Universidade Federal do ABC

Graduação em Engenharia de Informação

Tales Gouveia Fernandes

IMPLEMENTAÇÃO DIGITAL DE MULTI-EFEITOS PARA GUITARRA

Santo André - SP

Fevereiro 2014

Tales Gouveia Fernandes

IMPLEMENTAÇÃO DIGITAL DE MULTI-EFEITOS PARA GUITARRA

Trabalho apresentado ao curso de Engenharia de Informação

Orientador: Prof. Dr. Claudio José Bordin Júnior

Santo André - SP

Fevereiro 2014

AGRADECIMENTOS

Primeiramente a Deus por ter ajudado a manter a fé nos momentos mais difíceis e por fazer com que mais esse sonho se realize.

Ao meu pai Nivaldo e minha mãe Maria Helena que, com muito carinho e apoio, não mediram esforços para que eu chegasse até esta etapa de minha vida.

Ao professor Claudio, pela paciência na orientação e por ter compartilhado sua experiência, a qual foi de suma importância para realização desse trabalho.

A todos os colegas e professores presentes nesse período em que estive na UFABC.

RESUMO

Este trabalho tem como objetivo desenvolver uma implementação digital, em tempo real, de alguns dos efeitos de guitarra mais conhecidos. A partir de modelos matemáticos obtidos da literatura de processamento de sinais de áudio, fez-se uma prototipagem dos efeitos na plataforma do Simulink. A partir disso, foram desenvolvidos códigos na linguagem de programação C++ utilizando *framework*[1] STK (*Syntesis Toolkit*), que interage com o sistema operacional permitindo o processamento de sinais de áudio em tempo real. Complementando o STK, foi desenvolvida uma GUI (*Graphical User Interface*) na língua portuguesa. Foram abordados os efeitos *distortion*, *delay*, *flanging*, *chorus*, *reverb* e *vibe*.

[1] Pelo fato de a literatura da área de processamento de sinais ser primordialmente de língua inglesa, os termos técnicos utilizados neste trabalho serão mantidos nessa língua.

ABSTRACT

This work aims at developing a digital implementation in real time of some of the most popular guitar effects. Mathematical models obtained from the audio signal processing literature were employed to prototype the effects on the Simulink platform. From those prototypes, code have been developed in the C++ programming language using the STK (Synthesis Toolkit) framework, which interacts with the operating system to allow the processing of audio signals in real time. Complementing the STK, we developed a GUI (Graphical User Interface) in Portuguese. We addressed the following effects: delay, flanging, chorus, reverb and vibe.

ÍNDICE DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

O Violão, ou guitarra acústica, é um instrumento derivado da cítara romana, tendo seu uso expandido com a dominação do império romano. Ao longo de décadas de adaptações e modificações, este instrumento de cordas chegou ao que é hoje conhecido como violão. Em paralelo a este desenvolvimento, surgiram variações, as quais introduziam componentes eletromagnéticos ao instrumento. Por consequência a estas variações e modificações surgiu-se a guitarra elétrica. Em suma, inicialmente desenvolveu-se uma guitarra elétrica com base em um violão utilizando a caixa de ressonância e captadores. No entanto, a captação dos sons é feita pelos captadores indutivos, os quais captam as vibrações mecânicas das cordas de aço, transformando-as em sinais elétricos.

Os captadores indutivos, além de possuírem a função de microfones, também são capazes de amplificar a potência sonora do instrumento. Isso acabou por gerar um pequeno problema, pois os mesmos faziam com que o corpo das guitarras vibrasse, provocando uma alteração sonora devido à realimentação popularmente conhecida como “microfonia”. Esta por sua vez consiste na realimentação dos captadores, a partir de reverberações do som original. Como solução deste problema, o corpo da guitarra passou a ser maciço, e assim chegou à forma contemporânea.

Com a popularidade da guitarra principalmente através do surgimento do *rock and roll* nos anos 60, o qual é marcado por timbres graves e sons sujos, teve-se inicio a era da aplicação de efeitos sonoros. O primeiro registro da característica de efeito sonoro foi o timbre obtido pelo guitarrista Dave Davies, da banda inglesa *The Kinks*. Ele furou propositalmente um alto-falante com um lápis, dando ao som uma característica trêmula e duplicada. Paralelo a isso, descobriu-se que a saturação no ganho dos amplificadores valvulados gerava uma distorção, que se tornou o símbolo marcante do *rock and roll*. Atualmente a distorção é feita por meio de amplificadores valvulados ou transistorizados, e pedais analógicos ou digitais.

Além da distorção, existem vários outros efeitos que auxiliam o guitarrista na busca pelos timbres mais diversos, como por exemplo, o *delay*,

(atraso), o qual causa um determinado atraso no som; o *chorus* (coro), que produz uma sensação de aumento na quantidade de fontes sonoras; o *flanging* (flange), o qual produz o efeito do rebobinar de fitas, baseado na modulação sonora; o compressor, que atenua sinais com amplitude alta e incrementa sinais de baixa amplitude, fornecendo sustentação prolongada deixando o som uniforme; e o *reverb* (reverberação), o qual como o nome sugere, mistura uma reverberação ao som original.

Existem vários outros que se encaixam dentro destes grupos, por exemplo, os efeitos *acoustic* (acustico), *overdrive*, *fuzz* e metal, entram no grupo dos efeitos baseados em distorção das ondas. Já os efeitos *chorus* e *flanging*, tem a característica de alterar a modulação do sinal.

O que começou com o *rock and roll*, hoje se espalha por muitos segmentos musicais, utilizando estes efeitos com o objetivo de melhorar o som, e personalizar as músicas, dando características diversas a solos e arranjos. Os efeitos são acessórios considerados como essenciais para os músicos, pois ajudam na timbragem do som, ou seja, possibilitando novas nuances do som.

Os pedais, como são chamados alguns geradores de efeito, recebem esse nome por serem acionados com os pés. Possuem a principal característica de serem totalmente analógicos e compostos por componentes eletrônicos discretos. Ao entrar no pedal, o som passa por estágios de adequação para que então possa ser modificado. Tais modificações são feitas utilizando-se estágios de saturação, modulação ou filtragem.

Embora os pedais analógicos possam ser combinados com o intuito de oferecessem mais de um efeito, estes ainda são vulneráveis a defeitos e interferências. Além disso, possuem um número reduzido de efeitos. Com isso surgiram as pedaleiras digitais, as quais além de minimizar os defeitos e interferências, possuíam uma maior variedade de efeitos em um único módulo. Elas são compostas por processadores digitais de sinal (DSP's), os quais emulam e processam digitalmente o sinal da guitarra. Com o avanço tecnológico, os efeitos digitais se aproximam cada vez mais aos timbres dos efeitos analógicos.

1.1 OBJETIVO

Este trabalho tem por objetivo construir um simulador de pedaleira multiefeitos para guitarra que funcione em tempo real em um PC. Para isso, utilizou-se um *framework* de processamento de áudio em tempo real chamado *Synthesis Toolkit* (STK) [6], que é programável na linguagem C++. A aquisição do áudio se dá através da entrada de áudio do próprio computador. Os efeitos que serão implementados são os mais utilizados por guitarristas consagrados como, *distortion*, *delay*, *flanging*, *chorus*, *reverb* e *vibe* (vibração).

O texto a seguir está organizado da seguinte forma: na Seção 2 descrevem-se os modelos matemáticos relatados na literatura para os referidos efeitos. Na Seção 3, por sua vez, apresentam-se os modelos em Simulink desenvolvidos neste trabalho com o intuito de validar e determinar conjuntos de parâmetros adequados para os modelos matemáticos da Seção 2. A implementação dos efeitos em C++/STK é descrita na Seção 4, deixando as conclusões deste trabalho para a Seção 5.

Em complemento, relatam-se no Apêndice A os estudos realizados com vistas à implementação dos efeitos em hardware dedicado (DSP). No Apêndice B, fornece-se um guia para a instalação do software STK num computador e, finalizando, lista-se no Apêndice C o código fonte dos programas gerados neste trabalho.

2. DESCRIÇÃO MATEMÁTICA DOS EFEITOS

2.1 *DISTORTION*

O efeito *distortion*, também conhecido como *clipping* (limitação) é uma forma de distorção que corta o sinal em limites pré-estabelecidos. Pode ocorrer quando um sinal tem seu ganho saturado, quando um sinal é digitalizado ou ainda ocorrer na ocasião onde o sinal analógico ou digital é transformado em uma outra forma de onda. O *clipping* pode ser classificado em dois tipos:

- *hard clipping* (rígida) – casos onde o sinal é bruscamente cortado.
- *soft clipping* (suave) – casos onde o sinal continua a ter a forma do sinal original porém com o ganho diminuído.

A Figura 1 mostra os gráficos em função do tempo dos sinais com *soft clipping* e *hard clipping*.

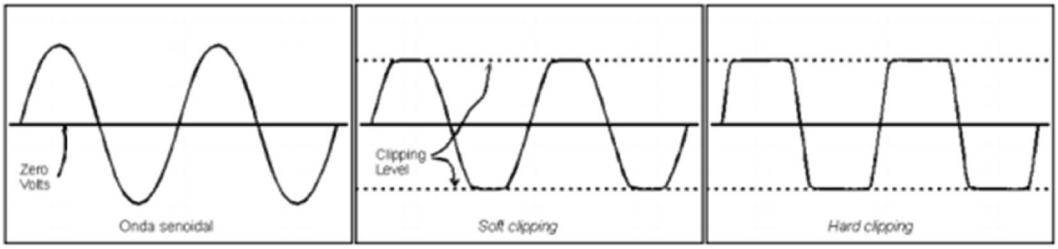


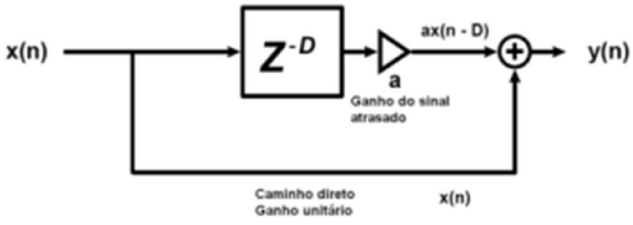
Figura 1: Tipos de clipagem de um efeito distortion. Fonte: www.geofex.com/effxfaq/d101_01.gif.

O *hard clipping* resulta em muitas harmônicas de alta frequência enquanto o *soft clipping* resulta em menos harmônicas de ordem superior e com componentes de distorção intermoduladas. O *distortion* pode ser criado digitalmente com a inserção de harmônicas do sinal original, ou programando-se um limiar máximo de ganho, toda vez que o sinal passar pelo limite.

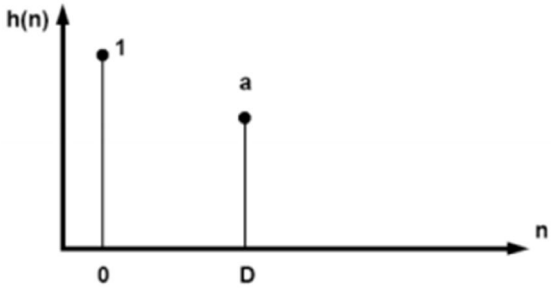
2.2 *DELAY*

O *delay* digital é o efeito de áudio do tipo atraso no tempo. O efeito *delay* é frequentemente a base para que se produzam outros efeitos, tais como *flanging* e *chorus*, os que variam o tempo de forma dinamicamente. É também utilizado em algoritmos de reverberação. Quando uma fonte de som é refletida

em uma superfície rígida distante, uma versão atrasada do sinal original é ouvida em um momento posterior. Para recriar essa reflexão digitalmente, unidades de efeito de *delay*, como no MATLAB, por exemplo, codificam os sinais de entrada e os armazena digitalmente em um vetor até ser novamente requisitados e decodificados de volta à forma analógica. Na Figura 2, (a) pode-se ver a implementação de um *delay* digital, juntamente com um gráfico mostrando uma reflexão simples (b).



(a)



(b)

Figura 2: (a) Implementação delay digital; (b) Gráfico reflexão simples. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

Para criar a reflexão simples de um sinal de entrada, a implementação mostrada acima é representada na seguinte equação de diferenças:

$$y[n] = x[n] + ax[n - D]$$

E sua função de transferência é:

$$H(z) = 1 + az^{-D}$$

Observa-se que o sinal de entrada *x[n]* é adicionado a uma cópia deslocado de *D*. O sinal pode ser atenuado por um fator que é menor do que um, devido ao fato de que superfícies refletivas, bem como o ar, contém uma constante de perda *a* em função da absorção da energia do sinal de origem. O *delay D* representa o tempo total que se leva para retornar da parede que o reflete. *D* é criado utilizando-se um vetor de *buffer* de comprimento específico.

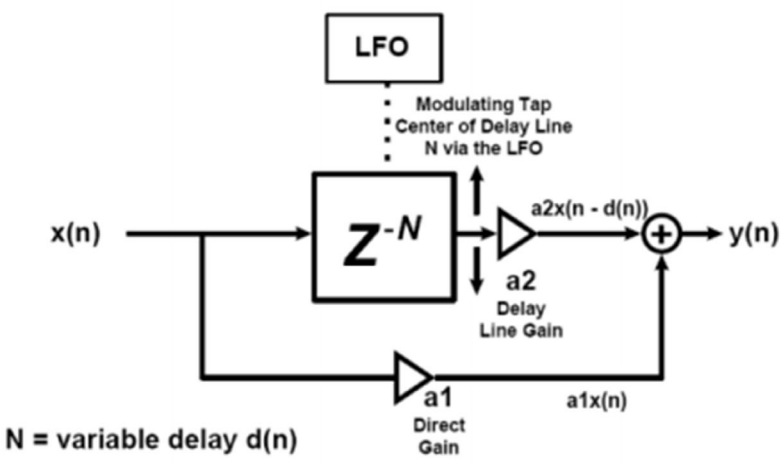
2.3 *CHORUS*

O efeito *chorus* é utilizado para "afinar" os sons. Este algoritmo de *delay* (atrasos entre 15ms e 35ms) tem por função duplicar o efeito que ocorre quando muitos músicos tocam o mesmo instrumento e a mesma música simultaneamente.

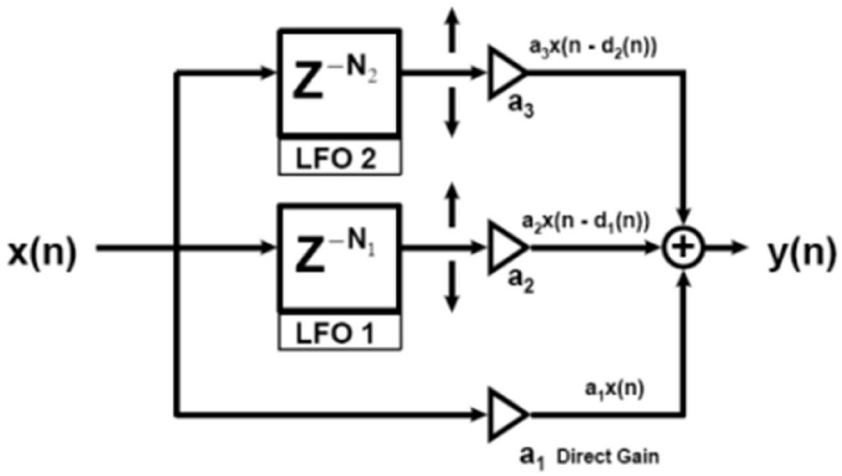
Geralmente, os músicos tocam de maneira sincronizada uns com os outros, mas há sempre ligeiras diferenças no momento, volume, e espaçamento entre cada instrumento que toca as mesmas notas musicais.

