ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KỸ THUẬT MÁY TÍNH

ĐỒ ÁN MÔN HỌC VI XỬ LÝ – VI ĐIỀU KHIỂN

Đề tài:

ĐIỀU KHIỂN DẢI LED RGB VỚI HIỆU ỨNG "LIGHT SHOW"

Sinh viên thực hiện :

VŨ ĐẠI DƯƠNG – 23520359

NGUYỄN TRỌNG BẢO DUY – 23520379

NGUYỄN ĐẶNG PHƯƠNG DUY - 23520372

PHAN THÀNH DUY - 23520386

Giảng viên hướng dẫn : TRẦN NGỌC ĐỨC

MỤC LỤC

I. THÀNH VIÊN NHÓM

1. Giới thiệu thành viên

Họ và tên	MSSV	Chức vụ
Nguyễn Đặng Phương Duy	23520372	Nhóm trưởng
Nguyễn Trọng Bảo Duy	23520379	Thành viên
Phan Thành Duy	23520386	Thành viên
Vũ Đại Dương	23520359	Thành viên

2. Phân công công việc

Họ và tên	Công việc				
	DMA + PWM (Timer 1)	**Timer trigger (Timer 2) + ADC**	**FFT**	**Hiệu ứng LED**	**Tổng hợp và viết báo cáo**
Nguyễn Đặng Phương Duy	**x**		**x**	**x**	
Nguyễn Trọng Bảo Duy		**x**	**x**		
Phan Thành Duy					**x**
Vũ Đại Dương	**x**			**x**	

II. GIỚI THIỆU CHUNG

1. Mục tiêu của đề tài

Sử dụng vi điều khiển **STM32** để điều khiển dải LED **WS2812B** thể hiện hiệu ứng ánh sáng đồng bộ với tín hiệu âm thanh đầu vào.

Khi âm thanh thay đổi (như nhịp bass mạnh hay yếu), hệ thống thay đổi hiệu ứng đèn (như tăng độ sáng, thay đổi màu sắc hoặc tốc độ hiệu ứng) nhằm tạo hiệu ứng ánh sáng "nhảy nhót theo nhạc".

Đây cũng là một ứng dụng minh hoạ cho việc sử dụng STM32 với ADC, DMA,PWM để làm dự án thị giác âm thanh.

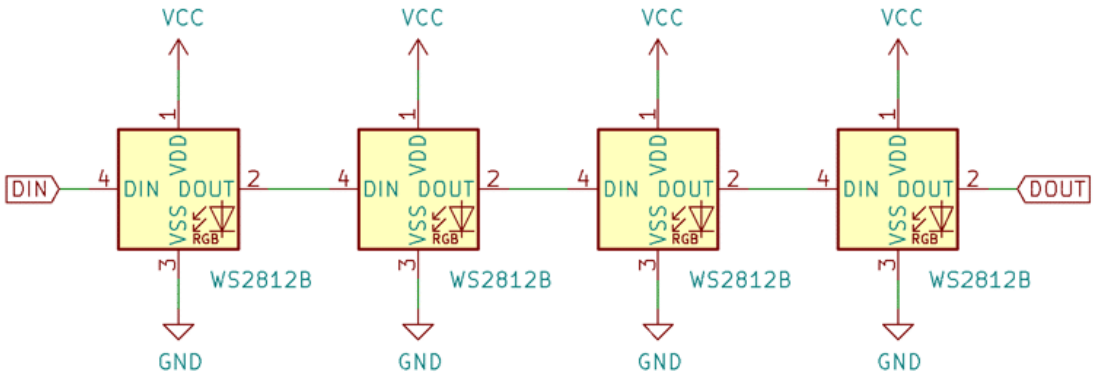
2. Mô tả chung

Hệ thống gồm ba thành phần chính: **STM32F407VET6**, dải LED **WS2812B** (54 đèn), và module micr **MAX9814,** với mục tiêu tạo ra các hiệu ứng ánh sáng sống động như rainbow wave, gradient… và có thể đồng bộ với nhạc thông qua module micro MAX9814. Dự án tận dụng nhiều ngoại vi phần cứng của vi điều khiển STM32 để đạt được hiệu quả cao, đồng thời giảm tải xử lý cho CPU nhờ vào sự hỗ trợ của DMA.

III. CƠ SỞ LÝ THUYẾT VÀ LINH KIỆN SỬ DỤNG

1. Về LED WS2812B

Led RGB **WS2812B** linh hoạt, dễ sử dụng và có thể điều khiển riêng biệt. Các đèn LED này có IC điều khiển tích hợp cho phép điều khiển màu sắc và độ sáng của từng LED một cách độc lập. Mỗi đèn LED có chân VCC , GND, DIN và DOUT độc lập. Các chân VCC và GND là chân chung cho tất cả các đèn LED, trong đó DIN của đèn LED đầu tiên được kết nối với nguồn tín hiệu, có thể là một bộ vi điều khiển. DOUT của đèn LED đầu tiên được kết nối với DIN của đèn LED thứ hai, v.v., như trong sơ đồ bên dưới.

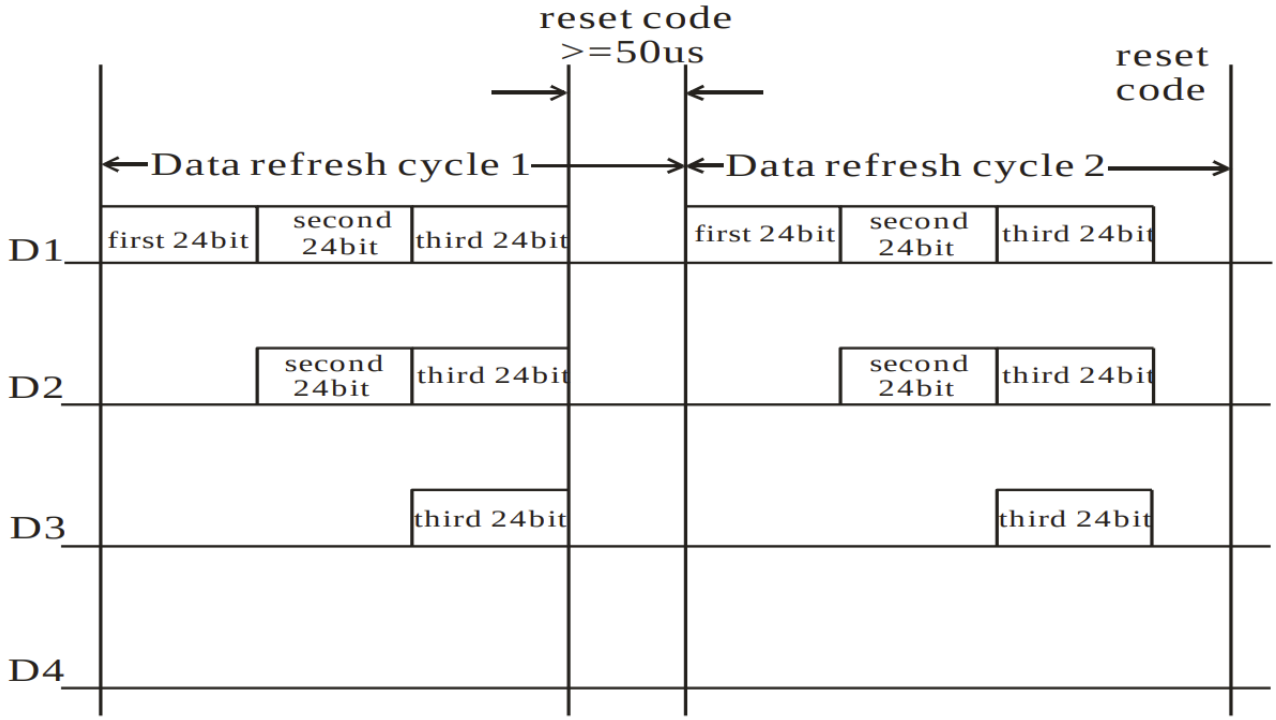


WS2812B sử dụng bộ điều chế độ rộng xung (PWM) để phân biệt giữa logic 0 và 1. Logic 1 yêu cầu độ rộng xung dài hơn, trong khi đó logic 0 yêu cầu độ rộng xung ngắn hơn. Tổng độ rộng xung là 1,25 µs, đồng nghĩa với tần số là 800kHz, với các chu kỳ nhiệm vụ (duty cycles) tương ứng với logic 0 và 1 là 36% và 64%.

Dung sai của mỗi độ rộng xung là ± 150ns. Xung reset phải là 50 ms hoặc lâu hơn trước khi dữ liệu tiếp theo được đưa đến đèn LED.

Các đèn LED phải được gửi tín hiệu theo thứ tự - 24 bit đầu tiên đến đèn LED thứ nhất, 24 bit thứ hai đến đèn LED thứ hai, v.v., cho đến khi tất cả các đèn LED trong chuỗi đều được gửi tín hiệu.

Sau khi tập hợp dữ liệu đầu tiên được gửi, phải có một xung reset giữ từ 50ms trở lên để đèn LED đạt thời gian ổn định và sau đó tập dữ liệu thứ hai có thể được gửi.



2. Về STM32F4

STM32 là dòng vi điều khiển 32-bit dựa trên kiến trúc ARM Cortex-M của hãng STMicroelectronics. STM32 được chia thành nhiều dòng: STM32F0, F1, F3, F4, F7, H7, G0, G4,… trong đó dòng **STM32F4** là dòng trung – cao cấp, cân bằng giữa hiệu năng và giá thành, rất phổ biến trong nhiều ứng dụng thực tế.

Trong các ứng dụng chiếu sáng nghệ thuật, trình diễn ánh sáng, hoặc hiển thị hiệu ứng, LED RGB WS2812B (NeoPixel) là loại LED phổ biến nhờ khả năng hiển thị màu linh hoạt và điều khiển độc lập từng LED thông qua một dây tín hiệu duy nhất. Tuy nhiên, WS2812B yêu cầu giao tiếp dữ liệu rất chính xác về mặt thời gian, khiến việc điều khiển nó trở nên thách thức đối với các vi điều khiển thông thường.

Vi điều khiển **STM32F4** (trong đề tài dùng **STM32F407VET6**) thuộc dòng ARM Cortex-M4 là một trong những lựa chọn tối ưu để thực hiện nhiệm vụ này vì :

a. Tốc độ xử lý mạnh mẽ

STM32F4 có xung nhịp cao (72–180 MHz), đảm bảo có đủ chu kỳ lệnh để xử lý và tạo xung chính xác, đặc biệt khi sử dụng kỹ thuật bit-banging hoặc điều khiển thời gian bằng Timer.

b. Hỗ trợ DMA và PWM/SPI để "giả lập" giao thức WS2812B

Nhờ DMA, ta có thể cấu hình Timer hoặc SPI để tự động gửi dữ liệu tạo ra các xung theo đúng chuẩn thời gian WS2812B. Điều này **giảm tải cho CPU**, giúp hệ thống hoạt động ổn định và tiết kiệm tài nguyên.

c. Dung lượng RAM và Flash lớn

Khi điều khiển hàng trăm LED (mỗi LED cần 3 byte dữ liệu), việc lưu trữ toàn bộ dữ liệu cần RAM lớn. STM32F4 cung cấp bộ nhớ đủ để điều khiển cả ma trận LED cỡ lớn.

d. Xử lý tín hiệu số (DSP, FPU)

Khi cần tạo hiệu ứng phức tạp hoặc xử lý tín hiệu âm thanh đồng bộ theo nhạc (music visualization), STM32F4 có thể thực hiện các thuật toán như FFT (biến đổi Fourier nhanh) nhờ các khối xử lý DSP và FPU tích hợp.

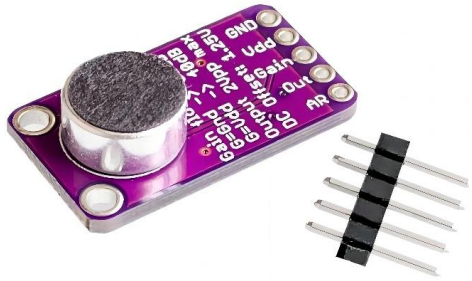
e. Giao tiếp ngoại vi đa dạng

Cho phép dễ dàng kết nối với các cảm biến âm thanh, nút nhấn, màn hình OLED, module giao tiếp không dây để điều khiển hiệu ứng.

Như vậy**,** với tốc độ xử lý cao, khả năng sử dụng DMA/SPI/PWM để tạo xung chuẩn xác, hỗ trợ các thuật toán hiệu ứng phức tạp và tài nguyên phần cứng dồi dào, **STM32F4 là nền tảng rất phù hợp để điều khiển dải LED WS2812B**

3. Về MAX9814

MAX9814 là một IC khuếch đại micro (microphone amplifier module) tích hợp sẵn chức năng Tự động Điều Chỉnh Độ Khuếch Đại (AGC – Automatic Gain Control) và bộ lọc dải thông (band-pass filter) để giảm nhiễu tần số rất thấp và rất cao.



