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. 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, Timer/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 micro thoại **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.

➢ **Các công cụ và ngoại vi sử dụng trong dự án**

1. **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 phát PWM ở tần số chính xác 800 kHz.
- 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.

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

   DMA được dùng song song ở hai phần:

   - 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.
   - DMA cho ADC1: Đọc tín hiệu analog từ module micro MAX9814 một cách liên tục và ổn định, tránh mất mẫu khi lấy tín hiệu âm thanh.

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

3. **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. ADC1 được cấu hình:

   - Ở chế độ continuous conversion hoặc scan mode.
   - Có thể kết hợp với DMA để thu mẫu liên tục, tần số lấy mẫu đủ lớn (khoảng 8–20 kHz) để xử lý tín hiệu nhạc cơ bản.
   - Dữ liệu sau khi được lấy có thể dùng để trực tiếp điều chỉnh độ sáng, màu LED, hoặc làm đầu vào cho thuật toán xử lý tín hiệu (như Fast Fourier Transform – FFT).

4. **GPIO** – Kết nối vật lý với các thiết bị ngoại vi

   Các chân GPIO được sử dụng như sau:

   - PA8: TIM1_CH1, xuất tín hiệu PWM cho WS2812B.
   - PA0: ADC1_IN0, nhận tín hiệu từ MAX9814.
   - Các chân khác có thể dùng để điều khiển thêm nút bấm, chuyển chế độ hiệu ứng, hoặc đèn trạng thái.

5. **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,
   - Timer1 được cấp xung từ APB2, nhân đôi khi cần để đảm bảo có thể tạo được PWM 800 kHz.

- ADC được cấp xung từ APB2 hoặc chia xung phù hợp để đạt được tốc độ lấy mẫu mong muốn.

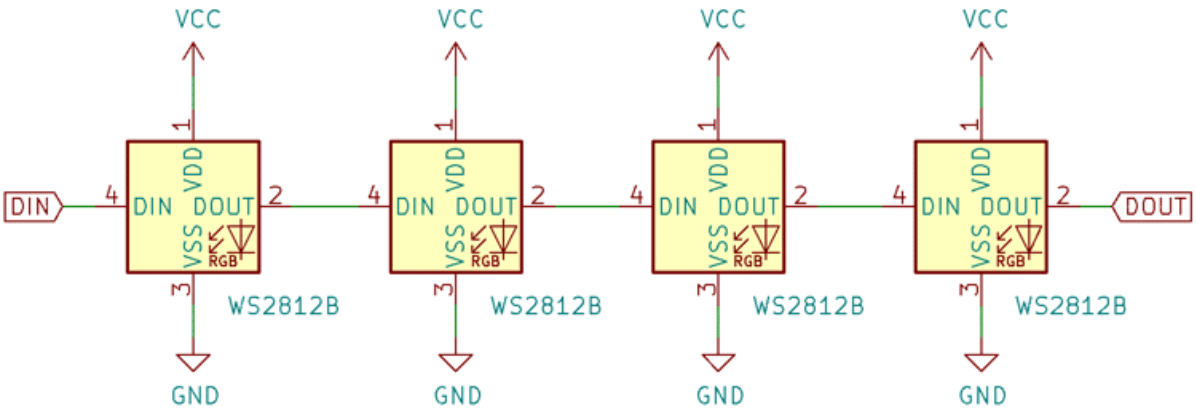
6. **NVIC** – Quản lý ngắt

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.

3. 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.

