ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KỸ THUẬT MÁY TÍNH

---------☆------ ☙-❧ -------☆---------

BẢNG LED RGB (NEOPIXEL) VỚI HIỆU ỨNG LIGHT SHOW

ĐỒ ÁN MÔN VI XỬ LÝ – VI ĐIỀU KHIỂN

Dương Thanh Hiếu – 23520475

Trần Triệu Dân – 23520223

Thành phố Hồ Chí Minh 5/2025

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA KỸ THUẬT MÁY TÍNH

Dương Thanh Hiếu – 23520475
Trần Triệu Dân – 23520223

BẢNG LED RGB (NEOPIXEL) VỚI HIỆU ỨNG LIGHT SHOW

ĐỒ ÁN MÔN VI XỬ LÝ – VI ĐIỀU KHIỂN

GIÁO VIÊN HƯỚNG DẪN:
Trần Ngọc Đức

Thành phố Hồ Chí Minh 5/2025

MỤC LỤC

DANH MỤC BẢNG

DANH MỤC HÌNH ẢNH

LỜI NÓI ĐẦU

Trong xu thế phát triển mạnh mẽ của khoa học kỹ thuật, đặc biệt là trong lĩnh vực điều khiển tự động và điện tử nhúng, các hệ thống hiển thị bằng ánh sáng thông minh đang ngày càng trở nên phổ biến và đóng vai trò quan trọng trong đời sống hiện đại. Các thiết bị LED RGB địa chỉ như NeoPixel không chỉ được ứng dụng trong công nghiệp, quảng cáo mà còn góp phần tạo nên các hiệu ứng nghệ thuật, giải trí và trang trí linh hoạt, sống động.

Hình 1: Ứng dụng của LED trong đời sống

Từ niềm yêu thích với lĩnh vực điện tử lập trình, cùng với mong muốn được áp dụng kiến thức lý thuyết vào thực tế, em đã lựa chọn đề tài: "Sử dụng vi điều khiển STM32 để điều khiển bảng LED RGB (NeoPixel) với hiệu ứng Light Show" cho đồ án kết thúc môn. Đây là một đề tài vừa mang tính học thuật, vừa có tính thực tiễn cao, cho phép khai thác tối đa khả năng xử lý của vi điều khiển STM32 cũng như sự phong phú của các hiệu ứng ánh sáng hiện đại.

Trong suốt quá trình thực hiện đồ án, em đã có cơ hội củng cố kiến thức về vi điều khiển STM32, giao tiếp ngoại vi, lập trình C nhúng, xử lý tín hiệu và thiết kế hệ thống phần cứng - phần mềm tích hợp. Đồng thời, em cũng rèn luyện được kỹ năng tư duy logic, giải quyết vấn đề và làm việc độc lập.

LỜI CẢM ƠN

Chúng em xin chân thành cảm ơn quý thầy cô trong bộ môn đã tận tình giảng dạy và hướng dẫn trong suốt thời gian học tập. Đặc biệt, chúng em xin gửi lời cảm ơn sâu sắc đến thầy Trần Ngọc Đức đã dành thời gian, kiến thức và kinh nghiệm quý báu để giúp chúng em hoàn thành tốt đồ án này.

Chúng em cũng xin gửi lời cảm ơn đến các bạn bè, người thân đã luôn đồng hành, hỗ trợ và động viên chúng em trong suốt thời gian học tập và làm đồ án. Những chia sẻ, góp ý và khích lệ của mọi người chính là nguồn động lực lớn giúp chúng em vượt qua những khó khăn, thử thách trong quá trình thực hiện.

Chương 1: Tổng quan

1.1. Giới thiệu đồ án

Trong thời đại công nghệ số phát triển mạnh mẽ, các hệ thống hiển thị ánh sáng thông minh đang ngày càng được ứng dụng rộng rãi trong nhiều lĩnh vực như quảng cáo, trình diễn nghệ thuật, trang trí sự kiện và nhà thông minh. Một trong những công nghệ nổi bật đó là dải LED RGB địa chỉ (NeoPixel), cho phép điều khiển từng bóng LED một cách độc lập để tạo ra các hiệu ứng ánh sáng rực rỡ và sống động.

Đồ án này tập trung vào việc thiết kế và xây dựng một hệ thống điều khiển bảng LED RGB (NeoPixel) sử dụng vi điều khiển STM32, với mục tiêu tạo ra các hiệu ứng Light Show hấp dẫn và có thể tùy biến dễ dàng. Vi điều khiển STM32 được lựa chọn vì sở hữu tốc độ xử lý cao, tài nguyên phong phú và hỗ trợ tốt các giao tiếp ngoại vi, rất phù hợp cho việc điều khiển các LED đòi hỏi tốc độ truyền dữ liệu chính xác như WS2812.

1.2. Thiết kế hệ thống

1.2.1. Kiến trúc hệ thống

Để đảm bảo hệ thống điều khiển bảng LED RGB (NeoPixel) hoạt động ổn định, linh hoạt và dễ bảo trì, cấu trúc phần mềm được xây dựng theo mô hình ba lớp (Three-Layer Architecture). Cách tiếp cận này cho phép tách biệt rõ ràng giữa phần cứng, xử lý nghiệp vụ và giao diện người dùng, giúp việc phát triển, kiểm thử và mở rộng hệ thống trở nên dễ dàng hơn.:

- Hardware Layer: STM32F407VET6, LCD ILI9341, WS2812, MAX9814.
- Bussiness Layer: thư viện xử lý.
- Presentation Layer: giao diện người dùng.

1.2.2. Mô tả các thành phần trong hệ thống

STT	Thành phần	Diễn giải
1	Hardware Layer	Đây là tầng vật lý, chứa các thiết bị phần cứng mà hệ thống tương tác trực tiếp. Tầng này cung cấp dữ liệu

		đầu vào (âm thanh), thực hiện xuất dữ liệu (hiển thị, LED), và là nền tảng cho toàn bộ hệ thống hoạt động. • Vi điều khiển STM32F407VET6: Là bộ xử lý trung tâm của hệ thống, đóng vai trò điều khiển toàn bộ hoạt động, xử lý dữ liệu đầu vào và đầu ra. • Màn hình LCD ILI9341: Người dùng có thể quan sát trạng thái hệ thống, chọn chế độ hiệu ứng ánh sáng, điều chỉnh tốc độ hoặc độ sáng thông qua màn hình cảm ứng hoặc nút nhấn. • Dải LED WS2812 (NeoPixel): Là thành phần đầu ra chính tạo nên các hiệu ứng ánh sáng. Mỗi LED WS2812 có thể điều khiển màu và độ sáng độc lập thông qua một giao tiếp dữ liệu duy nhất, giúp thiết kế mạch đơn giản nhưng hiệu ứng hiển thị rất phong phú. • Microphone MAX9814: Dùng để thu tín hiệu âm thanh môi trường. Tín hiệu này sau đó sẽ được đưa vào ADC của vi điều khiển để phân tích và phản hồi bằng hiệu ứng ánh sáng tương ứng với âm thanh (ví dụ: nháy theo nhạc).
2	Bussiness Layer	Tầng trung gian chứa các thuật toán và thư viện xử lý. Đây là nơi mọi logic xử lý được thực hiện, giúp hệ thống hoạt động đúng theo yêu cầu chức năng. • Thư viện điều khiển WS2812: Xử lý việc truyền tín hiệu theo chuẩn thời gian nghiêm ngặt của WS2812 bằng PWM hoặc DMA.

		• Thư viện xử lý âm thanh: Lọc và phân tích tín hiệu từ microphone MAX9814, có thể sử dụng kỹ thuật trung bình trượt, FFT (biến đổi Fourier nhanh) để phát hiện cường độ và tần số âm thanh. • Thuật toán hiệu ứng ánh sáng (Light Show Engine): Bao gồm các hiệu ứng như nháy theo nhạc, sóng màu, chạy đuổi, pha màu, v.v... Các hiệu ứng này có thể thay đổi theo thời gian, theo âm thanh hoặc theo lựa chọn của người dùng. • Bộ điều khiển màn hình (ILI9341): Thư viện vẽ và cập nhật giao diện trên LCD như hiển thị chế độ hiện tại, mức âm thanh, tốc độ hiệu ứng,...
3	Presentation Layer	Tầng tương tác giữa hệ thống và người sử dụng. Giúp người dùng quan sát hoặc tương tác với hệ thống một cách trực quan. • Giao diện hiển thị trên màn hình LCD ILI9341: Hiển thị thông tin hệ thống, hiệu ứng đang chạy, mức âm thanh thu được, trạng thái của dải LED,... • Các nút cảm ứng: Cho phép người dùng thay đổi hiệu ứng, điều chỉnh thông số và chuyển chế độ thủ công một cách thuận tiện.

Bảng 1: Các thành phần trong hệ thống

Chương 2: Cấu tạo phần cứng và sơ đồ kết nối

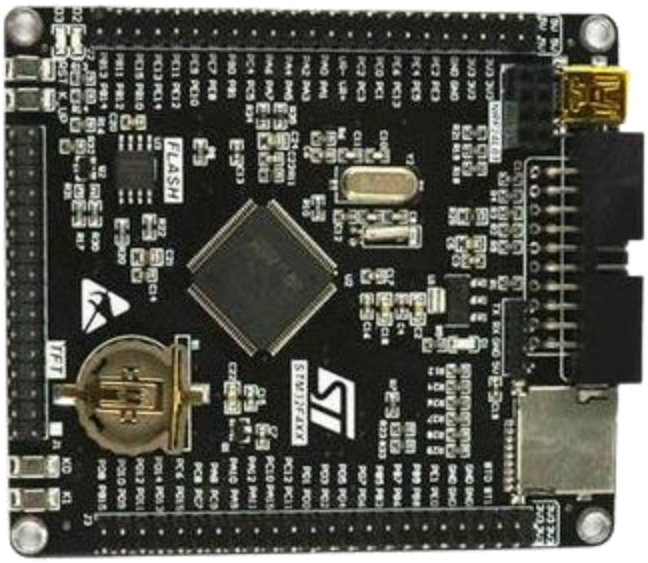
2.1. Thông tin chi tiết về linh kiện trong mạch

2.1.1. Khối điều khiển trung tâm STM32F407VET6 (Black board)

a. STM32F407VET6 (Black board) là gì ?

STM32F407VET6 là một vi điều khiển (microcontroller) thuộc dòng STM32F4 series của hãng STMicroelectronics, dựa trên kiến trúc ARM Cortex-M4. Đây là một MCU mạnh mẽ, được sử dụng phổ biến trong các hệ thống nhúng yêu cầu xử lý tín hiệu nhanh, điều khiển thời gian thực, hoặc ứng dụng đồ họa, âm thanh, và giao tiếp ngoại vi phức tạp.

Bo mạch này được trang bị lõi ARM Cortex-M4 tốc độ 168 MHz, bộ nhớ Flash 512 KB và RAM 192 KB, tích hợp nhiều tính năng mạnh mẽ như ADC 12-bit, DAC, PWM, UART, SPI, I2C, CAN, USB OTG, và Ethernet. Với thiết kế 2 hàng chân header dễ cắm trên breadboard, mạch có thể cấp nguồn qua USB hoặc chân 3.3V và thường đi kèm LED, nút nhấn reset, thạch anh 8 MHz và 32.768 kHz. Model được trang bị tổng cộng 100 chân (LQFP100), trong đó có khoảng 82 chân có thể cấu hình làm I/O.