O *chorus* pode ser recriado digitalmente com um atraso variável na derivação central, adicionando o resultado variante no tempo atrasado junto com o sinal de entrada. Ao utilizar-se deste efeito criado digitalmente, uma guitarra de seis cordas pode soar como uma guitarra de doze cordas. Os vocais também podem ser reproduzidos de modo a soar como mais do que um vocalista cantando.

O algoritmo de *chorus* é semelhante ao *flanging*, utilizando a mesma equação de diferenças, com exceção do tempo de atraso que é mais longo. Com um tempo da linha de atraso mais longo, o efeito do filtro c*omb* (pente) é reduzido à frequência fundamental com harmônicos de ordem inferior. A Figura 3 (a) e (b) mostram a estrutura do efeito *chorus* simulando dois e três instrumentos respectivamente.



(a)



(b)

Figura 3: (a) Chorus simulando dois instrumentos; (b) Chorus simulando três instrumentos. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

A equação de diferenças da Figura 3 (b) é

$$y[n] = a_1 x[n] + a_2 x[n - d[n]]$$

onde os coeficientes de ganho a_1 e a_2 pode ser um número aleatório de baixa-frequência com média unitária. O valor *d[n]* pode vir de uma tabela LFO (*Low-Frequency Oscilator* – Oscilador de Baixa Frequência) conforme a Figura 4.

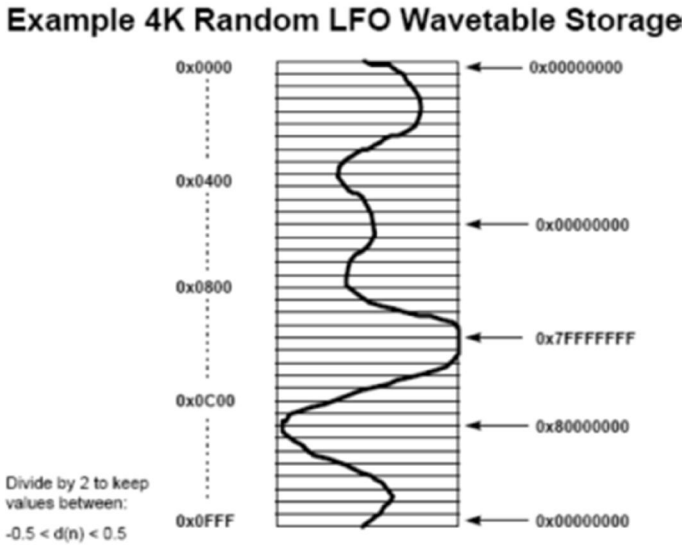


Figura 4: LFO (Low-Frequency oscilator). Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

2.4 *FLANGING*

O efeito *Flanging* ganhou este nome devido a um truque utilizado em estúdios de gravação onde a mesma faixa de áudio era reproduzida em duas

máquinas de fita, bobina a bobina, e engenheiros de gravação sutilmente tocaram a *flange* de uma das bobinas para produzir um curto *delay* entre as máquinas. Então, ao tocar a *flange* da outra bobina, trazeria as máquinas de volta à sincronização, removendo o *delay*. Isto gerou um efeito sonoro muito parecido com o de um "avião a jato".

Em processamento digital de sinais, o efeito *flanging* é produzido ao misturar um sinal atrasado variante (geralmente entre 5ms e 15ms) com o sinal original, o que produzirá uma série de fendas no espectro de frequência. Deste modo, a resposta em frequência de um *flanging* resulta em um filtro c*omb*, como ilustrado na Figura 5.

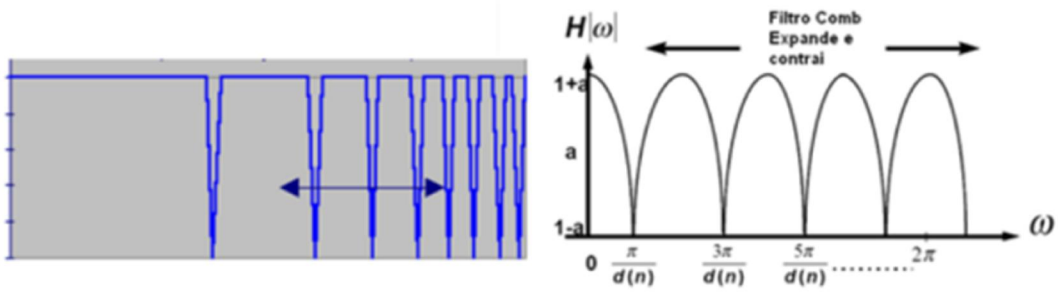


Figura 5: Resposta em frequência de um flanger. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

Com o aumento do *delay*, aumenta-se o número de picos. Modificando-se o *delay*, modifica-se o filtro c*omb*, que por sua vez afeta as frequências que são amplificadas ou canceladas. A Figura 6 mostra o diagrama funcional do efeito *flanging*.

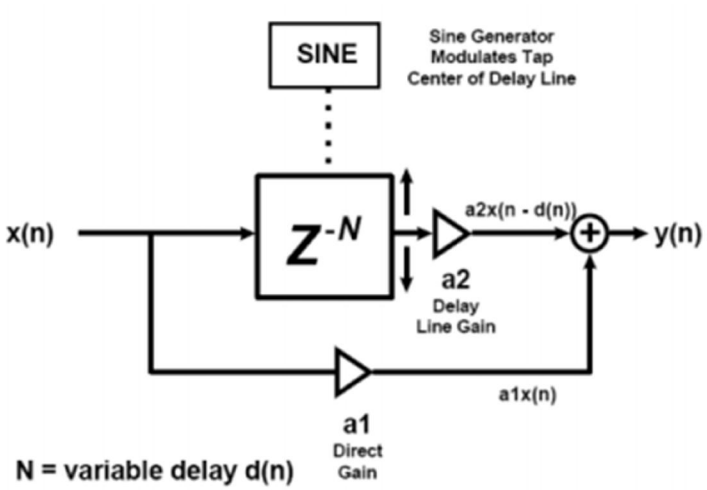


Figura 6: Diagrama funcional flanging. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

Uma importante diferença entre o efeito *flanging* e o *phaser* (fase) é que o *flanging* produz uma grande quantidade de fendas, e os picos entre estas fendas são harmonicamente (musicalmente) relacionados. Já um *phaser* produz um número menor de fendas que são uniformemente espalhados pelo espectro de frequências.

As maiorias dos *flangings* provêm um controle de ressonância para utilizar realimentação interna de modo a aperfeiçoar os picos na frequência de resposta.

Os controles comuns são:

- Taxa de profundidade - controla a velocidade e a distância com que as fendas se movem.
- Intensidade (ou Efeito ou ainda Mixagem) controla o nível do sinal atrasado e consequentemente a profundidade das fendas na frequência.
- Ressonância adiciona ênfase ao aplicar realimentação interna.

2.5 *REVERB*

A Reverberação é outro efeito baseado no tempo. Embora tenha um processamento mais complexo do que os efeitos *eco*, *chorus* e *flanging*, a reverberação é frequentemente confundida com os efeitos de *delay* e *eco*. A maioria das pedaleiras multiefeitos fornece uma variação em ambos os efeitos.

A primeira unidade de simulação de reverberação criada entre os anos 60 e 70 consistia no uso de uma mola mecânica ou placa conectada ao transdutor de áudio, de modo que passasse um sinal elétrico através dos mesmos. Outro transdutor do lado oposto convertia as reflexões mecânicas de volta para o transdutor de saída. Entretanto, isto não criava uma reverberação realística. M. A. Schroeder and James A. Moorer [5] desenvolveram algoritmos de modo a produzir a reverberação realística utilizando um DSP (processador digital de sinais).

O efeito reverberação simula o efeito de reflexão sonora em um grande auditório ou sala, como mostra a Figura 7.

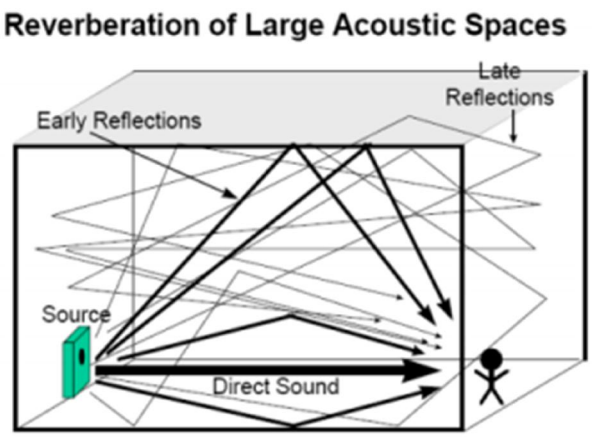


Figura 7: Efeito de reverberação em um auditório. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

Ao invés de poucas repetições discretas de um som como um *delay* de múltiplos atrasos, a reverberação gera muitas repetições atrasadas muito próximas no tempo, de modo que o ouvido não pode distinguir as diferenças entres os atrasos. As repetições são continuamente misturadas ao som. A fonte sonora soa em todas as direções a partir da fonte, reflete nas paredes e tetos e retorna de muitos ângulos com diferentes atrasos. A reverberação está quase sempre presente em ambientes fechados, e as reflexões são ainda maiores quando refletidas de superfícies rígidas. A Figura 8 mostra um som reverberado classificado em três principais faixas: Som direto, Reflexões adiantadas e reflexões misturadas com eco.

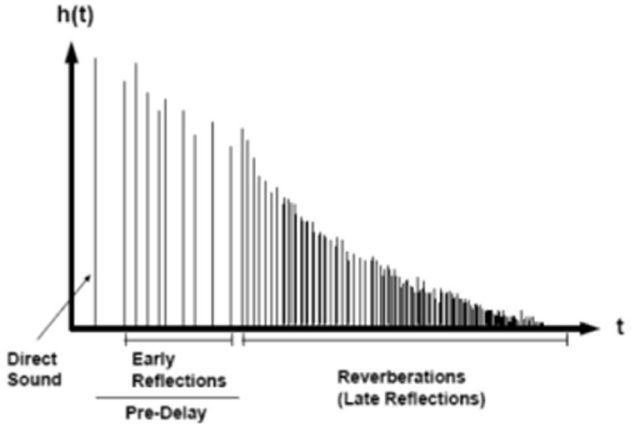
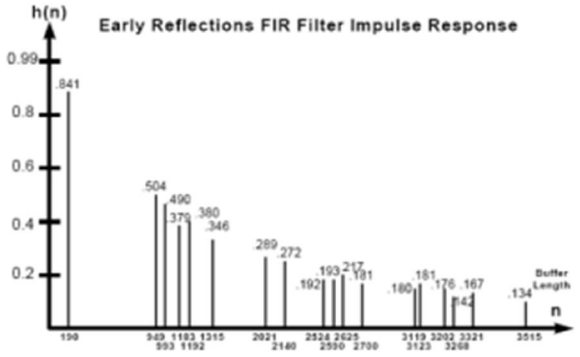


Figura 8: Classificação do som reverberado. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

- Som direto: atinge o ouvinte com som diretamente vindo da fonte
- Reflexões adiantadas: são ecos adiantados, os quais chegam num intervalo entre 10ms e 100ms, oriundos das reflexões de superfícies próximas à fonte sonora.

- Reflexões misturadas com eco: são reflexões sonoras que chegam ao ouvindo após 100ms de atraso.

A Figura 9 apresenta valores típicos de amplitude em função do comprimento do vetor buffer.



Example Reverb Specifications for a Large Auditorium response at 44.1 kHz Sampling Rate

Delay Line Buffer Length		*Time Delay*
Comb 1	*1759*	*40 ms*
Comb 2	*1949*	*44 ms*
Comb 3	*2113*	*48 ms*
Comb 4	*2293*	*52 ms*
Comb 5	*2467*	*56 ms*
Comb 6	*2647*	*60 ms*
Early Reflections	*3520*	*80 ms*
All-Pass Filter	*307*	*7 ms*

Figura 9: Amplitude em função do comprimento do vetor buffer. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

A Figura 10 mostra a estrutura do algoritmo de [5] para uma resposta de um grande auditório assumindo a taxa de amostragem de 44.1 kHz. Em [5], Moorer demonstrou uma técnica que envolve seis filtros *comb* em paralelo com uma saída passa-baixa, somando suas saídas e então mandando o resultado somado para um filtro “passa tudo” antes de produzir o sinal final.

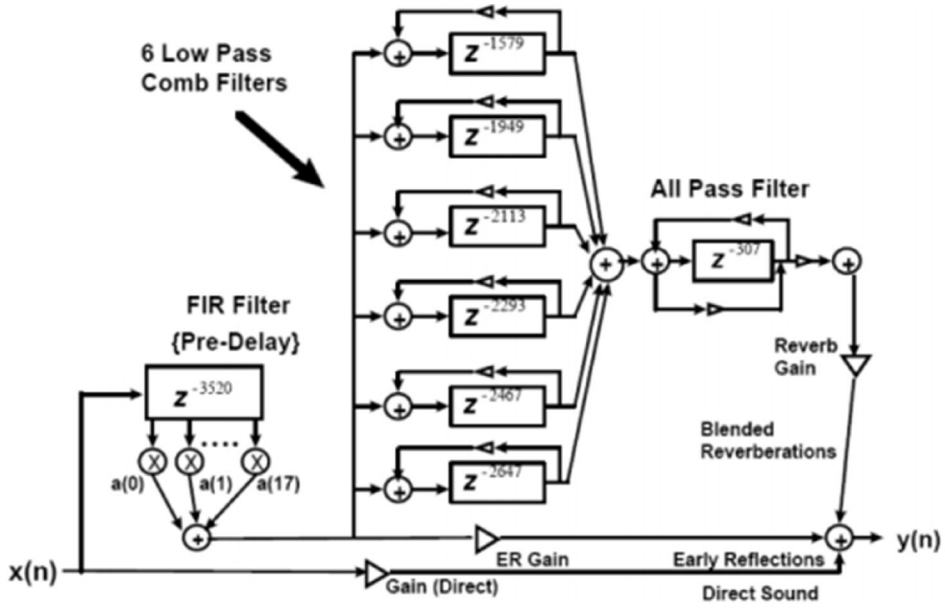


Figura 10: Diagrama do algoritmo de reverberação. Fonte: TOMARAKOS, JOHN E LEDGER, DAN.

2.6 *VIBE*

Existem também efeitos que são baseados na modulação por amplitude do sinal. No caso do efeito *vibe*, o processamento tem a mesma forma da modulação AM feita analogicamente, sendo implementada como

$$y(n) = [1 + \alpha m(n)] \cdot x(n)$$

onde se assume que a amplitude de pico sinal *m(n)* esteja normalizada. O coeficiente α determina a profundidade da modulação do sinal, sendo que para valores próximos de α = 1, a modulação é máxima, e, geralmente, bastante perceptível. O sinal modulante *x(n)* é gerado por um LFO. Com isto, o sinal de saída terá *y(n)* à amplitude variando de acordo com o modulador. A Figura 11 mostra o diagrama esquemático deste efeito.

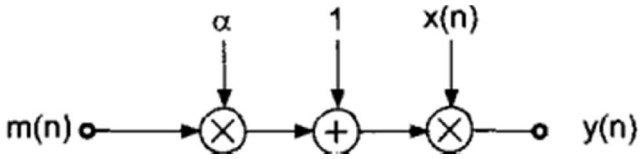


Figura 11: Diagrama esquemático do efeito vibe. Fonte: Udo Zölzer.

