Revista Ilha Digital
Endereço eletrônico:
http://ilhadigital.florianópolis.ifsc.edu.br/

IMPLEMENTAÇÃO EM MICROCONTROLADOR DE UM COMPRESSOR DE ÁUDIO

JOÃO ANTÔNIO CARDOSO

Resumo: Compressores de áudio estão entre os principais ferramentas das mais variadas etapas nas produções de áudio, permitindo que profissional manipule e modele as características estáticas e dinâmicas no que se diz respeito ao nível de pressão sonora do áudio. Este trabalho implementa um compressor de áudio digital na topologia *feedforward* com *makeup* automático, tendo sua cadeia de processamento inteiramente sendo executada no domínio logarítmico, dentro de um microcontrolador *STM32F4*, implementado em linguagem C. Para tal, um ambiente de desenvolvimento combinando hardware e software foi montado, permitindo a comparação em tempo real com compressores virtuais, servindo de referência durante a implementação de cada etapa, de modo a facilitar seu desenvolvimento. Como resultado, têm-se um compressor micro controlado inteiramente funcional e com características semelhantes ao compressor de referência utilizado.

Palavras-chave: DSP, ARM, Cortex M4, Áudio, Compressor.

Abstract: Audio compressors are among the main tools of the most varied stages of audio production, allowing professionals to manipulate and shape the static and dynamic characteristics with regard to sound pressure level. This work implements a digital audio compressor using a feedforward topology with automatic makeup gain, having its processing signal chain entirely working in logarithm domain inside a Cortex M4 microcontroller implemented using C language. To that end, a development ambient was configured enabling comparison with other compressors in real time, providing a reference during the implementation of each step, facilitating its development. The result is fully functional microcontrolled compressor with similar characteristics to the compreessor used as reference.

Keywords: *DSP, ARM, Cortex M4, Audio, Compressor*

1 INTRODUÇÃO

Sinais de áudio, sejam aqueles captados por microfones ou gerados sinteticamente, dificilmente são utilizados em produções sem haver processamento e um dos processamentos mais utilizados pela indústria é a compressão, que por meio de diversos ajustes, permite que os profissionais modelem a dinâmica dos sinais.

O processo de compressão, que pode ser realizado por um equipamento físico ou por um *software / plugin* de efeitos em um computador, como o exemplo didático desenvolvido por Nair (2013), pode ser entendido como um mapeamento entre os sinais de entrada e saída que resulta na redução da pressão sonora acima de um determinado limiar ajustável, enquanto os valores abaixo deste limiar, podem permanecer intocados (GIANNOULIS; MASSBERG; REISS, 2012).

Um dos casos comuns de sua utilização é no processo de mixagem musical, no qual um engenheiro de áudio utiliza o processo de compressão para reduzir os picos de pressão dos instrumentos musicais percussivos, e a partir desta redução ele pode realçar os sons menos intensos, como a reverberação natural da sala em que a tal

instrumento percussivo foi tocado, ou um outro instrumento gravado ou misturado no mesmo áudio (BROWN, 2019).

2 PRINCÍPIOS E PARÂMETROS DA COMPRESSÃO

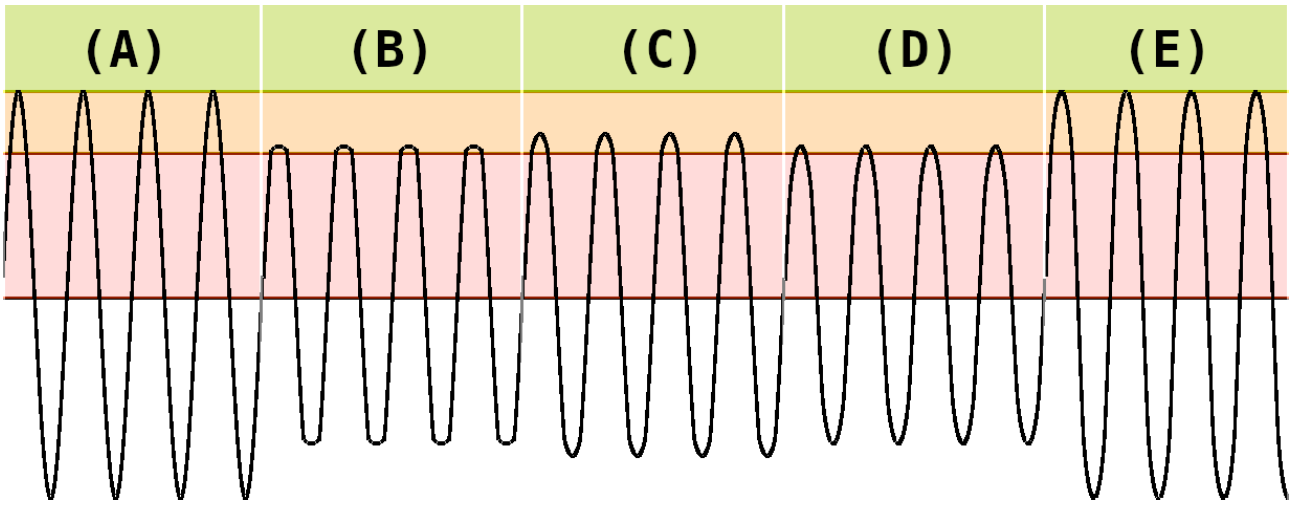
Para compreender com maiores detalhes o compressor, podemos considerar um compressor fictício como um bloco com uma entrada e uma saída de sinal, com seu nível de limiar de compressão (parâmetro *threshold*) ajustado para comprimir sinais mais intensos que -6 dB, como pode ser visto na 1.

Considerando como sinal de entrada do compressor um tom com intensidade máxima de -3dB, se ajustarmos a taxa de compressão (parâmetro *ratio*) para 10:1, o sinal original (mostrado em (A) na Figura 1) ultrapassa o limiar com -3dB, e se ativarmos a compressão, irá resultar num sinal em -6 +3/10 = -5,7 dB (sinal (B) na Figura 1). Seguindo o mesmo processo, se ajustarmos a taxa de compressão para -3dB, teremos o caso (C), onde podemos ver que o pico é maior que em (B), chegando em -6 +3/3 = -5 dB.

Para diminuir possíveis descontinuidades no sinal e por consequência diminuir as distorções e adição de harmônicos no sinal original, a taxa de compressão pode ser suavizada na região do limiar, criando-se um joelho (*knee*) mais suave, antecipando o início da compressão (GIANNOULIS; MASSBERG; REISS, 2012). Se aplicarmos uma largura de joelho (parâmetro *knee width*) de 3dB com a mesma taxa que em (B), obtemos (D).

