

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE
SANTA CATARINA – CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE ELETRÔNICA
CURSO SUPERIOR DE ENGENHARIA ELETRÔNICA**

OTÁVIO AUGUSTO JOENCK FREIRE

**DESENVOLVIMENTO E IMPLEMENTAÇÃO DE EFEITOS DIGITAIS EM TEMPO
REAL PARA GUITARRA: uma abordagem de Processamento Digital de Sinais**

Florianópolis, 2024

OTÁVIO AUGUSTO JOENCK FREIRE

**DESENVOLVIMENTO E IMPLEMENTAÇÃO DE EFEITOS DIGITAIS EM TEMPO
REAL PARA GUITARRA: uma abordagem de Processamento Digital de Sinais**

Monografia apresentada ao Curso de
Engenharia Eletrônica do Campus
Florianópolis do Instituto Federal de Santa
Catarina para a obtenção do diploma de
Engenheiro Eletrônico

Orientador: Carlos Gontarski Speranza

Florianópolis, 2024

Ficha de identificação da obra elaborada pelo autor.

Freire, Otávio Augusto Joenck
Desenvolvimento e implementação de efeitos digitais
em tempo real para guitarra: uma abordagem de Processamento
Digital de Sinais / Otávio Augusto Joenck Freire;
orientação de Carlos Gontarski Speranza. - Florianópolis,
SC, 2024.

89 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal
de Santa Catarina, Câmpus Florianópolis. Bacharelado
em Engenharia Eletrônica. Departamento Acadêmico
de Eletrônica.
Inclui Referências.

1. Efeitos digitais de guitarra. 2. Processamento
digital de sinais. 3. Kit de desenvolvimento STM32F4DISCOVERY.
4. Processamento em tempo real. 5. Características
Sonoras. I. Speranza, Carlos Gontarski . II.
Instituto Federal de Santa Catarina. III. Desenvolvimento
e implementação de efeitos digitais em tempo real para

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE EFEITOS DIGITAIS EM TEMPO REAL PARA GUITARRA: uma abordagem de Processamento Digital de Sinais

OTÁVIO AUGUSTO JOENCK FREIRE

Este trabalho foi julgado adequado para obtenção do título em Engenheiro Eletrônico, pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina, e aprovado na sua forma final pela comissão avaliadora abaixo indicada.

Florianópolis, 19 de dezembro de 2023.

Documento assinado digitalmente
 **CARLOS GONTARSKI SPERANZA**
Data: 07/02/2024 10:39:41-0300
Verifique em <https://validar.iti.gov.br>

Prof. Carlos Gontarski Speranza
Orientador
Instituto Federal de Santa Catarina

Documento assinado digitalmente
 **PEDRO GIASSI JUNIOR**
Data: 09/02/2024 10:39:50-0300
Verifique em <https://validar.iti.gov.br>

Prof. Pedro Giassi Junior
Instituto Federal de Santa Catarina

Documento assinado digitalmente
 **ALEXANDRE LEIZOR SZCZUPAK**
Data: 09/02/2024 15:58:46-0300
Verifique em <https://validar.iti.gov.br>

Prof. Alexandre Leizor Szczupak
Instituto Federal de Santa Catarina

AGRADECIMENTOS

A conclusão deste trabalho acadêmico é fruto de um esforço coletivo, e eu gostaria de agradecer a todos que contribuíram para o seu sucesso.

Em primeiro lugar, gostaria de agradecer à minha família, que sempre me apoiou e incentivou em minha jornada acadêmica. Minha mãe, Maria Cristina, meu pai, René, e minha irmã, Manoela, sempre estiveram ao meu lado, oferecendo-me amor, carinho e orientação.

Também gostaria de agradecer ao Instituto de Educação, Ciência e Tecnologia de Santa Catarina (IFSC), que me proporcionou uma excelente formação acadêmica. Em especial, gostaria de agradecer ao meu professor orientador, Carlos Gontarski Speranza, que me guiou e me apoiou durante todo o desenvolvimento deste trabalho.

Além disso, gostaria de agradecer aos professores do curso superior de engenharia eletrônica do IFSC, que me transmitiram seus conhecimentos e me ajudaram a crescer como profissional e como pessoa.

Por fim, gostaria de agradecer aos meus colegas de curso, que sempre foram um apoio e uma fonte de inspiração.

Agradeço a todos por fazerem parte desta conquista.

Com grandes poderes vêm grandes
responsabilidades (Stan Lee).

RESUMO

Este trabalho investiga alguns exemplos de efeitos sonoros digitais de guitarra, concentrando-se na implementação destes através de processamento digital de sinais (DSP). O estudo abrange a análise teórica e prática dos efeitos, explorando suas características sonoras e aplicações musicais. Para implementação de diversos efeitos digitais em tempo real foi utilizando o kit de desenvolvimento STM32F4DISCOVERY, uma plataforma versátil e de baixo custo. A combinação de conceitos teóricos com a aplicação prática por meio do desse *hardware* permite uma compreensão aprofundada da interação entre os aspectos teóricos e práticos do processamento de efeitos de guitarra. Para comprovar de forma quantitativa a aplicação dos efeitos, diversos resultados são apresentados e analisados neste trabalho.

Palavras-chave: Efeitos digitais de guitarra; processamento digital de sinais; kit de desenvolvimento STM32F4DISCOVERY; processamento em tempo real; características sonoras; aplicações musicais.

ABSTRACT

This work investigates some examples of digital guitar sound effects, focusing on their implementation through digital signal processing (DSP). The study encompasses both theoretical and practical analysis of these effects, exploring their sonic characteristics and musical applications. For the real-time implementation of various digital effects, the STM32F4DISCOVERY development kit was utilized, offering a versatile and cost-effective platform. The combination of theoretical concepts with practical application using this hardware allows for a comprehensive understanding of the interaction between theoretical and practical aspects of guitar effects processing. To quantitatively validate the application of these effects, various results are presented and analyzed in this study.

Keywords: Digital guitar effects; digital signal processing; STM32F4DISCOVERY development kit; real-time processing; sonic characteristics; musical applications.

LISTA DE ILUSTRAÇÕES

Figura 1 – Diagrama de blocos do efeito <i>delay</i> . O sinal de saída contém o sinal de entrada (<i>input</i>) que será misturado com o sinal de entrada atrasado (<i>D samples delay</i>) com uma atenuação(<i>a</i>).....	21
Figura 2– Diagrama de blocos do efeito vibrato. O sinal de saída (<i>y(n)</i>) contém o sinal de entrada (<i>x(n)</i>) que sofrerá um atraso variante no tempo devido ao uso de um oscilador de baixa frequência LFO ($\sin(2\pi fmn)$).....	22
Figura 3 - Diagrama de blocos do efeito chorus. O sinal de saída (<i>y(n)</i>) contém o sinal de entrada(<i>x(n)</i>) íntegro misturado com o sinal de entrada (<i>x(n)</i>) que sofrerá um atraso variante no tempo devido ao uso de um oscilador de baixa frequência LFO ($\sin 2\pi fmn$). Esse atraso variante no tempo sofrerá uma atenuação ou ganho <i>g</i>	24
Figura 4 – Representação gráfica do <i>REVERB</i>	25
Figura 5 – Diagrama de blocos do efeito flanger. O sinal de saída(Output) contém o sinal de entrada(<i>x(n)</i>) íntegro misturado com o sinal de entrada(Input) que sofrerá um atraso variante no tempo devido ao uso de um oscilador de baixa frequência LFO ($\sin 2\pi fmn$), o atraso variante no tempo é caracterizado(<i>D(n) samples delay</i>). Esse atraso variante no tempo sofrerá uma atenuação ou ganho <i>a</i>	26
Figura 6 – Representação gráfica de tipos de saturações	28
Figura 7 – Periféricos STM32F4DISCOVERY	29
Figura 8 – STM32F4DISCOVERY (Foto da placa retirada no manual do fabricante) .	30
Figura 9 – Diagrama de blocos do kit de desenvolvimento	30
Figura 10 – Diagrama de blocos do kit de desenvolvimento STM32F4DISCOVERY em conjunto com Placa <i>Wolfson Pi</i>	32
Figura 11 – Efeito <i>chorus</i> no tempo.....	39
Figura 12 – Efeito <i>chorus</i> em frequência.....	40
Figura 13 – Efeito vibrato no tempo.....	41
Figura 14 – Efeito vibrato em frequência.....	42
Figura 15 – Efeito <i>flanger</i> no tempo	43
Figura 16 – Efeito <i>flanger</i> em frequência.....	43
Figura 17 – Efeito de distorção no tempo.....	45
Figura 18 – Efeito de distorção em frequência	45

LISTA DE ABREVIATURAS E SIGLAS

AD	Analógico-Digital
DA	Digital-Analógico
DAELN	Departamento Acadêmico de Eletrônica
DSP	<i>Digital Signal Processing</i>
FIR	<i>Finite Impulse Response</i>
FS	<i>Sampling Frequency</i>
Fs	Frequência de Amostragem
IIR	<i>Infinite Impulse Response</i>
IFSC	Instituto Federal de Santa Catarina
LED	<i>Light-Emitting Diode</i>
LFO	<i>Low Frequency Oscillator</i>
N	Número de Amostras
TS	Período de Amostragem

SUMÁRIO

1	INTRODUÇÃO	13
1.1	Justificativa	14
1.2	Definição do Problema	14
1.3	Objetivo Geral	15
1.4	Objetivos Específicos.....	15
2	REFERENCIAL TEÓRICO	17
2.1	Conceitos de DSP	17
2.1.1	Amostragem	17
2.1.2	<i>Buffer</i>	17
2.1.3	Filtro <i>Finite Impulse Response</i>	19
2.1.4	Filtro <i>Infinite Impulse Response</i>	19
2.2	Efeitos Sonoros	19
2.2.1	Histórico	19
2.2.2	<i>Delay</i>	20
2.2.3	Vibrato	22
2.2.4	<i>Chorus</i>	23
2.2.5	<i>Reverb</i> (reverberação).....	24
2.2.6	<i>Flanger</i>	25
2.2.7	Distorção.....	27
2.3	Hardware utilizado	29
2.3.1	Kit de desenvolvimento STM32F4DISCOVERY	29
2.3.2	Placa <i>Wolfson Pi</i>	30
3	METODOLOGIA.....	33
3.1.	Saturação	33
3.1.1	Pseudocódigo	33
3.1.2	Explicação do pseudocódigo	34
3.2	<i>Chorus</i>	34
3.2.1	Pseudocódigo	34
3.2.2	Explicação do pseudocódigo	36
3.3	Vibrato	38
3.3.1	Pseudocódigo	38
3.3.2	Explicação	38
3.4	<i>Flanger</i>.....	38
3.4.1	Pseudocódigo	38
3.4.2	Explicação	38
4	RESULTADOS.....	38
4.1	<i>Chorus</i>	39
4.2	Vibrato	41
4.3	<i>Flanger</i>.....	43
4.4	Distorção	45
5	CONCLUSÕES E DISCUSSÕES.....	47
6	TRABALHOS FUTUROS	48
	REFERÊNCIAS	49
	APÊNDICE A – EFEITO <i>CHORUS</i> IMPLEMENTADO EM C	51

APÊNDICE B – EFEITO <i>FLANGER</i> IMPLEMENTADO EM C	61
APÊNDICE C - EFEITO VIBRATO IMPLEMENTADO EM C	71
APÊNDICE D - EFEITO DE SATURAÇÃO(DISTORÇÃO) IMPLEMENTADO EM C	81

1 INTRODUÇÃO

O universo dos efeitos de guitarra é um campo fascinante e em constante evolução, onde a criatividade musical encontra a inovação tecnológica. Desde os primórdios da música amplificada, a guitarra elétrica tem desempenhado um papel fundamental na formação do cenário musical contemporâneo. Ao longo das décadas, a busca por novas texturas sonoras e expressividade levou ao desenvolvimento de uma variedade de efeitos que moldaram a sonoridade da guitarra e influenciaram gêneros musicais inteiros. Neste contexto, é interessante revisitar os marcos históricos que pavimentaram o caminho para a era digital dos efeitos de guitarra na contemporaneidade. A década de 1940 marcou um ponto seminal na evolução musical com a eletrificação da guitarra, impulsionada pelos captadores e válvulas, que não apenas amplificaram os sons, mas também inauguraram um novo paradigma na sonoridade instrumental. A descoberta acidental da distorção, resultado de um amplificador valvulado com uma válvula defeituosa, estabeleceu um efeito sonoro distintivo que se tornaria emblemático no universo das guitarras (Mata, 2019). A empresa *Dunlop*, por sua vez, teve um papel pioneiro ao introduzir o primeiro pedal de distorção, o *Fuzz Face*, símbolo associado ao lendário Jimi Hendrix (Mata, 2019). Esse avanço permitiu aos músicos modular a intensidade da saturação dos transistores, conferindo-lhes controle sobre a distorção introduzida no sinal de entrada. No entanto, a evolução tecnológica não parou por aí. A chegada da era digital trouxe consigo um novo leque de possibilidades. Efeitos como *delay*, vibrato, *chorus* e *reverb* passaram a ser explorados com maior flexibilidade por meio do *Digital Signal Processing* (DSP) chamado de Processamento Digital de Sinal em português. Este trabalho foca nos efeitos digitais de guitarra, explorando a variedade disponível e os fundamentos teóricos, incluindo equações e processos de DSP. A implementação prática ocorreu por meio de algoritmos em linguagem C, utilizando um kit de desenvolvimento em tempo real da STMicroelectronics. Esta abordagem visa consolidar a compreensão teórica e destacar as complexidades por trás da manipulação digital do som. Além da implementação, a pesquisa inclui análise gráfica dos resultados obtidos, proporcionando uma visão visual das nuances dos efeitos digitais na sonoridade da guitarra. Este estudo busca contribuir não apenas teoricamente, mas também na oferta de *insights* práticos sobre o impacto desses

efeitos no universo sonoro da guitarra elétrica.

1.1 Justificativa

O estudo de efeitos digitais de guitarra é importante devido à sua relevância histórica, cultural, tecnológica e artística na música contemporânea. Ele proporciona uma compreensão aprofundada da evolução desses efeitos, desde a descoberta da distorção nas válvulas dos amplificadores até as capacidades avançadas de processamento digital de sinal. A pesquisa destaca a interseção entre criatividade musical e inovação tecnológica, destacando os efeitos digitais como veículos para a materialização da imaginação musical.

No contexto atual, a facilidade do acesso a efeitos digitais impulsiona a diversidade e inovação na música, permitindo que músicos de todos os níveis de habilidade expressem sua identidade sonora. Além disso, este estudo busca contribuir para a compreensão acadêmica e musical, mas também busca fornecer recursos práticos para músicos, produtores e entusiastas, capacitando-os a explorar novas fronteiras sonoras.

Quanto às vantagens dos efeitos digitais em comparação com os analógicos, a pesquisa abordará capacidades de manipulação e personalização oferecidas pelos processadores digitais, destacando sua versatilidade e eficácia na produção musical contemporânea.

1.2 Definição do Problema

O presente trabalho visa abordar um desafio central no âmbito da música e da tecnologia de áudio: a análise, implementação e avaliação de efeitos de guitarra em ambiente digital em tempo real, por meio da utilização de DSP. O problema a ser tratado reside na busca por uma compreensão abrangente dos principais efeitos de guitarra, sua complexidade técnica e criativa, bem como na investigação das capacidades e limitações do processamento em tempo real para a reprodução fiel e expressiva desses efeitos.

Este estudo confronta questões técnicas e artísticas intrincadas que emergem da crescente demanda por efeitos de guitarra altamente versáteis e

convincentes. A aplicação eficaz de efeitos de guitarra, que são fundamentais para a criação de timbres distintos e atmosferas musicais envolventes, requer um entendimento profundo das características de cada efeito, além da habilidade de implementar algoritmos de DSP capazes de reproduzir com fidelidade as nuances sonoras desejadas.

A necessidade de abordar esse problema é acentuada pelo constante avanço tecnológico no campo da música e do processamento de áudio, bem como pela busca contínua dos músicos por ferramentas criativas que expandam suas possibilidades artísticas. A implementação de efeitos de guitarra em tempo real, usando plataformas como STM32F4DISCOVERY, acrescenta uma dimensão desafiadora à pesquisa, uma vez que a latência, a capacidade de processamento e a precisão na reprodução tornam-se fatores críticos para a experiência do músico.

