ESP32-WROOM-32E module and ESP32-DevKitC kit - Part 2

Arquiteturas para Sistemas Embutidos

João Gameiro, 93097
Pedro Abreu, 93240

Turma TP1
Grupo 3

29 Março 2022

universidade
de aveiro

Development Framework

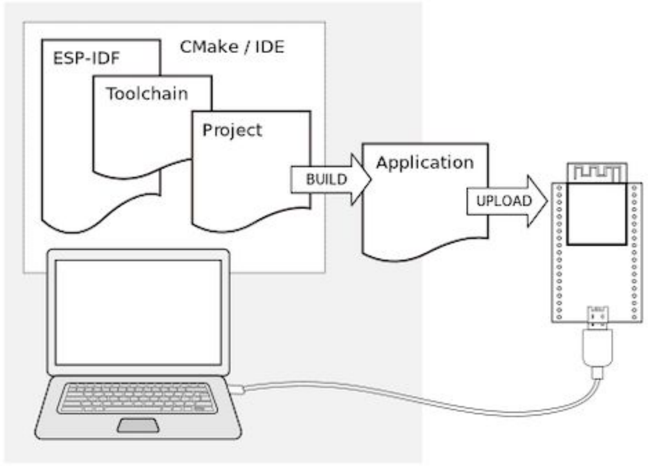
- ESP-IDF (**Espressif IoT Development Framework)** is Espressif's official IoT Development Framework for the ESP32, ESP32-S and ESP32-C series of SoCs.
- It provides a self-sufficient SDK for any generic application development on these platforms, using programming languages such as C and C++.
- ESP-IDF currently powers millions of devices in the field, and enables building a variety of network-connected products, ranging from simple light bulbs and toys to big appliances and industrial devices.
- The ESP-IDF is a collection of different things. We have the development framework which provides a comprehensive Software Development Kit, and it also includes the toolchain, documentation, example codes, and a set of tools.

Development Framework

ESP-IDF requires a set of tools which include:

- Cross Compiler toolchains;
- CMake and Ninja to build the applications;
- ESP-IDF (itself) that contains libraries and frameworks for ESP32 and scripts to operate with the toolchain.

The installation can be made one of two different ways, through extensions and plugins in IDEs or manually.



Standard toolchain for Linux (manual installation)

- bison (parser) and flex (fast lexical analyzer generator)
- wget for retrieving content from web servers
- python3, python3-pip, python3-setuptools
- cmake and ninja for building the project
- dfu-util to download and upload firmware to/from devices connected over USB
- libusb-1.0-0 that provides generic access to USB devices
- Ccache that caches the output of C/C++ compilation
- git, libffi-dev libssl-dev

- It is necessary to get the ESP-IDF from the espressif repository in github

IDE - Eclipse Plugin

- Brings developers an easy-to-use Eclipse-based development environment for developing ESP32 based IoT applications.
- It provides better tooling capabilities, which simplifies and enhances standard Eclipse CDT for developing and debugging ESP32 IoT applications.
- It offers advanced editing, compiling, flashing and debugging features with the addition of Installing the tools, SDK configuration and CMake editors
- The eclipse runs in Windows, Linux or macOS
- To use the extensions is necessary to have Java >= 11, Python >= 3.6, Eclipse IDE for C/C++ and Git

IDE - VSCode extension

There is also support for ESP32 development in Visual Studio Code to provide complete end-to-end support actions related to ESP-IDF such has:

- building and flashing
- monitoring, debugging and tracing
- System trace viewer, core dump, ….

Installation of ESP-IDF and ESP-IDF Tools is being done from this extension itself

IDE - PlatformIO

- PlatformIO is a user-friendly and extensible IDE, with a set of professional development instruments, providing modern and powerful features to speed up yet simplify the creation and delivery of embedded products that can be integrated with VSCode or CLion.
- Is a cross-platform embedded development environment with out-of-the-box support for ESP-IDF.

- **PlatformIO Core** (CLI tool) is the heart of the entire PlatformIO ecosystem and contains:
  - Multi-Platform Build System, Development platform and package managers
  - Library management, Library Dependency Finder
  - Serial Port Monitor and Integration Components (for CI/CD)

Programming Languages

- C/C++
    - used in VSCode, Eclipse, CLion, ….
- Python
    - **MicroPython -** Python3 programming language that includes a small subset of the python standard library and is optimized to run on ESP micro-controllers.
    - **CircuitPython -** designed to simplify experimenting and learning to code on low-cost microcontrollers
- LUA
    - **LuaRTOS** - operating system based on the Lua language, which has been designed to run on embedded systems. Contains a LUA interpreter.
    - **NodeMCU -** Open source LUA based firmware. Programming model in LUA.
- Javascript
    - **Espruino -** Javscript interpreter designed for microcontrollers and devices with small amount of RAM.

