IMPLEMENTAÇÃO DE UM SISTEMA COMPACTO DE CONVERSÃO TEXTO-FALA PARA O PORTUGUÊS

Rodrigo Coura Torres

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Aprovada por:

Prof. José Manoel de Seixas, D.Sc.

Prof. Sergio Lima Netto, Ph.D.

Prof. Marcio Nogueira de Souza, D.Sc.

Prof. Paulo Leo Manassi Osório, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

JUNHO DE 2004

TORRES, RODRIGO COURA

Implementação de um Sistema Compacto de Conversão Texto-Fala para o Português [Rio de Janeiro] 2004

XIII,158 pp 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia Elétrica, 2004)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1.Síntese de Voz 2.Codificação de Voz 3.DSP 4.Aplicações em Tempo Real 5.Sistemas Digitais

I.COPPE/UFRJ II.Título (série)

Agradecimentos

- A Deus, pela saúde e disposição que me permitiram a realização deste trabalho.
- Aos meus pais, Osvani e João Carlos, que com seu amor incondicional, permitiram que me tornasse o homem que sou hoje.
- À minha irmã, Yasmine, por sempre me apoiar e ser um pilar de apoio nos momentos difíceis.
- À minha avó, Olinda, por seu carinho e sabedoria adquirida ao longo dos anos, e ao meu avô Oswaldo, que infelizmente não se encontra mais fisicamente entre nós, mas que tenho certeza que está sempre ao meu lado.
- Aos meus amigos, que sempre estavam dispostos a me ajudar nos momentos de dificuldade.
- Aos meus orientadores, José Manoel de Seixas e Sergio Lima Netto, por toda ajuda, indispensável na elaboração deste trabalho.
- Ao professor Diamantino Freitas e todos os colegas da Universidade do Porto (Portugal), que me acolheram de tal maneira que pude me sentir em casa, mesmo estando tão longe.
- Aos colegas Eduardo Fonseca Brasil, Filipe Castello da Costa Beltrão Diniz e Ranniery da Silva Maia pela ajuda fundamental na codificação da base de dados.
- Ao professor Casé, e aos colegas Paulo Gentil Fernandes e Fabrício Faria Santana do LAPSI (UFRJ), por toda a ajuda na elaboração da placa de circuito impresso desenvolvida para esta tese.
- Aos funcionários e amigos do LPS, pela companhia sempre agradável durante a elaboração deste trabalho.
- Por fim, aos amigos do CGTOTAL, pelas conversas animadas durante o almoço ou tarde da noite no laboratório, permitindo aliviar o estresse do dia a dia.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO DE UM SISTEMA COMPACTO DE CONVERSÃO TEXTO-FALA PARA O PORTUGUÊS

Rodrigo Coura Torres

Junho/2004

Orientadores: José Manoel de Seixas

Sergio Lima Netto

Programa: Engenharia Elétrica

Síntese de voz tem se mostrado, ao longo dos anos, como uma poderosa técnica para resolver a falta de versatilidade dos sistemas atuais baseados em voz gravada. Uma aplicação adicional, bastante útil, para sintetizadores de voz são os sistemas de apoio a indivíduos com necessidades especiais. Atualmente, existem sistemas de síntese de voz implementados em computadores de uso geral, o que gera problemas de mobilidade. Com foco na mobilidade, foi desenvolvido um sistema compacto de síntese de voz para o português, implementado na tecnologia dos processadores digitais de sinais (DSP). Estes processadores possuem características que os tornam bastante atraentes para sistemas portáteis, como baixo consumo, alta performance e custo reduzido.

A técnica de síntese de voz implementada foi a *Time Domain Pitch Synchronous Overlap and Add* (TD-PSOLA). Visando aumentar a naturalidade do texto sintetizado, implementou-se um sistema simples de geração de prosódia baseado nos modelos de Fujisaki. Entretanto, um problema deste tipo de sintetizador é que a base de dados a ser utilizada é bastante grande (aproximadamente 23 Mbytes, para esta aplicação). Para enfrentar este problema, compactou-se a base de dados e codificaram-se as amostras de áudio utilizando a técnica de codificação de voz *Code Excited Linear Prediction* (CELP), reduzindo os requisitos de memória para menos de 200 kbytes, com perdas mínimas na qualidade devido à codificação.

O sistema está apto a operar com qualquer dispositivo que tenha a capacidade de se comunicar através da *interface* RS-232, utilizando o protocolo UART, podendo, assim, ser conectado a um PC, PDA, ou a qualquer dispositivo customizado capaz de produzir e transmitir texto escrito.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

COMPACT IMPLEMENTATION OF A TEXT-TO-SPEECH CONVERSION SYSTEM FOR PORTUGUESE

Rodrigo Coura Torres

June/2004

Advisors: José Manoel de Seixas

Sergio Lima Netto

Department: Electrical Engineering

Voice synthesizing has become a powerful technique for solving the lack of versatility of the today's voice based systems. A useful application for voice synthesizers is in the support for people with special needs. Today, systems are mostly developed for general purpose personal computers (PC), which seems to be a problem for users that want mobility. In order to solve this mobility problem, a Portuguese voice synthesizing system was developed. This synthesizer was implemented on a digital signal processor (DSP) which is an attractive digital solution for portable devices, due to its low cost, low power and excellent performance for digital signal processing applications.

The voice synthesizing technique implemented was the *Time Domain Pitch Synchronous Overlap And Add* (TD-PSOLA). In order to improve the speech naturality, a basic prosody generation system based on the Fujisaki's model was also developed. A major problem with this kind of synthesizer is the large amount of memory needed to store the whole database (approximately 23 Mbytes in its original version for this application). In order to overcome this problem, the database was compressed, and the audio data coded using a *Code Excited Linear Prediction* (CELP) voice coding system. Doing so, the amount of memory needed for the database dropped to about 200 kbytes, with minimal loss in quality due to the codification process.

The system is able to operate with any device that supports an RS-232 interface using the UART serial protocol. So, the developed system is able to operate with a PC, PDA, or any custom made device which is able to generate and send written text.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Atualmente, existem inúmeros sistemas automáticos que interagem com o usuário através da voz, fornecendo as informações necessárias oralmente. Isto é particularmente útil em aplicações onde não existe nenhuma espécie de contato visual com o sistema em operação. Praticamente qualquer serviço de informação por telefone atualmente em funcionamento interage com o usuário através de sistemas automáticos que transmitem a informação desejada oralmente. Serviços de atendimento bancário por telefone são um bom exemplo deste tipo de aplicação, onde o usuário pode receber informações sobre saldos, realizar transferências, entre outras aplicações, apenas digitando as opções no teclado do telefone, e recebendo as informações desejadas oralmente.

Hoje em dia, a maioria dos sistemas que disponibilizam informações oralmente operam através de frases pré-gravadas, de forma que a implementação torna-se relativamente simples, bastando apenas a contratação de um locutor para gravar as frases que serão reproduzidas ao usuário, no momento em que a informação contida numa dada frase for solicitada. Entretanto, sistemas concebidos desta maneira apresentam a deficiência de que só podem fornecer exatamente a informação previamente gravada, o que reduz muito a versatilidade destes, uma vez que não é possível reproduzir informações geradas dinamicamente. Outro problema oriundo desta técnica surge quando se deseja disponibilizar novas informações, pois, neste caso, devemos contratar novamente um locutor (preferencialmente o mesmo, para que todas as frases sejam reproduzidas com a mesma voz) para gravar as novas frases, o que pode tornar difícil a atualização do projeto.

Para resolver problemas como os supracitados, pode-se recorrer à técnicas de síntese de voz. Estas procuram gerar uma base genérica que contenha a informação de todas as unidades fonéticas de um dado idioma, de forma que possamos agrupar estas unidades e, assim, gerar qualquer tipo de frase oral, partindo-se apenas da frase escrita. Com isso, sistemas cujas informações são muito diversificadas, como sistemas de atendimento por telefone, por exemplo, podem gerar a informação na forma de texto escrito, que é muito mais simples, e, através de um sistema de síntese de voz, convertê-lo para texto oral, a fim de ser disponibilizado ao usuário por telefone.

1.1 Motivação

Uma aplicação bastante útil para sintetizadores de voz são os sistemas de apoio a indivíduos com necessidades especiais. Pessoas com deficiências visuais graves, por exemplo, são incapazes de ler informações escritas disponibilizadas pela Internet. Indivíduos com problemas de fala não podem se comunicar pelo telefone. Assim, um sistema que possibilitasse a conversão de texto escrito em fala seria, nestes casos, de extrema importância, principalmente porque, atualmente, existe um enorme volume de informação textual disponível (Internet, jornais, revistas, etc.), bastando apenas converter o formato da mídia, de visual para oral. Entretanto, sistemas de frases pré-gravadas são praticamente inúteis, uma vez que as informações a serem comunicadas pela fonte de informação textual seriam bastante aleatórias. Assim, sintetizadores de voz podem ser utilizados para servir de meio de comunicação oral para estas pessoas, fornecendo soluções eficazes para as necessidades especiais em questão.

Atualmente, existem sistemas de síntese de voz implementados em computadores de uso geral, tipo PC. Entretanto, desta maneira, indivíduos com necessidades especiais podem usufruir destes sistemas apenas quando estão próximos a um computador, limitando bastante a utilização do mesmo. Assim, surge a necessidade de um sistema de síntese de voz que seja portátil, barato e simples de operar, permitindo sua utilização em qualquer situação, o que aumentaria ainda mais a independência deste segmento de usuários.

A tecnologia atual já permite a compactação de sistemas digitais complexos, como telefones celulares, por exemplo. Este nível de compactação é atingido através da exploração das características inerentes ao sinal digital a ser processado, de forma a otimizar o processamento deste. Além disso, o desenvolvedor já conta com dispositivos digitais extremamente velozes, pequenos, de baixo consumo de energia e que podem, ainda, realizar várias tarefas simultaneamente.

1.2 Objetivo

Tendo a mobilidade do usuário em mente, foi desenvolvido um sistema compacto de síntese de voz para o Português, implementado em um processador digital de sinais (DSP). Nas famílias dos DSPs comerciais, encontram-se dispositivos que possuem características tais que os tornam bastante atrativos para sistemas portáteis, como:

- Preço reduzido.
- Baixo consumo de energia.
- Tamanho reduzido.
- Velocidade computacional elevada.
- Dispositivos integrados para execução de funções especiais de interconexão.

Optou-se por utilizar o modelo ADSP-21160M da *Analog Devices*, uma vez que este processador atende todos os requisitos supracitados, com o atrativo adicional de um fácil desenvolvimento de código em alto nível. A técnica de síntese de voz utilizada foi a *Time Domain Pitch Synchronous Overlap and Add* (TD-PSOLA), que utiliza uma base de dados contendo todos os arranjos fonéticos tomados dois a dois, considerando-se os 37 fonemas mais importantes da língua portuguesa, sendo cada arranjo denominado *difone*. Assim, a síntese do texto ocorre quando agrupamos estes difones, formando a frase oral desejada.

A mera concatenação dos difones, embora reproduza o texto de maneira inteligível, não acrescenta nenhuma naturalidade ao texto, reduzindo a compreensão do mesmo. Para resolver tal problema, sistemas geradores de prosódia vêm sendo

desenvolvidos, visando-se aumentar a naturalidade do texto sintetizado. Este trabalho implementou um sistema simples de geração de prosódia baseado nos modelos de Fujisaki.

Um dos problemas deste tipo de sintetizador é que a base de dados a ser utilizada é bastante volumosa, necessitando de um sistema com muita memória de armazenamento, o que poderia resultar no aumento nos custos e nas dimensões do sistema, tornando-o menos acessível e compacto. Para reduzir este problema, decidiu-se codificar a base de dados, utilizando-se a técnica de codificação de voz *Code Excited Linear Prediction* (CELP), que, baseada na modelagem do trato vocal e na determinação da excitação a ser aplicada a esta modelagem, reduz de maneira drástica a quantidade de bits necessária para o armazenamento de toda a base de dados.

Assim, o sistema desenvolvido funciona recebendo pela sua porta serial a seqüência de caracteres correspondente ao texto a ser sintetizado. Em seguida, o sistema realiza a análise prosódica do texto e determina quais são os difones que serão necessários, decodifica-os utilizando o decodificador CELP e, finalmente, utilizando o algoritmo TD-PSOLA, gera a seqüência de amostras de áudio correspondentes ao texto escrito. Esta seqüência é enviada para um conversor digital-analógico, que gerará a forma de onda a ser reproduzida por um alto-falante.

O sistema está apto a operar com qualquer dispositivo que tenha a capacidade de se comunicar através da *interface* RS-232 e que utiliza o protocolo UART, podendo, assim, ser conectado a um PC, um PDA (*Personal Digital Assistant*), ou a qualquer dispositivo customizado capaz de gerar texto escrito e transmiti-lo através desta *interface*.

1.3 Organização do Trabalho

Este documento está organizado da seguinte forma: o capítulo 2 apresenta uma rápida introdução à técnica de síntese de voz. Nele será apresentada a teoria do algoritmo de síntese utilizado neste trabalho. Em seguida, será apresentado o conceito de prosódia, e como este conceito é utilizado para a melhoria da qualidade de sistemas de conversão texto-fala. Também será apresentado, de maneira resumida,

alguns sistemas comerciais de síntese de voz para o Português.

O capítulo 3 apresenta algumas técnicas de codificação utilizadas para o armazenamento das informações do sintetizador. Em seguida, será apresentado o sistema de codificação CELP, que atualmente, é amplamente utilizado para a codificação de voz, de tal forma que se torna interessante utilizá-lo para codificar as informações da base de dados do sintetizador.

Em seguida, o capítulo 4 apresenta o conceito de processadores digitais de sinais (DSPs), apresentando suas características gerais, e, em seguida, detalhando a especificação técnica do DSP escolhido para a implementação deste trabalho. Também serão apresentados os componentes (*hardware* e *software*) utilizados para o desenvolvimento do sistema.

Iniciando a parte prática deste trabalho, o capítulo 5 apresenta a versão original do sistema de síntese utilizando TD-PSOLA, e, em seguida, a análise que resultou em algumas melhorias para este algoritmo, de forma que, após a implementação destas, o algoritmo de síntese estava apto a ser codificado. Também foi realizada, neste capítulo, a análise da base de dados do sintetizador, que resultou na formulação de técnicas para reduzir consideravelmente o tamanho da mesma, de forma que pudesse ser armazenada no DSP.

Dando continuidade a parte prática, o capítulo 6 apresenta todos os passos seguidos para a realização física do projeto. Neste capítulo estão apresentadas as classes que compõem o sistema de síntese, como funcionam, e como interagem entre si. Também serão apresentados os dispositivos de *software* e *hardware* desenvolvidos para permitir a comunicação do sintetizador com o usuário. Por fim, serão apresentadas as análises de desempenho realizadas com o sistema desenvolvido.

Por fim, o capítulo 7 apresenta as conclusões obtidas ao final da elaboração deste trabalho, apresentando as contribuições do mesmo, e finalizando com os possíveis trabalhos futuros.

Capítulo 2

Tecnologias de Síntese de Voz

Para facilitar o acompanhamento da implementação do sistema de síntese de voz, torna-se necessário uma pequena introdução teórica aos sistemas de conversão texto-fala. Neste capítulo, serão abordadas, na seção 2.1, as duas técnicas de síntese atualmente em estudo. Na seção 2.2, será apresentado o algoritmo TD-PSOLA, utilizado na implementação deste trabalho. Em seguida, a seção 2.3 apresentará uma rápida introdução ao estudo da prosódia. Por fim, a seção 2.4 apresentará, resumidamente, alguns sintetizadores comerciais para o Português.

2.1 Técnicas de Síntese de Voz

Intuitivamente, a operação de síntese automática da fala é análoga a controlar-se dinamicamente os músculos do trato vocal e a freqüência de vibração das cordas vocais, de tal forma que o sinal de saída corresponda aos requerimentos de entrada. Sabe-se [1] que transições fonéticas são mais importantes para a compreensão da fala do que porções estacionárias da mesma. Assim, os sistemas automáticos de geração de fala realizam estas transições de duas maneiras:

1. Explicitamente, na forma de uma série de regras que descrevem formalmente a influência de um fonema nos demais.

2. Implicitamente, armazenando estas transições fonéticas e co-articulações em uma base de dados segmentada, e usando estas na forma de unidades acústicas.

Assim, duas classes principais de sintetizadores surgiram destas duas abordagens: os sintetizadores por regra e os sintetizadores por concatenação.

2.1.1 Sintetizadores Baseados em Regras

Sintetizadores baseados em regras são preferidos pelos lingüistas, uma vez que estes sintetizadores utilizam uma abordagem mais analítica para a geração da fala.

Na sua fase preliminar, um grande número de palavras, geralmente contendo seqüências do tipo consoante-vogal-consoante (CVC), são lidas por um orador profissional, digitalizadas e armazenadas. Estas palavras, então, formam o "corpo de fala", que representa as transições e coarticulações a serem estudadas. Graças ao analisador de fala, é atribuída uma forma paramétrica para os dados digitais originalmente aplicados ao analisador de fala. Esta forma separa as contribuições da vibração glotal e do trato vocal, e as apresenta em uma forma mais compacta, mais adequada para estudos posteriores. Em seguida, ocorre o estágio de obtenção de regras, que é geralmente executado por especialistas. Inicialmente, uma inspeção dos dados da fala de todos os oradores é feita para se obter a forma da regra. Os valores dos parâmetros (duração ou freqüência do formante, por exemplo) são, então, ajustados para apenas um orador, embora as regras sejam extraídas de vários usuários, uma vez que é de interesse modelar características articulatórias gerais ao invés de peculiaridades de um único orador. Finalmente, um longo processo de tentativa e erro é empregado para otimizar a qualidade da síntese.

Quando um número satisfatório de regras é determinado, a síntese pode ser finalmente executada. As regras são associadas às entradas fonéticas e um sinal de voz paramétrico é produzido. Por fim, tem-se voz digital sendo gerada pelo sintetizador que implementa o modelo escolhido no estágio de análise.

Conseqüentemente, a qualidade final do segmento de um sintetizador baseado em regras depende dos seguintes fatores:

- Da eficiência das regras internas, ou seja, na capacidade e descrever o corpo paramétrico básico com baixa quantidade de erros.

- Da qualidade do *corpus*, ou seja:

  - Na escolha e na qualidade da gravação das emissões armazenadas no mesmo.
  - Na acurácia do modelo de fala usado na análise de blocos para efetivamente descrever falas de alta qualidade: mesmo que nenhuma regra de busca ou associação seja aplicada, emissões sintéticas podem divergir do original devido a erros de modelagem intrínsecos. Na verdade, um modelo tipicamente restringe o tipo de sinal a ser analisado. Se o modelo for muito simplificado, este pode não ser capaz de representar a riqueza da fala, levando a distorções.
  - No algoritmo utilizado para consertar os valores (variantes no tempo) dos parâmetros de um dado modelo, o que pode ser responsável por erros de modelagem extrínsecos, denominados desta forma devido ao fato de resultarem da inabilidade do algoritmo de análise em produzir uma representação paramétrica correta, e não da incapacidade do modelo propriamente dito.

- Das melhorias obtidas na fase de tentativa e erro, que podem, eventualmente, compensar falhas geradas pela gravação ou pelos algoritmos.

Por motivos históricos e de praticidade, sintetizadores baseados em regras sempre aparecem na forma de sintetizadores por formantes. Estes descrevem a fala como uma evolução dinâmica de até 60 parâmetros, a maioria relacionados a freqüências de formantes e anti-formantes, e larguras de banda, aliadas com formas de ondas glotais. Conseqüentemente, estes parâmetros podem ser obtidos praticamente sem erros de modelagem intrínsecos. Por outro lado, um grande número de parâmetros pode complicar a fase de análise e tende a gerar erros extrínsecos. Além disso, freqüências de formantes e larguras de banda são inerentemente difíceis de estimar a partir de dados de voz. A necessidade intensiva de tentativas e erros, até a redução máxima de erros extrínsecos, faz com que se leve muito tempo (anos, normalmente) para a elaboração de um sintetizador baseado nesta abordagem. E mesmo assim, a implementação final apresenta ruídos originados das próprias regras. Por fim, conseguir um alto grau de naturalidade é possível, mas as regras para tal ainda não foram descobertas [2].

2.1.2 Sintetizadores por Concatenação

Em contrapartida aos sintetizadores baseados em regras, sintetizadores baseados em concatenação possuem conhecimento bastante limitado a respeito dos dados que manipulam. A maioria destes dados estão embutidos em segmentos que serão encadeados. Este fato pode ser observado na figura 2.1, onde todas as operações que poderiam ser utilizadas de maneira transparente em um sintetizador musical (ou seja, sem nenhuma referência explícita à natureza interna do som a ser processado) foram agrupadas em um bloco chamado de "Processamento do Sinal". Em contrapartida, as operações contidas no bloco chamado de "Ciência da Fala" precisam possuir algum conhecimento sobre fonética.

A grande vantagem deste tipo de síntese em relação à baseada em regras é o maior grau de naturalidade que oferece. E possível obter bons resultados, mesmo com um pequeno conjunto de dados, desde que se tenha cuidado e precisão na elaboração da base de dados. É muito mais simples produzir um sintetizador por concatenação de qualidade aceitável, do que um baseado em modelos fonológicos, dado que o primeiro não requer conhecimentos tão aprofundados dos processos de produção da fala, sendo, desta forma, preferidos pelos engenheiros em geral.

Consistindo esta técnica em concatenar segmentos fonéticos pré-gravados para sintetizar a fala, deve ser dada alguma importância à forma como esses sons vão ser armazenados. Como exemplos de formas como os segmentos fonéticos podem ser armazenados, têm-se:

- O armazenamento da formas de onda.
- O armazenamento da informação espectral de seus vários segmentos (formantes ou coeficientes LPC).

Os difones são os tipos de segmentos mais utilizados em sistemas de síntese da fala a partir do texto, mas pode-se usar outras unidades, desde fones a frases inteiras. A grande vantagem do uso de difones é o fato destes preservarem a zona menos estável dos fonemas, sendo a concatenação feita nas suas zonas mais estáveis, mas, mesmo assim, é necessário minimizar as variações espectrais nos pontos de concatenação. Outra vantagem do uso desta unidade de concatenação em relação a unidades maiores é relativa ao tamanho da base de dados. Uma base de dados

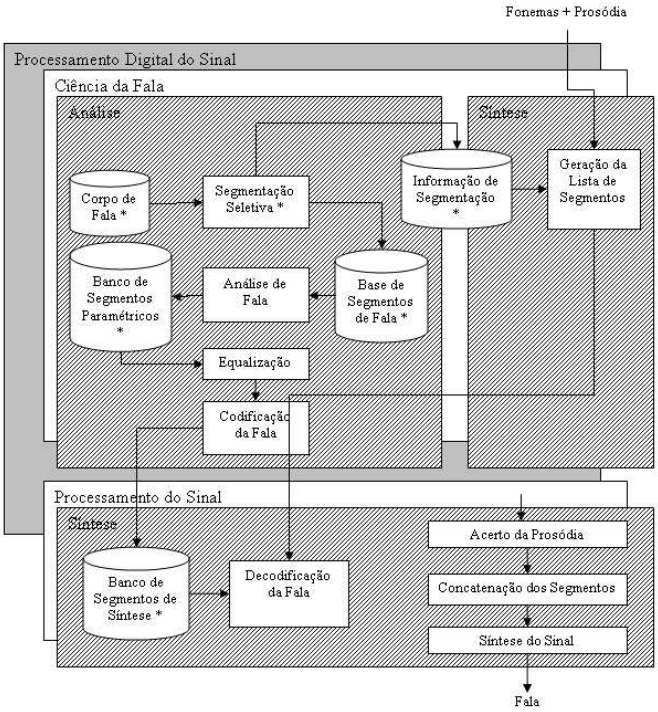

Figura 2.1: Estrutura geral de um sintetizador de voz baseado em concatenação. Os blocos dependentes do idioma estão representados por um "*" (extraído de [2]).

de difones que cubra todas as combinações de segmentos compreende o quadrado do número de fonemas de uma língua, enquanto que, se for utilizado trifones, teria-se o cubo deste valor. Desta maneira, construir uma base de dados de trifones, sílabas ou unidades ainda maiores para uma língua, com fins genéricos, mostra-se extremamente extensa, em termos de recursos [3].

Alguns dos aspectos mais importantes deste tipo de síntese são:

- **Tipo de segmentos:** a seleção dos segmentos deve considerar os efeitos de coarticulação, as transições, a facilidade de conexão, o comprimento e o número de unidades.

- **Corpus:** deve ser o menor possível, mas ao mesmo tempo conter todas as unidades necessárias.

- **Segmentação:** a segmentação pode ser automática ou realizada por um especialista, manualmente. Em ambos os casos, deve-se ter atenção à escolha dos limites dos segmentos.

- **Modelos do sinal da fala:** para modelar os parâmetros de entrada. Existem três modelos possíveis: *modelos articulatórios*, que modelam a produção da fala usando parâmetros capazes de dar uma interpretação física da produção; *modelos de produção de fala*, que, embora tentem modelar o mecanismo de produção, usam parâmetros que não são interpretáveis em termos do processo articulatório; e *modelos fenomenológicos*, que desprezam qualquer referência à produção da fala, focando-se na modelagem do espectro ou da evolução do sinal. Todos eles introduzem erros que poderão degradar a naturalidade do sinal sintético.

- **Codificação:** para que se reduza o tamanho da base de dados, os sinais podem ser codificados, sendo necessário ter atenção à perdas inerentes ao processo de codificação.

- **Prosódia:** os modelos de fala devem permitir alterações prosódicas, ou seja, da freqüência fundamental, das durações ou mesmo da intensidade dos sinais originais, de forma a melhorar a naturalidade do sinal sintético.

- **Algoritmo de concatenação:** na fala natural, verifica-se que as transições entre sons, salvo algumas exceções, apresentam uma variação lenta da trajetória dos articuladores; logo, deve-se conseguir o mesmo efeito usando a concatenação de segmentos de fala.

2.2 O Algoritmo *Time Domain Pitch Synchronous Overlap and Add* (TD-PSOLA)

Dentre as técnicas de conversão texto-fala recentemente desenvolvidas, o algoritmo TD-PSOLA (*Time Domain Pitch Synchronous Overlap and Add*) tem recebido considerável atenção, dado a sua excelente eficiência segmental e supra-segmental [2], associada com sua simplicidade quase inigualável, para a versão temporal (*Time Domain*) do mesmo. A principal característica do método TD-PSOLA é que este mostra que é possível executar modificações no *pitch* de forma de ondas contínuas diretamente, sem a necessidade de nenhum modelo paramétrico, aumentando, assim, a qualidade segmental dada pelos conversores texto-fala.

Supondo um sinal restrito $s(n)$ puramente periódico, é possível obter, a partir deste, uma versão $\widetilde{s}(n)$ com o *pitch* modificado somando com sobreposição *frames* $s_i(n)$, extraídos de maneira síncrona em relação ao *pitch* de $s(n)$, aplicando-se uma janela $w(n)$ a este *frame*, e trocando o deslocamento temporal entre *frames* originalmente com um período de *pitch* T_0 para o período de *pitch* T desejado.

$$s_i(n) = s(n)w(n - iT_0) \tag{2.1}$$

$$\widetilde{s}(n) = \sum_{i=-\infty}^{\infty} s_i(n - i(T - T_0)) \tag{2.2}$$

Caso $T \neq T_0$, de acordo com a fórmula de Poisson[1], a operação resulta na rearmonização do espectro de $s_i(n)$ (que se assumirmos periodicidade perfeita, não dependerá de i quando o som vozeado original for puramente periódico) com freqüência fundamental $1/T$.

$$\text{Se } s_i(n) \leftrightarrow S_i(\omega) \Rightarrow \widetilde{s}(n) \leftrightarrow \frac{2\pi}{T} \sum_{i=-\infty}^{\infty} S_i\left(i\frac{2\pi}{T}\right) \tag{2.3}$$

Conseqüentemente, $w(n)$ pode se escolhida de forma que o espectro de $s_i(n)$ fique bastante próximo do envelope espectral de $s(n)$. Assim, a equação 2.1 apresenta uma maneira simples e eficiente de mudar o *pitch* de um sinal periódico. Este processo pode ser melhor observado na figura 2.2. No gráfico superior da mesma, tem-se o sinal original de voz, cujo *pitch* deseja-se modificar. O gráfico do meio apresenta o posicionamento das janelas obtidas do gráfico superior. Por fim, o último gráfico apresenta a versão final, após a soma com sobreposição, resultando na forma de onda com *pitch* modificado. Em qualquer uma das figuras, percebe-se que o envelope espectral do sinal não foi alterado.

TD-PSOLA tem sido amplamente utilizado para síntese concatenativa, devido a sua alta qualidade de síntese e custo computacional extremamente reduzido. Entretanto, a voz sintetizada resultante não é perfeita. A simplicidade do algoritmo é alcançada ao custo da facilidade de casamento entre segmentos adjacentes, um problema que é mais facilmente resolvido com sintetizadores paramétricos.

[1] De acordo com a fórmula de Poisson, somar um número infinito de versões deslocadas de um dado sinal $f(t)$ resulta na amostragem da transformada de Fourier com período de amostragem igual ao inverso do deslocamento no tempo.

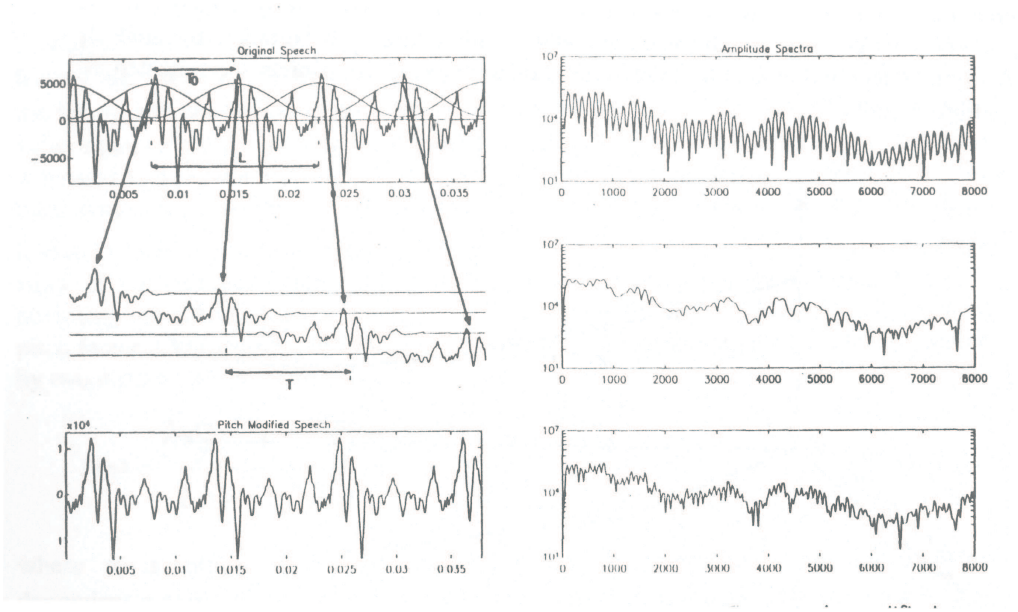

Figura 2.2: Processo de rearmonização do TD-PSOLA. A versão com *pitch* modificado (último gráfico) tem a mesma envoltória espectral que a forma de onda original (gráfico superior) (extraído de [2]).

Existe também uma implementação baseada em PSOLA que trabalha no domínio freqüencial. Esta técnica, chamada de *Frequency Domain Pitch Synchronous Overlap and Add* (FD-PSOLA) [4] baseia-se em modificar o sinal de voz no domínio da freqüência, usando a transformada de Fourier. A vantagem deste método é que ele é capaz de modificar com grande flexibilidade as características espectrais do sinal de voz, porém, com um custo computacional bem mais elevado.

2.3 Estudo da Prosódia

Nesta seção, o conceito de prosódia será abordado, uma vez que um adequado estudo da mesma torna-se fundamental para a implementação de sintetizadores de voz com maior naturalidade da fala produzida. Assim, torna-se necessário compreender o que é este conceito, como é gerado, e como pode ser manipulado, de forma a ser utilizado em conversores texto-fala.

2.3.1 O que é Prosódia?

Segundo [5], prosódia é a parte da fonética que trata da correta acentuação e entonação dos fonemas. Entretanto, para os engenheiros da área de processamento de voz, a prosódia surge através de certas características do sinal de fala, como mudanças no *pitch*, volume e duração. A prosódia resulta da maneira pela qual seres humanos manejam estes parâmetros na hora de falar, de forma a melhorar a comunicação oral. Através da prosódia, frases idênticas podem assumir significados distintos. Por exemplo, a frase "Você vai estudar?" e a frase "Você vai estudar!", nota-se que, apesar de serem exatamente idênticas, na primeira trata-se de uma pergunta, e na segunda, de uma afirmação. Da mesma maneira que a pontuação distingue os dois sentidos em um texto, a prosódia é responsável por distinguí-los durante a fala. Além disso, a prosódia é fundamental para o dinamismo da fala, de forma que, sem esta, a fala tende a se tornar extremamente enfadonha.

2.3.2 Componentes Principais da Prosódia

Os principais componentes da prosódia são: tonicidade, duração silábica e variação do *pitch*.

Tonicidade é uma propriedade prosódica que vem sendo descrita desde o primeiro estudo sobre prosódia. A tonicidade é a energia aplicada a cada sílaba de uma frase, de tal maneira que a variação da energia em cada sílaba permite ao orador transmitir idéias relacionadas aos seus sentimentos, como raiva, alegria, etc, além de servir como reforço a uma parte da frase que mereça maior importância.

Duração é uma medida prosódica que determina a duração de cada sílaba na frase. Este parâmetro também ajuda o orador a dar maior ênfase a uma determinada palavra que mereça mais atenção, além de ser necessário para a introdução de ritmo na frase.

Variação do *pitch* é, dos três parâmetros, o mais importante. A variação do *pitch* ao longo da frase é fundamental para a exata compreensão do sentido da mesma, dando, também, indicativos do sentimento que o orador está querendo transmitir.

2.3.3 Geração Automática da Prosódia

Como visto anteriormente, a prosódia é fundamental para a naturalidade da fala, aumentando a eficiência da comunicação e o dinamismo da oratória. Assim, pesquisas [6, 7] vêm sendo desenvolvidas de maneira que a prosódia seja gerada automaticamente, de tal forma que possa ser incorporada aos conversores texto-fala.

Um dos métodos mais utilizado é o modelo de Fujisaki. Este modelo busca obter a curva de *F0* para uma dada frase, baseado na suposição fundamental de que curvas de entonação, embora contínuas no tempo e na freqüência, originam eventos discretos ativados pelo orador, que aparece como um mecanismo fisiológico contínuo relacionado ao controle fundamental de freqüência. Assim, Fujisaki distingue dois tipos de eventos discretos, denominados comandos de *frase* e *acento*, e estes são, respectivamente, modelados como pulsos e funções degrau. Estes comandos geram filtros lineares de segunda ordem criticamente amortecidos, cujas saídas são somadas para originar os valores de *F0* (ver figura 2.3). Fujisaki argumenta que a escolha destas funções de transferência e dos dois tipos de comando baseiam-se em fundamentos fisiológicos. Com isso, a equação para a geração da curva de *F0* é dada por [8]

$$\ln F_0(t) = \ln F_b + \sum_{i=1}^{I} A_{pi} G_p(t - T_{0i}) + \sum_{j=1}^{J} A_{aj} [G_a(t - T_{1j}) - G_a(t - T_{2j})] \quad (2.4)$$

onde

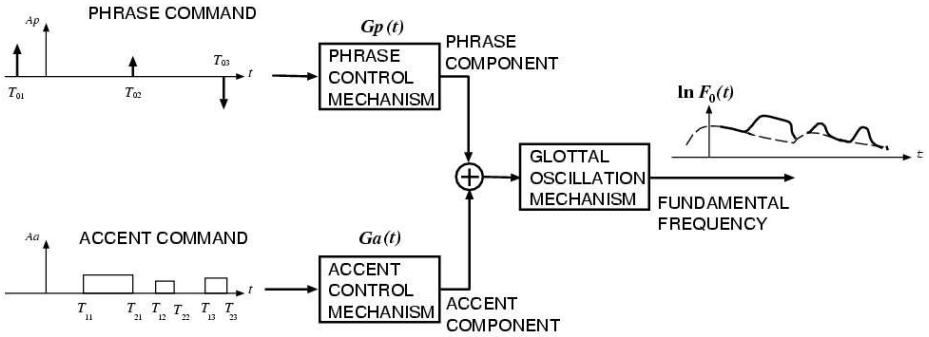

Figura 2.3: Modelo de produção de prosódia de Fujisaki (retirado de [8]).

$$Gp(t) = \begin{cases} \alpha^2 t e^{-\alpha t}, & \text{para } t \geq 0, \\ 0, & \text{para } t < 0 \end{cases} \tag{2.5}$$

e

$$Ga(t) = \begin{cases} \min[1 - (1 + \beta t)e^{-\beta t}, \gamma], & \text{para } t \geq 0, \\ 0, & \text{para } t < 0 \end{cases} \tag{2.6}$$

Assumindo que os parâmetros dos filtros lineares são fixos para um dado orador (uma vez que eles estão relacionados a restrições fisiológicas), os parâmetros relevantes para este modelo são relativamente poucos:

- Uma freqüência de polarização F_b sobre o qual todos os comandos de frase e acento serão sobrepostos.
- O número (I) de comandos de frase e dois parâmetros para cada comando: sua amplitude (A_p) e posição temporal (T_0).
- O número (J) de comandos de acento e três parâmetros para cada comando: sua amplitude (Aa) e as posições temporais de subida (T_1) e descida (T_2) do pulso.
- O parâmetro α na equação 2.5 denota a constante de tempo do mecanismo de controle de frase, a qual é assumida como constante dentro de uma emissão.
- O parâmetro β na equação 2.6 denota a constante de tempo do mecanismo de controle do acento, e também é tida como constante para uma emissão. O valor γ é o limiar máximo para o mecanismo de controle do acento, e assegura que o componente do acento atinja seu valor máximo em um tempo finito.

A extração automática destes parâmetros é o principal desafio para a implementação deste modelo, uma vez que envolve o conhecimento das funções sintáticas da frase, bem como análise de pontuação, conhecimentos estes que podem ser bastantes intuitivos. Em [7, 9] encontram-se algumas técnicas utilizadas para a extração destes parâmetros.

2.4 Sistemas de Conversão Texto-Fala Existentes

Esta seção apresentará, resumidamente, alguns sistemas comerciais de conversão texto-fala atualmente existentes para o Português [9].

2.4.1 DeltaTalk

O DeltaTalk [10] é um sistema de conversão texto-fala para o Português brasileiro desenvolvido pela *MicroPower Software*.Os requisitos mínimos de *hardware* para a última versão são um Pentium 100 com 16 Mbytes de memória e 30 Mbytes de espaço em disco. As plataformas suportadas são as versões 9x e NT 4.0 do Microsoft Windows. O DeltaTalk pode se comunicar com o *Virtual Vision*, da mesma empresa, permitindo que deficientes visuais usem o computador. O *Virtual Vision* pode ler textos digitados pelo usuário, em documentos ou páginas da Internet, inclusive informando os *links* disponíveis na página e o tipo e tamanho das fontes usadas. Pode, também, informar os objetos que estão sendo apontados pelo *mouse*, ler mensagens enviadas por aplicativos e fornecer detalhes sobre o *status* dos controles do Windows.

O DeltaTalk faz a conversão texto-fala através da concatenação de difones. O sistema faz normalização de texto e possui um dicionário de exceções no qual novas pronúncias podem ser adicionadas pelo usuário. O sistema possui modelagem prosódica, o que significa que deve ser feito algum tipo de processamento lingüístico de alto nível. É possível controlar a velocidade da pronúncia e a tonalidade da voz, bem como editar o contorno de *pitch* mediante um editor de melodias.

2.4.2 Digalo

O Digalo [11] é uma divisão da *Elan Informatique*, uma empresa com sede em Toulouse, França. O Digalo não é um *software* de síntese embutido em um aplicativo, mas um *plug-in* que contém o motor de síntese. Qualquer aplicativo pode ter acesso ao motor de síntese através da *interface* Microsoft SAPI (*Speech Application Program Interface*) 4.0. Com a adoção deste padrão, o motor de conversão texto-fala pode ser compartilhado por vários aplicativos compatíveis com esta *interface*, e os aplicativos podem utilizar quaisquer motores que tenham sido instalados, o que permite grande

integração entre aplicativos.

Os motores da Digalo utilizam o algoritmo PSOLA, e estão disponíveis em 7 línguas, entre elas, o Português brasileiro. A única plataforma suportada é o Microsoft Windows (9x, 2000 e NT), e o *hardware* mínimo é um PC com um processador Pentium 133 MHz ou equivalente, com 32 Mbytes de RAM e, de 3 a 10 Mbytes de espaço livre em disco, dependendo da língua do motor de síntese.

2.4.3 Lernout & Hauspie

Os recursos de fala e linguagem da Lernout & Hauspie [12] foram adquiridos pela *ScanSoft inc.*, que fornece um sistema TTS chamado RealSpeak. Este sistema possui motores em 19 línguas, entre elas o Português. As plataformas suportadas são: Linux, Solaris, AIX e várias versões do Microsoft Windows. Este sistema possui processamento lingüístico e modelagem prosódica. A *interface* com os motores de síntese com os aplicativos é feita através do padrão Microsoft SAPI.

2.4.4 Texto-Fala

O Texto-Fala [13] é um sistema TTS desenvolvido pela fundação CPqD, em colaboração com a *Elan Informatique*. O sistema é concatenativo e usa polifones como unidades, bem como a técnica PSOLA para modificar os segmentos.

O Texto-Fala e os motores de síntese da Elan são um dos melhores sistemas TTS existentes, possuindo boa qualidade segmental, e melodia e ritmo razoáveis [9]. O sistema possui, ainda, processamento lingüístico e prosódico. O modelo de entonação consiste em duas etapas: a primeira mapeia o texto numa representação fonológica, e a última mapeia esta representação no contorno de *pitch*. O primeiro mapeamento consiste em um *parsing* que encontra as fronteiras dos constituintes prosódicos e os níveis de acento de cada palavra. São usadas 20 marcações, reunidas em 4 grupos, para especificar o centro da palavra. O segundo mapeamento utiliza estas marcações para consultar um dicionário de contornos de *pitch*, encolhendo a entrada mais similar. Os contornos obtidos do dicionário são ajustados e concatenados para produzir o contorno de *pitch* final [9].

2.5 Conclusão

Neste capítulo, foi apresentado uma rápida introdução à ciência de síntese de voz. A teoria básica de prosódia também foi apresentada, e uma introdução teórica ao algoritmo TD-PSOLA foi realizada. Com esta apresentação, pode-se destacar as qualidades que tornam esta técnica interessante para a implementação do projeto. Entretanto, esta técnica apresenta o problema de requerer uma quantidade demasiadamente grande de memória para armazenar sua base de dados, de forma que uma técnica de compressão desta base de dados deve ser desenvolvida, como forma de minimizar este problema, como será visto no próximo capítulo.

Apresentou-se, por fim, alguns sistemas comerciais de conversão texto-fala, mostrando que estes são desenvolvidos apenas para computadores de uso geral, de tal forma que, em nenhum destes sistemas foi priorizada a mobilidade do produto, através do desenvolvimento de um conversor texto-fala portátil, limitando a sua utilização.

Capítulo 3

O Sistema de Codificação CELP

Como visto no capítulo 2, um dos principais problemas de um sintetizador por concatenação de segmentos é a elevada quantidade de memória que o mesmo necessita para o armazenamento da sua base de dados. Tendo em mente o desenvolvimento de um sistema de síntese de voz portátil e de baixo custo, esta grande necessidade de memória resultaria no encarecimento e no aumento do tamanho do produto, uma vez que módulos com grande capacidade de memória (da ordem de dezenas de Mbytes) seriam necessários. Para minimizar este problema, decidiu-se compactar esta base de dados utilizando codificação CELP (*Code Excited Linear Prediction*), que é, atualmente, a técnica de compressão de voz mais utilizada. Neste capítulo, apresentam-se os conceitos fundamentais de um sistema de codificação, com ênfase em CELP, uma vez que esta será a técnica utilizada neste trabalho.

Inicialmente, será apresentado na seção 3.1 as duas formas de codificação atualmente utilizadas em sistemas de conversão texto-fala. Em seguida, a seção 3.2 apresentará os conceitos envolvidos na codificação CELP, de forma a criar o embasamento necessário para a posterior descrição do algoritmo, que será feita na seção 3.3. Por fim, a seção 3.4 apresentará as alterações propostas para melhorar ainda mais a taxa de codificação do sistema CELP, ao mesmo tempo em que mantém a qualidade praticamente inalterada.

3.1 Sistemas de Codificação

Antes de ser abordado o sistema CELP propriamente dito, torna-se necessário apresentar brevemente alguns métodos de codificação utilizados. Estes são divididos em dois grupos: codificadores de forma de onda e codificadores de fonte ou paramétricos. Nesta seção, será dada uma rápida introdução a estes codificadores.

3.1.1 Codificadores de Forma de Onda

Estes codificadores utilizam, para a codificação, propriedades do sinal de caráter temporal, ou até mesmo espectral. Entretanto, nenhum tipo de suposição sobre o sinal a ser codificado é feita, de forma que qualquer tipo de sinal pode ser codificado eficientemente (do ponto de vista de reconstrução do sinal original) por codificadores de forma de onda. Entretanto, esta generalidade acarreta numa maior taxa de bits, uma vez que características inerentes a um determinado tipo de sinal não podem ser exploradas para que seja possível atingir uma melhor eficiência. Dentre os codificadores de forma de onda existentes, o codificador PCM [14] é o mais utilizado, sendo, inclusive, o mais utilizado para a codificação da base de dados de sintetizadores de voz.

Neste método, divide-se a faixa dinâmica do sinal a ser codificado em partes iguais, onde cada parte corresponde a um código binário, de tal forma que, quanto maior o código, maior o número de espaçamentos, e melhor a qualidade da codificação. Assim, por exemplo, se forem utilizados 8 bits, um total de 256 níveis estarão disponíveis para a codificação.

Para a realização da codificação, o sinal é observado a um período fixo (período de amostragem), e a cada observação, determina-se em qual intervalo aquela amostra ficou, sendo atribuída a esta amostra, a seqüência binária correspondente àquele intervalo. Assim, supondo que o sistema opere com resolução de N bits, e utilizando uma freqüência de amostragem de F *Hertz*, tem-se que a taxa de bits T_b gerada por este codificador é dada por:

$$T_b = N \times F \text{ bits / s} \tag{3.1}$$

Este tipo de codificação é o que apresenta a melhor qualidade, ao custo de ser

o que apresenta a maior taxa de bits. Isto faz com que uma base de dados codificada por este método fique demasiadamente grande, uma vez que, para sinais de voz, normalmente utiliza-se $N = 8$ bits e $F = 8$ kHz. Assim, atinge-se, tipicamente, uma taxa de 64 kbits por segundo.