Em geral, as frequências da portadora *x(n)* e do sinal *m(x)* são diferentes. Se estas estiverem nas faixas altas de frequências, percebe-se audivelmente três componentes: a portadora, a diferença entre as frequências e a soma das frequências. Considerando a modulação abaixo da faixa de 20 Hertz, ouve-se algo parecido com o efeito *tremolo* (vibração sonora).

3. SIMULAÇÃO

A fim de executar a prototipagem e assim, implementar a simulação dos modelos de efeitos descritos anteriormente, utilizou-se o ambiente Simulink do MATLAB. Esse ambiente proporciona processamento em tempo real e é programável a partir de um ambiente gráfico de fácil manuseio.

Os modelos Simulink descritos a seguir implementam os efeitos a partir de suas descrições matemáticas, dadas na Seção 2. A partir da simulação desses modelos, foi possível estabelecer um conjunto de parâmetros para a implementação em tempo real definitiva, descrita na Seção 4.

Os modelos referentes a todos os efeitos foram reunidos em um *set* inicial. Este *set* tem o intuito de otimizar o processamento, dedicando o processamento apenas para os efeitos que foram realmente selecionado para a simulação.

A Figura 12 mostra a configuração do arranjo dos efeitos.

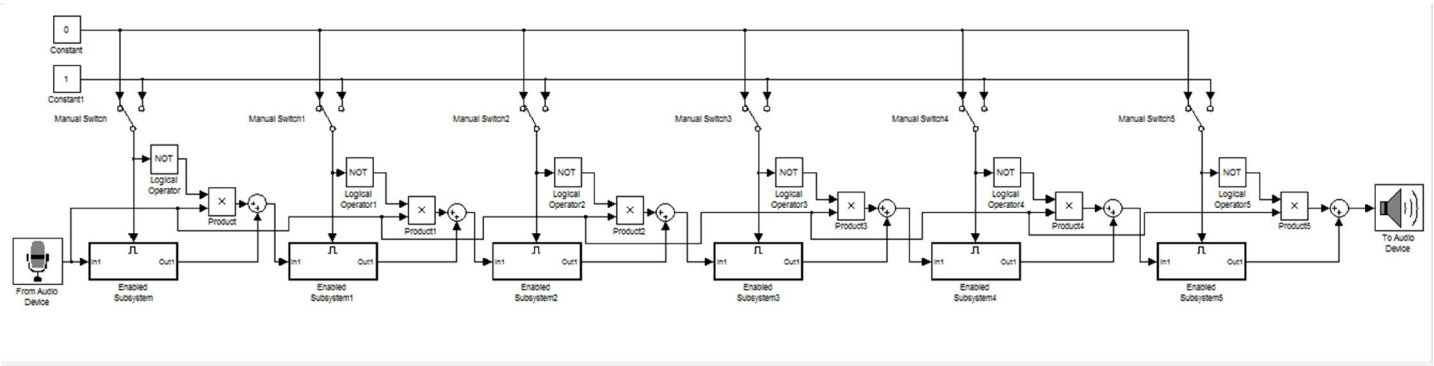


Figura 12: Arranjo geral dos efeitos. Fonte: Elaborado pelo autor.

Cada bloco denominado *Enabled Subsystem*, contem os respectivos efeitos de *distortion*, *delay*, *chorus*, *flanging*, *reverb* e *vibe*. Houve a necessidade de confinar os efeitos em sistemas distintos, os quais são ativados por nível lógico alto, pois assim evita-se que haja processamento desnecessário em efeitos que não são acionados. A estrutura montada fora dos *subsystem* é necessária para que se tenha o *by-pass* do sinal original. Com isso, se nenhum *subsystem* for ativado, o sinal da saída será igual ao sinal de entrada. Porém, se ao menos um *subsystem* for ativado, o sinal original sofrerá o processamento do efeito. Também pode ocorrer

concatenação dos efeitos, ou seja, a saída de cada efeito passará pelo próximo, se este último também estiver ativado.

A montagem desta ordem dos efeitos segue o padrão geral utilizado por músicos, os quais empregam no inicio do arranjo, logo na sequencia da guitarra, efeitos de distorção, posteriormente efeitos de atraso e por fim efeitos de modulação, os quais seguiram para o amplificador do musico.

Expandido cada *subsystem*, pode-se observar como foi feita a construção de cada efeito. A Figura 13 mostra o efeito *distortion*, o qual está presente no *Enabled Subsystem*.

A construção deste efeito tomou-se por base a característica de saturação não linear dos efeitos *distortion*. Com tudo, a simulação da função tangente hiperbólica caracteriza um envelopamento e timbragem bem corportada do áudio, respeitando assim os limites aceitáveis de pico do audio. Por fim, tanto os valores de volume como os valores de saturação podem ser regulados nos blocos correspondentes aos ganhos.

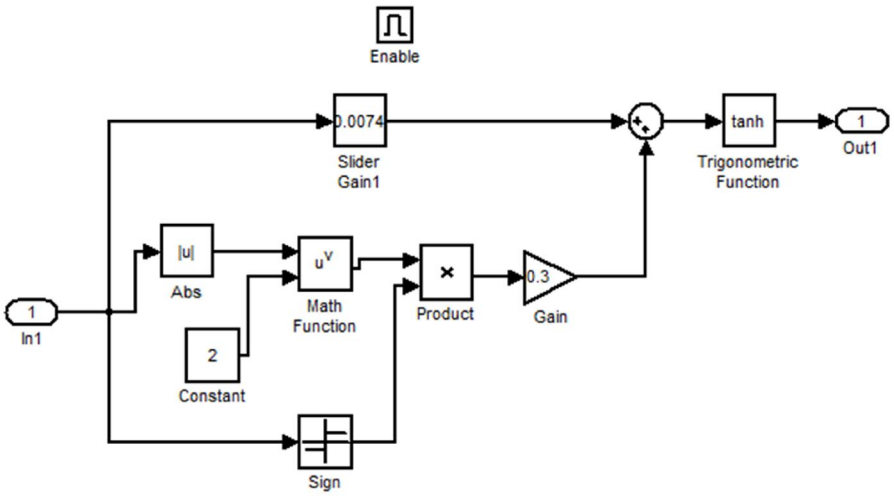


Figura 13: Prototipagem do efeito distortion. Fonte: Elaborado pelo autor.

Já a Figura 14 mostra o *Enabled Subsystem* 1, o qual corresponde ao segundo efeito do arranjo sendo equivalente ao *delay*.

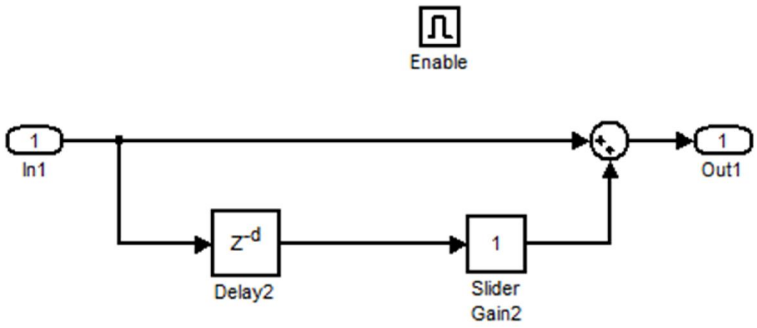


Figura 14: Prototipagem do efeito delay. Fonte: Elaborado pelo autor.

O efeito *delay* possui na saída o sinal original somado ao sinal atrasado no tempo. O bloco denominado *delay*, corresponde a um filtro *comb* do tipo FIR. Os parâmetros do bloco de atraso correspondem ao tempo τ de atraso, o qual é ajustado no próprio bloco de *delay* e também o parâmetro de volume relativo ao atraso, o qual é ajustado no bloco *slider gain*. Na construção deste efeito, utilizou-se no bloco de *delay* a opção de "buffer circular", sendo útil para evitar perdas de amostras no codec.

Seguindo o arranjo de montagem dos efeitos, o *Enabled Subsystem* 2 corresponde ao terceiro efeito sendo ele o *chorus*. A Figura 15 mostra a construção correspondente ao efeito.

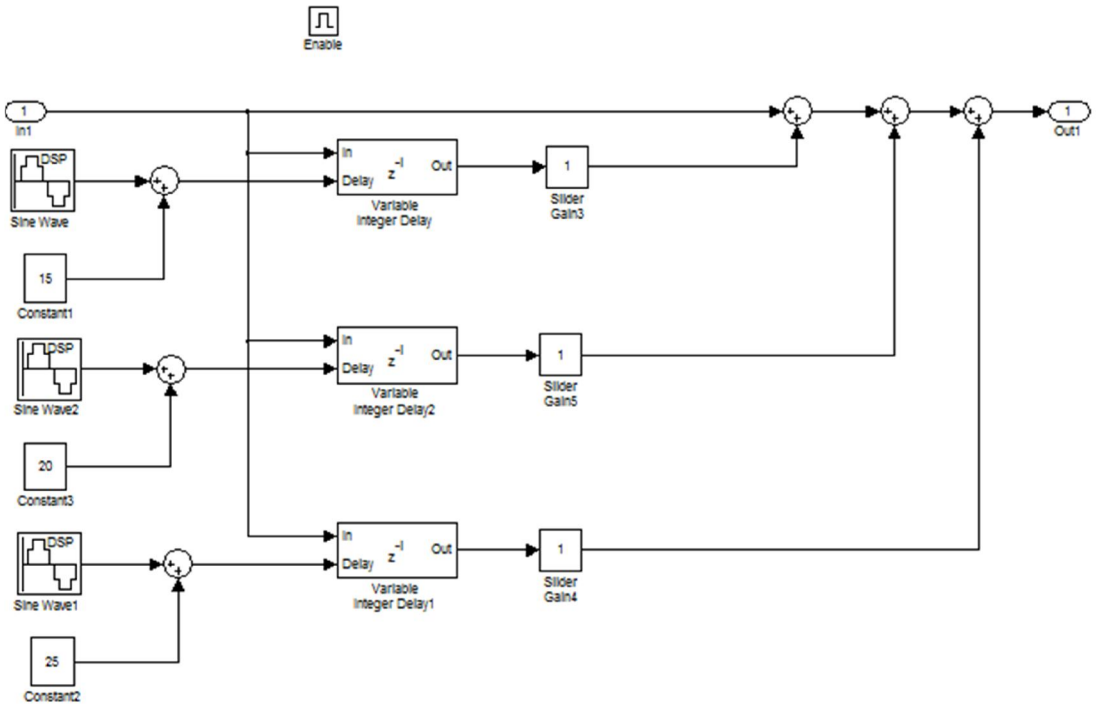


Figura 15: Prototipagem efeito chorus. Fonte: Elaborado pelo autor.

O efeito de *chorus* é constituído por três filtros *comb* do tipo FIR, sendo que eles possuem a opção de variação temporal, e, ao final, ambas as amostras de atrasos variados são somadas. Os atrasos das amostras nestes três filtros estão dentro da faixa de 10 a 25 ms. Com esta faixa de atrasos, será possível ouvir uma rápida repetição do som, o que caracteriza o efeito através da duplicação do sinal. A variação do *delay* de cada filtro é feita através do bloco *Sine Wave*, no entanto a frequência da modulação varia entre os filtros, assim pode-se levar em consideração que a variação do atraso é aproximada a uma variação pseudo-aleatória. A regulagem do efeito é feita através de cada volume na saída do filtro e, também, pode-se provocar uma leve alteração dentre a faixa de tempo do atraso de cada filtro, sendo chamada de regulagem *depth*.

Na sequência do arranjo, o *Enabled Subsystem* 3 corresponde ao quarto efeito sendo ele o *flanging*. A Figura 16 mostra a construção correspondente a este efeito.

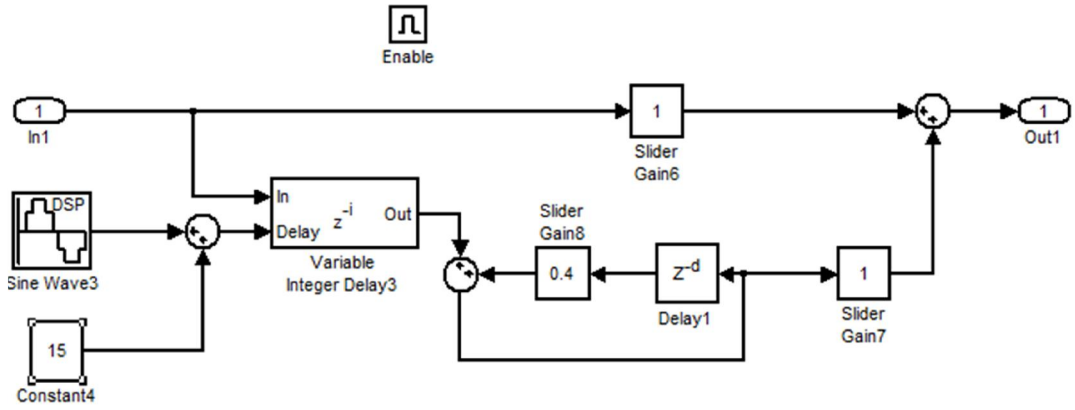


Figura 16: Prototipagem efeito flanging. Fonte: Elaborado pelo autor.

O efeito *flanging* segue basicamente a mesma ideia do efeito *chorus*. A principal diferença está na utilização de filtros FIR com atraso variante no tempo. A variação do atraso no filtro está dentro da faixa de 0 a 15 ms. Por possuir apenas uma única variação neste atraso, a característica da modulação do atraso não é mais aleatória e sim segue um padrão senoidal. Outra diferença é a realimentação na saída do filtro, está por sua vez ocasiona um fundo sonoro de vibração. A regulagem do efeito é feita através do volume na saída da realimentação do filtro e, também, pode-se provocar uma leve

alteração dentre a faixa de tempo do atraso do filtro, sendo chamada de regulagem *depth*. No entanto a regulagem da realimentação da saída do filtro pode ser levada em consideração e este tipo de regulagem é normalmente chamada de *feedback*.

A próxima etapa do arranjo corresponde ao efeito de *reverb*, o qual está confinado dentro do *Enabled Subsystem* 4 correspondendo ao quinto efeito. A Figura 17 mostra como é a montagem deste efeito.

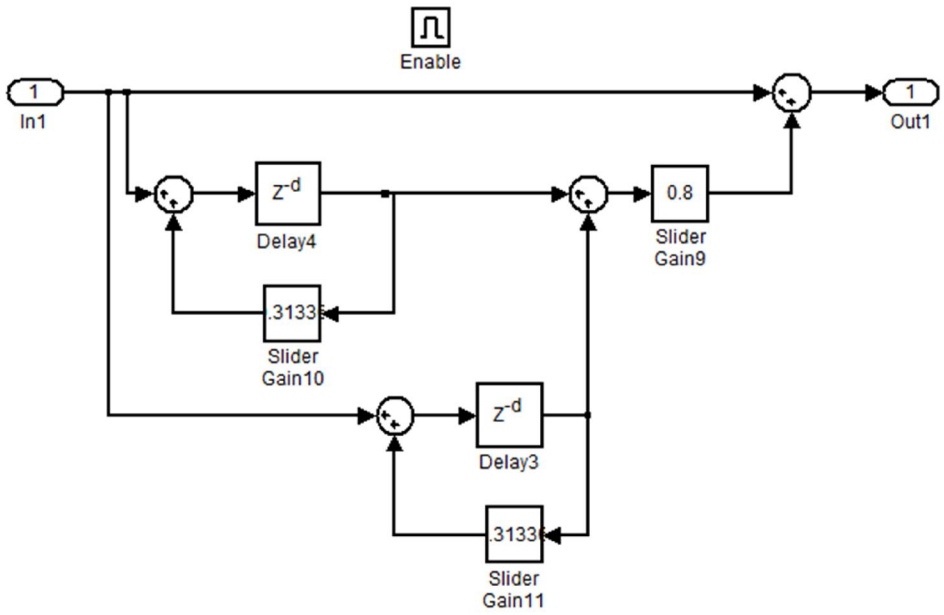


Figura 17: Prototipagem efeito reverb. Fonte: Elaborado pelo autor.

