UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
ESCOLA POLITÉCNICA
DEPARTAMENTO DE ELETRÔNICA E DA COMPUTAÇÃO

SISTEMA DE AQUISIÇÃO E REPRODUÇÃO DE ÁUDIO PARA APLICAÇÕES DE PROCESSAMENTO EM TEMPO REAL

Autor: _____
Felipe Brandão Taveira de Oliveira

Orientador: _____
Prof. Mariane Rembold Petraglia, Ph.D.

Co-Orientador: _____
Prof. Julio César Boscher Torres, D.Sc

Examinador: _____
Prof. Antônio Petraglia, Ph.D.

Examinador: _____
Prof. José Gabriel R.C. Gomes, Ph.D.

DEL

Julho de 2005

Resumo

Ultimamente, muita atenção vem sendo dada às aplicações de telefonia de viva voz e teleconferência. Essa atenção pode ser explicada também, por alguns problemas que com ela surgem, como o cancelamento de eco e problemas com segurança na transmissão.

Para que seja possível realizar o cancelamento, uma codificação de sinal ou qualquer outro tipo de processamento com o sinal de voz, é necessário fazer sua aquisição digital. Considerando o sistema para uma teleconferência como sendo um computador, esse projeto tem como objetivo escrever um algoritmo que seja capaz de ler e escrever um sinal de voz em uma placa de som.

Para que isso se torne prático, em um primeiro momento, o experimento foi feito considerando a entrada como sendo o microfone do computador. Esse sinal de voz que está na entrada será o sinal lido pelo programa da placa. Após essa leitura, não só o cancelamento de eco, mas qualquer outro processamento, como codificação de voz, se faz possível.

O passo seguinte é escrever esse sinal na placa para que seja feita uma conversão analógica dele e envie para a saída, que nesse caso, são os alto-falantes.

Para concluir e testar a funcionalidade do projeto, é feito um simples processamento para codificação do sinal, através da inversão do seu espectro.

Palavras-chave

Cancelamento de Eco

Codificação de Voz

Conversor AD/DA

Placa de Som

“Scrambling”

“Software”

Índice

Lista de Figuras

Sigla e Abreviaturas

CAD – Conversor analógico para digital (*Analog to Digital converter*)

CDA – Conversor digital para analógico (*Digital to Analog Converter*)

DSP – Processamento/Processador digital de sinais (*Digital Signal Processor/Processing*)

PC – Computador Pessoal (*Personal Computer*)

FIR – Resposta ao Impulso Finito

IIR – Resposta ao Impulso Infinito

LMS – Mínimo Quadrado Médio

CPU – Unidade Central de Processamento

BUS – ede

CD – Compact Disk

Capítulo 1 - Introdução

0.1 Introdução

Muitos problemas ocorrem em sistemas de tele-conferências e em telefones viva-voz, devido principalmente à interferência acústica do ruído ambiente e do eco acústico.

A maneira de corrigir estes problemas é fazendo com que o sinal passe por algum processamento. Porém, para isso, é necessário que o sinal seja capturado do meio, e após feitas as modificações, seja novamente introduzido a ele.

No caso de uma teleconferência com dois participantes, uma das possibilidades, seria a introdução de um PC entre eles. Com isso, sempre que um deles se comunicar, o sinal é capturado no sistema (lido da placa de som), processado e reproduzido (escrito na placa de som) na saída para que o outro participante escute um sinal sem ecos ou ruídos indesejáveis. Esse esquema vale para os dois participantes.

Essas leituras e escritas do sinal no computador, como descrito acima, ocorrem através da placa de som. Ela funciona como um “mixer”, já que lê várias entradas de áudio, combina-as e gera o sinal de saída, que será enviado para o alto-falante ou para a entrada de um equipamento de som. As entradas geralmente incluem, entre outras, CD e microfone.

A placa faz essa mixagem independentemente da CPU. Entretanto, ela está conectada ao BUS para receber alimentação da fonte, e para que o micro possa controlar os sinais a serem mixados, ou ler os sinais que uma ou mais entradas estão capturando.

0.2 Hardware Layout

A fim compreender como a leitura e a escrita funcionam em uma placa de som, é importante primeiramente compreender seu *layout*.

Se considerarmos uma placa de som padrão (como as encontradas integradas nas placas mãe ou as utilizadas para jogos), para que o sistema seja capaz de gravar o áudio, é necessário um conversor analógico-digital (CAD),

independente do dispositivo de entrada, que pode ser um microfone, a entrada de linha, ou a saída de um drive de CD ROM.

O sistema pode ser representado como pelo diagrama de blocos da Figura 1, com o fluxo do sinal entre eles indicado por uma seta.

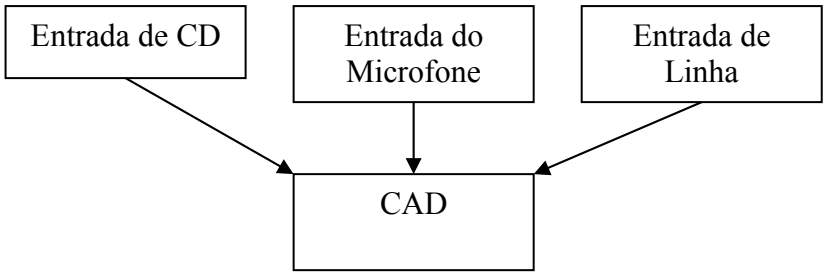

Figura 1 – Diagrama de blocos do sistema de entrada da placa de som

O áudio digital gravado pode ser reproduzido pela placa de som devido à existência de um conversor digital-analógico (CDA), que também é independente do dispositivo de saída a ser usado, podendo este ser um alto-falante ou um fone de ouvido.

Caso a placa tenha componentes capazes de sintetizar áudio em formato MIDI, sua saída também estará ligada aos alto-falantes através do conversor, conforme a figura 2.

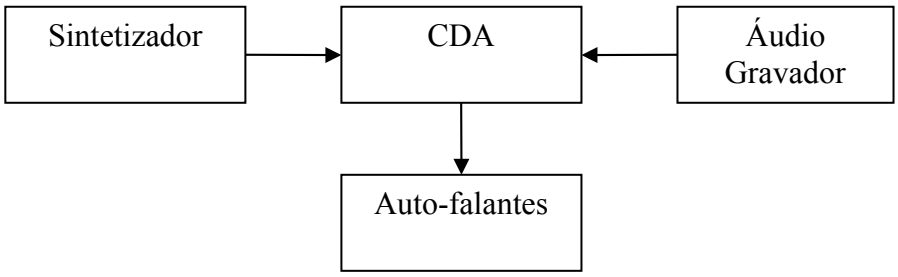

Figura 2 – Diagrama de blocos do sistema da saída da placa de som

Logo, o sistema final de uma placa de som pode ser representado pelo diagrama de blocos da Figura 3, que contém sete componentes e cinco fluxos do sinal correspondentes às setas que os interconectam.

