Microcontroladores e Interação com Sensores e Atuadores

Equalizador automático para headphones

22 de junho 2023

Grupo 3:
João Cunha, 107573; Mafalda Garrelhas, 107625; João Anacleto, 108052; João Rodrigues, 103947; Ricardo Cotrim, 107704; Luís Martins, 107660;

Resumo:

Neste projeto propõe-se a criação de um dispositivo capaz de criar perfis de equalização para headphones. Para tal, foi usado o STM32 NUCLEO F446RE para fazer o processamento de dados, ligado a um display touchscreen para interface humano-computador. Na captação dos dados, recorre-se a um microfone ligado à ADC integrada do STM, montado num suporte onde assentam os headphones. Para reproduzir os sons desejados no processo de calibração foi utilizada a DAC integrada do STM para criação de ondas sinusoidais de frequência crescente.
Essencialmente, este projeto tem como fim criar um perfil de equalizador automático de headphones de uma forma relativamente simples e acessível.

Índice

1 Introdução teórica

A qualidade sonora que uns headphones são capazes de reproduzir é um aspeto crucial para a experiência auditiva do utilizador. No entanto, cada pessoa tem preferências sonoras únicas, o que significa que as características tonais ideais podem variar de pessoa para pessoa.

Um equalizador, permite aos utilizadores ajustar o equilíbrio tonal do áudio reproduzido, otimizando o perfil sonoro dos headphones consoante os seus gostos. Um equalizador tradicional permite ajustar manualmente a intensidade das diferentes frequências sonoras, aumentando-as ou diminuindo-as conforme desejado. No entanto, nem todos os utilizadores têm conhecimento técnico para ajustar manualmente o perfil sonoro dos seus headphones, para isso introduziu-se o conceito de "equalizador automático" para headphones. O equalizador automático é uma ferramenta que ajusta automaticamente o perfil sonoro dos headphones a uma curva de preferência geral chamada de Harman Target.

Em suma, o equalizador automático é uma tecnologia que visa otimizar a experiência sonora dos utilizadores, adaptando-se às suas preferências. Um equalizador automático ajusta o perfil sonoro dos headphones de modo a proporcionar um som mais equilibrado e agradável de uma forma simples e ráp-ida, tornando a experiência auditiva mais imersiva.

2 Objetivos do trabalho

- Obter o gráfico de intensidade em função da frequência emitida pelos headphones.
- Criar um perfil de equalizador baseado nas medições efetuadas para aproximar a resposta dos headphones àquela da curva Harman Target.

3 Idealização do funcionamento do projeto

Para se obter o gráfico pretendido, tem-se que registar as várias intensidades sonoras em função da frequência ao longo de todo o espetro sonoro. Assim sendo, decidiu-se que faria sentido gerar sequencialmente as frequências a analisar e simultaneamente captar a intensidade produzida pelos headphones para as mesmas. Verificou-se que no repositório de GitHub do autoEQ todas medições realizadas tinham por base as mesmas frequências. Assim sendo, basta medir a intensidade sonora para cada uma destas frequências.

Após a aquisição de dados, estes seriam processados, o gráfico da intensidade sonora em função da frequência gerado e o perfil do equalizador criado.

Em realação à calibração do dispositivo, considerou-se que se fosse medida a curva de uns headphones com uma medida já conhecida, poderíamos depois subtrair à curva conhecida essa medida. Assim sendo, obter-se-ia a cuva característica do microfone. Desta forma, faria sentido ter

dois programas de Python ligeiramente diferentes um do outro, um para calibração do aparelho de medida e outro para a correção de headphones desconhecidos.

4 Escolha de materiais

Para concluir os objetivos propostos, é necessário pensar em vários subsistemas que interagem entre si para obter um produto finalizado.

Em primeiro lugar, como se pretende gerar ondas sonoras, é necessário haver um sistema para a criação das mesmas. Para o fazer, decidiu-se utilizar o STM32 NUCLEO F446RE por ter uma DAC incluída de 12bits. Isto possibilitaria a criação de ondas com menos erros de quantização. Para ser mais fácil trocar entre vários headphones, ligou-se uma saída Jack de 3.5mm ao circuito. Verificou-se que o volume produzido pelo STM era demasiado elevado e com receio de danificar os headphones criou-se um sistema de controlo de volume para os mesmos.

De seguida, verificou-se que seria necessário haver um sistema para a aquisição de dados sonoros. Para isso, um sensor capaz de captar este tipo de ondas seria um microfone. Assim sendo, como já tinha sido decidido incluir um microcontrolador no projeto devido à porção de geração do sinal, concluiu-se que deveria ser usada a ADC integrada do STM, mais uma vez, por ter 12bits. Verificou-se posteriormente que o sensor adquirido não incluia um circuito de amplificação. Assim sendo, foi necessário ser o grupo responsável por este trabalho a criá-lo.

Depois, visto ser necessário interagir com os vários sistemas, um display mostrou-se uma opção interessante. Como já se tinha definido o objetivo de gerar gráficos, um ecrã LCD adequar-se-ia mais devido à sua maior qualidade de imagem. Como estavam disponíveis ecrãs touchscreen, optou-se por estes uma vez que a interação com os mesmos é mais intuitíva.

De seguida, como o projeto já tinha um número considerável de subsistemas, decidiu-se que se deveria fazer uma caixa para o mesmo. Devido à liberdade de forma e à facilidade de fabrico decidiu-se utilizar impressão 3D para tal.

Finalmente, seria necesária alguma forma de exportar os resultados obtidos. Como o programa do autoEQ foi escrito em Python, faria sentido ter um computador para processar os dados incluído no projeto. Com este, também se evitaria utilizar uma PowerBank para ligar o projeto. Para simplificar a comunicação do STM com o coputador, decidiu-se simplesmente enviar os dados pela porta de série do microcontrolador.

