UCDAVIS
UNIVERSITY OF CALIFORNIA

Internet of Things

Senior Design Project Course

Digital Communication protocols

Part 1

Lecturer: Avesta Sasan

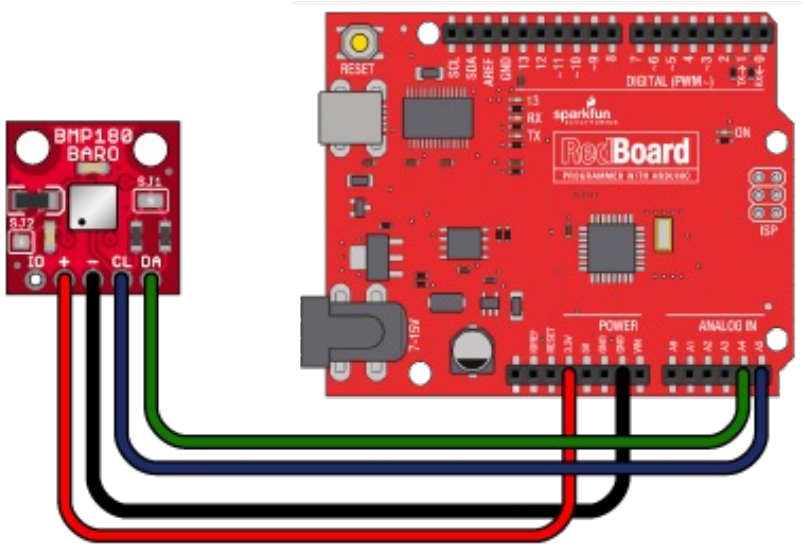
University of California Davis

Focus of Today's Lecture

Image source: http://www.cchc.cl/informacion-a-la-comunidad/industria-de-la-construccion/personaje/

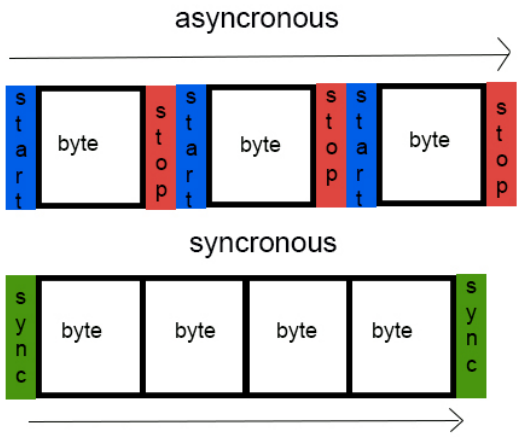
Communication with Digital Sensors

- To read digital sensors you need to use the sensor's **interface protocols**
- Interface Protocols could be **Synchronous** or **Asynchronous**.
- Interface Protocol could be **Serial** or **Parallel**.



Synchronous vs Asynchronous

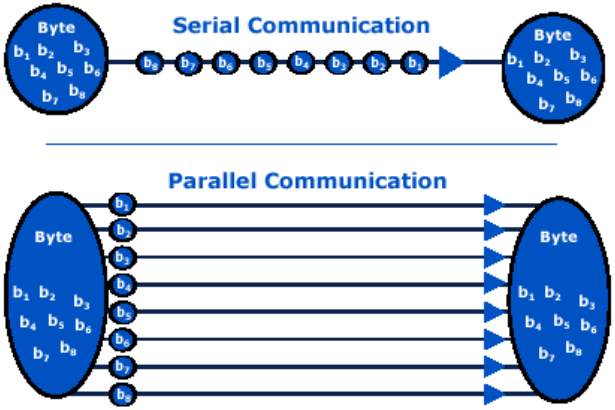
- **Synchronous**
  - Uses Clock!
  - Pairs its data line(s) with a clock signal.
  - Clock is shared among all devices
  - Faster serial transfer
  - Straight forward implementation
  - Examples: **SPI, I2C**
- **Asynchronous**
  - Doesn't need a clock
  - Need lower number of wires (no clock!)
  - Usually slower transfer
  - Implementation is more difficult
  - Examples: **TTL, RS32**



Parallel vs Serial

- **Parallel Interface**
  - Transfer multiple bits at a time.
  - Require a bus (group of wires)
  - Higher performance
  - Require too many IO pins

- **Serial Interface**
  - Transfer a single bit at a time.
  - Require a single wire
  - Higher overhead
  - Lower performance
  - Require single or only a few wires



How about a hybrid approach?

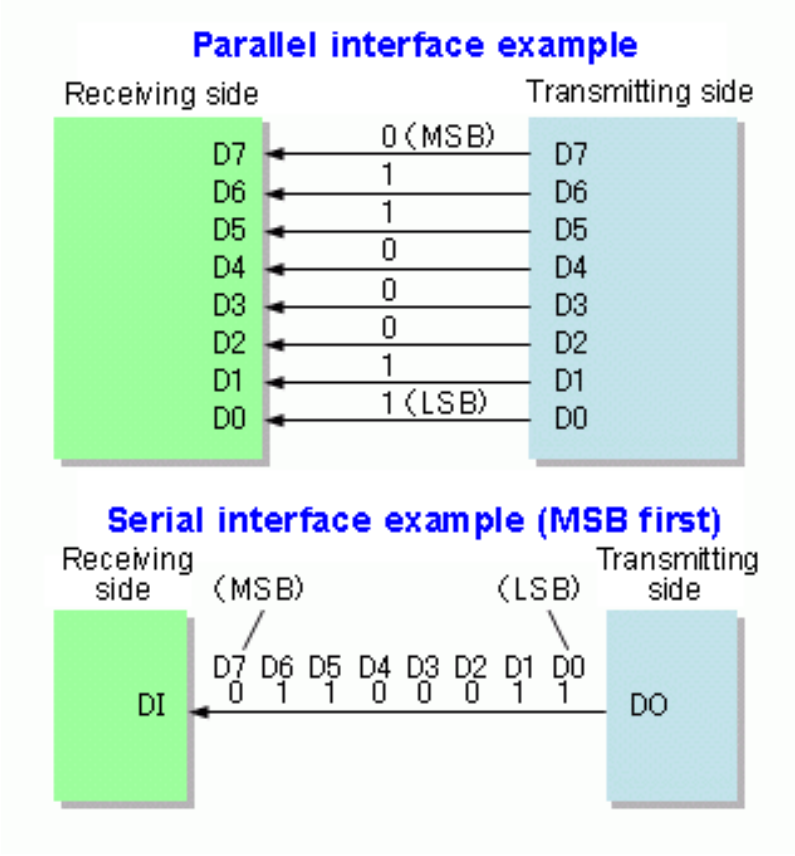
Which one is easier to implement?