ESP-IDF - Libraries and Frameworks

- Espressif Audio Development Framework
  - Comprehensive framework for audio applications
  - Music Players and Recorders, Audio Processing, Speech Recognition, Hands-free devices, etc
- Espressif DSP library
  - Provides algorithms optimized specifically for digital signal processing applications.
  - Supports FFT (Fast Fourier Transform), Vector math operations, Dot product, matrix multiplication, etc
- ESP CSI
  - Experimental implementation that uses the Wi-Fi Channel State Information to detect the presence of a human body.
- ESP IoT-Solution
  - Solution that contains frameworks and device drivers for IoT systems.
  - Contains device drivers for sensors, display, audio, actuators, etc

ESP-IDF - Libraries and Frameworks

- ESP WIFI-MESH Development Framework
  - Framework based on the ESP-WIFI-MESH protocol
  - Allows for fast network configuration, LAN control, Efficient debugging, etc
- ESP WHO
  - Face detection and recognition framework using the ESP32 and camera.
- ESP RainMaker
  - Solution for accelerated AIoT development
  - Creating AIoT devices from the firmware to the integration with voice-assistant, phone apps and cloud backend
- ESP Protocols
  - Repository that contains a collection of protocol components for ESP-IDF
  - Protocol that enables connectivity with GSM/LTE modems, …

ESP-IDF - Cloud Frameworks

ESP32 supports multiple cloud frameworks using agents built on top of ESP-IDF. All of these frameworks are open source and their code is available on Github.

- AWS IoT, Azure IoT, Google IoT Core
- Baidu IoT, Tecentyun IoT, Alyun IoT
- Tecent IoT, Joylink IoT

ESP-IDF - APIs

Some examples of APIs that allow for the programming of the ESP32 include:

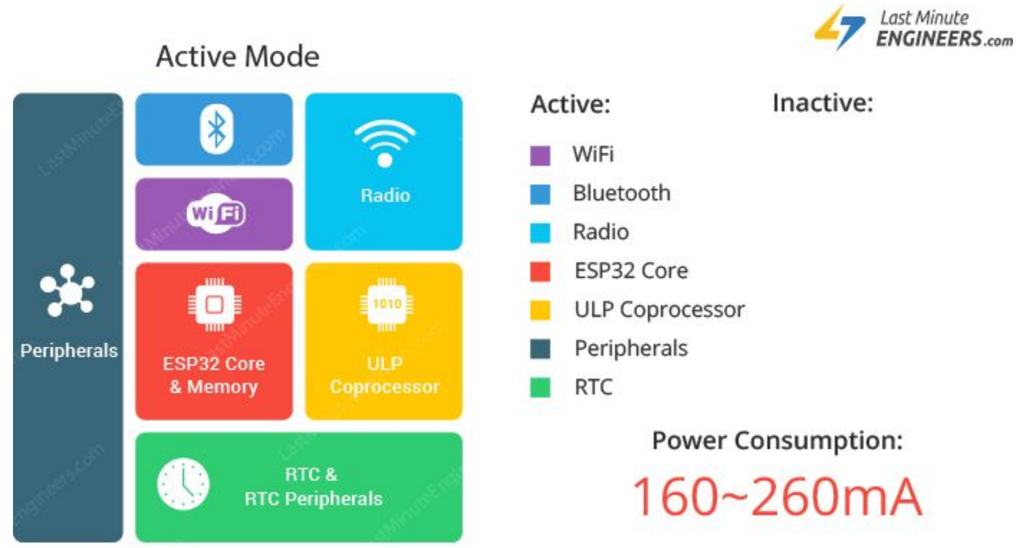
- **Bluetooth API**
  - Contains methods that allow for the programming and configuration of the bluetooth module
- **Networking APIs**
  - For configuration and utilization of networking modules
  - WiFi, Ethernet, Thread (IP based mesh networking), etc
- **Peripherals APIs**
  - For the configuration of the several ESP32 peripherals
  - SPI, I2C, I2S, ADC, DAC, etc
- **System API**
  - For aspects related with ESP32 system
  - Power Management, Sleep Modes, ULP Programming, Random Number Generation, Interrupt Allocation, etc
- **Storage API**
  - For reading and writing from flash memory and for dealing with several Fylesystems
  - SPI Flash API, FAT Fylesystem Support, SPIFFS Fylesystem support, etc

Power Management

- ESP32 offers efficient and flexible power-management technology to achieve the best balance between power consumption, wakeup latency and available wakeup sources.
- Users can select out of five predefined power modes of the main processors to suit specific needs of the application.
- To save power in power-sensitive applications, control may be executed by the Ultra-Low-Power coprocessor (ULP coprocessor), while the main processors are in Deep-sleep mode.