Hình 2: STM32F407VET6 Black Board

b. Thông số kỹ thuật

Vi điều khiển (MCU)	STM32F407VET6 – ARM Cortex-M4, FPU, DSP
Tốc độ xung nhịp (Clock)	168 MHz
Bộ nhớ Flash	512 KB
Bộ nhớ SRAM	192 KB (112 KB CCM + 64 KB SRAM1 + 16 KB SRAM2)
Số chân I/O	~82 chân GPIO (trong tổng số 100 chân LQFP100)
Giao tiếp UART/USART	Lên đến 6 cổng USART/UART
Giao tiếp SPI	3 cổng SPI
Giao tiếp I2C	3 cổng I2C
ADC	3 bộ ADC 12-bit, tổng cộng 24 kênh
DAC	2 kênh DAC 12-bit
PWM / Timer	17 bộ định thời (timer), gồm cả advanced và general-purpose
USB	USB OTG FS (thiết bị/phụ kiện hoặc chủ)
CAN	1 cổng CAN
SDIO	Có (giao tiếp thẻ nhớ SD)
Ethernet MAC	Có sẵn (cần PHY ngoài để hoạt động)
RTC (Real-Time Clock)	Có, với thạch anh ngoài 32.768 kHz (PC14, PC15)
Thạch anh HSE	8 MHz (xung ngoài chính)
Thạch anh LSE	32.768 kHz (RTC)

Hình 3: Thông số STM32F407VET6 Black Board

c. Cấp nguồn và các chân nguồn

❖ **Cấp nguồn**

Bo mạch STM32F407VET6 Black Board có thể được cấp nguồn linh hoạt qua nhiều cách khác nhau. Thông thường, nguồn 5V sẽ được cung cấp qua cổng USB (micro USB hoặc USB Type-C), sau đó được chuyển đổi xuống 3.3V bằng mạch ổn áp tích hợp trên bo để cấp điện cho vi điều khiển và các linh kiện khác. Ngoài ra, người dùng cũng có thể cấp nguồn trực tiếp qua chân VIN hoặc chân 5V trên header nếu sử dụng nguồn ngoài như adapter hoặc pin. Trong trường hợp đã có nguồn 3.3V ổn định từ bên ngoài, có thể cấp trực tiếp vào chân 3.3V của board mà không cần qua bộ chuyển đổi. Các chân GND được bố trí để làm mát và nối đất chung cho toàn hệ thống.

❖ **Các chân nguồn**

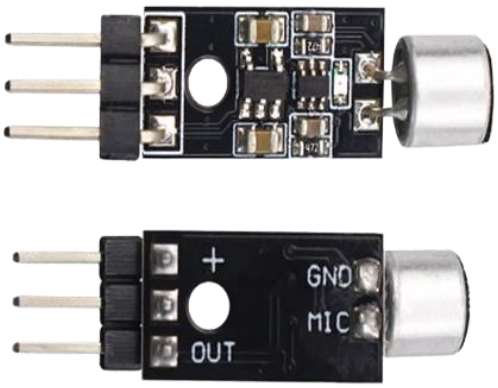
- ✓ 5V (VUSB hoặc VIN): Chân này nhận nguồn 5V từ cổng USB hoặc nguồn ngoài, dùng làm đầu vào cho mạch ổn áp trên board.
- ✓ 3.3V: Chân cấp nguồn 3.3V đã được ổn áp, dùng để cung cấp điện trực tiếp cho vi điều khiển và các thiết bị ngoại vi hoạt động ở mức điện áp thấp.
- ✓ GND (Ground): Các chân nối đất chung, rất quan trọng để tạo mạch hoàn chỉnh và ổn định điện áp.
- ✓ VBAT: Chân cấp nguồn cho pin dự phòng (nếu có), dùng để duy trì hoạt động của bộ đồng hồ thực (RTC) khi nguồn chính tắt.

2.1.2. Khối điều khiển Module MAX9814

a. MAX9814 là gì ?

MAX9814 là một mạch khuếch đại micro tích hợp sẵn bộ lọc tự động (AGC - Automatic Gain Control), được thiết kế để khuếch đại tín hiệu âm thanh thu từ micro nhỏ một cách rõ nét và ổn định. Đây là một chip chuyên dụng giúp tăng cường chất lượng tín hiệu âm thanh đầu vào mà không làm méo tiếng, rất phù hợp cho các ứng

dụng như thu âm, nhận dạng giọng nói, hoặc các dự án xử lý âm thanh trên vi điều khiển.



Hình 4: MAX9814

b. Thông số kỹ thuật

Điện áp cấp nguồn (VDD)	2.7V – 5.5V
Dòng điện tiêu thụ	~440 µA
Băng thông tín hiệu	50 Hz – 20 kHz
Tỷ lệ tín hiệu trên nhiễu (SNR)	58 dB (typical)
Tỷ lệ méo hài tổng (THD)	0.03%
Mức độ khuếch đại (Gain)	40dB, 50dB, 60dB
Đầu ra	Analog
Kích thước IC	3mm x 3mm

Hình 5: Thông số MAX9814

c. Cấp nguồn và các chân nguồn

❖ **Cấp nguồn**

Mạch khuếch đại micro **MAX9814** có thể được cấp nguồn linh hoạt từ 2.7V đến 5.5V, phổ biến nhất là sử dụng mức 3.3V hoặc 5V tùy theo hệ thống vi điều khiển đang sử dụng.

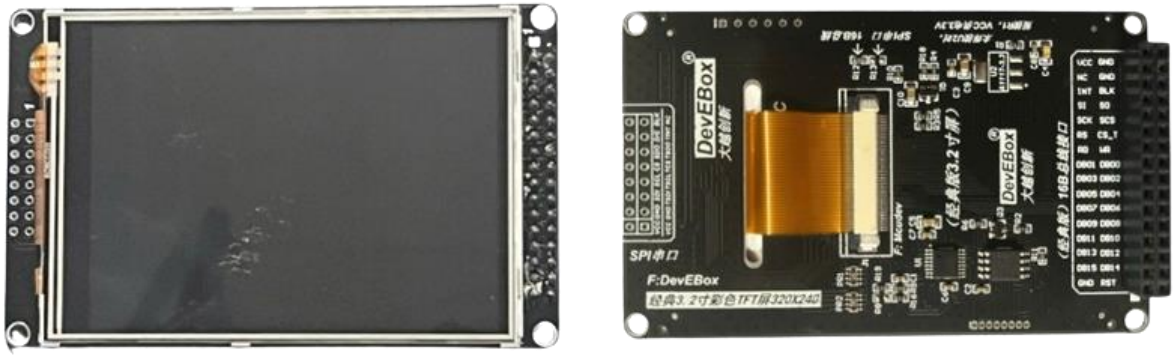
❖ **Các chân nguồn**

- ✓ VDD: Chân này nhận nguồn 3V từ vi điều khiển
- ✓ GND: Chân này phải được nối đất chung với vi điều khiển STM32 để đảm bảo tín hiệu ổn định.
- ✓ OUT: Tín hiệu âm thanh đã khuếch đại sẽ được xuất ra tại chân này dưới dạng analog,

2.1.3. Màn hình LCD ILI9341

a. LCD ILI9341 là gì ?

LCD ILI9341 là một loại màn hình màu TFT phổ biến, sử dụng driver **ILI9341** do hãng Ilitek sản xuất. Đây là màn hình được sử dụng rộng rãi trong các dự án nhúng, vi điều khiển (như STM32, Arduino, ESP32...) để hiển thị giao diện đồ họa, văn bản hoặc hình ảnh.



Hình 6: LCD ILI9341

b. Thông số kỹ thuật

Kích thước màn hình	2.2", 2.4", 2.8", 3.2"... (phổ biến nhất là 2.4")
Độ phân giải	240 × 320 pixel (QVGA)
Công nghệ hiển thị	TFT LCD – Màn hình màu
IC điều khiển	ILI9341 (Tích hợp sẵn trên màn hình)
Giao tiếp	SPI (phổ biến), đôi khi hỗ trợ cả 8/16-bit Parallel
Điện áp hoạt động	3.3V logic (cần chuyển mức khi dùng với 5V MCU)
Màu hiển thị	65K – 262K màu (16-bit)
Cảm ứng	Một số phiên bản có tích hợp cảm ứng điện trở (resistive touch) hoặc điện dung (capacitive touch)

Hình 7: Thông số LCD ILI9341

c. Cấp nguồn và các chân nguồn

❖ Cấp nguồn

LCD ILI9341 thường hoạt động với điện áp 3.3V cho cả nguồn cấp và tín hiệu điều khiển, tuy nhiên một số module bán sẵn trên thị trường được tích hợp sẵn mạch ổn áp (thường dùng IC AMS1117), cho phép cấp nguồn 5V vào chân VCC hoặc VIN. Trong trường hợp không có mạch ổn áp tích hợp, màn hình chỉ nên được cấp 3.3V trực tiếp để tránh hư hỏng. Ngoài ra, các chân điều khiển như SCK, MOSI, CS, DC và RESET chỉ chấp nhận mức logic 3.3V. Đèn nền của màn hình (chân LED) cũng có thể cấp 3.3V hoặc 5V tùy theo thiết kế module.

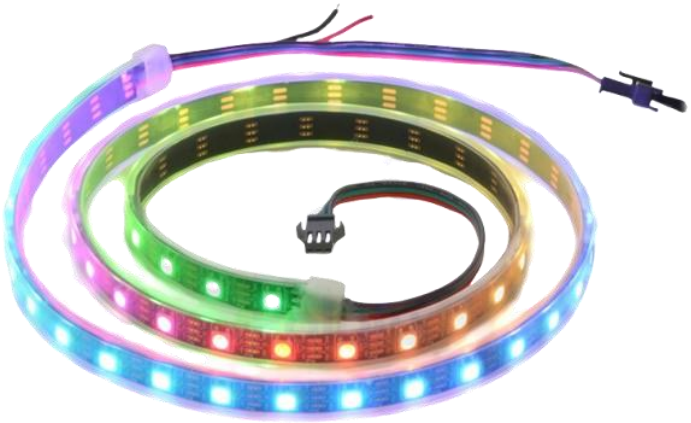
❖ Các chân nguồn

- ✓ VCC / VIN: Cấp nguồn chính cho module LCD (có thể qua ổn áp tích hợp)
- ✓ GND: Nối đất (mass) – bắt buộc chung với GND của vi điều khiển
- ✓ LED: Cấp nguồn cho đèn nền màn hình (backlight)

2.1.4. Dải đèn LED WS2812

a. WS2812 là gì ?

WS2812 là một loại LED RGB thông minh tích hợp mạch điều khiển bên trong từng chip LED, cho phép điều khiển màu sắc và độ sáng của từng đèn riêng lẻ chỉ bằng một đường tín hiệu dữ liệu duy nhất (1-Wire).



