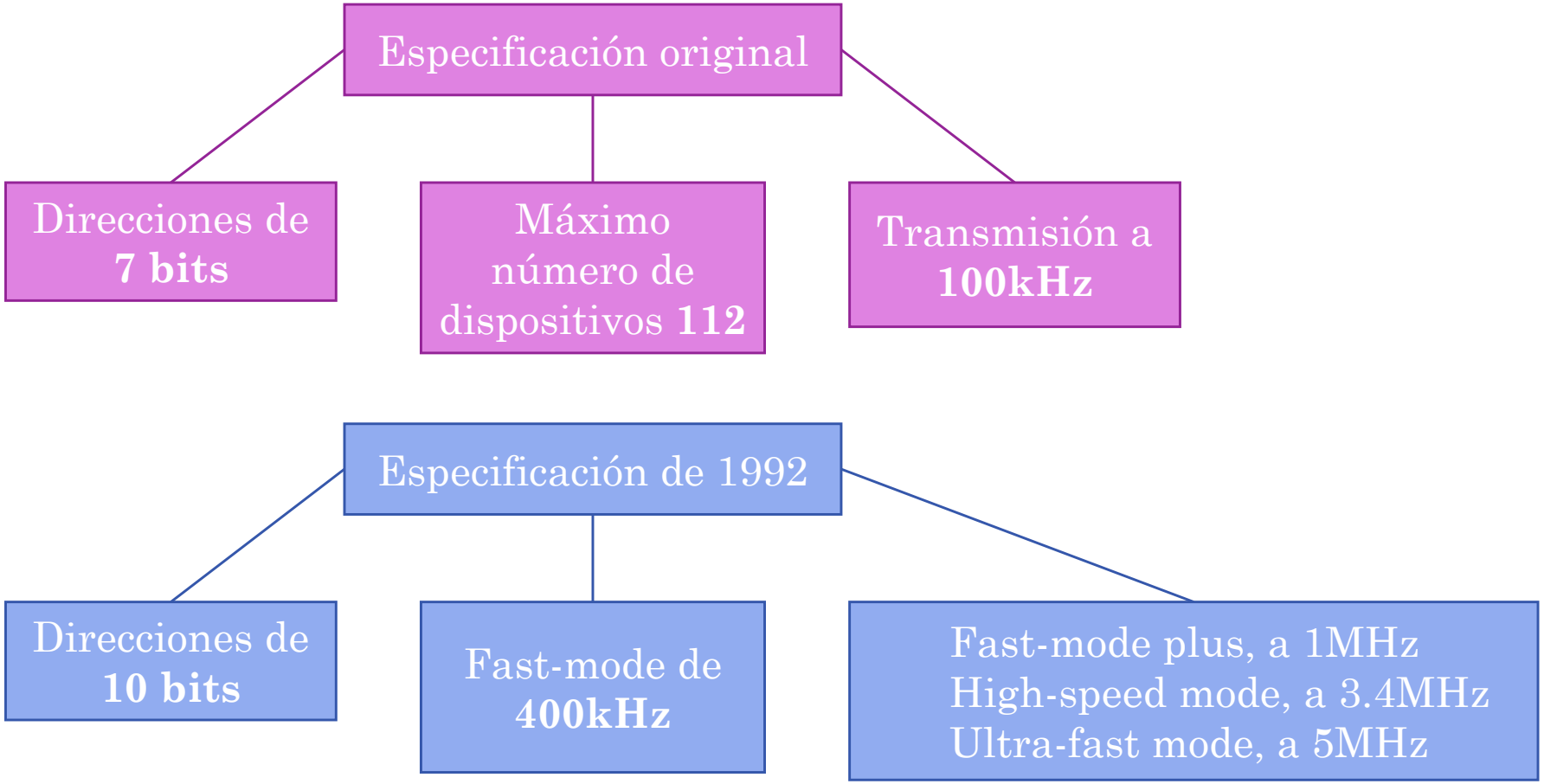
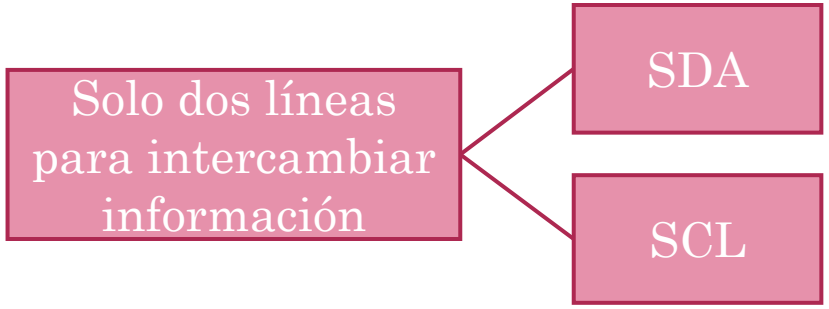
I2C (Inter-integrated-circuit)

