UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Carolina Bosquetti Westphal

EFEITOS DE ÁUDIO DIGITAIS: IMPLEMENTAÇÃO DO EFEITO WAH WAH UTILIZANDO ARDUINO E PEDALSHIELD

Florianópolis

2021

Carolina Bosquetti Westphal

EFEITOS DE ÁUDIO DIGITAIS: IMPLEMENTAÇÃO DO EFEITO WAH WAH UTILIZANDO ARDUINO E PEDALSHIELD

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Engenharia Elétrica para a obtenção do Grau de Bacharel em Engenharia Elétrica.
Orientador: Prof. Sidnei Noceti Filho, Dr.

Florianópolis

2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Westphal, Carolina Bosquetti
Efeitos de áudio digitais : Implementação do efeito Wah Wah utilizando Arduino e PedalShield / Carolina Bosquetti Westphal ; orientador, Sidnei Noceti Filho, 2021.
67 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia Elétrica, Florianópolis, 2021.

Inclui referências.

1. Engenharia Elétrica. 2. Efeitos de áudio digitais. 3. Wah Wah. 4. Arduino. 5. PedalSHIELD. I. Filho, Sidnei Noceti. II. Universidade Federal de Santa Catarina. Graduação em Engenharia Elétrica. III. Título.

Carolina Bosquetti Westphal

Efeitos de áudio digitais: Implementação do efeito Wah Wah utilizando Arduino e PedalShield

Este Trabalho Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Engenharia Elétrica" e aceito, em sua forma final, pelo Curso de Graduação em Engenharia Elétrica.

Florianópolis, 01 de julho de 2021.

Documento assinado digitalmente
Jean Vianei Leite
Data: 02/07/2021 15:49:38-0300
CPF: 003.474.909-80
Verifique as assinaturas em https://v.ufsc.br

Prof. Jean Vianei Leite, Dr.
Coordenador do Curso de Graduação em Engenharia Elétrica

Banca Examinadora:

Prof. Sidnei Noceti Filho, Dr.
Orientador
Universidade Federal de Santa Catarina

Prof. Walter Antonio Gontijo, Me.
Universidade do Vale do Itajaí

Prof. Walter Pereira Carpes Junior, Dr.
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço primeiramente ao meu noivo Rafael, pois ele me apoiou nos melhores e piores momentos de minha graduação, e sempre me incentivou a arriscar e testar meus limites. Sem o apoio dele, eu não conseguiria concluir este trabalho.

Agradeço também à minha mãe Margareth, e minha irmã Luciana, que me ajudaram financeiramente e emocionalmente para que eu pudesse chegar aonde cheguei. Me inspiro nas duas e espero um dia ser tão incrível quanto elas.

Também agradeço aos meus amigos, sejam os que já mantinham contato antes da graduação, ou sejam aqueles cujo laço foi criado no decorrer destes últimos anos, incluindo aqueles que conheci em meus tempos de trabalho; eles tornaram os dias mais alegres e fáceis de tolerar.

Deixo ainda um agradecimento aos professores e funcionários do departamento de Engenharia Elétrica, pois sempre tentam estar à disposição do aluno - em especial ao professor Sidnei, que foi muito paciente e atencioso comigo e sempre se interessou pelo meu crescimento profissional.

Por fim, mas não menos importante, gostaria de agradecer ao meu pai Valter, que já se foi antes mesmo de eu ingressar neste curso, mas me ensinou muita coisa até hoje. Penso nele todos os dias, e sei que estaria muito orgulhoso de olhar para mim e ver o quanto eu me tornei independente e me mantive forte.

RESUMO

Este trabalho trata dos procedimentos a serem realizados para a dedução, simulação e implementação de um efeito de áudio utilizando técnicas digitais. O leitor poderá replicar a implementação do exemplo realizado, que é o Wah Wah, ou implementar outro efeito de áudio desejado através dos passos descritos neste documento. No projeto, é feita a obtenção de um diagrama de blocos e equação da função do efeito de áudio, e a partir destas informações, a simulação no software MATLAB é realizada. Após a confirmação do funcionamento do efeito através do programa, é realizada a implementação do efeito utilizando o conjunto de hardware formado por uma placa Arduino DUE e um periférico PedalSHIELD. O funcionamento da implementação física é averiguado e relatado no presente documento.

Palavras-chave: Efeitos de áudio digitais. Wah Wah. Arduino. PedalSHIELD.

ABSTRACT

This work deals with the procedures to be performed for the deduction, simulation and implementation of an audio effect using digital techniques. The reader will be able to replicate the implementation of the performed example, which is the Wah Wah effect, or implement another audio effect through the steps described in this document. In the project, the diagram and equation of the audio effect function is obtained, and from this information, the simulation in MATLAB is performed. After confirming that the effect works through the program, the effect is implemented using the hardware set formed by an Arduino DUE board and a PedalSHIELD peripheral. The functioning of the physical implementation is verified and reported in this document.

Keywords: Digital audio effects. Wah Wah. Arduino. PedalSHIELD.

LISTA DE FIGURAS

Figura 1 Placa do Arduino DUE 22
Figura 2 Hardware do PedalSHIELD DUE 23
Figura 3 Etapas de funcionamento do PedalSHIELD 23
Figura 4 Operação de soma de dois sinais no domínio da amostra 26
Figura 5 Diagrama de blocos e equação do efeito *delay* 26
Figura 6 Bloco de configuração do algoritmo 31
Figura 7 Algoritmo iterativo do Wah Wah 32
Figura 8 Bloco de resultados do algoritmo 33
Figura 9 Sinal de áudio sem processamento na simulação do MATLAB 34
Figura 10 Sinal de áudio após o processamento na simulação do MATLAB 34
Figura 11 Sinal de áudio sem processamento, visualizado no Ocenaudio 35
Figura 12 Sinal de áudio após o processamento, visualizado no Ocenaudio 35
Figura 13 Definição dos limites e parâmetros ajustáveis 36
Figura 14 Declaração de constantes para facilitar leitura do código 36
Figura 15 Cálculo das constantes para os parâmetros ajustáveis .. 37
Figura 16 Setup do código do Wah Wah 39
Figura 17 Loop do código do Wah Wah 40
Figura 18 Interrupção com a equação de diferenças do Wah Wah . 41
Figura 19 Sinal de áudio sem processamento na implementação física 45
Figura 20 Sinal de áudio após o processamento na implementação física 46

SUMÁRIO

1 INTRODUÇÃO 15
1.1 OBJETIVOS 15
1.2 JUSTIFICATIVA 15
1.3 METODOLOGIA 16
2 EFEITOS DE ÁUDIO 17
2.1 TIPOS DE IMPLEMENTAÇÃO, VANTAGENS E DESVANTAGENS 17
2.2 EFEITOS EXISTENTES 18
2.3 WAH WAH 19
3 FERRAMENTAS 21
3.1 MATLAB 21
3.2 OCENAUDIO 21
3.3 ARDUINO 22
3.4 PEDALSHIELD 22
4 IMPLEMENTAÇÃO 25
4.1 DIAGRAMA DE BLOCOS E EQUAÇÃO DE IMPLEMENTAÇÃO 25
4.2 TRANSFORMADA Z 28
4.3 SIMULAÇÃO NO MATLAB 30
4.4 IMPLEMENTAÇÃO FÍSICA 35
5 RESULTADOS 45
6 CONCLUSÕES E TRABALHOS FUTUROS 47
REFERÊNCIAS 49
APÊNDICE A – Equação do Wah Wah e Transformada Z 53
APÊNDICE B – Código implementado no MATLAB ... 59
APÊNDICE C – Código implementado no Arduino 63

1 INTRODUÇÃO

Neste trabalho, a apresentação sobre efeitos de áudio, o efeito Wah Wah, e a implementação de efeitos de áudio através de técnicas digitais são realizadas. Através desta documentação, o leitor poderá utilizar fundamentos básicos para a compreensão e desenvolvimento de um efeito de áudio, e assim reproduzir o projeto.

Ao final do trabalho, será possível averiguar os resultados da implementação física do efeito de áudio Wah Wah. Tais técnicas poderão ser reproduzidas para obter outros efeitos de áudio desejados.

1.1 OBJETIVOS

Este trabalho tem como objetivo descrever os procedimentos necessários para a construção, desenvolvimento e implementação de efeitos de áudio digitais utilizando o conjunto de hardware formado pela plataforma Arduino e PedalSHIELD, e utilizar estes procedimentos para implementar o efeito Wah Wah.

A proposta deste trabalho é, resumidamente, apresentar um tutorial com instruções que o leitor pode seguir para obter um efeito de áudio implementado de maneira digital através do ambiente Arduino e PedalSHIELD, e replicar o efeito Wah Wah, inclusive.

1.2 JUSTIFICATIVA

Ferramentas como microcontroladores e dispositivos lógicos programáveis se tornaram grandes aliados na busca por soluções que possam substituir implementações analógicas. Estas ferramentas vêm evoluindo de maneira a possuir maior capacidade de processamento, e também mais recursos para suportar a implementação de efeitos de áudio da forma digital.

Na área musical, é essencial que existam avanços nas ferramentas utilizadas por músicos, visando uma maior variedade de efeitos, parâmetros de variação, construção do equipamento, e qualidade do produto final.

À medida que o mercado musical e tecnológico avançam, há a necessidade de unir os conhecimentos de efeitos de áudio, filtros e dispositivos analógicos com o uso de microcontroladores. A aplicação de

efeitos de áudio através do processamento digital de sinais torna-se muito vantajosa para o usuário por ser mais simples e barata, e assim é possível entregar um produto final simplificado e que cumpre os requisitos do consumidor.

Assim, os estudos relatados neste projeto são de grande importância para a área da tecnologia e de processamento digital de sinais, bem como para o estudo de filtros e efeitos de áudio.

1.3 METODOLOGIA

Durante o desenvolvimento do efeito de áudio, foi utilizado o software MATLAB para realizar simulações envolvendo os métodos algébricos e para a obtenção de um sistema que implementasse o efeito. O MATLAB permitiu implementar a equação de diferenças do efeito Wah Wah no domínio da amostra, que foi obtida a partir da transformada Z, além de ser possível gerar arquivos de áudio resultantes destas simulações.

Para a implementação física, foi utilizado o Arduino Due e o periférico PedalSHIELD. O Arduino DUE é um microcontrolador que possui uma IDE (*Integrated Development Environment*) própria para programá-lo, e o PedalSHIELD é um periférico dedicado para programação de efeitos de áudio, e que também permite o uso de bibliotecas prontas envolvendo efeitos de áudio, que se encontram disponíveis online.

Através do Arduino DUE, é possível programar as entradas e saídas do PedalSHIELD, e implementar o efeito de pedal desejado através do algoritmo baseado na simulação do MATLAB. O PedalSHIELD possui uma entrada e uma saída de áudio, ambas consistindo em um pino P10, possibilitando a conexão diretamente para um instrumento musical na entrada (neste caso, uma guitarra), e um amplificador de áudio na saída.

2 EFEITOS DE ÁUDIO

Um efeito de áudio pode ser definido como a manipulação de sinais de áudio. São ferramentas muito utilizadas nas áreas de música, cinema, ou ainda para hobbies.

Estes efeitos podem ser controlados por parâmetros que variam de acordo com o tipo de efeito, existindo diversas maneiras de se implementar um efeito de áudio.