Como a ação de compressão resulta em uma redução do pico do nosso sinal, podemos maquiar (parâmetro *makeup gain*) esta redução de volume aplicando um ganho adicional em todo o sinal após a compressão, diminuindo a diferença dinâmica entre os picos e vales do sinal, resultando em (E) para a mesma compressão que em (D).

Figura 1 – Diferentes Parâmetros Estáticos de Compressão



Limiar em -6dB, sinal senoidal com -3dB de pico. (A) Sinal sem compressão; (B) Compressão com razão 10:1; (C) Compressão com razão 3:1; (D) 10:1 com suavização no joelho de 3dB; (E) Ganho de maquiagem (automático). Fonte: Do Autor.

De acordo com Brown (2019) o ganho ideal estático do compressor pode ser descrito pela Equação 1, onde x é o sinal entrada (em decibels) $y(x)$ a saída processada, T o limiar, R a taxa de compressão e M a maquiagem.

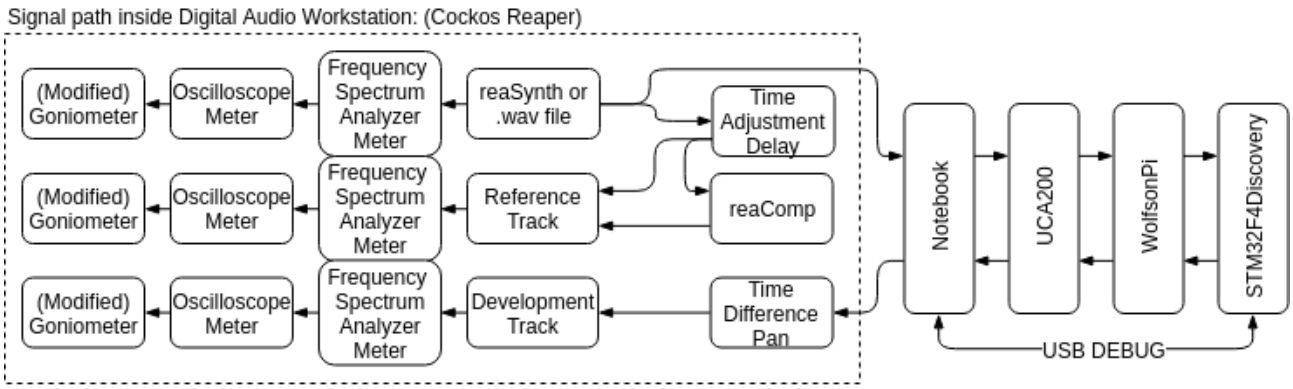
$$y(x) = \frac{x - T}{R} + T + M \tag{1}$$

Outra maneira de suavizar a ação do compressor pode ser adicionando uma inércia, criando um atraso que vai lentamente permitindo que a compressão atue ou que deixe de atuar, gerando características dinâmicas para o compressor. Este processo, descrito por uma região não linear (NAIR, 2013), pode ser chamado de balística e é definido por um tempo de ataque (parâmetro *attack time*) e um tempo de liberação (parâmetro *release time*), que em é geralmente trabalhado na escala de milissegundos (HICKS, 2017).

3 METODOLOGIA PARA O DESENVOLVIMENTO E TESTES

Com a finalidade de realizar o projeto utilizando o processamento digital de sinais (*DSP*), um conjunto de ferramentas foi utilizado, conforme mostra a Figura 2.

Figura 2 – Diagrama de Blocos do Ambiente de Desenvolvimento



Fonte: Do Autor.

Na parte de *hardware*, a plataforma de estudo e desenvolvimento de áudio embarcado tem como principais componentes uma placa *STM32F4Discovery*, que utiliza um microcontrolador *Cortex M4*, e a placa WolfsonPi, que contém o *codec* de áudio *WM5102*, comunicando via *SPI / I2S*, contendo todos os periféricos necessários para duas entradas e duas saídas de áudio. Para que fosse possível utilizar dois canais de entrada e dois canais de saída no *notebook*, uma interface de áudio *UCA200* também foi utilizada, ligando via *USB* no *notebook*.

Para que fosse possível testar e comparar o compressor em desenvolvimento, cada estágio e parâmetro do compressor foi comparado diretamente com um *plugin* nativo da estação de trabalho de áudio digital (*DAW - Digital Audio Workstation*) *Cockos Reaper*, distribuído com licença gratuita para teste (limitada para uso não profissional).

Seguindo o diagrama mostrado na Figura 2, um sinal era gerado dentro da *DAW* e roteado para a saída de áudio do *notebook*, conectado na inteface *UCA200*, que por sua vez conecta no *WolfsonPi*, que transmite seus dados para o compressor programado dentro do microcontrolador, que retorna o sinal processado (comprimido) para o *notebook* até a trilha (de gravação) de desenvolvimento *Development Track*, que por fim tem suas saídas conectadas à blocos para o monitoramento visual das formas de onda deste áudio.

Paralelamente à este fluxo, o mesmo sinal de origem também foi roteado internamente para o compressor digital *reaComp*, terminando em uma trilha (de gravação) chamada de referência (*Reference Track*), que também tinha suas saídas roteadas para blocos de monitoramento visual.

Além do *reaComp*, alguns outros *plugins* distribuidos nesta *DAW* foram utilizados: *reaSynth*, para gerar os tons de teste, *Osciloscope Meter* para visualizar as formas de onda no domínio temporal, *Frequency Spectrum*

Analyzer Meter para visualizar o conteúdo harmônico, *Goniometer* para visualizar as características estáticas do compressor, comparando os canais direita-esquerda, no qual a esquerda é o sinal processado e o canal da direita é o sinal sem processamento. O *Goniometer* é um *plugin* escrito na linguagem *javascript* e foi brevemente customizado para mostrar seus pontos (saída/direita *versus* entrada/esquerda) sem angulação em relação ao eixo horizontal (esquerda/entrada) e vertical (direita/saída).

Por conta do atraso de processamento entre o roteamento de sinal interno da *DAW* (processado pelo compressor *reaComp*) e externo à placa (processado pelo compressor desenvolvido), um atraso de compensação foi ajustado utilizando o *plugin Time Adjustment Delay*.