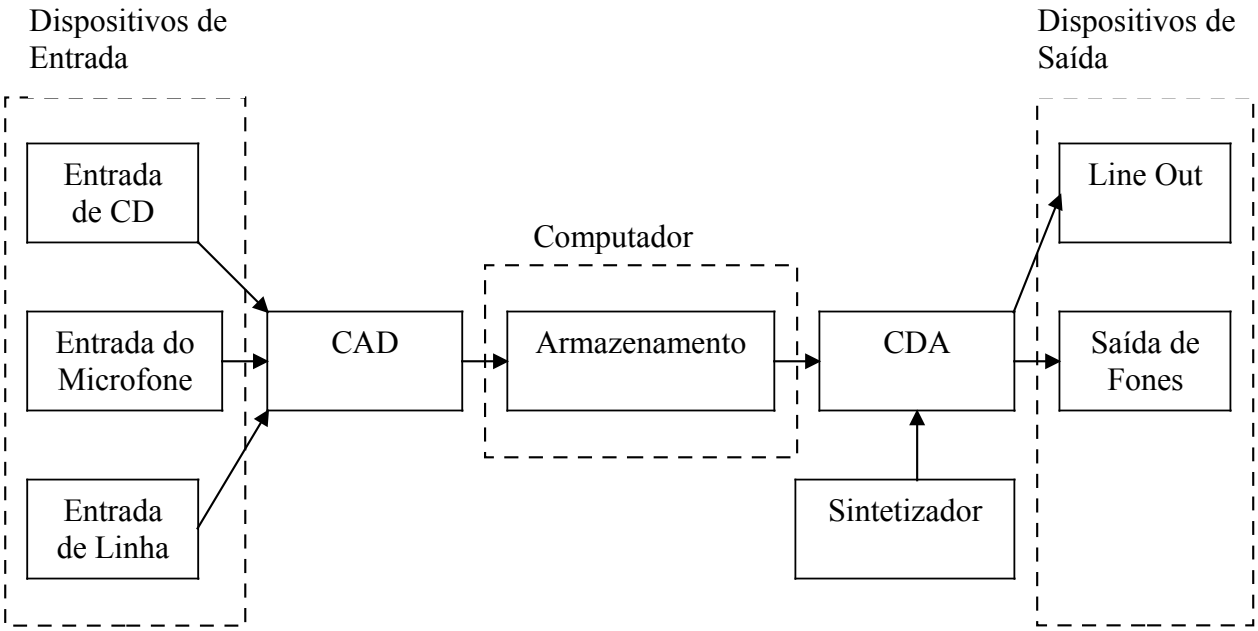

Figura 3 – Diagrama de blocos da placa de som

Cada um desses componentes possui parâmetros individuais de controle. Por exemplo, o Sintetizador tem geralmente seu nível de volume (ganho individual), independente do CD de áudio ou microfone.

Se o usuário estiver tocando um CD de áudio e um arquivo de mídia simultaneamente, pode-se balancear o volume dos componentes à medida que eles são enviados para o alto-falante. Logo, o componente do alto-falante terá seu próprio volume, ou seja, um volume mestre que afeta a mistura total dos componentes conduzidos para ele independente do volume dos fones de ouvido.

Do mesmo modo, o microfone e a entrada de linha também têm níveis separados de volume, também podendo ser balanceados ao gravar simultaneamente.

0.3 Windows Application Programming Interface (API)

0.3.1 O que é API?

O Microsoft Windows Application Programming Interface (API) é um conjunto de funções básicas, fornecidas através de bibliotecas do Windows, que possibilita que os aplicativos tenham acesso aos dispositivos do sistema, sem a necessidade de conhecer o hardware específico.

Suas funções dão acesso aos recursos do computador, tais como memória, arquivos, dispositivos e processos. Um aplicativo usa estas funções para controlar e

monitorar os recursos necessários para terminar o seu trabalho. Um exemplo, são as funções de gerenciamento de memória, utilizadas por aplicativos para alocar ou liberar memória.

As funções de I/O (entrada e saída) fornecem acesso a arquivos, diretórios, e dispositivos de entrada e saída (I/O). Estas funções dão acesso a arquivos e diretórios nos discos e outros dispositivos de armazenamento em um computador específico ou em uma rede. As funções de I/O suportam uma variedade de sistemas.

Os aplicativos podem compartilhar código ou informação com outros aplicativos através de DLLs. As funções de comunicação lêem e escrevem nas portas assim como controlam as modalidades operacionais delas.

0.3.2 Bibliotecas

Para que se possa utilizar o API, duas bibliotecas se fazem necessárias, são elas:

- Windows.h
- Mmsystem.h

0.3.3 Formas de Onda de Áudio

Essa parte que segue, irá mostrar a principal função que utiliza formas de onde e os serviços de áudio auxiliares do API do Windows para adicionar áudio aos aplicativos.

- WaveinOpen

Essa função abre o dispositivo de entrada fornecido para gravação. Para que possa fazer isso, é necessário que ela receba quatro parâmetros:
- Um ponteiro para um buffer, que irá definir se o dispositivo de áudio será aberto;
- Ponteiro que identifica qual o dispositivo de entrada será utilizado;
- Ponteiro para uma estrutura que informa qual o formato de áudio é desejado para gravação;

- WaveinClose

  Essa função fecha o dispositivo de entrada.

Assim como existem as funções que abrem e fecham os dispositivos para gravação, existem também as funções que tem o objetivo de abrir e fechar os dispositivos de saída na reprodução do sinal:

- WaveoutOpen
- WaveoutClose

OBS: Todas as funções contidas neste relatório são funções da biblioteca API do Windows.

Capítulo 1 - Sistema para Leitura e Escrita

Nesse capítulo, será focalizado o problema e os dispositivos já descritos de maneira geral acima, porém agora, com ênfase nas características particulares do projeto desenvolvido.

1.1 Introdução

1.1.1 Sistema - Configurações

Este trabalho visa implementar um sistema que faça a leitura e a escrita em uma placa de som, para que possa ser utilizado por diversos sistemas de processamento de áudio.