Portanto, o cerne do problema reside na necessidade de equilibrar a complexidade técnica do processamento digital de efeitos de guitarra com as demandas estéticas e musicais, buscando soluções que garantam a qualidade e autenticidade das sonoridades geradas. A investigação aprofundada desse problema contribuirá não somente para a compreensão do funcionamento dos efeitos de guitarra em ambiente digital, mas também para aprimorar a aplicação prática dessas tecnologias, enriquecendo a produção musical e a experiência dos músicos e ouvintes.

1.3 Objetivo Geral

O objetivo geral deste trabalho é investigar a produção e a implementação em tempo real de alguns efeitos digitais aplicados à guitarra elétrica no contexto da criação musical contemporânea. Para tanto, utiliza-se uma plataforma de DSP adaptada para processamento de áudio.

1.4 Objetivos Específicos

- Investigar a evolução histórica dos efeitos de guitarra, realizando uma análise abrangente da sua origem e evolução, desde os primórdios analógicos até a era moderna dos efeitos digitais, destacando marcos importantes e sua influência nas

práticas musicais.

- Explorar os principais efeitos digitais utilizados em guitarra, identificando, descrevendo e classificando os principais tipos de efeitos digitais utilizados em guitarras, como modulação, distorção, *delay*, *reverb*, entre outros, compreendendo suas características sonoras distintas e aplicações.
- Analisar o funcionamento dos efeitos digitais, examinando os princípios de funcionamento por trás dos efeitos digitais de guitarra, abordando conceitos de processamento de sinal, algoritmos de áudio, conversão A/D e D/A, bem como a simulação de circuitos clássicos.
- Implementar os efeitos em tempo real com o kit de DSP STM32F4DISCOVERY, em conjunto com um CODEC áudio.
- Apresentar os resultados e avaliar quantitativamente e qualitativamente os resultados da implementação dos efeitos em tempo real
- Contextualizar as limitações encontradas durante a implementação dos efeitos, relacionando-as com desafios conhecidos na área de processamento de áudio em tempo real e propondo possíveis melhorias.

2 REFERENCIAL TEÓRICO

2.1 Conceitos de DSP

2.1.1 Amostragem

A amostragem, no contexto do Processamento Digital de Sinais (DSP) e efeitos digitais de guitarra, é uma etapa crucial que envolve a conversão de um sinal analógico contínuo em sua forma digital discreta. Essa discretização é realizada por meio da coleta periódica de amostras do sinal original, a fim de criar uma representação digital que seja compatível com a manipulação e processamento pelos sistemas digitais. A frequência de amostragem, conforme Siemens (2020), é o parâmetro-chave que governa esse processo. De acordo com o teorema de *Nyquist-Shannon*, é necessário amostrar um sinal analógico com uma taxa maior do que o dobro da frequência mais alta presente no sinal para garantir a fidelidade da reconstrução digital subsequente. Essa condição assegura que a informação contida no sinal original seja mantida e preservada no domínio digital.

A relação entre a frequência de amostragem (F_s) e o período de amostragem (T_s) é expressa pela equação:

$$F_s = \frac{1}{T_s} \quad (1)$$

Essencialmente, a frequência de amostragem determina quantos pontos do sinal original serão coletados em um intervalo de tempo específico. Quanto maior a frequência de amostragem, mais detalhes do sinal original serão capturados, permitindo uma representação digital mais precisa. No contexto dos efeitos digitais de guitarra, a escolha adequada da frequência de amostragem é vital para evitar a perda de informações relevantes do sinal de áudio. Uma frequência de amostragem inadequada pode resultar em artefatos indesejados, como *aliasing*, que degradam a qualidade sonora do efeito produzido.

2.1.2 Buffer

No âmbito do processamento de dados, um elemento crucial que assume o papel de intermediário entre a aquisição e o processamento é o *buffer*.

Com base em Bhargav (2023), o *buffer* é uma ferramenta fundamental para otimizar a manipulação de dados, desempenhando um papel vital especialmente em cenários onde a memória é um recurso limitado. Um *buffer*, em sua essência, configura-se como um espaço de armazenamento temporário que age como uma ponte entre a coleta de dados e seu subsequente processamento. É uma peça-chave para gerenciar o fluxo contínuo de informações, permitindo que os dados sejam preparados e organizados antes de serem submetidos a análises ou transformações mais profundas. Essa funcionalidade se torna particularmente valiosa ao lidar com volumes de dados consideráveis ou quando há desafios relacionados à velocidade e eficiência. Os *buffers* têm um papel na compensação de atrasos relacionados à memória, como destacado pela PCMag (2023). Eles funcionam como intermediários, permitindo que o processamento avance sem interrupções significativas, mesmo quando a alocação de memória é um recurso escasso. Essa capacidade de suavizar e otimizar o fluxo de dados é especialmente vital em cenários de processamento em tempo real, onde até pequenos atrasos podem ter implicações substanciais. Um aspecto crucial do *buffer* é a sua aplicação no processamento amostra por amostra versus processamento em blocos. Esse ponto, discutido em detalhes por Woon-Seng Gan e Sen M. Kuo em "*Embedded Signal Processing with the Micro Signal Architecture*," distingue as estratégias de processamento offline e em tempo real. Processar amostra por amostra, em tempo real, exige que o *buffer* seja gerenciado de forma ágil para acomodar a entrada contínua e a saída do processamento, mantendo a latência sob controle (Gan; Kuo, 2007). Em contrapartida, o processamento em blocos envolve a acumulação de dados em um *buffer* antes que um bloco completo seja processado, otimizando a eficiência do processamento, mas potencialmente introduzindo maior latência. No desenvolvimento de *hardware* e *software*, especialmente em contextos de processamento de sinais incorporados, o uso do *buffer* é uma prática comum.

O termo "*buffer ping pong*" refere-se a uma técnica utilizada no processamento de áudio em tempo real para otimizar a transmissão e manipulação de dados sonoros. Em ambientes de processamento de áudio em tempo real, como em aplicações musicais ou em sistemas de comunicação, a latência mínima é crucial para garantir uma experiência fluida e sem interrupções. No caso do "*buffer ping ping*", são utilizados dois *buffers* alternados para processar e transmitir áudio de forma mais eficiente. O termo "*ping pong*" refere-se à alternância contínua entre esses dois

buffers: enquanto um *buffer* (por exemplo, o “ping”) está sendo preenchido com dados de entrada, o outro (o “pong”) está sendo esvaziado para reprodução ou processamento adicional. Quando todos os dados são adquiridos/processados/transmitidos, ocorre a troca dos papéis, permitindo que o processamento ocorra de forma contínua e sem interrupções, e reduzindo o risco do DSP não conseguir realizar o processamento no tempo disponível de aquisição de cada bloco (*buffer*), o seja, em tempo real.

2.1.3 Filtro *Finite Impulse Response*

Os filtros *Finite Impulse Response* (FIR) são uma classe de filtros digitais em que a resposta ao impulso é finita. A saída de um filtro desse tipo é uma combinação linear de entradas passadas e a atual. Eles são amplamente utilizados em DSP devido às suas propriedades e facilidade de implementação. No contexto de efeitos digitais de guitarra, um filtro FIR pode ser utilizado para criar efeitos como o *delay*. O *delay* é uma cópia atrasada do sinal original que é adicionada a ele, criando uma sensação de eco ou *reverberação*. O processo de aplicar um *delay* a um sinal de áudio envolve a convolução do sinal original com a resposta ao impulso do filtro.

2.1.4 Filtro *Infinite Impulse Response*

Os filtros *Infinite Impulse Response* (IIR) são uma classe de filtros digitais em que a resposta ao impulso é infinita. A saída de um filtro desse tipo é uma combinação linear de entradas passadas e presentes, bem como saídas passadas. Os filtros IIR são comuns em aplicações de processamento de sinais devido à sua capacidade de alcançar respostas em frequência mais complexas do que os filtros FIR.

2.2 Efeitos Sonoros

2.2.1 Histórico

Conforme abordado por Mata (2019), a década de 1940 marca um ponto importante na evolução musical, com a introdução dos captadores em guitarras

elétricas, culminando na eletrificação do instrumento. Este marco revolucionário não apenas possibilitou a amplificação dos sons emitidos, mas também inaugurou um novo paradigma na sonoridade instrumental, impulsionando a progressão deste campo artístico. O surgimento das válvulas aconteceu antes desse contexto e proporcionou uma transformação notável, propiciando uma amplificação mais potente e cristalina dos instrumentos em uma banda musical. Tal avanço, como observado por Mata (2019), redefiniu a paisagem musical da época ao substituir os instrumentos tradicionais dos músicos por essas novas formas eletrificadas, catalisando a evolução contínua da música. Uma interseção intrigante ocorreu durante esse período, quando um guitarrista, em uma sessão de estúdio, se viu diante de um amplificador valvulado com uma válvula defeituosa, desencadeando involuntariamente o primeiro registro conhecido de uma gravação com sinal distorcido. Essa descoberta acidental, conforme enfatizado por Mata (2019), estabeleceu um efeito sonoro distintivo que viria a desempenhar um papel seminal no universo das guitarras. Essa saturação controlada de amplificadores, alimentada pelas características intrínsecas das válvulas, gerou uma distorção sonora cativante, moldando um elemento de estilo que seria posteriormente incorporado em diversos gêneros musicais, notadamente no emergente cenário do *rock*. É relevante destacar que a empresa *Dunlop*, conforme apontado por Mata (2019), desempenhou um papel pioneiro na história dos efeitos de guitarra ao introduzir o primeiro pedal de distorção, conhecido como *Fuzz Face*, que se tornou marca do lendário guitarrista Jimi Hendrix. Esse avanço representou uma mudança significativa, uma vez que permitiu aos músicos modular a intensidade da saturação dos transistores presentes nos circuitos elétricos.

2.2.2 Delay

O efeito de *delay*, conforme discutido por França (2015), desempenha um papel fundamental no universo dos efeitos de guitarra e do DSP. Este efeito, muitas vezes empregado por guitarristas em suas performances, consiste essencialmente em reproduzir um sinal de entrada com um atraso determinado, combinando-o com o sinal original. Tal prática resulta em um sinal de saída que incorpora tanto o conteúdo original quanto sua versão atrasada, criando uma textura sonora característica. A equação característica de um *delay* simples, conforme definido por França (2015), pode ser expressa da seguinte maneira:

$$y[n] = x[n] + \alpha \cdot x[n - N] \quad (2)$$

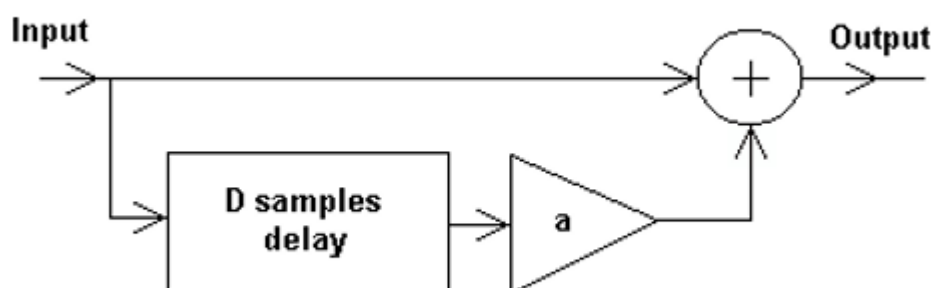
O valor de N pode ser calculado com a seguinte equação:

$$N = F_s \cdot \beta \quad (3)$$

N sendo o valor de atrasos (em amostras), frequência de amostragem (F_s) utilizado para a conversão Analógico-Digital (AD) e Digital-Analógico (DA) e tempo de *delay* (β) dado em segundos (Casado, 2018, tradução nossa).

Com isso se produz um sinal original somado com um sinal atrasado (de N amostras) multiplicado com um fator de - um ganho adicional para ser um resultado audível favorável para os ouvidos (Casado, 2018). Casado (2018) expande essa compreensão ao observar que o efeito de *delay* não se restringe apenas à guitarra, mas também pode ser aplicado a uma variedade de instrumentos musicais. Ele ressalta que o *delay* pode ser modelado utilizando um filtro FIR, um elemento-chave do processamento digital de sinais. Os filtros FIR oferecem a flexibilidade necessária para ajustar os parâmetros de atraso e atenuação de maneira precisa, permitindo a criação de efeitos de *delay* sob medida para diversas aplicações musicais. Este efeito é a base para outros efeitos como: *chorus*, vibrato e *flanger* que apresentam um *delay* variante no tempo.

Figura 1 – Diagrama de blocos do efeito *delay*. O sinal de saída contém o sinal de entrada (*input*) que será misturado com o sinal de entrada atrasado (*D samples delay*) com uma atenuação(*a*).

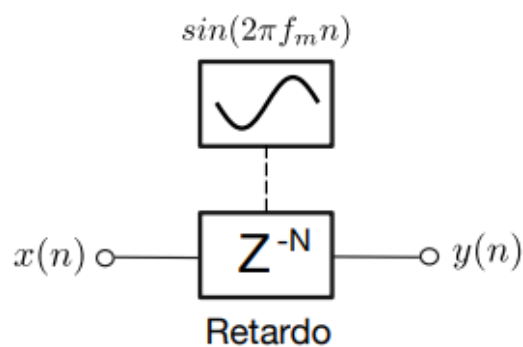


Fonte: Michalski (2023)

2.2.3 Vibrato

Dentre os efeitos digitais de guitarra que merecem destaque, encontra-se o vibrato. Conforme esclarece França (2015, p. 23), o vibrato é reconhecido por introduzir uma variação pulsante na altura da nota tocada. Em termos técnicos, o efeito vibrato é gerado por um oscilador que gera uma onda senoidal. Essa onda é então modulada a frequência da nota tocada. Quando a onda moduladora está em fase com a frequência da nota, a altura da nota é aumentada. Quando a onda moduladora está fora de fase com a frequência da nota, a altura da nota é diminuída. No entanto, no contexto das modernas possibilidades proporcionadas pela tecnologia digital, o vibrato pode ser incorporado por meio de DSP. Através da aplicação de algoritmos e cálculos, os pedais e processadores de efeitos digitais são capazes de reproduzir com fidelidade e flexibilidade o efeito de vibrato, emulando não apenas a variação de altura da nota, mas também ajustando parâmetros como velocidade e profundidade de forma altamente controlada. Assim, ao explorar as capacidades do DSP para criar e manipular o efeito de vibrato, os guitarristas podem expandir sua paleta sonora de maneira versátil e personalizada.

Figura 2– Diagrama de blocos do efeito vibrato. O sinal de saída ($y(n)$) contém o sinal de entrada ($x(n)$) que sofrerá um atraso variante no tempo devido ao uso de um oscilador de baixa frequência LFO ($\sin(2\pi f_m n)$).



Fonte: Ortuño (2019).

Equação:

$$y[n] = A \cdot x[n - \text{Amp} \cdot \sin(2\pi f_m n)] \quad (4)$$

No caso da equação acima representa o sinal de saída composto pelo sinal

de entrada que sofrerá um atraso variante no tempo devido o uso de um LFO. A onda base para esse oscilador de baixa frequência é uma senoide. Amp seria a amplitude desta onda senoidal e A o ganho ou atenuação desse efeito vibrato.