5 Materiais

5.1 Sensores:

- 1 microfone "KS0035 Keyestudio":

  Os microfones utilizados neste projeto foram microfones capacitivos de eletreto. Este tipo de microfones baseia-se no funcionamento de um condensador para a captação de ondas sonoras. Isto é, quando uma onda sonora

choca com a membrana deste sensor, são induzidas vibrações na mesma. Tanto esta membrana como a placa oposta à mesma, estão carregadas com cargas, tal como um condensador. Assim sendo, as vibrações anteriormente referidas provocam a sucessiva aproximação e afastamento das placas carregadas. Isto faz com que a capacitância instantânea criada por este condensador seja variável. Esta variação irá provocar o aparecimento de uma corrente no circuito. A partir disto, é possível a conversão de sinais sonoros em sinais elétricos. No âmbito deste projeto usou-se um microfone de eletreto para captar as ondas emitidas pelos headphones, que serão, posteriormente, analisadas.

Figura 1 - Imagem do microfone utilizado no projeto

5.2 Atuadores:

- 1 Headphones:

  O tipo mais comum de headphones são os headphones dinâmicos. Nestes, a variação da corrente que atravessa a bobine de cada um dos lados dos headphones induz um campo magnético. Por sua vez, este induzirá o deslocamento do íman do cone dos headphones. Este deslocamento provocará a criação de variações de pressão à superfície do cone sonoro que consequentemente provoca ondas sonoras. Neste projeto os headphones irão receber um sinal elétrico através do Jack 3.5mm, convertendo-o num sinal sonoro, que depois será captado pelo microfone, para posterior análise.

5.3 Outros:

- 1 Display LCD touchscreen NEXTION NX4832T035:

  Um display LCD touchscreen é tanto um sensor como um atuador. Neste caso usámos um painel touchscreen resistivo. Estes painéis são compostos por várias camadas. A camada mais externa é normalemente de plástico resistente a riscos, já a camada interior é geralmente composta por vidro. Cada uma destas camadas está revestida por um material condutor transparente, ficando as duas camadas condutoras frente a frente. Quando o utilizador faz pressão na tela, irá forçá-las a entrar em contacto, alterando a resistência nesse local. O controlador RTP deteta essa mudança e calcula a sua posição.

  Para entender como funciona a transmissão da luz num LCD é necessário entender como funcionam os polarizadores.

Quando dois polarizadores estão paralelos, ou seja, alinhados um ao outro, a luz atravessa-os sem qualquer dificuldade. À medida que a inclinação de um dos polarizadores se vai alterando a quantidade de luz que os atravessa diminui, até aos dois polarizadores ficarem perpendiculares entre si que será quando a luz deixa de atravessar completamente. É importante reeferir que a luz ao incidir no primeiro polarizador sofre algumas perdas visto que o polarizador só deixa passar luz com uma determinada polarização.

O display LCD touchscreen será usado, no projeto, para navegar os menus da interface gráfica. Isto é, será este que permite iniciar o programa e exibir os dados obtidos pelo microfone servindo como fonte de interação humano-computador.

Figura 2 - Imagem do display utilizado no projeto

- 1 STM32:

  O STM32 NUCLEO F446RE é um microcontrolador com vários conversores analógico para digital integrados de 12 bits. Usou-se um deles para a recolha dos dados obtidos pelos microfones. Além disso, este também contém um conversor digital para analógico de 12bits. Este componente é responsável pela criação das diversas ondas sinusoidais. Além da geração e captação de sinal, o STM32 também é responsável pelo processamento dos dados necessários à realização do projeto e pela interligação dos vários subsistemas.

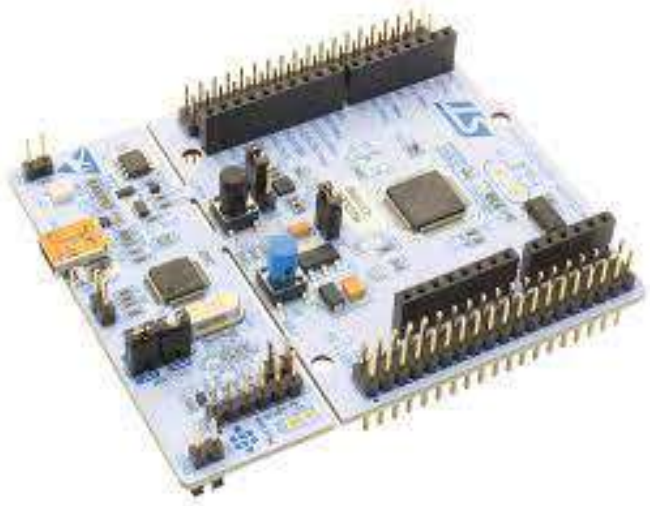


Figura 3 - Microcontrlador utilizado para o projeto

- 1 Placa branca:

  Esta foi usada para a ligação dos circuitos elétricos.