Mạch thường kèm sẵn micro electret, các tụ và điện trở, chỉ cần cấp nguồn (2.7 V–5.5 V), xuất ra tín hiệu analog đã được khuếch đại và lọc.

❖ **Vì sao chọn MAX9814 cho đề tài ?**

Ưu điểm	**Vai trò trong dự án**
AGC tự động	Giữ tín hiệu ổn định để ADC không bị méo hoặc quá yếu
Bộ lọc dải thông tích hợp	Loại nhiễu không mong muốn, tăng độ chính xác cho FFT
Phản ứng nhanh	Theo kịp biến đổi cường độ âm thanh theo beat nhạc
Gọn nhẹ, dễ kết nối	Phù hợp với mạch nhúng STM32 và yêu cầu về không gian

Kết luận: MAX9814 là lựa chọn phù hợp nhất cho đề tài "điều khiển LED theo âm thanh" nhờ AGC tích hợp, bộ lọc giúp loại bỏ nhiễu, và đáp ứng nhanh.

4. Các mô-đun ngoại vi STM32F407VET6 được sử dụng

a) **Timer1** (TIM1) – Phát xung PWM điều khiển WS2812B

Dải LED WS2812B yêu cầu tín hiệu điều khiển rất chính xác với tần số khoảng 800 kHz, mà mỗi bit dữ liệu được mã hóa bằng độ rộng xung PWM khác nhau (high time dài hoặc ngắn). Để đạt

được yêu cầu này mà không dùng bit-banging (tốn CPU), Timer1 được cấu hình ở chế độ PWM và kết hợp với DMA để phát tín hiệu dữ liệu LED. Cụ thể:

- TIM1 được cấu hình để tạo ra các xung PWM có độ rộng tương ứng với từng bit 0/1, trong đó mỗi chu kỳ bit kéo dài 1.25 µs tương ứng với tần số cập nhật 800 kHz theo chuẩn WS2812B.
- DMA được sử dụng để truyền các giá trị duty cycle tương ứng với từng bit 0/1 của dữ liệu màu từng LED.
- Khi DMA hoàn tất, ta biết quá trình cập nhật LED đã hoàn tất và có thể chuyển sang frame kế tiếp.

b) **Timer2** (TIM2) – (General Purpose Timer) bộ định thời hệ thống

TIM2 được cấu hình ở chế độ periodic interrupt, có thể dùng để điều khiển tần số lấy mẫu ADC từ microphone MAX9814, hoặc đơn giản là tạo timer cho hiệu ứng LED.

Trong một số cấu hình, TIM2 cũng có thể được chọn làm trigger source cho ADC (thay vì để ADC chạy liên tục), từ đó giúp thu thập tín hiệu âm thanh có chu kỳ và tránh lặp mẫu không đồng đều.

Trong project này, TIM2 được sử dụng để tạo ra tín hiệu trigger source cho ADC hoạt động ở tần số 14.4 kHz – tức là mỗi 69.4 µs sẽ yêu cầu ADC thực hiện một phép đo điện áp từ MAX9814 (tín hiệu âm thanh analog). Dữ liệu ADC sau đó được lưu vào buffer, và khi đủ mẫu, sẽ thực hiện FFT để phân tích tần số.

c) **DMA** (Direct Memory Access) – Truyền dữ liệu không cần CPU

DMA được dùng:

- DMA cho TIM1 PWM: Truyền dữ liệu duty cycle từ RAM sang thanh ghi CCR của Timer1 theo đúng thời gian để tạo ra tín hiệu điều khiển LED.

Việc sử dụng DMA giúp CPU giảm tải rất lớn, chỉ tập trung vào xử lý chính như chuyển frame, xử lý FFT.

d) **ADC1** – Đọc tín hiệu âm thanh từ micro

Module micro MAX9814 cho tín hiệu analog mức điện áp tương ứng với cường độ âm thanh. Tín hiệu này được đưa vào chân PA0 (kênh ADC1_IN0) của vi điều khiển. Trong project đang triển khai, ADC1 được tối ưu hóa để phục vụ cho việc xử lý tín hiệu âm thanh chính xác và hiệu quả

Trigger lấy mẫu: Dùng ngắt trigger từ Timer 2 (TIM2) để đảm bảo lấy mẫu theo chu kỳ đều đặn với tần số 14.4 kHz.

Xử lý dữ liệu: Sau khi đủ số mẫu, dữ liệu được dùng để:

- Tính FFT để phân tích tần số.
- Cập nhật các hiệu ứng LED dựa trên năng lượng hoặc tần số tín hiệu.

e) **GPIO** – Kết nối vật lý với các thiết bị ngoại vi

Các chân GPIO được sử dụng như sau:

- PE9: TIM1_CH1, xuất tín hiệu PWM cho WS2812B.
- PA0: ADC1_IN0, nhận tín hiệu từ MAX9814.
- PA4: MODE_BUTTON - nút nhấn chuyển chế độ hiệu ứng
- PE11: BRIGHTNESS_MODE_BUTTON- nút nhấn điều chỉnh độ sáng

f) **RCC** (Reset and Clock Control) – Quản lý xung nhịp hệ thống

Để hệ thống hoạt động chính xác, RCC được cấu hình:

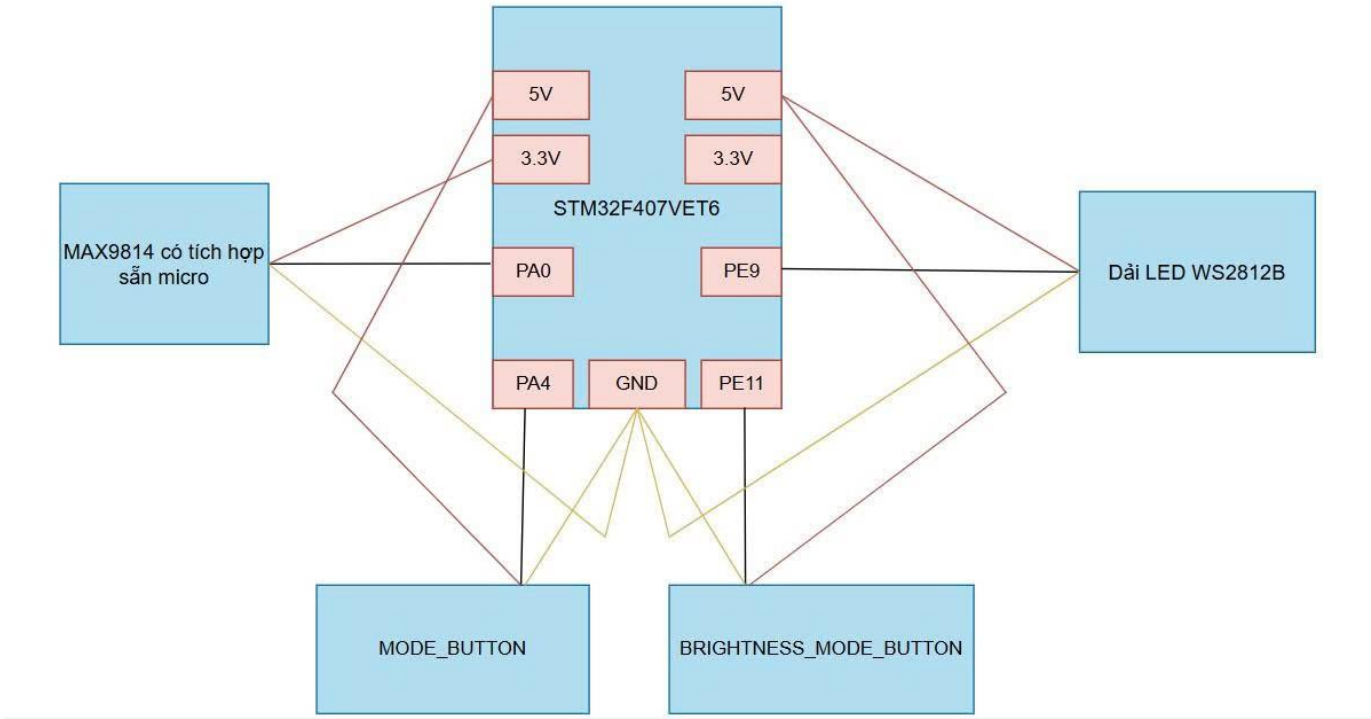
- Sử dụng HSE 8 MHz (thạch anh ngoài).
- Dùng PLL để tăng tốc độ hệ thống lên đến 168 MHz.
- TIM1 nằm trên bus APB2, có thể nhận xung gấp đôi (Prescaler x2) nhờ tính năng ‘x2 clock multiplier for timers’ trong cấu trúc STM32, giúp đảm bảo tần số PWM đủ cao.
- ADC được cấp xung từ APB2 hoặc chia xung phù hợp để đạt được tốc độ lấy mẫu mong muốn.

g) **NVIC** – Quản lý ngắt

- Hệ thống sử dụng các ngắt sau:
  - Ngắt DMA: Biết khi nào quá trình cập nhật dải LED hoàn tất, từ đó chuẩn bị frame tiếp theo.
  - Ngắt ADC (nếu không dùng DMA): Phục vụ đọc và xử lý tín hiệu âm thanh theo thời gian thực.
  - Ngoài ra, ngắt từ các nút bấm (nếu có) cũng được xử lý trong hệ thống.

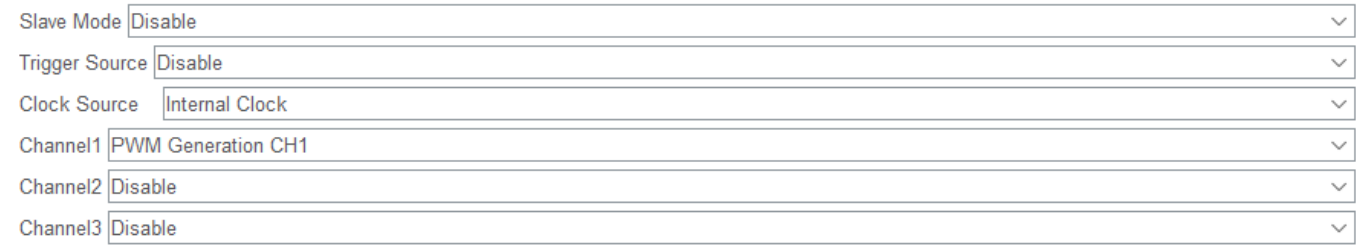
IV. THIẾT KẾ HỆ THỐNG

1. Sơ đồ khối hệ thống tổng thể

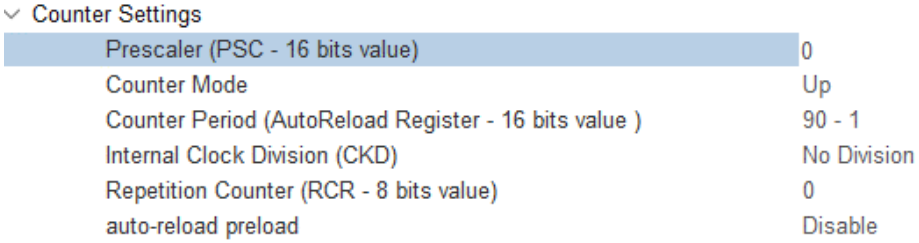


2. Cấu hình các mô-đun ngoại vi

2.1. Timer1



- Sử dụng TH1_CH1.



Prescaler (PSC) = 0

- Bộ chia xung không được sử dụng, tức là Timer chạy ở tốc độ xung gốc của APB2 Timer Clock (72 MHz).

AutoReload Register (ARR) = 90 - 1

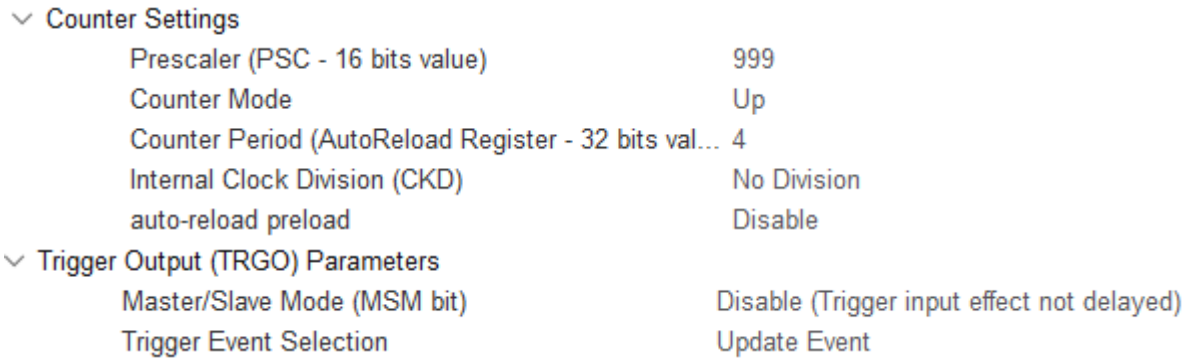
- Chu kỳ đếm là 90 xung clock.
- Với clock 72 MHz:

$$TPWM=\frac{90}{72x10^6} \approx 1.25\ \mu s$$

Chu kỳ này rất quan trọng vì WS2812B yêu cầu mỗi bit dài đúng 1.25 µs.

