ENSEA

Beyond Engineering

APLM

(**A**utomated **P**iano **L**earning **M**odule)

Project semester 7 2024-2025

Contents

ENSEA
Beyond Engineering

I. Component Search and Order

ENSEA
Beyond Engineering

I. Component Search and Order

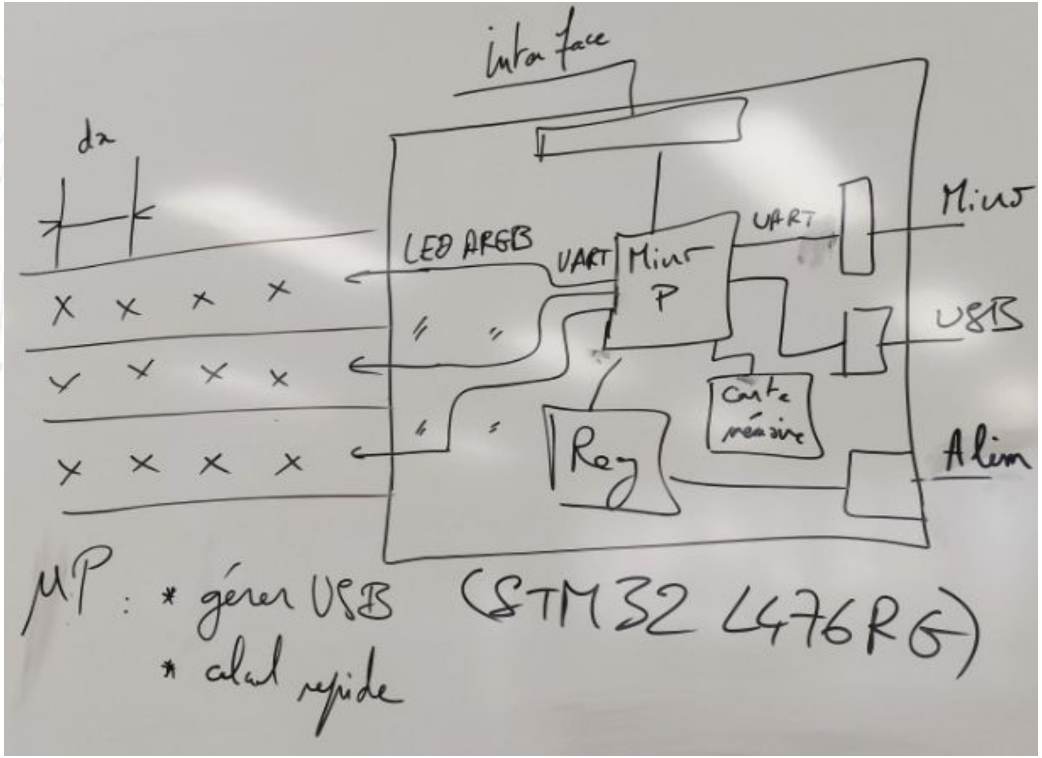


Figure 1: Main components and their place

I. Component Search and Order

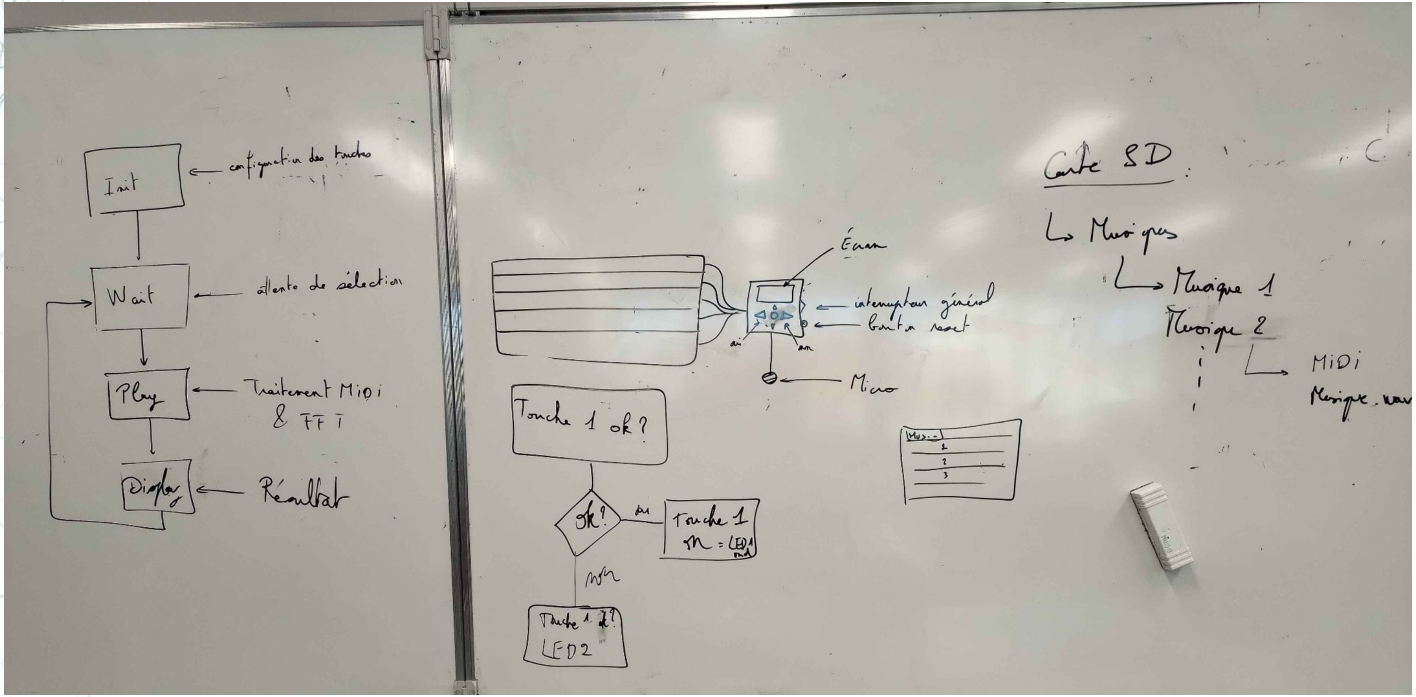


Figure 2: Model

I. Component Search and Order

Case modeling idea

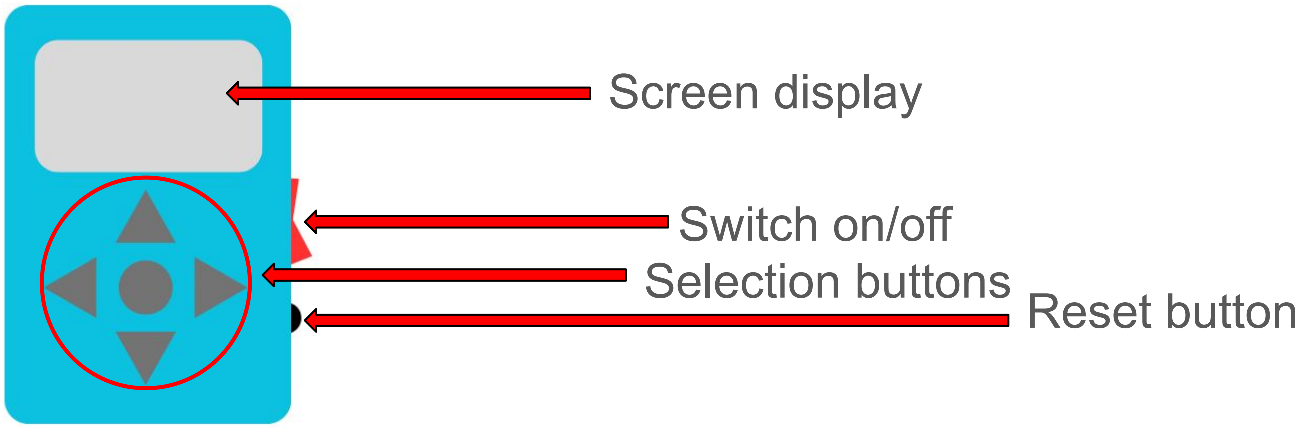


Figure 3: Led ARGB

I. Component Search and Order

MAPA Initialisation (Assign the piano keys to an LED)

```c
volatile uint32_t compteur = 0; /* cette variable peut être modifiée de manière
                                   imprévisible par des événements extérieurs*/

UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */
int __io_putchar(int ch) {
    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
    return ch;
}
```

```c
  while (1)
  {
    /* USER CODE END WHILE */
      printf("Compteur : %lu\r\n", compteur); // Envoie la valeur du compteur via UART
              HAL_Delay(500); // Envoie la valeur toutes les X ms
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == GPIO_PIN_13) // Vérifie que l'interruption vient de PC13
    {
        compteur++;   // Incrémente le compteur
    }
}
```

Figure 4: Test on the only button of the STM32L476RG

Choosing the right LED strip

- That can be divisible
- RGB
- That can be controlled

ENSEA
Beyond Engineering

II. Schematic and PCB Design

ENSEA
Beyond Engineering

II. Schematic and PCB Design

IN PROGRESS!

Figure 5: KiCad schematic

ENSEA
Beyond Engineering

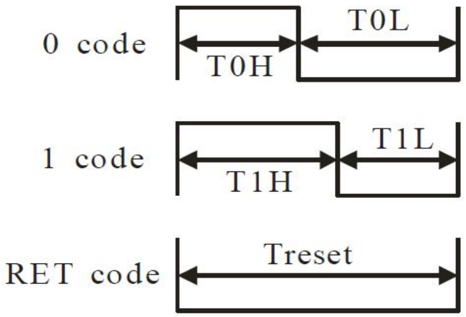
III. Processing a MIDI file

ENSEA
Beyond Engineering

III. Processing a MIDI file

Data transfer time(TH+TL=1.25μs±600ns)

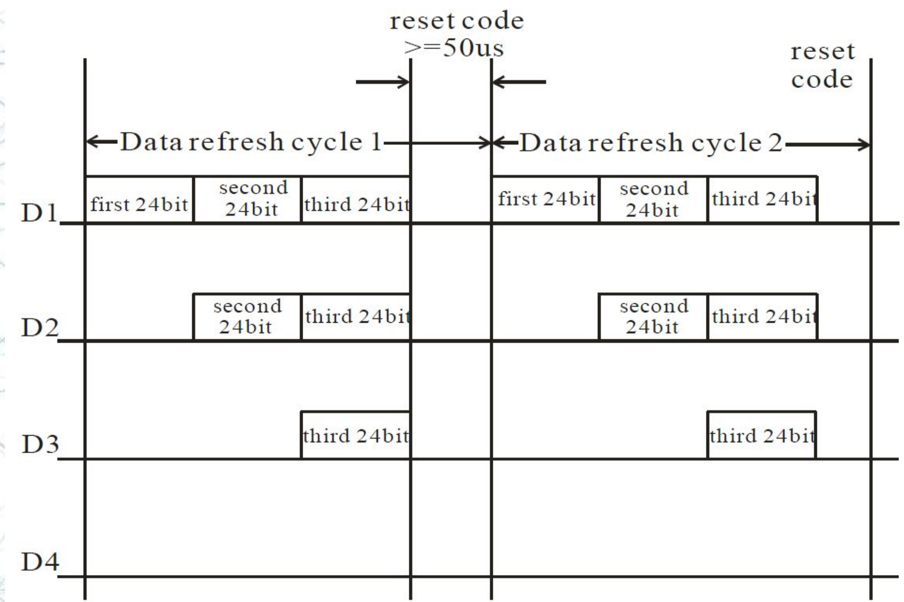
T0H	0 code ,high voltage time	0.4us	±150ns
T1H	1 code ,high voltage time	0.8us	±150ns
T0L	0 code , low voltage time	0.85us	±150ns
T1L	1 code ,low voltage time	0.45us	±150ns
RES	low voltage time	Above 50μs	




```c
#define T0H   (80 * 0.4)  // 0.4us * 80 MHz
#define T1H   (80 * 0.8)  // 0.8us * 80 MHz
#define T0L   (80 * 0.85) // 0.85us * 80 MHz
#define T1L   (80 * 0.45) // 0.45us * 80 MHz
#define LED_COUNT 30
void WS2812_Reset(void) {
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);
    HAL_Delay(1);  // Wait more than 50µs
```