O efeito *reverb* caracteriza-se por possuir filtros do tipo IIR ajustáveis. Por ser um filtro baseado na realimentação, obtém-se na saída muitas amostras atrasadas as quais são somadas ao sinal original, fornecendo assim a sensação de um ambiente com reverberação. Quanto mais filtros possuir a prototipagem mais haverá na saída uma sensação de reverberação. A regulagem deste efeito pode ser feita através do ajuste do volume final de todos os filtros, ou também pode-se variar o tempo de atraso de cada filtro o qual normalmente chama-se *time*.

Por fim, o ultimo efeito do arranjo chama-se *vibe*, o qual está confinado no *Enabled Subsystem* 5. A Figura 18 mostra como é constituído a prototipagem do efeito.

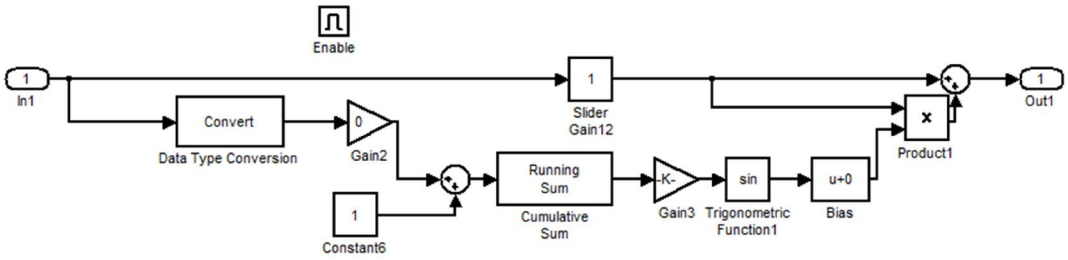


Figura 18: Prototipagem efeito vibe. Fonte: Elaborado pelo autor.

Este efeito possui a característica de modulação por amplitude do sinal. Assim seguindo a ideia da modulação AM, utiliza-se a portadora senoidal para modular o sinal original. A frequência da portadora é duas vezes maior que a frequência de amostragem do sinal original. Nesta prototipagem foi necessário adicionar fatores de correção de unidades, por isso há blocos de conversão dos valores das amostras.

A regulagem deste efeito pode ser feita através do volume do sinal modulado e também pode-se regular a frequência da portadora, sendo está regulagem geralmente chamada de *speed*.

4. IMPLEMENTAÇÃO EM TEMPO REAL ATRAVÉS DO *SYNTHESIS TOOLKIT*

Seguindo o planejamento de desenvolvimento do trabalho, implementaram-se os efeitos descritos anteriormente utilizando-se o *framework* denominado *Synthesis Toolkit* (STK). O STK é um conjunto de bibliotecas de código aberto para o processamento de sinais de áudio escrito na linguagem de programação C++. O STK é compatível com os sistemas operacionais Windows e Unix; ele abstrai complemente as interfaces do sistema operacional, permitindo que o programador trate amostras de áudio em tempo real através de uma API (*Application Programming Interface*) de alto nível.

4.1 SINTETIZAÇÃO DOS EFEITOS COM O STK

Utilizou-se como base para a implementação dos efeitos o programa “effects”, disponível no diretório “projects” do código fonte (versão 4.4.4).

Este programa configura os *codecs* de áudio do PC e exibe uma interface gráfica (GUI), como na Figura 19, a partir da qual se pode selecionar e executar um efeito e regular os seus parâmetros. As alterações nessas escolhas são processadas em tempo real.

A seguir, descreve-se a estrutura do programa “effects”. As contribuições desse trabalho, que consistiram na implementação dos efeitos *Distortion* e *Flanging*, são descritas posteriormente.

Figura 19: Interface gráfica do sintetizador de efeitos. Fonte: Elaborado pelo autor.

4.1.1 O PROGRAMA “EFFECTS”

O programa “effects” é implementado através dos arquivos “effects.cpp”, que contém a função *main*(), e “effects.tcl”, que contém a GUI escrita na linguagem interpretada TCL [8], utilizando o *toolkit* gráfico Tk [9].

A comunicação entre a GUI e o programa principal se dá através de um *socket* (método de comunicação entre processos). Quando o usuário interage com a GUI, esta envia mensagens através do *socket*. Essas mensagens consistem de sequências de caracteres, sendo tratadas pela função “processMessage” do programa principal, que é acionada periodicamente.

O ponto de entrada do arquivo “effects.cpp” (função main()) executa as seguintes tarefas: 1) configura a taxa de amostragem suportada pelo codec, 2) escolhe a interface de leitura das amostras do sinal, ou pelo *line in* ou pelo *mic in* do PC e a interface de saída do sinal processado, ou para o alto-falante ou para o *headphone* do PC, 3) elege a função “tick” para o tratamento da interrupção gerada pela chegada de um *frame* de áudio, e 4) entra em estado de espera, aguardando por essa interrupção.

A função “tick” do arquivo “effects.cpp” é acionada a cada *frame* de áudio recebido pelo *codec*. Esta função é responsável por chavear e acionar cada tipo de efeito selecionado pelo usuário através da GUI. Cada um desses efeitos é implementado através de uma *classe*. Essas classes, que são o âmago do *framework* STK, exibem em comum a existência de um método “tick” (que não deve ser confundida com a função “tick” citada anteriormente), que gera a saída do efeito e “consome” uma *amostra* de áudio de entrada.

Tome-se, por exemplo, o efeito *Chorus*. Esse efeito é implementado por uma classe homônima, fornecida pelo *framework* STK, cujo código se encontra nos arquivos “Chorus.cpp” e “Chorus.h”. Quando esse feito é selecionado através do “botão de rádio” da GUI (Figura 19), a GUI envia uma mensagem via *socket* que atribui o valor inteiro “3” à variável “effectId”. Esta atribuição desencadeia a execução do método “tick” de um objeto da classe “Chorus”. Assim, as amostras do sinal de áudio de entrada serão tratadas por essa classe. Este método “tick” da classe “Chorus” executa então o seguinte código (linhas 158 a 161 do arquivo “Chorus.h”):

```
delayLine_[0].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ * mods_[0].tick() ) );
delayLine_[1].setDelay( baseLength_  * 0.5 *  ( 1.0 - modDepth_ * mods_[1].tick() ) );
*oSamples = effectMix_ * ( delayLine_[0].tick( *iSamples ) - *iSamples ) + *iSamples;
* (oSamples+1) = effectMix_ * ( delayLine_[1].tick( *iSamples ) - *iSamples ) + *iSamples;
```

A execução deste trecho corresponde à obtenção das amostras de saída (“*oSamples”) através da combinação das amostradas de entrada atuais (“*iSamples”) com a saída de uma linha de atraso, implementada pelo objeto “delayLine” da classe “DelayL”. O atraso proporcionado por esta linha é ajustado através do método “setDelay” contido na mesma classe. Este ajuste varia a cada amostra seguindo os valores criados por um gerador de onda senoidal implementado pelo objeto “mods_” da classe “SineWave”. As classes “DelayL” e “SineWave” são fornecidas pelo *framework* STK.

4.1.2 IMPLEMENTAÇÃO DO EFEITO *DISTORTION*

Complementando a funcionalidade do *framework* STK, implementou-se o efeito *distortion*. Para isto, foi criada uma nova classe “Distortion”, derivada da classe “Echo”. O método “tick” da classe “Distortion” implementa o código:

```
lastFrame_[0] = delayLine_.tick(tanh(gain*inputL));
lastFrame_[1] = delayLine_.tick(tanh(gain*inputR));
```

A função *tanh* (tangente hiperbólica) gera uma distorção cuja intensidade é controlada pela variável “gain”, cujo valor é ajustado através da GUI pelo usuário.

4.1.3 IMPLEMENTAÇÃO DO EFEITO *FLANGING*

Complementando o *framework* do STK, implementou-se o efeito *flanging* através do código listado no anexo C (arquivos Flange.cpp e Flange.h). Este efeito está implementado através da classe “Flange”, cujo código deriva da classe “Chorus” contida no *framework* STK.

A função “inline StkFloat Flange :: tick” da classe “Flange” contém o seguinte código, que implementa o modelo descrito na Figura 16:

```
delayLine_[0].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ * mods_[0].tick() ) );
delayLine_[1].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ * mods_[1].tick() ) );
StkFloat Lchan= (delayLine_[0].tick( input ) + 0.4*atrasoiir_[0].nextOut() )/(1.4);
StkFloat Rchan= (delayLine_[1].tick( input ) + 0.4*atrasoiir_[1].nextOut() )/(1.4);
atrasoiir_[0].tick(Lchan);
atrasoiir_[1].tick(Rchan);
lastFrame_[0]= effectMix_* (Lchan - input) + input;
lastFrame_[1]= effectMix_* (Rchan - input) + input;
```

A diferença do efeito Flange para o Chorus, tal como implementado no pacote STK, está na utilização de um filtro IIR em sequência ao filtro FIR variante no tempo que caracteriza o efeito. Este filtro IIR consiste de uma linha de atraso de 1 segundo de duração ligada de modo a produzir uma realimentação.

A regulagem do efeito é feita através do volume na saída da realimentação do filtro e, também, pode-se provocar uma leve alteração dentre a faixa de tempo do atraso do filtro, sendo chamada de regulagem depth. Esta regulagem é feita através da manipulação pelo usuário através da GUI, que produz alterações no método “delayLine_.tick”. A regulagem da frequência de modulação se dá através da configuração do método “baseLength_”, o qual também é manipulado pelo usuário através da GUI.

5. CONCLUSÃO

Neste trabalho construiu-se um simulador em tempo real para os efeitos mais tradicionais de guitarra, os quais são os mais utilizados por músicos que criaram grandes clássicos musicais que, sem o processamento de efeitos, não seriam tão marcantes e referenciados hoje em dia.

O ambiente Simulink do MATLAB permitiu a prototipagem dos efeitos num ambiente de programação de alto nível, mas produziu atrasos largamente perceptíveis, sendo assim inadequado para uma implementação plena. Esses atrasos também são verificados em softwares comerciais de efeitos de guitarra, e só podem ser mitigados com o emprego de computadores de hardware avançado.

Utilizando os protótipos criados no Simulink a partir da modelagem usual de cada efeito, os efeitos foram personalizados seguindo a sensibilidade auditiva do aluno, através de adaptações e variações dos parâmetros dos mesmos.

Os modelos personalizados foram então programados na linguagem C++ utilizando o *framework Synthesis Toolkit* (STK) [6]. Este *framework* proporciona a interação com o sistema operacional Windows de forma bastante eficiente, permitindo que o sintetizador funcionasse em tempo real, com apenas um discreto atraso. O código fonte do STK [11] contém diversos efeitos já implementados. Assim, a contribuição deste trabalho consistiu do desenvolvimento de código para os efeitos não disponíveis originalmente no STK (*distortion* e *flanging*) e da adaptação de uma GUI (interface gráfica) para esse código.

De maneira geral pode-se perceber que atrasos experimentados ao se utilizar o software desenvolvido variam muito de PC para PC. Em alguns computadores menos avançados, para se obter um atraso razoável, foi necessária a instalação de um *driver* dedicado a processamento de áudio chamado ASIO4ALL, que, infelizmente, não suporta uma vasta gama de *hardware*.

Durante o trabalho, considerou-se a opção de desenvolver código para hardware dedicado através de uma ferramenta de integração do ambiente Simulink com o Visual DSP++, a partir da qual é possível gerar automaticamente código na linguagem C tendo como base os modelos em Simulink, o que permitiria a sua execução em hardware dedicado. Esta opção se mostrou inviável, porém, por suportar apenas uma determinada placa (ADSP-BF537 EzKIT Lite), disponível em quantidade reduzida na UFABC, e ser admitir modelos em Simulink bastante restritos. Especificamente, todos os blocos constantes do modelo considerado só podem executar operações aritméticas de ponto fixo, uma vez que o DSP BF537 tem essa característica.

Este projeto, assim, considerou o desenvolvimento de multi-efeitos para guitarra em plataformas distintas, tendo sido possível observar pontos favoráveis e desfavoráveis em cada uma delas.

6. REFERÊNCIAS BIBLIOGRÁFICAS

[1] GM Arts. Disponível em http://www.gmarts.org/index.php.

[2] TOMARAKOS, JOHN E LEDGER, DAN – Using The Low-Cost, High-Performance ADSP-21065L Digital Signal Processor For Digital Audio Applications. Norwood, MA 02062, EUA, Abril, 1998.

[3] História do violão. Disponível em http://www.infoescola.com/musica/historia-do-violao/

[4] Udo Zölzer - DAFX - Digital Audio Effects (Second Edition) ISBN: 978-0-470-66599-2

[5] James Moorer, "About this reverberation business", Computer Music Journal, 1979.

[6] The Synthesis Toolkit. Disponível em https://ccrma.stanford.edu/software/stk/information.html, Outubro de 2013.

[7] MinGW disponível em: http://sourceforge.net/projects/mingw/files/latest/download?source=files, Janeiro 2014

[8] TCL disponível em: http://sourceforge.net/projects/tcl/files/Tcl/8.6.0/tcl860-src.zip/download?use_mirror=ufpr, Janeiro 2014

[9] Tk disponível em: http://sourceforge.net/projects/tcl/files/Tcl/8.6.0/tk860-src.zip/download?use_mirror=ufpr, Janeiro 2014

[10] STK disponível em: http://ccrma.stanford.edu/software/stk/release/stk-4.4.4.tar.gz, Janeiro 2014

[11] Sobre TCL disponível em: http://www.tcl.tk/about/, Janeiro 2014

[12] DSP BF537 disponível em: http://www.analog.com/static/imported-files/data_sheets/ADSP-BF534_BF536_BF537.pdf, Janeiro 2014

A. HARDWARE DEDICADO (DSP)

Os Processadores digitais de sinais (DSPs) possuem, em sua arquitetura interna, um núcleo voltado a solucionar cálculos matemáticos, fazendo com que o mesmo seja indicado a aplicações de processamento de sinais, sejam elas de áudio, de vídeo ou providos de um gerador de funções que tem o intuito de testar alguns algoritmos.

O DSP utilizado para o desenvolvimento dos efeitos anteriormente construídos é da família Blackfin da Analog Device/Intel Micro Signal Architecture (MSA), mais especificamente, o ADSP-BF537.

