

Arduino Lab

Session 09: SPI

Instructor: Dr. Mohsen Raji

Authors:

- Hossein Dehghanipour
- Mohammad Hossein Allah Akbari

Shiraz University – Spring 2021



Brief Introduction

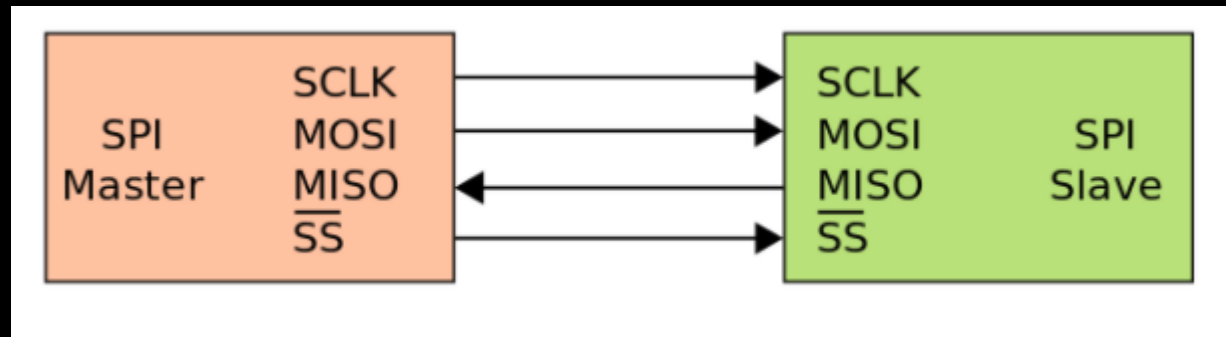
- **Serial Peripheral Interface (SPI)** is a **synchronous serial** data protocol used by microcontrollers for communicating with one or more peripheral devices quickly over **short distances (why?)**. It can also be used for communication between two microcontrollers.
 - With an SPI connection there is **always** one master device (usually a microcontroller) which controls the peripheral devices.
 - Typically there are three lines common to all the devices:
 - **MISO (Master In Slave Out)** - The Slave line for sending data to the master,
 - **MOSI (Master Out Slave In)** - The Master line for sending data to the peripherals,
 - **SCK (Serial Clock)** - The clock pulses which synchronize data transmission generated by the master
 - and one line specific for every device:
 - **SS (Slave Select)** - the pin on each device that the master can use to enable and disable specific devices.
 - When a device's Slave Select pin is **low**, it **communicates** with the master. When it's **high**, it **ignores** the master. This allows you to have **multiple SPI devices** sharing the same MISO, MOSI, and CLK lines.
-

Extended Introduction



Typical applications include [Secure Digital](#) cards and [liquid crystal displays](#).

SPI devices communicate in [full duplex](#) mode using a [master-slave](#) architecture with a single master. The master device originates the [frame](#) for reading and writing. Multiple slave-devices are supported through selection with individual [slave select](#) (SS), sometimes called chip select (CS), lines.



Interface

- The SPI bus specifies four logic signals:
 - **SCLK**: Serial Clock (output from master)
 - **MOSI**: Master Out Slave In (data output from master)
 - **MISO**: Master In Slave Out (data output from slave)
 - **SS**: Slave Select (often active low, output from master)

MOSI on a master connects to **MOSI** on a slave.

MISO on a master connects to **MISO** on a slave.

Slave Select has the same functionality as chip select and is used instead of an addressing concept.

Note: on a slave-only device, MOSI may be labeled as **SDI (Slave Data In)** and MISO may be labeled as **SDO (Slave Data Out)**.

The signal names above can be used to label both the master and slave device pins as well as the signal lines between them in an unambiguous way, and are the most common in modern products. Pin names are always capitalized e.g. "**Slave Select**," not "~~slave-select~~."

Operation

The SPI bus can operate with a single master device and with one or more slave devices.

