Documented Tasks

// SHOULD I EXPLAIN WHAT SYNCHRONOUS AND ASSYNCHRONOUS COMMUNICATION IS?

// SHOULD I EXPLAIN WHAT A MASTER AND SLAVE CONFIG IS? FEEL LIKE ITS UNECESSARY

// SHOULD I EXPLAIN THE ADVANTAGES AND DISADVANTAGES OF SPI

**Development Environment for ESP32 and read manual**

I’m currently reading the manual as I go and complete tasks on my project.

**Test ESP32 W/ some simple code**

* Imported hello\_world from Expressif examples and learnt how to flash micro and monitor serial terminal;
* Imported blink project from Expressif examples – found where LED is in the pins;
* Created a project that measures the internal temperature of the micro – code from YouTube;
* Created a project that takes measures from Hall effect sensors of the micro – code from YouTube;
* Created a project that simulates a traffic light with 3 leds: green, yellow and red.
* Created a standard project and made 2 tasks turn on and off the LED22;

**Search how to make libraries in ESP-IDF and search about Project Layering**

Given that the build manager toolchain is Cmake, the way the layering is done in Expressif IDE is using components you create/call from their API/import from their IDF Component Repository. Components, as far as I’ve understood, are meant to be thought as libraries but with additional features – pretty much libs with additional Cmake wrapper commands. As such, after investigating multiple components from their repo and comparing the file and folder structure of each one (they were varied), decided on layering my project with a look I’m more familiar with – something that came from SE subject project structuring.

**How to set up an IDE - for annex**

Even for a setup, it is easy to start with the IDE. I did it myself while looking at this [link](https://docs.espressif.com/projects/esp-idf/en/v5.1/esp32/get-started/windows-setup.html#get-started-windows-first-steps) (ESP-IDF Programming Guide) and at the IDF Documentation (after installing the IDE it is presented as an option to open this documentation. Initially, I copied and edited code that the IDE comes with (examples folder in the IDE installation directory) though their IDE, using their ESP-IDF CMD (tool that lets you execute build/config/monitor-related commands through a terminal interface) tool to set the target (esp32), build the project files, flash the micro and monitor its serial console. These are simple one-line commands that are on the programming guide link above.

After that, I explored the basic parts of the IDE interface until I could run the code on the micro through the IDE: setting the target through their interface, building and cleaning projects, and running code. **Debugging** is difficult here, so I researched the 3 debugging methods available to me as a user of the IDE: GDBStub Debugging, Core Dump Debugging (both are post-mortem) and OpenOCD Debugging. Since the LOLIN32 Lite also does not support JTAG debugging over USB, OpenOCD Debugging is out of the question as [I would need an external adaptor to be connected to certain pins in the board](https://www.espboards.dev/esp32/lolin32-lite/) and I don’t have one. As a result, I decided to stick with the basic GDBStub Debugging, that is pretty much the monitor command I’ve mentioned above.

**Check if I can change CPU frequency**

I found how to change the CPU frequency in a project and that is to access via ESP-IDF CMD terminal to its menuconfig (idf.py menuconfig) and change a setting in the sdkconfig file (Component config -> ESP System Settings - > CPU Frequency); or to access the file itself via project explorer on the project I’m on, and then find the same choices In the menu the other method had to change the settings of the file. I can change the CPU frequency to one of three values: 80, 160 or 240 MHz, the default being 160MHz.

**Think about architecture (queue with raw values - filtering - server task with filtered values)**

Following what the Embedded Systems has been teaching in class and using some references online of Expressif slideshow presentation on YouTube, I slowly came up with something that might be, in my opinion, viable as the architecture of the project.