6 Diagramas de montagem

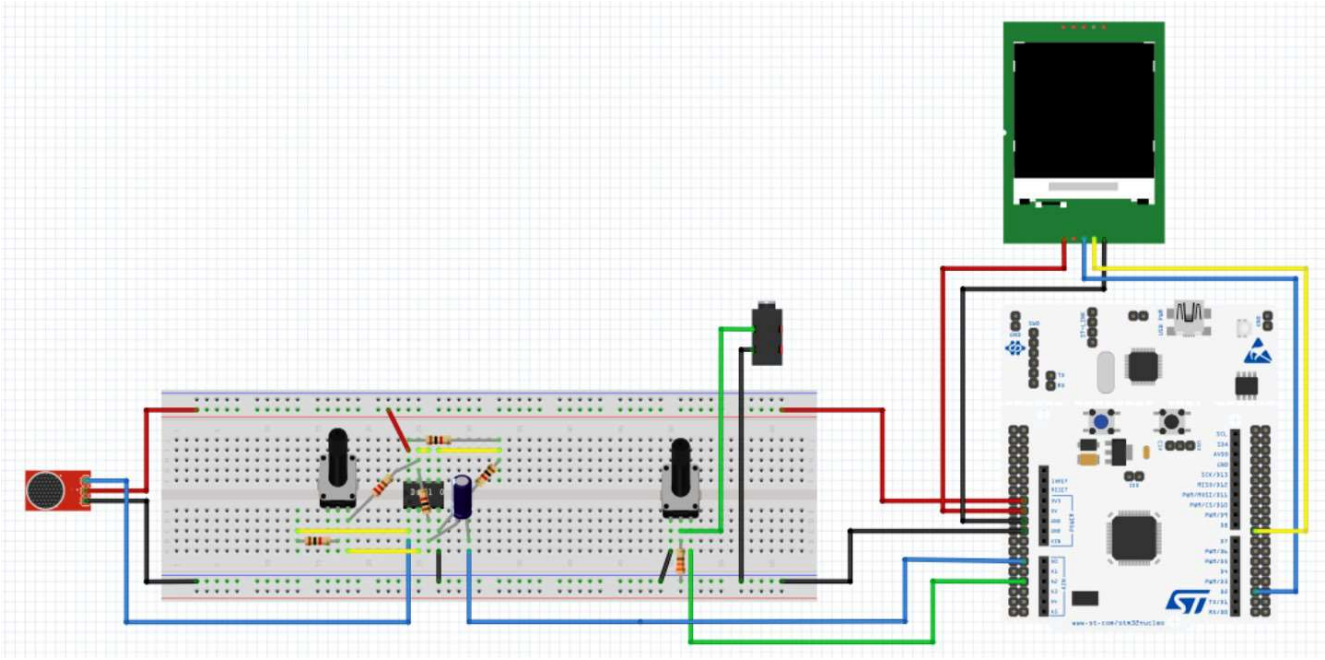


Figura 4 - Diagrama completo do circuito

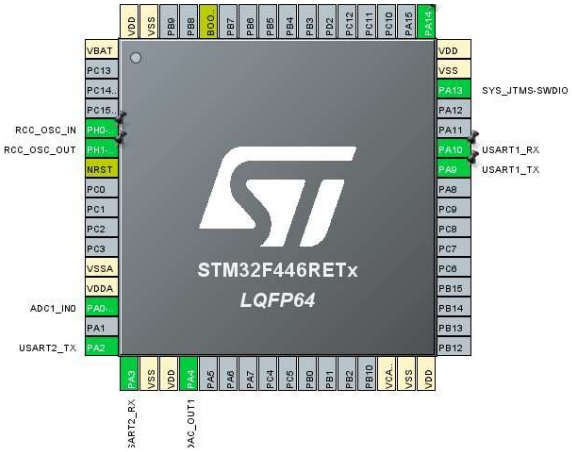


Figura 5 - Diagrama de pinos do STM

Podem-se inferir algumas conclusões deste diagrama:

1. Os pinos PA10 e PA9 foram configurados para ser o RX e o TX, respectivamente da USART1. Estes serão usados para comunicação com o ecrã.
2. O pino PA0 tem a entrada da ADC que foi utilizada para ligar o microphone.
3. O pino PA4 tem a saída da DAC utilizada para gerar as diversas ondas sinusoidais
4. Os pinos PA3 e PA2 têm respectivamente o RX e o TX da USART2. Esta será utilizada para enviar dados para o computador através da porta de série.

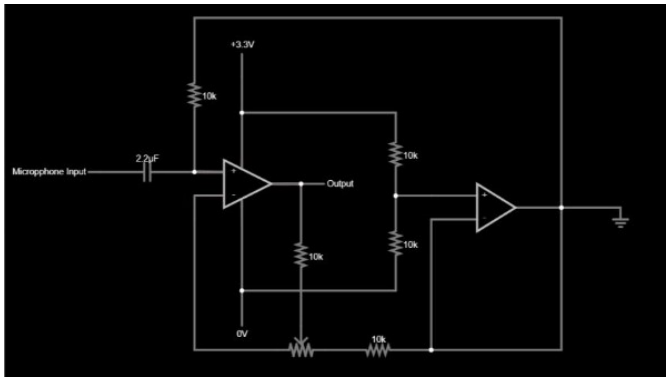


Figura 6 - Diagrama do circuito de amplificação do microfone

Este circuito foi proposto pelo Professor Rui Escadas, uma vez que o microfone não vinha incluído com um circuito de amplificação.

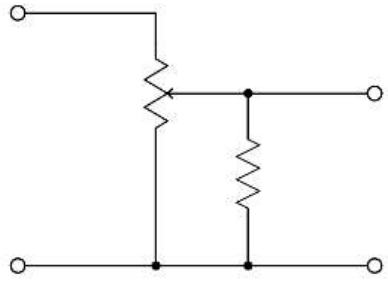


Figura 7 - Diagrama do sistema do controlo de volume dos headphones

Uma vez que se verificou que o volume reproduzido nos headphones quando ligados diretamente ao microcontrolador era muito elevado. Assim sendo, decidiu-se criar um sistema simples com um potenciómetro para controlo de volume para evitar danificá-los. Seguiu-se o circuito recomendado pelo blog do Tom Jewel. Isto porque com este circuito consegue-se aproximar a resposta de um potenciómetro linear áquela dum potenciómetro logarítmico. Isto é importante porque os ouvidos humanos percebem a intensidade sonora de forma logarítmica.

7 Programa implementado

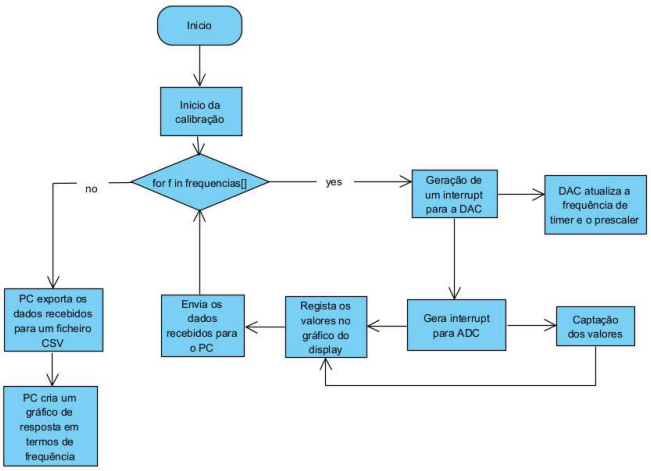


Figura 8 - Fluxograma que descreve o funcionamento lógico do projeto

O programa implementado pode ser descrito nos neguintes passos:

1. Início do programa quando ligado a montagem ao computador

2. Quando recebido o comando enviado pelo display para iniciar a sequência de calibração, é chamada a função responsável pelo funcionamento do projeto.