No caso da trilha de desenvolvimento (*Development Track*), referente ao áudio processado pelo compressor em desenvolvimento, havia um atraso que dependia da frequência do sinal de origem, provavelmente um defasamento não homogêneo para os filtros analógicos dos canais direita/esquerda da placa *WolfsonPi* ou da interface *UCA200*, para que o *Goniometer* mostrasse entrada vs saída com as amostras referentes ao mesmo instante de tempo, o *plugin Time Difference Pan* foi utilizado para ajustar a fase entre os canais.

Desta forma é possível, comparar a resposta estática, dinâmica, e conteúdo harmônico em tempo real entre o compressor implementado e um compressor de referência, sendo possível validar cada estágio e ajuste dos parâmetros do compressor.

4 IMPLEMENTAÇÃO

Um compressor pode ser implementado utilizando diferentes filosofias (KRUSCH, 2013), sendo uma das principais, o modo com que o controlador de ganho atua sob o fluxo de sinal: ele pode ser *antealimentado* (*feedforward*), que utiliza uma cópia do sinal de entrada para calcular o ganho de um amplificador de compressão, podendo atuar (teoricamente) sem nenhum atraso; ou ele pode ser com *retroalimentado* (*feedback*), agindo de acordo conforme suas ações passadas para calcular o ganho do amplificador de compressão, atuando necessariamente com um atraso, mas podendo corrigir não idealidades que provenientes do próprio processo não ideal (ou não modelado) de compressão.

Neste trabalho foi escolhido trabalhar com uma topologia *feedforward* operando inteiramente no domínio logarítmico (Figura 3), tendo como vantagem a utilização de operações de soma ao invés de multiplicações para a aplicação do ganho dentro do microcontrolador, o que pode reduzir o peso de processamento.

Figura 3 – Compressor *Feedforward* no domínio logarítmico

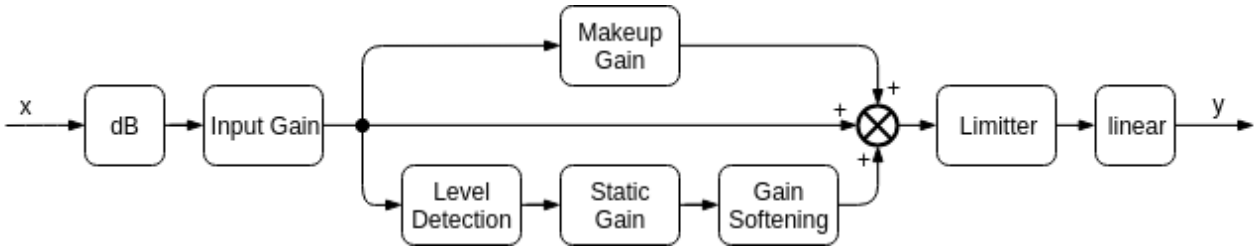


Diagrama de blocos da topologia *feedforward* para compressores atuando inteiramente no domínio logarítmico (em decibéis). Fonte: Do Autor.

Seguindo o fluxo do sinal de entrada x, o primeiro estágio se trata da conversão do domínio linear para logarítmico, executado conforme Equação 2.

É importante notar que aqui, x é um valor representado em ponto flutuante de precisão simples (*float 32*

bits) e provém da conversão de um conversor analógico digital localizado no *codec* do *WolfsonPi*, que precisa ser corretamente configurado para que um sinal gerado pelo *reaSynth* de, por exemplo, -3dB seja de fato convertido para -3dB dentro do microcontrolador, e por conta de uma série de filtros não ideais no caminho do sinal, este valor foi aqui ajustado para que um sinal tom puro em 220 Hz de -6dB fosse identificado pelo microcontrolador, após a conversão para o domínio logarítmico, com exatamente -6dB. Como o ganho dos filtros externos não foi possível ser compensado com precisão pelos PGAs (*Programmable Gain Amplifiers* de entrada e saída do *WolfsonPi*, a conversão entre os valores entregues pelo *codec*, que está configurado para utilizar representação no formato inteiro de 16 *bits* com sinal por amostra, foi modificado para uma razão de $40313.44708198012 bits/V$ ao invés de $2^{15} bits/V$.

Outro detalhe importante é que ao passarmos o valor para decibel, perdemos a informação de seu sinal e passamos a trabalhar com o módulo de seu nível de pressão, sendo necessário guardar esta informação para que seja possível reconstruir o sinal original, no caso de trabalharmos aplicando o ganho no domínio logarítmico.

$$x[n]_{dB} = 20 \log_{10}(x[n]_{linear}) \tag{2}$$

O ganho de entrada pode ser simplesmente somando-se o ganho ao sinal no domínio já logarítmico (Equação 3).

$$x[n] = x[n] + G_{in} \tag{3}$$

O próximo estágio é a detecção de nível (*Level Detection*), que foi optado por ser por meio da detecção de pico com tempo de integração nulo, pela simplicidade de implementação. Outras possibilidades são amplamente exploradas pelos fabricantes, se destacando principalmente a detecção por nível RMS (GIANNOULIS; MASSBERG; REISS, 2012), possuindo um tempo e integração definido pelo fabricante, ou disponibilizado como um parâmetro para o usuário.

Após a detecção de nível, para um sinal de entrada $x[n]$, o ganho de compressão (G) com largura de joelho suave (W) marginal ao nível de limiar T pode ser aproximado por uma equação de segundo grau, obtendo o sistema de inequações descritos em 4, conforme demonstrado por Dutilleux (2011), Giannoulis, Massberg e Reiss (2012) e (WEI; SHIMIZU, 2004).

$$G(x)[n] = \begin{cases} x[n] & , \ T - \frac{W}{2} > x[n] \\ x[n] + \frac{\left(\frac{1}{R} - 1\right)\left(x[n] - T + \frac{W}{2}\right)^2}{2W} & , \ T - \frac{W}{2} \leq x[n] \leq T + \frac{W}{2} \\ T + \frac{x[n] - T}{R} & , \quad x[n] > T + \frac{W}{2} \end{cases} \tag{4}$$

Obtido o ganho de compressão G, o próximo estágio realiza a suavização do ganho utilizando os parâmetros de tempo ataque τ_A e de liberação τ_R, obtendo um ganho suave $G_s[n]$, também utilizando o ganho suave anterior $G_s[n-1]$ (DUTILLEUX, 2011).

