

Configuración ADC DMA Joystick



ayudame revisando el siguiente codigo y dime como está configurado el adc con dma y necesito otro tambien con dma la idea es que ambos valores del adc tome el vrX y el vrY de un joystick /*

```

USER CODE BEGIN Header */
/**
*****
* @file      : main.c
* @brief     : Main program body
*****
* @attention
*
* Copyright (c) 2025 STMicroelectronics.
* All rights reserved.
*
* This software is licensed under terms that can be found in the LICENSE file
* in the root directory of this software component.
* If no LICENSE file comes with this software, it is provided AS-IS.
*
*****
*/
/* USER CODE END Header */
/* Includes -----*/
#include "main.h"

/* Private includes -----*/
/* USER CODE BEGIN Includes */
#include "fsm.h"
#include "stdint.h"
#include "stdio.h"
#include "string.h"
#include <stdlib.h>
#include "arm_math.h"
/* USER CODE END Includes */

/* Private typedef -----*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define -----*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -----*/

```

```

/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables -----*/
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;
DMA_HandleTypeDef hdma_tim3_ch2;

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */
#define FFT_SIZE_MAX 2048
const uint32_t timer_clk = 84000000UL;
static uint16_t adc_buffer[FFT_SIZE_MAX];
static uint16_t fft_size = 1024;
static uint16_t display_value = 0;
static uint8_t nextDigit_FSM = 1;
static uint8_t tx_buffer[256] = {0};
#define FREQ_BUFFER_SIZE 16
float freq_buffer[FREQ_BUFFER_SIZE] = {0};
uint8_t freq_index = 0;
uint8_t freq_full = 0;
char rx_buffer[128] = {0};
uint8_t rx_char = 0;
uint8_t rx_index=0;
volatile e_PosiblesEvents pending_event = IDLE;
volatile uint8_t data_snapshot = 0;

volatile uint8_t flagCapture = 0;
volatile uint32_t firstCapture = 0;
volatile uint32_t secondCapture = 0;
volatile uint32_t elapsedTicks = 0;
volatile uint32_t counterOverflow = 0;
/* USER CODE END PV */

/* Private function prototypes -----*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM4_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM3_Init(void);
/* USER CODE BEGIN PFP */
void displayNumber(uint8_t digitValue);

```

```

void HandleLEDDelayCmd(const char *arg);
void HandleSampleFreqCmd(const char *arg);
void HandleRGBCmd(const char *arg);
void HandlePWMFreqCmd(const char *arg);
void HandleUnknownCmd(void);
void HandleFFTSIZECmd(const char *arg);
void HandleStatusCmd(void);
void HandlePrintADC(void);
void HandleFreqDisplayCmd(void);
void HandleClearCmd(void);
/* USER CODE END PFP */

/* Private user code -----*/
/* USER CODE BEGIN 0 */

e_PosiblesStates state_machine_action(e_PosiblesEvents event) {
    switch (event) {
        case IDLE:
            break;
        case EVENT_ENCODER:
            if (data_snapshot) // DATA snapshot
                display_value = (display_value == 0) ? 4095 : display_value - 1;
            else
                display_value = (display_value == 4095) ? 0 : display_value + 1;
            break;

        case EVENT_SW:
            display_value = 0;
            nextDigit_FSM = 1;
            break;

        case EVENT_TIMER_TICK:{
            // Apagar todos los dígitos
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET); // userDis1
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_RESET); // userDis2
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET); // userDis3
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_RESET); // userDis4

            switch (nextDigit_FSM) {
                case 1:
                    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET);
                    displayNumber(display_value % 10);
                    break;
                case 2:
                    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_SET);
                    displayNumber((display_value / 10) % 10);
                    break;
                case 3:
                    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_SET);
                    displayNumber((display_value / 100) % 10);
                    break;
            }
        }
    }
}

```

```

        case 4:
            HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_SET);
            displayNumber((display_value / 1000) % 10);
            break;
    }
    nextDigit_FSM = (nextDigit_FSM < 4) ? nextDigit_FSM + 1 : 1;
    break;}
case EVENT_USART: {
    // eco
    int len = snprintf((char*)tx_buffer, sizeof(tx_buffer),
        "Comando recibido: %s\r\n", rx_buffer);
    HAL_UART_Transmit(&huart2, tx_buffer, len, 1000);

    // dispatch
    if (strncmp((char*)rx_buffer,"led=",4)==0)  HandleLEDDelayCmd(rx_buffer+4);
    else if (strncmp((char*)rx_buffer,"fmuestreo=",10)==0)
        HandleSampleFreqCmd(rx_buffer+10);
    else if (strncmp((char*)rx_buffer,"rgb=",4)==0)  HandleRGBCmd(rx_buffer+4);
    else if (strncmp((char*)rx_buffer,"fftSize=",8)==0) HandleFFTSizeCmd(rx_buffer+8);
    else if (strncmp((char*)rx_buffer,"status",6)==0) HandleStatusCmd();
    else if (strncmp((char*)rx_buffer,"print",5)==0) HandlePrintADC();
    else if (strncmp((char*)rx_buffer,"freq",4)==0) HandleFreqDisplayCmd();
    else if (strncmp((char*)rx_buffer,"fft",3)==0) HandlePrintFFT();
    else if (strncmp((char*)rx_buffer,"info",4)==0) HandleFFTInfo();
    else if (strncmp((char*)rx_buffer,"help",4)==0) HandleHelpCmd();
    else if (strncmp((char*)rx_buffer, "clear", 5) == 0) HandleClearCmd();
    else
        ;

    memset(rx_buffer,0,sizeof(rx_buffer));
    break;
}
case EVENT_IC_CAPTURE:{
    float periodo_ms = elapsedTicks * (1.0f / 84000000.0f) * 1000.0f; // En ms
    float frecuencia_hz = 84000000.0f / elapsedTicks;
    // Ejemplo: enviar por serial
    freq_buffer[freq_index++] = frecuencia_hz;
    if (freq_index >= FREQ_BUFFER_SIZE) {
        freq_index = 0;
        freq_full = 1;
    }
    break;
}
return event;
}}

void displayNumber(uint8_t digitValue) {
    uint32_t start = DWT->CYCCNT;
    static uint8_t contador_muestras = 0;
    // Apagar todos los segmentos
    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET); // A
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_RESET); // B
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_RESET); // C
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_RESET); // D