Essa família de processadores é composta por uma combinação de um núcleo com alto desempenho para processamento de sinais, com um conjunto de periféricos que incluem duas portas UART, na qual proporciona uma interface assíncrona de recebimento ou envio de dados. Uma porta SPI, na qual proporciona uma interface sincronizada de comunicação operando nos modos *máster*, *slave* ou *multimaster*. Duas portas seriais de uso geral (SPORTs). Também possui oito temporizadores de uso geral, os quais podem ser configurados para serem usados como um modulador PWM (*Pulse Width Modulator*), podendo até ser sincronizados por um clock externo ou então para gerar interrupções de eventos. Um relógio de tempo real (RTC), o qual oscila com frequência de 32,768 kHz devido ao cristal externo ao núcleo, sendo que este pode ser configurado para contagens de 60 segundos, 60 minutos, 24 horas e 32768 dias. Um watchdog (“cão de guarda”) timer, o qual pode ser utilizado para forçar o processador a um estado pré-determinado de reset do hardware ou uma interrupção. Uma interface periférica paralela, a qual pode ser ligada diretamente a conversores A/D ou D/A. Uma porta 10/100 Ethernet MAC. E por fim um regulador de voltagem interna de 0,8V a 1,2V que utiliza uma alimentação externa de 2,25V a 3,6V. Este conjunto núcleo e periféricos pode ser visualizado pelo diagrama de blocos da Figura 20, o qual foi retirado do próprio datasheet fornecido pelo fabricante ANALOG DEVICE.

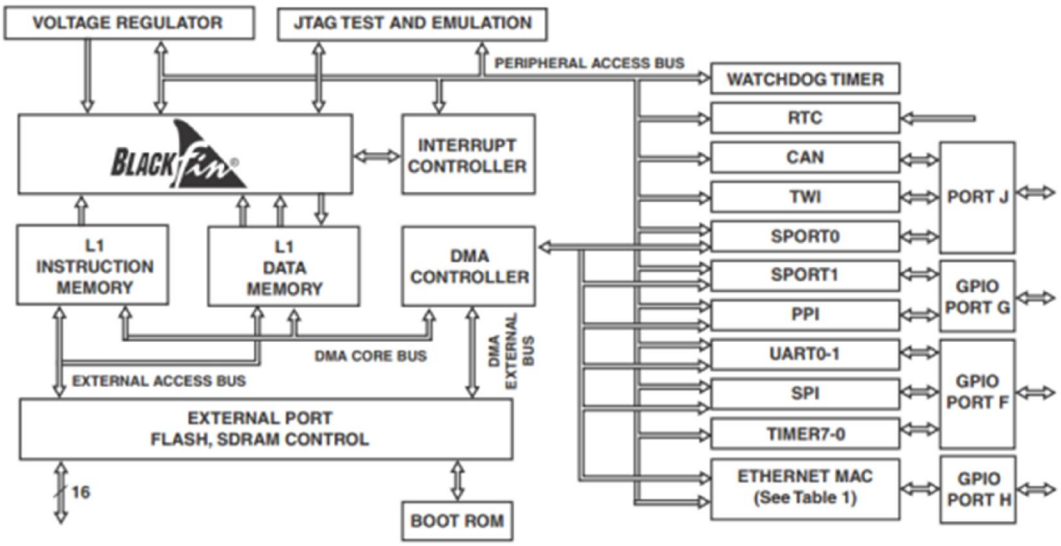


Figura 20: Diagrama funcional do ADSP BF-537. Fonte: Datasheet BF537.

Além dos periféricos, o ADSP-BF537 possui múltiplos e independentes controladores de acesso direto a memória DMA (Direct Memory Access), os quais permitem que haja transferência de dados sem sobre carga do processador. Essencialmente uma transferência DMA copia um bloco de memória de um dispositivo para outro, no entanto esta operação não irá bloquear o processador, o deixando livre para outras tarefas. Os DMAs do BF-537 são capazes de realizar transferências de até 64.000 dados e são formados por um controlador de eventos e duas memórias RAM, uma para armazenamento de dados com 64 kB e a outra para armazenamento de instruções com 80 kB.

Outras características importantes do ADSP-BF537, são:

• Processador de 32 bits.

• Clock do core (núcleo) do DSP de 250 MHz.

• Trabalha com ponto flutuante.

• Core com 2 MACS de 16 bits, 2 ULAs de 40 bits, 4 ULAs de 8 bits (dedicado ao tratamento de vídeo) e 1 deslocador de 40 bits.

• RAM interna de 148 KB (80 kB para instruções, 64 kB para dados e 4 kB para rascunho ou Scratchpad).

• 16 pinos de E/S de propósito geral, todos com recurso de interrupção.

• Interface para controle de até 64 MB de memória externa.

A Figura 21 mostra um diagrama das funções presentes no core do processador Blackfin. Nela é possível observar a unidade aritmética de endereçamento, a ULA de dados, os registradores de uso específico, os registradores de uso geral e a unidade de controle.

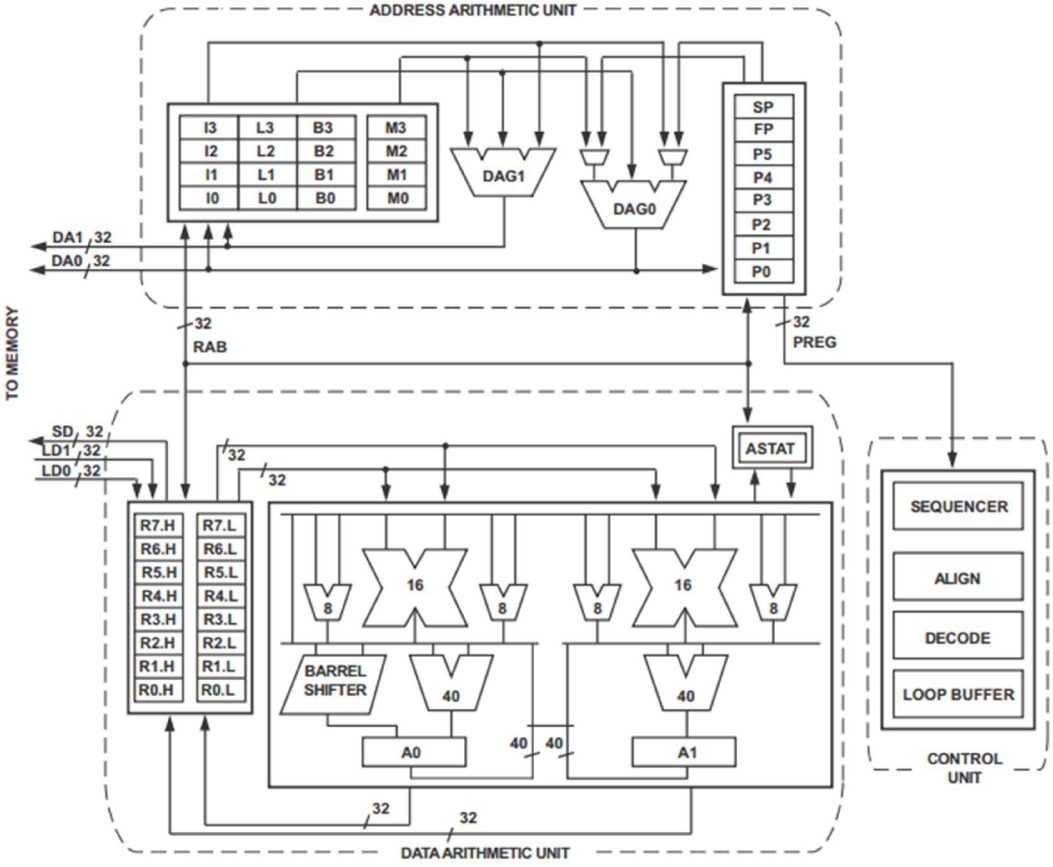


Figura 21: Arquitetura interna do processador Blackfin. Fonte: Datasheet BF537.

O kit de desenvolvimento do ADSP-BF537 EzKIT Lite, é uma placa de circuito impresso que possui além do processador, diversos outros circuitos integrados e também codecs de áudio e de video. Com ele é possível implementar diversas aplicações baseadas nos recursos encontrados no próprio ADSP-BF537, como, por exemplo, aplicações de áudio, vídeo e processamento digital de sinais.

Alguns dos recursos de hardware disponíveis no kit são: interface de comunicação RS-232; memórias RAM e FLASH externas de 32 MB e 2 MB, respectivamente; conector JTAG que permite acesso direto ao núcleo do DSP; extensão de todos os pinos do processador por meio de três conectores localizados na parte inferior da placa; LEDs e chaves do tipo push-button (PB's) para a implementação e teste de aplicações variadas; e conversores A/D de 24 bits e D/A de 10 bits integrados em um mesmo chip; três entradas de vídeo RCA utilizando o CODEC ADV 7183; três saídas de vídeos RCA utilizando o CODEC ADV 7171; quatro entradas e seis saídas de áudio RCA utilizando o CODEC AD 1836. A Figura 22 mostra o diagrama esquemático do EzKIT Lite.

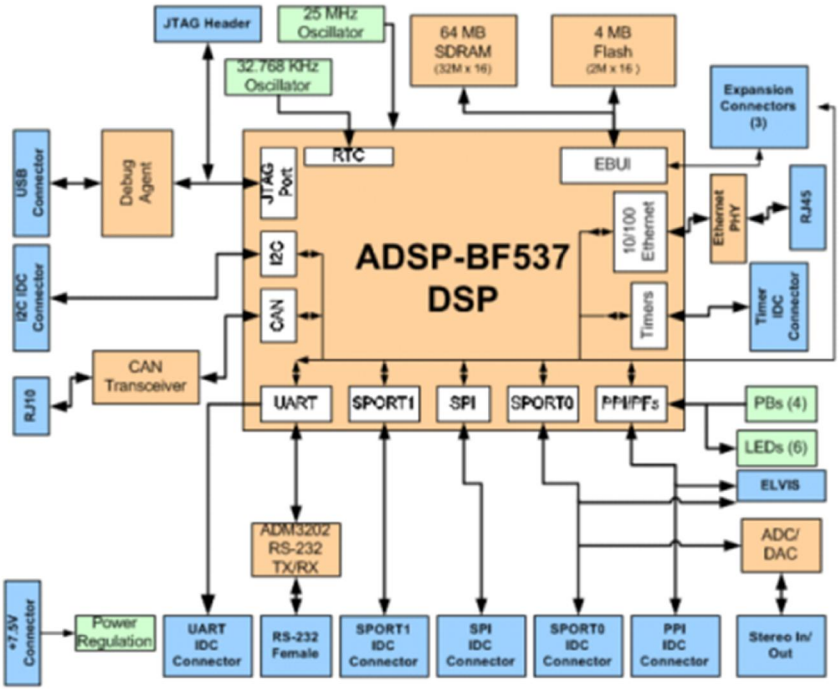


Figura 22: Diagrama de blocos do EzKIT. Fonte: Manual do EzKIT Lite BF537.

O ADSP-BF537 EzKIT Lite possui como ambiente de desenvolvimento o software VisualDSP++, também de propriedade da ANALOG DEVICE. Dentre os recursos que este software possue, pode-se citar sua capacidade de:

• Criar, compilar, montar (assemble) e linkar programas de aplicação escritos em linguagem de programação C, C++ e Assembly.

• Carregar, executar, parar, executar passo a passo e alocar pontos de parada (breakpoints) em programas de aplicação.

• Ler e escrever dados na memória de programa.

• Ler e escrever nos registradores tanto do núcleo do processador como dos periféricos.

• Plotar gráficos com dados provenientes da memória, previamente armazenados em buffers.

A.1 INTERFACE SIMULINK E VISUAL DSP++

Com o intuito de utilizar os mesmos efeitos construídos no Simulink, integrou-se a prototipagem ao Visual DSP++.

Para isto, utilizou-se inicialmente o toolbox *Embedded Coder*, o qual esta disponível no Simulink e é responsável por disponibilizar os blocos funcionais dos conversores ADC (*Analog to Digital Converter*) e DAC (*Digital to Analog Converter*) do respectivo DSP.

Os blocos dos conversores deste toolbox estão na seção *Embedded Targets*, a qual também possui blocos de ferramentas de outros tipos de fabricantes além dos da Analog Device. Devido à utilização do BF537 neste trabalho, seleciona-se na seção *Processors* o subitem referente ao fabricante e modelo do produto, sendo o item *Analog Devices Blackfin* e a biblioteca *ADSP-BF537 EZ-KIT Lite*.

A Figura 23 mostra o caminho no Simulink referente à obtenção dos blocos necessários para a integração das ferramentas utilizadas. No caso deste trabalho, serão utilizados os dois primeiros blocos, específicos para o BF537, correspondendo aos blocos de aquisição do sinal tanto de entrada como de saída.

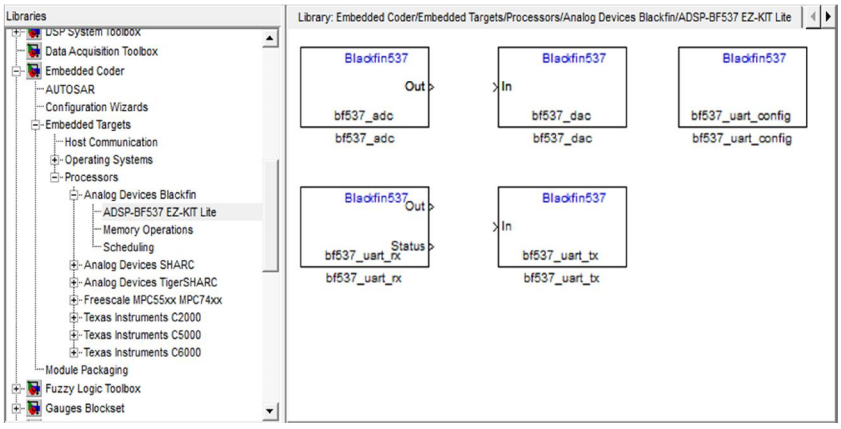


Figura 23: Seleção dos blocos dos conversores analógicos e digitais. Fonte: Elaborado pelo autor.

Agora, ao invés de utilizar os blocos de *From Audio Device* e *To Audio Device* na entrada e saída dos efeitos construídos no Simulink, deve-se substituir pelos blocos bf537_adc e bf537_dac respectivamente. Assim no momento em que será feito a construção e compilação do projeto no Visual DSP++ haverá a garantia de habilitação dos conversores que estão disponíveis e pré-configurados na própria placa do kit.

Por fim, é necessária a utilização do bloco denominado *Target Preferences*. Este bloco será responsável por intermediar, ou seja, fara o papel do driver de comunicação entre os dois ambientes, Simulink e Visual DSP++. Este bloco pertence à seção *Embedded Targets*, a qual pertence ao mesmo toolbox, *Embedded Coder*.

A configuração do bloco Target Preferences é feita da seguinte maneira. Seleciona-se a IDE Analog Device Visual DSP++. Em seguida define-se a placa, no caso será *ADSP-BF537 EZ-KIT Lite*, e assim todos os endereços de memória e o clock do núcleo já estará pré-definido e configurado. Por fim deve-se escolher o suporte a IDE, tendo as opções de simulação de uma placa conectada ao computador ou a opção de debug direto na placa física, a qual deverá estar realmente conectada e detectada por ambos os ambientes de programação.

A Figura 24 ilustra como será feita esta edição do bloco de funções *Target Preferences*. Neste exemplo optou-se por utilizar a IDE suporte como sendo o simulador do kit físico.