Hình 8: WS2812B

b. Thông số kỹ thuật

Kích thước	5.0mm x 5.0mm
Điện áp hoạt động	5V DC
Dòng tối đa mỗi LED	~60mA (tối đa khi sáng trắng ở độ sáng cao nhất)
Giao tiếp	1 dây (1-Wire), mức logic 5V
Độ sáng	Điều chỉnh 256 mức (8-bit) cho từng màu R/G/B
Thời gian truyền dữ liệu	~1.25 µs mỗi bit (800 Kbps)

Hình 9: Thông số WS2812

c. Cấp nguồn và các chân nguồn

❖ **Cấp nguồn**

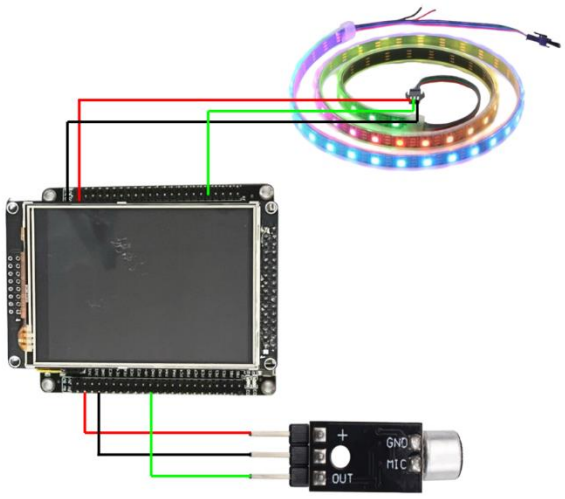
WS2812 hoạt động với điện áp cấp 5V DC ổn định và có mức tiêu thụ dòng khoảng 60mA cho mỗi bóng LED khi phát sáng ở mức sáng trắng tối đa. Do đó, nếu sử dụng một dải gồm 8 LED WS2812, nguồn cấp cần đảm bảo cung cấp ít nhất 500mA để hệ thống hoạt động ổn định.

❖ **Các chân nguồn**

- ✓ VCC: Cấp nguồn 5V DC ổn định, vì mỗi LED WS2812 hoạt động chuẩn ở 5V.
- ✓ GND: Nối với mass (GND) chung của hệ thống, ví dụ GND của STM32.
- ✓ DIN (Data): Nhận tín hiệu điều khiển từ vi điều khiển (STM32, Arduino,...)

2.2. Sơ đồ kết nối

2.2.1. Sơ đồ kết nối mạch



Hình 10: Sơ đồ kết nối mạch

2.2.2. Sơ đồ kết nối dây

STM32F407VET6	MAX9814	LCD ILI9341	WS2812
3.3V	RED	VCC	
5V			RED
GND	BLACK	GND	BLACK

PA0	GREEN		
PC6			GREEN
NC		NC	
PC_5		INT	
PB_1		BLK	
PB_15		SI	
PB_14		SO	
PB_13		SCK	
PB_12		SCS	
PD_13		RS	
PD_7		CS_T	
PD_4		RD	
PD_5		WR	
...		DB00 – DB15	
NRST		RST	

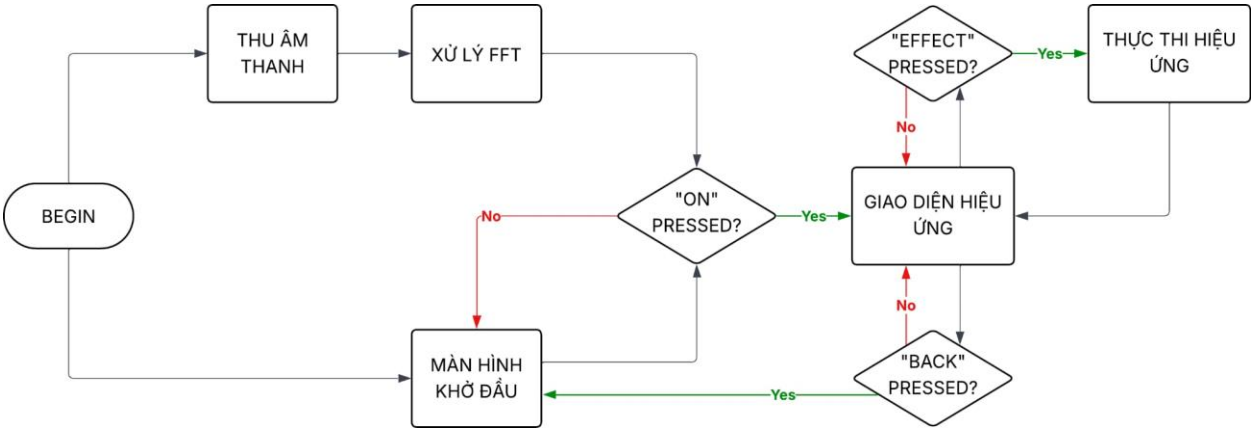
Bảng 2: Sơ đồ kết nối dây

2.3. Danh sách các linh kiện cần chuẩn bị

- ✓ 1 Vi điều khiển STM32F407VET6 Black board
- ✓ 1 Dải đèn LED WS2812B
- ✓ 1 Mạch thu và khuếch đại âm thanh MAX9814
- ✓ 1 màn hình LCD ILI9341
- ✓ dây đực – cái, cái – cái
- ✓ 1 ST - Links

Chương 3: Chương trình điều khiển

3.1. Lưu đồ hoạt động và nguyên lý của mạch



Hình 11: Lưu đồ hoạt động

❖ **Nguyên lý hoạt động:** Hệ thống bắt đầu bằng việc thu âm thanh từ môi trường, sau đó xử lý tín hiệu âm thanh bằng thuật toán FFT (Fast Fourier Transform) để phân tích các thành phần tần số. Dựa trên kết quả phân tích, người dùng có thể bật/tắt chế độ hiệu ứng, chọn hiệu ứng, và thực thi hiệu ứng ánh sáng tương ứng với âm thanh hoặc theo lựa chọn của người dùng bằng LCD.

3.2. Cài đặt trên STM32CubeIDE

3.2.1. System core

Thành phần	Giá trị
SYS	
Debug	Disable
Timebase Source	Sys Tick
RCC	
High speed clock (HSE)	Crystal/Ceramic Resonator
Low speed clock (LSE)	Disable

Bảng 3: Cài đặt System Core

3.2.2. Clock Configuration

Thành phần	Giá trị
Input frequency	8
PLL Source MUX	HSE
System Clock MUX	PLLCLK
HCLK	72
APB2 Prescaler	1
APB1peripheral clocks	72
APB2 Prescaler	2
APB2 peripheral clocks	36

Bảng 4: Cài đặt Clock Configuration

3.2.3. Timer

❖ **Timer 2**

Thành phần	Giá trị
Clock Source	Internal Clock
Channel1	PWM Generation CH1
Prescaler	71
Counter Mode	Up
Counter Period	99
Auto – reload preload	Enable
Master/Salve Mode	Enable
Trigger Event Selection	Update Event

Bảng 5: Cài đặt Timer 2

❖ **Tiner 3**

Thành phần	Giá trị
Clock Source	Internal Clock

Channel1	Disable
Prescaler	0
Counter Mode	Up
Counter Period	89
Auto – reload preload	Enable
GPIO Settings	PC6

Bảng 6: Cài đặt Timer 3

3.2.4. ADC

Thành phần	Giá trị
Mode	IN0
ADCs_Common_Settings	Independent Mode
Clock Prescaler	PCLK2 devided by 2
Resolution	12 bits (15 ADC Clock cycles)
Data Alignment	Right
Continous Conversion Mode	Enable
DMA Continous Requests	Enable
Number of Conversion	1
External Trigger Conversion Source	Timer 2 Trigger Out event
External Trigger Conversion Edge	Trigger detection on the rising edge
Channel	Channel 0
Sampling Time	15 Cycles
GPIO	PA0

Bảng 7: Cài đặt ADC

3.2.5. DMA

Thành phần	Giá trị
TIMER 3	
DMA Request	TIM3_CH1/TRIG

Stream	DMA1 Stream 4
Direction	Memory To Peripheral
Priority	Low
Mode	Normal
ADC1	
DMA Request	ADC1
Stream	DMA2 Stream 0
Direction	Peripheral To Memory
Priority	Low
Mode	Circular

Bảng 8: Cài đặt DMA

3.2.6. FSCM

Thành phần	Giá trị
Mode	NOR Flash/PSRAM/SRAM/ROM/LCD 1
Memory type	LCD interface
LCD Register Select	A18
Data	16
Address setup time in HCLK clock cycles	15
Data setup time in HCLK clock cycles	255
Bus turn around time in HCLK clock cycles	15

Bảng 9: Cài đặt FSCM

3.2.7. SPI

Thành phần	Giá trị
Mode	Full-Duplex Master

Hardware NSS Signal	Disable

Bảng 10: Cài đặt SPI

3.3. Thư viện xử lý

3.3.1. Thu âm thanh

a. Nguyên lý hoạt động

Microphone MAX9814 cung cấp một tín hiệu âm thanh dạng analog đã được khuếch đại. Tín hiệu này dao động theo biên độ của âm thanh và thường có điện áp dao động quanh mức trung bình (ví dụ 1.65V nếu Vcc = 3.3V). STM32 sử dụng ADC1 để liên tục đo giá trị điện áp tại chân đầu vào analog, và chuyển đổi giá trị điện áp này thành số nguyên (digital value) có độ phân giải 12-bit (tức là từ 0 đến 4095).

Tuy nhiên, nếu CPU phải đọc từng mẫu ADC bằng tay (polling hoặc interrupt), hiệu suất sẽ kém và không đảm bảo tốc độ lấy mẫu liên tục. Vì vậy, chương trình sử dụng DMA để cho phép ADC gửi dữ liệu trực tiếp vào mảng adc_buffer[] mà không cần CPU làm gì cả trong quá trình truyền.

Khi sử dụng ADC để phân tích tín hiệu âm thanh bằng FFT, nếu không giới hạn tần số lấy mẫu, hệ thống có thể hoạt động ở tần số quá cao so với nhu cầu thực tế. Với Sampling Time = 15 cycles và ADC Clock = 36 MHz, tần số lấy mẫu có thể lên tới khoảng 1.3 MHz, nghĩa là theo định lý Nyquist, ta có thể phân tích tín hiệu đến khoảng 645 kHz.

$$Sampling\ Frequency = \frac{36000000}{15 + 12.5} \approx 1.3MHz$$

Tuy nhiên, dải tần số cần thiết cho phân tích âm thanh thông thường chỉ nằm trong khoảng 0–5000 Hz, nên việc lấy mẫu ở tần số cao như vậy là không cần thiết, gây lãng phí tài nguyên và có thể ảnh hưởng đến độ chính xác của FFT do chứa nhiều thông tin dư thừa. Để đảm bảo lấy mẫu ở tần số phù hợp hơn, ví dụ 10 kHz, ta cần sử dụng cơ chế kích hoạt bằng Timer cho ADC. Timer trigger cho phép lấy mẫu

chính xác và ổn định theo thời gian, đảm bảo dữ liệu đưa vào FFT phản ánh đúng tín hiệu âm thanh cần phân tích.

b. Đoạn code chính:

- ADC_HandleTypeDef hadc1;
  - Cấu trúc dùng để quản lý cấu hình và trạng thái của ADC1.
  - Được khởi tạo trong MX_ADC1_Init() trong CubeMX hoặc thủ công.
- DMA_HandleTypeDef hdma_adc1;
  - Cấu trúc dùng để quản lý DMA kết nối với ADC1.
  - Được cấu hình để truyền dữ liệu ADC sang RAM.
- #define DMA_BUFFER_SIZE 1024
  - Kích thước bộ đệm: 1024 mẫu 16-bit.
  - Bạn đang lấy mẫu ADC với độ phân giải 12 bit, nên dùng uint16_t là hợp lý.
- uint16_t adc_buffer[1024] = {0};
  - Mảng lưu trữ các mẫu ADC thu được từ tín hiệu âm thanh.
  - DMA sẽ tự động ghi dữ liệu ADC vào đây mỗi lần lấy mẫu.
- HAL_ADC_Start_DMA(...)
  - Kích hoạt ADC và DMA cùng lúc.
  - Khi hoạt động:
    - ADC1 liên tục chuyển đổi tín hiệu analog (từ MAX9814).
    - DMA tự động ghi dữ liệu vào adc_buffer[].
  - Sau khi bộ đệm đầy, sẽ có:
    - Callback HAL_ADC_ConvCpltCallback(): khi DMA ghi xong toàn bộ.
    - (Hoặc HAL_ADC_ConvHalfCpltCallback() nếu bạn dùng kỹ thuật "double buffering").

```
ADC_HandleTypeDef hadc1;
DMA_HandleTypeDef hdma_adc1;

#define DMA_BUFFER_SIZE 1024
uint16_t adc_buffer[1024] = {0};

HAL_ADC_Start_DMA(&hadc1, (uint32_t *)adc_buffer, DMA_BUFFER_SIZE);
```

3.3.2. Thư viện WS2812.c

a. Nguyên lý hoạt động

- **Giao tiếp với WS2812:** Mỗi LED WS2812 sử dụng một giao tiếp nối tiếp đơn giản chỉ qua một chân dữ liệu (gọi là single-wire). Tuy nhiên, điểm đặc biệt là mỗi LED cần chính xác 24 bit dữ liệu để hiển thị một màu, trong đó bao gồm 8 bit cho màu xanh lá (Green), 8 bit cho màu đỏ (Red) và 8 bit cho màu xanh dương (Blue), theo đúng thứ tự GRB. Dữ liệu được truyền từ vi điều khiển xuống LED theo dạng nối tiếp tuần tự, LED đầu tiên sẽ nhận 24 bit đầu tiên, sau đó truyền tiếp các bit còn lại cho các LED phía sau.