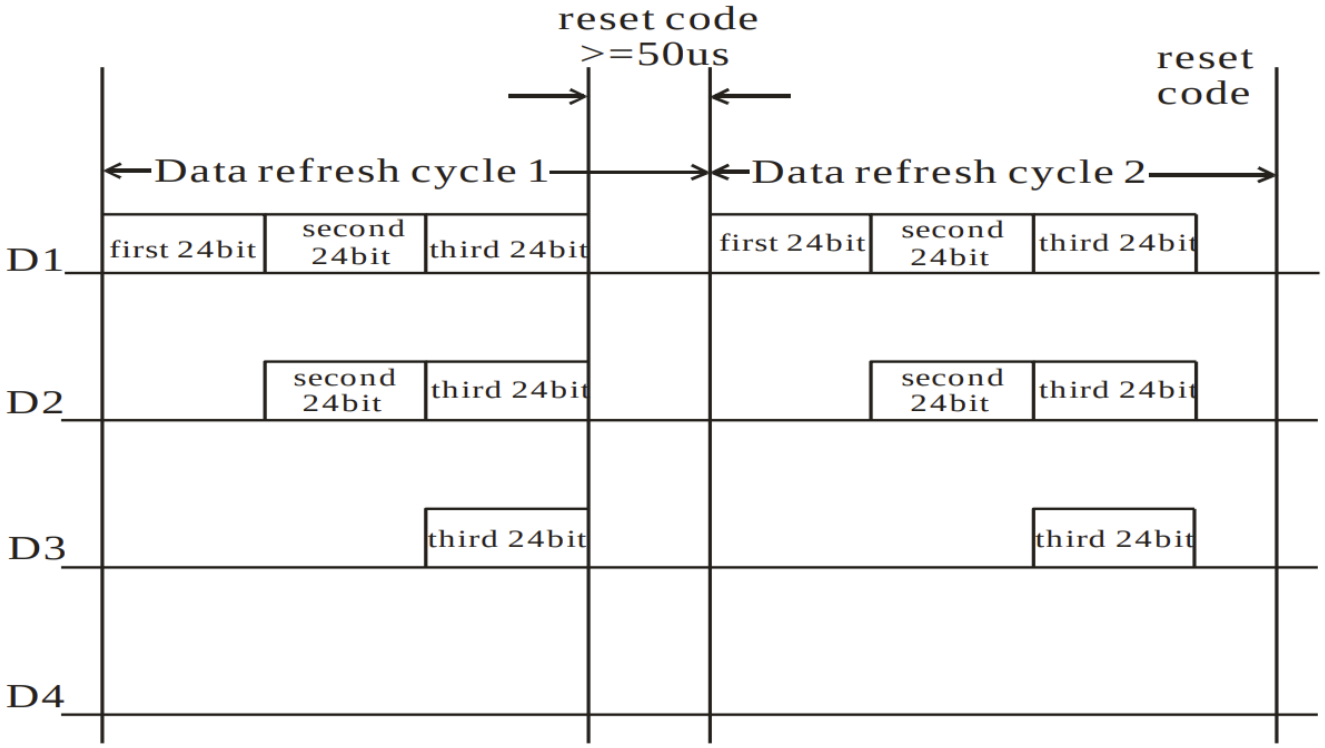


WS2812B sử dụng bộ điều chế độ rộng xung (PWM) để phân biệt giữa logic 0 và 1. Logic 1 yêu cầu độ rộng xung dài hơn, trong khi đó logic 0 yêu cầu độ rộng xung ngắn hơn. Tổng độ rộng xung là 1,25 µ s, đồng nghĩa với tần số là 800kHz, với các chu kỳ nhiệm vụ (duty cycles) tương ứng với logic 0 và 1 là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

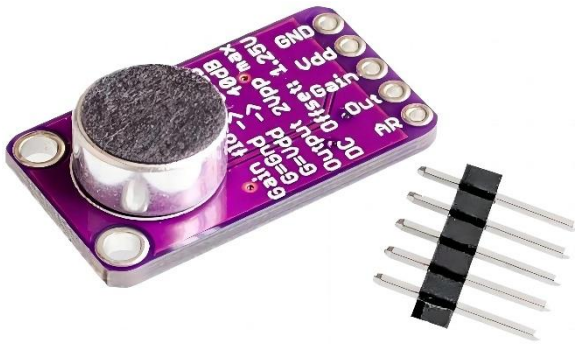
Dung sai của mỗi độ rộng xung là ± 150ns. Xung reset phải là 50 ms hoặc lâu hơn trước khi dữ liệu tiếp theo được đưa đến đèn LED.

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.



4. 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 dự án?**

1. Tự động điều chỉnh độ khuếch đại (AGC):

Khi âm thanh đầu vào (từ micro) có mức quá nhỏ, MAX9814 sẽ tự động tăng độ khuếch đại để tín hiệu đầu ra được nâng cao, tránh ADC đọc giá trị quá thấp (kém nhạy).

Ngược lại, nếu âm thanh đầu vào quá lớn, AGC sẽ giảm độ khuếch đại để đầu ra không bị saturate (cắt đỉnh) gây méo tín hiệu.

Nhờ AGC, tín hiệu đi vào ADC luôn nằm trong khoảng hoạt động tốt (tránh đầu ra quá bé hoặc quá lớn).

