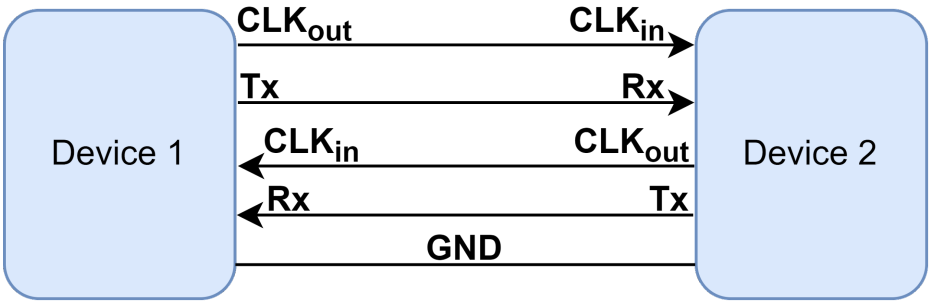
# Lab 1 - USART




## Implementation:

We will be implementing a Full-Duplex communication scheme (see above image). *Each device* will have its own channels:

- Clock Channel: the clock channel synchronizes the bits transmitted on the data channel.
  Implementation-wise, it's a constant “0101…” stream from the transmitter to the receiver. Each bit is “`BIT_TIME`” 1/50 seconds (i.e., the clock is 50Hz).
- Data Channel: the data channel will transmit data between the boards. Data will be transmitted serially (bit by bit).
  Data is written (sent to channel) when the clock channel rises, and data is read when the clock falls.

You are required to create the following two functions. The functions will work with 1 Byte (8 bits) of data, sending the LSB first. **Note that the number of bits will vary in the following labs, so make sure you can change the number of bits swiftly (by using defines, up to 12 bits). Your work should transmit and receive indefinitely (non-stop)**:

1. **usart_tx()** - sends data and clock to the other device. After the transmission, the transmitter will not send information for a random amount of time. **Do not use any sleep/delay/while/for commands.**
2. **usart_rx()** - reads the clock pin to find if a transmission is in progress. If it does, read the data pin when the clock falls from ‘1’ to ‘0’.

You cannot use “Busy Waiting” in your implementation. Each call to ”`usart_tx()`“ is followed by ”`usart_rx()`“ (or vice-versa). I.e., your code should look like this (up to order to your liking):

```
void loop() {
      usart_tx();
      usart_rx();
}
```

# General Tips:

1. **Read all general tips; they might help you with your code, questions, and defenses.**
2. First you must set the pins: `pinMode(TX_PIN, OUTPUT)`, `pinMode(RX_PIN, INPUT)`
3. Use "`digitalWrite(PIN_NUMBER, HIGH/LOW)`" and "`digitalRead(PIN_NUMBER)`" to set and get the logical state of some pin.
4. The functions "`millis()`" and "`micros()`" return timestamps relative to when the device is turned on. They return an "`unsigned long`" value, allowing to measure times like `BIT_TIME`.
   A rule of thumb - if the times are orders of milliseconds, use "`millis()`" to work with integers instead of floating numbers.
5. To use `Serial.print()` you need to set `BAUD_RATE = 9600` (the speed of communication over the data channel ~ bits per second), used in `Serial.begin(BAUD_RATE)`. **There is no debugger in the Arduino IDE - use prints instead.** Knowing how to debug using only print commands is, possibly, an essential skill required for a Software-Phy Engineer.
6. It's highly recommended to implement your code for `BIT_TIME` = 1 sec and then change it to 1/50 when your code works.
7. Instead of shifts and bitwise operations, you can use "`bitRead(buffer, i)`" or "`bitWrite(buffer, i, HIGH/LOW)`".
8. Arduino has a built-in random number generator using the "`random(min, max)`" function.
9. The easy and organized implementation uses a switch-case statement. We recommend using **defines** (for numeral parameters) and **enums** (state names). *Google is your friend when it comes to proper coding techniques.*
10. The program should transmit and receive bytes without limitations. For example, changing the number of bits sent should be done quickly (using define).
11. Your work should work in loopback (Tx connected to the Rx of the same board) **and** with another board (*perhaps one not running your code*).
12. Tx and Rx **do not** share variables. Using global or static variables is a good choice.
13. Arduino IDE's compiler tends to be rather weak; it's a good practice to divide calculations. E.g. "`a=(a++)%2;`" sometimes doesn't run, unlike "`a++; a=a%2;`".
14. **Your code should be well documented!**

# Tools That May Help:

Simulator (login with BGU via Google account):
https://www.tinkercad.com

Arduino Reference:
https://www.arduino.cc/reference/en/