Para isso foi utilizado um PC, uma placa de som e seus dispositivos de entrada e saída de áudio.

A escolha da utilização de Computadores Pessoais e não de sistemas mais dedicados, como um DSP, se deve à dificuldade de acesso aos mesmos, já que são dispositivos caros, enquanto que PCs são bastante flexíveis.

O PC e a placa de som utilizados não necessitam de nenhuma característica especial. As únicas exigências necessárias são que a placa de som possua capacidade *full-duplex* (capacidade de gravação e reprodução simultânea), visto que o sistema estará enviando e recebendo informações ao mesmo tempo (teleconferência) e que o PC possua capacidade de processamento suficiente para manter o sistema operacional utilizado e o processamento do sinal.

Como escolha natural a um sistema que exige flexibilidade, a linguagem de programação escolhida para a implementação do sistema foi a C/C++, pois sendo uma linguagem orientada a objetos torna mais fácil a manutenção do *software* no caso de mudanças.

1.1.2 Sistema - Tecnologias

A fim de demonstrar a funcionalidade da leitura e escrita do sinal em uma placa de som configurada por *software,* foi escolhido implementar um *scrambling* de voz, que realiza um simples embaralhamento do espectro do sinal para uma

codificação da voz. Tanto o processo de embaralhamento, quanto outros que podem ser utilizados, serão detalhados posteriormente.

1.1.3 Sistema – Características Técnicas

Nesta seção serão apresentadas algumas características que o sistema irá possuir.

Baseado na principal limitação que existe, a velocidade dos conversores AD/DA, pode-se exemplificar algumas características que o sistema irá apresentar.

As placas de som possuem a característica de ter sua freqüência de amostragem alterada por software, porém, pela maioria dos padrões de sons digitais terem escolhido a freqüência de amostragem de 44100Hz, geralmente as placas de som possuem amostragem mais estável e precisa nesta freqüência.

De forma a facilitar a sincronização e identificação do símbolo recebido, o sinal deve possuir alguns bits por amostra. Para efeitos de cálculos iniciais, consideraremos a utilização de 8 bits por amostra.

Também como forma de simplificar, o sistema será considerado monofônico.

1.2 Visão Geral do Sistema

Como o sistema será igual para qualquer um dos participantes da teleconferência, por simplicidade, será descrito apenas um sentido do sinal.

Para esse projeto, os dispositivos de entrada e saída utilizados são o microfone e os alto-falantes do computador respectivamente.

No primeiro momento, o sinal de voz irá para o microfone para que depois seja feita sua aquisição pela placa de som (será feita uma digitalização do sinal pelo conversor analógico-digital).

Após a aquisição, os dados devem ser lidos da placa para que daí possamos trabalhar com ele da maneira desejada, que no caso será uma codificação do sinal de voz (*scrambling*).

No momento em que o processamento é concluído, os dados são escritos de volta na placa de som, para que possam ser convertidos novamente para sinal analógico e sejam reproduzidos no sistema através dos alto-falantes.

De forma geral, o sistema completo pode ser ilustrado através do diagrama de blocos da Figura 4:

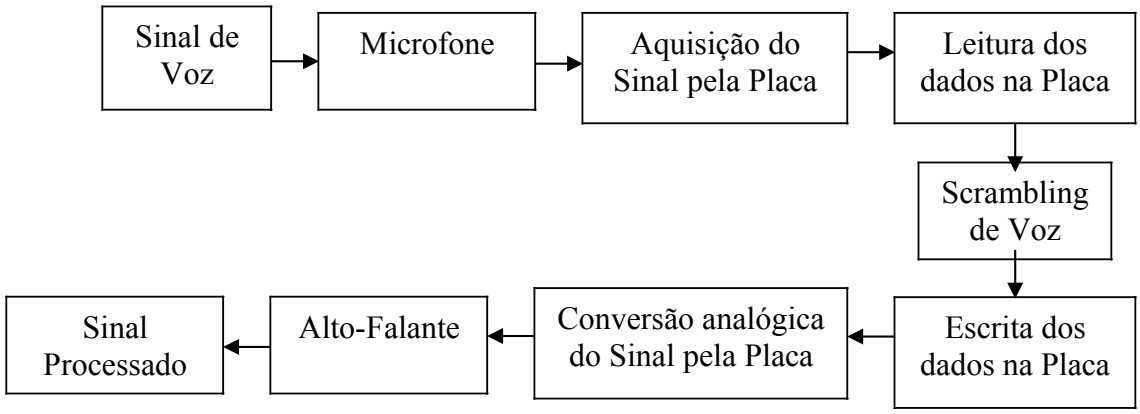

Figura 4 – Diagrama de blocos do sistema completo

1.3 Seleção dos dispositivos de áudio

Diversos caminhos podem ser levados em conta quando se deseja escolher um dispositivo de áudio para a entrada ou a saída, dependendo apenas de quão flexível é o programa.

Tomando como base a leitura, na qual o sinal é gravado, todos os dispositivos de entrada do sistema apresentam um número identificador chamado de ID. Para escolhermos qual dispositivo desejamos utilizar, basta usar o identificador (ID) do dispositivo disponibilizado pelo mapeador de som do Windows (Wave_Mapper), através da função WaveInOpen, como descrito no código C++ listado a seguir:

```
HWAVEIN      inHandle;
```

```
WAVEFORMATEX     waveformat;

// Inicializando o formato da onda para 8-bit, e mono
waveFormat.wFormatTag = WAVE_FORMAT_PCM;
waveFormat.nChannels = 1;
waveFormat.nSamplesPerSec = SamplePerSec;
waveFormat.wBitsPerSample = 8;
waveFormat.nBlockAlign = waveFormat.nChannels * (waveFormat.wBitsPerSample/8);
waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
waveFormat.cbSize = 0;

// Abrindo o dispositivo preferido para entrada de áudio.
err = waveInOpen(&WaveInHandle, WAVE_MAPPER, &waveFormat, (DWORD)err, 0,
CALLBACK_THREAD);
        if (err)
        {
                fileLog.printMsg("Cannot Open Device\n");
                success = false;
        }
```

Se o dispositivo não suportar a escolha desejada de taxa da amostragem e canais, então o Windows abrirá algum outro dispositivo que suporte (supondo haver outro dispositivo disponível).

Naturalmente, se não tiver nenhum dispositivo capaz instalado, a chamada acima retorna um erro.

