Lab 2

General Purpose Input/Output (GPIO)

The goal of this second lab is to get you familiarized with the various APIs exposed by the ESP-IDF framework. Firstly, you will discover the pinout of the development board and program with the GPIOs API. After that, you will discover by yourself the Serial Peripheral Interface (SPI) protocol used for transmission. The main goal is to correctly connect the BME280 sensor to the ESP32 via SPI and get measures from it. Another goal of the lab is to put you in the context of reading documentation from different sources in order to get devices communicate with each other.

2.1 Introduction

The ESP32 chip features 40 physical GPIO pads. Some GPIO pads cannot be used or do not have the corresponding pin on the chip package(refer to technical reference manual). Each pad can be used as a general purpose I/O or can be connected to an internal peripheral signal.

Note that GPIO6-11 are usually used for SPI flash. GPIO34-39 can only be set as input mode and do not have software pullup or pulldown functions [1].

2.2 GPIO API [2]

GPIO set direction

Configure GPIO direction,such as `GPIO_MODE_INPUT`, `GPIO_MODE_OUTPUT`, `GPIO_MODE_INPUT_OUTPUT`, etc.

```
esp_err_t gpio_set_direction(gpio_num_t gpio_num, gpio_mode_t mode)
```

Return
- `ESP_OK Success`
- `ESP_ERR_INVALID_ARG GPIO error`

Parameters
- `gpio_num` : Configure GPIO pins number, it should be GPIO number. If you want to set direction of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16) ;

1. Take a look at Section Pin Description of the ESP32 datasheet p.7
2. For a complete API reference, please refer to the official documentation of the ESP-IDF `https://esp-idf.readthedocs.io/en/v1.0/api/gpio.html`(link).

— `mode` : GPIO direction

GPIO set output level

```
esp_err_t gpio_set_level(gpio_num_t gpio_num, uint32_t level)
```

Return
— `ESP_OK Success`
— `ESP_ERR_INVALID_ARG GPIO number error`
Parameters
— `gpio_num` : GPIO number. If you want to set the output level of e.g. GPIO16, `gpio_num` should be `GPIO_NUM_16` (16) ;
— level : Output level. 0 : low ; 1 : high

GPIO get input level

```
int gpio_get_level(gpio_num_t gpio_num)
```

Return
— 0 the GPIO input level is 0
— 1 the GPIO input level is 1
Parameters
— `gpio_num` : GPIO number. If you want to get the logic level of e.g. pin GPIO16, `gpio_num` should be `GPIO_NUM_16` (16) ;

2.3 Breadboard

From now on, you will be using a breadboard to hookup all the wires. Figure 2.1 depicts how energy flows through a typical breadboard. More precisely, energy in blocks A and D is distributed horizontally while in blocks B and C, it is distributed vertically. Note that all blocks are independant.

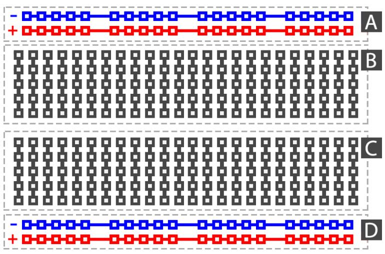

FIGURE 2.1 – Basic breadboard layout. Source : `https://www.tweaking4all.com/wp-content/uploads/2013/12/basic_breadboard_layout.png`

2.1 EXERCICES

1. Create a new project by following this structure (`main.c` containing all the stuff you are implementing) :

```
– src/
        – main.c
        – component.mk
– Makefile
```

Note that, the ESP-IDF has a particular build system which is based on Makefiles. The file `component.mk`, which is by the way empty, is used to specify to the build system that all files in this subdirectory have to be included into the project and thus compiled using the toolchain. [3]

2. Write a function `blink_task` which turns on then off the integrated LED of the ESP32 indifinitely. Note that the integrated LED is connected to `GPIO_2`. You can make the function call `vTaskDelay(1000/portTICK_RATE_MS);` in your program so as to add a delay of 1000ms between pull-up and pull-down. Do not forget to include `freertos/FreeRTOS.h` as well as `freertos/task.h` headers.

3. The entry point of ESP-IDF programs is not the usual `main` but `app_main`. Define an entry point for your program and launch the previously defined function in a `freeRTOS` task [4].