If a single slave device is used, the SS pin *may* be fixed to [logic low](#) if the slave [permits](#) it. Some slaves require a falling [edge](#) of the chip select signal to initiate an action. An example is the [Maxim](#) MAX1242 [ADC](#), which starts conversion on a high→low transition. **With multiple slave devices, an independent SS signal is required from the master for each slave device.**

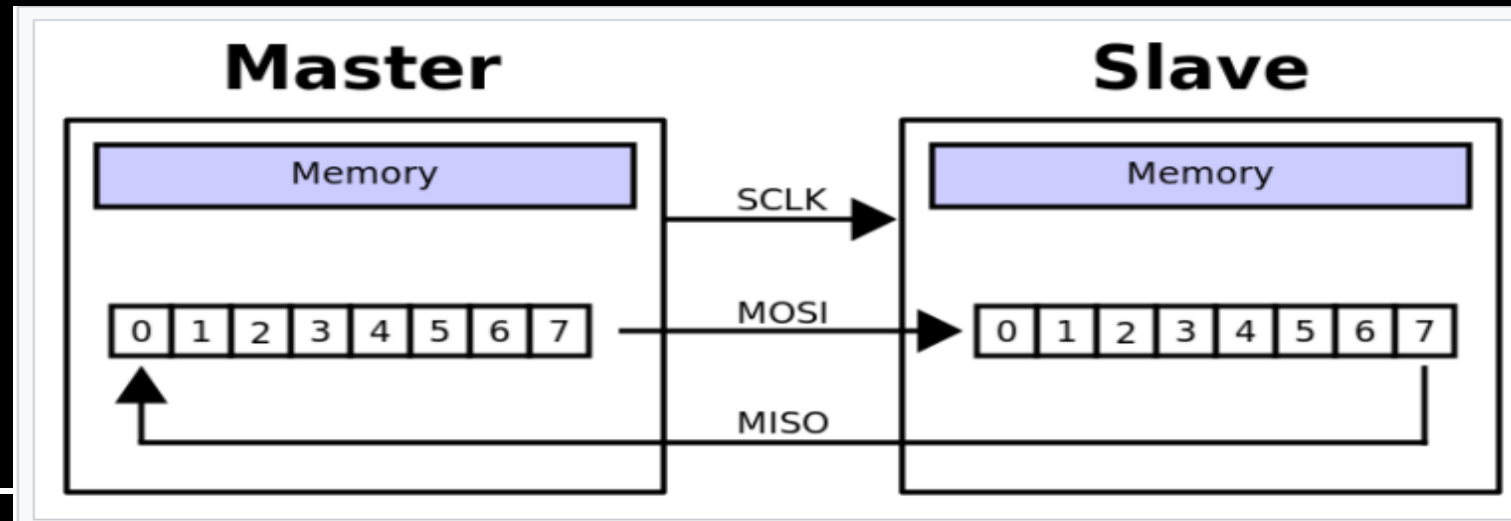
Most slave devices have [tri-state outputs](#) so their MISO signal becomes [high impedance](#) (*electrically disconnected*) when the device is not selected. Devices **without** *tri-state outputs* **cannot** share SPI bus segments with other devices without using an external *tri-state buffer*.



Data Transmission

During each SPI clock cycle, a full-duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only one-directional data transfer is intended.