Da mesma maneira que é feita a escolha do dispositivo de entrada, é feita a escolha do dispositivo de saída na reprodução do sinal, com a diferença que o mapeador nesse caso será usado através da função WaveOutOpen, listada abaixo:

```
HWAVEOUT     outHandle;
```

```
WAVEFORMATEX    waveformat;

// Inicializando o formato da onda para 8-bit, e mono
waveFormat.wFormatTag = WAVE_FORMAT_PCM;
waveFormat.nChannels = 1;
waveFormat.nSamplesPerSec = SamplePerSec;
waveFormat.wBitsPerSample = 8;
waveFormat.nBlockAlign = waveFormat.nChannels * (waveFormat.wBitsPerSample/8);
waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec * waveFormat.nBlockAlign;
waveFormat.cbSize = 0;
// Abrindo o dispositivo preferido para saída de áudio.
if ((err = waveOutOpen(&HWaveOut, WAVE_MAPPER, &waveFormat, (DWORD)WaveOutProc, 0,
CALLBACK_FUNCTION)))
{
        if (verbose) fileLog.printMsg("ERROR: Can't open WAVE Out Device!\n");
                success = false;
}
```

A maneira mais flexível para esta escolha seria apresentar ao usuário todos os nomes na lista de dispositivos de entrada e saída de áudio e deixá-lo escolher o que deseja (ou se o programa suportasse múltiplos dispositivos de saída, poderia deixá-lo selecionar diversos nomes da lista).

Visto que o Windows mantém listas separadas de dispositivos de entrada e saída, existem funções diferentes para identificar os dispositivos em cada lista.

O Windows possui uma função que permite ao usuário determinar o número de dispositivos que suportam entrada ou saída de áudio, estas funções são chamadas waveInGetNumDevs e waveOutGetNumDevs, respectivamente. Lembrando que os identificadores (IDs) começam com 0 e incrementam, se o Windows disser que existem três dispositivos na lista é porque os IDs serão 0, 1, e 2.

Assim como as quantidades de dispositivos, existem funções que podem ser chamadas para buscar informações de um dos dispositivos na lista (por exemplo, seu nome, que taxa de amostragem suporta entre outras). Para isso, basta passar o ID do dispositivo do qual se deseja informação. Os nomes dessas funções são waveInGetDevCaps e waveOutGetDevCaps.

O código a seguir apresenta um exemplo que imprime a lista de dispositivos de entrada de áudio:

```
WAVEINCAPS     wic;
unsigned long  iNumDevs, i;
// Número de dispositivos de entrada de áudio
iNumDevs = waveInGetNumDevs();
//Vai a todos os dispositivos mostrando seus nomes
for (i = 0; i < iNumDevs; i++)
        {
         // Informação sobre o próximo dispositivo
        if (!waveInGetDevCaps(i, &wic, sizeof(WAVEINCAPS)))
        {
                // Mostra seu ID e nome
                printf("Device ID #%u: %s\r\n", i, wic.szPname);
        }
}
```

1.4 Aquisição de Dados

Como as versões mais recentes do Windows não permitem trabalhar por interrupção (não é possível enviar amostra por amostra para a placa de som), por isso devemos fazer uso de *drivers*. A maioria dos *drivers* utilizados atualmente faz uso de buffers de onde obtêm as amostras que serão convertidas de acordo com a freqüência de amostragem escolhida.

O *software* implementado neste trabalho para comunicação com a placa de som irá utilizar no mínimo dois *buffers.* Um dos *buffers* será enviado para a placa de som que irá utilizá-lo para obter as amostras a serem convertidas, enquanto estaremos preenchendo o segundo com as amostras atuais. Quando a placa tiver terminado de converter todos os dados do primeiro *buffer*, o endereço do segundo deve ser passado à placa e o primeiro preenchido, de forma que o ciclo fique contínuo.

Para melhorar a proporção de tempo em que a placa está convertendo os dados e o tempo em que temos para preencher o *buffer,* utilizaremos mais de dois *buffers* no software do conversor.

Logo, as etapas que o software de leitura da placa deve seguir, estão demonstradas a seguir:

- Inicialmente todos o *buffers* se encontram vazios, portanto, ao receber uma amostra, o software deve começar a encher o primeiro buffer.
- Ao receber n amostras onde n é o tamanho do *buffer*, o software deve enfileirar este primeiro *buffer* para que a placa de som possa converter suas amostras em sinal analógico e pular para o próximo. A partir do momento em que esse primeiro é enfileirado, a conversão se inicia.
- Devemos preencher e enfileirar os K *buffers*.
- Caso seja recebida uma amostra durante o intervalo em que todos os *buffers* se encontram cheios, a classe deve retornar um aviso que não conseguiu enviar a amostra com sucesso.
- Quando a placa de som tiver processado todas as amostras do sinal, ela informará a nossa classe que existe um *buffer* liberado. Neste momento devemos verificar se o próximo *buffer* que está na fila encontra-se totalmente cheio, caso não esteja, um erro deve ser gerado, pois, o tempo de processamento do sinal não está sendo suficientemente rápido para processar uma amostra do sinal. Caso não existam problemas, devemos alterar o *status* do *buffer* liberado para vazio e torná-lo disponível para ser preenchido.

Para que possamos fazer a aquisição de dados de uma placa, algumas funções se fazem necessárias:

- WaveFormatex*

Estrutura que especifica o formato dos dados que um dispositivo de entrada de áudio suporta. Esta estrutura também é utilizada para os dispositivos de saída.

Para dados áudio PCM em não mais de dois canais e com amostras de 8-bit ou 16-bit, a estrutura de WAVEFORMATEX para especificar o formato de dados é utilizada.

- WaveHDR*

Estrutura usada como cabeçalho para um bloco de dados de entrada. Assim como a função acima, também é utilizada para os dispositivos de saída.

- WaveinCaps*

  Utilizada para saber das capacidades de um dispositivo de entrada particular.

- WaveinProc*

Essa função é chamada todas as vezes que o buffer for preenchido, com a função de verificar se ele foi totalmente preenchido, chamar e preparar outro buffer, dentre outras.

- WaveinAddBuffer*

  Envia um novo buffer para ser preenchido.

- WaveinReset*

  É essa função que para com a entrada de dados e volta para o ponto inicial.

- WaveinPrepareHeader*

  Prepara o buffer para ser utilizado.

- WaveinStart*

  Inicia a gravação.

Durante os testes do algoritmo de aquisição de dados, foi possível observar que para reduzir o atraso, era necessário diminuir o número de amostras, porém ele deve ser sub múltiplo de 44100Hz.