2. Bộ lọc nội tại:

MAX9814 tích hợp sẵn bộ lọc thông cao (high-pass filter) cắt tần số rất thấp (< 20 Hz) và tần số rất cao (> 20 kHz), loại bỏ tiếng ồn nền (như tần số lưới 50/60 Hz hoặc nhiễu sóng cao tần).

Kết quả: tín hiệu vào ADC sạch hơn, ít nhiễu, giúp việc tính biên độ âm thanh trung thực và chính xác hơn khi ánh xạ sang LED.

3. Độ nhạy cao, tần số phản hồi nhanh (~1 ms):

MAX9814 có thời gian phản hồi AGC nhanh, phù hợp cho việc đồng bộ ánh sáng theo nhạc (music visualization) yêu cầu tốc độ phản ứng nhanh khi biên độ âm thay đổi từng beat.

4. Kích thước nhỏ, dễ tích hợp:

Mạch thường được bào trên bo mạch kích thước ~14×20 mm, có các chân GND, VCC, OUT (tín hiệu), tùy chọn chân đặt mức gain (GAIN), rất dễ nối với STM32.

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

5. 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** (như STM32F103, STM32F407, v.v.) 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**

II. DMA, ADC VÀ CÁC HIỆU ỨNG LED :

1. DMA:

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.

```
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;  // 2/3 of 90
            }

            else pwmData[indx] = 30;  // 1/3 of 90

            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;  // 2/3 of 90
            }

            else pwmData[indx] = 30;  // 1/3 of 90

            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. 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:

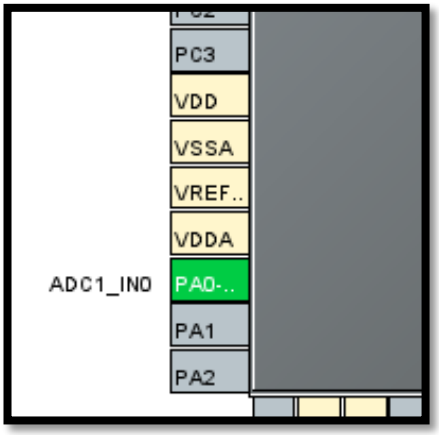
- Tốc độ lấy mẫu (sampling).
- Độ phân giải.

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; giá trị trung bình (≈ 1.65 V) tương ứng mã ~2048.
2. **Tốc độ chuyển đổi (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.
   - Trong hệ thống thu âm, thường chỉ cần sampling rate vài kHz (vd. 8 kHz hoặc 12 kHz) để bắt đủ dải âm thanh từ 20 Hz đến 20 kHz.
   - Tần số lấy mẫu được điều khiển bởi Timer hoặc dùng chế độ Continuous Conversion.

3. **Chế độ hoạt động với DMA**
   - Để tăng hiệu năng và giảm gánh nặng cho CPU, ADC thường được cấu hình ở chế độ **DMA Circular**. Khi ADC vừa hoàn thành một mẫu, DMA tự động ghi kết quả vào một vùng đệm (buffer) trong RAM, sau đó tiếp tục lấy mẫu tiếp mà không cần CPU can thiệp.
   - Hệ thống có thể đặt buffer gồm N mẫu (ví dụ N = 256), DMA ghi lần lượt các giá trị ADC vào buffer. Khi buffer đầy (Complete) hoặc nửa buffer đầy (Half-Complete), CPU được báo qua interrupt để xử lý một khối mẫu. Kỹ thuật này giúp CPU chỉ phải xử lý khối mẫu một lần thay vì phục vụ từng mẫu một, tránh trường hợp ngắt quá dày gây jitter.
4. **Chân ADC**
   - Thông thường, tín hiệu analog từ MAX9814 được đưa vào một chân ADC kênh đơn (ví dụ PA0 → ADC1_IN0).