4. Make the necessary hook-ups with many LEDs on the breadboard in order to make light effects of you choice. Modify you program to operate these light effects.

3. If you are interested to know more about the ESP-IDF build system, you can dig into the contents of `esp-idf/make/` folder.

4. The tiny microcontrollers that you are handling run a true operating system which features multithreading ! We will discover freeRTOS in more details during the next labs.

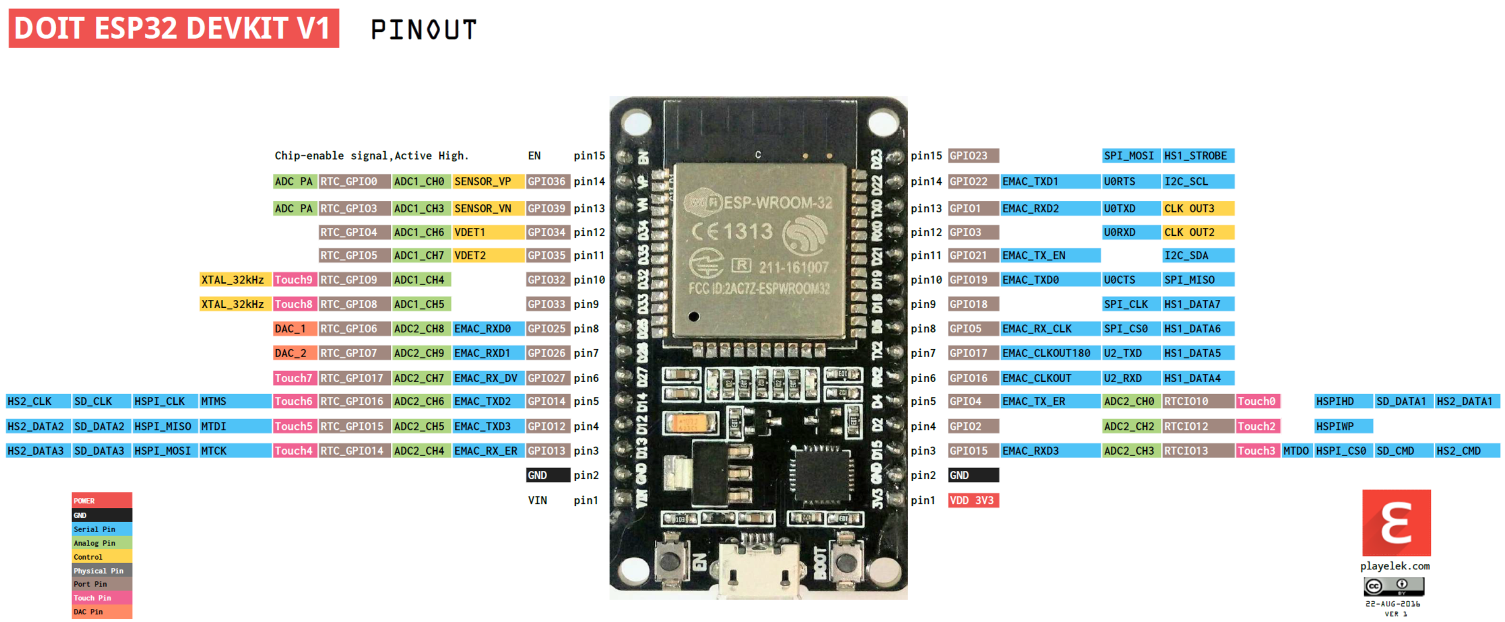

FIGURE 2.2 – ESP32 development board pinout.

2.3.1 Serial Peripheral Interface (SPI) — a hardware perspective

The Serial Peripheral Interface bus (SPI) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The interface was developed by Motorola in the mid 1980s and has become a de facto standard.

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) lines.

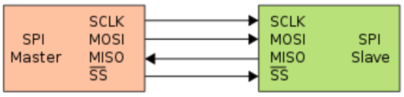

FIGURE 2.3 – Single master to single slave basic connection. Source : Wikipedia.

The SPI bus specifies four logic signals :

- SCLK : Serial Clock (output from master).
- MOSI : Master Output Slave Input, or Master Out Slave In (data output from master).
- MISO : Master Input Slave Output, or Master In Slave Out (data output from slave).
- SS : Slave Select (often active low, output from master).