1.5 Reprodução do Áudio

Assim como o *software* implementado para aquisição de dados da placa, o de escrita (reprodução) também irá utilizar no mínimo 2 *buffers.* Um dos *buffers* será enviado para a placa de som que irá utilizá-lo para obter as amostras a serem convertidas, enquanto o segundo estará sendo esvaziado com as amostras na saída.

Como na leitura, também utilizaremos mais de 2 *buffers* no software do conversor.

Logo, as etapas que o software de escrita na placa são:

- Inicialmente todos o *buffers* se encontram cheios.
- Ao enviar n amostras onde n é o tamanho do *buffer*, o software deve enfileirar este primeiro *buffer* para que a placa de som possa converter suas amostras em sinal digital e pular para o próximo. A partir do momento em que esse primeiro é enfileirado, a conversão se inicia.
- Devemos preencher e enfileirar os K *buffers*.

Todas as funções descritas para a aquisição do sinal também são válidas para a reprodução, com a diferença que suas chamadas são com a palavra out e não in e elas são válidas para os dispositivos de saída:

- WaveFormatex*
- WaveHDR*
- WaveoutCaps*
- Waveoutproc*
- WaveoutAddBuffer*
- WaveoutReset*
- WaveoutPrepareHeader*
- WaveoutStart*

Capítulo 2 - Aplicações

2.1 Cancelamento de Eco

O campo de processamento de Sinais Digitais desenvolveu-se muito rapidamente nas duas últimas décadas em virtude das crescentes técnicas disponíveis para implementação de algoritmos para processamento de sinais.

O problema de cancelamento de eco acústico vem ganhando muita atenção em aplicações de telefonia viva-voz e em sistemas de teleconferência, principalmente por razões de segurança e de conveniência. Para melhorar a qualidade de voz em comunicação móvel em viva-voz, é necessário remover ou reduzir a interferência acústica, tais como, ruído ambiente e eco acústico. O grande problema nesse tipo de sistema é a realimentação do sinal de voz entre auto-falantes e microfone, que gera o eco, o qual precisa ser cancelado. Outro problema freqüente é a presença de ruídos, como no caso de aplicações viva-voz no ambiente de um carro, a conversa a ser transmitida é distorcida por vários ruídos ambientes oriundos, entre outros, do motor, do sistema de exaustão, do barulho dos pneus e do vento.

As dificuldades em se resolver esse problema resultam das seguintes características deste tipo de sistema: a movimentação de objetos ou de pessoas resulta em um sistema de resposta ao impulso variável com o tempo. O acoplamento acústico é formado pelo caminho direto entre o alto-falante e o microfone, e por um grande número de reflexões nos objetos e nas paredes. Os sinais envolvidos são sinais de voz, cujas características espectrais variam muito.

O cancelamento de eco pode ser alcançado se uma réplica do sistema alto-falante/microfone for obtida e implementada em paralelo. Um modelo variante no tempo para este sistema pode ser obtido através de filtragem adaptativa. Entretanto, devido a longa reverberação de uma sala convencional, a resposta ao impulso é longa (milhares de amostras para uma taxa de amostragem de 8KHz), e os sinais que controlam a adaptação são altamente correlacionados e não estacionários. Desta forma, os algoritmos adaptativos convencionais apresentam convergência lenta e alta complexidade computacional para esta aplicação.

Para resolver os problemas mostrados acima, técnicas de processamento em banda única e sub-bandas são propostas para filtros adaptativos. As vantagens do

processamento em sub-bandas são: a complexidade computacional é reduzida de um fator aproximadamente igual ao fator de redução da taxa de amostragem, porque tanto o número de coeficientes quanto a taxa de adaptação podem ser reduzidos em cada sub-banda; a taxa de convergência é aumentada porque a faixa dinâmica espectral é em geral reduzida em cada sub-banda.

2.1.1 Filtragem Adaptativa

Um filtro adaptativo é exigido quando as especificações fixas são desconhecidas ou quando as especificações não podem ser satisfeitas por filtros invariantes no tempo. Os filtros adaptativos são variáveis com o tempo, uma vez que seus parâmetros estão continuamente mudando de maneira a satisfazer uma exigência de desempenho. Assim, a filtragem adaptativa consiste na escolha de estruturas e algoritmos, para que um filtro possa ter seus parâmetros (ou coeficientes) adaptados, de maneira a melhorar o critério de desempenho pré-estabelecido.

Os três itens que caracterizam um filtro adaptativo são:

a) Aplicação

O tipo de aplicação é definido pela escolha dos sinais extraídos do ambiente para servirem de sinal de entrada e de sinal desejado de saída. O número de aplicações bem sucedidas cresceu muito na última década, tais como:

- cancelamento de eco acústico;
- equalização de canais;
- ganho;
- formação de feixes adaptativos;
- cancelamento de ruído;
- sistema de controle.

b) Estrutura do filtro adaptativo

O filtro adaptativo pode ser implementado em um número de diferentes estruturas, e sua escolha pode influenciar a complexidade computacional do processo.

- filtro adaptativo FIR (finite impulse response): a estrutura de filtro adaptativo FIR mais utilizado é o filtro transversal também chamado tapped delay line;
- filtro adaptativo IIR (infinite impulse response): a estrutura mais utilizada de filtros IIR é a forma direta canônica, devido à simplicidade de sua implementação e análise.

c) Algoritmo

O algoritmo é o procedimento usado para ajustar os coeficientes do filtro adaptativo de maneira a melhorar um critério pré-estabelecido. Ele é determinado definindo-se o método de busca (ou algoritmo de minimização), a função objetivo e a natureza do sinal erro.

A escolha do algoritmo determina vários aspectos cruciais do processo adaptativo total, tais como, a existência de soluções sub-ótimas, ótimas polarizadas e complexidade computacional.

O algoritmo LMS (Least-Mean Square) é um importante membro da família de algoritmos gradientes estocásticos e está entre os mais utilizados em filtragem adaptativa por apresentar baixa complexidade computacional e desempenho confiável. Este algoritmo não exige medidas das funções de correlação pertinentes, nem exige inversão de matriz.

A filtragem adaptativa em sub-bandas é principalmente direcionada às aplicações onde se exigem filtros de ordem elevada como, por exemplo, no cancelamento de eco acústico, onde o modulo de sistema desconhecido (eco) apresenta uma resposta longa ao impulso.