A finalidade do uso destes efeitos é tornar o som reproduzido mais interessante, e também diferenciado. Para músicos, é interessante o uso de pedais de efeitos variados de áudio para compor canções, melodias, e proporcionar entretenimento.

Os efeitos de áudio podem ser implementados de forma analógica e digital. A respeito dos efeitos implementados de forma analógica, é possível utilizar circuitos contínuos ou amostrados. Com relação aos efeitos implementados de forma digital, pode-se utilizar de técnicas de processamento digital de sinais implementadas em um microcontrolador, como uma placa Arduino com periféricos adequados.

2.1 TIPOS DE IMPLEMENTAÇÃO, VANTAGENS E DESVANTAGENS

Os efeitos de áudio podem ser implementados de duas formas: analógica e digital.

Através da implementação analógica, o áudio original é processado através de um circuito contendo elementos como amplificadores operacionais, resistores, capacitores e transistores. Estes por sua vez podem compor blocos mais complexos como filtros, osciladores e equalizadores, variando parâmetros do som como ganho, fase e conteúdo espectral.

Nos efeitos de áudio analógicos, pode-se notar que os mesmos apresentam um timbre mais natural. O timbre é uma característica de diferenciação da fonte sonora - um violão e um teclado possuem timbres diferentes, por exemplo.

Por não utilizarem conversores analógico-digital, os efeitos de áudio analógicos não sofrem 'picotamento' ou perda de informação, pois não dependem de operações como amostragem para obter os dados digitalmente e aplicar o efeito desejado.

A implementação de efeitos de áudio de maneira analógica tem um alto custo de implementação - diversos componentes necessários para o efeito, por exemplo, são caros. Ainda, há uma menor flexibilidade nos parâmetros ajustáveis, mas tal desvantagem pode ser contornada às custas da modificação do hardware, que por consequência pode elevar o custo total da implementação.

Com relação aos efeitos de áudio digitais, equações matemáticas são implementadas em microcontroladores que recebem o áudio amostrado. É possível implementar um algoritmo para realizar modificações em parâmetros de um áudio, como sua amplitude, fase, frequência, e também realizar operações com outros sinais. Estas opções possibilitam ao usuário gerar uma grande variedade de efeitos com apenas um dispositivo físico.

É possível obter uma maior flexibilidade de parâmetros e efeitos utilizando o mínimo de modificações no hardware através da implementação na forma digital. Isso ocorre devido ao uso de algoritmos e equações matemáticas envolvidos na implementação, que podem ser facilmente alterados. Essa variação dos parâmetros e efeitos também permite uma diversidade maior de efeitos implementados.

A utilização de um microcontrolador como o Arduino também dispensa o uso de um hardware exclusivo para cada efeito de áudio, já que é possível implementar vários efeitos no mesmo hardware, dispensando alterações externas e reduzindo o custo de implementação.

2.2 EFEITOS EXISTENTES

Alguns efeitos de áudio já foram implementados em trabalhos anteriores e encontram-se disponíveis na internet. Ainda, a PedalSHIELD também possui alguns exemplos disponíveis em sua página na web.

A seguir, alguns efeitos já foram implementados:

- Eco e *Reverb*: Repetições do som;
- *Vibrato*: Variação da fase de um sinal;
- *Chorus* e *Flanger*: Várias vozes reproduzidas simultaneamente, adicionando atrasos curtos destas vozes, gerando interferências construtivas e destrutivas;
- Tremolo: Realiza a modulação em amplitude, resultando na variação dos níveis de volume de maneira rítmica.

A documentação referente a estes efeitos implementados anteriormente (PAVEI, 2017) não será descrita no trabalho atual, ficando a cargo do leitor o estudo da mesma. No entanto, a implementação destes efeitos também pode ser realizada seguindo a estrutura de desenvolvimento descrita neste trabalho. O objetivo do tutorial deste trabalho é permitir que o leitor possa implementar não somente o Wah Wah, mas também os efeitos mencionados anteriormente.

2.3 WAH WAH

O efeito Wah Wah é assim nomeado pela sua tentativa de imitação do som feito pela boca quando alguém tenta pronunciar seu nome. O Wah Wah pode ser obtido ao variar o seu conteúdo espectral utilizando um filtro seletor passa-faixa. Outros filtros também podem ser utilizados para obter o efeito Wah Wah, como um filtro passa-baixa com fator de qualidade Q $\gg$ 0,707, ou um equalizador *bump* ou *peaking filter*.

Os parâmetros que podem ser controlados na construção do efeito Wah Wah são:

- Fator de qualidade Q;
- Frequências mínima e máxima onde ocorrem picos de amplitude do efeito (controladas por um oscilador de baixa frequência, ou *LFO*);
- Mixer, responsável por equilibrar a intensidade do efeito aplicado ao sinal original.

Um oscilador de baixa frequência consiste em um gerador de sinais utilizado para realizar a modulação do sinal de áudio ao variar ciclicamente parâmetros de interesse do sinal - no caso do Wah Wah, deseja-se variar a frequência central de um filtro passa-faixa, de um filtro bump ou a freqüência de corte de um filtro passa-baixa de alto Q. fator este que está relacionado com a largura de banda (NOCETI FILHO, 2020).

No presente trabalho, será implementado um Auto Wah Wah na forma digital por causa do custo baixo e customização alta com o auxílio de um Arduino DUE.

Na forma analógica é comum usar de indutores e potenciômetros de custo elevado, e também de um pedal físico com parte mecânica mais complexa. O pedal físico é responsável pela ativação do efeito, sendo

necessário manuseá-lo com o pé para realizar a variação da frequência do filtro.

3 FERRAMENTAS

Nesta seção, serão apresentadas as ferramentas para o desenvolvimento e implementação do efeito de áudio na forma digital.

3.1 MATLAB

O MATLAB, ou MATrix LABoratory, é uma plataforma de programação e computação numérica amplamente utilizada para aplicações envolvendo cálculo de matrizes, processamento de sinais, entre outras opções. A ferramenta é desenvolvida e distribuída pela MATHWORKS.

Através do MATLAB, é possível criar algoritmos que resolvem problemas numéricos em um tempo menor do que em outras linguagens de programação. Além disso, o programa permite a visualização de gráficos no domínio do tempo, espectros de frequência, gráficos no domínio discreto, entre outras funcionalidades que são úteis para aplicação em engenharia e ciência.

Na implementação do efeito de áudio, o MATLAB é utilizado para desenvolver e testar o algoritmo de um efeito de áudio; um arquivo de áudio é lido pelo programa e é realizado um processamento no áudio, gerando um novo arquivo de áudio com o efeito desejado. Através da visualização dos diagramas e gráficos, pode-se comparar os resultados e observar as características do efeito de áudio obtido.

3.2 OCENAUDIO

O Ocenaudio é um editor de áudio desenvolvido para analisar e alterar arquivos de áudio que utiliza como base a biblioteca Ocen Framework. O software foi desenvolvido por um grupo de pesquisa do Laboratório de Circuitos e Processamento de Sinais (LINSE), situado na Universidade Federal de Santa Catarina.

O programa possui uma interface simples e suporta diversos formatos de arquivos de áudio, além de permitir a análise espectral dos arquivos, gravação e criação de sinais de áudio. Estas duas últimas funcionalidades são de grande importância para verificar a funcionalidade do efeito implementado.

3.3 ARDUINO

O Arduino é uma plataforma de prototipagem e integração entre software e hardware livres, ou *free software/hardware*. Os seus hardwares são placas compostas por um microcontrolador, entradas e saídas programáveis (sendo essas digitais ou analógicas), e fazem uso de interface serial ou USB para realizarem a comunicação com seu ambiente de desenvolvimento integrado, ou *Integrated Development Environment* (*IDE*).

O software referente ao Arduino, ou seja, o Arduino IDE, é uma aplicação que utiliza linguagem C/C++ para programar as placas Arduino.

A Figura 1 mostra a placa do Arduino DUE, que é utilizada neste trabalho. Por ser uma ferramenta simples e com uma biblioteca diversificada de exemplos, é utilizada para o desenvolvimento de projetos simples e avançados.

Figura 1 – Placa do Arduino DUE

Em conjunto com o periférico PedalSHIELD DUE, o ambiente integrado do Arduino foi utilizado para desenvolver o software referente ao pedal de guitarra com o efeito de áudio desejado.

3.4 PEDALSHIELD

O PedalSHIELD é um periférico open source e open hardware que opera em conjunto com o Arduino para realizar processamento digital de sinais e aplicação de efeitos de áudio.

Neste trabalho, foi utilizado o PedalShield DUE, que funciona somente com a placa Arduino DUE. A Figura 2 ilustra os principais componentes do PedalSHIELD DUE:

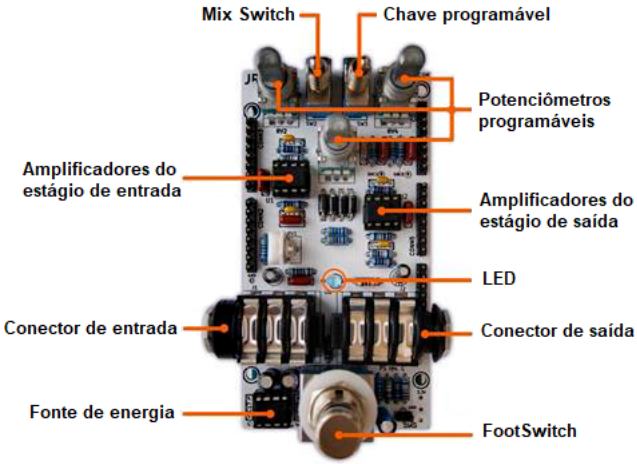

Figura 2 – Hardware do PedalSHIELD DUE

O PedalSHIELD foi desenvolvido visando uma plataforma compatível com todas as bibliotecas do Arduino, de baixo custo e design compacto. O funcionamento do PedalSHIELD consiste em três etapas: estágio de entrada, interface com o Arduino DUE, e estágio de saída. Estas etapas são ilustradas na Figura 3.

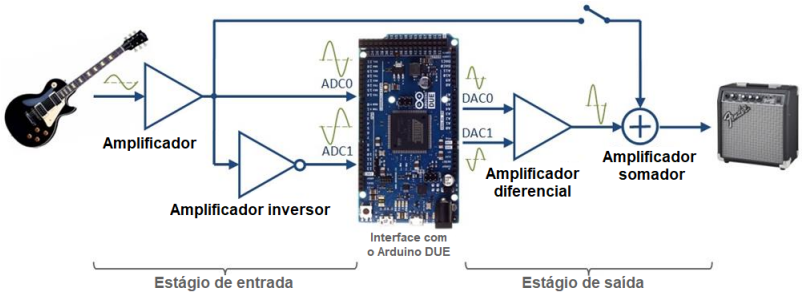

Figura 3 – Etapas de funcionamento do PedalSHIELD

- **Estágio de entrada:** Também chamado de pré-amplificação, esta etapa amplifica o sinal de entrada que vem do instrumento (por exemplo, uma guitarra), e envia o sinal amplificado ao microcontrolador do Arduino DUE, para que seja processado.

- **Interface com o Arduino DUE:** Esta etapa consiste no processamento digital do sinal, ou seja, a implementação de algoritmos que realizam o processamento e a modificação do sinal recebido, adicionando os efeitos desejados ao áudio.