```
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_DIV2;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  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();
  }
```

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

- Instance: ADC1
- ClockPrescaler: PCLK2 ÷ 2 (ADC clock ≈ 42 MHz nếu PCLK2 = 84 MHz)
- Resolution: 12 bit (kết quả 0…4095)
- ScanConvMode: DISABLE (chỉ 1 kênh)
- ContinuousConvMode: DISABLE (chuyển đổi khi phần mềm gọi)
- ExternalTrigConvEdge: NONE → dùng ADC_SOFTWARE_START
- 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: 28 cycles (khoảng 0,667 µs để tụ "lấy mẫu" ổn định)

Cấu hình này thiết lập ADC1 để ngang hàng đo một kênh (PA0) với độ phân giải 12 bit (0–4095), lấy mẫu khi phần mềm gọi (không liên tục hay DMA), dùng "software start" thay vì trigger từ timer, và mỗi lần đo tụ bên trong ADC sẽ giữ mẫu trong 28 chu kỳ ADC clock (~0,67 µs) để đảm bảo tín hiệu từ mạch MAX9814 ổn định trước khi chuyển đổi.

3. CÁC HÀM QUAN TRỌNG TRONG ĐIỀU KHIỂN 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()

```
void Set_Brightness (int brightness)  // 0-NORMAL_BRIGHTNESS
{
#if USE_BRIGHTNESS
    if (brightness > NORMAL_BRIGHTNESS) brightness = NORMAL_BRIGHTNESS;
    float scale = brightness / (float)NORMAL_BRIGHTNESS;

    for (int i = 0; i < MAX_LED; i++)
    {
        // preserve the “LED number” byte
        LED_Mod[i][0] = LED_Data[i][0];
        // scale each color channel linearly
        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: Nhận một giá trị brightness (thường 0–max). Mỗi giá trị màu trong LED_Data được chia cho một hệ số phụ thuộc brightness.
- 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) Tính bias với calculate_middle_point()

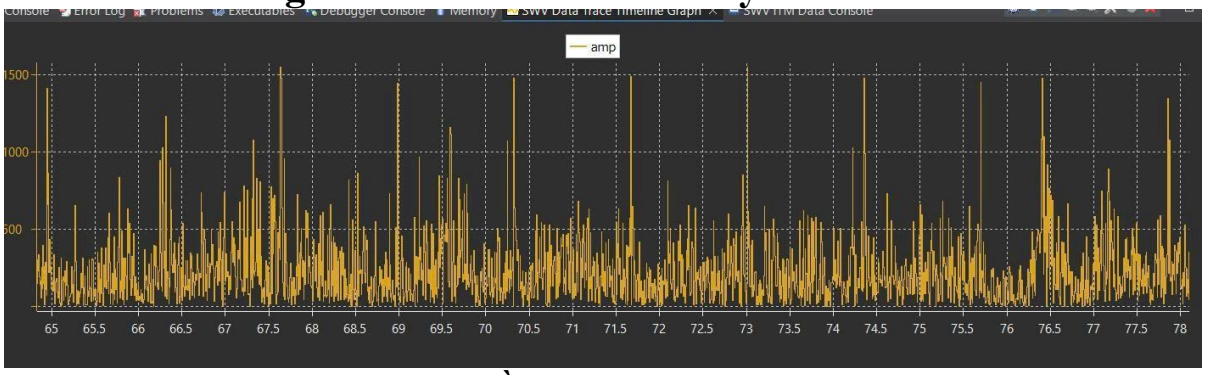
```
void calculate_middle_point(){
     uint32_t sum = 0;
     for (int i = 0; i < 32; i++){
           HAL_ADC_Start(&hadc1);
           if (HAL_ADC_PollForConversion(&hadc1, 10) == HAL_OK)
                 sum += HAL_ADC_GetValue(&hadc1);
           HAL_ADC_Stop(&hadc1);
           HAL_Delay(1);
       }

     middle_point = (uint16_t)(sum / 32);
}
```

- Mục đích: Tín hiệu âm thanh dao động xung quanh mức ~1.25 V, chứ không quanh 0 V. Để tính được "biên độ" thực, phải trừ đi bias (giá trị giữa không có âm).
- Nguyên lý hoạt động:
  1. Lặp 32 lần: mỗi lần:
     - Gọi HAL_ADC_Start(&hadc1) để khởi chuyển đổi.
     - Đợi HAL_ADC_PollForConversion(&hadc1, 10) (timeout 10 ms).
     - Đọc HAL_ADC_GetValue(&hadc1) → một giá trị 0…4095.
     - Gọi HAL_ADC_Stop(&hadc1) rồi HAL_Delay(1) (giãn 1 ms để tín hiệu ổn định).
     - Cộng dồn vào sum.
  2. Sau 32 mẫu, middle_point = sum / 32 → giá trị trung bình.
- Kết quả : middle_point (khoảng 1500…1600) lưu trong biến toàn cục.

d) get_amp()

- Mục đích:Tính biên độ âm thanh hiện tại.
- Hoạt động:
  Đọc 1 giá trị ADC hiện tại.
  Trừ cho middle_point → lấy độ lệch (biên độ).
  Dùng abs() để bỏ dấu âm.
- Vai trò: Tạo ra giá trị biên độ (mức dao động của âm thanh) → **dùng để điều khiển LED nháy theo âm thanh**.