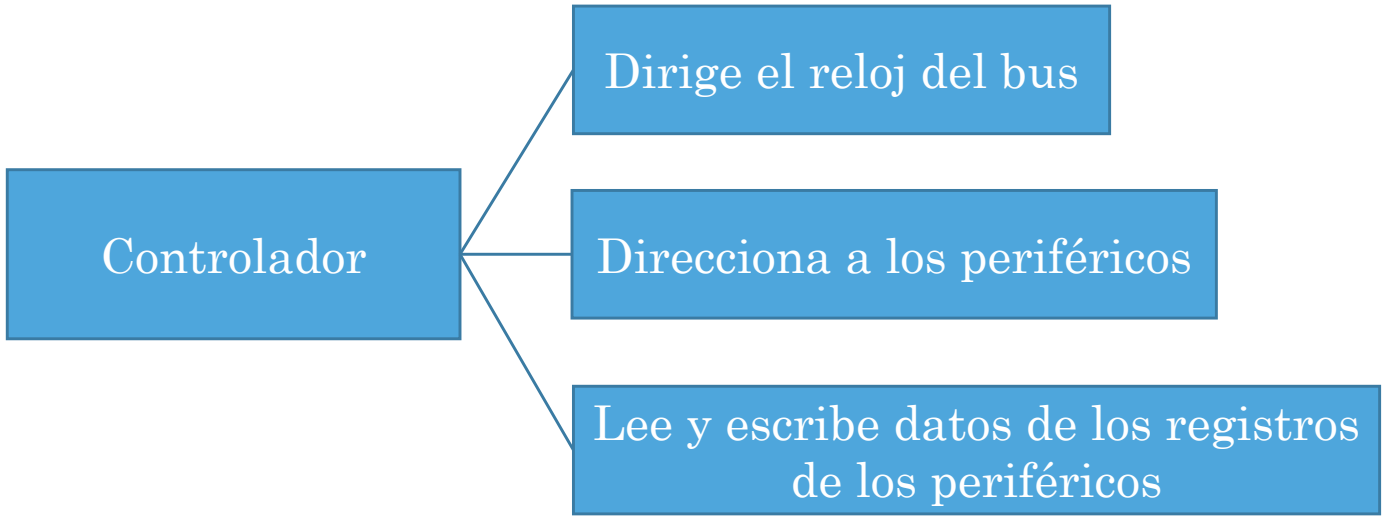


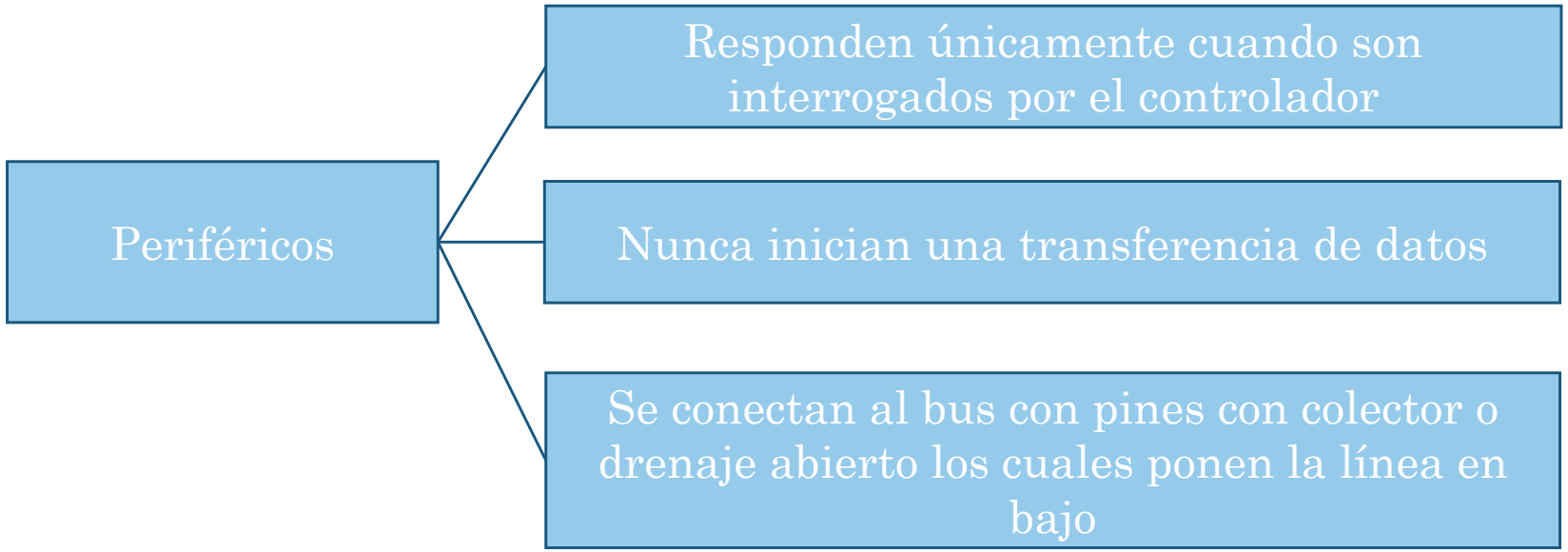
*No todos los dispositivos soportan estos modos



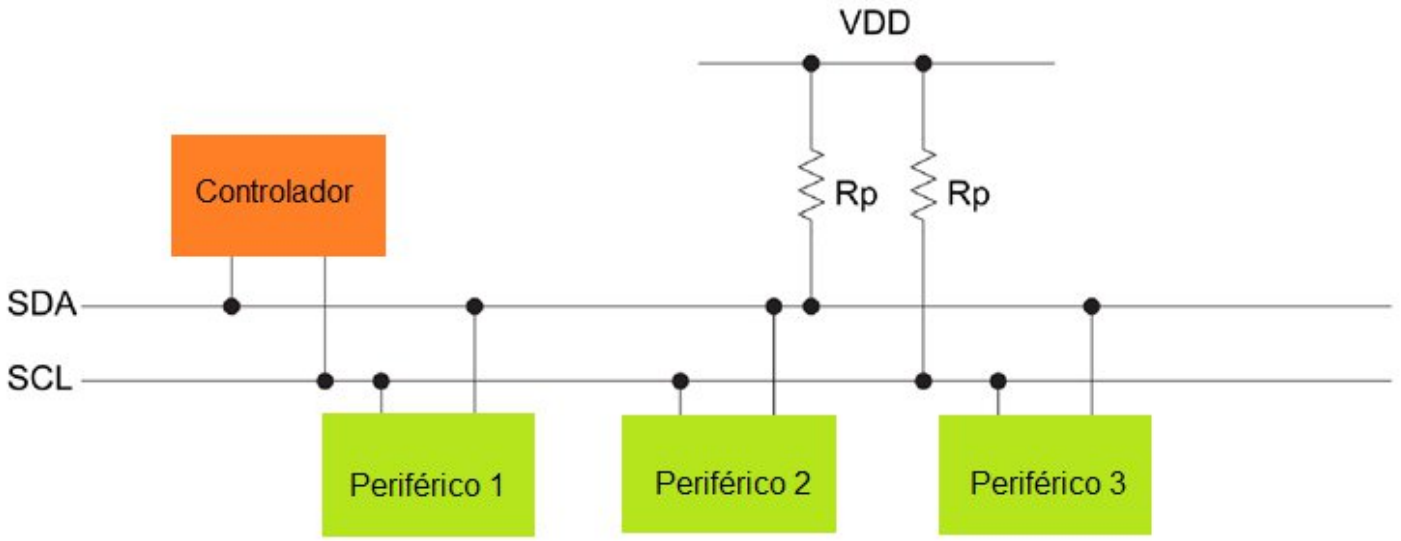
Bus bidireccional

Jerarquía controlador/periférico


Controlador
Dirige el reloj del bus
Direcciona a los periféricos
Lee y escribe datos de los registros de los periféricos