- **Estágio de saída:** Após o processamento através da interface com o Arduino DUE, o sinal é enviado dos conversores digital-analógico do Arduino para a saída, e consequentemente para o amplificador do instrumento.

No estágio de saída, há um amplificador somador, que auxilia na implementação de efeitos como delay ou chorus. Este amplificador é ativado ou desativado com o manuseio do Mix Switch, somando o sinal original e o sinal processado, e enviando o resultado para a saída.

O amplificador diferencial permite que dois conversores digital-analógico sejam lidos paralelamente para aumentar a resolução dos bits, mas também pode funcionar somente como buffer.

O periférico possui um total de três potenciômetros programáveis, duas chaves (uma delas sendo o Mix Switch), um LED e o FootSwitch. O Mix Switch soma o sinal não processado ao sinal processado na saída, e o FootSwitch ativa ou desativa o efeito de áudio na saída. Ambas as funcionalidades não podem ser alteradas via software.

Como o conjunto do PedalSHIELD com o Arduino utiliza programação C/C++, apenas um conhecimento básico desta linguagem de programação é necessário, e o leitor pode usufruir das aplicações do PedalSHIELD instalando a plataforma de programação gratuita do próprio Arduino. Ainda, é fácil encontrar auxílio no uso da linguagem e implementação de algoritmos na internet. Adicionalmente, o fórum da ElectroSmash contém códigos com exemplos de efeitos de áudio caso o leitor deseje se aprofundar no uso do PedalShield.

4 IMPLEMENTAÇÃO

Para fins didáticos, foi criado um tutorial para a implementação de efeitos de áudio na forma digital. Desta forma, este documento descreve os passos a serem seguidos para construir, simular e implementar efeitos de áudio de maneira digital, e também reproduzir o efeito que foi implementado com base nesta documentação - o Wah Wah.

Assim como qualquer tutorial ou método de implementação, estes passos podem não se aplicar a qualquer efeito. Efeitos de áudio não-lineares, por exemplo, são efeitos cuja saída não depende exclusivamente da entrada, e portanto podem ser mais complexos para realizar a implementação. Ainda, podem exigir um embasamento teórico mais avançado, e portanto mais tempo seria necessário para seu estudo e implementação.

Ainda, no momento da implementação, as limitações dos softwares e hardwares envolvidos (ou seja, as ferramentas) devem ser levadas em consideração, além das vantagens e desvantagens já mencionadas.

4.1 DIAGRAMA DE BLOCOS E EQUAÇÃO DE IMPLEMENTAÇÃO

O primeiro levantamento a ser feito para realizar o projeto de um efeito de áudio é a análise do diagrama de blocos que compõem o efeito desejado, pois desta forma é possível realizar o levantamento das variáveis que serão controladas na simulação e, futuramente, na implementação física.

O diagrama de blocos é um fluxograma composto de operações e sinais, e que segue passos para implementar algum procedimento. A relação entre a saída e a entrada deste diagrama de blocos é chamada de função de transferência, e também pode ser diretamente utilizada pelo usuário caso este já possua conhecimento desta função.

Muitos materiais bibliográficos possuem diagramas disponíveis para os efeitos de áudio mais comumente utilizados; de fato, os diagramas de blocos são utilizados em sistemas lineares, processamento digital de sinais, comunicações, entre outras aplicações, para representar a relação entre entrada e saída, e o processamento envolvendo estes dois sinais.

Sendo assim, o usuário pode utilizar de ferramentas como MATLAB e Arduino para implementar um processamento de uma entrada

através de um sistema composto por um diagrama de blocos, e obter uma saída. Para obter uma função de transferência a partir do diagrama de blocos, o usuário deve aplicar a álgebra de blocos necessária para a obtenção de uma saída em função da entrada. Estas técnicas são estudadas em seções curriculares envolvendo, primariamente, sistemas lineares, seja no domínio do tempo ou no domínio da amostra.

Na Figura 4, há um exemplo simples de um bloco realizando a soma de dois sinais de entrada, e a saída resultante é igual aos dois sinais de entrada somados; tudo isso realizado no domínio da amostra, que será utilizado para os próximos passos.

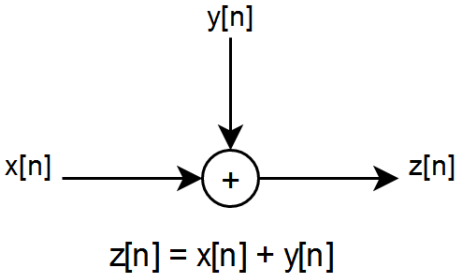

Figura 4 – Operação de soma de dois sinais no domínio da amostra

Outro exemplo já implementado em trabalhos anteriores é o efeito *delay*, que é composto pela soma de um sinal de entrada e da sua entrada com um atraso e um ganho especificados (PAVEI, 2017). O diagrama de blocos e sua equação podem ser visualizados na Figura 5.

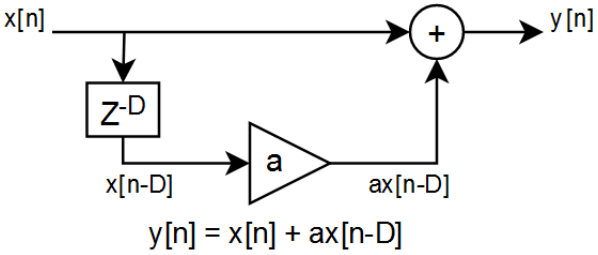

Figura 5 – Diagrama de blocos e equação do efeito *delay*

Como a equação encontrada já está no domínio da amostra, o usuário pode implementá-la sem a necessidade de aplicar outros procedimentos de transformação. Para realizar esta simulação no MATLAB, é necessário que o usuário tenha conhecimento básico da linguagem e lógica de programação, que serão usados para implementar processos iterativos e para o manuseio de dados no código.

Adicionalmente, é possível utilizar uma função de transferência já obtida anteriormente, sem a necessidade de analisar um diagrama de blocos. Caso o efeito a ser implementado seja mais complexo, é possível que o procedimento de obtenção da equação através do diagrama de blocos não seja o suficiente, então o usuário pode optar por funções que não necessariamente se encontram no domínio da amostra.

O Wah Wah é um exemplo desta situação, pois neste trabalho o mesmo é implementado através de um filtro passa-faixa de segunda-ordem com dois pólos complexos e com dois zeros no infinito. Adicionalmente, a complexidade deste filtro pode aumentar à medida que parâmetros e pré-requisitos de projeto são adicionados, e a análise de um diagrama de blocos se torna muito trabalhosa.

Neste trabalho o projeto foi iniciado a partir da Transformada de Laplace (função de transferência) de um filtro passa-faixa. A função de transferência deste filtro é exibida na Equação (4.1):

$$T(s) = \frac{Bs}{s^2 + Bs + W}, \quad B = \frac{\omega_0}{Q}, \quad W = \omega_0^2 \tag{4.1}$$

Será disponibilizado ao músico que deseja usar o efeito Wah Wah a variação dos parâmetros Q, as frequências máximas e mínimas dos picos das funções passa-faixa e a taxa de variação (rate) entre as frequências máxima e mínima. Este procedimento é adotado mundialmente em pedais dos mais variados efeitos para permitir que o usuário (músico) ajuste os parâmetros ao seu gosto pessoal.

O parâmetro Q poderá ser variado entre 1,5 e 4. As frequências mínima e máxima onde ocorrem os picos de amplitude do efeito são representadas pelo parâmetro ω_0 poderá variar entre as seguintes opções:

- Frequência mínima: Entre 300 Hz e 400 Hz;
- Frequência máxima: Entre 2500 Hz e 4000 Hz.

Será permitido ao usuário controlar a frequência do LFO desde 1 Hz até 4 Hz. Como a intenção é utilizar este efeito em conjunto com um instrumento e amplificador, o volume geral do áudio é fixo na implementação física.

Como a Equação 4.1 não está adequada para a simulação do algoritmo de processamento no MATLAB, outras ferramentas serão abordadas na próxima seção para demonstrar como obter uma equação no domínio da amostra, simulá-la e implementá-la no Arduino.

4.2 TRANSFORMADA Z

A transformada Z é uma ferramenta matemática de transformação linear utilizada na análise de sinais digitais e sinais discretos resultantes de conversão analógico-digital (LATHI; GREEN, 2018).

Quando se aplica a transformada Z em uma equação, esta passa a fazer parte do domínio Z, e a resolução de tais equações se torna mais simples. Adicionalmente, é possível obter funções de transferência em função de z.

Conforme mencionamos na seção anterior, existem situações onde o usuário pode obter a equação referente ao efeito de áudio manuseando diretamente o diagrama de blocos. Esta equação obtida do diagrama encontra-se no domínio das amostras, ou seja, está em função de n.

Para casos onde não é possível obter a equação diretamente no domínio da frequência, ou ainda, o usuário tem acesso apenas à função de transferência em um domínio diferente do domínio da amostra, a transformada Z simplifica o procedimento de transformação por ser muito utilizada em processamento digital de sinais e construção de filtros.

No caso do efeito Wah Wah, o bloco que o compõe é um filtro passa-faixa, com sua frequência dependendo de um LFO. Neste caso, o bloco que representaria o filtro pode ser expressado através de uma função de transferência no domínio da frequência, ou seja, domínio 's'. Como a adequação ao domínio da amostra é necessária, a transformada Z é utilizada para obter uma função de transferência em função de z, e em seguida, em função de n.

Como a Equação 4.1 está no domínio da frequência, utiliza-se a relação entre a transformada Z e a transformada de Laplace. Esta relação também é chamada de transformada bilinear, e está expressa na Equação 4.2, sendo necessária para realizar o procedimento de transformação e utilizar o domínio Z:

$$s = F\frac{(z-1)}{(z+1)}, \quad F = 2f_s \tag{4.2}$$

Na Equação 4.2, F está em função de f_s, que é a frequência de amostragem. Essa frequência é necessária na equação porque o domínio Z é um domínio discreto, e o resultado da equação após aplicar a transformada serão valores também no domínio discreto.

O Apêndice A mostra os procedimentos algébricos que foram realizados, passo a passo, para obter uma função de transferência expressa em Z, exibida na Equação 4.3:

$$T(z) = \frac{BF(1 - z^{-2})}{(F^2 + BF + W) + (2W - 2F^2)z^{-1} + (F^2 - BF + W)z^{-2}} \tag{4.3}$$

Após a obtenção da função de transferência no domínio discreto Z, é necessário adequar a função novamente para que possa ser escrita em um algoritmo do MATLAB no domínio da amostra. Isso deve ser feito aplicando a transformada inversa entre Z e o domínio da amostra, e estes passos também encontram-se disponíveis detalhadamente no Apêndice A.