While the above pin names are the most popular, in the past alternative pin naming conventions were sometimes used, and so SPI port pin names for older IC products may differ from those depicted in these illustrations :

- Serial Clock :
  - SCLK : SCK.
- Master Output → Slave Input :
  - MOSI : SIMO, SDI, DI, DIN, SI, MTSR.
- Master Input ← Slave Output :
  - MISO : SOMI, SDO, DO, DOUT, SO, MRST.
- Slave Select : SS : $\overline{SS}$, SSEL, CS, $\overline{CS}$, CE, nSS.

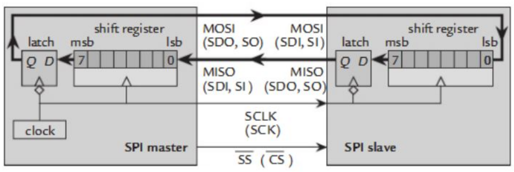

FIGURE 2.4 – SPI convention clarification. Source : pjrc forum.

the MOSI/MISO convention requires that, on devices using the alternate names, SDI on the master be connected to SDO on the slave, and vice versa (See Figure 2.4. Slave Select is the same functionality as chip select and is used instead of an addressing concept. Pin names are always capitalized as in Slave Select, Serial Clock, and Master Output Slave Input. from (Wikipedia)

General Purpose SPI	HSPIQ_in/_out	Any GPIO Pins	Standard SPI consists of clock, chip-select, MOSI and MISO. These SPIs can be connected to LCD and other external devices. They support the following features: • both master and slave modes; • 4 sub-modes of the SPI format transfer that depend on the clock phase (CPHA) and clock polarity (CPOL) control; • configurable SPI frequency; • up to 64 bytes of FIFO and DMA.
	HSPID_in/_out		
	HSPICLK_in/_out		
	HSPI_CS0_in/_out		
	HSPI_CS1_out		
	HSPI_CS2_out		
	VSPIQ_in/_out		
	VSPID_in/_out		
	VSPICLK_in/_out		
	VSPI_CS0_in/_out		
	VSPI_CS1_out		
	VSPI_CS2_out		

FIGURE 2.5 – SPI peripheral pin configuration. Source : ESP32 datasheet p. 29.

2.3.2 SPI peripheral devices in the ESP32

The ESP32 has four SPI peripheral devices, called SPI0, SPI1, HSPI and VSPI. SPI0 is entirely dedicated to the flash cache the ESP32 uses to map the SPI flash device it is connected to into memory. SPI1 is connected to the same hardware lines as SPI0 and is used to write to the flash chip. HSPI and VSPI are free to use. SPI1, HSPI and VSPI all have three chip select lines, allowing them to drive up to three SPI devices each as a master. The SPI peripherals also can be used in slave mode, driven from another SPI master.

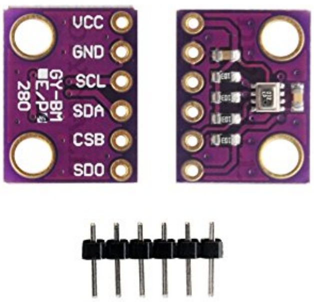

FIGURE 2.6 – BME280 pinout. Source : BME280 datasheet p.30.

2.2 EXERCICES

1. Establish a correspondance between the naming convention used in the BME280 and the one used in the ESP32 pinout.
2. At the end of the lab, suggest a possible hook-up to attach the sensor device to the development board. Which pins (of the ESP32) are you using ? What can you say about the pins that are prefixed with HSPI_* and SPI_* ?

2.3.3 Serial Peripheral Interface (SPI) — a software perspective

SPI peripherals featured by the ESP32 can operate in two modes ; slave and master. In the first mode, the `spi_slave` driver allows using the HSPI and/or VSPI as a full-duplex SPI slave. It can make use of DMA to send/receive transactions of arbitrary length. In the second, The `spi_master` driver allows easy communication with SPI slave devices, even in a multithreaded environment. It fully transparently handles DMA transfers multiplexing between differnet SPI slaves on the same master. In the following we will use the ESP32 in master mode and the BME280 in slave mode.