2.2 *Embaralhador de Voz*

Aplicações em nível tático de segurança das comunicações em canais de voz têm-se constituído em um tema de interesse proporcionando resistência significativa à violação da informação.

No projeto, o embaralhamento (scrambling) de voz realizado, é uma simples inversão no espectro do sinal. No momento da escrita do sinal na placa de som, todas as amostras ímpares foram multiplicadas por -1 e as amostras pares foram mantidas, conforme segue abaixo:

```
// AudioInOut.cpp
#include "stdafx.h"
#include "audioIn.h"
#include "audioOut.h"
#include "logFile.h"
#include <string>
#include<stdio.h>
#include<math.h>
#include<list>
extern logFile fileLog;
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
	audioOut lineOut(44100,44100);
	list<int> fifo;
	int flag = -1;
	int sample;
	FILE * teste;
	teste =	fopen("teste.txt","rb");
	while(true)
	{
		if ((sample = fgetc(teste))!= EOF)
		{
			flag *= -1;
			sample = sample * flag;
			fifo.push_back (sample);
		}
		// Início da escrita.
		if(!fifo.empty())
		{
			unsigned char byte;
			byte = (int) fifo.front();
			if (lineOut.sendSample(byte))
			{
				fifo.pop_front();
			}
		}
	}
	fclose(teste);
	printf ("Fim da Execução\n");
	exit(0);
	}
```

Capítulo 3 - Conclusões

O algoritmo proposto demonstrou ser capaz implementar um sistema de escrita e leitura em uma placa de som. O *software* implementado foi capaz de gravar em arquivo as amostras de sinal que estavam sendo direcionadas para a entrada, independente do tempo de gravação e após um pequeno processamento do sinal foi possível jogá-lo na saída com o devido processamento funcionando.

Um dos problemas verificados no projeto e como ponto para melhoria em trabalhos futuros é a melhoria no atraso durante a escrita, que mesmo com mudanças tanto na quantidade quanto no tamanho dos buffers, não foi possível uma melhora expressiva, fator importantíssimo para uma teleconferência em tempo real.

Os valores final utilizados foram:

- Número de Buffers de leitura = 2;
- Número de Buffers de escrita = 2;
- Bits por amostra = 8;
- Número de canais = 1;
- Amostras por segundo = 2205.