Para que seja possível escrever a função referente ao efeito Wah Wah no MATLAB, é necessário decompor a função de transferência em função da entrada e saída. Para isso, utiliza-se da relação expressa na Equação 4.4:

$$T(z) = \frac{Y(z)}{X(z)} \tag{4.4}$$

Ainda, através das relações expressas na Equação 4.5, que são referentes à transformada Z, são realizadas as substituições e operações necessárias para obter uma equação de funcionamento do efeito Wah Wah. A Equação 4.6 expressa o resultado obtido após o procedimento algébrico.

$$Y(z)z^{-N} = y[n - N], \quad X(z)z^{-N} = x[n - N] \tag{4.5}$$

$$y[n] = \frac{\begin{array}{r} BF(x[n] - x[n-2]) - (2W - 2F^2)y[n-1] \\ - (F^2 - BF + W)y[n-2] \end{array}}{F^2 + BF + W} \tag{4.6}$$

Em seguida, é necessário substituir os parâmetros desconhecidos para deixá-los em função dos parâmetros que serão modificados na simulação e na implementação física. Estes parâmetros são expressos na Equação 4.7:

$$B = \frac{\omega_0}{Q}, \quad W = \omega_0^2, \quad F = \frac{2}{T} = 2f_s \tag{4.7}$$

Após a substituição, o último passo é a normalização da frequência ω_0. Este procedimento é necessário pois as variáveis podem atingir valores muito elevados, e o Arduino vai enfrentar problemas de overflow se isso acontecer. A normalização é realizada utilizando a Equação 4.8:

$$\omega_0' = \frac{\omega_0}{f_s}, \quad \omega_0 = \omega_0' f_s \tag{4.8}$$

Apesar do MATLAB conseguir lidar com variáveis de grande tamanho, este procedimento será necessário na etapa de implementação física. Então pode-se adiantar esta etapa realizando a normalização antes de simular e validar o algoritmo no MATLAB.

Realizando esta última substituição, a Equação 4.9 é obtida e passa a ser adequada para utilização nos algoritmos do MATLAB.

$$y[n] = \frac{\begin{array}{r}\frac{2}{Q}\omega_0'(x[n] - x[n-2]) - (2{\omega_0'}^2 - 8)y[n-1] \\ - (4 - \frac{2}{Q}\omega_0' + {\omega_0'}^2)y[n-2]\end{array}}{(4 + \frac{2}{Q}\omega_0' + {\omega_0'}^2)} \tag{4.9}$$

Estes procedimentos podem ser realizados para funções no domínio da frequência, ou ainda no domínio z, e que representem qualquer efeito de áudio que o usuário deseje implementar, levando em consideração as limitações e os parâmetros da implementação.

4.3 SIMULAÇÃO NO MATLAB

A simulação através do software MATLAB é uma etapa essencial porque valida a implementação do diagrama de blocos e da equação referente ao efeito de áudio desejado.

Os passos para a simulação podem ser realizados para qualquer efeito de áudio, levando em consideração suas limitações, parâmetros e aplicações. Em relação ao efeito Wah Wah implementado neste documento, o código na íntegra referente à sua simulação encontra-se disponível no Apêndice B.

A Figura 6 mostra o trecho do código referente às configurações básicas necessárias para a implementação do efeito Wah Wah. Essas

configurações podem sofrer alterações caso sejam utilizadas para simular outros efeitos.

```
audiofileInputAddress = 'guitar_solo.wav';
audiofileOutputAddress = 'guitar_solo_wahwah.wav';
[audioInput, samplingFrequency] = audioread(audiofileInputAddress);

audioSize = length(audioInput);

audioOutput = zeros(audioSize, 2);

LFOFrequency = 1; % Low Frequency Oscilator frequency, in Hertz
qualityFactor = 1.5; % 'Quality' factor, determines the band-pass filter bandwidth
minFrequency = 300; % Lowest frequency, in Hertz
maxFrequency = 2500; % Highest frequency, in Hertz
frequencyMean = (minFrequency + maxFrequency) / 2;
frequencyVariance = (minFrequency - maxFrequency) / 2;

leftInputBuffer = [0 0];
rightInputBuffer = [0 0];
leftOutputBuffer = [0 0];
rightOutputBuffer = [0 0];
sample = 0;
```

Figura 6 – Bloco de configuração do algoritmo

Inicialmente, é necessário informar ao MATLAB como realizar a leitura do arquivo de áudio a ser processado. Isso é realizado informando o endereço do arquivo a ser lido no programa, utilizando a função *audioread*. Este procedimento é realizado para arquivos em formato *wav*, e deve-se adequar a função de leitura ao tipo de arquivo que será lido.

As matrizes com as amplitudes de áudio (*audioInput*) e a frequência de amostragem (*samplingFrequency*) são resultados diretos da função *audioread* e são armazenados para aplicar o efeito.

A matriz referente ao sinal de saída (*audioOutput*) é inicializada com valor zero para possuir um tamanho fixo (*audioSize*). No algoritmo, há buffers (*leftInputBuffer*, *rightInputBuffer*, *leftOutputBuffer* e *rightOutputBuffer*) utilizados no processo iterativo para armazenar os valores intermediários de iterações anteriores de entrada e saída, e devem ser inicializados com valor zero para não terem valores aleatórios que afetarão o áudio.

Para fins de simulação, os parâmetros a serem variados são declarados de maneira fixa, pois na implementação física isso é realizado com o manuseio de chaves e potenciômetros na Placa Shield. Caso o usuário deseje alterar estes parâmetros na simulação, estes são alterados manualmente e uma nova simulação é realizada. Na simulação do Wah Wah, os valores são mostrados na Figura 6.

Os parâmetros utilizados na simulação são a frequência do LFO (*LFOFrequency*), o fator de qualidade do filtro passa-faixa (*qualityFactor*), e as frequências mínima e máxima onde ocorrem os picos de amplitude do efeito (*minFrequency* e *maxFrequency*).

Para parâmetros de outros efeitos, há a necessidade de definilos no escopo do algoritmo, e de preferência utilizando nomenclatura intuitiva para que seja mais fácil localizar e alterar tais parâmetros posteriormente.

Após estes procedimentos, são utilizados métodos iterativos para implementar a equação do efeito de áudio desejado. O processo iterativo deve processar a saída, utilizando a função de transferência do efeito, para todas as amostras obtidas. A Figura 7 mostra o algoritmo iterativo que foi implementado.

```
for index = 1:audioSize
    leftInput = audioInput(index, 1);
    rightInput = audioInput(index, 2);

    omega0 = 2 * pi *(frequencyMean + frequencyVariance * ...
                      sin(2 * pi * LFOFrequency / samplingFrequency * sample)) ...
             / samplingFrequency;

    % `buffer(mod(sample, 2) + 1)` is the oldest value that is being
    % replaced, therefore it is equal to `y[n-2]` or `x[n-2]`
    leftOutput = ((2 / qualityFactor) * omega0 * (leftInput - leftInputBuffer(mod(sample, 2) + 1)) ...
              - (2 * (omega0 * omega0) - 8) * leftOutputBuffer(mod(sample + 1, 2) + 1) ...
              - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0)) * leftOutputBuffer(mod(sample, 2) + 1) ...
                ) / (4 + (2 / qualityFactor) * omega0 + (omega0 * omega0));
    rightOutput = ((2 / qualityFactor) * omega0 * (rightInput - rightInputBuffer(mod(sample, 2) + 1)) ...
              - (2 * (omega0 * omega0) - 8) * rightOutputBuffer(mod(sample + 1, 2) + 1) ...
              - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0)) * rightOutputBuffer(mod(sample, 2) + 1) ...
                ) / (4 + (2 / qualityFactor) * omega0 + (omega0 * omega0));

    leftInputBuffer(mod(sample, 2) + 1) = leftInput;
    rightInputBuffer(mod(sample, 2) + 1) = rightInput;
    leftOutputBuffer(mod(sample, 2) + 1) = leftOutput;
    rightOutputBuffer(mod(sample, 2) + 1) = rightOutput;

    sample = sample + 1;
    if (sample >= samplingFrequency)
        sample = 0;
    end

    audioOutput(index, :) = [leftOutput rightOutput];
end
```

Figura 7 – Algoritmo iterativo do Wah Wah

O parâmetro ω_0 (*omega0*) é alterado a cada iteração por depender do LFO. Dentro de uma iteração, os valores dos buffers, de ω_0 e da amostra atual de entrada (*leftInput* e *rightInput*, um para cada canal) são utilizados para calcular a amostra de saída atual (*leftOutput* e *rightOutput*) com base na função de transferência do efeito. Os valores

de saída são armazenados na matriz referente ao sinal de saída. No fim da iteração os valores dos buffers são atualizados, e uma nova iteração é iniciada.

A Figura 8 mostra o bloco referente aos resultados, onde são gerados os gráficos e o sinal de saída é gravado.

```
figure(1);
plot(audioInput, 'b');
title('Audio Input');
xlabel('Samples');
ylabel('Amplitude');

figure(2);
plot(audioOutput, 'r');
title('Audio Output');
xlabel('Samples');
ylabel('Amplitude');

audiowrite(audiofileOutputAddress, audioOutput, samplingFrequency)
```

Figura 8 – Bloco de resultados do algoritmo

Para escrever um novo arquivo com o áudio processado, utiliza-se a função *audiowrite*. Esta função é compatível com arquivo em formato *wav* e outros arquivos de áudio mais comumente utilizados.

Comandos para a geração dos gráficos de entrada e saída foram utilizados na simulação, de maneira a comparar os dois áudios visualmente e observar características que apontem o processamento do áudio. Ainda, com a utilização do Ocenaudio, é possível ouvir a faixa de áudio gerada e observar os gráficos do mesmo, o que valida a simulação do efeito.

A Figura 9 mostra o gráfico contendo as amostras do sinal de áudio de entrada (ou seja, sem processamento). A Figura 10 mostra o gráfico contendo as amostras do sinal de áudio de saída, após seu processamento utilizando o efeito Wah Wah.

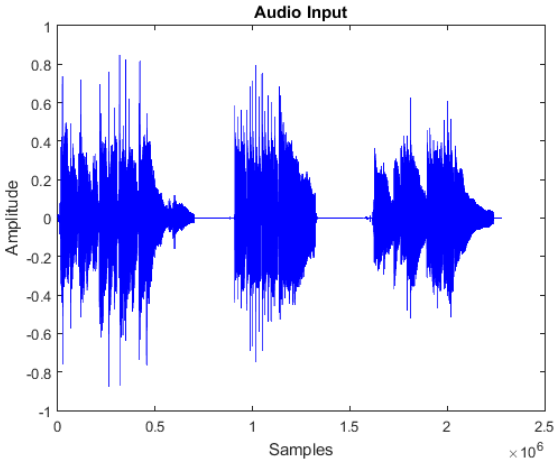

Figura 9 – Sinal de áudio sem processamento na simulação do MATLAB

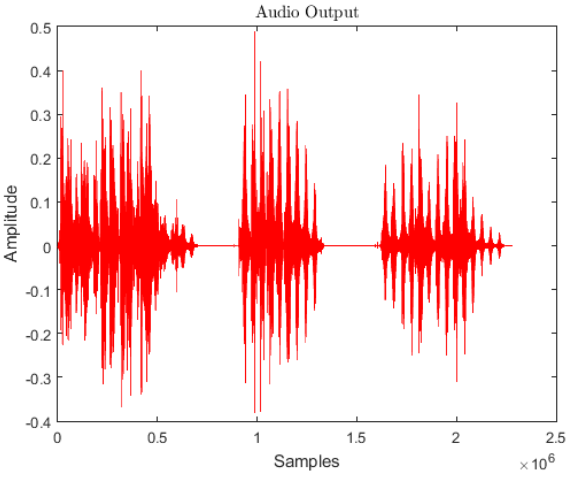

Figura 10 – Sinal de áudio após o processamento na simulação do MATLAB

Na Figura 11, pode-se observar o gráfico do áudio original no Ocenaudio, e na Figura 12, o gráfico do áudio processado.

