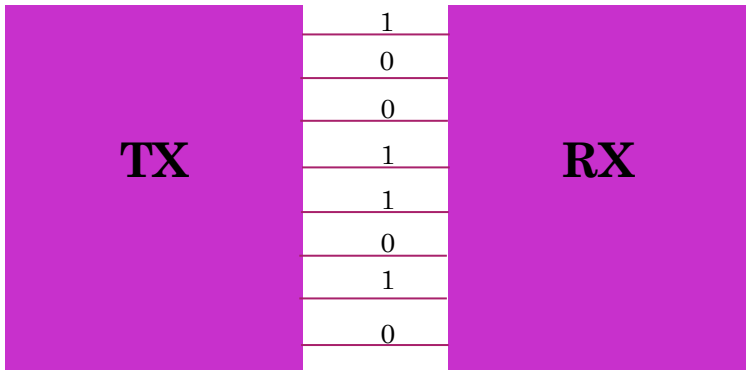
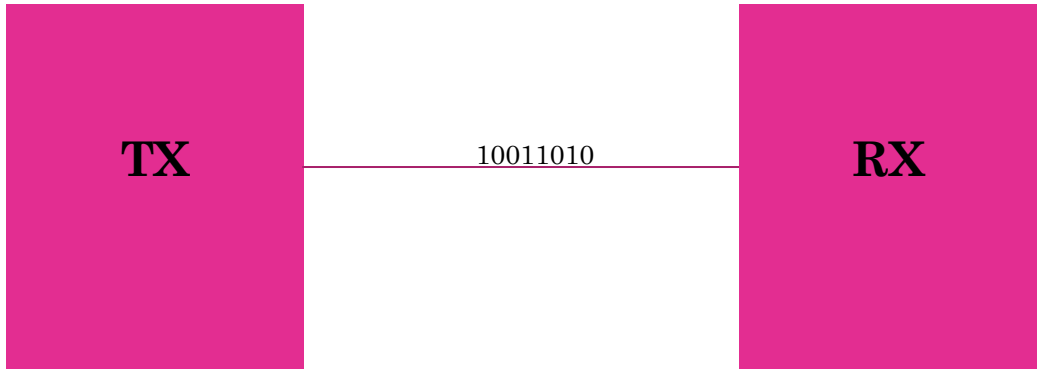
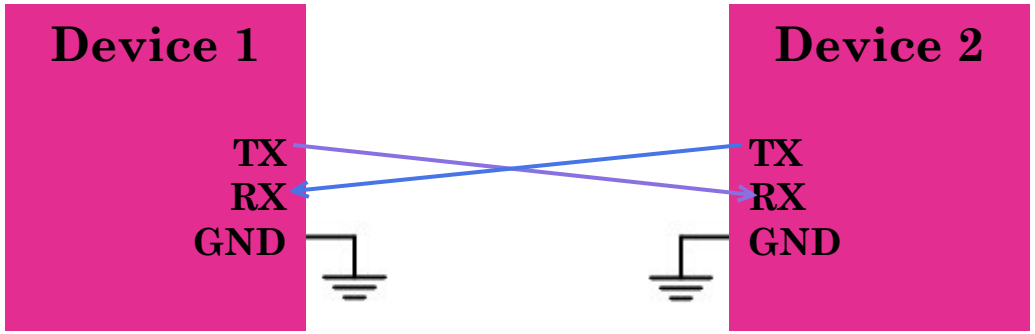
UART (Universal Asynchronous Receiver-Transmitter)






Device 1
TX
RX
GND
Device 2
TX
RX
GND

Pines predefinidos:

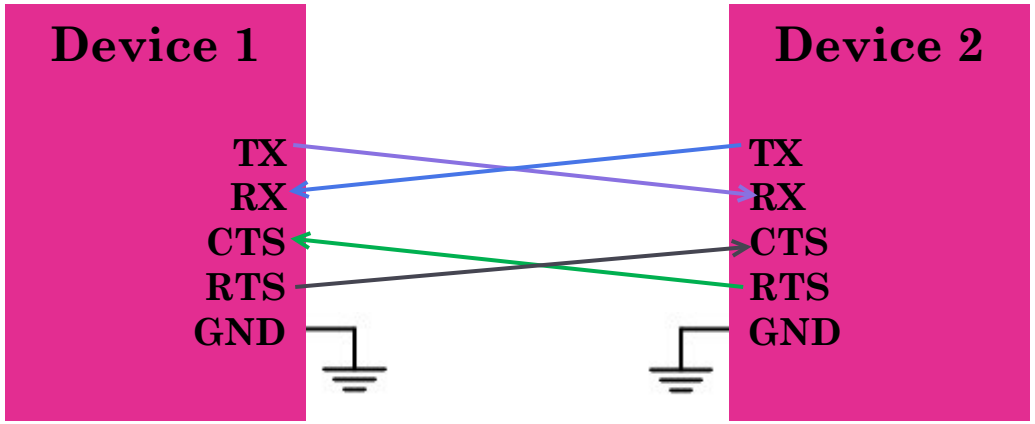
UART	RX	TX	CTS	RTS	
UART0	GPIO3	GPIO1	N/A	N/A	← Usado para depuración
UART1	GPIO9	GPIO10	GPIO6	GPIO11	← Requiere reasignar pines
UART2	GPIO16	GPIO17	GPIO8	GPIO7	

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

C:\Users\<usuario>\esp\esp-idf\examples\peripherals\uart

Control de flujo por hardware

RTS (Request to Send)
CTS (Clear to Send)



enum **uart_hw_flowcontrol_t**

UART hardware flow control modes.

Values:

UART_HW_FLOWCTRL_DISABLE = 0x0

disable hardware flow control

UART_HW_FLOWCTRL_RTS = 0x1

enable RX hardware flow control (rts)