2.3.4 SPI API [5]

Software-accessible SPI peripherals (`spi_common.h`)

```c
/**
 * @brief Enum with the three SPI peripherals that are software-accessible in it
 */
typedef enum {
    SPI_HOST=0,                 ///< SPI1, SPI
    HSPI_HOST=1,                ///< SPI2, HSPI
    VSPI_HOST=2                 ///< SPI3, VSPI
} spi_host_device_t;
```

Configuration structure for a SPI bus (`spi_master.h`)

```c
/**
 * @brief This is a configuration structure for a SPI bus.
 *
 * You can use this structure to specify the GPIO pins of the bus. Normally, the driver will use the
 * GPIO matrix to route the signals. An exception is made when all signals either can be routed through
 * the IO_MUX or are -1. In that case, the IO_MUX is used, allowing for >40MHz speeds.
 *
 * @note Be advised that the slave driver does not use the quadwp/quadhd lines and fields in spi_bus_config_t
 * refering to these lines will be ignored and can thus safely be left uninitialized.
 */
typedef struct {
    int mosi_io_num;            ///< GPIO pin for Master Out Slave In (=spi_d) signal, or -1 if not used.
    int miso_io_num;            ///< GPIO pin for Master In Slave Out (=spi_q) signal, or -1 if not used.
    int sclk_io_num;            ///< GPIO pin for Spi CLocK signal, or -1 if not used.
    int quadwp_io_num;          ///< GPIO pin for WP (Write Protect) signal which is used as D2 in 4-bit
                                ///  communication modes, or -1 if not used.
    int quadhd_io_num;          ///< GPIO pin for HD (HolD) signal which is used as D3 in 4-bit
                                ///  communication modes, or -1 if not used.
    int max_transfer_sz;        ///< Maximum transfer size, in bytes. Defaults to 4094 if 0.
} spi_bus_config_t;
```

Configuration of the device interface (`spi_master.h`)

```c
        /**
         * @brief This is a configuration for a SPI slave device that is connected to one of the SPI buses.
         */
        typedef struct {
            uint8_t command_bits;           ///< Default amount of bits in command phase (0-16), used when
                                            /// ''SPI_TRANS_VARIABLE_CMD'' is not used, otherwise ignored.
            uint8_t address_bits;           ///< Default amount of bits in address phase (0-64), used when
                                            /// ''SPI_TRANS_VARIABLE_ADDR'' is not used, otherwise ignored.
            uint8_t dummy_bits;             ///< Amount of dummy bits to insert between address and data phase
            uint8_t mode;                   ///< SPI mode (0-3)

                [...]

            int clock_speed_hz;             ///< Clock speed, in Hz
            int spics_io_num;               ///< CS GPIO pin for this device, or -1 if not used
            uint32_t flags;                 ///< Bitwise OR of SPI_DEVICE_* flags
            int queue_size;                 ///< Transaction queue size. This sets how many transactions
                                            ///  can be 'in the air' (queued using spi_device_queue_trans but not
                                            ///  yet finished
```

5. For a complete reference, take a look at the ESP-IDF documentation `https://esp-idf.readthedocs.io/en/v2.0/api/peripherals/spi_master.html`(link)

```
                                        ///  using spi_device_get_trans_result) at the same time
    transaction_cb_t pre_cb;            ///< Callback to be called before a transmission is started.
                                        ///  This callback is called within interrupt context.
    transaction_cb_t post_cb;           ///< Callback to be called after a transmission has completed.
                                        ///  This callback is called within interrupt context.
} spi_device_interface_config_t;
```

Buffer allocation (esp_heap_caps.h)

```
/**
 * @brief Allocate a chunk of memory which has the given capabilities
 *
 * Equivalent semantics to libc malloc(), for capability-aware memory.
 *
 * In IDF, ``malloc(p)`` is equivalent to ``heap_caps_malloc(p, MALLOC_CAP_8BIT)``.
 *
 * @param size Size, in bytes, of the amount of memory to allocate
 * @param caps        Bitwise OR of MALLOC_CAP_* flags indicating the type
 *                    of memory to be returned
 *
 * @return A pointer to the memory allocated on success, NULL on failure
 */
void *heap_caps_malloc(size_t size, uint32_t caps);
```