- **Phân biệt bit 0 và 1 theo độ rộng xung:** Điểm cốt lõi trong giao tiếp WS2812 là bit 0 và bit 1 không phân biệt bằng mức điện áp, mà phân biệt thông qua thời gian giữ mức cao (logic 1) và mức thấp (logic 0) trong một chu kỳ xung kéo dài khoảng 1.25 micro giây (µs). Cụ thể, để biểu diễn một bit 1, vi điều khiển cần giữ chân dữ liệu ở mức cao khoảng 0.7 µs rồi xuống thấp 0.6 µs. Trong khi đó, để biểu diễn bit 0, mức cao chỉ cần giữ khoảng 0.35 µs rồi xuống thấp 0.9 µs.

- **STM32 sử dụng PWM và DMA để tạo xung chính xác:** Do yêu cầu thời gian rất nghiêm ngặt như trên, việc tạo xung bằng delay phần mềm (dùng for, HAL_Delay...) là không đáng tin cậy và dễ sai lệch nếu có ngắt hoặc tác vụ khác đang chạy. Vì thế, trong thư viện này, STM32 sử dụng Timer ở chế độ PWM kết hợp DMA để tạo ra chuỗi xung chính xác. PWM sẽ tạo ra các xung vuông có độ rộng (duty cycle) được

lập trình sẵn – ví dụ duty cycle khoảng 66% để biểu diễn bit 1, hoặc 33% để biểu diễn bit 0. DMA sẽ lo việc truyền từng giá trị độ rộng đó đến Timer mà không cần CPU can thiệp nhiều, đảm bảo truyền liên tục và chính xác đến từng micro giây.

❖ **Cách mã hóa dữ liệu màu thành xung PWM:** Thư viện trước tiên sẽ lấy từng màu RGB từ mảng dữ liệu LED_Data[], sau đó hợp nhất thành một số 24 bit kiểu color với thứ tự Green → Red → Blue. Tiếp theo, từng bit trong color sẽ được duyệt từ trái sang phải (MSB đến LSB). Nếu bit hiện tại là 1, giá trị tương ứng trong mảng pwmData[] sẽ là 60 (tương đương mức cao dài); nếu là 0, giá trị sẽ là 30 (mức cao ngắn). Giá trị này tương ứng với độ rộng xung (duty) trong PWM trên Timer. Mỗi LED sẽ chiếm 24 phần tử trong mảng pwmData[].

❖ **Truyền toàn bộ dữ liệu bằng DMA và tạo reset:** Sau khi đã tạo xong mảng pwmData[] chứa toàn bộ dữ liệu của các LED, thư viện sẽ gọi HAL_TIM_PWM_Start_DMA() để truyền mảng đó đến Timer 3 (TIM3) qua kênh DMA. Khi DMA hoạt động, nó sẽ gửi liên tục từng xung PWM tương ứng cho từng bit màu mà không ngắt quãng, đảm bảo mỗi LED nhận đúng dữ liệu. Cuối chuỗi dữ liệu, thư viện thêm vào khoảng 50 xung 0 (tức là mức thấp kéo dài >50 μs), đây chính là tín hiệu reset bắt buộc để WS2812 bắt đầu hiển thị màu mới.

b. Các hàm chức năng

❖ **Set_LED(int LEDnum, int Red, int Green, int Blue):**

- Gán giá trị RGB cho LED tại vị trí LEDnum.
- Thứ tự trong mảng LED_Data:
    - LED_Data[LEDnum][0] = LEDnum;
    - LED_Data[LEDnum][1] = Green;
    - LED_Data[LEDnum][2] = Red;
    - LED_Data[LEDnum][3] = Blue;

```
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;
}
```

❖ **Get_LED(int LEDnum, uint8_t *r, uint8_t *g, uint8_t *b):**

- Truy xuất màu hiện tại của một LED và lưu kết quả vào các biến đầu ra con trỏ.

```
void Get_LED(int LEDnum, uint8_t *r, uint8_t *g, uint8_t *b)
{
    *r = LED_Data[LEDnum][2];
    *g = LED_Data[LEDnum][1];
    *b = LED_Data[LEDnum][3];
}
```

❖ **Reset_All_LED():**

- Đặt tất cả LED về màu đen (0, 0, 0).
- Thiết lập lại độ sáng về mặc định (13).
- Gửi dữ liệu đến WS2812 bằng WS2812_Send().

```
void Reset_All_LED()
{
    for (int i = 0 ; i < MAX_LED ; i++)
    {
        Set_LED(i, 0, 0, 0);
    }
    Set_Brightness(13);
    WS2812_Send();
}
```

❖ **Set_Brightness(int brightness):**

- Làm mượt độ sáng bằng cách giảm giá trị RGB theo hàm tan()**:**

- Khi brightness tăng, angle giảm → tan(angle) nhỏ lại → chia ra số lớn hơn → giá trị LED nhỏ hơn → LED tối hơn (vì giảm độ sáng).
- Đây là cách làm phi tuyến tính để tăng độ mượt thay vì chia đơn thuần.

```
void Set_Brightness (int brightness)
{
    if (brightness > 45) brightness = 45;
    for (int i=0; i<MAX_LED; i++)
    {
        LED_Mod[i][0] = LED_Data[i][0];
        for (int j=1; j<4; j++)
        {
            float angle = 90-brightness;
            angle = angle*PI / 180;
            LED_Mod[i][j] = (LED_Data[i][j])/(tan(angle));
        }
    }
}
```

❖ **WS2812_Send():**

- Ghép 3 màu thành một số 24-bit theo chuẩn GRB.
- Mã hóa từng bit thành xung PWM:
  - Bit 1: độ rộng PWM dài hơn (60 → duty cycle cao hơn)
  - Bit 0: độ rộng PWM ngắn hơn (30 → duty cycle thấp hơn)
- Reset: Gửi thêm 50 xung bằng 0 (~50 µs) để reset dải LED.
- Truyền bằng DMA:
  - Dữ liệu PWM được truyền tự động bằng DMA từ mảng pwmData[] đến Timer.
  - Khi DMA kết thúc, nó gọi callback để đặt datasentflag = 1.

```
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(&htim3, TIM_CHANNEL_1, (uint32_t *)pwmData, indx);
    while (!datasentflag){};
    datasentflag = 0;
}
```

3.3.3. Thư viện FFT.c

a. Nguyên lý hoạt động

❖ **Chuyển đổi tín hiệu sang miền tần số bằng FFT:**

Sau khi bộ chuyển đổi ADC đã thu thập đủ dữ liệu mẫu từ tín hiệu âm thanh (với số lượng điểm mẫu bằng FFT_SIZE), hệ thống chuyển sang giai đoạn xử lý tín hiệu bằng thuật toán FFT (Fast Fourier Transform). Thuật toán này có nhiệm vụ biến đổi tín hiệu từ miền thời gian—nơi các giá trị phản ánh biên độ tại mỗi thời điểm—sang miền tần số, nơi mỗi giá trị thể hiện cường độ (năng lượng) của một tần số cụ thể xuất hiện trong tín hiệu.

Việc thực hiện FFT trong chương trình sử dụng thư viện CMSIS-DSP, đặc biệt là hàm arm_rfft_fast_f32, giúp rút ngắn thời gian xử lý và tận dụng tối đa phần cứng FPU của STM32F4 để thực hiện các phép toán dấu chấm động (float) hiệu quả hơn. Kết quả trả về là một mảng gồm các số thực và ảo (real + imaginary), thể hiện dưới dạng cặp liên tiếp trong mảng fft_output. Sau đó, hệ thống tính biên độ (magnitude) của mỗi tần số bằng công thức căn bậc hai của tổng bình phương phần thực và phần ảo. Mỗi giá trị biên độ này phản ánh mức độ xuất hiện (hay năng lượng) của một tần số trong tín hiệu âm thanh đầu vào.

❖ **Phân tích năng lượng theo dải tần (Bass, Mid, Treble)**

Sau khi có dữ liệu FFT hoàn chỉnh, bước tiếp theo là phân tích và phân loại phổ tần thành ba vùng cơ bản thường gặp trong âm thanh: bass, mid và treble. Hệ thống chia toàn bộ phổ tần (từ 0 Hz đến một nửa tần số lấy mẫu - do FFT đối xứng) thành ba khoảng:

- Bass (âm trầm): thường từ 0 đến khoảng 250 Hz
- Mid (âm trung): từ 250 Hz đến khoảng 4000 Hz
- Treble (âm cao): từ 4000 Hz trở lên

Việc phân chia này dựa vào chỉ số tần số tương ứng với mỗi bin FFT, được tính theo công thức: f = index * (sampling_freq / FFT_SIZE). Trong quá trình phân tích, mỗi giá trị biên độ trong phổ được chuẩn hóa bằng hàm logarit (log10(magnitude + 1.0)) để giảm ảnh hưởng của những giá trị quá lớn và làm nổi bật các thay đổi nhỏ trong âm thanh.