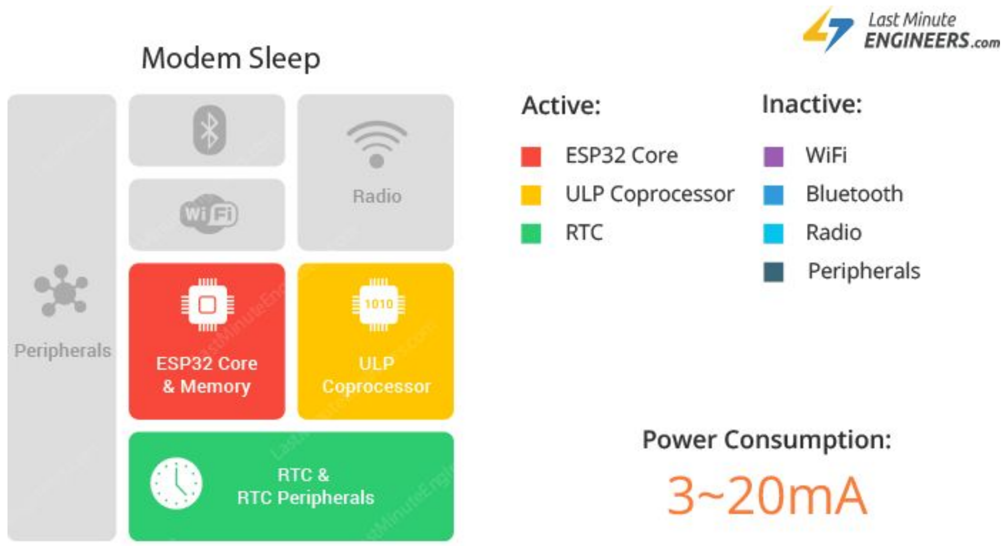
Power Management - Active Mode

- All the features of the chip are active.
- Active mode keeps everything (especially the WiFi module, the Processing Cores and the Bluetooth module) ON at all times, so the chip requires more than 240mA current to operate.
- Most inefficient mode and will drain the most current



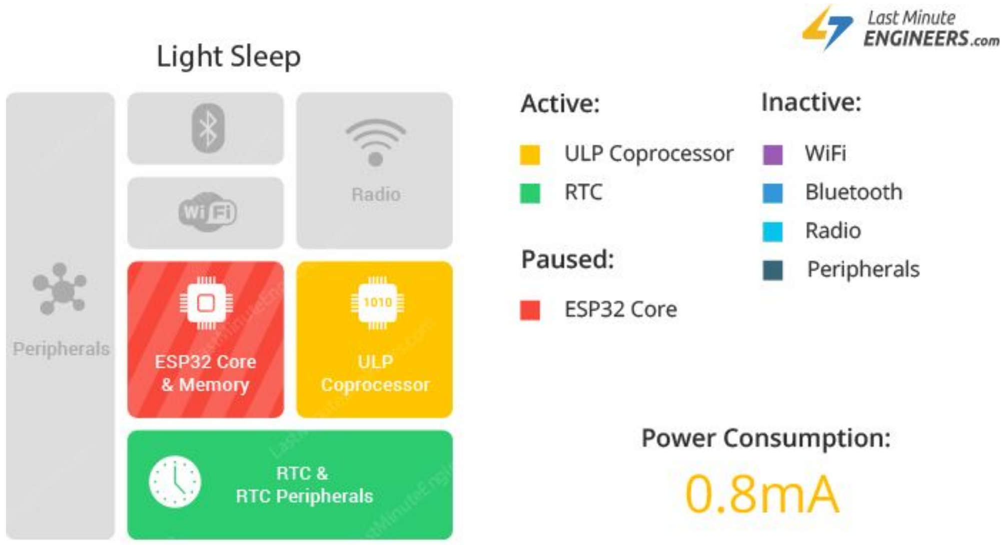
Power Management - Modem Sleep

- Everything is active while only WiFi, Bluetooth and radio are disabled. The CPU is also operational and the clock is configurable.
- To keep WiFi/Bluetooth connections alive, the Wi-Fi, Bluetooth, and radio are woken up at predefined intervals.
- The Wi-Fi/Bluetooth baseband is clock-gated or powered down. The radio is turned off.
- Immediate wake-up.