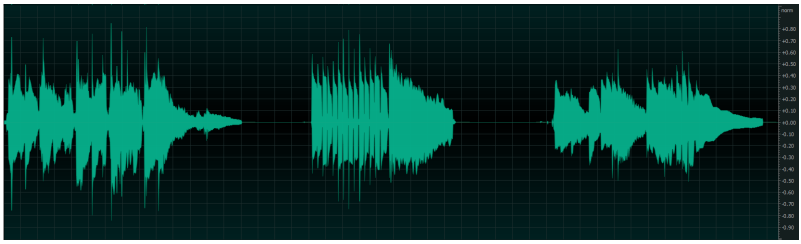

Figura 11 – Sinal de áudio sem processamento, visualizado no Ocenaudio

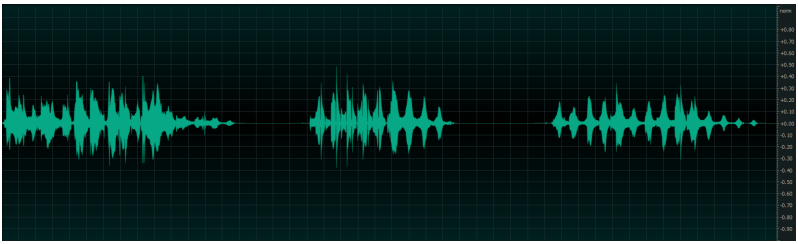

Figura 12 – Sinal de áudio após o processamento, visualizado no Ocenaudio

Ao comparar os dois gráficos, é possível observar as oscilações características do efeito Wah Wah no sinal de saída e seus picos de amplitude. É possível notar que no sinal de saída, houve uma atenuação na amplitude devido ao filtro utilizado para implementar o efeito. Isso pode ser corrigido adicionando um ganho na saída.

4.4 IMPLEMENTAÇÃO FÍSICA

Para a implementação física foi utilizado o mesmo algoritmo da simulação, já que este já foi escrito pensando na implementação física. Alguns ajustes foram necessários, como a mudança de leitura/escrita de um arquivo de áudio para a leitura/escrita dos conversores analógico-digital, além da configuração das interrupções. O código da implementação física na íntegra se encontra no Apêndice C.

Como a implementação física precisa ser em tempo real, algumas otimizações precisam ser feitas. Uma delas é declarar todas as variáveis

que não vão mudar de valor durante a execução como constantes, para que o compilador possa reduzir o número de instruções por iteração.

Para facilitar a customização do efeito, todos os parâmetros ajustáveis foram declarados no início do código, e depois utilizados para calcular os valores que serão utilizados durante as iterações, conforme Figura 13.

```
// * USER-DEFINED VARIABLES **** **** **** **** **** **** **** **** **** ****
const float lowestQualityFactor = 1.5;
const float highestQualityFactor = 4.0;
const float lowestLFOFrequency = 1.0;
const float highestLFOFrequency = 4.0;
const int lowestMaxFrequency = 300;
const int highestMaxFrequency = 400;
const int lowestMinFrequency = 2500;
const int highestMinFrequency = 4000;
// **** **** **** **** **** **** **** **** **** **** **** **** **** **** ****

// Constants used for the map functions
const int lowQualityFactor = lowestQualityFactor * 1000;
const int highQualityFactor = highestQualityFactor * 1000;
const int lowLFOFrequency = lowestLFOFrequency * 1000;
const int highLFOFrequency = highestLFOFrequency * 1000;
const int lowFrequencyMean = (lowestMaxFrequency + lowestMinFrequency) * 10;
const int highFrequencyMean = (highestMaxFrequency + highestMinFrequency) * 10;
const int lowFrequencyVariance = (lowestMaxFrequency - lowestMinFrequency) * 10;
const int highFrequencyVariance = (highestMaxFrequency - highestMinFrequency) * 10;
```

Figura 13 – Definição dos limites e parâmetros ajustáveis

Também são declaradas constantes para facilitar a leitura e compreensão do código, como feito na Figura 14.

```
// Constants to name pin numbers:
const int LED = 3;
const int FOOTSWITCH = 7;
const int TOGGLE = 2;

// The sampling frequency is set to 44.1 kHz since it's the usual frequency for sampling audio.
const int samplingFrequency = 44100;

// Maximum and minimum values for the different ADC quantifiers.
// Input ADC has 12 bits of resolution, and starts from 0, therefore it goes from 0 to 2^12 - 1.
const int lowAnalog = 0;
const int highAnalog = pow(2, 12) - 1;
// Digital audio has 16 bits of resolution, and is centered on 0, therefore it goes from -2^15 to 2^15 - 1.
const int lowDigital = - pow(2, 15);
const int highDigital = pow(2, 15) - 1;
// Potentiometers ADC has 10 bits of resolution, and starts from 0, therefore it goes from 0 to 2^10 - 1.
const int lowPotentiometer = 0;
const int highPotentiometer = pow(2, 10) - 1;
```

Figura 14 – Declaração de constantes para facilitar leitura do código

Por fim, todas as variáveis que serão usadas no código precisam ser declaradas no início dele, como feito na Figura 15. É importante também notar que os tipos das variáveis (*int, float, uint_ 16...*) precisam se adequar aos valores que as variáveis assumirão, e os nomes delas precisam ser descritivos para que as expressões possam ser compreendidas sem necessitar de comentários no código.

```
// Variables to store ADC outputs:
int leftInputADC, rightInputADC;
int potentiometer0, potentiometer1, potentiometer2;

// Variables to be used during the loops.
int sample = 0;
int leftInput, rightInput;
int leftInputBuffer[2] = {0, 0};
int rightInputBuffer[2] = {0, 0};
float leftOutput, rightOutput;
int leftOutputBuffer[2] = {0, 0};
int rightOutputBuffer[2] = {0, 0};
float LFOFrequency;
float qualityFactor;
float frequencyVariance;
float frequencyMean;
float omega0;
```

Figura 15 – Cálculo das constantes para os parâmetros ajustáveis

O *setup* do Arduino é a função que roda apenas uma vez no momento que a placa é ligada, dentro dele são realizadas as configurações que necessitam ser feitas antes de a placa começar a funcionar, como por exemplo ativar pinos ou iniciar e ajustar as interrupções. O *setup* do código do Wah Wah se encontra na Figura 16.

Para configurar corretamente a interrupção por timer, foi usado o conjunto de comandos *pmc_ set_ writeprotect(false)* para permitir a escrita das configurações do contador de timer no microcontrolador, e *pmc_ enable_ periph_ clk(ID_ TC4)* para habilitar o contador de timer de número 4. Depois, a função *TC_ Configure* foi usada para configurar esse contador, sendo que foi selecionado o grupo *TC1*, que contém os contadores 3, 4 e 5, e selecionado o contador de índice 1 desse grupo, que é o contador de número 4. Como último argumento para essa função, foram setadas as configurações para usar o modo de onda *(TC_ CMR_ WAVE)*, contar cada vez que o registrador C for reiniciado *(TC_ CMR_ WAVSEL_ UP_ RC)*, e para usar como base o clock 2

(TC_CMR_TCCLKS_TIMER_CLOCK2), que tem como frequência base 10,5 MHz. Então, o registrador C é configurado para ser reiniciado cada vez que chegar no valor 10,5 MHz dividido pela frequência de amostragem, 44,1 kHz, que é aproximadamente 238 (*TC_SetRC(TC1, 1, 10500000 / samplingFrequency)*) (isso faz com que a frequência de amostragem não seja exatamente de 44,1 kHz, mas isso não é problemático), e por fim o contador 4 foi iniciado (*TC_START(TC1, 1)*).

Com o contador configurado, foi então ajustado para que ele dispare a interrupção 4, configurando ao mesmo tempo o contador para habilitar a interrupção quando ficar ativo (*TC1->TC_CHANNEL[1].TC_IER=TC_IER_CPCS*), e desabilitar a interrupção quando o contador ficar inativo (*TC1->TC_CHANNEL[1].TC_IDR= TC_IER_CPCS*), e por fim habilitando a interrupção do contador 4 no controlador de interrupções.

Lembrando que essas configurações servem especificamente para o Arduino DUE, para outros microcontroladores é necessário pesquisar como configurar a interrupção.

No setup, também é necessário configurar os conversores analógico-digital, e a configuração neste código foi a seguinte: os conversores são configurados para fazerem a leitura constantemente (*ADC->ADC_MR |= 0x80*), os conversores são iniciados (*ADC->ADC_CR = 2*), e então os canais utilizados (0, 1 para o input de áudio, e 8, 9, 10 para os potenciômetros) são habilitados (*ADC->ADC_CHER = 0x1CC0*).

Como último passo do setup, os conversores digital-analógico da saída são iniciados (*analogWrite(DAC0, 0)* e *analogWrite(DAC1, 0)*), o pino do LED é configurado para ser escrito (*pinMode(LED, OUTPUT)*) e o pino do FootSwitch é configurado para ser lido (*pinMode(FOOTSWITCH, INPUT_PULLUP)*).

```
void setup()
{
  // Enable timer clock for interrupt TC4.
  pmc_set_writeprotect(false);
  pmc_enable_periph_clk(ID_TC4);

  // Configure the timer for interrupt TC4.
  TC_Configure(
    TC1, // Points to TC3.
    1, // Move the pointer 1 position to select TC4.
    TC_CMR_WAVE // Waveform mode.
    | TC_CMR_WAVSEL_UP_RC // UP mode with automatic trigger on RC Compare
    | TC_CMR_TCCLKS_TIMER_CLOCK2 // Use clock of 10.5 MHz (DUE clock / 8 = 84 MHz / 8)
  );
  // Set interrupt rate of around the sampling frequency.
  // For a sampling frequency of 44.1 kHz, interrupts whenever the counter reaches 10.5 MHz / 44.1 kHz =~ 238.
  TC_SetRC(TC1, 1, 10500000 / samplingFrequency);
  // Start clock for interrupt TC4.
  TC_Start(TC1, 1);

  // Enable timer interrupts on the timer
  TC1->TC_CHANNEL[1].TC_IER=TC_IER_CPCS;
  TC1->TC_CHANNEL[1].TC_IDR=~TC_IER_CPCS;

  // Enable the interrupt in the nested vector interrupt controller
  // TC4_IRQn where 4 is the timer number * timer channels (3) + the channel number
  // (=(1*3)+1) for timer1 channel1
  NVIC_EnableIRQ(TC4_IRQn);

  // ADC Configurations:
  // DAC in free running mode.
  ADC->ADC_MR |= 0x80;
  // Start ADC conversion.
  ADC->ADC_CR=2;
  // Enable ADC channels 0, 1, 8, 9, 10.
  ADC->ADC_CHER=0x1CC0;

  // DAC Configuration:
  // Enable DAC0.
  analogWrite(DAC0,0);
  // Enable DAC1.
  analogWrite(DAC1,0);

  // Set pin modes
  pinMode(LED, OUTPUT);
  pinMode(FOOTSWITCH, INPUT_PULLUP);
}
```

Figura 16 – Setup do código do Wah Wah

O *loop* é a função que começa a rodar quando o *setup* finaliza, e só para quando a placa é desligada. Em geral, é no *loop* que a função principal do programa é realizada (como o controle de um carrinho, por exemplo). No caso de efeitos de áudio é necessário se utilizar de interrupções porque estas conseguem facilmente ser disparadas com uma frequência fixa. Então o *loop* é usado apenas para ler os valores dos conversores analógico-digital. O *loop* do código do Wah Wah é apresentado na Figura 17.

```
void loop()
{
  // Wait for ADC 0, 1, 8, 9, 10 conversion to complete.
  while((ADC->ADC_ISR & 0x1CC0) != 0x1CC0);

  // Read data from ADCs 0, 1.
  leftInputADC = ADC->ADC_CDR[7];
  rightInputADC = ADC->ADC_CDR[6];

  // Read data from ADCs 8, 9, 10.
  potentiometer0 = ADC->ADC_CDR[10];
  potentiometer1 = ADC->ADC_CDR[11];
  potentiometer2 = ADC->ADC_CDR[12];

  // Light the LED when the FOOTSWITCH is active.
  digitalWrite(LED, digitalRead(FOOTSWITCH));
}
```

Figura 17 – Loop do código do Wah Wah

As interrupções são funções que são chamadas apenas quando uma dada condição é atingida. No caso do código do Wah Wah, a condição é que um contador de pulsos de *clock* do hardware atinja um certo valor, para que a interrupção seja disparada com uma frequência fixa, mas poderia também ser disparada cada vez que um botão fosse pressionado, por exemplo.