$$G_s[n] = \begin{cases} \alpha_A G_s[n-1] + (1 - \alpha_A) G[n] & , \ T - G \leq G_s[n-1] \\ \alpha_R G_s[n-1] + (1 - \alpha_R) G[n] & , \ T - G > G_s[n-1] \end{cases} \tag{5}$$

$$\alpha_A = e^{\left(\frac{-1}{F_s * \tau_A}\right)} \tag{6}$$

$$\alpha_B = e^{\left(\frac{-1}{F_s * \tau_B}\right)} \tag{7}$$

O estágio de maquiagem automática implementado utiliza uma compensação da metade do ganho de compressão (sem suavização) aplicado para um sinal de $0dB$ (GIANNOULIS; MASSBERG; REISS, 2012), mantendo margem suficiente para suportar a maior parte dos picos que ultrapassam o limiar sem serem comprimidos devido às dinâmicas não lineares proporcionadas pelo bloco de suavização de ganho. Por meio de comparação foi possível verificar que o compressor de referência também utiliza esta implementação para seu cálculo de ganho de maquiagem automático (M), que pode ser descrito pela Equação 8.

$$M = -\frac{G(0)}{2} \tag{8}$$

Tendo o ganho suavizado G_s e o ganho de maquiagem M, o ganho final de compressão pode ser aplicado simplesmente pela soma dos ganhos ao sinal original, como descrito em Equação 9 (WEI; SHIMIZU, 2004).

$$y[n] = x[n] + G_s + M \tag{9}$$

Após ser aplicado o sinal ser comprimido e devidamente maquiado, é possível que sobre algum resquício de sinal com um nível suficientemente alto para distorcer a saída deste equipamento e a ainda a entrada do próximo, por isso, um limitador/ceifador pode ser empregado, saturando o sinal em um valor determinado L, resultando em Equação 10.

$$x[n] = \begin{cases} x[n] & , x[n] \leq L \\ L & , x[n] > L \end{cases} \tag{10}$$

Por fim, é necessário realizar a conversão do domínio logarítmico para o domínio linear para que o conversor digital-analógico possa reproduzir o sinal corretamente, tal conversão pode ser realizada pela Equação 11.

$$y[n]_{linear} = 10^{(0.05\,x[n]_{dB})} \tag{11}$$

Para reconstruir cada amostra corretamente, é necessário considerar o sinal (negativo ou positivo) que foi armazenado antes de realizar a conversão para o domínio logarítmico.

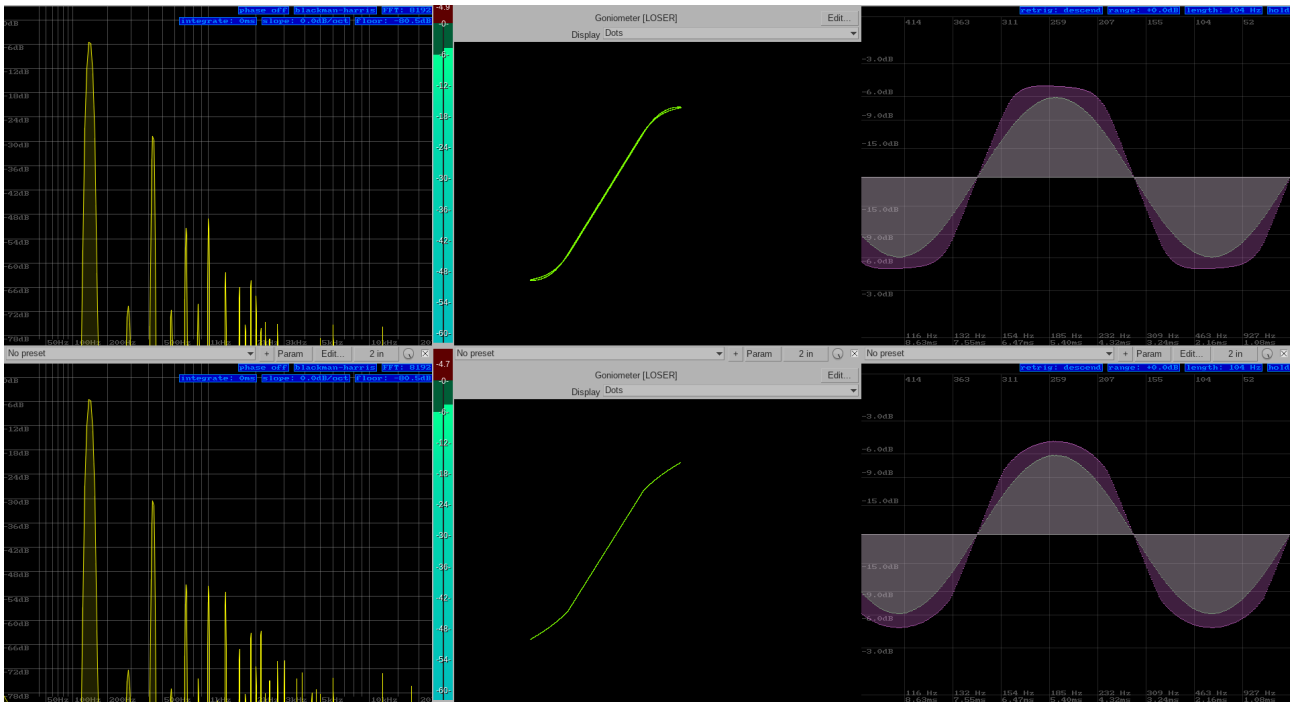
5 RESULTADOS E DISCUSSÃO

A partir da implementação na linguagem de programação *C* diretamente no microcontrolador das equações aqui apresentadas (Apêndice B), bloco-a-bloco elas foram testadas e validadas.

Aplicando um tom puro as característica estáticas do compressor implementado podem ser comparadas com o compressor de referência (Figura 4). Pode ser percebido que seu conteúdo harmônico, sua forma de onda e sua curva de ganho estático estão bastante semelhantes, o que significa que as características estáticas desejadas foram alcançadas. É possível notar, através da curva de ganho estático, que o compressor de referência utiliza uma aproximação menos suave que a implementada aqui, possivelmente sendo uma aproximação de primeira ordem ao invés de segunda ordem.