Tham số	Giá trị	Giải thích
Mode	PWM mode 1	Ở chế độ này, chân PWM ở mức HIGH từ đầu chu kỳ đến khi counter = CCRx (Pulse), sau đó xuống LOW. Đây là chế độ chuẩn để tạo xung PWM có độ rộng thay đổi (duty cycle).
Pulse	0	Giá trị ban đầu trong thanh ghi CCR1. Nghĩa là ban đầu PWM không tạo xung (duty = 0%). Trong thực tế, giá trị này sẽ được cập nhật liên tục qua DMA.
Output Compare Preload	Enable	Cho phép ghi trước giá trị mới vào CCR1 nhưng chỉ cập nhật vào đầu chu kỳ kế tiếp. Điều này rất quan trọng để cập nhật PWM mà không tạo ra xung sai lệch giữa chừng. Bắt buộc phải Enable khi dùng DMA.
CH Polarity	High	PWM bắt đầu ở mức HIGH, rồi xuống LOW (phù hợp với yêu cầu xung của WS2812B).
CH Idle State	Reset	Khi timer dừng, chân PWM sẽ ở mức LOW. Đây là trạng thái an toàn.

2.2. Timer2



Prescaler : 999

$$\text{Tần số đếm} = \frac{72\ \text{MHz}}{(999 + 1)} = 72\ \text{kHz}$$

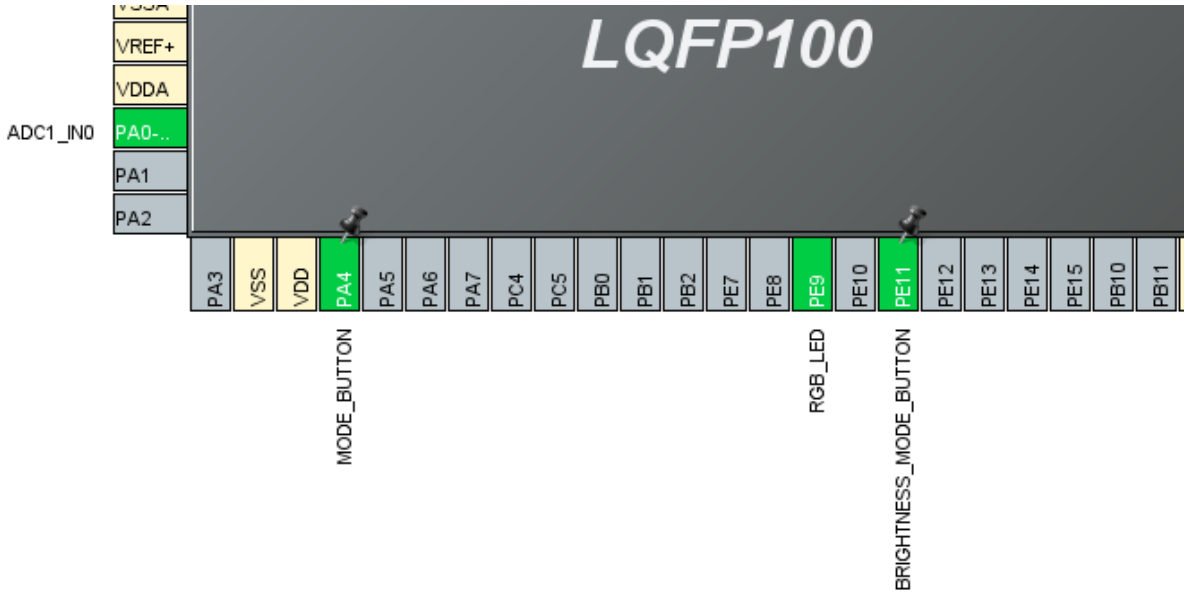
Tạo tần số thấp hơn từ 72 MHz → 72 kHz

Counter Period : 4

$$\text{Chu kỳ tràn} = (4 + 1) \times \frac{1}{72\text{kHz}} \approx 69.4\ \mu\text{s}\ (\sim 14.4\ \text{kHz})$$

Trigger với tần số ~14.4 kHz.
TRGO : Update Event
Gửi trigger ra khi Timer đếm xong 1 chu kỳ.
Kết nối với ADC1 với mục đích khiến mỗi chu kỳ của TIM2 (tần số ~14.4 kHz) sẽ kích hoạt một lần ADC để lấy mẫu âm thanh.

2.3. GPIO



Pinout view

Pin Name	Signal on Pin	GPIO output ...	GPIO mode	GPIO Pull-up...	Maximum ou...	User Label	Modified
PA0-WKUP	ADC1_IN0	n/a	Analog mode	No pull-up an...	n/a		☐

Pin Name	Signal on Pin	GPIO output ...	GPIO mode	GPIO Pull-up...	Maximum ou...	User Label	Modified
PA4	n/a	n/a	Input mode	Pull-up	n/a	MODE_BUT...	☑
PE11	n/a	n/a	Input mode	Pull-up	n/a	BRIGHTNES...	☑

Pin Name	Signal on Pin	GPIO output ...	GPIO mode	GPIO Pull-up...	Maximum ou...	User Label	Modified
PE9	TIM1_CH1	n/a	Alternate Fu...	No pull-up an...	Low	RGB_LED	☑

Chân	Vai trò	Lưu ý quan trọng
PA0	ADC1_IN0	Đặt chế độ Analog là bắt buộc để tránh dòng rò nội.
PA4	MODE_BUTTON	Dùng GPIO_Input + Pull-up là cấu hình tiêu chuẩn cho nút nhấn nối xuống GND.
PE11	BRIGHTNESS_MODE_BUTTON	Dùng GPIO_Input + Pull-up là cấu hình tiêu chuẩn cho nút nhấn nối xuống GND.
PE9	TIM1_CH1	Phải dùng Alternate Function (AF) đúng với Timer (TIM1_CH1).

2.4. NVIC

ADC1, ADC2 and ADC3 global interrupts	☑	0	0
TIM1 break interrupt and TIM9 global interrupt	☐	0	0
TIM1 update interrupt and TIM10 global interrupt	☐	0	0
TIM1 trigger and commutation interrupts and TIM11 global interrupt	☐	0	0
TIM1 capture compare interrupt	☐	0	0
TIM2 global interrupt	☑	0	0

Ngắt	Ưu tiên	Giải thích
ADC_IRQn	Bật và cao nhất	ADC dùng để lấy mẫu tín hiệu âm thanh nên cần ưu tiên cao nhất.
DMA2_Stream1_IRQn	5 (thấp hơn)	DMA có mức ưu tiên vừa phải → đúng với vai trò phụ trợ.
TIM2_IRQn	0 (cao nhất)	TIM2 có thể dùng để xử lý hiệu ứng LED theo chu kỳ nhanh → ưu tiên cao là hợp lý.

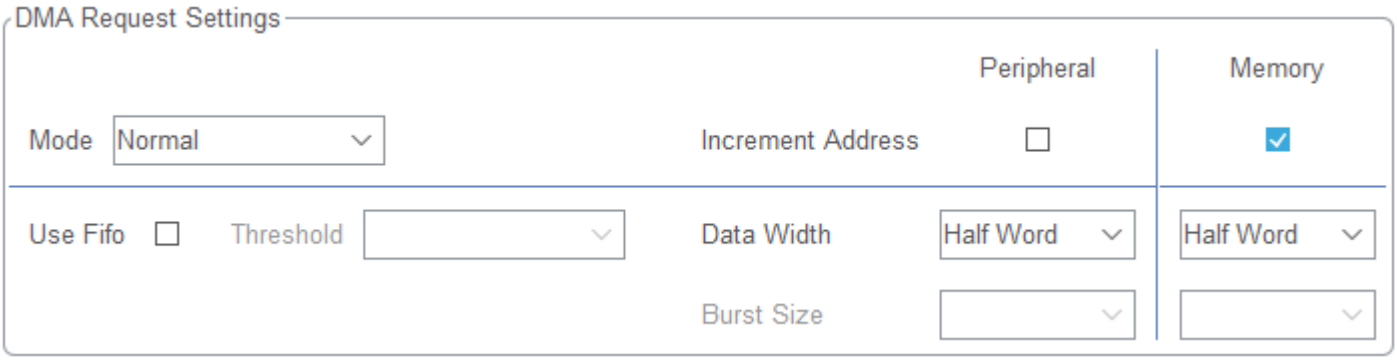
2.5. DMA + PWM

Việc tạo xung chính xác cho chuỗi WS2812B đòi hỏi khả năng tạo PWM với độ phân giải tốt (~1.25 µs) và truyền nhanh dữ liệu. Trực tiếp bit-banging trên CPU sẽ rất tốn tài nguyên. Giải pháp tối ưu là dùng phần cứng Timer/PWM kết hợp DMA. Điều này cho phép vi điều khiển chỉ việc nạp dữ liệu vào bộ nhớ và "bật" DMA, sau đó phần cứng tự động phát xung mà không can thiệp CPU.

Cấu hình DMA:

DMA Request	Stream	Direction	Priority
TIM1_CH1	DMA2 Stream 1	Memory To Peripheral	High

Thành phần	Giá trị cấu hình	Giải thích
DMA Request	TIM1_CH1	DMA được kích hoạt mỗi khi TIM1 cần cập nhật giá trị PWM mới để tạo ra một xung dữ liệu cho LED.
Stream	DMA2 Stream 1	Đây là một trong các stream của bộ DMA2 mà vi điều khiển cho phép liên kết với TIM1_CH1.
Direction	Memory To Peripheral	DMA truyền dữ liệu từ RAM (bộ nhớ) → Peripheral (TIM1). Dữ liệu thường là 1 mảng chứa các giá trị độ rộng xung (duty cycle).
Priority	High	WS2812B yêu cầu độ chính xác cao về thời gian. Việc đặt DMA ở ưu tiên cao giúp tránh bị chậm trễ bởi các DMA khác hoặc ngắt khác.



Tham số	Giá trị	Mô tả cụ thể
Mode	Normal	DMA chỉ chạy một lần hết buffer. Dùng cho từng khung dữ liệu LED. Sau khi xong, cần gọi lại hàm DMA để truyền tiếp.
Memory Increment (MemInc)	Enable	DMA sẽ lấy từng giá trị tiếp theo trong buffer chứa duty cycle cho LED
Memory Data Alignment	Half-word (16-bit)	Mỗi phần tử của buffer là uint16_t, tương ứng với thanh ghi CCR1 (16-bit)
Peripheral Data Alignment	Half-word (16-bit)	Ghi đúng kích thước vào thanh ghi 16-bit của timer

```
void WS2812_Send (void)
{
	uint32_t indx=0;
	uint32_t color;

	for (int i= 0; i<MAX_LED; i++)
	{
#if USE_BRIGHTNESS
		color = ((LED_Mod[i][1]<<16) | (LED_Mod[i][2]<<8) | (LED_Mod[i][3]));
#else
		color = ((LED_Data[i][1]<<16) | (LED_Data[i][2]<<8) | (LED_Data[i][3]));
```

```
#endif

        for (int i=23; i>=0; i--)
        {
                if (color&(1<<i))
                {
                        pwmData[indx] = 60;
                }
                else pwmData[indx] = 30;
                indx++;
        }
    }
    for (int i=0; i<50; i++)
    {
        pwmData[indx] = 0;
        indx++;
    }

    HAL_TIM_PWM_Start_DMA(&htim1, TIM_CHANNEL_1, (uint32_t *)pwmData, indx);
    while (!datasentflag){};
    datasentflag = 0;
}
```

Nguyên lý :

Bước	Mô tả
1	Gộp 3 giá trị GRB thành số 24 bit
2	Mã hóa 24 bit thành các xung PWM duty cycle (60 hoặc 30)
3	Thêm khoảng 50 xung 0 để tạo tín hiệu reset
4	Gửi toàn bộ mảng PWM bằng DMA thông qua Timer1
5	Chờ callback báo hoàn tất truyền thì mới cho phép xử lý tiếp

BƯỚC 1: Tạo dữ liệu màu GRB cho từng LED

Mỗi LED cần một dữ liệu màu gồm 3 thành phần: Green, Red, Blue (theo chuẩn WS2812B, không phải RGB). Mỗi thành phần có 8 bit → 1 LED cần tổng cộng 24 bit.

```
color = ((LED_Data[i][1]<<16) | (LED_Data[i][2]<<8) | (LED_Data[i][3]));
```

Ví dụ: G = 255, R = 128, B = 64 → color = 0xFF8040 (hex) → 24 bit nhị phân.

