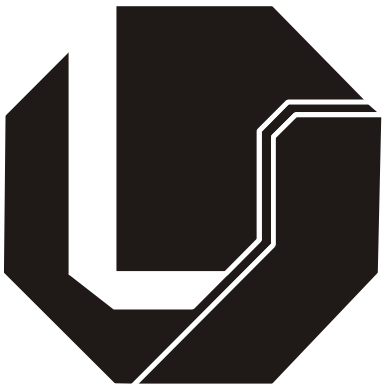
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



Editor MIDI para violão com articulação humanizada nota a nota e qualidade acústica em linguagem funcional pura

Carlos Roberto Ferreira de Menezes Júnior

Julho
2007

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Editor MIDI para violão com articulação humanizada nota a nota e qualidade acústica em linguagem funcional pura

Carlos Roberto Ferreira de Menezes Júnior

Texto da dissertação apresentada à Universidade Federal de Uberlândia, perante banca de examinadores, como parte dos requisitos necessários para a obtenção do título de Mestre em Ciências.

Banca Examinadora:

Prof. Luciano Vieira Lima, Dr. - Orientador (UFU)

Prof. Keiji Yamanaka, Dr. (UFU)

Prof. Haroldo Rodrigues de Azevedo, Dr. (Faculdades LOGATTI de São Carlos)

* A bolsa de estudo para esta pesquisa foi concedida pela CAPES, Brasil.

Dados Internacionais de Catalogação na Publicação (CIP)

M543e Menezes Júnior, Carlos Roberto Ferreira de, 1973-
Editor MIDI para violão com articulação humanizada nota a nota e qualidade acústica em linguagem funcional pura / Carlos Roberto Ferreira de Menezes Júnior. - 2007.
134 f. : il.

Orientador: Luciano Vieira Lima.
Dissertação (mestrado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.
Inclui bibliografia.

1. Música por computador - Teses. 2. Processamento do som por computador - Teses. 3. MIDI (Linguagem de programação de computador) - Teses. I. Lima, Luciano Vieira. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. III. Título.

CDU: 78:681.3

Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

Editor MIDI para violão com articulação humanizada nota a nota e qualidade acústica em linguagem funcional pura

Carlos Roberto Ferreira de Menezes Júnior

Texto da dissertação apresentada à Universidade Federal de Uberlândia como parte dos requisitos necessários para a obtenção do título de Mestre em Ciências.

Prof. Luciano Vieira Lima, Dr.	Prof. Darizon Alves de Andrade, Ph.D
Orientador	Coordenador do curso de Pós-Graduação

Agradecimentos

Primeiramente agradeço a Deus pelo amparo, pelas oportunidades oferecidas e pela vida.

Agradeço a Simone, minha companheira de existência. Sua luz, seu amor, sua paciência e seus carinhos foram fundamentais para realização e conclusão de mais uma etapa dessa nossa jornada.

Agradeço a Sofia, minha filha querida, pela abundância de vida e alegria que trouxe para nossas vidas. Cada sorriso seu faz meu coração transbordar de felicidade e paz.

Agradeço ao meu orientador Prof. Dr. Luciano Vieira Lima. Seu incentivo, sua dedicação e seu companherismo foram essenciais para realização deste trabalho. Suas orientações me proporcionaram vasto conhecimento que guardo com muito carinho pelo resto de minha vida.

Agradeço ao Hélcio pelo companherismo e pelas grandes contribuições que foram dadas a este trabalho.

Agradeço ao Reny pelas valiosas conversas no laboratório de inteligência artificial e pela solidariedade.

Agradeço às minhas amigas do Quarteto VagaMundo, Juliana, Vânia e Daniela pelo imenso privilégio de caminharmos juntos nesta bela tarefa de levar arte às pessoas.

Agradeço ao amigo Fanuel, meu primeiro professor de violão, cujo sua dedicação foi essencial para eu trilhar os caminhos musicais.

Agradeço ao povo brasileiro pelo suporte financeiro que me foi concedido ao longo destes dois anos de mestrado.

Por fim agradeço a minha mãe Mari Neide, meu pai Carlos e meu irmão Lúcio que me deram suporte, amor e condições de estar realizando mais um trabalho.

Resumo

Este trabalho apresenta como objetivo principal implementar um aplicativo computacional que permita ao usuário criar seqüências musicais MIDI para violão. O diferencial dos arquivos (seqüências musicais) gerados reside no fato de que cada nota musical da seqüência MIDI poderá ser editável para inclusão de articulação humanizada, individualizada e com qualidade timbral de um instrumento acústico. O resultado sonoro destas seqüências, além de gerar arquivos MIDI SMF, é renderizado em outros formatos musicais, tais como Wave. O aplicativo é desenvolvido em paradigma funcional puro, baseado em cálculo lâmbda, tanto no desenvolvimento das funções de manipulação dos arquivos, quanto no desenvolvimento da interface visual. Esta escolha evita a utilização de dlls e outros recursos com validade temporal fortemente dependente da versão do sistema operacional. A escolha da linguagem Clean é justificada no trabalho, entre outros motivos por ser uma das duas linguagens mais eficientes da atualidade, conforme análise de benchmarks especializados. Na construção das funções de manipulação dos arquivos SMF e da interface são geradas bibliotecas que permitirão estender o trabalho não apenas para o instrumento violão, mas, também, para outros instrumentos, principalmente para os solistas (nonofônicos). O aplicativo disponibiliza uma interface aderente ao usuário, a qual permite a um músico leigo na manipulação de programas de computação musical, utilizá-la sem dificuldades.

Palavras-chave: MIDI, Wave, SoundFont, violão, Clean, musica, computação sônica, instrumento virtual.

Abstract

This work presents as main goal to implement a computational application that allows the user create MIDI musical sequences for guitar. The difference of the musical sequences generated by the software resides in the fact that each musical note of the MIDI sequence can be modified for the inclusion of humanized articulation, individualized and with timbral qualities like an acoustic instrument. The sound resulted of these sequences is render in other musical formats, such as Wave. The application interface and functions are developed in pure functional paradigm (based on lambda calculus). This choice avoids the use of dlls and other resources strongly dependent of operational system version. The choice of the functional language Clean is justified in this work, by the reason of being one of the two more efficient languages of the actuality (according to analysis of specialized benchmarks). In the implementation of the manipulation functions of SMF files and interface generated libraries that will allow extending this work for another soloists (monophonic) instruments. This application presents an adherent interface with all kinds of users. In order to manipulate SMF files, many libraries were developed, allowing extending this work for other soloists (monophonic) instruments. This application presents an adherent interface with all levels of users.

Key-words: MIDI, Wave, SoundFont, acustic guitar, Clean, music, computer music, virtual instrument.

Publicações

O autor desta dissertação é músico, formado em bacharelado em violão e licenciatura em música pela Universidade Federal de Uberlândia, com especialização em Computação Sônica pela mesma universidade. Possui, também, graduação em engenharia mecânica pela UFU. Durante seu trabalho de pesquisa e desenvolvimento da mesma, publicou, entre outros, os seguintes artigos:

Trabalhos completos em congressos e eventos

1. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; CAMARGO JÚNIOR, Hélcio; PINHEIRO Alan Petrônio. **ALGORITMOS GENÉTICOS APLICADOS À GERAÇÃO AUTOMÁTICA DE ARQUIVOS MIDI SMF COM ARRANJO A QUATRO VOZES** - WCCSETE'2006

2. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; CAMARGO JÚNIOR, Hélcio. **EDITOR MIDI PARA VIOLÃO COM ARTICULAÇÃO HUMANIZADA NOTA A NOTA E QUALIDADE ACÚSTICA EM LINGUAGEM FUNCIONAL PURA -** WCCA'2007.

Artigos completos publicados em revista internacional (Brasil e Portugal)

1. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; MACHADO, André Campos, LIMA, Sandra Fernandes de Oliveira. **Sonar 5 – as novidades da nova versão** .Playmusic, São Paulo, v.97, Novembro de 2005, ISSN/ISBN: 14151871.

2. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; MACHADO, André Campos, LIMA, Sandra Fernandes de Oliveira. **Sonar 5 – as novidades da nova versão parte II** .Playmusic, São Paulo, v.98, Dezembro de 2005, ISSN/ISBN: 14151871.

3. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; MACHADO, André Campos, LIMA, Sandra Fernandes de Oliveira. **Antares Auto-Tune 4** .Playmusic, São Paulo, v.99, Janeiro de 2006, ISSN/ISBN: 14151871.

4. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; LIMA, Sandra Fernandes de Oliveira. **Izotope Spectron 1.07** .Playmusic, São Paulo, v.100, Fevereiro de 2006, ISSN/ISBN: 14151871.

5. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; LIMA, Sandra Fernandes de Oliveira. **Configurando o Sonar 6** .Playmusic, São Paulo, v.100, Fevereiro de 2006, ISSN/ISBN: 14151871.

6. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; LIMA, Sandra Fernandes de Oliveira; CAMARGO JÚNIOR, Hélcio. **Melodyne 3 – editando e corrigindo afinações em áudio** .Playmusic, São Paulo, v.103, Maio de 2006, ISSN/ISBN: 14151871.

7. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; LIMA, Sandra Fernandes de Oliveira; CAMARGO JÚNIOR, Hélcio. **Finale 2006 – a**

ferramenta Measure Tools e o botão direito do mouse .Playmusic, São Paulo, v.103, Maio de 2006, ISSN/ISBN: 14151871.

8. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; LIMA, Sandra Fernandes de Oliveira; CAMARGO JÚNIOR, Hélcio. **PSP Vintage Warmer – Simulador de compressor analógico valvulado** .Playmusic, São Paulo, v.105, Julho de 2006, ISSN/ISBN: 14151871.

9. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira; LIMA, Sandra Fernandes de Oliveira; CAMARGO JÚNIOR, Hélcio. **Sonar 5 – Utilizando o V-Vocal para corrigir afinação** .Playmusic, São Paulo, v.105, Julho de 2006, ISSN/ISBN: 14151871.

10. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Audacity 1.2.4 – Editor de áudio leve e gratuito** .Playmusic, São Paulo, v.106, Agosto de 2006, ISSN/ISBN: 14151871.

11. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Sonar 5 – Como utilizar arquivos SoundFonts com o SFZ** .Playmusic, São Paulo, v.106, Agosto de 2006, ISSN/ISBN: 14151871.

12. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Vienna SoundFonts Studio 2.4** .Playmusic, São Paulo, v.107, Setembro de 2006, ISSN/ISBN: 14151871.

13. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Finale 2006 – A ferramenta Transpose** .Playmusic, São Paulo, v.108, Outubro de 2006, ISSN/ISBN: 14151871.

14. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Sonar 6 – As novidades da nova versão** .Playmusic, São Paulo, v.110, Dezembro de 2006, ISSN/ISBN: 14151871.

15. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Chord**

Dictionary .Playmusic, São Paulo, v.111, Janeiro de 2007, ISSN/ISBN: 14151871.

16. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Sonar 6 – Utilizando a gravação Punch** .Playmusic, São Paulo, v.112, Fevereiro de 2007, ISSN/ISBN: 14151871.

17. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **Finale NotePad 2007** .Playmusic, São Paulo, v.113, Março de 2007, ISSN/ISBN: 14151871.

18. MENEZES JÚNIOR, Carlos Roberto Ferreira; LIMA, Luciano Vieira. **MidiNotate Player** .Playmusic, São Paulo, v.114, Abril de 2007, ISSN/ISBN: 14151871.

19. MENEZES JÚNIOR, Carlos Roberto Ferreira; MARRA, George Mendes; LIMA, Luciano Vieira. **Rosegarden** .Playmusic, São Paulo, v.115, Maio de 2007, ISSN/ISBN: 14151871.

Conteúdo

Capítulo 1 1
Introdução 1
1.1 Justificativa 2
1.2 Objetivo Geral 4
1.3 Objetivos Específicos 5

Capítulo 2 – Os formatos MIDI e WAVE 6
2.1 O Formato WAVE 6
2.1.1 Estrutura do cabeçalho do formato WAVE 9
2.2 O Protocolo MIDI 11
2.2.1 Standard Midi File 15
2.2.2 Estrutura de um Standard Midi File 17
2.2.3 PPQ 20
2.2.4 Delta Time 21
2.2.5 Eventos e Meta eventos 23
2.2.6 Exemplo de um arquivo SMF comentado byte a byte 27

Capítulo 3 - SoundFonts 31
3.1 Criação e edição de SoundFonts 33
3.2 Renderizando SondFonts com o TiMidity++ 38

Capítulo 4 – O Paradigma Funcional e a Linguagem Clean 39
4.1 A opção pela linguagem Clean 39

4.2 Tipos de dados 42
4.3 Funções em Clean 44
4.4 Notação Zermelo-Fraenkel 45
4.5 Técnica de tipos únicos 50
4.6 Implementação de um conversor MIDI --> WAVE 52
4.6.1 A interface gráfica 53
4.6.2 A implementação das funções de cada botão 57
4.7 Implementação de uma interface gráfica para um violão virtual com botões animados 62
4.8 Implementação de um compilador TEXTO --> MIDI 68
4.8.1 A função “geraMidiF1” 78

Capítulo 5 – Implementação do editor MIDI para violão com articulação humanizada nota a nota 83
5.1 A interface gráfica e seus componentes 84
5.1.1 Descrição de cada componete do aplicativo 85
5.2 Implementação das funções de manipulação de texto e partitura 97

Capítulo 6 – Estudo de caso e análise comparativa com outros softwares semelhantes 102
6.1 Análise comparativa dos recursos oferecidos pelos três softwares 104

Capítulo 7 – Conclusão e trabalhos futuros 106
7.1 Trabalhos futuros 108

Referências 109

Anexo I – Parâmetros do TiMidity++ 110
Anexo II – Especificação MIDI 1.0 115
Anexo III – Figuras Musicais 133

Capítulo 1

Introdução

A utilização do computador como um instrumento de criação e manipulação musical está cada vez mais presente na cultura contemporânea. Historicamente, este fato pode ser justificado ao analisar a música ocidental ao longo do tempo, no qual percebe-se que os avanços científicos e tecnológicos de cada período influenciaram o fazer musical e, conseqüentemente, o arcabouço teórico que serve de base para o mesmo. Segundo ABDOUNUR (1999), o primeiro registro científico associando matemática e música ocorreu por volta do século VI a.C. por Pitágoras, relacionando o conceito musical de intervalos com o conceito matemático de frações, fazendo uso de um instrumento de uma corda desenvolvido por ele denominado "Monocórdio". Mais adiante, no começo do século XVII, Jonh Napier introduz o cálculo logarítmico, que possibilitou na música o surgimento de um sistema de organização de notas chamado "sistema temperado", que revolucionou a criação musical no período barroco (século XVII e começo do século XVIII). Outro exemplo é o surgimento da "série de Fourier" no início do século XIX, possibilitando um melhor entendimento dos fenômenos acústico e musicais e novas formulações teóricas e artísticas. Os exemplos supracitados e tantos outros influenciaram também na construção de novos instrumentos, introduzindo novos timbres nos diversos sistemas de criação musical que foram sendo propostos.

No século XX, com o avanço da eletrônica, surgiram instrumentos que não se baseavam mais na produção acústica do som através de sua constituição física e sim na manipulação dos elementos elétricos e magnéticos. Assim surgiu o "Theremin", instrumento musical inventado em 1923 pelo cientista russo Leon Theremin.

Posteriormente, na década de 50, foi criado o sintetizador, que era capaz de sintetizar sons de vários instrumentos. Seu criador foi o americano Bob Moog. Com o surgimento e o desenvolvimento do computador, os músicos passaram a utilizar a imensa gama de possibilidades que tal equipamento poderia oferecer no campo da produção musical. Deste modo surge uma nova área do conhecimento: a Computação Musical ou Sônica (no Brasil). Ela é interdisciplinar pois integra vários campos do conhecimento humano tais como engenharia eletrônica, ciências da computação, física, psicologia, música, lingüística, filosofia, entre outros. Esta área tem sido foco de pesquisas no mundo inteiro resultando desde novas possibilidades estéticas no campo da composição eletroacústica até novas ferramentas no campo da inteligência artificial integrando, por exemplo, técnicas como Algoritmos Genéticos e Redes Neurais Artificiais.

1.1 - Justificativa

Com o advento das pesquisas em Computação Sônica surgiu a necessidade de armazenamento das informações musicais no meio computacional, para que as mesmas pudessem ser processadas e manipuladas, retornando ao usuário os resultados pretendidos. Sendo assim, temos duas tecnologias básicas para registro da informação musical: O protocolo MIDI e seu arquivo padrão SMF (Santard Midi File) e o formato de armazenamento WAVE. Estas duas tecnologias se diferem da seguinte forma:

O formato WAVE é um arquivo de áudio digitalizado. Sendo assim ele armazena pontos do sinal sonoro que é amostrado e quantizado. Desta forma o som analógico passa a ser “discretizado” e armazenado em Bytes. A fidelidade do som armazenado em relação ao som original vai depender principalmente da taxa de amostragem e da quantização. O arquivo MIDI é um protocolo de comunicação padrão entre dispositivos eletrônicos, deste modo ele não armazena o som digitalizado e sim as informações de como um dispositivo MIDI deverá executar a música. Ele contém desde informações gerais tais como tonalidade, número de instrumentos, até as notas e dinâmicas de cada instrumento. Deste modo ele permite que um sintetizador físico ou virtual recrie a música original mantendo-se fiel às características de dinâmica e do conteúdo da informação, mas sem o compromisso timbral dos instrumentos originais.

Cada uma destas tecnologias têm seus pontos fortes e fracos. No caso do formato WAVE, seu ponto forte é a fidelidade com o som original, porém seus pontos fracos são a quantidade excessiva de memória exigida e a dificuldade de extrair as informações musicais dos mesmos por não estarem explicitamente armazenadas. No caso do MIDI, seu ponto forte é que a análise musical fica mais completa a partir de suas informação já que o que é armazenado são as instruções de como cada nota deve ser tocada, utilizando um determinado instrumento, com uma determinada armadura de clave, etc. Deste modo pode-se editar estas informações de maneira rápida e precisa. Tudo isso necessitando uma quantidade extremamente pequena de memória. Sendo assim o MIDI tornou-se a tecnologia mais utilizada no desenvolvimento de softwares voltados para edição de partituras, seqüênciamento musical, jogos, sites multimídias, etc. Os pontos fracos são que o mesmo depende totalmente da qualidade dos sintetizadores para geração dos timbres e que muitas vezes o resultado sonoro soa um pouco "artificial" mesmo utilizando equipamentos caros.

Diante disto abre-se uma questão: Como integrar as vantagens que o protocolo MIDI oferece em termos de armazenamento e manipulação das informações musicais com as qualidades que a tecnologia WAVE oferece em termos de fidelidade com o som original e preservação das nuances acústicas?

Como uma alternativa a esta questão surgiu uma tecnologia chamada SoundFont. Ela foi desenvolvida pela empresa CREATIVE para ser utilizada em suas placas de som no início da década de 90. Em 1994, nasceu a placa "Sound Blaster AWE 32", primeira a utilizar tal tecnologia. A idéia de criar este formato veio a partir da concepção de síntese através dos chamados "wavetables" que ainda estavam no começo do seu desenvolvimento. Deste modo podemos entender que SoundFonts são bancos de timbres que podem ser gravados diretamente de instrumentos acústico, editados e armazenados na memória das placas de som que suportam esta tecnologia. Eles funcionam como "fontes de sons" tendo uma operacionabilidade semelhante às das fontes de textos, sendo que a qualidade de tais fontes vai depender da competência técnica e musical de quem as produzir. Hoje em dia, esta tecnologia é amplamente utilizada tanto por usuários Windows como Linux que têm como principais editores de SoundFonts os programas VIENNA e o SMURF, respectivamente. Deste modo esta tecnologia abre possibilidades de criar bancos de timbres não previstos nos padrões General Midi (GM1) e no General Midi 2 (GM2). Se analisarmos apenas o instrumento violão veremos que existem pelo menos 30 sonoridades diferentes que variam de acordo

com a forma de execução (ataques com apoio, trinados ascendentes, trinados descendentes, ligados, arrastados, entre outros) e que não são contemplados pelos bancos de fontes existentes e, tão pouco, pelos sintetizadores virtuais. Ao levar em consideração estas “nuances” de sonoridades poderemos obter um resultado sonoro mais humanizado e com qualidade acústica melhor, podendo ser utilizado profissionalmente em trabalhos que exijam um acabamento mais minucioso e preciso em termos musicais. Um exemplo de utilização desses recursos são as empresas de comunicação, publicidade e entretenimento que estão abrindo um vasto campo de trabalho para músicos profissionais que dominem não só seus respectivos instrumentos musicais mas, também, os instrumentos “virtuais”. Porém o formato SoundFont resolve apenas o problema de bancos de timbres e não o de criação e edição de arquivos MIDI voltados para um instrumento específico.