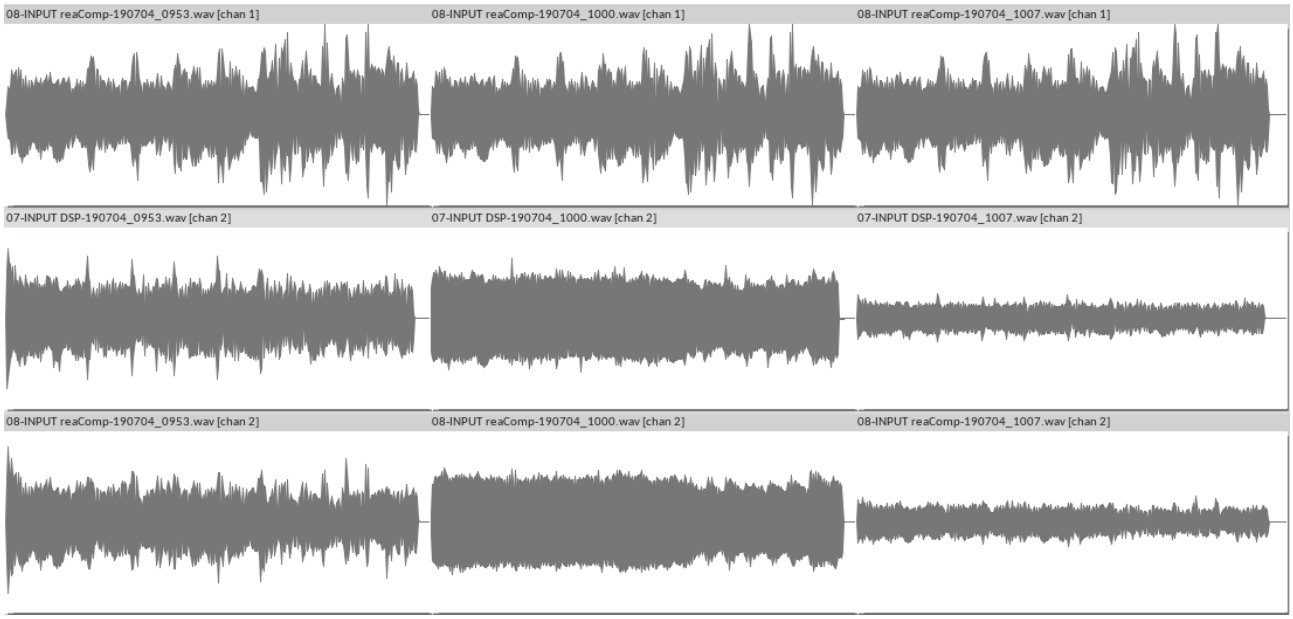
Por último, foram escolhidas algumas configurações para ser realizada uma análise auditiva e visual comparando a implementação com o compressor de referência (*reaComp*), que pode ser visto na Figura 5, na mesma ordem que as formas de onda estão apresentadas, da direita para a esquerda, a Tabela 1 apresenta as configurações utilizadas.

Figura 4 – Comparação das características estáticas do compressor



Da direita para a esquerda temos: As imagens superiores representam o sinal processado pelo compressor implementado e as inferiores representam o sinal processado pelo compressor de referência (*reaComp*). Fonte: Do Autor.

Figura 5 – Comparação das características dinâmicas do compressor



De cima para baixo: Áudio original, áudio processado pela implementação apresentada e por último, o áudio processado pelo compressor de referência (*reaComp*). Fonte: Do Autor.

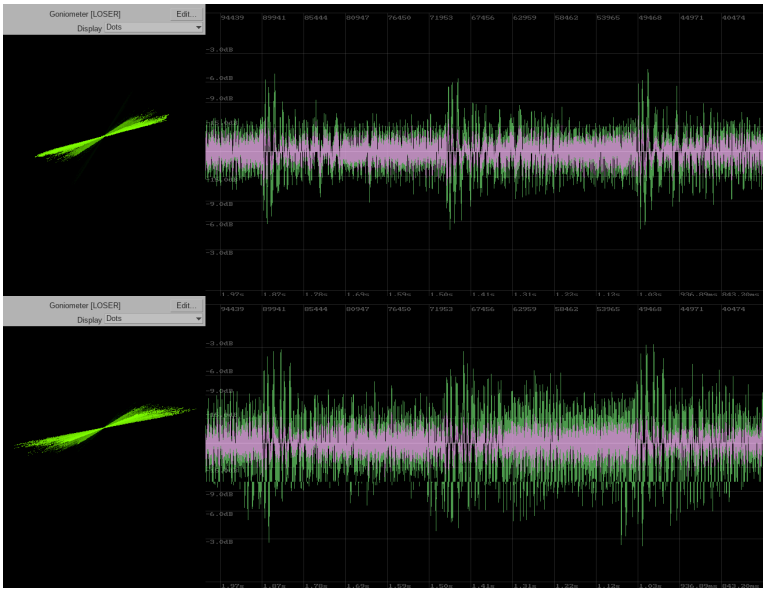
Tabela 1 – Parâmetros utilizados nos testes

Parâmetro	Teste 1	Teste 2	Teste 3
Input Gain (dB)	0	0	0
Ratio	10	13.4	19.7
Threshold (dB)	-25.4	-21.4	-21.1
Knee Width (dB)	1	3	5.9
Attack Time (ms)	8.8	0.08	5.6
Release Time (ms)	135	0	44
Auto Makeup	SIM	SIM	NÃO
Clipping (dB)	-2	-2	-2

Os resultados individuais de cada um dos três testes pode ser visto com mais detalhes em Apêndice A.

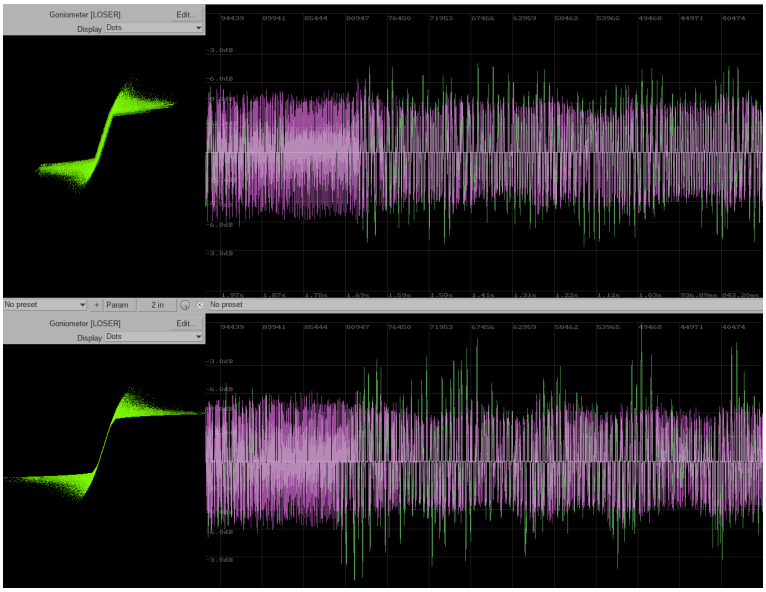
APÊNDICE A – RESULTADOS INDIVIDUAIS DOS TESTES