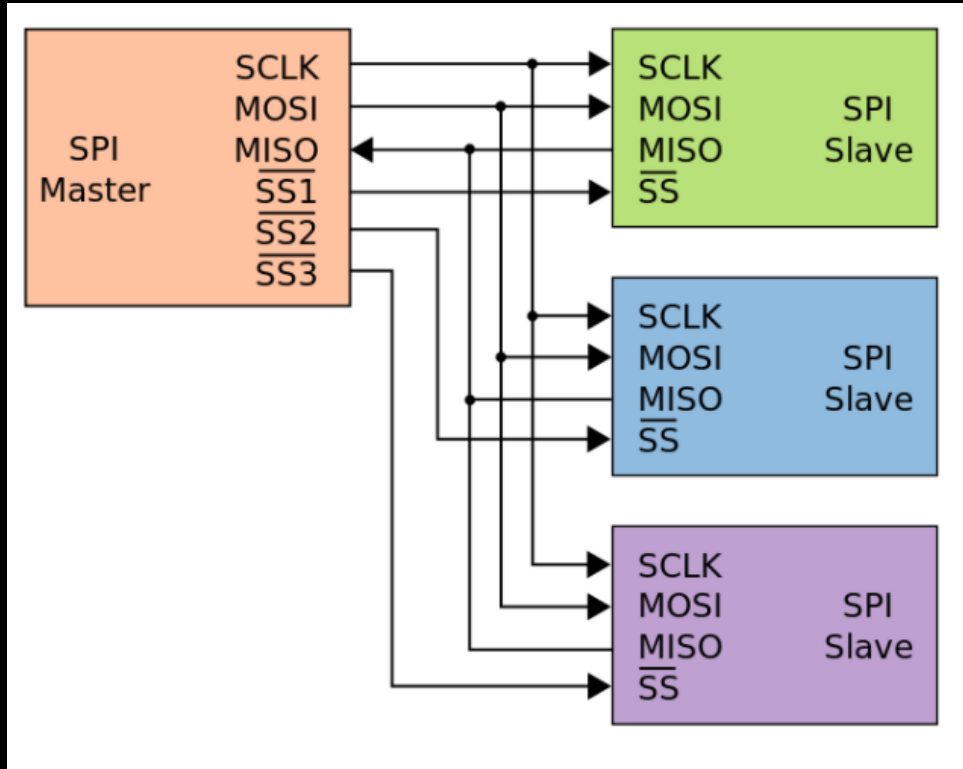
Transmissions normally involve **two shift registers** of some given **word-size**, such as **eight bits**, one in the master and one in the slave; they are connected in a virtual ring topology. Data is usually shifted out with the **most significant bit** first (can be changed in the Arduino). On the clock edge, both master and slave shift out a bit and output it on the transmission line to the counterpart. On the next clock edge, at each receiver the bit is sampled from the transmission line and set as a new least-significant bit of the shift register. After the register bits have been shifted out and in, the master and slave have exchanged register values. If more data needs to be exchanged, the shift registers are reloaded and the process repeats. Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.