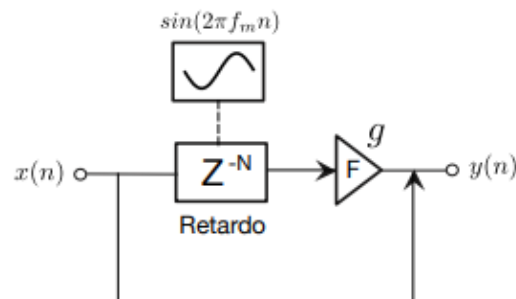
2.2.4 Chorus

O *Chorus* é um efeito de considerável proeminência, que se destaca como uma mescla sofisticada de *delay* e vibrato. Nesse contexto, o *chorus* pode ser interpretado como uma evolução do *delay*, com o vibrato sendo uma manifestação derivada do próprio *delay*. Do ponto de vista técnico e de DSP, o *chorus* pode ser entendido como uma combinação de atraso (*delay*) e modulação de frequência (vibrato) aplicados ao sinal de áudio original. A essência do *chorus* é concebida para evocar a sensação auditiva de um coro, como destacado por França (2015), onde o som se assemelha à harmonização de múltiplos instrumentos, emitindo notas de frequências quase idênticas simultaneamente. A origem histórica do *chorus* pode ser rastreada até a década de 30, quando as caixas Leslie, inicialmente incorporadas a órgãos musicais, desempenharam um papel pioneiro na incubação do conceito do efeito. Por meio de suas rotações mecânicas, as caixas Leslie foram capazes de criar uma dispersão tonal distinta, servindo como precursoras dessa característica sonora única. No entanto, o verdadeiro avanço veio na década de 70, quando a renomada empresa britânica *Roland*, conforme citado por Marshal (2021), lançou o primeiro pedal compacto de *chorus*. O sinal de entrada é copiado e atrasado por um curto período de tempo, geralmente na faixa de 20 a 50 milissegundos. Isso cria uma segunda fonte sonora que soa como uma repetição do sinal original. O oscilador de baixa frequência (LFO de *Low Frequency Oscillator*) controla o atraso aplicado ao sinal copiado. Isso faz com que o atraso varie suavemente no tempo, criando o efeito de oscilação nas frequências das notas.

$$y[n] = x[n] + g \cdot x[n - \text{sen}(2\pi f_m n)] \quad (5)$$

No caso da equação acima representa o sinal de saída composto pelo sinal de entrada misturado com a parte da equação a qual utiliza um oscilador de baixa frequência LFO modulado pela senoide(sen) com a sua respectiva frequência de oscilação(f) e atenuação(g). Essa segunda parte da equação é caracterizada pelo atrasado variante no tempo devido ao uso de um LFO.

Figura 3 - Diagrama de blocos do efeito chorus. O sinal de saída ($y(n)$) contém o sinal de entrada ($x(n)$) íntegro misturado com o sinal de entrada ($x(n)$) que sofrerá um atraso variante no tempo devido ao uso de um oscilador de baixa frequência LFO ($\sin(2\pi f_m n)$). Esse atraso variante no tempo sofrerá uma atenuação ou ganho g .



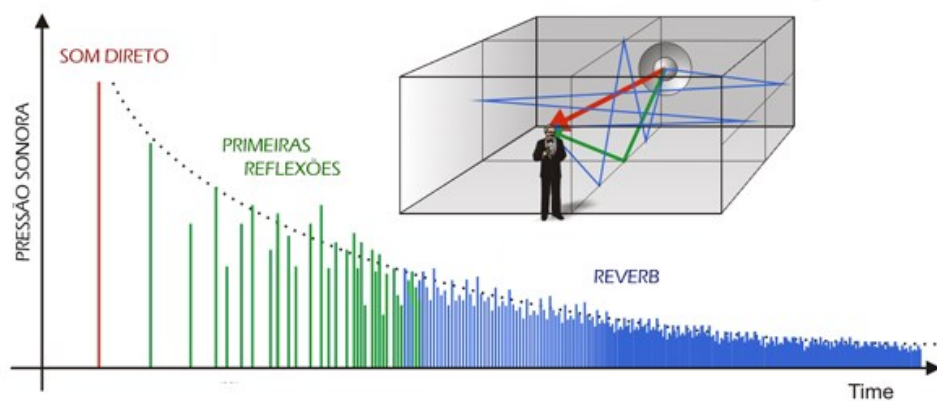
Fonte: Ortuño (2019).

2.2.5 Reverb (reverberação)

O *reverb*, um efeito instrumental inconfundível, encontra suas raízes em um fenômeno acústico poderoso: a *reverberação*. Em consonância com Casado (2018), o *reverb* é uma emulação da ambiência que ocorre quando uma onda sonora encontra superfícies e ecoa com um atraso característico, evocando a ressonância que pode ser encontrada, por exemplo, em espaços amplos. A reflexão do som ecoado pelas paredes, tetos e chão em um ambiente fechado, como uma sala, gera uma teia intrincada de ondas mecânicas que, após múltiplas reflexões, alcança os ouvidos. O *reverb* é, essencialmente, o som resultante dessas reflexões e eco, moldando uma percepção envolvente que desenha o espaço sonoro com profundidade e realismo. Na esfera da produção musical moderna, o *reverb* transcende os limites do ambiente físico, graças ao DSP. O DSP capacita os músicos e produtores a replicarem com precisão uma variedade de ambientes acústicos, desde salas de concertos majestosas até pequenas salas de ensaio intimistas, tudo dentro de um sistema microprocessado. O *reverb hall*, com seu tempo de *reverberação* prolongado, simula espaços expansivos, evocando a grandiosidade de salas vastas e imponentes. O *reverb stage room* replica ambientes mais compactos, imbuindo as gravações de um caráter acústico mais intimista e localizado. A evolução tecnológica também trouxe à tona variantes do *reverb* que moldam distintas características sonoras. O *reverb* de mola, conforme identificado por Casado (2018), busca simular as nuances do *reverb* obtido a partir das *reverberações* em molas metálicas. A equação do *reverb*, um pilar

de seu entendimento técnico, se desdobra como um conjunto de algoritmos que modelam a trajetória do som através de ambientes virtuais.

Figura 4 – Representação gráfica do *REVERB*



Fonte: Borges (2021).

A equação deste efeito é mostrada a seguir:

$$y[n] = x[n] + \text{atenuação1} \times x[n - \text{atraso1}] + \text{atenuação2} \times x[n - \text{atraso2}] + \text{atenuação3} \times x[n - \text{atraso3}] \quad (6)$$

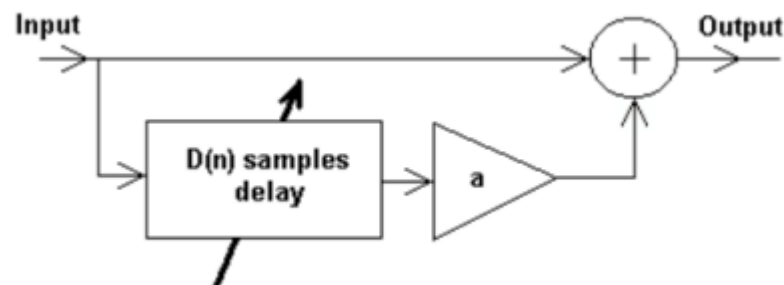
A equação acima representa de maneira simplificada a ideia do efeito. Segundo o gráfico representando o *reverb*, o efeito nada mais é que o sinal com atenuações variáveis e atrasos variáveis.

2.2.6 *Flanger*

Como ilustrado por Casado (2018), a essência do *flanger* reside na utilização de um atraso (*delay*) variável no tempo. O *flanger* opera através da combinação de duas fontes de áudio idênticas, uma das quais é ligeiramente atrasada e, subsequentemente, misturada novamente com o sinal original. Esse deslocamento de tempo, muitas vezes variável através do controle do usuário, resulta em um fenômeno conhecido como interferência construtiva e destrutiva, onde certas frequências se reforçam e outras se anulam, gerando uma variação rítmica característica. É frequentemente associado a um componente conhecido como *Low*

Frequency Oscillator (LFO). O LFO é responsável por controlar o atraso variável no tempo, que é aplicado ao sinal de áudio. Ele gera uma forma de onda de baixa frequência que modula os parâmetros do atraso, criando as flutuações características que definem o efeito *flanger*. Ao ajustar a velocidade e a amplitude do LFO, é possível moldar a intensidade e a velocidade da variação sonora, permitindo que os músicos personalizem o efeito de acordo com suas preferências artísticas. No contexto da tecnologia de DSP os dispositivos de efeitos digitais podem recriar o efeito *flanger* de maneira extremamente precisa e adaptável. Além disso, os processadores de sinal modernos frequentemente oferecem controles detalhados sobre parâmetros como taxa de variação, profundidade, forma de onda do LFO e *feedback*, permitindo uma personalização minuciosa do efeito.

Figura 5 – Diagrama de blocos do efeito flanger. O sinal de saída(Output) contém o sinal de entrada(x(n)) íntegro misturado com o sinal de entrada(Input) que sofrerá um atraso variante no tempo devido ao uso de um oscilador de baixa frequência LFO ($\sin(2\pi f_m n)$), o atraso variante no tempo é caracterizado(D(n) samples delay). Esse atraso variante no tempo sofrerá uma atenuação ou ganho a.



Fonte: Casado (2018).

Equação:

$$Output[n] = Input[n] + a \cdot Input[n - \sin(2\pi f_m n)] \quad (5)$$

A diferença entre o efeito *chorus* e o *flanger*. No *flanger*, a frequência do Oscilador de Baixa Frequência (LFO) é tipicamente menor em comparação com o *chorus*. O LFO é responsável por modular o atraso aplicado ao sinal de áudio, criando variações cíclicas. No *flanger*, essa modulação ocorre em uma taxa mais baixa, resultando em variações mais lentas e distintas.

Tamanho do *Delay*:

Uma das diferenças marcantes é o tamanho do *delay* utilizado. No *flanger*, o tamanho do *delay* é geralmente menor quando comparado ao *chorus*. O *delay* é a principal característica que cria o efeito de variação temporal entre o sinal original e suas cópias atrasadas. Com um *delay* menor no *flanger*, as variações temporais são mais rápidas e perceptíveis.

Ganho:

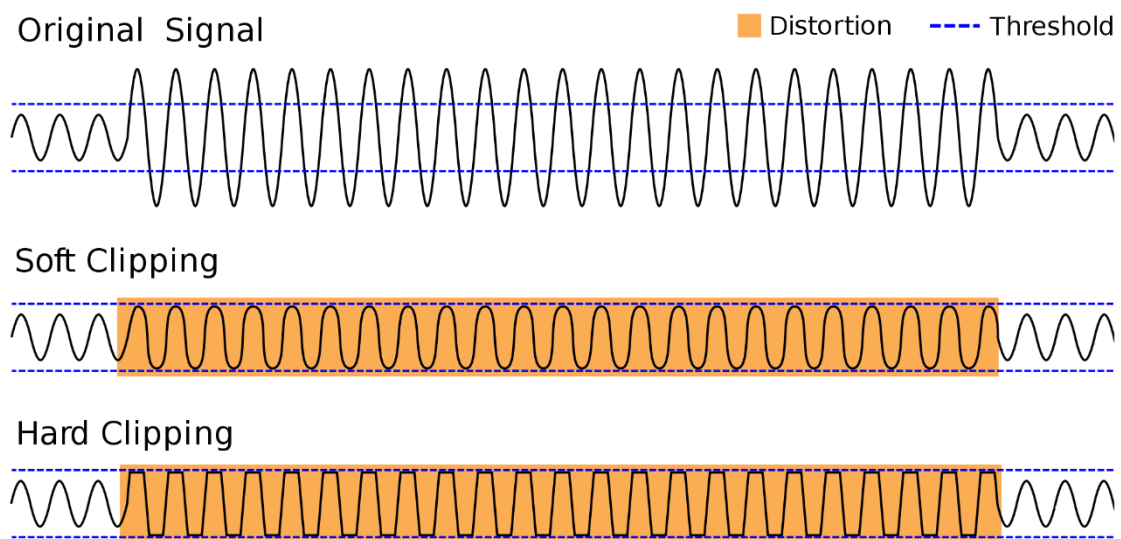
Em ambos os efeitos, *flanger* e *chorus*, o ganho é mantido constante. O ganho refere-se à amplificação do sinal resultante após o processamento. Manter o ganho inalterado contribui para garantir que a intensidade geral do sinal não seja afetada negativamente durante a aplicação desses efeitos.

2.2.7 Distorção

A descoberta da distorção, como anteriormente abordado, ocorreu de maneira fortuita, sendo essa circunstância um marco crucial em sua evolução. Inquestionavelmente, a distorção emerge como um dos efeitos mais proeminentes no universo da guitarra, por simbolizar a gênese de um estilo que redefiniu o panorama musical. Esse fenômeno se manifesta notavelmente no advento do *rock and roll*, um gênero cujo desenvolvimento foi impulsionado significativamente pela incorporação da distorção como um efeito fundamental da guitarra. Contudo, é relevante esclarecer que a guitarra, intrinsecamente, possui um timbre limpo, e a distorção constitui uma camada adicional aplicada a essa natureza primordial. No contexto histórico, os amplificadores de válvulas precederam a era dos transistores, sendo um componente crucial na origem da distorção. A compreensão abrangente da distorção demanda uma análise mais profunda desse efeito. Uma abordagem instrutiva envolve uma exploração da representação da onda sonora. O termo "distorção" tem raízes na alteração da forma de uma onda senoidal. Esse processo, conhecido como "*clipping*", envolve a limitação dos picos positivos ou negativos de uma onda, resultando na distorção da forma original. A convergência da saturação com a tecnologia de DSP abriu um novo horizonte de possibilidades para sua aplicação. Os sistemas de DSP conseguem emular fielmente os fenômenos analógicos de distorção, permitindo uma manipulação minuciosa das características sonoras. A amplitude da distorção, a

resposta de frequência e outros parâmetros podem ser ajustados de maneira controlada e reprodutível, concedendo uma maior flexibilidade aos músicos e produtores. Além disso, a era do DSP possibilitou a experimentação além dos limites do analógico. Modelagem de amplificadores, simulações de circuitos de válvulas e a criação de novas formas de distorção se tornaram possíveis através das capacidades computacionais.

Figura 6 – Representação gráfica de tipos de saturações



Fonte: Wikipédia (2023).

Conforme ilustrado no gráfico da figura 6, um exemplo de sinal sem a saturação seria uma senoide pura. Em contrapartida, uma distorção suave (*soft clipping*), delineada no segundo gráfico, é caracterizada por um ceifamento suave do sinal, resultando em uma leve alteração em sua forma de onda. O terceiro gráfico, por sua vez, retrata uma distorção mais intensa (*hard clipping*), uma terminologia frequentemente empregada pelos guitarristas para evocar a qualidade sonora. A observação atenta revela um ceifamento mais pronunciado na senoide, refletindo uma saturação mais acentuada do sinal original capturado pela guitarra. Uma forma de se modular a distorção por ceifamento é dado pela equação (8)

$$y[n] = A \cdot \tanh(B \cdot x[n]) \quad (8)$$

Este efeito como demonstrado pela equação é caracterizado por ser não linear.

2.3 Hardware utilizado

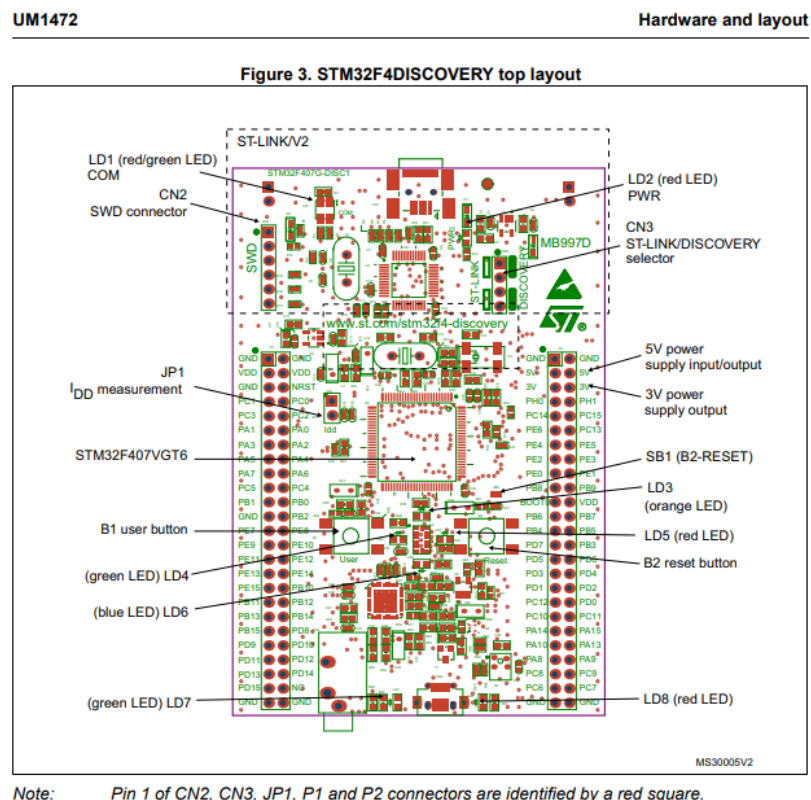
2.3.1 Kit de desenvolvimento STM32F4DISCOVERY

O Kit de desenvolvimento STM32F4DISCOVERY é uma placa de desenvolvimento baseada no microcontrolador STM32F407VGT6 da STMicroelectronics. Ele é projetado para ser uma plataforma de baixo custo e fácil de usar para o desenvolvimento de aplicações embarcadas.

