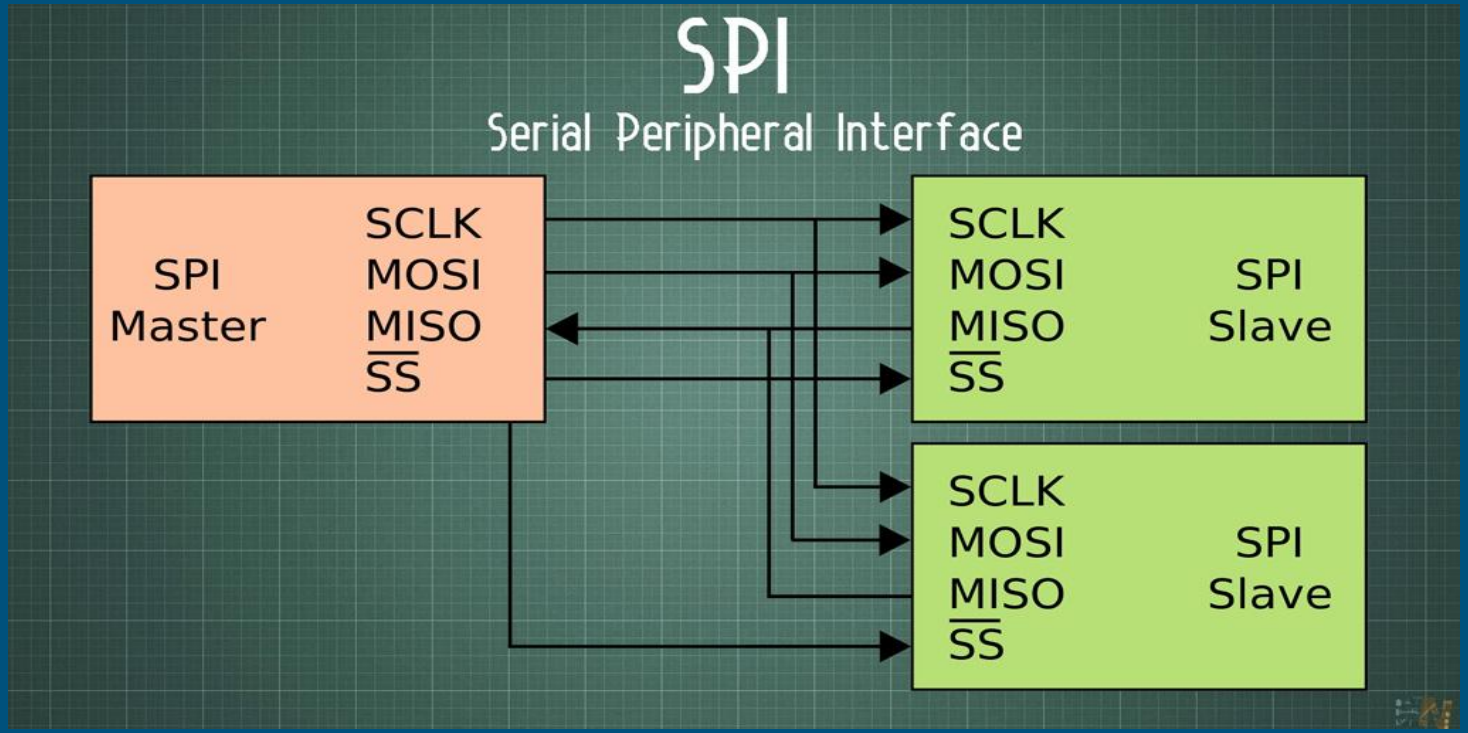
1. SPI (Serial Peripheral Interface)
2. I2C (Inter-Integrated Circuit)
3. UART (Universal Asynchronous Transmitter Receiver)

SPI (Serial Peripheral Interface)

This is a synchronous type serial communication protocol which consists of two data lines (MOSI and MISO), one clock line (SCK) and a slave select line (SS). Before moving ahead here are some terms that you should be aware of:

- **Master** – Device which provides clock for communication
- **Slave** – Device other than master which utilises master's clock to communicate
- **MOSI** – Master Out Slave In (line though which master sends data to its slaves)
- **MISO** – Master In Slave Out (line though which salves responds back to the master)
- **SCK** – Serial Clock (clock provided by master device)
- **SS** – Slave Select (line used to select slave to which master wants to communicate)

SPI (Serial Peripheral Interface)



SPI (Serial Peripheral Interface)

There are few register which are used to implement SPI communication. We have these below: SPDR, SPSR and SPCR

- ➔ **SPDR (SPI Data Register)**
- ➔ This is used to store one byte of data which is to be transferred or received.
- ➔ **SPSR (SPI Status Register)**
- ➔ This register holds the status bits involved in SPI communication
- ➔ **SPCR (SPI Control Register)**
- ➔ This register holds the control bits involved in SPI communication.