3.1.2 Codificadores de Fonte ou Paramétricos

Esta classe de codificadores surgiu como uma alternativa aos codificadores de forma de onda. Sua principal função é reduzir as altas taxas de bit dos codificadores de forma de onda, sem reduzir a qualidade da reconstrução do sinal original. Para realizar esta tarefa, esta classe de codificadores procura explorar ao máximo características específicas de um determinado tipo de sinal. Desta maneira, para a codificação de sinais de voz, por exemplo, toda a estrutura de produção da voz (trato vocal, conceitos lingüísticos, etc) são estudados de forma que se encontrem características que permitam a codificação com a menor taxa de bits possível para este tipo de sinal, sem deteriorar significativamente a qualidade do sinal resultante após a decodificação. Entretanto, como as características de um sinal mudam conforme o tipo de sinal (voz, música, imagem, etc), tem-se que codificadores paramétricos são específicos para cada tipo de sinal. O codificador paramétrico de voz mais utilizado é o LPC (*Linear Predictive Coding*) [15].

O LPC visa, ao invés de codificar o sinal de voz, codificar a estrutura que o produz. A voz é produzida quando o ar dos pulmões passa pelas cordas vocais, e, em seguida, dirige-se ao trato vocal. A configuração deste trato vocal (posição da língua, do maxilar, ou a possibilidade do ar passar pelo nariz) dá a forma final ao fonema a ser pronunciado. Assim, tem-se uma excitação (o ar dos pulmões), que pode ser periódica (caso as cordas vocais vibrem, como no caso de sons vozeados) ou não (para sons surdos), e um sistema que altera as propriedades desta excitação. Assim, pode-se interpretar o trato vocal como um filtro que se encarrega de modelar o sinal de excitação de forma a produzir o som que se deseja.

A codificação LPC baseia-se nesta filosofia. Primeiramente, escolhe-se uma janela de tamanho reduzido (de 10 a 30 ms), de tal forma que o sinal possa ser considerado estacionário. Depois, para cada janela, devem-se extrair as informações fundamentais que compõem este sinal, que são:

- Informação de sonoridade
- A freqüência e a amplitude da excitação.
- A configuração do trato vocal para aquela janela.

A freqüência, amplitude e informação de sonoridade podem ser facilmente determinadas observando-se a forma de onda a ser codificada. O trato vocal, como já dito, pode ser interpretado como um filtro. Assim, utiliza-se um filtro digital para representá-lo. Com isso, após determinar a freqüência de vibração do sinal de excitação, o codificador LPC determina os coeficientes que compõem o filtro digital modelante do trato vocal para aquela janela através de regressão linear. Com isso, a taxa de bits fica bastante reduzida, uma vez que se reduz significativamente o número de parâmetros a serem enviados.

Exemplificando, para uma janela de 20 ms de duração, e supondo 10 coeficientes para o filtro, seriam enviados apenas 13 valores a cada 20 ms, enquanto que o sistema PCM, uma vez que codifica todas as amostras, enviaria 160 valores. Desta forma, comparando o sistema LPC com o PCM, temos que o LPC possui, para este caso, uma taxa de bits 71,5% menor.

O problema do codificador LPC é que a excitação é sempre caracterizada como sonora ou surda. Como existem outras classificações para a excitação, e estas não são consideradas neste classificador, tem-se como resultado final uma voz um tanto "robotizada", o que degrada consideravelmente a qualidade da reprodução.

3.2 O Sistema de Codificação CELP

O sistema de codificação CELP [16, 17] é a união das características dos codificadores apresentados nas sessões 3.1.1 e 3.1.2, no sentido de que se mantém a parametrização LPC ao mesmo tempo em que se determina a excitação pelo formato da onda. Através da determinação da melhor excitação com o auxílio de dicionários, é possível que se alcancem taxas de codificação bem reduzidas, mantendo uma qualidade perfeitamente aceitável.

Este sistema utiliza como base o sistema LPC. Entretanto, o CELP explora aquela que talvez seja a principal desvantagem do LPC: a questão da excitação uti-

lizada. O CELP utiliza dicionários para manipular estas excitações. Desta maneira, é possível se obter um número bem mais amplo de excitações, o que torna a voz reconstituída, após a decodificação, muito mais natural que aquela que se obteria com o sistema LPC básico [17].

3.2.1 O Janelamento do Sinal de Voz

O codificador CELP, de maneira semelhante aos codificadores paramétricos, opera sobre um conjunto de amostras do sinal de voz a ser codificado. Assim, torna-se necessário o janelamento do sinal de entrada. Neste trabalho, optou-se por usar uma janela de *Hanning* [18] com $\alpha = 0,54$ e duração de 20 ms, o que, para uma freqüência de amostragem de 8 kHz, corresponde a 160 amostras.

3.2.2 Filtro de Síntese

O filtro de síntese [16] $H(z)$ é o filtro que representa os efeitos do trato vocal, e é dado por:

$$H(z) = \frac{1}{A(z)} = \frac{1}{1 - \sum_{i=1}^{p} a_i z^{-i}} \tag{3.2}$$

onde a constante p representa a ordem do modelo LPC e denota a precisão com a qual os efeitos do trato vocal serão modelados por $H(z)$. Neste trabalho, foi utilizado $p = 10$, de forma a obter um compromisso entre qualidade e taxa de bits. Os coeficientes a_i são os coeficientes LPC obtidos desta análise.

3.2.3 Filtro de Ponderação ou Perceptivo

De acordo com [16], foi verificado que erros e ruídos em componentes de menor amplitude fazem mais diferença ao ouvido humano do que em componentes de maior amplitude. Como conseqüência, componentes de menor amplitude são mais importantes no cálculo do erro contido no processo de análise por síntese. Assim, a função do filtro perceptivo é enfatizar as componentes de menor amplitude, ao mesmo tempo em que realiza o processo inverso com as componentes de maior amplitude. O filtro perceptivo $W(z)$ é dado por:

$$W(z) = \frac{A(z)}{A(\frac{z}{\gamma})} \tag{3.3}$$

onde γ é o fator de ponderação, que possui o valor típico de 0,8 [19]. Pode-se observar na figura 3.1 o efeito do filtro perceptivo, onde se nota a conseqüente atenuação das componentes de maior amplitude, ao mesmo tempo em que enfatiza as componentes de menor amplitude.

Contudo, o que realmente é utilizado neste trabalho é uma aglutinação do filtro de síntese com o filtro perceptivo. Assim, o filtro de síntese modificado é dado por:

$$H(z) \cdot W(z) = \frac{1}{A(z)} \cdot \frac{A(z)}{A\left(\frac{z}{\gamma}\right)} = \frac{1}{A\left(\frac{z}{\gamma}\right)} \tag{3.4}$$

3.2.4 Dicionário Fixo

Em um segmento do sinal de fala digital existem correlações entre amostras vizinhas, devido ao efeito do trato vocal humano, que são geralmente chamadas de correlações de curto termo. Além disso, se o segmento considerado for sonoro, existem também correlações entre amostras da ordem de períodos de *pitch*, e que são geralmente chamadas de correlação de longo termo. Se o segmento for surdo, somente correlações de curto termo existirão [16]. Desta maneira, se forem removidas [20] as correlações de curto e longo termo de determinado segmento de sinal de fala, o resultado será um sinal com características semelhantes a um ruído branco. E este sinal resultante é justamente a excitação que é aplicada ao trato vocal. Assim, o dicionário fixo é formado por um conjunto fixo de excitações, onde cada termo é uma seqüência de um processo estocástico gaussiano de média zero. Além disso, o dicionário fixo é permanente para o sistema, e deve ser o mesmo tanto no codificador, como no decodificador.

3.2.5 Dicionário Adaptativo

O dicionário adaptativo tem como finalidade fornecer uma espécie de "ajuste fino" para a codificação, representando a parte da excitação que o dicionário fixo não conseguiu representar, complementando assim, a função do mesmo. Isto é possível

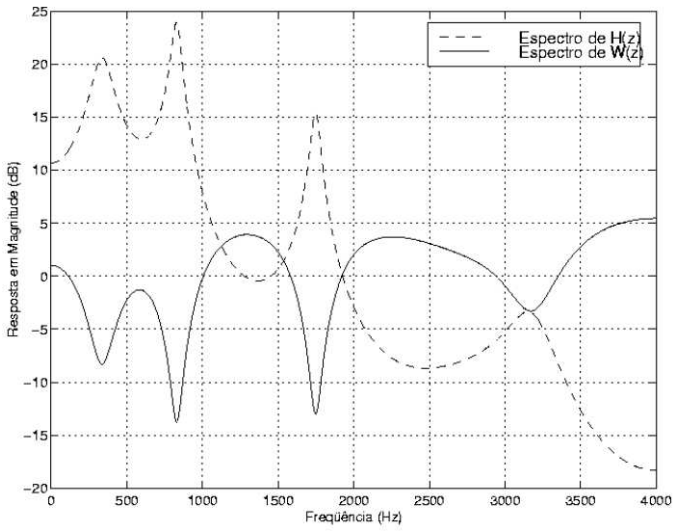

Figura 3.1: Efeito do filtro perceptivo para $\gamma = 0,8$.

uma vez que este dicionário, ao longo do processo de codificação, adapta-se dinamicamente (a forma como esta adaptação é feita será vista na seção 3.3) às características vocais do sinal a ser codificado, buscando, principalmente, as correlações de longo termo existentes naquele sinal, de forma que excitações que não estejam contidas no dicionário fixo possam ser anexadas ao sistema de codificação e, assim, aumentar a fidelidade do sinal de excitação final gerado pelo codificador.

3.2.6 Cálculo dos Ganhos Fixos e Adaptativos

Os sinais de excitação dados pelos dicionários fixos e adaptativos possuem certa amplitude, que pode não corresponder à amplitude real da excitação original. Assim, torna-se necessário calcular o valor de ganho que será aplicado às seqüências escolhidas de cada dicionário. O ganho, então, é dado por

$$G = \frac{\mathbf{S}_a \cdot \mathbf{R}_d^T}{\mathbf{R}_d \cdot \mathbf{R}_d^T} \tag{3.5}$$

onde $\mathbf{S}_a$ é o vetor com o sinal alvo de excitação e $\mathbf{R}_d$ é a seqüência escolhida do dicionário em questão. Assim, o ganho é dado pela razão entre a correlação do sinal alvo com a seqüência escolhida do dicionário e a autocorrelação da seqüência do dicionário escolhida. O sinal alvo é o sinal de voz do qual deve ser feita a análise por síntese.

3.2.7 Análise por Síntese

A excitação que deverá ser usada para reconstituir a voz do usuário na saída do sistema será determinada através do processo de "Análise por Síntese". Este processo é realizado dividindo-se, inicialmente, a janela de 160 amostras (20 ms) em quatro sub-blocos de 40 amostras (5 ms), uma vez que a excitação varia mais rapidamente do que o trato vocal propriamente dito [16]. Assim, a análise por síntese se dá do seguinte modo:

1. Para cada sub-bloco de 5 ms, cada excitação contida no dicionário é submetida ao filtro calculado pelo filtro de síntese modificado, gerando uma resposta.

2. Esta resposta, então, é subtraída do sinal de voz presente na entrada do sistema.

3. A diferença resultante é, então, guardada.

4. A excitação que gerar a menor diferença será aquela escolhida para reconstruir o sinal de voz na saída do sistema.

O nome deste processo pode ser justificado pelo fato de ser necessário sintetizar o sinal de voz várias vezes para que se possa realizar a análise para determinar a melhor excitação.

3.3 Funcionamento do Sistema CELP

O sistema pode ser visualizado como um todo no diagrama de blocos da figura 3.2. Nesta figura, existem marcações para auxiliar a identificação de cada ponto que consta nesta descrição. Um detalhe importante é que este esquema relata o que ocorre após o janelamento, a análise LPC e a divisão em sub-blocos. A descrição do algoritmo [17] segue abaixo:

1. O sub-bloco do sinal de voz entra no sistema.

2. O sub-bloco do sinal de voz é filtrado pelo filtro de ponderação.

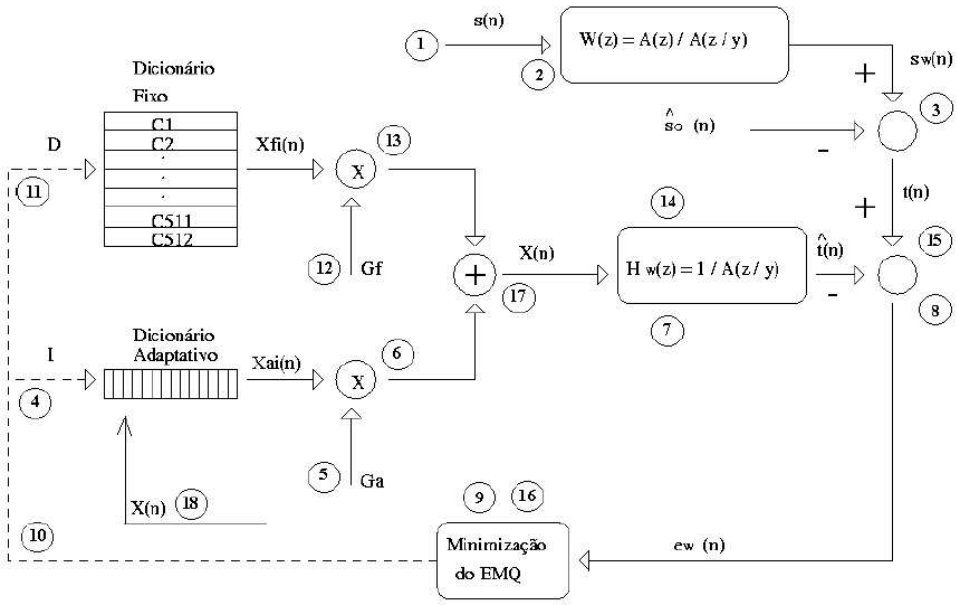

Figura 3.2: Diagrama em blocos do codificador CELP.

3. A resposta à entrada zero relativa ao sub-bloco anterior é subtraída da saída do filtro de ponderação, formando o denominado "sinal-alvo". A resposta à entrada zero pode ser entendida como um resquício da resposta do filtro de síntese que aparece na saída do filtro no momento em que a excitação não está mais presente.

4. O processo de análise por síntese tem início com a procura no dicionário adaptativo.

5. Tendo em mãos o índice da excitação do dicionário adaptativo, deve-se calcular o ganho relativo a esta excitação.

6. Multiplica-se esta excitação pelo ganho calculado;

7. Filtra-se a excitação atual (multiplicada pelo ganho calculado) pelo filtro de síntese alterado pelo filtro de ponderação, gerando-se o sinal estimado.

8. Subtrai-se o sinal estimado do sinal-alvo, gerando-se um sinal de erro.

9. A excitação que gerar o EMQ (erro médio quadrático) mínimo será a excitação ótima. Seu índice e o ganho relativo a esta serão guardados.

10. Atualiza-se o sinal-alvo, subtraindo do mesmo a melhor excitação do dicionário adaptativo (multiplicada pelo devido ganho) submetida ao filtro de síntese alterado pelo filtro de ponderação.

11. Realiza-se a busca no dicionário fixo já considerando o novo sinal-alvo (atualizado).

12. Tendo em mãos o índice da excitação do dicionário fixo, deve-se calcular o ganho relativo a esta excitação.

13. Multiplica-se esta excitação pelo ganho calculado.

14. Filtra-se a excitação atual (multiplicada pelo ganho calculado) pelo filtro de síntese alterado pelo filtro de ponderação, gerando-se o sinal estimado.

15. Subtrai-se o sinal estimado do sinal-alvo original, gerando-se um sinal de erro.

16. A excitação que gerar o EMQ mínimo será a excitação ótima. Seu índice e o ganho relativo a esta serão guardados.

17. Obtém-se a excitação completa somando-se as excitações de ambos os dicionários multiplicadas pelos respectivos ganhos.

18. Atualiza-se o dicionário adaptativo colocando-se ao fim do mesmo a resposta ótima completa.

Para a decodificação, utiliza-se um algoritmo semelhante, mas sem as buscas nos dicionários, visto que tais buscas já haviam sido feitas no processo de codificação. O único trabalho que o decodificador terá será de buscar a excitação completa, multiplicá-la pelo ganho, filtrá-la com o filtro de síntese e atualizar o dicionário adaptativo. Todos esses parâmetros (índices, ganhos e coeficientes) virão como produto do codificador.

Com esta versão inicial do sistema CELP, para cada janela de 160 amostras, devemos armazenar somente os seguintes parâmetros:

- O vetor com os dez coeficientes LPC.

- O vetor com os índices do dicionário fixo correspondentes a cada sub-bloco de 40 amostras (um índice para cada sub-bloco, totalizando quatro índices).

- O vetor com os índices do dicionário adaptativo correspondentes a cada sub-bloco.
- O vetor com os ganhos do dicionário fixo correspondentes a cada sub-bloco.
- O vetor com os ganhos do dicionário adaptativo correspondentes a cada sub-bloco.

Assim, supondo a alocação de 16 bits para cada valor de índice, e supondo 32 bits (ponto flutuante) para os demais valores, temos, para cada bloco de 160 amostras:

$$T = (10 + 4 + 4) \times 32 + (4 + 4) \times 16 = 704 \text{ bits} \tag{3.6}$$

o que representa uma redução de 45 % na taxa de bits, se compararmos com um codificador PCM que utilize 8 bits por amostra, de tal forma que, para armazenar 160 amostras, seriam necessários $160 \times 8 = 1280$ bits. A seguir, serão apresentados alguns métodos que tornarão a taxa de compressão do sistema CELP ainda mais expressiva.

3.4 Modificações do Sistema CELP

Na seção anterior, observou-se que, para cada bloco de 160 amostras, são necessários 704 bits. Tem-se como principal fator limitante deste valor, o fato de haver a necessidade de armazenar valores em ponto flutuante, que precisam de muitos bits (normalmente 32) para possuírem uma precisão adequada. As duas próximas sessões apresentarão as técnicas utilizadas para reduzir o número de bits necessários para a transmissão de valores em ponto flutuante. Tal redução na taxa de bits torna-se extremamente desejável na hora de codificar a base de dados do sintetizador de voz desenvolvido, uma vez que, quanto menos memória for utilizada, menor e mais barato torna-se o produto final.

3.4.1 Quantização dos Ganhos

Como visto na descrição do algoritmo, para cada janela de 20ms de sinal, é necessário o envio de oito valores de ganhos (quatro para as excitações selecionadas

do dicionário fixo, e mais quatro para aquelas oriundas do dicionário adaptativo). Devido ao fato destes valores serem em ponto flutuante, torna-se necessário o envio de oito palavras de 32 bits para cada janela, totalizando 256 bits por janela.

Visando reduzir este número, optou-se por realizar a quantização dos valores de ganho. Assim, foi realizado a quantização não uniforme dos mesmos, através da geração de um vetor de quantização para cada tipo de ganho, que possui a faixa dinâmica dos valores de ganho dividida em espaços não-uniformes, de acordo com o número de bits que se deseja utilizar para representar cada ganho. Neste trabalho, optou-se por utilizar 6 bits para a quantização de cada valor de ganho, de tal maneira que o vetor de quantização possui $2^6 = 64$ elementos. Assim, em uma única palavra de 32 bits, pode-se enviar os quatro índices[1] correspondentes aos valores de ganho de um dado dicionário, de tal forma que se reduz o número de bits gastos com informação de ganho para apenas 64 bits (32 bits para cada dicionário), o que representa uma redução de 75% na quantidade de bits necessários para representar estes valores.

3.4.2 Quantização dos Coeficientes LPC

De maneira análoga ao que acontece com os valores de ganho, temos que os coeficientes LPC, ao se utilizar 32 bits para cada valor em ponto flutuante, representarão um total de 320 bits para cada janela de 20 ms. Assim, da mesma forma como foi feita com os valores de ganho, foi realizada a quantização dos valores LPC. Entretanto, os coeficientes LPC são extremamente sensíveis a ruído de quantização. Por isso, neste trabalho, foi utilizado os coeficientes LSF (*Linear Spectral Frequency*) [16], que são obtidos através dos coeficientes LPC. Para o cálculo dos LSF, deve-se seguir a seguinte regra [17]:

[1] De fato, cada índice está ocupando 8 bits, de tal forma que os quatro valores concatenados ocupam 32 bits, visto que a palavra do DSP é de 32 bits, como será visto no capítulo 4.

Tabela 3.1: Tabela de distribuição de bits dos coeficientes LSF.

Quantizador	W_1	W_2	W_3	W_4	W_5	W_6	W_7	W_8	W_9	W_10	**Total**
QDLSF	4	4	3	3	3	3	3	3	3	3	32

1. Calculam-se os polinômios $P(z)$ (que é simétrico) e $Q(z)$ (que é anti-simétrico) a partir de $A(z)$ da seguinte forma:

$$P(z) = A(z) + z^{-p-1}A(z^{-1}) \tag{3.7}$$

$$Q(z) = A(z) - z^{-p-1}A(z^{-1}) \tag{3.8}$$

   onde p é o número de coeficientes LPCs.

2. Obtêm-se os polinômios $P_1(z)$, que é definido como sendo $P(z)$ sem a raiz em -1, e $Q(z)$, que é definido como sendo $Q(z)$ sem a raiz em $+1$:

$$P_1(z) = \frac{P(z)}{1+z^{-1}} \text{ e } Q_1(z) = \frac{Q(z)}{1-z^{-1}} \text{ para } p \text{ par} \tag{3.9}$$

$$P_1(z) = P(z) \text{ e } Q_1(z) = \frac{Q(z)}{1-z^{-2}} \text{ para } p \text{ ímpar} \tag{3.10}$$

3. Os polinômios $P_1(z)$ e $Q_1(z)$ são simétricos de ordem par. Como as raízes ocorrem em pares de números complexos conjugados, somente metade delas precisa ser determinada. Assim, para p par, $p/2$ raízes de $P_1(z)$ mais $p/2$ raízes de $Q_1(z)$, totalizando p, podem representar os polinômios $P(z)$ e $Q(z)$, e, conseqüentemente, o filtro $1/A(z)$, que é o filtro de síntese. Como as p raízes estão sobre o círculo unitário, somente os ângulos (ou freqüências) são necessários para representar $A(z)$. Tais freqüências são as LSFs.

Contudo, são transmitidas somente as diferenças entre os coeficientes LSF, denominadas DLSF, visando reduzir ainda mais o erro de quantização, uma vez que os coeficientes LSF podem ser ordenados. Assim, torna-se válido utilizar mais bits para os primeiros coeficientes. A tabela 3.1 apresenta a distribuição de bits utilizada neste trabalho, visando o compromisso entre compactação e qualidade de codificação.

Com a quantização dos coeficientes LPC, tem-se que o número de bits gastos para representar estes valores cai para 32 bits por janela, representando uma redução de 90% quando comparado ao número original de bits, anterior à quantização destes valores.

3.5 Conclusão

Foi apresentado o sistema de codificação CELP utilizado para a compressão das amostras de áudio da base de dados. O sistema inclui, ainda, diversas melhorias, de forma a aumentar ainda mais a taxa de compressão das amostras de áudio. Este sistema foi comparado com os dois métodos clássicos de codificação (PCM e LPC), onde se mostrou que a codificação CELP é a técnica que alia taxas de bits reduzidas, com excelente qualidade de codificação.

De posse de um codificador dotado destas qualidades, torna-se viável a redução da base de dados do sintetizador através da codificação das amostras de áudio da base utilizando está técnica, como será visto no capítulo 5.

Capítulo 4

O Processador ADSP-21160M

Como este trabalho trata da implementação de um sistema digital, torna-se necessária a utilização de um *hardware* digital para a execução do mesmo. Para o desenvolvimento de um protótipo portátil e de baixo custo, é desejável que o dispositivo escolhido tenha as seguintes características:

- Preço reduzido.
- Tamanho reduzido.
- Baixo consumo de energia.
- Grande poder de processamento.
- Boa quantidade de dispositivos integrados.

Para atender estas exigências, um DSP foi escolhido como dispositivo digital para a implementação do sistema de síntese de voz. Este capítulo apresentará o processador ADSP-21160M, que foi o DSP utilizado neste trabalho. A seção 4.1 apresentará a motivação para a escolha de um DSP para a implementação do sistema. A seção 4.2 apresentará as principais características do DSP escolhido para a implementação do projeto. A seção 4.3 apresentará toda a estrutura interna do DSP, apresentando os dispositivos que o mesmo contém, para interagir com dispositivos externos, requisito fundamental para a aplicação proposta. Já a seção 4.4 apresentará a placa protótipo que foi utilizada para a elaboração do projeto. Por fim, a ferramenta de desenvolvimento existente para a programação deste processador será apresentada na seção 4.5.

4.1 Motivação para o Uso de DSPs

Para a implementação de sistemas de processamento digital de sinais, pode-se utilizar qualquer dispositivo digital programável. Os computadores de uso geral (tipo PC), embora sejam bastante rápidos e fáceis de programar, são caros e consomem muita energia, além de não possuírem tamanho reduzido. Uma FPGA (*Field Programmable Gate Array*) possui tamanho reduzido, excelente performance e baixo consumo, mas a implementação de um sistema como o apresentado neste trabalho tomaria uma complexidade tal, neste tipo de dispositivo, que tornaria a escolha deste dispositivo pouco viável.

Em processamento digital de sinais, existem operações que são bastante comuns, como por exemplo somas com acumulação, operações modulares, forte iteratividade, entre outras. Assim sendo, um tipo de processador foi desenvolvido especialmente para aplicações de processamento digital de sinais. Estes processadores, chamados de DSP (*Digital Signal Processors*) exploram características inerentes do processamento digital de sinais, visando otimizar a execução destas e, desta maneira, atender os requisitos em tempo real [21] que aplicações, como a apresentada neste trabalho, possuem. Suas principais qualidades são:

- ***Hardware* dedicado para multiplicar e acumular:** DSPs podem executar uma multiplicação com acumulação em apenas um ciclo, diferentemente de um PC, que normalmente leva entorno de 12 ciclos para realizar a mesma operação. Desta maneira, operações com matrizes, em geral, são executadas eficientemente.

- **Memória interna:** estes dispositivos são dotados de memória interna, com múltiplos barramentos, de forma a permitir acesso veloz, na velocidade do *clock*, a dados e instruções necessários.

- **Implementação em *hardware* de *buffers* circulares:** operações modulares são bastante utilizadas em filtros digitais, desta maneira, DSPs realizam estas operações em *hardware*, para que o dado necessário esteja disponível no próximo ciclo.

- **Implementação de *loops* em *hardware*:** para evitar o *overhead* do teste de

contagem de iterações, DSPs possuem dispositivos especializados em controlar processos iterativos, de maneira que não se gastam ciclos para o controle destes processos.

- **Facilidade de interfaceamento:** como um DSP normalmente entra como parte de um sistema maior, torna-se necessário o envio e recebimento de dados do mesmo. Assim, estes dispositivos são dotados de controladores internos próprios para a conexão de dispositivos externos, como CODECs, memórias, etc. Reduzindo ao mínimo o *overhead* de processamento, ao mesmo tempo em que simplifica o desenvolvimento do código. Além disso, a integração destes controladores reduz o número de componentes do sistema, reduzindo, conseqüentemente, as dimensões e a complexidade do mesmo.

- **Facilidade de codificação:** DSPs são normalmente codificados em C, C++ ou *Assembly*. Além disso, possuem muitas bibliotecas de funções comuns em processamento digital de sinais fornecidas pelo fabricante, de forma que se reduz o tempo necessário para o desenvolvimento de aplicações.

- **Escalabilidade:** estes processadores, quando ficam obsoletos, são sempre substituídos por dispositivos com características de pinagem e tensão equivalentes, de tal maneira que a migração para um novo modelo ocorre, normalmente, com um mínimo de alterações no sistema, o que permite a continuidade da aplicação por vários anos.

4.2 Características Gerais do Processador ADSP-21160M

Para a implementação do sistema proposto, optou-se por utilizar o processador ADSP-21160M da *Analog Devices*. Este processador é um DSP de 32 bits baseado na família de DSPs ADSP-21000, formando um sistema de processamento completo *on-chip*. Ele integra uma memória SRAM interna de duas portas, periféricos de *I/O* e uma unidade de processamento adicional para suporte ao processamento SIMD (*Single Instruction on Multiple Data*) [22], permitindo que a mesma instrução seja realizada em dois dados distintos. Desta maneira, em certas ocasiões,

consegue-se dobrar o número efetivo de operações por ciclo, aumentando consideravelmente a eficiência de processamento, sem a necessidade de aumento do *clock* do processador, evitando, assim, problemas de aquecimento e consumo exagerado de energia.

Este processador apresenta características gerais que o tornam bastante atraente para o projeto proposto neste trabalho, uma vez que atendem a todos os requisitos mencionados no começo deste capítulo. A seguir, estão mencionadas estas características.

- ***Clock* de 80 MHz, com velocidade de pico de 480 *MFlops* (320 *MFlops* sustentado):** este poder de processamento garante que o sistema proposto atenderá os requisitos de tempo real, e ainda suportará futuras modificações que aumentem a complexidade do código.

- **Memória interna com 4 MBits de capacidade total:** para o armazenamento da base de dados, e o processamento destas informações, torna-se necessário um DSP com alta capacidade de memória. A quantidade oferecida por este processador (alta para um DSP) é suficiente para garantir que o sistema não sofrerá por falta de memória, desde que seja bem projetado.

- **Consumo de 2 *watts* de energia:** sendo bastante reduzido, quando comparado com processadores de uso geral (aproximadamente 12 *watts* [23]), tornando o sistema apto a operar com uma bateria de tamanho reduzido, aumentando sua portabilidade.

- **Preço de 175 dólares a unidade:** valor reduzido, quando comparado com outros processadores comerciais, o que reduz significativamente os custos do projeto.

- **Dimensões de 24 x 27 milímetros:** com dimensões reduzidas, aumenta-se a portabilidade do projeto, condição essencial para o desenvolvimento do mesmo.

Na figura 4.1, pode-se observar o diagrama em blocos do processador, ilustrando as seguintes características da arquitetura:

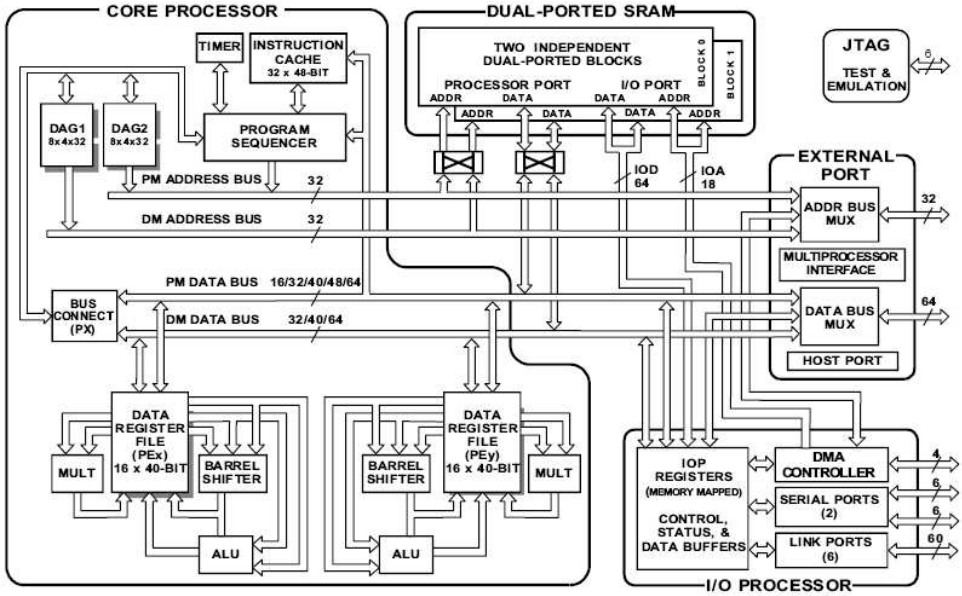

Figura 4.1: Diagrama em blocos do SHARC ADSP-21160M (extraído de [22]).

- Dois elementos de processamento (PEx e PEy), cada um contendo uma unidade lógica aritmética (ALU), um multiplicador e um deslocador (*Shifter*). Todos operando com dados de 32 bits em ponto flutuante, padrão IEEE [24].
- Seqüenciador de programa com *cache* de instruções, temporizadores e geradores de endereços (DAG1 e DAG2).
- Memórias SRAM de duas portas.
- Portas externas para conexão com dispositivos externos como memórias, periféricos, um computador hospedeiro ou um sistema de multiprocessamento.
- Processador de *I/O* com controlador de DMA (*Direct Memory Access*), portas seriais e portas de ligação para comunicações ponto a ponto em sistemas de multiprocessamento.
- *Interface* jTAG para emulação e testes de sistema.

Também se pode observar, na figura 4.1, três barramentos internos: o barramento da memória de programa (PM), o barramento da memória de dados (DM) e o barramento de *I/O* (IO). Durante um único ciclo, estes barramentos permitem ao processador acessar dois operandos (um da memória de programa e outro da

memória de dados), acessar uma instrução que esteja no *cache* e realizar ainda uma transferência via DMA.

Os barramentos conectam-se às portas externas do DSP, permitindo que o processador acesse memórias externas, um processador hospedeiro ou um sistema de multiprocessamento com memória compartilhada. As portas externas realizam a arbitração do barramento e provêem sinais de controle para dispositivos de *I/O* e memórias compartilhadas.

Apresenta-se, na figura 4.2, um sistema típico contendo apenas um processador. Porém, o ADSP-21160M oferece suporte extensivo para sistema de multiprocessamento. Além disso, como DSP atual e eficiente, o ADSP-21160M contempla os cinco requerimentos básicos para processamento digital de sinais:

1. **Aritmética rápida e flexível:** a família de processadores ADSP-21000 executa todas as instruções em um único ciclo. Ela provê tanto um período de *clock* curto, bem como um conjunto completo de operações aritméticas, incluindo *min*, *max*, $1/x$, $1/\sqrt{x}$, etc. O DSP é compatível com o padrão IEEE de ponto flutuante e permite a geração de interrupções no caso de exceções aritméticas.

2. **Fluxo de dados sem restrições:** o ADSP-21160M possui a arquitetura da família SHARC (*Super Harvard Architecture*), combinada com um arquivo de registro de dados de 10 portas, de tal maneira que, em um único ciclo, o processador pode:

   - Ler ou escrever dois operandos no arquivo de registro.
   - Fornecer dois operandos para a ALU.
   - Fornecer dois operandos para o multiplicador.
   - Receber três resultados da ALU e do multiplicador.

3. **Precisão estendida e larga faixa dinâmica nas unidades computacionais:** o ADSP-21160M opera no formato de ponto flutuante de 32 bits no padrão da IEEE, 32 bits em ponto fixo e fracionário (complemento a dois e sem sinal). Além disso, possui precisão estendida de 40 bits (formato IEEE) em ponto flutuante. O processador propaga esta precisão estendida através

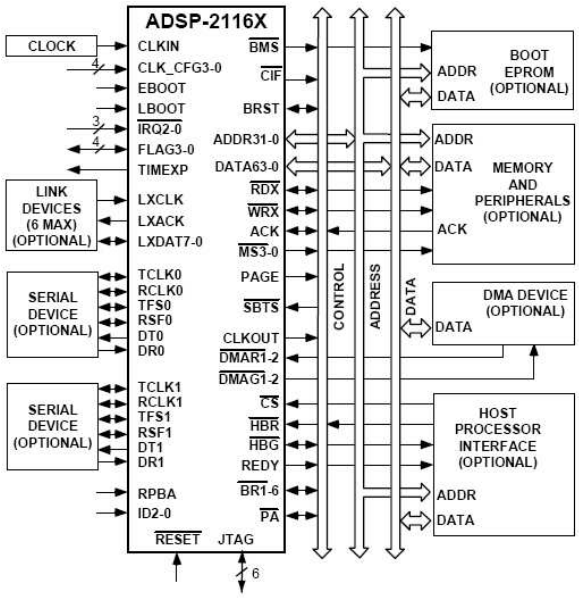

Figura 4.2: Sistema ADSP-21160M (extraído de [22]).

de suas unidades computacionais, limitando erros intermediários devido ao truncamento.

4. **Geradores de endereços duais:** o ADSP-21160M possui dois geradores de endereços (DAGs) que permitem endereçamento imediato ou indireto (com pré ou pós-modificação). Eles permitem operações de módulo, bit-reverso e escrita por irradiação (*broadcasting*), sem restrições quanto a localização do *buffer* de dados.

5. **Seqüenciamento eficiente de programa:** em adição aos *loops* sem *overhead*, o ADSP-21160M permite configurações e saídas de *loops* em um único ciclo. Os *loops* podem ser cascateados (seis níveis em *hardware*) e interrompidos. O processador suporta ainda ramificações atrasadas ou não (*delayed or non-delayed branches*).

4.3 Arquitetura do ADSP-21160M

O ADSP-21160M forma um sistema completo de processamento *on-chip*, integrando uma memória SRAM de alta capacidade e velocidade e periféricos de *I/O* com um barramento dedicado. Esta seção resume algumas das principais carac-

terísticas de cada bloco funcional (vide figura 4.1) do processador ADSP-21160M.

4.3.1 Núcleo de Processamento

O núcleo de processamento do ADSP-21160M consiste de dois elementos de processamento. Cada um com uma unidade lógica e aritmética (ALU), um multiplicador, um deslocador e um arquivo de registro de dados. Além disso, o núcleo de processamento ainda contém um seqüenciador de programa, dois geradores de endereços, um temporizador e um *cache* de instruções. Todo o processamento digital de sinais ocorre no núcleo de processamento.

4.3.1.1 Elementos de Processamento

O núcleo de processamento contém dois elementos de processamento:

- **Elemento de Processamento Primário (PEx):** processa todas as instruções se o DSP se encontra em modo SISD (*Single Instruction on Single Data*). Este elemento de processamento corresponde ao elemento de processamento dos processadores anteriores da família ADSP-21000.
- **Elemento de Processamento Secundário (PEy):** processa sempre as mesmas instruções que o PEx, só que em dados diferentes. Ativado somente quando o modo SIMD do processador é habilitado.

O modo SIMD é útil quando se deseja executar as mesmas instruções em dois conjuntos paralelos de dados, ou, em um caso especial, programas podem usar e estrutura SIMD para executar uma única tarefa (somar dois vetores, por exemplo), dividindo-a em duas partes, que são executadas em paralelo [25].

Cada elemento de processamento contém um arquivo de registro de dados e três unidades computacionais: uma ALU, um multiplicador com um multiplicador em ponto fixo, e um deslocador. Para atender a vasta variedade de necessidades de processamento, as unidades computacionais processam dados em três formatos: ponto fixo de 32 bits e ponto flutuante de 32 e 40 bits. O formato de 32 bits em ponto flutuante está no formato padrão da IEEE, enquanto que o formato de 40 bits possui mais 8 bits na mantissa, para maior precisão.

A ALU executa um conjunto de operações aritméticas e lógicas tanto nos formatos em ponto fixo quanto em ponto flutuante. O multiplicador executa multiplicações e multiplicações com acumulação, tanto em ponto fixo quanto em ponto flutuante. O deslocador executa deslocamentos lógicos e aritméticos, manipulação de bits, depósito e extração de campos e derivação do expoente em operandos de 32 bits. Estas unidades computacionais executam todas as operações em um único ciclo, não existindo *pipeline* de execução. Além disso, todas as unidades computacionais estão conectadas em paralelo, de forma que a saída de uma unidade pode ser entrada de outra no ciclo seguinte. Em operações multifunções, a ALU e o multiplicador executam operações independentes simultaneamente.

Cada elemento de processamento possui um arquivo de registro de dados de uso geral que transfere dados entre as unidades computacionais e os barramentos de dados, além de armazenar resultados intermediários. Um arquivo de registros possui dois conjuntos (primário e alternativo), cada um com dezesseis registradores, para rápida mudança de contexto. Todos os registradores possuem 40 bits. O arquivo de registros, combinado com a arquitetura Harvard do processador permite um fluxo de dados sem restrições entre as unidades computacionais e a memória interna.

4.3.1.2 Controle de Seqüenciamento de Programa

Controles internos para a execução de programas através do ADSP-21160M originam-se de quatro blocos funcionais: seqüenciador de programa, geradores de endereços, temporizador e *cache* de instruções. Dois geradores de endereços e um seqüenciador de programa fornecem endereços para acessos à memória. Juntos, permitem que operações computacionais sejam executadas com máxima eficiência, uma vez que as unidades computacionais podem se ater exclusivamente ao processamento dos dados. Com seu *cache* de instruções, o DSP pode simultaneamente fornecer uma instrução do *cache* e acessar dois dados da memória. Os geradores de endereços implementam *buffers* circulares em *hardware*, sem *overhead*. Algumas funções do controle de seqüenciamento de programa estão listadas a seguir:

1. **Seqüenciador de Programa:** o seqüenciador de programa fornece endereços para a memória de programa. Ele controla iterações de *loops* e avalia instruções condicionais. Com um contador e uma pilha para *loops*, o ADSP-21160M

executa códigos iterativos sem *overhead*. Nenhuma instrução é requerida para decrementar e testar o contador.

2. **Geradores de Endereços de Dados:** os geradores de endereços de dados (DAGs) provêem endereços de memória quando dados são transferidos entre a memória e os registradores. Geradores de endereços duais permitem ao processador calcular dois endereços para a leitura ou escrita de dois operandos. DAG1 fornece endereços de 32 bits para a memória de dados, enquanto que o DAG2 fornece endereços de 32 bits para dados armazenados na memória de programa.

   Cada DAG rastreia até oito ponteiros de endereços, oito modificadores e oito valores de comprimento de vetores. Um ponteiro usado para endereçamento indireto pode se modificado por um valor em registrador de modificação especificado, tanto antes (*pre-modify addressing*) quanto depois (*pos-modify addressing*) do acesso. Um valor de comprimento de vetor pode ser associado a cada ponteiro para implementar endereçamento modular automático para a implementação de *buffers* circulares sem *overhead*, e inicializados em qualquer endereço de memória. Cada registrador do DAG possui um registrador alternativo que pode ser ativado para rápida troca de contexto.

3. **Interrupções:** o ADSP-21160M possui quatro interrupções de *hardware* externas: três interrupções de uso geral ($\overline{\text{IRQ2-0}}$), e uma interrupção especial para *reset*. O processador também possui interrupções geradas internamente para o temporizador, operações de DMA, *overflow* de *buffers* circulares, *overflow* de pilhas, exceções aritméticas, interrupções de multiprocessamento, e interrupções definidas pelo usuário (*software interrupts*).

4. **Mudança de Contexto:** muitos dos registradores do processador possuem registradores alternativos que podem ser ativados durante o tratamento de interrupções, permitindo rápida mudança de contexto, uma vez que um grupo alternativo de registradores evita a necessidade de se salvar todos os registradores em uso no momento do atendimento da interrupção. Os registradores primários estão habilitados no momento de *reset*, enquanto que os secundários (alternativos) são habilitados através da ativação de bits de controle no regis-

trador de controle do processador.

5. **Temporizador:** um temporizador de intervalo programável provê a geração de interrupções periodicamente. Quando habilitado, o temporizador decrementa um registrador de contagem de 32 bits a cada ciclo. Quando este contador chega a zero, o mesmo gera uma interrupção e, em seguida, o registrador de contagem é automaticamente recarregado através do registrador de armazenamento de período e a contagem se reinicia automaticamente.

6. ***Cache* de Instruções:** o seqüenciador de programa contém um *cache* de 32 instruções que permite liberar o barramento de programa para a leitura/escrita de dados que estejam na memória de programa. O *cache* é seletivo. Apenas instruções cuja busca entram em conflito com acesso a dados na memória de programa são colocadas no *cache*, para que o barramento de programa possa ser liberado para o acesso simultâneo a esses dados.

4.3.1.3 Barramento Interno

Para atender os requisitos da arquitetura Harvard, O processador ADSP-21160M possui seis barramentos:

- Endereçamento da memória de programa.
- Dados da memória de programa.
- Endereçamento da memória de dados.
- Dados da memória de dados.
- Endereçamento de *I/O*.
- Dados de *I/O*.

Os barramentos de endereço das memórias de programa e dados transferem, respectivamente, os endereços das instruções e dados necessários, enquanto que os barramentos de dados das memórias de programa e dados transferem, respectivamente, as instruções ou os dados acessados. O barramento de endereços da memória de programa é de 32 bits, podendo acessar 4 giga-palavras de dados e instruções.

O barramento de dados da memória de programa possui comprimento de 64 bits, permitindo a acomodação de instruções de 48 bits e dados de 64 bits.

O barramento de endereço de dados compreende 32 bits, podendo acessar, conseqüentemente, até 4 giga-palavras de dados. O barramento de dados possui comprimento de 64 bits. O barramento de dados da memória de dados provê um caminho para o conteúdo de qualquer registrador no processador que deva ser transferido para qualquer outro registrador ou para qualquer endereço na memória de dados em um único ciclo. O endereço da memória de dados surge de um valor absoluto especificado por uma instrução (endereçamento direto) ou da saída de um gerador de endereço de dados (endereçamento indireto).

Os barramentos de endereçamento e dados de *I/O* permitem ao processador de *I/O* acessar a memória interna para operações de DMA sem gerar atraso ao processador núcleo. O barramento de endereçamento de *I/O* possui comprimento de 32 bits, enquanto que o barramento de dados de *I/O* possui comprimento de 64 bits.

4.3.2 Memória Interna SRAM com Portas Duais

O ADSP-21160M possui 4 MBits de memória SRAM integrada, organizada em dois blocos de 2 MBits cada, que podem ser configurados para diferentes combinações de armazenamento de código e dados. Cada bloco de memória possui duas portas para acesso independente, e em um único ciclo, pelo núcleo do processador e pelo processador de *I/O* ou controlador de DMA. Assim, num único ciclo, pode-se transferir dois dados para o núcleo do processador e um dado para o processador de *I/O*.

Toda a memória do DSP pode ser acessada como palavras de 16, 32, 48 ou 64 bits. Assim, o processador pode conter um máximo de 128 Kpalavras de dados de 32 bits, 256 Kpalavras de dados de 16 bits e 80 Kpalavras de instruções de 48 bits (ou dados de 40 bits), ou qualquer combinação de diferente tamanho, até um máximo de 4 MBits.

O DSP oferece suporte a dados em 16 bits de ponto flutuante, o que efetivamente dobra a quantidade de dados que podem ser armazenados na memória. Conversões entre 32 e 16 bits em ponto flutuante são executadas em um único ciclo.

Embora cada bloco de memória possa armazenar código e dado, acessos de memória geralmente são mais eficientes quando um bloco armazena dados, usando os barramentos de dados e o outro armazena instruções e dados utilizando os barramentos de programa. Utilizando esta implementação, assegura-se a execução em um único ciclo de uma instrução utilizando dois operandos, caso a instrução esteja no *cache*. O DSP também mantém a execução em um único ciclo quando um dos operandos é transferido de ou para fora do *chip*, através da utilização do barramento de *I/O*.