3. Enquanto não forem realizadas as medições para todas as frequências (Enquanto o array com as frequências em análise não for totalmente percorrido) será repetido:

   a) É gerado um interrupt para a DAC. Neste, os valores da frequência do timer e o prescaler são alterados.

   b) São gerados vários interrupt para a ADC. Estes serão o número de amostras que serão captados para depois serem tratados. Destes valores são escolhidos os 100 maiores e depois disto é feita uma média dos 90 valores mais pequenos deste conjunto de 100. Isto foi feito para tentar atenuar a captação de picos de intensidade.

   c) O valor de intensidade média medida é registada no gráfico do display e a percentagem é atualizada.

   d) A medida captada é enviada para o computador

4. Após o ciclo ter terminado, o computador reconhece que recebeu o número correto de medidas (O loop chega ao fim) e os dados das medições são exportados para um ficheiro CSV.

5. Com recurso ao MatPLotLib é criado o gráfico em estudo para podermos observar os resultados das medidas.

Infelizmente verificou-se que para frequências baixas (<100Hz) havia uma grande quantidade de harmónicos presentes no sinal. Isto foi verificado com um osciloscópio. Para corrigir este problema seria necessário implementar um sistema de filtragem.
Nota: Para uma descrição mais detalhada ver anexos com o código utilizado.

8 Componente Mecânica

Através do programa SolidWorks modelou-se uma peça para garantir que os headphones ficariam posicionados corretamente em relação aos microfones. Depois de terminado o modelo, com as devidas tolerâncias, este teve de ser dividido em várias peças para ser possível se proceder à sua

impressão, uma vez que as dimensões da peça eram superiores às suportadas pela impressora disponível. Depois de terminadas todas as peças, procedeu-se à sua montagem. Inicialmente, as duas peças que formam o corpo da peça foram coladas, depois colocaram-se os suportes para os microfones, aparafusando-os ao corpo da peça. Para além disso foram instalados o STM, placa branca, microfones e também o display, já com as devidas conexões feitas, para que o programa funcionasse corretamente. Por último foram coladas as duas partes que compõem a "tampa", e posteriormente, esta foi aparafusada ao corpo, para garantir a fixação dos componentes no interior da peça.

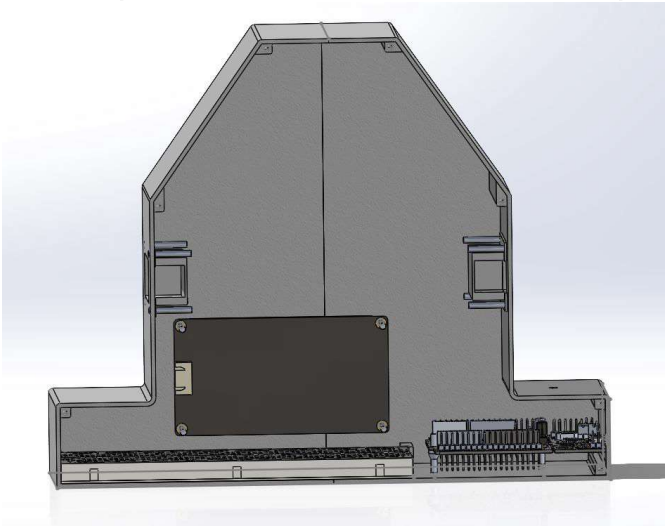


Figura 9 - Modelo 3D do projeto

9 Resultados e Conclusão

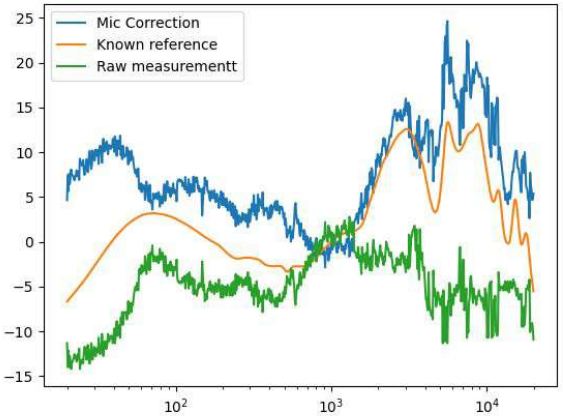


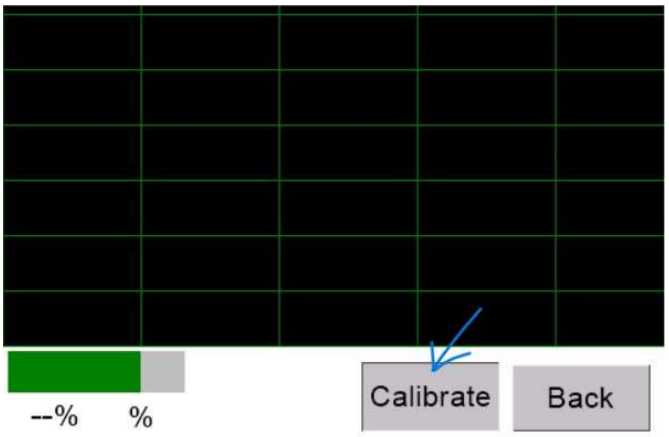
Figura 10 - Gráfico da correção

Figura 11 - Aparelho Finalizado

10 Extra - Manual de utilização

1. Ligar o STM32 a uma porta USB de um computador com o python instalado
2. Correr o programa de Python desejado (calibração ou correção) e escolher a porta de série à qual o STM está ligado:

   Para tal, deve-se escrever no terminal o respectivo número da porta de série da lista de dispositivos nesse momento ligados. Depois, clica-se em enter para este aguardar pelos valores do STM.
3. Ligar os headphones à audio jack e verificar qual dos lados dos headphones é que está a reproduzir som. Depois, alinhar esse lado com o microfone existente na montagem.
4. Navegar os menus do display da NEXTION para iniciar a calibração



5. Aguardar que apareça um gráfico no ecrã do PC. Quando isto acontecer quer dizer que o programa chegou ao fim e se tem um ficheiro CSV com as medidas realizadas.