Sendo assim surge a necessidade de desenvolver técnicas de programação que solucionem os problemas apresentados acima. É nesta perspectiva que este trabalho se encontra, que é de pesquisar tais técnicas utilizando linguagem funcional pura que permita ao usuário criar sequências de violão mais humanizadas. Para tanto, justifica-se a escolha do paradigma funcional e a linguagem Clean, como sendo uma solução aderente para manipular os objetos do domínio musical, além de ser uma linguagem eficiente e já consagradas em benchmarks confiáveis como “http://shootout.alioth.debian.org/gp4/benchmark.php” .

1.2 – Objetivo Geral

O propósito geral deste trabalho é desenvolver e apresentar técnicas de programação utilizando linguagem funcional pura para implementação de um sistema computacional voltado para a criação e edição de arquivos MIDI SMF que permita ao usuário criar seqüências musicais de violão onde a cada nota pode-se definir tipos de articulações realizada por humanos, renderizando o resultado sonoro em arquivos com definição e qualidade acústica.

1.3 - Objetivos Específicos

- Criar rotinas em linguagem funcional Clean voltadas para manipulação de arquivos SMF (Standard Midi File).
- Desenvolver funções e implementar exemplos que integrem programas em Clean com renderizadores de arquivos MIDI para Wave já consagrados e que utilizem bancos SoundFonts tais como o TiMidity++.
- Apresentar os conceitos e implementar exemplos de criação de GUI (Graphical User Interface) voltados para aplicativos multimídia em linguagem funcional Clean
- Desenvolver técnicas de visualização de partituras em interfaces gráficas.
- Desenvolver um compilador Texto -> MIDI.
- Criar um banco SoundFont editável com articulações humanizadas de violões.
- Apresentar os conceitos de como criar um banco SoundFont com articulações humanizadas.

Capítulo 2

Os formatos MIDI e WAVE

A utilização de sistemas digitais tanto na produção musical quanto nas pesquisas de fenômenos acústico e psicoacústicos proporcionou aos profissionais destas áreas um ganho significativo de ferramentas voltadas para a manipulação do som que vieram facilitar e ampliar as possibilidades de pesquisas sonoras tanto para fins artísticos como para fins científicos. Deste modo as fitas analógicas tornaram-se obsoletas. No caso específico da música, a subtituição das fitas pelo computador fez com que o mesmo ocupasse um papel central não só em termos de registro sonoro mas também como instrumento musical que proporciona possibilidades tímbricas inéditas amplamente pesquisadas nas áreas da composição musical, seja na eletroacústica, seja na música popular. Sendo assim o protocolo MIDI, seu arquivo padrão SMF (Standard Midi File) e o arquivo WAVE tornaram-se os principais formatos a serem utilizados na produção musical contemporânea por se adequarem perfeitamente aos propósitos e anseios dos músicos da atualidade. O que ocorre é que estas duas tecnologias apresentam uma filosofia metodológica de implementação totalmente diferentes entre si, cada uma com suas qualidades e limitações, o que será apresentado a seguir.

2.1 – O formato WAVE

WAVE é um arquivo de áudio digitalizado. Sendo assim ele armazena pontos do sinal sonoro que é amostrado e quantizado. Desta forma o som analógico passa a ser “discretizado” e armazenado em Bytes. A fidelidade do som armazenado em relação ao som original vai depender principalmente da taxa de amostragem e da quantização.

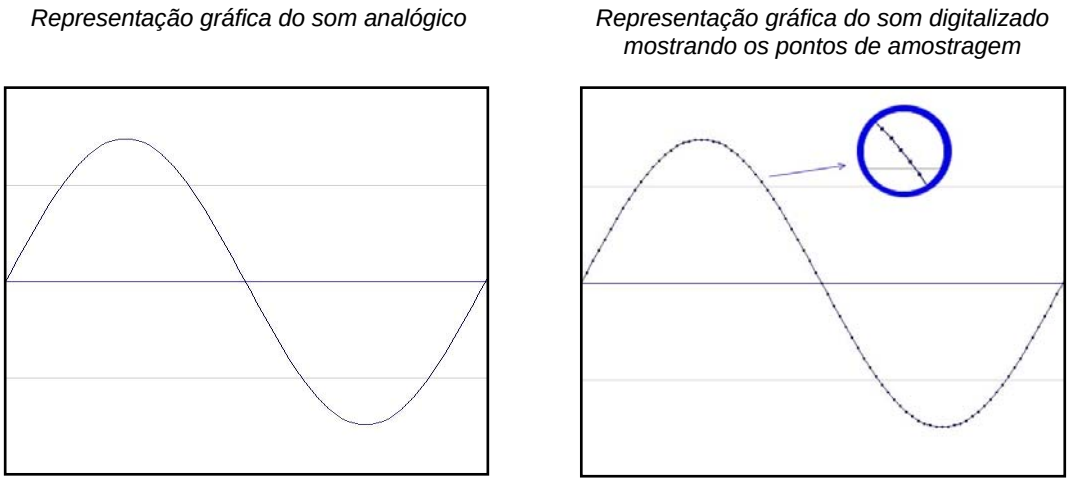


Figura 2.1 – Diferença entre o som analógico e o som digital

Taxa de amostragem é a frequência que o conversor A/D (Analógico→Digital) efetua as amostras de um som durante sua digitalização. Sendo assim, quando um arquivo WAVE apresenta uma taxa de amostragem de 44.100 Hertz significa que no processo de digitalização o conversor A/D "capturou" e armazenou 44.100 pontos a cada segundo. Quanto maior a taxa de amostragem mais fiel ao som original o arquivo WAVE será.

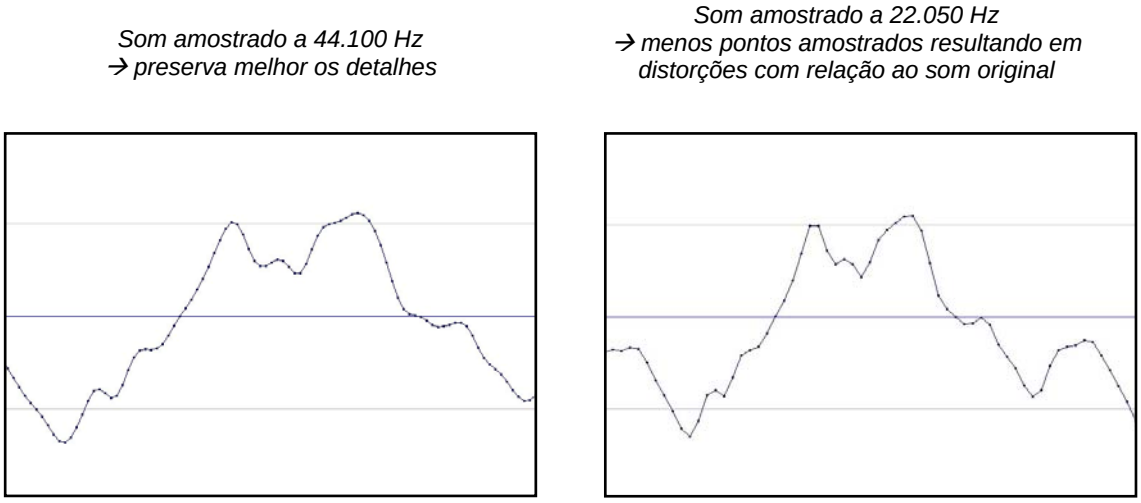


Figura 2.2 – Arquivos WAVE com taxas de amostragem diferentes

A quantização é o número de bits que é disponibilizado à cada ponto amostrado no processo de digitalização para armazenamento das informações. Deste modo um arquivo WAVE com quantização de 8 bits tem a capacidade de representar apenas 256

valores diferentes a cada ponto amostrado. Um arquivo WAVE de 16 bits tem a capacidade de representar 65.536 valores diferentes a cada ponto, sendo assim ele acumula menos erros no processo de digitalização. Quanto maior a quantização, melhor a qualidade dos arquivos e mais fiel ele será ao som original. O padrão do CD é de 44.100 Hz no que se refere a taxa de amostragem e quantização de 16 bits.

Os arquivos do tipo WAVE podem ser monos ou stereos. No arquivo stereo são armazenadas informações sonoras em dois canais independentes possibilitando uma reprodução com capacidade de simular ambientes reais e distribuir espacialmente os instrumentos nestes mesmos ambientes. O arquivo mono armazena as informações em apenas um canal e distribui o mesmo sinal em todas as vias.

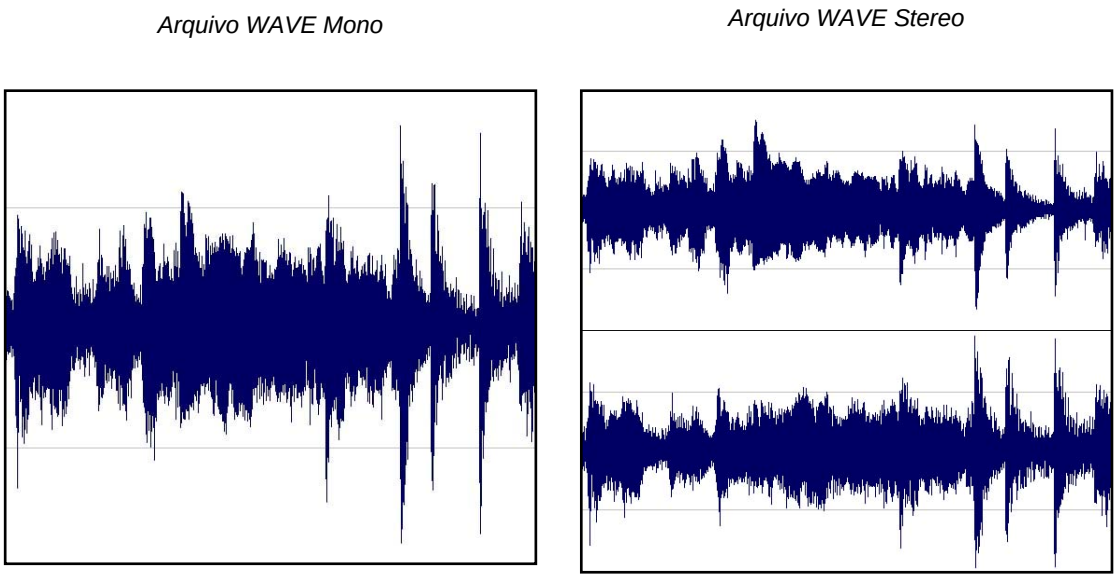


Figura 2.3 – Arquivo WAVE mono e stereo

Estas características são registradas no formato WAVE em um cabeçalho padrão, permitindo que o sinal seja recuperado e analisado com precisão pelas ferramentas matemáticas dedicadas a este fim. Deste modo a estrutura do arquivo WAVE é de um cabeçalho e um corpo de dados. No cabeçalho estão contidas as informações relevantes tais como taxa de amostragem, quantização, número de canais e tamanho do arquivo (em bytes). No corpo de dados estão contidos os pontos amostrados.

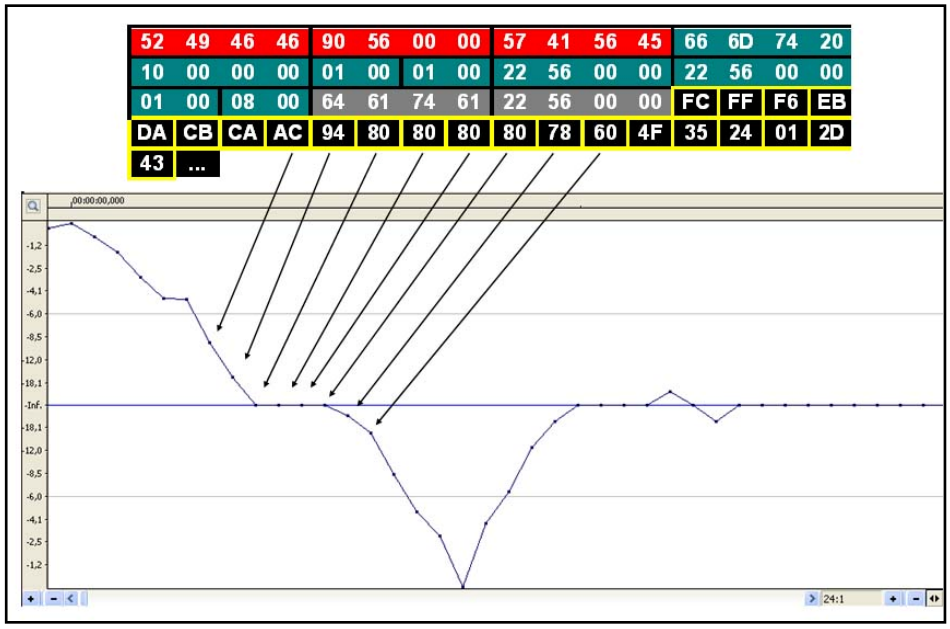


Figura 2.4 – Representação dos bytes em hexadecimal e gráfica de um Arquivo WAVE mono de 8 bits e taxa de amostragem de 22.050 Hz

2.1.1 – Estrutura do Cabeçalho do formato WAVE

O arquivo Wave inicia por um cabeçalho de 36 bits, sendo:

4 (quatro) Bytes	contém a string “RIFF” (caracteres 52 49 46 46 em Hexadecimal)
4 (quatro) Bytes	contém a string “WAVEfmt” (caracteres 57 41 56 45 66 6D 74 20 em Hexadecimal)
2 (dois) Bytes	contém a estrutura, como, por exemplo: PCM (Pulse Code Modulation)
2 (dois) Bytes	contém o número de canais
4 (quatro) Bytes	contém a taxa de amostragem
4 (quatro) Bytes	contém a taxa média de transferência de dados
2 (dois) Bytes	representa a quantidade mínima de bytes utilizados para representar um ponto amostrado: 8 bits mono: 01, 8 bits estéreo: 02, 16 bits mono: 02, 16 bits estéreo: 04.
2 (dois) Bytes	contém o número de bits por amostra. Oito bits = 08, dezesseis bits = 16.

4 (quatro) Bytes	representa a string “data”, caracteres 64 61 74 61 em hexadecimal.
4 (quatro) Bytes	contém o número de Bytes de dados (pontos digitalizados) a serem lidos.
• Obs: Quando uma informação possui mais de um Byte no arquivo Wave, o primeiro Byte da informação é o Byte mais significativo.	

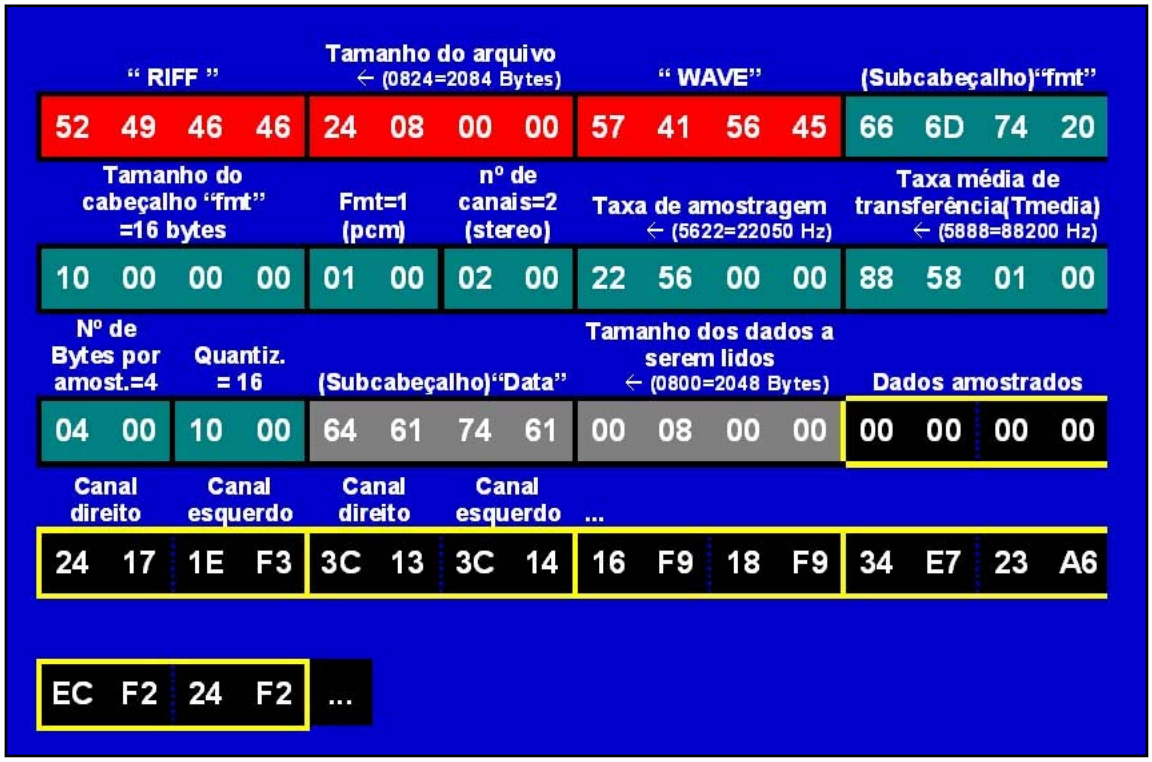


Figura 2.5 – Representação Byte a Byte do cabeçalho de um arquivo WAVE em hexadecimal

2.2 – O protocolo MIDI

MIDI (Musical Instrumento Digital Interface) é um protocolo[1] padrão que apresenta um conjunto de mensagens capazes de levar toda a informação necessária a um equipamento musical eletrônico digital para torná-lo capaz de gerar ou reproduzir músicas. Deste modo o músico pode controlar vários instrumentos musicais simultaneamente, por exemplo os sintetizadores, pois basta editar as informações em apenas um dos equipamentos e transmitir via MIDI para os outros. Sendo assim, ao se tocar uma tecla no instrumento MIDI, ele transmite uma mensagem codificada digitalmente que informa qual foi a tecla pressionada e com que força ela foi pressionada. Essa mensagem é enviada pelo cabo MIDI e pode ser recebida por outro instrumento, que então executará, com seu próprio timbre, a mesma nota. Outras informações musicais podem ser transmitidas via MIDI tais como mudança de andamento, ajuste de volumes, mudança de timbres, entre outras.