Como a interrupção consegue manter uma frequência fixa, ela causa a menor distorção possível na saída de áudio devido à função de transferência do efeito, e por isso é nela que foi implementada a equação de diferenças que compõe o efeito de áudio do Wah Wah.

A interrupção do código do Wah Wah está na Figura 18, e pode ser dividas em algumas partes:

1. A transformação das leituras dos pinos do Arduino (entrada de áudio e potenciômetros) em valores que possam ser utilizados nas equações, utilizando a função *map*;

2. O cálculo do valor de ω_0 (*omega0*), que é feito usando o oscilador de baixa frequência para escolher qual frequência o LFO deve assumir;

3. A entrada e os valores dos buffers são passados pela equação de diferenças para determinar a saída;

4. Os buffers são atualizados com os valores das entradas e saídas atuais;

5. A saída é enviada pelo conversor digital-analógico para ser ouvida/gravada.

```
// Interrupt TC4 handler.
void TC4_Handler()
{
  // Convert the inputs from the ADCs 12 bits to digital audio 16 bits:
  leftInput = map(leftInputADC, lowAnalog, highAnalog, lowDigital, highDigital);
  rightInput = map(rightInputADC, lowAnalog, highAnalog, lowDigital, highDigital);

  // Convert the potentiometer values to the range of the variables:
  qualityFactor = map(potentiometer0, lowPotentiometer, highPotentiometer, lowQualityFactor, highQualityFactor) / 1000.0;
  LFOFrequency = map(potentiometer1, lowPotentiometer, highPotentiometer, lowLFOFrequency, highLFOFrequency) / 1000.0;
  frequencyMean = map(potentiometer2, lowPotentiometer, highPotentiometer, lowFrequencyMean, highFrequencyMean) / 10.0;
  frequencyVariance = map(potentiometer2, lowPotentiometer, highPotentiometer, lowFrequencyVariance, highFrequencyVariance) / 10.0;

  // Calculate the variable frequency of omega0:
  omega0 = 2 * PI *(
             frequencyMean + frequencyVariance * sin(2 * PI * LFOFrequency / samplingFrequency * sample)
           ) / samplingFrequency;

  // Calculate outputs using the differences equation:
  leftOutput = ((2 / qualityFactor) * omega0 * (leftInput - leftInputBuffer[sample%2])
                - (2 * (omega0 * omega0) - 8) * leftOutputBuffer[(sample+1)%2]
                - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0)) * leftOutputBuffer[sample%2]
               ) / (4 + (2 / qualityFactor) * omega0 + (omega0 * omega0));
  rightOutput = ((2 / qualityFactor) * omega0 * (rightInput - rightInputBuffer[sample%2])
                 - (2 * (omega0 * omega0) - 8) * rightOutputBuffer[(sample+1)%2]
                 - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0)) * rightOutputBuffer[sample%2]
                ) / (4 + (2 / qualityFactor) * omega0 + (omega0 * omega0));

  // Store current readings:
  leftInputBuffer[sample%2] = leftInput;
  rightInputBuffer[sample%2] = rightInput;
  leftOutputBuffer[sample%2] = leftOutput;
  rightOutputBuffer[sample%2] = rightOutput;

  // Increse/reset sample number:
  ++sample;
  if (sample >= samplingFrequency) {
    sample = 0;
  }

  // Write to the DACs:
  // Select DAC channel 0.
  dacc_set_channel_selection(DACC_INTERFACE, 0);
  // Write output to DAC, mapping it from digital levels (16 bits) back to ADC converter levels (12 bits).
  dacc_write_conversion_data(DACC_INTERFACE, map(leftOutput, lowDigital, highDigital, lowAnalog, highAnalog));
  // Select DAC channel 1.
  dacc_set_channel_selection(DACC_INTERFACE, 1);
  // Write output to DAC, mapping it from digital levels (16 bits) back to ADC converter levels (12 bits).
  dacc_write_conversion_data(DACC_INTERFACE, map(rightOutput, lowDigital, highDigital, lowAnalog, highAnalog));

  // Clear status allowing the interrupt to be fired again.
  TC_GetStatus(TC1, 1);
}
```

Figura 18 – Interrupção com a equação de diferenças do Wah Wah

Na primeira parte, as entradas de áudio são mapeadas dos valores de 12 bits iniciando em 0 (de 0 à $2^{12} - 1$) da saída dos conversores analógico-digital, para um valor de 16 bits centrado em 0 (de -2^{15} à $2^{15} - 1$), compatível com o formato de áudio digital. Essa conversão é necessária porque a equação do Wah Wah gerou valores negativos para amostras de áudio no formato original, então a conversão facilitou na hora de escrever os valores calculados na saída.

Ainda na primeira parte, os valores parâmetros ajustáveis são retirados dos potenciômetros, que são lidos por um conversor analógico-digital com resolução de 10 bits (valores indo de 0 à $2^{10} - 1$: o fator de qualidade é retirado do potenciômetro 0 e convertido para um valor de 1,5 à 4, a frequência do oscilador de baixa frequência é retirada do potenciômetro 1 e convertida para um valor de 1 à 4 Hz, e os limites de frequência do filtro são retirados do potenciômetro 2 e mapeados para os valores de 300-2500 Hz à 400-4000 Hz (para facilitar os cálculos, esses valores foram tratados como uma frequência média (*frequencyMean*, (frequência máxima + frequência mínima) / 2) e uma variação de frequências (*frequencyRange*, (frequência máxima - frequência mínima) / 2)).

Vale ressaltar que seria mais eficiente utilizar tabelas de valores no lugar da função map para converter os valores vindos dos conversores analógico-digital, porém isso não foi feito neste código para se manter a simplicidade. Isso também é verdadeiro para o cálculo de ω_0 a seguir, e a escrita da saída no final.

Na segunda parte, o valor de ω_0, que é a frequência com ganho máximo no filtro passa-banda, é calculado. O cálculo é feito usando um seno com a frequência do oscilador, o valor instantâneo desse seno é multiplicado pelo valor de variação de frequência, e depois somado com a frequência média do filtro.

Na terceira parte, os valores calculados até então são passados pela equação de diferenças do Wah Wah, que é determinada no Apêndice A. Essa equação também poderia ser otimizada, salvando operações intermediárias em variáveis ou utilizando bibliotecas mais eficientes para fazer essas operações, mas isso também não foi feito neste código para manter a simplicidade.

Na quarta parte, os valores dos buffers são atualizados com os valores atuais da entrada e da saída, e a variável usada para controlar o seno do oscilador de baixa frequência é incrementada, e zerada caso chegue no valor máximo. Para manter os buffers com um tamanho reduzido e facilitar a lógica de acesso deles, os buffers tem o tamanho do número de amostras anteriores (como essa equação necessita de duas amostras anteriores já que tem termos *x[n-2]* e *y[n-2]*, o buffer tem tamanho 2), e são acessados usando a operação resto de divisão (operador % no Arduino) para manter os acessos congruentes (assim, o índice (*sample-1*)%2 se refere aos valores do ciclo anterior, e o índice (*sample-2*)%2 = (*sample*)%2 se refere ao ciclo antes deste). Essa parte também poderia ser feita de maneira mais compreensível usando um *queue* da linguagem C++.

Na quinta parte, o valor da saída calculado é passado por um map para que ele volte aos 12 bits necessários para o conversor digital-analógico, e então é escrito no conversor. Por fim, é necessário chamar a função *TC_ GetStatus* com o grupo e índice do contador para sinalizar que a interrupção foi feita, e assim permitir que ela seja disparada novamente.

5 RESULTADOS

Para obter os resultados, foi conectado o cabo de saída do Arduino na entrada de microfone do computador para que este pudesse obter o sinal e gravá-lo através do Ocenaudio.

O áudio utilizado para a implementação foi o mesmo utilizado na simulação do MATLAB. Ao comparar a gravação do áudio não processado do Arduino, mostrado na Figura 19, com o áudio original (não processado), mostrado na Figura 11, é possível notar que ambos são muito similares, e portanto a gravação do áudio do Arduino funciona corretamente.

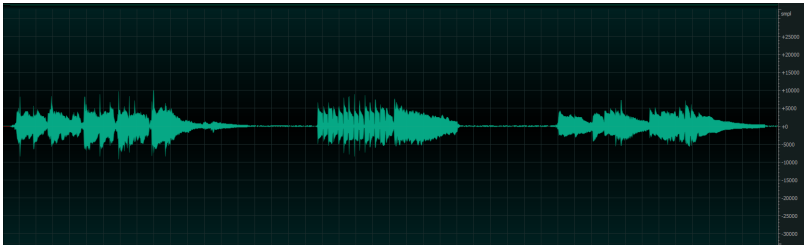

Figura 19 – Sinal de áudio sem processamento na implementação física

Para comparar o áudio após o processamento, foi realizado o mesmo procedimento de gravação, configurando todos os parâmetros variáveis com os valores mínimos, que são os mesmos utilizados na simulação.

Ao comparar os resultados da implementação física, mostrados na Figura 20, com os resultados da simulação, mostrados na Figura 12, é possível observar que a qualidade da implementação física é inferior à simulada. Isto era esperado por conta dos diversos fatores inerentes à implementação física, como ruído de quantização dos conversores analógico-digital, estática no hardware e nos conectores utilizados para a gravação do áudio e tolerâncias dos componentes. Apesar disso, é possível notar a presença do efeito, verificando que os 'picos' e 'vales' do sinal da implementação física ocorrem aproximadamente nos mesmos pontos do sinal simulado.