Minh họa về vai trò của hàm get_amp()

e) Set

```
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);          // purple
    }
}
```

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().

f) 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 (0,0,0) tức là tắt. Sau đó đặt lại độ sáng và gửi dữ liệu LED.	Toàn bộ LED được tắt ngay lập tức, không còn phát sáng.

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

h) get_rainbow_color()

```
void get_rainbow_color(uint16_t index, uint16_t effStep, uint8_t *red, uint8_t *green, uint8_t *blue)
{
    // Compute a "phase" value in [0, 59], matching the original rainbow_effect logic
    int16_t temp = (int16_t)(effStep - index * 1.2f);
    int16_t mod  = temp % 60;            // may be negative or positive
    if (mod < 0) mod += 60;              // ensure 0 ≤ mod < 60
    uint16_t ind = 60 - mod;             // ind in [1, 60]
    ind = ind % 60;                      // now in [0, 59]

    // Determine which third of the 60-step "wheel" we're in
    uint16_t segment = (ind % 60) / 20;  // 0 → red→green, 1 → green→blue, 2 → blue→red

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

    switch (segment) {
        case 0:
            // Transition red → green
            // factor1 goes from 1 → 0 over 20 steps; factor2 goes from 0 → 1
            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:
            // Transition green → blue
            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:
            // Transition blue → red
            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;
    }

    // Store results
    *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.

4. CÁC HÀM HIỆU ỨNG ĐIỀU KHIỂN LED:

a. effect_sound_color()

```
void effect_sound_color() {
    float ratio = (float)amp / amp_maxn;
    if (ratio > 1.0) ratio = 1.0;
    if (ratio <= 0.05f) {
     Turn_off_all_at_once();
     WS2812_Send();
     return;
    }

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

    uint8_t r, g, b;

    if (ratio < 0.5f) {
        float t = ratio / 0.5f;
        r = (uint8_t)(0 + t * (148 - 0));
        g = 0;
        b = (uint8_t)(255 + t * (211 - 255));
    } else {
        float t = (ratio - 0.5f) / 0.5f;
        if (t < 0.5f) {
            float t2 = t / 0.5f;
            r = 255;
            g = (uint8_t)(0 + t2 * (127 - 0));
            b = 0;
        } else {
            float t2 = (t - 0.5f) / 0.5f;
            r = 255;
```

```
            g = (uint8_t)(127 + t2 * (255 - 127));
            b = 0;
        }
    }

    for (int i = 0; i < MAX_LED; i++) {
        Set_LED(i, r, g, b);
    }
    WS2812_Send();
}
```

Mục đích:
- Tất cả LED đổi màu đồng thời theo biên độ âm thanh.

Nguyên lý hoạt động:
1. Tính tỉ lệ âm thanh:
```
float ratio = (float)amp / amp_maxn;
```
- Chuẩn hóa amp thành ratio trong khoảng 0.0–1.0.
- Giới hạn ratio = 1.0 nếu vượt ngưỡng.

2. Kiểm tra ngưỡng tắt LED:
```
if (ratio <= 0.05f) {
    Turn_off_all_at_once();
    WS2812_Send();
    return;
}
```
- Nếu âm lượng ≤ 5% → tắt hết LED và thoát hàm.

3. Cài đặt độ sáng LED:
```
int brightness = 5 + (int)(ratio * (NORMAL_BRIGHTNESS - 5));
Set_Brightness(brightness);
```
- Tính brightness từ 5 đến NORMAL_BRIGHTNESS tỉ lệ với ratio để LED mờ hoặc rực.

4. Tính màu LED theo 2 vùng:
- Vùng 1 (ratio < 0.5): màu xanh → tím
```
float t = ratio / 0.5f;
r = (uint8_t)(0 + t * 148);
g = 0;
b = (uint8_t)(255 + t * (211 - 255));
```
- Khi t tăng 0→1: R tăng 0→148, B giảm 255→211, G = 0.

- Vùng 2 (ratio ≥ 0.5): màu tím → đỏ → vàng
```
float t = (ratio - 0.5f) / 0.5f;
```