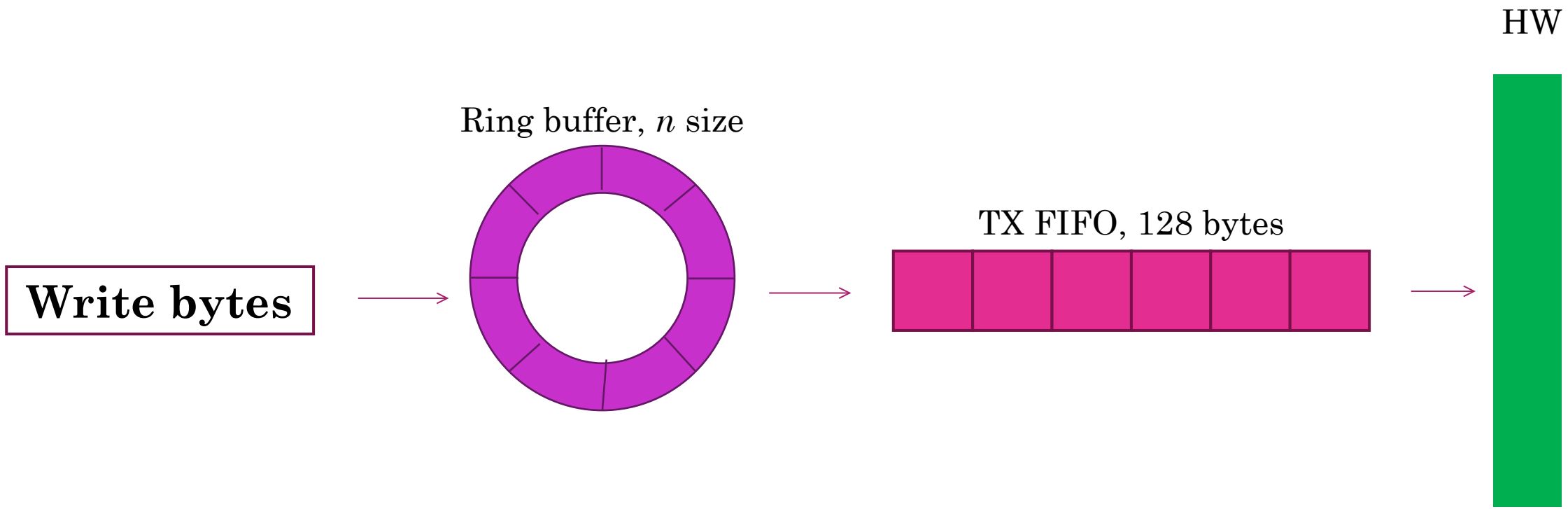
UART_HW_FLOWCTRL_CTS = 0x2

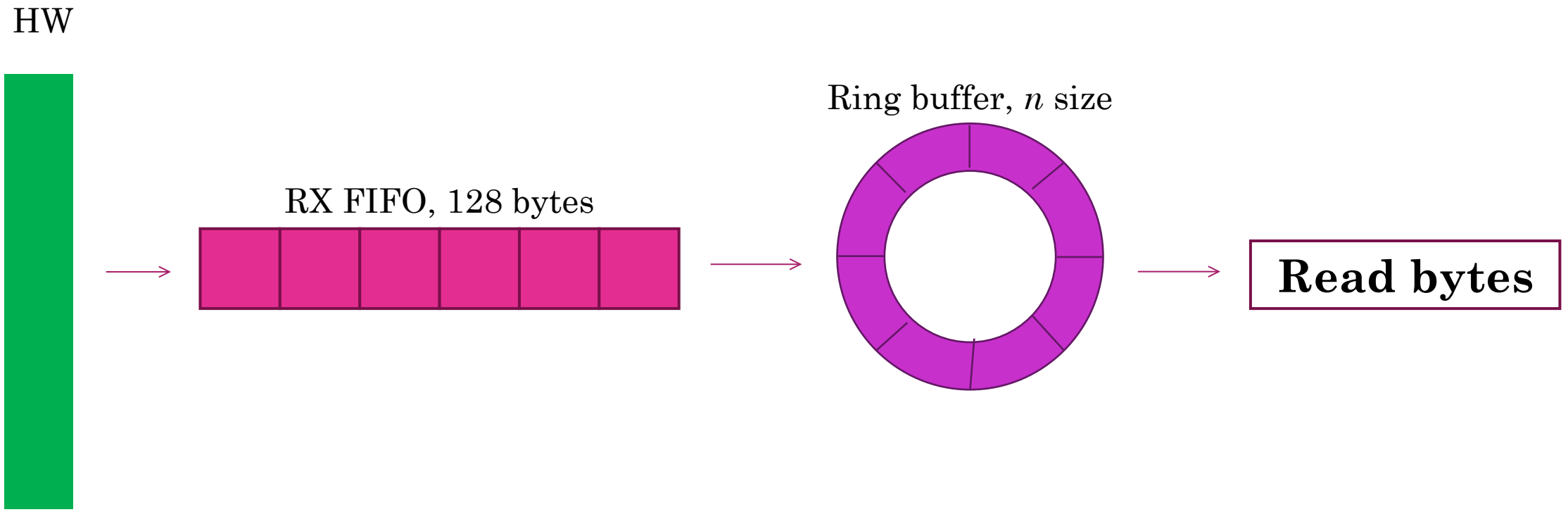
enable TX hardware flow control (cts)

UART_HW_FLOWCTRL_CTS_RTS = 0x3

enable hardware flow control

UART_HW_FLOWCTRL_MAX = 0x4


HW
Ring buffer, n size
Write bytes
TX FIFO, 128 bytes


HW
RX FIFO, 128 bytes
Ring buffer, n size
Read bytes

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/uart.h"
```

```c
#define BUF_SIZE          (1024)
#define UART_RX_PIN       (3)
#define UART_TX_PIN       (1)
```

```c
void app_main(void)
{
    xTaskCreate(echo_task, "uart_echo_task", CONFIG_EXAMPLE_TASK_STACK_SIZE, NULL, 10, NULL);
}
```

```c
static void echo_task(void *arg)
{
    uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    // Configure UART parameters
    ESP_ERROR_CHECK(uart_param_config(UART_NUM_0, &uart_config));