❖ **Tạo hiệu ứng đèn LED theo âm thanh:**

Khi hệ thống đã có dữ liệu năng lượng cho từng dải tần (bass, mid, treble) và mức năng lượng tổng thể của tín hiệu, bước cuối cùng là áp dụng các thông tin đó để điều khiển dải LED WS2812B. Thư viện cung cấp nhiều kiểu hiệu ứng ánh sáng sinh động, mỗi kiểu được thiết kế để phản hồi theo một khía cạnh cụ thể của âm thanh:

- **Light Bar**: tăng số LED sáng theo cường độ âm thanh.
- **Alternating Flash**: nháy màu xen kẽ với tốc độ thay đổi theo âm lượng.
- **EchoSplit**: Ánh sáng tỏa ra như tiếng vang, chia đôi từ hai hướng.
- **Sparkle**: tạo hiệu ứng nhấp nháy ngẫu nhiên.

b. Các hàm chức năng

❖ **void Calculate_FFT():**

- Chuyển đổi dữ liệu ADC sang phổ tần bằng FFT và tính năng lượng trung bình.
    - Chuyển adc_buffer[] (kiểu uint16_t) sang fft_input[] (kiểu float).
    - Thực hiện FFT bằng arm_rfft_fast_f32.
    - Tính biên độ (magnitude) cho từng bin bằng căn bậc hai của ($real^2 + imag^2$).
    - Tính giá trị trung bình của toàn bộ phổ FFT (fft_output_average), dùng để đo âm lượng tổng thể.

```
void Calculate_FFT(void)
{
    for (int i = 0 ; i < FFT_SIZE ; i++)
    {
        fft_input[i] = (float)adc_buffer[i];
    }

    arm_rfft_fast_f32(&S, fft_input, fft_output, 0);
    for (int i = 0; i < FFT_SIZE / 2; i++)
    {
        float real = fft_output[2*i];
        float imag = fft_output[2*i + 1];
        fft_output[i] = sqrtf(real * real + imag * imag);
        fft_output_average += fft_output[i];
    }
    fft_output_average /= (FFT_SIZE / 2 - 1);
}
```

❖ **void Analyze_Frequency_Bands(float* bass, float* mid, float* treble):**

- Chia phổ tần thành ba vùng: bass, mid, treble, và chuẩn hóa năng lượng từng vùng.
    - Duyệt toàn bộ phổ FFT (chỉ nửa đầu do FFT đối xứng).
    - Dựa trên tần số mỗi bin (i * bin_freq) để phân loại vào bass, mid hoặc treble.
    - Dùng log10(mag + 1) để nén dữ liệu và loại bỏ ảnh hưởng của cực trị.
    - Tính tổng năng lượng và chuẩn hóa ba vùng theo phần trăm.

```
void Analyze_Frequency_Bands(float* bass, float* mid, float* treble)
{
    *bass = *mid = *treble = 0;

    float bin_freq = SAMPLING_FREQ / FFT_SIZE;
    float total_energy = 0.0f;

    for (int i = 0; i < FFT_SIZE / 2; i++)
    {
        float freq_output = i * bin_freq;
        float magnitude = fft_output[i];
        float norm_mag = log10f(magnitude + 1.0f);

        if (freq_output <= BASS_FREQ_MAX)
            *bass += norm_mag;
        else if (freq_output <= MID_FREQ_MAX)
            *mid += norm_mag;
        else if (freq_output <= TREBLE_FREQ_MAX)
            *treble += norm_mag;
        else
            break;
    }

    total_energy = *bass + *mid + *treble;
    if (total_energy > 0)
    {
        *bass /= total_energy;
        *mid  /= total_energy;
        *treble /= total_energy;
    }
}
```

❖ **void FFT_SETUP():**

- Hàm này thiết lập hệ thống FFT và thực hiện xử lý âm thanh ban đầu.
  - Gọi arm_rfft_fast_init_f32(&S, FFT_SIZE) để khởi tạo cấu trúc FFT cho bộ xử lý CMSIS.
  - Sau đó gọi Calculate_FFT() để tính FFT từ dữ liệu ADC.
  - Cuối cùng, gọi Analyze_Frequency_Bands(&bass, &mid, &treble) để phân tích năng lượng theo dải tần.
- Đây là hàm tổng khởi động quá trình xử lý tín hiệu âm thanh – từ thu mẫu đến phân tích tần số. Nên được gọi mỗi khi cần cập nhật dữ liệu FFT và kết quả phân tích.

```
void FFT_SETUP(void)
{
    arm_rfft_fast_init_f32(&S, FFT_SIZE);

    Calculate_FFT();
    Analyze_Frequency_Bands(&bass, &mid, &treble);
}
```

❖ **void Light_Bar():**

- Hiệu ứng thanh sáng đơn giản nhất, trong đó số lượng LED được bật sẽ tăng giảm theo mức cường độ âm thanh. Điều này tạo ra cảm giác "âm lượng càng lớn, ánh sáng càng mạnh".
  - Dựa trên fft_output_average để xác định mức cường độ âm thanh.
  - Tính số LED cần sáng (led_on_count) theo tỷ lệ với SOUND_MAX.
  - Bật từng LED với mức màu xanh dương tăng dần, còn lại tắt.

```c
void Light_Bar(void)
{
    float intensity = fft_output_average - SOUND_BASE;
    if (intensity > SOUND_MAX) intensity = SOUND_MAX;
    if (intensity < 0.0f) intensity = 0.0f;

    uint8_t led_on_count = (uint8_t)((intensity / SOUND_MAX) * MAX_LED);
    if (led_on_count > MAX_LED) led_on_count = MAX_LED;

    int Color_Steps = 255 / MAX_LED;
    for (int i = 0 ; i < led_on_count ; i++)
    {
        Set_LED(i, Color_Steps * i, 0, 255);
    }

    for (int i = led_on_count ; i < MAX_LED ; i++)
    {
        Set_LED(i, 0, 0, 0);
    }
    Set_Brightness(13);
    WS2812_Send();
    HAL_Delay(20);
}
```

❖ **void Alternating_Flash(uint32_t MIN_DELAY, uint32_t MAX_DELAY):**

- Đèn LED sẽ nhấp nháy luân phiên giữa các vị trí chẵn – lẻ với màu sắc khác nhau, tốc độ thay đổi theo mức âm lượng. Càng lớn thì càng nhấp nháy nhanh, tạo hiệu ứng động mạnh và kích thích thị giác.
    - Chuẩn hóa fft_output_average để tính tốc độ delay (delayrate).
    - Lần lượt bật LED chẵn với màu hồng, rồi LED lẻ với màu xanh dương.
    - Mỗi lần đều có delay phụ thuộc âm lượng.

```
void Alternating_Flash(uint8_t MIN_DELAY, uint8_t MAX_DELAY)
{
    if (fft_output_average < SOUND_BASE) fft_output_average = SOUND_BASE;
    if (fft_output_average > SOUND_MAX) fft_output_average = SOUND_MAX;

    float norm = (fft_output_average - SOUND_BASE) / (SOUND_MAX - SOUND_BASE);
    delayrate = (uint32_t)(MAX_DELAY - (norm * (MAX_DELAY - MIN_DELAY)));

    for (int i = 0 ; i < MAX_LED; i+=2)
    {
        Set_LED(i, 255, 0, 150);
    }
    Set_Brightness(13);
    WS2812_Send();
    HAL_Delay(delayrate);

    Reset_All_LED();

    for (int i = 1 ; i < MAX_LED ; i+=2)
    {
        Set_LED(i, 0, 150, 255);
    }
    Set_Brightness(13);
    WS2812_Send();
    HAL_Delay(delayrate);

    Reset_All_LED();
}
```

❖ **void EchoSplit():**

- Hiệu ứng mô phỏng sao băng di chuyển, nơi một "đầu sao" sáng trắng quét qua dải LED, kèm theo phần "đuôi" mờ dần. Tốc độ rơi của sao băng được điều chỉnh theo âm lượng âm thanh, tạo cảm giác âm thanh đang "vẽ" lên dải LED.
    - Đầu LED trắng sáng di chuyển từ đầu đến cuối dải.
    - Các LED phía sau được làm mờ theo hệ số giảm (60 đơn vị).
    - Tốc độ chạy sao băng điều chỉnh theo âm lượng.

```
void EchoSplit()
{
    Reset_All_LED();

    if (fft_output_average < SOUND_BASE) fft_output_average = SOUND_BASE;
    if (fft_output_average > SOUND_MAX) fft_output_average = SOUND_MAX;

    const int blockSize = 5;
    for (int i = 0; i <= (MAX_LED / 2) - blockSize; i++) {
        Reset_All_LED();
        readtouch();
        if (Signal.End_Flag) {
            Reset_All_LED();
            Signal.End_Flag = 0;
            return;
        }

        FFT_SETUP();
        if (fft_output_average < SOUND_BASE) fft_output_average = SOUND_BASE;
        if (fft_output_average > SOUND_MAX) fft_output_average = SOUND_MAX;

        for (int j = 0; j < blockSize; j++) {
            int idx = i + j;
            if (fft_output_average < 2000.0f)
                Set_LED(idx, 8, 247, 254);
            else if (fft_output_average >2000.0f && fft_output_average < 2800.0f)
                Set_LED(idx, 9, 251, 211);
            else
                Set_LED(idx, 254, 83, 187);
        }
        for (int j = 0; j < blockSize; j++) {
            int idx = MAX_LED - 1 - i - j;
            if (fft_output_average < 2000.0f)
                Set_LED(idx, 8, 247, 254);
            else if (fft_output_average >2000.0f && fft_output_average < 2800.0f)
                Set_LED(idx, 9, 251, 211);
            else
                Set_LED(idx, 254, 83, 187);
        }
        Set_Brightness(45);
        WS2812_Send();
    }
    for (int i = 0; i <= (MAX_LED / 2) - blockSize; i++) {
        readtouch();
        if (Signal.End_Flag) {
            Reset_All_LED();
            Signal.End_Flag = 0;
            return;
        }

        FFT_SETUP();
        if (fft_output_average < SOUND_BASE) fft_output_average = SOUND_BASE;
        if (fft_output_average > SOUND_MAX) fft_output_average = SOUND_MAX;

        for (int j = 0; j < blockSize; j++) {
            int idx = (MAX_LED / 2 - 1) - i - j;
            if (idx >= 0) {
                if (fft_output_average < 2000.0f)
                    Set_LED(idx, 8, 247, 254);
                else if (fft_output_average >2000.0f && fft_output_average < 2800.0f)
                    Set_LED(idx, 9, 251, 211);
                else
                    Set_LED(idx, 254, 83, 187);
            }
        }

        for (int j = 0; j < blockSize; j++) {
            int idx = (MAX_LED / 2) + i + j;
            if (idx < MAX_LED) {
                if (fft_output_average < 2000.0f)
                    Set_LED(idx, 8, 247, 254);
                else if (fft_output_average >2000.0f && fft_output_average < 2800.0f)
                    Set_LED(idx, 9, 251, 211);
                else
                    Set_LED(idx, 254, 83, 187);
            }
        }
        Set_Brightness(45);
        WS2812_Send();
    }
}
```

❖ void Sparkle():

- Hiệu ứng nhấp nháy ngẫu nhiên các LED với màu sắc và cường độ thay đổi, giống như ánh sáng nhấp nháy của kim cương hoặc pháo hoa. Độ sáng tổng thể sẽ thay đổi theo âm lượng, tạo sự sống động và bất ngờ.
    - Dựa trên âm lượng để tính độ sáng tối đa (brightness).
    - Mỗi lần gọi, màu LED được chọn ngẫu nhiên.

```
void Sparkle(void)
{
    Reset_All_LED();

    float normalized = (fft_output_average - SOUND_BASE) / (SOUND_MAX - SOUND_BASE);
    if (normalized > 1.0f) normalized = 1.0f;
    if (normalized < 0.0f) normalized = 0.0f;

    brightness = (uint8_t)(normalized * 45.0f);

    for (int i = 0 ; i < MAX_LED ; i++)
    {
        uint8_t R = rand() % 256;
        uint8_t G = rand() % 256;
        uint8_t B = rand() % 256;

        Set_LED(i, R, G, B);
    }
    Set_Brightness(brightness);
    WS2812_Send();
    HAL_Delay(50);
}
```

3.3.4. Thư viện ILI9341.c

a. Nguyên lý hoạt động

Giao tiếp với màn hình ILI9341 thông qua mô phỏng bộ nhớ cho phép vi điều khiển điều khiển màn hình như thể nó là một vùng nhớ ngoài. Cụ thể, các lệnh và dữ liệu được truyền bằng cách ghi vào các địa chỉ bộ nhớ cụ thể, nhờ đó giao tiếp diễn ra nhanh chóng và đơn giản như khi truy cập RAM thông thường.