```
if (t < 0.5f) {
    float t2 = t / 0.5f;
    r = 255;
    g = (uint8_t)(0 + t2 * 127);
    b = 0;
} else {
    float t2 = (t - 0.5f) / 0.5f;
    r = 255;
    g = (uint8_t)(127 + t2 * (255 - 127));
    b = 0;
}
```

- Khi t từ 0→1: G tăng 0→127 rồi 127→255, R = 255, B = 0.

5. Cập nhật màu lên toàn bộ LED:

```
for (int i = 0; i < MAX_LED; i++) {
    Set_LED(i, r, g, b);
}
WS2812_Send();
```

- Gọi Set_LED để gán màu cho từng LED, sau đó WS2812_Send để xuất dữ liệu.

Hiệu ứng tạo ra:

- LED sáng mờ khi âm nhỏ, sáng rực khi âm to.
- Màu LED chuyển dần từ xanh lam → tím → đỏ → vàng theo mức âm lượng.
- Phản hồi gần như tức thời, chuyển màu mềm mại.

b. ripple_effect()

```
void ripple_effect(uint16_t amplitude) {
    float ratio = (float)amplitude / amp_maxn;
    if (ratio > 1.0f) ratio = 1.0f;

    if (ratio <= 0.05f) {
        Turn_off_all_at_once();
        WS2812_Send();
        return;
    }

    int center = MAX_LED / 2;
    int spread = 1 + (int)(ratio * (MAX_LED / 2));

    for (int i = 0; i < MAX_LED; i++) {
        int dist = abs(i - center);
        float brightness = 1.0f - ((float)dist / spread);
        if (brightness < 0.0f) brightness = 0.0f;

        // Get rainbow color based on position
        uint8_t r, g, b;
        get_rainbow_color(i, effStep, &r, &g, &b);
```

```
        // Apply brightness modulation to color
        Set_LED(i, (uint8_t)(r * brightness), (uint8_t)(g * brightness), (uint8_t)(b * brightness));
    }

    Set_Brightness(NORMAL_BRIGHTNESS);
    WS2812_Send();

    effStep = (effStep + 1) % 60;
}
```

Mục đích:

- Tạo hiệu ứng "gợn sóng ánh sáng" lan tỏa từ giữa dải LED theo biên độ âm thanh.

Nguyên lý hoạt động:

1. Tính tỉ lệ âm thanh:

   float ratio = (float)amplitude / amp_maxn;

   - Giới hạn ratio ≤ 1.0.

2. Nếu âm thanh nhỏ (≤ 5%):

   Turn_off_all_at_once()
   WS2812_Send()
   return;

   - Tắt hết LED khi im lặng.

3. Xác định tâm và độ lan tỏa (spread):

   int center = MAX_LED / 2;
   int spread = 1 + (int)(ratio * (MAX_LED / 2));

   - Tâm: vị trí giữa dải.
   - Spread tăng theo ratio để sóng càng lan xa khi âm to.

4. Với mỗi LED (index i):

   int dist = abs(i - center); // khoảng cách đến tâm
   float brightness = 1.0f - (dist / (float)spread);
   if (brightness < 0.0f) brightness = 0.0f;

   - Độ sáng giảm dần khi LED càng xa tâm.

   get_rainbow_color(i, effStep, &r, &g, &b);
   // Lấy màu cầu vồng dựa trên thứ tự LED và bước effStep.

   Set_LED(i, (uint8_t)(r * brightness), (uint8_t)(g * brightness), (uint8_t)(b * brightness));

- Nhân màu cầu vồng với độ sáng để LED gần tâm rực, xa tâm mờ.

5. Set_Brightness(NORMAL_BRIGHTNESS)
   WS2812_Send();
   - Cố định độ sáng toàn cục, xuất dữ liệu.

6. effStep = (effStep + 1) % 60;
   - Tăng bước hiệu ứng để màu cầu vồng di chuyển mượt.

Hiệu ứng tạo ra:
- Ánh sáng cầu vồng lan tỏa từ giữa dải LED.
- Độ sáng giảm theo khoảng cách, tạo hiệu ứng gợn sóng.
- Màu chuyển động mượt qua từng bước effStep.

c. sound_bar_hue_gradient()

```
void sound_bar_hue_gradient(uint16_t amplitude) {
    static int current_led_count = 1;
    int target_led_count;