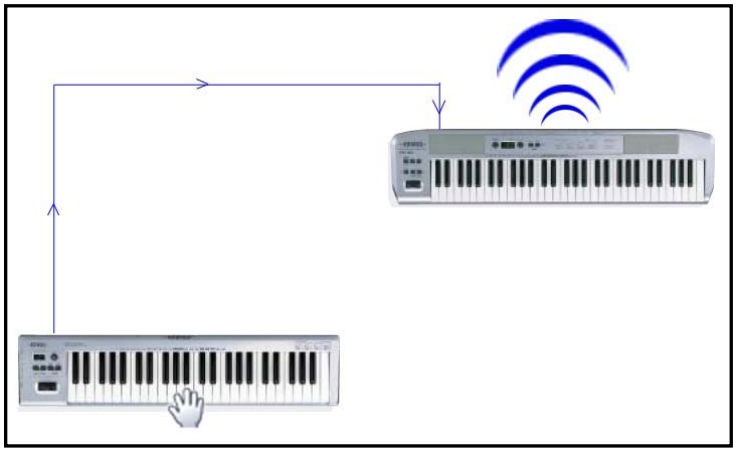


Figura 2.6 – Mensagens MIDI sendo enviadas de um teclado para o outro através de um cabo MIDI

Esta tecnologia surgiu na década de 80, do século XX, quando houve um crescente interesse por música eletrônica, surgindo inúmeros equipamentos musicais que interligavam entre si para gerar novos tipos de sons, permitindo performances

[1] Protocolo - linguagem de comunicação. Uma ferramenta que permite dois individuos ou esquipamentos se comunicarem como por exemplo o inglês, o português, etc.

originais. Muitas vezes era exigido destes músicos um conhecimento razoável de eletrônica, bem como, do sistema, que era bastante complexo e com vários arranjos de fios de interconexão entre os equipamentos. Isso se dava pelo fato de que nesta época os sintetizadores ainda eram monofônicos[2] e monotimbrais[3]. Na busca de soluções para este problema alguns grandes fabricantes de equipamentos musicais se reuniram, em junho de 1981, na feira do NAMM (National Association of Music Merchants), onde definiram a criação de uma interface padrão e um protocolo permitindo que se pudesse conectá-los sem a necessidade de um conhecimento dos circuitos eletrônicos de cada fabricante, simplificando, desta forma, o interfaceamento dos mesmos. Deste modo o músico poderia aproveitar melhor e de uma maneira mais prática e fácil os recursos que cada fabricante oferecia em termos de sonoridade. Esta interface inicialmente foi denominada por **USI** (Universal Synthesizer Interface), gerando a necessidade de um padrão de comunicação com a mesma, denominada por **MIDI** (**M**usical **I**nstrument **D**igital **I**nterface), cuja primeira divulgação dos mesmos (**USI** e **MIDI**) foram apresentados ao público na mostra do **NAMM** de junho de 1982, disponibilizando, em janeiro de 1983, a primeira especificação do protocolo MIDI: a MIDI **Specification 1.0** contendo todos os detalhes do protocolo MIDI e da interface de comunicação. Atualmente existem duas organizações que administram o desenvolvimento do MIDI, são elas: **MMA** (**Midi M**anufacturers **A**ssociation) e **JMSC** (**J**apanese **Midi S**tandards **C**omitee). Embora a implementação inicial tenha sido orientada para sintetizadores, o sistema foi idealizado de tal forma que permitiu ser expandido. Também foi criado um conjunto de especificações de como armazenar as informações musicais em arquivos digitais, para que, futuramente, o músico possa reproduzi-las em seu equipamento. Estes arquivos foram denominados **S**tandard **M**IDI **F**iles (**SMF).**

Com os avanços no campo da informática a *Roland Corporation* resolveu lançar no mercado uma placa de interface MIDI denominada MPU401 para ser utilizada em computadores. O Macintosh foi um dos primeiros a incorporar esta tecnologia sendo que apenas em 1987 os computadores da linha PCs da IBM e compatíveis começaram a utilizar este padrão. Hoje em dia todos os sistemas de música profissional digital utilizam o padrão MIDI (interface e protocolo). Os que não utilizam a interface física emulam as mesmas por software criando uma máquina MPU401 virtual.

[2] Monofônico – equipamento capaz de tocar apenas uma nota de cada vez não sendo possível gerar duas ou mais notas simultaneamente.

[3] Monotimbral – equipamento capaz de gerar apenas um timbre de cada vez.

Figura 2.7 – Mensagens MIDI sendo enviadas de um computador para dois teclados

A transferência das mensagens MIDI de um equipamento para outro é efetuado na forma “serial”, isto quer dizer que os bits das mensagens são transferidos um a um. Como a transmissão ocorre numa velocidade suficientemente rápida (31.250 bits por segundo) os dados são recebidos e interpretados em tempo hábil por um equipamento na outra extremidade do cabo. Como a transmissão serial do MIDI é do tipo assíncrona, é necessário sinalizar ao receptor o início do bloco de dados, para que ele possa temporizar a recepção. Isso é feito adicionando-se dois bits a cada byte, um antes chamado de “start bit” e tem o valor =0 outro no fim chamado de “stop bit” e tem o valor = 1. Com esses bits o circuito receptor pode determinar com precisão onde começa e onde termina o pacote de oito bits e detectar seguramente cada um dos bits do código. Dessa forma, o código 11010011 é transmitido pelo cabo MIDI da seguinte forma : 0110100111 totalizando dez bits.

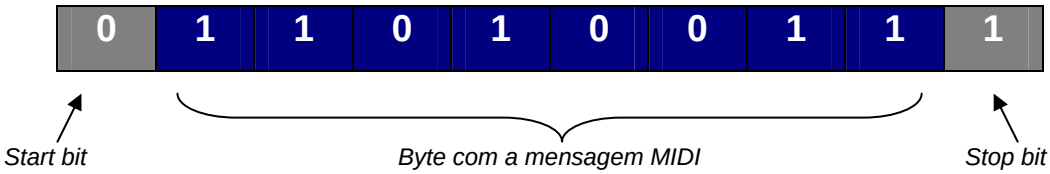


Figura 2.8 – Representação dos bits da mensagem MIDI com mais um bit no início (start bit) e outro no fim (stop bit).

As mensagens MIDI são organizadas da seguinte forma:

Tabela 2.1 – classificação dos tipos de mensagens MIDI

Mensagens de Canal	Mensagens de voz →	Note on Note off Pitch bender Program change Control change Channel aftertouch Polyphonic aftertouch
	Mensagens de modo →	Local control All notes off Omni on Omni off Poly mode Mono mode
Mensagens de Sistema	Mensagens comuns →	Midi time code Song position pointer Song select Tune request EOX (end of SysEx)
	Mensagens de tempo-real →	Timing clock Start Continue Stop Active sensing Reset
	Mensagens exclusivas →	Manufacture data dump
	Mensagens exclusives universais →	MIDI time code (sincronismo e edição) MIDI show control Notation information MIDI machine control MIDI tuning standard Sample dump standard General MIDI

2.2.1 – Standard MIDI Files (SMF).

Cinco anos depois da criação do MIDI 1.0 Specification (1983) foram criadas as especificações do arquivo MIDI padrão ou Standard MIDI Files 1.0 (1988). Estas especificações definem as características do formato de arquivo para armazenamento de mensagens MIDI registradas seqüêncialmente. O objetivo foi oferecer um formato universal que possa ser compartilhado por equipamentos e softwares de diversos fabricantes.

Para armazenar e executar estes arquivos, foi criada uma máquina MIDI com uma arquitetura bastante simples e que pudesse ser implementada a baixo custo e com a tecnologia limitada da época de sua concepção. Assim, a mesma deveria possuir um número pequeno de componentes que pudessem executar todos os comandos necessários para se registrar e reproduzir uma seqüência musical. Este objetivo resultou em uma máquina que possui apenas um contador, um registrador/decodificador de eventos (status) e um registrador de dados. Ou seja, uma máquina que consegue armazenar uma mensagem MIDI, decodificá-la e saber quando deverá executá-la.

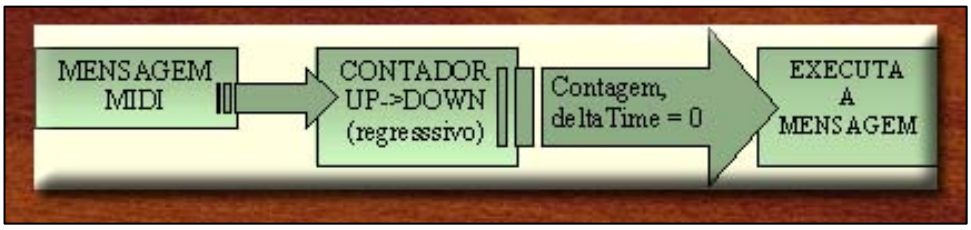


Figura 2.9 – Estrutura da máquina MIDI

Deste modo o registro de informações em um arquivo SMF feito pela máquina MIDI, como por exemplo ativar e desativar notas, ocorre da seguinte maneira: Primeiro é armazenada a mensagem de ativar uma nota musical em um determinado canal e seu volume, depois é disparado o contador que acumulará uma contagem até que ocorra um novo evento que gere uma nova mensagem, ao receber a nova mensagem, a contagem é finalizada e armazenada[4], feito isto o contador é zerado para armazenar o tempo da nova mensagem e então ocorre o armazenamento desta nova mensagem (desativar a nota musical) e repete-se o processo até que os eventos MIDI terminem e se finalize a geração do SMF.

[4] A contagem é armazenada em um ou mais Bytes (de 1 a 4), conforme o tempo de duração do evento. Estes bytes são chamados de DeltaTime.

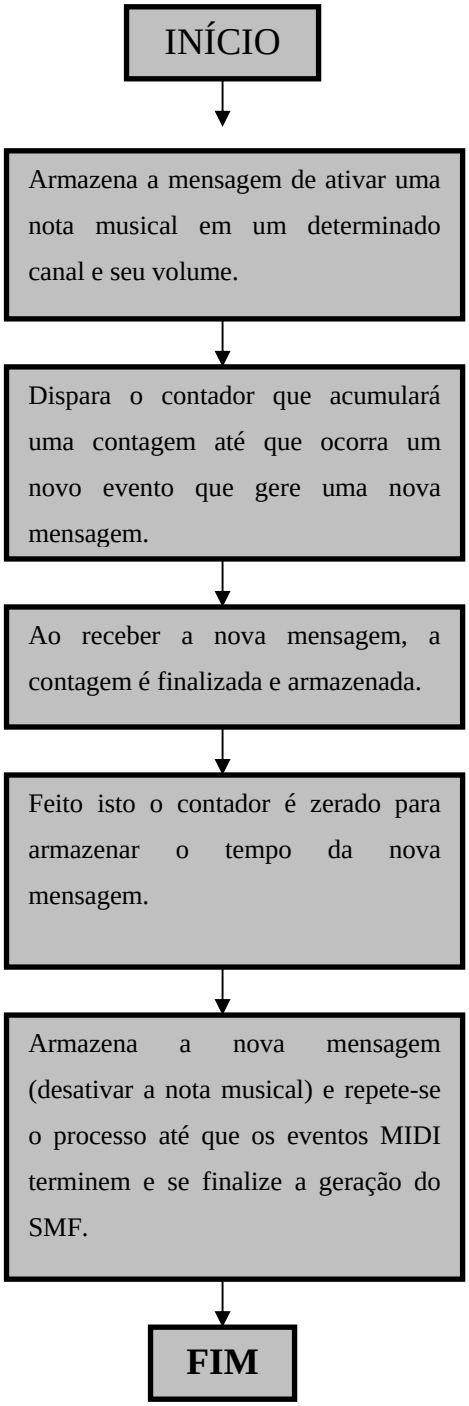


Figura 2.10 – esquema de como ocorre o registro dos eventos MIDI em um SMF.

No que se refere a reprodução de um arquivo MIDI padrão, a máquina MIDI recebe de um a quatro bytes de informação indicando uma contagem regressiva ao tempo em que a mesma deve esperar para executar a mensagem que virá logo após esta contagem. Sendo assim quando o contador chegar a zero, o sistema lê e executa a mensagem MIDI em questão. Posteriormente o sistema lê nova contagem e repete o processo até que uma mensagem de fim de track seja recebida.

2.2.2 – Estrutura de um Standard MIDI Files

A estrutura básica de um arquivo MIDI padrão é a seguinte:

- Cabeçalho (Header Chunck): Representado pelos primeiros 14 bytes, contendo informações sobre tipo de arquivo, formato do arquivo, número de tracks (trilhas) e a PPQ (Pulse per Quarter)
- Trilhas (Track Chunck): Onde ficam as informações musicais tais como notas, duração, fórmula de compasso, timbres, etc.

Figura 2.11 – Estrutura do cabeçalho de um SMF em hexadecimal.

Inicialmente foram criados 3 (três) tipos básicos de formatos: o formato 0, o formato 1 e o formato 2. Destes, sobreviveram, comercialmente, os formatos 0 e 1. As diferenças entre o SMF formato 0 e o SMF formato 1 são as seguintes: O formato 0 é destinado para teclados e equipamentos que vão ler o arquivo e executá-los praticamente em tempo real, ou seja, não precisa ler todo o arquivo e interpretá-lo para depois reproduzi-lo. Assim, no instante da geração dos arquivos SMF formato 0, os eventos são armazenados na ordem em que são gerados, independente de qual canal MIDI o tenha executado. Assim, quando a máquina MIDI for ler este arquivo, bastará ao mesmo executar cada evento na ordem em que aparecem, esperando a contagem de cada DeltaTime, precedente ao evento, chegar a zero antes de executá-los. Portanto, nos SMF formato 0, todos os eventos de todos os canais são armazenados em seqüência em um único track de informação.

O formato 1 é destinado principalmente para softwares de editoração de grades orquestrais de partituras, onde, a princípio, o formato 0 é inapropriado, não aderente ao formato de uma partitura. Diz-se não aderente porque em uma grade orquestral tem-se registrado a partitura de todos os instrumentos musicais em pentagramas diferentes.

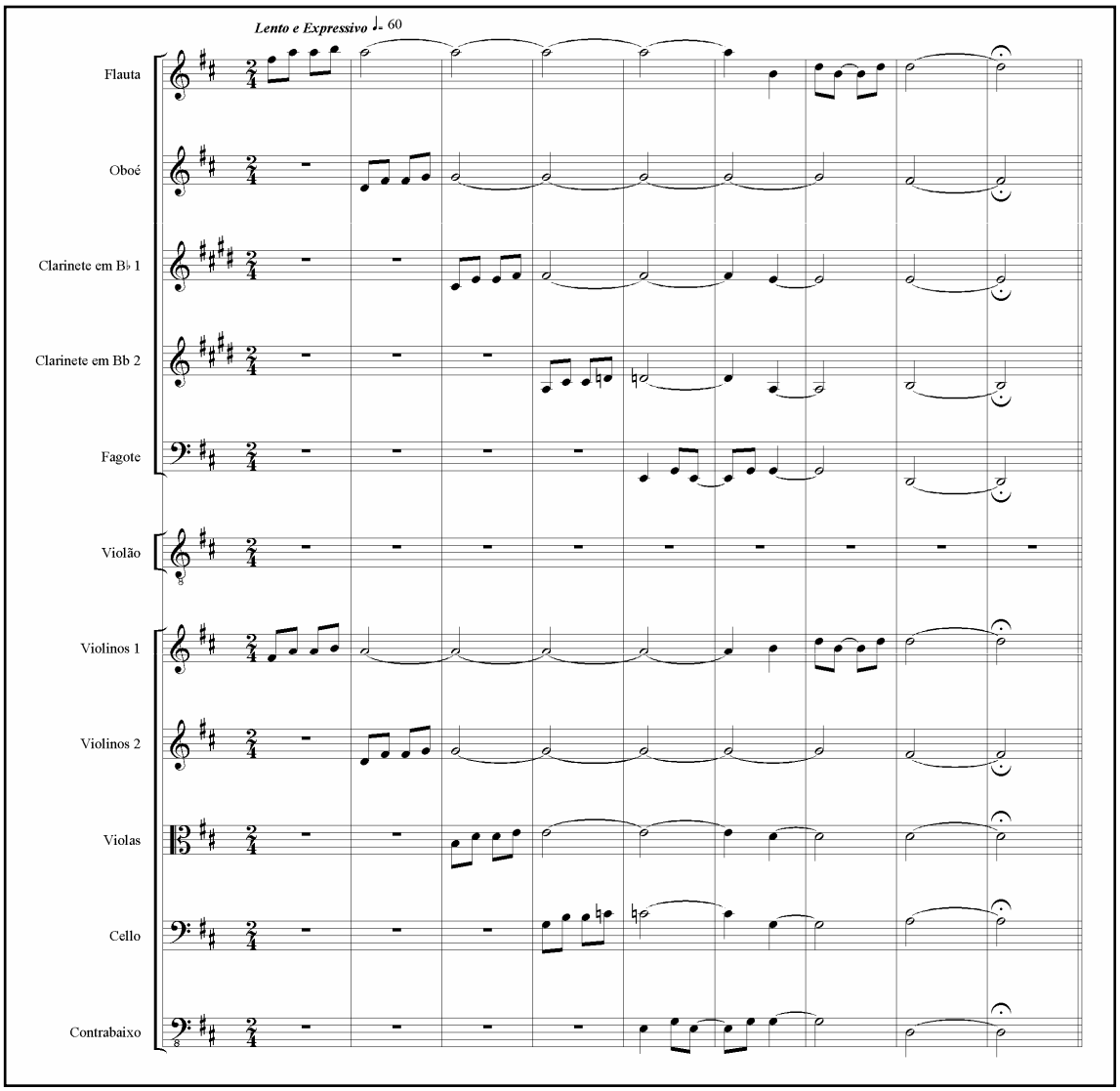


Figura 2.12 – Exemplo de uma grade orquestral

Observe que em uma grade orquestral se tem o registro de todos os eventos separadamente, mas temporalmente dependentes, ou seja, sincronizados. No SMF formato 0 isto não ocorre, os eventos, registrados em mensagens MIDI, estão registrados em um só track de informação, sequenciadamente de acordo com que ocorrem, necessitando que sejam separados por instrumentos (canais MIDI), recalculando o tempo por canal para que se possa ter uma visão paralela, sincronizada, de todos os eventos ao mesmo tempo.

Assim, surge o formato 1, onde cada ocorrência de eventos de um canal MIDI (instrumento da grade orquestral), será registrado em uma ou mais trilhas (tracks) de informação independente. Diz-se uma ou mais trilhas devido ao fato de que em uma grade orquestral pode-se ter mais de um pentagrama de um mesmo instrumento, como no caso de um quarteto de violões (4 violões tocando melodias diferentes, por exemplo). Desta forma, em um arquivo SMF formato 1, também pode-se ter mais de uma trilha de informação de **um mesmo instrumento** (canal), com conteúdos diferentes.

De posse das informações separadas de cada execução musical, um programa de editoração de partituras não terá dificuldade em plotar um pentagrama para cada uma das trilhas (tracks) musicais registradas. Geralmente as informações gerais tais como formula de compasso, tempo do metrônomo, armadura de clave, entre outras, são armazenadas no primeiro track[5]. Estas informações são conhecidas como MetaEventos.

Figura 2.13 – Estrutura do SMF Formato 0 e do Formato 1

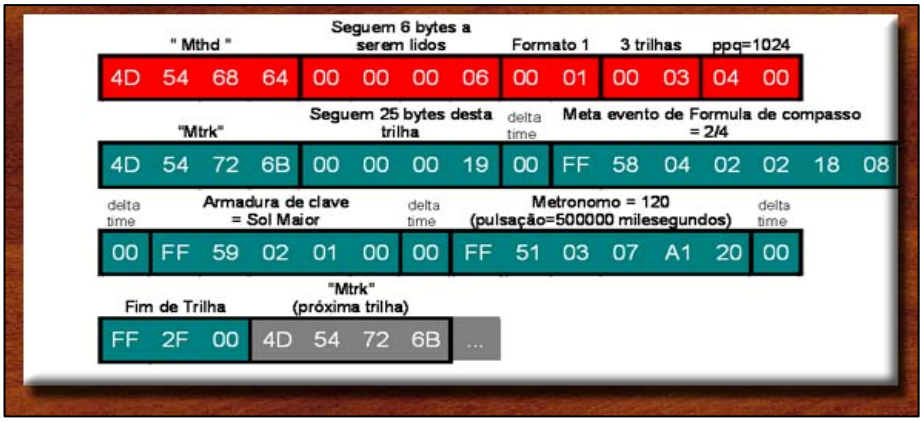


Figura 2.14 – Exemplo de um arquivo SMF formato 1 em hexadecimal

[5] Estas informações também podem ser armazenadas em qualquer outro Track

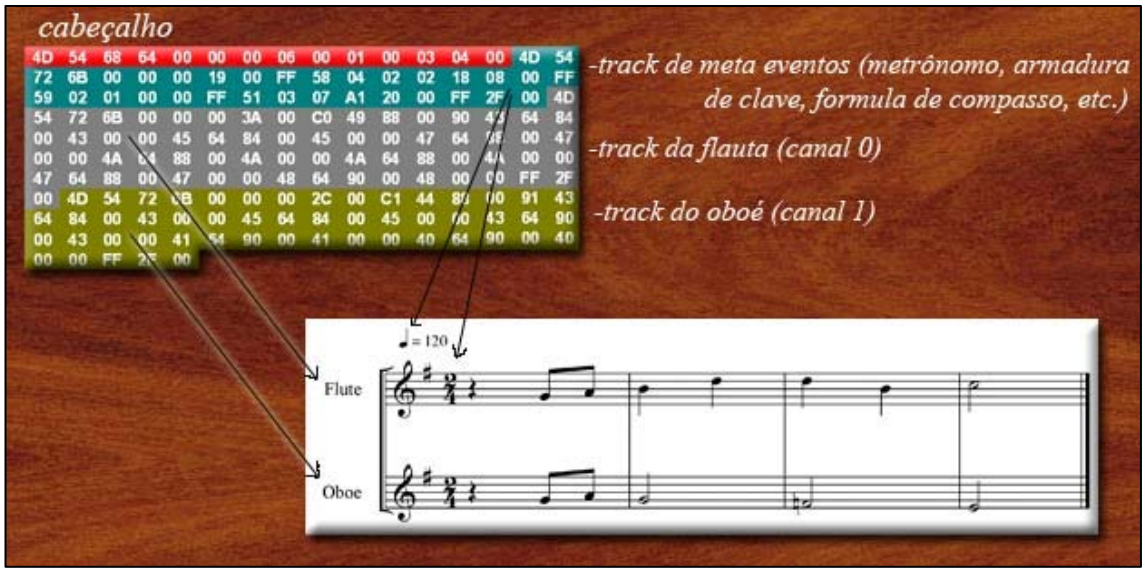


Figura 2.15 – Exemplo de um arquivo SMF formato 1 e sua respectiva partitura.