```

```
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_RESET); // E
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET); // F
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_RESET); // G
```

```
switch (digitValue) {
    case 0:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
        break;
    case 1:
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        break;
    case 2:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
    case 3:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
    case 4:
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
    case 5:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
    case 6:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
}
```

```

        break;
    case 7:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        break;
    case 8:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
    case 9:
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_10, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
        break;
}
uint32_t elapsed = DWT->CYCCNT - start;

if (contador_muestras < 10) {
    printf("Ciclos para mostrar digito %d: %lu\n", digitValue, (unsigned long)elapsed);
    contador_muestras++;
}
}

void HandleLEDDelayCmd(const char *arg) {
    uint32_t nuevo = atoi(arg);
    // Ajusta TIM2 para el blinky
    __HAL_TIM_SET_AUTORELOAD(&htim2, nuevo);
    __HAL_TIM_SET_COUNTER(&htim2, 0);
    // Feedback
    int len = snprintf((char*)tx_buffer, sizeof(tx_buffer),
        "LED delay = %lu ms\r\n", (unsigned long)nuevo);
    HAL_UART_Transmit(&huart2, tx_buffer, len, 1000);
}

void HandleSampleFreqCmd(const char *arg) {
    uint32_t fs;
    char option = arg[0];
    // Verifica la opción seleccionada (1, 2, 3, 4)
    switch(option) {
        case '1': fs = 44100; break;
        case '2': fs = 48000; break;
        case '3': fs = 96000; break;
        case '4': fs = 128000; break;
        default: {

```

```

        const char *msg = "Opciones válidas para 'fmuestreo=' son: 1, 2, 3, 4\r\n";
        HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 1000);
        return;
    }
}

// calcular ARR y PSC
uint32_t arr = timer_clk / fs - 1;
uint32_t psc = 0;
if (arr > 0xFFFF) {
    // escalar PSC si supera 16 bits
    psc = (arr / 0x10000) + 1;
    arr = (timer_clk / (psc+1) / fs) - 1;
}
// aplicar a TIM3 y reiniciar
__HAL_TIM_SET_PRESCALER(&htim3, psc);
__HAL_TIM_SET_AUTORELOAD(&htim3, arr);
__HAL_TIM_SET_COUNTER(&htim3, 0);
HAL_TIM_Base_Stop(&htim3);
HAL_TIM_Base_Start(&htim3);
// Feedback
int len = snprintf((char*)tx_buffer, sizeof(tx_buffer),
    "Sample TIM3 @ %lu Hz (PSC=%lu, ARR=%lu)\r\n",
    (unsigned long)fs, (unsigned long)psc, (unsigned long)arr);
HAL_UART_Transmit(&huart2, tx_buffer, len, 1000);
}

void HandleRGBCmd(const char *arg) {
    // R, G, B si aparecen en arg
    GPIO_PinState R = (strchr(arg, 'R') ? GPIO_PIN_SET : GPIO_PIN_RESET);
    GPIO_PinState G = (strchr(arg, 'G') ? GPIO_PIN_SET : GPIO_PIN_RESET);
    GPIO_PinState B = (strchr(arg, 'B') ? GPIO_PIN_SET : GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_3, R);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_2, G);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, B);
    // Feedback
    int len = snprintf((char*)tx_buffer, sizeof(tx_buffer),
        "RGB -> R:%c G:%c B:%c\r\n",
        R==GPIO_PIN_SET?'1':'0',
        G==GPIO_PIN_SET?'1':'0',
        B==GPIO_PIN_SET?'1':'0');
    HAL_UART_Transmit(&huart2, tx_buffer, len, 1000);
}

void HandleFFTSIZECmd(const char *arg) {
    char option = arg[0];

    if (option == '1') {
        fft_size = 1024;
    } else if (option == '2') {
        fft_size = 2048;
    } else {

```

```

const char *msg = "Opciones válidas para 'fftSize=' son:\r\n"
    "1 -> 1024 puntos\r\n"
    "2 -> 2048 puntos\r\n";
HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), 1000);
return;
}

// reiniciar DMA con nuevo tamaño
HAL_ADC_Stop_DMA(&hadc1);
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, fft_size);

int len = snprintf((char*)tx_buffer, sizeof(tx_buffer),
    "FFT size set to %u\r\n", fft_size );
HAL_UART_Transmit(&huart2, tx_buffer, len, 1000);
}

void HandleStatusCmd(void) {
    uint32_t psc = htim3.Init.Prescaler;
    uint32_t arr = htim3.Init.Period;
    float fs = 84000000.0f / ((psc + 1) * (arr + 1));
    float bin_res = fs / fft_size;

    int len = snprintf((char*)tx_buffer, sizeof(tx_buffer),
        "Config:\r\nSample TIM3 @ %.2f Hz (PSC=%lu, ARR=%lu)\r\nFFT size: %u\r\nResolucion
espectral: %.2f Hz/bin\r\nCanal ADC: 6\r\nTrigger ADC: TIM3_TRGO\r\n",
        fs, (unsigned long)psc, (unsigned long)arr, fft_size, bin_res);
    HAL_UART_Transmit(&huart2, tx_buffer, len, 1000);
}

void HandlePrintADC(void) {
    static uint8_t continuous_mode = 0;
    char msg[32];

    // Toggle del modo continuo
    if (continuous_mode) {
        continuous_mode = 0;
        return;
    }

    continuous_mode = 1;

    while (continuous_mode) {
        for (int i = 0; i < fft_size; i++) {
            // Enviar valor ADC normalizado (0.0-3.3V) como float
            float voltage = adc_buffer[i] * (3.3f / 4095.0f);
            int len = snprintf(msg, sizeof(msg), "%.4f\r\n", voltage);

            if (HAL_UART_Transmit(&huart2, (uint8_t*)msg, len, 10) != HAL_OK) {
                continuous_mode = 0;
                break;
            }
        }

        // Verificar comando de parada sin bloquear