Nota: Esta secção foi criada para uma futura utilização do projeto

11 Bibliografia

Youtube - ControllersTech
AutoEQ Código - https://github.com/jaakkopasanen/AutoEq
https://www.audio-technica.com/pt-br/support/um-guia-rapido-para-microfones-o-que-um-microfone-faz/
https://www.orientdisplay.com/pt/difference-between-resistive-and-capacitive-touch-panel/

12 Anexos

Código STM32:

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * Copyright (c) 2023 STMicroelectronics.
  * All rights reserved.
  *
  * This software is licensed under terms that can be found in the LICENSE file
  * in the root directory of this software component.
  * If no LICENSE file comes with this software, it is provided AS-IS.
  *
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "math.h"
#include "stdio.h"
#include "stdlib.h"
/* USER CODE END Includes */

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

/* USER CODE END PTD */

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

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
```

```
40 /* USER CODE BEGIN PM */
41
42 /* USER CODE END PM */
43
44 /* Private variables ---------------------------------------------------------
      */
45 ADC_HandleTypeDef hadc1;
46
47 DAC_HandleTypeDef hdac;
48 DMA_HandleTypeDef hdma_dac1;
49
50 TIM_HandleTypeDef htim2;
51
52 UART_HandleTypeDef huart1;
53 UART_HandleTypeDef huart2;
54
55 /* USER CODE BEGIN PV */
56
57 /* USER CODE END PV */
58
59 /* Private function prototypes -----------------------------------------------
      */
60 void SystemClock_Config(void);
61 static void MX_GPIO_Init(void);
62 static void MX_DMA_Init(void);
63 static void MX_DAC_Init(void);
64 static void MX_TIM2_Init(void);
65 static void MX_ADC1_Init(void);
66 static void MX_USART2_UART_Init(void);
67 static void MX_USART1_UART_Init(void);
68 /* USER CODE BEGIN PFP */
69 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef*);
70 void NewMeasurement(void);
71 /* USER CODE END PFP */
72
73 /* Private user code ---------------------------------------------------------
      */
74 /* USER CODE BEGIN 0 */
75
76 uint16_t readValue; //Stores the microphone value
77 float value = 0.001; //Output voltage
78 uint32_t var; //Store the respective digital value
79 uint8_t RxData = 0;
80 uint8_t Cmd_End[3] = {0xFF, 0xFF, 0xFF};
81
82 uint32_t sine_val[100];
83
84 #define PI 3.1415926
85 uint32_t i = 0;
86 uint32_t j = 0;
87 int result; //Will be passed over serial
88 int sampleSize = 10000;
89 uint8_t resultStr[5] = "0000\n";
90 int Reference;
91
92 // Creating the array of all possible Periods
93 const float Periods[][3] =
      {{499,89.0,0},{494.0495,89.0,0.02},{489.19608,89.0,0.03},{484.20136,89.0,0.05},{479.5382,89.0,0.06},{4
```

```
94
95
96  void get_sineval ()
97  {
98          for (int i = 0; i < 100; i++) {
99                  sine_val[i] = (((sin(i*2*PI/100) + 1)*(2048/10)))/2+100;
100         }
101 }
102
103 /* USER CODE END 0 */
104
105 /**
106   * @brief  The application entry point.
107   * @retval int
108   */
109 int main(void)
110 {
111   /* USER CODE BEGIN 1 */
112
113   /* USER CODE END 1 */
114
115   /* MCU Configuration--------------------------------------------------------
      */
116
117   /* Reset of all peripherals, Initializes the Flash interface and the Systick.
       */
118   HAL_Init();
119
120   /* USER CODE BEGIN Init */
121
122   /* USER CODE END Init */
123
124   /* Configure the system clock */
125   SystemClock_Config();
126
127   /* USER CODE BEGIN SysInit */
128
129   /* USER CODE END SysInit */
130
131   /* Initialize all configured peripherals */
132   MX_GPIO_Init();
133   MX_DMA_Init();
134   MX_DAC_Init();
135   MX_TIM2_Init();
136   MX_ADC1_Init();
137   MX_USART2_UART_Init();
138   MX_USART1_UART_Init();
139   /* USER CODE BEGIN 2 */
140
141   // INitalizing the DAC
142   HAL_TIM_Base_Start(&htim2);
143   get_sineval();
144   HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, sine_val, 100, DAC_ALIGN_12B_R);
145
146   // Setting default values
147   htim2.Init.Period = 100-1;
148   htim2.Init.Prescaler = 90-1;
```

```
149   HAL_TIM_Base_Start(&htim2);
150   MX_TIM2_Init();
151
152   /* USER CODE END 2 */
153
154   /* Infinite loop */
155   /* USER CODE BEGIN WHILE */
156   while (1)
157   {
158     /* USER CODE END WHILE */
159
160     /* USER CODE BEGIN 3 */
161         HAL_UART_Receive_IT(&huart1, &RxData,1);
162
163         if (RxData == 0x01) {
164                 NewMeasurement();
165         }
166
167   }
168   /* USER CODE END 3 */
169 }
170
171 /**
172   * @brief System Clock Configuration
173   * @retval None
174   */
175 void SystemClock_Config(void)
176 {
177   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
178   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
179
180   /** Configure the main internal regulator output voltage
181   */
182   __HAL_RCC_PWR_CLK_ENABLE();
183   __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
184
185   /** Initializes the RCC Oscillators according to the specified parameters
186   * in the RCC_OscInitTypeDef structure.
187   */
188   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
189   RCC_OscInitStruct.HSEState = RCC_HSE_ON;
190   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
191   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
192   RCC_OscInitStruct.PLL.PLLM = 4;
193   RCC_OscInitStruct.PLL.PLLN = 180;
194   RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
195   RCC_OscInitStruct.PLL.PLLQ = 2;
196   RCC_OscInitStruct.PLL.PLLR = 2;
197   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
198   {
199     Error_Handler();
200   }
201
202   /** Activate the Over-Drive mode
203   */
204   if (HAL_PWREx_EnableOverDrive() != HAL_OK)
205   {
206     Error_Handler();
```