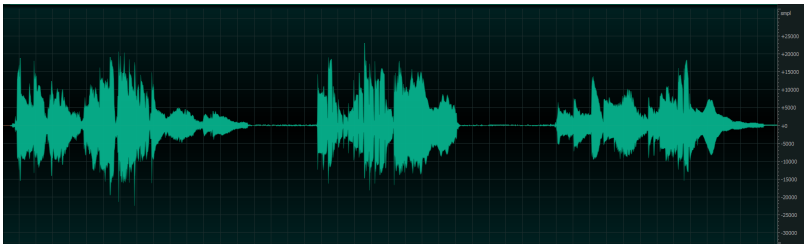

Figura 20 – Sinal de áudio após o processamento na implementação física

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, foi possível realizar o estudo da implementação de efeitos de áudio da forma digital. Diversas técnicas foram aplicadas para realizar o processamento do áudio e alcançar o efeito desejado. Através deste estudo, também foi possível criar um tutorial para a construção de efeitos de áudio de maneira digital, através do uso do Arduino DUE com PedalShield DUE.

Ao implementar o efeito Wah Wah utilizando o tutorial criado, os resultados foram satisfatórios. Após a conclusão da implementação, foi possível realizar um teste físico conectando um violão e um amplificador, respectivamente, na entrada e na saída do conjunto Arduino e PedalSHIELD, e foi possível escutar a presença do efeito e usufruir da implementação física realizada.

Para trabalhos futuros, pode-se comparar os resultados com um pedal analógico que implementa o Wah Wah, para verificar as diferenças existentes nos dois tipos de implementação. Também, pode-se fazer as otimizações no código como sugerido na seção de implementação. Adicionalmente, pode-se passar a entrada por um filtro redutor de ruídos/interferência para reduzir seus impactos na saída de áudio.

REFERÊNCIAS

ARDUINO. *Arduino DUE - Arduino Official Store*. <https://store.arduino.cc/usa/due>. Acessado em 14/04/2021.

ARDUINO. *Products*. <https://www.arduino.cc/en/Main/Products>. Acessado em 10/05/2021.

ARDUINO. *Software*. <https://www.arduino.cc/en/software>. Acessado em 15/05/2021.

ELECTROSMASH. *ElectroSmash Forum*. <https://www.electrosmash.com/forum>. Acessado em 17/06/2021.

ELECTROSMASH. *PedalSHIELD DUE Arduino Guitar Pedal*. <https://www.electrosmash.com/pedalshield>. Acessado em 08/04/2021.

GONTIJO, W. A.; PAVEI, A. H.; FILHO, S. N. Implementação de efeitos de Áudio utilizando arduino due e pedalshield. In: *14º Congresso de Engenharia de Áudio da AES Brasil, 20ª Convenção Nacional da AES Brasil*. São Paulo, Brasil. 17 a 19 de Maio de 2016. p. 81–89.

KO7M. *Arduino DUE Timers*. <http://ko7m.blogspot.com/2015/01/arduino-due-timers-part-1.html>. Acessado em 01/07/2021.

LATHI, B. P.; GREEN, R. *Linear systems and signals*. New York: Oxford University Press, 2018.

MATHWORKS. *MATLAB and Simulink*. <https://www.mathworks.com/products/matlab.html>. Acessado em 15/04/2021.

NOCETI FILHO, S. *Efeitos de áudio digitais e analógicos - Aspectos práticos*. <http://www.linse.ufsc.br/ sidnei/filtros-efeito-audio.html>. Acessado em 17/06/2021.

NOCETI FILHO, S. *Filtros seletores de sinais*. Florianópolis: Editora UFSC, 2020.

OCENAUDIO. <https://www.ocenaudio.com/>. Acessado em 27/04/2021.

PAVEI, A. H. *Several codes based on PedalSHIELD library*. Maio 2016. <https://www.electrosmash.com/forum/software-pedalshield/133-several-codes-based-on-pedalshield-library>. Acessado em 08/04/2021.

PAVEI, A. H. *Implementações de efeitos de áudio utilizando Arduino Due e PedalShield DUE*. Trabalho de Conclusão de Curso — Universidade Federal de Santa Catarina, Florianópolis, 2017.

APÊNDICE A – Equação do Wah Wah e Transformada Z

$$2\pi 300 \text{ rad/s} \leq \omega_0 \leq 2\pi 2500 \text{ rad/s}$$

$$\omega_0 = 2\pi 1400 + 2\pi 1100 \cdot \sin(2\pi \cdot \text{LFO\_freq}) \text{ rad/s}$$

$$T(s) = \frac{Bs}{s^2 + Bs + W}$$

$$T(z) = T(s)|_{s=F\frac{(z-1)}{(z+1)}}$$

$$T(z) = \frac{BF\frac{(z-1)}{(z+1)}}{F^2\frac{(z-1)^2}{(z+1)^2} + BF\frac{(z-1)}{(z+1)} + W}$$

$$T(z) = \frac{BF\frac{(z-1)}{(z+1)}}{F^2\frac{(z-1)^2}{(z+1)^2} + BF\frac{(z-1)}{(z+1)} + W} \cdot \frac{(z+1)^2}{(z+1)^2}$$

$$T(z) = \frac{BF(z-1)(z+1)}{F^2(z-1)^2 + BF(z-1)(z+1) + W(z+1)^2}$$

$$T(z) = \frac{BF(z^2-1)}{F^2(z^2-2z+1) + BF(z^2-1) + W(z^2+2z+1)}$$

$$T(z) = \frac{BF(z^2-1)}{F^2(z^2-2z+1) + BF(z^2-1) + W(z^2+2z+1)} \cdot \frac{z^{-2}}{z^{-2}}$$

$$T(z) = \frac{BF(1-z^{-2})}{F^2(1-2z^{-1}+z^{-2}) + BF(1-z^{-2}) + W(1+2z^{-1}+z^{-2})}$$

$$T(z) = \frac{BF(1-z^{-2})}{(F^2+BF+W) + (2W-2F^2)z^{-1} + (F^2-BF+W)z^{-2}}$$

$$T(z) = \frac{Y(z)}{X(z)}$$

$$\frac{Y(z)}{X(z)} = \frac{BF(1 - z^{-2})}{(F^2 + BF + W) + (2W - 2F^2)z^{-1} + (F^2 - BF + W)z^{-2}}$$

$$((F^2 + BF + W) + (2W - 2F^2)z^{-1} + (F^2 - BF + W)z^{-2}) \cdot Y(z) = BF(1 - z^{-2}) \cdot X(z)$$

$$(F^2 + BF + W) \cdot Y(z) = BF(1 - z^{-2}) \cdot X(z) - ((2W - 2F^2)z^{-1} + (F^2 - BF + W)z^{-2}) \cdot Y(z)$$

$$Y(z) = \frac{BF(1 - z^{-2}) \cdot X(z) - ((2W - 2F^2)z^{-1} + (F^2 - BF + W)z^{-2}) \cdot Y(z)}{F^2 + BF + W}$$

$$Y(z) = \frac{BF(X(z) - X(z)z^{-2}) - ((2W - 2F^2)Y(z)z^{-1} + (F^2 - BF + W)Y(z)z^{-2})}{F^2 + BF + W}$$

$$Y(z)z^{-N} = y[n - N], \quad X(z)z^{-N} = x[n - N]$$

$$y[n] = \frac{BF(x[n] - x[n - 2]) - (2W - 2F^2)y[n - 1] - (F^2 - BF + W)y[n - 2]}{F^2 + BF + W}$$

$$B = \frac{\omega_0}{Q}, \quad W = \omega_0^2, \quad F = \frac{2}{T} = 2f_s$$

$$y[n] = \frac{\frac{\omega_0}{Q}2f_s(x[n] - x[n - 2]) - (2\omega_0^2 - 2(2f_s)^2)y[n - 1] - ((2f_s)^2 - \frac{\omega_0}{Q}2f_s + \omega_0^2)y[n - 2]}{(2f_s)^2 + \frac{\omega_0}{Q}2f_s + \omega_0^2}$$

$$y[n] = \frac{\frac{2}{Q}\omega_0 f_s(x[n] - x[n - 2]) - (2\omega_0^2 - 8f_s^2)y[n - 1] - (4f_s^2 - \frac{2}{Q}\omega_0 f_s + \omega_0^2)y[n - 2]}{4f_s^2 + \frac{2}{Q}\omega_0 f_s + \omega_0^2}$$

$$\omega_0' = \frac{\omega_0}{f_s}, \quad \omega_0 = \omega_0' f_s$$

$$y[n] = \frac{\frac{2}{Q}\omega_0 f_s(x[n] - x[n-2]) - (2\omega_0^2 - 8f_s^2)y[n-1] - (4f_s^2 - \frac{2}{Q}\omega_0 f_s + \omega_0^2)y[n-2]}{4f_s^2 + \frac{2}{Q}\omega_0 f_s + \omega_0^2}$$

$$y[n] = \frac{\begin{array}{r}\frac{2}{Q}(\omega_0' f_s) f_s(x[n] - x[n-2]) - (2(\omega_0' f_s)^2 - 8f_s^2)y[n-1] \\ - (4f_s^2 - \frac{2}{Q}(\omega_0' f_s) f_s + (\omega_0' f_s)^2)y[n-2]\end{array}}{4f_s^2 + \frac{2}{Q}(\omega_0' f_s) f_s + (\omega_0' f_s)^2}$$

$$y[n] = \frac{f_s^2 \frac{2}{Q}\omega_0'(x[n] - x[n-2]) - f_s^2(2{\omega_0'}^2 - 8)y[n-1] - f_s^2(4 - \frac{2}{Q}\omega_0' + {\omega_0'}^2)y[n-2]}{f_s^2(4 + \frac{2}{Q}\omega_0' + {\omega_0'}^2)}$$

$$y[n] = \frac{f_s^2}{f_s^2}\frac{\frac{2}{Q}\omega_0'(x[n] - x[n-2]) - (2{\omega_0'}^2 - 8)y[n-1] - (4 - \frac{2}{Q}\omega_0' + {\omega_0'}^2)y[n-2]}{(4 + \frac{2}{Q}\omega_0' + {\omega_0'}^2)}$$

$$y[n] = \frac{\frac{2}{Q}\omega_0'(x[n] - x[n-2]) - (2{\omega_0'}^2 - 8)y[n-1] - (4 - \frac{2}{Q}\omega_0' + {\omega_0'}^2)y[n-2]}{(4 + \frac{2}{Q}\omega_0' + {\omega_0'}^2)}$$