Cuando no se están transmitiendo datos, las líneas están en alto


VDD
Rp
Rp
Controlador
SDA
SCL
Periférico 1
Periférico 2
Periférico 3

Condiciones de Inicio y Fin

Condición de Inicio

- Ocurre al inicio de la transmisión
- Es iniciada por el controlador para despertar a los periféricos en el bus del estado inactivo
- Es una transición de la línea SDA de estado alto a bajo

*Es una de las dos ocasiones donde se permite que SDA cambie de estado cuando SCL está en alto

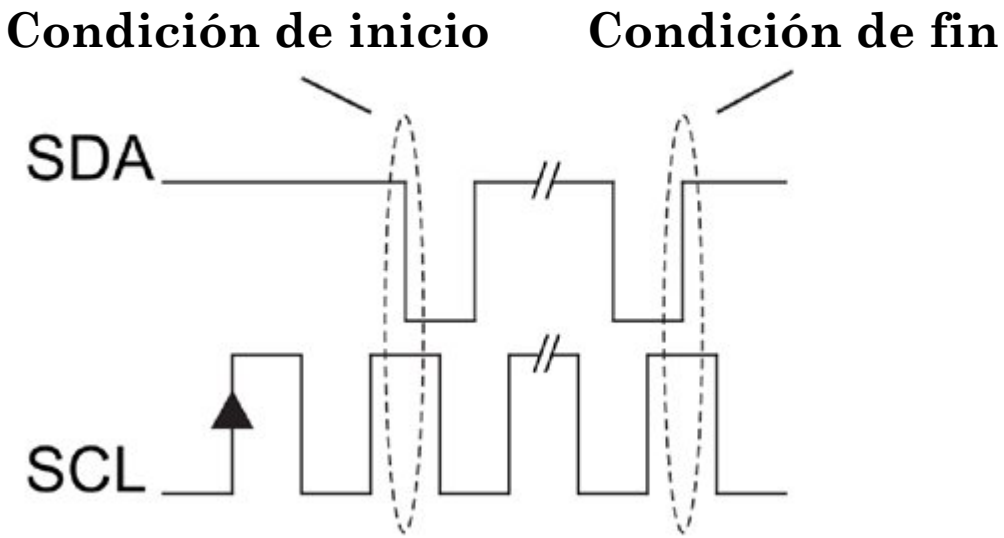
Condición de Fin

Ocurre al final de la transmisión

Es generada por el controlador para indicar a los periféricos que deben regresar al estado inactivo, liberar la línea SDA y no enviar más datos en el bus.

Es una transición de SDA del estado bajo a alto, mientras SCL está en alto

*Es la segunda ocasión donde se permite que SDA cambie de estado cuando SCL está en alto



En el resto de las condiciones, la línea SDA solo cambia cuando SCL está en bajo.

Tipos de frame

Frame de dirección

Es siempre el primero en cualquier secuencia de comunicación

Para direcciones de 7 bits, la dirección es enviada empezando del MSB

La dirección es seguida de un bit que indica la operación: un **1** para lectura o **0** para escritura

El noveno bit es **NACK/ACK**

Una vez que el octavo bit ha sido enviado, el dispositivo receptor toma control de SDA.

Si el receptor no coloca a SDA en bajo antes del noveno pulso de reloj, se infiere que el receptor no recibió los datos o no supo como procesar el mensaje.

En este caso, el intercambio de información se detiene y depende del controlador decidir como se procede