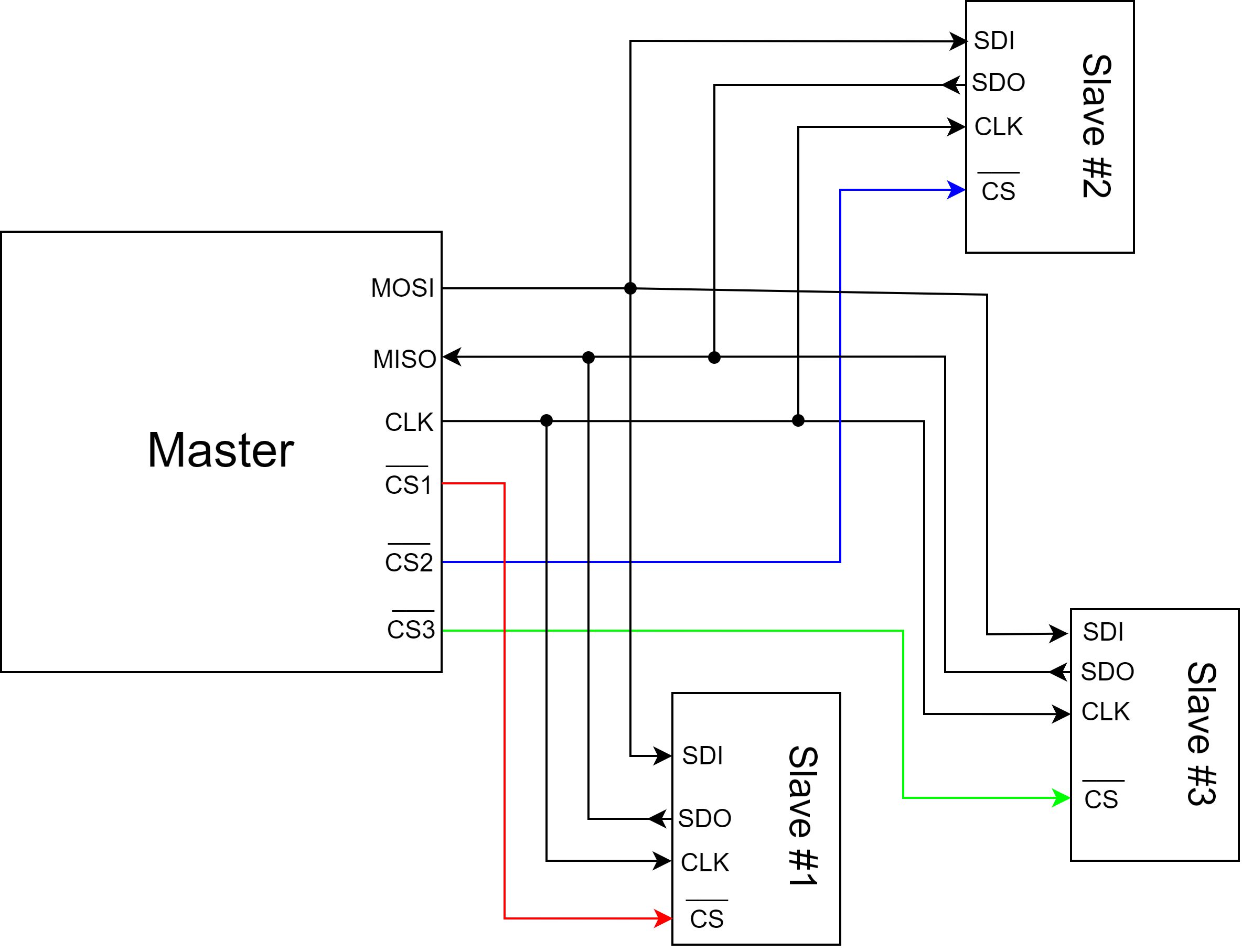
The project consists of a main task that creates the other tasks and starts the scheduler.

**Study SPI**

I used SPI in the MPU-9250\6500 communication instead of the I2C because not only is it faster (1MHz vS 400kHz) but also because I wanted to learn a new communication protocol.

SPI (Serial Peripheral Interface) is a protocol widely used between microcontrollers and peripherals such as sensors, ADCs e DACs. SPI communication is a synchronous type of communication protocol. Using a synchronous type of communication protocol means that all the devices that are communicating with each other have the same clock signal.

Usually, people use this communication protocol as it is very fast and supports very high communication speeds and high CPU clock frequencies as well. SPI communication is a full duplex communication protocol, having a master and slave configuration, something that I’m already familiar with since I learned about the I2C communication protocol in the Embedded Systems subject. In this configuration, two devices (using a slave and a master as an example, although there can be multiple slaves connected to one master) can transmit and receive data at the same time: SPI communication doesn’t rely only on one data bus.



At the master side, there’s the MOSI pin (Master Out Slave In), the MISO pin (Master In Slave Out), the CLK signal pin (clock signal) and the CS pin (Chip Select).

In communication, the MOSI line has the data that’s being transmitted from the Master to the Slave (hence Master Out Slave In) and the MISO line has the data that’s being sent by the Slave and the Master receives it (hence Master In Slave Out). The clock signal is very important for synchronization: when the master initiates the clock signal, and all slaves receive the signal, they send the data as per clock signal. The CS line is dedicated for each existent slave connected to the master, so if there are 3 slaves to a master, then there are 3 Chip Select lines connected to each one of the slaves one by one. This can be seen as the main limitation in SPI communication, since the number of GPIOs restrict the maximum number of slaves. The CS signal is active low – generally it is pulled high (which is at VCC level) if the master and slave are not in communication with each other. When the master needs to communicate with the slave, it will put the signal to ground, notifying the receiver that it wants to send data.