```



```

        if (rx_index > 0 && strstr(rx_buffer, "print") != NULL) {
            continuous_mode = 0;
            memset(rx_buffer, 0, sizeof(rx_buffer));
            rx_index = 0;
            break;
        }
    }
}

void HandleFreqDisplayCmd(void) {
    char msg[64];
    float suma = 0.0f;
    int count = freq_full ? FREQ_BUFFER_SIZE : freq_index;

    HAL_UART_Transmit(&huart2, (uint8_t*)"Frecuencias IC (Hz):\r\n", 24, 100);

    for (int i = 0; i < count; i++) {
        int idx = (freq_index + i) % FREQ_BUFFER_SIZE;
        suma += freq_buffer[idx];
        int len = snprintf(msg, sizeof(msg), "%.2f\r\n", freq_buffer[idx]);
        HAL_UART_Transmit(&huart2, (uint8_t*)msg, len, 100);
    }

    float promedio = (count > 0) ? suma / count : 0.0f;
    int len = snprintf(msg, sizeof(msg), "Promedio: %.2f Hz\r\n", promedio);
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, len, 100);
}

void HandlePrintFFT(void) {
    float input_f32[FFT_SIZE_MAX];
    float output_fft[FFT_SIZE_MAX];

    // Convertir y normalizar datos ADC
    for (int i = 0; i < fft_size; i++) {
        input_f32[i] = (float)adc_buffer[i] - 2048.0f; // Eliminar offset DC
    }

    // Configurar y calcular FFT
    arm_rfft_fast_instance_f32 S;
    arm_rfft_fast_init_f32(&S, fft_size);
    arm_rfft_fast_f32(&S, input_f32, output_fft, 0);

    // Calcular parámetros de frecuencia
    float fs = 84000000.0f / ((htim3.Init.Prescaler + 1) * (htim3.Init.Period + 1));
    float bin_res = fs / fft_size;

    // Cabecera simple para CoolTerm
    HAL_UART_Transmit(&huart2, (uint8_t*)"FFT_DATA_START\n", 15, 100);

    // Enviar magnitudes positivas

```

```

for (int i = 1; i < fft_size / 2; i++) {
    float real = output_fft[2 * i];
    float imag = output_fft[2 * i + 1];
    float mag = sqrtf(real * real + imag * imag) / (fft_size/2);

    // Convertir a valor positivo absoluto (para histograma)
    float positive_mag = fabsf(mag);

    char msg[32];
    // Formato: "frecuencia,magnitud\n" (sin texto adicional)
    int len = snprintf(msg, sizeof(msg), "%.1f,%.4f\n", i * bin_res, positive_mag);
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, len, 100);
}

// Finalización
HAL_UART_Transmit(&huart2, (uint8_t*)"FFT_DATA_END\n", 13, 100);
}

void HandleFFTInfo(void) {
    float input_f32[FFT_SIZE_MAX];
    float output_fft[FFT_SIZE_MAX];
    float mag_fft[FFT_SIZE_MAX / 2];

    for (int i = 0; i < fft_size; i++) {
        input_f32[i] = (float)adc_buffer[i] - 2048.0f;
    }

    arm_rfft_fast_instance_f32 S;
    arm_rfft_fast_init_f32(&S, fft_size);
    arm_rfft_fast_f32(&S, input_f32, output_fft, 0);

    for (int i = 0; i < fft_size / 2; i++) {
        float real = output_fft[2 * i];
        float imag = output_fft[2 * i + 1];
        mag_fft[i] = sqrtf(real * real + imag * imag);
    }

    uint32_t max_index = 0;
    float max_val = 0.0f;
    arm_max_f32(&mag_fft[1], (fft_size / 2) - 1, &max_val, &max_index);
    max_index += 1; // porque empezamos en bin 1

    float fs = 84000000.0f / ((htim3.Init.Prescaler + 1) * (htim3.Init.Period + 1));
    float freq_bin = fs / fft_size;
    float freq_detected = freq_bin * max_index;

    float sum = 0.0f, sum_sq = 0.0f;
    for (int i = 0; i < fft_size; i++) {
        float x = (float)adc_buffer[i];
        sum += x;
    }
}