2.2.3 – PPQ

PPQ significa: Pulses Per Quarter Note, ou seja, número de pulsos por unidade de tempo musical igual a semínima (quarter note em inglês). Em outras palavras, ppq significa em quantas partes temporais será dividida uma semínima para contagem de tempo. Deste modo se a ppq tiver o valor de 240, significa que a semínima vale 240, a colcheia vale 120, a semicolcheia vale 60 e assim por diante. O valor da ppq é definido por dois Bytes, o qual é registrado no cabeçalho principal de um arquivo MIDI SMF, de acordo com o que foi apresentado anteriormente**.**

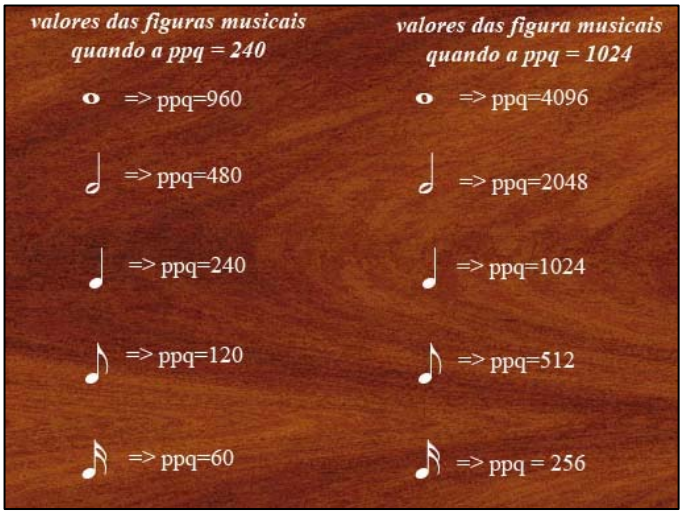


Figura 2.16 – valores das figuras musicais de acordo com o valor da PPQ.

2.2.4 – Delta Time

Delta Time é a contagem do tempo em que um evento MIDI fica ativado. Por padrão a mensagem MIDI sempre possui um Delta Time seguido de um evento ou meta evento. Este valor é registrado com 1(um) a 4(quatro) Bytes.

Figura 2.17 – Delta time seguido de eventos ou meta eventos (padrão adotado nas mensagens MIDI)

Desta forma, com uma Ppq = 240, se um Delta Time de um evento possuir valor igual a 720, isto significa que o mesmo possui uma duração equivalente ao tempo de uma mínima (480) mais o tempo de uma semínima (240), ou seja: 480 + 240 = 720. Portando, com este recurso, não se tem mais necessidade de conhecer a velocidade da máquina MIDI que gerou as contagens do arquivo SMF. Isto se dá devido ao fato de que o mesmo contador que conta o tempo de uma semínima (do padrão de conversão), contará, também, na mesma velocidade, o tempo de todos os eventos do SMF.

Figura 2.18 – Exemplo de uma mensagem MIDI com delta time seguido de um evento ou meta evento em hexadecimal.

A forma como o Delta Time é registrado se difere da PPQ da seguinte maneira: A PPQ é registrada utilizando bytes completos, ou seja, utilizando os 8 bits de um byte. Os Delta Time são registrados utilizando apenas os 7 bits menos significativos de um byte. O motivo de se grafar os Delta Time desta forma reside no fato de que o oitavo bit é utilizado para informar à máquina MIDI quantos bytes de contagem o Delta Time possuirá. Deste modo quando este oitavo bit tiver o valor = 1 (um), significa que o próximo byte ainda pertence ao registro do Delta Time, quando o valor do oitavo bit for igual a 0 (zero) significa que este é o último byte de registro do Delta Time. Sendo assim, quando é preciso utilizar 8 bits para armazenar o valor do Delta Time utiliza-se os 7 bits do byte menos significativo e o bit menos significativo do byte mais significativo. O limite de número de bytes utilizados para registrar o Delta Time é de 4 bytes.

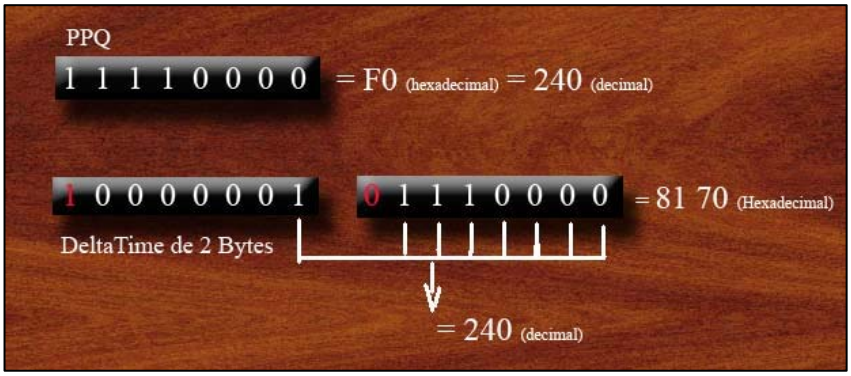


Figura 2.19 – Para registrar a PPQ com valor = 240 (decimal) utiliza-se os 8 bits do byte, para registrar o Delta Time com valor =240 utiliza-se utiliza-se os 7 bits do byte menos significativo e o bit menos significativo do byte mais significativo.

Em uma CPN **(** Common Practice Notation , ou seja, partitura tradicional), o valor absoluto, em segundos, do tempo das notas musicais era obtido pelo conhecimento da figura musical desejada, associado ao valor do metrônomo adotado e registrado pelo músico na partitura. Nos arquivos MIDI SMF, o tempo absoluto, em segundos, de um evento, é calculado determinando-se quantas semínimas equivale uma determinada contagem, e, de posse deste valor, deve-se multiplicar pelo tempo de cada semínima, conforme equação a seguir:

Tempo de uma contagem = (valor do Delta Time / Ppq) x (tempo de uma semínima)

O valor temporal, em segundos, da semínima, é registrado em um Meta Evento chamado: **Set Tempo, ou Tempo do Metrônomo**[6]. Este Meta Evento informa o tempo da semínima em microssegundos e utiliza 6(seis) bytes sendo que os dois primeiros informam o tipo de Meta Evento (no caso o Set Tempo), o próximo byte informa o número de bytes de dados deste Meta Evento e os 3(três) últimos Bytes o tempo em microsegundos ao qual é levado em consideração os 8 (oito) bits de cada Byte.

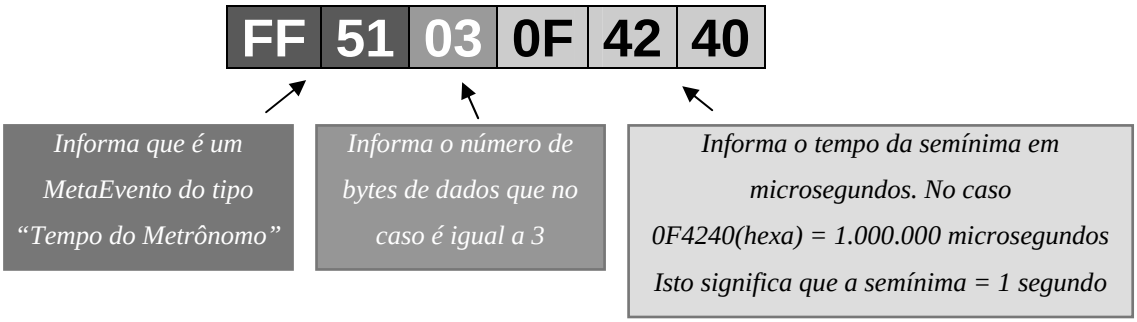


Figura 2.20 – Exemplo de um Meta Evento do tipo Set Tempo (tempo do metrônomo)

2.2.5 – Eventos e MetaEventos

Eventos MIDI é qualquer mensagem de canal, ou seja, eventos de ativar ou desativar determinada nota, mudar de instrumento, mudar volume do canal, etc. Os Meta Eventos são eventos gerais e não de um canal específico. Neles estão contidas informações úteis e necessárias para os equipamentos que executarão os eventos MIDI, tais como armadura de clave, formula de compasso, tempo do metrônomo, entre outros. Um Meta Evento inicia-se com o byte FF(hexadecimal), logo em seguida vem um byte contendo a informação do tipo de Meta Evento, depois vem um byte informando quando bytes de dados virão a seguir e por fim vem os bytes de dados. O Meta Evento "Set Tempo" abordado acima é um exemplo deste tipo de mensagem e segue esta mesma estrutura apresentada.

[6] Metrônomo é um aparelho utilizado para indicar o andamento de uma música. Quando uma partitura apresenta metrônomo = 96 significa que o mesmo irá tocar batidas regulares (semelhante aos "tique" de um relógio) com uma freqüência de 96 por minuto. Sendo assim se o mesmo tiver um valor de 60, o tempo de cada "tique" será de 1 (um) segundo. Quanto maior o valor do metrônomo, mais rápido será o andamento da música.

Veja a seguir a codificação dos eventos e meta eventos contidos nas mensagens MIDI.

Hexadecimal	Decimal	função
80 – 8F	128 – 143	Desativar nota nos canais 0 – 15
90 – 9F	144 – 159	Ativar nota nos canais 0 – 15
A0 – AF	160 – 175	After Touch nos canais 0 – 15
B0 – BF	176 – 191	Control Change nos canais 0 – 15
C0 – CF	192 – 207	Mudar instrumento nos canais 0 – 15
D0 – DF	208 – 223	Pressão (fixa) nos canais 0 – 15
E0 – EF	224 – 239	Pitch Wheel nos canais 0 – 15
F0	240	Mensagens exclusivas
F7	247	Fim das Mensagens exclusivas

Figura 2.21 – Codificação dos Eventos MIDI

Hexadecimal	Decimal	função
FF 01	255 1	Texto
FF 02	255 2	Direito autoral
FF 03	255 3	Título
FF 04	255 4	Nome do instrumento
FF 05	255 5	Lirismo
FF 06	255 6	Marcador
FF 07	255 7	Sugestão de ponto
FF 20	255 32	Prefixo de canal
FF 2F	255 47	Fim de Track

Hexadecimal	Decimal	função
FF 51	255 51	Tempo do metrônomo
FF 54	255 84	Inicio SMPTE
FF 58	255 88	Fórmula de compasso
FF 59	255 89	Armadura de clave
FF 7F	255 127	Sequenciador específico

Figura 2.22 – Codificação dos Meta Eventos mais utilizados.

Os eventos de ativar e desativar as notas sempre vem seguidos de um byte representando a nota musical a ser ativada ou desativada e um outro byte com o valor do volume.

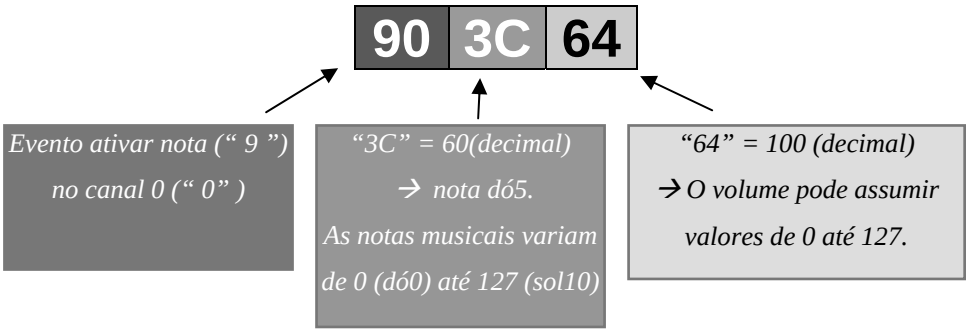


Figura 2.23 – Exemplo de um Evento de ativar nota

Os eventos de mudança de instrumento (Program Change) é indicado por apenas 2 (dois) bytes sendo que no segundo está indicado o código do instrumento.

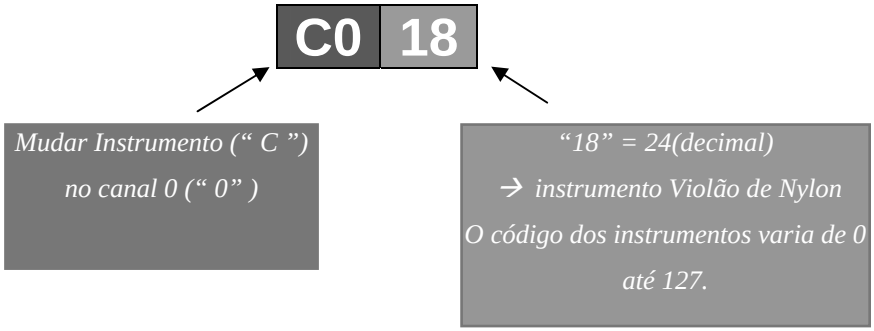


Figura 2.24 – Exemplo de um Evento de mudança de instrumento

Foi criado um padrão no que se refere a codificação dos instrumentos demonimado General MIDI (GM). A seguir será apresentada uma tabela com os códigos em decimal e hexadecimal dos instrumentos padrão GM.

• **Piano**	• **Chrom Perc**	• **Organ**	• **Guitar**
0=Acoustic Grand Piano 00 1=Bright Acoustic Piano 01 2=Electric Grand Piano 02 3=Honky-tonk Piano 03 4=Electric Piano 1 04 5=Electric Piano 2 05 6=Harpsichord 06 7=Clavi 07	8=Celesta 08 9=Glockenspiel 09 10=Music Box 0A 11=Vibraphone 0B 12=Marimba 0C 13=Xylophone 0D 14=Tubular Bells 0E 15=Dulcimer 0F	16=Drawbar Organ 10 17=Percussive Organ 11 18=Rock Organ 12 19=Church Organ 13 20=Reed Organ 14 21=Accordion 15 22=Harmonica 16 23=Tango Accordion 17	24=Acou Guitar (nylon) 18 25=Acou Guitar (steel) 19 26=Elect Guitar (jazz) 1A 27=Elect Guitar (clean) 1B 28=Elect Guitar (muted) 1C 29=Overdriven Guitar 1D 30=Distortion Guitar 1E 31=Guitar Harmonics 1F
• **Bass**	• **Strings**	• **Ensemble**	• **Brass**
32=Acoustic Bass 20 33=Electric Bass (finger) 21 34=Electric Bass (pick) 22 35=Fretless Bass 23 36=Slap Bass 1 24 37=Slap Bass 2 25 38=Synth Bass 1 26 39=Synth Bass 2 27	40=Violin 28 41=Viola 29 42=Cello 2A 43=Contrabass 2B 44=Tremelo Strings 2C 45=Pizzicato Strings 2D 46=Orchestral Harp 2E 47=Timpani 2F	48=String Ensemble 1 30 49=String Ensemble 2 31 50=SynthStrings 1 32 51=SynthStrings 2 33 52=Choir Aahs 34 53=Voice Oohs 35 54=Synth Voice 36 55=Orchestra Hit 37	56=Trumpet 38 57=Trombone 39 58=Tuba 3A 59=Muted Trumpet 3B 60=French Horn 3C 61=Brass Section 3D 62=Synth Brass 1 3E 63=Synth Brass 2 3F
• **Red**	• **Pipe**	• **Synth Lead**	• **Synth Pad**
64=Soprano Sax 40 65=Alto Sax 41 66=Tenor Sax 42 67=Baritone Sax 43 68=Oboe 44 69=English Horn 45 70=Bassoon 46 71=Clarinet 47	72=Piccolo 48 73=Flute 49 74=Recorder 4A 75=Pan Flute 4B 76=Bottle Blow 4C 77=Shakuhachi 4D 78=Whistle 4E 79=Ocarina 4F	80=Lead 1 (square) 50 81=Lead 2 (sawtooth) 51 82=Lead 3 (calliope lead) 52 83=Lead 4 (chiff lead) 53 84=Lead 5 (charang) 54 85=Lead 6 (voice) 55 86=Lead 7 (fifths) 56 87=Lead 8 (bass + lead) 57	88=Pad 1 (new age) 58 89=Pad 2 (warm) 59 90=Pad 3 (polysynth) 5A 91=Pad 4 (choir) 5B 92=Pad 5 (bowed) 5C 93=Pad 6 (metallic) 5D 94=Pad 7 (halo) 5E 95=Pad 8 (sweep) 5F
• **Synth Effects**	• **Ethnic**	• **Percussive**	• **Sound Effects**
96=FX 1 (rain) 60 97=FX 2 (soundtrack) 61 98=FX 3 (crystal) 62 99=FX 4 (atmosphere) 63 100=FX 5 (brightness) 64 101=FX 6 (goblins) 65 102=FX 7 (echoes) 66 103=FX 8 (sci-fi) 67	104=Sitar 68 105=Banjo 69 106=Shamisen 6A 107=Koto 6B 108=Kalimba 6C 109=Bagpipe 6D 110=Fiddle 6E 111=Shanai 6F	112=Tinkle Bell 70 113=Agogo 71 114=Steel Drums 72 115=Woodblock 73 116=Taiko Drum 74 117=Melodic Tom 75 118=Synth Drum 76 119=Reverse Cymbal 77	120=Guitar Fret Noise 78 121=Breath Noise 79 122=Seashore 7A 123=Bird Tweet 7B 124=Telephone Ring 7C 125=Helicopter 7D 126=Applause 7E 127=Gunshot7F

Tabela 2.2 – Instrumentos do padrão GM (Geneal MIDI)

2.2.6 – Exemplo de um arquivo SMF comentado Byte a Byte

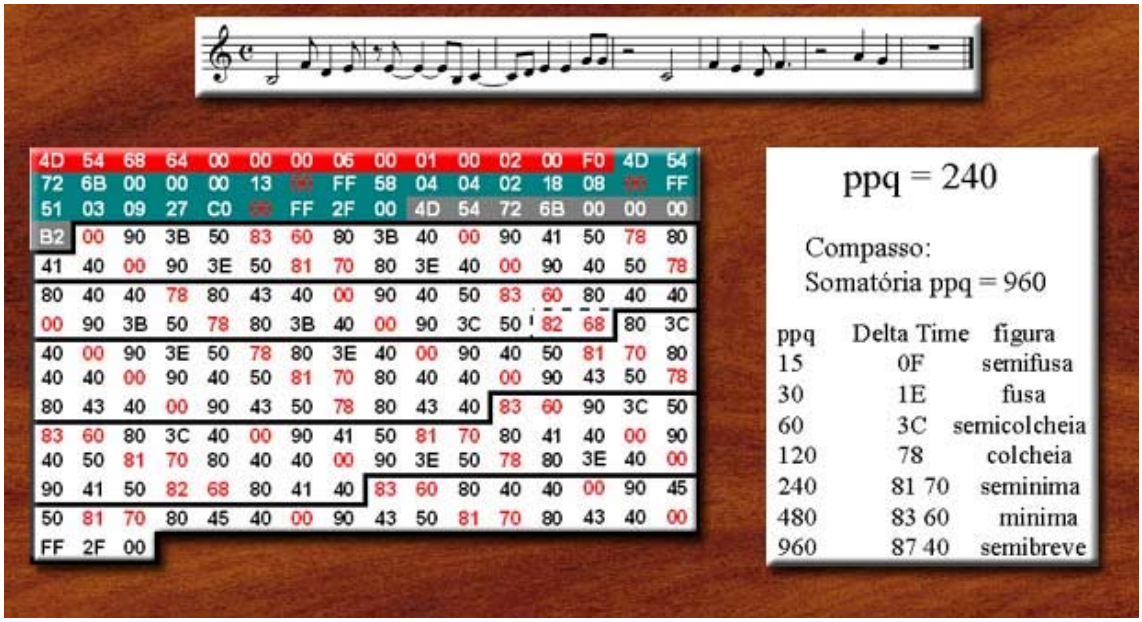


Figura 2.25 – Exemplo de um arquivo MIDI

A seguir será discriminado o significado de cada byte do arquivo SMF apresentado acima.