SPI bus initialization (spi_master.h)

```
/**
 * @brief Initialize a SPI bus
 *
 * @warning For now, only supports HSPI and VSPI.
 *
 * @param host SPI peripheral that controls this bus
 * @param bus_config Pointer to a spi_bus_config_t struct specifying how the host should be initialized
 * @param dma_chan Either channel 1 or 2, or 0 in the case when no DMA is required. Selecting a DMA channel
 *                 for a SPI bus allows transfers on the bus to have sizes only limited by the amount of
 *                 internal memory. Selecting no DMA channel (by passing the value 0) limits the amount of
 *                 bytes transfered to a maximum of 32.
 *
 * @warning If a DMA channel is selected, any transmit and receive buffer used should be allocated in
 *          DMA-capable memory.
 *
 * @return
 *         - ESP_ERR_INVALID_ARG   if configuration is invalid
 *         - ESP_ERR_INVALID_STATE if host already is in use
 *         - ESP_ERR_NO_MEM        if out of memory
 *         - ESP_OK                on success
 */
esp_err_t spi_bus_initialize(spi_host_device_t host, const spi_bus_config_t *bus_config, int dma_chan);
```

Attaching a device to the SPI bus (spi_master.h)

```
/**
 * @brief Allocate a device on a SPI bus
 *
 * This initializes the internal structures for a device, plus allocates a CS pin on the indicated SPI master
 * peripheral and routes it to the indicated GPIO. All SPI master devices have three CS pins and can thus control
 * up to three devices.
 *
 * @note While in general, speeds up to 80MHz on the dedicated SPI pins and 40MHz on GPIO-matrix-routed pins are
 *       supported, full-duplex transfers routed over the GPIO matrix only support speeds up to 26MHz.
 *
 * @param host SPI peripheral to allocate device on
 * @param dev_config SPI interface protocol config for the device
 * @param handle Pointer to variable to hold the device handle
 * @return
 *         - ESP_ERR_INVALID_ARG   if parameter is invalid
 *         - ESP_ERR_NOT_FOUND     if host doesn't have any free CS slots
 *         - ESP_ERR_NO_MEM        if out of memory
 *         - ESP_OK                on success
 */
esp_err_t spi_bus_add_device(spi_host_device_t host, spi_device_interface_config_t *dev_config,
                             spi_device_handle_t *handle);
```

2.3 EXERCICES

1. Create a new project. Define in the beginning of your `.c` file, just after headers inclusions, the necessary macros which will be used to specify the pins, or GPIOs, that you have selected in the previous part in order to perform transmissions between the sensor and the development board.
2. Allocate necessary structures and fill the corresponding fields. Some fields are specified for more complex usage and are not required in our case.
3. Now, you have to initialize the SPI bus then attach the sensor device using the right function calls. Pay attention in the case of using Direct Memory Access (DMA) for data transfers when you initialize the SPI bus. In this case, you have to allocate the transfer buffers in a memory which can be accessed in DMA mode, *i.e.* using the macro `MALLOC_CAP_DMA` defined in `esp_heap_caps.h`.

2.4 BME280 driver

2.4.1 BME280 device structure (bme280_defs.h)

```
        /*!
         * @brief bme280 device structure
         */
        struct bme280_dev {
                /*! Chip Id */
                uint8_t chip_id;
                /*! Device Id */
                uint8_t dev_id;
                /*! SPI/I2C interface */
❶               enum bme280_intf intf;
                /*! Read function pointer */
❷               bme280_com_fptr_t read;
                /*! Write function pointer */
❸               bme280_com_fptr_t write;
                /*! Delay function pointer */
                bme280_delay_fptr_t delay_ms;
                /*! Trim data */
                struct bme280_calib_data calib_data;
                /*! Sensor settings */
                struct bme280_settings settings;
        };
```

The exposed device structure allows you to specify some configurations to match your specific architecture.

— ❶ : you can switch between SPI and I2C serial protocols ;

— ❷, ❸ : you can provide a costum read/write function that is based on the communication schema that is available in your architecture ;