Frame de datos

Después de enviar la dirección, inicia la transmisión de datos

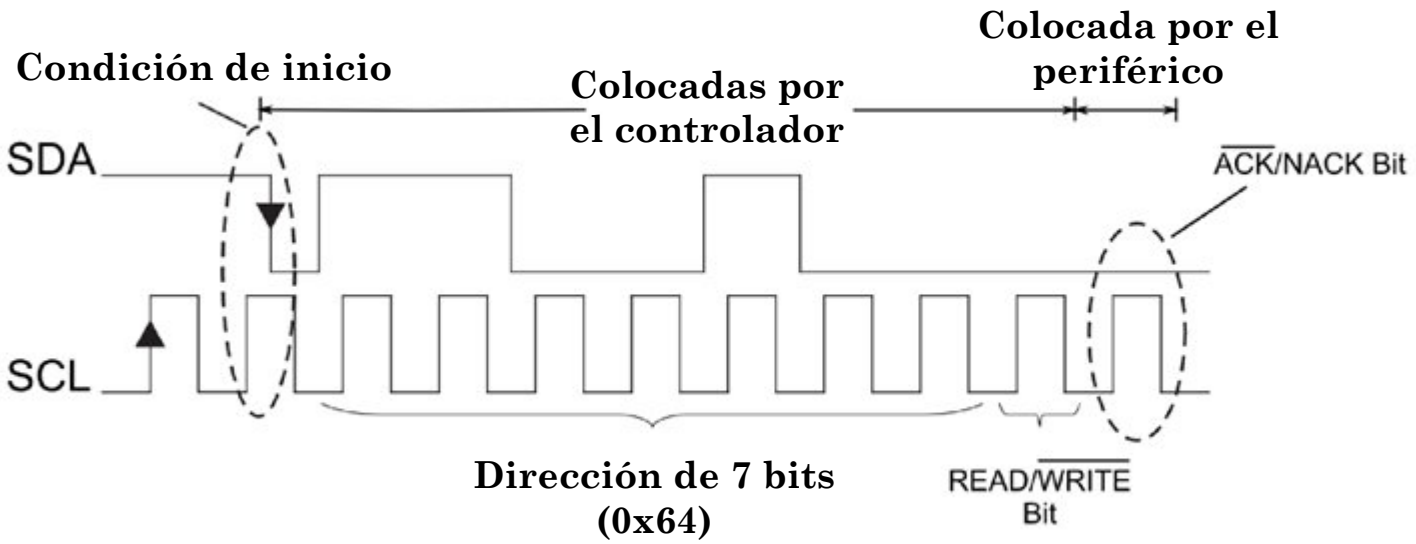
El controlador genera pulsos de reloj a un intervalo regular

Los datos son colocados en SDA por el controlador o el periférico dependiendo si fue una lectura o escritura

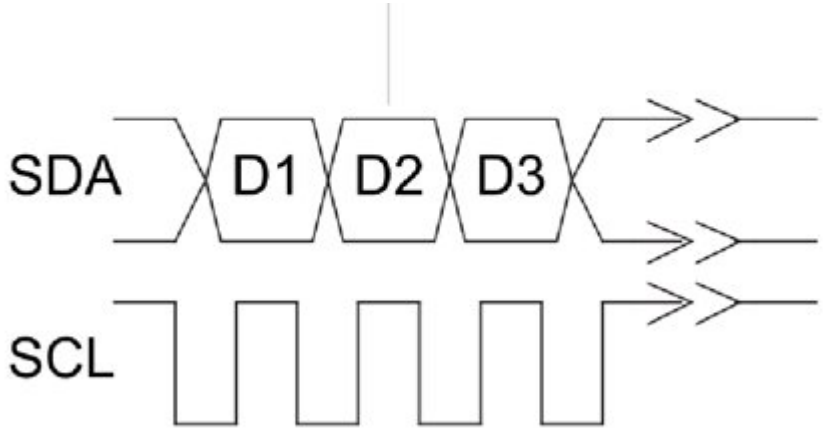
Los datos se transmiten en formatos de 8 bits, iniciando del MSB

Frame de datos

- Cada bit esta sincronizado con la señal de reloj
- El controlador genera pulsos de reloj a un intervalo regular
- No hay límite respecto a la cantidad de bytes a transmitir
- Cada byte debe ser seguido de un ACK generado por el receptor


Condición de inicio
Colocadas por el controlador
Colocada por el periférico
SDA
SCL
ACK/NACK Bit
Dirección de 7 bits (0x64)
READ/WRITE Bit

SDA permanece estable cuando SCL está en alto



Los bits en SDA pueden cambiar únicamente cuando SCL está en bajo.

De manera que el receptor lee un bit cuando SCL está en alto y el transmisor envía cada bit cuando SCL está en bajo.