4D 54 68 64	MThd → significa que é um arquivo MIDI padrão (SMF)
00 00 00 06	06 (hexadecimal) = 6(decimal) → significa que virão 6 bytes até o final do cabeçalho.
00 01	Tipo de formato → neste caso formato 1
00 02	Número de Tracks → neste caso 2 Tracks
00 F0	Valor da PPQ → neste caso F0(hexadecimal) = 240 (decimal)
4D 54 72 6B	MTrk → significa início de uma trilha (Track)
00 00 00 13	Número de bytes do Track → neste caso 13(hexadeciaml) = 19(decimal) bytes.
00	DeltaTime= 0. Executa a próxima mensagem imediatamente.
FF 58 04	"FF" → Status de Meta Evento "58" → Do tipo Fórmula de compasso "04"→ Indica o tamanho dos dados deste Meta Evento

04 02 18 08	Dados da fórmula de compasso "04" → 4 figuras por compasso (numerador da formula de compasso. "02" → tipo de figura que representa a unidade de tempo (denominador da formula de compasso). Neste caso semínima pois o valor é 4 = ($2^{"02"}$). Os dois Bytes que seguem podem atualmente ter qualquer valor entre 0 e 127(7F), já que as máquinas MIDI atuais não necessitam mais destes dados. "18" → resolução: especifica o número de pulsos MIDI por tempo. "08" → Especifica o número de fusas em um compasso**.**
00	DeltaTime= 0. Executa a próxima mensagem imediatamente.
FF 51 03	"FF" → Status de Meta Evento "51" → Do tipo Tempo do Metrônomo (Set Tempo) "03"→ Indica o tamanho dos dados deste Meta Evento
09 27 C0	"09 27 C0"(hexadecimal) → 0927C0(hexadecimal) = 600.000(decimal) microsegundos (0,6 segundo).
00	DeltaTime= 0. Executa a próxima mensagem imediatamente.
FF 2F 00	"FF" → Status de Meta Evento "2F" → Do tipo Fim de Track "00"→ Indica o tamanho dos dados deste Meta Evento
4D 54 72 6B	Mtrk → início de Track
00 00 00 B2	Número de bytes do Track → neste caso B2(hexadeciaml) = 178(decimal) bytes.
00	DeltaTime= 0. Executa a próxima mensagem imediatamente.
90 3B 50	"90" → ativar nota (9) no canal 0 (0) "3B" → nota si4 "50" → volume 80(decimal) (pode ser qualquer valor entre 0 e 127
83 60	DeltaTime = 480(decimal). A máquina MIDI espera o tempo de uma mínima para executar o próxima mensagem

80 | 3B | 40	"80" → desativar nota (8) no canal 0 (0) "3B" → nota si4 "40" → volume 64(decimal). Neste caso mudar este valor não causará nenhuma diferença.
00	DeltaTime= 0. Executa a próxima mensagem imediatamente.
90 | 41 | 50	"90" → ativar nota (9) no canal 0 (0) "41" → nota fa5 "50" → volume 80(decimal) (pode ser qualquer valor entre 0 e 127
78	DeltaTime = 120(decimal). A máquina MIDI espera o tempo de uma colcheia para executar o próxima mensagem
80 | 41 | 40	"80" → desativar nota (8) no canal 0 (0) "41" → nota fa5 "40" → volume 64(decimal). Neste caso mudar este valor não causará nenhuma diferença.
00	DeltaTime= 0. Executa a próxima mensagem imediatamente.
90 | 3E | 50	"90" → ativar nota (9) no canal 0 (0) "3E" → nota re5 "50" → volume 80(decimal) (pode ser qualquer valor entre 0 e 127
81 | 70	DeltaTime = 240(decimal). A máquina MIDI espera o tempo de uma semínima para executar o próxima mensagem
80 | 3E | 40	"80" → desativar nota (8) no canal 0 (0) "3E" → nota re5 "40" → volume 64(decimal). Neste caso mudar este valor não causará nenhuma diferença.
Ao longo deste Track só aparece eventos de ativar e desativar as notas sempre precedido de um Delta Time de 1, 2, 3 ou 4 bytes. Isto ocorre até encontar a mensagem de fim de Track mostrada a seguir:	
FF 2F 00	"FF" → Status de Meta Evento "2F" → Do tipo Fim de Track "00"→ Indica o tamanho dos dados deste Meta Evento

O estudo do MIDI é vasto e rico, com muitos detalhes não abordados pois não é objetivo deste capítulo esgotar o assunto e sim apresentar os conceitos e definições considerados suficientes para o bom entendimento das técnicas de implementação de um sistema computacional voltado para a criação e edição de arquivos MIDI SMF voltados especificamente para seqüências de violão, foco deste trabalho.

Capítulo 3

SoundFonts

Com a popularização do protocolo MIDI, a demanda por produtos voltados para produção musical baseada nesta tecnologia cresceu de forma acentuada. Os sintetizadores foram sendo aprimorados e os recursos expandidos. O processo de geração do som que era mais utilizado por estes instrumentos era a chamada síntese subtrativa.

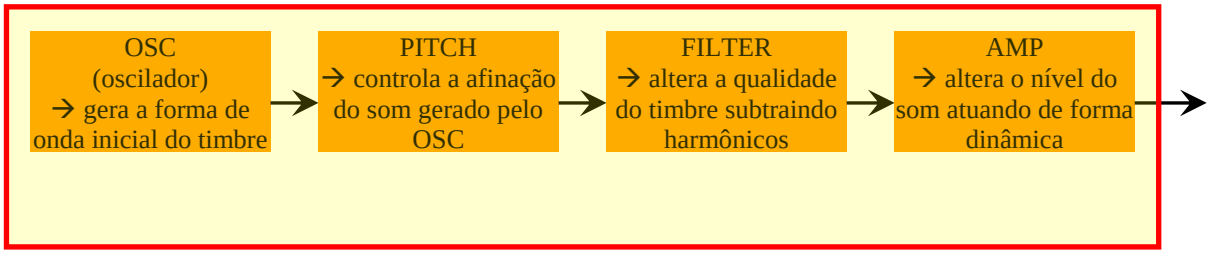


Figura 3.1 – Síntese subtrativa

Segundo RATTON (2004),

> *"O filtro tem um importante papel na síntese subtrativa, pois é ele quem molda efetivamente a composição harmônica do som resultante. Com um gerador de envoltória acoplado ao filtro, a atuação deste último pode ser alterada no decorrer do tempo, variando-se a freqüência de corte no ataque, decaimento e demais estágios do som, o que permite a produção de timbres cuja composição harmônica sofre alterações do início ao fim da execução da nota."*

Com o advento dos sintetizadores digitais a síntese subtrativa começou a ser substituída pela síntese por wavetable cuja a geração do som é produzida a partir de amostras digitais de sons naturais.

No caso das placas de som utilizadas em computadores, os sintetizadores MIDI já vêm incorporados, na maioria delas. Deste modo algumas empresas começaram a desenvolver tecnologias que explorassem os recursos da síntese por wavetable em seus equipamentos. A partir disto surgiu a tecnologia SoundFont. Ela foi desenvolvida pela empresa CREATIVE para ser utilizada em suas placas de som no início da década de 90. Em 1994, surgiu a placa “Sound Blaster AWE 32” , primeira a utilizar tal tecnologia. Em 1996 surgiu o formato SF2 (SoundFont 2.0) que substituiu o antigo formato SF1 (SoundFonts 1.0) e passou a ser incorporado em todas as placas da CREATIVE (Sound Blaster modelos AWE 64, PCI 128, Live!, Audigy e Audigy 2) e outros fabricantes começaram a incorporar em suas placas o suporte ao formato SF2 tais como Turtle Beach e a TerraTec. Alguns softwares de edição de audio e MIDI passaram a trabalhar com SoundFonts como, por exemplo, o Cakewalk, o Cubase e o Logic Audio.

SoundFonts são, deste modo, bancos de timbres que podem ser gravados diretamente de instrumentos acústicos, editados e armazenados na memória das placas de som que suportam esta tecnologia ou armazenados no disco rígido para serem manipulados por softwares e instrumentos virtuais. Eles funcionam como “fontes de Sons” tendo uma operacionabilidade semelhante às das fontes de textos. Veja o exemplo a seguir:

	Tipo de fonte de texto
Esta frase é um exemplo	→ Comic Sans MS
ESTA FRASE É UM EXEMPLO	→ Castellar

O conteúdo destas frases é o mesmo, o que muda é apenas a visualização. Para tornar o conteúdo destas frases visíveis o sistema precisa utilizar alguma fonte de texto instalada. É nestas fontes que estão armazenados os tipos de figuras ou “desenhos” que cada letra assumirá na visualização do conteúdo do texto. Os SoundFonts funcionam de forma semelhante, porém voltados para produção sonora e não para visualização. É neles que estão armazenados os timbres que serão utilizados pelo sistema para audição do conteúdo musical contido no arquivo MIDI. Os dados contidos no arquivo

SoundFont são as amostras de audio digitalizadas e as instruções para síntese, tais como características de filtros, loop, parâmetros do envelope de volume tais como tempos de ataque, sustain, release, entre outros.

3.1 – Criação e edição de SoundFonts

Uma das etapas deste trabalho foi criar SoundFonts a partir de amostras de áudio digitalizado com qualidade profissional. A captação e edição foi feita em estúdio com equipamentos de alta definição sonora e utilizando violões profissionais. Os procedimentos que serão aqui apresentados valem para criação de qualquer instrumento no formato SoundFont.

O software utilizado para criar e editar estes arquivos foi o VIENNA SOUNDFONT STUDIO 2.3 desenvolvido pela CREATIVE, mesma empresa que criou os SoundFonts. Este aplicativo é gratuito e pode ser baixado da Internet no endereço http://www.soundblaster.com/soundfont/downloads.asp . Com ele pode-se manipular todos os parâmetros necessários para síntese sonora a partir das amostras de áudio.

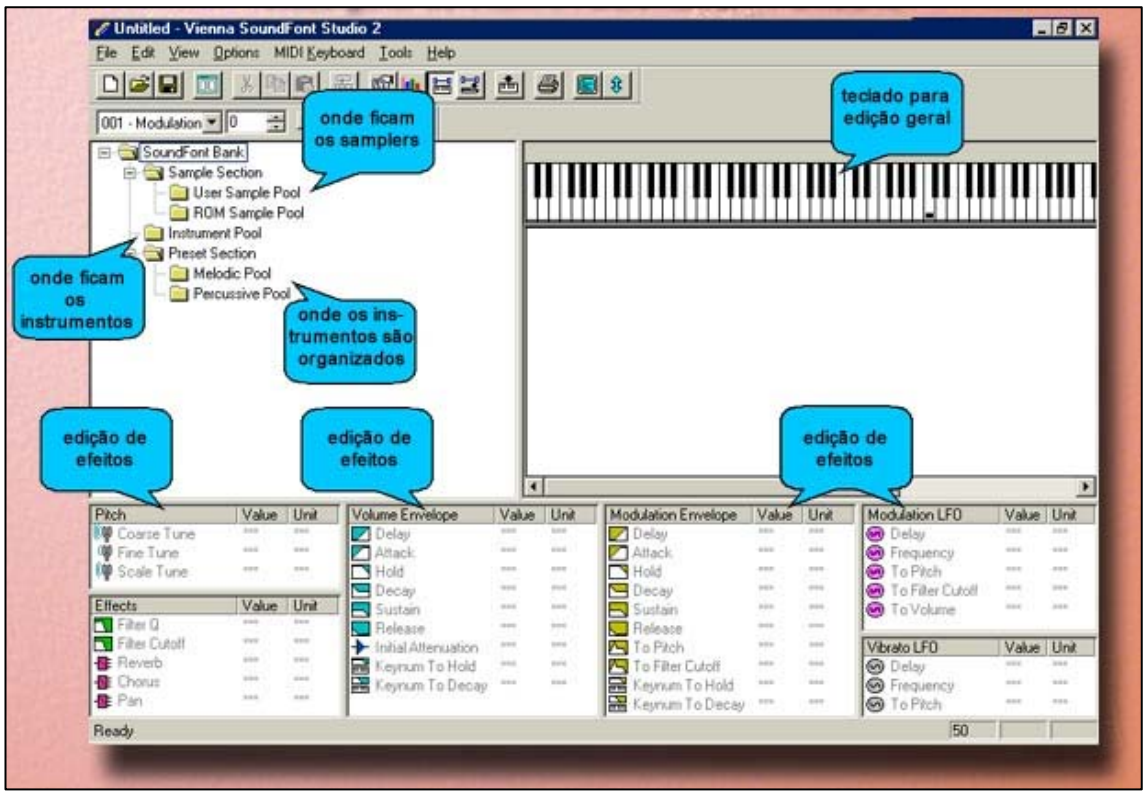


Figura 3.2 – Visão geral do software Vienna SoundFont Studio 2.3

A visualização do arquivo através deste software é organizada por pastas onde ficam armazenadas as amostras e os instrumentos criados a partir destas amostras.

O primeiro passo para criar um SoundFont é importar estas amostras que devem estar em formato WAVE (*.wav). Elas ficam armazenadas na pasta *"User Sample Pool"*. Para importá-las basta clicar com o botão direito do mouse sobre ela e selecionar os arquivos de audio.

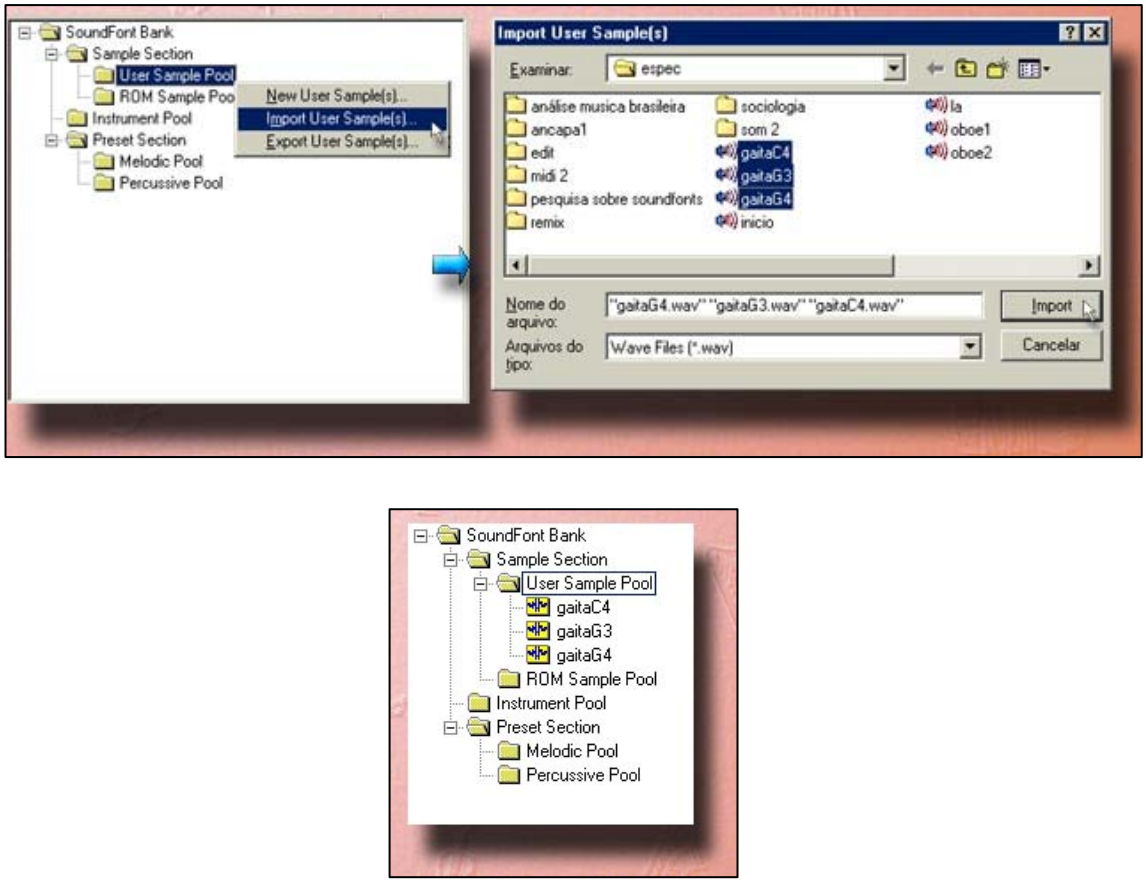


Figura 3.3 – Importando as amostras de audio

Um dos parâmetros a serem editados após as amostras serem importadas é definir a região de Loop. Ela é necessária no caso do sintetizador receber uma mensagem de ativar nota com duração superior a duração da amostra de audio. Quando isto ocorre o sintetizador entra na região do Loop e reproduz a mesma continuamente até receber a mensagem de desativar a nota. Isto faz com que não seja necessário utilizar amostras longas (com mais de 5 segundos), pois assim os arquivos SoundFonts ficam mais compactos e eficientes.

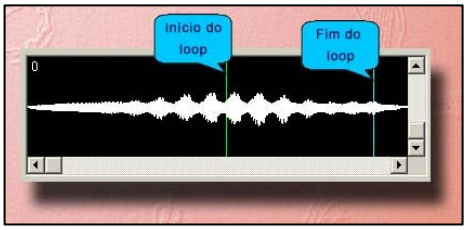


Figura 3.4 – Definindo a região de Loop

A pasta *"Instrument Pool"* é onde ficam os instrumentos criados a partir das amostras de audio. O SoundFont permite criar até 128 instrumentos em cada banco, sendo que são disponibilizados 128 bancos. Deste modo pode-se criar 16.384 instrumentos diferentes em apenas um SoundFont. Para isto basta utilizar o botão direito do mouse nesta pasta, nomear o novo instrumento e escolher quais amostras de audio armazenadas na pasta *"User Sample Pool"* serão utilizadas.

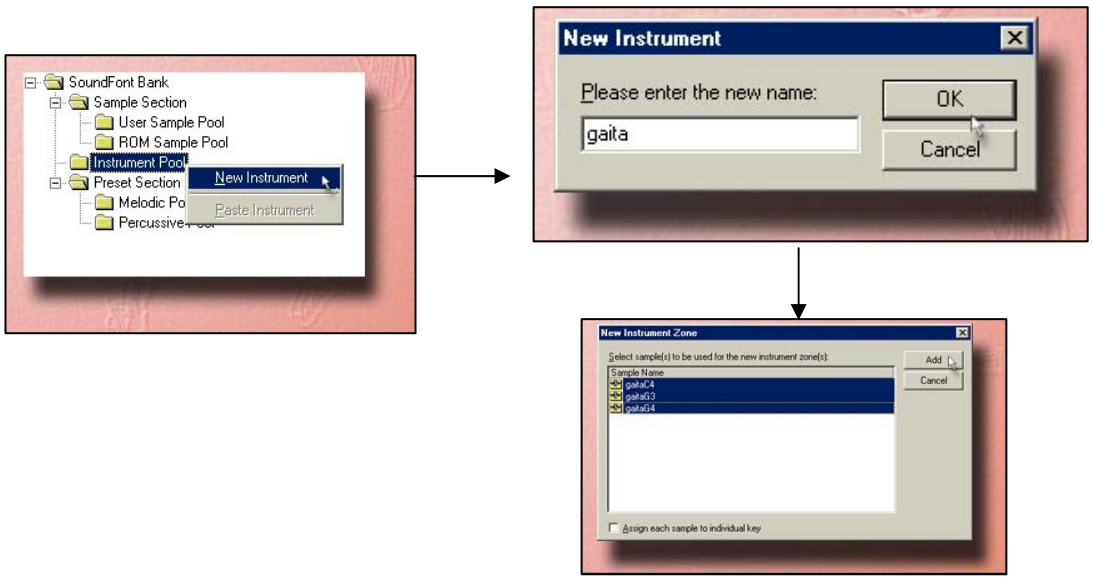


Figura 3.5 – Criando um novo instrumento

Cada uma destas amostras utilizadas na criação do novo instrumento precisa ser endereçada para a sua respectiva "Root Key", que é a altura original captada e armazenada nos arquivos de audio. Por exemplo, suponhamos que a amostra de audio contém o som de gaita tocando a nota "sol 5"[1], o "Root Key" desta amostra deve ser configurado para a nota "sol 5". É a partir desta "Root Key" que é feita a síntese das outras notas.