Which one is more natural to use?

Serial Communication

- To exchange information, we always need a well defined protocol.
- Many to/from sensor communication protocols exist, in this lecture we will cover a few!

- Sensor communication could be
  - **Parallel Interface:**
    communicating multiple bits at the same time
  - **Serial Interface:**
    communicating one bit at a time

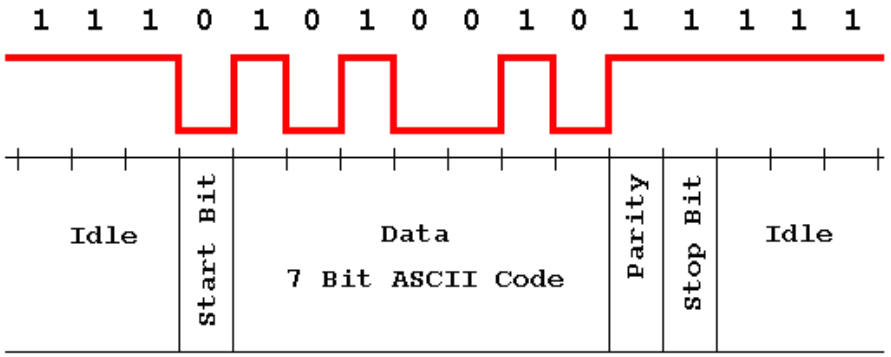


Asynchronous Comm. Protocol

- To implement the Asynchronous communication protocol, we need to build our data packets using a predefined (agreed upon) format.

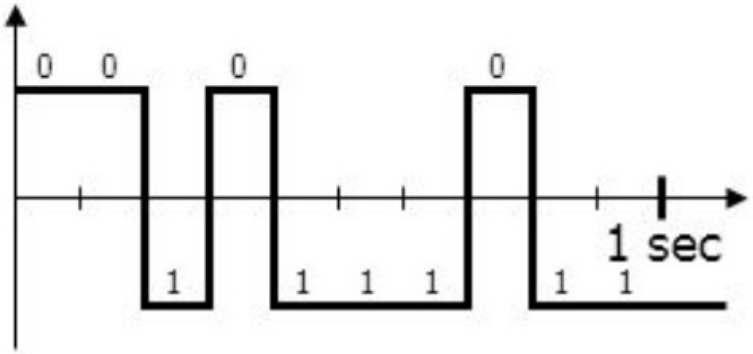
- A common way to build this data packet is by using the following sections on a data transfer packet:
  - Start bit
  - Data bits
  - Parity Bit
  - Stop Bit(s)

- We also need to agree on the speed that the data is transferred!

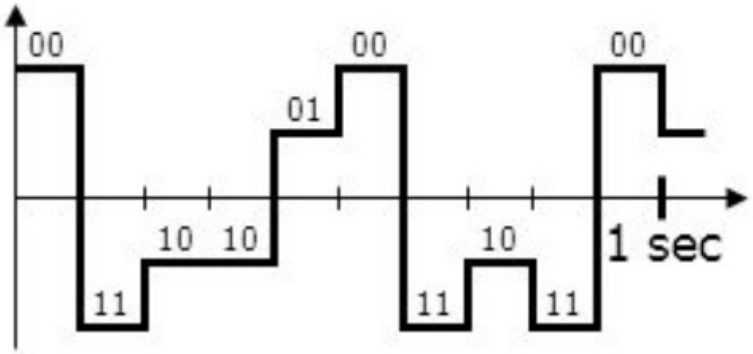


Baud Rate

- **Baud rate:** How many times a signal changes per second.
- **Bit Rate:** How many bits an be sent per second.
- Bit rate is controlled by Baud rate and the number of signal levels.
- **Can you find the relationship between Baud rate and Bitrate below?**
- **If we only have signals with two level, what is the relationship?**
- Baud rate is measure by the unit: **b**its **p**er **s**econd (bps)
- Standard Baud rates are 200, 2400, 4800, 9600, 19200, 38400, 57600, and 115200bps.