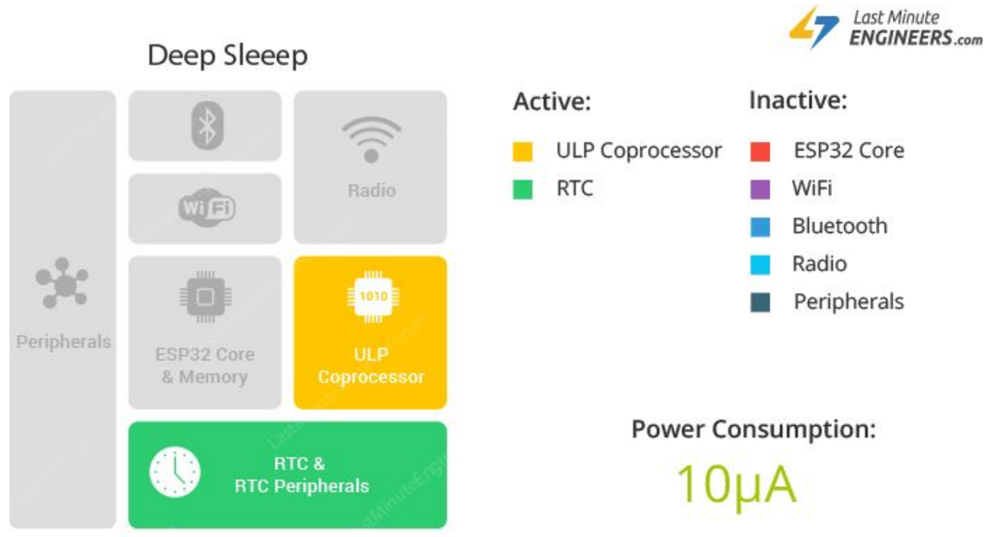
Power Management - Light Sleep

- Digital peripherals, most of the RAM and CPU are clock-gated.
- The CPU is paused by powering off its clock pulses, while RTC and ULP-coprocessor are kept active.
- Before entering light sleep mode, ESP32 preserves its internal state and resumes operation upon exit from the sleep. It is known Full RAM Retention.
- Less than 1 ms to wake up.



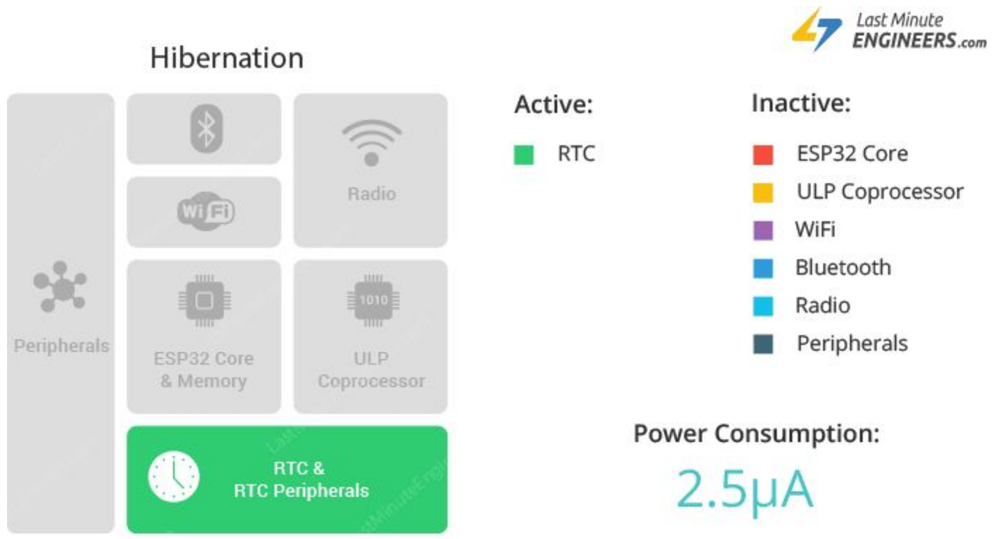
Power Management - Deep Sleep

- CPU is powered down, while the ULP co-processor does sensor measurements and wakes up the main system. The main memory of the chip is also disabled.
- RTC memory is kept powered on.
- Power is shut off to the entire chip except RTC module. So, any data that is not in the RTC recovery memory is lost, and the chip will thus restart with a reset.
- Less than 1 ms to wake up.



Power Management - Hibernation Mode

- The chip disables CPU and ULP-coprocessor. The RTC recovery memory is also powered down, meaning there’s no way we can preserve any data during hibernation mode.
- Everything is shut off except only one RTC timer on the slow clock and some RTC GPIOs are active. They are responsible for waking up the chip from the hibernation mode.
- Less than 1 ms to wake up.



Power Management - System API (Sleep Modes)

- Configuration of Wake Up Sources
  - `esp_sleep_enable_timer_wakeup()` function can be used enable deep sleep wakeup using a timer
  - `esp_sleep_enable_touchpad_wakeup()` function can be used to enable a touchpad wakeup source
  - `esp_sleep_enable_ulp_wakeup()` function can be used to enable this wakeup source
- Entering in sleep modes
  - `esp_light_sleep_start()` function can be used to enter light sleep mode
  - `esp_deep_sleep_start()` function can be used to enter deep sleep mode
- Checking wake up sources
  - `esp_sleep_get_wakeup_cause()` check which wakeup source has triggered wakeup from sleep mode
  - `esp_sleep_get_touchpad_wakeup_status()` to identify touchpad which has caused wakeup using
- Disabling sleep wake up sources
  - `esp_sleep_disable_wakeup_source()` to disable a previously wakeup source

Power Management - System API

- The System API also contains methods that can adjust advanced peripheral bus (APB) frequency or CPU frequency.
- Application components can express their requirements by creating and acquiring power management locks (change of cpu frequency, exiting from sleep mode, etc).
    - When an application acquires a lock, the power management algorithm operation is restricted
    - Ex.: a peripheral driver may need interruptions activated and request for the exit of the light sleep mode
- Power management can be enabled at compile time however it comes at cost of comes at the cost of increased interrupt latency