[1] De acordo com as especificações MIDI o sol 5 está localizado logo acima da nota dó central (do 5).

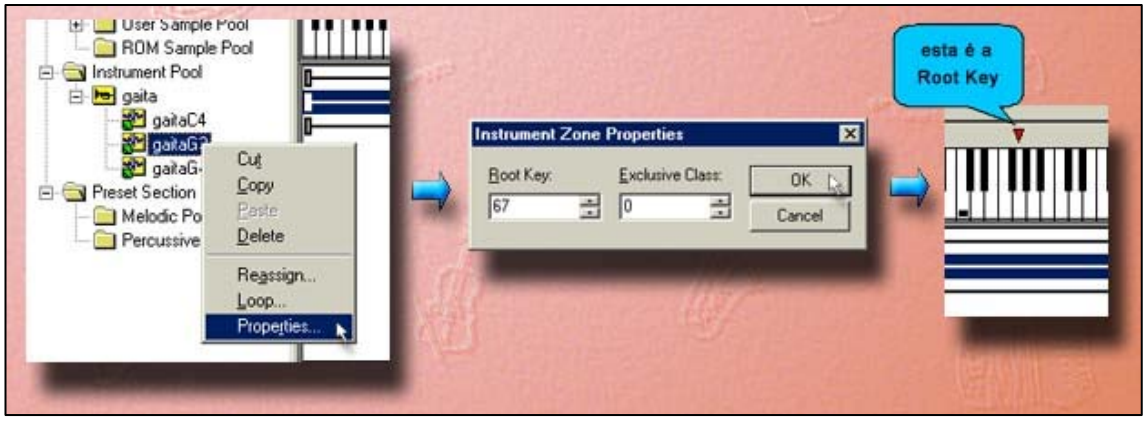


Figura 3.6 – Configuração do "Root Key"

É preciso definir, também, o conjunto de notas que serão sintetizadas por cada amostra. A isto denominou-se "Range".

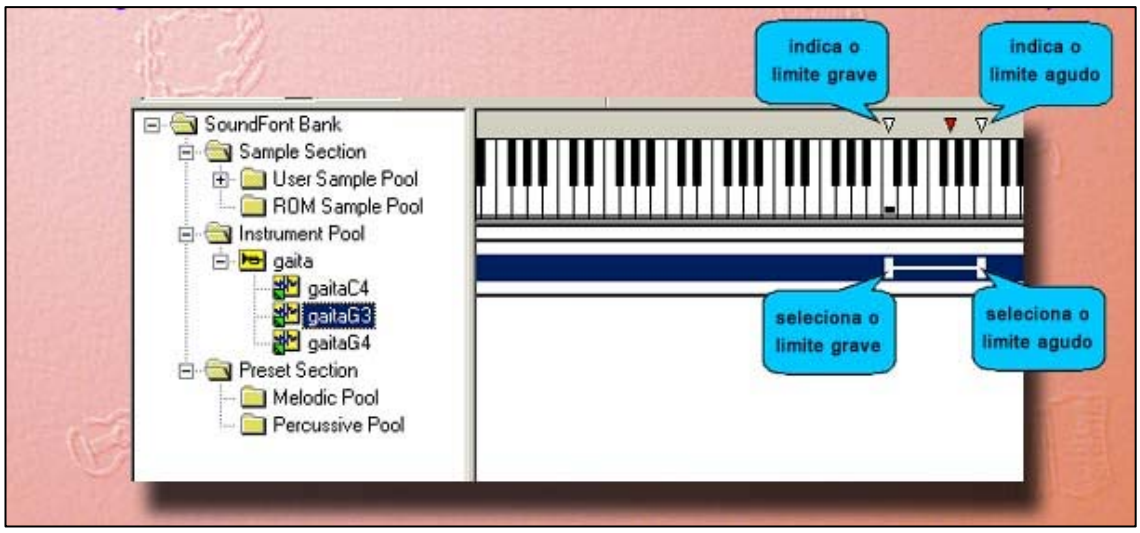


Figura 3.7 – Configuração do "Root Key"

Depois de criado o instrumento é preciso definir qual será o seu número, que pode variar de 0 (zero) até 127, e em qual banco estará localizado, que pode variar de 0 (zero) até 127 também. Geralmente quando um arquivo MIDI não manda nenhuma mensagem para o sintetizador especificando o número do banco, ele executa os timbres do primeiro banco que é o 0 (zero). É na pasta *"Melodic Pool"* contida na pasta *"Preset Section"* que são criadas estas definições. Ao selecionar "New Melodic Preset" o programa disponibiliza uma janela onde é definido o número do instrumento, o banco

ao qual ele pertencerá e o nome do mesmo. Em seguida é pedido para escolher qual dos instrumentos contidos na pasta “Instrument Pool” serão utilizadas estas configurações.

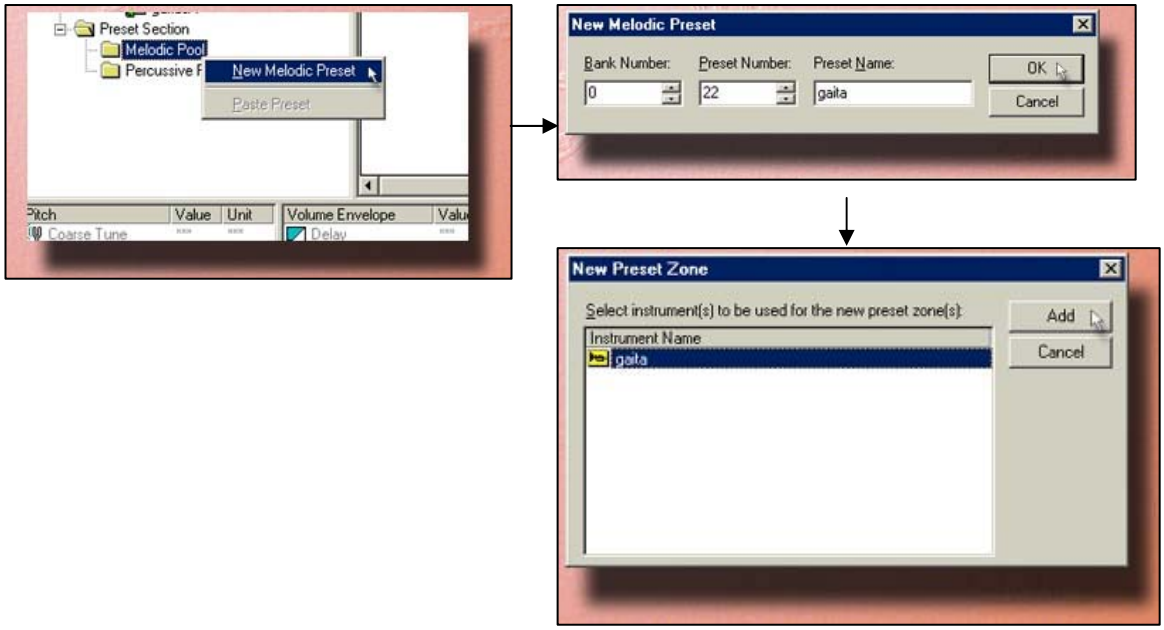


Figura 3.8 – Configuração do número correspondente ao instrumento.

É possivel modificar parâmetros do envelope de volume tais como Attack, Decay, Sustain, Release, além de poder acrescentar efeito como filtros de freqüências, vibratos, reverb, chorus, pan, entre outros.

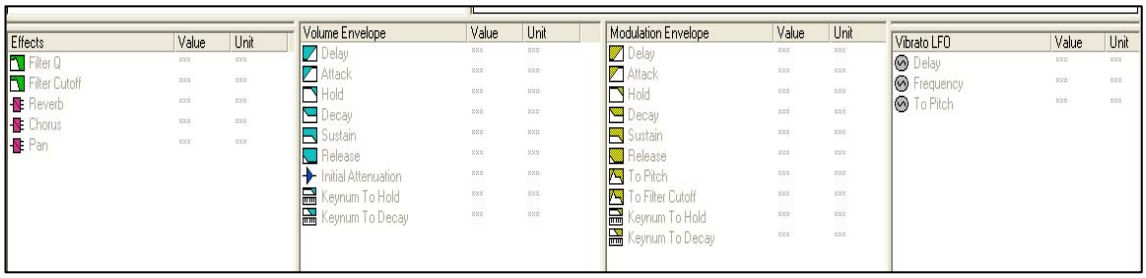


Figura 3.9 – Janela de edição de efeitos

Pesquisar formas de criar e editar SoundFonts foi essencial para o desenvolvimento deste trabalho, pois a partir destes procedimentos foram desenvolvidos os bancos de sons contendo os timbres e articulações humanizadas. Para maiores detalhes sobre o formato SoundFont basta consultar o site da CREATIVE http://www.soundblaster.com/soundfont/ , pois no mesmo é disponibilizado material ricamente detalhado.

3.2 – Renderizando SoundFonts com TiMidity++

Vários softwares foram desenvolvidos para trabalhar com SoundFonts, tanto em sistemas Windows quanto Linux. Um deles se mostrou bastante útil no desenvolvimento deste trabalho que é o TiMidity++. Este aplicativo de código aberto renderiza[2] arquivos MIDI padrão em arquivos WAVE utilizando os timbres no formato SoundFont. Os argumentos de entrada são inseridos através da linha de comando do Prompt do DOS. Ele é extremamente leve e eficiente, ideal para trabalhos que envolvam manipulação de arquivos sonoros.

Para renderizar um arquivo MIDI em WAVE utilizando o TiMidity++ basta entrar com o endereço (Path) do arquivo MIDI, alguns parâmetros de configurações e pronto. O arquivo SoundFont que será utilizado no processo de renderização é definido em um arquivo que se encontra na mesma pasta do executável, denominado "timidity.cfg". Ele é um arquivo texto contendo o endereço (Path) deste SoundFont. As opções dos parâmetros de entrada encontram-se em anexo. Veja exemplo abaixo.

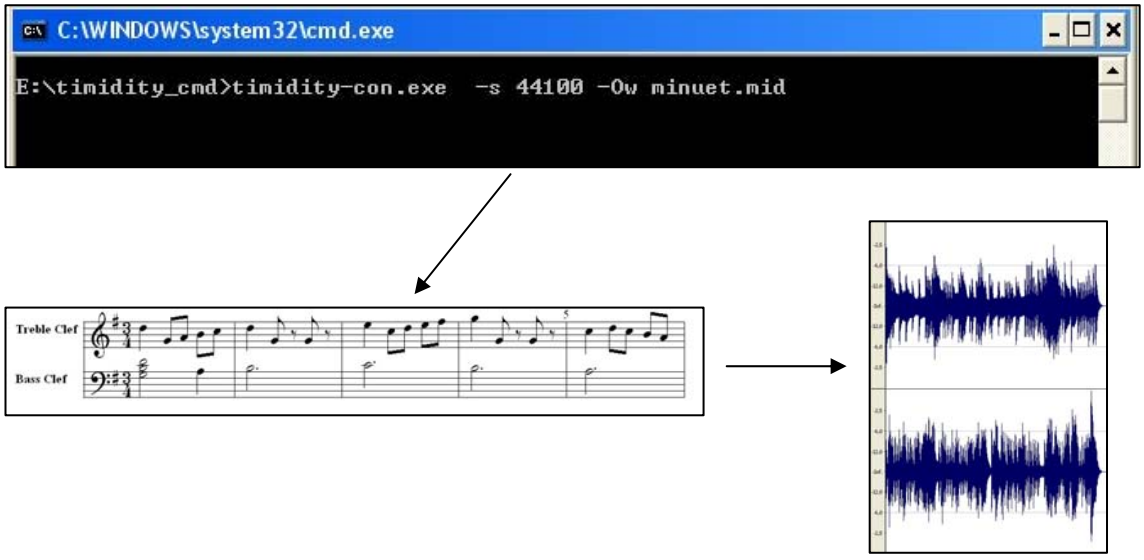


Figura 3.10 – Comando que converte o arquivo MIDI em WAVE com qualidade de CD

No exemplo acima o arquivo "minuet.mid", que aparece aberto em um editor de partitura, é convertido em um arquivo WAVE com taxa de amostragem de 44.100 Hz, 16 bits, stereo, que aparece aberto em um editor de audio.

[2] Renderização é um termo utilizado na computação gráfica que significa converter uma série de símbolos gráficos num arquivo visual. É empregado também como processo de converter um tipo de arquivo para outro, ou ainda "traduzir" de uma linguagem para outra. (http://pt.wikipedia.org/wiki/Renderizar)

Capítulo 4

O paradigma funcional e a linguagem Clean

Neste capítulo serão apresentados os recursos e potencialidades que a linguagem Clean oferece no desenvolvimento de aplicativos multimídias bem como os motivos que levaram o autor a escolher o paradigma funcional. Para facilitar o entendimento das ferramentas e dos conceitos a serem abordados foram criados pequenos aplicativos com algumas implementações básicas com o objetivo de exemplificar alguns recursos utilizados no desenvolvimento do editor MIDI para violão com articulação humanizada nota a nota, foco deste trabalho.

4.1 A Opção pela Linguagem Clean

Inicialmente alguns pontos básicos nortearam a escolha de uma linguagem de programação que fosse adequada ao desenvolvimento deste trabalho. Primeiramente a linguagem escolhida deveria ser a mais aderente no que se refere a manipulação dos elementos do domínio do música. Buscou-se também uma linguagem que gere aplicativos de fácil execução, sem depender da instalação conjunta de dlls e frameworks que exigissem constantes atualizações do código. Outro critério importante foi de optar por uma linguagem livre.

Sendo assim, dos vários paradigmas de programação existentes, a escolha recaiu no paradigma funcional, principalmente pela facilidade de se implementar problemas recursivos em listas e vetores utilizando funções relativamente complexas. Segundo

CAMARGO (2007) em seu trabalho *"Desenvolvimento de aplicativos MIDI em linguagem funcional CLEAN"*, esta linguagem é a mais adequada para implementação de aplicativos MIDI por possuir as seguintes características:

- Por ser de código aberto e gratuito (pode-se baixá-la em: http://clean.cs.ru.nl/);
- Por possuir alto nível de abstração na implementação de funções, evitando algoritmos complexos para modelagem das funções;
- Por possuir funções matemáticas de alta ordem, como ***map*** e ***fold***, que permitem aplicar uma função a um domínio completo e complexo de dados;
- Por aceitar funções como parâmetro de outras funções, fatos comuns nas abstrações em música;
- Por possuir implementação de notação Zermelo-Fraenkel (http://en.wikipedia.org/wiki/Zermelo%E2%80%93Fraenkel_set_theory), a qual permite se definir o conjunto imagem diretamente da especificação da função e de seu domínio;
- Por possuir transparência referencial, ou seja, o resultado de uma função, com mesmos argumentos, sempre dará o mesmo resultado, independendo do contexto;
- Por minimizar efeitos colaterais, mesmo em interfaces gráficas (utilizando a técnica de Tipos Únicos). Nesta técnica, tipos únicos, para se evitar a perda da transparência referencial, quando uma variável, um arquivo ou outro registro do sistema estiver sendo utilizado, o mesmo é indisponibilizado a qualquer usuário ou sistema, até que uma nova instância do mesmo seja disponibilizada e liberada por quem iniciou sua manipulação. Assim, pode-se até fazer avaliações destrutivas nesta etapa, já que ela não será vista por mais ninguém além do processo que a está manipulando;
- Por ser fortemente tipada, evitando que programas com problemas de tipo sejam compilados e, desta forma, não transferindo tal erro para avaliação pelo usuário. Assim, da mesma forma com que a matemática procede, uma função apenas manipula argumentos de mesmo tipo de dado. Estruturas computacionais, como listas e vetores, também só poderão ter elementos de mesmo tipo, podendo-se, desta forma, utilizá-la como domínio e conjuntos de dados a serem manipulados com segurança pelas funções desejadas;

- Por não aceitar avaliação destrutiva de variáveis[1]. Desta forma, evita-se que o programador tenha que monitorar todos os processos que utilizem a mesma variável, permitindo que se possa modularizar projetos com vários programadores e reaproveitar código de outros programas. Assim, uma vez que a variável é instanciada, recebe o valor, o mesmo se mantém durante todo o processamento do aplicativo;
- Por possuir avaliação lazy[2], e, quando desejado, avaliação eager[3]. Com a avaliação lazy, o programa apenas avalia uma expressão ou variável quando for necessário. Assim, por exemplo, pode-se conhecer o primeiro elemento de uma lista ou de um vetor sem que se tenha de conhecê-lo como um todo, agilizando a execução do programa. Caso se deseje utilizar avaliação eager, como no caso das linguagens procedurais, o Clean também disponibiliza tal recurso. Em alguns casos é até desejado que isto seja possível, como, no caso, de se querer validar toda uma lista de dados antes de se iniciar seu processamento, sem que seja necessário fazer uma função para tal procedimento;
- Por possuir coleta automática de lixo, o que, mesmo para programadores experientes, é uma tarefa difícil e trabalhosa de ser implementada com eficiência. O ato de utilizar e liberar memória durante o processamento de um aplicativo é uma tarefa que demanda um bom conhecimento de software e do hardware (da máquina) onde o sistema irá rodar;
- Por ser polimórfica[4] , permitindo uma mesma função, em diferentes contextos, seja aplicada de forma diferente aos dados recebidos. Esta característica, em música é relevante. Música, como toda arte, é fortemente dependente de contexto;
- Por permitir a utilização de códigos gerados por outras linguagens (obj, dll), como, por exemplo: C. Assim, evita-se ter que desenvolver programas já

[1] A avaliação destrutiva só é aceita em interfaces gráficas, onde é necessária, mas tutelada pela técnica de tipos únicos que evita que tal avaliação cause efeitos colaterais e a perda da transparência referencial.
[2] Lazy – do inglês: preguiçosa. Um termo que, traduzido, não expressa a intenção. Laze, no caso, significa que a linguagem não avalia os dados ou funções enquanto não for necessário. Isto não é preguiça, é eficiência. (http://en.wikipedia.org/wiki/Lazy_evaluation)
[3] Eager – do inglês – ávida, desejada. No caso, o termo seria melhor a tradução precoce, prévia. A linguagem avalia todos os dados antes de manipulá-los, mesmo que a maioria deles não venha a ser utilizada. (http://en.wikipedia.org/wiki/Eager_evaluation)
[4] Não confundir polimorfismo com sobrecarga de operadores. Na sobrecarga apenas se define como um operador ou função vai manipular um determinado tipo de dado. A sobrecarga não se preocupa com o contexto. Uma função polimórfica, por trabalhar com dados de tipos diferentes, necessita que os operadores utilizados por ela sejam sobrecarregados para os tipos utilizados.

eficientes e consagrados por outras linguagens, principalmente quando se tem que desenvolver *drivers* para controle de placas e programas para manipular portas de dados, os quais demandam programação em baixo nível de abstração, como no caso de enviar e ler dados da placa de som;

- Por ser, dentre todas as linguagens e paradigmas, uma das quatro linguagens mais eficientes e rápidas em quase todos os tipos de aplicações, usando vários *benchmarks* conhecidos e consagrados. No paradigma funcional escolhido, o Clean é a que possui melhor *benchmark*, superando *Eiffel, Haskell* e mesmo o Lisp (http://shootout.alioth.debian.org/gp4/benchmark.php?test=all&lang=all);
- Por possuir código legível e limpo, mesmo em interfaces gráficas e visuais:

A seguir, serão apresentados alguns conceitos e ferramentas utilizados nesta linguagem, de forma a permitir que interessados no desenvolvimento de aplicativos utilizando esta linguagem possam ter uma visão geral da mesma e de suas potencialidades.

4.2 – Tipos de dados

Algumas linguagens podem acarretar efeitos colaterais indesejáveis e imprevisíveis nos programas implementados. As linguagens procedurais são aquelas que mais efeitos colaterais produzem. CLEAN foi criado de tal forma a procurar evitá-los.

Na computação os conjuntos não são infinitos, existe uma quantidade máxima de elementos que se pode armazenar na memória de um computador. Por outro lado, a forma de se armazenar um tipo de dado é completamente diferente da forma de se armazenar os outros. Por exemplo, a forma de armazenar uma variável declarada como do tipo “Inteiro” é diferente da forma como é armazenada uma variável declarada como do tipo “Real”. Sendo assim se tentássemos somar inteiros com reais, como o resultado seria armazenado: como real ou como inteiro? Desde modo a linguagem Funcional CLEAN, assim como as linguagens tipadas modernas, não permite que, mesmo por distração, o programador opere com tipos de dados diferentes.