Figura 6 – Resultados para o Teste 1



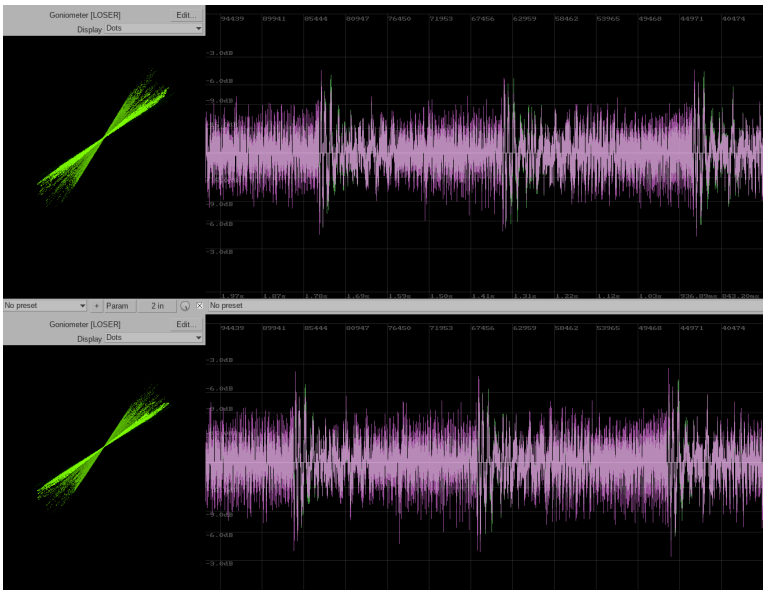
Parte superior representa os resultados visuais do compressor implementado, e a parte inferior, o compressor de referência. Fonte: Do Autor.

Figura 7 – Resultados para o Teste 2



Parte superior representa os resultados visuais do compressor implementado, e a parte inferior, o compressor de referência. Fonte: Do Autor.

Figura 8 – Resultados para o Teste 3



Parte superior representa os resultados visuais do compressor implementado, e a parte inferior, o compressor de referência. Fonte: Do Autor.