Applications

- Generic Low-power IoT Sensor Hub
- Generic Low-power IoT Data Loggers
- Cameras for Video Streaming
- Over-the-Top (OTT) Devices
- Speech Recognition
- Image Recognition
- Mesh Network

Applications

- Home Automation
  - Light Control, Smart Plugs, Smart Door Locks
- Smart Buildings
  - Smart Lighting, Energy monitoring
- Industrial Automation
  - Industrial wireless control and industrial robotics
- Smart agriculture
  - Smart greenhouses, smart irrigation, agriculture robotics
- Health Care applications
  - Health monitoring, baby monitors
- Retail & Catering Applications
  - Service robots, POS machines

Applications

- Audio Applications
  - Internet music players, Live streaming devices, Audio headsets
- WiFi enabled toys
  - Remote control toys, Proximity sensing toys, Educational toys
- Wearable Electronics
  - Smart watches, Smart bracelets

Remote Access

- ESP Local Control (**esp_local_ctrl**) component in ESP-IDF provides capability to control an ESP device over Wi-Fi + HTTPS or BLE.

- There is the possibility of creating lightweight HTTP(S) servers that run on the board and can be configured to be accessible from the network and add the necessary methods to control the kit
  - **esp_http_server** and **esp_https_server** allow for the creating and configuration of the server

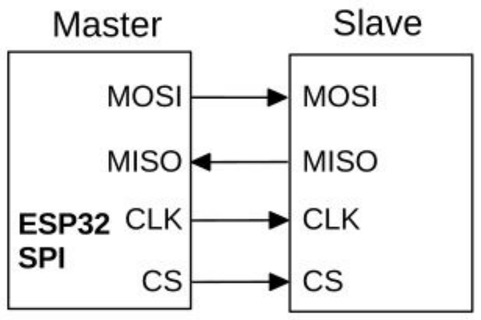
Firmware upgrades

- `esp_https_ota` provides simplified APIs to perform firmware upgrades over HTTPS. It’s an abstraction layer over existing **OTA** APIs.
- `esp_https_ota` function allocates HTTPS OTA Firmware upgrade context, establishes HTTPS connection, reads image data from HTTP stream and writes it to **OTA** partition and finishes HTTPS **OTA** Firmware upgrade operation.
  - This API is capable of dealing with the entire OTA upgrade process
  - If more control information regarding the operation is necessary there available other APIs to be called in succession to accomplish the upgrade
- It is also possible to perform OTA upgrades with pre-encrypted firmware by enabling an option to decrypt the data received before being processed.

Firmware upgrades

- The OTA update mechanism allows a device to update itself based on data received while the normal firmware is running (for example, over Wi-Fi or Bluetooth.)
- The OTA operation functions write a new app firmware image to whichever OTA app slot that is currently not selected for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot.

- It may be necessary to perform an application rollback to keep the device working after the update.
    - In case of successful update the application is marked with a valid state for that image
    - Otherwise is rollbacked to the previous versions and marked as invalid (not selectable for booting)



SPI - Master Driver

- SPI Master Driver is a program that controls ESP32s SPI peripherals while they function as masters. The SPI master driver governs communications of Hosts with Devices.
- SPI Transactions include
  - **Command** -> In this phase, a command (0-16 bit) is written to the bus by the Host.
  - **Address** -> In this phase, an address (0-64 bit) is transmitted over the bus by the Host.
  - **Write** -> Host sends data to a Device.
  - **Dummy** -> Configurable phase used for timing and synchronization purposes
  - **Read** -> Device sends data to Host
- The attributes of a transaction are determined by the bus configuration structure, device configuration structure and transaction configuration structure.

SPI - Master Driver

- An SPI Host can send full-duplex transactions, during which the read and write phases occur simultaneously.
- The total transaction length is determined by the sum of the following members:
    - **`spi_device_interface_config_t::command_bits`**
    - **`spi_device_interface_config_t::address_bits`**
    - **`spi_transaction_t::length`**
- In half-duplex transactions, the read and write phases are not simultaneous (one direction at a time).
    - The lengths of the write and read phases are determined by **`length`** and **`rxlength`** members of the struct **`spi_transaction_t`** respectively.
- The command and address phases are optional, as not every SPI device requires a command and/or address.
- The read and write phases can also be optional, as not every transaction requires both writing and reading data.

SPI - Master Driver

- During the command and address phases, the members `cmd` and `addr` in the struct `spi_transaction_t` are sent to the bus, nothing is read at this time.
- The data that needs to be transferred to or from a Device will be read from or written to a chunk of memory indicated by the members `rx_buffer` and `tx_buffer` of the structure `spi_transaction_t`.
  - if DMA is enabled for transfers the buffers need to be allocated in DMA internal memory
- When the transaction data size is equal to or less than 32 bits, it will be sub-optimal to allocate a buffer for the data.
  - For transmitted data, it can be achieved by using the `tx_data` member and setting the `SPI_TRANS_USE_TXDATA` flag on the transmission. For received data, use `rx_data` and set `SPI_TRANS_USE_RXDATA`.