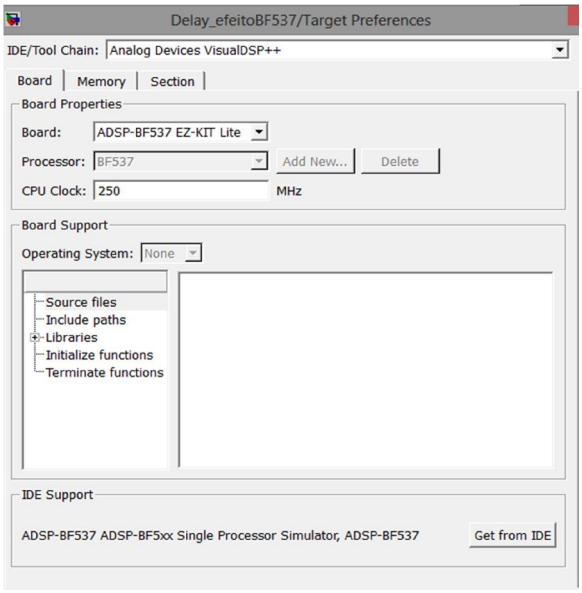


Figura 24: Configuração do *Target Preferences*. Fonte: Elaborado pelo autor.

Feito toda esta etapa de configuração dos blocos de funcionalidades essenciais para a integralização dos ambientes, construiu-se os efeitos anteriormente prototipados com esta nova configuração, para que assim posteriormente possa ser construído de maneira correta o projeto na linguagem C de cada função dos protótipos.

A Figura 25 mostra a nova configuração do efeito *Delay*, com os blocos correspondentes a integralização.

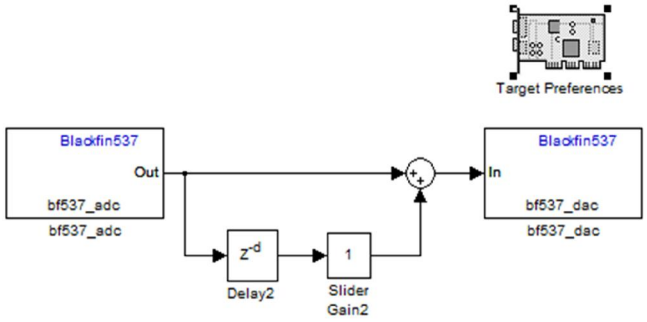


Figura 25: Efeito *Delay* configurado para a integralização. Fonte: Elaborado pelo autor.

Ao termino de todas as mudanças em todos os efeitos, inicia-se o processo de construção do projeto e compilação do código C no Visual DSP++.

Nesta etapa devem-se ter alguns cuidados com relação aos ambientes, pois a construção do projeto não poderá ser executada de forma correta.

Inicialmente na janela de comando do MATLAB, o qual devera estar em modo administrador, executa-se o comando “adivdspsetup”, com o intuito de o Simulink reconhecer a licença de instalação do ambiente Visual DSP++. Sem esta verificação não há comunicação entre os dois softwares.

Feito esta verificação, deve-se ter a certeza de que a seção iniciada no Visual DSP++ corresponde ao mesmo suporte IDE do protótipo feito no Simulink. Independentemente se será utilizado o simulado ou o debug da placa, ambos devem estar configurados e aceitos nos ambientes de integralização. Este pode ser um dos grandes causadores de erros no momento da construção do projeto.

Será necessária a mudança com relação ao valor das entradas de sinal dos blocos funcionais. Devido o DSP BF537 não trabalhar com números de ponto flutuante, todos os dados de entrada e saída deveram ser alterados para

números de ponto fixo de 32 bits. A não alteração acarretara em um erro de construção de projeto e compilação do código no Visual DSP++.

Se não ocorrer nenhum erro, será criado um novo projeto no Visual DSP++, sendo correspondente ao protótipo feito no Simulink. O conteúdo deste projeto terá as pastas “Source Files”, “Linker File” e “Header File”.

A pasta de arquivos fonte terá todo o código C correspondente a cada bloco do efeito construído. Já a pasta de arquivos de cabeçalho terá toda a parte de código referente a tipos de variáveis, estruturas de funções e macros, sendo que estes arquivos serão compartilhados por todos os arquivos fonte. Por fim a pasta de arquivos linker terá códigos referentes aos eventuais logs de banco de dados, ou seja, logs de acesso à memória do kit do DSP.

Se seção escolhida foi a de simulação, será possível verificar se haverá erros de código em uma eventual gravação do projeto no próprio kit. Porem não será possível testar o comportamento dos efeitos fornecendo algum sinal de áudio. No entanto se a seção for à de debug do kit, será possível gravar o projeto e executa-lo direto na placa, assim testes práticos, com sinais de entra e saída audíveis, poderão mostrar qual o real comportamento de toda a implementação da prototipagem. Para se gravar diretamente na placa o código, basta conecta-la ao computador via USB e selecionar a opção “load”.

B. INSTALANDO OS PACOTES STK E TCL

Para iniciar a implementação dos efeitos propostos com o *framework* STK, instalou-se no computador primeiramente o compilador da linguagem de programação C++. O compilador escolhido foi o MinGW (GNU Minimalista para Windows). O MinGW fornece um conjunto de ferramentas de programação o*pen source*, sendo adequado para o desenvolvimento de aplicações nativas do MS-Windows. Ele utiliza uma série de DLLs fornecidas pela própria Microsoft, possui um instalador de linha de comando (msys) e principalmente utiliza a biblioteca de execução em tempo real da Microsoft C. O arquivo executável de instalação do MinGW encontra-se disponível em [7]. Ao executar a instalação deve-se escolher as opções de instalação referentes ao compilador C++ e também as opções referentes ao msys. Após a instalação será necessário à edição da variável de ambiente do sistema operacional chamada “Path”, pois sem esta edição não será possível executar a janela de linhas de comando do compilador. Para isto deve-se acessar o editor de variável de ambiente, o qual se encontra no seguinte caminho: Painel de controle; Sistemas e Segurança; Sistema; Configurações avançadas do sistema; Variáveis de Ambiente; Variáveis do sistema; Path. Esta variável já contem caminhos referentes a outros softwares, assim deve-se adicionar o seguinte caminho: “;C:\MinGW\bin;C:\MinGW\msys\1.0\local\bin”. Este caminho está considerando a instalação do MinGW na unidade “C:\”, caso o mesmo foi instalado em outra unidade, deve-se fazer a alteração referente apenas a unidade. Feito isto será possível utilizar a janela de linha de comando do MinGW. Por último, deverá ser feito uma cópia e renomeação do arquivo “wish86.exe” para “wish.exe”, o qual se encontra no seguinte caminho: “C:\MinGW\msys\1.0\local\bin”. Ambos os arquivos executáveis deverão estar presentes no diretório citado anteriormente. Sem está alteração não será possível futuramente executar corretamente a GUI (Interface gráfica para usuário) criada com o pacote TCL.

Após a instalação do compilador MinGW, deve-se instalar o pacote TCL juntamente com a extensão Tk. Para isso, deve-se fazer o download em [8] e [9] respectivamente, ambos estão disponíveis em sua ultima versão 8.6. O download do pacote virá compactado no formato “zip”, assim deve-se

descompacta-lo em duas pastas próprias, de preferencia no diretório “C:\”. Feito a descompactação, deve-se executar o seguinte comando na janela de linha de comando do MinGW: “cd c:/tcl8.6.0/win”, em que “tcl8.6.0” é o diretório criado ao descompactar o arquivo “tcl860-src.zip”, baixado em [8]. Pode-se perceber que o MinGW tem uma descrição de acesso a diretórios que segue os padrões do Unix. Após acessar o diretório do pacote TCL, deve-se executar os comando “make” e posteriormente, “make install”. Será executado o instalador automático do pacote TCL. Terminado a instalação do mesmo, deverá ser repetido o mesmo processo para a extensão Tk, o qual está compactado no arquivo “tk860-src.zip”, baixado em [9]. Também será feito o mesmo procedimento para o *framework* STK, o qual está compactado no arquivo “stk-4.4.4.rar”, baixado em [10].

Estes passos de instalação deixarão todo o conjunto de ferramentas pronto para o uso e desenvolvimento dos efeitos digitais.

C. CÓDIGO FONTE

Levando em conta os comentários das seções 4.1.2 e 4.1.3, segue em C.1, C.2 e C.3, C.4 os códigos fonte, tanto o ".cpp" como o ".h", criados para os efeitos *Distortion* e *Flanging* respectivamente.

Em C.5 e C.6 estão os códigos alterados dos arquivos "effects.cpp" e "effects.tcl" respectivamente, conforme comentado na seção 4.1.1.

Por fim em C.7 está o código alterado do arquivo "Makefile", responsável por indicar os arquivos a serem compilados.

C.1 "Distortion.cpp"

```
#include "Distortion.h"
#include <cmath>
#include <iostream>

using namespace stk;

const StkFloat Distortion::fixedGain = 0.014;
const StkFloat Distortion::scaleWet = 3;
const StkFloat Distortion::scaleDry = 2;
const StkFloat Distortion::scaleDamp = 0.4;
const StkFloat Distortion::scaleRoom = 0.28;
const StkFloat Distortion::offsetRoom = 0.7;
int Distortion::cDelayLengths[] = {1617, 1557, 1491, 1422, 1356, 1277,
1188, 1116};
int Distortion::aDelayLengths[] = {225, 556, 441, 341};

Distortion::Distortion( void ) {

  delayLine_.setDelay( 44100 );

  lastFrame_.resize( 1, 2, 0.0 );

  Effect::setEffectMix( 0.75 );
  roomSizeMem_ = (0.75 * scaleRoom) + offsetRoom;
  dampMem_ = 0.25 * scaleDamp;
  width_ = 1.0;
  frozenMode_ = false;
  update();

  gain_ = fixedGain;
  g_ = 0.5;
```

```
  double fsScale = Stk::sampleRate() / 44100.0;
  if ( fsScale != 1.0 ) {

    for ( int i = 0; i < nCombs; i++ ) {
      cDelayLengths[i] = (int) floor(fsScale * cDelayLengths[i]);
    }


    for ( int i = 0; i < nAllpasses; i++ ) {
      aDelayLengths[i] = (int) floor(fsScale * aDelayLengths[i]);
    }
  }


  for ( int i = 0; i < nCombs; i++ ) {
    combDelayL_[i].setMaximumDelay( cDelayLengths[i] );
    combDelayL_[i].setDelay( cDelayLengths[i] );
    combDelayR_[i].setMaximumDelay( cDelayLengths[i] + stereoSpread );
    combDelayR_[i].setDelay( cDelayLengths[i] + stereoSpread );
  }


  for (int i = 0; i < nAllpasses; i++) {
    allPassDelayL_[i].setMaximumDelay( aDelayLengths[i] );
    allPassDelayL_[i].setDelay( aDelayLengths[i] );
    allPassDelayR_[i].setMaximumDelay( aDelayLengths[i] + stereoSpread
);
    allPassDelayR_[i].setDelay( aDelayLengths[i] + stereoSpread );
  }
}

Distortion::~Distortion(){
}

void Distortion::setEffectMix( StkFloat mix ){
  Effect::setEffectMix( mix );
  update();
}

void Distortion::setRoomSize( StkFloat roomSize ){
  roomSizeMem_ = (roomSize * scaleRoom) + offsetRoom;
  update();
}

StkFloat Distortion::getRoomSize(){
  return (roomSizeMem_ - offsetRoom) / scaleRoom;
}

void Distortion::setDamping( StkFloat damping ){
  dampMem_ = damping * scaleDamp;
  update();
}
```

```
StkFloat Distortion::getDamping(){
  return dampMem_ / scaleDamp;
}

void Distortion::setWidth( StkFloat width ){
  width_ = width;
  update();
}

StkFloat Distortion::getWidth(){
  return width_;
}

void Distortion::setMode( bool isFrozen ){
  frozenMode_ = isFrozen;
  update();
}

StkFloat Distortion::getMode(){
  return frozenMode_;
}

void Distortion::update(){
  StkFloat wet = scaleWet * effectMix_;
  dry_ = scaleDry * (1.0-effectMix_);


  wet /= (wet + dry_);
  dry_ /= (wet + dry_);

  wet1_ = wet * (width_/2.0 + 0.5);
  wet2_ = wet * (1.0 - width_)/2.0;

  if ( frozenMode_ ) {
    roomSize_ = 1.0;
    damp_ = 0.0;
    gain_ = 0.0;
  }
  else {
    roomSize_ = roomSizeMem_;
    damp_ = dampMem_;
    gain_ = fixedGain;
  }

  for ( int i=0; i<nCombs; i++ ) {
    combLPL_[i].setCoefficients(1.0 - damp_, -damp_);
    combLPR_[i].setCoefficients(1.0 - damp_, -damp_);
  }
}

void Distortion::clear(){
    for ( int i = 0; i < nCombs; i++ ) {
    combDelayL_[i].clear();
```

```
    combDelayR_[i].clear();
  }

  for ( int i = 0; i < nAllpasses; i++ ) {
    allPassDelayL_[i].clear();
    allPassDelayR_[i].clear();
  }

  lastFrame_[0] = 0.0;
  lastFrame_[1] = 0.0;
}

StkFrames& Distortion::tick( StkFrames& frames, unsigned int channel
){

#if defined(_STK_DEBUG_)
  if ( channel >= frames.channels() - 1 ) {
    oStream_ << "Distortion::tick(): channel and StkFrames arguments
are incompatible!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif

  StkFloat *samples = &frames[channel];
  unsigned int hop = frames.channels();
  for ( unsigned int i=0; i<frames.frames(); i++, samples += hop ) {
    *samples = tick( *samples, *(samples+1) );
    *(samples+1) = lastFrame_[1];
  }

  std::cout << "Option 1\n";

  return frames;
}

StkFrames& Distortion::tick( StkFrames& iFrames, StkFrames &oFrames,
unsigned int iChannel, unsigned int oChannel )
{

#if defined(_STK_DEBUG_)
  if ( iChannel >= iFrames.channels() || oChannel >=
oFrames.channels() - 1 ) {
    oStream_ << "Distortion::tick(): channel and StkFrames arguments
are incompatible!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif

  StkFloat *iSamples = &iFrames[iChannel];
  StkFloat *oSamples = &oFrames[oChannel];
  unsigned int iHop = iFrames.channels();
  unsigned int oHop = oFrames.channels();
```

```
  bool stereoInput = ( iFrames.channels() > iChannel+1 ) ? true :
false;
  for ( unsigned int i=0; i<iFrames.frames(); i++, iSamples += iHop,
oSamples += oHop) {
    if ( stereoInput )
      *oSamples = tick( *iSamples, *(iSamples+1) );
    else
      *oSamples = tick( *iSamples );

    *(oSamples+1) = lastFrame_[1];
  }

  std::cout << "Option 2\n";
    return oFrames;
}
```