APÊNDICE B – CÓDIGO EM C COM PRINCIPAIS DEFINIÇÕES

```
//#include <stm32f4xx.h>
//#include <stm32f4xx_hal.h>
//#include <stm32f4xx_hal_gpio.h>
//#include <stm32f4_discovery.h>
#include <dwt.h>
#include "compressor.h"
#include "audio_config.h"
#include <wolfson_pi_audio.h>

int16_t TxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
int16_t RxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];

__IO BUFFER_StateTypeDef buffer_offset = BUFFER_OFFSET_NONE;
__IO uint8_t Volume = VOLUME;
GPIO_InitTypeDef GPIO_InitStructure;

void wolfson_pi_init(void);
void stm_init(void);

void wolfson_pi_init(void)
{
    WOLFSON_PI_AUDIO_Init((INPUT_DEVICE_LINE_IN << 8) | OUTPUT_DEVICE_BOTH, 80, AUDIO_FREQUENCY);
    WOLFSON_PI_AUDIO_SetInputMode(INPUT_DEVICE_LINE_IN);
    WOLFSON_PI_AUDIO_SetMute(AUDIO_MUTE_ON);
    WOLFSON_PI_AUDIO_Play(TxBuffer, RxBuffer, WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE);
    WOLFSON_PI_AUDIO_SetVolume(Volume);
```

```
}

void stm_init(void)
{
        // Initialise the HAL Library; it must be the first
        // instruction to be executed in the main program.
        HAL_Init();
        DWT_Enable();

    //Push Button
        __GPIOA_CLK_ENABLE();
        GPIO_InitStructure.Pin   = GPIO_PIN_0;
        GPIO_InitStructure.Mode  = GPIO_MODE_INPUT;
        GPIO_InitStructure.Pull  = GPIO_NOPULL;
        //GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);

        //LED:
        __GPIOD_CLK_ENABLE();
        GPIO_InitStructure.Pin   = GPIO_PIN_12;
        GPIO_InitStructure.Mode  = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStructure.Pull  = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
        HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);

        __GPIOD_CLK_ENABLE();
        GPIO_InitStructure.Pin   = GPIO_PIN_13;
        GPIO_InitStructure.Mode  = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStructure.Pull  = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
        HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);

        __GPIOD_CLK_ENABLE();
        GPIO_InitStructure.Pin   = GPIO_PIN_14;
        GPIO_InitStructure.Mode  = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStructure.Pull  = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
        HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);

        __GPIOD_CLK_ENABLE();
        GPIO_InitStructure.Pin   = GPIO_PIN_15;
        GPIO_InitStructure.Mode  = GPIO_MODE_OUTPUT_PP;
        GPIO_InitStructure.Pull  = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_HIGH;
        HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);

    LED0_on();
    LED1_on();
    LED2_on();
    LED3_on();
}

int main(int argc, char* argv[])
{
    UNUSED(argc);
```

```
    UNUSED(argv);

    stm_init();
    wolfson_pi_init();

    // Compressor initialization
    compressor_parameter_t compressor_parameter;
    compressor_t compressor;
    compressor_init(&compressor_parameter, &compressor);

    LED0_off();
    LED1_off();
    LED2_off();
    LED3_off();

    // Left-Right matrix selection macros
    #ifdef BYPASS_L
    #define FUN_L(i)  RxBuffer[i]
    #else
    #define FUN_L(i)  compressor_stage(RxBuffer[i], &compressor)
    #endif
    #ifdef BYPASS_R
    #define FUN_R(i)  RxBuffer[i]
    #else
    #define FUN_R(i)  compressor_stage(RxBuffer[i], &compressor)
    #endif

    for(;;){
            static uint32_t i = 0;
                if(buffer_offset == BUFFER_OFFSET_HALF){
                        DWT_Reset();
                        for(i = 0; i < (WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE / 2); i++){
                                        TxBuffer[i] = FUN_L(i);
                                        i++;
                                        TxBuffer[i] = FUN_R(i);
                        }
                        buffer_offset = BUFFER_OFFSET_NONE;
                }

                if(buffer_offset == BUFFER_OFFSET_FULL){
                        DWT_Reset();
                        for(i = (WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE / 2); i < WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++){
                                        TxBuffer[i] = FUN_L(i);
                                        i++;
                                        TxBuffer[i] = FUN_R(i);
                        }
                        buffer_offset = BUFFER_OFFSET_NONE;
                }
        }

    return 0;
}

/*-------------------------------------------------------
 Callbacks implementation:
```

```
 ------------------------------------------------------*/

/**
 * @brief    Manages the DMA full Transfer complete event.
 */
void WOLFSON_PI_AUDIO_TransferComplete_CallBack (void)
{
    buffer_offset = BUFFER_OFFSET_FULL;
}

/**
 * @brief    Manages the DMA Half Transfer complete event.
 */
void WOLFSON_PI_AUDIO_HalfTransfer_CallBack (void)
{
    buffer_offset = BUFFER_OFFSET_HALF;
}

/**
 * @brief    Manages the DMA FIFO error interrupt.
 * @param    None
 * @retval None
 */
void WOLFSON_PI_AUDIO_OUT_Error_CallBack (void)
{
    /* Stop the program with an infinite loop */
    for(;;);
}
```

```
/*
 * audio_config.h
 *
 *  Created on: Jul 3, 2019
 *      Author: joaoantoniocardoso
 */

#ifndef AUDIO_CONFIG_H_
#define AUDIO_CONFIG_H_

#include <wolfson_pi_audio.h>

#define AUDIO_FREQUENCY   AUDIO_FREQUENCY_96K
#define BYPASS_L
#define VOLUME 0x80                                          // 0x80 = 0dB

#endif /* AUDIO_CONFIG_H_ */
```

```
/*
 * compressor.h
 *
 *  Created on: Jul 3, 2019
 *      Author: joaoantoniocardoso
```

```
 */

#ifndef COMPRESSOR_H_
#define COMPRESSOR_H_

#include <arm_math.h>
#include "audio_config.h"
#include <leds_and_buttons.h>

#define COMPRESSOR_SETUP_DEFAULT
//#define COMPRESSOR_SETUP_ILHA_DIGITAL_1
//#define COMPRESSOR_SETUP_ILHA_DIGITAL_2
//#define COMPRESSOR_SETUP_ILHA_DIGITAL_3
#define AUTO_MAKEUP_ON

/**
 * @brief   Compressor Parameters
 * @param   sample is the signal
 * @param   gain_db is the pre-gain or input-gain level (dB)
 * @param   ratio is the compression ratio
 * @param   th_db is the threshold level (dB)
 * @param   w_db is the knee-width (dB)
 * @param   att is the attack time (ms)
 * @param   rlt is the release time (ms)
 * @param   makeup_db is the makeup gain level (dB)
 * @param   clip_db is the clip level (dB)
 */
typedef struct compressor_parameter_t{
    float32_t gain_db;
    float32_t ratio;
    float32_t th_db;
    float32_t w_db;
    float32_t w_db_x2_inv;
    float32_t k1_db;
    float32_t k2_db;
    float32_t att;
    float32_t rlt;
    float32_t tau_attack;
    float32_t tau_release;
    float32_t clip_db;
    float32_t alpha_attack;
    float32_t alpha_release;
}compressor_parameter_t;

typedef struct compressor_t{
    compressor_parameter_t *param;
    float32_t sample;
    int32_t sign;
}compressor_t;

#define int_to_float(x)   ((float32_t)(x/40313.44708198012f))
#define float_to_int(x)   ((int16_t)(x*40313.44708198012f))

float32_t linear_to_db(float32_t linear_value);
float32_t db_to_linear(float32_t db_value);
```

```
void compressor_parameter_init(compressor_parameter_t *p);
void compressor_init(compressor_parameter_t *p, compressor_t *c);
float pre_gain_db(compressor_t *c);
void sign_detection(compressor_t *c);
void limitter_stage_db(compressor_t *c);
float gain_stage_db(compressor_t *c, float sample_db);
float ballistics_stage_db(compressor_t *c, float gain_compressor_stage_db);
int16_t compressor_stage(int16_t b, compressor_t *c);
float auto_makeup_stage_db(compressor_t *c);

// Compressor setups
#ifdef COMPRESSOR_SETUP_DEFAULT
       #define GAIN_DB                0                       // pre-gain level (dB)
       #define RATIO                 10                      // compress ratio
       #define TH_DB                 -9                      // threshold level (dB)
       #define W_DB                3                       // knee-width (dB)
       #define ATT                         0                       // attack time (ms)
       #define RLT                         0                       // release time (ms)
       #define CLIP                -2                      // clipping level (dB)
#endif
#ifdef COMPRESSOR_SETUP_ILHA_DIGITAL_1
       #define GAIN_DB                0                       // pre-gain level (dB)
       #define RATIO                 10                      // compress ratio
       #define TH_DB                 -25.4                 // threshold level (dB)
       #define W_DB                1                       // knee-width (dB)
       #define ATT                         8.8                         // attack time (ms)
       #define RLT                         135.0                 // release time (ms)
       #define CLIP                -2                      // clipping level (dB)
#endif
#ifdef COMPRESSOR_SETUP_ILHA_DIGITAL_2
       #define GAIN_DB                0                       // pre-gain level (dB)
       #define RATIO                 13.4                  // compress ratio
       #define TH_DB                 -21.4                  // threshold level (dB)
       #define W_DB                3                       // knee-width (dB)
       #define ATT                         0.08                  // attack time (ms)
       #define RLT                         0                       // release time (ms)
       #define CLIP                -2                      // clipping level (dB)
#endif
#ifdef COMPRESSOR_SETUP_ILHA_DIGITAL_3
       #define GAIN_DB                0                       // pre-gain level (dB)
       #define RATIO                 19.7                  // compress ratio
       #define TH_DB                 -21.1                  // threshold level (dB)
       #define W_DB                5.9                           // knee-width (dB)
       #define ATT                         5.6                 // attack time (ms)
       #define RLT                         44                          // release time (ms)
       #define CLIP                -2                      // clipping level (dB)
#endif


#endif /* COMPRESSOR_H_ */
```

```
/*
 * compressor.c
 *
```

```
 *  Created on: Jul 3, 2019
 *      Author: joaoantoniocardoso
 */


#include "compressor.h"


float32_t linear_to_db(float32_t linear_value)
{
        return 20.0f * log10f(linear_value);
}
float32_t db_to_linear(float32_t db_value)
{
        return powf(10, 0.05f * db_value);
}

void compressor_parameter_init(compressor_parameter_t *p)
{
    // Apply parameters limits
    if(p->gain_db > 20)                 p->gain_db = 20;
    else if(p->gain_db < -10)           p->gain_db = -10;
    else if(p->ratio < 1)                   p->ratio = 1;
    if(p->th_db > 0)                    p->th_db = 0;
    if(p->w_db < 0)                     p->w_db = 0;
    else if(p->w_db > -p->th_db)        p->w_db = -p->th_db;
    if(p->att > 500)                        p->att = 500;
    if(p->att < 0)                          p->att = 0;
    if(p->rlt > 1000)                       p->rlt = 1000;
    if(p->rlt < 0)                          p->rlt = 0;
    if(p->clip_db > -2)                 p->clip_db = -2;
    else if(p->clip_db < -20)           p->clip_db = -20;

    p->w_db_x2_inv = 1.f / (4.f * p->w_db);
    p->k1_db = p->th_db -p->w_db;                                   // knee minimum level (dB)
    p->k2_db = p->th_db +p->w_db;                                   // knee maximum level (dB)
    p->ratio = 1 / p->ratio;                                       // ratio to multiply instead divide
    p->tau_attack = p->att * 0.001f;                                       // attack time
    p->tau_release = p->rlt * 0.001f;                                      // release time
    p->alpha_attack = expf(-1 / (p->tau_attack * AUDIO_FREQUENCY));
    p->alpha_release = expf(-1 / (p->tau_release * AUDIO_FREQUENCY));
}

void compressor_init(compressor_parameter_t *p, compressor_t *c)
{
    p->gain_db = GAIN_DB;
    p->ratio = RATIO;
    p->th_db = TH_DB;
    p->w_db = W_DB;
    p->att = ATT;
    p->rlt = RLT;
    p->clip_db = CLIP;

    compressor_parameter_init(p);
    c->param = p;
```

```
    c->sign = 1;
}

float pre_gain_db(compressor_t *c)
{
        float pre_gain_db = c->param->gain_db;
        return pre_gain_db;
}

void sign_detection(compressor_t *c)
{
    if(c->sample < 0)   c->sign = -1;
    else                c->sign = 1;
}

void limitter_stage_db(compressor_t *c)
{
    if(c->sample > c->param->clip_db){
        LED2_on();
        c->sample = c->param->clip_db;
    }else{
        LED2_off();
    }
}

float gain_stage_db(compressor_t *c, float sample_db)
{
        float gain_compressor_stage_db;

        if(sample_db < c->param->k1_db){
                LED0_off();
                gain_compressor_stage_db = 0;
        }else if(sample_db <= c->param->k2_db){
                LED0_on();
                gain_compressor_stage_db = (c->param->ratio -1) *
                                (sample_db -c->param->k1_db) *
                                (sample_db -c->param->k1_db) *
                                c->param->w_db_x2_inv;
        }else{
                LED0_on();
                gain_compressor_stage_db =
                                (c->param->ratio -1) *
                                (sample_db -c->param->th_db);
        }

        return gain_compressor_stage_db;
}

float ballistics_stage_db(compressor_t *c, float gain_compressor_stage_db)
{
        static float gain_ballisctics_stage_db = 0;

        if(gain_compressor_stage_db <= gain_ballisctics_stage_db){
                gain_ballisctics_stage_db =
                                c->param->alpha_attack * gain_ballisctics_stage_db +
```

```
                                (1 - c->param->alpha_attack) * gain_compressor_stage_db;
        }else{
                gain_ballisctics_stage_db =
                                c->param->alpha_release * gain_ballisctics_stage_db +
                                (1 - c->param->alpha_release) * gain_compressor_stage_db;
        }

        return gain_ballisctics_stage_db;
}

float auto_makeup_stage_db(compressor_t *c)
{
        float gain_makeup_stage_db;
        float sample_db = 0;

        gain_makeup_stage_db = -gain_stage_db(c, sample_db)/2;

        return gain_makeup_stage_db;
}

int16_t compressor_stage(int16_t b, compressor_t *c)
{
    c->sample = int_to_float(b);
    sign_detection(c);

    // COMPRESSOR STAGE
    c->sample = linear_to_db(c->sign * c->sample);
    static float gain_db = 0;

    gain_db = pre_gain_db(c);
    c->sample += gain_db;

    gain_db = gain_stage_db(c, c->sample);
    gain_db = ballistics_stage_db(c, gain_db);
#ifdef AUTO_MAKEUP_ON
    gain_db += auto_makeup_stage_db(c);
#endif

        if(BT0_on()){
                LED3_on();
        }else{
                LED3_off();
                c->sample += gain_db;
        }

        limitter_stage_db(c);

    c->sample = c->sign * db_to_linear(c->sample);

    return float_to_int(c->sample);
}
```