    // Set UART pins
    ESP_ERROR_CHECK(uart_set_pin(UART_NUM_0, UART_TX_PIN, UART_RX_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE));

    // Install UART driver
    ESP_ERROR_CHECK(uart_driver_install(UART_NUM_0, BUF_SIZE * 2, \
                                        BUF_SIZE * 2, 0, NULL, 0));


    // Configure a temporary buffer for the incoming data
    uint8_t *data = (uint8_t *) malloc(BUF_SIZE);

    while (1) {
        // Read data from the UART
        int len = uart_read_bytes(UART_NUM_0, data, (BUF_SIZE - 1), 20 / portTICK_PERIOD_MS);
        // Write data back to the UART
        uart_write_bytes(UART_NUM_0, (const char *) data, len);
    }
}
```

esp_err_t **uart_param_config**(
uart_port_t **uart_num**, /* UART_NUM_0, UART_NUM_1 o UART_NUM_2 */
const uart_config_t ***uart_config** /* Configuración de los parámetros del UART */)

struct **uart_config_t**

UART configuration parameters for uart_param_config function.

Public Members

int **baud_rate**

UART baud rate

uart_word_length_t **data_bits**

UART byte size

uart_parity_t **parity**

UART parity mode

uart_stop_bits_t **stop_bits**

UART stop bits

uart_hw_flowcontrol_t **flow_ctrl**

UART HW flow control mode (cts/rts)

uint8_t **rx_flow_ctrl_thresh**

UART HW RTS threshold

uart_sclk_t **source_clk**

UART source clock selection

```
uart_config_t uart_config = {
    .baud_rate = 115200,
    .data_bits = UART_DATA_8_BITS,
    .parity = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
    .source_clk = UART_SCLK_DEFAULT,
};
```

```
esp_err_t uart_set_pin(
uart_port_t uart_num, /* UART_NUM_0, UART_NUM_1 o UART_NUM_2  */
int tx_io_num, /* GPIO del pin UART TX. */
int rx_io_num, /* GPIO del pin RX de UART */
int rts_io_num, /* GPIO del pin UART RTS */
int cts_io_num) /* GPIO del pin UART CTS */
```

```
uart_set_pin(UART_NUM_0, UART_TX_PIN, UART_RX_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE)
```

UART_PIN_NO_CHANGE

Constant for uart_set_pin function which indicates that UART pin should not be changed

esp_err_t **uart_driver_install**(
uart_port_t **uart_num**, /* UART_NUM_0, UART_NUM_1 o UART_NUM_2 */
int **rx_buffer_size**, /* Tamaño del ring buffer de RX */
int **tx_buffer_size**, /* Tamaño del ring buffer de TX. Si es cero, el controlador no utilizará el búfer TX y la función TX bloqueará la tarea hasta que se hayan enviado todos los datos. */
int **queue_size**, /* Tamaño de la cola para eventos de UART */
QueueHandle_t ***uart_queue**, /* Cola de eventos UART (parámetro de salida) */
int **intr_alloc_flags**) /* Banderas utilizadas para asignar la interrupción */

QueueHandle_t ***uart_queue**
En caso de éxito, escribe aquí un nuevo identificador de cola para proporcionar acceso a los eventos UART. Si se configura como NULL, el controlador no utilizará una cola de eventos.

```
uart_driver_install(UART_NUM_0, BUF_SIZE * 2, \
                    BUF_SIZE * 2, 0, NULL, 0)
```

```
int uart_read_bytes(
uart_port_t uart_num, /* UART_NUM_0, UART_NUM_1 o UART_NUM_2 */
uint8_t *buf, /* Buffer donde se almacenarán los datos leídos */
uint32_t length, /* Cantidad máxima de bytes a leer desde el buffer de recepción */
TickType_t ticks_to_wait /* Tiempo máximo de espera */
)
```

TickType_t **ticks_to_wait**
El tiempo máximo que la función debe esperar por los datos antes de devolver el control. Se indica en ticks. Se puede usar portMAX_DELAY para esperar indefinidamente.

Retorno:
(-1) Error
OTROS (>=0) El número de bytes leídos del UART FIFO