SPI - Master Driver

- **Interrupt Transactions**
  - Interrupt transactions will block the transaction routine until the transaction completes, thus allowing the CPU to run other tasks.
- **Polling Transactions**
  - The routine keeps polling the SPI Host's status bit until the transaction is finished. Smaller transaction duration, but CPU is busy polling the Host.

- **Important Data Structures**
  - **`spi_transaction_t` -** describes one SPI transaction
  - **`spi_device_interface_config_t` -** configuration for a SPI slave that is connected to one of the buses
  - **`spi_transaction_ext_t` -** for SPI transactions which may change their address and command length.
  - **`spi_bus_config_t` -** configuration structure for a SPI bus

SPI - Slave Driver

SPI Slave driver is a program that controls ESP32’s SPI peripherals while they function as slaves. ESP32 integrates two general purpose SPI controllers which can be used as slave nodes (SPI2 and SPI3)

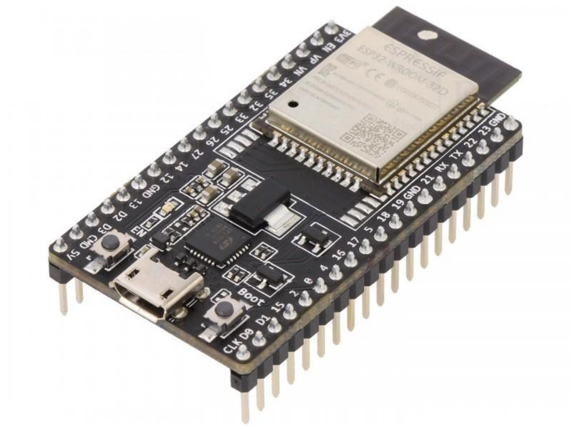
- A full-duplex SPI transaction begins when the Host asserts the CS line and starts sending out clock pulses on the SCLK line.
- Initialization of an SPI peripheral as slave is done by making a call to **`spi_slave_initialize`** with it’s specific configurations

SPI - Slave Driver

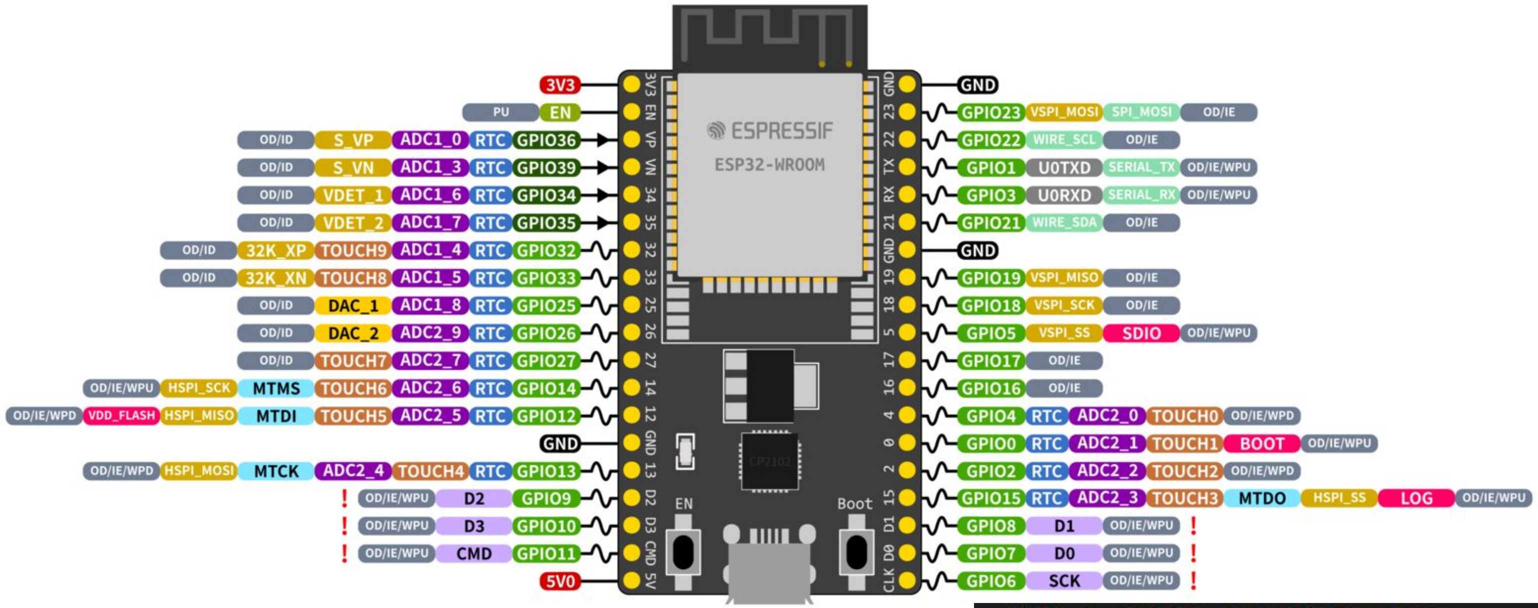
- Like in Master driver data is written and read from memory portions indicated by the members **`rx_buffer`** and **`tx_buffer`** of the structure **`spi_transaction_t`**.
- The amount of data that the driver driver can read or write to the buffers is limited by **`spi_transaction_t::length`**