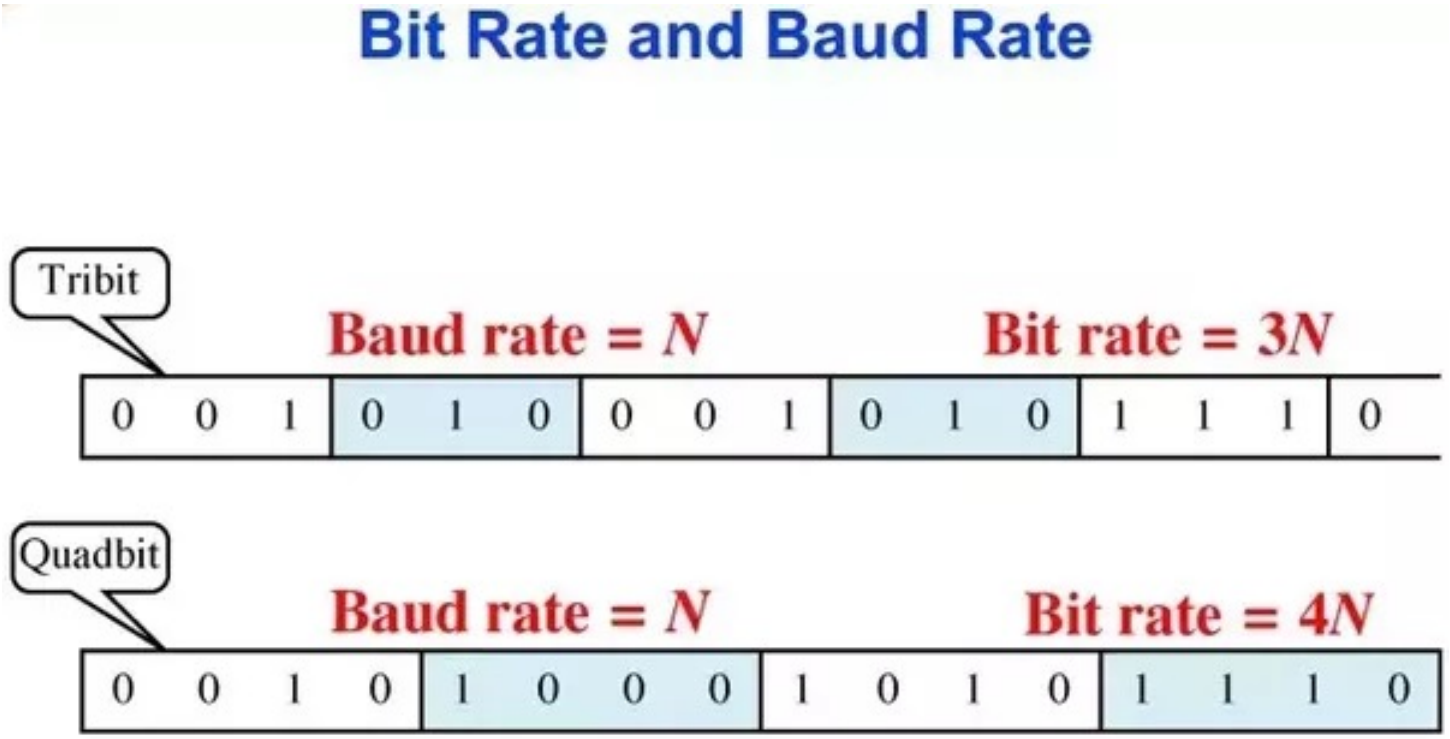


Baud = 10
Bit rate = 10 bps



Baud = 10
Bit rate = 20 bps

Baud Rate vs Bit Rate



Packaging the Data

- Each packet of data is called a **frame**.
- **Start Bit:** A synchronization bit that mark the beginning of the data,
  - Start bit is always one bit.
  - Indicate the start of a transition by going from 1 to 0.
- **End bit:** Synchronization bit(s) to mark the end of the data.
  - One or two bits (depending on the implementation)
  - Indicates the end of a transition by pulling the line to 1

Packaging the Data

- **Data bits:** contain the information of interest!
  - It is also called a **chunk**
  - Data could be set to a value between 5 to 9 bits
  - **8 bits is a natural choice (size of a Byte), but why use other sizes?**
  - We also have to agree if data is **Big Endian** or **Little Endian**?

Big Endian vs Little Endian

- The order that bits are transferred! MSB first or LSB first:

- **Exercise**: A Microprocessor has received the following byte (**01100100**) of information. What value it represent if it is Big Endian? What if it was Small Endian?

ANIMATED RESPONSE FOR BIG ENDIAN

ANIMATED RESPONSE FOR LITTLE ENDIAN

Packaging the Data

- Parity bit (**optional**): Very simple way of error checking
  - Add up the bits, if even number then it is 0, if odd number it is 1. (or vise versa)
  - Can detect one bit flip error. (can it detect two)?
  - Slow down the data transfer (need error coding encoder and decoder)
  - Improves reliability

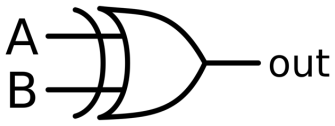
Original Data	Even Parity	Odd Parity
0 0 0 0 0 0 0 0	0	1
0 1 0 1 1 0 1 1	1	0
0 1 0 1 0 1 0 1	0	1
1 1 1 1 1 1 1 1	0	1
1 0 0 0 0 0 0 0	1	0
0 1 0 0 1 0 0 1	1	0

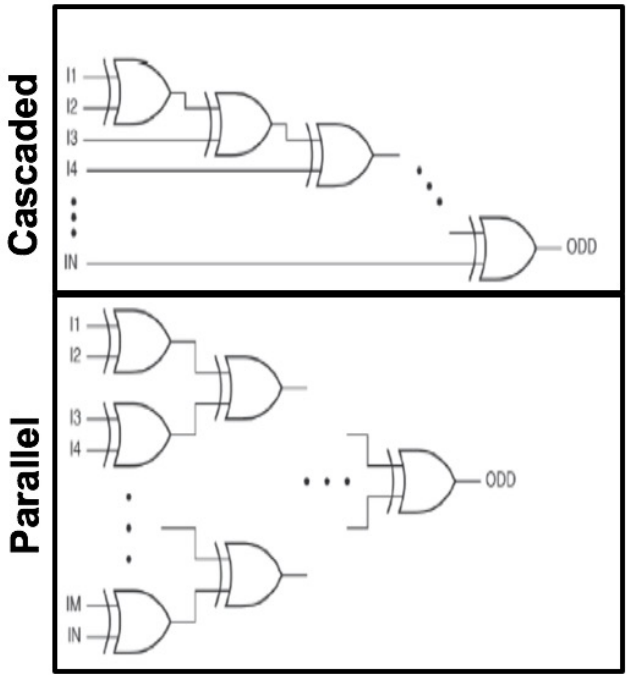
- Now you now everything that you need to know to build a packet.