```
207   }
208
209   /** Initializes the CPU, AHB and APB buses clocks
210   */
211   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
212                                 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
213   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
214   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
215   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
216   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
217
218   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK)
219   {
220     Error_Handler();
221   }
222 }
223
224 /**
225   * @brief ADC1 Initialization Function
226   * @param None
227   * @retval None
228   */
229 static void MX_ADC1_Init(void)
230 {
231
232   /* USER CODE BEGIN ADC1_Init 0 */
233
234   /* USER CODE END ADC1_Init 0 */
235
236   ADC_ChannelConfTypeDef sConfig = {0};
237
238   /* USER CODE BEGIN ADC1_Init 1 */
239
240   /* USER CODE END ADC1_Init 1 */
241
242   /** Configure the global features of the ADC (Clock, Resolution, Data
        Alignment and number of conversion)
243   */
244   hadc1.Instance = ADC1;
245   hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
246   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
247   hadc1.Init.ScanConvMode = DISABLE;
248   hadc1.Init.ContinuousConvMode = ENABLE;
249   hadc1.Init.DiscontinuousConvMode = DISABLE;
250   hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
251   hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
252   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
253   hadc1.Init.NbrOfConversion = 1;
254   hadc1.Init.DMAContinuousRequests = DISABLE;
255   hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
256   if (HAL_ADC_Init(&hadc1) != HAL_OK)
257   {
258     Error_Handler();
259   }
260
261   /** Configure for the selected ADC regular channel its corresponding rank in
        the sequencer and its sample time.
262   */
```

```
263   sConfig.Channel = ADC_CHANNEL_0;
264   sConfig.Rank = 1;
265   sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
266   if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
267   {
268     Error_Handler();
269   }
270   /* USER CODE BEGIN ADC1_Init 2 */
271
272   /* USER CODE END ADC1_Init 2 */
273
274 }
275
276 /**
277   * @brief DAC Initialization Function
278   * @param None
279   * @retval None
280   */
281 static void MX_DAC_Init(void)
282 {
283
284   /* USER CODE BEGIN DAC_Init 0 */
285
286   /* USER CODE END DAC_Init 0 */
287
288   DAC_ChannelConfTypeDef sConfig = {0};
289
290   /* USER CODE BEGIN DAC_Init 1 */
291
292   /* USER CODE END DAC_Init 1 */
293
294   /** DAC Initialization
295   */
296   hdac.Instance = DAC;
297   if (HAL_DAC_Init(&hdac) != HAL_OK)
298   {
299     Error_Handler();
300   }
301
302   /** DAC channel OUT1 config
303   */
304   sConfig.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
305   sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
306   if (HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1) != HAL_OK)
307   {
308     Error_Handler();
309   }
310   /* USER CODE BEGIN DAC_Init 2 */
311
312   /* USER CODE END DAC_Init 2 */
313
314 }
315
316 /**
317   * @brief TIM2 Initialization Function
318   * @param None
319   * @retval None
320   */
```

```
321 static void MX_TIM2_Init(void)
322 {
323
324   /* USER CODE BEGIN TIM2_Init 0 */
325
326   /* USER CODE END TIM2_Init 0 */
327
328   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
329   TIM_MasterConfigTypeDef sMasterConfig = {0};
330
331   /* USER CODE BEGIN TIM2_Init 1 */
332
333   /* USER CODE END TIM2_Init 1 */
334   htim2.Instance = TIM2;
335   //htim2.Init.Prescaler = 90-1;
336   htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
337   //htim2.Init.Period = 100-1;
338   htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
339   htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
340   if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
341   {
342     Error_Handler();
343   }
344   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
345   if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
346   {
347     Error_Handler();
348   }
349   sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
350   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
351   if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
352   {
353     Error_Handler();
354   }
355   /* USER CODE BEGIN TIM2_Init 2 */
356
357   /* USER CODE END TIM2_Init 2 */
358
359 }
360
361 /**
362   * @brief USART1 Initialization Function
363   * @param None
364   * @retval None
365   */
366 static void MX_USART1_UART_Init(void)
367 {
368
369   /* USER CODE BEGIN USART1_Init 0 */
370
371   /* USER CODE END USART1_Init 0 */
372
373   /* USER CODE BEGIN USART1_Init 1 */
374
375   /* USER CODE END USART1_Init 1 */
376   huart1.Instance = USART1;
377   huart1.Init.BaudRate = 9600;
378   huart1.Init.WordLength = UART_WORDLENGTH_8B;
```

```
379   huart1.Init.StopBits = UART_STOPBITS_1;
380   huart1.Init.Parity = UART_PARITY_NONE;
381   huart1.Init.Mode = UART_MODE_TX_RX;
382   huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
383   huart1.Init.OverSampling = UART_OVERSAMPLING_16;
384   if (HAL_UART_Init(&huart1) != HAL_OK)
385   {
386     Error_Handler();
387   }
388   /* USER CODE BEGIN USART1_Init 2 */
389
390   /* USER CODE END USART1_Init 2 */
391
392 }
393
394 /**
395   * @brief USART2 Initialization Function
396   * @param None
397   * @retval None
398   */
399 static void MX_USART2_UART_Init(void)
400 {
401
402   /* USER CODE BEGIN USART2_Init 0 */
403
404   /* USER CODE END USART2_Init 0 */
405
406   /* USER CODE BEGIN USART2_Init 1 */
407
408   /* USER CODE END USART2_Init 1 */
409   huart2.Instance = USART2;
410   huart2.Init.BaudRate = 115200;
411   huart2.Init.WordLength = UART_WORDLENGTH_8B;
412   huart2.Init.StopBits = UART_STOPBITS_1;
413   huart2.Init.Parity = UART_PARITY_NONE;
414   huart2.Init.Mode = UART_MODE_TX_RX;
415   huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
416   huart2.Init.OverSampling = UART_OVERSAMPLING_16;
417   if (HAL_UART_Init(&huart2) != HAL_OK)
418   {
419     Error_Handler();
420   }
421   /* USER CODE BEGIN USART2_Init 2 */
422
423   /* USER CODE END USART2_Init 2 */
424
425 }
426
427 /**
428   * Enable DMA controller clock
429   */
430 static void MX_DMA_Init(void)
431 {
432
433   /* DMA controller clock enable */
434   __HAL_RCC_DMA1_CLK_ENABLE();
435
436   /* DMA interrupt init */
```