A seguir serão apresentados alguns tipos de dados primitivos presentes no CLEAN.

● **Inteiro** → Números pertencentes ao conjunto dos inteiros. Este tipo é referenciado no CLEAN como **Int.** Exemplo: 2

● **Real** → Número pertencente ao conjunto dos Reais. Este tipo é referenciado no CLEAN como **Real.** Exemplo: 2.1

● **Caractere** → É um símbolo de um alfabeto. No caso do computador o alfabeto utilizado é o ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange). Este tipo é referenciado no CLEAN como **Char.** Ele é representado por um símbolo colocado entre apóstrofos. Exemplo: '2' ou 'c'

● **String** → É uma cadeia de caracteres colocados entre aspas. Exemplos: "carro" , "Nome", "caixa de som", etc. Uma String é um vetor de caracteres que só possui sentido se lido de uma vez só. Este tipo é referenciado no CLEAN como **{#Char}.**

● **Vetor** → Um vetor possui elementos de mesmo tipo de dados delimitados por chaves e separados por vírgula. Exemplos: **vetor de inteiros** →{1,2,3,4} ou **vetor de caracteres** →{'c', 'a', 's' ,'a'} ou **vetor de String**→{"bicicleta", "carro", "Antonio"}, etc.

● **Matriz** → A matriz é um vetor de vetores. Exemplo: **matriz de vetores de inteiros** →{{1,2,3,4}, {10,20}, {4,8,9}}. O tipo de dados desta matriz é referenciado no Clean como **{{Int}}**

● **Lista** → Uma lista possui elementos de mesmo tipo de dados delimitados por colchetes e separados por vírgula. Exemplo: **[4,5,8,12] = lista de inteiros** ou **['v', 'a', 'i'] = lista de caracteres**.

A declaração de tipo de uma lista é dada pelo tipo de seus elementos colocado entre colchetes. Exemplo:

- a lista **[0,1,2,3]** possui o tipo **[Int]**,
- a lista **['a', 'b']** possui o tipo **[Char]**,
- a lista **["tecla", "palavra " , "vida"]** possui o tipo **[{#Char}]**,
- a lista **[[12,22],[6],[4,5,9]]** possui o tipo **[[Int]].**

Uma lista possui um número variado de elementos, ou seja, você pode eliminar um elemento de uma lista ou mesmo acrescentar.

Uma lista em CLEAN possui a seguinte estrutura: **[c:r],** onde **c** é a cabeça da lista (o primeiro elemento dela) e **r** o resto da lista (sua calda, ou seja, a lista sem a cabeça). Uma lista vazia é representada por **[ ]**, ou seja, o abrir e fechar de colchetes com tantos espaços em branco internamente quanto se deseje.

● **Tuplas** → É uma coleção de elementos separados por vírgula. Tal coleção é delimitada por parênteses. Ela é muito útil para agrupar informações de diferentes tipos. Exemplo:

- (‘c`,50 ,“carro”) – uma tupla formada por elementos de três diferentes tipos de dados. É referenciada no Clean da seguinte forma **(Char , Int , {#Char})**
- ([1, 2, 3,4], [10, 20], [0,1,2]) – uma tupla formada por três listas. É referenciada no Clean da seguinte forma **([Int] , [Int] , [Int])**

Muitas vezes você pode desejar utilizar um tipo especial de dados que a linguagem não possui. Como já foi dito, se utilizarmos as primitivas do CLEAN em tipos de dados diferentes (a menos das tuplas) o sistema acusa um erro de tipo. Muitas vezes é interessante que trabalhemos dados com tipos diferentes e não queremos trabalhá-los como tuplas, já que, se trabalhados como listas teríamos um número bem maior de funções primitivas para implementar nossos programas. Sendo assim, podemos criar nossos próprios tipos de forma a poder manipulá-los utilizando a maioria das primitivas que o CLEAN possui. Para isto temos o Construtor de Tipo. Não iremos abordar como utilizar o construtor de tipo no corpo desta dissertação. No CD fornecido com este trabalho existe um vasto material onde o leitor pode conhecer melhor este recurso e possa aprofundar no que se refere as ferramentas oferecidas pela linguagem CLEAN.

4.3 – Funções em Clean

Criar e manipular funções em CLEAN é algo rápido e direto, geralmente com códigos limpos e enxutos. Basta digitar o nome da função desejada, seus argumentos e o que deseja que a mesma faça.Veja um exemplo de criação de uma função em CLEAN que soma o quadrado (x^2) com o cubo (x^3) de um número inteiro “x”.

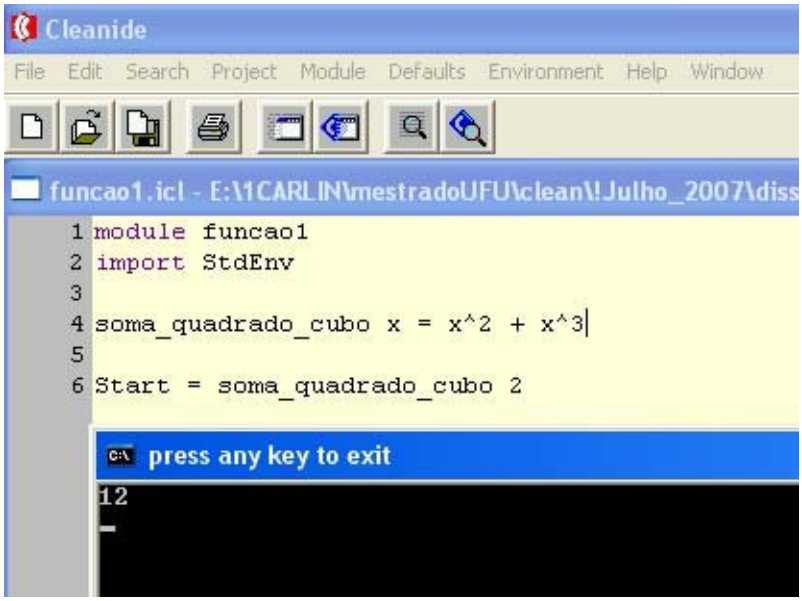


Figura 4.1 – exemplo de uma Função em CLEAN

Observe que a sintaxe da função foi primeiro escrever o nome da mesma, neste caso "soma_quadrado_cubo", depois a variável de entrada, neste caso "x", o símbolo de "=" e posteriormente o que a função deve fazer que é somar o quadrado com o cubo da variável de entrada "x". Ao chamar a função com argumento de entrada igual a 2 (dois), a mesma retorna o resultado = 12

A função "Start" é a rotina padrão do CLEAN para iniciar o programa. Sempre que o mesmo é executado, busca-se a função Start para iniciar o processo.

Abordaremos a criação das funções com mais detalhes à medida que forem apresentadas as técnicas de implementação de alguns aplicativo que servirão de exemplos para melhor compreensão deste trabalho.

4.4 – Notação Zermelo-Fraenkel

Esta é uma das ferramentas mais eficientes desta linguagem, amplamente utilizada no corpo deste trabalho e extremamente aderente na manipulação dos objetos do domínio musical. Além de permitir a criação de listas de elementos gerados por funções extremamente complexas, de uma forma clara e simples, o código gerado é compacto e veloz.

Esta notação, proposta por Ernest Zermelo, um matemático alemão, e por Adolf Frankel, um lógico Israelita, é um recurso, uma ferramenta matemática poderosa, veloz e eficiente na arte de computar valores de uma função. A notação Zermelo-Frankel formaliza, explicita a lei de formação de um conjunto imagem, dado o domínio da função e as regras de inferência da mesma.

Na teoria de conjuntos a seguinte notação é comumente utilizada na definição de conjuntos:

$$\mathbf{S = \{x^3 \mid x \in N\}}$$

Esta expressão deve ser lida da seguinte forma: S é o conjunto de elementos de todos os cubos de x (x elevado a 3) tal que x pertence ao conjunto dos números naturais (N). Em CLEAN esta notação é formalizada da seguinte forma:

[função \\ domínio | restrições (condições) do domínio]

Veja o exemplo abaixo:

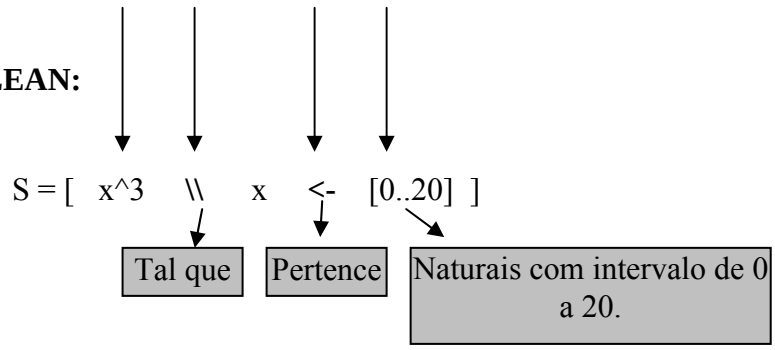
Domínio: {x ∈ **N**}

Função : f(x) = x^3

Notação Matemática da formalização do conjunto imagem:

$$S = \{ \quad x^3 \quad | \quad x \quad \in \quad \mathbf{N} \quad \}$$

Implementação em CLEAN:



Obs. O símbolo ^ é o operador de potenciação utilizado pelo CLEAN.

Para exemplificar esta notação, observe o seguinte exemplo:

- Mostrar o conjunto dos quadrados dos inteiros entre 1 e 5.

Da matemática tem-se : $S = \{ x^2 \mid 1 \leq x \leq 5 \}$

Em CLEAN tem-se : **S = [x^x \\ x <- [1..5]]**

A implementação no CLEAN é mostrada a seguir.

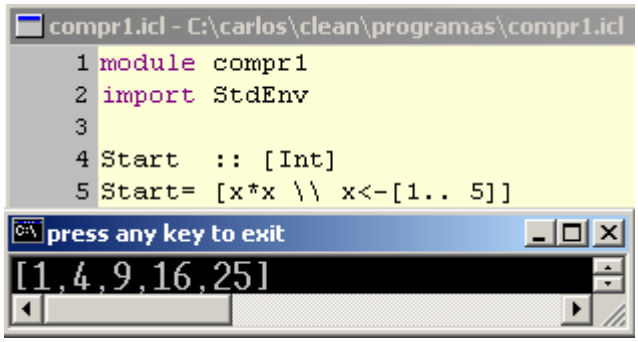


Figura 4.2 - Quadrado dos inteiros de 1 a 5

Obs. :: **[Int]** declara o tipo do dado devolvido pela função Start, ou seja: uma **lista de inteiros**.

Em CLEAN, a notação pode ser implementada:

1- Para listas: **[expressão]**

2- Para vetores: **{ expressão }**

A única diferença é que para vetores a notação se delimita com chaves e para listas com colchetes.

A seguir temos outro exemplo extraído do trabalho de LIMA (2006) que mostra de uma maneira bastante clara como implementar esta notação na linguagem CLEAN.

Exemplo:

Implementar a Notação Zermelo-Frankel em CLEAN de tal forma que a mesma gere o conjunto imagem formado pela lista de todos os números naturais pares de 0 a 9.

Domínio: $\{x \in \mathbf{N} \mid 0 \leq x \leq 9 \wedge x \text{ é par}\}$

ou seja-> **{2,4,6,8}**

Função: f(x) = x

ou regra -> o valor de x será igual ao do argumento, sem qualquer modificação.

A pergunta que deveria ser suscitada agora é:

- Como montar corretamente a notação para que tal lista seja gerada?

Observe como fazer isto passo a passo.

1- Primeiro, o que se deseja é uma lista, assim, inicia-se colocando os colchetes que delimitam uma lista:

[ ]

2- O próximo passo é colocar o separador entre a função e o domínio utilizados por tal notação:

[\\ ]

Função ⇦ ⇨ **Domínio e restrições**

3- Quando a função possuir apenas uma regra, coloca-se a regra do lado esquerdo do separador \\.

- Se a mesma possuir mais de uma regra, coloca-se do lado esquerdo do separador a função com seu(s) argumento(s).
- No caso, a função é **f(x)=x**, possuindo apenas uma regra: **x**, ou seja, o valor inferido é igual ao argumento da função.
- Neste caso, coloca-se uma letra ou palavra qualquer como regra, desde que a mesma comece com uma letra minúscula (exigência da linguagem CLEAN).
- Feito isto, obtém-se:

[x \\ ]

4- O próximo passo, é colocar o domínio sem suas restrições (mesmo que elas existam).

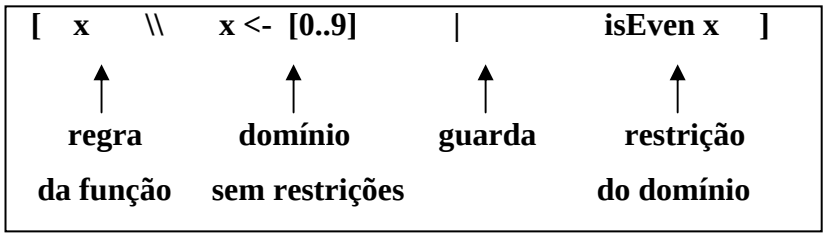
- O domínio é: **x pertence ao conjunto dos números naturais, tal que x é maior ou igual a 0 e menor ou igual a 9** (**$\{x \in \mathbf{N}, 0 \leq x \leq 9\}$**).

- Assim, como o teclado do computador não possui o símbolo de pertence (∈) nem o do conjunto dos naturais (**N**), os mesmos são substituídos no CLEAN por <- e **[0..9]** respectivamente, onde [0..9] corresponde à lista com todos os naturais de 0 a 9.
- Esta lista ([0..9]) é equivalente a **[0,1,2,3,4,5,6,7,8,9]**.
- A notação em CLEAN fica:

[x \\ x<-[0..9]]

5- Se o que se deseja fosse apenas isto, a lista produzida seria uma cópia do domínio, ou seja, uma lista de elementos x, tal que cada elemento da imagem é igual a um elemento do domínio.

- No exercício proposto, o domínio possui uma regra de restrição:
  - O elemento do domínio **x**, argumento da função, tem que **ser par.**
- Para iniciar uma regra no CLEAN, deve-se colocar uma **guarda**, uma barra vertical, após a descrição geral do domínio.
- Logo após esta guarda (|) coloca-se a(s) restrição(ões).
- A restrição, neste caso, é que só devem ser considerados pela função os argumentos pares do domínio.
- Uma restrição é reconhecida quando a mesma necessita de aplicação de alguma (ou mais de uma) função aos elementos do domínio.
- Neste presente caso, deve-se aplicar uma função que verifica cada argumento do domínio se o mesmo é **par**.
- Se for, a regra da função é aplicada ao mesmo e o valor é inferido. Se não for, o elemento do domínio é recusado e nele não é aplicada a regra de inferência da função em questão que inferirá os valores do conjunto imagem.
- Em CLEAN, existe uma função que testa se um inteiro é par: **isEven**.
- A notação Zermelo-Frankel para listas fica, finalmente, da seguinte forma:



ou seja:

[x \\ x <- [0..9] | isEven x]

a lista gerada por esta notação é:

[0, 2, 4, 6, 8]

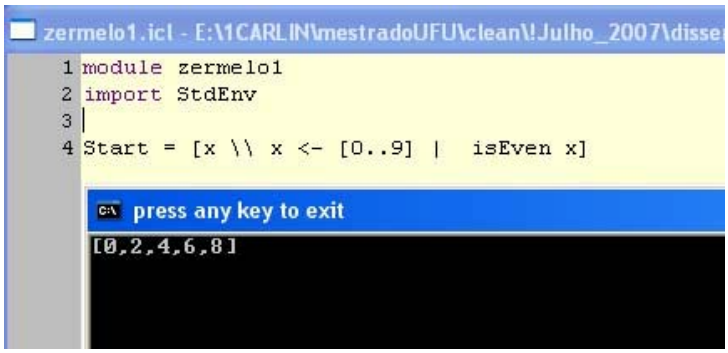


figura 4.3 – Expressão Zermelo-Frankel com o resultado.

4.5 – Técnica de Tipos Únicos

O CLEAN oferece uma biblioteca voltada para manipulação de entradas e saídas denominada de ***StdIO*** e encontra-se nos diretórios padrões onde o CLEAN está instalado. Manipular arquivos é tarefa que não possui transparência referencial e que necessita realizar avaliações destrutivas em dados, memória e arquivos. Para tanto, o CLEAN utiliza a técnica de **Tipos Únicos** para evitar efeitos colaterais indesejados. Segundo CAMARGO (2007).

> *"Nesta técnica, Tipos Únicos, a linguagem, antes de começar a trabalhar com arquivos e processos assume o status do computador com seu conteúdo e passa a monitorar todas as modificações de IO que ocorrem no sistema, impedindo que mais de um processo ou função manipule o mesmo dado. Somente após o dado, arquivo ou qualquer outro elemento global ser liberado pela função ou processo que o estava utilizando é que o mesmo, alterado ou não, poderá ser manipulado por outro processo, por outra função. Assim, uma vez referenciados, um*

registro, uma variável, memória, arquivos, etc., os mesmos passam a ficar indisponíveis. Desta forma, para iniciar um processo que utiliza ***IO*** *(entrada e saída de dados), a sintaxe dos programas muda radicalmente. A função* ***Start*** *passa a ter um argumento, denominado de "mundo", ou seja, uma variável de controle das unidades de* ***IO*** *do computador".*

Assim, um programa que envolve ***IO*** inicia e termina como segue:

```
1  module testeMundo
2  import StdEnv,StdIO
3
4
5
6  Start world
7
8
9  =world
10
```

```
press any key to exit
65536
```

figura 4.4 - Listagem e resultado do programa testeMundo. A resposta devolve o valor 65536, que é o status atual de seu mundo, o qual, depois de "aberto" foi retornado sem nenhuma modificação em suas unidades de ***IO****.*

Caso neste processo fosse realizada uma ação de leitura de um arquivo qualquer, este arquivo seria monitorado por esta variável, alterando seu valor para um valor que traduza, para o compilador da linguagem, esta ocorrência. Desta forma, a variável ***world*** inicial seria alterada. Assim, como a variável de entrada e a de saída possuem o mesmo nome e conteúdos diferentes, ocorreria, aí, uma avaliação destrutiva, mas controlada.
Entre iniciar a função ***Start*** e seu fim, cada linha de programa deve ser precedida do símbolo " **#** ". O programa, com a técnica de Tipos Únicos, passa a tomar algumas características procedurais, tais como:

- O programa é executado linha a linha, top->down. Desta forma, se uma linha utiliza uma função ou variável, as mesmas deverão ter sido criadas anteriormente, ou seja, em linhas precedentes.

- As variáveis e os resultados das funções passam a ser globais no escopo da função ***Start***. As variáveis, o resultado fica do lado esquerdo da igualdade e a chamada da função do lado direito.

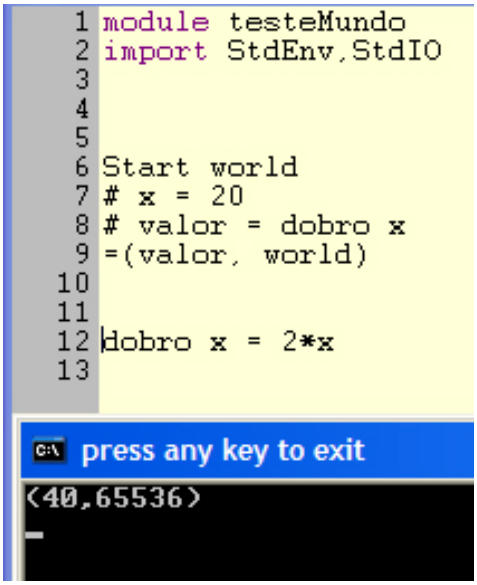


figura 4.5 - Listagem e resultado de um primeiro programa utilizando sintaxe de Tipos únicos

Ao longo da dissertação serão mostrados os procedimentos de abrir e salvar arquivos de dados, tarefas fundamentais para o desenvolvimento de aplicativos multimídia. Para tanto, implementou-se uma biblioteca, a ***StdArqNovo***, a qual possui várias funções de manipulação e criação de arquivos e diretórios. A mesma é apresentada no CD em anexo, já locada no subdiretório "\BIBLIOTECAS MIDI_StdMIDI\StdArq" dentro do diretório onde o editorMidiViolão se encontra.