SPI transaction (spi_master.h)

```
/**
 * This structure describes one SPI transaction. The descriptor should not be modified until the transaction finishes.
 */
struct spi_transaction_t {
    uint32_t flags;                 ///< Bitwise OR of SPI_TRANS_* flags
    uint16_t cmd;                   ///< Command data, of which the length is set in the ``command_bits`` of
                                    ///spi_device_interface_config_t.
                                    ///< <b>NOTE: this field, used to be "command" in ESP-IDF 2.1 and before,
                                    ///is re-written to be used in a new way in ESP-IDF 3.0.</b>
                                    ///< - Example: write 0x0123 and command_bits=12 to send command 0x12, 0x3_
                                    ///(in previous version, you may have to write 0x3_12).
    uint64_t addr;                  ///< Address data, of which the length is set in the ``address_bits`` of
                                    ///spi_device_interface_config_t.
                                    ///< <b>NOTE: this field, used to be "address" in ESP-IDF 2.1 and before,
                                    ///is re-written to be used in a new way in ESP-IDF3.0.</b>
                                    ///< - Example: write 0x123400 and address_bits=24 to send address of 0x12,
                                    ///0x34, 0x00 (in previous version, you may have to write 0x12340000).
    size_t length;                  ///< Total data length, in bits
    size_t rxlength;                ///< Total data length received, should be not greater than ``length`` in
                                    ///full-duplex mode (0 defaults this to the value of ``length``).
    void *user;                     ///< User-defined variable. Can be used to store eg transaction ID.
    union {
        const void *tx_buffer;      ///< Pointer to transmit buffer, or NULL for no MOSI phase
        uint8_t tx_data[4];         ///< If SPI_USE_TXDATA is set, data set here is sent directly from this variable.
    };
    union {
        void *rx_buffer;            ///< Pointer to receive buffer, or NULL for no MISO phase. Written by 4
                                   ///bytes-unit if DMA is used.
        uint8_t rx_data[4];         ///< If SPI_USE_RXDATA is set, data is received directly to this variable
    };
} ;         //the rx data should start from a 32-bit aligned address to get around dma issue.
```

Queueing a SPI transaction for execution (spi_master.h)

```
/**
 * @brief Queue a SPI transaction for execution
 *
 * @param handle Device handle obtained using spi_host_add_dev
 * @param trans_desc Description of transaction to execute
 * @param ticks_to_wait Ticks to wait until there's room in the queue; use portMAX_DELAY to
 *                      never time out.
 * @return
 *         - ESP_ERR_INVALID_ARG   if parameter is invalid
 *         - ESP_ERR_TIMEOUT       if there was no room in the queue before ticks_to_wait expired
 *         - ESP_ERR_NO_MEM        if allocating DMA-capable temporary buffer failed
 *         - ESP_OK                on success
 */
esp_err_t spi_device_queue_trans(spi_device_handle_t handle, spi_transaction_t *trans_desc, TickType_t ticks_to_wait);
```

Get the result of a SPI transaction (spi_master.h)

```
/**
 * @brief Get the result of a SPI transaction queued earlier
 *
 * This routine will wait until a transaction to the given device (queued earlier with
 * spi_device_queue_trans) has succesfully completed. It will then return the description of the
 * completed transaction so software can inspect the result and e.g. free the memory or
 * re-use the buffers.
 *
 * @param handle Device handle obtained using spi_host_add_dev
 * @param trans_desc Pointer to variable able to contain a pointer to the description of the transaction
       that is executed. The descriptor should not be modified until the descriptor is returned by
       spi_device_get_trans_result.
 * @param ticks_to_wait Ticks to wait until there's a returned item; use portMAX_DELAY to never time
                       out.
 * @return
 *         - ESP_ERR_INVALID_ARG   if parameter is invalid
 *         - ESP_ERR_TIMEOUT       if there was no completed transaction before ticks_to_wait expired
 *         - ESP_OK                on success
 */
esp_err_t spi_device_get_trans_result(spi_device_handle_t handle, spi_transaction_t **trans_desc,
                                      TickType_t ticks_to_wait);
```

Send SPI transaction (spi_master.h)

```
/**
 * @brief Do a SPI transaction
 *
 * Essentially does the same as spi_device_queue_trans followed by spi_device_get_trans_result. Do
 * not use this when there is still a transaction queued that hasn't been finalized
 * using spi_device_get_trans_result.
 *
 * @param handle Device handle obtained using spi_host_add_dev
 * @param trans_desc Description of transaction to execute
 * @return
 *         - ESP_ERR_INVALID_ARG   if parameter is invalid
 *         - ESP_OK                on success
 */
esp_err_t spi_device_transmit(spi_device_handle_t handle, spi_transaction_t *trans_desc);
```