```
437   /* DMA1_Stream5_IRQn interrupt configuration */
438   HAL_NVIC_SetPriority(DMA1_Stream5_IRQn, 0, 0);
439   HAL_NVIC_EnableIRQ(DMA1_Stream5_IRQn);
440
441 }
442
443 /**
444   * @brief GPIO Initialization Function
445   * @param None
446   * @retval None
447   */
448 static void MX_GPIO_Init(void)
449 {
450 /* USER CODE BEGIN MX_GPIO_Init_1 */
451 /* USER CODE END MX_GPIO_Init_1 */
452
453   /* GPIO Ports Clock Enable */
454   __HAL_RCC_GPIOH_CLK_ENABLE();
455   __HAL_RCC_GPIOA_CLK_ENABLE();
456
457 /* USER CODE BEGIN MX_GPIO_Init_2 */
458 /* USER CODE END MX_GPIO_Init_2 */
459 }
460
461 /* USER CODE BEGIN 4 */
462 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
463 {
464   readValue = HAL_ADC_GetValue(&hadc1);
465   /*If continuousconversion mode is DISABLED uncomment below*/
466   //HAL_ADC_Start_IT (&hadc1);
467 }
468
469 void NewMeasurement(void) {
470         int max;
471         int index;
472         int temp;
473         int sum;
474         int ADC_vals[sampleSize];
475         ////////////////////////
476         uint8_t *buffer = malloc(30*sizeof (char));
477         int len = sprintf((char *) buffer, "percentagem.val=0");
478         HAL_UART_Transmit(&huart1, buffer, len, 1000);
479         HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
480         free(buffer);
481         uint8_t *bufferx = malloc(30*sizeof (char));
482         int lenx = sprintf((char *) bufferx, "x.val=0");
483         HAL_UART_Transmit(&huart1, bufferx, lenx, 1000);
484         HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
485         free(bufferx);
486         uint8_t *buffery = malloc(30*sizeof (char));
487         int leny = sprintf((char *) buffery, "y.val=0");
488         HAL_UART_Transmit(&huart1, buffery, leny, 1000);
489         HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
490         free(buffery);
491         uint8_t *bufferxref = malloc(30*sizeof (char));
492         int lenxref  = sprintf((char *) bufferxref , "xref.val=0");
493         HAL_UART_Transmit(&huart1, bufferxref , lenxref , 1000);
494         HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
```

```
495         free(bufferxref );
496         uint8_t *bufferyref  = malloc(30*sizeof (char));
497         int lenyref = sprintf((char *) bufferyref, "yref.val=0");
498         HAL_UART_Transmit(&huart1, bufferyref, lenyref, 1000);
499         HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
500         free(bufferyref);
501
502
503
504         for (i =0 ; i < 695; i++) {
505                 htim2.Init.Period = Periods[i][0];
506                 htim2.Init.Prescaler = Periods[i][1];
507                 HAL_TIM_Base_Start(&htim2);
508                 MX_TIM2_Init();
509
510                 // Initalizing the microphone
511                 for (j = 0; j < sampleSize; j++) {
512                 HAL_ADC_Start_IT (&hadc1);
513                 ADC_vals[j] = readValue;
514                 }
515
516                 // Calculating the read Value
517                 // Selection algorithm
518                 // Order biggest 100 terms
519                 for (int iter = 0; iter < 100; iter++) {
520                         j = iter;
521                         max = 0;
522                         while (j < sampleSize) {
523                                 if (ADC_vals[j]>max) {
524                                         max = ADC_vals[j];
525                                         index = j;
526                             }
527                                 j++;
528                         }
529                         //Podia poupar 1 var
530                         temp = ADC_vals[iter];
531                         ADC_vals[iter] = max;
532                         ADC_vals[index] = temp;
533                 }
534
535                 // Getting the average value
536                 sum = 0;
537                 for (j = 9; j < 100; j++) sum+=ADC_vals[j];
538                 result = sum/90;
539
540                 // Converting to uint8
541                 for (j = 0; j < 4; j++) {resultStr[j] = 48 + (((int)result)/ (
                        int) pow(10,3-j)) % 10;}
542
543                 // sending to Display
544                 uint8_t *bufferGB = malloc(30*sizeof (char));
545                 int lenGB = sprintf((char *) bufferGB, "greenBar.val=%d", (int)
                        (Periods[i][2]*100));
546                 HAL_UART_Transmit(&huart1, bufferGB, lenGB, 1000);
547                 HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
548                 free(bufferGB);
549
550                 uint8_t *bufferCOVX = malloc(30*sizeof (char));
```

```
551                     int lenCOVX = sprintf((char *) bufferCOVX, "covx greenBar.val,
                            t0.txt,0,0");
552                     HAL_UART_Transmit(&huart1, bufferCOVX, lenCOVX, 1000);
553                     HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
554                     free(bufferCOVX);
555 
556                     uint8_t *bufferSETY = malloc(30*sizeof (char));
557                     int lenSETY = sprintf((char *) bufferSETY, "y.val=%d", result);
558                     HAL_UART_Transmit(&huart1, bufferSETY, lenSETY, 1000);
559                     HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
560                     free(bufferSETY);
561 
562                     uint8_t *bufferADD2 = malloc(30*sizeof (char));
563                     int lenADD2 = sprintf((char *) bufferADD2, "add 2,0,x.val");
564                     HAL_UART_Transmit(&huart1, bufferADD2, lenADD2, 1000);
565                     HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
566                     free(bufferADD2);
567 
568                     uint8_t *bufferXY = malloc(30*sizeof (char));
569                     int lenXY = sprintf((char *) bufferXY, "x.val=y.val");
570                     HAL_UART_Transmit(&huart1, bufferXY, lenXY, 1000);
571                     HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
572                     free(bufferXY);
573                     uint8_t *bufferDO = malloc(30*sizeof (char));
574                     int lenDO = sprintf((char *) bufferDO, "doevents");
575                     HAL_UART_Transmit(&huart1, bufferDO, lenDO, 1000);
576                     HAL_UART_Transmit(&huart1, Cmd_End, 3, 100);
577                     free(bufferDO);
578 
579                     // Sending to serial
580                     HAL_UART_Transmit(&huart2, resultStr, 5, 10);
581             }
582 }
583 /* USER CODE END 4 */
584 
585 /**
586   * @brief  This function is executed in case of error occurrence.
587   * @retval None
588   */
589 void Error_Handler(void)
590 {
591   /* USER CODE BEGIN Error_Handler_Debug */
592   /* User can add his own implementation to report the HAL error return state
        */
593   __disable_irq();
594   while (1)
595   {
596   }
597   /* USER CODE END Error_Handler_Debug */
598 }
599 
600 #ifdef  USE_FULL_ASSERT
601 /**
602   * @brief  Reports the name of the source file and the source line number
603   *         where the assert_param error has occurred.
604   * @param  file: pointer to the source file name
605   * @param  line: assert_param error line source number
606   * @retval None
```