Unlike what happens in UART and I2C communication, in the SPI protocol there’s no specific bit frame for the SPI communication, simplifying the process of sending and receiving data.

There are 4 modes in the SPI that give flexibility to program data in four different clock mode combinations. The modes depend on the CLK signal polarity and on the CLK signal phase, as you can see in the following table.

|  |  |  |
| --- | --- | --- |
|  | CLK signal polarity | CLK signal phase |
| 1st Mode | 0 | 0 |
| 2nd Mode | 0 | 1 |
| 3rd Mode | 1 | 0 |
| 4th Mode | 1 | 1 |

In the **first mode**, there is no delay for clock pulses (starts from the low state) and the data is output immediately on the rising edge of the SPI CLK signal – input data is latched on the falling edge of the clock.

A diagram of a clock polarity

AI-generated content may be incorrect.

A diagram of a clock polarity

AI-generated content may be incorrect.In the **second mode**, the SPI CLK starts after some delay (starts from the low state) and the data is output one half cycle before the first rising edge of the SPI CLK. The input data is latched onto the rising edge of the SPI CLK and later output on the subsequent falling edges of the SPI CLK.

In the **third mode**, there is no delay for clock pulses, but it starts from the high state and the data is output immediately on the falling edge of the SPI CLK signal – input data is latched on the rising edge of the clock.

A diagram of a clock polarity

AI-generated content may be incorrect.

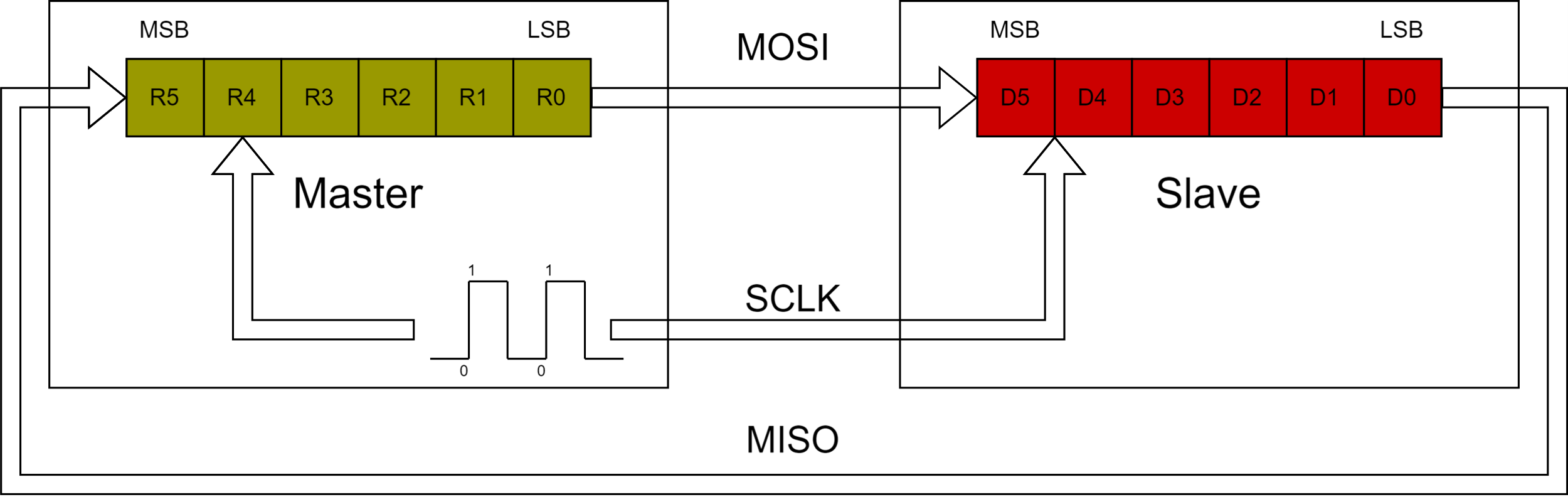
In the **fourth mode**, the SPI CLK starts after some delay (starts from the high state) and data is output one half cycle before the first falling edge of the SPI CLK – input data is latched on the falling edge of the SPI CLK but later output on the subsequent rising edges of the SPI CLK.