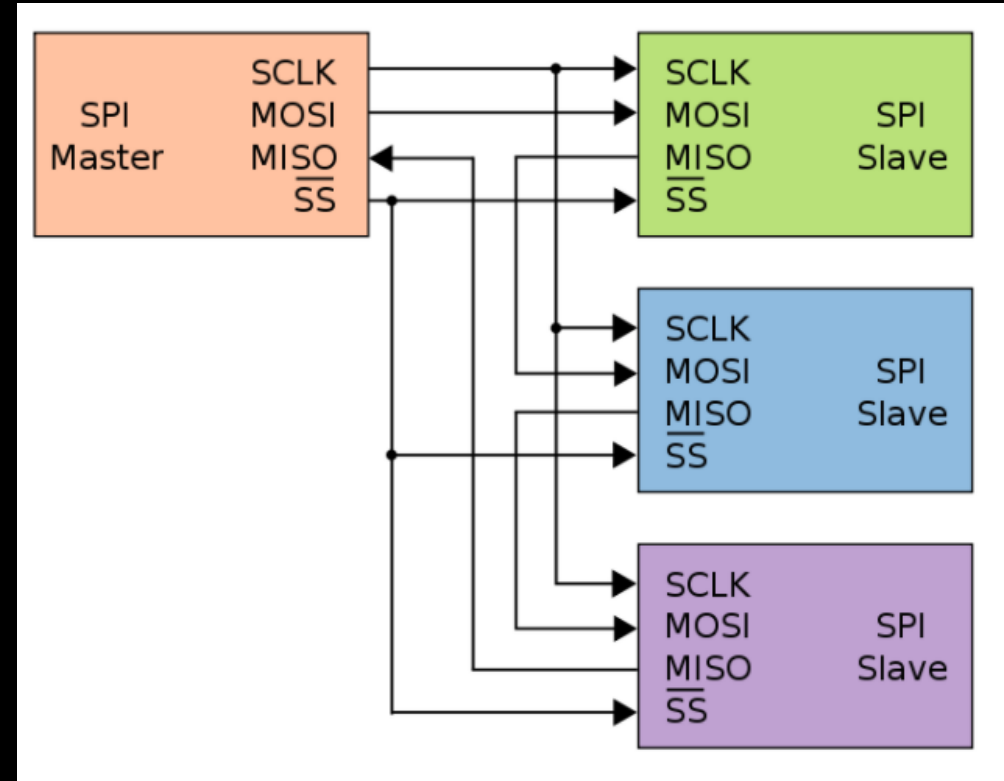
Slave Configurations



Independent Configuration



Daisy Chain Configuration



Can you see the difference?

You thought that we were done with the introduction????

Nope. You are wrong



Pros and Cons



Advantages

- Full duplex communication in the default version of this protocol
- Higher [throughput](#) than [I²C](#) or [SMBus](#). Not limited to any maximum clock speed, enabling potentially high speed.
- Complete protocol flexibility for the bits transferred
 - Not limited to 8-bit words
 - Arbitrary choice of message size, content, and purpose
- Extremely simple hardware interfacing
 - Typically lower power requirements than [I²C](#) or SMBus due to less circuitry (including pull up resistors)
 - No arbitration or associated failure modes - unlike [CAN-bus](#)
 - Slaves use the master's clock and do not need precision oscillators
 - Slaves do not need a unique [address](#) – unlike [I²C](#) or [GPIB](#) or [SCSI](#)
 - Transceivers are not needed - unlike [CAN-bus](#)
- Uses only four pins on IC packages, and wires in board layouts or connectors, much fewer than parallel interfaces
- At most one unique bus signal per device (chip select); all others are shared
- Signals are unidirectional allowing for easy [galvanic isolation](#)
- Simple software implementation