III. Processing a MIDI file

Data transmission method:




```c
void WS2812_SendColor(uint8_t red, uint8_t green, uint8_t blue) {
    WS2812_SendByte(green);
    WS2812_Reset();
    WS2812_SendByte(red);
    WS2812_Reset();
    WS2812_SendByte(blue);
    WS2812_Reset();
}

void WS2812_SendByte(uint8_t byte) {
    for (int i = 0; i < 8; i++) {
        if (byte & (1 << (7 - i))) {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
            delay_cycles(T1H);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);
            delay_cycles(T1L);
        } else {
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_SET);
            delay_cycles(T0H);
            HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_RESET);
            delay_cycles(T0L);
        }
    }
}
```

Composition of 24bit data:

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

Note: Follow the order of GRB to sent data and the high bit sent at first.

Figure: Functionment of the led strip

III. Processing a MIDI file

```c
uint8_t led_colors[LED_COUNT][3] = {0};

int midi_note_to_led(uint8_t note) {
    if (note >= 21 && note < 21 + LED_COUNT) {
        return note - 21;
    }
    return -1;
}

void set_led_color(int index, uint8_t red, uint8_t green, uint8_t blue) {
    if (index < 0 || index >= LED_COUNT) {
        return;
    }
    led_colors[index][0] = green;
    led_colors[index][1] = red;
    led_colors[index][2] = blue;
    WS2812_Update();
}

void WS2812_Update(void) {
    for (int i = 0; i < LED_COUNT; i++) {
        WS2812_SendColor(led_colors[i][1], led_colors[i][0], led_colors[i][2]);
    }
    WS2812_Reset();
}
```

Figure: Control of the led color

ENSEA
Beyond Engineering

III. Processing a MIDI file

```
// MIDI Header Chunk
0x4D, 0x54, 0x68, 0x64, // "MThd" - Header chunk identifier
0x00, 0x00, 0x00, 0x06, // Header length (6 bytes for standard MIDI headers)
0x00, 0x01,             // Format type (Type 1: multiple tracks)
0x00, 0x01,             // Number of tracks (1 in this case)
0x00, 0x60,             // Division (96 ticks per quarter note)
```

- **"MThd" (0x4D 0x54 0x68 0x64)**: This is the ASCII identifier for the MIDI header chunk.
- **Header Length** (0x00, 0x00, 0x00, 0x06): Specifies the length of the header (always 6 bytes).
- **Format Type** (0x00, 0x01): Defines the file as a Type 1 MIDI file, where multiple tracks are allowed (Type 0 is single-track).
- **Number of Tracks** (0x00, 0x01): Indicates that there is one track in this file.
- **Division** (0x00, 0x60): Defines timing in ticks per quarter note.

Figure: MIDI Header Chunk

ENSEA
Beyond Engineering

III. Processing a MIDI file

```
// MIDI Track Chunk
0x4D, 0x54, 0x72, 0x6B, // "MTrk" - Track chunk identifier
0x00, 0x00, 0x00, 0x2F, // Track length (example length here)
```

- **"MTrk"** (0x4D 0x54 0x72 0x6B): Identifies the start of a track chunk.
- **Track Length** (0x00, 0x00, 0x00, 0x2F): Specifies the byte length of the track data (in this case, 47 bytes). This should be adjusted based on actual track content.