O Kit STM32F4DISCOVERY inclui os seguintes componentes:

- Um microcontrolador STM32F407VGT6 com um núcleo ARM Cortex-M4 de 32 bits e 1 MB de memória flash.
- Um acelerômetro MEMS STMicroelectronics.
- Um microfone digital STMicroelectronics.
- Um conversor digital-analógico com um driver de alto-falante de classe D integrado.
- *Light-Emitting Diode* (LEDs), botões e um conector USB OTG.

Figura 7 – Periféricos STM32F4DISCOVERY



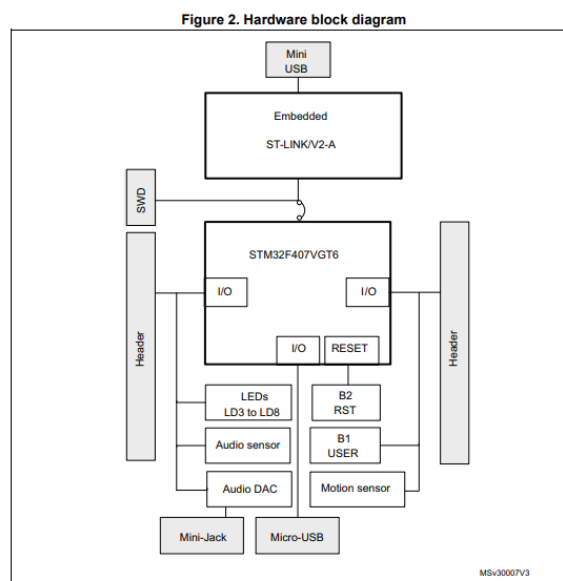
Fonte: STMicroelectronics (2020).

Figura 8 – STM32F4DISCOVERY (Foto da placa retirada no manual do fabricante)



Fonte: STMicroelectronics (2020).

Figura 9 – Diagrama de blocos do kit de desenvolvimento



Fonte: STMicroelectronics (2020).

2.3.2 Placa Wolfson Pi

A placa *Wolfson Pi* é uma placa de desenvolvimento baseada no *Raspberry Pi* que foi projetada para processamento de áudio em tempo real. Ela possui um processador ARM Cortex-A72 de 64 bits, 4 GB de RAM e um codec de áudio de alta

qualidade. A placa também possui uma interface de áudio de 3,5 mm, que pode ser usada para conectar fones de ouvido ou alto-falantes.

A *Wolfson Pi* pode ser usada para uma variedade de aplicações de processamento de áudio em tempo real, incluindo:

- Geração de som, como música ou efeitos sonoros
- Processamento de voz, como reconhecimento de fala ou sintetização de voz

- Filtragem de áudio, como redução de ruído ou equalização
- Análise de áudio, como detecção de som ou reconhecimento de padrões
- A placa é relativamente fácil de usar e vem com uma ampla gama de bibliotecas e ferramentas de *software* disponíveis.

- O processador ARM Cortex-A72 de 64 bits é rápido o suficiente para executar tarefas de processamento de áudio complexas.

- Os 4 GB de RAM fornecem espaço suficiente para armazenar dados de áudio e resultados de processamento.

- O codec de áudio de alta qualidade produz um som claro e nítido.
- A interface de áudio de 3,5 mm facilita a conexão de fones de ouvido ou alto-falantes.

- A placa *Wolfson Pi* trabalha com frequências de amostragem de 8kHz, 16kHz, 24kHz e 48kHz. O tamanho de palavra pode ser de 8 bits, 16 bits ou 24 bits.

A interconexão do kit de desenvolvimento STM32F4DISCOVERY com a placa de áudio *Wolfson Pi* foi realizada por profissionais do Departamento de Eletrônica (DAELN) do Instituto Federal de Santa Catarina (IFSC) - campus Florianópolis, para uso em disciplinas de DSP dos cursos de Engenharia Eletrônica e Superior de Tecnologia em Eletrônica Industrial.

Figura 10 – Diagrama de blocos do kit de desenvolvimento STM32F4DISCOVERY em conjunto com Placa Wolfson Pi



Fonte: acervo do autor (2023).

3 METODOLOGIA

A metodologia adotada para este estudo abrange o desenvolvimento de efeitos digitais para guitarra por meio de DSP. O processo é estruturado em etapas específicas, cada uma focada na implementação de um dos efeitos principais: distorção, *chorus*, vibrato e *flanger*.

O trabalho é iniciado com a apresentação de pseudocódigo para cada efeito, que refletem a implementação realizada em linguagem de programação C e apresentada nos anexos.

Ao detalhar cada etapa do pseudocódigo, é exposta a lógica por trás da implementação de cada efeito. Isso inclui a explicação das operações matemáticas, a manipulação de amostras de áudio e o ajuste de parâmetros específicos de cada efeito.

Essa abordagem tem como objetivo oferecer uma compreensão abrangente do desenvolvimento e implementação de efeitos digitais de guitarra, sendo acessível tanto para profissionais da área de DSP quanto para músicos e entusiastas. A frequência de amostragem utilizada foi de 48 kHz e foi utilizado operações por ponto flutuante.

3.1. Saturação

3.1.1 Pseudocódigo

```
// Para cada amostra no bloco de processamento
PARA cada k de 0 até BLOCK_SIZE - 1 FAÇA:
// Obtenha a amostra distorcida do buffer 2
distortedSample = inputBuffer2[k];
// Aplique distorção usando tangente hiperbólica (tanh)
distortedSample = tanh(distortedSample * 9);
// Armazene a amostra distorcida no buffer de saída
outputBuffer[k] = distortedSample;
FIM PARA
```

3.1.2 Explicação do pseudocódigo

PARA cada k de 0 até BLOCK_SIZE - 1 FAÇA

Aqui se inicia o laço principal que percorre cada amostra no bloco de áudio (BLOCK_SIZE), sendo k é uma variável que representa a posição da amostra neste bloco.

BLOCK_SIZE é uma constante que representa o tamanho do bloco de processamento.

distortedSample = *inputBuffer2*[k];

Se obtém a amostra atual do *buffer 2* para processamento.

Utilizar dois *buffers* (*inputBuffer1* e *inputBuffer2*) sugere um esquema de "ping- ping". A cada iteração, o código opera em uma amostra de *inputBuffer2* e armazena a amostra distorcida em *outputBuffer*. No próximo bloco de processamento, os *buffers* são alternados, ou seja, *inputBuffer2* pode se tornar *inputBuffer1*, e vice-versa. Esse método ajuda a evitar problemas de dependência de dados, permitindo leitura e gravação simultâneas.

distortedSample = tanh(distortedSample * 9);

Depois aplica a função tangente hiperbólica (tanh) à amostra do *buffer 2* multiplicada por um fator de distorção (9 neste caso). A função tanh é comumente usada para criar efeitos de distorção em áudio.

***outputBuffer*[k] = distortedSample;**

Por fim, a amostra distorcida resultante é armazenada no *buffer* de saída.

3.2 Chorus

3.2.1 Pseudocódigo

// Parâmetros para o efeito de coro profundo

CONSTANTE DELAY_LENGTH = 500 // comprimento do atraso

CONSTANTE LFO_RATE = 0.0002 // frequencia da LFO

CONSTANTE LFO_DEPTH = 0.5 // profundidade da LFO

CONSTANTE incrementoLFO = 2 * PI * LFO_RATE;

CONSTANTE faseLFO = 0

```

// Para cada amostra no bloco de áudio
PARA cada k de 0 até BLOCK_SIZE - 1 FAÇA:
    // Obtenha a amostra de entrada
    amostraEntrada = inputBuffer1[k]
    // Aplique o efeito de coro no canal ESQUERDO
    amostraAtrasada = 0
    // Calcule o atraso com base na fase da LFO
    atraso = (int)(1 + DELAY_LENGTH + LFO_DEPTH * seno(faseLFO) *
DELAY_LENGTH)
    atrasoFracionado = atraso - (1 + DELAY_LENGTH + LFO_DEPTH *
seno(faseLFO) * DELAY_LENGTH)
    // Crie a amostra atrasada interpolada
    SE (k - atraso == -1) ENTÃO:
        amostraAtrasada = inputBuffer2[k - atraso] * (1 - atrasoFracionado) +
inputBuffer1[k - atraso + 1] * atrasoFracionado
    SENÃO SE (k - atraso < 1) ENTÃO:
        amostraAtrasada = inputBuffer2[k - atraso] * (1 - atrasoFracionado) +
inputBuffer2[k - atraso + 1] * atrasoFracionado
    SENÃO:
        amostraAtrasada = inputBuffer1[k - atraso] * (1 -
atrasoFracionado) + inputBuffer1[k - atraso + 1] * atrasoFracionado
    FIM SE
    outputBuffer[k] = amostraEntrada + LFO_DEPTH *
amostraAtrasada
    // Atualize a fase da LFO
    faseLFO += incrementoLFO
    // Verifique e ajuste a fase LFO para evitar estouro
    SE (faseLFO >= 2 * PI) ENTÃO:
        faseLFO -= 2 * PI
    FIM SE
FIM PARA

```

3.2.2. Explicação do pseudocódigo

// Parâmetros para o efeito de coro profundo

constante *DELAY_LENGTH* = 500 // comprimento do atraso

constante *LFO_RATE* = 0.0002 // frequencia da LFO

constante *LFO_DEPTH* = 0.5 // profundidade da LFO

Para obter o valor do *delay* em milissegundos é necessário dividir o tamanho do *delay(DELAY_LENGTH)* pela frequência de amostragem(*Fs*). A frequência do LFO é calculada multiplicado o *LFO_RATE* pela frequência de amostragem. No caso da amplitude seria a constante *LFO_DEPTH*.

PARA cada k de 0 até *BLOCK_SIZE* - 1 FAÇA:

Este trecho é um laço que percorre cada amostra no bloco de áudio. *k* é uma variável que representa a posição da amostra no bloco.

***amostraEntrada* = *inputF32Buffer1*[*k*]**

Aqui, a amostra de entrada (*amostraEntrada*) é extraída do *buffer inputF32Buffer1* na posição *k*.

***amostraAtrasada* = 0**

A variável *amostraAtrasada* é inicializada como zero, pois será usada para acumular as amostras atrasadas.

***atraso* = (int)(1 + *DELAY_LENGTH* + *LFO_DEPTH* * seno(*faseLFO*) * *DELAY_LENGTH*)**

***atrasoFracionado* = *atraso* - (1 + *DELAY_LENGTH* + *LFO_DEPTH* * seno(*faseLFO*) * *DELAY_LENGTH*)**

O atraso é calculado com base na fase da LFO. A função seno é usada para gerar uma forma de onda que varia entre -1 e 1. Isso é multiplicado pela profundidade da LFO (*LFO_DEPTH*) e ajustado para obter um valor inteiro representando o número de amostras de atraso (*atraso*). A variável *atrasoFracionado* é a parte fracionária do atraso.

SE (*k* - *atraso* == -1) ENTÃO:

amostraAtrasada* = *inputBuffer2*[*k* - *atraso*] * (1 - *atrasoFracionado*) + *inputBuffer1*[*k* - *atraso* + 1] * *atrasoFracionado

SENÃO SE (*k* - *atraso* < 1) ENTÃO:

amostraAtrasada* = *inputBuffer2*[*k* - *atraso*] * (1 - *atrasoFracionado*) + *inputBuffer2*[*k* - *atraso* + 1] * *atrasoFracionado

SENÃO:

$\text{amostraAtrasada} = \text{inputBuffer1}[\text{k} - \text{atraso}] * (1 - \text{atrasoFracionado}) + \text{inputBuffer1}[\text{k} - \text{atraso} + 1] * \text{atrasoFracionado}$

FIM SE

$\text{outputBuffer}[\text{k}] = \text{amostraEntrada} + \text{LFO_DEPTH} * \text{amostraAtrasada}$

Esta parte do código trata da criação da amostra atrasada interpolada. Dependendo da condição ($\text{k} - \text{atraso} == -1$, $\text{k} - \text{atraso} < 1$ ou caso contrário), diferentes *buffers* são e a interpolação linear é aplicada entre duas amostras adjacentes para obter a amostra atrasada. A amostra resultante é somada à amostra de entrada multiplicada pela profundidade da LFO (LFO_DEPTH).

O uso de condições ($\text{k} - \text{atraso} == -1$), ($\text{k} - \text{atraso} < 1$), e SENÃO neste algoritmo está relacionado ao conceito de "*buffer ping pong*".

Isso envolve o uso de dois *buffers* (neste caso, *inputBuffer1* e *inputBuffer2*), alternando entre eles em cada iteração para evitar problemas de dependência de dados e permitir a execução simultânea de leitura e gravação.

A condição ($\text{k} - \text{atraso} == -1$) é um caso especial que ocorre quando a amostra atrasada está exatamente no limite entre os dois *buffers*. Neste ponto, o código interpola entre o último elemento do *buffer* anterior (*inputBuffer2*) e o primeiro elemento do *buffer* atual (*inputBuffer1*).

A condição ($\text{k} - \text{atraso} < 1$) está lidando com o caso em que o índice calculado ($\text{k} - \text{atraso}$) está fora dos limites do *buffer* atual (*inputBuffer2*). Isso ocorre quando a amostra atrasada que você está tentando acessar foi gravada no *buffer* anterior (*inputBuffer2*). Nesse caso, o código interpola entre os elementos do *buffer* anterior para obter a amostra atrasada.

A condição SENÃO lida com o caso geral em que o índice resultante está dentro do *buffer* atual (*inputBuffer1*). Nesse caso, a amostra atrasada é calculada interpolando entre os elementos do *buffer* atual.

$\text{faseLFO} += \text{incrementoLFO}$

A fase da LFO (*faseLFO*) é atualizada adicionando o incremento definido por *incrementoLFO*. Isso modula a forma de onda da LFO.

SE ($\text{faseLFO} \geq 2 * \text{PI}$) ENTÃO:

$\text{faseLFO} -= 2 * \text{PI}$

FIM SE

Essa parte verifica se a fase da LFO ultrapassou um ciclo completo ($2 * \text{PI}$)

e ajusta-a, se necessário, para evitar estouro.

3.3 Vibrato

3.3.1 Pseudocódigo

O pseudocódigo do vibrato é semelhante ao *chorus*, com a diferença que aqui só temos na saída o *buffer* de entrada com a modulação senoidal (a entrada atual não vai para a saída):

$outputBuffer[k] = LFO_DEPTH * amostraAtrasada$

3.3.2 Explicação

A explicação detalhada é a mesma ao *chorus*, com a única mudança que no vibrato não possui o *input buffer* na equação do *output*.

3.4 Flanger

3.4.1 Pseudocódigo

O algoritmo do *flanger* é também é muito semelhante ao do *chorus*, sendo que o que muda de um para o outro são alguns parâmetros iniciais e é isso que difere um efeito do outro.

// Parâmetros para o efeito de *flanger*
constante $DELAY_LENGTH = 200$ // comprimento do atraso
constante $LFO_RATE = 0.0001$ // frequencia da LFO
constante $LFO_DEPTH = 0.5$ // profundidade da LFO
CONSTANTE $incrementoLFO = 2 * PI * LFO_RATE;$
CONSTANTE $faseLFO = 0$

3.4.2 Explicação

A explicação detalhada é a mesma ao *chorus* pois o que mudou foi as constantes e por isso não será apresentado novamente.

4 RESULTADOS

Para obter os resultados gráficos dos efeitos digitais de áudio implementados, foi utilizado o seguinte procedimento:

Geração de um tom de frequência 2 kHz de acordo com a frequência de uma guitarra: Um tom de frequência de 2 kHz foi gerado usando um *software* de geração de áudio. O tom foi gerado em formato PCM com uma taxa de amostragem de 48 kHz e uma resolução de 16 bits.

Exportação do áudio: O áudio gerado foi exportado em formato WAV.

Conexão do áudio ao DSP: O áudio foi conectado à entrada do DSP usando uma interface de áudio.

Aplicação do efeito: O código em C do efeito foi executado no DSP. O efeito foi aplicado ao áudio em tempo real.

Gravação da saída: A saída do efeito foi gravada em formato WAV.