A diagram of a clock polarity

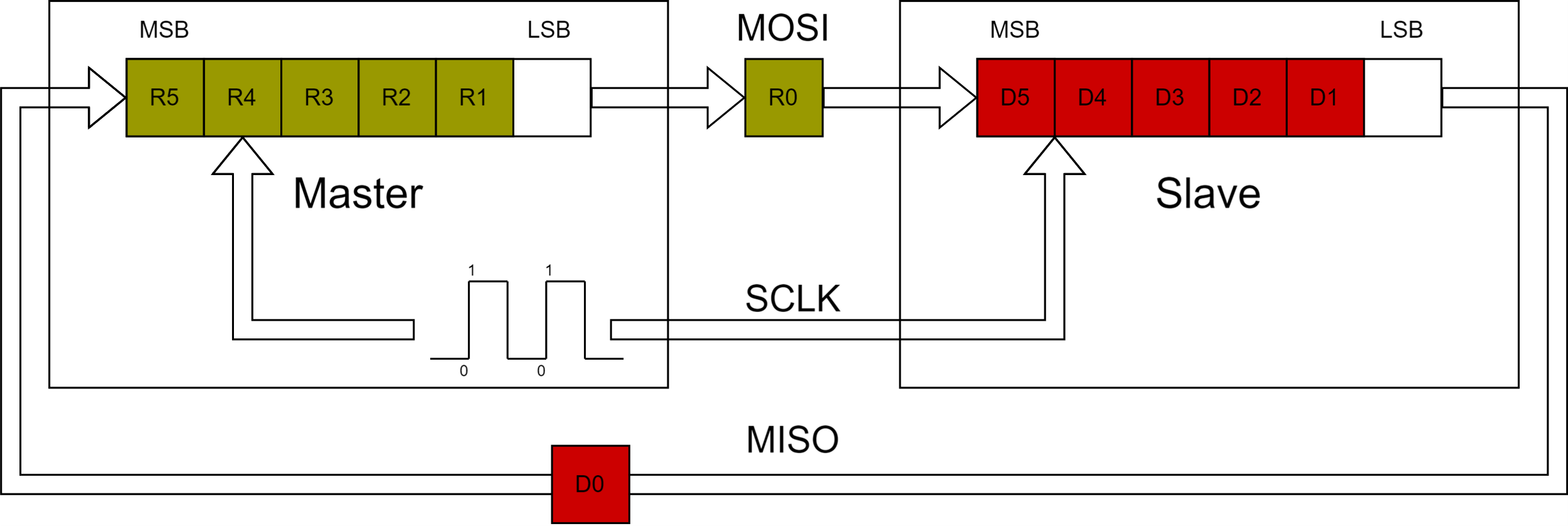
AI-generated content may be incorrect.

This allows SPI communication to interface with different types of serial devices.

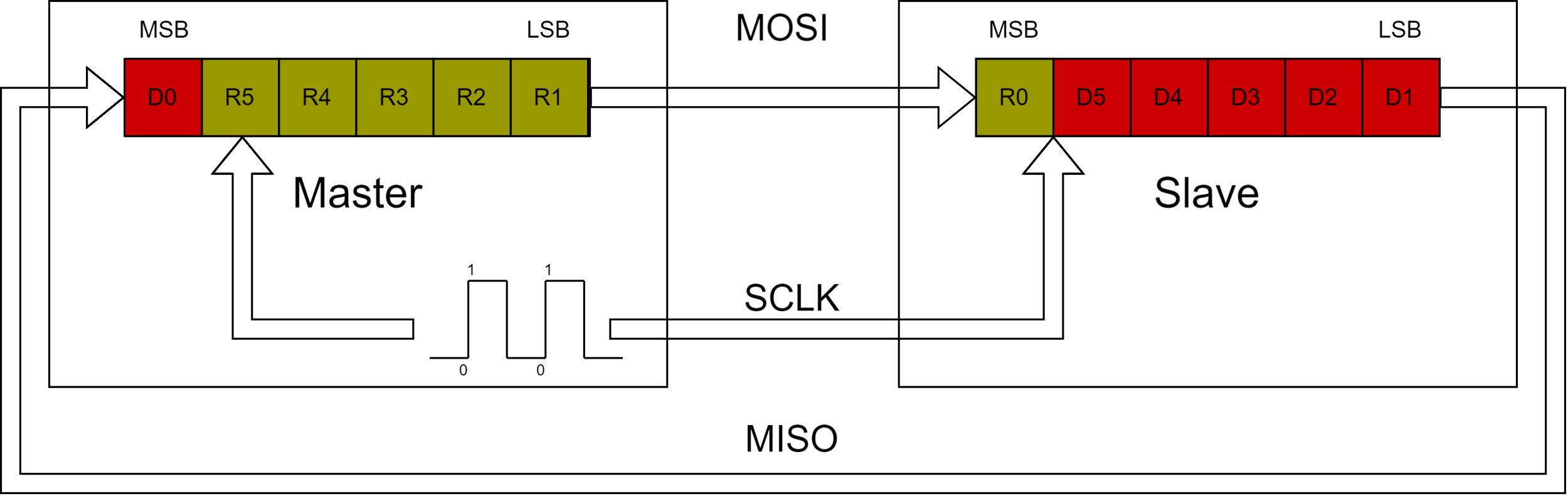
Looking at the hardware and internal blocks of a master and a slave, one can see that there are shift registers inside both. The master and slave are connected in such a way that the two shift registers form an inter-device circular buffer. These shift registers operate in SISO (Serial In Serial Out) mode, meaning that the output of the master’s shift register is connected to the input of the slave’s shift register and the output of the slave’s shift register is connected to input of the master’s shift register: these connections operate in a loop.