- **Exercise**: Build an **optimized** format for a frame that uses **little endian** format. It is used to read sensor information that are between values [**0, 232**]. It uses **1 bit even Parity** and we intend to send **value 208** from sensor to the receiver, and use a **single bit for end signal**.

Parity Bit Generator Circuit

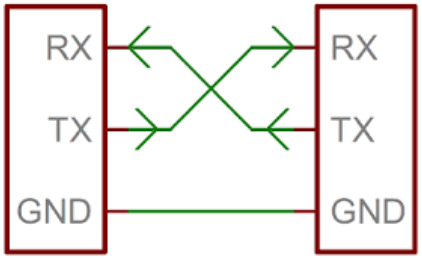
- Simplest way: Use XORs
- Could cascade the XORs or form a XOR tree
  - What is the tradeoff?
- **Question:** How do you change the parity between **ODD** and **EVEN**?
- **Answer**: XOR the results with value one or zero to change parity mode!



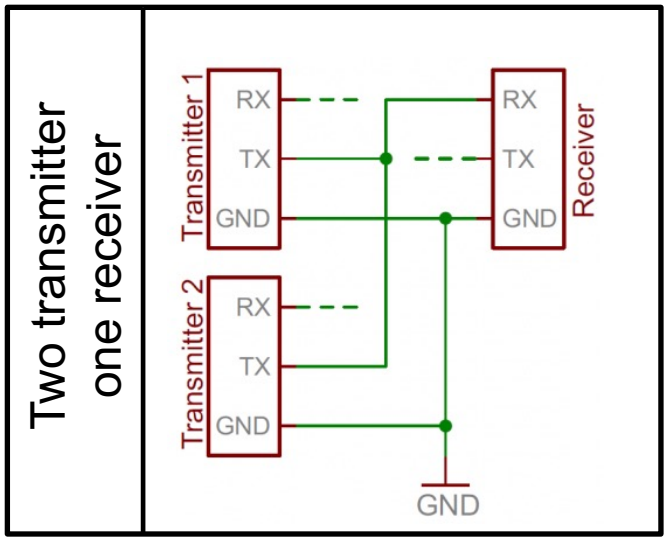


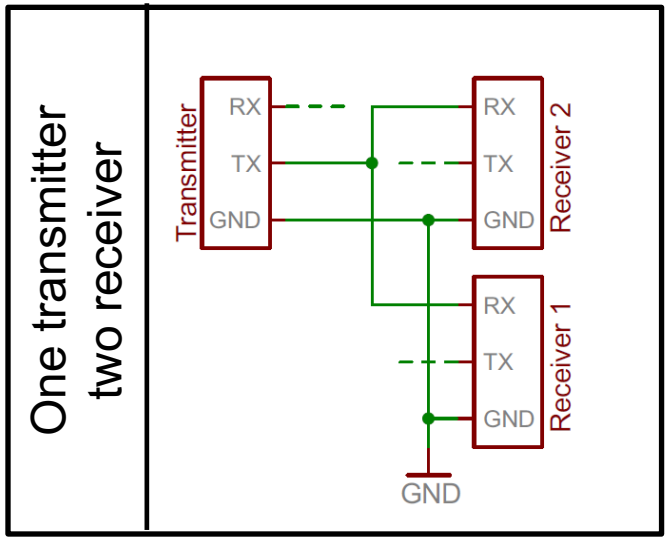
How to Connect Asynch Devices?

- TX to RX



- **Question**: How do we setup the circuit if we have multiple transmitters and/or multiple receivers?

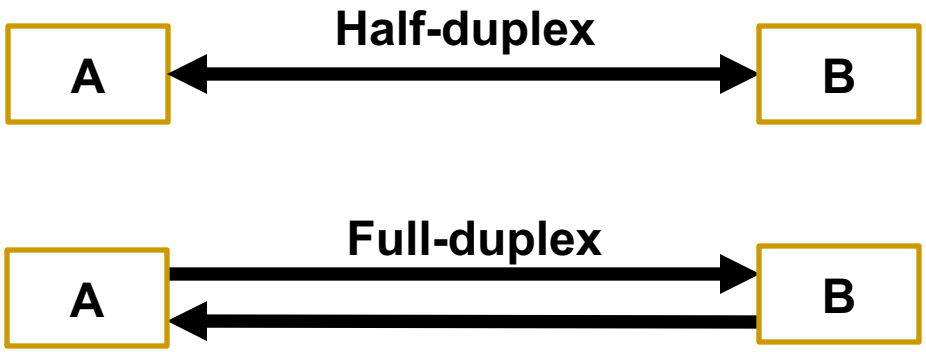




- **Question:** Why do you need a common GND?

More Details!

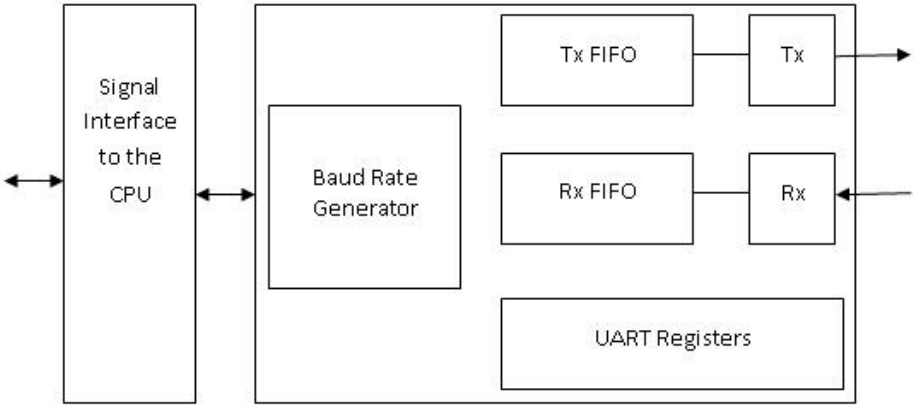
- Serial Asynchronous communication could be
  - **Full-duplex**: both devices can send and receive data simultaneously.
  - **Half-duplex**: devices should take turn in transmitting data.