4.3.3 Portas Externas

As portas externas do ADSP-21160M provêem a *interface* necessária para o acesso a memórias externas e periféricos. O espaço de endereçamento externo de 4 giga-palavras é incluído no espaço de endereçamento unificado do DSP. Os barramentos de endereço e dados de *I/O* e da memória de dados e programa são multiplexados nas portas externas, para criar um barramento externo com 32 bits de tamanho para endereços e 64 bits para dados. Memórias externas podem ter comprimento de 16, 32, 48 ou 64 bits, uma vez que o controlador de DMA do DSP automaticamente realiza o empacotamento dos dados para o tamanho adequado durante as transferências.

A decodificação de endereços que ultrapassem os limites da memória interna do processador geram sinais de seleção de bancos de memória para o acesso a dispositivos de memória externa. Linhas separadas de controle permitem endereçamento simplificado de memórias DRAM. O DSP permite ainda controle programável de *waitstates* e de reconhecimento, permitindo a conexão de periféricos com requerimentos de tempos diferentes de acesso, espera e desabilitação.

As portas externas do ADSP-21160M possuem as seguintes interfaces:

4.3.3.1 *Interface* com Hospedeiros

Permite conexão simplificada com barramentos padrões de microcomputadores, tanto com 16 ou 32 bits, com pouco *hardware* adicional. Esta *interface* permite transferências síncronas ou assíncronas com velocidades de até a metade do *clock* interno do processador. Esta *interface* opera através das portas externas e seus

endereços para acesso se encontram mapeados no espaço unificado de endereços. Quatro canais de DMA estão disponíveis para esta *interface*. Tanto a transmissão de dados como a de instruções ocorre com pouco *overhead* de *software*. O computador hospedeiro pode acessar diretamente a memória interna do DSP, podendo também acessar canais de configuração do controlador DMA, servindo como forma de realizar o *boot* do DSP.

4.3.3.2 *Interface* com Sistemas de Multiprocessamento

Sua função é simplificar a implementação de sistemas de multiprocessamento. O espaço unificado de endereços permite acesso direto à memória interna de qualquer ADSP-21160M contido no sistema de multiprocessamento. A arbitração da lógica de barramento do DSP permite a implementação simples de sistemas multiprocessados contendo até seis processadores ADSP-21160M e um processador hospedeiro. Cada processador pode ser o mestre do barramento, e assim, acessar a memória e os registradores de qualquer outro DSP no sistema. A mudança de estado de um processador (de escravo para mestre, por exemplo) ocorre em um único ciclo. Esta *interface* também permite a escrita por irradiação (*broadcast*), permitindo que o DSP mestre escreva simultaneamente na memória de todos os escravos do sistema.

4.3.4 Processador de *I/O*

O processador de entrada/saída (IOP) do ADSP-21160M inclui duas portas seriais, seis portas de ligação e um controlador DMA. Uma das funções de *I/O* que este processador automatiza é o *boot*. O DSP pode ser inicializado pelas portas externas (através dados vindo de uma memória não volátil ou de um computador hospedeiro). Alternativamente, o DSP pode ser não inicializado, executando instruções diretamente da memória externa.

4.3.4.1 Portas Seriais

O ADSP-21160M contém duas portas seriais síncronas que permitem uma maneira simples e eficaz de conectar o DSP a dispositivos externos como CODECs, entre outros. As portas seriais podem operar a uma velocidade máxima de metade do *clock* do processador. Funções independentes de transmissão e recepção oferecem

grande flexibilidade para comunicações seriais. As portas seriais podem ainda transferir dados de ou para a memória interna usando DMA. Cada porta serial oferece modo multicanal TDM (*Time Division Multiplexing*) e ainda permite compressões *A-law* e μ-*law*.

As portas seriais podem organizar os dados a serem transferidos tanto no formato *big-endian* quanto *little-endian*, em palavras de tamanho entre 3 e 32 bits. Elas oferecem, também, modos de transmissão e sincronização ajustáveis. Por fim, o *clock* de transferência e os pulsos de sincronismo podem ser gerados internamente ou externamente.

4.3.4.2 Portas de Ligação

O ADSP-21160M possui seis portas de ligação de 8 bits, permitindo capacidades adicionais de *I/O*. Portas de ligação são especialmente úteis em comunicações ponto-a-ponto entre DSPs em sistemas de multiprocessamento, como se pode observar na figura 4.3. As portas de ligação podem operar independentemente ou simultaneamente, e cada uma pode ser configurada tanto para recepção, quanto para transmissão de dados. Elas possibilitam, ainda, que os dados sejam empacotados em palavras de 32 ou 48 bits, permitindo que o núcleo do processador acesse diretamente os dados, ou que os mesmos sejam transmitidos diretamente para a memória interna, via DMA.

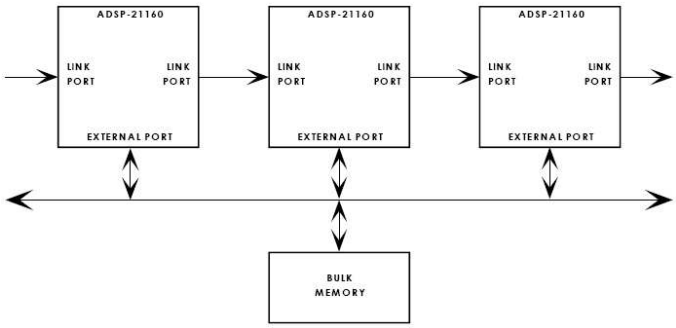

Figura 4.3: Exemplo de sistema de multiprocessamento utilizando portas de ligação (extraído de [22]).

4.3.4.3 Controlador de DMA

O controlador de DMA do ADSP-21160M permite transferência de dados sem *overhead* e sem a intervenção do núcleo do processador, operando de maneira independente e invisível para o mesmo. Tal controlador permite que o núcleo do processador execute seu programa enquanto novos dados são transferidos de/para a memória interna do DSP, estando assim, disponíveis para o núcleo do processador quando o mesmo precisar destes.

Uma transferência DMA pode ocorrer entre a memória interna do ADSP-21160M e a memória externa, periféricos externos ou um processador hospedeiro. Transmissões DMA também podem ocorrer entre a memória interna e as portas de ligação ou entre as portas seriais. Outra opção é a transmissão de dados entre a memória externa e algum outro dispositivo externo. Em qualquer caso, os dados podem ser empacotados em palavras de 16, 32, 48 ou 64 bits, processo este que é feito automaticamente pelo controlador DMA.

Quatorze canais de DMA estão disponíveis no ADSP-21160M. Seis deles para as portas de ligação, quatro para as portas seriais, e quatro para as portas externas do DSP. Os canais de DMA destinados às portas externas servem para transferência de ou para um computador hospedeiro, outros processadores DSPs ou transferências gerais de *I/O*.

4.3.5 *Interface* jTAG

A *interface* jTAG do ADSP-21160M aceita o padrão 1149.1 *Joint Test Action Group* do IEEE para testes de sistema. Este padrão define um método para serialmente vasculhar o estado de *I/O* de cada componente do sistema. Emuladores usam a *interface* jTAG para monitorar e controlar o DSP durante a emulação do mesmo. Emuladores utilizando esta *interface* provêem emulação à máxima velocidade, podendo inspecionar ou modificar a memória, registradores e pilhas do processador. A emulação baseada em jTAG é não intrusiva e não afeta a carga e nem a execução de programas no DSP.

4.4 Placa de Avaliação ADSP-21160M EZKIT Lite

Quando se desenvolve um sistema baseado em DSP, inicialmente avalia-se o desempenho do processador escolhido através de simulações em *software* do mesmo. Entretanto, a simulação não permite ao usuário avaliar de maneira significativa como o DSP escolhido irá interagir com os outros componentes do sistema.

Para atender a essa necessidade, surgem placas de avaliação, que são produzidas pelo fabricante ou parceiros e, por isso, podem possuir custo bastante reduzido[1]. Assim, o desenvolvedor pode desenvolver seu protótipo de sistema DSP utilizando estas placas de avaliação, que permitirão a conexão do DSP com os outros dispositivos integrantes do sistema.

A placa de avaliação ADSP-21160M EZKIT Lite é apresentada na figura 4.4. Esta placa contém um processador ADSP-21160M para avaliação. Ela funciona como uma *interface* entre o DSP e dispositivos externos, facilitando o desenvolvimento de protótipos. Além disso, ela já disponibiliza alguns recursos comumente utilizados em projetos baseados em DSP, como:

- Módulos de memória externa (total de 4 MBits).
- Unidade de memória *Flash* (total de 4 MBits).
- CODEC para processamento de áudio.
- Conector jTAG
- *LEDs* de uso geral, controlados por *software*.
- Botões conectados às interrupções em *hardware* do processador, para geração manual de interrupções.
- Conectores para portas de ligação e portas seriais.
- Conectores para conexão com qualquer outro dispositivo que não pertença a placa.

[1] O custo da placa utilizada neste trabalho é da ordem de 500 dólares [26].

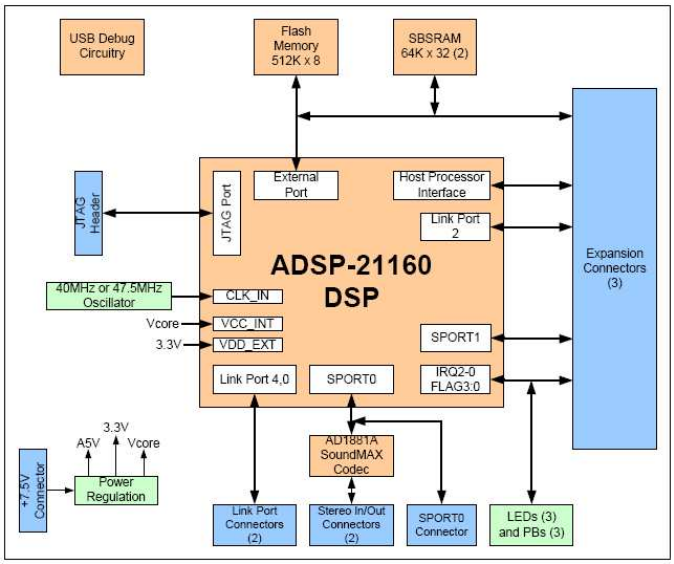

Figura 4.4: Diagrama em blocos da placa ADSP-21160M EZKIT Lite (extraído de [27]).

4.5 Ferramenta de Desenvolvimento

Para o desenvolvimento de projetos utilizando DSPs, é fundamental a utilização de uma ferramenta que simplifique ao máximo esta tarefa, reduzindo assim, o tempo necessário para a elaboração do projeto.

O *VisualDSP++* é uma ferramenta de desenvolvimento que integra em um único ambiente a edição, compilação e depuração de um projeto DSP, simplificando ao máximo o projeto através de *interfaces* visuais, permitindo maior escalabilidade ao mesmo [28].

Na figura 4.5 tem-se a visão geral da ferramenta de desenvolvimento. As principais funcionalidades da mesma são:

- **Atividades de depuração simplificadas:** é possível depurar sistemas através de um única *interface* para todos os simuladores, emuladores ou avaliações de *hardware* de sistemas customizados.

- **Suporte a várias linguagens:** pode-se desenvolver códigos em C/C++ ou *Assembly*, este último, algébrico, facilitando a inteligibilidade do código. Pode-se ainda observar o código em linguagem de máquina. Para programas desenvolvidos em C/C++, pode-se observar o fonte em C/C++, ou em modo misto

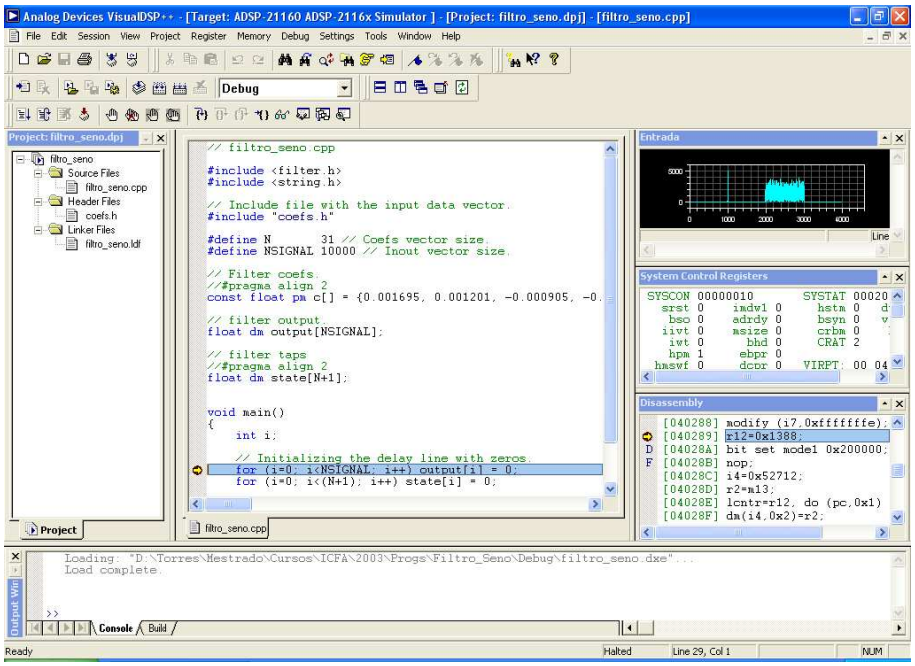

Figura 4.5: Ambiente de desenvolvimento *VisualDSP++*.

(C e *Assembly*).

- **Controle eficiente de depuração:** pode-se inserir *breakpoints* ao longo do código, e assim, executá-lo passo a passo, buscando por possíveis erros de programação. Pode-se também inserir *breakpoints* condicionais em registradores, pilhas e posições de memória, para saber quando os mesmos são acessados.

- **Ferramentas para otimização de código:** pode-se utilizar ferramentas de *profiling* para identificar gargalos de execução, e assim, determinar pontos que necessitem de otimização. Pode-se apresentar vetores na forma de gráficos e utilizar interrupções de entrada e saída utilizando arquivos para simular aplicações em tempo real.

- **Depuração de sistemas multiprocessados:** pode-se depurar sistemas baseados em vários processadores utilizando uma única *interface*. Pode-se inserir *breakpoints* para execução passo a passo sincronizada com todos os processadores do sistema.

- **Bibliotecas de funções otimizadas:** a ferramenta de desenvolvimento apresenta um conjunto de bibliotecas que implementam funções comuns em pro-

cessamento digital de sinais, como *FFT*, filtros *FIR* e *IIR*, operações com matrizes, entre outras. Todas as funções são escritas em *Assembly* e altamente otimizadas, explorando todos os recursos do processador, e podem ser chamadas diretamente de um programa feito em C/C++.

4.6 Conclusão

Neste capítulo, foi apresentado o dispositivo digital a ser utilizado na elaboração do sintetizador de voz. As características apresentadas neste capítulo (consumo, performance, dimensões, preço e recursos integrados) mostram que este processador atende todos os requisitos exigidos pelo projeto. No próximo capítulo será abordada a metodologia utilizada para fazer com que o projeto possa ser encaixado nas especificações do DSP apresentado, no que diz respeito à automação e requerimentos de memória.

Capítulo 5

Análise do Projeto

Até este ponto do trabalho, apresentou-se os conceitos teóricos necessários para a implementação do sistema proposto. Neste capítulo, será elaborado o algoritmo do sintetizador a ser implementado em DSP, de forma que a teoria abordada até então forneça ferramentas suficientes para a realização do projeto. A elaboração do algoritmo TD-PSOLA a ser utilizado neste trabalho será apresentada na seção 5.1. Como visto no capítulo 2, têm-se o problema da base de dados ser bastante volumosa, não podendo ser armazenada na placa de desenvolvimento do DSP, conforme visto no capítulo 4. Desta maneira, este problema será contornado através da compressão da base de dados, conforme será visto na seção 5.2 deste capítulo.

5.1 Elaboração Prática do Algoritmo de Síntese

A versão inicial deste algoritmo provém do projeto "Antígona", que é um projeto de *interface* de voz para o português europeu a ser utilizado em comércio eletrônico [29]. No âmbito da colaboração internacional entre a Escola de Engenharia da Universidade Federal do Rio de Janeiro e a Faculdade de Engenharia da Universidade do Porto, teve-se acesso à implementação base, em Matlab, do TD-PSOLA. Esta seção descreverá as partes integrantes desta implementação, bem como algumas adaptações propostas para este sistema, introduzidas neste trabalho.

5.1.1 Base de Dados

Como visto, o TD-PSOLA é um algoritmo de síntese por concatenação de difones, de tal maneira que é necessária uma base de dados contendo todos os arranjos tomados dois a dois dos 37 fonemas mais importantes da língua portuguesa (a lista com os fonemas portugueses considerados neste trabalho esta apresentada na tabela 5.1). Além disso, para a inserção da prosódia, torna-se necessário também a existência de um vetor contendo a posição das marcas de *pitch* de análise dos fonemas vozeados, uma vez que as janelas são determinadas por estas marcas. Também se faz necessária a existência de um vetor com a informação de vozeamento de cada fonema de um difone, para que se possa determinar o tipo de janela que será utilizada. Por fim, deve-se também armazenar a posição de início do segundo fonema do difone. Assim, na figura 5.1 pode-se observar, graficamente, como estas informações estão armazenadas, para melhor compreensão da relação entre elas.

A base de dados foi obtida do projeto "Antígona" [29], dada a qualidade da mesma. Para o desenvolvimento desta base [30], inicialmente, um texto contendo várias realizações de todos os difones existentes considerados (total de 922 difones) foi pronunciado por um orador profissional, em estúdio, e posteriormente digitalizado, com freqüência de amostragem de 22,05 kHz e com resolução de 16 bits. Em seguida, foi feito o recorte manual de cada difone. O vozeamento foi determinado, e as marcas de *pitch* inseridas através de métodos automáticos. Como resultado final, tem-se uma base de dados de alta qualidade, porém, exigindo uma elevada quantidade de memória (aproximadamente 23 Mbytes) para o seu armazenamento, além de também exigir muita memória para o conseqüente processamento dos difones nela armazenados.

5.1.2 Descrição do Sintetizador Original

No trabalho desenvolvido, como já dito, foi utilizado o algoritmo TD-PSOLA para a realização da síntese, de acordo com a teoria apresentada no capítulo 2. Assim, a versão do sistema de síntese original, recebida da colaboração UFRJ / FEUP, é dada pelo fluxograma apresentado na figura 5.2. As descrições dos principais blocos funcionais desta versão original encontram-se abaixo:

Tabela 5.1: Lista dos fonemas portugueses considerados para a geração da base de dados.

Índice	**Classe**	**Símbolo SAMPA**	**Exemplo**
01	Silêncio	X	N/A
02	Oclusivas	p	*p*ai
03	Oclusivas	b	*b*ar
04	Oclusivas	t	*t*ia
05	Oclusivas	d	*d*ata
06	Oclusivas	k	*c*asa
07	Oclusivas	g	*g*ato
08	Fricativas	f	*f*érias
09	Fricativas	v	*v*aca
10	Fricativas	s	*s*elo
11	Fricativas	z	a*z*ul
12	Fricativas	S	*ch*ave
13	Fricativas	Z	a*g*ir
14	Nasais	m	*m*eta
15	Nasais	n	*n*eta
16	Nasais	J	se*nh*a
17	Líquidas	I	*i*odo
18	Líquidas	I~	sa*l*
19	Líquidas	L	fo*lh*a
20	Vibrantes	r	ca*r*o
21	Vibrantes	R	ca*rr*o
22	Vogais	i	f*i*ta
23	Vogais	e	p*e*ra
24	Vogais	E	s*e*ta
25	Vogais	a	c*a*ro
26	Vogais	6	cam*a*
27	Vogais	O	*c*orda
28	Vogais	o	s*o*pa
29	Vogais	u	m*u*da
30	Vogais	@	dest*e*
31	Vogais	i~	p*in*ta
32	Vogais	e~	m*en*ta
33	Vogais	6~	m*an*ta
34	Vogais	o~	p*on*ta
35	Vogais	u~	m*un*dial
36	Semivogais	j	pa*i*
37	Semivogais	w	pa*u*

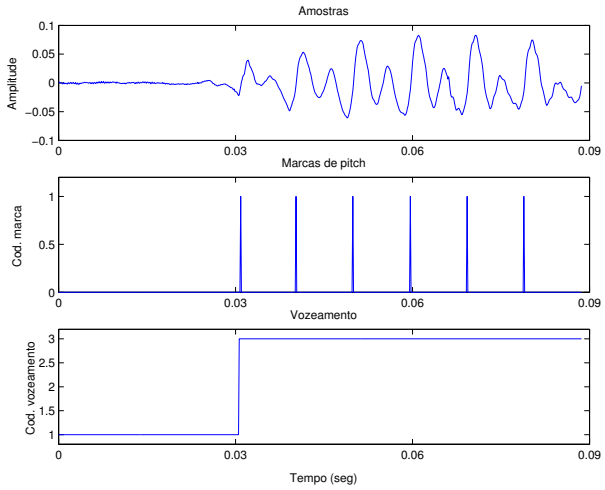

Figura 5.1: Representação gráfica das informações correspondentes ao difone "Xd" armazenadas na base de dados.

1. **Introduza o texto fonético a sintetizar:** Neste momento, o usuário do sistema digita o texto a ser sintetizado. Entretanto, como não há, nesta versão, um módulo de conversão grafema-fonema, o texto a ser introduzido já precisa estar convertido em caracteres do alfabeto fonético SAMPA.

2. **Identificação do próximo difone:** este bloco verifica na *string* de caracteres fonéticos, qual será o próximo par de fonemas que será processado.

3. **Identificação dos seus fonemas:** os fonemas do par selecionado no passo anterior são separados, e o índice correspondente à posição de cada um no vetor de fonemas (ver tabela 5.1) é determinado.

4. **Busca do difone na base de dados:** uma vez que os índices dos fonemas do difone foram selecionados, estes são utilizados para a localização das informações do difone na base de dados.

5. **Concatenação das informações do difone nos vetores de saída:** como a síntese é realizada no conjunto de dados como um todo, deve-se concatenar as amostras de cada difone em um único vetor de amostras. O mesmo deve ser feito para os vetores de marcas de *pitch* e de informação de vozeamento. Para facilitar a execução do algoritmo TD-PSOLA propriamente dito, esta etapa do algoritmo insere sempre uma marca de *pitch* no fim dos fonemas, e insere,

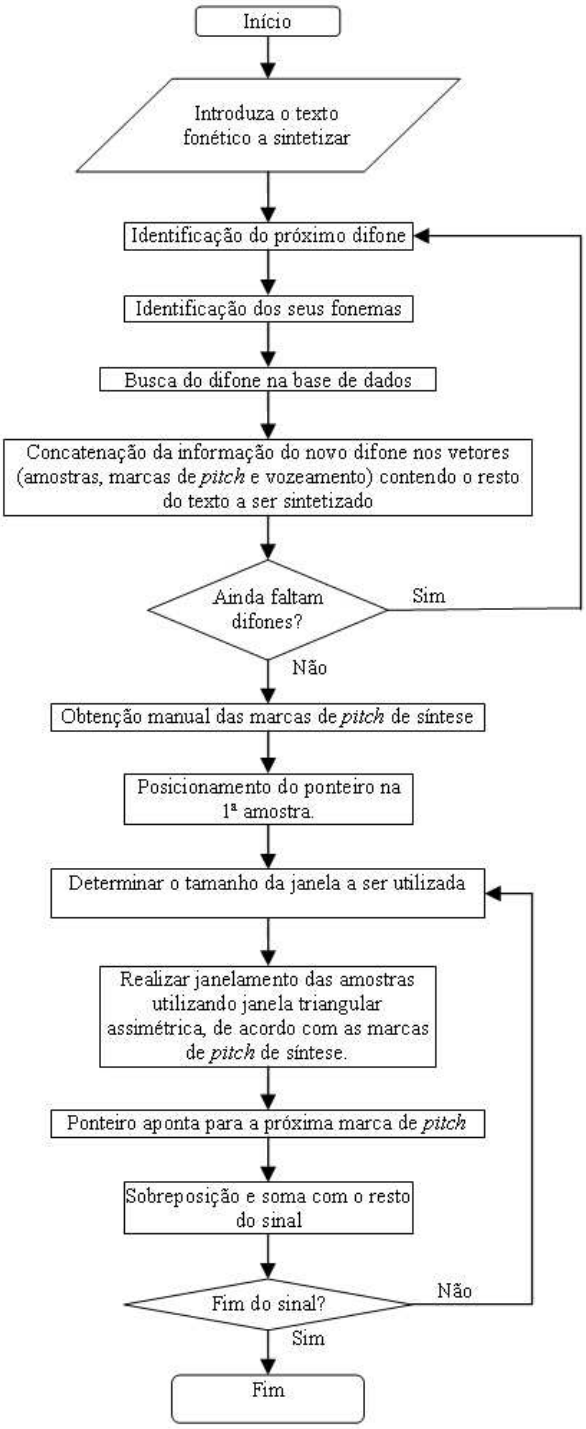

Figura 5.2: Fluxograma do algoritmo original de síntese baseado em TD-PSOLA.

também, uma no início dos fonemas não vozeados, caso estes sejam o primeiro fonema do difone em questão.

6. **Ainda faltam difones:** Verifica se ainda existem difones no texto fonético aguardando para serem processados. Em caso afirmativo, o algoritmo volta para o passo 2. Do contrário, vai para o próximo passo.

7. **Obtenção manual das marcas de *pitch*:** como esta versão inicial não gera a curva de *pitch* automaticamente, uma tela é apresentada ao usuário, para que o mesmo desenhe a curva desejada. O sistema, a partir da interpolação desta curva, obtém as marcas de *pitch* de síntese, que fornecem ao TD-PSOLA a informação de prosódia que este necessita.

8. **Posicionamento do ponteiro na primeira amostra:** o algoritmo posiciona os ponteiros dos vetores de amostras, marcas de *pitch* de análise, marcas de *pitch* de síntese, informação de vozeamento e vetor de saída em suas posições iniciais, preparando para a execução do algoritmo TD-PSOLA propriamente dito.

9. **Determinar tamanho da janela:** o algoritmo determina o comprimento da janela a ser utilizada. Este comprimento é igual a $p_s(i+2) - p_s(i)$, onde $p_s(i)$ é o índice correspondente a i-ésima marca de *pitch* de síntese, de forma que o pico da janela ocorre na posição dada por $p_s(i+1)$.

10. **Realizar janelamento das amostras:** nesta etapa, utiliza-se uma janela triangular assimétrica, uma vez que o número de amostras do lado esquerdo da marca de *pitch* $(p_s(i+1) - p_s(i))$ pode ser diferente do número de amostras do lado direito $(p_s(i+2) - p_s(i+1))$. A janela é, então, centrada na amostra do vetor de amostras correspondente a posição de $p_a(i+1)$, onde $p_a(i)$ é o índice correspondente a i-ésima marca de *pitch* do vetor de análise. Por fim, uma vez determinado o comprimento e a posição da janela, o janelamento das amostras desejadas nesta etapa, é, finalmente, realizado.

11. **Atualização do ponteiro:** uma vez que a informação para àquela janela foi extraída, os ponteiros dos vetores de marca de *pitch* de síntese e de análise são

incrementados em uma unidade, de forma a apontar para o próximo índice correspondendo a próxima marca de *pitch*.

12. **Sobreposição e soma:** a janela obtida é somada com sobreposição (a metade esquerda da mesma sobreposta à metade direita da janela anterior). O ponteiro do vetor de saída aponta, então, para a amostra cuja posição é dada por $p_{tr} = p_a(i+1) + 1$.

13. **Fim do sinal:** caso não hajam mais marcas de *pitch*, o sistema considera encerrada a síntese, e reproduz a fala resultante. Do contrário, retorna ao passo 9.

5.1.3 Adaptações Propostas para o Sintetizador Original

A versão original do algoritmo possui algumas características que o tornariam inviável para sua utilização no sistema proposto neste trabalho, como a necessidade de geração manual da curva de *pitch* de síntese, e a necessidade de inserção de texto fonético. Assim, uma versão adaptada, mas ainda baseada na versão inicial foi elaborada. A seguir, segue a descrição dos blocos da nova versão, cujo fluxograma pode ser observado na figura 5.3, que foram inseridos, ou modificados, em relação à versão original.

1. **Introdução do texto silábico:** neste bloco, permite-se a apresentação de texto silábico para o sintetizador, permitindo que qualquer pessoa possa utilizá-lo.

2. **Conversão grafema-fonema:** como o algoritmo PSOLA utiliza como unidade segmental o difone, torna-se necessária a conversão do texto silábico apresentado para texto fonético. Desta maneira, este bloco realiza esta conversão. Além disso, este bloco também marca os fonemas correspondentes às sílabas tônicas. Este processo é realizado de acordo com o algoritmo descrito em [31].

3. **Busca do difone na base de dados:** Como a base de dados se encontra compactada, a extração das informações segue várias etapas, graças a oti-

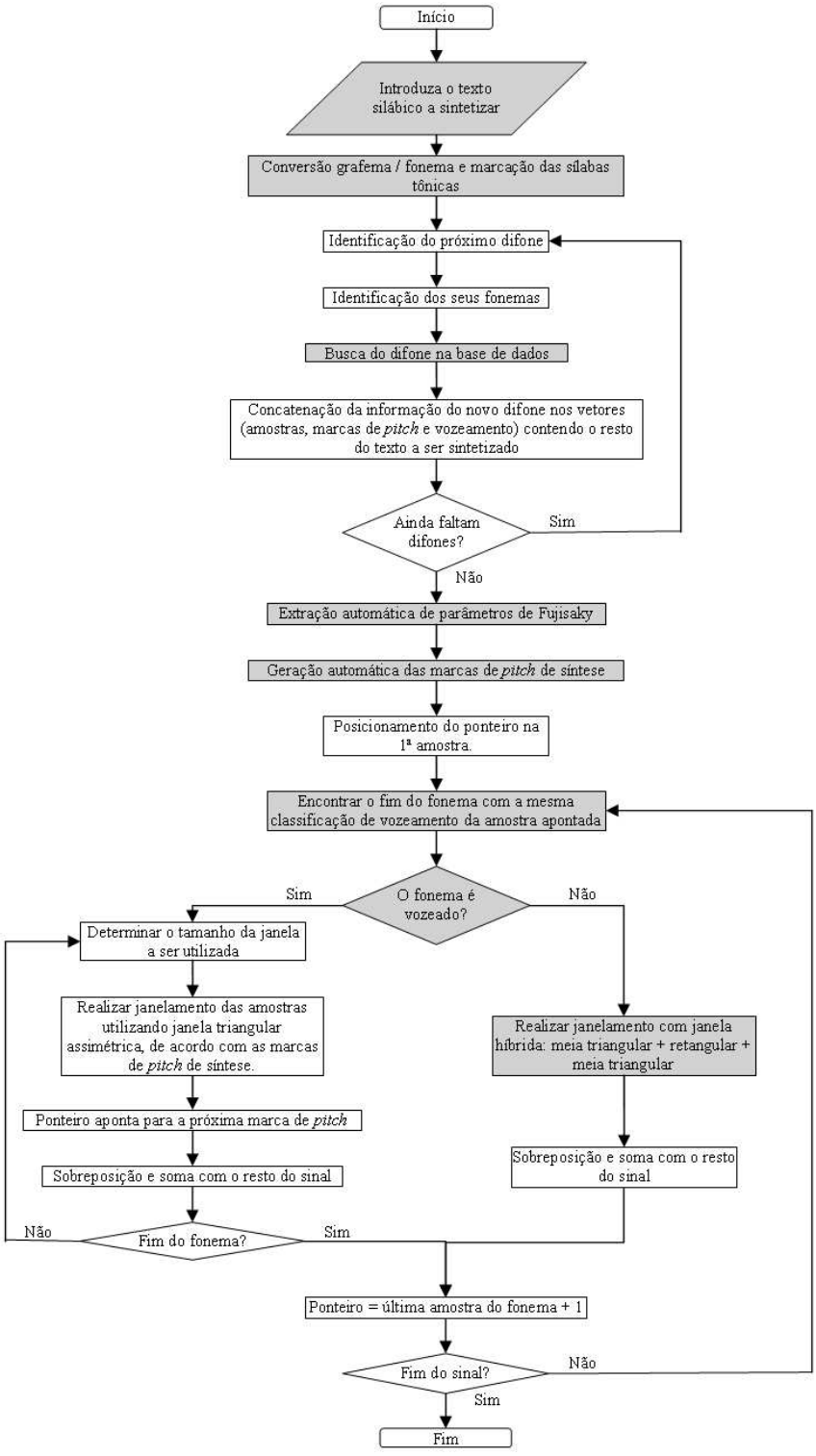

Figura 5.3: Fluxograma do algoritmo modificado de síntese baseado em TD-PSOLA. Os blocos em cinza representam funcionalidades que foram melhoradas, ou que foram adicionadas à versão original.

mização do espaço necessário para o armazenamento da mesma. Esta etapa será descrita, em detalhes, na seção 5.2.

4. **Extração automática dos parâmetros de Fujisaki:** a curva de *pitch* é gerada automaticamente. Para tal, uma implementação bastante simples do modelo de Fujisaki foi implementada. Os parâmetros são extraídos segundo os seguintes critérios:

   - Existe apenas um único comando de frase G_p, posicionado sempre no início da frase, com $A_p = 0,02$ e $\alpha = 0,03$.
   - Os comandos de acento estão posicionados nas sílabas tônicas da frase, de forma que T_1 é o instante de tempo correspondente ao início do primeiro fonema pertencente a uma sílaba tônica, e, T_2, ao fim do último fonema pertencente a esta mesma sílaba. O valor A_a é sempre igual a 1,1, e por fim, tem-se $\gamma = 0,2$ e $\beta = 0,005$.
   - A freqüência de polarização F_b é constante e igual a 100.

5. **Geração automática da curva de *pitch* de síntese:** uma vez obtidos os parâmetros de Fujisaki, a curva de *pitch* desejada é gerada de acordo com o método descrito no capítulo 2.

6. **Encontrar o fim do fonema com o mesmo vozeamento:** como esta versão não insere marcas de *pitch* adicionais, o passo do processo não é mais dado por estas marcas, e sim pela informação de vozeamento, de tal forma que se processa um fonema por ciclo de processamento. Este bloco, então, seleciona as amostras (e as marcas de *pitch* de análise e de síntese) correspondentes ao fonema a ser processado nesta etapa.

7. **O fonema é vozeado:** De acordo com a informação de vozeamento, este bloco determina como o fonema corrente será processado. Caso seja considerado vozeado, o processamento das amostras deste fonema é equivalente ao realizado na versão original.

8. **Janelamento com janelas híbridas:** para os fonemas não vozeados, uma janela retangular está sendo utilizada, de forma que ocorre a mera cópia das

amostras não vozeadas, reduzindo-se o *overhead* de processamento, uma vez que se diminui o número de operações aritméticas. Entretanto, se o fonema em questão for precedido por um fonema vozeado, a janela possui, do seu lado esquerdo, meia janela triangular. Se o fonema seguinte também for vozeado, o lado direito de uma janela triangular é acrescentado à direita da janela retangular, formando um janela híbrida, como pode ser observado na figura 5.4.

Como se percebe nesta nova versão, uma vez recebido o texto, o sistema realiza todo o processo de síntese automaticamente, sem qualquer tipo de intervenção humana, de forma que esta versão atende, agora, os requisitos de automação desejados para a implementação do projeto proposto neste trabalho.

5.2 Compressão da Base de Dados

Nesta seção, será apresentada a estratégia utilizada para a compressão total da base de dados apresentada na seção 5.1. Esta base de dados original foi gerada com freqüência de amostragem de 22.050 Hz e possuía a peculiaridade de conter o vetor de marcas de *F0* e de informação de vozeamento com comprimento igual ao vetor de amostras do difone, o que simplificava ligeiramente o algoritmo, mas requeria uma quantidade muito grande de memória. Seguindo estas características, a base de dados na memória do DSP conterá, inicialmente, os seguintes parâmetros:

- Identificador do primeiro fonema do difone.
- Identificador do segundo fonema do difone.
- Total (N) de amostras do difone.
- Posição do início do segundo fonema.
- Vetor de informação de vozeamento.
- Vetor de amostras de áudio.
- Vetor de marcas de *F0*.

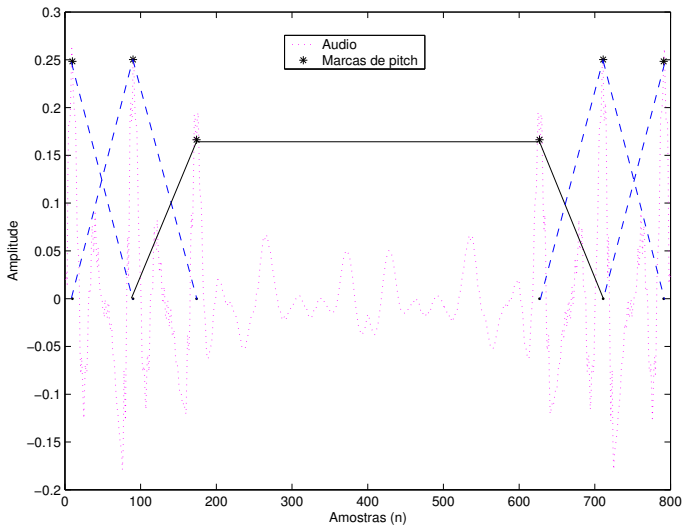

Figura 5.4: Visualização da janela híbrida (linha cheia) utilizada quando um fonema não vozeado encontra-se entre dois vozeados. As janelas triangulares (linhas tracejadas) são as janelas assimétricas utilizadas em fonemas vozeados. Ao fundo (linha pontilhada), pode-se visualizar as amostras de áudio a serem janeladas.

Como a palavra de dados do DSP em questão (ver capítulo 4) possui 32 bits[1], cada valor desta base de dados possui 32 bits de comprimento. Assim, está apresentada na tabela 5.2 a quantidade total de bits gasta para armazenar cada parâmetro da base de dados original. Como fica claro, a quantidade total de memória requeria (23,3 Mbytes) tornaria a implementação em DSP demasiada complexa, dada a enorme quantidade de memória que seria necessária para conter toda essa base de dados.

A solução para este problema foi estudar a composição da base de dados, procurando informações que não sejam necessárias ou redundantes. Assim, as seguintes modificações foram feitas:

1. As amostras do difone foram inicialmente reamostradas a 8kHz de freqüência de amostragem (com o conseqüente recálculo da posição das marcas de *F0* e do início do segundo fonema) e, em seguida, codificadas utilizando o sistema

[1] De fato, o DSP pode endereçar palavras de comprimento diferentes, entretanto, seria necessário criar seções distintas de memória para dados de comprimentos diferentes, o que aumentaria a complexidade de acesso às informações da base de dados, bem como se diminuiria a versatilidade de utilização da memória.

Tabela 5.2: Distribuição dos bits na base de dados original.

Parâmetro	**Tamanho por Difone (bits)**	**Total da Base de Dados (bits)**
Id do primeiro fonema	32	29.504
Id do segundo fonema	32	29.504
Total de amostras (N)	32	29.504
Índice do segundo fonema	32	29.504
Vozeamento	$N \times 32$	62.165.120
Amostras do difone	$N \times 32$	62.165.120
Marcas e *F0*	$N \times 32$	62.165.120
Total	N/A	186.642.880

CELP apresentado no capítulo 3, com todas as melhorias propostas. Assim, além do tamanho total do difone, armazenou-se também o número de janelas que compõem um dado difone.

2. O vetor de vozeamento foi reduzido a um tamanho fixo de apenas dois valores de vozeamento, um para cada fonema. E cada valor está representado por apenas 3 bits.

3. O vetor de marcas de *F0*, originalmente, era um vetor de comprimento igual ao vetor de amostras do difone, com seus valores iguais a "1" nos índices correspondentes a uma marca de *F0* no vetor de amostras, e zero nos outros índices (ver figura 5.1). Como o importante é a posição destes valores unitários, decidiu-se guardar somente o índice destes, de forma que, supondo um vetor de marcas de *F0* $f = [0;0;0;0;1;0;0;0;0;0;1;0;0;0;0;0;0;0;1]$, o mesmo seria reduzido para um vetor $f' = [4;10;18]$, reduzindo drasticamente o tamanho do mesmo, uma vez que a maioria dos valores no vetor de marcas de *F0* são nulos. Além disso, foi alocado para cada valor apenas 16 bits, de forma que cada par de índices ocupa uma palavra de 32 bits. Caso o número total de índices seja ímpar, o último índice ocupará uma palavra inteira. Assim, torna-se necessário armazenar, também, o número total de índices, para que seja possível determinar se a última palavra contém um (total de índices ímpar) ou dois (total de índices par) índices. Para reduzir o *overhead* de processamento, armazenou-se também o número de palavras de 32 bits (blocos) de marcas de *F0*, já que houve sobra de espaço, como será explicado mais a frente.

4. Estudando a base de dados, observou-se que 14 bits eram suficientes para

armazenar o índice do começo do segundo fonema, o que gerou uma economia de 18 bits por difone.

5. Para os parâmetros referentes ao tamanho dos vetores e à identificação de cada fonema que compõe o difone, foram utilizados apenas o número de bits necessários. Este número foi determinado através da observação da base de dados e pela determinação do valor máximo de cada parâmetro.

Assim, após todas as modificações acima propostas, gerou-se a nova base de dados, cujos parâmetros contidos na mesma podem ser observados na tabela 5.3. Como pode-se perceber, houve uma drástica redução nos requisitos de memória da base de dados como um todo. A base final possui apenas 190,0 kbytes, o que representa uma redução de 99,19%, quando comparada com a base de dados original.

A inserção na base de dados dos parâmetros referentes ao número de janelas de cada difone e do número de palavras de 32 bits (blocos) contendo marcas de *F0* pode parecer redundante a princípio. Entretanto, somando o número de bits utilizados para representar os outros seis valores[2] de comprimento inferior a 32 bits, chegamos ao total de 52 bits necessários. Como se deve utilizar múltiplos de 32 bits, uma vez que este é o tamanho da palavra de dados do DSP, 64 bits seriam necessários para armazenar somente estes 52 bits, de tal maneira que se optou por inserir estas duas informações extras, uma vez que não resultaria no aumento da base de dados, e tal inserção reduziria ligeiramente o *overhead* de processamento, dado que estes valores já estariam armazenados e não precisariam ser calculados em tempo de execução.

A geração desta base de dados final não foi obtida diretamente, e sim em etapas, onde cada etapa trabalhava em um determinado aspecto da base de dados, de maneira a reduzir o tamanho da mesma. A etapa seguinte, conseqüentemente, herdava todas as modificações introduzidas pelas etapas anteriores, e aplicava suas próprias modificações. Assim, um total de cinco versões para a base de dados foram geradas, resultando na progressiva redução da mesma, como fica claro na figura 5.5, onde é apresentado o tamanho em bits de cada versão. Abaixo, segue a descrição de cada versão de base de dados gerada.

[2]Identificador e vozeamento do primeiro e segundo fonema (4 valores), número de marcas de *F0* (1 valor) e índice de início do segundo fonema (1 valor).

Tabela 5.3: Distribuição dos bits na base de dados compactada.

Parâmetro	**Tamanho por Difone (bits)**	**Total da Base de Dados (bits)**
Id do primeiro fonema	6	5.532
Id do segundo fonema	6	5.532
Total de amostras	14	12.908
Total de janelas (N_j)	6	5.532
Marcas de *F0*	6	5.532
Blocos de *F0* (N_b)	6	5.532
Índice do segundo fonema	14	12.908
Vozeamento	6	5.532
Marcas de *F0*	$N_b \times 32$	100.160
Coeficientes LPC	$N_j \times 32$	155.584
Ganhos adaptativos	$N_j \times 32$	155.584
Ganhos fixos	$N_j \times 32$	155.584
Índices do dicionário adaptativo	$4 \times N_j \times 16$	311.168
Índices do dicionário fixo	$4 \times N_j \times 16$	311.168
Dicionário adaptativo	4135×32	132.320
Dicionário fixo	4135×32	132.320
Matriz de quantização LPC	$(2 \times 16 + 8 \times 8) \times 32$	3.072
Matriz de quantização dos ganhos adaptativos	64×32	2.048
Matriz de quantização dos ganhos fixos	64×32	2.048
Total	N/A	1.520.064

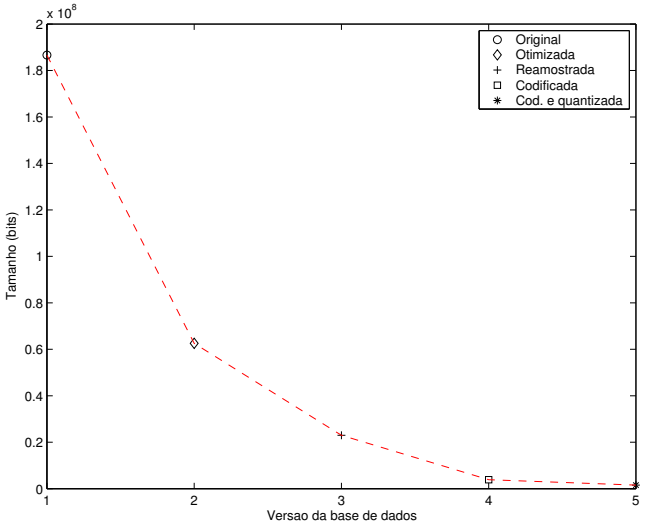

Figura 5.5: Curva de decaimento do tamanho total da base de dados.

1. **Base de Dados Original:** é a base inicial recebida, sem qualquer tipo de modificação.

2. **Base de Dados Otimizada:** versão onde os vetores de marcas de *F0* e de vozeamento foram reestruturados para terem seus tamanhos reduzidos, sem perda de informação, conforme anteriormente mencionado nesta seção.

3. **Base de Dados Reamostrada:** nesta versão, a base de dados foi reamostrada a 8kHz, reduzindo o número de amostras de áudio do difone.

4. **Base de Dados Codificada:** neste ponto, foi aplicado o sistema de codificação CELP, porém, sem nenhuma das modificações propostas no capítulo 3.

5. **Base de Dados Codificada e Quantizada:** a implementação final, com a base de dados codificada em CELP, e os coeficientes LPC e de ganho quantizados.

Por fim, observa-se na tabela 5.4 apresenta o tamanho total de cada versão da base de dados, o tamanho percentual de uma versão em relação a versão anterior, e na última coluna, a representação percentual de uma versão em relação a versão original, mostrando que cada etapa foi capaz de aplicar um fator de redução bastante intenso no tamanho global da base de dados.

5.3 Conclusão

Neste capítulo, partiu-se de uma versão básica de um conversor texto-fala, oriunda da colaboração internacional entre a UFRJ e a Universidade do Porto. Esta versão foi, então, modificada, de forma a reduzir a complexidade computacional, e,

Tabela 5.4: Tamanhos totais e percentuais de cada versão da base de dados.