REFERÊNCIAS

BROWN, G. *Audio Dynamics 101: Compressors, Limiters, Expanders, and Gates*. 2019. Disponível em: <https://www.izotope.com/en/blog/music-production/audio-dynamics-101-compressors-limiters-expanders-and-gates.html>. 2

DUTILLEUX, P. Nonlinear processing. In: ____. *DAFX: digital audio effects*. [S.l.]: Wiley, 2011. p. 554. 5

GIANNOULIS, D.; MASSBERG, M.; REISS, J. Digital dynamic range compressor design—a tutorial and analysis. *AES: Journal of the Audio Engineering Society*, v. 60, 07 2012. 1, 2, 5, 6

HICKS, M. *Audio Compression Basics*. 2017. Disponível em: <https://www.uaudio.com/blog/audio-compression-basics/>. 3

KRUSCH, M. *Tegeler Audio Manufaktur*. 2013. Disponível em: <https://www.tegeler-audio-manufaktur.de/News/side-chain-what-it-is-and-what-it-isnt-part-1>. 4

NAIR, V. *Tutorial: A compressor in Pure Data*. 2013. Disponível em: <http://designingsound.org/2013/06/28/tutorial-a-compressor-in-pure-data/>. 1, 3

WEI, S.; SHIMIZU, K. Audio dynarnic range compression characteristics based on an interpolating polynomial. In: *The 2nd Annual IEEE Northeast Workshop on Circuits and Systems, 2004. NEWCAS 2004.* IEEE, 2004. Disponível em: <https://doi.org/10.1109/newcas.2004.1359040>. 5, 6