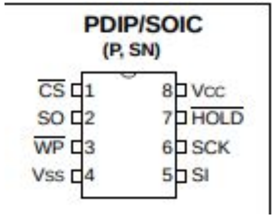
- **Important Data Structures**
  - **`spi_slave_interface_config_t -`** Configuration for a SPI host acting as a slave device.
  - **`spi_slave_transaction_t -`** describes one SPI transaction

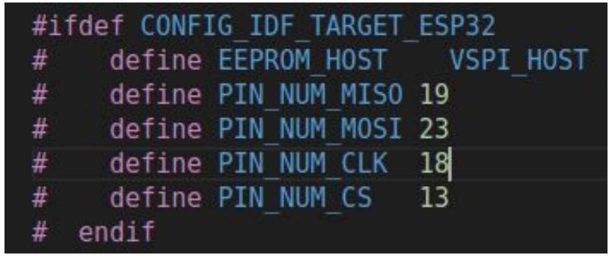
Demonstração SPI

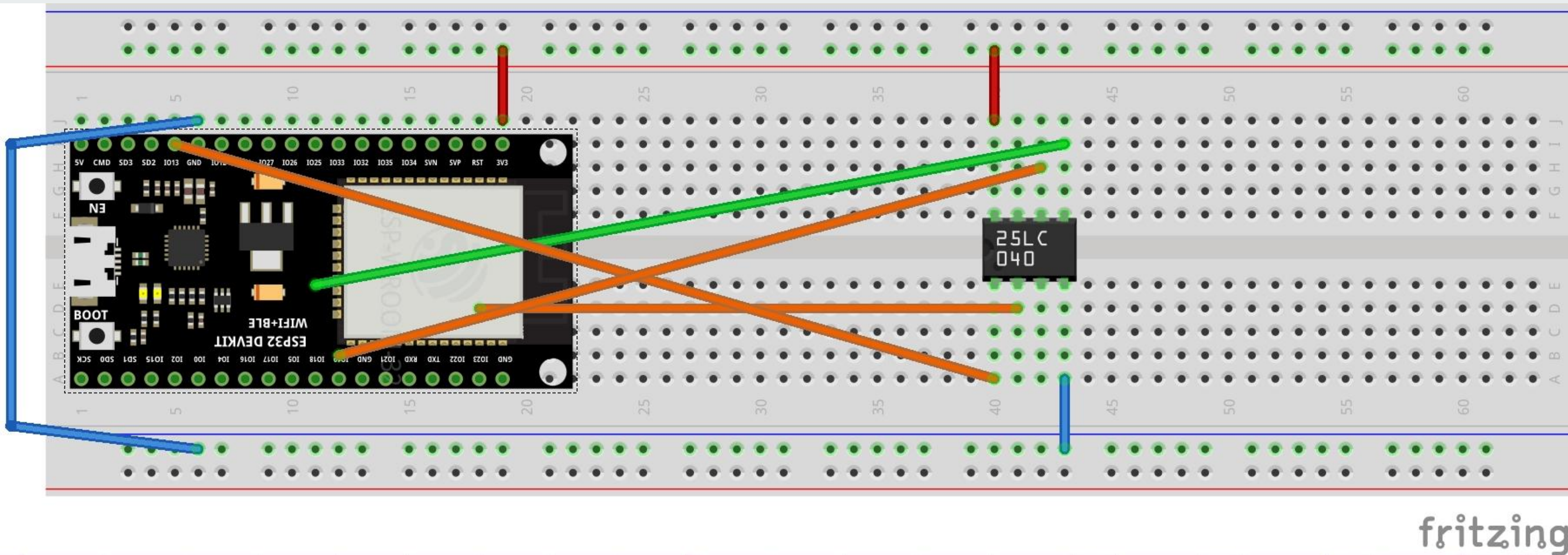

ESPRESSIF
ESP32-WROOM-32D
3V3 EN VP VN 34 35 32 33 25 26 27 14 12 GND 13 D2 D3 CMD 5V
GND 23 22 TX RX 21 GND 19 18 5 17 16 4 0 2 15 D1 D0 CLK
Boot