Versão	Tamanho (bits)	% em Relação a Anterior	% em Relação a Original
Original	186.642.880	100.00	100.00
Otimizada	62.559.264	33.52	33.52
Reamostrada	22.963.168	36.71	12.30
Codificada	3.846.656	16.75	2.06
Quantizada	1.520.064	39.51	0.81

principalmente, reduzir ao mínimo a interferência humana na execução do algoritmo. Com a versão final atendendo estes objetivos, a mesma pode ser utilizada no sistema proposto por este trabalho.

Também foi apresentada a estratégia abordada para a redução global da base de dados fornecida inicialmente. Foi alcançada uma redução de 99,19% no tamanho da mesma. Entretanto, esta nova base de dados gera um intenso *overhead* de processamento, uma vez que, antes da síntese ser realizada, torna-se necessário a decodificação de todos os difones componentes da frase que se deseja sintetizar.

Com a redução da base de dados para um tamanho inferior a 200 kbytes, a implementação em DSP torna-se perfeitamente possível. O *overhead* gerado devido a esta compressão não será um problema, tendo em vista o alto poder de processamento que este dispositivo possui, como foi visto no capítulo 4.

Capítulo 6

Implementação do Sistema e Resultados

Até este ponto do trabalho, foram apresentadas as técnicas e metodologias necessárias para tornar a implementação do sistema de síntese de voz em DSP possível. Neste capítulo, todos os conceitos estudados, e os resultados obtidos serão, finalmente, unidos para que o sistema proposto se torne realidade.

A figura 6.1 apresenta o diagrama em blocos do sintetizador de voz desenvolvido, mostrando como as três partes principais estão interligadas. Com isso, este capítulo divide-se da seguinte forma: a seção 6.1 apresentará o *interface* gráfica desenvolvida para operar com o sistema de síntese de voz. A seção 6.2 apresentará o *hardware* desenvolvido para a conversão do nível de tensão entre a RS-232 e o DSP. Em seguida, o seção 6.3 apresentará toda a implementação do algoritmo de síntese no DSP, bem como o algoritmo utilizado para a comunicação do DSP com a *interface* de usuário. A seguir, os resultados serão apresentados na seção 6.4.

Figura 6.1: Diagrama em blocos do sintetizador de voz.

6.1 *Interface* com o Usuário

Para a implementação deste protótipo, foi utilizado um computador de uso geral, tipo PC, como *interface* com o usuário. Entretanto, é válido afirmar que qualquer sistema de aquisição de texto pode ser usado, inclusive sistemas customizados, desde que os mesmos possuam uma *interface* serial compatível com a *interface* RS-232, e utilizem o protocolo serial UART, usado nos computadores convencionais.

Para a utilização de um computador de uso geral, tipo PC, como *interface* de usuário, foi desenvolvido um editor de texto implementado na linguagem de programação Java, que foi escolhida por possuir as seguintes vantagens:

- O mesmo código compilado roda em qualquer sistema operacional que possua uma Máquina Virtual Java [32].
- É bastante segura e robusta.
- É gratuita.
- É simples desenvolver *interfaces* gráficas para o usuário.

Além disso, foi utilizado um pacote de funções para acesso à porta serial do PC [33]. Assim, foi implementado um editor de texto bastante simples, com algumas funções genéricas como:

- Copiar, recortar e colar textos.
- Abrir e salvar textos.

Esta *interface* também apresenta duas funções específicas:

1. Um botão para enviar o texto para o sintetizador de voz.
2. Uma opção de *menu* para cancelar o envio de qualquer texto que esteja na espera para ser sintetizado, e, assim, cancelar a síntese.

Pode-se visualizar na figura 6.2 a *interface* desenvolvida. Para sintetizar um texto, o usuário só precisa digitá-lo (ou abrir um texto já escrito) e pressionar o botão "FALAR". Deste modo, a *interface* enviará para a porta serial do PC o texto escrito.

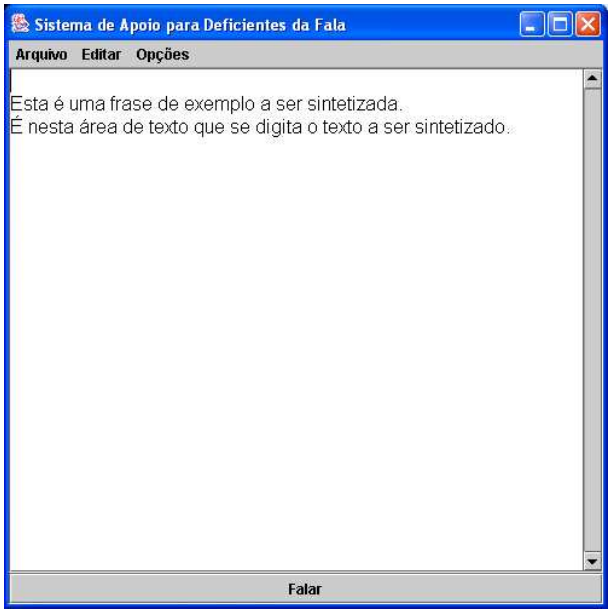

Figura 6.2: Ilustração da *interface* com o usuário.

Assim, após o usuário entrar com o texto que deseja sintetizar e apertar o botão "FALAR", o programa executará o seguinte algoritmo:

1. O texto digitado é separado por caracteres marcadores de frase, como ".,:;?!".

2. Para cada frase gerada, o sistema verifica se a frase possui um número de caracteres maior do que o máximo permitido (20 caracteres). Isto é importante para assegurar que o DSP terá memória suficiente para processar a frase.

3. Caso a frase seja maior do que o permitido, a mesma é dividida em frases menores. Neste caso, a separação é feita de forma a não se dividir uma palavra. Por fim, cada frase menor gerada é adicionada à fila de envio.

4. Caso a frase seja menor do que o máximo permitido, a frase é adicionada diretamente à fila de envio.

A fila de envio é usada para guardar de forma ordenada as frases geradas pelo algoritmo anterior. Assim, pode-se escrever um novo texto enquanto o sistema envia um texto anterior. Os dados da fila de envio são enviados ao controlador serial do PC da seguinte maneira:

1. Se a frase adicionada à fila é a primeira (fila inicialmente vazia), o programa chama diretamente o método que enviará a frase para a porta serial.

2. Se houver outras frases a serem enviadas, o envio ocorrerá de acordo com a requisição do DSP. Esta requisição é feita utilizando-se o pino de CTS (*Clear to Send*) da porta serial, e que está ligado a um *flag* de saída do DSP. Desta maneira, quando o DSP está pronto para receber uma nova frase, ele ativa o pino de CTS, de forma que o editor de texto reconheça esta transição e atenda a requisição mandando uma nova frase, caso esta exista. Como o evento é gerado tanto na transição positiva, quanto negativa do CTS, a rotina de envio antes verifica o estado final do pino de CTS, para confirmar se o estado final do pino é "ativado", situação na qual a transmissão deverá ser realizada.

3. Repete-se o item 2 até que todas as frases tenham sido enviadas.

Com este algoritmo de envio de frases para o DSP, tem-se como principal vantagem o fato de que, como a transmissão é gerida por eventos, o programa não precisa ficar aguardando o envio de todo o texto, atendendo a um pedido de transmissão somente quando solicitado, de forma que está livre para outras atividades enquanto não surge uma nova solicitação. Entretanto, o evento é gerado somente com a transição positiva do pino de CTS, fato que só ocorrerá após o término da síntese da primeira frase pelo DSP. Assim, para a primeira frase a ser enviada, é necessário forçar a chamada da função de envio, como visto no primeiro passo do algoritmo anterior.

Por fim, faz-se necessário dizer que o sistema, imediatamente antes de enviar a frase para o controlador serial do PC, insere um *byte* no início da frase contendo o tamanho da mesma[1]. Isto é fundamental para que o DSP possa alocar a memória necessária para receber a frase adequadamente. A relação das classes desenvolvidas, e seus respectivos métodos podem ser vistos no apêndice A

6.2 Conversão de Nível de Tensão RS-232/DSP

Como visto em [34], o nível de tensão usado pela RS-232 é de -15 volts para o bit "1" (*Mark Bit*) e +15 volts para o bit "0" (*Space Bit*). Entretanto, o

[1]Com isso, limita-se o tamanho máximo de uma frase enviada em 255 caracteres, embora um máximo de apenas 20 caracteres tenha sido utilizado nesta implementação, para evitar problemas de falta de memória por parte do DSP.

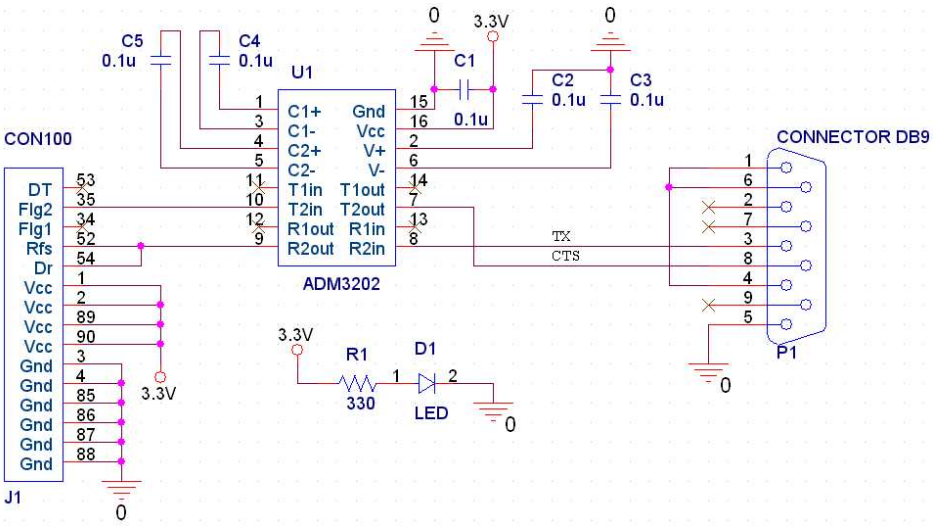

Figura 6.3: Esquemático do circuito utilizado para a conversão do nível de tensão entre a RS-232 e o DSP.

controlador externo do DSP utiliza +3 volts para o bit "1" e zero volts para o bit "0", de tal maneira que um dispositivo em *hardware* foi desenvolvido para realizar esta conversão de nível de tensão.

Para tal, foi utilizado o circuito integrado ADM3202 da *Analog Devices* [35], que realiza tanto a conversão do nível de tensão do DSP para a RS-232, quanto a conversão da RS-232 para o DSP. O esquemático do circuito projetado é apresentado na figura 6.3. O circuito é bastante simples, contendo alguns capacitores utilizados pelo integrado para implementar o multiplicador de tensão interno, um capacitor para desacoplamento da fonte, e um *LED* para sinalização de alimentação da placa. A alimentação do circuito provém da placa de desenvolvimento que contém o DSP.

Na figura 6.4 pode-se observar a placa já acabada. Como se pode notar, os componentes do circuito (excluindo-se os conectores) são todos do tipo SMD (*Surface Mount Device*), possuindo a vantagem de serem mais baratos e menores, reduzindo significativamente o tamanho do circuito final.

6.3 Sintetizador de Voz em DSP

O *software* que executa as funções do sintetizador foi implementado utilizando a linguagem C++. Ele é composto por cinco classes principais:

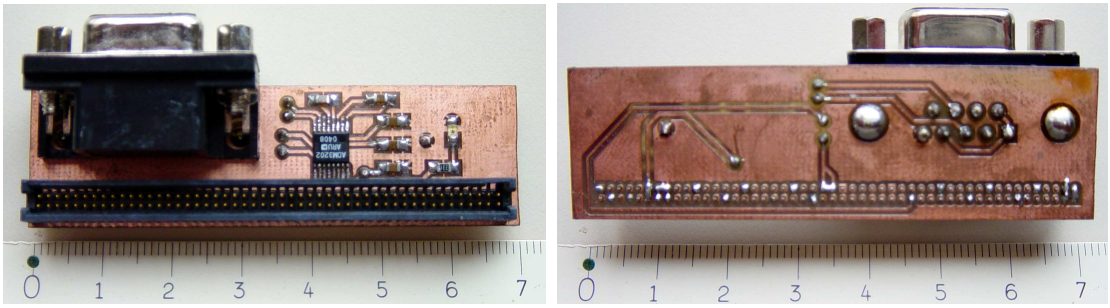

Figura 6.4: Vista superior (esquerda) e inferior (direita) do circuito conversor de nível de tensão (a unidade de medida apresentada na figura está graduada em centímetros).

- Classe de comunicação com a *interface* de usuário (CSerial).
- Classe de processamento de texto (CTextProc).
- Classe de gerenciamento da base de dados (CDataBase).
- Classe de implementação do algoritmo TD-PSOLA (CPSOLA).
- Classe de acesso ao CODEC da placa (CSound).

O diagrama de classes do sintetizador implementado é apresentado na figura 6.5. Como se nota, a execução do algoritmo é bastante linear, de forma que o resultado produzido por uma determinada classe é fornecido como dado de entrada à classe seguinte. Esta seção descreverá o funcionamento de cada uma destas cinco classes. A relação dos métodos de cada uma das classes desenvolvidas pode ser vista no apêndice A.

6.3.1 Classe de Comunicação com a *Interface* de Usuário

Esta classe encarrega-se de toda a comunicação com a *interface* de usuário, sendo responsável por receber as frases a serem sintetizadas. Como os dados são transmitidos serialmente, foi utilizada a *interface* serial do DSP, de tal maneira que a recepção dos dados ocorre independente do núcleo de processamento. Entretanto, como a porta serial do DSP é síncrona, e a do padrão UART usado como protocolo é assíncrono [36], torna-se impossível o envio e o recebimento simultâneo de dados.

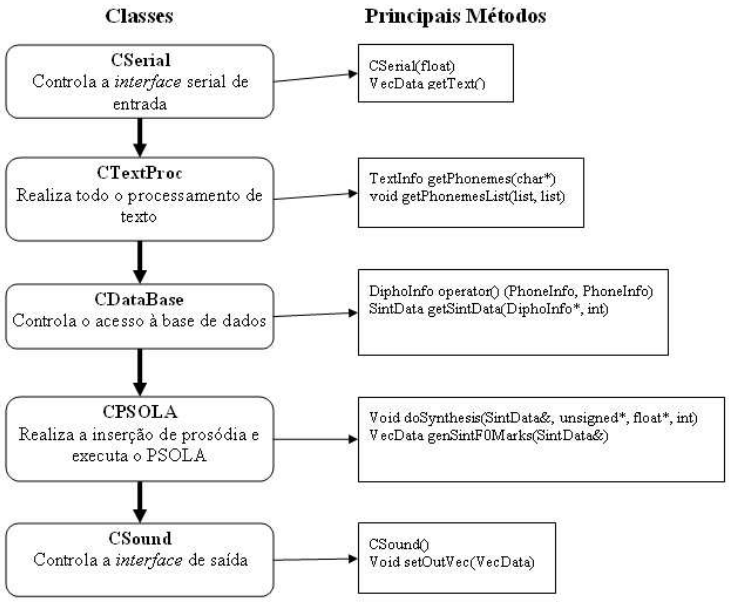

Figura 6.5: Diagrama de classes do sintetizador de voz implementado no DSP.

Mas isso não é um problema, visto que o DSP só recebe dados. Assim, a rotina de inicialização da porta serial é a seguinte:

1. A porta serial é habilitada apenas para a recepção de dados.

2. Em seguida, o controlador serial do DSP é configurado pra receber os dados com uma freqüência de transmissão três vezes superior à freqüência de transmissão usada pela UART (no caso 115.200 bits/seg). Isto se faz necessário, uma vez que não há nenhum controle de fase entre o controlador UART e a porta serial do DSP. Assim, realizamos esta superamostragem dos bits recebidos para resolver este problema, uma vez que pequenas variações na fase só afetarão os *chips*[2] no início ou no fim do bloco, de forma que se utiliza sempre o *chip* do meio como o valor do bit recebido.

3. Configura-se a porta serial para receber o pulso de sincronismo. Este pulso de sincronismo é o bit de início (*Start Bit*) da transmissão serial, de tal forma que basta ligar o pino de recebimento dos dados com o pino de sincronismo, como pode ser observado na figura 6.3, onde os pino de recebimento de dados e de

[2]O termo *chip* é utilizado em transmissão digital e representa cada componente do bit superamostrado. Desta forma, para este trabalho, como cada bit é amostrado com uma freqüência três vezes superior à freqüência de transmissão, cada bit será dividido em três *chips*.

sincronismo (pinos 54 e 52 do conector *CON100*, respectivamente) estão em curto-circuito. Assim, o recebimento de uma palavra ocorrerá somente quando o primeiro bit (o *Start Bit*) chegar ao pino de sincronismo do DSP, seguido dos bits de dados, bits estes que serão desprezados pelo pino de sincronismo, da mesma forma que o pino de dados desprezará o *Start Bit*.

4. Um *flag* (*FLAG2*) do DSP é configurado como *flag* de saída, de forma que seja utilizado pelo DSP como indicador de CTS (*Clear to Send*), ficando encarregado de notificar a *interface* de usuário que o DSP já está pronto para receber uma nova frase, conforme descrito na seção 6.1. Inicialmente, este *flag* se encontra desabilitado, impedindo qualquer envio de dados para o DSP.

Após a inicialização da porta serial do DSP, os dados poderão ser recebidos, quando o método *getText()* é chamado. Este método recebe os dados da seguinte maneira:

1. Inicialmente, o método ativa o pino de CTS, notificando a *interface* de usuário que está pronto para receber uma nova frase, e fica aguardando a recepção do primeiro *byte*, que contém o tamanho da frase a ser recebida.

2. De posse do tamanho da frase a ser recebida, o método desativa o pino de CTS, impedindo que mais de uma frase seja enviada, aloca a memória necessária e recebe os bits da frase propriamente dita.

3. Como cada bit é superamostrado, para cada palavra serial recebida, um método é chamado para extrair desta o *byte* transmitido pela UART.

4. Ao final da execução, o método retorna a frase completa recebida, e que deverá ser sintetizada.

6.3.2 Classe de Processamento de Texto

Esta classe é responsável por todo o pré-processamento do texto a ser sintetizado. Como visto no capítulo 5, o texto chega no DSP na forma de sílabas, de tal forma que é necessário, inicialmente, convertê-lo para o alfabeto SAMPA de fonemas. Além disso, para a inserção da prosódia, é preciso marcar as sílabas tônicas.

Ambos os procedimentos são realizados de acordo com o algoritmo desenvolvido em [31]. Este algoritmo, através de uma base de regras, gerada através de estudos lingüísticos, realiza a separação silábica do texto, a marcação das sílabas tônicas e a conseqüente conversão para fonemas. Contudo, a separação silábica deve separar *sílabas fonéticas*, ou seja, pretende-se separar a palavra em sílabas da forma como se soletra, não fazendo sentido, por exemplo, dividir o dígrafo "rr" em sílabas diferentes porque juntos realizam apenas um fonema [31]. Para as exceções, uma tabela contém palavras que não seguem a nenhuma das regras, reduzindo a probabilidade de erro de conversão do sistema.

Esta classe também possui um método que, de posse da seqüência de caracteres fonéticos, agrupa-os em difones e, para cada fonema de um difone, guarda a informação de tonicidade do mesmo, que será utilizada para a inserção da prosódia. Assim, na saída do método, tem-se a lista de difones a serem buscados na base de dados e a respectiva tonicidade de cada um.

6.3.3 Classe de Gerenciamento da Base de Dados

Esta classe se encarrega de realizar toda e qualquer comunicação com a base de dados do sintetizador. A base de dados possui as informações codificadas de todos os difones dispostas seqüencialmente, na forma de um grande vetor. As informações, bem como o tipo de dado e o número de bits necessários para a representação encontram-se na tabela 6.1.

Uma vez que a base de dados foi comprimida, conforme visto no capítulo 5, para apenas 191,5 kbytes, torna-se possível armazená-la na memória *Flash* da placa de desenvolvimento do DSP, que é de 512 kbytes, junto com o código de todo o sistema, permitindo a operação *stand alone* do sintetizador desenvolvido.

Para que o acesso à base de dados possa ser realizado eficientemente, esta classe gera internamente uma tabela de partição. Como o alfabeto fonético considerado está armazenado em um vetor, cada caracter fonético possui um índice específico (variando de 0 a 36, considerando 37 fonemas), de tal maneira que a tabela de partição é uma matriz de 37×37 elementos, onde cada difone é representado por dois índices. O primeiro (o número da linha) indica o índice do primeiro fonema, e o segundo (o número da coluna) representa o índice do segundo fonema.

Tabela 6.1: Organização das informações de cada difone.

Variável	**Descrição**	**Tipo**	**Tamanho**	**N^o de Bits**
ph1	Índice do 1^o fonema	*Unsigned*	1	6
ph2	Índice do 2^o fonema	*Unsigned*	1	6
diphSize	Tamanho real do difone	*Unsigned*	1	14
numBlocks	Número de janelas LPC	*Unsigned*	1	6
f0Size	Número de marcas de *F0*	*Unsigned*	1	6
f0CompSize	Número de palavras de marcas de *F0*	*Unsigned*	1	6
halfDiph	Índice de início do segundo fonema	*Unsigned*	1	14
voiceInfo[0]	Vozeamento do 1^o fonema	*Unsigned*	1	3
voiceInfo[1]	Vozeamento do 2^o fonema	*Unsigned*	1	3
f0Marks	Vetor com as marcas de *F0*	*Unsigned*	*f0CompSize*	32
coefLSF	Coeficientes LSF quantizados	*Unsigned*	*numBlocks*	32
adapGain	Ganhos adaptativos quantizados	*Unsigned*	*numBlocks*	32
fixedGain	Ganhos fixos quantizados	*Unsigned*	*numBlocks*	32
adapInd	Índices adaptativos	*Unsigned*	*numBlocks*$\times$ 4	16
fixedInd	Índices fixos	*Unsigned*	*numBlocks*$\times$4	16

Para a base de dados, tem-se que a primeira informação armazenada representa justamente os índices dos fonemas que compõem o difone, como apresentado na tabela 6.1. Assim, é possível gerar a tabela de partição rapidamente, varrendo a base de dados e armazenando na tabela o endereço do início dos dados referentes a cada difone na posição correspondente aos índices do mesmo. Como se sabe previamente a quantidade de dados de cada difone, uma vez lidos os índices de um difone, pode-se calcular dinamicamente o *offset* necessário para que o ponteiro seja posicionado no inicio do difone seguinte, tornando a geração da tabela de partição um processo bastante veloz.

Esta classe possui duas funções principais de acesso:

1. A função "*operator()*", que recebe como parâmetros os caracteres correspondentes ao primeiro e ao segundo fonema do difone, e esta retorna o endereço na base de dados correspondente aos dados deste difone, após a busca dos mesmos na tabela de partição.

2. A função "*getSintData*", que recebe a lista com os endereços de cada difone, e realiza a decodificação dos mesmos através do algoritmo de decodificação CELP apresentado no capítulo 3. Esta função retorna uma estrutura contendo toda a informação necessária para a síntese de voz (informação de vozeamento,

marcas de *F0*, informações de tonicidade, etc), pois será esta a informação utilizada pela classe de síntese.

6.3.4 Classe de Implementação do Algoritmo TD-PSOLA

Esta classe é responsável por aplicar o algoritmo de síntese de voz abordado neste trabalho. Ela recebe como parâmetros as seguintes informações:

- As amostras sonoras concatenadas de cada difone.
- As marcas concatenadas de *F0* de cada difone.
- A informação de vozeamento de cada fonema.
- A informação de tonicidade de cada fonema.

Inicialmente, o algoritmo final de síntese normaliza as amostras de áudio. Isto é importante para suavizar problemas de diferenças de energia entre um difone e outro. A normalização é realizada fazendo-se

$$y[i] = x[i] \left(\frac{M - m_i}{M} \right) \tag{6.1}$$

onde $x[i]$ é a i-ésima amostra de áudio, m_i é a média do penúltimo período vozeado do fonema (caso este exista, pois do contrário, esta média é zero), por ser um período relativamente estável do fonema. Por fim, M é dado por

$$M = \sum_{i=1}^{N} m_i \tag{6.2}$$

onde N é o total de fonemas da frase a ser sintetizada.

Em seguida, o algoritmo base de *Overlap and Add* é realizado, de acordo com o descrito no capítulo 5. O resultado da síntese é armazenado em um vetor, que será retornado à função chamadora. Uma vez este algoritmo finalizado, o resultado final da síntese já está pronto para ser enviado ao conversor *D/A* da placa de desenvolvimento.

6.3.5 Classe de Acesso ao CODEC da Placa

Esta classe, como o próprio nome sugere, encarrega-se de realizar a comunicação do DSP com o CODEC contido na placa de desenvolvimento. Este conversor comunica-se com o DSP pela porta serial do mesmo [37]. Como a porta serial do DSP opera independentemente e em paralelo com o núcleo de processamento, pode-se implementar uma espécie de paralelismo entre a síntese de uma frase e o processo de envio, ao CODEC, de uma frase previamente sintetizada, o que é bem mais eficiente do que, a cada frase sintetizada, aguardar o completo envio da mesma para o CODEC e, somente ao final deste envio, iniciar a síntese da frase seguinte. Assim, foi implementado um sistema de fila, gerido por esta classe de acesso ao CODEC, que funciona da seguinte maneira:

1. A frase f_0 (a primeira do texto) é sintetizada, e o resultado da síntese é armazenado no vetor v_0 alocado dinamicamente. O endereço deste vetor é passado para uma fila, que se encontra inicialmente vazia.

2. Imediatamente após a inclusão do vetor v_0 na fila, o sistema inicia a síntese da frase f_1. Ao mesmo tempo, as amostras contidas em v_0 são enviadas ao CODEC, uma vez que o envio de uma amostra é determinado por interrupção.

3. O sistema continua sintetizando as frases posteriores, sendo que, para cada nova frase, um vetor v_n precisa ser alocado dinamicamente. O sistema, antes de inicializar a síntese de uma nova frase, verifica, então, se há memória suficiente para uma nova frase, uma vez que ainda podem existir vetores na fila, aguardando para serem enviados ao CODEC, o que vai, gradativamente, reduzindo a memória total disponível para estes vetores de saída, uma vez que o DSP sintetiza frases muito mais velozmente do que o CODEC consegue recebê-las.

4. Caso a memória tenha sido exaurida, o sistema entra em estado de inoperância (*idle*), aguardando o término do envio das amostras do primeiro vetor da fila. Ao término do envio, a rotina de interrupção, que gerencia o envio das amostras automaticamente, libera a memória deste vetor, fazendo com que o DSP desperte e consiga finalmente alocar a memória desejada, dando prosseguimento, assim, ao processo de síntese.

5. Este processo paralelo prossegue até todas as frases do texto terem sido sintetizadas.

Como visto no algoritmo acima, a memória do DSP determina o quão "longe" o DSP pode ir, até ser obrigado a esperar que o CODEC termine o envio de alguns vetores, liberando memória para o prosseguimento da síntese. Uma vez que o DSP opera a 80 MHz e o CODEC recebe amostras a 8 kHz, a cada amostra recebida pelo CODEC, o DSP executa 10.000 instruções, o que, para um algoritmo de síntese bem implementado, significa uma enorme vantagem, garantindo que a demanda por processamento em tempo real do sistema seja atendida.

A região de memória utilizada para a alocação destes vetores de saída é de acesso exclusivo desta classe, e é utilizada para este fim somente, permitindo que esta falta de memória ocorra sempre, e somente devido a esta condição pois, do contrário, pode se correr o risco do sistema ficar sem memória em um ponto aleatório do algoritmo. Isto inviabilizaria esta implementação, já que em certos pontos do programa (dentro de funções da biblioteca padrão, por exemplo), não é possível remediar problemas de alocação.

6.3.6 Funcionamento Geral do Sistema

Uma vez que foi visto o papel desempenhado por cada classe, segue abaixo o algoritmo completo de execução do sistema.

1. O sistema recebe pela porta serial da *interface* de usuário uma seqüência de caracteres correspondentes à frase que se deseja sintetizar.

2. Os fonemas, bem como as informações de tonicidade são extraídos da frase.

3. Com os fonemas, gera-se a lista de difones desejados da base de dados.

4. As informações destes difones são decodificadas da base de dados e concatenadas.

5. É realizada a análise prosódica da frase, gerando como resultado a curva de *F0* a ser utilizada na síntese.

6. A memória necessária para o armazenamento do vetor de resultado é alocada, caso seja possível. Do contrário, o sistema aguarda até que vetores previamente alocados sejam liberados pela rotina de gerenciamento de interrupção da porta serial conectada ao CODEC.

7. O algoritmo TD-PSOLA é executado e o resultado é armazenado no vetor alocado no passo anterior.

8. O vetor com o resultado da síntese é armazenado na fila de envio ao CODEC.

9. O sistema volta novamente ao passo 1.

6.4 Análise de Desempenho do Sistema

Como forma de validação do sistema proposto, foram realizados três testes: análise de desempenho, teste de qualidade objetiva e teste de qualidade subjetiva.

6.4.1 Velocidade de Processamento

O primeiro teste realizado com a implementação final do projeto é verificar a velocidade do mesmo. Para tal, realizou-se uma análise, em ambiente de simulação, do sistema desenvolvido. Neste ambiente, utilizou-se a ferramenta de *profiling* do *VisualDSP++* [38], que realiza a contagem do número de instruções executadas em cada função do algoritmo de síntese implementado. Entretanto, conforme visto no capítulo 4, cada instrução é executada em um único ciclo de *clock*, de forma que o tempo pode ser calculado como

$$t = N_c \times T_{clk} \tag{6.3}$$

onde N_c é o número de instruções de uma dada função e T_{clk} é o período de *clock* do processador utilizado (no caso deste trabalho, tem-se $T_{clk} = 12,5000 \pm 0,0025\ ns$).

Realizadas as medidas, tem-se na tabela 6.2 o tempo gasto em cada etapa principal do processo de conversão texto-fala, e na figura 6.6, os valores percentuais destas medidas. Pode-se observar que a maior parte do tempo é gasto na decodificação dos difones necessários, e que o processo de síntese propriamente dito consome, comparativamente, um tempo bastante pequeno. Entretanto, mesmo com

o enorme *overhead* introduzido por esta etapa de decodificação, o tempo decorrido entre o momento em que o texto é adquirido, até o momento onde o vetor final com as amostras de saída é gerado, é de apenas 34.287012 ± 0.006857 ms para uma frase de tamanho pequeno (tipo "Bom dia a todos!"), o que mostra que o sistema implementado atende os requisitos de tempo real do projeto, uma vez que o ser humano não é sensível a um atraso desta magnitude. Além disso, conforme visto neste capítulo, enquanto o DSP envia as amostras de uma frase sintetizada, ao mesmo tempo, o mesmo já está realizando a síntese da frase seguinte, de forma que, quando a reprodução da primeira acabar, o processo de síntese da segunda frase também já terá sido efetuado, reduzindo o atraso real para praticamente zero.

Em outra análise realizada, desta vez com o *software* sendo executado no DSP[3], foi medido o número de ciclos em que o DSP se encontrava em processo de recebimento ou envio de dados, através da *interface* jTAG do mesmo (vide capítulo 4), de forma que esta medida pudesse ser realizada sem impactar a velocidade de processamento do DSP. Entretanto, esta medida é feita de maneira diferente daquela realizada no modo de simulação. Enquanto que no modo de simulação todas as instruções são contadas, dando uma visão bastante precisa do número de ciclos gastos em cada função, neste novo modo, chamado de medida estatística, a medida da performance do programa no DSP é feita através da amostragem do contador de programas (PC) com freqüência aleatória, enquanto o DSP executa o código. As áreas do programa onde o contador de programas é mais encontrado são justamente

[3]Devido à restrições da ferramenta de desenvolvimento, a análise de tempo anterior (*linear profiling*) só pode ser realizada em ambiente de simulação, de tal forma que o algoritmo é executado no PC de desenvolvimento, e não no DSP.

Tabela 6.2: Tempo gasto em etapa da síntese.

Etapa	**Tempo** (ms)
Conversão grafema / fonema	$0,147434 \pm 0,000029$
Decodificação dos difones	$21,168802 \pm 0,004234$
Geração da curva de *F0*	$0,267439 \pm 0,000053$
Realização do TD-PSOLA	$2,715531 \pm 0,000543$
Outros	$9,987807 \pm 0,001998$

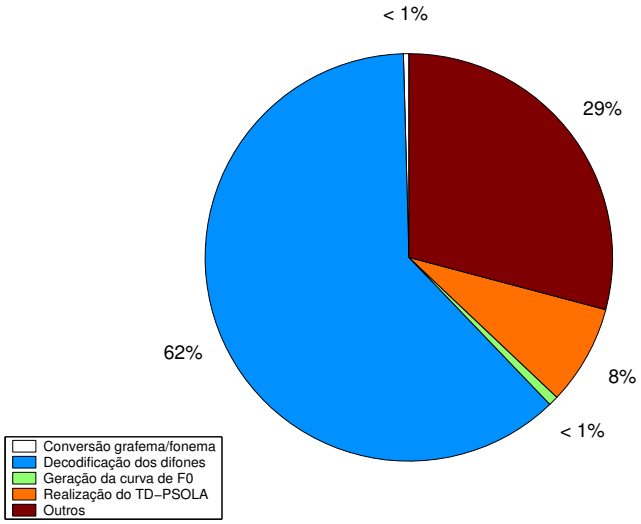

Figura 6.6: Visualização gráfica do percentual de tempo gasto no DSP para a conclusão da síntese de uma frase pequena.

aquelas onde se supõe que a maior parte do tempo de processamento está sendo gasto [38].

Por se tratar de um processo aleatório, para esta análise em tempo real, foram realizadas 10 medidas consecutivas, e os valores médios obtidos, e suas respectivas variâncias estão na tabela 6.3. Como visto na seção 6.3.6, a função de recebimento de texto é chamada apenas uma vez para cada frase a ser sintetizada. Entretanto, a função de envio das amostras sintetizadas é chamada um número muito maior de vezes (aproximadamente 3 ordens de grandeza acima), uma vez que é chamada para cada amostra que deva ser enviada. Desta maneira, embora os percentuais de números de ciclos observados em cada etapa estejam relativamente próximos, devido a esta enorme diferença de número de chamadas, conclui-se que a etapa onde o DSP recebe o texto a ser sintetizado é a que mais consome tempo. Isto pode ser justificado uma vez que, como os bits são recebidos a uma taxa de 115.200 bits/seg, e sincronamente (o DSP precisa aguardar o recebimento de toda a frase), tem-se que muito tempo é gasto nesta etapa. O mesmo não ocorre com a fase de envio das amostras resultantes para o conversor D/A, dado que este processo de envio é assíncrono, de tal forma que o DSP não consome tempo aguardando o fim do envio, conforme explicado na seção 6.3.5. Por fim, a grande diferença entre os desvios destes dois valores deve-se ao fato de que o inicio da contagem, e, em seguida, do

envio da primeira frase a ser sintetizada são controlados manualmente, deixando a medida da função de recebimento do texto com maior variância, quando comparada com a medida de envio das amostras ao CODEC.

6.4.2 Análise Objetiva da Qualidade de Síntese

A análise objetiva tem como finalidade fornecer ao leitor um parâmetro que torne possível a avaliação da qualidade da síntese, mesmo quando não se pode ouvir os resultados gerados. Desta maneira, optou-se por utilizar a distância de Itakura [39] como figura de mérito.

A distância de Itakura entre um determinado segmento $s(n)$ e o segmento que deseja-se comparar $\hat{s}(n)$ é dada por [16]

$$D_I(\hat{\alpha}, \alpha) = \frac{1}{2}\left[10\log\left(\frac{\hat{\alpha}\cdot\mathbf{R_S}\cdot\hat{\alpha}}{\alpha\cdot\mathbf{R_S}\cdot\alpha}\right) + 10\log\left(\frac{\alpha\cdot\mathbf{R_{\hat{S}}}\cdot\alpha}{\hat{\alpha}\cdot\mathbf{R_{\hat{S}}}\cdot\hat{\alpha}}\right)\right] \tag{6.4}$$

onde $\alpha = [1 - \alpha_1 \ldots - \alpha_p]^T$ e $\hat{\alpha} = [1 - \hat{\alpha}_1 \ldots - \hat{\alpha}_p]^T$ são os vetores obtidos, respectivamente, a partir dos coeficientes de predição linear $\{\alpha_1, \ldots, \alpha_p\}$ e $\{\hat{\alpha}_1, \ldots, \hat{\alpha}_p\}$ dos segmentos $s(n)$ e $\hat{s}(n)$, sendo $\mathbf{R_S}$ e $\mathbf{R_{\hat{S}}}$, suas respectivas matrizes de autocorrelação [16].

A distância de Itakura possui a vantagem de não ser sensível a variações de energia. Entretanto, esta medida é, na verdade, uma espécie de pseudo-distância, uma vez que $D_I(\hat{\alpha}, \alpha) \neq D_I(\alpha, \hat{\alpha})$, característica esta que viola o conceito clássico de distância. Entretanto, quanto maior for esta medida, mais afastados estarão os modelos comparados. Desta maneira, decidiu-se adotar como medida final da distância de Itakura a relação abaixo:

$$D_{Im}(\hat{\alpha}, \alpha) = \frac{D_I(\hat{\alpha}, \alpha) + D_I(\alpha, \hat{\alpha})}{2} \tag{6.5}$$

Tabela 6.3: Percentual do número de ciclos observados nas fases de recebimento e envio de dados entre o DSP e o resto do sistema.

Etapa	**Ciclos (%)**
Receber frase	$1,1 \pm 0,4$
Enviar amostras	$1,421 \pm 0,003$

Para a obtenção destas medidas, foi utilizado um pacote pronto para Matlab, obtido de [40]. Assim, calculou-se a matriz de distâncias D_I e D_{Im} entre uma frase gerada pelo sistema descrito neste capítulo, o sintetizador original, operando com a base de dados amostrada a 22.050 Hz, e o mesmo sintetizador, porém com a base de dados original reamostrada para 8 kHz.

As tabelas 6.4 e 6.5 contêm as médias obtidas, e seus respectivos desvios, das matrizes de distâncias de Itakura obtidas através da análise de um conjunto de 35 frases de comprimento aleatório. Como se percebe, a distância da base de dados a 22.050 Hz para a mesma versão, com base de dados a 8 kHz é bastante grande, como esperado. O mesmo ocorre para a distância da base de dados a 22.050 Hz e a base de dados codificada implementada neste trabalho. Entretanto, quando se analisa a distância da base de dados a 8 kHz e a base de dados codificada, percebe-se que esta diferença é relativamente pequena, mostrando que, objetivamente, a diferença entre estas duas bases é bastante reduzida, significando que a codificação não degradou de maneira significativa a qualidade da síntese. Percebe-se, por fim, que a grande diferença entre a base de dados a 22.050 Hz e as outras duas bases deve-se, em sua maioria, às perdas introduzidas pelo processo de reamostragem das amostras da base de dados para uma freqüência mais baixa, e não à codificação utilizada posteriormente.

6.4.3 Análise Subjetiva

A análise objetiva, embora possa fornecer uma avaliação da qualidade de síntese, não é capaz de apurar características perceptuais, uma vez que o ser humano pode não ser sensível a certas variações do resultado final em relação ao ideal, de

Tabela 6.4: Matriz de distância de Itakura (D_I) da versão final do sintetizador para a versão com base de dados amostrada a 22.050 Hz e a 8 kHz (sem codificação).

	22 kHz	**8 kHz**	**Codificada**
22 kHz	$0,000000 \pm 0,000000$	$3,523634 \pm 0,820706$	$3,623994 \pm 0,873791$
8 kHz	$3,981042 \pm 0,947577$	$0,000000 \pm 0,000000$	$0,712728 \pm 0,695297$
Codificada	$3,938025 \pm 0,861246$	$0,668129 \pm 0,568218$	$0,000000 \pm 0,000000$

Tabela 6.5: Distância média de Itakura (D_{Im}) da versão final do sintetizador para a versão com base de dados amostrada a 22.050 Hz e 8 kHz (sem codificação).

	22 kHz	**8 kHz**	**Codificada**
22 kHz	$0,000000 \pm 0,000000$	$3,752338 \pm 0,695611$	$3,781010 \pm 0,674143$
8 kHz	$3,752338 \pm 0,695611$	$0,000000 \pm 0,000000$	$0,690429 \pm 0,628934$
Codificada	$3,781010 \pm 0,674143$	$0,690429 \pm 0,628934$	$0,000000 \pm 0,000000$

forma que, mesmo que a análise objetiva deixe a desejar, a análise subjetiva pode, ainda sim, aprovar o modelo. Desta forma, é importante que este tipo de análise seja realizada para a correta validação do sistema. Para tal, foram utilizados dois testes subjetivos: teste de inteligibilidade e de qualidade.

6.4.3.1 Teste de Inteligibilidade

Este teste, tem por finalidade, avaliar o grau de compreensão do sistema desenvolvido. Para tal, um conjunto de 20 palavras isoladas foi selecionado. As palavras, retiradas de [41], foram as seguintes:

01 -	Abacate	11 -	Canhão
02 -	Morango	12 -	Violeta
03 -	Carro	13 -	Jarro
04 -	Zumbi	14 -	Lapiseira
05 -	Salada	15 -	Notícia
06 -	Didática	16 -	Xícara
07 -	Gruta	17 -	Lacrado
08 -	Filho	18 -	Plástico
09 -	Palhaço	19 -	Caderno
10 -	Garoto	20 -	Profissão

Em seguida, estas palavras foram sintetizadas pelo sintetizador desenvolvido neste trabalho e pronunciadas, sem repetição, para uma audiência de 14 pessoas, onde cada pessoa, após ouvir a palavra, escreveu a palavra compreendida. Antes da realização deste teste, 4 palavras (não inclusas no grupo de teste) foram pronunciadas, para familiarização do sistema por parte dos participantes.

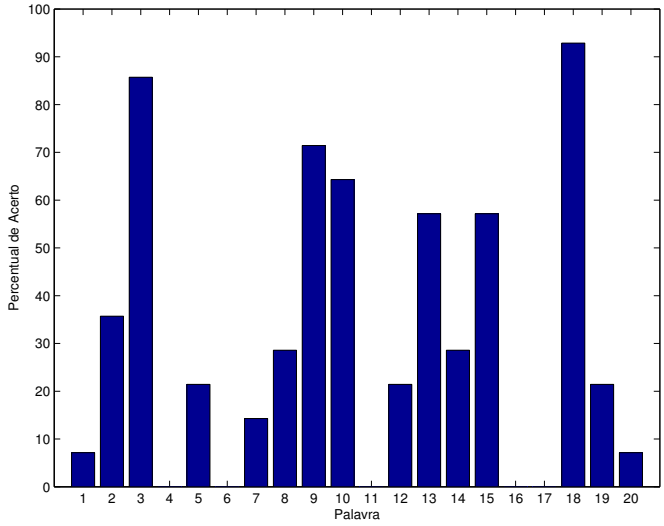

Figura 6.7: Percentual de acerto obtido para cada uma das 20 palavras pronunciadas para uma audiência de 14 pessoas.

Observa-se na figura 6.7 o percentual de acerto para cada palavra (indicada no gráfico pelo seu respectivo número) pronunciada, e os valores exatos obtidos, podem ser observados na tabela 6.6. Na média, cada avaliador foi capaz de identificar corretamente apenas $31 \pm 14\%$ das palavras apresentadas, mostrando que a inteligibilidade ficou a desejar. Entretanto, observando-se a figura 6.7 observa-se que existe grande flutuação da taxa de acerto de cada palavra, mostrando que um conjunto específico delas foi responsável por este fraco resultado. Estes resultados devem-se a uma combinação de três fatores:

1. Primeiramente, a pronúncia fica prejudicada, uma vez que a base de dados foi reamostrada para 8 kHz, enquanto que a versão original possuía taxa de amostragem igual a 22.050 Hz. Com a resolução espectral reduzida em aproximadamente 67%, é de se esperar que componentes espectrais de mais altas freqüências sejam descartadas. Assim, a compreensão da palavra como um todo fica comprometida, uma vez que parte da informação espectral da mesma foi perdida.

2. Em segundo lugar, tem-se um problema relacionado à codificação da base de dados. Como já dito no capítulo 5, a codificação se dá para cada difone individualmente. Como visto no capítulo 3, a qualidade da codificação esta bas-

tante relacionada ao uso do dicionário adaptativo. Este dicionário encontra-se inicialmente zerado, de forma que seu conteúdo vai surgindo ao longo da codificação, tornando a qualidade da mesma melhor ao longo do processo. Desta forma, inicialmente, a qualidade da codificação pode ficar bastante a desejar, uma vez que o dicionário adaptativo ainda não possui informação suficiente sobre o sinal a ser codificado, necessitando de aproximadamente 100 ms de voz para atingir este ponto ótimo de operação. No caso do sintetizador implementado, tem-se a presença de difones bastante curtos (alguns com apenas 50 ms de duração), o que resulta na má codificação dos mesmos, resultando numa má síntese de uma dada frase que utilize estes difones, dificultando a compreensão da palavra como um todo. Este efeito foi mais visível, no teste realizado, nas palavras "Gruta" (7), "Canhão" (11) e "Xícara" (16).

3. Por último, tem-se o problema do sotaque. O sintetizador desenvolvido apresenta sotaque do Português europeu, enquanto que 86% dos participantes do teste de inteligibilidade eram brasileiros, e cada palavra foi reproduzida uma única vez. Desta maneira, houve problemas de compreensão, dado ao sotaque, que para certas palavras, era bastante intenso para alguém que não estivesse habituado ao sotaque europeu. Para que a importância do sotaque pudesse ser avaliada, está apresentado na figura 6.8 o percentual de acerto deste teste gerado a partir das avaliações obtidas de dois[4] portugueses que integravam a equipe de avaliação do sistema. Como se percebe, obteve-se, para este grupo, resultados consideravelmente melhores, com cada avaliador identificando corretamente $50 \pm 7\%$ das palavras apresentadas. Duas palavras em especial, não puderam ser determinadas por este segundo grupo composto apenas por portugueses por motivos regionais. A primeira foi "Zumbi" (4), uma vez que em Portugal utiliza-se a palavra originada do inglês ("*Zombie*"). A segunda foi a palavra "Xícara" (16), uma vez que a mesma não existe no Português europeu, onde se utiliza o vocábulo "Chávena" para referir-se a este objeto.

[4] Apenas duas pessoas foram utilizadas devido a impossibilidade de encontrar outros participantes familiarizados com o Português europeu. Entretanto, mesmo não sendo um conjunto estatisticamente representativo, o mesmo pode dar, ao menos, uma aproximação do resultado a ser esperado para um conjunto maior.

Esta análise comparativa mostra o forte impacto que o sotaque pode causar na qualidade subjetiva de conversores texto-fala, de forma que, para uma boa inteligibilidade, a adaptação do sistema para as características idiomáticas do local a ser utilizado é fundamental.