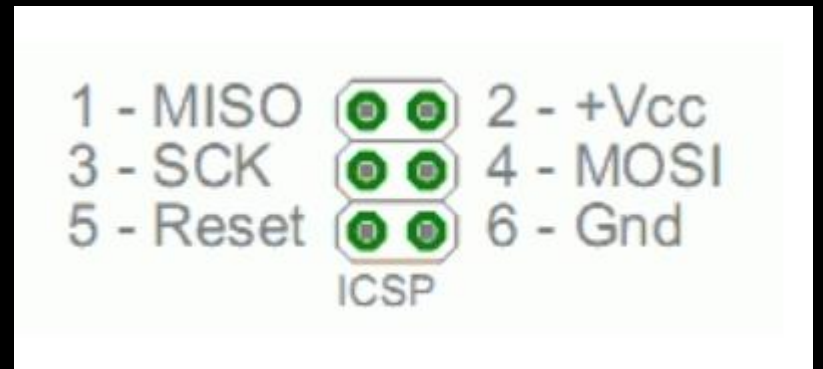
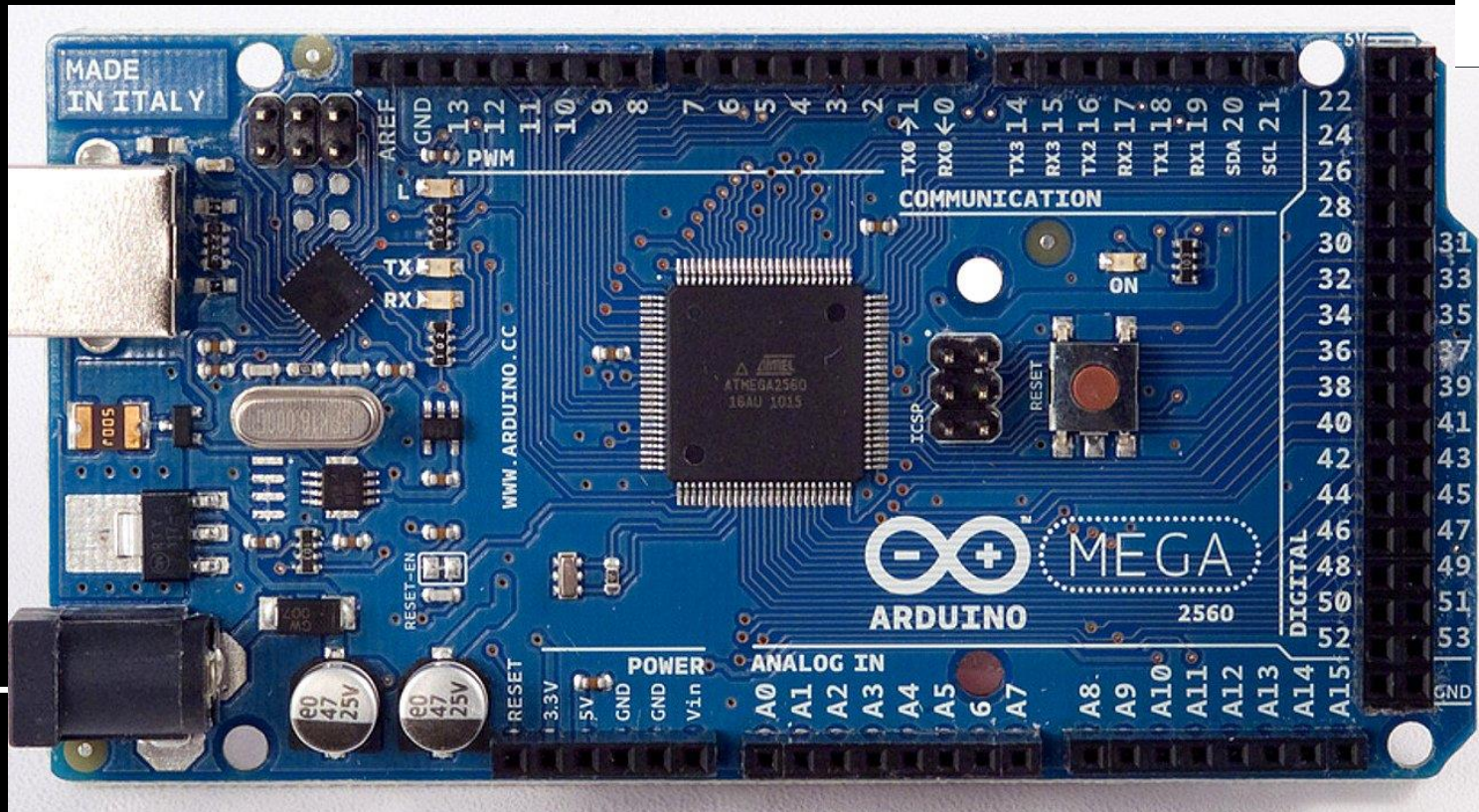
DisAdvantages

- Requires more pins on IC packages than [I²C](#).
 - No in-band addressing; out-of-band chip select signals are required on shared buses
 - Extensibility severely reduced when multiple slaves using different SPI Modes are required. Access is slowed down when master frequently needs to reinitialize in different modes.
 - No hardware flow control by the slave (but the master can delay the next clock edge to slow the transfer rate)
 - No hardware slave acknowledgment (the master could be transmitting to nowhere and not know it)
 - Typically supports only one master device (depends on device's hardware implementation)
 - No error-checking protocol is defined
 - Without a formal standard, validating conformance is not possible
 - Only handles short distances compared to [RS-232](#), [RS-485](#), or [CAN-bus](#). (Its distance can be extended with the use of transceivers like [RS-422](#).)
 - Many existing variations, making it difficult to find development tools like host adapters that support those variations
 - SPI does not support [hot swapping](#) (dynamically adding nodes).
 - Some variants like [dual SPI](#), [quad SPI](#) are half-duplex.
-