```

```

float offset = sum / fft_size;
for (int i = 0; i < fft_size; i++) {
    float x = (float)adc_buffer[i] - offset;
    sum_sq += x * x;
}
float rms = sqrtf(sum_sq / fft_size);

float db_val = 20.0f * log10f(max_val + 1e-6f);

char msg[160];
int len = snprintf(msg, sizeof(msg),
    "Info de FFT:\r\nFrecuencia dominante: %.2f Hz\r\nMagnitud: %.2f dB\r\nOffset: %.2f niveles
ADC\r\nRMS: %.2f niveles ADC\r\n",
    freq_detected, db_val, offset, rms);
HAL_UART_Transmit(&huart2, (uint8_t*)msg, len, 100);
}

void HandleClearCmd(void) {
    const char *clear_screen = "\033[2J\033[H"; // Secuencia ANSI para limpiar la pantalla
    HAL_UART_Transmit(&huart2, (uint8_t*)clear_screen, strlen(clear_screen), 1000);
}

void HandleHelpCmd(void) {
    const char *help_msg =
        "\r\n===== AYUDA =====\r\n"
        "\r\nComandos disponibles:\r\n"
        " led=<ms>@    - Cambia la frecuencia del LED Blinky\r\n"
        " fmuestreo=<1|2|3|4>@ - Frecuencia de muestreo del ADC:\r\n"
        "     - 1 -> 44100 Hz\r\n"
        "     - 2 -> 48000 Hz\r\n"
        "     - 3 -> 96000 Hz\r\n"
        "     - 4 -> 128000 Hz\r\n"
        " rgb=<RGB>@    - Control de LED RGB, ej: rgb=RG\r\n"
        " fftSize=<size>@ - Tama\xC3\xB1o FFT: \r\n"
        "     - 1 -> 1024  \r\n"
        "     - 2 -> 2048  \r\n"
        " status@      - Mostrar configuraci3n actual\r\n"
        " print@       - Imprimir datos crudos del ADC\r\n"
        " fft@         - Imprimir espectro FFT\r\n"
        " info@        - Frecuencia dominante, magnitud y offset\r\n"
        " freq@        - Historial de frecuencia medida por IC\r\n"
        " help @       - Mostrar esta ayuda\r\n"
        "===== \r\n";
    HAL_UART_Transmit(&huart2, (uint8_t*)help_msg, strlen(help_msg), 1000);
}

void imprimirBinario(uint8_t valor) {
    for (int i = 7; i >= 0; i--) {
        printf("%d", (valor >> i) & 1);
    }
}

void mostrarOperacion(const char *desc, uint8_t resultado) {
    printf("%s\n", desc);
}

```

```

printf("Binario: "); imprimirBinario(resultado);
printf("\nHex: 0x%02X\n", resultado);
printf("ASCII: %c\n\n", (resultado >= 32 && resultado <= 126) ? resultado : '.');
}

void verificarOperacionesCrucigrama(void) {
    // 1. Caracter ASCII 'B'
    uint8_t op1 = 'B';
    mostrarOperacion("1. ASCII 'B'", op1);

    // 2. ((!!0xC) << 0b11) | (~0x45)
    uint8_t op2 = ((!!0xC) << 0b11) | (~0x45);
    mostrarOperacion("2. ((!!0xC) << 0b11) | (~0x45)", op2);

    // 3. ((0b00111100 & 0xAC) << 0x3)
    uint8_t op3 = (0b00111100 & 0xAC) << 0x3;
    mostrarOperacion("3. ((0b00111100 & 0xAC) << 0x3)", op3);

    // 4. -121
    uint8_t op4 = (uint8_t)(-121);
    mostrarOperacion("4. -121 (uint8_t)", op4);

    // 5. (0xCAFE & 0xFF0) >> 7
    uint16_t val5 = (0xCAFE & 0xFF0) >> 7;
    uint8_t op5 = (uint8_t)(val5 & 0xFF);
    mostrarOperacion("5. (0xCAFE & 0xFF0) >> 7", op5);

    // 6. (1 << 4) + (0x3 << 4) - 10 + (0b11 << 0x6)
    uint8_t op6 = (1 << 4) + (0x3 << 4) - 10 + (0b11 << 6);
    mostrarOperacion("6. (1<<4)+(0x3<<4)-10+(0b11<<6)", op6);

    // 7. ('T' | '5')
    uint8_t op7 = 'T' | '5';
    mostrarOperacion("7. ('T' | '5')", op7);

    // 8. (2921 & 0xD6)
    uint8_t op8 = 2921 & 0xD6;
    mostrarOperacion("8. (2921 & 0xD6)", op8);

    // 9. 01100110 ^ 00101011
    uint8_t a9 = 0b01100110;
    uint8_t b9 = 0b00101011;
    uint8_t op9 = a9 ^ b9;
    mostrarOperacion("9. 01100110 ^ 00101011", op9);
}

/* USER CODE END 0 */

/**
 * @brief The application entry point.
 * @retval int
 */

```

```

int main(void)
{

    /* USER CODE BEGIN 1 */

    /* USER CODE END 1 */

    /* MCU Configuration-----*/

    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
    HAL_Init();

    /* USER CODE BEGIN Init */
    CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk; // Habilitar DWT
    DWT->CYCCNT = 0;                               // Reiniciar contador
    DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;           // Encender contador de ciclos
    /* USER CODE END Init */

    /* Configure the system clock */
    SystemClock_Config();

    /* USER CODE BEGIN SysInit */

    /* USER CODE END SysInit */

    /* Initialize all configured peripherals */

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART2_UART_Init();
    MX_TIM2_Init();
    MX_TIM4_Init();
    MX_ADC1_Init();
    MX_TIM3_Init();
    /* USER CODE BEGIN 2 */
    HAL_TIM_Base_Start_IT(&htim2);
    HAL_TIM_Base_Start_IT(&htim3);
    HAL_TIM_Base_Start_IT(&htim4);
    HAL_UART_Receive_IT(&huart2,&rx_char,1);
    HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, fft_size);
    HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_2);
    HandleSampleFreqCmd(3); //96000 hz muestreo
    HandleFFTSizeCmd(1); // 1-> 1024

    verificarOperacionesCrucigrama();
    medirTiempoEnvioUART();
    HandleHelpCmd();

    /* USER CODE END 2 */

    /* Infinite loop */

```

```

/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    if (pending_event != IDLE){
        state_machine_action(pending_event);
        pending_event = IDLE;
    }
}
/* USER CODE END 3 */
}

/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
    */
    __HAL_RCC_PWR_CLK_ENABLE();
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /** Initializes the RCC Oscillators according to the specified parameters
    * in the RCC_OscInitTypeDef structure.
    */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLM = 16;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV4;
    RCC_OscInitStruct.PLL.PLLQ = 4;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
    */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
        |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSClkSource = RCC_SYSCCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;