Para os valores acima, o atraso encontrado foi de aproximadamente 0,5 segundos.

```
// AudioIn.cpp
// É nele que fazemos todas as operações para a escrita na placa de som.

#include "StdAfx.h"
#include ".\audioin.h"
#include "global.h"
#include <stdio.h>

unsigned char           DoneAll;

// Variável usada para indicar se estamos ou não gravando.
BOOL                    InRecord;

HWAVEIN                         WaveInHandle;
BUFFERID bufferId[NUM_BUFFERS];
bool verboseIn;

//---------------------------------
WaveInProc---------------------------------
//   Esta é a linha na qual a Wave In de nível baixo de Windows API passa
as
// mensagens a respeito da gravação audio digital.
//---------------------------------------------------------------------------
-

 MSG        msg;
//   É chamada todas as vezes que o buffer for preenchido, com a função de
// verificar se ele foi totalmente preenchido, chamar e preparar outro
// buffer, dentre outras.
 DWORD WINAPI waveInProc(LPVOID arg)
{
      // Aguarda uma menssagem ser enviada pelo Driver de Áudio
      while (GetMessage(&msg, 0, 0, 0) == 1)
      {
            switch (msg.message)
            {
                  // Caso um buffer tenha sido preenchido
                  case MM_WIM_DATA:
                  {
                        if (InRecord)
                        {
                              // Verifica se o próximo buffer foi
completamente processado
                              BUFFERID * buffId = (BUFFERID *)(((WAVEHDR
*)msg.lParam)->dwUser);
                              BUFFERID * nextBuffId = (BUFFERID *)((WAVEHDR
*)buffId->next)->dwUser;
                              if( nextBuffId ->filled == true)
                              {
                                    if (verboseIn)fileLog.printMsg("Error:
The AudioIn buffer hadn´t been already completly processed\n");
                              }
                              ((BUFFERID *)(((WAVEHDR *)msg.lParam)-
>dwUser))->filled = true;

                              // Aponta para o WAVEHDR que vai ser
processado
```

```
                              // Nesse momento é necessário realinhar o
buffer para a placa poder utiliza-la para outro bloco de audio
                              waveInAddBuffer(WaveInHandle, (WAVEHDR
*)msg.lParam, sizeof(WAVEHDR));
                        }

                        else
                        {
                              ++DoneAll;
                        }
                        // Espera por mais menssagens
                  continue;
                    }

                    case MM_WIM_OPEN:
                    {
                        DoneAll = 0;
                        // Espera por mais menssagens
                  continue;
                    }

                    // Fecha o dispositivo de áudio
                    case MM_WIM_CLOSE:
                    {
                        // Fim da Função
                        break;
                    }
              }
        }
        return(0);
}

//   Essa função tem como principais objetivos, alocar memória, criar
// estruturas, abrir os drivers e ajustar as configurações.
audioIn::audioIn(unsigned int numSamples, unsigned int samplesPerSec )
{
        MMRESULT                                    err;
        success = true;
        InRecord = FALSE;
        currBufIdx = 0;
        verboseIn = true;

        waveInThread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)waveInProc,
0, 0, (LPDWORD)&err);
        if (!waveInThread)
        {
              fileLog.printMsg("Can't create sample recording thread! \n");
              success = false;
        }
        CloseHandle(waveInThread);

        waveFormat.wFormatTag = WAVE_FORMAT_PCM;
        waveFormat.nChannels = 1;
        waveFormat.nSamplesPerSec = samplesPerSec;
        waveFormat.wBitsPerSample = 8;
        waveFormat.nBlockAlign = waveFormat.nChannels *
(waveFormat.wBitsPerSample/8);
        waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec *
waveFormat.nBlockAlign;
        waveFormat.cbSize = 0;
```

```
      //   Abre o dispositivo de WAVE In. Vale ressaltar que se o
dispositivo
      // não suportar a configuração, o Windows automaticamente tenta abrir
      // outro dispositivo que suporte.
      err = waveInOpen(&WaveInHandle, WAVE_MAPPER, &waveFormat, (DWORD)err,
0, CALLBACK_THREAD);
      if (err)
      {
            fileLog.printMsg("Cannot Open Device\n");
            success = false;
      }
      else
      {
            //   Aloca, prepara e alinha 2 buffers que o driver possa usar
para
            // gravar dados de áudio.
            ZeroMemory(&WaveHeader[0], sizeof(WAVEHDR) * NUM_BUFFERS);
            currBuffer = &WaveHeader[0]; /* Point the first Header */
            if (!(WaveHeader[0].lpData = (char *)VirtualAlloc(0,
numSamples*waveFormat.nBlockAlign* NUM_BUFFERS, MEM_COMMIT,
PAGE_READWRITE)))
            {
                  fileLog.printMsg("ERROR: Can't allocate memory for WAVE
buffer!\n");
                  success = false;
            }
            else
            {
                  ZeroMemory(&bufferId, sizeof(BUFFERID)*NUM_BUFFERS);
                  for (int i = 0; i<NUM_BUFFERS; i++)
                  {
                        WaveHeader[i].dwUser= (DWORD_PTR)&bufferId[i];
                        ((BUFFERID *)WaveHeader[i].dwUser)->next =
&WaveHeader[(i+1)%NUM_BUFFERS];
                        ((BUFFERID *)WaveHeader[i].dwUser)->filled = false;
                        WaveHeader[i].dwBufferLength =
numSamples*waveFormat.nBlockAlign ;
                        WaveHeader[i].lpData = WaveHeader[0].lpData +
i*WaveHeader[0].dwBufferLength;
                        if ((err = waveInPrepareHeader(WaveInHandle,
&WaveHeader[i], sizeof(WAVEHDR))))
                        {
                              fileLog.printMsg("Error preparing WAVEHDR(IN)
1! \n");
                              success = false;
                              DoneAll = 1;
                        }
                        if ((err = waveInAddBuffer(WaveInHandle,
&WaveHeader[i], sizeof(WAVEHDR))))
                        {
                              printf("Error queueing WAVEHDR 1! -- %08X\n",
err);
                              success = false;
                              DoneAll = 1;
                        }
                  }
            }
      }
}

// Faz o oposto da função acima
```

```
audioIn::~audioIn(void)
{
	MMRESULT                                    err;
	InRecord = FALSE;

	//   Pára a gravação e informa o driver para desalinhar e retornar
	// todos os WAVEHDRs.
	//   O driver vai retornar qualquer buffer parcialmente preenchido
que
	// estava gravando.
	waveInReset(WaveInHandle);

	while (DoneAll < 2) Sleep(100);

	for( int i = 0; i < NUM_BUFFERS; i++)
	{
	  if ((err = waveInPrepareHeader(WaveInHandle, &WaveHeader[i],
sizeof(WAVEHDR))))
		{
			fileLog.printMsg("Error unpreparing WAVEHDR 2\n");
		}
		// Libera qualquer memória alocada para os buffers
		if (WaveHeader[i].lpData) VirtualFree(WaveHeader[i].lpData, 0,
MEM_RELEASE);
	}
	// Fecha o dispositivo WAVE In
	do
	{
		err = waveInClose(WaveInHandle);
		if (err) fileLog.printMsg("Can't close WAVE In Device!\n");
	} while (err);
}
// Faz a escrita byte a byte
bool audioIn::getCurrSample(unsigned char * sample)
{
	if ((currBufIdx<currBuffer->dwBufferLength)&&(((BUFFERID
*)currBuffer->dwUser)->filled == true))
	{
		*sample = (unsigned char)(currBuffer->lpData[currBufIdx]);
		currBufIdx++;
		return(true);
	}
	//Se acabou de processar este e o próximo não foi enchido pela placa
de som
	else if(currBufIdx>currBuffer->dwBufferLength-1)
	{
		((BUFFERID *)currBuffer->dwUser)->filled = false;
		currBufIdx = 0;
		ZeroMemory(currBuffer->lpData,currBuffer-
>dwBufferLength*sizeof(char));