2.4 EXERCICES

1. Download the `BME280` driver that is available from bosch sensortech git `https://github.com/BoschSensortec/BME280_driver`. Take a closer look at the `BME280` driver and by referring to the subset of the ESP-IDF SPI API provided above, suggest an implementation for the read and write functions. Make sure your implementation is working correctly with the hook-up you proposed earlier.
   Try to follow this project structure (`main.c` containing all the stuff you are implementing) :

```
        - lib/
                - include/
                   - bme280.h
                   - bme280_defs.h
                - bme280.c
                - changelog.md
                - README.md
                - component.mk
        - src/
                - main.c
                - component.mk
        - Makefile
```

2.5 `BME280` API

Device initialization

```
/*!
 *  @brief This API is the entry point.
 *  It reads the chip-id and calibration data from the sensor.
 *
 *  @param[in,out] dev : Structure instance of bme280_dev
 *
 *  @return Result of API execution status
 *  @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme280_init(struct bme280_dev *dev);
```

Getting measures from device

```
/*!
 * @brief This API reads the pressure, temperature and humidity data from the
 * sensor, compensates the data and store it in the bme280_data structure
 * instance passed by the user.
 *
 * @param[in] sensor_comp : Variable which selects which data to be read from
 * the sensor.
 *
 * sensor_comp |   Macros
 * ------------|-------------------
 *     1       | BME280_PRESS
 *     2       | BME280_TEMP
 *     4       | BME280_HUM
 *     7       | BME280_ALL
 *
 * @param[out] comp_data : Structure instance of bme280_data.
 * @param[in] dev : Structure instance of bme280_dev.
 *
 * @return Result of API execution status
 * @retval zero -> Success / +ve value -> Warning / -ve value -> Error
 */
int8_t bme280_get_sensor_data(uint8_t sensor_comp, struct bme280_data *comp_data, struct bme280_dev *dev);
```

Settings structure

```
/*!
 * @brief bme280 sensor settings structure which comprises of mode,
 * oversampling and filter settings.
 */
struct bme280_settings {
        /*! pressure oversampling */
        uint8_t osr_p;
        /*! temperature oversampling */
        uint8_t osr_t;
        /*! humidity oversampling */
        uint8_t osr_h;
        /*! filter coefficient */
        uint8_t filter;
        /*! standby time */
        uint8_t standby_time;
};
```

2.5 EXERCICES

1. Extend your program and exploit the implementation of the read and write functions you provided to get measures from the sensor and display them with the correct format. Make sure everything works well. Don't forget to initialize the device.
2. Explore the `BME280` driver, especially the calibration part. Extend your program so as to allow the user to parametrize the calibration process. Same thing with the parameters exposed in the settings structure.
3. Now, try to connect mutiple sensors (slaves) to the ESP32 using SPI.
4. Bonus : Re-do the same work in this lab using I2C protocol.

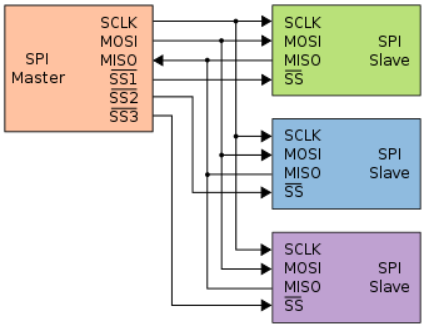

FIGURE 2.7 – Typical SPI bus : master and three independant slaves. Source : Wikipedia.

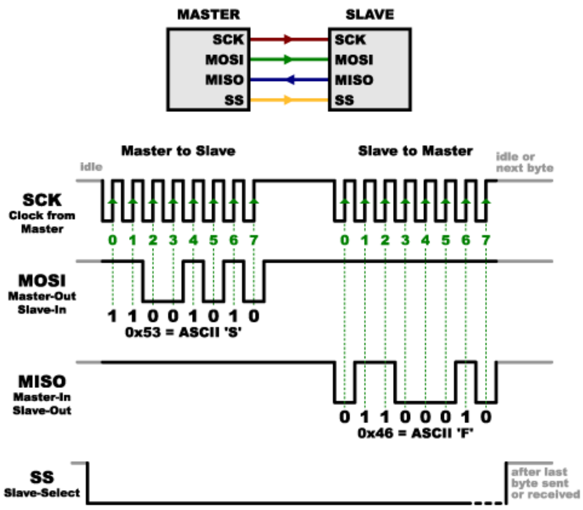

FIGURE 2.8 – SPI protocol timing diagram. Source : SparkFun.