```

```

RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
{
    Error_Handler();
}
}

/**
 * @brief ADC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_ADC1_Init(void)
{

    /* USER CODE BEGIN ADC1_Init 0 */

    /* USER CODE END ADC1_Init 0 */

    ADC_ChannelConfTypeDef sConfig = {0};

    /* USER CODE BEGIN ADC1_Init 1 */

    /* USER CODE END ADC1_Init 1 */

    /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of
    conversion)
    */
    hadc1.Instance = ADC1;
    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
    hadc1.Init.ScanConvMode = DISABLE;
    hadc1.Init.ContinuousConvMode = ENABLE;
    hadc1.Init.DiscontinuousConvMode = DISABLE;
    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
    hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T3_TRGO;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.NbrOfConversion = 1;
    hadc1.Init.DMAContinuousRequests = ENABLE;
    hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    if (HAL_ADC_Init(&hadc1) != HAL_OK)
    {
        Error_Handler();
    }

    /** Configure for the selected ADC regular channel its corresponding rank in the sequencer and its
    sample time.
    */
    sConfig.Channel = ADC_CHANNEL_6;

```

```

sConfig.Rank = 1;
sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN ADC1_Init 2 */

/* USER CODE END ADC1_Init 2 */

}

/**
 * @brief TIM2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM2_Init(void)
{

    /* USER CODE BEGIN TIM2_Init 0 */

    /* USER CODE END TIM2_Init 0 */

    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM2_Init 1 */

    /* USER CODE END TIM2_Init 1 */
    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 42000-1;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 500;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
    if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
    {
        Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
    {
        Error_Handler();
    }
}

```



```

/* USER CODE BEGIN TIM2_Init 2 */

/* USER CODE END TIM2_Init 2 */

}

/**
 * @brief TIM3 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM3_Init(void)
{

/* USER CODE BEGIN TIM3_Init 0 */

/* USER CODE END TIM3_Init 0 */

TIM_ClockConfigTypeDef sClockSourceConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};
TIM_IC_InitTypeDef sConfigIC = {0};

/* USER CODE BEGIN TIM3_Init 1 */

/* USER CODE END TIM3_Init 1 */
htim3.Instance = TIM3;
htim3.Init.Prescaler = 0;
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = 433;
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
{
    Error_Handler();
}
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
{
    Error_Handler();
}
if (HAL_TIM_IC_Init(&htim3) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;

```

```

sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
sConfigIC.ICFilter = 8;
if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM3_Init 2 */
HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_2);
/* USER CODE END TIM3_Init 2 */

}

/**
 * @brief TIM4 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM4_Init(void)
{

    /* USER CODE BEGIN TIM4_Init 0 */

    /* USER CODE END TIM4_Init 0 */

    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};

    /* USER CODE BEGIN TIM4_Init 1 */

    /* USER CODE END TIM4_Init 1 */
    htim4.Instance = TIM4;
    htim4.Init.Prescaler = 840-1;
    htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim4.Init.Period = 200-1;
    htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim4) != HAL_OK)
    {
        Error_Handler();
    }
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim4, &sClockSourceConfig) != HAL_OK)
    {
        Error_Handler();
    }
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
    {
        Error_Handler();
    }

```

```

}
/* USER CODE BEGIN TIM4_Init 2 */

/* USER CODE END TIM4_Init 2 */

}

/**
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init(void)
{

/* USER CODE BEGIN USART2_Init 0 */

/* USER CODE END USART2_Init 0 */

/* USER CODE BEGIN USART2_Init 1 */

/* USER CODE END USART2_Init 1 */
huart2.Instance = USART2;
huart2.Init.BaudRate = 115200;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.OverSampling = UART_OVERSAMPLING_16;
if (HAL_UART_Init(&huart2) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN USART2_Init 2 */

/* USER CODE END USART2_Init 2 */

}

/**
 * Enable DMA controller clock
 */
static void MX_DMA_Init(void)
{

/* DMA controller clock enable */
__HAL_RCC_DMA2_CLK_ENABLE();
__HAL_RCC_DMA1_CLK_ENABLE();

/* DMA interrupt init */

```

```

/* DMA1_Stream5_IRQn interrupt configuration */
HAL_NVIC_SetPriority(DMA1_Stream5_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(DMA1_Stream5_IRQn);
/* DMA2_Stream0_IRQn interrupt configuration */
HAL_NVIC_SetPriority(DMA2_Stream0_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(DMA2_Stream0_IRQn);

}

/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* USER CODE BEGIN MX_GPIO_Init_1 */
    /* USER CODE END MX_GPIO_Init_1 */

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOC, LedB_Pin|segD_Pin|segE_Pin|LedR_Pin
        |LedG_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(userLed_GPIO_Port, userLed_Pin, GPIO_PIN_SET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOA, dis1_Pin|dis2_Pin|dis4_Pin|segA_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOB, dis3_Pin|segF_Pin|segB_Pin|segC_Pin
        |segG_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pins : LedB_Pin segD_Pin segE_Pin LedR_Pin
        LedG_Pin */
    GPIO_InitStruct.Pin = LedB_Pin|segD_Pin|segE_Pin|LedR_Pin
        |LedG_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    /*Configure GPIO pin : userLed_Pin */
    GPIO_InitStruct.Pin = userLed_Pin;

```

```

GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(userLed_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pins : dis1_Pin dis2_Pin dis4_Pin segA_Pin */
GPIO_InitStruct.Pin = dis1_Pin | dis2_Pin | dis4_Pin | segA_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/*Configure GPIO pins : dis3_Pin segF_Pin segB_Pin segC_Pin
                        segG_Pin */
GPIO_InitStruct.Pin = dis3_Pin | segF_Pin | segB_Pin | segC_Pin
                        | segG_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

/*Configure GPIO pin : userData_Pin */
GPIO_InitStruct.Pin = userData_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(userData_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : userSw_Pin */
GPIO_InitStruct.Pin = userSw_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(userSw_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : userClk_Pin */
GPIO_InitStruct.Pin = userClk_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(userClk_GPIO_Port, &GPIO_InitStruct);

/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);

HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