The connection between the master’s shift register’s output and the slave’s shift register’s input is called the MOSI line and the connection between the slave’s shift register’s output and the master’s shift register’s input is called the MISO line. The clock is generated by the master so that the master gets control of the communication: this is a synchronous communication protocol and so, there will be a clock signal in both master and slave for synchronization.

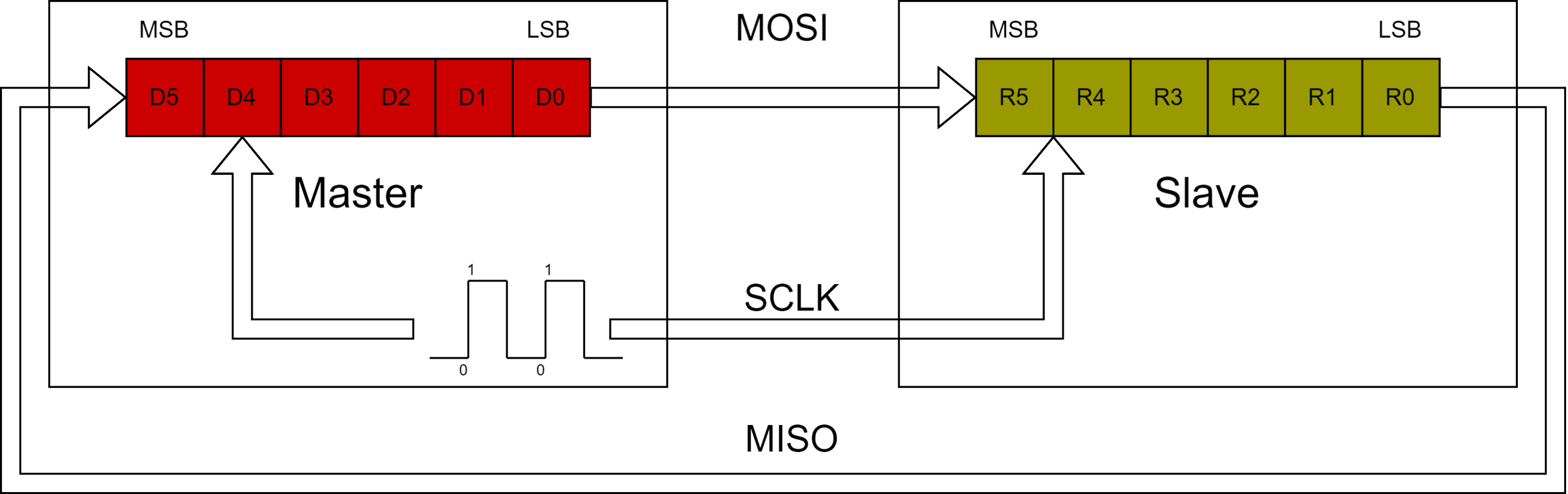
The master initiates the communication by generating a clock pulse, and as soon as that clock pulse starts, shift registers from both devices shift their first bit (LSB), that in this case is master shifting R0 and slave shifting D0, both to the right.



As the two devices are connected in a ring, the ejected LSB from the master (R0) is stored in the MSB position of the slave shift register and the ejected LSB from the slave (D0) is stored in the MSB position of the master shift register.



By repeating this process is how each bit travels over the MOSI and MISO lines at the same time in the SPI communication protocol.



**Study UART**

**NMEA frames check**

**Map communication protocol in micro: make document about protocols and read about them in ESP**

**Compare 3 different modules to explain the protocols**

**Tabela comparativa com os parâmetros que me interessam para decidir no melhor**

**Make communication protocol code (to do)**

**Make display font**