SPI in Arduino



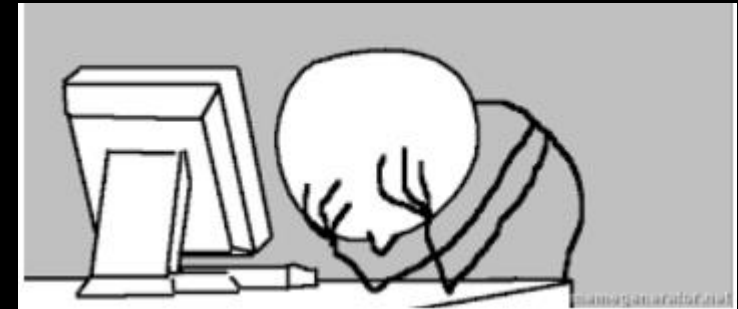
Arduino / Genuino Board	MOSI	MISO	SCK	SS (slave)	SS (master)	Level
Uno or Duemilanove	11 or ICSP- 4	12 or ICSP- 1	13 or ICSP- 3	10	-	5V
Mega1280 or Mega2560	51 or ICSP- 4	50 or ICSP- 1	52 or ICSP- 3	53	-	5V



To write code for a new SPI device you need to note a few things:

- What is the maximum SPI speed your device can use? This is controlled by the **first parameter** in **SPISettings**. If you are using a chip rated at 15 MHz, use 15000000. Arduino will automatically use the best speed that is equal to or less than the number you use with SPISettings.
- Is data shifted in Most Significant Bit (MSB) or Least Significant Bit (LSB) first? This is controlled by **second SPISettings parameter**, either MSBFIRST or LSBFIRST. Most SPI chips use MSB first data order.
- Is the data clock idle when high or low? Are samples on the rising or falling edge of clock pulses? These modes are controlled by the **third parameter** in **SPISettings**.

The SPI standard is loose and each device implements it a little differently. This means you have to pay special attention to the device's datasheet when writing your code



Modes of Transmission

Generally speaking, there are **four** modes of transmission. These modes control whether data is **shifted in and out** on the **rising or falling edge** of the data clock signal (called the clock phase), and whether the **clock is idle** when **high or low** (called the clock polarity). The four modes combine polarity and phase according to this table:

Mode	Clock Polarity (CPOL)	Clock Phase (CPHA)	Output Edge	Data Capture
SPI_MODE0	0	0	Falling	Rising
SPI_MODE1	0	1	Rising	Falling
SPI_MODE2	1	0	Rising	Falling
SPI_MODE3	1	1	Falling	Rising

Functions: `SPI.beginTransaction()`

Once you have your SPI parameters, use `SPI.beginTransaction()` to begin using the SPI port.

The SPI port will be configured with your all of your settings. The simplest and most efficient way to use `SPISettings` is directly inside `SPI.beginTransaction()`. For example:

```
SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0));
```



Functions: SPI.beginTransaction()

When we are done working with SPI, we use SPI.beginTransaction() to tell the microprocessor that our work with this unit is done.

- If other libraries use SPI from interrupts, they will be prevented from accessing SPI until you call SPI.beginTransaction().
 - The SPI settings are applied at the begin of the transaction and SPI.beginTransaction() doesn't change SPI settings. Unless you, or some library, calls beginTransaction a second time, the setting are maintained.
 - You should attempt to minimize the time between before you call SPI.beginTransaction(), for best compatibility if your program is used together with other libraries which use SPI.
-