```

```

void medirTiempoEnvioUART(void) {
    const char *msg = "Hola mundo! - Examen Taller V\n";
    uint32_t start = DWT->CYCCNT;

    printf("%s", msg);

    uint32_t elapsed = DWT->CYCCNT - start;
    printf("Ciclos para enviar mensaje UART: %lu\n", (unsigned long)elapsed);
}

int __io_putchar(int ch) {
    HAL_UART_Transmit(&huart2, (uint8_t*)&ch, 1, HAL_MAX_DELAY);
    return ch;
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if(htim->Instance==TIM2){
        HAL_GPIO_TogglePin(userLed_GPIO_Port,userLed_Pin);
    }else if(htim->Instance==TIM4){
        if (pending_event == IDLE)
            pending_event = EVENT_TIMER_TICK;
    }else if (htim->Instance==TIM3){

        counterOverflow++;
    }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == userClk_Pin) {
        data_snapshot = HAL_GPIO_ReadPin(userData_GPIO_Port, userData_Pin);
        if (pending_event == IDLE)
            pending_event = EVENT_ENCODER;
    } else if (GPIO_Pin == userSw_Pin) {
        if (pending_event == IDLE)
            pending_event = EVENT_SW;
    }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
    if(huart->Instance == USART2){
        if(rx_index<sizeof(rx_buffer)-1){
            rx_buffer[rx_index++]=rx_char;
        }
        if (rx_char == '@') {
            rx_buffer[rx_index]='\0';
            rx_index=0;
            pending_event = EVENT_USART; //aquí levanta tu bandera
        }
        HAL_UART_Receive_IT(huart, &rx_char,1);
    }
}

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM3 && htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2) {
        static uint8_t ready = 0;

```

```

if (ready == 0) {
    firstCapture = TIM3->CCR2;
    counterOverflow = 0;
    ready = 1;
} else {
    secondCapture = TIM3->CCR2;

    if (secondCapture >= firstCapture) {
        elapsedTicks = secondCapture - firstCapture + (counterOverflow * 65536);
    } else {
        elapsedTicks = (0xFFFF - firstCapture + secondCapture + 1) + (counterOverflow * 65536);
    }

    if (elapsedTicks > 0 && pending_event == IDLE) {
        pending_event = EVENT_IC_CAPTURE;
    }

    ready = 0; // volver a medir siguiente ciclo
}
}
}
/* USER CODE END 4 */

/**
 * @brief This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

#ifdef USE_FULL_ASSERT
/**
 * @brief Reports the name of the source file and the source line number
 *        where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,

```

```
ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
/* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

He revisado tu código completo y puedo confirmar lo siguiente respecto a la **configuración actual del ADC con DMA**:

Estado actual de la configuración ADC con DMA

- **ADC usado:** ADC1
- **Canal único configurado:** ADC_CHANNEL_6 (probablemente conectado a uno de los ejes del joystick)
- **Modo de escaneo:** DISABLE (solo un canal se está leyendo)
- **Conversión continua:** ENABLE
- **Trigger externo:** TIM3_TRG0
- **DMA:** Habilitado con `HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, fft_size);`
- **Buffer DMA:** `adc_buffer[FFT_SIZE_MAX]`

Objetivo: Leer ambos ejes del joystick (VRx y VRy) usando DMA

Para eso, necesitas configurar el ADC1 con **Scan mode habilitado** y `NbrOfConversion = 2` para leer dos canales (por ejemplo, ADC_CHANNEL_6 y ADC_CHANNEL_7, dependiendo de tus pines). Aquí te muestro cómo modificarlo.

Pasos para configurar ADC1 con dos canales (VRx y VRy) usando DMA

1. Modificar la inicialización del ADC1 en `MX_ADC1_Init()`:

```
c
hadc1.Init.ScanConvMode = ENABLE; // Escaneo habilitado
hadc1.Init.NbrOfConversion = 2; // Dos conversiones (VRx y VRy)
hadc1.Init.ContinuousConvMode = ENABLE; // 0 usar modo discontinuo según necesidad
```

2. Configurar ambos canales (ejemplo usando canales 6 y 7):

```
c
// VRx -> ADC_CHANNEL_6
sConfig.Channel = ADC_CHANNEL_6;
sConfig.Rank = 1;
sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);

// VRy -> ADC_CHANNEL_7
sConfig.Channel = ADC_CHANNEL_7;
sConfig.Rank = 2;
sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);
```


3. Modificar el buffer DMA:

Si estás leyendo 1024 muestras por canal, el buffer debe ser:

```
c
uint16_t adc_buffer[2 * FFT_SIZE_MAX]; // Doble tamaño si haces FFT de ambos ejes o un canal por separado
```

4. Llamar al DMA correctamente:

```
c
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, 2 * fft_size);
```

Acceso a los datos

Después de la transferencia DMA, los datos quedan intercalados así:

```
c
VRx: adc_buffer[0], adc_buffer[2], adc_buffer[4], ...
VRy: adc_buffer[1], adc_buffer[3], adc_buffer[5], ...
```

Recomendación

- Si solo quieres usar los valores actuales de VRx y VRy (no FFT), puedes usar un buffer de tamaño 2 (DMA en modo circular), y simplemente hacer:

```
c
uint16_t adc_joystick[2];
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_joystick, 2);
```

Y luego leer:

```
c
uint16_t VRx = adc_joystick[0];
uint16_t VRy = adc_joystick[1];
```

¿Deseas que te prepare ese código ya listo para reemplazar el `MX_ADC1_Init()` y cómo iniciar el DMA para VRx y VRy con buffer de 2 muestras?



quiero que VRx y VRy con tengan buffer diferentes ademas puedo hacer que el led rgb funcione con señal pwm ? para lograr diferentes tonos con los mismos pines o debo cambiar el pin

Perfecto. Vamos por partes:

✓ Parte 1: Configurar VRx y VRy con buffers DMA diferentes

En STM32 **no puedes usar dos buffers independientes en un mismo ADC con DMA directamente**, ya que el ADC entrega los datos de los canales de forma **intercalada** (por ejemplo, [VRx0, VRy0, VRx1, VRy1, ...]).

Sin embargo, sí puedes hacer lo siguiente:

✓ Solución: usar un solo buffer intercalado y separar los datos manualmente

1. Configura el ADC1 con escaneo de 2 canales (como se explicó antes).
2. Usa un solo buffer compartido:

```
c
#define NUM_MUESTRAS 100
uint16_t adc_buffer[2 * NUM_MUESTRAS]; // Intercalado: VRx, VRy, VRx, VRy...
uint16_t vrx_buffer[NUM_MUESTRAS];
uint16_t vry_buffer[NUM_MUESTRAS];
```

3. Luego, en tu función de procesamiento (por ejemplo, en una interrupción o tarea periódica), haces la separación:

```
c
for (int i = 0; i < NUM_MUESTRAS; i++) {
    vrx_buffer[i] = adc_buffer[2 * i]; // VRx
    vry_buffer[i] = adc_buffer[2 * i + 1]; // VRy
}
```

👉 Esto te da dos buffers separados para procesar de forma independiente cada eje del joystick.

✓ Parte 2: ¿Puedes usar PWM en los mismos pines del LED RGB actual?

🔍 Revisión de tu código actual

En tu función `HandleRGBCmd()` estás controlando el LED RGB así:

```
c
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_3, R); // Rojo
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_2, G); // Verde
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, B); // Azul
```

Estos pines están configurados como salidas digitales simples (`GPIO_MODE_OUTPUT_PP`).

⚠ Problema:

Para generar tonos RGB con mayor precisión, necesitas **PWM hardware** (por temporizador). Esto **requiere que los pines estén conectados a canales PWM (Timer x, Channel y)**.

¿Puedo seguir usando los mismos pines?

Depende del microcontrolador STM32 exacto que estés usando y del mapeo de funciones alternas (AF).

Por ejemplo, en un STM32F4:

- **PC13** → No soporta salida de PWM
- **PC2 y PC3** → Sí pueden tener funciones alternativas, pero no siempre para PWM

🔧 **Solución más robusta:** cambia los pines a unos que estén conectados a canales PWM (por ejemplo: PA8, PB0, PB1, etc., según tu modelo STM32).

✓ Recomendación final

- ✓ **Sí puedes usar PWM para el LED RGB**, pero necesitas **reubicar los pines** a salidas PWM reales.
- ✓ Para **usar VRx y VRy con DMA separados**, usas un único buffer intercalado y separas los valores en código.
- ✗ **No puedes usar dos buffers DMA separados directamente** con un solo ADC.

¿Quieres que te dé el código completo para:

- Cambiar los pines del LED RGB a PWM?
- Usar TIMx con HAL_TIM_PWM_Start()?
- Y el ejemplo de inicialización de ADC con dos canales para joystick?

Solo dime qué modelo exacto de STM32 estás usando (por ejemplo STM32F103C8, STM32F411, STM32F407, etc.) para darte los pines correctos.



```
este algoritmo como sabe que si los pares son x o Y? y no los intercala ? for (int i = 0; i <
NUM_MUESTRAS; i++) {
    vrx_buffer[i] = adc_buffer[2 * i];    // VRx
    vry_buffer[i] = adc_buffer[2 * i + 1]; // VRy
} ademas se usa el dma ?
```

Excelente pregunta. Vamos a responderla con precisión técnica y clara:

🔍 **Pregunta 1: ¿Cómo sabe el algoritmo que `adc_buffer[2*i]` es VRx y `adc_buffer[2*i+1]` es VRy?**

Esto **no es por arte de magia**: es porque el STM32 convierte los canales del ADC en el **orden en que tú los configuras en el escaneo (Scan Mode)**.

⚙️ Ejemplo concreto:

Supongamos que en la inicialización de `MX_ADC1_Init()` tú haces esto:

```
c
sConfig.Channel = ADC_CHANNEL_6; // VRx
sConfig.Rank = 1;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);

sConfig.Channel = ADC_CHANNEL_7; // VRy
sConfig.Rank = 2;
HAL_ADC_ConfigChannel(&hadc1, &sConfig);
```

👉 Con esto, el ADC hace lo siguiente en cada ciclo de muestreo:

1. Convierte el canal **6 (VRx)** → se guarda primero en el buffer DMA
2. Convierte el canal **7 (VRy)** → se guarda después

Es decir, el orden de los datos que DMA guarda en el buffer es:

```
adc_buffer[] = [VRx0, VRy0, VRx1, VRy1, VRx2, VRy2, ...]
```

Por eso, en tu código:

```
c
vr_x_buffer[i] = adc_buffer[2*i]; // posiciones 0,2,4,...
vr_y_buffer[i] = adc_buffer[2*i + 1]; // posiciones 1,3,5,...
```

Funciona correctamente, **siempre y cuando respetes el orden de Rank 1 y Rank 2** al configurar los canales del ADC.

🔄 Pregunta 2: ¿Se usa DMA?

✅ Sí, esta técnica usa DMA completamente.

Cuando haces:

```
c
HAL_ADC_Start_DMA(&hadc1, (uint32_t*)adc_buffer, 2 * NUM_MUESTRAS);
```

- STM32 comienza a hacer conversiones en modo escaneo (con los canales definidos)
- Por cada ciclo, **DMA copia automáticamente los datos al buffer `adc_buffer[]`**
- El procesador **no hace polling ni interrupciones**: todo es manejado por DMA

⚠️ Pero el buffer `vr_x_buffer[]` y `vry_buffer[]` los llenas tú manualmente, después de que el DMA haya llenado `adc_buffer[]`. Es decir, no hay DMA doble, tú haces la separación.

📌 Resumen técnico:

Pregunta	Respuesta
¿Cómo se sabe que los pares/impares son VRx/VRy?	Por el orden de los Rank en la configuración del ADC
¿Se usa DMA?	Sí. <code>adc_buffer[]</code> es llenado automáticamente por DMA
¿Por qué separo los valores manualmente?	Porque el DMA pone los datos intercalados y el ADC no crea buffers separados por canal

¿Quieres que te genere una sección de código `.c` y `.h` completa para esta separación con interrupción de fin de conversión o usando un flag?