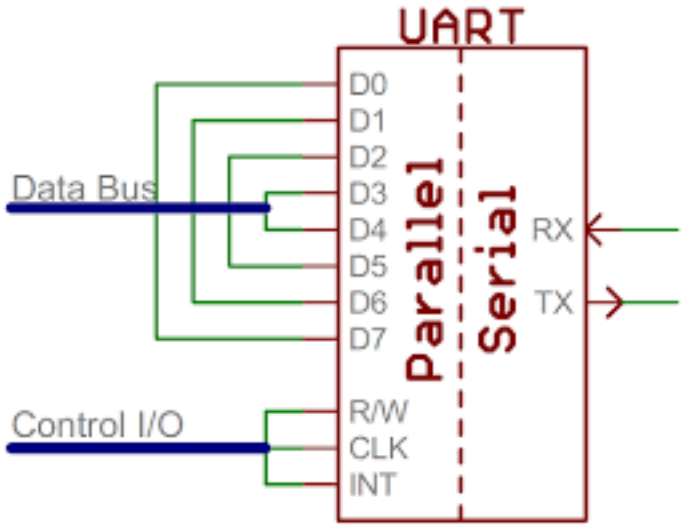


- Make sure the voltages match up!

UART

- **U**niversal **A**synchronous **R**eceiver/**T**ransmitter (**UART**)
- Implements the serial communication!
- Act as intermediate between parallel and serial data
- Many microcontrollers have the UART block internally, but you can also have it as a standalone block.
- Refer to the following link to learn how to use UART functions in Arduino:
  - https://www.arduino.cc/en/Reference/Serial





Asynchronous Communication Problems

- No precise control over when data is sent/received
- **Tx** and **Rx** should agree prior to data transfer on Baud rate
- **What if clocks are slightly off?**
  - Extra start and stop bits to somewhat take care of this problem, but it is communication overhead!
- Hardware required to compose or decode the data frame is more complex than parallel communication protocols.
- **Synchronous communication solve all this problems!**
  - **As an example we see how SPI protocol work, and slightly cover I2C.**

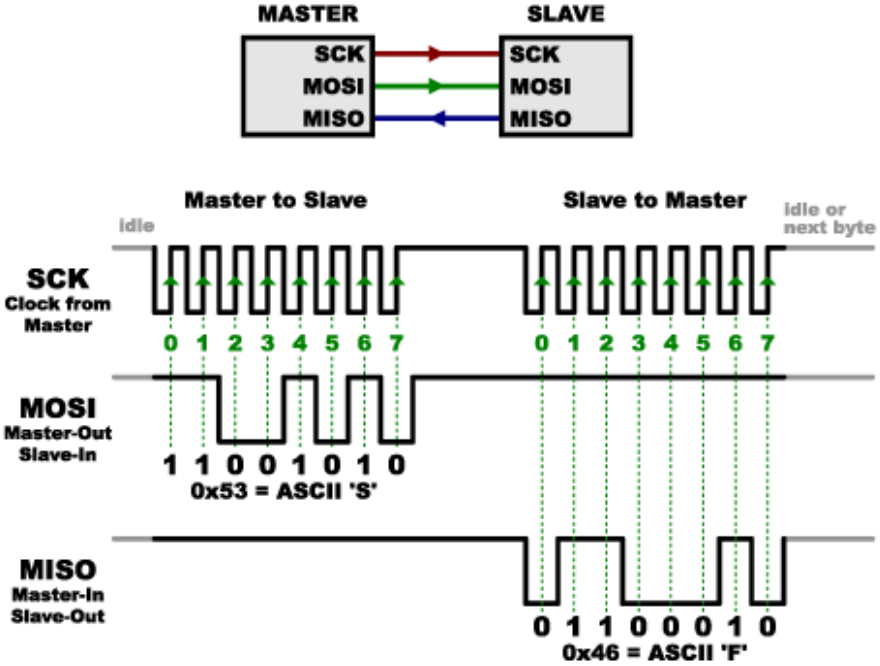
SPI: A Synchronous Comm. Protocol

- Separate data and clock wires!
- Data is read on rising or falling edge of clock
- Clock and data are send at the same time
  - Hence, no need to pre-arranged baud rate
- Receiving hardware can be as simple as a shift register
- Clock is always generated by Master
  - But both master and slave can receive and transfer data
- Always one master, but could have multiple slaves!
  - I2C protocol solves this problem!

SPI: Data Exchange!