Exportação da gravação: A gravação foi exportada para um computador.

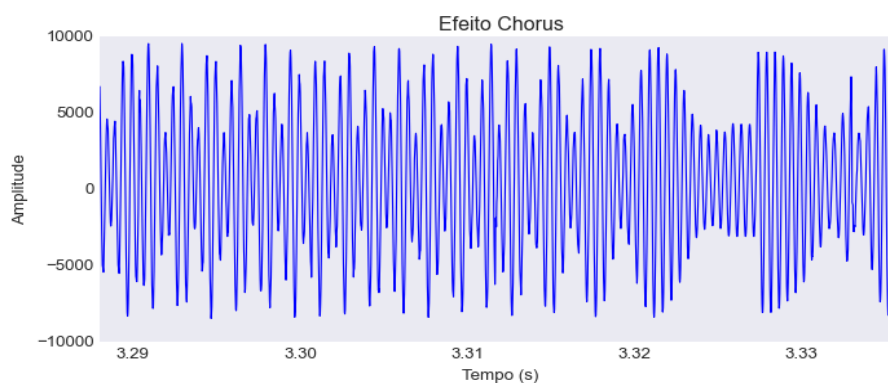
Leitura do áudio: O áudio gravado foi lido usando a linguagem de programação Python.

Geração dos gráficos: Os gráficos de espectro de frequência e domínio do tempo foram gerados usando a biblioteca de processamento de áudio da linguagem Python.

O procedimento descrito acima foi repetido para cada efeito digital de áudio implementado. Os resultados obtidos foram comparados com os resultados esperados para cada efeito.

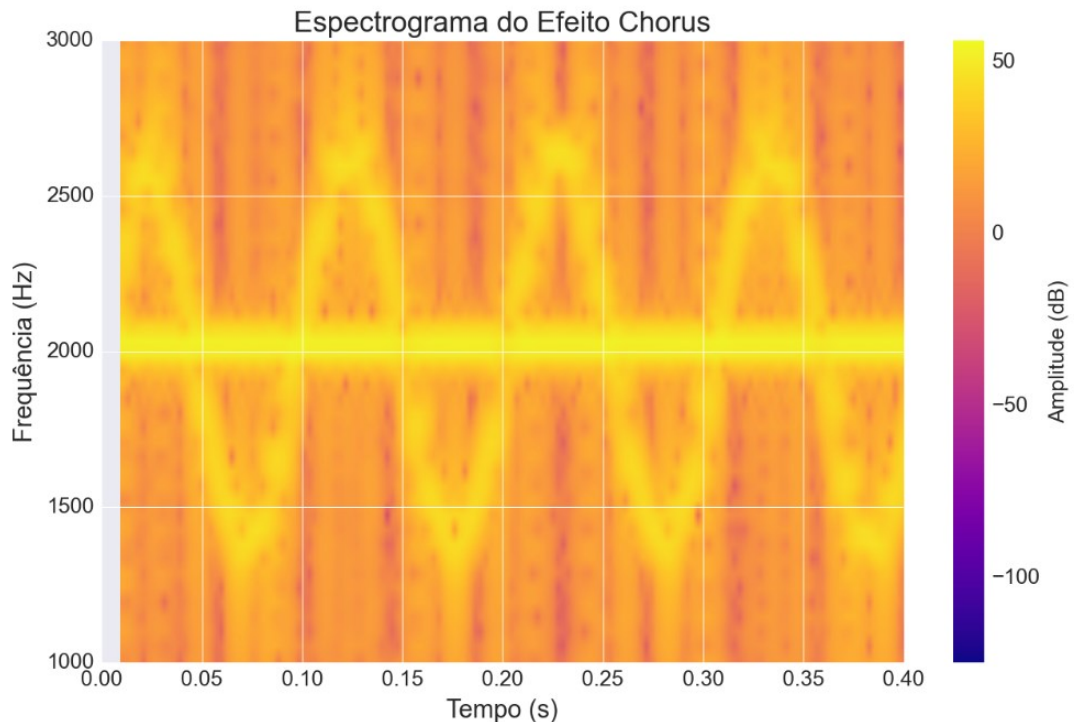
4.1 Chorus

Figura 11 – Efeito *chorus* no tempo



Fonte: elaborado pelo autor (2023).

Figura 12 – Efeito *chorus* em frequência



Fonte: elaborado pelo autor (2023).

A figura 10 mostra o sinal de saída no tempo, resultante do efeito de *chorus* quando a entrada for um sinal senoidal puro. Já a figura 11 ilustra o espectrograma desse mesmo sinal de saída. Na análise espectral, é notável a manifestação da modulação em frequência, evidenciada pela presença de componentes espectrais distintos. O espectro reflete claramente a influência da modulação em frequência, destacando uma senoide adicional na frequência correspondente ao efeito de *chorus* (especificamente o LFO deste efeito), quando comparado ao sinal de referência não afetado.

Ao examinar a saída, observa-se a soma do sinal sem efeito com a componente modulada pelo efeito *chorus*. O espectro resultante exibe uma senoide em frequência, indicando a contribuição significativa do efeito ao sinal global. É interessante notar que na ausência do efeito de *chorus*, a representação espectral exibe uma linha reta correspondente a uma frequência constante, destacando a diferença perceptível quando o efeito é aplicado. Este fenômeno ressalta a natureza dinâmica e modulatória do *chorus* na faixa de frequência considerada.

Para calcularmos a frequência do oscilador de baixa frequência LFO é necessário observar o gráfico do espectrograma de frequência apresentado na figura

11 e então fazemos o cálculo da sua frequência sabendo que:

$$f = \frac{1}{T} \quad (9)$$

A equação número 9 representa o cálculo da frequência do LFO visto que o gráfico fornece o período T em segundos da senoide. Com o seguinte cálculo observando o gráfico da figura 11 obtemos a frequência do LFO

$$f = \frac{1}{0,175 - 0,75} = 10 \text{ Hz}$$

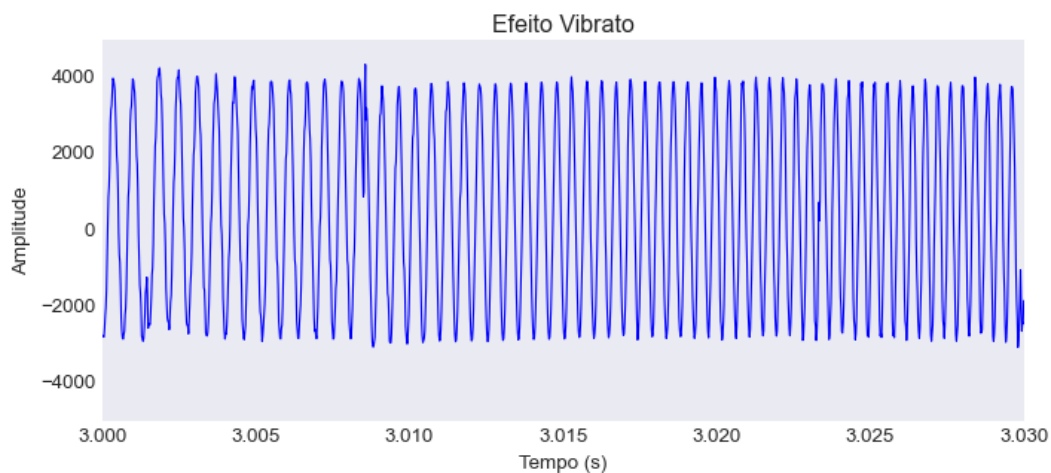
Para calcular a frequência de operação do LFO pelo algoritmo desenvolvido como já mencionado anteriormente basta utilizar a seguinte equação:

$$\begin{aligned} f_{\text{algoritmo}} &= LFO_{\text{RATE}} \cdot \text{frequência}_{\text{amostragem}} \\ f_{\text{algoritmo}} &= 0,0002.48k = 9,6 \text{ Hz} \end{aligned} \quad (10)$$

Foi obtido um valor muito próximo da frequência inicial declarada no algoritmo do efeito visto que foi definido o valor de aproximadamente 9,6 Hz e obtivemos o valor de 10 Hz

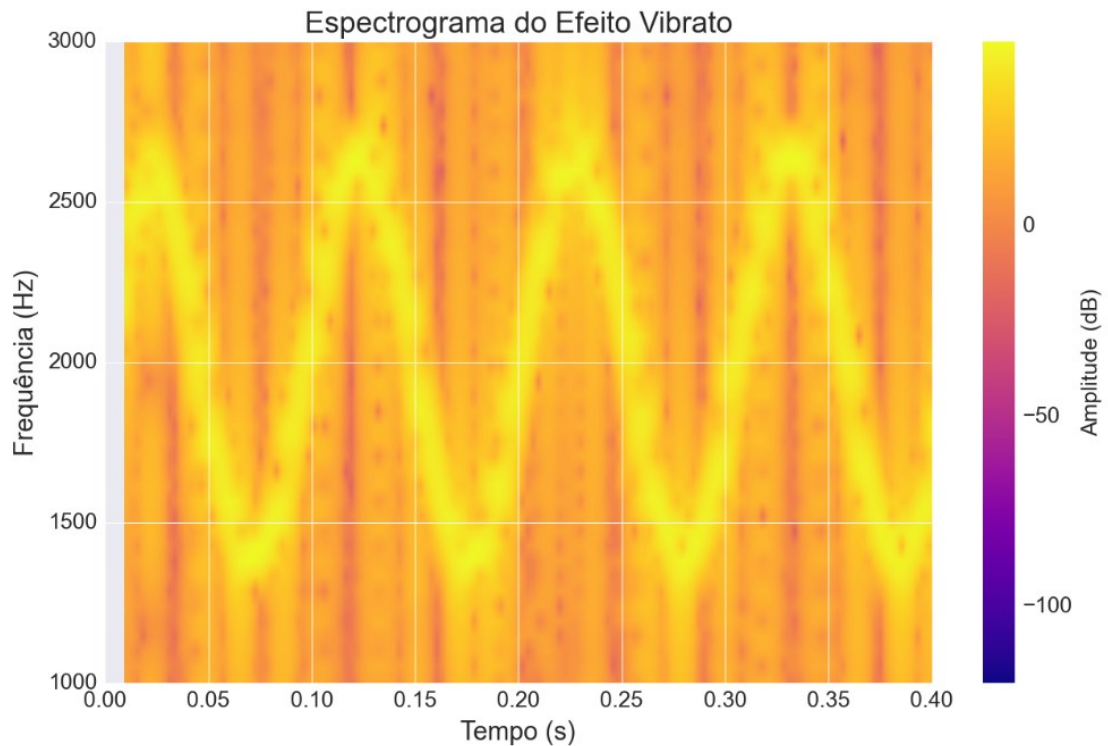
4.2 Vibrato

Figura 13 – Efeito vibrato no tempo



Fonte: elaborado pelo autor (2023).

Figura 14 – Efeito vibrato em frequência



Fonte: elaborado pelo autor (2023).

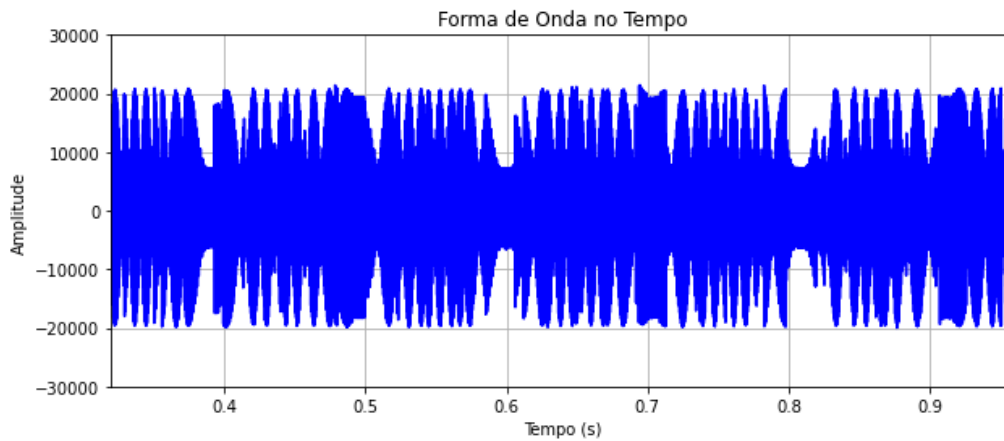
A imagem apresentada retrata o espectrograma obtido do efeito de Vibrato. Nessa representação espectral, é claramente discernível o impacto da modulação em frequência, evidenciado pelos componentes espectrais variáveis ao longo do tempo. Ao contrário do *Chorus*, a entrada original foi removida, e apenas a modulação em frequência é visível, sem a presença da linha reta constante associada à ausência de efeito.

Na saída, é possível observar exclusivamente a contribuição da modulação em frequência ao sinal, refletida no espectro resultante. A representação espectral revela uma senoide em frequência, indicativa do efeito de vibrato sem a interferência da linha reta constante associada ao sinal não modulado. Esse comportamento destaca a natureza específica da modulação em frequência proporcionada pelo vibrato, sem a presença da componente constante observada na ausência do efeito.

Para o cálculo da frequência de operação do LFO desse efeito basta seguir os mesmos passos do efeito anterior visto que os parâmetros iniciais são os mesmos

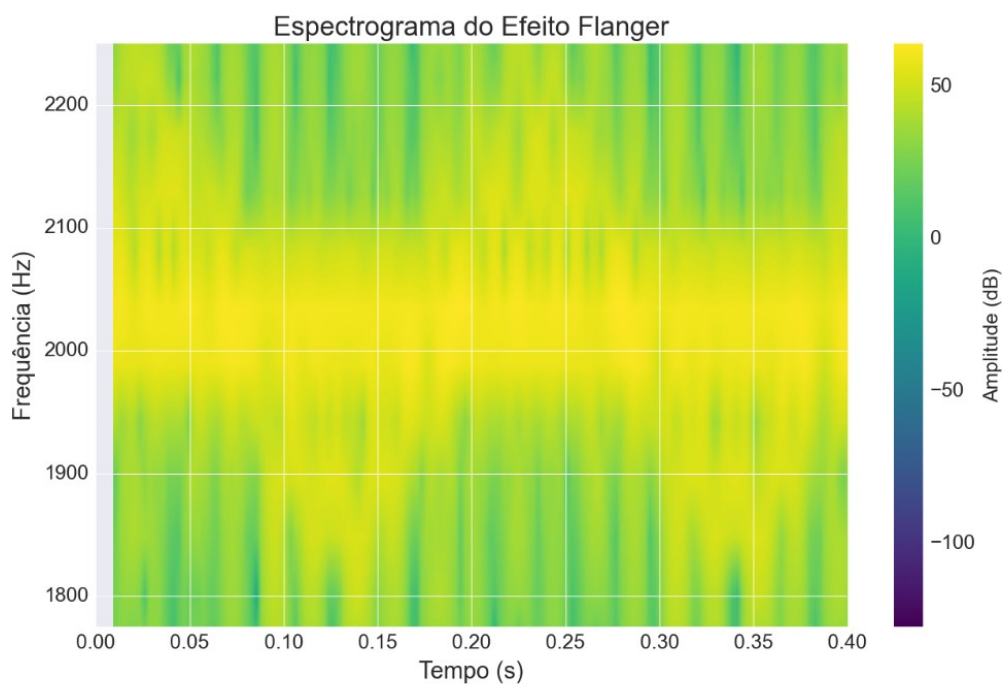
4.3 Flanger

Figura 15 – Efeito *flanger* no tempo



Fonte: elaborado pelo autor (2023).

Figura 16 – Efeito *flanger* em frequência



Fonte: elaborado pelo autor (2023).

A representação gráfica apresentada ilustra o resultado obtido pelo efeito de *flanger*. Na análise espectral, torna-se evidente a presença de modulação em frequência, manifestada por componentes espectrais distintos. O espectro reflete de maneira clara a influência da modulação em frequência, destacando uma senoide adicional na frequência correspondente ao efeito de *flanger*, quando comparado ao

signal de referência não afetado.

Ao examinar a saída, observa-se a combinação do sinal sem efeito com a componente modulada pelo efeito de *flanger*. O espectro resultante exibe uma senoide em frequência, indicando a contribuição notável do efeito ao sinal global. Vale ressaltar que, na ausência do efeito de *flanger*, a representação espectral demonstra uma linha reta correspondente a uma frequência constante, evidenciando a distinção perceptível quando o efeito é aplicado. Este fenômeno destaca a natureza dinâmica e modulatória do *flanger* na faixa de frequência considerada.