BƯỚC 2: Mã hóa 24 bit thành PWM duty cycle

WS2812B không truyền bit trực tiếp mà dùng tín hiệu PWM để biểu diễn bit. Mỗi bit được biểu diễn bằng 1 xung PWM:
- Bit '1' → xung rộng (duty cycle = 66% → giá trị 60)
- Bit '0' → xung hẹp (duty cycle = 33% → giá trị 30)

```
for (int i=23; i>=0; i--)
	{
		if (color&(1<<i))
		{
			pwmData[indx] = 60;
		}
		else pwmData[indx] = 30;
		indx++;
	}
```

BƯỚC 3: Thêm xung reset vào cuối frame

Sau khi gửi hết dữ liệu LED, cần giữ mức thấp ~50 micro giây để LED reset và hiển thị.
Với tần số PWM ~800kHz, mỗi xung là ~1.25us → 50 xung PWM `0` tương đương ~62.5us.

```
for (int i=0; i<50; i++){
	pwmData[indx] = 0;
	indx++;
    }
```

BƯỚC 4: Gửi mảng pwmData[] bằng DMA + Timer PWM

```
HAL_TIM_PWM_Start_DMA(&htim1, TIM_CHANNEL_1, (uint32_t *)pwmData, indx);
```

- Gọi hàm này sẽ bắt đầu truyền indx phần tử của mảng pwmData[] đến kênh PWM của Timer1.
- DMA sẽ tự động lấy từng phần tử và ghi vào thanh ghi CCR1 của TIM1 → tạo chuỗi PWM liên tục.

BƯỚC 5: Chờ DMA hoàn tất

```
while (!datasentflag){};
    datasentflag = 0;
```

- Sau khi DMA truyền xong toàn bộ mảng, callback

HAL_TIM_PWM_PulseFinishedCallback() được gọi và gán datasentflag = 1.

- Vòng while dùng để đảm bảo truyền xong trước khi tiếp tục xử lý khác.

2.6. ADC

ADC (Analog to Digital Convert) là bộ chuyển đổi tương tự sang số. Đại lượng tương tự là Điện áp Vin được so sánh với điện áp mẫu Vref (giá trị lớn nhất), sau đó được chuyển đổi thành số lưu vào thanh ghi DATA của bộ chuyển đổi đó.

Có 2 tham số quan trọng của bộ ADC cần lưu ý:

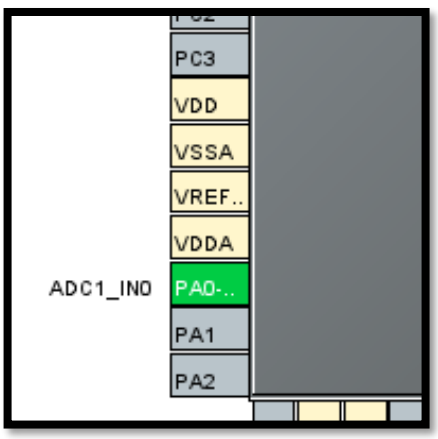
- Tốc độ lấy mẫu (sampling).
- Độ phân giải.

Trong đề tài này, **ADC** là thành phần chịu trách nhiệm "đọc" tín hiệu âm thanh analog từ module micro MAX9814 và chuyển nó thành giá trị số trên STM32F407VET6. Nhờ ADC, vi điều khiển thu được biên độ tín hiệu mic và từ đó tính toán các thông số như độ lớn tức thời (amplitude) hoặc phân tích phổ (FFT). Kết quả đầu ra của ADC là đầu vào quan trọng để xác định hiệu ứng ánh sáng LED WS2812B phải hiển thị như thế nào (VD: bật tối đa, chuyển màu, lan tỏa, v.v.) theo nhịp của nhạc.

- **Nguyên lý hoạt động của ADC trên STM32**

1. **Độ phân giải 12 bit**
   - Kết quả chuyển đổi có độ rộng 12 bit, tức giá trị số nằm trong [0 … 4095].
   - Ví dụ, 0 → ngõ vào 0 V; 4095 → ngõ vào 3.3 V.
2. **Tốc độ lấy mẫu (sampling rate)**
   - ADC có thể hoạt động ở speed tối đa khoảng 2.4 MSPS khi sử dụng xung PCLK2 cao nhất và prescaler phù hợp.
   - Theo định lý Nyquist, tần số âm thanh tối đa có thể thu được chính xác là bằng một nửa tần số lấy mẫu. Trong project này, tần số lấy mẫu được cấu hình là 14.4 kHz, đảm bảo đủ để thu nhận dải tần âm thanh phổ biến (bass, mid, treble).
   - Tần số lấy mẫu được điều khiển bởi Timer hoặc dùng chế độ Continuous Conversion. Trong trường hợp này sẽ sử dụng Timer 2 để tối đa hóa độ chính xác.
3. **Chân ADC**
   - Thông thường, tín hiệu analog từ MAX9814 được đưa vào một chân ADC kênh đơn

     (ví dụ thông thường dùng chân PA0 → ADC1_IN0).



```
static void MX_ADC1_Init(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T2_TRGO;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 1;
  hadc1.Init.DMAContinuousRequests = DISABLE;
  hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }
  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_15CYCLES;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
```

Trong MX_ADC1_Init(), ADC1 được cấu hình như sau:

- Instance: ADC1
- ClockPrescaler: PCLK2 ÷ 2
- Resolution: 12 bit (kết quả 0…4095)
- ScanConvMode: DISABLE (chỉ 1 kênh)
- ContinuousConvMode: DISABLE (chuyển đổi khi phần mềm gọi)
- ExternalTrigConvEdge: ADC_EXTERNALTRIGCONVEDGE_RISING
- ExternalTrigConv: ADC_EXTERNALTRIGCONV_T2_TRGO
- DataAlign: RIGHT (kết quả căn phải)
- NbrOfConversion: 1 (đọc đúng 1 kênh)
- DMAContinuousRequests: DISABLE (không dùng DMA)
- EOCSelection: ADC_EOC_SINGLE_CONV (cờ EOC sau mỗi mẫu)

Cấu hình kênh ADC1_IN0 (PA0):

- Channel: ADC_CHANNEL_0
- Rank: 1

- SamplingTime: 15 cycles

Cấu hình này thiết lập ADC1 để đo một kênh (PA0) với độ phân giải 12 bit (0–4095), sử dụng trigger ngoài từ Timer 2 (TIM2_TRGO) để bắt đầu chuyển đổi, không chạy liên tục và không dùng DMA. Dữ liệu được căn phải, và mỗi lần đo ADC giữ mẫu trong 15 chu kỳ ADC clock trước khi chuyển đổi, phù hợp với tín hiệu thay đổi nhanh như đầu ra từ mạch MAX9814.

3. Các hàm hỗ trợ việc thực hiện các hiệu ứng led

a) Set_LED()

```
void Set_LED (int LEDnum, int Red, int Green, int Blue)
{
    LED_Data[LEDnum][0] = LEDnum;
    LED_Data[LEDnum][1] = Green;
    LED_Data[LEDnum][2] = Red;
    LED_Data[LEDnum][3] = Blue;
}
```

Mục đích: Thiết lập giá trị màu sắc cho một LED cụ thể tại vị trí chỉ định.

Nguyên lý hoạt động: Hàm nhận tham số là số LED (LEDnum) và các giá trị Red , Green , Blue (0–255). Nó lưu các giá trị này vào mảng LED_Data (hoặc tương đương) dùng để chứa dữ liệu màu của từng LED.

Hiệu ứng tạo ra: Khi sau đó gọi WS2812_Send , LED thứ LEDnum sẽ sáng với màu đã thiết lập. Nếu không gọi WS2812_Send , thay đổi chỉ nằm trong bộ nhớ đệm.

b) Set_Brightness()

Cấu hình ban đầu :

```
#define MAX_BRIGHTNESS 255
#define NORMAL_BRIGHTNESS 128
#define USE_BRIGHTNESS 1
```

Code :

```
void Set_Brightness (int brightness)
{
#if USE_BRIGHTNESS
   if (brightness > MAX_BRIGHTNESS) brightness = MAX_BRIGHTNESS;
   float scale = brightness / (float)MAX_BRIGHTNESS;
   for (int i = 0; i < MAX_LED; i++) {
```

```
        LED  Mod[i][0] = LED  Data[i][0];

        LED_Mod[i][1] = (uint8_t)(LED_Data[i][1] * scale);
        LED_Mod[i][2] = (uint8_t)(LED_Data[i][2] * scale);
        LED_Mod[i][3] = (uint8_t)(LED_Data[i][3] * scale);
    }
#endif
}
```

Mục đích: Điều chỉnh độ sáng tổng thể của dải LED (dim/bight).

Nguyên lý hoạt động:

```
if (brightness > MAX_BRIGHTNESS) brightness = MAX_BRIGHTNESS;
```

Đảm bảo rằng giá trị brightness không vượt quá giới hạn cho phép (255).

```
float scale = brightness / (float)MAX_BRIGHTNESS;
```

Tính hệ số giảm sáng (scale), ví dụ :

- brightness = 255 → scale = 1.0 → giữ nguyên
- brightness = 128 → scale ≈ 0.5 → giảm nửa sáng

```
for (int i = 0; i < MAX_LED; i++)

{

        LED_Mod[i][0] = LED_Data[i][0];

        LED_Mod[i][1] = (uint8_t)(LED_Data[i][1] * scale);

        LED_Mod[i][2] = (uint8_t)(LED_Data[i][2] * scale);

        LED_Mod[i][3] = (uint8_t)(LED_Data[i][3] * scale);

    }
```

Mỗi màu sẽ được nhân với scale (tỉ lệ độ sáng từ 0.0 đến 1.0), rồi ép kiểu về uint8_t để giữ trong khoảng 0–255. Kết quả được lưu vào LED_Mod để điều chỉnh sáng mà không làm thay đổi màu gốc.

Hiệu ứng tạo ra: Giúp giảm sáng/mờ toàn bộ LED một cách **đồng đều**, giữ nguyên màu gốc nhưng nhẹ hơn. Thích hợp để điều chỉnh sáng trong môi trường tối hoặc tiết kiệm điện.

c) Set_LEDs_color_at_once()

```
void Set_LEDs_color_at_once(int start, int end, int step, int r, int g, int b){
    for (int pos = start; pos < end; pos += step){
        Set_LED(pos, r, g, b);
    }
}
```

Mục đích: Tô màu đồng loạt cho một dãy LED RGB cách đều nhau, giúp tạo hiệu ứng đều, nhịp hoặc mẫu lặp (ví dụ: sọc, ánh sáng đuổi...).

Nguyên lý hoạt động:

- Duyệt qua các vị trí LED từ start đến end – 1 với bước nhảy step.
- Tại mỗi vị trí, gọi Set_LED() để gán màu RGB.
- Ví dụ: tô màu LED ở vị trí 0, 3, 6, 9... nếu step = 3.

Kết quả: Một chuỗi LED cách đều được phát sáng với cùng một màu sắc, tạo hiệu ứng ánh sáng theo mẫu hoặc theo nhóm.

Hiển thị sẽ có hiệu lực sau khi gọi WS2812_Send().

d) Turn_on/off_all_at_once()

```
void Turn_on_all_at_once(int r, int g, int b, int to_led){
    Set_LEDs_color_at_once(0, to_led, 1, r, g, b);
    Set_Brightness(NORMAL_BRIGHTNESS);
    WS2812_Send();
}

void Turn_off_all_at_once(void){
    Set_LEDs_color_at_once(0, MAX_LED, 1, 0, 0, 0);
    Set_Brightness(NORMAL_BRIGHTNESS);
    WS2812_Send();
}
```

Hàm	Mục đích	Nguyên lý hoạt động	Kết quả
Turn_on_all_at_once (r, g, b, to_led)	Bật sáng tất cả LED từ vị trí 0 đến to_led – 1 với màu RGB tùy chọn.	Gọi Set_LEDs_color_at_once() từ 0 đến to_led với bước 1 và màu (r,g,b). Sau đó đặt độ sáng mặc định và gửi dữ liệu LED.	Một đoạn LED phát sáng đồng loạt theo màu chỉ định.
Turn_off_all_at_once ()	Tắt tất cả LED trên dải.	Gọi Set_LEDs_color_at_once() từ 0 đến MAX_LED với màu	Toàn bộ LED được tắt ngay lập tức, không còn phát sáng.