ACK/NACK

Después de cada byte transmitido, el receptor envía un bit de ACK/NACK

Un ACK denota que un byte (ya sea una dirección o un dato) se recibió exitosamente

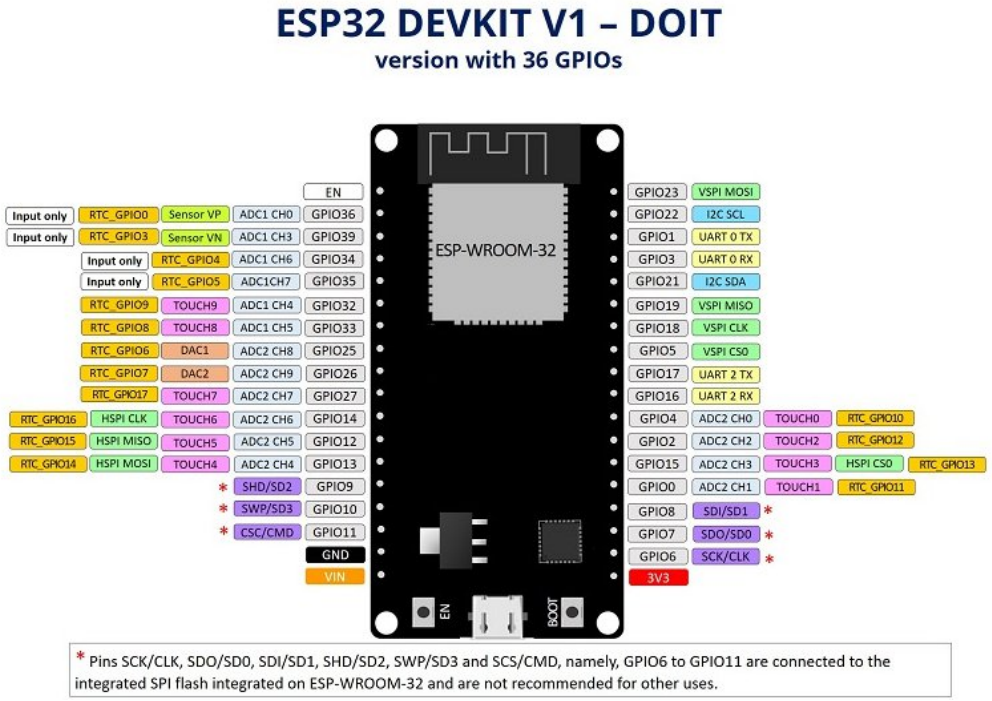
Un NACK indica que ocurrió un error en la transmisión de datos

El controlador usa un NACK después del último byte que quiere leer del periférico para indicarle al periférico que debe terminar la transmisión

El receptor genera el ACK al poner la línea SDA en bajo

Se genera un NACK cuando el receptor deja a SDA en alto y no responde de ninguna manera

Default	
SDA	GPIO 21
SCL	GPIO 22



https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/i2c.html

MASTER

```c
#include <stdio.h>
#include <string.h>
#include "esp_log.h"
#include "driver/i2c.h"

static const char *TAG = "master";

#define I2C_MASTER_SCL      22
#define I2C_MASTER_SDA      21
#define I2C_SLAVE_ADDRESS   0x1C
#define ACK_CHECK_EN        0x1
```

```c
void app_main(void)
{
    uint8_t led_on[] = "ON";
    uint8_t led_off[] = "OFF";

    ESP_ERROR_CHECK(i2c_master_init());
    ESP_LOGI(TAG, "I2C inicializado");

    while(1)
    {
        i2c_master_send(led_on, sizeof(led_on));
        ESP_LOGI(TAG, "LED on request");
        vTaskDelay(500/ portTICK_PERIOD_MS);
        i2c_master_send(led_off, sizeof(led_off));
        ESP_LOGI(TAG, "LED off request");
        vTaskDelay(500/ portTICK_PERIOD_MS);
    }
}
```

```
esp_err_t i2c_master_init(void)
{
    i2c_config_t i2c_master_conf = {
        .mode = I2C_MODE_MASTER,
        .sda_io_num = I2C_MASTER_SDA,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_io_num = I2C_MASTER_SCL,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,
        .master.clk_speed = 400000,
    };

    esp_err_t error = i2c_param_config(I2C_NUM_0, &i2c_master_conf);

    if (error != ESP_OK) {
        return error;
    }
    return i2c_driver_install(I2C_NUM_0, i2c_master_conf.mode, 0, 0, 0);
}
```