- **MOSI:** Master Out Slave In
  - Line to send data from Master to Slave

- **MISO:** Master In Slave Out
  - Line to send data from Slave to Master
  - The number of cycles is **pre-arranged**, generated by Master, during which slave toggle the MISO



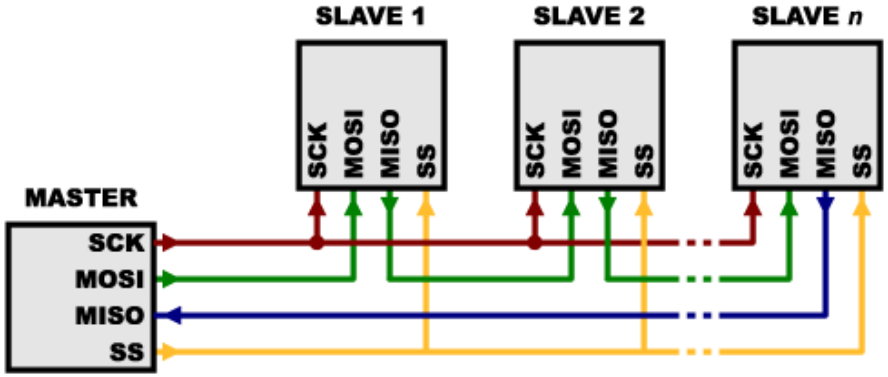
SPI: Data Exchange

- Pre-arranging the number of needed clock cycles could be achieved by
  - **Knowing the Command behavior in advance!**
    Using known command to communicate with sensors whose responses are known and are always fixed in length!

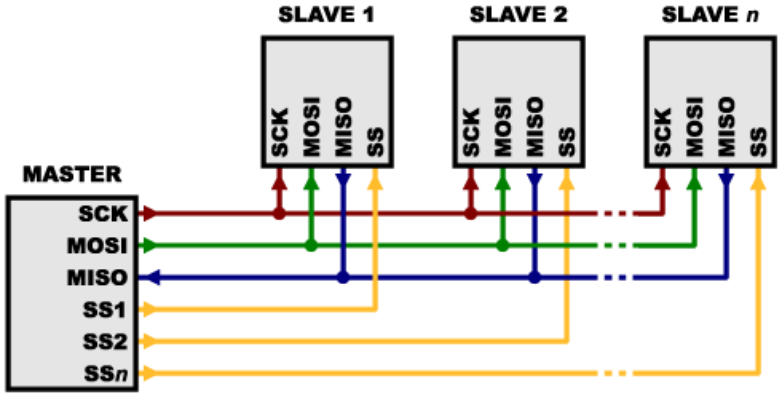
  - **Data length is communicated:**
    This is a two step process!
    1. The sensor send a fixed response (1 or 2 bytes), by which it specifies the length of data.
    2. The Master generates the number of needed clock cycles to transfer the specified data from Slave to Master.

SPI: Slave Select

- SPI has **one Master**, but could have **multiple slaves**
  - How do we select the slave?
- Slave Select (SS) or Chip Select (CS) is used to select the slave node that SPI communicate with.
- Two way to connect multiple Slaves:
  - **Direct:** Data is directly transferred to the target.
  - **Daisy Chain:** Data flow through many slaves to reach target.



Daisy Chain Connection



Direct Connection

Using SPI on Arduino Microprocessor

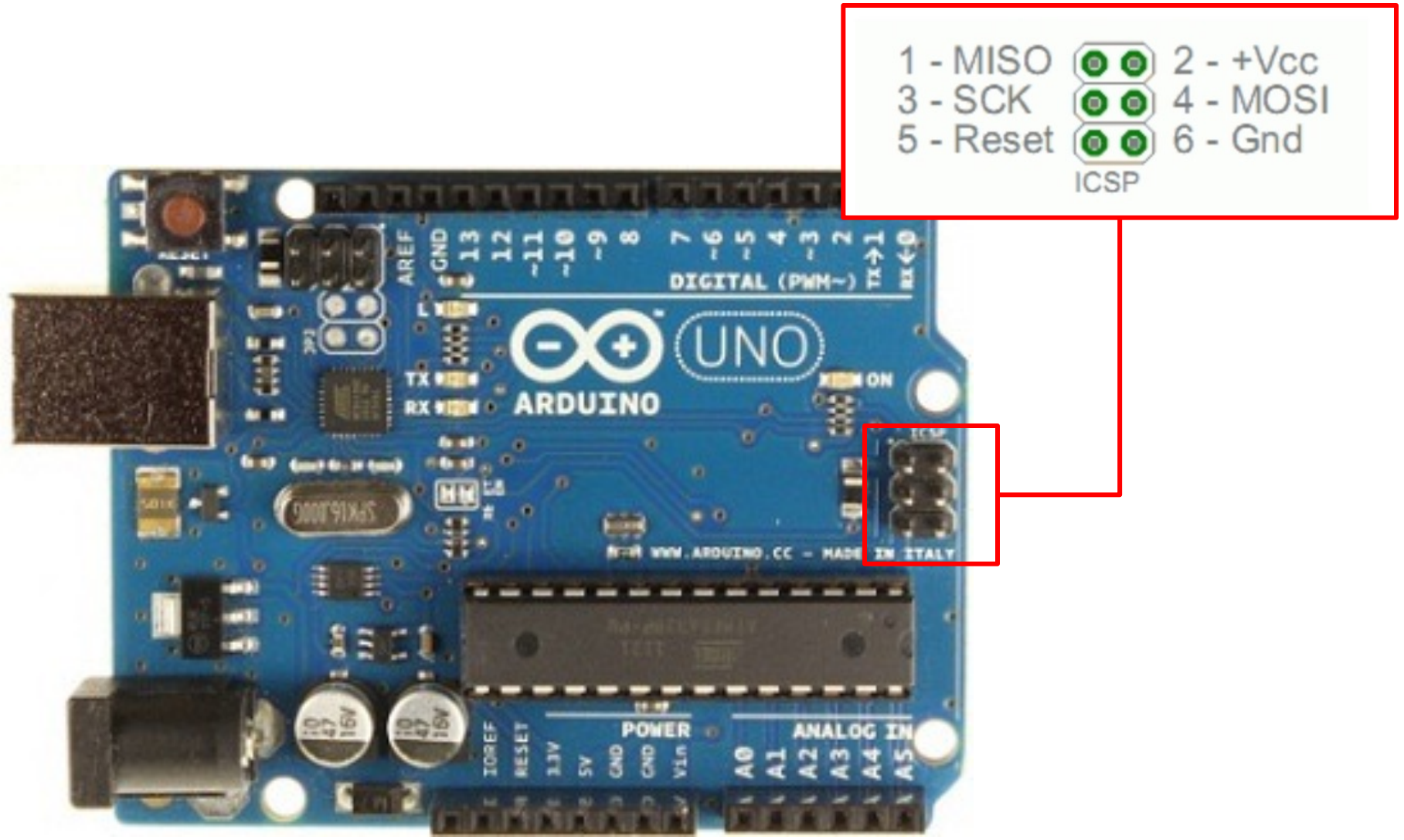
- Some Microprocessors have **build in SPI peripherals**
- SPI is simple, so your could also write your SPI protocol
  - http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus
  - https://www.arduino.cc/en/Reference/SPI
- Example Arduino Microprocessor:
  - Has SPI peripheral support and the associated library
  - SPI could be configured using library defined function:
  - **setBitOrder():** sets the data transfer endian (Big vs Little).
    - https://www.arduino.cc/en/Reference/SPISetBitOrder
  - **setDataMode():** sets whether clock is idle at high or low.
    - https://www.arduino.cc/en/Reference/SPISetDataMode
  - **setClockDivider():** sets he clock frequency.
    - https://www.arduino.cc/en/Reference/SPISetClockDivider
  - **Transfer():** performs chip select, data transfer and chip deselect
    - https://www.arduino.cc/en/Reference/SPITransfe
- Read more: https://www.arduino.cc/en/Reference/DueExtendedSPI