		(0,0,0) tức là tắt. Sau đó đặt lại độ sáng và gửi dữ liệu LED.	

e) HSV_to_RGB()

```
void HSV_to_RGB(float h, float s, float v, uint8_t* r, uint8_t* g, uint8_t* b) {
  int i = (int)(h / 60.0f) % 6;
  float f = (h / 60.0f) – i;
  float p = v * (1.0f – s);
  float q = v * (1.0f – f * s);
  float t = v * (1.0f - (1.0f - f) * s);

  float r_, g_, b_;

  switch (i) {
    case 0: r_ = v; g_ = t; b_ = p; break;
    case 1: r_ = q; g_ = v; b_ = p; break;
    case 2: r_ = p; g_ = v; b_ = t; break;
    case 3: r_ = p; g_ = q; b_ = v; break;
    case 4: r_ = t; g_ = p; b_ = v; break;
    default: r_ = v; g_ = p; b_ = q; break;
  }

  *r = (uint8_t)(r_ * 255);
  *g = (uint8_t)(g_ * 255);
  *b = (uint8_t)(b_ * 255);
}
```

Mục đích: Chuyển đổi một giá trị màu từ không gian màu HSV (Hue, Saturation, Value) sang RGB (Red, Green, Blue) để sử dụng trong điều khiển LED RGB.

Nguyên lý hoạt động:

- h: màu (0° đến 360°), chia thành 6 đoạn (mỗi 60° đại diện một chuyển màu cơ bản).
- s: độ bão hòa màu (0 đến 1): 0 là xám, 1 là màu sắc nguyên gốc.
- v: độ sáng (0 đến 1): 0 là đen, 1 là sáng nhất.

Bước chuyển đổi:

1. Tìm đoạn màu tương ứng từ h chia 60 (mỗi đoạn là một pha chuyển màu).
2. Tính phần thập phân f của pha để xác định tỷ lệ trộn giữa hai màu gần nhau.
3. Tính các giá trị trung gian p, q, t để pha màu tùy vào s và v.

4. Dựa trên i (đoạn hiện tại), gán đúng công thức nội suy để tính RGB.
5. Nhân các giá trị kết quả với 255 để chuyển từ phạm vi [0,1] sang [0,255].

Kết quả: Hàm trả về các giá trị RGB tương ứng với màu HSV đã cho, thông qua các con trỏ *r, *g, *b. Đây là bước cần thiết khi bạn muốn dùng HSV để điều khiển màu sắc dải LED RGB, ví dụ như tạo hiệu ứng cầu vồng hoặc gradient màu mượt mà.

f) get_rainbow_color()

```
void get_rainbow_color(uint16_t index, uint16_t effStep, uint8_t *red, uint8_t *green, uint8_t
*blue)
{
  int16_t temp = (int16_t)(effStep - index * 1.2f);
  int16_t mod  = temp % 60;
  if (mod < 0) mod += 60;
  uint16_t ind = 60 - mod;
  ind = ind % 60;

  uint16_t segment = (ind % 60) / 20;

  float factor1, factor2;
  uint8_t rr = 0, gg = 0, bb = 0;

  switch (segment) {
    case 0:
      factor1 = 1.0f - ((float)(ind % 20) / 20.0f);
      factor2 = ((float)(ind % 20) / 20.0f);
      rr = (uint8_t)(255.0f * factor1);
      gg = (uint8_t)(255.0f * factor2);
      bb = 0;
      break;

    case 1:
      factor1 = 1.0f - ((float)((ind % 60) - 20) / 20.0f);
      factor2 = ((float)((ind % 60) - 20) / 20.0f);
      rr = 0;
      gg = (uint8_t)(255.0f * factor1);
      bb = (uint8_t)(255.0f * factor2);
      break;

    case 2:
```

```
        factor1 = 1.0f - ((float)((ind % 60) - 40) / 20.0f);
        factor2 = ((float)((ind % 60) - 40) / 20.0f);
        rr = (uint8_t)(255.0f * factor2);
        gg = 0;
        bb = (uint8_t)(255.0f * factor1);
        break;
    }
    *red = rr;
    *green = gg;
    *blue = bb;
}
```

Mục đích: Hàm get_rainbow_color() được dùng để tính toán màu sắc dạng cầu vồng cho từng LED dựa trên chỉ số index (vị trí LED) và bước hiệu ứng effStep. Mục tiêu là tạo ra một hiệu ứng màu sắc chuyển động liên tục như cầu vồng lan tỏa dọc theo dải LED.

Nguyên lý:

- Mỗi LED sẽ được gán một pha màu khác nhau dựa trên công thức: effStep - index * 1.2. Nhờ vậy, màu sắc trên các LED sẽ lệch pha, tạo hiệu ứng chuyển động.
- Pha màu được đưa về giá trị từ 0 đến 59 (vòng tròn màu 60 bước).
- Vòng tròn được chia làm 3 phần:
    1. Đỏ chuyển sang xanh lá.
    2. Xanh lá chuyển sang xanh dương.
    3. Xanh dương chuyển về đỏ.
- Trong mỗi đoạn, dùng nội suy tuyến tính (factor1, factor2) để pha trộn giữa hai màu liên tiếp.
- Kết quả là các giá trị RGB được trả về thông qua con trỏ *red, *green, *blue.

Kết quả: Mỗi lần gọi hàm với index và effStep, bạn sẽ nhận được một màu RGB nằm trong dải cầu vồng, có thể thay đổi theo thời gian và vị trí, giúp tạo hiệu ứng màu sắc động và mượt mà cho dải LED.

g) void HAL_TIM_PWM_PulseFinishedCallback()

```
void HAL_TIM_PWM_PulseFinishedCallback(TIM_HandleTypeDef *htim) {
    if (htim == &htim1) {
        HAL_TIM_PWM_Stop_DMA(&htim1, TIM_CHANNEL_1);
        datasentflag = 1;
    }
}
```

Mục đích: Ngắt khi PWM+DMA hoàn tất gửi.

Nguyên lý hoạt động: Khi htim1 kết thúc DMA, gọi HAL_TIM_PWM_Stop_DMA, set datasentflag=1.

Kết quả: WS2812_Send có thể kết thúc chờ và tiếp tục.

h) frequency_to_color_temp()

```
void frequency_to_color_temp(uint16_t freq, uint8_t* r, uint8_t* g, uint8_t* b) {
    if (freq < 250) {
        float ratio = (float)freq / 250.0f;
        *r = 255;
        *g = (uint8_t)(ratio * 165);
        *b = 0;
    } else if (freq <= 2000) {
        float ratio = (float)(freq - 250) / 1750.0f;
        *r = (uint8_t)(255 * (1.0f - ratio));
        *g = 255;
        *b = 0;
    } else {
        float ratio = (freq - 2000) / 18000.0f;
        if (ratio > 1.0f) ratio = 1.0f;

        if (ratio < 0.5f) {
            float t = ratio * 2.0f;
            *r = 0;
            *g = (uint8_t)(255 * (1.0f - t));
            *b = (uint8_t)(255 * t);
        } else {// Blue to Violet
            float t = (ratio - 0.5f) * 2.0f;
            *r = (uint8_t)(128 * t);
            *g = 0;
            *b = 255;
        }
    }
}
```

Chức năng:
Chuyển tần số âm thanh thành màu sắc nhiệt tương ứng – từ màu ấm (đỏ, cam) đến màu lạnh (xanh, tím).

Nguyên lý hoạt động:

1. Tần số thấp (< 250Hz):
   → Chuyển từ đỏ sang cam
   → r = 255, g tăng dần từ 0 đến 165, b = 0
2. Tần số trung (250–2000Hz):
   → Chuyển từ vàng sang xanh lá
   → g = 255, r giảm dần từ 255 về 0, b = 0
3. Tần số cao (> 2000Hz):
   - Từ 2000 đến 11000Hz: xanh lá → xanh dương
   - Từ 11000 đến 20000Hz: xanh dương → tím
     → g giảm, b tăng rồi giữ nguyên; sau đó r tăng, g = 0

Kết quả: Cho ra màu RGB phù hợp với cảm giác "nhiệt độ" của âm thanh:

- Bass → đỏ / cam (ấm)
- Giọng nói → vàng / xanh lá (trung)
- Treble → xanh / tím (lạnh)

i) frequency_to_full_spectrum()

```
void frequency_to_full_spectrum(uint16_t freq, uint8_t* r, uint8_t* g, uint8_t* b) {
  float hue = (float)freq / 7000.0f * 300.0f;
  if (hue > 300.0f) hue = 300.0f;
  HSV_to_RGB(hue, 1.0f, 1.0f, r, g, b);
}
```

Mục đích: Biến tần số âm thanh thành giá trị Hue và quy đổi thành màu RGB.

Nguyên lý hoạt động: Map freq từ 0 đến 7000Hz thành hue từ 0 đến 300 độ rồi gọi HSV_to_RGB().

Kết quả: LED thể hiện màu động theo nhạc với phổ màu đầy đủ từ đỏ tới tím.

j) Xử lý FFT

```
void arm_rfft_fast_f32(arm_rfft_fast_instance_f32 * S,float32_t * p, float32_t * pOut, uint8_t
ifftFlag);
```

Hàm arm_rfft_fast_f32() là hàm trong thư viện CMSIS-DSP của ARM với vai trò chuyển tín hiệu ADC sang tín hiệu thực bằng thuật toán Cooley-Tukey FFT.

- **record_sample_and_maybe_runFFT**()

```
void record_sample_and_maybe_runFFT(uint16_t raw) {
  float centered = ((float)(raw - (uint16_t)(middle_point))) * UINT16_TO_FLOAT;
  fftBufIn[fftIndex] = centered;
  fftIndex++;

  if (fftIndex >= FFT_BUFFER_SIZE) {
    arm_rfft_fast_f32(&fftHandler, fftBufIn, fftBufOut, 0);

    fftReady = 1;
    fftIndex = 0;
  }
}
```

Mục đích : Đây là phần xử lý thu mẫu tín hiệu âm thanh và thực hiện Fast Fourier Transform qua arm_rfft_fast_f32() với nhiệm vụ **:**

- Thu mẫu tín hiệu âm thanh từ mic (MAX9814) qua ADC
- Xử lý trung tâm hóa tín hiệu
- Đưa dữ liệu vào buffer
- Khi buffer đầy, thực hiện Fast Fourier Transform để chuyển từ miền thời gian sang miền tần số

Nguyên lý hoạt động

- **Bước 1:** Loại bỏ thành phần DC và chuẩn hóa

float centered = ((float)(raw_adc - (uint16_t)(middle_point))) * UINT16_TO_FLOAT;

- middle_point: giá trị ADC trung bình để loại bỏ lệch DC

- UINT16_TO_FLOAT = 0.00001525879f là hằng số dùng để đưa giá trị ADC về dải (1.0 ; +1.0)

- Kết quả là tín hiệu đã trung tâm hóa, sẵn sàng cho xử lý FFT

- **Bước 2:** Ghi vào buffer FFT

fftBufIn[fftIndex] = centered;

fftIndex++;

-Ghi tín hiệu vào mảng fftBufIn

-fftIndex là chỉ số đếm mẫu thu được

- **Bước 3:** Khi đủ mẫu – thực hiện FFT

```
if (fftIndex >= FFT_BUFFER_SIZE) {
    arm_rfft_fast_f32(&fftHandler, fftBufIn, fftBufOut, 0);
    fftReady = 1;
    fftIndex = 0;
}
```

- Khi đã thu đủ FFT_BUFFER_SIZE mẫu

- Gọi hàm arm_rfft_fast_f32() từ thư viện CMSIS-DSP để thực hiện FFT

  - Đầu vào: fftBufIn (dạng float)
  - Đầu ra: fftBufOut (dạng phức liên hợp hoặc biên độ)

-Gán fftReady = 1 để báo hiệu đã có dữ liệu FFT mới

- **Trong vòng lặp while**

```
while (1)
  {
                    if (fftReady){
                            peakVal = 0.0f;
                            peakHz = 0.0f;

                            uint16_t halfBins = FFT  BUFFER  SIZE / 2;
                            for (uint16_t k = 1; k < halfBins; k++) {
                                    float re = fftBufOut[2 * k];
                                    float im = fftBufOut[2 * k + 1];
                                    float mag = sqrtf(re * re + im * im);
                                    if (mag > peakVal) {
                                            peakVal = mag;
                                            peakHz  = (uint16_t)(k  * SAMPLE_RATE_HZ /
(float)(FFT_BUFFER_SIZE));
                                    }
                            }

                            fftReady = 0;
                    }
      … (effect and button) …
                HAL_Delay(2);
  }
```

Mục đích: Tìm tần số mạnh nhất (có biên độ lớn nhất) trong tín hiệu âm thanh sau khi xử lý FFT.