Cabe notar que, embora o algoritmo base seja similar ao do *chorus*, os parâmetros iniciais foram ajustados, resultando em uma modulação em senoide diferenciada. Essas alterações nas constantes iniciais proporcionam uma característica sonora única ao efeito de *flanger*, distinguindo-o do *chorus* e conferindo-lhe uma abordagem sônica única.

Com o seguinte cálculo observando o gráfico da figura 15 obtemos a frequência do LFO

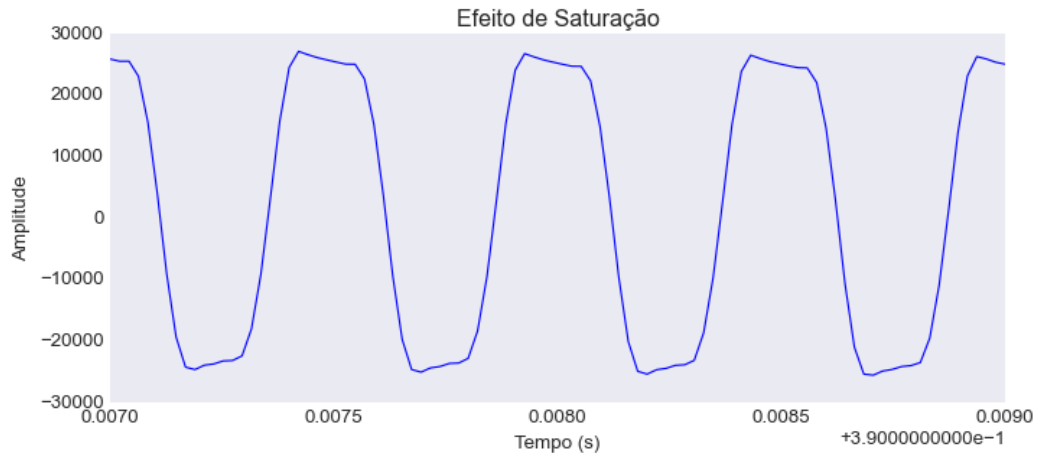
$$f = \frac{1}{0,325 - 0,125} = 5 \text{ Hz}$$

$$f_{\text{algoritmo}} = 0,0001.48k = 4,8 \text{ Hz}$$

Foi obtido um valor muito próximo da frequência inicial declarada no algoritmo do efeito visto que foi definido o valor de aproximadamente 4,8 Hz e obtivemos o valor de 5 Hz

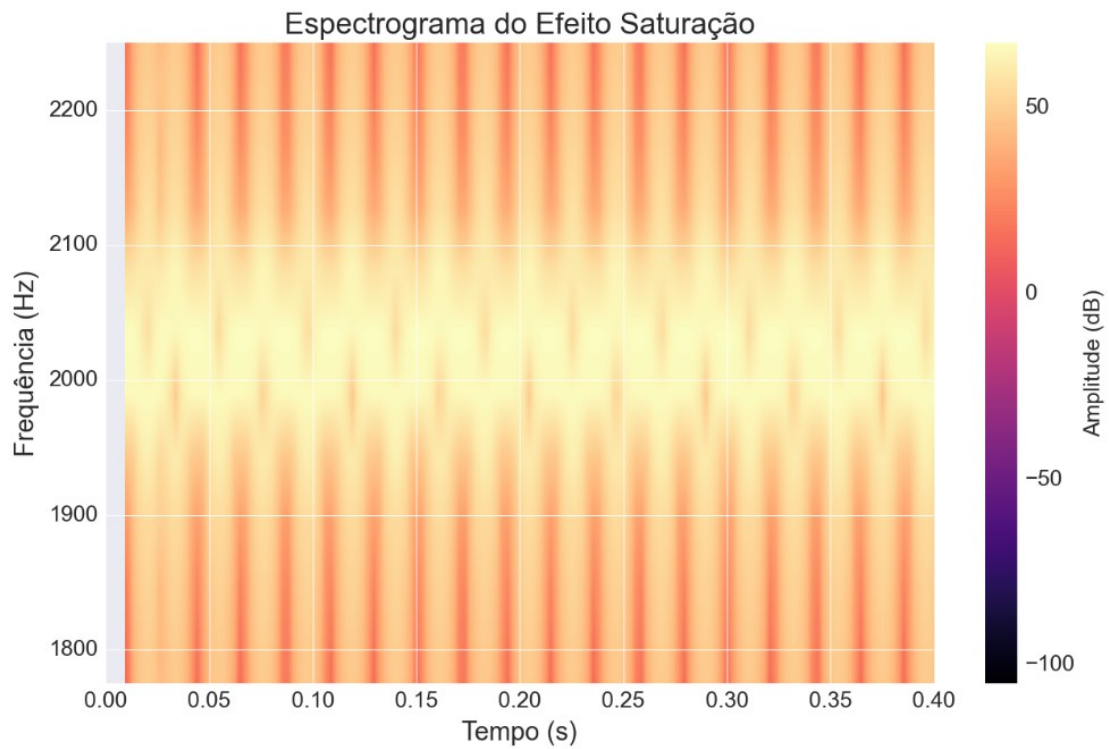
4.4 Distorção

Figura 17 – Efeito de distorção no tempo



Fonte: elaborado pelo autor (2023).

Figura 18 – Efeito de distorção em frequência



Fonte: elaborado pelo autor (2023).

A partir da imagem anterior, notamos que, no domínio temporal, a distorção foi aplicada à senoide, resultando em uma amplificação não linear da forma de onda. A senoide, que anteriormente apresentava uma amplitude uniforme, agora exibe uma

saturação notável, onde os picos foram comprimidos e enriquecidos.

Em termos de frequência, a distorção introduziu nuances harmônicas adicionais à senoide. Essas componentes harmônicas são manifestações de frequências adicionais presentes no sinal.

5 CONCLUSÕES E DISCUSSÕES

O presente trabalho teve como objetivo implementar algoritmos de efeitos digitais de áudio em C, para processamento em tempo real. Os efeitos escolhidos foram vibrato, *chorus*, *flanger* e distorção.

As dificuldades encontradas durante o desenvolvimento do trabalho foram:

A implementação dos algoritmos de cada efeito em C, por ser processamento em tempo real.

Dificuldades para a aquisição dos resultados no domínio do tempo e frequência.

Apesar das dificuldades, os objetivos propostos foram alcançados. Os algoritmos foram implementados com sucesso e os resultados obtidos foram satisfatórios.

6 TRABALHOS FUTUROS

O presente projeto pode ser aprimorado com a seguinte proposta: buscar investidores interessados no avanço de produtos já consolidados no mercado de guitarras elétricas, os quais têm uma longa presença na indústria. A ideia central é desenvolver pedais de efeitos digitais para guitarra, utilizando os conceitos apresentados anteriormente. A abordagem pode incluir a criação de pedais focados em um efeito específico, ou a exploração da concepção de pedaleiras com uma variedade de efeitos integrados para guitarra.

Ao oferecer esses produtos, a intenção é competir de maneira sólida no mercado brasileiro, proporcionando preços acessíveis aos futuros compradores. O foco principal seria assegurar um excelente custo-benefício para os consumidores. No entanto, para que esse objetivo seja alcançado, é fundamental formar uma equipe altamente especializada. A proposta envolve a criação de uma *startup* em Florianópolis especializada no desenvolvimento de efeitos digitais para guitarra elétrica.

Dessa forma, a iniciativa não apenas busca inovar no segmento, mas também se compromete a atender às demandas do mercado brasileiro, oferecendo produtos de qualidade a preços competitivos. A formação de uma equipe especializada em Florianópolis, conhecida por seu potencial inovador, contribuirá significativamente para o sucesso e crescimento da *startup* no campo de desenvolvimento de tecnologias para guitarras elétricas.

REFERÊNCIAS

- BHARGAV, Nikhil. What's a Buffer? **Baeldung**, 25 jan. 2023. Disponível: <https://www.baeldung.com/cs/buffer#:~:text=In%20computer%20science%2C%20a%20buffer,to%20better%20explain%20the%20concepts>. Acesso em: 15 nov. 2023.
- BORGES, Tiago. **Loops nos efeitos de áudio** – Cuidado com esta pegadinha. 16 dez. 2021. Disponível em: <https://tiagoborges.net/loops-nos-efeitos/>. Acesso em: 15 nov. 2023.
- CASADO, Carlos López. **Implementación de algoritmos para efectos de áudio digital**. 2018. 132f. Monografia (Ingeniería Electrónica Industrial y Automática) - Departamento Ingeniería Eléctrica, Electrónica y Física Aplica, Universidad Politécnica de Madrid, Madrid, jun. 2018.
- FRANÇA, André Wagner. **Uso de Processamento Digital de Áudio na Implementação de Efeitos em Instrumentos Musicais**. 2015. 59f. Monografia (Bacharelado em Ciência da Computação) - Instituto de Ciências Exatas, Universidade de Brasília, 2015.
- GAN, Woon-Seng; KUO, Sen M. **Embedded Signal Processing with the Micro Signal Architecture**. 1. ed. Nova Jersey: Wiley-IEEE Press, 2007.
- MATA, Alejandro Victor Rejon. **Análisis de circuitos analógicos para efectos en guitarra eléctrica**. 2019. 62f. Monografia (Física) – Departamento de Electrónica y Electromagnetismo, Facultad de Física, Universidad de Sevilla, Sevilla, 2019.
- MICHALSKI, Adam. **Sounds Effects**. Disponível em: <https://sound.eti.pg.gda.pl/student/eim/synteza/adamx/eindex.html>. Acesso em: 15 out. 2023.
- ORTUÑO, Manuel Larrosa. **Desarrollo de herramientas de usuario para la generación de efectos digitales sobre señales de audio en tiempo real**. 2019. Monografia (Ing. Sist. de Telecom., Sonido e Imagen) – Escola Politecnica Superior de Gandia, Univeritat Politècnica de València, 2019.
- PCMag. **Buffer**. 2023. Disponível em: <https://www.pcmag.com/encyclopedia/term/buffer>. Acesso em: 15 nov. 2023.
- SIEMENS. **Digital Signal Processing**: Sampling Rates, Bandwidth, Spectral Lines, and more... 10 jul. 2020. Disponível: [https://community.sw.siemens.com/s/article/digital-signal-processing-sampling-rates-bandwidth-spectral-lines-and-more#:~:text=Sampling%20rate%20\(sometimes%20called%20sampling,as%202000%20Hertz%20sample%20frequency](https://community.sw.siemens.com/s/article/digital-signal-processing-sampling-rates-bandwidth-spectral-lines-and-more#:~:text=Sampling%20rate%20(sometimes%20called%20sampling,as%202000%20Hertz%20sample%20frequency). Acesso em: 15 nov. 2023.
- STMICROELECTRONICS. **STM32F4DISCOVERY**. 2020. Disponível em: https://www.st.com/resource/en/data_brief/stm32f4discovery.pdf. Acesso em: 23 out. 2023.

WIKIPEDIA. **Distortion (music)**. 2023. Disponível em:
https://en.wikipedia.org/wiki/Distortion_%28music%29. Acesso em: 23 out. 2023.

APÊNDICE A – EFEITO *CHORUS* IMPLEMENTADO EM C

EFEITO *CHORUS* IMPLEMENTADO EM C:

```
#include <stm32f4xx.h>
#include <arm_math.h>
#include <stm32f4_discovery.h>
#include <stm32f4_discovery_accelerometer.h>
#include <wolfson_pi_audio.h>
#include <diag/Trace.h>
#include <tests.h>
#include <dwt.h>
#include "filter.h"

// #define CYCLE_COUNTER
#define NUM_TAPS 6
#define BLOCK_SIZE (WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE)/4
int16_t TxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
int16_t RxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
// Define the parameters for the deep chorus effect
#define DELAY_LENGTH 500 // Adjust the delay length as needed
#define LFO_RATE 0.0002 // Increase the LFO rate for a deeper effect (in
Hz)
#define LFO_DEPTH 0.5 // Increase the LFO depth for a deeper effect
#define SAMPLE_RATE 48000.0f // Adjust the sample rate as needed
#define PI 3.14159265358979323846f // Define PI if not already defined
// Create a delay buffer for the chorus effect
float32_t chorusDelayBuffer[DELAY_LENGTH] = {0};
uint32_t delay = 0;
uint32_t delayfrac = 0;
// Create an LFO variable for modulation
float32_t lfoPhase = 0.0f;
float32_t lfoIncrement = 2 * PI * LFO_RATE;
__IO BUFFER_StateTypeDef buffer_offset = BUFFER_OFFSET_NONE;
__IO uint8_t Volume = 70;
uint32_t AcceleroTicks;
```

```

int16_t AcceleroAxis[3];
/* -----
 * Declare State buffer of size (numTaps + blockSize - 1)
 * ----- */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1];
static q15_t firStateQ15[BLOCK_SIZE + NUM_TAPS - 1];
/* -----
 ** FIR Coefficients buffer. Moving average.
 ** ----- */
const float32_t firCoeffs32[NUM_TAPS] = {0.2, 0.2, 0.2, 0.2, 0.2, 0};
const q15_t firCoeffsQ15[NUM_TAPS] = {6553, 6553, 6553, 6553, 6553, 0}; //num
taps should be even and >4 for arm_filter_q15
/* -----
 * Global variables for FIR LPF Example
 * ----- */
FilterTypeDef filterType=FIR_FLOAT32;
int main(int argc, char* argv[])
{
    UNUSED(argc);
    UNUSED(argv);
#ifdef CYCLE_COUNTER
    uint32_t cycleCount;
#endif
    uint32_t i, k;
    float32_t inputF32Buffer1[BLOCK_SIZE];
    float32_t inputF32Buffer2[BLOCK_SIZE];
    float32_t outputF32Buffer[BLOCK_SIZE];
    q15_t inputQ15Buffer[BLOCK_SIZE];
    q15_t outputQ15Buffer[BLOCK_SIZE];
#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    FILE *CoefficientsFile;
#endif
#ifdef CYCLE_COUNTER
    FILE *CycleFile;

```

```

#endif

    float Coefficients[NUM_TAPS];
#endif

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

#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    CoefficientsFile = fopen("coefficients.txt", "r");
    if (!CoefficientsFile) {
        trace_printf("Error trying to open CoefficientsFile. Check the
name/location.\n");
        while(1);
    }
    for(i=0; i<5; i++)
        fscanf(CoefficientsFile, "%f", &Coefficients[i]);
    for(i=0; i<5; i++)
        trace_printf("Coefficient %d: %f\n", i, Coefficients[i]);
    fclose(CoefficientsFile);
#endif

#ifdef CYCLE_COUNTER
    CycleFile = fopen("cyclecounter.txt", "w");
    if (!CycleFile) {
        trace_printf("Error trying to open cycle counter file\n.");
        while(1);
    }
#endif

#endif

    WOLFSON_PI_AUDIO_Init((INPUT_DEVICE_LINE_IN << 8) |
OUTPUT_DEVICE_BOTH, 80, AUDIO_FREQUENCY_48K);
    //WOLFSON_PI_AUDIO_SetInputMode(INPUT_DEVICE_DIGITAL_MICROP
HONE);

    WOLFSON_PI_AUDIO_SetMute(AUDIO_MUTE_ON);

```

```

    WOLFSON_PI_AUDIO_Play(TxBuffer, RxBuffer,
WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE);
    WOLFSON_PI_AUDIO_SetVolume(Volume);
    BSP_ACCELERO_Init();
    TEST_Init();
    arm_fir_instance_f32 S;
    float32_t *inputF321,*inputF322, *outputF32;
    arm_fir_instance_q15 S15;
    q15_t *inputQ15, *outputQ15;
    arm_status status;
    /* Initialize input and output buffer pointers */
    inputF321 = &inputF32Buffer1[0];
    inputF322 = &inputF32Buffer2[0];
    outputF32 = &outputF32Buffer[0];
    inputQ15 = &inputQ15Buffer[0];
    outputQ15 = &outputQ15Buffer[0];
    /* Call FIR init function to initialize the instance structure. */
    arm_fir_init_f32(&S, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0],
BLOCK_SIZE);
    status=arm_fir_init_q15(&S15, NUM_TAPS, (q15_t *)&firCoeffsQ15[0],
&firStateQ15[0], BLOCK_SIZE);
    if (status==ARM_MATH_ARGUMENT_ERROR) {
        trace_printf("Problem at arm_fir_init_q15. Check if num_taps is even
and greater than 4.\n");
        while(1);
    }
    trace_printf("End of filter initialization.\n filterType is %d\n", filterType);
    while (1) {
        // Add your code here.
        if(buffer_offset == BUFFER_OFFSET_HALF)
        {
#ifdef CYCLE_COUNTER
            DWT_Reset();
            cycleCount = DWT_GetValue();