i2c_config_t

```c
/**
 * @brief I2C initialization parameters
 */
typedef struct{
    i2c_mode_t mode;      /*!< I2C mode */
    int sda_io_num;       /*!< GPIO number for I2C sda signal */
    int scl_io_num;       /*!< GPIO number for I2C scl signal */
    bool sda_pullup_en;   /*!< Internal GPIO pull mode for I2C sda signal*/
    bool scl_pullup_en;   /*!< Internal GPIO pull mode for I2C scl signal*/

    union {
        struct {
            uint32_t clk_speed;      /*!< I2C clock frequency for master mode, (no higher than 1MHz for now) */
        } master;                    /*!< I2C master config */
#if SOC_I2C_SUPPORT_SLAVE
        struct {
            uint8_t addr_10bit_en;   /*!< I2C 10bit address mode enable for slave mode */
            uint16_t slave_addr;     /*!< I2C address for slave mode */
            uint32_t maximum_speed;  /*!< I2C expected clock speed from SCL. */
        } slave;                     /*!< I2C slave config */
#endif // SOC_I2C_SUPPORT_SLAVE
    };
    uint32_t clk_flags;              /*!< Bitwise of ``I2C_SCLK_SRC_FLAG_**FOR_DFS**`` for clk source choice*/
} i2c_config_t;
```

enum `i2c_mode_t`

Values:

`I2C_MODE_SLAVE` = 0

I2C slave mode

`I2C_MODE_MASTER`

I2C master mode

`I2C_MODE_MAX`

```
esp_err_t i2c_param_config(
i2c_port_t i2c_num, /* Puerto I2C (I2C_NUM_0 o I2C_NUM_1) */
const i2c_config_t *i2c_conf /* Configuración de los parámetros de I2C */
)
```

```
esp_err_t i2c_driver_install(
i2c_port_t i2c_num, /* Puerto I2C (I2C_NUM_0 o I2C_NUM_1) */
i2c_mode_t mode, /* Modo de operación (I2C_MODE_MASTER o I2C_MODE_SLAVE) */
size_t slv_rx_buf_len, /* Tamaño del buffer de recepción para modo esclavo. En modo maestro, es 0. */
size_t slv_tx_buf_len, /* Tamaño del buffer de transmisión para modo esclavo. En modo maestro, es 0. */
int intr_alloc_flags /* Banderas de interrupciones */
)
```

```
esp_err_t i2c_master_send(uint8_t * data, uint8_t len)
{
    esp_err_t res;
    i2c_cmd_handle_t cmd_handle = i2c_cmd_link_create();
    i2c_master_start(cmd_handle);
    i2c_master_write_byte(cmd_handle, I2C_SLAVE_ADDRESS << 1 | I2C_MASTER_WRITE, ACK_CHECK_EN);
    i2c_master_write(cmd_handle, data, len, ACK_CHECK_EN);
    i2c_master_stop(cmd_handle);

    /* send all queued commands */
    res = i2c_master_cmd_begin(I2C_NUM_0, cmd_handle, 1000 / portTICK_PERIOD_MS);
    i2c_cmd_link_delete(cmd_handle);
    return res;
}
```

i2c_cmd_handle_t **i2c_cmd_link_create**(void): Crea un enlace de comandos I2C.
Retorna un puntero a la estructura que va a contiene los comandos de I2C a ejecutar.

esp_err_t **i2c_master_start**(
i2c_cmd_handle_t **cmd_handle** /* Manejador creado con **i2c_cmd_link_create** */
)

esp_err_t **i2c_master_write_byte**(
i2c_cmd_handle_t **cmd_handle**, /* Manejador creado con **i2c_cmd_link_create** */
uint8_t **data**, /* Byte a escribir en el bus */
bool **ack_en** /* Si se debe esperar un ACK del esclavo */
)

esp_err_t **i2c_master_write**(
i2c_cmd_handle_t **cmd_handle**, /* Manejador creado con **i2c_cmd_link_create** */
uint8_t ***data**, /* Datos a enviar */
size_t **data_len**, /* Longitud de los datos */
bool **ack_en** /* Si se debe esperar un ACK del esclavo */
)

esp_err_t **i2c_master_stop**(
i2c_cmd_handle_t **cmd_handle**, /* Manejador creado con **i2c_cmd_link_create** */
)

esp_err_t **i2c_master_cmd_begin**(
i2c_port_t **i2c_num**, /* Número de puerto I2C */
i2c_cmd_handle_t **cmd_handle**, /* Manejador creado con **i2c_cmd_link_create** */
TickType_t **ticks_to_wait** /* Tiempo máximo en ticks que la función esperará para ejecutar los comandos en I2C */
)

void **i2c_cmd_link_delete**(
i2c_cmd_handle_t **cmd_handle**, /* Manejador creado con **i2c_cmd_link_create** */
)