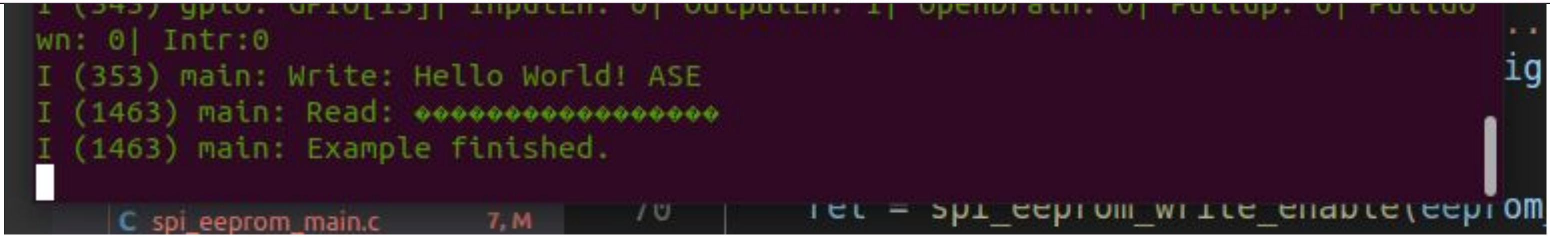
ESP32-DevKitC


3V3
PU EN
OD/ID S_VP ADC1_0 RTC GPIO36
OD/ID S_VN ADC1_3 RTC GPIO39
OD/ID VDET_1 ADC1_6 RTC GPIO34
OD/ID VDET_2 ADC1_7 RTC GPIO35
OD/ID 32K_XP TOUCH9 ADC1_4 RTC GPIO32
OD/ID 32K_XN TOUCH8 ADC1_5 RTC GPIO33
OD/ID DAC_1 ADC1_8 RTC GPIO25
OD/ID DAC_2 ADC2_9 RTC GPIO26
OD/ID TOUCH7 ADC2_7 RTC GPIO27
OD/IE/WPU HSPI_SCK MTMS TOUCH6 ADC2_6 RTC GPIO14
OD/IE/WPD VDD_FLASH HSPI_MISO MTDI TOUCH5 ADC2_5 RTC GPIO12
GND
OD/IE/WPD HSPI_MOSI MTCK ADC2_4 TOUCH4 RTC GPIO13
! OD/IE/WPU D2 GPIO9
! OD/IE/WPU D3 GPIO10
! OD/IE/WPU CMD GPIO11
5V0
ESPRESSIF
ESP32-WROOM
EN
Boot
GND
GPIO23 VSPI_MOSI SPI_MOSI OD/IE
GPIO22 WIRE_SCL OD/IE
GPIO1 U0TXD SERIAL_TX OD/IE/WPU
GPIO3 U0RXD SERIAL_RX OD/IE/WPU
GPIO21 WIRE_SDA OD/IE
GND
GPIO19 VSPI_MISO OD/IE
GPIO18 VSPI_SCK OD/IE
GPIO5 VSPI_SS SDIO OD/IE/WPU
GPIO17 OD/IE
GPIO16 OD/IE
GPIO4 RTC ADC2_0 TOUCH0 OD/IE/WPD
GPIO0 RTC ADC2_1 TOUCH1 BOOT OD/IE/WPU
GPIO2 RTC ADC2_2 TOUCH2 OD/IE/WPD
GPIO15 RTC ADC2_3 TOUCH3 MTDO HSPI_SS LOG OD/IE/WPU
GPIO8 D1 OD/IE/WPU !
GPIO7 D0 OD/IE/WPU !
GPIO6 SCK OD/IE/WPU !




```
#ifdef CONFIG_IDF_TARGET_ESP32
#    define EEPROM_HOST    VSPI_HOST
#    define PIN_NUM_MISO 19
#    define PIN_NUM_MOSI 23
#    define PIN_NUM_CLK  18
#    define PIN_NUM_CS   13
#  endif
```


5V CMD SD3 SD2 IO13 GND IO12 IO14 IO27 IO26 IO25 IO33 IO32 IO35 IO34 SVN SVP RST 3V3
EN
BOOT
ESP32 DEVKIT
WIFI+BLE
CLK SDO SD1 IO15 IO2 IO0 IO4 IO16 IO17 IO5 IO18 IO19 GND IO21 RXD TXD IO22 IO23 GND
25LC
040
fritzing



```
I (343) gpio: GPIO[13]| InputEn: 0| OutputEn: 1| OpenDrain: 0| Pullup: 0| Pulldo
wn: 0| Intr:0
I (353) main: Write: Hello World! ASE
I (1463) main: Read: ������������������
I (1463) main: Example finished.
```

```
C spi_eeprom_main.c   7, M      70      ret = spi_eeprom_write_enable(eeprom
```

Bibliografia

- ESP-IDF Programming Guide
  - https://docs.espressif.com/projects/esp-idf/en/latest/esp32/index.html
- Insight Into ESP32 Sleep Modes & Their Power Consumption - Last Minute Engineers
  - https://lastminuteengineers.com/esp32-sleep-modes-power-consumption/
- Programming Languages for ESP32 and ESP8266
  - https://www.electronicdiys.com/2018/11/programming-languages-for-esp32-esp8266.html
- ESP32 - Technical Reference Manual
  - https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
- ESP32 Series - Datasheet
  - https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf
- ESP32WROOM32E ESP32 WROOM 32UE - Datasheet
  - https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf
- Espressif - Github
  - https://github.com/espressif
- Memória microchip eeprom 25LC040A
  - http://ww1.microchip.com/downloads/en/devicedoc/21827c.pdf