```

```

#endif

    if (filterType==FIR_FLOAT32) {
        for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
            if(i%2) {
                inputF32Buffer1[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
            else {
                TxBuffer[i] = RxBuffer[i]; // RIGHT (canal de
baixo no OcenAudio)
            }
        }
        for(k=0; k < BLOCK_SIZE; k++) {
            float32_t inputSample = inputF32Buffer1[k];
            // Apply chorus effect to the LEFT channel
            float32_t delayedSample = 0;
            delay = (int)(1+DELAY_LENGTH+LFO_DEPTH *
sinf(lfoPhase) * DELAY_LENGTH);
            delayfrac = delay -
(1+DELAY_LENGTH+LFO_DEPTH*sinf(lfoPhase)*DELAY_LENGTH);
            if(k-delay== -1){
                delayedSample = inputF32Buffer2[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
                outputF32Buffer[k] = inputF32Buffer1[k] +
LFO_DEPTH*delayedSample;
            }
            if(k-delay<1){
                delayedSample = inputF32Buffer2[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
                outputF32Buffer[k] = inputF32Buffer1[k] +
LFO_DEPTH*delayedSample;
            }
        }
    }
}

```

```

        else{
            delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
            outputF32Buffer[k] = inputF32Buffer1[k] +
LFO_DEPTH*delayedSample;
        }
        // Update the LFO phase
        lfoPhase += lfoIncrement;
        if (lfoPhase >= 2 * PI) {
            lfoPhase -= 2 * PI;
        }
    }
    for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
        if(i%2){
            TxBuffer[i] =
(int16_t)(outputF32Buffer[k]*32768); //back to 1.15
            k++;
        }
    }
}
if (filterType==FIR_Q15) {
    for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i]; //pass-through
        }
    }
}
arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
for(i=0, k=0;

```

```

i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
    if(i%2){
        TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]); //back to 1.15
        k++;
    }
}

}

}

#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nHALF: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}
if(buffer_offset == BUFFER_OFFSET_FULL)
{
#ifdef CYCLE_COUNTER
    DWT_Reset();
    cycleCount = DWT_GetValue();
#endif

    if (filterType==FIR_FLOAT32) {
        for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
            if(i%2) {
                inputF32Buffer2[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
            else {
                TxBuffer[i] = RxBuffer[i]; //pass-
through(int16_t)0.3*32768; //
            }
        }
        for(k=0; k< BLOCK_SIZE; k++) {
            float32_t inputSample = inputF32Buffer2[k];

```

```

// Apply chorus effect to the LEFT channel
float32_t delayedSample = 0;
delay = (int)(1+DELAY_LENGTH+LFO_DEPTH *
sinf(lfoPhase) * DELAY_LENGTH);
delayfrac = delay -
(1+DELAY_LENGTH+LFO_DEPTH*sinf(lfoPhase)*DELAY_LENGTH);
if(k-delay==1){
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer2[k] +
LFO_DEPTH*delayedSample;
}
if(k-delay<1){
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer2[k] +
LFO_DEPTH*delayedSample;
}
else{
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer2[k] +
LFO_DEPTH*delayedSample;
}
}
// Update the LFO phase
lfoPhase += lfoIncrement;
if (lfoPhase >= 2 * PI) {
    lfoPhase -= 2 * PI;
}
}
for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
    if(i%2){
        TxBuffer[i] =

```

```

(int16_t)(outputF32Buffer[k]*32768.0);//back to 1.15
        k++;
    }
}
}
if (filterType==FIR_Q15) {
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i];//pass-through
        }
    }
    arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2){
            TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]);//back to 1.15
            k++;
        }
    }
}
}
#if defined(OS_USE_SEMIHOSTING) && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nFULL: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}
TEST_Main();
}
#if defined(OS_USE_SEMIHOSTING) && defined(CYCLE_COUNTER)

```

```

        fclose(CycleFile);
#endif

        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 */
    while (1);
}

```

APÊNDICE B – EFEITO *FLANGER* IMPLEMENTADO EM C

EFEITO *FLANGER* IMPLEMENTADO EM C:

```
#include <stm32f4xx.h>
#include <arm_math.h>
#include <stm32f4_discovery.h>
#include <stm32f4_discovery_accelerometer.h>
#include <wolfson_pi_audio.h>
#include <diag/Trace.h>
#include <tests.h>
#include <dwt.h>
#include "filter.h"

// #define CYCLE_COUNTER
#define NUM_TAPS 6
#define BLOCK_SIZE (WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE)/4
int16_t TxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
int16_t RxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
// Define the parameters for the deep chorus effect
#define DELAY_LENGTH 200 // Adjust the delay length as needed
#define LFO_RATE 0.0001 // Increase the LFO rate for a deeper effect (in
Hz)
#define LFO_DEPTH 0.5 // Increase the LFO depth for a deeper effect
#define SAMPLE_RATE 48000.0f // Adjust the sample rate as needed
#define PI 3.14159265358979323846f // Define PI if not already defined
// Create a delay buffer for the chorus effect
float32_t chorusDelayBuffer[DELAY_LENGTH] = {0};
uint32_t delay = 0;
uint32_t delayfrac = 0;
// Create an LFO variable for modulation
float32_t lfoPhase = 0.0f;
float32_t lfoIncrement = 2 * PI * LFO_RATE;
__IO BUFFER_StateTypeDef buffer_offset = BUFFER_OFFSET_NONE;
__IO uint8_t Volume = 70;
uint32_t AcceleroTicks;
```

```

int16_t AcceleroAxis[3];
/* -----
 * Declare State buffer of size (numTaps + blockSize - 1)
 * ----- */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1];
static q15_t firStateQ15[BLOCK_SIZE + NUM_TAPS - 1];
/* -----
 ** FIR Coefficients buffer. Moving average.
 ** ----- */
const float32_t firCoeffs32[NUM_TAPS] = {0.2, 0.2, 0.2, 0.2, 0.2, 0};
const q15_t firCoeffsQ15[NUM_TAPS] = {6553, 6553, 6553, 6553, 6553, 0}; //num
taps should be even and >4 for arm_filter_q15
/* -----
 * Global variables for FIR LPF Example
 * ----- */
FilterTypeDef filterType=FIR_FLOAT32;
int main(int argc, char* argv[])
{
    UNUSED(argc);
    UNUSED(argv);
#ifdef CYCLE_COUNTER
    uint32_t cycleCount;
#endif
    uint32_t i, k;
    float32_t inputF32Buffer1[BLOCK_SIZE];
    float32_t inputF32Buffer2[BLOCK_SIZE];
    float32_t outputF32Buffer[BLOCK_SIZE];
    q15_t inputQ15Buffer[BLOCK_SIZE];
    q15_t outputQ15Buffer[BLOCK_SIZE];
#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    FILE *CoefficientsFile;
#endif
#ifdef CYCLE_COUNTER
    FILE *CycleFile;

```

```

#endif

    float Coefficients[NUM_TAPS];
#endif

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

#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    CoefficientsFile = fopen("coefficients.txt", "r");
    if (!CoefficientsFile) {
        trace_printf("Error trying to open CoefficientsFile. Check the
name/location.\n");
        while(1);
    }
    for(i=0; i<5; i++)
        fscanf(CoefficientsFile, "%f", &Coefficients[i]);
    for(i=0; i<5; i++)
        trace_printf("Coefficient %d: %f\n", i, Coefficients[i]);
    fclose(CoefficientsFile);
#endif

#ifdef CYCLE_COUNTER
    CycleFile = fopen("cyclecounter.txt", "w");
    if (!CycleFile) {
        trace_printf("Error trying to open cycle counter file\n.");
        while(1);
    }
#endif

#endif

    WOLFSON_PI_AUDIO_Init((INPUT_DEVICE_LINE_IN << 8) |
OUTPUT_DEVICE_BOTH, 80, AUDIO_FREQUENCY_48K);
    //WOLFSON_PI_AUDIO_SetInputMode(INPUT_DEVICE_DIGITAL_MICROP
HONE);

    WOLFSON_PI_AUDIO_SetMute(AUDIO_MUTE_ON);

```

```

    WOLFSON_PI_AUDIO_Play(TxBuffer, RxBuffer,
WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE);
    WOLFSON_PI_AUDIO_SetVolume(Volume);
    BSP_ACCELERO_Init();
    TEST_Init();
    arm_fir_instance_f32 S;
    float32_t *inputF321,*inputF322, *outputF32;
    arm_fir_instance_q15 S15;
    q15_t *inputQ15, *outputQ15;
    arm_status status;
    /* Initialize input and output buffer pointers */
    inputF321 = &inputF32Buffer1[0];
    inputF322 = &inputF32Buffer2[0];
    outputF32 = &outputF32Buffer[0];
    inputQ15 = &inputQ15Buffer[0];
    outputQ15 = &outputQ15Buffer[0];
    /* Call FIR init function to initialize the instance structure. */
    arm_fir_init_f32(&S, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0],
BLOCK_SIZE);
    status=arm_fir_init_q15(&S15, NUM_TAPS, (q15_t *)&firCoeffsQ15[0],
&firStateQ15[0], BLOCK_SIZE);
    if (status==ARM_MATH_ARGUMENT_ERROR) {
        trace_printf("Problem at arm_fir_init_q15. Check if num_taps is even
and greater than 4.\n");
        while(1);
    }
    trace_printf("End of filter initialization.\n filterType is %d\n", filterType);
    while (1) {
        // Add your code here.
        if(buffer_offset == BUFFER_OFFSET_HALF)
        {
#ifdef CYCLE_COUNTER
            DWT_Reset();
            cycleCount = DWT_GetValue();

```

```

#endif

    if (filterType==FIR_FLOAT32) {
        for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
            if(i%2) {
                inputF32Buffer1[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
            else {
                TxBuffer[i] = RxBuffer[i]; // RIGHT (canal de
baixo no OcenAudio)
            }
        }
        for(k=0; k < BLOCK_SIZE; k++) {
            float32_t inputSample = inputF32Buffer1[k];
            // Apply chorus effect to the LEFT channel
            float32_t delayedSample = 0;
            delay = (int)(1+DELAY_LENGTH+LFO_DEPTH *
sinf(lfoPhase) * DELAY_LENGTH);
            delayfrac = delay -
(1+DELAY_LENGTH+LFO_DEPTH*sinf(lfoPhase)*DELAY_LENGTH);
            if(k-delay== -1){
                delayedSample = inputF32Buffer2[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
                outputF32Buffer[k] = inputF32Buffer1[k] +
LFO_DEPTH*delayedSample;
            }
            if(k-delay<1){
                delayedSample = inputF32Buffer2[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
                outputF32Buffer[k] = inputF32Buffer1[k] +
LFO_DEPTH*delayedSample;
            }
        }
    }
}

```

```

else{
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer1[k] +
LFO_DEPTH*delayedSample;
}
// Update the LFO phase
lfoPhase += lfoIncrement;
if (lfoPhase >= 2 * PI) {
    lfoPhase -= 2 * PI;
}
}
for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
    if(i%2){
        TxBuffer[i] =
(int16_t)(outputF32Buffer[k]*32768); //back to 1.15
        k++;
    }
}
}
if (filterType==FIR_Q15) {
    for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i]; //pass-through
        }
    }
}
arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
for(i=0, k=0;

```

```

i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
    if(i%2){
        TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]); //back to 1.15
        k++;
    }
}

}

}

#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nHALF: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}
if(buffer_offset == BUFFER_OFFSET_FULL)
{
#ifdef CYCLE_COUNTER
    DWT_Reset();
    cycleCount = DWT_GetValue();
#endif

    if (filterType==FIR_FLOAT32) {
        for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
            if(i%2) {
                inputF32Buffer2[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
            else {
                TxBuffer[i] = RxBuffer[i]; //pass-
through(int16_t)0.3*32768; //
            }
        }
        for(k=0; k< BLOCK_SIZE; k++) {
            float32_t inputSample = inputF32Buffer2[k];

```

```

// Apply chorus effect to the LEFT channel
float32_t delayedSample = 0;
delay = (int)(1+DELAY_LENGTH+LFO_DEPTH *
sinf(lfoPhase) * DELAY_LENGTH);
delayfrac = delay -
(1+DELAY_LENGTH+LFO_DEPTH*sinf(lfoPhase)*DELAY_LENGTH);
if(k-delay==1){
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer2[k] +
LFO_DEPTH*delayedSample;
}
if(k-delay<1){
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer2[k] +
LFO_DEPTH*delayedSample;
}
else{
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
    outputF32Buffer[k] = inputF32Buffer2[k] +
LFO_DEPTH*delayedSample;
}
}
// Update the LFO phase
lfoPhase += lfoIncrement;
if (lfoPhase >= 2 * PI) {
    lfoPhase -= 2 * PI;
}
}
for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
    if(i%2){
        TxBuffer[i] =

```

```

(int16_t)(outputF32Buffer[k]*32768.0);//back to 1.15
        k++;
    }
}
}
if (filterType==FIR_Q15) {
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i];//pass-through
        }
    }
    arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2){
            TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]);//back to 1.15
            k++;
        }
    }
}
}
#if defined(OS_USE_SEMIHOSTING) && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nFULL: %lu", (DWT_GetValue()- cycleCount));
#endif
    buffer_offset = BUFFER_OFFSET_NONE;
}
TEST_Main();
}
#if defined(OS_USE_SEMIHOSTING) && defined(CYCLE_COUNTER)

```

```

        fclose(CycleFile);
#endif

        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 */
    while (1);
}

```

APÊNDICE C - EFEITO VIBRATO IMPLEMENTADO EM C

EFEITO VIBRATO IMPLEMENTADO EM C

```
#include <stm32f4xx.h>
#include <arm_math.h>
#include <stm32f4_discovery.h>
#include <stm32f4_discovery_accelerometer.h>
#include <wolfson_pi_audio.h>
#include <diag/Trace.h>
#include <tests.h>
#include <dwt.h>
#include "filter.h"

// #define CYCLE_COUNTER
#define NUM_TAPS 6
#define BLOCK_SIZE (WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE)/4
int16_t TxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
int16_t RxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
// Define the parameters for the deep chorus effect
#define DELAY_LENGTH 500 // Adjust the delay length as needed
#define LFO_RATE 0.0002 // Increase the LFO rate for a deeper effect (in
Hz)
#define LFO_DEPTH 0.5 // Increase the LFO depth for a deeper effect
#define SAMPLE_RATE 48000.0f // Adjust the sample rate as needed
#define PI 3.14159265358979323846f // Define PI if not already defined
// Create a delay buffer for the chorus effect
float32_t chorusDelayBuffer[DELAY_LENGTH] = {0};
uint32_t delay = 0;
uint32_t delayfrac = 0;
// Create an LFO variable for modulation
float32_t lfoPhase = 0.0f;
float32_t lfoIncrement = 2 * PI * LFO_RATE;
__IO BUFFER_StateTypeDef buffer_offset = BUFFER_OFFSET_NONE;
__IO uint8_t Volume = 70;
uint32_t AcceleroTicks;
```

```

int16_t AcceleroAxis[3];
/* -----
 * Declare State buffer of size (numTaps + blockSize - 1)
 * ----- */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1];
static q15_t firStateQ15[BLOCK_SIZE + NUM_TAPS - 1];
/* -----
 ** FIR Coefficients buffer. Moving average.
 ** ----- */
const float32_t firCoeffs32[NUM_TAPS] = {0.2, 0.2, 0.2, 0.2, 0.2, 0};
const q15_t firCoeffsQ15[NUM_TAPS] = {6553, 6553, 6553, 6553, 6553, 0}; //num
taps should be even and >4 for arm_filter_q15
/* -----
 * Global variables for FIR LPF Example
 * ----- */
FilterTypeDef filterType=FIR_FLOAT32;
int main(int argc, char* argv[])
{
    UNUSED(argc);
    UNUSED(argv);
#ifdef CYCLE_COUNTER
    uint32_t cycleCount;
#endif
    uint32_t i, k;
    float32_t inputF32Buffer1[BLOCK_SIZE];
    float32_t inputF32Buffer2[BLOCK_SIZE];
    float32_t outputF32Buffer[BLOCK_SIZE];
    q15_t inputQ15Buffer[BLOCK_SIZE];
    q15_t outputQ15Buffer[BLOCK_SIZE];
#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    FILE *CoefficientsFile;
#endif
#ifdef CYCLE_COUNTER
    FILE *CycleFile;