		currBuffer = (WAVEHDR *)((BUFFERID *)currBuffer->dwUser)->next;
		return(false);
	}
	else
	{
		return(false);
	}
}

// Indica o momento da gravação
bool audioIn::beginCapture(void)
```

```
{
	MMRESULT                                        err;

	InRecord = TRUE;
	if ((err = waveInStart(WaveInHandle)))
	{
		fileLog.printMsg("Error starting sampling!\n");
		return(false);
	}
	else
    {
		#ifdef DEBUG
		fileLog.printMsg("Sampling\n");
		#endif
		return(true);
	}
}
void audioIn::setVerbose(bool value)
{
	verboseIn = value;
}


// AudioOut.cpp
// É nele que fazemos todas as operações para a leitura da placa de som.

#include "StdAfx.h"
#include ".\audioout.h"
#include "global.h"
#include <stdio.h>

HWAVEOUT    HWaveOut;
WAVEHDR *       currBufferOut;
BUFFERID bufferIdO[NUM_BUFS];

bool verbose;

//   É chamada todas as vezes que o buffer for esvaziado, com a função de
// verificar se ele foi totalmente esvaziado, chamar e preparar outro
// buffer, dentre outras.
void CALLBACK WaveOutProc(HWAVEOUT waveOut, UINT uMsg, DWORD dwInstance,
DWORD dwParam1, DWORD dwParam2)
{
	if (uMsg == MM_WOM_DONE)
	{
		if (((BUFFERID *)((WAVEHDR *)((BUFFERID *)(((WAVEHDR
*)dwParam1)->dwUser))->next)->dwUser)->filled == false)
		{
			if (verbose)fileLog.printMsg("Error : The AudiOut buffer
hadn´t been already completly processed\n");
		}
		ZeroMemory(((WAVEHDR *)dwParam1)->lpData,((WAVEHDR *)dwParam1)-
>dwBufferLength*sizeof(char));
		((BUFFERID *)(((WAVEHDR *)dwParam1)->dwUser))->filled = false;
		((WAVEHDR *)dwParam1)->dwFlags &= ~WHDR_DONE;

		waveOutWrite(HWaveOut, (WAVEHDR *)dwParam1, sizeof(WAVEHDR));
	}
}
void audioOut::setVerbose(bool value)
{
```

```
	verbose = value;
}

//   Essa função tem como principais objetivos, desalocar memória, fechar
// estruturas e drivers e ajustar as configurações.
audioOut::audioOut(DWORD numSamples, unsigned int samplesPerSec)
{
	DWORD err;
	success = true;
	firstBuffer = true;
	firstBufferIdx =0;
	currBufIdx = 0;
	verbose = true;

	waveFormat.wFormatTag = WAVE_FORMAT_PCM;
	waveFormat.nChannels = 1;
	waveFormat.nSamplesPerSec = samplesPerSec;
	waveFormat.wBitsPerSample = 8;
	waveFormat.nBlockAlign = waveFormat.nChannels *
(waveFormat.wBitsPerSample/8);
	waveFormat.nAvgBytesPerSec = waveFormat.nSamplesPerSec *
waveFormat.nBlockAlign;
	waveFormat.cbSize = 0;

	// Abre o dispositivo WAVE Out
	if ((err = waveOutOpen(&HWaveOut, WAVE_MAPPER, &waveFormat,
(DWORD)WaveOutProc, 0, CALLBACK_FUNCTION)))
	{
		if (verbose) fileLog.printMsg("ERROR: Can't open WAVE Out
Device!\n");
		success = false;
	}
	WaveBufSize = numSamples*waveFormat.nBlockAlign;

	ZeroMemory(&WaveHeader[0], sizeof(WAVEHDR) * NUM_BUFS);

	if (!(WaveHeader[0].lpData = (char *)VirtualAlloc(0,
WaveBufSize*NUM_BUFS, MEM_COMMIT, PAGE_READWRITE)))
	{
		if (verbose)fileLog.printMsg("ERROR: Can't allocate memory for
WAVE buffer!\n");
		success = false;

	}
	ZeroMemory(&bufferIdO, sizeof(BUFFERID)*NUM_BUFS);
	currBufferOut = &WaveHeader[0];
	for(int i=0;i<NUM_BUFS;i++)
	{
	  WaveHeader[i].lpData = WaveHeader[0].lpData + i*WaveBufSize;
		WaveHeader[i].dwBufferLength = WaveBufSize;

		bufferIdO[i].filled = false;
		bufferIdO[i].next = &WaveHeader[(i+1)%NUM_BUFS];

		// Buffers ficam vazios
		WaveHeader[i].dwUser = (DWORD_PTR)&bufferIdO[i];

		if ((err = waveOutPrepareHeader(HWaveOut, &WaveHeader[i],
sizeof(WAVEHDR))))
		{
			switch (err)
```

```
                  {
                        case MMSYSERR_INVALHANDLE:
                              if (verbose)fileLog.printMsg("0\n");
                        case MMSYSERR_NODRIVER:
                              if (verbose)fileLog.printMsg("1\n");
                        case MMSYSERR_NOMEM:
                              if (verbose)fileLog.printMsg("2\n");
                        default:
                        {
                              char msg[256];
                              sprintf(msg,"ERROR: preparing WAVEHDR Out 1!
-- %08X\n", err);
                              if (verbose)fileLog.printMsg(msg);
                              success = false;
                        }
                  }
            }
      }
}

// Faz o oposto da função acima
audioOut::~audioOut(void)
{
      waveOutReset(HWaveOut);

      for (int i = 0; i<NUM_BUFS;i++)
      {
            waveOutUnprepareHeader(HWaveOut, &WaveHeader[i],
sizeof(WAVEHDR));
      }

      // Desaloca WAVE buffers
      VirtualFree(WaveHeader[0].lpData, WaveBufSize<<NUM_BUFS, MEM_FREE);

      // Fecha o dispositivo WAVE Out
      waveOutClose(HWaveOut);
}

// Faz a leitura byte a byte
bool audioOut::sendSample(unsigned int sample)
{
      if ((currBufIdx<currBufferOut->dwBufferLength)&&(((BUFFERID
*)currBufferOut->dwUser)->filled == false))
      {
            (currBufferOut->lpData[currBufIdx]) = sample;
            currBufIdx++;
            return(true);
      }
      else if(currBufIdx>currBufferOut->dwBufferLength-1)
      {
            currBufIdx = 0;
            ((BUFFERID *)currBufferOut->dwUser)->filled = true;

            if (firstBuffer)
            {
                  firstBufferIdx++;
                  waveOutWrite(HWaveOut, currBufferOut, sizeof(WAVEHDR));
                  if (firstBufferIdx>(NUM_BUFS-1))
                        firstBuffer = false;
            }
            currBufferOut = ((BUFFERID *)currBufferOut->dwUser)->next;
```

```
            return(false);
      }
      else
      {
            return(false);
      }
}

// AudioInOut.cpp
//   É nele em que as funções de leitura e escrita são chamadas. A leitura
// e a escrita são colocadas de maneira alternada, para que seja possível
// que elas funcionem em conjunto, com o sinal que está sendo jogado na
// entrada indo diretamente para a saída.
//   É nele também que é feita a gravação em arquivo do sinal que está
// na entrada.
//   Caso seja necessário fazer algum processamento com o sinal que está
// entrando na placa, é nesse programa que isso é feito.

#include "stdafx.h"
#include "audioIn.h"
#include "audioOut.h"
#include "logFile.h"
#include <string>
#include<stdio.h>
#include<math.h>
#include<list>
extern logFile fileLog;
using namespace std;

//   Programa principal com chamada via linha de comando. Nesse caso
utilizamos
// um console.
int _tmain(int argc, _TCHAR* argv[])
{
      audioOut lineOut(1000,11025);
      audioIn lineIn(1000,11025);
      list<unsigned char> fifo;
      unsigned char sample;
      if (!lineIn.beginCapture())
        exit(1);
      bool sendS =true;
      bool getS = false;
      double dsample = 0;
      long int i =0;
      FILE * teste;
      teste =     fopen("teste.txt","wb");
    while(true)
      {
            // Começa a escrita.
            if (lineIn.getCurrSample(&sample))
                  {
                        fifo.push_back (sample);
                        getS = true;
                        i++;
                        //   Caso seja necessário analisar o sinal de
entrada, ele
                        // está sendo salvo como o arquivo "teste", no
mesmo diretório
                        // do programa.
                        fputc(sample,teste);
```

```
                }
            // Início da leitura.
            if(!fifo.empty())
            {
                unsigned char byte;
                byte = fifo.front();
                if (lineOut.sendSample(byte))
                {
                    fifo.pop_front();
                }
            }
        }
        fclose(teste);
        printf ("Fim da Execução\n");
        exit(0);
}
```

Bibliografia

[1] A. Oppenheim e R. Schafer, *Discrete-Time Signal Processing*. Upper Saddle River, NJ: Prentice-Hall: New Jersey, 1999.

[2] Haykin, S. *Communication Systems*. New York, NY: John Wiley & Sons, Inc, 4ª.ed., 2000.

[3] http://www.msdn.com

[4] http://www.borg.com/~jglatt/tech/lowaud

.