```
int uart_write_bytes(
uart_port_t uart_num, /* UART_NUM_0, UART_NUM_1 o UART_NUM_2 */
const char *src, /* Buffer con los datos a enviar. */
size_t size) /* la cantidad de bytes a escribir */
```

Si el parámetro del controlador UART **'tx_buffer_size' es cero**: esta función no regresará hasta que se hayan enviado todos los datos, o al menos se hayan introducido en TX FIFO.

De lo contrario, si **'tx_buffer_size' > 0**, esta función regresará después de copiar todos los datos al ring buffer de transmisión. Luego, la ISR de UART moverá los datos del ring buffer a TX FIFO gradualmente.

Retorno:
(-1) Error
OTROS (>=0) La cantidad de bytes enviados al FIFO TX

Macro ESP_ERROR_CHECK

La macro tiene una función similar a la de assert, excepto que verifica el valor de **esp_err_t** en lugar de una condición booleana. Si el argumento de ESP_ERROR_CHECK() no es igual a **ESP_OK**, se imprime un mensaje de error en la consola y se llama a abort() para terminar la ejecución del programa.

```c
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/uart.h"
#include "esp_log.h"

static const char *TAG = "uart_events";

#define BUF_SIZE (1024)
#define RD_BUF_SIZE (BUF_SIZE)
static QueueHandle_t uart0_queue;
static const char lf[] = "\n";
```

```c
void app_main(void)
{
    /* Configure parameters of an UART driver,
     * communication pins and install the driver */
    uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    //Install UART driver, and get the queue.
    uart_driver_install(UART_NUM_0, BUF_SIZE * 2, BUF_SIZE * 2, 20, &uart0_queue, 0);
    uart_param_config(UART_NUM_0, &uart_config);

    //Set UART pins (using UART0 default pins ie no changes.)
    uart_set_pin(UART_NUM_0, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE,
                 UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);

    //Create a task to handler UART event from ISR
    xTaskCreate(uart_event_task, "uart_event_task", 2048, NULL, 12, NULL);
}
```

```c
static void uart_event_task(void *pvParameters)
{
    uart_event_t event;
    uint8_t* buffer = (uint8_t*) malloc(RD_BUF_SIZE);
    for(;;) {
        //Waiting for UART event.
        if(xQueueReceive(uart0_queue, (void * )&event, (TickType_t)portMAX_DELAY)) {
            memset(buffer, 0, RD_BUF_SIZE);
            switch(event.type) {
                //Event of UART receving data
                /*We'd better handler data event fast, there would be much more data events than
                other types of events. If we take too much time on data event, the queue might
                be full.*/
                case UART_DATA:
                    uart_read_bytes(UART_NUM_0, buffer, event.size, portMAX_DELAY);
                    ESP_LOGI(TAG, "[UART_DATA]. Len: %d. Data: ", event.size);
                    uart_write_bytes(UART_NUM_0, (const char*) buffer, event.size);
                    uart_write_bytes(UART_NUM_0, lf, sizeof(lf));
                    break;
                //Event of HW FIFO overflow detected
                case UART_FIFO_OVF:
                    ESP_LOGI(TAG, "\nHW FIFO overflow");
                    // If fifo overflow happened, you should consider adding
                    // flow control for your application.
                    // The ISR has already reset the rx FIFO,
                    // As an example, we directly flush the rx buffer here in
                    // order to read more data.
                    uart_flush_input(UART_NUM_0);
                    xQueueReset(uart0_queue);
                    break;
```