```
pasa raro en el .ioc activo el adc1 canala 7 y al generar el codigo no aparece sConfig.Channel =
ADC_CHANNEL_7; // VRy
sConfig.Rank = 2;
HAL_ADC_ConfigChannel(&hadc1, &sConfig); debería agregarla manualmente o que puede estar
pasando ?
```

Eso que estás observando es un comportamiento común en STM32CubeMX/STM32CubeIDE, y te explico exactamente **qué está pasando** y cómo resolverlo correctamente:

🔍 ¿Por qué no aparece la línea `sConfig.Channel = ADC_CHANNEL_7;`?

Cuando usas CubeMX (a través del archivo `.ioc`) para configurar varios canales en un ADC **y no activas correctamente el modo de escaneo o no defines explícitamente el número de conversiones**, el generador de código **solo crea la configuración del primer canal**.

✅ Verifica estos puntos en STM32CubeMX (archivo `.ioc`):


1. ADC1 → Parameter Settings:

- ✅ **Scan Conversion Mode:** Debe estar en **Enabled**
- ✅ **Number of Conversion:** Debe ser 2 (si quieres VRx y VRy)

2. ADC1 → Configuration (en la pestaña de la derecha):

- Asegúrate de tener:
 - Rank 1: Channel 6 (VRx)
 - Rank 2: Channel 7 (VRy)

3. **Trigger externo (opcional):** Si usas un temporizador para disparar las conversiones, configúralo como `External Trigger Conversion Source`.


 **Si solo uno de los canales está configurado o el Scan Mode está en Disable, entonces el código generado será solo con un canal (Rank 1).**

¿Debes agregar la configuración del canal 7 manualmente?

Sí, puedes agregarla manualmente si el CubeMX no lo hace, pero es mejor corregirlo en el archivo `.ioc` para que quede persistente al regenerar el código.




Si decides hacerlo manualmente, agrégalo así justo después del canal 6:

```
c
sConfig.Channel = ADC_CHANNEL_7;
sConfig.Rank = 2;
sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
{
    Error_Handler();
}
```

 **Ten cuidado:** si regeneras el código con CubeMX, podrías perder esta línea si no corrigen el `.ioc` primero.

Recomendación final

Haz esto:

1. Abre tu proyecto en STM32CubeMX.
2. Ve a la configuración del ADC1.
3. Asegúrate de:
 -  Scan Conversion Mode: Enabled
 -  Number of Conversion: 2
 -  Que tengas dos canales configurados en el orden deseado
4. Guarda y vuelve a generar el código.

Con eso, CubeMX debería generar ambos `HAL_ADC_ConfigChannel()` automáticamente.




¿Deseas que te revise también el archivo `.ioc` si me lo pasas, o que te diga cómo hacer esto paso a paso en CubeMX con imágenes?



 Imagen cargada

que hace el modo continuo como debería configurar ?

Gracias por compartir la imagen. Veo que tienes activado lo siguiente:

-  **Scan Conversion Mode: Enabled**
-  **Continuous Conversion Mode: Enabled**
-  **DMA Continuous Requests: Enabled**

Vamos a explicarlo bien para que entiendas **cómo afecta eso al funcionamiento** y qué debes hacer para muestrear correctamente **dos canales (VRx y VRy)** usando **DMA sincronizado con temporizador (TIM3)**.

¿Qué hace el Continuous Conversion Mode?


Modo continuo (Enabled):

Cuando está habilitado:

- El ADC comienza a convertir los canales **una y otra vez sin parar**.
- Se repite la secuencia de canales automáticamente.
- Ideal si quieres una "muestra continua", pero **no sincronizada con un trigger externo (como un timer)**.



Problema:

Tú configuraste el ADC con **trigger externo TIM3_TRGO** (`hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T3_TRGO`), lo que **no tiene sentido si también activas el modo continuo**.

 **Conflicto:** Si tienes **Continuous Mode = Enabled** y **External Trigger**, se ignora el trigger y el ADC corre solo.

¿Cómo deberías configurar para usar TIM3 como trigger?

Configuración correcta para escaneo de 2 canales con DMA sincronizado al timer:

Parámetro	Valor correcto
Scan Conversion Mode	Enabled
Continuous Conversion	 Disabled
External Trigger Source	TIM3_TRGO
DMA Continuous Requests	 Enabled
Nbr of Conversions	2
Channel Rank 1	ADC_CHANNEL_6 (VRx)
Channel Rank 2	ADC_CHANNEL_7 (VRy)

✓ ¿Por qué esto es mejor?

- Cada vez que TIM3 genera un evento TRGO (por ejemplo a 96 kHz), se dispara una secuencia de 2 conversiones (VRx y VRy).
- DMA transfiere ambos datos en orden.
- No se pierde sincronización con el timer.
- Tú puedes tener control preciso del muestreo (útil si haces FFT o control basado en tiempo).

📌 ¿Qué debes hacer en CubeMX?

1. **Desactiva Continuous Conversion Mode** → ponlo en **Disabled**.
2. Asegúrate de que **Nbr of Conversion = 2**.
3. Confirma que tienes configurado:
 - **Channel 6** → Rank 1
 - **Channel 7** → Rank 2
4. **Verifica que TIM3 esté configurado como Trigger Output (TRGO)** para sincronizar.

¿Quieres que te genere un ejemplo visual o el .ioc corregido como archivo si me lo proporcionas?