SPI on Arduino (Power of Library!)

```
void setup(){
  // initialize the bus for a device on pin 4
  SPI.begin(4);
  // initialize the bus for a device on pin 10
  SPI.begin(10);
}
```

```
void setup(){
  // initialize the bus for the device on pin 4
  SPI.begin(4);
  // Set clock divider on pin 4 to 21
  SPI.setClockDivider(4, 21);
  // initialize the bus for the device on pin 10
  SPI.begin(10);
  // Set clock divider on pin 10 to 84
  SPI.setClockDivider(10, 84);
}
```

MISO, MOSI and other pins for SPI



SPI on Arduino (Power of Library!)

- **SPI.transfer(pin, value)** command in the library take care of chip select, data transfer, and chip deselect.

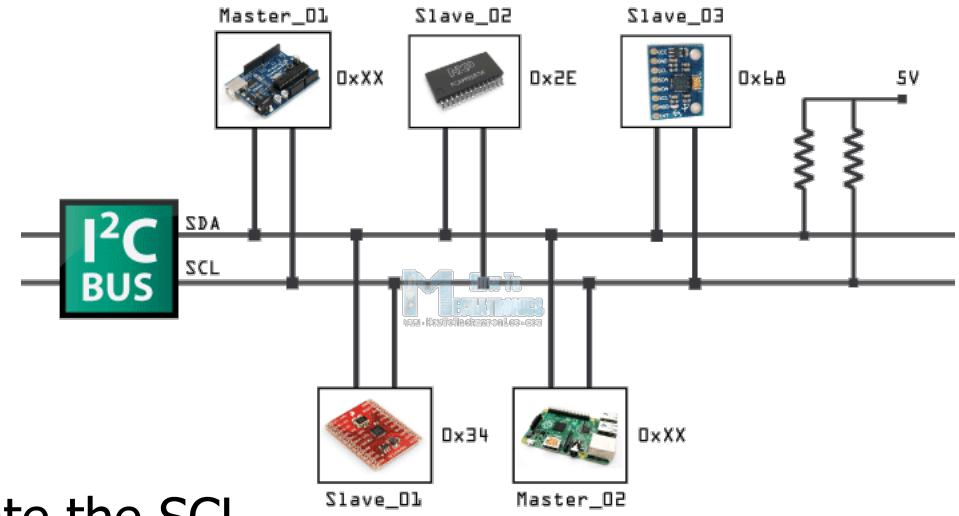
- For example:

```
void loop(){
  byte response = SPI.transfer(4, 0xFF);
}
```

  - **Select device** by setting pin 4 to LOW
  - **Send** 0xFF through the SPI bus and return the byte received
  - **Deselect device** by setting pin 4 to HIGH

I2C: A Synchronous Comm. Protocol

- Requires only two wires
  - **SDA:** Data Signal
  - **SCL:** Clock Signal



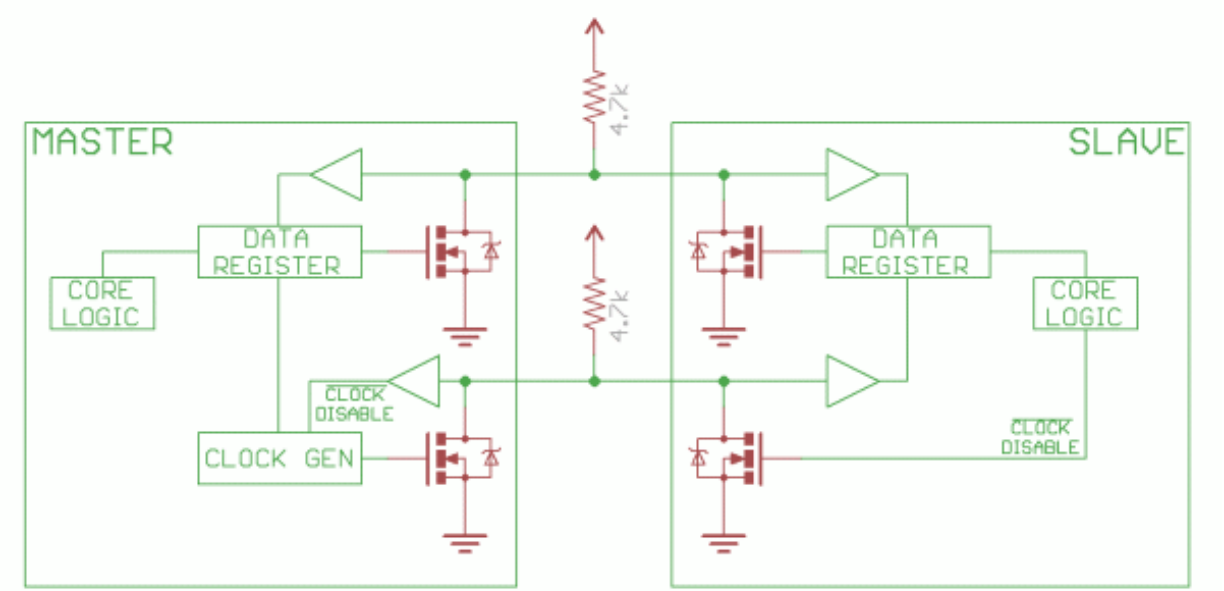
- Can have multiple Master
  - The **current** Master generate the SCL