Figure: MIDI Track Chunk

ENSEA
Beyond Engineering

III. Processing a MIDI file

- `// Set Tempo Meta-Event`
- `0x00, 0xFF, 0x51, 0x03, 0x07, 0xA1, 0x20`

- **0x00**: Delta time, indicating no delay before this event (plays immediately).
- **0xFF 0x51**: Indicates a "Set Tempo" meta-event.
- **0x03**: Length of the tempo data (3 bytes).
- **0x07 0xA1 0x20**: Sets the tempo in microseconds per quarter note. This value (500,000 in hexadecimal, or `0x07A120`) sets the tempo to 120 beats per minute (BPM), as: Tempo (µs per quarter note)=60,000,000/BPM. So, 500,000 µs = 120 BPM

Figure: MIDI Tempo

ENSEA
Beyond Engineering

III. Processing a MIDI file

```
0xB0, 0x40, 0x00, // Control change for Channel 1
0xC0, 0x00,       // Program change for Channel 1
0xB0, 0x07, 0x7F, // Another control change
0x90, 0x3E, 0x6E, // Note on (channel 1, note 62, velocity 110)
0x90, 0x3E, 0x00, // Note off (channel 1, note 62, velocity 0)
```

- **MIDI Status Bytes**:
  - **0xB0**: Control Change (Channel 1)
  - **0xC0**: Program Change (Channel 1)
  - **0x90**: Note On (Channel 1)
  - **0x80 or 0x90 with velocity 0**: Note Off (Channel 1)
- **Parameters**: Each status byte is followed by parameters, such as note number, velocity, or control values.

Figure: MIDI Events

ENSEA
Beyond Engineering

III. Processing a MIDI file

- `0x00, 0xFF, 0x2F, 0x00`

- **0x00**: Delta time (no delay before the end of track).
- **0xFF 0x2F**: End of track meta-event.
- **0x00**: Length (no additional data).

Figure 12: End of Track

III. Processing a MIDI file

```c
void process_midi_file_data(void) {
    size_t length = sizeof(midi_data) / sizeof(midi_data[0]);
    uint32_t tempo = 500000; // Default tempo (120 BPM, in microseconds per quarter note)
    uint32_t tick_duration = tempo / 96; // Duration per tick based on division in header

    for (size_t i = 0; i < length; ) {
        // Read the delta time
        uint32_t delta_time = 0;
        while (midi_data[i] & 0x80) {
            delta_time = (delta_time << 7) | (midi_data[i++] & 0x7F);
        }
        delta_time = (delta_time << 7) | midi_data[i++];

        // Apply delay based on delta time
        HAL_Delay(delta_time * tick_duration / 1000); // Convert microseconds to milliseconds

        uint8_t status = midi_data[i++];

        if (status == 0xFF) {
            uint8_t meta_type = midi_data[i++];
            uint8_t meta_length = midi_data[i++];

            if (meta_type == 0x51) { // Set Tempo
                tempo = (midi_data[i] << 16) | (midi_data[i + 1] << 8) | midi_data[i + 2];
                tick_duration = tempo / 96; // Update tick duration based on new tempo
            }
            i += meta_length;
        } else if ((status & 0xF0) == 0x90 && midi_data[i + 1] > 0) { // Note on
            uint8_t note = midi_data[i++];
            uint8_t velocity = midi_data[i++];
```

ENSEA
Beyond Engineering

III. Processing a MIDI file

```c
            int led_index = midi_note_to_led(note);
            set_led_color(led_index, 255, 0, 0); // Turn LED on (e.g., red)

        } else if ((status & 0xF0) == 0x80 || midi_data[i + 1] == 0) { // Note off
            uint8_t note = midi_data[i++];
            uint8_t velocity = midi_data[i++];

            int led_index = midi_note_to_led(note);
            set_led_color(led_index, 0, 0, 0); // Turn LED off
        } else {
            i += 2; // Skip other unhandled events
        }
    }
}
```