Nguyên lý hoạt động:

- if (fftReady): Khi FFT xong thì bắt đầu xử lý.
- Duyệt qua các bin FFT (bỏ bin 0).
- Tính biên độ mỗi bin: sqrt(re² + im²).
- So sánh để tìm bin có biên độ lớn nhất → tính ra tần số tương ứng (peakHz).
- Đặt lại fftReady = 0 để chờ lượt FFT mới.
- Tạo trễ 2 ms bằng HAL_Delay(2);

Kết quả: Ta có tần số mạnh nhất (peakHz) và độ lớn của nó (peakVal).

- **HAL_ADC_ConvCpltCallback()**

```
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
	if (hadc->Instance == ADC1)
	  {
			uint16_t raw = HAL_ADC_GetValue(&hadc1);
			if ( middle_point_index > 0){
				middle_point += raw;
				middle_point_index--;
			}

			else if (middle_point_index == 0){
				middle_point/= 32;
				middle_point_index--;
			}

			else{
				record_sample_and_maybe_runFFT(raw);
			}

	    amp = abs(raw - (uint16_t)middle_point);
	  }
}
```

Mỗi khi Timer2 tạo ra trigger, ADC1 bắt đầu đo tín hiệu analog. Sau khi đo xong, hệ thống tự động gọi hàm HAL_ADC_ConvCpltCallback()

Mục đích :

- Đọc tín hiệu ADC khi có ngắt hoàn thành chuyển đổi.
- Chuẩn hóa tín hiệu ADC bằng cách tính middle_point từ 32 sample đầu tiên.

- Ghi các mẫu ADC đã chuẩn hóa fftBufIn để chuẩn bị cho xử lý FFT.
- Kích hoạt thuật toán FFT khi đã thu đủ số lượng mẫu cần thiết.
- Tính toán độ lớn tín hiệu để làm hiệu ứng đồng bộ âm thanh**.**

Nguyên lý:

if (hadc->Instance == ADC1) → Nếu ADC đang chạy là **ADC1** thì mới xử lý tiếp (để tránh nhầm ADC khác).

uint16_t raw = HAL_ADC_GetValue(&hadc1); → raw bằng giá trị ADC vừa đọc được từ ADC1

```
if (middle_point_index > 0) {
   middle_point += raw;
   middle_point_index--;
}
```

→ Nếu còn đang tính trung bình:

- middle_point bằng middle_point cộng raw
- middle_point_index giảm đi 1

→ Mục đích: thu thập 32 mẫu đầu tiên để tính trung bình

```
else if (middle_point_index == 0) {
   middle_point /= 32;
   middle_point_index--;
}
```

→ Nếu vừa đủ 32 mẫu: middle_point = middle_point / 32 →middle_point_index giảm đi 1 để không vào nhánh này nữa.

```
else {
   record_sample_and_maybe_runFFT(raw);
}
```

→ Nếu đã tính xong middle_point, thì gửi giá trị raw vào hàm xử lý FFT

amp = abs(raw - (uint16_t)middle_point);

amp bằng giá trị tuyệt đối của raw trừ middle_point

→ Dùng để đo độ lớn tín hiệu tại thời điểm hiện tại (mức âm lượng)

k) Các nút bấm (Chuyển đổi hiệu ứng và thay đổi độ sáng) :

*(effect and button)

```
if(HAL_GPIO_ReadPin(MODE_BUTTON_GPIO_Port, MODE_BUTTON_Pin) == 0) {
    while(HAL_GPIO_ReadPin(MODE_BUTTON_GPIO_Port, MODE_BUTTON_Pin) ==
0) {}
    mode_button_index = (mode_button_index + 1) % 6;
}

                    switch (mode_button_index) {
                        case 0: effect_flash_fade_random_color(amp, peakHz,
brightness_mode); break;
                        case 1: effect_dynamic_vu_meter(amp, peakHz, brightness_mode);
break;
                        case 2: effect_spectrum_color_bands(amp, peakHz,
brightness_mode); break;
                        case 3: effect_frequency_chase_gradient(amp, peakHz,
brightness_mode); break;
                        case 4: effect_rainbow_roll(amp, peakHz, brightness_mode);
break;
                        case 5: effect_bass_pulse_glow(amp, peakHz, brightness_mode);
break;
                        default: effect_flash_fade_random_color(amp, peakHz,
brightness_mode); break;
                    }

                    if(HAL_GPIO_ReadPin(BRIGHTNESS_MODE_BUTTON_GPIO_Port,
BRIGHTNESS_MODE_BUTTON_Pin) == 0) {
while(HAL_GPIO_ReadPin(BRIGHTNESS_MODE_BUTTON_GPIO_Port,
BRIGHTNESS_MODE_BUTTON_Pin) == 0) {}
                        brightness_mode = (uint8_t)((float)MAX_BRIGHTNESS * ((25.0f
* (float)brightness_button_count) / 100.0f));
                        brightness_button_count = (brightness_button_count + 1) % 5;
                    }
```

Mục đích :

- Tạo nút MODE dùng để chuyển đổi các hiệu ứng
- Tạo nút BRIGHTNESS để thay đổi độ sáng

Nút MODE – chuyển hiệu ứng:

- `if(HAL_GPIO_ReadPin(MODE_BUTTON_GPIO_Port, MODE_BUTTON_Pin) == 0)`
    - Kiểm tra nút **MODE** có được nhấn không.

- **while(HAL_GPIO_ReadPin(MODE_BUTTON_GPIO_Port, MODE_BUTTON_Pin) == 0) {}**
  - Chờ nút được thả ra (chống rung).
- **mode_button_index = (mode_button_index + 1) % 6;**
  - Tăng chỉ số hiệu ứng, xoay vòng từ 0 đến 5.
- **switch (mode_button_index):** Gọi một trong 6 hiệu ứng:

1. effect_flash_fade_random_color(...)
2. effect_dynamic_vu_meter(...)
3. effect_spectrum_color_bands(...)
4. effect_frequency_chase_gradient(...)
5. effect_rainbow_roll(...)
6. effect_bass_pulse_glow(...)

Nút BRIGHTNESS – thay đổi độ sáng:

- if(HAL_GPIO_ReadPin(BRIGHTNESS_MODE_BUTTON_GPIO_Port, BRIGHTNESS_MODE_BUTTON_Pin) == 0)
  - Kiểm tra nút BRIGHTNESS có được nhấn không.
- while(HAL_GPIO_ReadPin(BRIGHTNESS_MODE_BUTTON_GPIO_Port, BRIGHTNESS_MODE_BUTTON_Pin) == 0) {}
  - Chờ thả nút (chống rung).
- brightness_mode = (uint8_t)((float)MAX_BRIGHTNESS * ((25.0f * (float)brightness_button_count) / 100.0f));: Tính độ sáng theo mức:
  - 0%, 25%, 50%, 75%, 100% (tương ứng với brightness_button_count)
- brightness_button_count = (brightness_button_count + 1) % 5;
  - Tăng chỉ số mức sáng, xoay vòng.

HAL_Delay(2) : Nghỉ 2 ms để ổn định hệ thống, tránh lặp nhanh.

4. Các hàm hiệu ứng led:

a. effect_flash_fade_random_color()

```
#define AMP_THRESHOLD 1000
#define FADE_DURATION_MS 500
void effect_flash_fade_random_color(uint16_t amp, uint16_t peakHz, uint8_t
brightness_mode) {
    static uint32_t last_flash_time = 0;

    uint32_t current  time = HAL_GetTick();

    if (amp > AMP_THRESHOLD) {
        uint8_t r = rand() % 256;
```

```
        uint8_t g = rand() % 256;
        uint8_t b = rand() % 256;

        for (int i = 0; i < MAX_LED; i++) {
            Set_LED(i, r, g, b);
        }
        last_flash_time = current_time;
    }
    uint32_t elapsed_time = current_time - last_flash_time;
    int brightness;
    if (elapsed_time >= FADE_DURATION_MS) {
        brightness = 0;
    } else {
        brightness = brightness_mode - (brightness_mode * elapsed_time) /
FADE_DURATION_MS;
    }

    Set_Brightness(brightness);
    WS2812_Send();
}
```

Chức năng: Tạo hiệu ứng đèn chớp màu ngẫu nhiên khi âm thanh vượt ngưỡng, sau đó mờ dần.

Nguyên lý hoạt động:

1. `if (amp > AMP_THRESHOLD)`
   `uint8_t r = rand() % 256;`
   `uint8_t g = rand() % 256;`
   `uint8_t b = rand() % 256;`
   Khi âm thanh vượt ngưỡng, tạo 1 màu RGB bất kỳ (0–255 mỗi kênh) → màu sẽ thay đổi theo từng beat.
2. `for (int i = 0; i < MAX_LED; i++) {`
   `Set_LED(i, r, g, b);`
   `}`
   Gán màu mới cho toàn bộ LED
3. `last_flash_time = current_time;`
   Ghi lại thời gian của lần chớp sáng gần nhất → dùng để tính thời gian mờ dần.
4. `uint32_t elapsed_time = current_time - last_flash_time;`

   Nếu chưa có beat nào thì elapsed_time = 0.

   Nếu có beat, sau mỗi lần gọi hàm, thời gian trôi qua tăng dần → phục vụ tính hiệu ứng mờ dần.

5. if (elapsed_time >= FADE_DURATION_MS) {
   brightness = 0;
   } else {
   brightness = brightness_mode - (brightness_mode * elapsed_time) / FADE_DURATION_MS;
   }
   Nếu đã qua 500ms → độ sáng = 0 (LED tắt).
   Nếu chưa qua 500ms:→Độ sáng giảm dần tuyến tính từ brightness_mode về 0.
6. Set_Brightness(brightness);
   Hàm này điều chỉnh độ sáng tổng thể của LED theo brightness
7. WS2812_Send();
   Gửi toàn bộ dữ liệu mới (màu và độ sáng) đến dải LED WS2812B.
   Hiển thị hiệu ứng ra ngoài thực tế.

Hiệu ứng tạo ra: Nháy màu theo nhạc, sau đó mờ dần một cách mượt mà.

b. effect_dynamic_vu_meter()

```
void effect_dynamic_vu_meter(uint16_t amplitude, uint16_t peak_freq, uint8_t brightness_mode) {
    float ratio = (float)amplitude / amp_maxn;
    if (ratio > 1.0f) ratio = 1.0f;

    if (ratio <= 0.05f) {
        Turn_off_all_at_once();
        return;
    }

    int total_leds_to_light = (int)(ratio * MAX_LED);
    if (total_leds_to_light < 1) total_leds_to_light = 1;

    uint8_t r, g, b;
    frequency_to_full_spectrum(peak_freq, &r, &g, &b);

    Turn_off_all_at_once();

    int center = MAX_LED / 2;
    int leds_per_side = total_leds_to_light / 2;

    if (total_leds_to_light % 2 == 1) {
        Set_LED(center, r, g, b);
    }
```

```
    for (int i = 1; i <= leds_per_side; i++) {
        if (center - i >= 0) {
            Set_LED(center - i, r, g, b);
        }
        if (center + i < MAX_LED) {
            Set_LED(center + i, r, g, b);
        }
    }

    int brightness = 5 + (int)(ratio * (brightness_mode - 5));
    Set_Brightness(brightness);
    WS2812_Send();
}
```

Chức năng:
Hiệu ứng VU meter (Volume Unit) – sáng dải LED từ giữa ra 2 bên theo âm lượng, màu sắc thay đổi theo tần số.