6.4.3.2 Teste de Qualidade

Inicialmente, pensou-se em realizar um teste MOS (*Mean Opinion Score*) [3] para análise da qualidade. Entretanto, este teste não seria fidedigno, tendo em vista que o sistema seria testado com um grupo de avaliadores, composto, em sua maioria, por brasileiros, enquanto que o sistema desenvolvido possui sotaque europeu. Desta forma, pensou-se em realizar um outro teste, que tem como finalidade comparar a qualidade do sistema desenvolvido com a versão onde a base de dados não sofreu nenhum tipo de codificação, e estava amostrada a 8 kHz. Para tal, foram selecionadas 10 frases, retiradas de [42], e cada frase foi sintetizada por ambos os sintetizadores. As frases utilizadas neste teste foram:

1. Ela saía discretamente
2. Desculpe se magoei o velho
3. Queremos discutir o orçamento
4. Ela tem muita fome
5. Hoje dormirei bem
6. Depois do almoço te encontro
7. Procurei Maria na copa
8. A pesca é proibida neste lago
9. O inspetor fez a vistoria completa
10. Temos muito orgulho da nossa gente

Tabela 6.6: Taxa de acerto para cada palavra do teste de inteligibilidade.

Índice	**Acerto (%)**	**Índice**	**Acerto (%)**
01	7 ± 7	11	0 ± 7
02	35 ± 7	12	21 ± 7
03	85 ± 7	13	57 ± 7
04	0 ± 7	14	28 ± 7
05	21 ± 7	15	57 ± 7
06	0 ± 7	16	0 ± 7
07	14 ± 7	17	0 ± 7
08	28 ± 7	18	92 ± 7
09	71 ± 7	19	21 ± 7
10	64 ± 7	20	7 ± 7

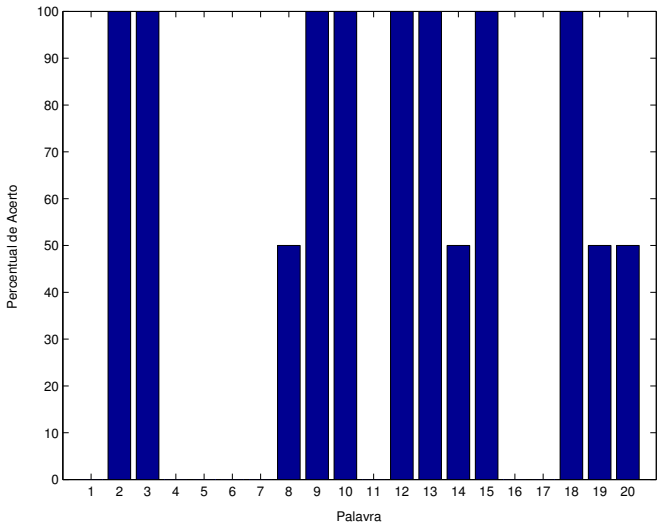

Figura 6.8: Percentual de acerto obtido para cada uma das 20 palavras pronunciadas para o grupo contendo apenas portugueses.

Em seguida, o grupo de teste ouviu, de maneira análoga à realizada no teste anterior, duas frases, também não inclusas na lista de teste, para que se familiarizassem com ambos os sintetizadores, porém, nenhuma informação a respeito do sintetizador utilizado foi dada ao grupo. Em seguida, cada frase foi apresentada ao grupo, sendo proferida por cada um dos sintetizadores (sem repetição, no que diz respeito ao sintetizador utilizado), porém, a ordem dos sintetizadores era aleatória, para evitar tendências de resultados. Por fim, cada ouvinte deveria escolher entre três opções: se a primeira realização lhe pareceu melhor, a segunda, ou se o mesmo não conseguiu observar diferenças entre ambas.

O resultado deste teste pode ser observado na figura 6.9. Ela apresenta, em valores percentuais, quantas frases foram tidas como melhor sintetizadas pelo sintetizador com base de dados amostrada a 8 kHz, quantas vezes foram consideradas melhor sintetizadas pelo sintetizador com base de dados codificada implementado, e, por fim, em quantas ocasiões não foi possível distinguir entre ambas as frases. Como se percebe, este resultado encontra-se bastante satisfatório, uma vez que, em 34% dos casos, não foram notadas diferenças entre os sintetizadores comparados, o que representa um número apenas 26% menor do que o percentual de vezes que o sintetizador de comparação (com base a 8kHz) foi considerado melhor. Considerando-se que, em 20% dos casos, o sintetizador implementado foi tido como o melhor, pode-se inferir que, de forma geral, as qualidades das duas versões comparadas estão bastante próximas, mostrando, mais uma vez, que a codificação não impactou de maneira significativa na qualidade da síntese, quando comparada com um sintetizador com base de dados sem codificação, amostrada a 8 kHz.

6.5 Conclusão

Foi apresentada neste capítulo a etapa de implementação prática do sistema proposto. A implementação foi bem sucedida, com todos os módulos operando segundo protocolos pré-estabelecidos. Entretanto, o foco da implementação foi na diminuição da base de dados, de forma a atender os requisitos de custo e portabilidade do projeto. Os testes de velocidade provaram que o sistema atende os requisitos de tempo real da aplicação, e os testes de qualidade provaram que a perda introdu-

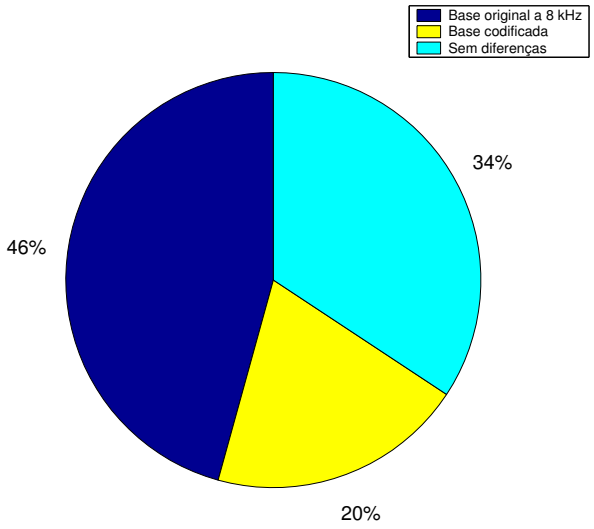

Figura 6.9: Valores percentuais resultantes da comparação do sintetizador implementado com base de dados codificada, com o mesmo sintetizador, porém com base e dados a 8 kHz, sem codificação.

zida pela codificação da base de dados é pequena, quando comparada com a perda introduzida pela reamostragem da base de dados para uma freqüência mais baixa.

A análise subjetiva da inteligibilidade ficou a desejar, quando se utilizou um grupo de avaliadores contendo, em sua maioria, brasileiros. Entretanto, quando se analisou o resultado do grupo contendo somente portugueses, observou-se que os resultados foram significativamente melhores, de tal maneira que este trabalho pôde apresentar o impacto que o sotaque causa na avaliação de sistemas de conversão texto-fala.

Capítulo 7

Conclusão

Foi desenvolvido um sistema compacto de conversão texto-fala para o Português. Inicialmente, abordou-se a teoria de síntese de voz, onde foi apresentado o algoritmo escolhido para ser implementado em DSP. O estudo da prosódia realizado foi capaz de mostrar a importância deste conceito na qualidade dos conversores texto-fala. Apresentou-se também, alguns sistemas de conversão texto-fala comerciais, onde se mostrou que o mercado, aparentemente, não investe em sistemas portáteis, uma vez que os sistemas comerciais desenvolvidos eram todos para computadores de uso geral, limitando, assim, a utilização destes sistemas.

Também foi apresentada a técnica de codificação CELP. Neste ponto, provou-se que esta técnica poderia ser bastante útil no desenvolvimento do sistema proposto neste trabalho, uma vez que alia qualidade com eficiência de codificação.

Apresentou-se o conceito de processadores digitais de sinais, bem como o processador utilizado na implementação deste projeto, onde foram destacadas as principais características deste dispositivo, mostrando que o mesmo pode atender a todos os requisitos do projeto.

Foi realizada uma análise detalhada do algoritmo TD-PSOLA a ser implementado originalmente, devido à colaboração UFRJ/FEUP. Através desta análise, diversas adaptações foram implementadas, resultando num sistema muito mais independente, uma vez que automatizou-se etapas onde era necessária intervenção humana. Também se realizou a análise da base de dados, onde se obteve uma redução de mais de 99% no tamanho da mesma, conseguindo-se, assim, reduzi-la, dos originais 23 Mbytes para 190 kbytes, tornando-a apropriada para a implementação em

DSP.

A realização prática do projeto foi bem sucedida. Todos os módulos desenvolvidos foram integrados, e o sistema operou sem problemas. A implementação foi feita em C++, explorando os conceitos de orientação a objetos, de forma a facilitar a manutenção e a implementação de melhorias futuras, uma vez que se realizou uma análise cuidadosa dos requisitos de *software* do sistema para que o máximo de abstração de componentes de *software* fosse atingido.

No desenvolvimento deste sistema, foi dado prioridade à compactação da base de dados, visto que o DSP possuía apenas 512 kbytes de memória flash. Por outro lado, esta quantidade de memória não é restritiva, de tal forma que módulos de memória flash de capacidades maiores podem ser utilizados, ao custo do encarecimento do projeto. Desta forma, existe um compromisso entre compactação e qualidade de síntese. Uma versão com maior qualidade de síntese pode ser desenvolvida, ao custo de maior utilização de memória. Do contrário, se for priorizado a redução de custos, a compactação torna-se uma solução atraente, mas que impactará, invariavelmente, na qualidade da síntese.

Nos testes de inteligibilidade, os resultados apresentados confirmaram o forte impacto que o sotaque apresenta na hora de avaliar o sistema desenvolvido. Além disso, existem palavras em uma versão do idioma que são inexistentes em outra, dificultando a migração para outra versão do idioma. Desta maneira, tem-se que escolher a finalidade do projeto. A implementação de uma única versão de sintetizador para uma determinada versão do idioma pode ser mais simples e barata, mas invariavelmente implicará em problemas de inteligibilidade em regiões com sotaques diferentes. Por outro lado, o desenvolvimento de diversas versões para um mesmo idioma pode aumentar consideravelmente a complexidade do projeto, uma vez que a análise lingüística será realizada para várias versões de um mesmo idioma.

O sotaque Português, aliado a reamostragem da base para 8 kHz, foram os principais fatores de perdas de inteligibilidade por parte do sistema desenvolvido. Por fim, é válido mencionar que a rigorosidade do teste (pouca familiarização com o sistema, ausência de repetição das palavras e utilização de palavras curtas) também impactou negativamente nos resultados do teste de inteligibilidade. Entretanto, este rigor serve para obter uma medida de inteligibilidade mínima do sistema.

Verificou-se que a implementação realizada atende aos requisitos de velocidade, e está totalmente adaptada para operar com qualquer dispositivo customizado que possa gerar texto escrito, e transmiti-lo através da *interface* serial RS-232, com protocolo UART. Além disso, foi desenvolvida uma *interface* gráfica em Java de forma a conectar o protótipo desenvolvido a computadores de uso geral, PDAs e outros sistemas digitais que suportem esta linguagem de programação.

Com a implementação de um sistema de baixo custo, portátil e de fácil utilização, foi dado um passo significativo na difusão de sistemas de conversão texto-fala, principalmente para indivíduos com necessidades especiais, tendo em vista que este trabalho pode aumentar de maneira significativa a independência destes grupo de pessoas.

Por fim, é valido afirmar que este trabalho é uma poderosa contribuição para a defesa do idioma, uma vez que a maioria dos sistemas desenvolvidos são para o inglês, de tal maneira que se corre o risco de, num futuro próximo, não haverem mais sistemas para o Português. Com este trabalho, espera-se estimular o contínuo desenvolvimento de projetos específicos para o Português.

7.1 Possíveis Trabalhos Futuros

Este trabalho, embora seja um protótipo completo, pode sofrer melhorias. Entre elas:

1. **Utilização de vetores de quantização específicos:** neste trabalho, foi utilizado vetores de quantização LPC e de ganhos genéricos. Entretanto, como se sabe previamente todos os coeficientes LPC e valores de ganho da base de dados (uma vez que a codificação é feita *offline*), novos vetores de quantização podem ser gerados utilizando-se estes valores previamente conhecidos, produzindo-se, assim, vetores de quantização totalmente condicionados ao sistema desenvolvido.

2. **Inicialização prévia do dicionário adaptativo:** para a codificação da base de difones do sintetizador, tem-se a presença de difones bastante curtos, o que resulta na má codificação dos mesmos. No futuro, este problema pode ser resolvido utilizando-se um dicionário adaptativo pré-inicializado, de forma

que o mesmo já possua alguma informação sobre a excitação que se busca, e em menos janelas, atinja seu estado ótimo de operação.

3. **Melhorias na prosódia:** métodos mais inteligentes podem ser utilizados para a extração dos parâmetros de Fujisaki, como, por exemplo, métodos baseados em redes neurais, que conseguem identificar eficientemente os padrões lingüísticos que fornecerão informação mais confiável para a extração dos parâmetros de Fujisaki. Além disso, um módulo de acerto de durações que colocaria a duração específica de cada fonema, de acordo com a entonação que se deseje obter, pode ser adicionado, melhorando ainda mais a naturalidade da fala.

4. **Alteração da base de dados e dos algoritmos de processamento de texto para o Português brasileiro:** para a completa migração do sistema desenvolvido neste trabalho para o Português brasileiro, pode-se utilizar uma base de dados desenvolvida para esta versão do idioma. Além disso, pode-se converter os algoritmos de processamento de texto, para que estes atendam as nuances do Português brasileiro.

5. **Desenvolvimento de um produto final:** este trabalho foi realizado em uma placa de avaliação. entretanto, para a futura comercialização deste projeto, será necessário o desenvolvimento de uma placa especialmente projetada para a aplicação, contendo somente os componentes necessários para o sistema de síntese, reduzindo as dimensões e o número de componentes do produto acabado, com a conseqüente queda no preço do mesmo.

Referências Bibliográficas

[1] LIBERMAN, A. L., INGEMANN, F., LISKER, L., *et al.*, "Minimal Rules for Synthesizing Speech", *Journal of the Acoustical Society of America*, pp. 1490–1499, 1959.

[2] DUTOIT, T., *An Introduction to Text-to-Speech Synthesis*, v. 3. Kluwer Academic, 1999.

[3] BARROS, M. J., *Estudo Comparativo e Técnicas de Geracão de Sinal para a Síntese da Fala*. Tese de mestrado, Universidade do Porto, 2002.

[4] MOULINES, E., CHARPENTIER, F., "Pitch-Synchronous Waveform Processing Techniques for Text-to-Speech Synthesis Using Diphones", *Speech Communications*, vol. 9, no. 6, pp. 453–467, dezembro 1990.

[5] BECHARA, E., *Moderna Gramática Portuguesa*. Lucerna, 2001.

[6] TEIXEIRA, J. P., FREITAS, D., "Segmental Durations Predicted with a Neural Network", *Proceedings of Eurospeech*, pp. 169–172, setembro 2003.

[7] TEIXEIRA, J. P., FREITAS, D., FUJISAKI, H., "Prediction of Accent Commands for the Fujisaki Intonation Model", *Proceedings of Speech Prosody*, pp. 451–455, março 2004.

[8] MIXDORFF, H., *Intonation Patterns of German - Quantitative Analysis and Synthesis of* F0 *Countours*. Tese de doutorado, Technische Universität Dresden, 1998.

[9] SILVA, S. S., *Um Estudo de Modelos de Prosódia para o Português Brasileiro*. Tese de mestrado, UFRJ, maio 2004.

[10] MICROPOWER SOFTWARE, *www.micropower.com.br*.

[11] DIGALO, *www.digalo.com*.

[12] SCANSOFT INC., *www.scansoft.com*.

[13] TEXTO-FALA, *www.cpqd.com.br/produtos/textofala/*.

[14] HAYKIN, S., *Communication Systems*. 3 ed. John Wiley & Sons, 1994.

[15] DELLER, J. R., PROAKIS, J. G., HANSEN, J. H. L., *Discrete-Time Processing of Speech Signals*. IEEE Press, 1993.

[16] MAIA, R. S., *Codificação CELP e Análise Espectral de Voz*. Tese de mestrado, UFRJ, 2000.

[17] DINIZ, F. C. C. B., *Implementação de um Codificador de Voz CELP em Tempo Real*, Projeto final, UFRJ, maio 2003.

[18] DINIZ, P. S. R., da SILVA, E. A. B., NETTO, S. L., *Digital Signal Processing: System Analysis and Design*. Cambridge University Press, 2002.

[19] KROON, P., SWAMINATHAN, K., "A High-Quality Multirate Real-Time CELP Coder", *IEEE Journal on Selected Areas in Communications*, vol. 10, no. 5, pp. 850–857, junho 1992.

[20] ATAL, S. B., "Predictive Coding of Speech At Low Bit Rates", *IEEE Transactions on Communications*, vol. 30, no. 4, pp. 600–614, abril 1982.

[21] ACKENHUSEN, J. G., *Real-Time Signal Processing*. Prentice Hall, 1999.

[22] ANALOG DEVICES, *ADSP-21160: SHARC DSP Hardware Reference*. 2 ed., maio 2002.

[23] INTEL, *Intel Pentium M Processor on 90nm Process with 2-MB L2 Cache*, maio 2004.

[24] THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, *IEEE Standard for Binary Floating-Point Arithmetic*, março 1985.

[25] ANALOG DEVICES, *Visual DSP++ 3.0 Manual: C/C++ Compiler and Library Manual For SHARC DSPs*. 4 ed., janeiro 2003.

[26] "Analog Devices", www.analog.com.

[27] ANALOG DEVICES, *ADSP-21160 EZ-KIT Lite: Evaluation System Manual*. 3 ed., janeiro 2003.

[28] ANALOG DEVICES, *Visual DSP++ 3.0 Manual: Getting Started Guided for SHARC DSPs*. 4 ed., janeiro 2003.

[29] BARROS, M. J., *Projeto Antígona: Site de Comércio Electrónico com* Interface *de Voz para o Português Europeu*, Relatório de projeto, Faculdade de Engenharia da Universidade do Porto, setembro 2001.

[30] TEIXEIRA, J. P., FREITAS, D., BRAGA, D., *et al.*, "Phonetic Events from the Labeling the European Portuguese Database for Speech Synthesis, FEUP/IPB-DB", *Eurospeech*, pp. 1707–1710, setembro 2001.

[31] GOUVEIA, P. D. F., TEIXEIRA, J. P., FREITAS, D., "Divisão Silábica Automática do Texto Escrito e Falado", *Actas do V PROPOR, Processamento Computacional da Língua Portuguesa Escrita e Falada*, pp. 65–74, novembro 2000.

[32] NAUGHTON, P., SCHILDT, H., *Java 2: the Complete Reference*. 3 ed. McGraw-Hill, 1999.

[33] SUN, "Java Communication API Overview", http://java.sun.com/products/javacomm/javadocs/index.html.

[34] PEACOCK, G., "Interfacing the Serial / RS-232 Port", www.beyondlogic.org/serial/serial.htm, agosto 2001.

[35] ANALOG DEVICES, *Low Power, 3.3 V, RS-232 Line Driver/Receiver*, 2001.

[36] LEDGER, D., *Implementing a Glueless UART Using the SHARC DSP SPORTs*. Analog Devices, maio 2003.

[37] ANALOG DEVICES, *AD1881A AC´97 SoundMAX® Codec Datasheet*, 2000.

[38] ANALOG DEVICES, *Visual DSP++ 3.0 User Guide for SHARC DSPs*. 4 ed., janeiro 2003.

[39] CHILDERS, D. G., *Speech Processing and Synthesis Toolboxes*. John Wiley & Sons, 1999.

[40] BROOKES, M., "VOICEBOX: Speech Processing Toolbox for MATLAB", www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html.

[41] SILVA, S. S., *Sistema de Conversão Texto-Fala Usando Unidades Silábicas*, Projeto final, UFRJ, dezembro 2001.

[42] ALCAIM, A., SOLEWICZ, J. A., MORAES, J. A., "Freqüência de Ocorrência dos Fones e Listas de Frases Foneticamente Balanceadas no Português Falado no Rio de Janeiro", *Revista da Sociedade Brasileira de Telecomunicações*, vol. 7, no. 1, pp. 23–41, dezembro 1992.

Apêndice A

Relação de Classes

Este apêndice lista todos os métodos contidos em cada uma das classes desenvolvidas para a *interface* de usuário e o sintetizador de voz implementado em DSP. Espera-se com esta descrição, facilitar a tarefa daqueles que desejem utilizar, ou aperfeiçoar o sistema desenvolvido.

A.1 *Interface* de Usuário

A.1.1 Métodos da Classe *Actions*

Retorno	**Método**	**Tipo**
N/A	Actions(window)	Público
void	clearList()	Público
void	closeApplication()	Público
void	closeFile()	Público
void	copyText()	Público
void	cutText()	Público
void	newFile()	Público
void	openFile()	Público
void	pasteText()	Público
boolean	saveFile()	Público
boolean	saveFileAs()	Público
void	selectAll()	Público
void	sendText()	Público
void	setTextModified()	Público
void	restoreFileStates()	Privado
void	saveFileStates()	Privado

A.1.2 Métodos da Classe *CSerial*

Retorno	**Método**	**Tipo**
N/A	CSerial(window)	Público
void	emptyList()	Público
void	serialEvent(SerialPortEvent)	Público
void	writeData(String, boolean)	Público

A.1.3 Métodos da Classe *sadf*

Retorno	**Método**	**Tipo**
void	main(String[])	Público

A.1.4 Métodos da Classe *TextFilter*

Retorno	Método	Tipo
boolean	accept(File)	Público
String	getDescription()	Público
String	getExtension(File)	Privado

A.1.5 Métodos da Classe *window*

Retorno	Método	Tipo
N/A	window()	Público
void	actionPerformed(ActionEvent)	Público
void	generateMenu()	Privado
void	generateTextArea()	Privado
void	keyPressed(KeyEvent)	Público
void	keyReleased(KeyEvent)	Público
void	keyTyped(KeyEvent)	Público

A.2 Sintetizador de Voz

A.2.1 Métodos da Classe *CSerial*

Retorno	Método	Tipo
N/A	CSerial(float)	Público
VecData<char>	getText()	Público
char	receiveWord()	Privado
void	rxISR(int)	Privado

A.2.2 Métodos da Classe *CTextProc*

Retorno	**Método**	**Tipo**
TextInfo	getPhonemes(char*)	Público
void	getPhonemesList(char*, list<PhoneInfo>, list<PhoneInfo>)	Público
TextInfo	textDecode(char*))	Privado
unsigned char*	markWords(unsigned char*)	Privado
char*	preTranscription(char*)	Privado
unsigned char*	markSyllables(unsigned char*)	Privado
unsigned char*	markStressesSyllable(unsigned char*)	Privado
char*	phoneticTranscription(char*)	Privado
void	init1252to850(unsigned char*)	Privado
unsigned char	conv2LowCaps(unsigned char)	Privado
void	wordProc(char*)	Privado
unsigned char*	splitWord(unsigned char*,unsigned char*)	Privado
void	markSyllable(unsigned char*)	Privado
char*	translatedWord(char*, char*)	Privado
char*	exceptionTable(char*)	Privado
void	init850to1252(unsigned char*)	Privado
bool	phoneticVowel(unsigned char)	Privado
bool	sequentialConsonants(unsigned char, unsigned char)	Privado
bool	accentedLetter(unsigned char)	Privado
bool	accentedTermination(unsigned char*)	Privado
bool	accentedVowel(unsigned char)	Privado
bool	isVowel(unsigned char)	Privado
bool	isConsonant(unsigned char)	Privado

A.2.3 Métodos da Classe *CDataBase*

Retorno	**Método**	**Tipo**
N/A	CDataBase()	Público
DiphoInfo	operator()(PhoneInfo, PhoneInfo)	Público
SintData	getSintData(DiphoInfo*, int)	Público
void	decodeDiphone(float*, unsigned*)	Privado
void	dequantLSF(QLSFWord&, float*)	Privado
void	interpolateLSFValues(float*, float*, int, float*)	Privado
void	lsf2lpc(float*, float*)	Privado
void	polyMult(float*, float*, float*, int, int)	Privado
void	genInputSignal(float, float, float*, int, float*, const float*, float*)	Privado
void	genOutputSamples(float*, float*, float*, float*)	Privado
void	dequantGains(QGainWord&, QGainWord&, float*, float*)	Privado

A.2.4 Métodos da Classe *CPSOLA*

Retorno	**Método**	**Tipo**
void	doSynthesis(SintData&, unsigned*, float*, int)	Público
VecData<unsigned>	getSintF0Marks(SintData&)	Público
void	getMeans(SintData&, float*, float&)	Privado
void	normSamples(SintData&)	Privado
void	applyWindow(float*, int, int)	Privado
void	applyLeftSideWindow(float*, int)	Privado
float	applyRightSideWindow(float*, int)	Privado
float	ga(int)	Privado
float	gp(int, int)	Privado
float	getF0Value(int, SintData&)	Privado

A.2.5 Métodos da Classe *CSound*

Retorno	**Método**	**Tipo**
N/A	CSound()	Público
void	setOutVec(VecData<float>&)	Público
void	codecInit()	Privado
void	clrSPT0Regs()	Privado
void	progSPORT0Regs()	Privado
void	progDMACtrl()	Privado
void	codecReset()	Privado
static void	sendSamples(int)	Privado
static void	rxIRQ(int)	Privado

Apêndice B

Códigos Fonte

Este capítulo apresenta todos os códigos fontes desenvolvidos para a realização do sistema apresentado. Este apêndice tem como finalidade, fornecer informações mais técnicas e aprofundadas sobre o sistema desenvolvido. Os códigos aqui descritos estão comentados, para facilitar a compreensão dos mesmos.

B.1 *Interface* de Usuário

B.1.1 Actions.java

```
/**
 * <p>Title: class Actions</p>
 * <p>Description: Implements the methods that will treat the system events (menu, buttons, etc)</p>
 * <p>Copyright: Copyright (c) Rodrigo Coura Torres 2003</p>
 * <p>Company: LPS / COPPE / EE / UFRJ</p>
 * @author Rodrigo Coura Torres
 * @version 1.0
 */

import javax.swing.*;
import java.io.*;
import java.util.StringTokenizer;


public class Actions
{
    // Class Constants.
    private static final String EXIT_MSG = "O texto não está salvo!\nDeseja salvá-lo?";
    private static final String FILE_SAVE_ERROR_MSG = "O arquivo não pode ser salvo!";
    private static final String FILE_OPEN_ERROR_MSG = "O arquivo não pode ser aberto!";
    private static final String LIST_CLEAR_QUESTION_MSG = "Deseja realmente cancelar a síntese do restante do texto?";
    private static final String LIST_CLEAR_MSG = "Síntese Cancelada!";
    private static final String EXIT_DLG_CAPTION = "Atenção!";
    private static final String FILE_DLG_CAPTION = "Erro!";
    private static final String LIST_CLEAR_QUESTION_DLG_CAPTION = "Atenção!";
    private static final String LIST_CLEAR_DLG_CAPTION = "Confirmação!";
    private static final String ID_YES = "Sim";
    private static final String ID_NO = "Não";
    private static final String ID_OK = "OK";
    private final int MAX_SENTENCE_SIZE = 20; // The maximum sentence size the DSP can hold.
    private final String TEXT_DELIMITERS = "\r\n,.:;?/\\!-()";

    private window win;
    private boolean fileUpdated = true;
    private File file = null;
    private CSerial serialInt;

    // Attributes for backup purpouses.
    File bckFile;
    boolean bckFileUpdated;


    /**
```

```
 * Class constructor
 * @param Win: the window (from class window) wich events will be treated by this instance.
 */
public Actions (window Win)
{
    win = Win;
    serialInt = new CSerial(Win);
}

/**
 * Save the file opened/modified information flags, in case of anything
 * goes wrong during save/open processes.
 */
private void saveFileStates()
{
    bckFileUpdated = fileUpdated;
    bckFile = file;
}

/**
 * Restore the file opened/modified information flags, if anything
 * went wrong during save/open processes.
 */
private void restoreFileStates()
{
    fileUpdated = bckFileUpdated;
    file = bckFile;
}


/**
 * Create a new file which will contains the JTextArea text.
 * @return true if the file was successifully created and saved.
 */
public boolean saveFileAs()
{
    final JFileChooser dlg = new JFileChooser();
    dlg.setFileSelectionMode(JFileChooser.FILES_ONLY);
    dlg.setFileFilter(new TextFilter());
    int res = dlg.showSaveDialog(win);

    this.saveFileStates();

    if (res == JFileChooser.APPROVE_OPTION)
    {
        file = dlg.getSelectedFile();

        try
        {
            FileWriter f = new FileWriter(file);
            f.write(win.TAText.getText());
            f.close();
            fileUpdated = true;
            return true;
        }
        catch (IOException e)
        {
            this.restoreFileStates();
            Object opt[] = {ID_OK};
            JOptionPane.showOptionDialog(win, FILE_SAVE_ERROR_MSG, FILE_DLG_CAPTION, JOptionPane.OK_OPTION,
                JOptionPane.ERROR_MESSAGE, null, opt, opt[0]);
            return false;
        }
    }

    this.restoreFileStates();
    return false;
}

/**
 * Saves the screen text in a already existing file. If the file doesn´t
 * exists, the method call the saveAs method.
 * @return true if the save process was successfull.
 */
public boolean saveFile()
{
    if (file == null) return saveFileAs();

    this.saveFileStates();

    try
    {
        FileWriter f = new FileWriter(file);
        f.write(win.TAText.getText());
        f.close();
        fileUpdated = true;
        return true;
    }
    catch(IOException e)
    {
        this.restoreFileStates();
        Object opt[] = {ID_OK};
        JOptionPane.showOptionDialog(win, FILE_SAVE_ERROR_MSG, FILE_DLG_CAPTION, JOptionPane.OK_OPTION,
            JOptionPane.ERROR_MESSAGE, null, opt, opt[0]);
        return false;
    }
}

/**
```

```
 * Close an existing file (saving it if any modification was made)
 * and erases the text area.
 */
public void newFile()
{
    if (!fileUpdated)
    {
        Object[] options = {ID_YES, ID_NO};
        int opt = JOptionPane.showOptionDialog(win, EXIT_MSG, EXIT_DLG_CAPTION, JOptionPane.YES_NO_OPTION,
            JOptionPane.QUESTION_MESSAGE, null, options, options[0]);

        if (opt == JOptionPane.YES_OPTION)
        {
            if (!saveFile()) return;
        }
        else if (opt == JOptionPane.CLOSED_OPTION) return;
    }

    win.TAText.setText("");
    file = null;
    fileUpdated = true;
}

/**
 * Does the same as newFile.
 */
public void closeFile()
{
    newFile();
}

/**
 * Verifies if the text is saved and exit if everything is OK.
 */
public void closeApplication()
{
    if (!fileUpdated)
    {
        Object[] options = {ID_YES, ID_NO};
        int opt = JOptionPane.showOptionDialog(win, EXIT_MSG, EXIT_DLG_CAPTION, JOptionPane.YES_NO_OPTION,
            JOptionPane.QUESTION_MESSAGE, null, options, options[0]);

        if (opt == JOptionPane.YES_OPTION)
        {
            if (!saveFile()) return;
        }
        else if (opt == JOptionPane.CLOSED_OPTION) return;
    }
    System.exit(0);
}


/**
 * Open a file, but first verifies if a possible opened file is saved.
 */
public void openFile()
{
    if (!fileUpdated)
    {
        Object[] options = {ID_YES, ID_NO};
        int opt = JOptionPane.showOptionDialog(win, EXIT_MSG, EXIT_DLG_CAPTION, JOptionPane.YES_NO_OPTION,
            JOptionPane.QUESTION_MESSAGE, null, options, options[0]);

        if (opt == JOptionPane.YES_OPTION)
        {
            if (!saveFile()) return;
        }
        else if (opt == JOptionPane.CLOSED_OPTION) return;
    }

    final JFileChooser dlg = new JFileChooser();
    dlg.setFileSelectionMode(JFileChooser.FILES_ONLY);
    dlg.setFileFilter(new TextFilter());
    int res = dlg.showOpenDialog(win);
    this.saveFileStates();

    if (res == JFileChooser.APPROVE_OPTION)
    {
        file = dlg.getSelectedFile();

        try
        {
            InputStream f = new FileInputStream(file);
            byte buff[] = new byte [(int) file.length()];
            f.read(buff);
            win.TAText.setText(new String(buff));
            fileUpdated = true;
        }
        catch (IOException e)
        {
            this.restoreFileStates();
            Object opt[] = {ID_OK};
            JOptionPane.showOptionDialog(win, FILE_OPEN_ERROR_MSG, FILE_DLG_CAPTION, JOptionPane.OK_OPTION,
                JOptionPane.ERROR_MESSAGE, null, opt, opt[0]);
            openFile();
        }
    }
}
```

```
/**
 * This method must be called when the screen text becomes diferent
 * from the text saved in a file.
 */
public void setTextModified()
{
    fileUpdated = false;
}

/**
 * Cut the select text to the clipboard.
 */
public void cutText()
{
    win.TAText.cut();
}

/**
 * Copy the select text to the clipboard.
 */
public void copyText()
{
    win.TAText.copy();
}

/**
 * Selects the entire text.
 */
public void selectAll()
{
    win.TAText.selectAll();
}

/**
 * Paste the text from the clipboard to the cursor position.
 */
public void pasteText()
{
    win.TAText.paste();
}

/**
 * Takes the typed text, process it, splitting in smaller sentences
 * an send through the serial port to the DSP.
 */
public void sendText()
{
    String text = win.TAText.getText();
    StringTokenizer st = new StringTokenizer(text, TEXT_DELIMITERS);

    boolean first = true;
    int startPos;
    int endPos;
    char size;

    while(st.hasMoreTokens())
    {
        String sentence = st.nextToken();
        startPos = 0;

        // This while brakes the sentences greater larger than MAX_SENTENCE_SIZE
        // in smaller sentences of maximum size of MAX_SENTENCE_SIZE.
        while ( (sentence.length() - startPos) > MAX_SENTENCE_SIZE)
        {
            String aux = sentence.substring(startPos, (startPos + MAX_SENTENCE_SIZE));
            endPos = aux.lastIndexOf(' ');

            if (endPos > 0)
            {
                endPos += startPos;
                String formatedData = sentence.substring(startPos, endPos);
                size = (char) formatedData.length();
                String data = size + formatedData;
                serialInt.writeData(data, first);
                first = false;
                startPos = endPos + 1;
            }
            else
            {
                StringTokenizer temp = new StringTokenizer(sentence.substring(startPos), " \r\n");
                String word = temp.nextToken();
                Object opt[] = {ID_OK};
                JOptionPane.showOptionDialog(win, "A palavra \"" + word + "\" é maior do que o máximo permitido e será descartada!",
                                             FILE_DLG_CAPTION,
                                             JOptionPane.OK_OPTION,
                                             JOptionPane.ERROR_MESSAGE, null,
                                             opt, opt[0]);

                startPos += word.length() + 1;
                first = true; // the synchronism was lost due to this error message.
            }
        }

        // The code below processes the last piece of the sentence, or the
        // single one, if the sentence is already smaller that MAX_SENTENCE_SIZE.
        if (startPos < sentence.length())
        {
            String formatedData = sentence.substring(startPos);
            size = (char) formatedData.length();
```

```
                String data = size + formatedData;
                serialInt.writeData(data, first);
                first = false;
            }
        }
    }


    /**
     * This method clears the output sentences list, so, the transmiter
     * will not send any remaining sentences.
     */
    public void clearList()
    {
        Object[] options = {ID_YES, ID_NO};
        int opt = JOptionPane.showOptionDialog(win, LIST_CLEAR_QUESTION_MSG, LIST_CLEAR_QUESTION_DLG_CAPTION, JOptionPane.YES_NO_OPTION,
            JOptionPane.QUESTION_MESSAGE, null, options, options[1]);

        if (opt == JOptionPane.YES_OPTION)
        {
            serialInt.emptyList();
            Object opt2[] = {ID_OK};
            JOptionPane.showOptionDialog(win, LIST_CLEAR_MSG, LIST_CLEAR_DLG_CAPTION, JOptionPane.OK_OPTION,
                JOptionPane.INFORMATION_MESSAGE, null, opt2, opt2[0]);
        }
    }
}
```

B.1.2 CSerial.java

```
/**
 * <p>Title: CSerial</p>
 * <p>Description: Class that send the data to the DSP trough the serial port.</p>
 * <p>Copyright: Copyright (c) Rodrigo Coura Torres 2004</p>
 * <p>Company: LPS / COPPE / UFRJ</p>
 * @author Rodrigo Coura Torres
 * @version 1.0
 */

import java.io.*;
import java.util.*;
import javax.comm.*;
import javax.swing.*;

public class CSerial implements SerialPortEventListener
{
    private final String comName = "COM2";
    private final String appName = "CSerial.class";
    private final int baudRate = 115200;

    private static final String SERIAL_TX_ERROR_DLG_CAPTION = "Erro na transmissão serial!\n";

    private static final String SERIAL_ERROR_DLG_CAPTION = ": porta serial inexistente!\n";
    private static final String SERIAL_OPEN_DLG_CAPTION = ": porta serial em uso!\n";
    private static final String SERIAL_STREAM_DLG_CAPTION = "Não foi possível pegar a stream de saída!\n";
    private static final String SERIAL_CONF_DLG_CAPTION = "Configuração serial inválida!\n";
    private static final String SERIAL_LIST_DLG_CAPTION = "Excesso de gerenciadores de eventos incluídos!\n";
    private static final String FILE_DLG_CAPTION = "Erro!";
    private static final String ID_OK = "OK";

    private CommPortIdentifier comPort;
    private SerialPort serialPort;
    private OutputStream out;
    private boolean recReady = false;
    private LinkedList textList;
    private window win;

    /**
     * Class constructor. Activates the serial interface between the Host computer
     * and the DSP.
     * @param Win The main programm dialog.
     */
    public CSerial(window Win)
    {
        try
        {
            // Getting the main dialog window.
            win = Win;

            // Taking the serial port we will use.
            comPort = CommPortIdentifier.getPortIdentifier(comName);
            // Opening the serial port.
            serialPort = (SerialPort) comPort.open(appName, 500);
            // Getting the output stream.
            out = serialPort.getOutputStream();
            // Configuring the serial port.
            serialPort.setSerialPortParams(baudRate, SerialPort.DATABITS_8, SerialPort.STOPBITS_1, SerialPort.PARITY_NONE);
            // Setting the event listener.
            serialPort.notifyOnCTS(true);
            serialPort.addEventListener(this);
            // Configuring the text linkedlist.
            textList = new LinkedList();
        }
        catch(NoSuchPortException e)
        {
```

```
        Object opt[] = {ID_OK};
        JOptionPane.showOptionDialog(win, comName + SERIAL_ERROR_DLG_CAPTION + e.getMessage(),
                        FILE_DLG_CAPTION,
                        JOptionPane.OK_OPTION,
                        JOptionPane.ERROR_MESSAGE, null,
                        opt, opt[0]);
        System.exit(1);
    }
    catch (PortInUseException e)
    {
        Object opt[] = {ID_OK};
        JOptionPane.showOptionDialog(win, comName + SERIAL_OPEN_DLG_CAPTION + e.getMessage(),
                        FILE_DLG_CAPTION,
                        JOptionPane.OK_OPTION,
                        JOptionPane.ERROR_MESSAGE, null,
                        opt, opt[0]);
        System.exit(2);

    }
    catch(IOException e)
    {
        Object opt[] = {ID_OK};
        JOptionPane.showOptionDialog(win, SERIAL_STREAM_DLG_CAPTION + e.getMessage(),
                        FILE_DLG_CAPTION,
                        JOptionPane.OK_OPTION,
                        JOptionPane.ERROR_MESSAGE, null,
                        opt, opt[0]);
        System.exit(3);

    }
    catch (UnsupportedCommOperationException e)
    {
        Object opt[] = {ID_OK};
        JOptionPane.showOptionDialog(win, SERIAL_CONF_DLG_CAPTION + e.getMessage(),
                        FILE_DLG_CAPTION,
                        JOptionPane.OK_OPTION,
                        JOptionPane.ERROR_MESSAGE, null,
                        opt, opt[0]);
        System.exit(4);

    }
    catch (TooManyListenersException e)
    {
        Object opt[] = {ID_OK};
        JOptionPane.showOptionDialog(win, SERIAL_LIST_DLG_CAPTION + e.getMessage(),
                        FILE_DLG_CAPTION,
                        JOptionPane.OK_OPTION,
                        JOptionPane.ERROR_MESSAGE, null,
                        opt, opt[0]);
        System.exit(5);

    }
}

/**
 * It writes the data to a linked list, so, the data is transmited as soon as possible
 * and without lock the application during the process.
 * @param data The string to be transmited.
 * @param first A boolean value that indicates if the string is the first one
 * of a batch of sentences.
 */
public void writeData(String data, boolean first)
{
    textList.addLast(data);

    /* If is the first sentence, we need to explicit kick off the transmition
     because the event responds to the CTS change only, so, even if the CTS
     is authorizing a trsnmition, the host will not know, since the CTS is
     unchanged. So, forcing the first one, we send the first sentence, and after that
     the event catches automatically the CTS state changes, sending the rest of the
     sentences.
    */
    if (first) this.serialEvent(null);
}

public void serialEvent(SerialPortEvent ev)
{
    // Checks if the list is not empty and the CTS signal is requesting another
    // data.
    if ( (!textList.isEmpty()) && (!serialPort.isCTS()) )
    {
        try
        {
            String data = (String) textList.removeFirst();
            out.write(data.getBytes());
            out.flush(); //  Force the imediate transmition.
        }
        catch (IOException e)
        {
                Object opt[] = {ID_OK};
                JOptionPane.showOptionDialog(win, SERIAL_TX_ERROR_DLG_CAPTION + e.getMessage(),
                                             FILE_DLG_CAPTION,
                                             JOptionPane.OK_OPTION,
                                             JOptionPane.ERROR_MESSAGE, null,
                                             opt, opt[0]);
        }
    }
}
```

```
    /**
     * Empties the entire list, so, the trasmition of any pending data
     * is aborted.
     */
    public void emptyList()
    {
        textList.clear();
    }
}
```

B.1.3 TextFilter.java

```
/**
 * <p>Title: class TextFilter</p>
 * <p>Description: Class that extends the Fileilter class, so its filter the
 * FileChooser class results, in order to show only the text (ASCII) files. </p>
 * <p>Copyright: Copyright (c) Rodrigo Coura Torres 2003</p>
 * <p>Company: LPS / COPPE / EE / UFRJ</p>
 * @author Rodrigo Coura Torres
 * @version 1.0
 */

import java.io.File;
import javax.swing.filechooser.FileFilter;

public class TextFilter extends FileFilter
{
    private final String TEXT_FILES = "txt";
    private final String DESCRIPTION = "Text files (" + TEXT_FILES + ")";

    /**
     * File filter to accept all directories and text files
     * @param f : The file you wish to evaluate.
     */
    public boolean accept(File f)
    {
        if (f.isDirectory())
        {
            return true;
        }

        String ext = getExtension(f);

        if (ext != null)
        {
            if (ext.equals(TEXT_FILES))
            {
                    return true;
            } else
            {
                return false;
            }
        }

        return false;
    }

    /**
     * @return The file filter description.
     */
    public String getDescription()
    {
        return DESCRIPTION;
    }

    /**
     * Gets a file name extension.
     * @param f : An instance of class File, witch points to a eisting file.
     * @return the string containing the file extension.
     */
    private String getExtension(File f)
    {
        String ext = null;
        String s = f.getName();
        int i = s.lastIndexOf('.');

        if ( (i > 0) &&  (i < (s.length() - 1)))
        {
            ext = s.substring(i+1).toLowerCase();
        }

        return ext;
    }
}
```

B.1.4 sadf.java

```
/**
 * <p>Title: class sadf</p>
 * <p>Description: Programm main file (start point)</p>
 * <p>Copyright: Copyright (c) Rodrigo Coura Torres 2003</p>
```

```
 * <p>Company: LPS / COPPE / EE / UFRJ</p>
 * @author Rodrigo Coura Torres
 * @version 1.0
 */

public class sadf
{
    static final private String title = "Sistema de Apoio para Deficientes da Fala";
    static final private int WindowHeight = 500;
    static final private int WindowWidth = 500;

    public static void main(String argv[])
    {
        window win = new window();
        win.setTitle(title);
        win.setSize(WindowHeight, WindowWidth);
        win.setVisible(true);
    }
}
```