Khi khởi động, thư viện điều khiển sẽ gửi một chuỗi lệnh cấu hình đến màn hình ILI9341. Những lệnh này có nhiệm vụ thiết lập các thông số như độ phân giải, chế độ màu sắc, hướng hiển thị, cấu hình gamma và nhiều yếu tố khác nhằm đưa màn hình vào trạng thái sẵn sàng để hiển thị nội dung.

Việc hiển thị đồ họa trên ILI9341 dựa trên cơ chế điều khiển từng điểm ảnh (pixel). Thư viện cung cấp nhiều hàm hỗ trợ vẽ các hình cơ bản như điểm, đường thẳng, hình chữ nhật, hình tròn, tam giác và cả hình ảnh dạng bitmap. Mỗi hình được tạo nên từ nhiều pixel được tô màu phù hợp theo yêu cầu.

Bên cạnh đồ họa, thư viện cũng hỗ trợ hiển thị văn bản thông qua bảng font bitmap. Mỗi ký tự được tạo thành từ các điểm ảnh mô phỏng hình dạng chữ cái, và được vẽ ra tại vị trí con trỏ hiện tại. Thư viện còn có thể tự động xuống dòng khi cần thiết, giúp việc hiển thị văn bản linh hoạt và trực quan.

Ngoài ra, thư viện cũng hỗ trợ thay đổi hướng hiển thị của màn hình như xoay ngang, dọc, hoặc lật gương. Việc thay đổi này được thực hiện bằng cách gửi lệnh điều chỉnh ma trận biến đổi hình ảnh bên trong màn hình. Đáng chú ý, thao tác này không làm thay đổi bản thân dữ liệu, mà chỉ ảnh hưởng đến cách trình bày trên màn hình.

Để tối ưu hiệu năng, thư viện còn sử dụng các kỹ thuật như giới hạn vùng vẽ (drawing window) để giảm lượng dữ liệu cần truyền, hoặc bật/tắt các hiệu ứng như đảo màu, hiển thị, và xé khung (tearing effect) nhằm cải thiện chất lượng hình ảnh và tốc độ cập nhật màn hình. Nhờ những cơ chế này, việc hiển thị trên ILI9341 trở nên mượt mà và hiệu quả hơn.

b. Các hàm chức năng

❖ **void tetrisDrawBackButton(uint16_t x, uint16_t y, uint16_t color)**

- Vẽ nút "Back" tại vị trí (x, y) với màu color
- lcdDrawCircle(x, y, DOWN_BUTTON_RADIUS - 3, color);

  → Vẽ một vòng tròn nhỏ hơn (radius - 3)
- lcdDrawCircle(x, y, DOWN_BUTTON_RADIUS, color);

  → Vẽ một vòng tròn lớn hơn (radius).
- lcdDrawTriangle(x+5, y-10, x+5, y+10, x-5, y, color);

  → Vẽ một hình tam giác bên trong nút (hướng qua trái).

```
void tetrisDrawBackButton(uint16_t x, uint16_t y, uint16_t color)
{

    lcdDrawCircle(x,y,BACK_BUTTON_RADIUS-3,color);
    lcdDrawCircle(x,y,BACK_BUTTON_RADIUS,color);
    lcdDrawTriangle(x+5,y-10,x+5,y+10,x-5,y,color);

}
```

❖ **unsigned long testDrawImage()**

- Vẽ màn hình khởi động gồm:
  - Nền màu đen
  - Logo (Ảnh bmSTLogo)
  - nút tròn bo góc có chữ “ ON EFFECT”
- lcdFillRGB(COLOR_BACK) : Xóa toàn bộ màn hình , đổ màu đen
- Vẽ Logo trung tâm:
  - Nếu LCD nằm ngang:

    → lcdDrawImage((lcdGetWidth() - bmSTLogo.xSize) / 2, 0, &bmSTLogo);
  - Nếu LCD nằm dọc:

    → lcdDrawImage(0, (lcdGetHeight() - bmSTLogo.ySize) / 2, &bmSTLogo);
- Vẽ nút bấm giao diện:

→ LCD_DrawRoundIcon(x, y, width, height, radius, "label", borderColor, fillColor, textColor);

```
unsigned long testDrawImage()
{
    unsigned long start;

    lcdFillRGB(COLOR_BLACK);
    start = HAL_GetTick();
    if (lcdGetOrientation() == LCD_ORIENTATION_LANDSCAPE || lcdGetOrientation() == LCD_ORIENTATION_LANDSCAPE_MIRROR)
    {
        lcdDrawImage((lcdGetWidth() - bmSTLogo.xSize) / 2, 0, &bmSTLogo);
    }
    else
    {
        lcdDrawImage(0, (lcdGetHeight() - bmSTLogo.ySize) / 2, &bmSTLogo);
    }

    //in cac icon
     LCD_DrawRoundIcon(90, 50, 120, 40, 10, "F_F_T", COLOR_BLUE, COLOR_BLACK , COLOR_WHITE);
     LCD_DrawRoundIcon(90, 120, 120, 40, 10, "N-O-R-M-A-L", COLOR_BLUE, COLOR_BLACK, COLOR_WHITE);
//  lcdDrawImage(20, 20, &bmSTIcon1 );

    return HAL_GetTick() - start;

}
```

❖ **void giao_dien_batdau()**

- Thiết lập cấu hình văn bản rồi gọi hàm testDrawImage() để hiển thị logo và các biểu tượng.
- testDrawImage();

  → Gọi hàm testDrawImage() để vẽ nội dung hình ảnh và các nút lên LCD.

```
void giao_dien_batdau(){
        lcdSetTextFont(&Font16);
        lcdSetCursor(0, lcdGetHeight() - lcdGetTextFont()->Height - 1);
        lcdSetTextColor(COLOR_WHITE, COLOR_BLACK);
        testDrawImage();
}
```

❖ **void LCD_DrawRoundIcon(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t radius,const char *text, uint16_t frameColor, uint16_t textColor, uint16_t bgColor)**

- Vẽ một nút hình chữ nhật bo góc (rounded icon) và in chữ vào giữa nút đó.
- lcdFillRoundRect(x, y, width, height, radius, bgColor);
  - Vẽ hình chữ nhật bo góc có nền màu bgColor.
  - Đây là phần nền của nút.
- lcdDrawRoundRect(x, y, width, height, radius, frameColor);

- Vẽ viền ngoài với bán kính bo góc bằng radius và màu viền frameColor.

- uint16_t textX = x + (width / 2) - (strlen(text) * 4); // mỗi ký tự ~8 pixel
- uint16_t textY = y + (height / 2) - 8; // chiều cao ~16 pixel
  - Giả định font chữ cao 16 pixel, rộng khoảng 8 pixel/char.
  - textX: canh giữa ngang.
  - textY: canh giữa dọc.
- LCD_Print(textX, textY, (char *)text, textColor, textColor);
  - In chữ với màu chữ textColor.
  - Nền chữ không đổi vì màu nền và màu chữ giống nhau (hoặc bạn có thể thay textColor, bgColor nếu muốn có nền khác).

```
void LCD_DrawRoundIcon(uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint16_t radius,
                       const char *text, uint16_t frameColor, uint16_t textColor, uint16_t bgColor)
{
    // 1. Vẽ nền bo góc tròn
    lcdFillRoundRect(x, y, width, height, radius, bgColor);
    // Vẽ khung bo góc tròn
        lcdDrawRoundRect(x, y, width, height, radius, frameColor);

        // Tính toán vị trí chữ để căn giữa
        uint16_t textX = x + (width / 2) - (strlen(text) * 4);  // Mỗi ký tự rộng ~8px
        uint16_t textY = y + (height / 2) - 8;                  // Mỗi ký tự cao ~16px

        // In chữ vào giữa khung mà không có nền
        LCD_Print(textX, textY, (char *)text, textColor, textColor);
}
```

3.3.5. Thư viện Calibrate.c

a. Nguyên lý hoạt động

Thư viện này có mục đích chính là xử lý quá trình hiệu chuẩn và ánh xạ tọa độ từ cảm ứng (touch screen) sang tọa độ thực tế trên màn hình hiển thị. Điều này rất quan trọng vì các tọa độ cảm ứng thu được thường không chính xác tuyệt đối, do ảnh hưởng của sai số phần cứng, môi trường hoặc vị trí lắp đặt.

Để thực hiện việc chuyển đổi tọa độ, thư viện sử dụng phép biến đổi affine tuyến tính hai chiều. Phép biến đổi này sử dụng một ma trận để ánh xạ từ tọa độ cảm ứng thu được (Xs, Ys) sang tọa độ hiển thị chính xác trên màn hình (Xd, Yd). Nhờ đó, người dùng tương tác với giao diện một cách chính xác hơn.

Quá trình hiệu chuẩn được thực hiện thông qua 3 điểm mẫu. Người dùng hoặc hệ thống sẽ cung cấp 3 điểm đã biết trên màn hình (gọi là display points), cùng với 3 điểm tương ứng mà cảm ứng ghi nhận được (touch points). Hàm *setCalibrationMatrix()* trong thư viện sẽ sử dụng các điểm này để tính toán ra 6 hệ số biến đổi (An, Bn, Cn, Dn, En, Fn) và một giá trị chia (Divider) cần thiết cho ma trận affine.

Sau khi hiệu chuẩn xong, mỗi lần người dùng chạm vào màn hình, hàm *getDisplayPoint()* sẽ sử dụng tọa độ cảm ứng hiện tại kết hợp với ma trận đã hiệu chuẩn để tính ra tọa độ thực trên LCD, từ đó đảm bảo độ chính xác trong việc phản hồi cảm ứng.

Thư viện cũng lưu ý đến giới hạn độ phân giải của cảm ứng. Nó được thiết kế tối ưu cho độ phân giải ≤ 10 bit, tức tối đa 1024 giá trị. Trong trường hợp cảm ứng có độ phân giải cao hơn (chẳng hạn 12 bit = 4096 giá trị), người dùng cần sử dụng biến kiểu 64-bit hoặc áp dụng tỷ lệ thu nhỏ để tránh hiện tượng tràn số nguyên trong quá trình tính toán.

b. Các hàm chức năng

❖ **Int setCalibrationMatrix(POINT_T * displayPtr,POINT_T * screenPtr, MATRIX * matrixPtr)**

- Tính toán các hệ số của ma trận hiệu chuẩn (calibration matrix) từ 3 điểm cảm ứng và 3 điểm hiển thị thực tế tương ứng.
    - isplayPtr: mảng chứa 3 điểm hiển thị thực tế trên LCD (tọa độ mong muốn).
    - screenPtr: mảng chứa 3 điểm cảm ứng tương ứng do người dùng chạm (tọa độ cảm ứng).
    - matrixPtr: con trỏ đến cấu trúc MATRIX – nơi lưu kết quả các hệ số biến đổi.
- Kết quả:
    - Tính ra các hệ số: An, Bn, Cn, Dn, En, Fn, và Divider.
    - Những hệ số này được dùng để chuyển đổi tọa độ cảm ứng sang tọa độ hiển thị trong hàm thứ hai.