Nguyên lý hoạt động:

```
float ratio = (float)amplitude / amp_maxn;
if (ratio > 1.0f) ratio = 1.0f;
```

→ Tính tỉ lệ biên độ hiện tại so với biên độ cực đại. Nếu lớn hơn 1.0 thì giới hạn lại để tránh lỗi.

```
if (ratio <= 0.05f) {
    Turn_off_all_at_once();
    return;
}
```

→ Nếu âm thanh quá nhỏ (≤ 5%) thì tắt toàn bộ LED. Tránh hiển thị khi không có tiếng hoặc có nhiễu nhỏ.

```
int total_leds_to_light = (int)(ratio * MAX_LED);
if (total_leds_to_light < 1) total_leds_to_light = 1;
```

→ Tính số lượng LED cần sáng theo biên độ. Đảm bảo ít nhất 1 LED sẽ sáng nếu có âm thanh.

```
uint8_t r, g, b;
frequency_to_full_spectrum(peak_freq, &r, &g, &b);
```

→ Chuyển tần số đỉnh (peak_freq) sang mã màu RGB. Giúp ánh sáng thể hiện màu sắc theo cao độ âm thanh.

```
Turn_off_all_at_once();
```

→ Xóa trạng thái LED cũ trước khi cập nhật hiệu ứng mới.

```
int center = MAX_LED / 2;
int leds_per_side = total_leds_to_light / 2;
```

→ Tính vị trí trung tâm của dải LED. LED sẽ sáng đối xứng ra hai phía từ giữa.

```
if (total_leds_to_light % 2 == 1) {
   Set_LED(center, r, g, b);
}
```

→ Nếu số lượng LED cần sáng là lẻ → bật thêm LED ở chính giữa.

```
for (int i = 1; i <= leds_per_side; i++) {
   if (center - i >= 0) {
      Set_LED(center - i, r, g, b);
   }
   if (center + i < MAX_LED) {
      Set_LED(center + i, r, g, b);
   }
}
```

→ Bật LED sang trái và phải của vị trí trung tâm. Tạo hiệu ứng lan ra từ giữa dải LED.

```
int brightness = 5 + (int)(ratio * (brightness_mode - 5));
Set_Brightness(brightness);
```

→ Điều chỉnh độ sáng theo biên độ âm thanh. Càng lớn tiếng → độ sáng càng cao (trong giới hạn brightness_mode).

```
WS2812_Send();
```

→ Gửi toàn bộ dữ liệu LED đã cập nhật để hiển thị ra dải WS2812B.

Hiệu ứng tạo ra: Một dải đèn sáng lan ra từ giữa, màu thay đổi theo tần số, độ dài và độ sáng theo âm lượng.

c. effect_spectrum_color_bands()

```
void effect_spectrum_color_bands(uint16_t amplitude, uint16_t peak_freq, uint8_t
brightness_mode) {
   float ratio = (float)amplitude / amp_maxn;
   if (ratio > 1.0f) ratio = 1.0f;

   if (ratio <= 0.05f) {
      Turn_off_all_at_once();
      return;
   }
```

```
    int low_start = 0;
    int low_end = MAX_LED / 3;
    int mid_start = low_end;
    int mid_end = (MAX_LED * 2) / 3;
    int high_start = mid_end;
    int high_end = MAX_LED;

    Turn_off_all_at_once();

    uint8_t r = 0, g = 0, b = 0;
    int start_led = 0, end_led = 0;

    if (peak_freq <= 250) {
        frequency_to_full_spectrum(peak_freq, &r, &g, &b);
        start_led = low_start;
        end_led = low_end;
    } else if (peak_freq <= 2000) {
            frequency_to_full_spectrum(peak_freq, &r, &g, &b);
        start_led = mid_start;
        end_led = mid_end;
    } else {
            frequency_to_full_spectrum(peak_freq, &r, &g, &b);
        start_led = high_start;
        end_led = high_end;
    }

    for (int i = start_led; i < end_led; i++) {
        Set_LED(i, r, g, b);
    }
    int brightness = 5 + (int)(ratio * (brightness_mode - 5));
    Set_Brightness(brightness);
    WS2812_Send();
}
```

Chức năng:

Chia dải LED thành 3 vùng màu theo dải tần số âm thanh:

Thấp → Trung → Cao → mỗi vùng sáng màu khác nhau tùy theo tần số.

Nguyên lý hoạt động:

void effect_spectrum_color_bands(uint16_t amplitude, uint16_t peak_freq, uint8_t brightness_mode)

→Đây là hàm xử lý hiệu ứng ánh sáng "phổ màu theo dải tần". Nó sử dụng biên độ và tần số đỉnh để xác định màu và vùng LED cần hiển thị.

```
float ratio = (float)amplitude / amp_maxn;
if (ratio > 1.0f) ratio = 1.0f;
```

→Tính tỷ lệ biên độ hiện tại so với giá trị biên độ lớn nhất. Tỷ lệ này được dùng để điều chỉnh độ sáng sau cùng.

```
if (ratio <= 0.05f) {
    Turn_off_all_at_once();
    return;
}
```

→Nếu biên độ quá nhỏ (gần như không có âm thanh), tất cả các LED sẽ tắt để tiết kiệm điện và tránh hiển thị sai.

```
int low_start = 0;
int low_end = MAX_LED / 3;
int mid_start = low_end;
int mid_end = (MAX_LED * 2) / 3;
int high_start = mid_end;
int high_end = MAX_LED;
```

→Chia dải LED thành 3 phần tương ứng với 3 dải tần: Low (âm trầm), Mid (âm trung), High (âm cao).

```
Turn_off_all_at_once();
```

→Tắt toàn bộ LED trước khi hiển thị dải màu mới.

```
uint8_t r = 0, g = 0, b = 0;
int start_led = 0, end_led = 0;
```

→Khai báo biến màu RGB và giới hạn của đoạn LED sẽ được bật sáng.

```
if (peak_freq <= 250) {
    frequency_to_full_spectrum(peak_freq, &r, &g, &b);
    start_led = low_start;
    end_led = low_end;
} else if (peak_freq <= 2000) {
    frequency_to_full_spectrum(peak_freq, &r, &g, &b);
    start_led = mid_start;
    end_led = mid_end;
} else {
    frequency_to_full_spectrum(peak_freq, &r, &g, &b);
    start_led = high_start;
    end_led = high_end;
}
```

→Xác định dải tần tương ứng với peak_freq: trầm, trung, cao. Gọi hàm lấy màu tương ứng.

```
for (int i = start_led; i < end_led; i++) {
    Set_LED(i, r, g, b);
}
```

→Bật sáng tất cả LED trong đoạn xác định với màu đã tính.

```
int brightness = 5 + (int)(ratio * (brightness_mode - 5));
Set_Brightness(brightness);
WS2812_Send();
```

→Tính độ sáng dựa trên biên độ âm thanh và gửi dữ liệu LED ra dải WS2812B.

Hiệu ứng tạo ra:

LED sáng một vùng duy nhất theo loại âm thanh:
→ Bass → đầu dải
→ Trung → giữa
→ Treble → cuối

Mỗi lần tần số thay đổi, vùng sáng sẽ đổi theo.

d. effect_frequency_chase_gradient()

```
typedef struct {
    int position
    uint8_t r, g, b;
    uint8_t brightness;
    uint8_t length;
    uint8_t active;
} Pulse;

#define MAX_PULSES 8
static Pulse pulses[MAX_PULSES];
static uint16_t last_amp = 0;
static uint32_t last_pulse_time = 0;

void effect_frequency_chase_gradient(uint16_t amplitude, uint16_t peak_freq, uint8_t
brightness_mode) {
    const uint16_t BEAT_THRESHOLD = 800;
    const uint32_t MIN_PULSE_INTERVAL = 100;

    uint32_t current_time = HAL_GetTick();

    if (amplitude > BEAT_THRESHOLD &&
```

```
    amplitude > (last  amp + 200) &&
    (current_time - last_pulse_time) > MIN_PULSE_INTERVAL) {

    for (int i = 0; i < MAX_PULSES; i++) {
      if (!pulses[i].active) {
        pulses[i].position = 0;
        frequency_to_full_spectrum(peak_freq, &pulses[i].r, &pulses[i].g, &pulses[i].b);

        float ratio = (float)amplitude / amp_maxn;
        if (ratio > 1.0f) ratio = 1.0f;

        pulses[i].brightness = (uint8_t)(ratio * 255);
        pulses[i].length = 3 + (uint8_t)(ratio * 8);
        pulses[i].active = 1;

        last_pulse_time = current_time;
        break;
      }
    }
  }

  Turn_off_all_at_once();

  for (int i = 0; i < MAX_PULSES; i++) {
    if (pulses[i].active) {
      for (int j = 0; j < pulses[i].length; j++) {
        int led_pos = pulses[i].position - j;
        if (led_pos >= 0 && led_pos < MAX_LED) {
          // Calculate fade factor for trail
          float fade = 1.0f - ((float)j / pulses[i].length);
          uint8_t r = (uint8_t)(pulses[i].r * fade);
          uint8_t g = (uint8_t)(pulses[i].g * fade);
          uint8_t b = (uint8_t)(pulses[i].b * fade);

          uint8_t existing_r = LED_Data[led_pos][2];
          uint8_t existing_g = LED_Data[led_pos][1];
          uint8_t existing_b = LED_Data[led_pos][3];

          r = (r + existing_r > 255) ? 255 : r + existing_r;
          g = (g + existing_g > 255) ? 255 : g + existing_g;
          b = (b + existing_b > 255) ? 255 : b + existing_b;

          Set_LED(led_pos, r, g, b);
        }
      }

      pulses[i].position++;
```

```
            if (pulses[i].position >= MAX_LED + pulses[i].length) {
                pulses[i].active = 0;
            }
        }
    }

    Set_Brightness(brightness_mode);
    WS2812_Send();

    last_amp = amplitude;
}
```

Chức năng:
Tạo hiệu ứng xung sáng chạy dọc dải LED, màu sắc theo tần số nhạc, độ dài và độ sáng theo âm lượng.
Mỗi beat âm thanh mạnh sẽ tạo ra một "pulse" chạy.

Nguyên lý hoạt động theo các bước:

1. Điều kiện phát hiện nhịp beat

```
if (amplitude > BEAT_THRESHOLD &&
    amplitude > (last_amp + 200) &&
    (current_time - last_pulse_time) > MIN_PULSE_INTERVAL) {
```

Kiểm tra nếu biên độ âm thanh:

- Lớn hơn ngưỡng beat
- Lớn hơn đáng kể so với biên độ trước đó
- Cách lần phát hiện nhịp gần nhất ít nhất 100ms

→ Khi đó, tạo một "pulse" (xung ánh sáng mới).

2. Tạo pulse mới nếu có slot trống

```
for (int i = 0; i < MAX_PULSES; i++) {
    if (!pulses[i].active) {
        pulses[i].position = 0;
        frequency_to_full_spectrum(peak_freq, &pulses[i].r, &pulses[i].g, &pulses[i].b);

        float ratio = (float)amplitude / amp_maxn;
        if (ratio > 1.0f) ratio = 1.0f;
```

```
            pulses[i].brightness = (uint8_t)(ratio * 255);
            pulses[i].length = 3 + (uint8_t)(ratio * 8);
            pulses[i].active = 1;

            last_pulse_time = current_time;
            break;
        }
    }
```

- Tìm pulse chưa kích hoạt để sử dụng.
- Gán vị trí = 0 và đặt màu dựa theo tần số âm thanh.
- Độ sáng và độ dài phụ thuộc vào biên độ.
- Đánh dấu pulse này là đang hoạt động.

3. Tắt toàn bộ LED trước khi vẽ mới

```
Turn_off_all_at_once();
```

- Đảm bảo LED sạch trước khi hiển thị hiệu ứng mới.

4. Cập nhật và vẽ các pulse đang hoạt động

```
for (int i = 0; i < MAX_PULSES; i++) {
    if (pulses[i].active) {
        for (int j = 0; j < pulses[i].length; j++) {
            int led_pos = pulses[i].position - j;
            if (led_pos >= 0 && led_pos < MAX_LED) {
                float fade = 1.0f - ((float)j / pulses[i].length);
                uint8_t r = (uint8_t)(pulses[i].r * fade);
                uint8_t g = (uint8_t)(pulses[i].g * fade);
                uint8_t b = (uint8_t)(pulses[i].b * fade);

                uint8_t existing_r = LED_Data[led_pos][2];
                uint8_t existing_g = LED_Data[led_pos][1];
                uint8_t existing_b = LED_Data[led_pos][3];
                r = (r + existing_r > 255) ? 255 : r + existing_r;
                g = (g + existing_g > 255) ? 255 : g + existing_g;
                b = (b + existing_b > 255) ? 255 : b + existing_b;
                Set_LED(led_pos, r, g, b);
            }
        }
        pulses[i].position++;
```

```
        if (pulses[i].position >= MAX_LED + pulses[i].length) {
            pulses[i].active = 0;
        }
    }
}
```

Với mỗi pulse còn hoạt động:

- Tính vị trí LED cần sáng, tạo hiệu ứng đuôi mờ dần.
- Kết hợp màu mới với màu LED hiện tại để tạo hiệu ứng mượt.
- Di chuyển pulse sang phải sau mỗi lần chạy.
- Dừng pulse nếu đã chạy hết dải LED.

5. Cập nhật độ sáng và gửi dữ liệu

```
Set_Brightness(brightness_mode);
WS2812_Send();
last_amp = amplitude;
```

- Đặt độ sáng tổng thể.
- Gửi dữ liệu xuống dải LED WS2812.
- Lưu biên độ hiện tại để so sánh lần sau.