All the above registers are 8 bit in length.

SPI (Serial Peripheral Interface)

Advantages:

- Provides synchronous serial communication which is much more reliable over asynchronous.
- Multiple devices(Slaves) can be connected to single master
- Faster form of serial communication.

Disadvantages:

- Requires multiple slave select wires for connecting multiple slaves
- Only master has control over entire communication process; no two slaves can communicate with each other directly.

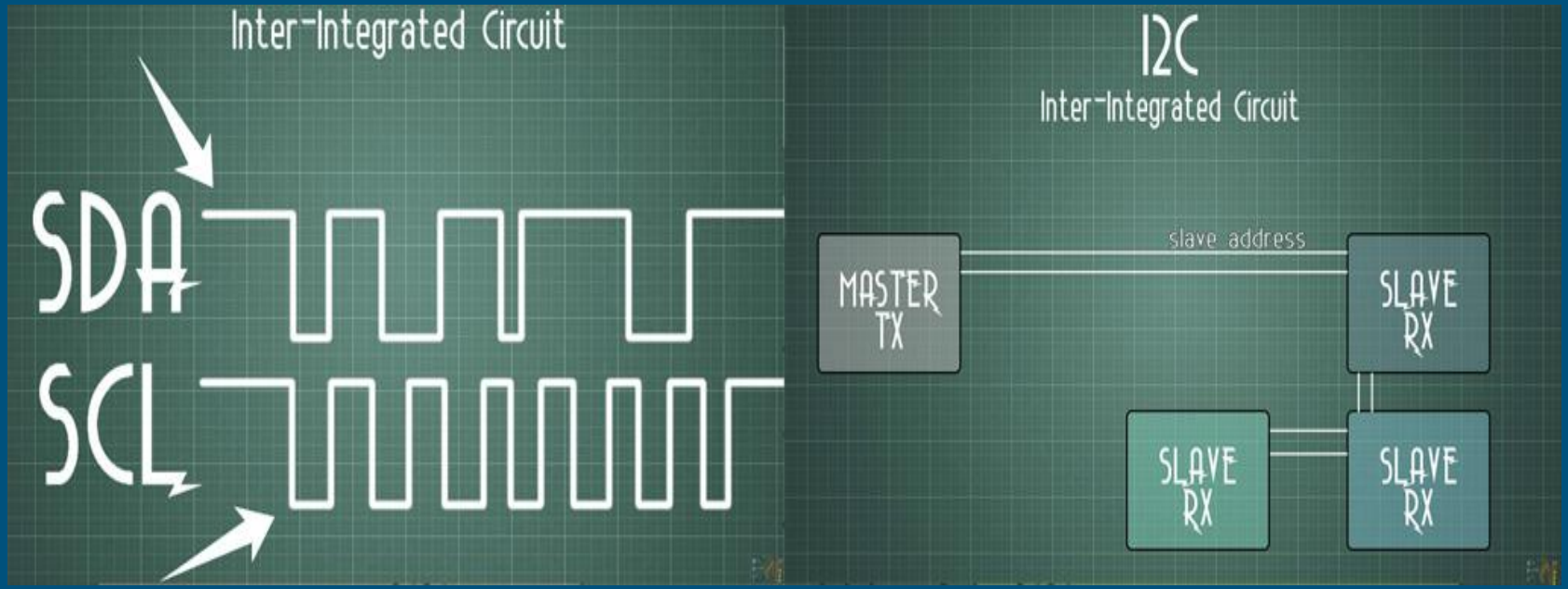
I2C (Inter-Integrated Circuit or Two wire interface)

Another very useful synchronous serial communication protocol is I2C or Inter-Integrated Circuit protocol. Unlike SPI, I2C uses only two wires for the entire process, maybe that's why it is also known as Two Wire Interface (TWI) protocol.

These two wires are SDA (Serial Data) and SCL (Serial Clock). I2C protocol can support multiple slave devices but unlike SPI, which only supports one master device, I2C can support multiple master devices as well.

Every device sends/receives data using only one wire which is SDA. SCL maintains sync between devices through common clock which is provided by the active master.

I2C (Inter-Integrated Circuit or Two wire interface)



I2C (Inter-Integrated Circuit or Two wire interface)

Each slave has its own unique 7 to 10 bit address which master uses to identify them. Whenever master wants to send data it first generates a request which has particular address of that slave.

Every slave matches this address with its own and the one whose address gets matched responds to the master.

Every message initiates with a start condition and ends with a stop condition.

A single message can hold multiple data bytes, each having an acknowledge (ACK) or negative acknowledge (NACK) bit in between them.