```
                //Event of UART ring buffer full
                case UART_BUFFER_FULL:
                    ESP_LOGI(TAG, "\nRing buffer full");
                    // If buffer full happened, you should consider
                    // increasing your buffer size
                    // As an example, we directly flush the rx buffer here in
                    // order to read more data.
                    uart_flush_input(UART_NUM_0);
                    xQueueReset(uart0_queue);
                    break;
                //Event of UART parity check error
                case UART_PARITY_ERR:
                    ESP_LOGI(TAG, "\nUART parity error");
                    break;
                //Event of UART frame error
                case UART_FRAME_ERR:
                    ESP_LOGI(TAG, "\nUART frame error");
                    break;
                //Others
                default:
                    ESP_LOGI(TAG, "\nUART event type: %d", event.type);
                    break;
            }
        }
    }
    free(buffer);
    buffer = NULL;
    vTaskDelete(NULL);
```

UART_DATA:

Ocurre cuando hay datos disponibles para leer en el UART. El campo **size** indica la cantidad de bytes.

UART_FIFO_OVF:

Ocurre cuando el buffer FIFO del UART se desborda, de manera que no se pudieron almacenar más datos en el buffer porque ya estaba lleno.

UART_BUFFER_FULL:

Indica que el buffer de recepción de UART está lleno.

UART_PARITY_ERR:

Ocurre cuando se detecta un error de paridad en los datos recibidos. Puede ser debido a ruido o una configuración incorrecta de la paridad.

UART_FRAME_ERR:

Indica que hubo un error de "framing" en los datos recibidos, es decir, los bits de inicio o fin no coincidieron correctamente con lo esperado.

ESP_LOGI

Macro utilizada para imprimir mensajes de información.

```
ESP_LOGI(tag, format, ...)
```

tag:

Cadena de caracteres usada para identificar el módulo desde donde se está generando el log.

format:

Formato del mensaje, similar a la función printf().

esp_log_level_set(TAG, ESP_LOG_INFO);

CONFIG_LOG_DEFAULT_LEVEL

```c
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "esp_log.h"
#include "driver/uart.h"
#include "string.h"
#include "driver/gpio.h"

static const int RX_BUF_SIZE = 1024;

#define TXD_PIN (GPIO_NUM_4)
#define RXD_PIN (GPIO_NUM_5)
```

```c
void app_main(void)
{
    init();
    xTaskCreate(rx_task, "uart_rx_task", 1024*2, NULL, 10, NULL);
    xTaskCreate(tx_task, "uart_tx_task", 1024*2, NULL, 9, NULL);
}
```

```
void init(void) {
    const uart_config_t uart_config = {
        .baud_rate = 115200,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_DISABLE,
        .stop_bits = UART_STOP_BITS_1,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    // We won't use a buffer for sending data.
    uart_driver_install(UART_NUM_1, RX_BUF_SIZE * 2, 0, 0, NULL, 0);
    uart_param_config(UART_NUM_1, &uart_config);
    uart_set_pin(UART_NUM_1, TXD_PIN, RXD_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE);
}
```

```
int sendData(const char* logName, const char* data)
{
    const int len = strlen(data);
    const int txBytes = uart_write_bytes(UART_NUM_1, data, len);
    ESP_LOGI(logName, "Wrote %d bytes", txBytes);
    return txBytes;
}

static void tx_task(void *arg)
{
    static const char *TX_TASK_TAG = "TX_TASK";
    esp_log_level_set(TX_TASK_TAG, ESP_LOG_INFO);
    while (1) {
        sendData(TX_TASK_TAG, "Hello world");
        vTaskDelay(2000 / portTICK_PERIOD_MS);
    }
}
```

```
static void rx_task(void *arg)
{
    static const char *RX_TASK_TAG = "RX_TASK";
    esp_log_level_set(RX_TASK_TAG, ESP_LOG_INFO);
    uint8_t* data = (uint8_t*) malloc(RX_BUF_SIZE+1);
    while (1) {
        const int rxBytes = uart_read_bytes(UART_NUM_1, data, RX_BUF_SIZE,
                                            1000 / portTICK_PERIOD_MS);
        if (rxBytes > 0) {
            data[rxBytes] = 0;
            ESP_LOGI(RX_TASK_TAG, "Read %d bytes: '%s'", rxBytes, data);
        }
    }
    free(data);
}
```

Tarea:
Revisar la documentación del ESP-IDF sobre UART.