```

```

#endif

    float Coefficients[NUM_TAPS];
#endif

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

#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    CoefficientsFile = fopen("coefficients.txt", "r");
    if (!CoefficientsFile) {
        trace_printf("Error trying to open CoefficientsFile. Check the
name/location.\n");
        while(1);
    }
    for(i=0; i<5; i++)
        fscanf(CoefficientsFile, "%f", &Coefficients[i]);
    for(i=0; i<5; i++)
        trace_printf("Coefficient %d: %f\n", i, Coefficients[i]);
    fclose(CoefficientsFile);
#endif

#ifdef CYCLE_COUNTER
    CycleFile = fopen("cyclecounter.txt", "w");
    if (!CycleFile) {
        trace_printf("Error trying to open cycle counter file\n.");
        while(1);
    }
#endif

#endif

    WOLFSON_PI_AUDIO_Init((INPUT_DEVICE_LINE_IN << 8) |
OUTPUT_DEVICE_BOTH, 80, AUDIO_FREQUENCY_48K);

    //WOLFSON_PI_AUDIO_SetInputMode(INPUT_DEVICE_DIGITAL_MICROP
HONE);
    WOLFSON_PI_AUDIO_SetMute(AUDIO_MUTE_ON);

```

```

    WOLFSON_PI_AUDIO_Play(TxBuffer, RxBuffer,
WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE);
    WOLFSON_PI_AUDIO_SetVolume(Volume);
    BSP_ACCELERO_Init();
    TEST_Init();
    arm_fir_instance_f32 S;
    float32_t *inputF321,*inputF322, *outputF32;
    arm_fir_instance_q15 S15;
    q15_t *inputQ15, *outputQ15;
    arm_status status;
    /* Initialize input and output buffer pointers */
    inputF321 = &inputF32Buffer1[0];
    inputF322 = &inputF32Buffer2[0];
    outputF32 = &outputF32Buffer[0];
    inputQ15 = &inputQ15Buffer[0];
    outputQ15 = &outputQ15Buffer[0];
    /* Call FIR init function to initialize the instance structure. */
    arm_fir_init_f32(&S, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0],
BLOCK_SIZE);
    status=arm_fir_init_q15(&S15, NUM_TAPS, (q15_t *)&firCoeffsQ15[0],
&firStateQ15[0], BLOCK_SIZE);
    if (status==ARM_MATH_ARGUMENT_ERROR) {
        trace_printf("Problem at arm_fir_init_q15. Check if num_taps is even
and greater than 4.\n");
        while(1);
    }
    trace_printf("End of filter initialization.\n filterType is %d\n", filterType);
    while (1) {
        // Add your code here.
        if(buffer_offset == BUFFER_OFFSET_HALF)
        {
#ifdef CYCLE_COUNTER
            DWT_Reset();
            cycleCount = DWT_GetValue();

```

```

#endif

    if (filterType==FIR_FLOAT32) {
        for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
            if(i%2) {
                inputF32Buffer1[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
            else {
                TxBuffer[i] = RxBuffer[i]; // RIGHT (canal de
baixo no OcenAudio)
            }
        }
        for(k=0; k < BLOCK_SIZE; k++) {
            float32_t inputSample = inputF32Buffer1[k];
            // Apply chorus effect to the LEFT channel
            float32_t delayedSample = 0;
            delay = (int)(1+DELAY_LENGTH+LFO_DEPTH *
sinf(lfoPhase) * DELAY_LENGTH);
            delayfrac = delay -
(1+DELAY_LENGTH+LFO_DEPTH*sinf(lfoPhase)*DELAY_LENGTH);
            if(k-delay== -1){
                delayedSample = inputF32Buffer2[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
                outputF32Buffer[k] =
LFO_DEPTH*delayedSample;
            }
            if(k-delay<1){
                delayedSample = inputF32Buffer2[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
                outputF32Buffer[k] =
LFO_DEPTH*delayedSample;
            }
        }
    }
}

```

```

        else{
            delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
            outputF32Buffer[k] =
LFO_DEPTH*delayedSample;
        }
        // Update the LFO phase
        lfoPhase += lfoIncrement;
        if (lfoPhase >= 2 * PI) {
            lfoPhase -= 2 * PI;
        }
    }
    for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
        if(i%2){
            TxBuffer[i] =
(int16_t)(outputF32Buffer[k]*32768); //back to 1.15
            k++;
        }
    }
}
if (filterType==FIR_Q15) {
    for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i]; //pass-through
        }
    }
}
arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
for(i=0, k=0;

```

```

i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
    if(i%2){
        TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]); //back to 1.15
        k++;
    }
}

}

#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nHALF: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}

if(buffer_offset == BUFFER_OFFSET_FULL)
{
#ifdef CYCLE_COUNTER
    DWT_Reset();
    cycleCount = DWT_GetValue();
#endif

    if (filterType==FIR_FLOAT32) {
        for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
            if(i%2) {
                inputF32Buffer2[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
            else {
                TxBuffer[i] = RxBuffer[i]; //pass-
through(int16_t)0.3*32768; //
            }
        }
    }
    for(k=0; k< BLOCK_SIZE; k++) {

```

```

float32_t inputSample = inputF32Buffer2[k];
// Apply chorus effect to the LEFT channel
float32_t delayedSample = 0;
delay = (int)(1+DELAY_LENGTH+LFO_DEPTH *
sinf(lfoPhase) * DELAY_LENGTH);
delayfrac = delay -
(1+DELAY_LENGTH+LFO_DEPTH*sinf(lfoPhase)*DELAY_LENGTH);
if(k-delay==1){
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
    outputF32Buffer[k] =
LFO_DEPTH*delayedSample;
}
if(k-delay<1){
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer1[k-delay+1]*delayfrac;
    outputF32Buffer[k] =
LFO_DEPTH*delayedSample;
}
else{
    delayedSample = inputF32Buffer1[k-
delay]*(1-delayfrac)+inputF32Buffer2[k-delay+1]*delayfrac;
    outputF32Buffer[k] =
LFO_DEPTH*delayedSample;
}
// Update the LFO phase
lfoPhase += lfoIncrement;
if (lfoPhase >= 2 * PI) {
    lfoPhase -= 2 * PI;
}
}
for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
    if(i%2){

```

```

        TxBuffer[i] =
(int16_t)(outputF32Buffer[k]*32768.0); //back to 1.15
        k++;
    }
}
}
if (filterType==FIR_Q15) {
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i]; //pass-through
        }
    }
    arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2){
            TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]); //back to 1.15
            k++;
        }
    }
}
}
#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nFULL: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}
TEST_Main();
}

```

```

#if defined(OS_USE_SEMIHOSTING) && defined(CYCLE_COUNTER)
    fclose(CycleFile);
#endif

    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 */
    while (1);
}

```

APÊNDICE D - EFEITO DE SATURAÇÃO(DISTORÇÃO) IMPLEMENTADO EM C

EFEITO DE SATURAÇÃO(DISTORÇÃO) IMPLEMENTADO EM C:

```
#include <stm32f4xx.h>
#include <arm_math.h>
#include <stm32f4_discovery.h>
#include <stm32f4_discovery_accelerometer.h>
#include <wolfson_pi_audio.h>
#include <diag/Trace.h>
#include <tests.h>
#include <dwt.h>
#include "filter.h"

// #define CYCLE_COUNTER
#define NUM_TAPS 6
#define BLOCK_SIZE (WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE)/4
int16_t TxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
int16_t RxBuffer[WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE];
// Define the parameters for the deep chorus effect
#define DELAY_LENGTH 500 // Adjust the delay length as needed
#define LFO_RATE 0.0002 // Increase the LFO rate for a deeper effect (in
Hz)
#define LFO_DEPTH 0.5 // Increase the LFO depth for a deeper effect
#define SAMPLE_RATE 48000.0f // Adjust the sample rate as needed
#define PI 3.14159265358979323846f // Define PI if not already defined
// Create a delay buffer for the chorus effect
float32_t chorusDelayBuffer[DELAY_LENGTH] = {0};
uint32_t delay = 0;
uint32_t delayfrac = 0;
// Create an LFO variable for modulation
float32_t lfoPhase = 0.0f;
float32_t lfoIncrement = 2 * PI * LFO_RATE;
__IO BUFFER_StateTypeDef buffer_offset = BUFFER_OFFSET_NONE;
__IO uint8_t Volume = 70;
uint32_t AcceleroTicks;
```

```

int16_t AcceleroAxis[3];
/* -----
 * Declare State buffer of size (numTaps + blockSize - 1)
 * ----- */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1];
static q15_t firStateQ15[BLOCK_SIZE + NUM_TAPS - 1];
/* -----
 ** FIR Coefficients buffer. Moving average.
 ** ----- */
const float32_t firCoeffs32[NUM_TAPS] = {0.2, 0.2, 0.2, 0.2, 0.2, 0};
const q15_t firCoeffsQ15[NUM_TAPS] = {6553, 6553, 6553, 6553, 6553, 0}; //num
taps should be even and >4 for arm_filter_q15
/* -----
 * Global variables for FIR LPF Example
 * ----- */
FilterTypeDef filterType=FIR_FLOAT32;
int main(int argc, char* argv[])
{
    UNUSED(argc);
    UNUSED(argv);
#ifdef CYCLE_COUNTER
    uint32_t cycleCount;
#endif
    uint32_t i, k;
    float32_t inputF32Buffer1[BLOCK_SIZE];
    float32_t inputF32Buffer2[BLOCK_SIZE];
    float32_t outputF32Buffer[BLOCK_SIZE];
    q15_t inputQ15Buffer[BLOCK_SIZE];
    q15_t outputQ15Buffer[BLOCK_SIZE];
#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    FILE *CoefficientsFile;
#endif
#ifdef CYCLE_COUNTER
    FILE *CycleFile;

```

```

#endif

    float Coefficients[NUM_TAPS];
#endif

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

#ifdef OS_USE_SEMIHOSTING
    //Semihosting example
    CoefficientsFile = fopen("coefficients.txt", "r");
    if (!CoefficientsFile) {
        trace_printf("Error trying to open CoefficientsFile. Check the
name/location.\n");
        while(1);
    }
    for(i=0; i<5; i++)
        fscanf(CoefficientsFile, "%f", &Coefficients[i]);
    for(i=0; i<5; i++)
        trace_printf("Coefficient %d: %f\n", i, Coefficients[i]);
    fclose(CoefficientsFile);
#endif

#ifdef CYCLE_COUNTER
    CycleFile = fopen("cyclecounter.txt", "w");
    if (!CycleFile) {
        trace_printf("Error trying to open cycle counter file\n.");
        while(1);
    }
#endif

#endif

    WOLFSON_PI_AUDIO_Init((INPUT_DEVICE_LINE_IN << 8) |
OUTPUT_DEVICE_BOTH, 80, AUDIO_FREQUENCY_48K);
    //WOLFSON_PI_AUDIO_SetInputMode(INPUT_DEVICE_DIGITAL_MICROP
HONE);
    WOLFSON_PI_AUDIO_SetMute(AUDIO_MUTE_ON);
    WOLFSON_PI_AUDIO_Play(TxBuffer, RxBuffer,

```

```

WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE);
    WOLFSON_PI_AUDIO_SetVolume(Volume);
    BSP_ACCELERO_Init();
    TEST_Init();
    arm_fir_instance_f32 S;
    float32_t *inputF321,*inputF322,*outputF32;
    arm_fir_instance_q15 S15;
    q15_t *inputQ15,*outputQ15;
    arm_status status;
    /* Initialize input and output buffer pointers */
    inputF321 = &inputF32Buffer1[0];
    inputF322 = &inputF32Buffer2[0];
    outputF32 = &outputF32Buffer[0];
    inputQ15 = &inputQ15Buffer[0];
    outputQ15 = &outputQ15Buffer[0];
    /* Call FIR init function to initialize the instance structure. */
    arm_fir_init_f32(&S, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0],
BLOCK_SIZE);
    status=arm_fir_init_q15(&S15, NUM_TAPS, (q15_t *)&firCoeffsQ15[0],
&firStateQ15[0], BLOCK_SIZE);
    if (status==ARM_MATH_ARGUMENT_ERROR) {
        trace_printf("Problem at arm_fir_init_q15. Check if num_taps is even
and greater than 4.\n");
        while(1);
    }
    trace_printf("End of filter initialization.\n filterType is %d\n", filterType);
    while (1) {
        // Add your code here.
        if(buffer_offset == BUFFER_OFFSET_HALF)
        {
#ifdef CYCLE_COUNTER
            DWT_Reset();
            cycleCount = DWT_GetValue();
#endif

```

```

        if (filterType==FIR_FLOAT32) {
            for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
                if(i%2) {
                    inputF32Buffer1[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                    k++;
                }
                else {
                    TxBuffer[i] = RxBuffer[i]; // RIGHT (canal de
baixo no OcenAudio)
                }
            }
            for(k=0; k < BLOCK_SIZE; k++) {
                float32_t distortedSample = inputF32Buffer2[k];
                // Aplicar distorção usando tangente hiperbólica
(tanh)

                distortedSample = tanhf(distortedSample*9);
                outputF32Buffer[k] = distortedSample;
            }
            for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
                if(i%2){
                    TxBuffer[i] =
(int16_t)(outputF32Buffer[k]*32768); //back to 1.15
                    k++;
                }
            }
        }
        if (filterType==FIR_Q15) {
            for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
                if(i%2) {
                    inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);

```

```

        k++;
    }
    else {
        TxBuffer[i] = RxBuffer[i]; //pass-through
    }
}
arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);
for(i=0, k=0;
i<(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2); i++) {
    if(i%2){
        TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]); //back to 1.15
        k++;
    }
}
}

#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nHALF: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}
if(buffer_offset == BUFFER_OFFSET_FULL)
{
#ifdef CYCLE_COUNTER
    DWT_Reset();
    cycleCount = DWT_GetValue();
#endif

    if (filterType==FIR_FLOAT32) {
        for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
            if(i%2) {
                inputF32Buffer2[k] =
(float32_t)(RxBuffer[i]/32768.0); //convert to float
                k++;
            }
        }
    }
}

```

```

    }
    else {
        TxBuffer[i] = RxBuffer[i]; //pass-
through(int16_t)0.3*32768; //
    }
}
for(k=0; k< BLOCK_SIZE; k++) {
    float32_t distortedSample = inputF32Buffer2[k];
    // Aplicar distorção usando tangente hiperbólica
    (tanh)
    distortedSample = tanhf(distortedSample*9);
    outputF32Buffer[k] = distortedSample;
}
for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
    if(i%2){
        TxBuffer[i] =
(int16_t)(outputF32Buffer[k]*32768.0); //back to 1.15
        k++;
    }
}
}
if (filterType==FIR_Q15) {
    for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
        if(i%2) {
            inputQ15Buffer[k] = (q15_t)(RxBuffer[i]);
            k++;
        }
        else {
            TxBuffer[i] = RxBuffer[i]; //pass-through
        }
    }
}
arm_fir_q15(&S15, inputQ15, outputQ15, BLOCK_SIZE);

```

```

        for(i=(WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE/2),
k=0; i<WOLFSON_PI_AUDIO_TXRX_BUFFER_SIZE; i++) {
            if(i%2){
                TxBuffer[i] =
(int16_t)(outputQ15Buffer[k]); //back to 1.15
                k++;
            }
        }
    }
}

#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fprintf(CycleFile, "\nFULL: %lu", (DWT_GetValue()- cycleCount));
#endif

    buffer_offset = BUFFER_OFFSET_NONE;
}
TEST_Main();
}

#ifdef OS_USE_SEMIHOSTING && defined(CYCLE_COUNTER)
    fclose(CycleFile);
#endif
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 */
    while (1);
}
```