Hiệu ứng tạo ra: Khi có beat âm thanh → LED sáng lên một đoạn màu (gradient) rồi chạy về bên phải. Màu mỗi lần thay đổi theo tần số → tạo cảm giác sống động, rực rỡ và đa sắc.

e. effect_rainbow_roll()

```
void effect_rainbow_roll(uint16_t amplitude, uint16_t peak_freq, uint8_t brightness_mode) {
    static float offset = 0.0f;

    float ratio = (float)amplitude / amp_maxn;
    if (ratio > 1.0f) ratio = 1.0f;

    offset += ((float)peak_freq / 100.0f);
    if (offset >= 360.0f) offset -= 360.0f;

    for (int i = 0; i < MAX_LED; i++) {
        float hue = fmodf(offset + (i * (360.0f / MAX_LED)), 360.0f);
        uint8_t r, g, b;
        HSV_to_RGB(hue, 1.0f, ratio, &r, &g, &b);
        Set_LED(i, r, g, b);
```

```
    }

    int brightness = 5 + (int)(ratio * (brightness_mode - 5));
    Set_Brightness(brightness);
    WS2812_Send();
}
```

Mục đích: Tạo hiệu ứng ánh sáng cầu vồng chuyển động liên tục trên dải LED, tốc độ và độ sáng thay đổi theo âm thanh đầu vào.

Nguyên lý hoạt động:

1. `static float offset = 0.0f;`

- offset giữ giá trị để dịch chuyển dải màu cầu vồng.
- Biến tĩnh này giúp giữ trạng thái giữa các lần gọi hàm.

2. `float ratio = (float)amplitude / amp_maxn;`
`if (ratio > 1.0f) ratio = 1.0f;`

- ratio biểu thị biên độ hiện tại so với cực đại.
- Dùng để điều chỉnh độ rực màu (saturation/brightness).

3. `offset += ((float)peak_freq / 100.0f);`
`if (offset >= 360.0f) offset -= 360.0f;`

- Tần số cao sẽ làm màu chuyển động nhanh hơn (cuộn nhanh).
- Offset được giữ trong khoảng [0, 360) để làm màu cuộn tròn.

4.
```
for (int i = 0; i < MAX_LED; i++) {
    float hue = fmodf(offset + (i * (360.0f / MAX_LED)), 360.0f);
    uint8_t r, g, b;
    HSV_to_RGB(hue, 1.0f, ratio, &r, &g, &b);
    Set_LED(i, r, g, b);
}
```

- Tính hue cho mỗi LED dựa vào vị trí và offset → tạo dải màu cầu vồng.
- HSV_to_RGB() chuyển đổi màu sang RGB để hiển thị.
- ratio điều khiển độ sáng của màu.

5.
```
int brightness = 5 + (int)(ratio * (brightness_mode - 5));
Set_Brightness(brightness);
WS2812_Send();
```

- Điều chỉnh độ sáng toàn dải theo biên độ âm thanh.
- Gửi dữ liệu đến dải LED để hiển thị hiệu ứng.

Kết quả: Dải LED hiển thị màu sắc chuyển động như cầu vồng, màu sắc và tốc độ thay đổi theo tần số âm thanh, độ sáng thay đổi theo cường độ âm thanh.

f. effect_bass_pulse_glow()

```
#define BASS_THRESHOLD 1200
#define BASS_FADE_MS 300

void effect_bass_pulse_glow(uint16_t amplitude, uint16_t peak_freq, uint8_t brightness_mode)
{
    static uint32_t last_bass_time = 0;
    static float hue = 0.0f;
    uint32_t now = HAL_GetTick();

    if (peak_freq <= 250 && amplitude > BASS_THRESHOLD) {
        last_bass_time = now;

        hue += 30.0f;
        if (hue >= 360.0f) hue -= 360.0f;
    }

    uint32_t elapsed = now - last_bass_time;
    int brightness;

    if (elapsed >= BASS_FADE_MS) {
        brightness = 0;
    } else {
        brightness = brightness_mode - (brightness_mode * elapsed) / BASS_FADE_MS;
    }

    uint8_t r, g, b;
    HSV_to_RGB(hue, 1.0f, 1.0f, &r, &g, &b);

    for (int i = 0; i < MAX_LED; i++) {
        Set_LED(i, r, g, b);
    }

    Set_Brightness(brightness);
    WS2812_Send();
}
```

Mục đích: Tạo hiệu ứng LED phát sáng nhấp nháy đổi màu theo nhịp bass trong âm thanh, với màu thay đổi dần và sáng mờ dần mềm mại.

Nguyên lý:

1. Khai báo:

```
#define BASS_THRESHOLD 1200
#define BASS_FADE_MS 300
```

- BASS_THRESHOLD: Ngưỡng biên độ âm bass để nhận biết tiếng trống mạnh.
- BASS_FADE_MS: Thời gian làm mờ ánh sáng sau khi có bass (ms).

2.
```
static uint32_t last_bass_time = 0;
 static float hue = 0.0f;
```

- last_bass_time: Thời điểm lần cuối phát hiện bass → dùng để tính độ mờ theo thời gian.
- hue: Tông màu hiện tại (giá trị từ 0 đến 360° trong hệ màu HSV) → chuyển màu cầu vồng.

3.
```
if (peak_freq <= 250 && amplitude > BASS_THRESHOLD) {
   last_bass_time = now;
   hue += 30.0f;
   if (hue >= 360.0f) hue -= 360.0f;
}
```

- Nếu tần số nằm trong vùng bass (≤ 250Hz) và cường độ lớn hơn ngưỡng → xem là tiếng bass mạnh.
- Cập nhật thời gian bass cuối cùng và thay đổi hue để màu chuyển dần theo cầu vồng.

4.
```
uint32_t elapsed = now - last_bass_time;
int brightness;
if (elapsed >= BASS_FADE_MS) {
   brightness = 0;
} else {
   brightness = brightness_mode - (brightness_mode * elapsed) / BASS_FADE_MS;
}
```

- Tính thời gian trôi qua kể từ tiếng bass gần nhất.
- Nếu đã quá BASS_FADE_MS, tắt sáng hoàn toàn.
- Nếu chưa, giảm sáng tuyến tính từ brightness_mode về 0.

5.
```
HSV_to_RGB(hue, 1.0f, 1.0f, &r, &g, &b);
```

- Chuyển màu sắc hue từ hệ HSV sang RGB.
- Saturation = 1.0 và Value = 1.0, nghĩa là luôn là màu thuần, độ sáng sẽ điều chỉnh sau.

6.
```
for (int i = 0; i < MAX_LED; i++) {
   Set_LED(i, r, g, b);
```

```
}
```

- Toàn bộ LED được thiết lập cùng một màu RGB vừa tính.

7. `Set_Brightness(brightness);`
   `WS2812_Send();`

- Gán độ sáng đã tính ở bước trước.
- Gửi dữ liệu đến dải WS2812B để hiển thị hiệu ứng.

Kết quả: LED sẽ phát sáng đổi màu khi có bass mạnh, sau đó độ sáng giảm dần tạo hiệu ứng nhấp nháy đồng bộ với âm thanh.

V. KẾT QUẢ VÀ KIỂM THỬ

1. Video thực tế mạch hoạt động (*)

2. Mô tả từng chức năng đã triển khai

Đọc tín hiệu âm thanh từ mic MAX9814: ADC lấy mẫu liên tục qua DMA, thu thập dữ liệu dạng sóng âm.

Xử lý tín hiệu FFT: Phân tích phổ âm thanh để phát hiện tần số và biên độ, phục vụ hiệu ứng ánh sáng.

Chuyển đổi ADC → PWM → LED: Giá trị ADC hoặc kết quả FFT được chuyển đổi sang tín hiệu PWM điều khiển WS2812B.

6 hiệu ứng ánh sáng:

effect_flash_fade_random_color()

effect_dynamic_vu_meter()

effect_spectrum_color_bands()

effect_frequency_chase_gradient()

effect_rainbow_roll()

effect_bass_pulse_glow()

Chuyển đổi hiệu ứng bằng nút nhấn: Hệ thống sử dụng 2 nút để đổi chế độ và độ sáng hiệu ứng.

3. Đánh giá độ hiệu quả:

Chức năng	Tốt	Bình thường	Không tốt
Ổn định tín hiệu PWM	**X**		
Độ trễ khi đổi hiệu ứng	**X**		
Độ chính xác xử lý ADC	**X**		
Khả năng xử lý dựa trên tín hiệu âm thanh	**X**		
Phản hồi từ nút nhấn	**X**		

4. So sánh với mục tiêu đề ra

Mục tiêu đề ra	Kết quả đạt được
Sử dụng STM32 điều khiển dải LED WS2812B	Đã triển khai thành công với điều khiển chính xác qua giao tiếp thời gian nghiêm ngặt
Hiển thị hiệu ứng ánh sáng đồng bộ với tín hiệu âm thanh	LED phản hồi tốt theo âm thanh, từ nhịp bass đến thay đổi tần số
Thay đổi màu sắc, độ sáng theo cường độ hoặc tần số âm thanh	Các hiệu ứng như effect_bass_pulse_glow, effect_frequency_chase_gradient… hoạt động đúng như mục tiêu
Hệ thống phản ứng với âm thanh theo thời gian thực, không có độ trễ lớn	Thời gian phản hồi nhanh, trễ không đáng kể nhờ kết hợp DMA và phân tích FFT
Minh họa ứng dụng ADC, DMA, PWM trong xử lý và thị giác hóa tín hiệu âm thanh	Thành công – toàn bộ hệ thống đều dùng ADC (đọc âm thanh), DMA (tăng tốc độ đọc), PWM (điều khiển LED)

Tổng kết: Hệ thống đã đáp ứng đầy đủ các mục tiêu kỹ thuật và chức năng đề ra ban đầu. Các hiệu ứng ánh sáng hoạt động mượt mà, đồng bộ với âm thanh đầu vào. Việc kết hợp các ngoại vi như ADC, DMA và PWM trên STM32 đã được khai thác hiệu quả, chứng minh tính khả thi và ứng dụng thực tiễn của đề tài.

VI. TÀI LIỆU THAM KHẢO

- Data sheet LED RGB: WS2812b

- Hướng dẫn code:

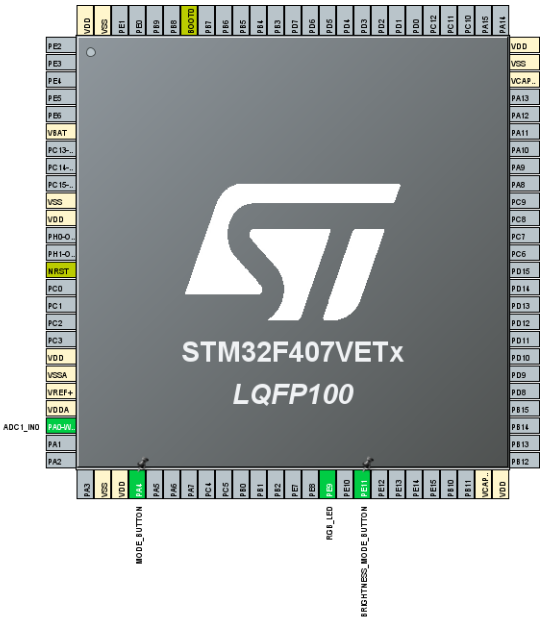
  - PWM & DMA
  - Sound sensor
  - ADC & Timer Trigger
  - FFT
  - Button

- Các hiệu ứng tham khảo:

  - effect1
  - effect2

VII. PHỤ LỤC

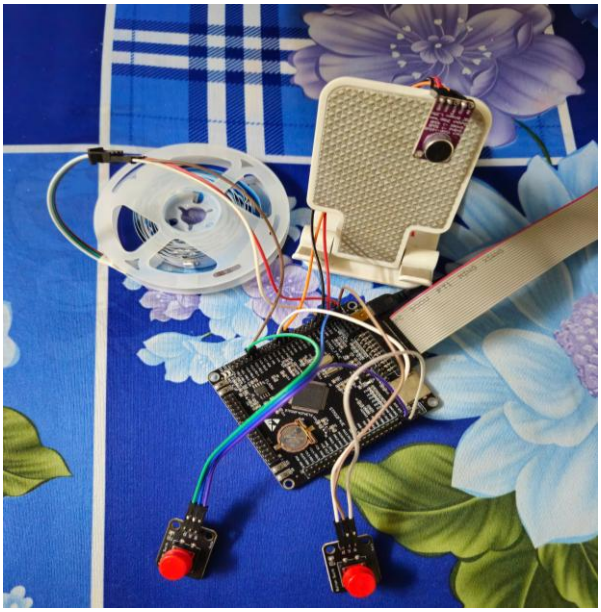
- Giao diện STM32CubeIDE:



VII-1 Mạch ioc và các chân GPIO

- Toàn bộ mã nguồn (Source Code): Link Github

- **(*)** Video thực tế mạch hoạt động: Link gg drive



VII-2 Mạch nối thực tế