- Trả về:
    - OK nếu tính toán thành công.
    - NOT_OK nếu không tính được ma trận (chia cho 0, tức là 3 điểm không tạo thành một tam giác hợp lệ).

```
int setCalibrationMatrix( POINT_T * displayPtr,
                          POINT_T * screenPtr,
                          MATRIX * matrixPtr)
{
    int  retValue = OK ;
    matrixPtr->Divider = ((screenPtr[0].x - screenPtr[2].x) * (screenPtr[1].y - screenPtr[2].y)) -
                         ((screenPtr[1].x - screenPtr[2].x) * (screenPtr[0].y - screenPtr[2].y)) ;
    if( matrixPtr->Divider == 0 )
    {
        retValue = NOT_OK ;
    }
    else
    {
        matrixPtr->An = ((displayPtr[0].x - displayPtr[2].x) * (screenPtr[1].y - screenPtr[2].y)) -
                        ((displayPtr[1].x - displayPtr[2].x) * (screenPtr[0].y - screenPtr[2].y)) ;
        matrixPtr->Bn = ((screenPtr[0].x - screenPtr[2].x) * (displayPtr[1].x - displayPtr[2].x)) -
                        ((displayPtr[0].x - displayPtr[2].x) * (screenPtr[1].x - screenPtr[2].x)) ;
        matrixPtr->Cn = (screenPtr[2].x * displayPtr[1].x - screenPtr[1].x * displayPtr[2].x) * screenPtr[0].y +
                        (screenPtr[0].x * displayPtr[2].x - screenPtr[2].x * displayPtr[0].x) * screenPtr[1].y +
                        (screenPtr[1].x * displayPtr[0].x - screenPtr[0].x * displayPtr[1].x) * screenPtr[2].y ;
        matrixPtr->Dn = ((displayPtr[0].y - displayPtr[2].y) * (screenPtr[1].y - screenPtr[2].y)) -
                        ((displayPtr[1].y - displayPtr[2].y) * (screenPtr[0].y - screenPtr[2].y)) ;
        matrixPtr->En = ((screenPtr[0].x - screenPtr[2].x) * (displayPtr[1].y - displayPtr[2].y)) -
                        ((displayPtr[0].y - displayPtr[2].y) * (screenPtr[1].x - screenPtr[2].x)) ;

        matrixPtr->Fn = (screenPtr[2].x * displayPtr[1].y - screenPtr[1].x * displayPtr[2].y) * screenPtr[0].y +
                        (screenPtr[0].x * displayPtr[2].y - screenPtr[2].x * displayPtr[0].y) * screenPtr[1].y +
                        (screenPtr[1].x * displayPtr[0].y - screenPtr[0].x * displayPtr[1].y) * screenPtr[2].y ;
    }

    return( retValue ) ;

} /* end of setCalibrationMatrix() */
```

❖ **Int getDisplayPoint(POINT_T * displayPtr,POINT_T * screenPtr,MATRIX * matrixPtr)**

- Dựa vào tọa độ cảm ứng thực tế và ma trận hiệu chuẩn đã tính được, tính ra tọa độ hiển thị thực tế cần vẽ lên màn hình.
    - displayPtr: con trỏ tới nơi lưu kết quả là tọa độ hiển thị tương ứng (tọa độ sau khi đã hiệu chuẩn).
    - screenPtr: tọa độ cảm ứng mà người dùng vừa chạm vào.
    - matrixPtr: con trỏ tới ma trận hiệu chuẩn đã được tính bằng setCalibrationMatrix().
- Trả về:
    - OK nếu tính toán thành công.

- NOT_OK nếu Divider == 0, tức là ma trận hiệu chuẩn không hợp lệ.

```
int getDisplayPoint( POINT_T * displayPtr,
                     POINT_T * screenPtr,
                     MATRIX * matrixPtr )
{
    int retValue = OK ;
    if( matrixPtr->Divider != 0 )
    {
        /* Operation order is important since we are doing integer */
        /*  math. Make sure you add all terms together before      */
        /*  dividing, so that the remainder is not rounded off     */
        /*  prematurely.                                           */

        displayPtr->x = ( (matrixPtr->An * screenPtr->x) +
                          (matrixPtr->Bn * screenPtr->y) +
                           matrixPtr->Cn
                        ) / matrixPtr->Divider ;

        displayPtr->y = ( (matrixPtr->Dn * screenPtr->x) +
                          (matrixPtr->En * screenPtr->y) +
                           matrixPtr->Fn
                        ) / matrixPtr->Divider ;
    }
    else
    {
        retValue = NOT_OK;
    }

    return (retValue);
} /* end of getDisplayPoint() */
```

```
int getDisplayPoint( POINT_T * displayPtr,
                     POINT_T * screenPtr,
                     MATRIX * matrixPtr )
{
    int retValue = OK ;


    if( matrixPtr->Divider != 0 )
    {

        /* Operation order is important since we are doing integer */
        /*  math. Make sure you add all terms together before      */
        /*  dividing, so that the remainder is not rounded off     */
        /*  prematurely.                                           */

        displayPtr->x = ( (matrixPtr->An * screenPtr->x) +
                          (matrixPtr->Bn * screenPtr->y) +
                           matrixPtr->Cn
                        ) / matrixPtr->Divider ;

        displayPtr->y = ( (matrixPtr->Dn * screenPtr->x) +
                          (matrixPtr->En * screenPtr->y) +
                           matrixPtr->Fn
                        ) / matrixPtr->Divider ;
    }
    else
    {
        retValue = NOT_OK;
    }

    return (retValue);

} /* end of getDisplayPoint() */
```

3.3.6. Thư viện LCD_Touch.c

a. Nguyên lý hoạt động

Thư viện có vai trò chính là nhận biết và xử lý tín hiệu từ màn hình cảm ứng điện trở, nhằm thu được tọa độ điểm chạm chính xác trên màn hình LCD sử dụng driver ILI9341. Việc này đảm bảo người dùng có thể tương tác chính xác với các thành phần hiển thị trên màn hình.

Khi người dùng chạm tay vào màn hình, một chân tín hiệu IRQ (interrupt request) trên chip điều khiển cảm ứng sẽ chuyển về mức thấp (LOW), báo hiệu có một thao tác chạm đang diễn ra. Thư viện liên tục kiểm tra trạng thái của chân IRQ này để phát hiện sự kiện chạm một cách kịp thời và chính xác.

Sau khi phát hiện có chạm, thư viện sẽ sử dụng giao tiếp SPI để gửi các lệnh đọc tọa độ X và Y tới IC điều khiển cảm ứng. Dữ liệu trả về từ IC là các tọa độ thô (raw) thể hiện vị trí điểm chạm trên tấm cảm ứng điện trở.

Do tín hiệu cảm ứng dễ bị nhiễu bởi các yếu tố như môi trường, nhiễu điện từ hoặc tiếp xúc không ổn định, thư viện áp dụng kỹ thuật lọc nhiễu để tăng độ chính xác. Cụ thể, nó sẽ thực hiện nhiều lần đọc tọa độ (thường là 10 lần), sau đó sắp xếp các giá trị đo được và lấy trung bình của hai giá trị ở giữa. Cách làm này giúp loại bỏ các giá trị lệch quá xa (outliers) và giảm thiểu ảnh hưởng của nhiễu.

Cuối cùng, tọa độ thô thu được sẽ được chuyển đổi sang tọa độ hiển thị thực tế thông qua một ma trận hiệu chuẩn. Ma trận này được thiết lập trước bằng thực nghiệm để bù trừ sai số phần cứng và căn chỉnh độ chính xác. Nhờ đó, thư viện có thể trả về tọa độ chính xác (x, y) trong vùng hiển thị của màn hình LCD, ví dụ như trong khung 320x240 pixel. Việc hiệu chuẩn này rất quan trọng để đảm bảo người dùng chạm đúng vị trí mong muốn trên giao diện hiển thị.

b. Các hàm chức năng

❖ **bool TouchIsTouched()**

- Kiểm tra xem màn hình cảm ứng có đang bị chạm hay không.
- Đọc trạng thái của chân IRQ (TOUCH_IRQ_PIN).
  - Nếu chân này ở mức LOW, tức là có chạm → trả về true.
  - Ngược lại → trả về false.

```
bool TouchIsTouched(void)
{
    GPIO_PinState pin_state = HAL_GPIO_ReadPin(TOUCH_IRQ_PORT, TOUCH_IRQ_PIN);
    return pin_state == GPIO_PIN_RESET;
}
```

❖ **static uint8_t SpiTransfer(uint8_t byte)**

- Gửi 1 byte dữ liệu qua SPI và nhận lại 1 byte từ IC cảm ứng.
- Sử dụng HAL_SPI_TransmitReceive() để giao tiếp SPI 2 chiều với IC cảm ứng.

```
static uint8_t SpiTransfer(uint8_t byte)
{
    uint8_t result;

    (void)HAL_SPI_TransmitReceive(&hspi2, &byte, &result, 1U, 1000U);

    return (result);
}
```

❖ **static bool GetPointRaw(uint16_t* x, uint16_t* y)**

- Chức năng: Đọc nhiều lần giá trị tọa độ thô X, Y từ màn hình cảm ứng. Trả về giá trị đã được lọc nhiễu (trung bình).
- Cách hoạt động:
  - Kiểm tra nếu đang chạm (TouchIsTouched()).
  - Nếu có:
    - Kích hoạt CS (CS_ON) để bắt đầu SPI.
    - Lặp lại quá trình đọc giá trị X/Y 10 lần.
    - Lưu kết quả vào databuffer[][].
  - Sau khi đọc xong:
    - Tắt CS (CS_OFF).
    - Kiểm tra xem đủ 10 mẫu chưa. Nếu chưa → false.
    - Sắp xếp từng mảng X và Y.

- Lấy trung bình 2 giá trị giữa (thứ 4 và 5) để giảm nhiễu.