APÊNDICE B – Código implementado no MATLAB

```
% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
%  wahwah.m
%  Simulates an wahwah effect on an audio file.
% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%% %%%
audiofileInputAddress = 'guitar_solo.wav';
audiofileOutputAddress = 'guitar_solo_wahwah.wav';
[audioInput, samplingFrequency] = audioread(audiofileInputAddress);
audioSize = length(audioInput);
audioOutput = zeros(audioSize, 2);
% Low Frequency Oscilator frequency, in Hertz
LFOFrequency = 1;
% Quality factor, determines the band-pass filter bandwidth
qualityFactor = 1.5;
% Lowest frequency, in Hertz
minFrequency = 300;
% Highest frequency, in Hertz
maxFrequency = 2500;
frequencyMean = (minFrequency + maxFrequency) / 2;
frequencyVariance = (minFrequency - maxFrequency) / 2;
leftInputBuffer = [0 0];
rightInputBuffer = [0 0];
leftOutputBuffer = [0 0];
rightOutputBuffer = [0 0];
sample = 0;
for index = 1:audioSize
    leftInput = audioInput(index, 1);
    rightInput = audioInput(index, 2);
    omega0 = 2 * pi *(frequencyMean + frequencyVariance * ...
                      sin(2 * pi * LFOFrequency ...
                            / samplingFrequency * sample)) ...
             / samplingFrequency;
    % `buffer(mod(sample, 2) + 1)` is the oldest value that will be
    % replaced, therefore it is equal to `y[n-2]` or `x[n-2]`
    leftOutput = ((2 / qualityFactor) * omega0 ...
                    * (leftInput - leftInputBuffer(mod(sample, 2) + 1)) ...
                - (2 * (omega0 * omega0) - 8) ...
                    * leftOutputBuffer(mod(sample + 1, 2) + 1) ...
                - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0)) ...
                    * leftOutputBuffer(mod(sample, 2) + 1) ...
                ) / (4 + (2 / qualityFactor) * omega0 + (omega0 * omega0));
    rightOutput = ((2 / qualityFactor) * omega0 ...
                    * (rightInput - rightInputBuffer(mod(sample, 2) + 1)) ...
                - (2 * (omega0 * omega0) - 8) ...
                    * rightOutputBuffer(mod(sample + 1, 2) + 1) ...
                - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0)) ...
                    * rightOutputBuffer(mod(sample, 2) + 1) ...
                ) / (4 + (2 / qualityFactor) * omega0 + (omega0 * omega0));
    leftInputBuffer(mod(sample, 2) + 1) = leftInput;
    rightInputBuffer(mod(sample, 2) + 1) = rightInput;
    leftOutputBuffer(mod(sample, 2) + 1) = leftOutput;
    rightOutputBuffer(mod(sample, 2) + 1) = rightOutput;
    sample = sample + 1;
```

```
    if (sample >= samplingFrequency)
        sample = 0;
    end
    audioOutput(index, :) = [leftOutput rightOutput];
end
figure(1);
plot(audioInput, 'b');
title('Audio Input');
xlabel('Samples');
ylabel('Amplitude');
figure(2);
plot(audioOutput, 'r');
title('Audio Output');
xlabel('Samples');
ylabel('Amplitude');
audiowrite(audiofileOutputAddress, audioOutput, samplingFrequency)
```

APÊNDICE C – Código implementado no Arduino

```
/* **** **** **** **** **** **** **** **** **** **** **** **** **** **** ****
 *  wahwah.ino
 *  Implements an wahwah effect on Arduino DUE with PedalSHIELD.
 * **** **** **** **** **** **** **** **** **** **** **** **** **** **** ***/
// * USER-DEFINED VARIABLES **** **** **** **** **** **** **** **** **** ****
const float lowestQualityFactor = 1.5;
const float highestQualityFactor = 4.0;
const float lowestLFOFrequency = 1.0;
const float highestLFOFrequency = 4.0;
const int lowestMaxFrequency = 300;
const int highestMaxFrequency = 400;
const int lowestMinFrequency = 2500;
const int highestMinFrequency = 4000;
// **** **** **** **** **** **** **** **** **** **** **** **** **** **** ****
// Constants used for the map functions
const int lowQualityFactor = lowestQualityFactor * 1000;
const int highQualityFactor = highestQualityFactor * 1000;
const int lowLFOFrequency = lowestLFOFrequency * 1000;
const int highLFOFrequency = highestLFOFrequency * 1000;
const int lowFrequencyMean =
  (lowestMaxFrequency + lowestMinFrequency) * 10;
const int highFrequencyMean =
  (highestMaxFrequency + highestMinFrequency) * 10;
const int lowFrequencyVariance =
  (lowestMaxFrequency - lowestMinFrequency) * 10;
const int highFrequencyVariance =
  (highestMaxFrequency - highestMinFrequency) * 10;
// Constants to name pin numbers:
const int LED = 3;
const int FOOTSWITCH = 7;
const int TOGGLE = 2;
// The sampling frequency is set to 44.1 kHz
// since it's the usual frequency for sampling audio.
const int samplingFrequency = 44100;
// Maximum and minimum values for the different ADC quantifiers.
// Input ADC has 12 bits of resolution, and starts from 0,
// therefore it goes from 0 to 2^12 - 1.
const int lowAnalog = 0;
const int highAnalog = pow(2, 12) - 1;
// Digital audio has 16 bits of resolution, and is centered on 0,
// therefore it goes from -2^15 to 2^15 - 1.
const int lowDigital = - pow(2, 15);
const int highDigital = pow(2, 15) - 1;
// Potentiometers ADC has 10 bits of resolution, and starts from 0,
// therefore it goes from 0 to 2^10 - 1.
const int lowPotentiometer = 0;
const int highPotentiometer = pow(2, 10) - 1;
// Variables to store ADC outputs:
int leftInputADC, rightInputADC;
int potentiometer0, potentiometer1, potentiometer2;
// Variables to be used during the loops.
int sample = 0;
```

```
int leftInput, rightInput;
int leftInputBuffer[2] = {0, 0};
int rightInputBuffer[2] = {0, 0};
float leftOutput, rightOutput;
int leftOutputBuffer[2] = {0, 0};
int rightOutputBuffer[2] = {0, 0};
float LFOFrequency;
float qualityFactor;
float frequencyVariance;
float frequencyMean;
float omega0;
void setup()
{
  // Enable timer clock for interrupt TC4.
  pmc_set_writeprotect(false);
  pmc_enable_periph_clk(ID_TC4);
  // Configure the timer for interrupt TC4.
  TC_Configure(
    // Points to TC3.
    TC1,
    // Move the pointer 1 position to select TC4.
    1,
    // Waveform mode.
    TC_CMR_WAVE
    // UP mode with automatic trigger on RC Compare
    | TC_CMR_WAVSEL_UP_RC
    // Use clock of 10.5 MHz (Arduino DUE clock / 8 = 84 MHz / 8)
    | TC_CMR_TCCLKS_TIMER_CLOCK2
  );
  // Set interrupt rate of around the sampling frequency.
  // For a sampling frequency of 44.1 kHz
  // interrupts whenever the counter reaches 10.5 MHz / 44.1 kHz =~ 238.
  TC_SetRC(TC1, 1, 10500000 / samplingFrequency);
  // Start clock for interrupt TC4.
  TC_Start(TC1, 1);
  // Enable timer interrupts on the timer
  TC1->TC_CHANNEL[1].TC_IER = TC_IER_CPCS;
  TC1->TC_CHANNEL[1].TC_IDR = ~TC_IER_CPCS;
  // Enable the interrupt in the nested vector interrupt controller TC4_IRQn,
  // where 4 is the timer number * timer channels (3) + the channel number
  // (=(1*3)+1) for timer1 channel1
  NVIC_EnableIRQ(TC4_IRQn);
  // ADC Configurations:
  // DAC in free running mode.
  ADC->ADC_MR |= 0x80;
  // Start ADC conversion.
  ADC->ADC_CR = 2;
  // Enable ADC channels 0, 1, 8, 9, 10.
  ADC->ADC_CHER = 0x1CC0;
  // DAC Configuration:
  // Enable DAC0.
  analogWrite(DAC0, 0);
```

```
  // Enable DAC1.
  analogWrite(DAC1, 0);
  // Set pin modes
  pinMode(LED, OUTPUT);
  pinMode(FOOTSWITCH, INPUT_PULLUP);
}
void loop()
{
  // Wait for ADC 0, 1, 8, 9, 10 conversion to complete.
  while ((ADC->ADC_ISR & 0x1CC0) != 0x1CC0);
  // Read data from ADCs 0, 1.
  leftInputADC = ADC->ADC_CDR[7];
  rightInputADC = ADC->ADC_CDR[6];
  // Read data from ADCs 8, 9, 10.
  potentiometer0 = ADC->ADC_CDR[10];
  potentiometer1 = ADC->ADC_CDR[11];
  potentiometer2 = ADC->ADC_CDR[12];
  // Light the LED when the FOOTSWITCH is active.
  digitalWrite(LED, digitalRead(FOOTSWITCH));
}
// Interrupt TC4 handler.
void TC4_Handler()
{
  // Convert the inputs from the ADCs 12 bits to digital audio 16 bits:
  leftInput = map(
    leftInputADC,
    lowAnalog,
    highAnalog,
    lowDigital,
    highDigital
  );
  rightInput = map(
    rightInputADC,
    lowAnalog,
    highAnalog,
    lowDigital,
    highDigital
  );
  // Convert the potentiometer values to the range of the variables:
  qualityFactor = map(
    potentiometer0,
    lowPotentiometer,
    highPotentiometer,
    lowQualityFactor,
    highQualityFactor
  ) / 1000.0;
  LFOFrequency = map(
    potentiometer1,
    lowPotentiometer,
    highPotentiometer,
    lowLFOFrequency,
    highLFOFrequency
```

```
) / 1000.0;
frequencyMean = map(
  potentiometer2,
  lowPotentiometer,
  highPotentiometer,
  lowFrequencyMean,
  highFrequencyMean
) / 10.0;
frequencyVariance = map(
  potentiometer2,
  lowPotentiometer,
  highPotentiometer,
  lowFrequencyVariance,
  highFrequencyVariance
) / 10.0;
// Calculate the variable frequency of omega0:
omega0 = 2 * PI * (
           frequencyMean + frequencyVariance
           * sin(2 * PI * LFOFrequency / samplingFrequency * sample)
         ) / samplingFrequency;
// Calculate outputs using the differences equation:
leftOutput = (
                (2 / qualityFactor) * omega0
                 * (leftInput - leftInputBuffer[sample % 2])
               - (2 * (omega0 * omega0) - 8)
                 * leftOutputBuffer[(sample + 1) % 2]
               - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0))
                 * leftOutputBuffer[sample % 2]
             ) / (
                 4 + (2 / qualityFactor) * omega0 + (omega0 * omega0)
             );
rightOutput = (
                 (2 / qualityFactor) * omega0
                  * (rightInput - rightInputBuffer[sample % 2])
                - (2 * (omega0 * omega0) - 8)
                  * rightOutputBuffer[(sample + 1) % 2]
                - (4 - (2 / qualityFactor) * omega0 + (omega0 * omega0))
                  * rightOutputBuffer[sample % 2]
              ) / (
                  4 + (2 / qualityFactor) * omega0 + (omega0 * omega0)
              );
// Store current readings:
leftInputBuffer[sample % 2] = leftInput;
rightInputBuffer[sample % 2] = rightInput;
leftOutputBuffer[sample % 2] = leftOutput;
rightOutputBuffer[sample % 2] = rightOutput;
// Increse/reset sample number:
++sample;
if (sample >= samplingFrequency) {
  sample = 0;
}
// Write to the DACs:
```

```
  // Select DAC channel 0.
  dacc_set_channel_selection(DACC_INTERFACE, 0);
  // Write output to DAC, mapping it from digital levels (16 bits)
  // back to ADC converter levels (12 bits).
  dacc_write_conversion_data(
    DACC_INTERFACE,
    map(leftOutput, lowDigital, highDigital, lowAnalog, highAnalog)
  );
  // Select DAC channel 1.
  dacc_set_channel_selection(DACC_INTERFACE, 1);
  // Write output to DAC, mapping it from digital levels (16 bits)
  // back to ADC converter levels (12 bits).
  dacc_write_conversion_data(
    DACC_INTERFACE,
    map(rightOutput, lowDigital, highDigital, lowAnalog, highAnalog)
  );
  // Clear status allowing the interrupt to be fired again.
  TC_GetStatus(TC1, 1);
}
```