ENSEA
Beyond Engineering

IV. Digital audio signal processing

IV. Digital audio signal processing

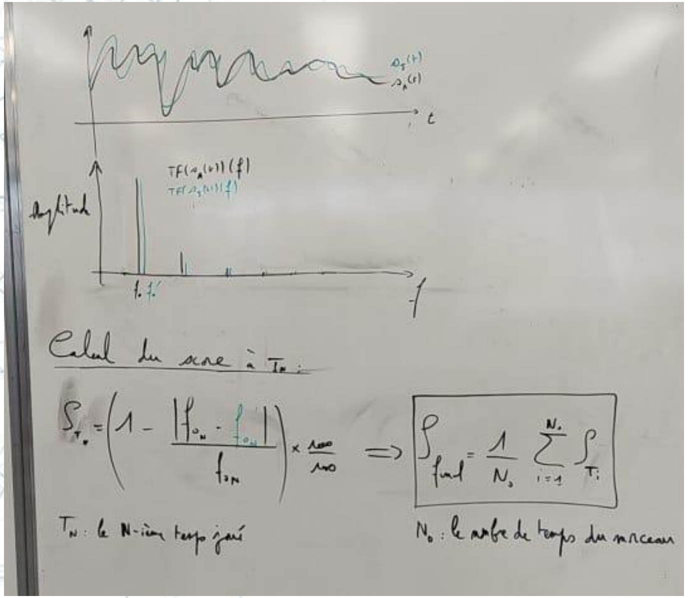


Figure 13: Method for evaluating the user score

CMSIS
COMPLIANT
ARM® Cortex™ Microcontroller
Software Interface Standard

IV. Digital audio signal processing

A reminder of previous conclusions:

Problems encountered:

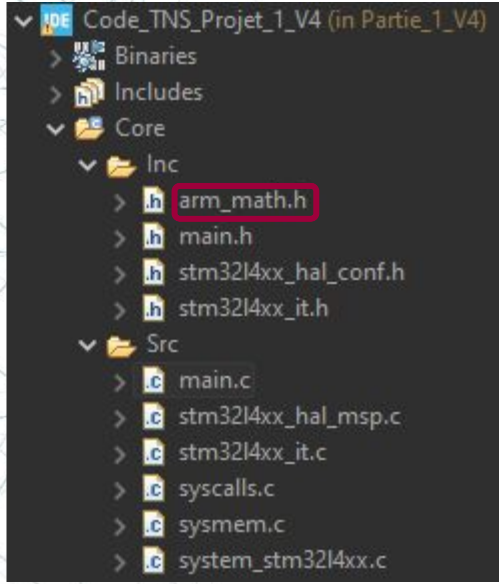
- importing the library
- inclusion problems

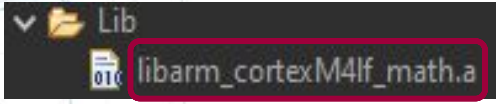
Possible solutions:

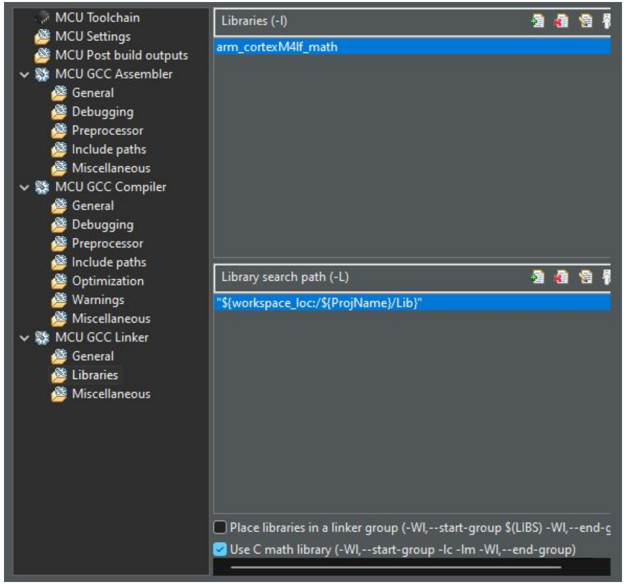
- follow an online step-by-step method for importing