- Gán vào *x, *y và trả về true.

```
static bool GetPointRaw(uint16_t* x, uint16_t* y)
{
    uint8_t i;
    bool sorted;
    uint16_t swap_value;
    uint16_t x_raw;
    uint16_t y_raw;
    uint16_t databuffer[2][MW_HAL_TOUCH_READ_POINTS_COUNT];
    uint8_t touch_count;

    if (!TouchIsTouched())
    {
        return false;
    }

    // get set of readings
    CS_ON;
    touch_count = 0U;
    do
    {
        SpiTransfer(COMMAND_READ_X);
        x_raw = (uint16_t)SpiTransfer(0U) << 8;
        x_raw |= (uint16_t)SpiTransfer(0U);
        x_raw >>= 3;

        SpiTransfer(COMMAND_READ_Y);
        y_raw = (uint16_t)SpiTransfer(0U) << 8;
        y_raw |= (uint16_t)SpiTransfer(0U);
        y_raw >>= 3;

        databuffer[0][touch_count] = x_raw;
        databuffer[1][touch_count] = y_raw;
        touch_count++;
    }
    while (TouchIsTouched() == true && touch_count < MW_HAL_TOUCH_READ_POINTS_COUNT);
    CS_OFF;

    // check that the touch was held down during all the readings
    if (touch_count != MW_HAL_TOUCH_READ_POINTS_COUNT)
    {
        return (false);
    }
```

```
    // sort the x readings
    do
    {
        sorted = true;
        for (i = 0U; i < touch_count - 1U; i++)
        {
            if(databuffer[0][i] > databuffer[0][i + 1U])
            {
                swap_value = databuffer[0][i + 1U];
                databuffer[0][i + 1U] = databuffer[0][i];
                databuffer[0][i] = swap_value;
                sorted = false;
            }
        }
    }
    while (!sorted);

    // sort the y readings
    do
    {
        sorted = true;
        for (i = 0U; i < touch_count - 1U; i++)
        {
            if (databuffer[1][i] > databuffer[1][i + 1U])
            {
                swap_value = databuffer[1][i + 1U];
                databuffer[1][i + 1U] = databuffer[1][i];
                databuffer[1][i] = swap_value;
                sorted = false;
            }
        }
    }
    while (!sorted);

    // take averaged middle 2 readings
    *x = (databuffer[0][4] + databuffer[0][5]) / 2U;
    *y = (databuffer[1][4] + databuffer[1][5]) / 2U;

    return (true);
}
```

❖ **bool TouchGetCalibratedPoint(int16_t* x, int16_t* y)**

- Trả về tọa độ chạm đã được hiệu chuẩn (tọa độ pixel trên LCD).
- Cách hoạt động:
  - Gọi GetPointRaw() để lấy tọa độ thô.
  - Chuyển từ raw_point sang display_point bằng hàm getDisplayPoint() (dùng ma trận hiệu chuẩn matrix).
  - Giới hạn phạm vi tọa độ đầu ra không vượt quá kích thước màn hình.
  - Gán vào *x, *y → trả về true.

```
bool TouchGetCalibratedPoint(int16_t* x, int16_t* y)
{
    POINT_T raw_point;
    POINT_T display_point;
    uint16_t raw_x;
    uint16_t raw_y;
    // get raw reading
    if (GetPointRaw(&raw_x, &raw_y) == false)
    {
        return false;
    }
    raw_point.x = (INT_32)raw_x;
    raw_point.y = (INT_32)raw_y;
    // apply calibration matrix
    (void)getDisplayPoint(&display_point, &raw_point, &matrix);
    // range check results
    if (display_point.x > 319)
    {
        display_point.x = 319;
    }
    if (display_point.y > 239)
    {
        display_point.y = 239;
    }

    if (display_point.x < 0)
    {
        display_point.x = 0;
    }
    if (display_point.y < 0)
    {
        display_point.y = 0;
    }
    *x = (int16_t)display_point.x;
    *y =(int16_t)display_point.y;
    return true;
}
```

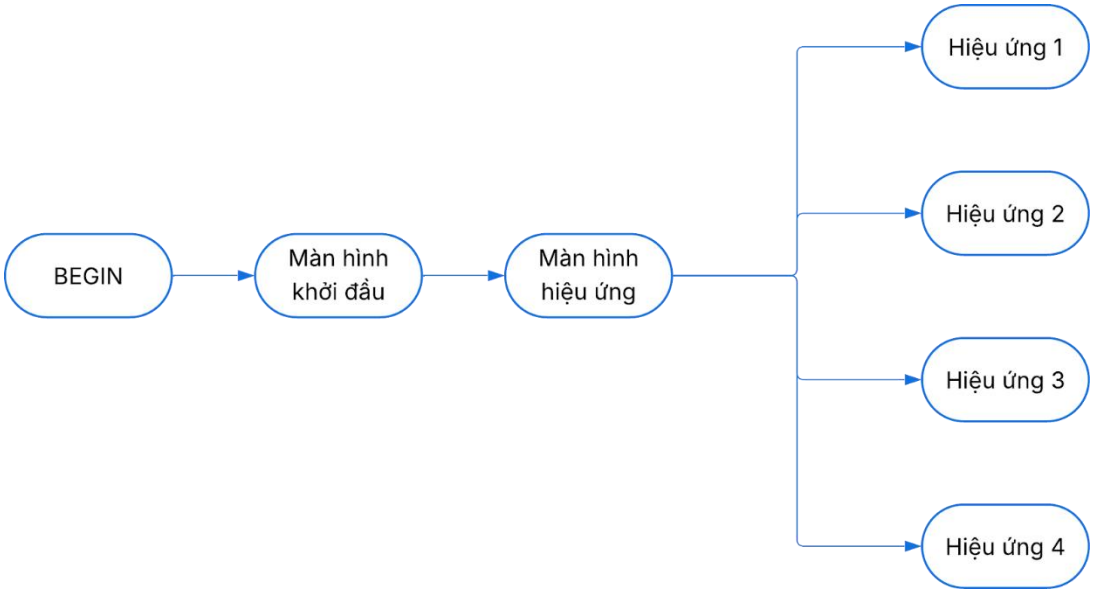
❖ **void TouchCalibrate()**

- Thiết lập ma trận hiệu chuẩn từ dữ liệu mẫu.
- Cách hoạt động:
  - Dựa trên 3 điểm thực nghiệm:
    - display_points[] là vị trí người dùng chạm trên màn hình.
    - raw_points[] là giá trị đo được thô từ cảm ứng
  - Gọi setCalibrationMatrix() để tính ra matrix hiệu chuẩn.

```
void TouchCalibrate(void)
{
//  uint16_t x;
//  uint16_t y;
    POINT_T raw_points[3];
    POINT_T display_points[3] = {{40, 40}, {280, 40}, {280, 200}};

    raw_points[0].x = 2970;
    raw_points[0].y = 670;



    /* second point */

    raw_points[1].x = 865;
    raw_points[1].y = 711;




    raw_points[2].x = 789;
    raw_points[2].y = 3239;



    (void)setCalibrationMatrix(display_points, raw_points, &matrix);
}
```

Chương 4: Thiết kế giao diện

4.1. Sơ đồ liên kết các màn hình



Hình 12: Sơ đồ liên kết các màn hình

4.2. Danh sách các màn hình

STT	Màn hình	Loại màn hình	Chức năng
1	Màn hình khởi đầu	Chọn/Nhấn nút	
2	Màn hình hiệu ứng	Chọn/Nhấn nút	Màn hình để lựa chọn hiệu ứng nào để thực thi và chuyển đổi giữa các hiệu ứng

Bảng 11: Danh sách các màn hình

4.3. Mô tả các màn hình

4.3.1. Màn hình khởi đầu

a. Giao diện

b. Mô tả các đối tượng trên màn hình

STT	Tên	Kiểu	Chức năng
1	O_N F_F_T	BUTTON	

Bảng 12: Các đối tượng trên màn hình khởi đầu

4.3.2. Màn hình hiệu ứng

a. Giao diện

b. Mô tả các đối tượng trên màn hình

STT	Tên	Kiểu	Chức năng
1	Light_Bar	BUTTON	Bật hiệu ứng
2	Sparkle	BUTTON	Bật hiệu ứng
3	Flash	BUTTON	Bật hiệu ứng
4	EchoSplit	BUTTON	Bật hiệu ứng
5	(<) tam giác hướng sang trái	BUTTON	Chuyển sang màn hình khởi đầu

Bảng 13: Các đối tượng trên màn hình ứng dụng

Chương 5: Tổng kết

5.1. Sản phẩm hoàn thiện

Hình 13: Sản phẩm hoàn thiện

5.2. Kết luận

5.2.1. Nhận xét và kết luận

- ❖ Sản phẩm đã đạt được các yêu cầu đề ra, cụ thể:
    - Vi điều khiển thu nhận tín hiệu âm thanh từ microphone thông qua module MAX9814, sau đó xử lý tín hiệu bằng thuật toán FFT để trích xuất các đặc trưng tần số.
    - Kết quả xử lý âm thanh được sử dụng để điều khiển dãy đèn LED WS2812, thông qua kỹ thuật băm xung PWM, tạo ra hiệu ứng ánh sáng phản hồi theo âm thanh.
    - Hệ thống hỗ trợ nhiều hiệu ứng ánh sáng khác nhau, tất cả đều dựa trên đặc tính âm thanh đầu vào, mang lại trải nghiệm thị giác sống động và tương tác.
    - Màn hình cảm ứng LCD tích hợp hiển thị tên hiệu ứng đang chạy, đồng thời cho phép người dùng chọn lựa hiệu ứng một cách trực quan và chủ động, nâng cao tính tương tác của sản phẩm.
- ❖ Một số điểm hạn chế còn tồn tại:
    - Hiệu ứng ánh sáng chưa phong phú, cần được mở rộng thêm để tăng tính hấp dẫn.
    - Khi sử dụng với số lượng LED lớn, việc đồng bộ giữa âm thanh và ánh sáng có thể bị trễ nhẹ, ảnh hưởng đến độ mượt của hiệu ứng.
    - Cảm ứng trên màn hình LCD đôi khi chưa ổn định, cần được hiệu chỉnh hoặc thay thế phần cứng để nâng cao độ nhạy và độ chính xác.

5.2.2. Hướng phát triển

- ✓ **Mở rộng bộ hiệu ứng ánh sáng:** Thiết kế thêm nhiều hiệu ứng mới, đa dạng về màu sắc, hình dạng và kiểu chuyển động, phù hợp với các thể loại âm nhạc khác nhau (như EDM, ballad, rock…).

- ✓ **Cải thiện khả năng đồng bộ với âm thanh**: Tối ưu thuật toán FFT và quá trình truyền dữ liệu đến LED để giảm độ trễ, đặc biệt khi điều khiển dãy LED dài hoặc tốc độ âm nhạc cao.
- ✓ **Tối ưu hiệu suất phần cứng:** Nâng cấp vi điều khiển hoặc sử dụng DMA, bộ định thời và các kỹ thuật lập trình song song để tăng hiệu suất xử lý, giảm tải CPU và cải thiện tốc độ phản hồi.
- ✓ **Cải thiện giao diện cảm ứng:** Tăng độ nhạy và độ ổn định cho cảm ứng LCD, hoặc thay thế bằng màn hình cảm ứng chất lượng cao hơn để nâng cao trải nghiệm người dùng.
- ✓ **Cá nhân hóa trải nghiệm người dùng:** Cho phép người dùng tùy chỉnh hoặc lập trình hiệu ứng theo sở thích, hoặc thiết lập hiệu ứng theo các dải tần cụ thể (bass, mid, treble).
- ✓ **Kết nối không dây:** Phát triển tính năng điều khiển và chọn hiệu ứng qua Bluetooth hoặc Wi-Fi, thông qua ứng dụng trên điện thoại.
- ✓ **Tích hợp lưu trữ và phát lại:** Lưu lại các đoạn âm thanh và hiệu ứng đã xử lý để có thể phát lại sau hoặc dùng trong trình diễn ánh sáng.

TÀI LIỆU THAM KHẢO

1. Datasheet STM32F407VET6 Black Board
2. How to Interface WS2812 with STM32 using Timer PWM
3. STM32 LED Strip (WS2812b) control 1: Theory and implementation
4. [STM32] Hướng Dẫn Tích Hợp CMSIS-DSP vào STM32CubeIDE: Ví Dụ SIMD Cộng Vector
5. Định lý Nyquist–Shannon
6. Thời gian chuyển đổi ADC trên STM32
7. STM32 FSMC || LCD PART 1 || How to configure
8. STM32 FSMC || LCD PART 2 || Add touch Interface

BẢNG PHÂN CÔNG

Thành viên	Nhiệm vụ	Tỉ lệ đóng góp
Dương Thanh Hiếu	Xử lý thu âm thanh từ MAX9814 Thiết kế thư viện xử lý LED, thư viện xử lý FFT, thư viện hiệu ứng	100%
	Viết báo cáo	60%
Trần Triệu Dân	Thiết kế thư viện xử lý LCD, thư viện xử lý cảm ứng màn hình	100%
	Viết báo cáo	40%