C.2 “Distortion.h”

```
#ifndef STK_DISTORTION_H
#define STK_DISTORTION_H
#include "Effect.h"
#include "Delay.h"
#include "OnePole.h"
namespace stk {
class Distortion : public Effect
{
 public:
   Distortion();
  ~Distortion();
  void setEffectMix( StkFloat mix );
  void setRoomSize( StkFloat value );
  StkFloat getRoomSize( void );
  void setDamping( StkFloat value );
  StkFloat getDamping( void );
  void setWidth( StkFloat value );
  StkFloat getWidth( void );
  void setMode( bool isFrozen );
  StkFloat getMode( void );
  void clear( void );
  StkFloat lastOut( unsigned int channel = 0 );
  StkFloat tick( StkFloat inputL, StkFloat inputR = 0.0, unsigned int
channel = 0 );
  StkFrames& tick( StkFrames& frames, unsigned int channel = 0 );
  StkFrames& tick( StkFrames& iFrames, StkFrames &oFrames, unsigned
int iChannel = 0, unsigned int oChannel = 0 );
 protected:
  void update( void );
  static const int nCombs = 8;
  static const int nAllpasses = 4;
  static const int stereoSpread = 23;
  static const StkFloat fixedGain;
```

```
  static const StkFloat scaleWet;
  static const StkFloat scaleDry;
  static const StkFloat scaleDamp;
  static const StkFloat scaleRoom;
  static const StkFloat offsetRoom;
  static int cDelayLengths[nCombs];

  static int aDelayLengths[nAllpasses];
  StkFloat g_;
  StkFloat gain_;
  StkFloat roomSizeMem_, roomSize_;
  StkFloat dampMem_, damp_;
  StkFloat wet1_, wet2_;
  StkFloat dry_;
  StkFloat width_;
  bool frozenMode_;
  Delay combDelayL_[nCombs];
  Delay combDelayR_[nCombs];
  OnePole combLPL_[nCombs];
  OnePole combLPR_[nCombs];
  Delay delayLine_;
  Delay allPassDelayL_[nAllpasses];
  Delay allPassDelayR_[nAllpasses];
};

inline StkFloat Distortion :: lastOut( unsigned int channel )
{
#if defined(_STK_DEBUG_)
  if ( channel > 1 ) {
    oStream_ << "Distortion::lastOut(): channel argument must be less
than 2!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif
  return lastFrame_[channel];
}
inline StkFloat Distortion::tick( StkFloat inputL, StkFloat inputR,
unsigned int channel ){
#if defined(_STK_DEBUG_)
  if ( channel > 1 ) {
    oStream_ << "Distortion::tick(): channel argument must be less
than 2!";
    handleError(StkError::FUNCTION_ARGUMENT);
  }
#endif
  if ( !inputR ) {
    inputR = inputL;
  }

  StkFloat fInput = (inputL + inputR) * gain_;
  StkFloat outL = 0.0;
  StkFloat outR = 0.0;
```

```
  for ( int i = 0; i < nCombs; i++ ) {
    StkFloat yn = fInput + (roomSize_ * combLPL_[i].tick(
combDelayL_[i].nextOut() ) );
    combDelayL_[i].tick(yn);
    outL += yn;
    yn = fInput + (roomSize_ * combLPR_[i].tick(
combDelayR_[i].nextOut() ) );
    combDelayR_[i].tick(yn);
    outR += yn;
  }
  for ( int i = 0; i < nAllpasses; i++ ) {
    StkFloat vn_m = allPassDelayL_[i].nextOut();
    StkFloat vn = outL + (g_ * vn_m);
    allPassDelayL_[i].tick(vn);
    outL = -vn + (1.0 + g_)*vn_m;
    vn_m = allPassDelayR_[i].nextOut();
    vn = outR + (g_ * vn_m);
    allPassDelayR_[i].tick(vn);
    outR = -vn + (1.0 + g_)*vn_m;
  }
  lastFrame_[0] = delayLine_.tick(tanh(gain*inputL));
  lastFrame_[1] = delayLine_.tick(tanh(gain*inputR));
  return lastFrame_[channel];
}
}
#endif
```

C.3 “Flange.cpp”

```
#include "Flange.h"
namespace stk {
Flange :: Flange( StkFloat baseDelay ){
  atrasoiir_[0].setMaximumDelay(44100);
  atrasoiir_[0].setDelay(44100);
  atrasoiir_[1].setMaximumDelay(44100);
  atrasoiir_[1].setDelay(44100);
   lastFrame_.resize( 1, 2, 0.0 );
  delayLine_[0].setMaximumDelay( (unsigned long) (baseDelay * 1.414) +
2);
  delayLine_[0].setDelay( baseDelay );
  delayLine_[1].setMaximumDelay( (unsigned long) (baseDelay * 1.414) +
2);
  delayLine_[1].setDelay( baseDelay );
  baseLength_ = baseDelay;

  mods_[0].setFrequency( 0.5 );
  mods_[1].setFrequency( 0.5 );
  modDepth_ = 0.05;
  effectMix_ = 0.5;
  this->clear();
}
```

```
void Flange :: clear( void ){
  atrasoiir_[0].clear();
  atrasoiir_[1].clear();
    delayLine_[0].clear();
  delayLine_[1].clear();
    lastFrame_[0] = 0.0;
  lastFrame_[1] = 0.0;
}
  void Flange :: setModDepth( StkFloat depth ){
  if ( depth < 0.0 || depth > 1.0 ) {
    oStream_ << "Flange::setModDepth(): depth argument must be between
0.0 - 1.0!";
    handleError( StkError::WARNING ); return;
  }
    modDepth_ = depth;
};
void Flange :: setModFrequency( StkFloat frequency ){
  mods_[0].setFrequency( frequency );
  mods_[1].setFrequency( frequency * 1.1111 );
}
}
```

C.4 “Flange.h”

```
#ifndef STK_FLANGE_H
#define STK_FLANGE_H
#include "Effect.h"
#include "DelayL.h"
#include "SineWave.h"

namespace stk {
class Flange : public Effect
{
 public:
 Flange( StkFloat baseDelay = 44100 );
  void clear( void );
  void setModDepth( StkFloat depth );
  void setModFrequency( StkFloat frequency );
  StkFloat lastOut( unsigned int channel = 0 );
  StkFloat tick( StkFloat input, unsigned int channel = 0 );
  StkFrames& tick( StkFrames& frames, unsigned int channel = 0 );
  StkFrames& tick( StkFrames& iFrames, StkFrames &oFrames, unsigned
int iChannel = 0, unsigned int oChannel = 0 );
 protected:
  DelayL atrasoiir_[2];
  DelayL delayLine_[2];
  SineWave mods_[2];
  StkFloat baseLength_;
  StkFloat modDepth_;
};
```

```
inline StkFloat Flange :: lastOut( unsigned int channel ){
#if defined(_STK_DEBUG_)
  if ( channel > 1 ) {
    oStream_ << "Flange::lastOut(): channel argument must be less than
2!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif
  return lastFrame_[channel];
}
inline StkFloat Flange :: tick( StkFloat input, unsigned int channel
){
#if defined(_STK_DEBUG_)
  if ( channel > 1 ) {
    oStream_ << "Flange::tick(): channel argument must be less than
2!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif

  delayLine_[0].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ *
mods_[0].tick() ) );
  delayLine_[1].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ *
mods_[1].tick() ) );

  StkFloat Lchan= (delayLine_[0].tick( input ) +
0.4*atrasoiir_[0].nextOut() )/(1.4);
  StkFloat Rchan= (delayLine_[1].tick( input ) +
0.4*atrasoiir_[1].nextOut() )/(1.4);

  atrasoiir_[0].tick(Lchan);
  atrasoiir_[1].tick(Lchan);

  lastFrame_[0]= effectMix_* (Lchan - input) + input;
  lastFrame_[1]= effectMix_* (Rchan - input) + input;

  return lastFrame_[channel];
}
inline StkFrames& Flange :: tick( StkFrames& frames, unsigned int
channel ){
#if defined(_STK_DEBUG_)
  if ( channel >= frames.channels() - 1 ) {
    oStream_ << "Flange::tick(): channel and StkFrames arguments are
incompatible!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif
  std::cout << "chorus.h - linha 135\n";
  StkFloat *samples = &frames[channel];
  unsigned int hop = frames.channels() - 1;
  for ( unsigned int i=0; i<frames.frames(); i++, samples += hop ) {
    delayLine_[0].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ *
mods_[0].tick() ) );
```

```
    delayLine_[1].setDelay( baseLength_  * 0.5 *  ( 1.0 - modDepth_ *
mods_[1].tick() ) );
    *samples = effectMix_ * ( delayLine_[0].tick( *samples ) -
*samples ) + *samples;
    samples++;
    *samples = effectMix_ * ( delayLine_[1].tick( *samples ) -
*samples ) + *samples;
  }
  lastFrame_[0] = *(samples-hop);
  lastFrame_[1] = *(samples-hop+1);
  return frames;
}
inline StkFrames& Flange :: tick( StkFrames& iFrames, StkFrames&
oFrames, unsigned int iChannel, unsigned int oChannel ){
#if defined(_STK_DEBUG_)
  if ( iChannel >= iFrames.channels() || oChannel >=
oFrames.channels() - 1 ) {
    oStream_ << "Flange::tick(): channel and StkFrames arguments are
incompatible!";
    handleError( StkError::FUNCTION_ARGUMENT );
  }
#endif
  std::cout << "chorus.h - linha 161\n";
  StkFloat *iSamples = &iFrames[iChannel];
  StkFloat *oSamples = &oFrames[oChannel];
  unsigned int iHop = iFrames.channels(), oHop = oFrames.channels();
  for ( unsigned int i=0; i<iFrames.frames(); i++, iSamples += iHop,
oSamples += oHop ) {
    delayLine_[0].setDelay( baseLength_ * 0.707 * ( 1.0 + modDepth_ *
mods_[0].tick() ) );
    delayLine_[1].setDelay( baseLength_  * 0.5 *  ( 1.0 - modDepth_ *
mods_[1].tick() ) );
    *oSamples = effectMix_ * ( delayLine_[0].tick( *iSamples ) -
*iSamples ) + *iSamples;
    *(oSamples+1) = effectMix_ * ( delayLine_[1].tick( *iSamples ) -
*iSamples ) + *iSamples;
  }
  lastFrame_[0] = *(oSamples-oHop);
  lastFrame_[1] = *(oSamples-oHop+1);
  return iFrames;
}
}
#endif
```

C.5 “effects.cpp”

```
#include "Skini.h"
#include "SKINI.msg"
#include "Envelope.h"
#include "PRCRev.h"
#include "JCRev.h"
```

```
#include "NRev.h"
#include "FreeVerb.h"
#include "Distortion.h"
#include "Flange.h"
#include "Echo.h"
#include "PitShift.h"
#include "LentPitShift.h"
#include "Chorus.h"
#include "Messager.h"
#include "RtAudio.h"

#include <signal.h>
#include <cstring>
#include <iostream>
#include <algorithm>
using std::min;

using namespace stk;

void usage(void) {
  std::cout << "\nuseage: effects flags \n";
  std::cout << "    where flag = -s RATE to specify a sample rate,\n";
  std::cout << "    flag = -ip for realtime SKINI input by pipe\n";
  std::cout << "           (won't work under Win95/98),\n";
  std::cout << "    and flag = -is <port> for realtime SKINI input by
socket.\n";
  exit(0);
}
bool done;
static void finish(int ignore){ done = true; }

struct TickData {
  unsigned int effectId;
  PRCRev   prcrev;
  JCRev    jcrev;
  NRev     nrev;
  FreeVerb frev;
  Distortion dist;
  Flange flange;
  Echo     echo;
  PitShift shifter;
  LentPitShift lshifter;
  Chorus   chorus;
  Envelope envelope;
  Messager messager;
  Skini::Message message;
  StkFloat lastSample;
  StkFloat t60;
  int counter;
  bool settling;
  bool haveMessage;

  TickData()
```

```
    : effectId(0), t60(1.0), counter(0),
      settling( false ), haveMessage( false ) {}
};

#define DELTA_CONTROL_TICKS 64

void processMessage( TickData* data ){
  register unsigned int value1 = data->message.intValues[0];
  register StkFloat value2 = data->message.floatValues[1];
  register StkFloat temp = value2 * ONE_OVER_128;

  switch( data->message.type ) {

  case __SK_Exit_:
    if ( data->settling == false ) goto settle;
    done = true;
    return;

  case __SK_NoteOn_:
    if ( value2 == 0.0 )
      data->envelope.setTarget( 0.0 );
    else
      data->envelope.setTarget( 1.0 );
    break;

  case __SK_NoteOff_:
    data->envelope.setTarget( 0.0 );
    break;

  case __SK_ControlChange_:
    switch ( value1 ) {

    case 20: {
      int type = data->message.intValues[1];
      data->effectId = (unsigned int) type;
      break;
    }

    case 22:
      data->echo.setDelay( (unsigned long) (temp * Stk::sampleRate() *
0.95) );
      data->lshifter.setShift( 1.4 * temp + 0.3 );
      data->shifter.setShift( 1.4 * temp + 0.3 );
      data->chorus.setModFrequency( temp );
          data->flange.setModFrequency( temp );
      data->prcrev.setT60( temp * 10.0 );
      data->jcrev.setT60( temp * 10.0 );
      data->nrev.setT60( temp * 10.0 );
      data->frev.setDamping( temp );
      break;

    case 23:
      data->chorus.setModDepth( temp * 0.2 );
```

```
          data->flange.setModDepth( temp * 0.2 );
      data->frev.setRoomSize( temp );
      break;

    case 44:
      data->echo.setEffectMix( temp );
      data->shifter.setEffectMix( temp );
      data->lshifter.setEffectMix( temp );
      data->chorus.setEffectMix( temp );
          data->flange.setEffectMix( temp );
      data->prcrev.setEffectMix( temp );
      data->jcrev.setEffectMix( temp );
      data->nrev.setEffectMix( temp );
      data->frev.setEffectMix( temp );
      break;

    default:
      break;
    }

  }

  data->haveMessage = false;
  return;

 settle:
  data->envelope.setTarget( 0.0 );
  data->counter = (int) (0.3 * data->t60 * Stk::sampleRate());
  data->settling = true;
}
int tick( void *outputBuffer, void *inputBuffer, unsigned int
nBufferFrames,
         double streamTime, RtAudioStreamStatus status, void
*dataPointer )
{
  TickData *data = (TickData *) dataPointer;
  register StkFloat *oSamples = (StkFloat *) outputBuffer, *iSamples =
(StkFloat *) inputBuffer;
  register StkFloat sample;
  Effect *effect;
  int i, counter, nTicks = (int) nBufferFrames;

  while ( nTicks > 0 && !done ) {

    if ( !data->haveMessage ) {
      data->messager.popMessage( data->message );
      if ( data->message.type > 0 ) {
        data->counter = (long) (data->message.time *
Stk::sampleRate());
        data->haveMessage = true;
      }
      else
        data->counter = DELTA_CONTROL_TICKS;
```

```
    }

    counter = min( nTicks, data->counter );
    data->counter -= counter;
    for ( i=0; i<counter; i++ ) {
      if ( data->effectId < 3 ) {
        if ( data->effectId == 0 )
          sample = data->envelope.tick() * data->echo.tick(
*iSamples++ );
        else if ( data->effectId == 1 )
          sample = data->envelope.tick() * data->shifter.tick(
*iSamples++ );
        else
          sample = data->envelope.tick() * data->lshifter.tick(
*iSamples++ );
        *oSamples++ = sample;
        *oSamples++ = sample;
      }
      else {
        if ( data->effectId == 3 ) {
          data->chorus.tick( *iSamples++ );
          effect = (Effect *) &(data->chorus);
        }
        else if ( data->effectId == 4 ) {
          data->prcrev.tick( *iSamples++ );
          effect = (Effect *) &(data->prcrev);
        }
        else if ( data->effectId == 5 ) {
          data->jcrev.tick( *iSamples++ );
          effect = (Effect *) &(data->jcrev);
        }
        else if ( data->effectId == 6 ) {
          data->nrev.tick( *iSamples++ );
          effect = (Effect *) &(data->nrev);
        }
        else if ( data->effectId == 8 ){
        data->dist.tick( *iSamples++ );
        effect = (Effect *) &(data->dist);
        }
            else if ( data->effectId == 9 ){
        data->flange.tick( *iSamples++ );
        effect = (Effect *) &(data->flange);
        }
        else {
          data->frev.tick( *iSamples++ );
          effect = (Effect *) &(data->frev);
        }
        const StkFrames& samples = effect->lastFrame();
        *oSamples++ = data->envelope.tick() * samples[0];
        *oSamples++ = data->envelope.lastOut() * samples[1];
      }
      nTicks--;
    }
```

```
    if ( nTicks == 0 ) break;

    if ( data->haveMessage ) processMessage( data );
  }

  return 0;
}

int main( int argc, char *argv[] )
{
  TickData data;
  RtAudio adac;
  int i;

  if ( argc < 2 || argc > 6 ) usage();

  Stk::setSampleRate( 44100.0 );

  unsigned int port = 2001;
  for ( i=1; i<argc; i++ ) {
    if ( !strcmp( argv[i], "-is" ) ) {
      if ( i+1 < argc && argv[i+1][0] != '-' ) port = atoi(argv[++i]);
      data.messager.startSocketInput( port );
    }
    else if (!strcmp( argv[i], "-ip" ) )
      data.messager.startStdInput();
    else if ( !strcmp( argv[i], "-s" ) && ( i+1 < argc ) &&
argv[i+1][0] != '-')
      Stk::setSampleRate( atoi(argv[++i]) );
    else
      usage();
  }

  RtAudioFormat format = ( sizeof(StkFloat) == 8 ) ? RTAUDIO_FLOAT64 :
RTAUDIO_FLOAT32;
  RtAudio::StreamParameters oparameters, iparameters;
  oparameters.deviceId = adac.getDefaultOutputDevice();
  oparameters.nChannels = 2;
  iparameters.deviceId = 3;
  iparameters.nChannels = 1;
  unsigned int bufferFrames = RT_BUFFER_SIZE;
  try {
    adac.openStream( &oparameters, &iparameters, format, (unsigned
int)Stk::sampleRate(), &bufferFrames, &tick, (void *)&data );
  }
  catch ( RtError& error ) {
    error.printMessage();
    goto cleanup;
  }

  data.envelope.setRate( 0.001 );

        (void) signal( SIGINT, finish );
```

```
  try {
    adac.startStream();
  }
  catch ( RtError &error ) {
    error.printMessage();
    goto cleanup;
  }

  while ( !done ) {
    Stk::sleep( 50 );
  }

  try {
    adac.closeStream();
  }
  catch ( RtError& error ) {
    error.printMessage();
  }

 cleanup:

        std::cout << "\neffects finished ... goodbye.\n\n";
  return 0;
}
```