- Slaves can not generate the clock, but can force the clock down to prevent data transfer (Clock Stretching) to consume or generate data.
  - http://howtomechatronics.com/tutorials/arduino/how-i2c-communication-works-and-how-to-use-it-with-arduino/

More Details on I2C

- Bus drivers are open drain: Can pull the bus down, but can't pull it up.
  - Prevents **bus contention** (two components, one trying to pull the bus down and one trying to pull up!)

- Each signal line has a pull up resistor to pull up the wire.
  - Larger resistors for short distances (2-3 meters), lower resistors for long distances (> 3 meters)



I2C: How to Connect?

- As easy as connecting the SDA and SCL pins to the existing wires.

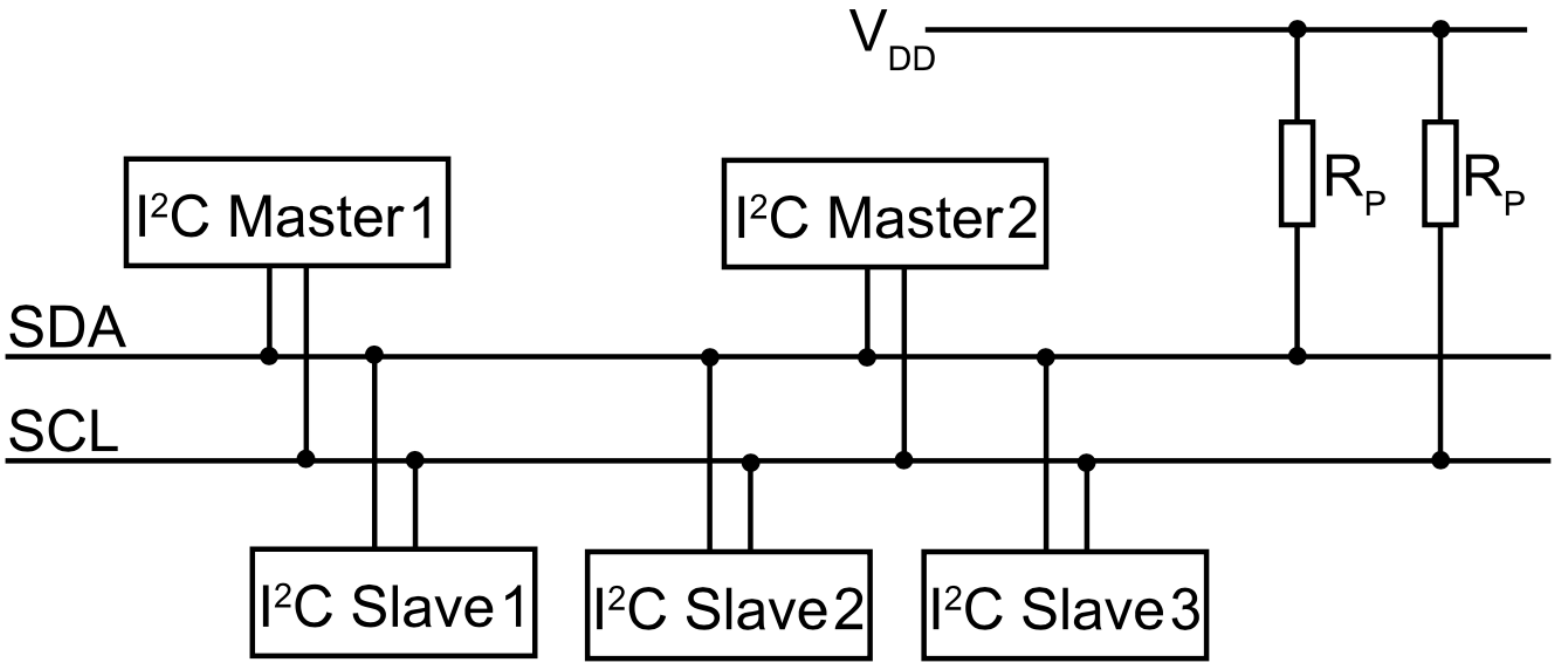


Image source: http://dlnware.com/i2c

SPI vs I2C

I2C	SPI
Speed limit varies from 100kbps, 400kbps, 1mbps, 3.4mbps depending on i2c version.	More than 1mbps, 10mbps till 100mbps can be achieved.
Half duplex synchronous protocol	Full Duplex synchronous protocol
Support Multi master configuration	Multi master configuration is not possible
Acknowledgement at each transfer	No Acknowledgement
Require Two Pins only SDA, SCL	Require separate MISO, MOSI, CLK & CS signal for each slave.
Addition of new device on the bus is easy	Addition of new device on the bus is not much easy a I2C
More Overhead (due to acknowledgement, start, stop)	Less Overhead