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

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

    if (amplitude < 100 && current_led_count > 1) {
        current_led_count--;
    } else if (target_led_count > current_led_count) {
        current_led_count++;
    }

    Turn_off_all_at_once();

    for (int i = 0; i < current_led_count; i++) {
        float t = (float)i / (current_led_count - 1);
        float hue = 270.0f * (1.0f - t); // tím (270°) → đỏ (0°)

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

    Set_Brightness(NORMAL_BRIGHTNESS);
    WS2812_Send();
}
```

Mục đích:
- Tạo hiệu ứng "cột đo âm thanh" với dải màu chuyển sắc từ tím → đỏ.

Nguyên lý hoạt động:
1. Tính tỉ lệ âm thanh:
   float ratio = (float)amplitude / amp_maxn;
   - Giới hạn ratio ≤ 1.0.

2. Tính số LED cần sáng (target_led_count):
   target_led_count = (int)(ratio * MAX_LED);
   - Ít nhất 1 LED nếu âm thanh ≥ ngưỡng.
3. Điều chỉnh dần số LED hiện tại (current_led_count):
   - Nếu amplitude < 100 và current_led_count > 1 → giảm dần.
   - Nếu target_led_count > current_led_count → tăng dần.
4. Tắt hết LED (Turn_off_all_at_once()).
5. Với mỗi LED index i trong current_led_count:
   float t = i / (current_led_count - 1);
   float hue = 270.0f * (1.0f - t); // hue từ 270° → 0°
   HSV_to_RGB(hue, 1.0f, 1.0f, &r, &g, &b);
   Set_LED(i, r, g, b);
   - LED đầu có hue = 270° (tím), đến LED cuối có hue = 0° (đỏ).
6. Set_Brightness(NORMAL_BRIGHTNESS)
   WS2812_Send();
   - Gửi dữ liệu đã cập nhật.

Hiệu ứng tạo ra:
- Một "cột sáng" LED từ trái sang phải, số lượng LED hiển thị tỉ lệ với âm lượng.
- Màu sắc chuyển từ tím sang đỏ dọc theo cột, dễ theo dõi cường độ âm thanh.
- Số LED sáng tăng/giảm mượt.

d. effect_random_one_in_six_leds_by_sound()

```
void effect_random_one_in_six_leds_by_sound(uint16_t amplitude) {
    float ratio = (float)amplitude / amp_maxn;
    if (ratio > 1.0f) ratio = 1.0f;

    // Ignore low amplitude (silence)
    if (ratio <= 0.05f) {
        Turn_off_all_at_once();
        WS2812_Send();
        return;
    }

    Turn_off_all_at_once();

    const int group_size = 6; //per n Led will have 1 random led lighted
    for (int start = 0; start < MAX_LED; start += group_size) {
        int rand_index = rand() % group_size;
        int led_index = start + rand_index;
        if (led_index >= MAX_LED) break;

        // Optional: brighter when louder
        uint8_t brightness = (uint8_t)(ratio * 255);
        uint8_t r = brightness;
        uint8_t g = rand() % brightness;
        uint8_t b = rand() % brightness;

        Set_LED(led_index, r, g, b);
    }
```

```
    Set_Brightness(5 + (int)(ratio * (NORMAL_BRIGHTNESS - 5)));
    WS2812_Send();
}
```

Mục đích:

- Tạo hiệu ứng "nhấp nháy ngẫu nhiên" theo âm thanh.

Nguyên lý hoạt động:

1. Tính tỉ lệ âm thanh (ratio) tương tự.
2. Nếu âm thanh ≤ 5% → tắt hết LED.
3. Turn_off_all_at_once() để xóa hiệu ứng cũ.
4. Chia dải LED thành nhóm 6 LED (group_size = 6).