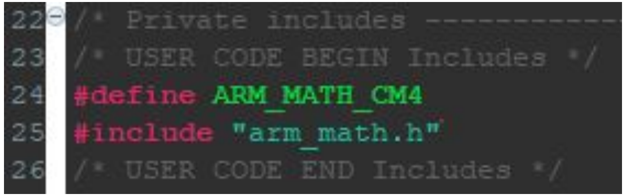
ENSEA
Beyond Engineering

IV. Digital audio signal processing









IV. Digital audio signal processing

```c
// Audio processing function
void Process_Audio(void) {
  // Perform FFT
  arm_rfft_fast_f32(&S, audioSource, fftOutput, 0);

  // Compute the magnitude of the FFT (complex values)
  arm_cmplx_mag_f32(fftOutput, fftMagnitude, AUDIO_BUFFER_SIZE / 2);

  // Find the maximum value and its index
  arm_max_f32(fftMagnitude, AUDIO_BUFFER_SIZE / 2, &maxValue, &maxIndex);

  // Compute the corresponding frequency
  freqMax = ((float32_t)maxIndex / (AUDIO_BUFFER_SIZE / 2)) * (SAMPLE_RATE / 2);
}
```

```c
// Function to send frequency via UART
void Send_Frequency(float32_t frequency) {
  char buffer[50];
  int len = sprintf(buffer, "Max Frequency: %.2f Hz\r\n", frequency); // Format the frequency
  HAL_UART_Transmit(&huart2, (uint8_t*)buffer, len, HAL_MAX_DELAY); // Send it over UART
}
```

Definition of functions for calculating the FFT and sending results back to the terminal

```c
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
#define AUDIO_BUFFER_SIZE 2048  // Increase the buffer size for better resolution
#define SAMPLE_RATE 32000       // Set your sample rate
/* USER CODE END PD */
```

```c
// Global variables
float32_t audioSource[AUDIO_BUFFER_SIZE];  // Audio buffer
float32_t fftOutput[AUDIO_BUFFER_SIZE * 2]; // FFT output buffer (complex)
float32_t fftMagnitude[AUDIO_BUFFER_SIZE]; // Magnitude of FFT
float32_t maxValue;                         // Max FFT value
uint32_t maxIndex;                          // Index of max FFT value
float32_t freqMax;                          // Frequency of max value
```

Defining constants and global variables

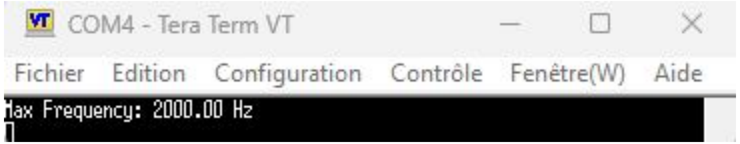
```c
  // Initialize the FFT structure
  arm_rfft_fast_init_f32(&S, AUDIO_BUFFER_SIZE);

  // Fill audio buffer with a 440 Hz sine wave
  for (uint32_t i = 0; i < AUDIO_BUFFER_SIZE; i++) {
    audioSource[i] = sinf(2 * M_PI * 2000.0f * i / SAMPLE_RATE);  // Simulating a 400 Hz signal
  }

  // Process the audio to compute FFT and find max frequency
  Process_Audio();
  Send_Frequency(freqMax);  // Send the calculated frequency over UART
```

Initialise the FFT, simulate a 2000 Hz sinusoidal signal and call up the functions

ENSEA
Beyond Engineering

IV. Digital audio signal processing

Results:



Future projects:

- Use this code with a microphone on an ADC port.
- Modify this code to work for an audio file from an SD card.
- Design a code that takes a score in comparison with a played song.

ENSEA
Beyond Engineering

Beyond Engineering

Thank you for your time and attention!