B.1.5 window.java

```
/**
 * <p>Title: class window</p>
 * <p>Description: It specifies the user interface window</p>
 * <p>Copyright: Copyright (c) Rodrigo Coura Torres 2003</p>
 * <p>Company: LPS / COPPE / EE / UFRJ</p>
 * @author Rodrigo Coura Torres
 * @version 1.0
 */

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class window extends JFrame implements ActionListener, KeyListener
{
    // Contant variables used across this class.

    // Button constants
    private final String BUTTON_NAME = "Falar";

    // Menu constants
    private final String MENU_FILE_NAME = "Arquivo";
    private final String MENU_EDIT_NAME = "Editar";
    private final String MENU_OPTIONS_NAME = "Opções";
    private final String FILE_NEW_NAME = "Novo";
    private final String FILE_OPEN_NAME = "Abrir";
    private final String FILE_CLOSE_NAME = "Fechar";
    private final String FILE_SAVE_NAME = "Salvar";
    private final String FILE_SAVEAS_NAME = "Salvar Como";
    private final String FILE_EXIT_NAME = "Sair";
    private final String EDIT_CUT_NAME = "Recortar";
    private final String EDIT_COPY_NAME = "Copiar";
    private final String EDIT_PASTE_NAME = "Colar";
    private final String EDIT_SELECT_ALL_NAME = "Selecionar Tudo";
    private final String OPTIONS_CLEAR_NAME = "Cancelar Síntese";
    private final int FILE_NEW_KEY_ID = KeyEvent.VK_N;
    private final int FILE_OPEN_KEY_ID = KeyEvent.VK_O;
    private final int FILE_CLOSE_KEY_ID = KeyEvent.VK_F4;
    private final int FILE_SAVE_KEY_ID = KeyEvent.VK_F2;
    private final int FILE_SAVEAS_KEY_ID = KeyEvent.VK_S;
    private final int FILE_EXIT_KEY_ID = KeyEvent.VK_F4;
    private final int EDIT_CUT_KEY_ID = KeyEvent.VK_X;
    private final int EDIT_COPY_KEY_ID = KeyEvent.VK_C;
    private final int EDIT_PASTE_KEY_ID = KeyEvent.VK_V;
    private final int EDIT_SELECT_ALL_KEY_ID = KeyEvent.VK_A;
    private final int OPTIONS_CLEAR_ID = KeyEvent.VK_Z;

    // Text Area constants.
    private final String FontName = "Arial";
    private final int FontSize = 16;
    private final int FontType = Font.PLAIN;

    // Window´s components.
    public JTextArea TAText;
    public JScrollPane Scrolls;
    public JButton BTalk;
    public JMenuBar MBMenuStr;
    public JMenu MFile;
    public JMenu MEdit;
    public JMenu MOptions;
    public JMenuItem MINew, MIOpen, MIClose, MISave, MISaveAs, MIExit;
    public JMenuItem MICut, MICopy, MIPaste, MISelectAll;
    public JMenuItem MIClear;
    public Container con;
    public Actions Action;

    /**
     * Deefault class constructor. Its generates the window and menu structures.
     */
    public window()
    {
        con = getContentPane();
        Action = new Actions(this);
```

```
    // Creating and adding the window´s components.
    generateTextArea();

    // Generating the button
    BTalk = new JButton(BUTTON_NAME);
    con.add(BTalk, BorderLayout.SOUTH);
    BTalk.addActionListener(this);

    // Generating the menu structure.
    this.generateMenu();

    // Adding listener.
    TAText.addKeyListener(this);

    // The Action.closeApplication will handle the exit event.
    this.setDefaultCloseOperation(window.DO_NOTHING_ON_CLOSE);

    addWindowListener(new WindowAdapter()
    {
        public void windowClosing(WindowEvent we)
        {
           Action.closeApplication();
        }
    });
}

/**
 * Creates and confugires the TextArea component.
 */
private void generateTextArea()
{
    TAText = new JTextArea();
    TAText.setFont(new Font(FontName, FontType, FontSize));
    TAText.setLineWrap(true);
    TAText.setWrapStyleWord(true);
    Scrolls = new JScrollPane(TAText);
    Scrolls.setVerticalScrollBarPolicy(JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);
    con.add(Scrolls, BorderLayout.CENTER);
}

/**
 * Generates the menu´s structure.
 */
private void generateMenu()
{
    MBMenuStr = new JMenuBar();
    MFile = new JMenu(MENU_FILE_NAME);
    MEdit = new JMenu(MENU_EDIT_NAME);
    MOptions = new JMenu(MENU_OPTIONS_NAME);
    MINew = new JMenuItem(FILE_NEW_NAME);
    MIOpen = new JMenuItem(FILE_OPEN_NAME);
    MIClose = new JMenuItem(FILE_CLOSE_NAME);
    MISave = new JMenuItem(FILE_SAVE_NAME);
    MISaveAs = new JMenuItem(FILE_SAVEAS_NAME);
    MIExit = new JMenuItem(FILE_EXIT_NAME);
    MICut = new JMenuItem(EDIT_CUT_NAME);
    MICopy = new JMenuItem(EDIT_COPY_NAME);
    MIPaste = new JMenuItem(EDIT_PASTE_NAME);
    MISelectAll = new JMenuItem(EDIT_SELECT_ALL_NAME);
    MIClear = new JMenuItem(OPTIONS_CLEAR_NAME);

    setJMenuBar(MBMenuStr);

    // Organizing the File Menu

    // Setting the shortcut keys.
    MINew.setAccelerator(KeyStroke.getKeyStroke(FILE_NEW_KEY_ID, ActionEvent.CTRL_MASK));
    MIOpen.setAccelerator(KeyStroke.getKeyStroke(FILE_OPEN_KEY_ID, ActionEvent.CTRL_MASK));
    MIClose.setAccelerator(KeyStroke.getKeyStroke(FILE_CLOSE_KEY_ID, ActionEvent.CTRL_MASK));
    MISave.setAccelerator(KeyStroke.getKeyStroke(FILE_SAVE_KEY_ID, ActionEvent.CTRL_MASK));
    MISaveAs.setAccelerator(KeyStroke.getKeyStroke(FILE_SAVEAS_KEY_ID, ActionEvent.CTRL_MASK));
    MIExit.setAccelerator(KeyStroke.getKeyStroke(FILE_EXIT_KEY_ID, ActionEvent.ALT_MASK));

    // Placing the listeners.
    MINew.addActionListener(this);
    MIOpen.addActionListener(this);
    MIClose.addActionListener(this);
    MISave.addActionListener(this);
    MISaveAs.addActionListener(this);
    MIExit.addActionListener(this);

    // Adding the File's menu itens
    MFile.add(MINew);
    MFile.add(MIOpen);
    MFile.add(MIClose);
    MFile.add(MISave);
    MFile.add(MISaveAs);
    MFile.addSeparator();
    MFile.add(MIExit);
    MBMenuStr.add(MFile);

    // Organizing the edit Menu.

    //Setting the shortcut keys.
    MICut.setAccelerator(KeyStroke.getKeyStroke(EDIT_CUT_KEY_ID, ActionEvent.CTRL_MASK));
    MICopy.setAccelerator(KeyStroke.getKeyStroke(EDIT_COPY_KEY_ID, ActionEvent.CTRL_MASK));
    MIPaste.setAccelerator(KeyStroke.getKeyStroke(EDIT_PASTE_KEY_ID, ActionEvent.CTRL_MASK));
    MISelectAll.setAccelerator(KeyStroke.getKeyStroke(EDIT_SELECT_ALL_KEY_ID, ActionEvent.CTRL_MASK));
```

```
        // Placing the listeners.
        MICut.addActionListener(this);
        MICopy.addActionListener(this);
        MIPaste.addActionListener(this);
        MISelectAll.addActionListener(this);

        // Adding the Edit's menu itens.
        MEdit.add(MICut);
        MEdit.add(MICopy);
        MEdit.add(MIPaste);
        MEdit.add(MISelectAll);
        MBMenuStr.add(MEdit);

        // Organizing the Options menu.

        // Setting the shortcut keys.
        MIClear.setAccelerator(KeyStroke.getKeyStroke(OPTIONS_CLEAR_ID, ActionEvent.CTRL_MASK));
        // Placing the listeners.
        MIClear.addActionListener(this);
        // Adding the Option's menu itens
        MOptions.add(MIClear);
        MBMenuStr.add(MOptions);
    }

    /**
     * Calls the specific method for an input event.
     * @param ae : the action that was peformed.
     */
    public void actionPerformed(ActionEvent ae)
    {
        String arg = (String) ae.getActionCommand();

        if (arg.equals(BUTTON_NAME))
        {
            Action.sendText();
        }
        else if (arg.equals(FILE_NEW_NAME))
        {
            Action.newFile();
        }
        else if (arg.equals(FILE_OPEN_NAME))
        {
            Action.openFile();
        }
        else if (arg.equals(FILE_CLOSE_NAME))
        {
            Action.closeFile();
        }
        else if (arg.equals(FILE_SAVE_NAME))
        {
            Action.saveFile();
        }
        else if (arg.equals(FILE_SAVEAS_NAME))
        {
            Action.saveFileAs();
        }
        else if (arg.equals(FILE_EXIT_NAME))
        {
            Action.closeApplication();
        }
        else if (arg.equals(EDIT_CUT_NAME))
        {
            Action.cutText();
        }
        else if (arg.equals(EDIT_COPY_NAME))
        {
            Action.copyText();
        }
        else if (arg.equals(EDIT_PASTE_NAME))
        {
            Action.pasteText();
        }
        else if (arg.equals(EDIT_SELECT_ALL_NAME))
        {
            Action.selectAll();
        }
        else if (arg.equals(OPTIONS_CLEAR_NAME))
        {
            Action.clearList();
        }
    }

    public void keyTyped(KeyEvent e)
    {
        Action.setTextModified();
    }

    public void keyPressed(KeyEvent e){}
    public void keyReleased(KeyEvent e){}
}
```

B.2 Sintetizador em DSP

B.2.1 cdatabase.h

```
//cdatabase.h

/*
This header file contains the CDataBase class declaration.
This class is responsible for the manipulation of the DataBase
*/

#ifndef CDATABASE_H

#include "structs.h"
#include "defines.h"

class CDataBase
{
private:
    static const unsigned dataBase[DBSIZE];                     // Vector containing the database information.
    static const float quantCB4[CB4ROWS][CB4COLS];              // Matrix with the quantization values with 4 bits resolution.
    static const float quantCB3[CB3ROWS][CB3COLS];              // Matrix with the quantization values with 3 bits resolution.
    static const char* phonList[NUMPHONES];                     // Vector with the considered phonemes of the database.
    static const float fixedCB[NFIXED];                         // Fixed codebook values.
    static const float adapCB[NADAP];                           // Adaptive codebook values.
    static const float qFixedGain[FIXED_QUANT_WEIGHT_VEC_SIZE]; // Vector with the quantized fixed gain values.
    static const float qAdapGain[ADAP_QUANT_WEIGHT_VEC_SIZE];   // Vector with the quantized adaptive gain values.
    unsigned *dataPos[NUMPHONES][NUMPHONES];                    // Matrix witch each cell poits to the begining of the data corresponding
                                                                // to the cell coordinates.
public:
    CDataBase();
    DiphoInfo operator()(PhoneInfo &phone1, PhoneInfo &phone2);     // returns the diphone index in the database.
    SintData getSintData(DiphoInfo *diphInd, int size);             // return the data structure to be sintesized.
private:
    // Decode the coded diphone.
    void decodeDiphone(float *vec, unsigned *addr);
    // Dequantize the quantized LSF values.
    void dequantLSF(QLSFWord &quantLSF, float* coefLSF);
    // Interpolates the LSF values,
    void interpolateLSFValues(float *coefLSF, float *prevCoefLSF, int subBlock, float *intLSF);
    // Gets the LPC values from the LSF coefs.
    void lsf2lpc(float *intCoefLSF, float *coefLPC);
    // Multiply two polynomials.
    void polyMult(float *pol1, float *pol2, float *polr, int ord1, int ord2);
    // Generates the input signal to be applied to the sint filter.
    void genInputSignal(float adapGain, float fixedGain, float *aCBInitAddr, int aCBInd, float *aCBBaseAddr, const float *fCB, float *out);
    // Generates the final audio result of a sub block.
    void genOutputSamples(float *inSignal, float *coefSint, float *vec, float *taps);
    // Dequantizes the quantized adap and fix gain values.
    void dequantGains(QGainWord &adap, QGainWord &fix, float *vAdap, float *vFix);
};

#ifdef CDATABASE_CPP
const char *CDataBase::phonList[NUMPHONES] = {"X", "p", "b", "t", "d", "k", "g", "f", "v", "s",
                                              "z", "S", "Z", "m", "n", "J", "l", "l~", "L", "r",
                                              "R", "i", "e", "E", "a", "6", "O", "o", "u", "@",
                                              "i~", "e~", "6~", "o~", "u~", "j", "w"};
#endif

#define CDATABASE_H

#endif
```

B.2.2 cpsola.h

```
//cpsola.h

/*
This header file declares the class that will apply the
sintesizing algorithm using the TD-PSOLA algorithm.
*/

#ifndef CPSOLA_H

#include "defines.h"
#include "structs.h"

class CPSOLA
{
public:
    void doSynthesis(SintData &data, unsigned *newF0Marcks, float *out, int &outSize);  // Perform the sinthesizing algorithm.
        VecData<unsigned> genSintF0Marcks(SintData &data);
private:
    void getMeans(SintData &data, float *means, float &sumMeans);                       // Calculates the means for normalization.
    void normSamples(SintData &data);                                                   // Normalize the input audio samples.
    void applyWindow(float *block, int left, int right);                                // Apply a non-uniform triangular window to a vector.
    void applyLeftSideWindow(float *block, int left);
    void applyRightSideWindow(float *block, int right);
    float ga(int t);
    float gp(int t, int dur);
```

```
    float getF0Value(int n, SintData &data);
};

#define CPSOLA_H

#endif
```

B.2.3 cserial.h

```
// CSerial.h

/*
    This class implements the communication between
    the DSP and a HOST computer through the RS-232 Interface.
*/

#ifndef CSERIAL_H

#include "structs.h"

class CSerial
{
private:
    static volatile VecData<char> text;
    static volatile int pos;

private:
    static char receiveWord();
    static void rxISR(int x);

public:
    CSerial(float baudRate);
    VecData<char> getText();
};

#define CSERIAL_H

#endif
```

B.2.4 csound.h

```
// cinterfaces.h

/*
    This class controls the access to the Board's Digital to
    Analog Converter.
*/

#ifndef CSOUND_H

#include <list>

#include "structs.h"
#include "defines.h"

class CSound
{
private:
    static volatile int     tx_buf[];
    static volatile int     rx_buf[];
    static int rcv_tcb[];
    static int xmit_tcb[];
    static int codecInitRegisters[];
    static list<VecData <float> > vecList;

public:
    CSound();
    void setOutVec(VecData<float> &vec)
    {
        vecList.push_back(vec);
    }

private:
    void codecInit();
    void clrSPT0Regs();
    void progSPORT0Regs();
    void progDMACtrl();
    void codecReset();
    static void sendSamples(int x);
    static void rxIRQ( int x){};
};

#define CSOUND_H

#endif
```

B.2.5 ctextproc.h

```
// ctextproc.h

/*
This include file dedclares the class that will be responsible for
the text processing (prosody analisys, syllables to phonemes conversion, etc).
*/

#ifndef CTEXTPROC_H

#include <list>
#include <cstring>

#include "defines.h"
#include "structs.h"

class CTextProc
{
public:
    TextInfo getPhonemes(char *text);
    void getPhonemesList(char *inText, list<PhoneInfo> &phone1, list<PhoneInfo> &phone2);
private:
    TextInfo textDecode(char *text);
    unsigned char *marckWords(unsigned char *strin);
    char *preTranscription(char *);
    unsigned char *marckSyllables(unsigned char *strin);
    unsigned char *marckStressedSyllable(unsigned char *strin);
    char *phoneticTranscription(char *strin);
    void init1252to850(unsigned char *);
    unsigned char conv2LowCaps(unsigned char ch);
    void wordProc(char *pal);
    unsigned char *splitWord(unsigned char *pal, unsigned char *out);
    void marckSyllable(unsigned char *pal);
    char *translateWord(char *pal, char *out);
    char *exceptionTable(char *out);
    void init850to1252(unsigned char * tab);
    bool phoneticVowel(unsigned char ch);
    bool sequentialConsonants(unsigned char c1, unsigned char c2);
    bool accentedLetter(unsigned char ch);
    bool accentedTermination(unsigned char *sil);
    bool accentedVowel(unsigned char ch);
    bool isVowel(unsigned char ch);
    bool isConsonant(unsigned char ch);
};

#define CTEXTPROC_H

#endif
```

B.2.6 defines.h

```
// defines.h

/*
    This header includes all the defines used in this programm.
*/

#ifndef DEFINES_H

/*************** DEFINES FOR MEMORY SEGMENTS ********/
#define DATABASE_MEM_SEG "database_seg"

/***** DEFINES FOR THE STRUCTS.H HEADER FILE *********/

// Defines of the SintData struct.
#define VINF_NUM_COLS   2   // The number of cols of the voiceInfovector.
#define PHONE_IND       0   // Specifies the col that contaisn the begining of the phoneme.
#define VOICE_INFO      1   // Specifies the col that contains the voice information of that phoneme.
#define STRESS_NUM_COLS 2   // The number of cols of the stressInfo vector.
#define T1              0   // Specifies the T1 mack position (Fujisaky model).
#define T2              1   // Specifies the T2 mack position (Fujisaky model).


/********* DEFINES OF THE CODEBOOKS.H HEADER FILE *******/
#define NFIXED  4135        // Fixed codebook size.
#define NADAP   4135        // Adaptive codebook size.


/********* DEFINES OF THE DATABASE.H HEADER FILE *******/
#define DBSIZE  39008       // Database size.


/********* DEFINES OF THE QUANTDATA.H HEADER FILE *******/
#define CB4ROWS                     2   // Number of coeffitients quantized with 4 bits.
#define CB4COLS                     16  // Number of quantized levels (4 bits).
#define CB3ROWS                     8   // Number of coeffitients quantized with 3 bits.
#define CB3COLS                     8   // Number of quantized levels (3 bits).
#define FIXED_QUANT_WEIGHT_VEC_SIZE 64  // Size of the vector with the quantized weight values.
#define ADAP_QUANT_WEIGHT_VEC_SIZE  64  // Size of the vector with the quantized weight values.


/********* DEFINES OF THE CDATABASE.H HEADER FILE *******/
#define NUMPHONES       37      // Number of considered phonemes.
#define NUM_SUBBLOCKS   4       // Number of sub-blocks of a coded blocks.
```

```
#define LPC_ORD         10      // The LPC filter order.
#define WFACTOR         0.8;    // The weight factor of the sint filter.
#define SUBWIN_SIZE     40      // The size (in samples) of a 5ms sub window size.
#define WIN_SIZE        160     // The size (in smples) of a processing window.


/***************** DEFINES OF THE CTEXTPROC.H HEADER FILE *****/
#define DIPH_VEC_SIZE           5   // The maximum size of a diphone if 4 chars.
#define PHONE_VEC_SIZE          3   // The maximum size of a phoneme if 2 chars.
#define STRESS_CODE             '1' // Code identifying a stressed syllable.
#define SYL_CODE                ' ' // Code identifying a syllable.
#define WORD_CODE               '2' // Code identifying the begining of a word.
#define NO_STRESSED             0   // Indicate that a diphone has any stressed phonemes.
#define FIRST_HALF_STRESSED     1   // Indicate that a diphone has the first phoneme stressed.
#define SECOND_HALF_STRESSED    2   // Indicate that a diphone has the second phoneme stressed.
#define FULL_STRESSED           3   // Indicate that a diphone has both phonemes stressed.
#define VALID_TEXT_DATA         0   // Indicate that the character is a valid phoneme char.
#define TamMaxPal       30
#define NumMaxSil       20
#define TamMaxSil       6
#define OUTex(x)        (*(out++)=(x))
#define OUTs(x)         (strcpy(out,(x)), out+=strlen(x))
#define  ESP            (char) 28       //Space
#define  PON            (char) 127      //Sentence marck
#define iFRA            (char) '\x10'   //Sentence begin
#define iPAL            (char) ' '      //179   //Word begin
#define  NUL            (char) 2        //Unrecognized character.
#define iSIL            (char) 7        //Syllable begining
#define sTON            (char) '\x18'   //24    //Begining of the stressed syllable.


/*************** DEFINES OF THE CPSOLA.H HEADER FILE *********/
#define NOT_VOICED          -10000000   // Specifies the mean value corresponding to a diphone that is not voiced.
#define VOICED_THRESHOLD    1           // Specifies the minimum value for a voiced phoneme.
#define OUT_SAMPLE_RATE     8000        // The output sample rate.

/*************** DEFINES OF THE CSERIAL.H HEADER FILE *********/
asm("#define CTS_FLAG FLG2");                   // The flag we will use as CTS pin.
asm("#define CTS_FLAG_ENABLE FLG20");           // The flag we will use as CTS pin enable.
#define BAUD_RATE               115200          // The serial baud rate.


/************** DEFINES OF THE CSOUND.H HEADER FILE (CODEC consts) *********/
#define OUT_GAIN 0.3    // This the gain applyed to the output samples.

// Codec register initializations
// Refer to AD1881 Data Sheet for register bit assignments
#define Select_LINE_INPUT   0x0404      // LINE IN - 0X0404, Mic In - 0x0000
#define Select_MIC_INPUT    0x0000
#define Line_Level_Volume   0x0000      // 0 dB for line inputs
#define Mic_Level_Volume    0x0F0F
#define Sample_Rate1        8000
#define Sample_Rate2        8000
#define FREQ_RATIO          6       // The ration between the SPORT frequency generation(48k) and the deesired sampling frequency (8k)

// AD1881 TDM Timeslot Definitions
#define     TAG_PHASE               0
#define     COMMAND_ADDRESS_SLOT    1
#define     COMMAND_DATA_SLOT       2
#define     STATUS_ADDRESS_SLOT     1
#define     STATUS_DATA_SLOT        2
#define     LEFT                    3
#define     RIGHT                   4

// Left and Right ADC valid Bits used for testing of valid audio data in current TDM frame
#define     M_Left_ADC              12
#define     M_Right_ADC             11
#define     DAC_Req_Left            0x80
#define     DAC_Req_Right           0x40


// AD1881 Codec Register Address Definitions
#define     REGS_RESET              0x0000
#define     MASTER_VOLUME           0x0200
#define     RESERVED_REG_1          0x0400
#define     MASTER_VOLUME_MONO      0x0600
#define     RESERVED_REG_2          0x0800
#define     PC_BEEP_Volume          0x0A00
#define     PHONE_Volume            0x0C00
#define     MIC_Volume              0x0E00
#define     LINE_IN_Volume          0x1000
#define     CD_Volume               0x1200
#define     VIDEO_Volume            0x1400
#define     AUX_Volume              0x1600
#define     PCM_OUT_Volume          0x1800
#define     RECORD_SELECT           0x1A00
#define     RECORD_GAIN             0x1C00
#define     RESERVED_REG_3          0x1E00
#define     GENERAL_PURPOSE         0x2000
#define     THREE_D_CONTROL_REG     0x2200
#define     RESERVED_REG_4          0x2400
#define     POWERDOWN_CTRL_STAT     0x2600
#define     SERIAL_CONFIGURATION    0x7400
#define     MISC_CONTROL_BITS       0x7600
#define     SAMPLE_RATE_GENERATE_0  0x7800
#define     SAMPLE_RATE_GENERATE_1  0x7A00
#define     VENDOR_ID_1             0x7C00
#define     VENDOR_ID_2             0x7E00
```

```
//  Mask bit selections in Serial Configuration Register for
//  accessing registers on any of the 3 codecs
#define     MASTER_Reg_Mask         0x1000
#define     SLAVE1_Reg_Mask         0x2000
#define     SLAVE2_Reg_Mask         0x4000
#define     MASTER_SLAVE1           0x3000
#define     MASTER_SLAVE2           0x5000
#define     MASTER_SLAVE1_SLAVE2    0x7000

// Macros for setting Bits 15, 14 and 13 in Slot 0 Tag Phase
#define     ENABLE_VFbit_SLOT1_SLOT2    0xE000
#define     ENABLE_VFbit_SLOT1          0xC000


// ad1881 RESET~ spec = 1.0(uS) min
// 80(MIPs) = 12.5 (nS) cycle time, therefore >=    80 cycles
#define     AD1881_RESET_CYCLES     80
// CYCLES1 + CYCLES2 = 80000 DSP instruction cycles total
// ad1881 warm-up = 1.0(mS)
// 80(MIPs) = 12.5 (nS) cycle time, therefore >= 80000 cycles
#define     AD1881_WARMUP_CYCLES1    20000
#define     AD1881_WARMUP_CYCLES2    60000


#define DEFINES_H

#endif
```

B.2.7 globals.h

```
// global.h

/*
This include file includes global function like error control
and identification, among others.
*/

#ifndef GLOBALS_H

#ifdef GLOBALS_CPP
#define DEF
#else
#define DEF extern
#endif

DEF int extHeapID;              // ID for the heap on the external memory.

void *mAlloc(int size);         // Memory allocation on the default heap.
void *mallocOutVec(int size);   // Memory allocation on the external memory.

#define GLOBALS_H

#endif
```

B.2.8 structs.h

```
// structs.h

/*
    This header files declares all structs that will be used in the
    programm.
*/

#ifndef STRUCTS_H

#include "defines.h"

/*
This struct holds the data needed by the sintesizer.
*/
typedef struct
{
    float *samples;         // The audio samples.
    unsigned sampleSize;    // The size of the audio samples vector.
    unsigned *f0Marcks;     // The F0 marcks vector.
    unsigned f0Size;        // The number of F0 marcks.
    unsigned **voiceInfo;   // Contains the voice information for each phoneme.
    unsigned voiceSize;     // The size of the voice vector.
    unsigned **stressInfo;  // The accent information for the Fujisaky model.
    unsigned stressSize;    // The size of the stress vector.

} SintData;


/*
This struct holds the data stored in the first word
of a specific diphone.
*/
typedef union
{
    struct
```

```
    {
        unsigned int numBlocks  :6;     // Number of codeed blocks.
        unsigned int diphSize   :14;    // Diphone's real size.
        unsigned int ph2        :6;     // Index of the second phoneme.
        unsigned int ph1        :6;     // Index of the first phoneme.
    } values;

    unsigned int word;
} ConfWord1;


/*
This struct holds the data stored in the second word
of a specific diphone.
*/
typedef union
{
    struct
    {
        unsigned int voiceInfo2 :3;     // Voice information of the second phoneme.
        unsigned int voiceInfo1 :3;     // Voice information of the first phoneme.
        unsigned int halphDiph  :14;    // Index of the first sample of the second phoneme.
        unsigned int f0CompSize :6;     // Number of words containing F0 marcks information.
        unsigned int f0Size     :6;     // Number of F0 marcks.
    } values;

    unsigned int word;
} ConfWord2;

/*
This struct holds the same kind of information (for instance f0Marcks)
that share the same data word.
*/
typedef union
{
    struct
    {
        unsigned int high   :16;    // Data stored in the high part of the word.
        unsigned int low    :16;    // Data stored in the low part of the word.
    } word;

    unsigned int value;
} DataWord;


/*
This struct holds the quantized LSF values.
*/
typedef union
{
    struct
    {
        unsigned int w9 :3; // 10th quantized LSF value.
        unsigned int w8 :3; // 9th quantized LSF value.
        unsigned int w7 :3; // 8th quantized LSF value.
        unsigned int w6 :3; // 7th quantized LSF value.
        unsigned int w5 :3; // 6th quantized LSF value.
        unsigned int w4 :3; // 5th quantized LSF value.
        unsigned int w3 :3; // 4th quantized LSF value.
        unsigned int w2 :3; // 3rd quantized LSF value.
        unsigned int w1 :4; // 2nd quantized LSF value.
        unsigned int w0 :4; // 1st quantized LSF value.
    } word;

    unsigned int value;
} QLSFWord;


/*
This struct holds the quantized gain values.
*/
typedef union
{
    struct
    {
        unsigned int g3 :8; // 1st gain value.
        unsigned int g2 :8; // 2nd gain value.
        unsigned int g1 :8; // 3rd gain value.
        unsigned int g0 :8; // 4th gain value.
    } word;

    unsigned int value;
} QGainWord;


/*
This structure is used for vectors information.
It carries the data, and also the size of the vector.
*/
template <class DataType> struct VecData
{
    DataType *data; // Pointer to the data.
    int size;       // Size of data.
};


/*
    This struct holds the text data and in the vector info,
    the data related to that text (word start, stress syllable, etc).
```

```
*/
typedef struct
{
    char *text;
    int *info;
    int size;
} TextInfo;


/*
    This struct holds a phoneme and the information about it.
*/
typedef struct
{
    char phoneme[PHONE_VEC_SIZE];
    int  info;
} PhoneInfo;


/*
    This struct hold the address to the diphone information
    and the information about stress syllables.
*/
typedef struct
{
    unsigned *addr;
    int stressInfo;
} DiphoInfo;

#define STRUCTS_H

#endif
```

B.2.9 cdatabase.cpp

```
//cdatabase.cpp

/*
    CDataBase class definition.
*/

#define CDATABASE_CPP

#include <cstring>
#include <cmath>
#include <cstdlib>

#include <21160.h>

#include "cdatabase.h"
#include "structs.h"
#include "defines.h"
#include "database.h"
#include "quantdata.h"
#include "codebooks.h"
#include "globals.h"

/*
This constructor generates the index matrix, where
the rows represent the index of the first phoneme, and the
cols, the index of the second phoneme. So, each cell contains
the index on the database vector corresponding to the begining
of the corresponding diphone information.
*/
CDataBase::CDataBase()
{
    ConfWord1 w1;
    ConfWord2 w2;
    const unsigned *p = dataBase;

    // Initializing the diphones matrix with zeros.
    memset(dataPos, 0, (NUMPHONES*NUMPHONES));

    while(p < &dataBase[(DBSIZE-1)])
    {
        // Getting the cell position.
        w1.word = *p;
        w2.word = *(p+1);

        // Saving the diphone's start position.
        dataPos[w1.values.ph1][w1.values.ph2] = (unsigned *) p;

        // Jumping to the next diphone's start position.
        p += ( w2.values.f0CompSize + ((NUM_SUBBLOCKS+3) * w1.values.numBlocks) + 2);
    }
}


/*
This method receives as a parameter the strings of each diphone's phonemes
the user wants ("X" and "o~", for instance). The method returns the index
corresponding to the begining  of the specified diphone information
on the database vector, and also the information about stress syllables..
*/
DiphoInfo CDataBase::operator()(PhoneInfo &phone1, PhoneInfo &phone2)
```

```
{
    int ind1, ind2;
    DiphoInfo ret;

    ind1 = ind2 = -1;

    for (int i=0; i<NUMPHONES; i++)
    {
        if ( !strcmp(phonList[i], phone1.phoneme) ) ind1 = i;
        if ( !strcmp(phonList[i], phone2.phoneme) ) ind2 = i;
    }

    ret.addr = ( (ind1 == -1) || (ind2 == -1) ) ? 0 : dataPos[ind1][ind2];

    if ( (phone1.info == VALID_TEXT_DATA) && (phone2.info == STRESS_CODE) )
    {
        ret.stressInfo = SECOND_HALF_STRESSED;
    }
    else if ( (phone1.info == STRESS_CODE) && (phone2.info == VALID_TEXT_DATA) )
    {
        ret.stressInfo = FIRST_HALF_STRESSED;
    }
    else if ( (phone1.info == STRESS_CODE) && (phone2.info == STRESS_CODE) )
    {
        ret.stressInfo = FULL_STRESSED;
    }
    else
    {
        ret.stressInfo = NO_STRESSED;
    }

    return ret;
}


/*
This method receives a vector of size "size" containing the
indexes of the diphones of the sentence to be sintesized.
The function returns a structure with the necessary information
for the sintesizer.
*/
SintData CDataBase::getSintData(DiphoInfo *diphInd, int size)
{
    int i, j, k, l, m, r, totalSize, diphSize, halphDiph;
    unsigned *pos;
    SintData sintData;
    ConfWord1 w1;
    ConfWord2 w2;
    DataWord data;

    sintData.sampleSize = sintData.f0Size = sintData.voiceSize = sintData.stressSize = 0;

    // Getting the amount of data we will need.
    for (i=0; i<size; i++)
    {
        // If the diphone doesn't exists, we proceed to the next one.
        if (!diphInd[i].addr)
        {
            if (diphInd[i].stressInfo == SECOND_HALF_STRESSED) sintData.stressSize++;
            continue;
        }

        pos = diphInd[i].addr;
        w1.word = *pos++;
        w2.word = *pos++;

        sintData.sampleSize += w1.values.diphSize;
        sintData.f0Size += w2.values.f0Size;
        sintData.voiceSize += 2;

        // Counting by the begining of a stressed syllable.
        if (diphInd[i].stressInfo == SECOND_HALF_STRESSED) sintData.stressSize++;
    }

    // Allocating memory.
    sintData.samples = (float *) mAlloc(sintData.sampleSize);
    sintData.f0Marcks = (unsigned *) mAlloc(sintData.f0Size);
    sintData.voiceInfo = (unsigned **) mAlloc(sintData.voiceSize);
    sintData.stressInfo = (unsigned **) mAlloc(sintData.stressSize);

    //Filling the structure with information for the sintesizer.
    totalSize = k = m = r = 0;

    for (i=0; i<size; i++)
    {
        if (!diphInd[i].addr)
        {
            // If the diphone doesn't exists, we use the end of the las diphone
            // as either the begining or the end of a stressed syllable..
            if (diphInd[i].stressInfo == SECOND_HALF_STRESSED)
            {
                sintData.stressInfo[r] = (unsigned *) mAlloc(STRESS_NUM_COLS);
                sintData.stressInfo[r][T1] = totalSize;
            }
            else if (diphInd[i].stressInfo == FIRST_HALF_STRESSED)
            {
                sintData.stressInfo[r][T2] = totalSize;
                r++;
            }
```

```
            continue;
        }

        pos = diphInd[i].addr;
        w1.word = *pos++;
        w2.word = *pos++;

        diphSize = w1.values.diphSize;
        halphDiph = w2.values.halphDiph;

        // Getting the correct indexex of the F0  marcks.
        for (j=0; j< (int) (w2.values.f0CompSize-1); j++)
        {
            data.value = *pos++;
            sintData.f0Marcks[k++] = data.word.low + totalSize;
            sintData.f0Marcks[k++] = data.word.high + totalSize;
        }

        if (w2.values.f0Size)
        {
            // If the size is an odd number, the last
            // f0 marck takes the whole word.
            data.value = *pos++;

            if (w2.values.f0Size & 0x1)
            {
                sintData.f0Marcks[k++] = data.value + totalSize;
            }
            else
            {
                sintData.f0Marcks[k++] = data.word.low + totalSize;
                sintData.f0Marcks[k++] = data.word.high + totalSize;
            }
        }

        // Storing the indexes of the stressed syllables.
        if (diphInd[i].stressInfo == SECOND_HALF_STRESSED)
        {
            sintData.stressInfo[r] = (unsigned *) mAlloc(STRESS_NUM_COLS);
            sintData.stressInfo[r][T1] = halphDiph + totalSize;
        }
        else if (diphInd[i].stressInfo == FIRST_HALF_STRESSED)
        {
            sintData.stressInfo[r][T2] = halphDiph + totalSize;
            r++;
        }

        // Saving the voice information of each phoneme.
        sintData.voiceInfo[m] = (unsigned *) mAlloc(VINF_NUM_COLS);
        sintData.voiceInfo[m][PHONE_IND] = totalSize;
        sintData.voiceInfo[m++][VOICE_INFO] = w2.values.voiceInfo1;
        sintData.voiceInfo[m] = (unsigned *)  mAlloc(VINF_NUM_COLS);
        sintData.voiceInfo[m][PHONE_IND] = halphDiph + totalSize;
        sintData.voiceInfo[m++][VOICE_INFO] = w2.values.voiceInfo2;

        decodeDiphone(&sintData.samples[totalSize], diphInd[i].addr);

        totalSize += diphSize;
    }

    return sintData;
}


/*
This method uses a CELP decoder to restore the original
diphone's samples. The encoded data is stored on the database vector
at the "addr" start position.
*/
void CDataBase::decodeDiphone(float *outVec, unsigned *addr)
{
    int i, j, k;
    float *adapPos;
    ConfWord1 w1;
    ConfWord2 w2;
    DataWord auxAdap, auxFix;
    QLSFWord quantLSF;
    QGainWord quantFixGain, quantAdapGain;
    float *vec, *baseVec;
    float adapGain[NUM_SUBBLOCKS];
    float fixedGain[NUM_SUBBLOCKS];
    int adapInd[NUM_SUBBLOCKS];
    int fixedInd[NUM_SUBBLOCKS];
    float coefLSF[LPC_ORD];
    float prevCoefLSF[LPC_ORD];
    float intCoefLSF[LPC_ORD];
    float coefLPC[LPC_ORD];
    float inSignal[SUBWIN_SIZE];
    float aCB[NADAP];
    float taps[LPC_ORD];

    w1.word = *addr++;
    w2.word = *addr++;

    int numBlocks = w1.values.numBlocks;
    int diphSize = w1.values.diphSize;

    // Getting the start addresses.
```

```
    unsigned *lsfAddr = addr + w2.values.f0CompSize;
    unsigned *aGainAddr = lsfAddr + numBlocks;
    unsigned *fGainAddr = aGainAddr + numBlocks;
    unsigned *aIndAddr = fGainAddr + numBlocks;
    unsigned *fIndAddr = aIndAddr + ((NUM_SUBBLOCKS/2)*numBlocks);

    // generating a fresh copy of the adaptive codebook.
    memcpy(aCB, adapCB, NADAP);

    // Initializing the prevVector with zeros.
    memset(coefLSF, 0, LPC_ORD);
    memset(taps, 0, LPC_ORD);

    // This var will point to the position where the new
    // adaptive codebbok samples will be placed.
    adapPos = aCB;

    // Allocating mamory for the output results.
    baseVec = (float *) mAlloc(numBlocks*WIN_SIZE);
    vec = baseVec;

    // Decoding the phoneme.
    for (i=0; i<numBlocks; i++)
    {
        // Getting the quantized LSF values
        quantLSF.value = *lsfAddr++;

        // Getting the adaptive and fixed gain values and dequantizing them.
        quantAdapGain.value = *aGainAddr++;
        quantFixGain.value = *fGainAddr++;
        dequantGains(quantAdapGain, quantFixGain, adapGain, fixedGain);

        // Getting the fixed and adaptive indexes.
        k = 0;
        while (k < NUM_SUBBLOCKS)
        {
            auxAdap.value = *aIndAddr++;
            auxFix.value = *fIndAddr++;
            adapInd[k] = auxAdap.word.low;
            fixedInd[k++] = auxFix.word.low;
            adapInd[k] = auxAdap.word.high;
            fixedInd[k++] = auxFix.word.high;
        }

        // Saving the previous LSF values.
        memcpy(prevCoefLSF, coefLSF, LPC_ORD);

        // Dequantizing the LSF values.
        dequantLSF(quantLSF, coefLSF);

        for (j=0; j<NUM_SUBBLOCKS; j++)
        {
            if (i)
            {
                // Interpolating the LSF values.
                interpolateLSFValues(coefLSF, prevCoefLSF, j, intCoefLSF);
            }
            else
            {
                memcpy(intCoefLSF, coefLSF, LPC_ORD);
            }

            // Getting the original LPC coefs.
            lsf2lpc(intCoefLSF, coefLPC);

            // Getting the input signal and updating
            // the adaptive codebook.
            genInputSignal(adapGain[j], fixedGain[j], adapPos, adapInd[j], aCB, &fixedCB[fixedInd[j]], inSignal);

            // Updating the adaptive codebook and pointing to the
            // next position for updating and codebook starting point.
            adapPos = (float *) circptr(adapPos, SUBWIN_SIZE, aCB, NADAP);

            // Generating the final output and
            // Pointing to the next sub block start position.
            genOutputSamples(inSignal, coefLPC, vec, taps);
            vec += SUBWIN_SIZE;
        }
    }

    // Copying just the original number of samples
    // to the real output vector.
    memcpy(outVec, baseVec, diphSize);
    free(baseVec);
}


/*
    Its returns on "coefLSF" the original LSF values witch were
    quantized and stored on the "quantLSF" var.
*/
void CDataBase::dequantLSF(QLSFWord &quantLSF, float* coefLSF)
{
    // Dequanting and restoring the original, non differential values.
    coefLSF[0] = quantCB4[0][quantLSF.word.w0];
    coefLSF[1] = quantCB4[1][quantLSF.word.w1] + coefLSF[0];
    coefLSF[2] = quantCB3[0][quantLSF.word.w2] + coefLSF[1];
    coefLSF[3] = quantCB3[1][quantLSF.word.w3] + coefLSF[2];
    coefLSF[4] = quantCB3[2][quantLSF.word.w4] + coefLSF[3];
```

```
    coefLSF[5] = quantCB3[3][quantLSF.word.w5] + coefLSF[4];
    coefLSF[6] = quantCB3[4][quantLSF.word.w6] + coefLSF[5];
    coefLSF[7] = quantCB3[5][quantLSF.word.w7] + coefLSF[6];
    coefLSF[8] = quantCB3[6][quantLSF.word.w8] + coefLSF[7];
    coefLSF[9] = quantCB3[7][quantLSF.word.w9] + coefLSF[8];
}


/*
    Interpolate the LSF values.
    The parameters are : the current LSF values (coefLSF),
    the previous coefLSF (prevCoefLSF) and the subblock number.
*/
void CDataBase::interpolateLSFValues(float *coefLSF, float *prevCoefLSF, int subBlock, float *intLSF)
{
    const float q[] = {0.25, 0.5, 0.75, 1.0};

    // Making the interpolation
    for (int i = 0; i < LPC_ORD; i++)
    {
        intLSF[i] = ( (1 - q[subBlock]) * prevCoefLSF[i] ) + ( q[subBlock]  * coefLSF[i] );
    }
}


/*
    Restores the original LPC values from the LFS values.
    "intCoefLSF" contains the interpolated LSF values, and
    the restored LPC values are stored in coefLPC.
*/
void CDataBase::lsf2lpc(float *intCoefLSF, float *coefLPC)
{
    float pa1[3], pa2[3], pa3[3], pa4[3], pa5[3], pa6[5], pa7[5], pa8[9];
    float qa1[3], qa2[3], qa3[3], qa4[3], qa5[3], qa6[5], qa7[5], qa8[9];
    float p[11], q[11];
    int j;

    // Calculating the Q(z) and P(z) polynomials
    pa1[0] = pa1[2] = pa2[0] = pa2[2] = pa3[0] = pa3[2] = pa4[0] = pa4[2] = pa5[0] = pa5[2] = 1.0;
    qa1[0] = qa1[2] = qa2[0] = qa2[2] = qa3[0] = qa3[2] = qa4[0] = qa4[2] = qa5[0] = qa5[2] = 1.0;

    pa1[1] = -2*cos(intCoefLSF[0]);
    pa2[1] = -2*cos(intCoefLSF[2]);
    pa3[1] = -2*cos(intCoefLSF[4]);
    pa4[1] = -2*cos(intCoefLSF[6]);
    pa5[1] = -2*cos(intCoefLSF[8]);

    qa1[1] = -2*cos(intCoefLSF[1]);
    qa2[1] = -2*cos(intCoefLSF[3]);
    qa3[1] = -2*cos(intCoefLSF[5]);
    qa4[1] = -2*cos(intCoefLSF[7]);
    qa5[1] = -2*cos(intCoefLSF[9]);

    polyMult(pa1, pa2, pa6, 2, 2);
    polyMult(pa3, pa4, pa7, 2, 2);
    polyMult(pa6, pa7, pa8, 4, 4);
    polyMult(pa8, pa5, p, 8, 2);

    polyMult(qa1, qa2, qa6, 2, 2);
    polyMult(qa3, qa4, qa7, 2, 2);
    polyMult(qa6, qa7, qa8, 4, 4);
    polyMult(qa8, qa5, q, 8, 2);

    // Calculating the LPCs
    for (j = 1; j <= LPC_ORD; j++)
    {
        coefLPC[j-1] = -(p[j] + p[j - 1] + q[j] - q[j - 1]) / 2;
    }
}


/*
    Multiplies two polynomials (poli1 and poli2).
    Each one with order ord1 and ord2, resp.
    The output´polyniomial is placed in "polr".
*/
void CDataBase::polyMult(float *pol1, float *pol2, float *polr, int ord1, int ord2)
{
    // Local variables
    int i, j;

    // Multiplying the polynomials
    for (j = 0; j <= (ord1 + ord2); j++)
    {
        polr[j] = 0;
        for (i = 0; i <= ord1; i++)
        {
            if ((j - i) >= 0 && (j - i) <= ord2)
            {
                polr[j] += pol1[i] * pol2[j - i];
            }
        }
    }
}



/*
    This function takes the adaptive gain (adapGain), the fixed gain
```

```
    (fixedGain), the adaptive codebook start address (aCB) and the fixed
    codebook start address (fCB) and generates the complet input signal
    that will be applied to the sintezis filter. The output response
    is stored in the "out" vector. The function also updates the adaptive
    codebook.
*/
void CDataBase::genInputSignal(float adapGain, float fixedGain, float *aCBInitAddr, int aCBInd, float *aCBBaseAddr, const float *fCB, float *out)
{
    float *aInd = (float *) circptr(aCBInitAddr, aCBInd, aCBBaseAddr, NADAP);

    for (int i=0; i<SUBWIN_SIZE; i++)
    {
        out[i] = *aCBInitAddr = ( adapGain * (*aInd) ) + ( fixedGain * (*fCB++) );
        aCBInitAddr = (float *) circptr(aCBInitAddr, 1, aCBBaseAddr, NADAP);
        aInd = (float *) circptr(aInd, 1, aCBBaseAddr, NADAP);
    }
}


/*
    This function applyes the synthesis filter (coefSint) to the input
    signal "inSignal", and the result is stored in "vec" and taps holds
    the delayed samples.
*/
void CDataBase::genOutputSamples(float *inSignal, float *coefSint, float *vec, float *taps)
{
    float *p = taps;
    float acc;

    for (int i=0; i<SUBWIN_SIZE; i++)
    {
        acc = 0.0;

        for (int j=0; j<LPC_ORD; j++)
        {
            acc += ( coefSint[j] * (*p) );
            p = (float *) circptr(p, 1, taps, LPC_ORD);
        }

        p = (float *) circptr(p, -1, taps, LPC_ORD);
        vec[i] = *p = inSignal[i] + acc;
    }
}

/*
    This method takes the quantizeds gain values stored in "adap" e "fix" (adaptive
    and fixed gains, respectively), dequantize them and stores the results in the
    adaptive gain vector "vAdap" and the fixed gain vector "vFix".
*/
void CDataBase::dequantGains(QGainWord &adap, QGainWord &fix, float *vAdap, float *vFix)
{
    vAdap[0] = qAdapGain[adap.word.g0];
    vAdap[1] = qAdapGain[adap.word.g1];
    vAdap[2] = qAdapGain[adap.word.g2];
    vAdap[3] = qAdapGain[adap.word.g3];

    vFix[0] = qFixedGain[fix.word.g0];
    vFix[1] = qFixedGain[fix.word.g1];
    vFix[2] = qFixedGain[fix.word.g2];
    vFix[3] = qFixedGain[fix.word.g3];
}
```