C.6 “Effects.tcl”

```
set mixlevel 64.0
set effect1 64.0
set effect2 64.0
set effect 0
set outID "stdout"
set commtype "stdout"

wm title . "Multi-Efeitos para Guitarra - TG3"
wm iconname . "Multi-Efeitos para Guitarra"
. config -bg blue

menu .menu -tearoff 0
menu .menu.communication -tearoff 0
.menu add cascade -label "Communication" -menu .menu.communication \
                        -underline 0
.menu.communication add radio -label "Console" -variable commtype \
                        -value "stdout" -command { setComm }
.menu.communication add radio -label "Socket" -variable commtype \
                        -value "socket" -command { setComm }
. configure -menu .menu

label .title -text "Multi-Efeitos para Guitarra - TG3" \
                        -font {Times 14 bold} -background white \
```

```
                        -foreground darkred -relief raised

pack .title -padx 5 -pady 10
pack .title2 -padx 5 -pady 10

frame .noteOn -bg blue

button .noteOn.on -text "Liga Efeitos" -bg grey66 -command { noteOn
64.0 64.0 }
button .noteOn.off -text "Desliga Efeitos" -bg grey66 -command {
noteOff 64.0 127.0 }
button .noteOn.exit -text "Fecha Programa" -bg grey66 -command myExit
pack .noteOn.on -side left -padx 5
pack .noteOn.off -side left -padx 5 -pady 10
pack .noteOn.exit -side left -padx 5 -pady 10

pack .noteOn

frame .left -bg blue

scale .left.effectsmix -from 0 -to 100 -length 400 \
-command {printWhatz "ControlChange    0.0 1 " 44} \
-orient horizontal -label "Volume do Efeito (0% vol - 100% vol)" \
-tickinterval 32 -showvalue true -bg grey66 \
-variable mixlevel

scale .left.effect1 -from 0 -to 100 -length 400 \
-command {printWhatz "ControlChange    0.0 1 " 22} \
-orient horizontal -label "Atraso do Echo" \
-tickinterval 32 -showvalue true -bg grey66 \
-variable effect1

scale .left.effect2 -from 0 -to 100 -length 400 \
-command {printWhatz "ControlChange    0.0 1 " 23} \
-orient horizontal -label "Desabilitado" \
-tickinterval 32 -showvalue true -bg grey66 \
-variable effect2

pack .left.effectsmix -padx 10 -pady 3
pack .left.effect1 -padx 10 -pady 3
pack .left.effect2 -padx 10 -pady 3

pack .left -side left

frame .effectSelect -bg blue
pack .effectSelect -side right -padx 5 -pady 5

radiobutton .effectSelect.echo -text "Echo" -variable effect -relief
flat \
    -value 0 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.shifter -text "Pitch Shift" -variable effect
-relief flat \
```

```
    -value 1 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.lshifter -text "Lent Pitch Shift" -variable
effect -relief flat \
    -value 2 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.chorus -text "Chorus" -variable effect -
relief flat \
    -value 3 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.prcrev -text "PRC Reverb" -variable effect -
relief flat \
    -value 4 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.jcrev -text "JC Reverb" -variable effect -
relief flat \
    -value 5 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.nrev -text "NRev Reverb" -variable effect -
relief flat \
    -value 6 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.freerev -text "FreeVerb" -variable effect -
relief flat \
    -value 7 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}

radiobutton .effectSelect.distortion -text "Distortion" -variable
effect -relief flat \
    -value 8 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}
radiobutton .effectSelect.flange -text "Flange" -variable effect -
relief flat \
    -value 9 -command {changeEffect "ControlChange    0.0 1 " 20
$effect}

pack .effectSelect.echo -pady 2 -padx 5 -side top -anchor w -fill x
pack .effectSelect.shifter -pady 2 -padx 5 -side top -anchor w -fill x
pack .effectSelect.lshifter -pady 2 -padx 5 -side top -anchor w -fill
x
pack .effectSelect.chorus -pady 2 -padx 5 -side top -anchor w -fill x
pack .effectSelect.prcrev -pady 2 -padx 5 -side top -anchor w -fill x
pack .effectSelect.jcrev -pady 2 -padx 5 -side top -anchor w -fill x
pack .effectSelect.nrev -pady 2 -padx 5 -side top -anchor w -fill x
pack .effectSelect.freerev -pady 2 -padx 5 -side top -anchor w -fill x

pack .effectSelect.distortion -pady 2 -padx 5 -side top -anchor w -
fill x
pack .effectSelect.flange -pady 2 -padx 5 -side top -anchor w -fill x

proc myExit {} {
    global outID
    puts $outID [format "NoteOff          0.0 1 64 100" ]
```

```
    flush $outID
    puts $outID [format "ExitProgram"]
    flush $outID
    close $outID
    exit
}

proc noteOn {pitchVal pressVal} {
    global outID
    puts $outID [format "NoteOn           0.0 1 %f %f" $pitchVal
$pressVal]
    flush $outID
}

proc noteOff {pitchVal pressVal} {
    global outID
    puts $outID [format "NoteOff          0.0 1 %f %f" $pitchVal
$pressVal]
    flush $outID
}

proc printWhatz {tag value1 value2 } {
    global outID
    puts $outID [format "%s %i %f" $tag $value1 $value2]
    flush $outID
}

proc changeEffect {tag value1 value2 } {
    global outID
    if ($value2==0) {
        .left.effect1 config -state normal -label "Atraso do Echo"
        .left.effect2 config -state disabled -label "Desabilitado"
      }
    if {$value2>=1 && $value2<=2} {
        .left.effect1 config -state normal -label "Valor Deslocamento
Pitch (centro = sem deslocamento)"
        .left.effect2 config -state disabled -label "Desabilitado"
    }
    if ($value2==3) {
        .left.effect1 config -state normal -label "Frequencia
Modulacao Chorus"
        .left.effect2 config -state normal -label "Profundidade
Modulacao Chorus"
    }
    if {$value2>=4 && $value2<=6} {
        .left.effect1 config -state normal -label "Tempo Decaimento
T60 ( 0 - 10 segundos)"
        .left.effect2 config -state disabled -label "Desabilitado"
    }
    if ($value2==7) {
        .left.effect1 config -state normal -label "Amortecimento
(baixo para alto)"
```

```
        .left.effect2 config -state normal -label "Tamanho Sala(ganho
de realimentacao)"
    }

      if ($value2==8) {
        .left.effect1 config -state normal -label "Distortion"
        .left.effect2 config -state disable -label "Desabilitado"
    }
      if ($value2==9) {
        .left.effect1 config -state normal -label "Frequencia
Modulacao Flange"
        .left.effect2 config -state normal -label "Profundidade
Modulacao Flange"
    }
    puts $outID [format "%s %i %f" $tag $value1 $value2]
    flush $outID
}

bind . <Destroy> +myExit

set d .socketdialog

proc setComm {} {
            global outID
            global commtype
            global d
            if {$commtype == "stdout"} {
                        if { [string compare "stdout" $outID] } {
                                    set i [tk_dialog .dialog "Break
Socket Connection?"\
                                    {You are about to break an existing
socket connection\
                                    ... is this what you want to do?}
"" 0 Cancel OK]
                                    switch $i {
                                                0 {set commtype
"socket"}
                                                1 {close $outID
                                                   set outID "stdout"}
                                    }
                        }
            } elseif { ![string compare "stdout" $outID] } {
                        set sockport 2001
        set sockhost localhost
                        toplevel $d
                        wm title $d "STK Client Socket Connection"
                        wm resizable $d 0 0
                        grab $d
                        label $d.message -text "Specify a socket host
and port number\
                        below (if different than the STK defaults
shown) and then click the \"Connect\"\
```

```
                        button to invoke a socket-client connection
attempt to the STK socket server." \
                                                -background white -font
{Helvetica 10 bold} \
                                                -wraplength 3i -justify
left
                        frame $d.sockhost
                        entry $d.sockhost.entry -width 15
                        label $d.sockhost.text -text "Socket Host:" \
                                                -font {Helvetica 10
bold}
                        frame $d.sockport
                        entry $d.sockport.entry -width 15
                        label $d.sockport.text -text "Socket Port:" \
                                                -font {Helvetica 10
bold}
                        pack $d.message -side top -padx 5 -pady 10
                        pack $d.sockhost.text -side left -padx 1 -pady
2
                        pack $d.sockhost.entry -side right -padx 5 -
pady 2
                        pack $d.sockhost -side top -padx 5 -pady 2
                        pack $d.sockport.text -side left -padx 1 -pady
2
                        pack $d.sockport.entry -side right -padx 5 -
pady 2
                        pack $d.sockport -side top -padx 5 -pady 2
                        $d.sockhost.entry insert 0 $sockhost
                        $d.sockport.entry insert 0 $sockport
                        frame $d.buttons
                        button $d.buttons.cancel -text "Cancel" -bg
grey66 \
                                                -command { set commtype
"stdout"
                                        set outID "stdout"
                                        destroy $d }
                        button $d.buttons.connect -text "Connect" -bg
grey66 \
                                                -command {
                                    set sockhost [$d.sockhost.entry
get]
                                    set sockport [$d.sockport.entry
get]
                                set err [catch {socket $sockhost
$sockport} outID]

                                    if {$err == 0} {
                                                destroy $d
                                    } else {
                                                tk_dialog $d.error
"Socket Error"\
                                                {Error: Unable to make
socket connection.\
```

```
                                                Make sure the STK
socket server is first\
                                                running and that the
port number is correct.}\
                                                "" 0 OK
                        }   }
                        pack $d.buttons.cancel -side left -padx 5 -pady
10
                        pack $d.buttons.connect -side right -padx 5 -
pady 10
                        pack $d.buttons -side bottom -padx 5 -pady 10
            }
}

bind . <Configure> {+ center_the_toplevel %W }
proc center_the_toplevel { w } {
    if { [string equal $w [winfo toplevel $w]] } {

        set width [winfo reqwidth $w]
        set height [winfo reqheight $w]
        set x [expr { ( [winfo vrootwidth  $w] - $width  ) / 2 }]
        set y [expr { ( [winfo vrootheight $w] - $height ) / 2 }]
        wm geometry $w ${width}x${height}+${x}+${y}
        bind $w <Configure> {}
        wm geometry $w ""
    }

    return
}
```

C.7 “Makefile”

```
PROGRAMS =
RM = /bin/rm
SRC_PATH = ../../src
OBJECT_PATH = Release
vpath %.o $(OBJECT_PATH)

OBJECTS     =     Stk.o Generator.o Envelope.o SineWave.o \
                            Filter.o Delay.o DelayL.o OnePole.o \
                            Effect.o Echo.o PitShift.o Chorus.o
LentPitShift.o \
                            PRCRev.o JCRev.o NRev.o FreeVerb.o \
                            FileRead.o WvIn.o FileWvIn.o WaveLoop.o
Skini.o Messager.o

INCLUDE =
ifeq ($(strip $(INCLUDE)), )
      INCLUDE = ../../include
endif
vpath %.h $(INCLUDE)
```

```
CC       = g++
DEFS     =   -DHAVE_GETTIMEOFDAY -D__WINDOWS_DS__ -D__WINDOWS_MM__
DEFS    += -D__LITTLE_ENDIAN__
CFLAGS   = -O3 -Wall
CFLAGS  += -I$(INCLUDE) -I$(INCLUDE)/../src/include
LIBRARY = -lole32 -lwinmm -lWsock32 -ldsound -lwinmm -lm

REALTIME = yes
ifeq ($(REALTIME),yes)
  PROGRAMS += effects
      OBJECTS += RtMidi.o RtAudio.o Thread.o Mutex.o Socket.o
TcpServer.o
endif

RAWWAVES =
ifeq ($(strip $(RAWWAVES)), )
      RAWWAVES = ../../rawwaves/
endif
DEFS    += -DRAWWAVE_PATH=\"$(RAWWAVES)\"

%.o : $(SRC_PATH)/%.cpp
      $(CC) $(CFLAGS) $(DEFS) -c $(<) -o $(OBJECT_PATH)/$@

%.o : ../../src/include/%.cpp
      $(CC) $(CFLAGS) $(DEFS) -c $(<) -o $(OBJECT_PATH)/$@

all : $(PROGRAMS)

effects: effects.cpp Distortion.cpp Flange.cpp $(OBJECTS)
      $(CC) $(LDFLAGS) $(CFLAGS) $(DEFS) -o effects effects.cpp
Distortion.cpp Flange.cpp $(OBJECT_PATH)/*.o $(LIBRARY)

libeffects: effects.cpp Distortion.cpp Flange.cpp
      $(CC) $(LDFLAGS) $(CFLAGS) $(DEFS) -o effects effects.cpp
Distortion.cpp Flange.cpp -L../../src -lstk $(LIBRARY)

$(OBJECTS) : Stk.h

clean :
      $(RM) -f $(OBJECT_PATH)/*.o
      $(RM) -f $(PROGRAMS) *.exe
      $(RM) -fR *~ *.dSYM

distclean: clean
      $(RM) Makefile

strip :
      strip $(PROGRAMS)
```