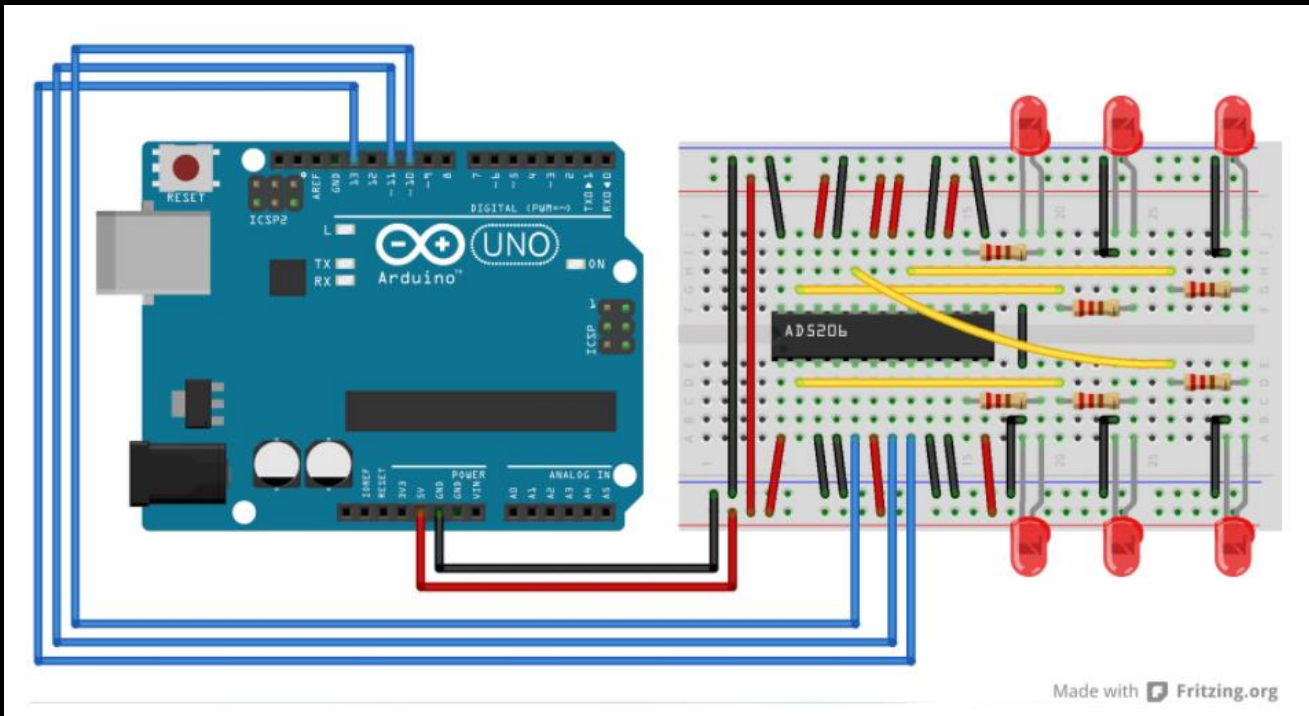
Functions: `SPI.transfer()`

With most SPI devices, after `SPI.beginTransaction()`, you will write the slave select pin LOW, call `SPI.transfer()` any number of times to transfer data, then write the SS pin HIGH, and finally call `SPI.endTransaction()`.

Note: Don't forget to include SPI Library: `#include <SPI.h>`

There is something odd about this code. Can you find it?

Hint: where is `SPI.end()` ???



```
#include <Arduino.h>
#include <SPI.h>

const int slaveSelectPin = 10;
void setup() {

    // initialize SPI:
    SPI.begin();
}

void loop() {

    for (int channel = 0; channel < 6; channel++) {

        for (int level = 0; level < 255; level++) {

            digitalWrite(slaveSelectPin, LOW);
            delay(100);
            SPI.transfer(channel);
            SPI.transfer(level);
            delay(100);
            digitalWrite(slaveSelectPin, HIGH);
            delay(10);

        }

    }

}
```

Note about Slave Select (SS) pin on AVR based boards

All AVR based boards have an SS pin that is useful when they act as a slave controlled by an external master. Since this library supports only master mode, this pin should be set always as OUTPUT otherwise the SPI interface could be put automatically into slave mode by hardware, rendering the library inoperative.

It is, however, possible to use any pin as the Slave Select (SS) for the devices. For example, the Arduino Ethernet shield uses pin 4 to control the SPI connection to the on-board SD card, and pin 10 to control the connection to the Ethernet controller.

Please Visit the Following Links:

[SPISettings](#)

[begin\(\)](#)

[end\(\)](#)

[beginTransaction\(\)](#)

[endTransaction\(\)](#)

[setBitOrder\(\)](#)

[setClockDivider\(\)](#)

[setDataMode\(\)](#)

[transfer\(\)](#)

[usingInterrupt\(\)](#)

[Due Extended SPI usage](#)

[shiftOut\(\)](#)

[shiftIn\(\)](#)



After all this time, if you still have any questions and you don't know what to do, go ahead;
ask your TAs.



I don't care anymore.



Good Luck ☺