B.2.10 cpsola.cpp

```
// cpsola.cpp

#include <cstring>
#include <cstdlib>
#include <cmath>

#include "cpsola.h"
#include "structs.h"
#include "defines.h"
#include "globals.h"


/*
    This method receives as input the struct containing the data that will be
    synthesized (data), and a vector containing the nem F0 marcks, as a result
    of the prosodical analysis. The output, syntesized vector is stored in "out",
    which has size = "outSize".
*/
void CPSOLA::doSynthesis(SintData &data, unsigned *newF0Marcks, float *out, int &outSize)
{
    unsigned i, j, k, left, right, size;
    unsigned analR, analL, sintR, sintL;
    unsigned analSize, sintSize;
    unsigned init, end, f0End;
    unsigned *f0, *nf0;
    float *block, *start;
    float *p = out;
    bool theLastWasVoiced = false;
    bool theNextIsVoiced;

    //Zeroing the output vector.
```

```
for (i=0; i<outSize; i++) out[i] = 0;

// Normalizing the voiced phonemes.
normSamples(data);

f0 = data.f0Marcks;
nf0 = newF0Marcks;

for (i=0; i<data.voiceSize; i++)
{
    // Is a voiced phoneme.
    if (data.voiceInfo[i][VOICE_INFO] > VOICED_THRESHOLD)
    {
        // F0 ptr points to the begining of the F0 marcks vector.
        // depending if the last was voiced or not. If not voiced, we
        // increment by 2, in order to start at the right position, since
        // the algorithm looks at the last two F0 marcks.
        if (!theLastWasVoiced)
        {
            f0 += 2;
            nf0 += 2;
        }

        f0End = (i == (data.voiceSize-1)) ? (data.f0Marcks[(data.f0Size-1)]) : data.voiceInfo[(i+1)][PHONE_IND];

        while ( (*f0 <= f0End) && (f0 <= &data.f0Marcks[data.f0Size]) )
        {
            // Calculating the block's left and right size.
            // For both analisys and sintesis F0 marcks.
            analR = *f0 - *(f0-1);
            analL = *(f0-1) - *(f0-2);
            analSize = analR + analL;

            sintR = *nf0 - *(nf0-1);
            sintL = *(nf0-1) - *(nf0-2);
            sintSize = sintR + sintL;

            if (sintSize < analSize)
            {
                right = sintR;
                left = sintL;
                size = sintSize;
            }
            else
            {
                right = analR;
                left = analL;
                size = analSize;
            }

            // Allocating the desired space.
            block = (float *) mAlloc(size*sizeof(float));

            // Getting the block of data.
            memcpy(block, &data.samples[*(f0-2)], size*sizeof(float));

            // Applying the non-symetric triangular window.
            applyWindow(block, left, right);

            // Applying the overlap and add.
            for(j=0, k=0; j<size; j++, k++)
            {
                *p++ += block[k];
            }

            p -= right;
            free(block);
            f0++;
            nf0++;
        }

        theLastWasVoiced = true;
    }
    else
    {

        if (theLastWasVoiced)
        {
            // Right and p uses the information of the last voiced phoneme.
            block = (float *) mAlloc(right*sizeof(float));

            // Getting the block of data and applying the window.
            memcpy(block, &data.samples[*(f0-1) - right], right*sizeof(float));
            applyLeftSideWindow(block, right);

            // Applying the overlap and add.
            for(j=0; j<right; j++)
            {
                *p++ += block[j];
            }

            free(block);
            init = *(f0-1);
        }
        else
        {
            init = data.voiceInfo[i][PHONE_IND];
        }
```

```
            if (i == (data.voiceSize-1))
            {
                end =  data.sampleSize;
                theNextIsVoiced = false;
            }
            else
            {
                if (data.voiceInfo[(i+1)][VOICE_INFO] > VOICED_THRESHOLD)
                {
                    end = *f0;
                    theNextIsVoiced = true;
                }
                else
                {
                    end = data.voiceInfo[(i+1)][PHONE_IND];
                    theNextIsVoiced = false;
                }
            }

            // Getting the size to be copied.
            end -= init;

            memcpy(p, &data.samples[init], end*sizeof(float));
            p += end;

            // Applying the right side of the hibrid window, if it is the case.
            if (theNextIsVoiced)
            {
                sintSize = *(nf0+1) - *nf0;
                analSize = *(f0+1) - *f0;
                size = (sintSize < analSize) ? sintSize : analSize;


                block = (float *) mAlloc(size*sizeof(float));

                // Getting the block of data and applying the window.
                memcpy(block, &data.samples[*f0], size*sizeof(float));
                applyRightSideWindow(block, size);

                // Applying the overlap and add.
                for(j=0; j<size; j++)
                {
                    *p++ += block[j];
                }

                free(block);
                p -= size;
            }

            theLastWasVoiced = false;

        }
    }

    // Updates the size of the vector with the length of valid data.
    outSize = (p - out);
}



/*
    Receives the data structure (data), and returns in "means" a vector
    with the mean vector of each phoneme and in "sumMeans", the mean's
    summation. Each a specific phoneme is not voiced, it has no mean,
    so, the NOT_VOICED value is used for identification.
*/
void CPSOLA::getMeans(SintData &data, float *means, float &sumMeans)
{
    int i, j;
    int end;
    int iBeg, iEnd;
    int k = 0;
    int pos = 0;
    int count;

    sumMeans = 0.0;
    for (i=0; i<data.voiceSize; i++) means[i] = 0;

    // Looking at each phoneme.
    for (i=0; i<data.voiceSize; i++)
    {
        if (data.voiceInfo[i][VOICE_INFO] > VOICED_THRESHOLD)
        {
            // The end will be either the end of the diphone,
            // or the begining of the next phoneme.
            end = (i == (data.voiceSize-1)) ? (data.f0Marcks[(data.f0Size-1)]) : data.voiceInfo[(i+1)][PHONE_IND];

            // Takes the first f0Marck of the next phoneme.
            // But in fact, we need the period just before
            // the last F0 period.
            count = 0;
            while (data.f0Marcks[k++] < end) count++;

            if (count > 4)
            {
                // The period we want resides 3 positions before the first
                // index of the next phoneme.
                iBeg = data.f0Marcks[(k - 3)];
                iEnd = data.f0Marcks[(k - 2)];
```

```
                // calculating the mean.
                for (j=iBeg; j<iEnd; j++)
                {
                    means[pos] += data.samples[j];
                }

                means[pos] /= (iEnd-iBeg);
            }
            else
            {
                means[pos] = 0;
            }

            // Adding the result to the mean's summation variable.
            sumMeans += means[pos++];
        }
    }
}


/*
    takes the samples in "data" structure and normalize them.
*/
void CPSOLA::normSamples(SintData &data)
{
    float *meanVec;
    float meanSum, normVal;
    int i, j, end, k;

    // Getting the mean values of each voiced phoneme,
    //as well as the mean summation.
    meanVec = (float *) mAlloc(data.voiceSize);
    getMeans(data, meanVec, meanSum);

    k = 0;

    // Looking at each phoneme to normalize.
    for (i=0; i<data.voiceSize; i++)
    {
        // Normalize only the voiced phonemes.
        if (data.voiceInfo[i][VOICE_INFO] > VOICED_THRESHOLD)
        {
            normVal = (meanSum - meanVec[k++]) / meanSum;

            end = (i == (data.voiceSize-1)) ? (data.sampleSize) : data.voiceInfo[(i+1)][PHONE_IND];

            for (j=data.voiceInfo[i][PHONE_IND]; j<end; j++)
            {
                data.samples[j] *= normVal;
            }
        }
    }

    free(meanVec);
}


/*
    Applies a non-uniform bartllet window in a vector. Parameters
    are the vector in wich the window will be applyed, the number of
    samples in the left and right sides of the window.
*/
void CPSOLA::applyWindow(float *block, int left, int right)
{
    applyLeftSideWindow(block, left);
    applyRightSideWindow( (block + left), right);
}


/*
    This method applies the left side of a triangular window.
    The result is stored in "block", so the original data is lost.
    The parameter "left" specifies the size of the left side window.
*/
void CPSOLA::applyLeftSideWindow(float *block, int left)
{
    float lv = (float) left;

    for (int i=0; i<left; i++)
    {
        *block++ *= ( (2 / ( 2*lv - 1)) * i );
    }
}


/*
    This method applies the right side of a triangular window.
    The result is stored in "block", so the original data is lost.
    The parameter "right" specifies the size of the right side window.
*/
void CPSOLA::applyRightSideWindow(float *block, int right)
{
    float rv = (float) right;

    for (int i=(right-1); i>=0; i--)
    {
        *block++ *= ( (2 / ( (2*rv) - 1)) * i );
    }
```

```
}


/*
    This function receives the struct that holds the
    information about the sentence we want to sintesize.
    Using this information, the fucntion returns the new
    F0 marcks vector, as a result of a prosodical analysis
    of the data.
*/
VecData<unsigned> CPSOLA::genSintF0Marcks(SintData &data)
{
    int i, f0End;
    float acc, gp_part;
    VecData<unsigned> ret;
    float invF0Val;
    int offset = 0;
    bool theLastWasVoiced = false;
    bool firstVoiced = true;

    ret.size = data.f0Size;
    ret.data = (unsigned *) mAlloc(ret.size);

    unsigned *f0 = data.f0Marcks;
    unsigned *nf0 = ret.data;

    for (i=0; i<data.voiceSize; i++)
    {
        // Is a voiced phoneme.
        if (data.voiceInfo[i][VOICE_INFO] > VOICED_THRESHOLD)
        {
            f0End = (i == (data.voiceSize-1)) ? (data.f0Marcks[(data.f0Size-1)]) : data.voiceInfo[(i+1)][PHONE_IND];

            if (firstVoiced)
            {
                offset = 0;
                firstVoiced = false;
            }

            if (!theLastWasVoiced)
            {
                *nf0++ = *f0++ + offset;
            }

            while(*f0 <= f0End)
            {
#if 1
                invF0Val = (1 / exp(getF0Value(*f0++, data)))*OUT_SAMPLE_RATE;
#else
                invF0Val = (1.0 / 50.0)*OUT_SAMPLE_RATE;
                f0++;
#endif

                *nf0 = *(nf0-1) + (int) invF0Val;
                nf0++;
            }

            offset = 0;
            theLastWasVoiced = true;
        }
        else
        {
            if (i != (data.voiceSize-1))
            {
                offset = (data.voiceInfo[(i+1)][VOICE_INFO] > VOICED_THRESHOLD) ? *f0 : data.voiceInfo[(i+1)][PHONE_IND];
                offset -=  (theLastWasVoiced) ? *(f0-1) : data.voiceInfo[i][PHONE_IND];
                theLastWasVoiced = false;
            }
        }
    }

    return ret;
}


/*
    This method is the impulse response of the accent control mechanism.
    for the Fujisaki model. It receives the index of the sample
    we want to calculate the new F0 value.
*/
float CPSOLA::ga(int t)
{
    const float gamma = 0.2;
    const float beta = 0.005;
    float ret;

    t *= (1000.0 / OUT_SAMPLE_RATE);

    if (t >= 0)
    {
        ret = 1 - ((1 + beta*t)*exp(-beta*t));
        return  (ret < gamma) ? ret : gamma;
    }

    return 0;
}


/*
```

```
    This method is the impulse response of the phrase control mechanism.
    for the Fujisaki model. It receives the index of the sample
    we want to calculate the new F0 value.
*/
float CPSOLA::gp(int t, int dur)
{
    float ret;

    float alpha = (float) 0.03;
    alpha *= (1800.0 / dur);

    t *= (1000.0 / OUT_SAMPLE_RATE);
    dur *= (1000.0 / OUT_SAMPLE_RATE);

    if (t >= 0)
    {
        ret = (alpha*alpha) * t * exp(-alpha*t);
        return (ret * 1000.0);
    }

    return 0;
}



/*
    This method takes the sinthesis information in "data"
    and the sample index we want to calculate the F0 frequency by using
    the fujisaki model. It returns the F0 frequency obtained.
*/
float CPSOLA::getF0Value(int n, SintData &data)
{
    float ret;
    const float Fb = (float) 100.0;
    float Ap = (float) 0.02;
    float Aa = (float) 1.1;
    const float T0 = (float) OUT_SAMPLE_RATE / 100.0;

    ret = logf(Fb) + Ap*gp( (n - T0), data.sampleSize);

    for (int i=0; i<data.stressSize; i++)
    {
        ret += Aa * ( ga((n - data.stressInfo[i][T1])) - ga((n - data.stressInfo[i][T2])) );
    }

    return ret;
}
```

B.2.11 cserial.cpp

```
//cserial.cpp

#include <cmath>

#include <cstring>
#include <cstdio>

#include <21160.h>
#include <def21160.h>
#include <cdef21160.h>
#include <signal.h>

asm("#include <def21160.h>");

#include "cserial.h"
#include "globals.h"
#include "structs.h"
#include "defines.h"


volatile VecData<char> CSerial::text;
volatile int CSerial::pos;


CSerial::CSerial(float baudRate)
{
#if 1
    const float DSPFreq = 80000000;

    // Setting the Flag 1 pin as output (CTS pin).
    asm("bit set mode2 CTS_FLAG_ENABLE;");

    // Clearing the SPORT 1 control registers.
    *pSTCTL1 = 0x0;
    *pSRCTL1 = 0x0;

    // Receive configuration word.
    unsigned conf = (SPEN | SLEN29 | ICLK | CKRE | RFSR | LRFS | LAFS);
    *pSRCTL1 = conf;

    // calculating the serial port frequency.
    float divVal = (DSPFreq / (6*baudRate)) - 1;

    // Rounding the value.
```

```
    float auxVal = divVal - floorf(divVal);
    float realDivVal = (auxVal > 0.5) ? ceilf(divVal) : floorf(divVal);

    // Seting the serial port clock divider register.
    *pRDIV1 = (int) realDivVal;

    //Configuring the serial port's interrupt.
    interruptf(SIG_SPR1I, rxISR);

    // Initializing the text structure.
    text.data = 0;
    text.size = 0;
    pos = -1;

    asm("bit clr flags CTS_FLAG;");
#endif
}


char CSerial::receiveWord()
{
    char ret = 0;

    asm("\
        r0 = dm(RX1);\
        r4 = fext r0 by 25:1;\
        r1 = fext r0 by 22:1;\
        r4 = r4 or fdep r1 by 1:1;\
        r1 = fext r0 by 19:1;\
        r4 = r4 or fdep r1 by 2:1;\
        r1 = fext r0 by 16:1;\
        r4 = r4 or fdep r1 by 3:1;\
        r1 = fext r0 by 13:1;\
        r4 = r4 or fdep r1 by 4:1;\
        r1 = fext r0 by 10:1;\
        r4 = r4 or fdep r1 by 5:1;\
        r1 = fext r0 by 7:1;\
        r4 = r4 or fdep r1 by 6:1;\
        r1 = fext r0 by 4:1;\
        r4 = r4 or fdep r1 by 7:1;\
        %0 = r4;" :
        "=d" (ret)
        );

    return ret;
}



void CSerial::rxISR(int x)
{
    char ret = receiveWord();

    if (pos < 0)
    {
        text.size = (int) ret;
    }
    else
    {
        text.data[pos] = ret;
    }

    pos++;
}


#if 0

VecData<char> CSerial::getText()
{
    VecData<char> vec;
    char *fraseEx = "Bom dia a todos";

    vec.size = strlen(fraseEx) + 1;
    vec.data = (char *) mAlloc(vec.size);
    strcpy(vec.data, fraseEx);

    return vec;
}

#else

/*
    This function communicates with the Host computer, and
    gets the new block of text to be sintesized.
    It returns the text infoirmation (data and size).
*/
VecData<char> CSerial::getText()
{
    VecData<char> ret;

    // Initializing control values.
    text.data = 0;
    text.size = 0;
    pos = -1;

    // Enebling the reception of another setence.
    asm("bit set flags CTS_FLAG;");
```

```
    // Waiting for the vector size info.
    while (!text.size) asm("idle;");

    // Disabling the serial transmition, so, the host will
    // send just this actual sentence.
    asm("bit clr flags CTS_FLAG;");

    // Allocating space for the result.
    ret.data = (char *) mAlloc(text.size+1);

    // text.data var will point to the output memory location.
    text.data = ret.data;
    ret.size = text.size;

    // Wait until the whole sentence has been aqquired.
    while(pos < ret.size) asm("idle;");

    // Placing the C's end string marck.
    text.data[ret.size] = 0;

    return ret;
}

#endif
```

B.2.12 csound.cpp

```
//cinterfaces.cpp

#include <list>

#include <def21160.h>
#include <21160.h>
#include <signal.h>

#include "csound.h"
#include "defines.h"
#include "structs.h"

list<VecData <float> > CSound::vecList;


/*
    Class contructor. This class initializes the DAC
    comunication with the DSP. It also enables the
    interrupts needed.
*/
CSound::CSound()
{
    // Reset the codec.
    codecReset();
    // Clear and Reset SPORT0 and DMAs.
    clrSPT0Regs();
    // Initialize SPORT0 for codec communications.
    progSPORT0Regs();
    // Start Serial Port 0 tx and rx DMA Transfers.
    progDMACtrl();
    // Initialize & program AD1881.
    codecInit();
    // start audio processing, enable SPORT0 tx int.
    interruptf(SIG_SPT0I, CSound::sendSamples);
}


/*
    This is the function that treats the send data
    interrupt of the seria por. As long as the interrupt
    is generates at 48kHz, we use a variable to perform
    a simply resample algorithm, in order to achieve
    the frequency of 8kHz that we need.
    When the buffer reaches it's end, the function
    automatically releases the allocated mnemory and
    starts sending the next vector in the list (if any).
*/
void CSound::sendSamples(int x)
{
    static VecData<float> outData = {0,0};
    static int dataNotReady = 0;
    static int pos = 0;

    if(!dataNotReady)
    {
        if(pos < outData.size)
        {
            tx_buf[TAG_PHASE] = 0x9800;
            tx_buf[LEFT] = tx_buf[RIGHT] = (int) (OUT_GAIN*outData.data[pos++]);
        }
        else
        {
            if (outData.data)
            {
                free(outData.data);
                vecList.pop_front();
            }
```

```
            if (!vecList.empty())
            {
                outData = vecList.front();
                pos = 0;
            }
            else
            {
                outData.data = 0;
                pos = outData.size = 0;
            }

            tx_buf[TAG_PHASE] = 0x9800;
            tx_buf[LEFT] = tx_buf[RIGHT] = 0x0;
        }
    }

    dataNotReady = circindex(dataNotReady, 1, FREQ_RATIO);
}
```

B.2.13 csound_ext.cpp

```
//cinterfaces.cpp

#include <def21160.h>
#include <21160.h>
#include <signal.h>

#include "csound.h"
#include "defines.h"


volatile int CSound::tx_buf[16] = {
    ENABLE_VFbit_SLOT1_SLOT2,   // set valid bits for slot 0, 1, and 2
    SERIAL_CONFIGURATION,       // serial configuration register address
    0xFF80,                     // initially set to 16-bit slot mode for ADI SPORT compatibility
    0x0000,                     // stuff other slots with zeros for now
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000,
    0x0000
};

volatile int CSound::rx_buf[5];             // receive buffer

int CSound::rcv_tcb[8]  = { 0, 0, 0, 0, 0, 5, 1, 0 };      // receive tcb
int CSound::xmit_tcb[8] = { 0, 0, 0, 0, 0, 16, 1, 0 };     // transmit tcb

int CSound::codecInitRegisters[34] = {
    MASTER_VOLUME,          0x0000,             // Master Volume set for no attenuation
    MASTER_VOLUME_MONO,     0x8000,             // Master Mono volume is muted
    PC_BEEP_Volume,         0x8000,             // PC volume is muted
    PHONE_Volume,           0x8008,             // Phone Volume is muted
    MIC_Volume,             0x8048,             // 20 dB gain stage on, MIC Input analog loopback is muted
    LINE_IN_Volume,         0x8808,             // Line Input analog loopback is muted
    CD_Volume,              0x8808,             // CD Volume is muted
    VIDEO_Volume,           0x8808,             // Video Volume is muted
    AUX_Volume,             0x8808,             // AUX Volume is muted
    PCM_OUT_Volume,         0x0000,             // PCM out from DACs is 0 db gain for both channels
    RECORD_SELECT,          Select_LINE_INPUT,  // Record Select on Line Inputs for L/R channels
    RECORD_GAIN,            Line_Level_Volume,  // Record Gain set for 0 dB on both L/R channels
    GENERAL_PURPOSE,        0x0000,             // 0x8000, goes through 3D circuitry
    THREE_D_CONTROL_REG,    0x0000,             // no phat stereo
    MISC_CONTROL_BITS,      0x0000,             // use SR0 for both Left and Right ADCs and DACs, repeat sample
    SAMPLE_RATE_GENERATE_0, Sample_Rate2,       // user selectable sample rate
    SAMPLE_RATE_GENERATE_1, Sample_Rate2
};      // Sample Rate Generator 1 not used in this example


void CSound::codecInit()
{
    short rxSlot;
    int codecReady = 0;
    int currentSlot;
    int i;
    short codecRegisterAddress;
    short codecRegisterData;

    asm("#include <def21160.h>");

    interrupts(SIG_SPT0I,   rxIRQ);

    // enable SPORT0 multichannel operation
    *(int *) SRCTL0 |= 0x800000;        // set the sport0 multichannel ena bit
```

```
    // get status bit 15 from AD1881 Tag - SLOT 0
    do
    {
        rxSlot = rx_buf[0];
        if ((rx_buf[0] & 0x8000) != 0) {codecReady = 1;}
    } while (!codecReady);

    asm("idle;");                   // wait for a couple of TDM audio frames to pass
    asm("idle;");

    currentSlot = 0;

    currentSlot += TAG_PHASE;
    tx_buf[currentSlot] = ENABLE_VFbit_SLOT1_SLOT2;

    for (i=0; i < 17; i++)
    {
        // fetch next codec register address
        codecRegisterAddress = codecInitRegisters[2*i];
        currentSlot = 0;
        currentSlot += COMMAND_ADDRESS_SLOT;
        // put fetched codec register address into transmit SLOT 1
        tx_buf[currentSlot] = codecRegisterAddress;

        // fetch codec register data
        codecRegisterData = codecInitRegisters[2*i+1];
        // put fetched codec register data into transmit SLOT 2
        currentSlot -= COMMAND_ADDRESS_SLOT;
        currentSlot += COMMAND_DATA_SLOT;
        tx_buf[currentSlot] = codecRegisterData;

        asm("idle;");   // wait until TDM frame is transmitted
    }

    interruptf( SIG_SPT0I,  SIG_IGN);
    asm("nop;");
}


void CSound::clrSPT0Regs()
{
    asm("#include <def21160.h>");
    asm("IRPTL = 0x00000000;");     // clear pending interrupts

    interruptf(SIG_SPT0I,   SIG_IGN);


    *(int *) SRCTL0 = 0;        // sport0 receive control register
    *(int *) RDIV0 = 0;         // sport0 receive frame sync divide register
    *(int *) STCTL0 = 0;        // sport0 transmit control register
    *(int *) MRCS0 = 0;         // sport0 receive multichannel word enable register
    *(int *) MTCS0 = 0;         // sport0 transmit multichannel word enable register
    *(int *) MRCCS0 = 0;        // sport0 receive multichannel companding enable register
    *(int *) MTCCS0 = 0;        // sport0 transmit multichannel companding enable register

    // reset SPORT1 DMA parameters back to the Reset Default State
    *(int *) II0 = 0x1FFFF;
    *(int *) IM0 = 0x0001;
    *(int *) C0 = 0xFFFF;
    *(int *) CP0 = 0x00000;
    *(int *) GP0 = 0x1FFFF;
    *(int *) II2 = 0x1FFFF;
    *(int *) IM2 = 0x0001;

    *(int *) C2 = 0xFFFF;
    *(int *) CP2 = 0x00000;
    *(int *) GP2 = 0x1FFFF;
}


void CSound::progSPORT0Regs()
{
    // sport0 receive control register
    *(int *) SRCTL0 = 0x1F0C40F0;   // 16 chans, int rfs, ext rclk, slen = 15, sden & schen enabled

    // sport0 receive frame sync divide register
    *(int *) RDIV0 = 0x00FF0000;            // SCKfrq(12.288M)/RFSfrq(48.0K)-1 = 0x00FF

    // sport0 transmit control register
    *(int *) STCTL0 = 0x001C00F0;           // 1 cyc mfd, data depend, slen = 15, sden & schen enabled

    // sport1 receive and transmit multichannel word enable registers
    *(int *) MRCS0 = 0x001F001F;            // enable receive channels 0-4
    *(int *) MTCS0 = 0xFFFFFFFF;            // enable transmit channels 0-6

    // sport1 transmit and receive multichannel companding enable registers
    *(int *) MRCCS0 = 0x00000000;           // no companding on receive
    *(int *) MTCCS0 = 0x00000000;           // no companding on transmit
}


void CSound::progDMACtrl()
{
    int txBlockChainPtr;
    int rxBlockChainPtr;
    int currentSlot = 0;

    currentSlot += 7;
```

```
    // sport1 dma control tx chain pointer register
    xmit_tcb[currentSlot] = (int)tx_buf;    // internal dma address used for chaining

    currentSlot -= 1;
    xmit_tcb[currentSlot] = 1;              // DMA internal memory DMA modifier

    currentSlot -= 1;
    xmit_tcb[currentSlot] = 16;             // DMA internal memory buffer count

    currentSlot = 0;
    currentSlot += 7;
    txBlockChainPtr = (int) (xmit_tcb + 7); // get DMA chaining internal mem pointer containing tx_buf address

    txBlockChainPtr &= 0x0003FFFF;          // mask the pointer
    txBlockChainPtr |= 0x40000;             // set the pci bit

    currentSlot -= 3;
    xmit_tcb[currentSlot] = txBlockChainPtr;// write DMA transmit block chain pointer to TCB buffer
    *(int *) CP2 = txBlockChainPtr;         // receive block chain pointer, initiate tx DMA transfers

    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    // - Note: Tshift0 & TX0 will be automatically loaded with the first 2 values in the   -
    // - Tx buffer. The Tx buffer pointer ( II3 ) will increment by 2x the modify value    -
    // - ( IM3 ).                                                                          -
    // - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    currentSlot = 0;
    currentSlot += 7;

    // sport1 dma control tx chain pointer register
    rcv_tcb[currentSlot] = (int)rx_buf;     // internal dma address used for chaining

    currentSlot -= 1;
    rcv_tcb[currentSlot] = 1;               // DMA internal memory DMA modifier

    currentSlot -= 1;
    rcv_tcb[currentSlot] = 5;               // DMA internal memory buffer count

    currentSlot = 0;
    currentSlot += 7;
    rxBlockChainPtr = (int)(rcv_tcb + 7);   // get DMA chaining internal mem pointer containing rx_buf address

    rxBlockChainPtr &= 0x0003FFFF;          // mask the pointer
    rxBlockChainPtr |= 0x40000;             // set the pci bit

    currentSlot -= 3;
    rcv_tcb[currentSlot] = rxBlockChainPtr; // write DMA transmit block chain pointer to TCB buffer
    *(int *) CP0 = rxBlockChainPtr;         // receive block chain pointer, initiate rx DMA transfers
}

void CSound::codecReset()
{
    int i;

    asm("#include <def21160.h>");               // definitions for inline assembly
    asm("bit set MODE2 FLG3O;");                // set flag 3 as an output
    asm("bit clr FLAGS FLG3;");                 // clear codec reset line
    for(i=0; i < (1000); i++) asm("nop;");      // hold in reset for a bit
    asm("bit set FLAGS FLG3;");                 // set codec reset line
}
```

B.2.14 ctextproc.cpp

```
//ctextproc.cpp

#include <list>
#include <cstring>
#include <cstdlib>
#include <cctype>

#include "ctextproc.h"
#include "defines.h"
#include "structs.h"
#include "globals.h"


/*
    Returns two list phone1 has the first phoneme of each diphone
    and phone2 is a list with the second phoneme of each diphone.
    So, for instance, the first diphone is composed by
    phone1(0) + phone2(0).
*/
void CTextProc::getPhonemesList(char *inText, list<PhoneInfo> &phone1, list<PhoneInfo> &phone2)
{
    char diphone[DIPH_VEC_SIZE];
    int stressInfo[DIPH_VEC_SIZE];
    PhoneInfo p1, p2;

    // Converting the syllables to phonemes.
    TextInfo ti = getPhonemes(inText);

    // Converting to C style string.
```

```
    const char *text = ti.text;
    int *info = ti.info;

    while ( *(text + 1) != 0)
    {
        if ( (*(text + 1) == '~') && (*(text + 3) == '~') )
        {
            // Getting a diphone like "i~e~", for instance.
            strncpy(diphone, text, 4);
            memcpy(stressInfo, info, 4);
            diphone[4] = 0;
            text += 2;
            info += 2;
        }
        else
        {
            if (*(text + 2) == '~')
            {
                //Getting a diphone like "ie~", for instance.
                strncpy(diphone, text, 3);
                memcpy(stressInfo, info, 3);
                diphone[3] = 0;
                text++;
                info++;
            }
            else
            {
                //Getting a diphone like "i~e", for instance.
                if (*(text + 1) == '~')
                {
                    strncpy(diphone, text, 3);
                    memcpy(stressInfo, info, 3);
                    diphone[3] = 0;
                    text += 2;
                    info += 2;
                }
                else
                {
                    //Getting a diphone like "ie", for instance.
                    strncpy(diphone, text, 2);
                    memcpy(stressInfo, info, 2);
                    diphone[2] = 0;
                    text++;
                    info++;
                }
            }
        }

        if (diphone[1] == '~')
        {
            strncpy(p1.phoneme, diphone, 2);
            p1.info = stressInfo[0];
            p1.phoneme[2] = 0;

            strcpy(p2.phoneme, &diphone[2]);
            p2.info = stressInfo[2];
        }
        else
        {
            p1.phoneme[0] = diphone[0];
            p1.info = stressInfo[0];
            p1.phoneme[1] = 0;

            strcpy(p2.phoneme, &diphone[1]);
            p2.info = stressInfo[1];
        }

        phone1.push_back(p1);
        phone2.push_back(p2);
    }

    free(ti.text);
    free(ti.info);
}


/*
    This function receives as a parameter the original text
    and returns a string with the syllables converted to
    phonemes.
*/
TextInfo CTextProc::getPhonemes(char *text)
{
    char * out = (char *) marckWords((unsigned char *)text);
    char * out2 = preTranscription(out);
    free(out);
    char * out3 = (char *) marckSyllables((unsigned char *)out2);
    free(out2);
    char * out4 = (char *) marckStressedSyllable((unsigned char *)out3);
    free(out3);
    char * phonemes = phoneticTranscription(out4);
    free(out4);

    TextInfo ret = textDecode(phonemes);
    free(phonemes);

    return ret;
}
```

```
/*
    This method takes the information added to the string and put it
    into the struct TextInfo. The method also removes redundant
    characters.
*/
TextInfo CTextProc::textDecode(char *text)
{
    int i;
    TextInfo ret;
    int textSize = strlen(text);
    int *auxInfo = (int *) mAlloc(textSize);
    bool stressFound = false;

    // This variable makes something lake this "a2a2a" becomes "aa".
    bool jumpNext = false;

    ret.size = 2;

    for (i=0; i<textSize; i++)
    {
        // If if is a char data.
        if ( ((!isdigit(text[i])) || (text[i] == '6')) && (text[i] != SYL_CODE) )
        {
            if (!jumpNext) // If the last char was not the same as the actual.
            {
                auxInfo[i] = (stressFound) ? STRESS_CODE : VALID_TEXT_DATA;
                ret.size++;
            }
            else
            {
                auxInfo[i] = NUL;
                jumpNext = false;
            }
        }
        // The stressFound var will indicate that we found the begining os a stress syllable.
        else if (text[i] == STRESS_CODE)
        {
            stressFound = true;
            auxInfo[i] = NUL;
        }
        else if (text[i] == SYL_CODE)
        {
            stressFound = false;
            auxInfo[i] = NUL;
        }
        // If it is a word code, we verify if we have redundant chars.
        else if (text[i] == WORD_CODE)
        {
            auxInfo[i] = NUL;

            if (text[(i-1)] == text[(i+1)])
            {
                jumpNext = !jumpNext;
            }
        }
        else // Anything else is marked for removal.
        {
            auxInfo[i] = NUL;
        }
    }

    int k = 1;
    ret.info = (int *) mAlloc(ret.size);
    ret.text = (char *) mAlloc(ret.size + 1);

    // Inserting the "no sound" phoneme.
    ret.text[0] = ret.text[ret.size-1] = 'X';
    ret.info[0] = ret.info[ret.size-1] = VALID_TEXT_DATA;

    // Removing the unwanted data.
    for (i=0; i<textSize; i++)
    {
        if (auxInfo[i] != NUL)
        {
            ret.info[k] = auxInfo[i];
            ret.text[k] = text[i];
            k++;
        }
    }

    ret.text[ret.size] = 0;
    free(auxInfo);
    return ret;
}
```