```
607   */
608 void assert_failed(uint8_t *file, uint32_t line)
609 {
610   /* USER CODE BEGIN 6 */
611   /* User can add his own implementation to report the file name and line
          number,
612      ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line)
          */
613   /* USER CODE END 6 */
614 }
615 #endif /* USE_FULL_ASSERT */
```

Código Python de calibração:

```
# This file basically creates the frequency response graph of a a microfone, in order to give proper
    compensation to the
# measuredRefs of the unknown headphones. Then, it computes the "corrected" measurements with these values
# It gets the input from the serial port from an STM32 recieving a 12b representing voltages from 0 to 3.3V

# Cdigo para a porta de srie retirado de https://www.tinkerassist.com/blog/arduino-serial-port-read.
# Importing data from arduino
import serial.tools.list_ports
# Needed to calculate the logs
import math

# Getting list of com ports
ports = serial.tools.list_ports.comports()
 serialInst = serial.Serial()

 portsList = []

# adding to the list
for onePort in ports:
    portsList.append(str(onePort))
    print(str(onePort))

# Getting the port input
val = input("Select Port: COM")

# Formatting
for x in range(0, len(portsList)):
    if portsList[x].startswith("COM" + str(val)):
        portVar = "COM" + str(val)

# Initializing the port with the UART baud rate
 serialInst.baudrate = 115200
 serialInst.port = portVar
 serialInst.open()

count = 0
readings = []
# Reading the file
while count < 695:
    if serialInst.in_waiting:
        packet = serialInst.readline()
        readings.append(int(packet.decode('utf').rstrip('\n')))
        count = count + 1

# Getting the reference at 1000Hz
reference = readings[393]

# Calculating the dBr
count = 0
for read in readings:
    readings[count] = 1000 * math.log(read/reference, 10)
    count = count + 1

##################################################################
# Pandas seems like the most sane choice for handling data
import pandas as pd
```

```
# Testing the final output
import matplotlib.pyplot as plt

# Creating the measurement data frame
frequencies =
    [20,20.2,20.4,20.61,20.81,21.02,21.23,21.44,21.66,21.87,22.09,22.31,22.54,22.76,22.99,23.22,23.45,23.69,23.92,24.16,24.4,24.6
measurement = pd.DataFrame(list(zip(frequencies, readings)),columns =['frequency', 'raw'])

# Importing the ccsv files that correspond to the reference headphones and the raw measuredRefs
reference = pd.read_csv ('Samson SR850.csv', usecols= ['frequency ',' raw']) # Theoretical

# Correcting theoretical - experimental = correction
correction = pd.DataFrame({'frequency':[], 'raw ':[]})
correction ['frequency '] = measurement['frequency']
correction ['raw'] = reference ["raw"] - measurement['raw']

# Rounding values
correction ['raw'] = correction ['raw'].round(2)

# Saving the dataframe into a CSV Fule
correction .to_csv('correction .csv', index=False)

# Vreating grapsh to better vizualuze corrections
plt.plot( correction ['frequency '], correction ['raw'], label='Mic Correction')
plt.plot( reference ['frequency '], reference ['raw'], label='Known reference')
plt.plot(measurement['frequency'], measurement['raw'], label='Raw measurementt')
plt.legend()
plt.xscale ('log')
plt.show()
```

Código Python de correção:

```
# This file basically creates the frequency response graph of the headphones.
# It uses the previous program output in order to correct the graph

# Cdigo para a porta serial retirado de https://www.tinkerassist.com/blog/arduino-serial-port-read.
# Importing data from arduino
import serial.tools.list_ports
# Needed to calculate the logs
import math

# Getting list of com ports
ports = serial.tools.list_ports.comports()
serialInst = serial.Serial()

portsList = []

# adding to the list
for onePort in ports:
    portsList.append(str(onePort))
    print(str(onePort))

# Getting the port input
val = input("Select Port: COM")

# Formatting
for x in range(0, len(portsList)):
    if portsList[x].startswith("COM" + str(val)):
        portVar = "COM" + str(val)

# Initializing the port with the UART baud rate
serialInst.baudrate = 115200
serialInst.port = portVar
serialInst.open()

count = 0
readings = []
# Reading the file
while count < 695:
    if serialInst.in_waiting:
        packet = serialInst.readline()
        readings.append(int(packet.decode('utf').rstrip('\n')))
        count = count + 1

# Getting the reference at 1000Hz
reference = readings[393]

# Calculating the dBr
count = 0
for read in readings:
    readings[count] = 1000 * math.log(read/reference, 10)
    count = count + 1

##################################################################
# Pandas seems like the most sane choice for handling data
import pandas as pd
# Testing the final output
import matplotlib.pyplot as plt
```

```
# Creating the measurement data frame
frequencies =
    [20,20.2,20.4,20.61,20.81,21.02,21.23,21.44,21.66,21.87,22.09,22.31,22.54,22.76,22.99,23.22,23.45,23.69,23.92,24.16,24.4,24.6
measurement = pd.DataFrame(list(zip(frequencies, readings)),columns =['frequency', 'raw'])

# Importing the ccsv files that correspond to the reference headphones and the raw measuredRefs
reference = pd.read_csv('correction.csv', usecols= ['frequency','raw']) # Theoretical

# Correcting Real = experimental + correction
result = pd.DataFrame({'frequency':[], 'raw':[]})
result['frequency'] = measurement['frequency']
result['raw'] = reference["raw"] + measurement['raw']

# Rounding values
result['raw'] = result['raw'].round(2)

# Saving the dataframe into a CSV Fule
result.to_csv('result.csv', index=False)

# Vreating grapsh to better vizualuze corrections
plt.plot(result['frequency'], result['raw'], label='Corrected')
plt.plot(reference['frequency'], reference['raw'], label='Mic Correction')
plt.plot(measurement['frequency'], measurement['raw'], label='Raw measurementt')
plt.legend()
plt.xscale('log')
plt.show()
```