```
for (int start = 0; start < MAX_LED; start += group_size) {
    int rand_index = rand() % group_size;
    int led_index = start + rand_index;
    if (led_index >= MAX_LED) break;

    uint8_t brightness = (uint8_t)(ratio * 255);
    uint8_t r = brightness;
    uint8_t g = rand() % brightness;
    uint8_t b = rand() % brightness;

    Set_LED(led_index, r, g, b);
}
```

- Mỗi nhóm chọn 1 LED ngẫu nhiên sáng.
- Màu: R = độ sáng phụ thuộc ratio, G và B ngẫu nhiên trong 0→brightness.

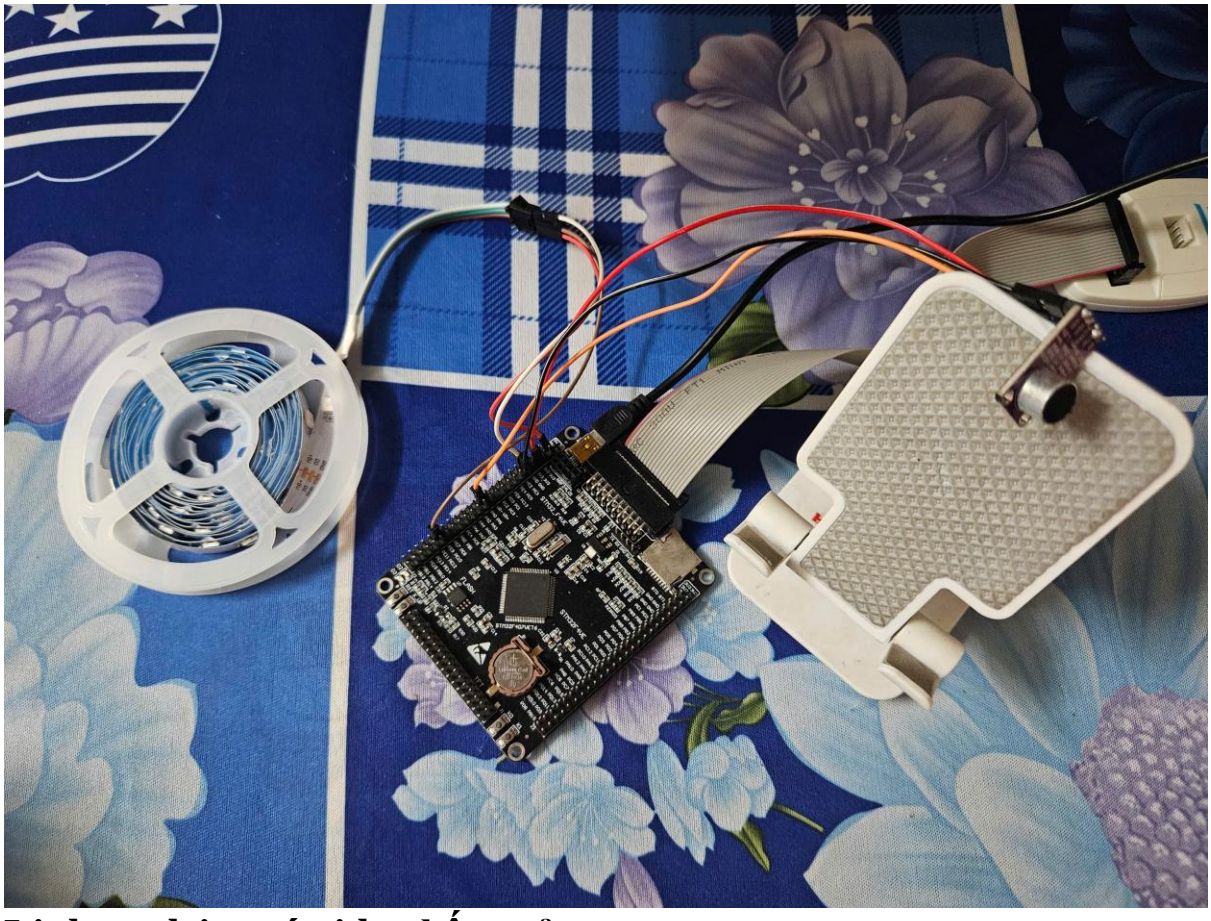
5. `Set_Brightness(5 + (ratio * (NORMAL_BRIGHTNESS - 5)));`
   `WS2812_Send();`
   - Điều chỉnh độ sáng tổng thể và xuất dữ liệu.

Hiệu ứng tạo ra:

- LED sẽ chớp ngẫu nhiên từng nhóm, giống đom đóm hoặc sao băng theo âm thanh.
- Khi âm thanh lớn, màu sắc và độ sáng rực hơn.

III. HÌNH ẢNH ,VIDEO KẾT QUẢ:

Hình ảnh mạch thực tế :



Link gg drive có video kết quả :
https://drive.google.com/drive/folders/16WzMkUMokDoD2AtBF4_X8ZynyhBq74Ls