B.2.15 ctextproc_ext.cpp

```
//ctextproc.cpp

#include <cstdlib>

#include "ctextproc.h"
#include "globals.h"
```

```
#include "defines.h"


unsigned char *CTextProc::marckWords(unsigned char *strin)
{
    unsigned char ch, marca=0, palavra[TamMaxPal], *strout, *out;
    int i=0;
    bool iniciarFrase=true, iniciarPalavra=true, chInvalido=false;
    unsigned char cp[256]; //CodePage

    strout=out=(unsigned char *) mAlloc(strlen((const char *)strin)*2+100);
    init1252to850(cp);
    while((ch= conv2LowCaps(cp[*(strin++)]))!= '\0') {
        switch(ch) {
            case '.': case ':': case ';': case '!':
            case '\?':iniciarFrase=true;
            case '-': case '(': case ')' : case '\"': case '\n': case '/':
            case ',': case '≪': case '≫': case 174: case 175: case '\'':
                        iniciarPalavra=true;
                      marca=PON;
                        break;
            case ' ': iniciarPalavra=true;
                        marca=ESP;
                        break;
            default: if (!isVowel(ch) && !isConsonant(ch)) chInvalido=true;
                     if (iniciarFrase) {
                           marca=iFRA;
                           iniciarFrase=iniciarPalavra=false;
                     }else  if (iniciarPalavra) {
                                   marca=iPAL;
                                   iniciarPalavra=false;
                            }else   marca= 0;
        }
        if (iniciarPalavra) {       // caracter fora de qq palavra
            if (i) {                            // palavra não nula
                palavra[i++]='\0';
                if (chInvalido) {
                    *(out++)=NUL;
                    chInvalido=false;
                }
                strcpy((char *)out,(char *)palavra); out+=strlen((char *)palavra);
                i=0;                                    // palavra passa a nula
            }
            if (marca!=ESP && marca) {              // caracter não é um espaço
                *(out++)=marca;
                if (marca==iFRA) *(out++)=iPAL;
                *(out++)=ch;
            }
        }else {                         // caracter dentro duma palavra
                if (marca) *(out++)=marca;  // 1° caracter da palavra
                if (marca==iFRA) *(out++)=iPAL;
                palavra[i++]=ch;
        }
    } // FIM DO WHILE

    if (!iniciarPalavra) {
        palavra[i++]='\0';
        if (chInvalido) {
            *(out++)=NUL;
            chInvalido=false;
        }
        strcpy((char *)out,(char *)palavra); out+=strlen((char *)palavra);
        i=0;                                    // palavra passa a nula
    }
    *out='\0';
    return strout;
}



char *CTextProc::preTranscription(char *strin)
{
    char ch, palavra[TamMaxPal], *strout, *out;
    int i=0;

    strout=out=(char *) mAlloc(strlen(strin)*2+100);

    while((ch=*(strin++))!= '\0') {
        switch(ch) {
            case PON:
            case NUL: case ESP:
            case iFRA: case iPAL:
                        if (i) {   // palavra não nula
                           palavra[i++]='\0';
                           wordProc(palavra);
                           strcpy(out,palavra); out+=strlen(palavra);
                           i=0;        // palavra passa a nula
                           }
                        if (ch==PON){
                             *(out++)=ch;
                             ch=*(strin++);
                        }
                        *(out++)=ch;
                        break;
            default:  palavra[i++]=ch;
        }
    }  // FIM WHILE

    if (i) {    // palavra não nula
```

```
        palavra[i++]='\0';
        wordProc(palavra);
        strcpy(out,palavra); out+=strlen(palavra);
        i=0;        // palavra passa a nula
    }
    *out='\0';
    return strout;
}



unsigned char *CTextProc::marckSyllables(unsigned char *strin)
{
    unsigned char ch, palavra[TamMaxPal], *strout, *out;
    int i=0;

    strout=out=(unsigned char *) mAlloc(strlen((char *)strin)*2+100);

    while((ch=*(strin++))!= '\0') {
        switch(ch) {
    //      case '\\':
            case PON:
            case NUL: case ESP:
            case iFRA: case iPAL:
                        if (i) {   // palavra não nula
                           palavra[i++]='\0';
                           out=splitWord(palavra, out);
                           i=0;        // palavra passa a nula
                           }
                        if (ch==PON){
                               *(out++)=ch;
                               ch=*(strin++);
                        }
                        *(out++)=ch;
//                      if (ch=='\\') do{ch=*(strin++); *(out++)=ch;}while(ch!='\\');
                        break;
            default:  palavra[i++]=ch;
        }
    }  // FIM WHILE

    if (i) {    // palavra não nula
        palavra[i++]='\0';
        out=splitWord(palavra, out);
        i=0;        // palavra passa a nula
    }
    *out='\0';
    return strout;
}


unsigned char *CTextProc::marckStressedSyllable(unsigned char *strin)
{
    unsigned char ch, palavra[TamMaxPal], *strout, *out;
    int i=0;
    unsigned char *ppm, mente[8]=".|mE.te";
    mente[0]=mente[4]=iSIL; mente[1]=sTON;

    strout=out=(unsigned char *) mAlloc(strlen((char *)strin)*2+100);

    while((ch=*(strin++))!= '\0') {
        switch(ch) {
            case PON:
            case NUL: case ESP:
            case iFRA: case iPAL:
                        if (i) {   // palavra não nula
                           palavra[i++]='\0';
                           if(ppm=(unsigned char *)strstr((char *)palavra,(char *) mente+2)){
                               *(ppm-1)='\0';
                               marckSyllable(palavra);
                               strcat((char *)palavra,(char *) mente);
                           } else marckSyllable(palavra);
                           strcpy((char *)out,(char *)palavra); out+=strlen((char *)palavra);
                           i=0;        // palavra passa a nula
                           }
                        if (ch==PON){
                               *(out++)=ch;
                               ch=*(strin++);
                        }
                        *(out++)=ch;
                        break;
            default:  palavra[i++]=ch;
        }
    }
    if (i) {    // palavra não nula
        palavra[i++]='\0';
        if(ppm=(unsigned char *)strstr((char *)palavra, (char *)mente+2)){
            *(ppm-1)='\0';
            marckSyllable(palavra);
            strcat((char *)palavra, (char *)mente);
        } else marckSyllable(palavra);
        strcpy((char *)out,(char *)palavra); out+=strlen((char *)palavra);
        i=0;        // palavra passa a nula
    }
    *out='\0';
    return strout;
}
```

```
char *CTextProc::phoneticTranscription(char *strin)
{
    char ch, palavra[TamMaxPal], *strout, *out;
    int i=0;
    unsigned char cp[256]; //CodePage

    strout=out=(char *) mAlloc(strlen(strin)*2+100);

    while((ch=*(strin++))!= '\0') {
        switch(ch) {
    //      case '\\':
            case PON:
            case NUL: case ESP:
            case iFRA: case iPAL:
                        if (i) {   // palavra não nula
                           palavra[i++]='\0';
                           out=translateWord(palavra, out);
                           out=exceptionTable(out);
                           i=0;        // palavra passa a nula
                           }
                        if (ch==PON){
                             *(out++)=ch;
                             ch=*(strin++);
                        }
                        *(out++)=ch;
                        if (ch=='\\') do{ch=*(strin++); *(out++)=ch;}while(ch!='\\');
                        break;
            default:  palavra[i++]=ch;
        }
    }
   if (i) { // palavra não nula
        palavra[i++]='\0';
        out=translateWord(palavra, out);
        out=exceptionTable(out);
        i=0;        // palavra passa a nula
    }
    *out='\0';

    init850to1252(cp);
    for (out=strout;*out!='\0';out++) *out=(char)cp[(unsigned char)*out];
    return strout;
}


void    CTextProc::init1252to850(unsigned char * tab)
{
  int i;
  for(i=0;i<256;i++) tab[i]= (unsigned char) i;
  tab[199]=128; tab[233]=130; tab[226]=131; tab[224]=133; tab[231]=135; tab[234]=136;
  tab[232]=138; tab[238]=140; tab[201]=144; tab[244]=147; tab[225]=160; tab[237]=161;
  tab[243]=162; tab[250]=163; tab[171]=174; tab[187]=175; tab[193]=181; tab[194]=182;
  tab[192]=183; tab[227]=198; tab[195]=199; tab[202]=210; tab[205]=214; tab[206]=215;
  tab[211]=224; tab[212]=226; tab[210]=227; tab[245]=228; tab[213]=229; tab[218]=233;
  tab[171]=174; tab[187]=175;
}


unsigned char CTextProc::conv2LowCaps(unsigned char ch)
{
    switch(ch) {
            case 128: return (unsigned char) 135;
            case 144: return (unsigned char) 130;
            case 181: return (unsigned char) 160;
            case 182: return (unsigned char) 131;
            case 183: return (unsigned char) 133;
            case 199: return (unsigned char) 198;
            case 210: return (unsigned char) 136;
            case 214: return (unsigned char) 161;
            case 215: return (unsigned char) 140;
            case 224: return (unsigned char) 162;
            case 226: return (unsigned char) 147;
            case 229: return (unsigned char) 228;
            case 233: return (unsigned char) 163;
            default:  if (ch>='A' && ch<='Z') return (unsigned char) (ch+32);
                        else return ch;
    }
}


void CTextProc::wordProc(char *pal)
{
    char *pp, *pc;

    //substitui parte da palavra (x->ss)
    if (pc=strstr(pal, "proxi")) *(pc+3)='#'; //ss                              "proxi"  -> "prosi"
    else if (pc=strstr(pal, "pr\xa2xi")) *(pc+3)='#'; //ss                      "próxi"
         else if (pc=strstr(pal, "m\xa0xim")) *(pc+2)='#'; //ss                 "máxim"
              else if (pc=strstr(pal, "auxili")) *(pc+2)='#'; //ss             "auxili"
                   else if (pc=strstr(pal, "aux\xa1li")) *(pc+2)='#'; //ss     "auxíli"
                        else if (pc=strstr(pal, "troux")) *(pc+4)='#'; //ss    "troux"

    // pré-transcrição do 'o'
                         /* palavras começadas por ....  */
    if (pc=strstr(pal, "hemo")) {*(pc+1)=(char)130; *(pc+3)=(char)162;} // hémó
    else if (pc=strstr(pal, "homeo")) {*(pc+1)=(char)162; *(pc+3)='i'; *(pc+4)=(char)162;} // hómió
    else if (pc=strstr(pal, "iso")) {*(pc+2)=(char)162;} // isó
    else if (pc=strstr(pal, "lito")) {*(pc+3)=(char)162;} // litó
    else if (pc=strstr(pal, "meso")) {*(pc+3)=(char)162;} // mesó
    else if (pc=strstr(pal, "neo")) {*(pc+1)=(char)130; *(pc+2)=(char)162;} // néó
```

```
else if (pc=strstr(pal, "octo")) {*(pc+3)=(char)162;} // octó
else if (pc=strstr(pal, "oftalmo")) {*(pc)=(char)162; *(pc+6)='u';} // óftalmu
else if (pc=strstr(pal, "onoma")) {*(pc)=(char)162; *(pc+2)=(char)147;} // ónôma
else if (pc=strstr(pal, "oxi")) {*(pc)=(char)162;} // óxi
else if (pc=strstr(pal, "orto")) {*(pc+3)=(char)162;} // ortó
else if (pc=strstr(pal, "quiro")) {*(pc+4)=(char)162;} // quiró
else if (pc=strstr(pal, "rino")) {*(pc+3)=(char)162;} // rinó
else if (pc=strstr(pal, "rizo")) {*(pc+3)=(char)162;} // rizó
else if (pc=strstr(pal, "xeno")) {*(pc+3)=(char)162;} // xenó
else if (pc=strstr(pal, "xilo")) {*(pc+3)=(char)162;} // xiló
else if (pc=strstr(pal, "auto")) {*(pc+3)=(char)162;} // autó
else if (pc=strstr(pal, "aero")) {*(pc+1)=(char)130; *(pc+3)=(char)162;} // aéró
else if (pc=strstr(pal, "agro")) {*(pc+3)=(char)162;} // agró
else if (pc=strstr(pal, "astro")) {*(pc+4)=(char)162;} // astró
else if (pc=strstr(pal, "bio")) {*(pc+2)=(char)162;} // bió
else if (pc=strstr(pal, "electro")) {*(pc+6)=(char)162;} // electró
else if (pc=strstr(pal, "fono")) {*(pc+1)=(char)162; *(pc+3)=(char)162;} // fónó
else if (pc=strstr(pal, "foto")) {*(pc+1)=(char)162; *(pc+3)=(char)162;} // fótó
else if (pc=strstr(pal, "hetero")) {*(pc+1)=(char)130; *(pc+5)=(char)162;} // héteró
else if (pc=strstr(pal, "horos")) {*(pc+1)=(char)162; *(pc+3)=(char)162;} // hórós
else if (pc=strstr(pal, "lipo")) {*(pc+3)=(char)162;} // lipó
else if (pc=strstr(pal, "megalo")) {*(pc+1)=(char)130; *(pc+5)=(char)162;} // mégaló
else if (pc=strstr(pal, "micro")) {*(pc+4)=(char)162;} // micró
else if (pc=strstr(pal, "moto")) {*(pc+1)=(char)162;} // móto
else if (pc=strstr(pal, "poli")) {*(pc+1)=(char)162;} // póli
else if (pc=strstr(pal, "pseudo")) {*(pc+5)=(char)162;} // pseudó
else if (pc=strstr(pal, "proto")) {*(pc+2)=(char)162; *(pc+4)=(char)162;} // prótó
else if (pc=strstr(pal, "retro")) {*(pc+1)=(char)130; *(pc+4)=(char)162;} // rétró
else if (pc=strstr(pal, "termo")) {*(pc+4)=(char)162;} // termó

// pré-transcrição do 'e'
                   /* palavras começadas por ....  */
if (pc=strstr(pal, "hiper")) {*(pc+3)=(char)130;} // hipér
else if (pc=strstr(pal, "ferro")) {*(pc+1)=(char)130;} // férro
else if (pc=strstr(pal, "enea")) {*(pc)=(char)130;} // énea
else if (pc=strstr(pal, "etno")) {*(pc)=(char)130;} // étno
else if (pc=strstr(pal, "helio")) {*(pc+1)=(char)130;} // hélio
else if (pc=strstr(pal, "hemato")) {*(pc+1)=(char)130;} // hémato
else if (pc=strstr(pal, "hemi")) {*(pc+1)=(char)130;} // hémi
else if (pc=strstr(pal, "hexa")) {*(pc+1)=(char)130;} // hégza
else if (pc=strstr(pal, "mega")) {*(pc+1)=(char)130;} // méga
else if (pc=strstr(pal, "tetra")) {*(pc+1)=(char)130;} // tétra
else if (pc=strstr(pal, "hepa")) {*(pc+1)=(char)130;} // hépa
else if (pc=strstr(pal, "herma")) {*(pc+1)=(char)130;} // hérma
else if (pc=strstr(pal, "herb")) {*(pc+1)=(char)130;} // hérb
else if (pc=strstr(pal, "super")) {*(pc+3)=(char)130;} // supér
else if (pc=strstr(pal, "pre")) {*(pc+2)=(char)130;} // pré


//remoção do h
if (*pal=='h') for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
if (*(pal+strlen(pal)-1)=='h') *(pal+strlen(pal)-1)='\0';

if (*pal=='r') *pal='R';

//  if (pp=strstr(pal, "muit")) *(pp+2)='I';
if (*pal=='q'){
        *pal='k';
        if (*(pal+2)=='e' || *(pal+2)=='i'|| *(pal+2)==(char) 161|| *(pal+2)==(char) 136|| *(pal+2)==(char) 130)
        for(pp=pal+1;*pp!='\0';pp++) *pp=*(pp+1);
} else
if (*pal=='g')
    if (*(pal+1)=='e' || *(pal+1)=='i'|| *(pal+1)==(char) 161|| *(pal+1)==(char) 136|| *(pal+1)==(char) 130)
        *pal='Z';
    else
        if (*(pal+1)=='u'){
            if (*(pal+2)=='e' || *(pal+2)=='i'|| *(pal+2)==(char) 161|| *(pal+2)==(char) 136|| *(pal+2)==(char) 130)
                for(pp=pal+1;*pp!='\0';pp++) *pp=*(pp+1);
        }

while(*(++pal)!='\0') {
    switch(*pal){
    case '#': *pal='s';
              break;
    case 'h': if (*(pal-1)=='l') {
                    *(pal-1)='L';
                    for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                    pal--;
                } else
                      if (*(pal-1)=='n') {
                           *(pal-1)='J';
                           for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                           pal--;
                       } else
                               if (*(pal-1)=='c') {
                                   *(pal-1)='S';
                                   for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                   pal--;
                               }
                break;
    case 'r': if (*(pal-1)=='r') {
                           *(pal-1)='R';
                           for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                       pal--;
                       }else
                           if (*(pal-1)=='l' || *(pal-1)=='s' || *(pal-1)=='A' || *(pal-1)=='E' || *(pal-1)=='I' ||
                                 *(pal-1)=='O' || *(pal-1)=='U' || *(pal-1)==(char)181 ||
                                   *(pal-1)==(char)182 || *(pal-1)==(char)210 || *(pal-1)==(char)144)
                               *(pal)='R';
```

```
                    break;
        case 's': if ((isVowel(*(pal-1)) || *(pal-1)==(char) 132 || *(pal-1)==(char) 134 || *(pal-1)==(char) 148)
                        && phoneticVowel(*(pal+1)))
                    *pal='z';
                else
                    if (*(pal-1)=='s'){
                       for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                       pal--;
                    }
                break;
        case 'q': *pal='k';
                if (*(pal+2)=='e' || *(pal+2)=='i'|| *(pal+2)==(char) 161|| *(pal+2)==(char) 136|| *(pal+2)==(char) 130)
                   for(pp=pal+1;*pp!='\0';pp++) *pp=*(pp+1);
                break;
        case 'g':
                if (*(pal+1)=='e' || *(pal+1)=='i'|| *(pal+1)==(char) 161|| *(pal+1)==(char) 136|| *(pal+1)==(char) 130)
                   *pal='Z';
                else
                   if (*(pal+1)=='u'){
                       if (*(pal+2)=='e' || *(pal+2)=='i'|| *(pal+2)==(char) 161|| *(pal+2)==(char) 136|| *(pal+2)==(char) 130)
                           for(pp=pal+1;*pp!='\0';pp++) *pp=*(pp+1);
                   }
                break;

        case (char) 198:   //  ã
                if (*(pal+1)=='o') *pal= (char) 134;  //  (ão)
                else if (*(pal+1)=='e' || *(pal+1)=='i') *pal=(char)132; // ä (ãe)
                       else break;
                   for(pp=pal+1;*pp!='\0';pp++) *pp=*(pp+1);
                break;
        case (char) 228:   //  õ
                *pal=(char)148; // ö (õe)
                for(pp=pal+1;*pp!='\0';pp++) *pp=*(pp+1);
                break;
        case (char) 136:                    // vêm e têm
                if (*(pal+1)=='m' && (*(pal-1)=='v' || *(pal-1)=='t')) {
                   *pal=(char)132;
                   *(pal+1)='E';
                }
                break;
        case 'm': if (*(pal+1)!='b' && *(pal+1)!='p' && *(pal+1)!='\0') break;
        case 'n': if (*(pal+1)!='h' && (*(pal+1)=='\0' || isConsonant(*(pal+1))))
                    switch(*(pal-1)) {
                       case (char) 160:              *(pal-1)=(char)181; // Á
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case (char) 131:              *(pal-1)=(char)182; // Â
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case 'a':         *(pal-1)='A';
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case (char) 136:              *(pal-1)=(char)210;  // Ê
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case (char) 130:              *(pal-1)=(char)144; // É
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case 'e':         if(*(pal+1)=='\0' && *(pal)=='n'){*(pal-1)='e'; break;};
                                        *(pal-1)='E';
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case (char) 140:
                       case 'i': case (char) 161: *(pal-1)='I';
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case (char) 147:
                       case 'o': case (char) 162: *(pal-1)='Õ';
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       case (char) 150:
                       case 'u': case (char) 163: *(pal-1)='U';
                                        for(pp=pal;*pp!='\0';pp++) *pp=*(pp+1);
                                            pal--;
                                        break;
                       }
                    break;
        }
    }
}



unsigned char *CTextProc::splitWord(unsigned char *pal, unsigned char *out)
{
    unsigned char *p=pal;
    if (*p=='\0') return out;
    *(out++)=*p;
    if (!phoneticVowel(*p)) {
        p++;
        if (*p=='\0') {*(out++)=NUL; return out;};      // palavra inválida
```

```
        *(out++)=*p;
        if (!phoneticVowel(*p)){
            p++;
            if (*p=='\0') {*(out++)=NUL; return out;};      // palavra inválida
            *(out++)=*p;
        }
    }
    for(;;){
        p++;
        if (*p=='\0') return out;
        if (phoneticVowel(*p))
            if ((*p=='o' && *(p-1)=='a') || ((*p=='i' || *p=='u' ) &&
                  *p!=*(p-1) && *(p+1)!='u' && ((*(p+1)!='l' && *(p+1)!='r') || phoneticVowel(*(p+2)))))
                *(out++)=*p;
            else {
                *(out++)=iSIL;
                *(out++)=*p;
            }
        else {
            p++;
            if (*p=='\0') {
                *(out++)=*(p-1);
                return out;
            }
            if (phoneticVowel(*p)) {
                *(out++)=iSIL;
                *(out++)=*(p-1);
                *(out++)=*p;
            }else{
                if (*(p+1)=='\0') {             // palavra inválida
                    *(out++)=*(p-1);
                    *(out++)=*p;
                    {*(out++)=NUL; return out;};
                }
                if (phoneticVowel(*(p+1))) {
                    if (sequentialConsonants(*(p-1), *p)) {
                        *(out++)=iSIL;
                        *(out++)=*(p-1);
                    }else {
                        *(out++)=*(p-1);
                        *(out++)=iSIL;
                    }
                }else{
                    if (*(p+2)=='\0') {             // palavra inválida
                        *(out++)=*(p-1);
                        *(out++)=*p;
                        *(out++)=*(p+1);
                        {*(out++)=NUL; return out;};
                    }
                    if (!phoneticVowel(*(p+2))) {
                        *(out++)=*(p-1);
                        *(out++)=*p;
                        *(out++)=iSIL;
                        p++;
                        *(out++)=*p;
                        p++;
                    }else
                        if (sequentialConsonants(*(p-1), *p) || *p=='s') {
                            *(out++)=*(p-1);
                            *(out++)=*p;
                            *(out++)=iSIL;
                            p++;
                        }else{
                            *(out++)=*(p-1);
                            *(out++)=iSIL;
                            *(out++)=*p;
                            p++;
                        }
                }
                *(out++)=*p;
                p++;
                *(out++)=*p;
            }
        }
    }
//  return out;
}


 void CTextProc::marckSyllable(unsigned char *pal)
 {
    static unsigned char silabas[NumMaxSil][TamMaxSil+1];
    int ns=0, i=0, s, sTon=-1;
    unsigned char *p;
    for(p=pal; *p!='\0'; p++)
        if (*p==iSIL) {
            silabas[ns][i]='\0';
            ns++;
            i=0;
        }else{
            silabas[ns][i++]=*p;
            if (sTon==-1) if (accentedLetter(*p)) sTon=ns;
        }
    silabas[ns][i]='\0';
    ns++;
   if (ns<2) sTon=-1;
    else {
        for(s=0; s<ns && sTon==-1; s++)
            for(i=0; silabas[s][i]!='\0'; i++)
```

```
                if (silabas[s][i]==132 || silabas[s][i]==134 || silabas[s][i]==148 || silabas[s][i]==198) sTon=s;
        if (sTon==-1) if (accentedTermination(silabas[ns-1])) sTon=ns-1;
        if (sTon==-1) sTon=ns-2;
    }
    i=0;
    for(s=0; s<ns ; s++) {
        if (s) pal[i++]=iSIL;
        if (sTon==s) pal[i++]=sTON;
        strcpy((char *)pal+i,(char *)silabas[s]); i+=strlen((char *)silabas[s]);
    }
}


char *CTextProc::translateWord(char *pal, char *out)
{
    int ns=1;
    char *p;
    bool tonica = false;

    for(p=pal; *p!='\0'; p++)
    switch(*p) {
    case iSIL:ns++;
            tonica=false;
                        OUTex(*p);
                        break;
    case sTON:tonica=true;
                        OUTex(*p);
                        break;
    case 'i': if (p>pal && phoneticVowel(*(p-1))) OUTex('j'); // semivogal
                        else OUTex(*p);
                        break;
    case 'u': if (p>pal && phoneticVowel(*(p-1))) OUTex('w');
                        else OUTex(*p);
                        break;
    case (char) 130:
                        OUTs("E");  // é
                        break;
    case (char) 136:
                        OUTs("e");  // ê
                        break;
    case (char) 147:
                        OUTs("o");  // ô
                        break;
    case (char) 228:
                        OUTs("o");  // õ
                        break;
    case (char) 162:
                        OUTs("O");  // ó
                        break;
    case (char) 161:
                        OUTs("i");  // í
                        break;
    case (char) 131:
                        OUTs("6");  // â
                        break;
    case (char) 198:
                        OUTs("6");  // ã
                        break;
    case (char) 133:
                        OUTs("a");  // à
                        break;
    case (char) 160:
                        OUTs("a");  // á
                        break;
    case (char) 163:
                        OUTs("u");  // ú
                        break;
    case 'j': OUTex('Z');
                        break;
    case 'c': if (*(p+1)=='e' || *(p+1)=='i'|| *(p+1)==(char) 161|| *(p+1)==(char) 136|| *(p+1)==(char) 130 ||
                                 *(p+1)=='E' || *(p+1)==(char) 144 || *(p+1)==(char) 210 || *(p+1)=='I')
                                OUTex('s');
                        else OUTex('k');
                        break;
    case (char) 135:
                        OUTex('s');     // ç
                        break;
    case (char) 134:
                        OUTs("6~w~");       // ão
                        break;
    case (char) 132:
                        OUTs("6~j~");       // ãe
                        break;
    case (char) 148:
                        OUTs("o~j~");       // õe
                        break;
    case (char) 181: // Á
    case (char) 182: // Â
    case 'A':
                        if (*(p+1)=='\0')
                                OUTs("6w~");
                        else OUTs("6~");
                        break;
    case (char) 210: // Ê  -> êm
    case (char) 144: // É  -> ém
    case 'E':
                        if (*(p+1)=='\0')
                                OUTs("6~j");
                        else OUTs("e~");
```

```
                    break;
case 'I': OUTs("i~");
                    break;
case 'O': OUTs("o~");
                    break;
case 'U': OUTs("u~");
                    break;
case 'z':
case 's': if (*(p+1)=='\0')
                        OUTex('S');
                    else    if (*(p+1)==iSIL)
                                if ((*(p+2)== 'p'||*(p+2)== 't'||*(p+2)== 'c'||*(p+2)== 'k'||*(p+2)== 'f'||
                                    *(p+2)== (char)135||*(p+2)== 's'||*(p+2)== 'S')    // p, t, k, f, s, S
                                     || (*(p+2)==sTON && (*(p+3)=='p'||*(p+3)=='t'||*(p+3)=='c'||*(p+3)=='k'||
                                     *(p+3)=='f'||*(p+3)==(char) 135||*(p+3)=='s'||*(p+3)== 'S')))
                                  OUTex('S');
                                else OUTex('Z');
                            else OUTex(*p);
                    break;

case 'x': if (p==pal || (p==pal+1 && tonica))  OUTex('S');      // x em início de palavra
                    else
                        if (*(p+1)=='\0')  OUTs("ks");                  // x em fim de palavra
                        else                                                    // x no meio de palavra
                            if (*(p+1)==iSIL) OUTex('S');
                            else
                                if ((*(p-1)==iSIL && phoneticVowel(*(p-3)) && (*(p-2)=='i' || *(p-2)=='u')) ||  //precedido de ditongo
                                      (*(p-1)==sTON && *(p-2)==iSIL && phoneticVowel(*(p-4)) && (*(p-3)=='i' || *(p-3)=='u')))
                                      OUTex('S');
                                else
                                    if ((*(p-1)==iSIL && (*(p-2)=='a' || *(p-2)==(char) 160)) ||//precedido de a
                                            (*(p-1)==sTON && *(p-2)==iSIL && (*(p-3)=='a' || *(p-3)==(char) 160)))
                                    { if (*(p+1)=='a' || *(p+1)==(char) 160 || *(p+1)=='A')   // e seguido de a
                                            OUTex('S');
                                        else
                                            if (*(p+1)=='e' || *(p+1)=='E') OUTex('S'); // precedido de 'a' e seguido de 'e'
                                            else OUTs("ks"); // apenas precedido de a
                                    }else
                                        if ((*(p-1)==iSIL && *(p-2)==(char) 162) ||   // precedido de ó
                                              (*(p-1)==sTON && *(p-2)==iSIL && *(p-3)==(char) 162))
                                            OUTs("ks");
                                    else
                                            // precedido de o a iniciar palavra
                                            if ((*(p-1)==iSIL && *(p-2)=='o' && (p-2==pal|| (*(p-2)==sTON && p-3==pal) )) ||
                                                (*(p-1)==sTON && *(p-2)==iSIL && *(p-3)=='o' && p-3==pal))
                                            OUTs("ks");
                                      else
                                           if ((*(p-1)==iSIL && *(p-2)=='o' && *(p+1)=='i') ||   // precedido de o e seguido de i
                                           (*(p-1)==sTON && *(p-2)==iSIL && *(p-3)=='o' && *(p+1)=='i'))
                                                   OUTs("ks");
                                        else
                                             if ( ( *(p-1)==iSIL && (*(p-2)=='i' || *(p-2)=='u') &&
                                             (*(p-3)=='f' || (*(p-3)=='l' && *(p-4)=='f')) ) ||  // precedido por fi fu fli flu
                                            ( *(p-1)==sTON && *(p-2)==iSIL && (*(p-3)=='i' || *(p-3)=='u' ) &&
                                             (*(p-4)=='f' || (*(p-4)=='l' && *(p-5)=='f')) ) )
                                                    OUTs("ks");
                                             else
                                                 if ((*(p-1)==iSIL && *(p-2)=='r') ||   // precedido de r
                                                     (*(p-1)==sTON && *(p-2)==iSIL && *(p-3)=='r'))
                                                       OUTs("ks");
                                                 else OUTex('S');   // restantes casos...
                    break;

case 'a': if ((*(p+1)=='l' || *(p+1)=='x') && (*(p+2)==iSIL || *(p+2)=='\0'))  OUTex('a');  // al. al ax. ax
                    else
                        if (*(p+1)=='z' && (*(p+2)==iSIL || *(p+2)=='\0'))  OUTex('a');  // az. az
                        else
                            if (*(p+1)=='r' && *(p+2)=='\0')  OUTex('a');  // ..ar
                            else
                                if (*(p+1)==iSIL && (*(p+2)=='m' || *(p+2)=='n' || *(p+2)=='J'))  OUTex('6');  // am an anh
                                else
                                    if (tonica) OUTex('a');  // em sílaba tónica
                                    else
                                        if (*(p+1)=='i' || *(p+1)=='o' || *(p+1)=='u')  OUTex('a');  // aj aw
                                        else
                                            if (*(p+1)=='c' && *(p+2)==iSIL &&  ((*(p+3)==(char) 135 || *(p+3)=='c') || (*(p+3)==sTON &&
                                                                    (*(p+4)==(char) 135 || *(p+4)=='c'))))  OUTex('a');  // acc  acç
                                            else
                                                if (*(p+1)==iSIL &&  (*(p+2)=='x' || (*(p+2)==sTON && *(p+3)=='x')))  OUTex('a');  // ax
                                                else OUTex('6');
                    break;

case 'e': switch (*(p+1)){
                        case '\0':if (p==pal || (p==pal+1 && tonica)) OUTex('i');       // e
                                        else OUTex('@');                // ...e
                                        break;
                        case 'i': OUTs("Ej"); p++; break;       // ..ei..
                        case 'u': OUTs("ew"); p++; break;       // ..eu..
                        case 'l': OUTs("El"); p++; break;       // ..el..
                        case 'c': if (*(p+2)==iSIL) {OUTs("Ek"); p++; break;}   // ..ec.?
                        case 'p': if (*(p+2)==iSIL) {OUTs("Ep"); p++; break;}   // ..ep.?
                        case 'g': if (*(p+2)==iSIL) {OUTs("Eg"); p++; break;}   // ..eg.?
                        case 's': if (*(p+2)=='\0') {OUTs("@S"); p++; break;}   // ..es
                        case 'r': if (*(p+2)=='\0') {*(p-1)=='k'?OUTs("Er"):OUTs("er"); p++; break;}    // ..er
                        case 'n': if (*(p+2)=='\0') {OUTs("En"); p++; break;}   // ..en
                        case 'z': if (*(p+2)=='\0') {OUTs("eS"); p++; break;}   // ..ez
                        case 'x': if (*(p+2)=='\0')     // ..ex
                                        {if (p==pal || (p==pal+1 && tonica)) OUTs("6jS");
                                         else OUTs("Eks");
```

```
                                        p++;   break;}
                    default:
        /*0*/       if (*(p+1)==iSIL) //...e.?....
                        {
                            if (*(p+2)=='x')                // ..e.x+vogal...
                            {
                                if (p==pal || (p==pal+1 && tonica)) // em início de palavra
                                    {OUTex('i');    OUTex(iSIL); OUTex('z');}
                                else                                    // no meio de palavra
                                  if ((*(p-1)=='n' && *(p-2)==iSIL && *(p-3)=='i' && p-3==pal) ||    // i.ne.x
                                     (*(p-1)=='n' && *(p-2)==sTON && *(p-3)==iSIL && *(p-4)=='i' && p-4==pal)) // i.|ne.x
                                     {OUTs("@j"); OUTex(iSIL); OUTex('z');}    //@jz
                                  else
                                        if (*(p-1)=='s') {OUTex('E'); OUTex(iSIL); OUTs("ks");}
                                        else
                                            if (tonica) {OUTex('E'); OUTex(iSIL); OUTs("ks");}
                                            else
                                                if (*(p-1)=='l' && isConsonant(*(p-2)))  // ..fle.x..
                                                    {OUTex('E'); OUTex(iSIL); OUTs("ks");}
                                                else {OUTex('@'); OUTex(iSIL); OUTex('S');}
                                p+=2;
                                ns++;
                                tonica=false;
                                break;
                            }

                            if (*(p+2)==sTON && *(p+3)=='x')    // ..e.|x..
                            {
                                if (p==pal || (p==pal+1 && tonica))     // em início de palavra
                                    {OUTex('i');    OUTex(iSIL); OUTex(sTON); OUTex('z');}
                                else                                    // no meio de palavra
                                  if (*(p-1)=='n' && *(p-2)==iSIL && *(p-3)=='i' && p-3==pal)    // i.ne.|x
                                        {OUTs("@j"); OUTex(iSIL); OUTex(sTON); OUTex('z');}    //@jz
                                  else
                                        if (*(p-1)=='s') {OUTex('E'); OUTex(iSIL); OUTex(sTON); OUTs("ks");}
                                        else
                                            if (*(p-1)=='l' && isConsonant(*(p-2)))  // ..fle.|x..
                                                {OUTex('E'); OUTex(iSIL); OUTex(sTON); OUTs("ks");}
                                            else
                                                if (accentedVowel(*(p+4))) {OUTex('E'); OUTex(iSIL); OUTex(sTON); OUTs("ks");}
                                                else {OUTex('@'); OUTex(iSIL); OUTex(sTON); OUTex('S');}
                                p+=3;
                                ns++;
                                tonica=true;
                                break;
                            }


                            if (*(p+2)=='e' || *(p+2)=='E' || (*(p+2)==sTON && (*(p+3)=='e' || *(p+3)=='E'))) {
                                OUTex('@');     // ..e.e..
                                break;
                            }
                            if (phoneticVowel(*(p+2)) || (*(p+2)==sTON && phoneticVowel(*(p+3)))) {
                                OUTex('i');     // ..e.vogal..
                                break;
                            }
                            if (!strcmp(p+2, "la") || !strcmp(p+2, "las")) {// ..e.la
                                OUTex('E'); OUTex(iSIL); OUTs("l6");
                                p+=3;
                                ns++;
                                tonica=false;
                                break;
                            }
                            if (p==pal ) {OUTex('i'); break;}   // elemento
                            //if (*(p+2)=='n' || *(p+2)=='m') {
                            //  OUTex('e');          // ..e.n..   ..e.m..
                            //  break;
                            //}
                            if (*(p+2)=='J' || *(p+2)=='L' || *(p+2)=='S' || *(p+2)=='j' ||
                                 (*(p+2)==sTON && (*(p+3)=='J' || *(p+3)=='L' || *(p+3)=='S' || *(p+3)=='j'))) {
                                OUTex('6');          // ..e.lh..   ..e.nh..  ..e.ch..   ..e.j..
                                break;
                            }
                            if (p==pal+1 && tonica) {OUTex('E'); break;} // essa
                            if (tonica && ns==1 && *(p+1)==iSIL && !strchr(p+2, iSIL) &&
                                ( *(p+strlen(p)-1)=='s' || *(p+strlen(p)-1)=='e' || *(p+strlen(p)-1)=='A' ||
                                  *(p+strlen(p)-1)=='E' || *(p+strlen(p)-1)=='I' || *(p+strlen(p)-1)=='O' || *(p+strlen(p)-1)=='U' ))
                                {OUTex('E'); break;} // deves, temes, segues, bebe, bebem

                            tonica? OUTex('e'): OUTex('@');
                            break;
        /*0*/           }else // ....e??..
                            {
                              if (*(p+1)=='r' && ns>1)  {OUTs("Er"); p++; break;}   // ..er..
                                if (*(p+1)=='x')    {OUTs("6jS"); p++; break;}  // ..ex+consoante..
                                if (p==pal ) {if(*(p+1)!='s') OUTex('i'); break;}   // estela
                                if (p==pal+1 && tonica) {OUTex('E'); break;} // essa
                                tonica? OUTex('e'): OUTex('@');
        /*0*/           }
                    }  //fim  do switch
                    break;

case 'o':   if (!strcmp(p, "o\x07o")) {     // ...oo
                OUTex('o'); OUTex(iSIL); OUTex('o');
                p+=2;
                ns++;
                tonica=false;
            }
            else
```

```
                if (!strcmp(p, "o\x07zo")) OUTex('o');      // ...oso
                else
                if (!strcmp(p, "o\x07zos") || !strcmp(p, "o\x07za") || !strcmp(p, "o\x07zas"))
                    OUTex('O');     // ...osos  ...osa  ...osas
                else
                if (!strcmp(p, "o\x07ra") || !strcmp(p, "o\x07re") || !strcmp(p, "o\x07ro") ||
                    !strcmp(p, "o\x07ras")|| !strcmp(p, "o\x07res")|| !strcmp(p, "o\x07ros"))
                    OUTex('O');     // ...ora  ...ore  ...oro      ...oras  ...ores  ...oros
                else
                if ((!strcmp(p, "o\x07ta") || !strcmp(p, "o\x07te") ||
                    !strcmp(p, "o\x07tas")|| !strcmp(p, "o\x07tes")) && tonica)
                    OUTex('O');     // ...ota  ...ote  ...otas  ...otes
                else
                if (p>pal && (!strcmp(p-1, "do\x07xo") || !strcmp(p-1, "do\x07xos")))
                    OUTex('O');     // ...doxo  ...doxos
                else
                if (!strcmp(p, "o\x07Ja") || !strcmp(p, "o\x07Jo") ||
                    !strcmp(p, "o\x07Jas")|| !strcmp(p, "o\x07Jos"))
                    OUTex('o');     // ...onha  ...onho  ...onhas  ...onhos
                else
                if (p>pal && *(p-1)=='a') OUTex('w');  // ditongo ..ao..
                else
                if (*(p+1)=='\0' || (*(p+1)=='s' && *(p+2)=='\0')) OUTex('u');   // ..o  ..os
                else
                if (*(p+1)=='i' || *(p+1)=='u') OUTex('o');// ...oi..  ...ou..
                else
                if (*(p+1)=='z' && *(p+2)=='\0') OUTex('O'); // ...oz
                else
                if ((*(p+1)=='p' || *(p+1)=='c') && *(p+2)==iSIL) OUTex('O'); // oct, opc, opt ...
                else
                if (*(p+1)=='r' && *(p+2)=='\0') OUTex('o'); // ...or
                else
                if ((tonica && p-1==pal && *(p+1)=='r' && *(p+2)==iSIL) ||
                   (p==pal && *(p+1)=='r' && *(p+2)==iSIL))
                    OUTex('O');     // or|...
                else
                if (tonica && *(p+1)=='r')
                    OUTex('O');     // ...|or...
                else
                if (*(p+1)=='l') (tonica || *(p+2)=='\0')? OUTex('O'): OUTex('o'); // ..ol  ..|ol..
                                                                                   //  ...ol...
                else
                if (tonica && *(p+1)==iSIL && *(p+2)=='a')  OUTex('o');     // ...|o.a...
                else
                if (!tonica && (*(p+1)==iSIL && *(p+2)=='a') || (*(p+1)==iSIL && *(p+2)==sTON && *(p+3)=='a') )
                    OUTex('u'); // ...o.a...
                else
                if ((tonica && p-1==pal && *(p+1)=='b' && *(p+2)==iSIL) ||
                   (p==pal && *(p+1)=='b' && *(p+2)==iSIL))
                    OUTex('O');     // ob|...
                else
                if ((tonica && p-1==pal && *(p+1)=='b' && *(p+2)=='s' && *(p+3)==iSIL) ||
                   (p==pal && *(p+1)=='b' && *(p+2)=='s' && *(p+3)==iSIL))
                    OUTex('O');     // ob|...
                else tonica? OUTex('o'): OUTex('u');
                break;
    default:    OUTex(*p);
                    break;
 }      //fim  do switch
return out;
}


char *CTextProc::exceptionTable(char *out)
{
        static char *tab[32][2]={   {"u\7\30la", "O\7\30la"},
                    {"\30ak\7tu", "\30a\7tu"},
                    {"6k\7ti\7vi\7\30da\7d@", "6\7ti\7vi\7\30da\7d@"},
                    {"\30tE\7muS", "\30te\7muS"},
                    {"su\7\30me~\7t@", "sO\7\30me~\7t@"},
                    {"\30pa\7r6", "\30p6\7r6"},
                    {"\30o\7LuS", "\30O\7LuS"},
                    {"u\7bri\7\30ga\7du", "o\7bri\7\30ga\7du"},
                    {"\30E\7l@", "\30e\7l@"},
                    {"6jS\7\30sEp\7tu", "6jS\7\30sE\7tu"},
                    {"\30Op\7ti\7mu", "\30O\7ti\7mu"},
                    {"i\7\30pEr\7mEr\7k6\7du", "i\7\30pEr\7m@r\7ka\7du"},
                    {"e~\7\30prE\7z6", "e~\7\30pre\7z6"},
                    {"se~\7\30prE", "se~\7\30pr@"},
                    {"i\7\30li\7t@", "E\7\30li\7t@"},
                    {"\30ter\7nu", "\30tEr\7nu"},
                    {"\30ser\7vu", "\30sEr\7vu"},
                    {"\30ler\7du", "\30lEr\7du"},
                    {"\30per\7ku", "\30pEr\7ku"},
                    {"s6\7sEr\30dO\7t@", "s6\7s@r\30dO\7t@"},
                    {"6\7\30ROS", "6\7\30RoS"},
                    {"gu\7lu\7\30zEj\7m6S", "gu\7lu\7\30z6j\7m6S"},
                    {"saw\7\30da\7d@", "s6w\7\30da\7d@"},
                    {"saw\7\30do\7zu", "s6w\7\30do\7zu"},
                    {"maj\7\30or", "maj\7\30Or"},
                    {"m@\7\30nor", "m@\7\30nOr"},
                    {"m6\7\30Lor", "m@\7\30LOr"},
                    {"pi\7\30or", "pi\7\30Or"},
                    {"su\7\30or", "su\7\30Or"},
                    {"por", "pur"},
                    {"\30o\7vuS", "\30O\7vuS"},
                    {"i\7ku\7nu\7\30mi\7\66", "i\7kO\7nu\7\30mi\7\66"} };

    char *p=out;
```

```
    int n;
    *p='\0';
    while(*p!=iPAL && *p!=iFRA) p--;
    p++;
    for (n=0; n<32 && strcmp(p,tab[n][0]); n++);
    if (n>=32) return out;
    else{
        strcpy(p,tab[n][1]);
        p+=strlen(tab[n][1]);
        return p;
    }
}


void    CTextProc::init850to1252(unsigned char * tab)
{
  int i;
  for(i=0;i<256;i++) tab[i]= (unsigned char) i;
  tab[128]=199; tab[130]=233; tab[131]=226; tab[133]=224; tab[135]=231; tab[136]=234;
  tab[138]=232; tab[140]=238; tab[144]=201; tab[147]=244; tab[160]=225; tab[161]=237;
  tab[162]=243; tab[163]=250; tab[174]=171; tab[175]=187; tab[181]=193; tab[182]=194;
  tab[183]=192; tab[198]=227; tab[199]=195; tab[210]=202; tab[214]=205; tab[215]=206;
  tab[224]=211; tab[226]=212; tab[227]=210; tab[228]=245; tab[229]=213; tab[233]=218;
  tab[174]=171; tab[175]=187;
  // códigos internos de control
  tab[iFRA]='3'; tab[iPAL]='2'; tab[iSIL]=' '; tab[PON]='4'; tab[sTON]='1'; tab[NUL]='5';
  tab[134]=229; tab[132]=228; tab[148]=246;
}



bool CTextProc::phoneticVowel(unsigned char ch)
{
   if (isVowel(ch)) return true;
    else
        switch(ch) {
            case 'A': case 181: case 182: case 'E': case 210: case 144: case 'I': case 'O': case 'U':
            case 134: case 132: case 148:
                            return true;
            default:        return false;
        }
}


bool CTextProc::sequentialConsonants(unsigned char c1, unsigned char c2)
{
    if (c2=='l' || c2=='r')
        if (c1=='b' || c1=='p' || c1=='d' || c1=='t' || c1=='g' ||
            c1=='v' || c1=='f' || c1=='c'|| c1=='k') return true;
    return false;
}


bool CTextProc::accentedLetter(unsigned char ch)
{
        // á é í ó ú à â ê ô Ê É Å Â
    if (ch==160 || ch==130 || ch==161 || ch==162 || ch==163 ||
         ch==133 || ch==131 || ch==136 || ch==147 || ch==210 || ch==144 || ch==181 || ch==182)  return true;
    else return false;
}


bool CTextProc::accentedTermination(unsigned char *sil)
{
    int nLet=strlen((char *)sil);
    // l r z i is im ins u us um uns
    switch (sil[nLet-1]){
        case 'l':
        case 'r':
        case 'z':
        case 'i':
        case 'I':
        case 'u':
        case 'U': return true;
        case 's': if ((sil[nLet-2]=='i' || sil[nLet-2]=='I') || (sil[nLet-2]=='u' || sil[nLet-2]=='U'))
                            return true;
                  else return false;
        case 'm': if (sil[nLet-2]=='u' || sil[nLet-2]=='i')
                            return true;
                  else return false;
        default:  return false;
    }
}


bool CTextProc::accentedVowel(unsigned char ch)
{
    switch(ch) {
            case 130: case 160: case 131: case 133: case 198: case 136:
            case 161: case 140: case 162: case 147: case 228: case 163:
            case 181: case 182: case 210: case 144: case 134: case 132: case 148:
                            return true;
            default:        return false;
    }
}



bool CTextProc::isVowel(unsigned char ch)
```

```
{
    switch(ch) {
            case 130: case 160: case 131: case 133: case 198: case 136:
            case 161: case 140: case 162: case 147: case 228: case 163:
            case 'a': case 'e': case 'i': case 'o': case 'u':
                        return true;
            default:        return false;
    }
}


bool CTextProc::isConsonant(unsigned char ch)
{
    switch(ch) {
            case 'b': case 'c': case 'd': case 'f': case 'g': case 'h':
            case 'j': case 'l': case 'm': case 'n': case 'p': case 'q':
            case 'r': case 's': case 't': case 'v': case 'x': case 'z':
            case 135: return true;
            default:     return false;
    }
}
```

B.2.16 globals.cpp

```
// global.cpp

#define GLOBALS_CPP

#include <cstdlib>

#include <21160.h>

#include "globals.h"


/*
    This global function is used for memory allocation.

*/
void *mAlloc(int size)
{
    void *ret;
    ret = malloc(size);

    if (!ret)
    {
        set_flag(SET_FLAG1, SET_FLAG);
        // never leaves this place.
        while (true) asm("idle;");
    }

    return ret;
}


void *mallocOutVec(int size)
{
    void *ret;

    do
    {
        ret = heap_malloc(extHeapID, size);
        if (!ret) set_flag(SET_FLAG0, SET_FLAG);
    }
    while(!ret);

    set_flag(SET_FLAG0, CLR_FLAG);

    return ret;
}
```

B.2.17 main.cpp

```
//main.cpp

#include <list>
#include <cstring>
#include <cstdio>

asm("#include <def21160.h>");
#include <21160.h>

#include "cdatabase.h"
#include "ctextproc.h"
#include "globals.h"
#include "cpsola.h"
#include "csound.h"
#include "cserial.h"
```

```
void main()
{
    // Generic variables.
    int nDiph, k;
    DiphoInfo *indList;
    SintData sintData;

    // Programm classes.
    CTextProc textProc;
    CPSOLA sint;
    CDataBase dataBase;
    CSound dac;
    CSerial serialInt(BAUD_RATE);

    // Custom structures.
    VecData<char> text;
    VecData<float> sintOut;
    VecData<unsigned> prosodyInfo;

    // STL classes
    list<PhoneInfo> p1,p2;
    list<PhoneInfo>::iterator itr1;
    list<PhoneInfo>::iterator itr2;

    // Getting the ID for the heap located on external memory.
    extHeapID = heap_lookup_name("ext_heap");

    while(true)
    {
        text = serialInt.getText();
        textProc.getPhonemesList(text.data, p1, p2);
        free(text.data);

        nDiph = p1.size();
        indList = (DiphoInfo *) mAlloc(nDiph*sizeof(DiphoInfo));

        itr1 = p1.begin();
        itr2 = p2.begin();

        k = 0;
        while(itr1 != p1.end())
        {
            indList[k++] = dataBase((*itr1++), (*itr2++));
        }

        p1.clear();
        p2.clear();

        sintData = dataBase.getSintData(indList, nDiph);

        free(indList);

        prosodyInfo = sint.genSintF0Marcks(sintData);
        sintOut.size = prosodyInfo.data[prosodyInfo.size - 1] + ( sintData.sampleSize - sintData.f0Marcks[sintData.f0Size - 1] );
        sintOut.data = (float *) mallocOutVec(sintOut.size);

        sint.doSynthesis(sintData, prosodyInfo.data, sintOut.data, sintOut.size);

        free(prosodyInfo.data);
        free(sintData.samples);
        free(sintData.f0Marcks);
        for (k=0; k<sintData.voiceSize; k++) free(sintData.voiceInfo[k]);
        free(sintData.voiceInfo);
        for (k=0; k<sintData.stressSize; k++) free(sintData.stressInfo[k]);
        free(sintData.stressInfo);

        dac.setOutVec(sintOut);
    }
}
```

B.2.18 seg_init.asm

```
/*****************************************************************************
 *
 * seg_init is needed by the C run time system to assist in the
 * runtime initialization of
 *  stack start address
 *  stack length
 *  malloc heaps
 *
 *  global variables (if mem21k was run)
 *
 * This data must reside in seg_init if mem21k is to be run on executables
 * generated with seg_init code.  Data for segments other than seg_init and
 * seg_rth is compressed into seg_init by mem21k; the __inits value in this
 * section points to this data and must be available to extract the data,
 * so obviously it cannot be *in* the data, which means it has to go in
 * seg_init or seg_rth.  In fact, __inits is in seg_init, below.  Mem21k
 * places a pointer to the initialization data in __inits as part of this
 * processing.
 *
 * There can be multiple heaps specified in this file.
 * Each heap specification consists of the following
 *  8 bytes - ASCII heap name
```

```
 *  2 bytes - unused
 *  2 bytes - heap location
 *        0x0001  PM location
 *        0xFFFF  DM location
 *  6 bytes   zero
 *  6 bytes   heap start address (in high order 32 bits)
 *  6 bytes   heap length (in high order 32 bits)
 *
 ***************************************************************************/

.segment/pm seg_init;

/*
 * The following initializations rely on several values being established
 * externally, typically by the linker description file.
 */

.extern ldf_stack_space;    /* The base of the stack */
.extern ldf_stack_length;   /* The length of the stack */
.extern ldf_heap_space;     /* The base of a primary DM heap "seg_heap" */
.extern ldf_heap_length;    /* The length of heap "seg_heap" */
.extern ldf_ext_heap_space;     /* The base of a primary DM heap "seg_heaq" */
.extern ldf_ext_heap_length;    /* The length of heap "seg_heaq" */


/*
 * The linker description file will typically look something like:
 *
 *  MEMORY
 *  {
 *      heap_memory  { START(0x2c000) LENGTH(0x2000) TYPE(DM RAM) }
 *      stack_memory { START(0x2e000) LENGTH(0x2000) TYPE(DM RAM) }
 *  }
 *
 *  ...
 *
 *  SECTIONS
 *  {
 *      stack
 *      {
 *      ldf_stack_space = .;
 *      ldf_stack_length = MEMORY_SIZEOF(stack_memory);
 *      } > stack_memory
 *
 *      heap
 *      {
 *      ldf_heap_space = .;
 *      ldf_heap_length = MEMORY_SIZEOF(heap_memory);
 *      } > heap_memory
 *  }
 */

.global ___lib_stack_space;
.var    ___lib_stack_space = ldf_stack_space;
.___lib_stack_space.end:

.global ___lib_stack_length;
.var    ___lib_stack_length = ldf_stack_length;
.___lib_stack_length.end:

.global ___inits;

#if defined(__2116x__) || defined(__2126x__)
.var    ___inits = 0;
#else
.var    ___inits = 0;
#endif
.___inits.end:

/* The first two PM words represent the heap name and the heap location */
/*   FFFFFFFF DM location   00000001 PM location */

.global ___lib_heap_space;
.var    ___lib_heap_space[5] =
    0x7365675F6865, /* 'seg_he' */
    0x6170FFFFFFFF, /* 'ap'     */
    0,
    ldf_heap_space,
    ldf_heap_length;
.___lib_heap_space.end:


.global ___lib_ext_heap_space;
.var    ___lib_ext_heap_space[5] =
    0x6578745F6865, /* 'ext_he' */
    0x6170FFFFFFFF, /* 'ap'     */
    0,
    ldf_ext_heap_space,
    ldf_ext_heap_length;
.___lib_ext_heap_space.end:


/* Add more heap descriptions here */

.global ___lib_end_of_heap_descriptions;
.var ___lib_end_of_heap_descriptions = 0; /* Zero for end of list */
___lib_end_of_heap_descriptions.end:

.endseg;
```

B.2.19 SADF.ldf

```
ARCHITECTURE(ADSP-21160)

//
// ADSP-21160 Memory Map:
//   ------------------------------------------------
//   Internal memory  0x0000 0000 to 0x000f ffff
//   ------------------------------------------------
//                    0x0000 0000 to 0x0000 0?ff  IOP Regs
//                    0x0000 0?00 to 0x0001 ffff  (reserved)
//           Block 0  0x0002 0000 to 0x0002 7fff  Long Word (64) Addresses
//           Block 1  0x0002 8000 to 0x0002 ffff  Long Word (64) Addresses
//                    0x0003 0000 to 0x0003 ffff  (reserved)
//           Block 0  0x0004 0000 to 0x0004 ffff  Normal Word (32/48) Addresses
//           Block 1  0x0005 0000 to 0x0005 ffff  Normal Word (32/48) Addresses
//                    0x0006 0000 to 0x0007 ffff  (reserved)
//           Block 0  0x0008 0000 to 0x0009 ffff  Short Word (16) Addresses
//           Block 1  0x000a 0000 to 0x000b ffff  Short Word (16) Addresses
//                    0x000c 0000 to 0x000f ffff  (reserved)
//   ------------------------------------------------
//   Multiproc memory 0x0010 0000 to 0x007f ffff
//   ------------------------------------------------
//                    0x0010 0000 to 0x001f ffff  ADSP-21160 ID=001 Internal memory
//                    0x0020 0000 to 0x002f ffff  ADSP-21160 ID=010 Internal memory
//                    0x0030 0000 to 0x003f ffff  ADSP-21160 ID=011 Internal memory
//                    0x0040 0000 to 0x004f ffff  ADSP-21160 ID=100 Internal memory
//                    0x0050 0000 to 0x005f ffff  ADSP-21160 ID=101 Internal memory
//                    0x0060 0000 to 0x006f ffff  ADSP-21160 ID=110 Internal memory
//                    0x0070 0000 to 0x007f ffff  ADSP-21160 ID=all Internal memory
//   ------------------------------------------------
//   External memory  0x0080 0000 to 0xffff ffff
//   ------------------------------------------------
//
// This architecture file allocates:
//         Internal 256 words of run-time header in memory block 0
//                  256 words of initialization code in memory block 0
//                  38K words of C code space in memory block 0
//                   6K words of C PM data space in memory block 0
//                  32K words of C DM data space in memory block 1
//                   8K words of C heap space in memory block 1
//                   8K words of C stack space in memory block 1

#ifdef __swfa
SEARCH_DIR( $ADI_DSP\211xx\lib\swfa )
#else
SEARCH_DIR( $ADI_DSP\211xx\lib )
#endif

#ifdef _ADI_THREADS
#ifdef __ADI_LIBEH__
$LIBRARIES = libehmt.dlb, libc160mt.dlb, libiomt.dlb, libdsp160.dlb, libcppmt.dlb, libcpprt.dlb;
#else
$LIBRARIES = libc160mt.dlb, libiomt.dlb, libdsp160.dlb, libcppmt.dlb, libcpprt.dlb;
#endif
#else
#ifdef __ADI_LIBEH__
$LIBRARIES = libeh.dlb, libc160.dlb, libio.dlb, libdsp160.dlb, libcpp.dlb, libcpprt.dlb;
#else
$LIBRARIES = libc160.dlb, libio.dlb, libdsp160.dlb, libcpp.dlb, libcpprt.dlb;
#endif
#endif

// Libraries from the command line are included in COMMAND_LINE_OBJECTS.
#ifdef _ADI_THREADS
#ifdef __ADI_LIBEH__
$OBJECTS = 160_cpp_hdr_adamagic_mt.doj, $COMMAND_LINE_OBJECTS;
#else
$OBJECTS = 160_cpp_hdr_mt.doj, $COMMAND_LINE_OBJECTS;
#endif
#else
#ifdef __ADI_LIBEH__
$OBJECTS = 160_cpp_hdr_adamagic.doj, $COMMAND_LINE_OBJECTS;
#else
$OBJECTS = 160_cpp_hdr.doj, $COMMAND_LINE_OBJECTS;
#endif
#endif

MEMORY
{
#ifdef __EZKIT_LICENSE_RESTRICTION_SHARC__

  // Only 21K available for PM

  seg_rth  { TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48) }
  seg_init { TYPE(PM RAM) START(0x00040100) END(0x000401ff) WIDTH(48) }
  seg_pmco { TYPE(PM RAM) START(0x00040200) END(0x00044a43) WIDTH(48) }
  seg_pmda { TYPE(PM RAM) START(0x00047724) END(0x00047ef4) WIDTH(32) }

#else

    seg_rth  { TYPE(PM RAM) START(0x00040000) END(0x000400ff) WIDTH(48)               }
```

```
    seg_init { TYPE(PM RAM) START(0x00040100) END(0x000401ff) WIDTH(48)                }
    seg_int_code { TYPE(PM RAM) START(0x00040200) END(0x00040287) WIDTH(48)                }
    seg_pmco { TYPE(PM RAM) START(0x00040288) END(0x00049aa9) WIDTH(48)                }
    seg_pmda { TYPE(PM RAM) START(0x0004e800) END(0x0004ffff) WIDTH(32)                }

#endif

        seg_ctdm { TYPE(DM RAM) START(0x00050000) END(0x000500ff) WIDTH(32)                }
    seg_dmda { TYPE(DM RAM) START(0x00050100) END(0x00057fff) WIDTH(32)                }
    seg_heap { TYPE(DM RAM) START(0x00058000) END(0x0005dfff) WIDTH(32)                }
    seg_stak { TYPE(DM RAM) START(0x0005e000) END(0x0005ffff) WIDTH(32)                }
    database_seg { TYPE(DM RAM) START(0x2800000) END(0x280b98f) WIDTH(32)               }
    ext_heap { TYPE(DM RAM) START(0x280b990) END(0x281ffff) WIDTH(32)                }
}



PROCESSOR P0
{
    KEEP( _main,___ctor_NULL_marker,___lib_end_of_heap_descriptions )
    OUTPUT( $COMMAND_LINE_OUTPUT_FILE )

    SECTIONS
    {
        // .text output section
        seg_rth
        {
            INPUT_SECTIONS( $OBJECTS(seg_rth) $LIBRARIES(seg_rth))
        } >seg_rth

        seg_init
        {
            ldf_seginit_space = . ;
            INPUT_SECTIONS( $OBJECTS(seg_init) $LIBRARIES(seg_init))
        } >seg_init

#ifdef __EZKIT_LICENSE_RESTRICTION_SHARC__
        seg_int_code
        {
            INPUT_SECTIONS( $OBJECTS(seg_int_code) $LIBRARIES(seg_int_code))
        } >seg_pmco
#else
        seg_int_code
        {
            INPUT_SECTIONS( $OBJECTS(seg_int_code) $LIBRARIES(seg_int_code))
        } >seg_int_code
#endif
        seg_pmco
        {
            INPUT_SECTIONS( $OBJECTS(seg_pmco) $LIBRARIES(seg_pmco))
        } >seg_pmco

        seg_pmda
        {
            INPUT_SECTIONS( $OBJECTS(seg_pmda) $LIBRARIES(seg_pmda))
        } >seg_pmda

                seg_ctdm
                {
                  __ctors = .;    /* points to the start of the section */
                  INPUT_SECTIONS( $OBJECTS(seg_ctdm) $LIBRARIES(seg_ctdm))
                } > seg_ctdm

        seg_dmda
        {
            INPUT_SECTIONS( $OBJECTS(seg_dmda) $LIBRARIES(seg_dmda))
        } > seg_dmda


        stackseg
        {

            // allocate a stack for the application
            ldf_stack_space = .;
            ldf_stack_length = MEMORY_SIZEOF(seg_stak);
        } > seg_stak

        heap
        {
            // allocate a heap for the application
            ldf_heap_space = .;
            ldf_heap_length = MEMORY_SIZEOF(seg_heap);
            ldf_heap_end = ldf_heap_space + ldf_heap_length - 1;
        } > seg_heap

        extHeap
        {
            // allocate a heap for the application in the extarnal memory
            ldf_ext_heap_space = .;
            ldf_ext_heap_length = MEMORY_SIZEOF(ext_heap);
            ldf_ext_heap_end = ldf_ext_heap_space + ldf_ext_heap_length - 1;
        } > ext_heap

        database_seg{
            INPUT_SECTIONS(cdatabase.doj(database_seg))
        }>database_seg

    }
}
```