I2C (Inter-Integrated Circuit or Two wire interface)

Advantages:

- Multiple masters and multiple slaves can be interfaced together
- Only two wires are required for this communication

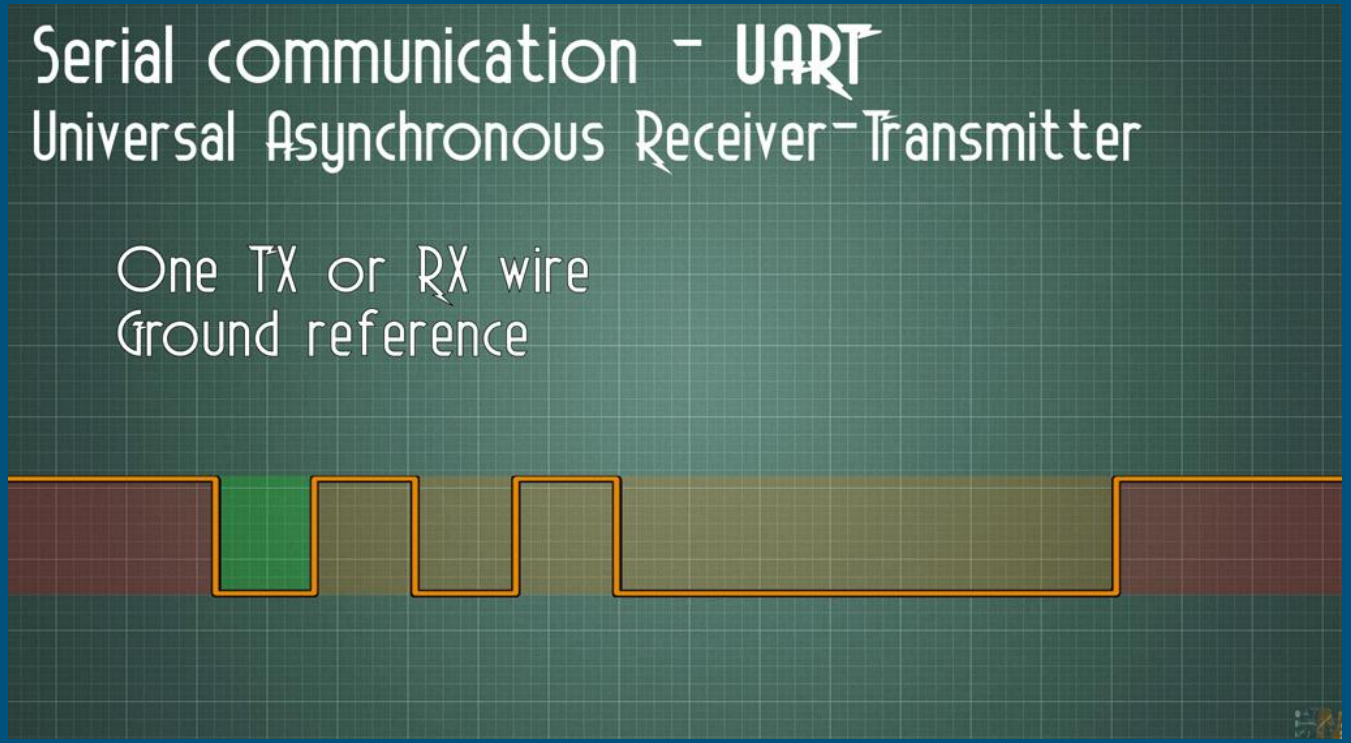
Disadvantages:

- It is slower as compared to SPI because a lot of framing work is done within this protocol

UART (Universal Asynchronous Transmitter Receiver)

- UART stands for Universal Asynchronous Receiver and Transmitter while USART stands for Universal Synchronous and Asynchronous Receiver and Transmitter. The difference between them is that UART performs only asynchronous serial communication while USART can perform both synchronous as well as asynchronous serial communication process.
- For Asynchronous mode, this protocol makes use of only two wires i.e. Rx and TX. Since no clock is needed here, both the devices have to make use of their independent internal clocks to work.

UART (Universal Asynchronous Transmitter Receiver)



UART (Universal Asynchronous Transmitter Receiver)

Advantages:

- Provides both synchronous as well as asynchronous serial communication
- Availability of various baud rates making it suitable for wide applications and devices
- One of the easiest form of serial communication

Disadvantages:

- Can connect only two devices at a time

Conclusion

Use SPI when you have only one master and multiple slave devices. SPI proves to be a faster protocol for this.

When you have multiple master devices as well, apart from multiple slave devices, then one should prefer using I2C or TWI over SPI.

This will also reduce the number of wires to be used.

Now if you are looking for a device to device serial communication then USART/UART proves itself the best as it is easy to deal with and widely used in many peripheral devices.