SLAVE

```
#include <stdio.h>
#include <string.h>
#include "esp_log.h"
#include "driver/i2c.h"
#include "driver/gpio.h"

static const char *TAG = "slave";

#define LED_PIN     GPIO_NUM_2

#define I2C_SLAVE_SCL              22
#define I2C_SLAVE_SDA              21
#define I2C_SLAVE_ADDRESS          0x1C
#define I2C_SLAVE_RX_BUF_LEN       1024

#define LED_ON_CMD                 "ON"
#define LED_OFF_CMD                "OFF"
```

```c
void app_main(void)
{
    uint8_t rx_buffer[I2C_SLAVE_RX_BUF_LEN] = {0};

    gpio_reset_pin(LED_PIN);
    gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);

    ESP_ERROR_CHECK(i2c_slave_init());
    ESP_LOGI(TAG, "I2C inicializado");

    while(1)
    {
        if (i2c_slave_read_buffer(I2C_NUM_1, rx_buffer, I2C_SLAVE_RX_BUF_LEN, 100 / portTICK_PERIOD_MS) > 0)
        {
            i2c_reset_rx_fifo(I2C_NUM_1);

            if(!strncmp((const char*)rx_buffer, LED_ON_CMD, sizeof(LED_ON_CMD)))
            {
                gpio_set_level(LED_PIN, 1);
                ESP_LOGI(TAG, "LED ON");
            }
            else if(!strncmp((const char*)rx_buffer, LED_OFF_CMD, sizeof(LED_OFF_CMD)))
            {
                gpio_set_level(LED_PIN, 0);
                ESP_LOGI(TAG, "LED OFF");
            }

            memset(rx_buffer, 0, I2C_SLAVE_RX_BUF_LEN);
        }
    }
}
```

```
esp_err_t i2c_slave_init(void)
{
    i2c_config_t i2c_slave_conf = {
        .mode = I2C_MODE_SLAVE,
        .sda_io_num = I2C_SLAVE_SDA,
        .sda_pullup_en = GPIO_PULLUP_ENABLE,
        .scl_io_num = I2C_SLAVE_SCL,
        .scl_pullup_en = GPIO_PULLUP_ENABLE,

        .slave.addr_10bit_en = 0,
        .slave.slave_addr = I2C_SLAVE_ADDRESS,
    };

    esp_err_t error = i2c_param_config(I2C_NUM_1, &i2c_slave_conf);

    if (error != ESP_OK) {
        return error;
    }
    return i2c_driver_install(I2C_NUM_1, i2c_slave_conf.mode, I2C_SLAVE_RX_BUF_LEN, 0, 0);
}
```

int **i2c_slave_read_buffer**(
i2c_port_t **i2c_num**, /* El puerto I2C */
uint8_t ***data**, /* Buffer donde se guardarán los datos leídos del bus */
size_t **max_size**, /* Tamaño máximo de datos a leer */
TickType_t **ticks_to_wait** /* Tiempo máximo en ticks que la función esperará para recibir datos */
)

Retorno:
ESP_FAIL(-1) Error
Otros(>=0) La cantidad de bytes leídos

Lee bytes del buffer de I2C interno. Cuando el bus I2C recibe datos, la ISR los copia desde el FIFO RX de hardware al ring buffer interno. Al invocar **i2c_slave_read_buffer**, se copian los bytes del ring buffer al buffer *data* del usuario.

```
esp_err_t i2c_reset_rx_fifo(
i2c_port_t i2c_num /* El puerto I2C */
)
```

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/migration-guides/release-5.x/5.2/peripherals.html

Tarea:

- Revisar la documentación del ESP-IDF sobre I2C.
- Ejecutar los ejemplos vistos en clase.
- Actualizar los ejemplos a la última versión del SDK.