4.6 – Implementação de um conversor MIDI →WAVE

Com o objetivo de facilitar o entendimento dos conceitos e técnicas aplicadas no desenvolvimento deste trabalho, foram criados pequenos aplicativo focados em uma determinada tarefa que está incorporada no aplicativo final. A Implementação de um conversor MIDI → WAVE tem os seguintes objetivos:

- Apresentar os conceitos básico para o desenvolvimento de uma interface com botões e campo de texto no CLEAN

- Apresentar técnicas de renderizar arquivos SMF em WAVE utilizando bancos SoundFonts integrando programas em CLEAN com o renderizador Timidity++.
- Apresentar técnicas básicas voltadas para a manipulação de entradas e saídas em interfaces gráficas no CLEAN.

Este aplicativo foi projeto para ser o mais simples e direto possível. Sendo assim o usuário poderá converter arquivos MIDI em WAVE com apenas dois "cliques" do mouse. Este aplicativo também oferece a possibilidade de ouvir o arquivo MIDI original sendo executado com os timbres do sintetizador padrão do computador do usuário ou ouvir o arquivo WAVE convertido. Além disso ele pode ser utilizado em computadores sem placa de som e ainda pode ser rodado diretamente de uma pen drive pois não necessita ser instalado.

Figura 4. 6 – Interface gráfica do aplicativo "Conversor Midi → Wave"

4.6.1 A interface gráfica

Neste aplicativo foi implementado uma interface do tipo NDI (**N**o **D**ocument **I**nterface). Esta interface possui uma janela de diálogo e é a mais simples que se pode implementar. Nela pode-se incluir botões, campos de texto, popups, entre outros.

O fluxograma da implementação desta interface é o seguinte:

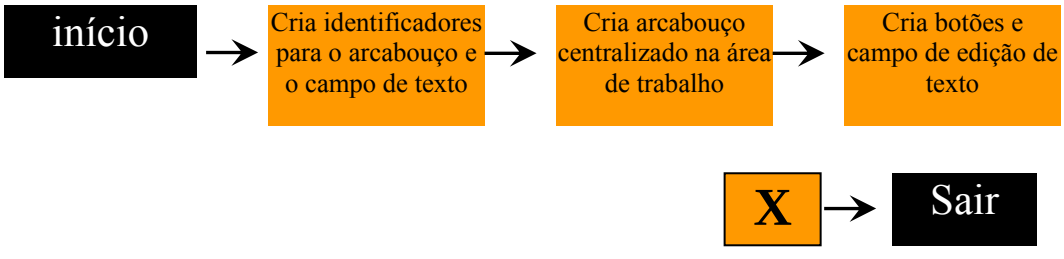


figura 4.7 – Fluxograma de implementação de uma interface NDI

A seguir será mostrado o código de implementação desta interface.

```
19 Start :: *World -> *World
20 Start world
21 = startIO NDI 0 (initialize) [ProcessClose closeProcess] world
22
23 initialize proc
24 #(certo1, arq1, proc) = fopen "8mbgm.sf2" FReadData proc
25 #(conteudo1, arq1) = freads arq1 70000000
26 #(certo1, proc) = fclose arq1 proc
27 # (ok,file, proc)= fopen ("c:\\soundfont1.sf2") FWriteData proc
28 | not ok = proc
29 # file = fwrites conteudo1 file
30 # (_, proc)= fclose file proc
31
32 #(ids,proc)=openIds 10 proc
33 #(_,proc)  =openDialog "carlos" (janela ids) proc
34 =proc
35
36 janela ids = Dialog "Conversor Midi-->Wave V-1.0"
37                     (
38                     ButtonControl "abrir Midi"  [ControlFunction (noLS abrirMIDI)]
39                     :+:
40                     ButtonControl "Converter para Wave"  [ControlFunction (noLS midiToWave)]
41                     :+:
42                     ButtonControl "Tocar arquivo Midi original"  [ControlFunction (noLS PlayMidi)]
43                     :+:
44                     ButtonControl "Tocar arquivo Wave convertido" [ControlFunction (noLS PlayWave)]
45                     :+:
46                     ButtonControl "   << STOP >>   "  [ControlFunction (noLS StopMidiWave)]
47                     :+:
48                     TextControl ("") [ControlId texto, ControlPos (Left,zero), ControlWidth (PixelWidth 650)]
49                     :+:
50                     EditControl "" (PixelWidth 650) 1 [ControlId endereco, ControlPos (Center,zero)]
51                     )
52                     [ WindowClose (noLS closeProcess), WindowId wd1]
53 where
54     wd1 = ids!!0
55     endereco = ids!!1
56     texto = ids!!2
```

figura 4.8 – código de implementação de uma interface NDI

A função que inicia uma interface é a ***startIO***.

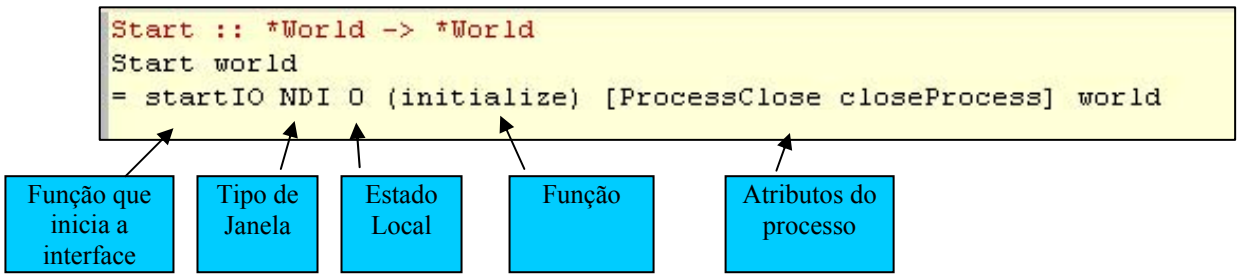


figura 4.9 – função que inicia a interface

onde:

- **Tipo de Janela** = NDI (**N**o **D**ocument **I**nterface), SDI (**S**ingle **D**ocument **I**nterface) ou MDI (**M**ultiple **D**ocument **I**nterface)
- **Estado Local** = Neste caso o valor é o inteiro 0 (zero). Poderia ser qualquer outro tipo de dado. O mesmo é utilizado quando se deseja passar um dado para os processos da interface de forma global, já que o Clean não possui variáveis

globais. Assim, o dado colocado nesta variável poderá ser visto por todos os processos através de seu estado local. Um processo possui um estado local, que é o que se precisa ter antes de iniciar o processo propriamente dito. Detalhes de como manipular o estado local e o processo podem ser vistos no material de apoio contido no CD que acompanha este trabalho.

- **Função** = função que abre um processo para a criação da interface, no caso é a função " ***initialize"***. Esta função agora é parte do mundo aberto na etapa inicial da construção da interface. Ela inicia o processo de criação da janela e outros processos que o programador queira realizar na inicialização.

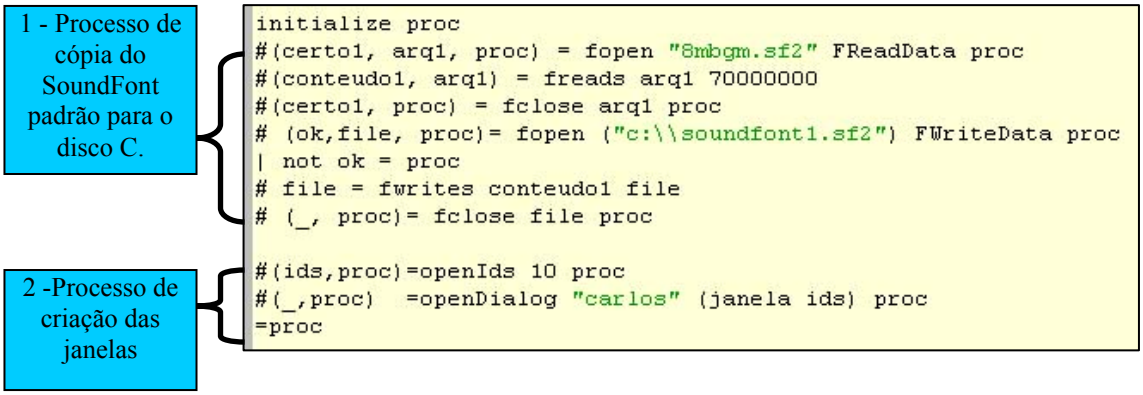


figura 4.10 – função initialize

O primeiro processo apresentado neste código realiza uma cópia do SoundFont padrão para o disco C. Ele será comentado mais adiante. O segundo processo apresentado na figura acima é o que cria as janelas. Ele começa com a função ***"openIds"*** ao qual é requisitado ao sistema operacional um número determinado de identificadores. Ao utilizar esta função, a mesma cria uma lista de identificadores, conforme valor solicitado a ela, neste caso 10 (dez). Estes identificadores servem para permitir que se acessem os dispositivos da interface, como, por exemplo, os campos de edição de texto. Assim, ao se criar os dispositivos pode-se destinar um identificador para cada um. Neste caso o arcabouço foi identificado como ***"wd1"***, o campo de edição de texto onde aparece o endereço do arquivo MIDI foi identificado como ***"endereco"*** e o campo de texto onde aparece algumas mensagem de confirmação da conversão foi identificado como ***"texto"***.

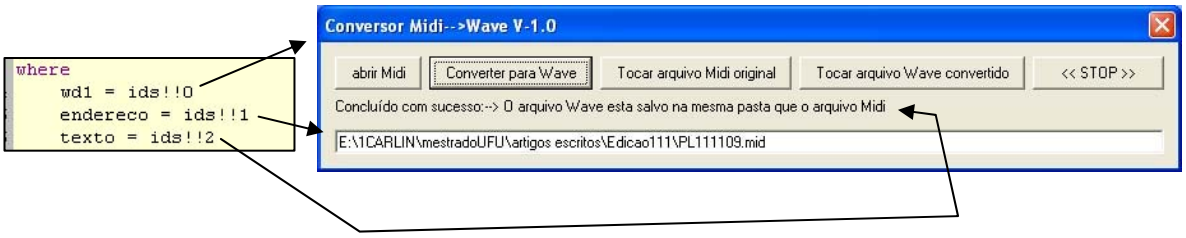


figura 4.11 –Identificadores do arcabouço e dos campos de textos

A função que cria a janela de diálogo é a ***"openDialog".*** Ela tem como argumento de entrada o estado local e a função ***"janela"***.

figura 4.12 –A função openDialog

Como não foi utilizado estado local colocou-se qualquer tipo de dado, no caso uma string *"carlos"*. A função ***"janela"*** tem como parâmetro de entrada os identificadores. Ela chama outra função, a ***"Dialog"*** . Esta, por sua vez, tem como parâmetros de entrada o texto do arcabouço, os componentes (botões, campo de textos, etc.) e os atributos da janela.

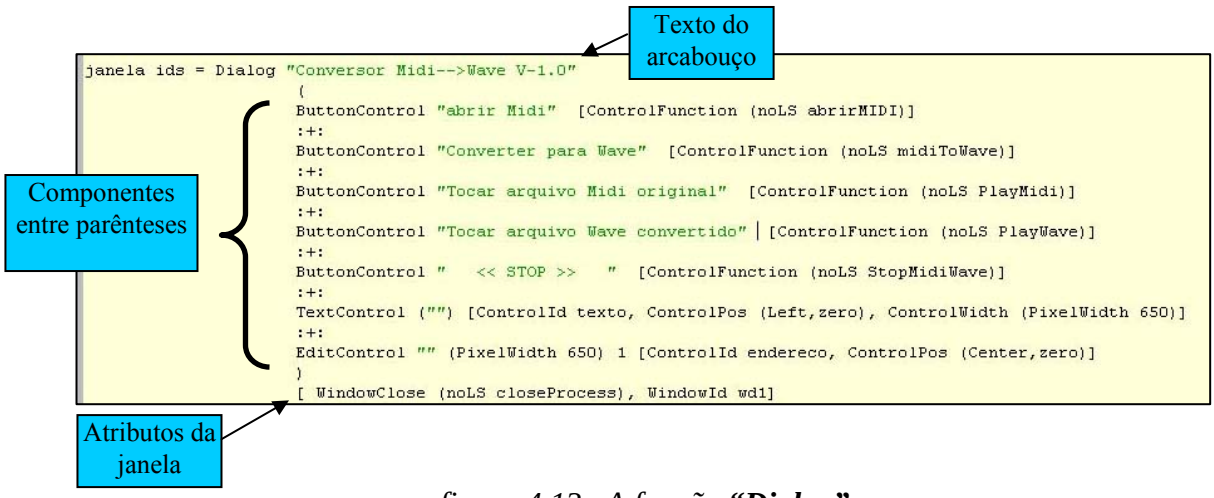


figura 4.13 –A função ***"Dialog"***

A forma como cada componente tais como os botões, popups, caixa de texto, caixa de seleção, entre outros, são implementados está ricamente exposto no livro "*A Tutorial to the Clean object I/O library 1.2"* que faz parte do material de apoio contido no CD fornecido com esta dissertação.

4.6.2 Implementação das funções de cada botão

A seguir serão apresentadas as funções que cada botão executa ao ser acionado.

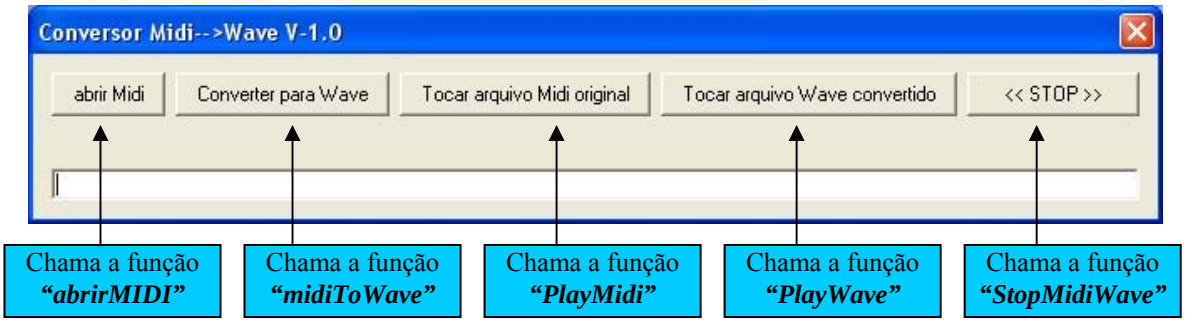


figura 4.14 –Funções que cada botão executa

● função ***"abrirMIDI"*** → Esta função abre uma caixa de diálogo do Explorer do Windows para que o usuário possa procurar e escolher o arquivo MIDI para conversão. Ela retorna o endereço do arquivo (Path) e plota no campo de edição de texto. O código desta função é o seguinte:

```
60    abrirMIDI proc
61        # (certo, proc) = selectInputFile proc
62        | isNothing certo = proc
63        # caminho = fromJust certo
64        # (certo, arq, proc) = fopen caminho FReadData proc
65        #(conteudo, arq) = freads arq 4000000
66        #(certo, proc) = fclose arq proc
67        # nome = toString (reverse (takeWhile ((<>) '\\') (reverse [x \\ x <-: caminho])))
68        | not (conteudo%(0,3) == "MThd") = appPIO (setControlText endereco "Não é um arquivo Midi.") proc
69        # proc = appPIO (setControlText endereco caminho) proc
70        = proc
```

figura 4.15 –código da função ***"abrirMIDI"***

Ao ser chamada, esta função retorna um processo contido na variável ***"proc"***. Este processo é constituido de acordo com o fluxograma apresentado a seguir.

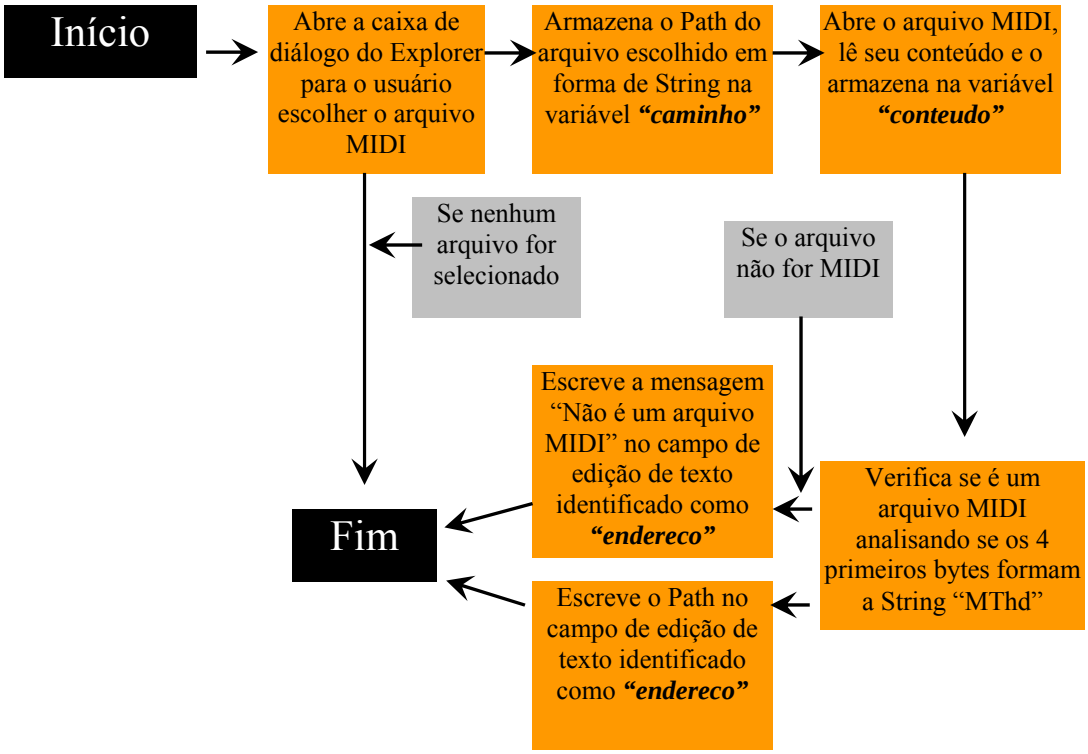


figura 4.16 –Fluxograma da função ***"abrirMIDI"***

● função ***"midiToWave"*** → Esta função lê o endereço do arquivo MIDI contido no campo de edição de texto. Posteriormente é chamada a função ***"Execute"*** que é utilizada para executar programas externos. Seu argumento de entrada é uma String contendo o nome do programa ao qual deseja-se acionar e os parâmetros de entrada do mesmo. Neste caso o programa acionado por esta função é o renderizador de SoundFonts ***"Timidity++"*** abordado no capítulo 3. Deste modo ele renderiza o arquivo MIDI para Wave utilizando os timbre do SoundFont que foi copiado para o disco C no processo de inicialização. Ao abrir ***"Conversor MIDI →Wave"***. Antes de ser criado o arcabouço da interface ele abre um arquivo SoundFont denominado "8mbgm.sf2" e copia seu conteúdo para o disco C em um arquivo SoundFont chamado "soundfont1.sf2". Isto é feito para facilitar a configuração do Timidity++, pois deste

modo ele fica configurado com um endereço (Path) fixo do SoundFont padrão que será utilizado no processo que é ***"C:\soundfont1.sf2"***.

Processo de cópia do SoundFont padrão para o disco C no momento da inicialização do programa.

```
initialize proc
#(certo1, arq1, proc) = fopen "8mbgm.sf2" FReadData proc
#(conteudo1, arq1) = freads arq1 70000000
#(certo1, proc) = fclose arq1 proc
# (ok,file, proc)= fopen ("c:\\soundfont1.sf2") FWriteData proc
| not ok = proc
# file = fwrites conteudo1 file
# (_, proc)= fclose file proc

#(ids,proc)=openIds 10 proc
#(_,proc)  =openDialog "carlos" (janela ids) proc
=proc
```

figura 4.17 –Rotina que copia o SoundFont fornecido com o programa para o disco C.

O código da função ***"midiToWave"*** é o seguinte:

```
midiToWave proc
    # (Just dial, proc) = accPIO(getWindow wd1) proc
    # (se,Just caminho2) = getControlText endereco dial
     | not se = proc
     | (caminho2 == "") = appPIO (setControlText texto "É preciso abrir o arquivo Midi antes de converter") proc
    # proc = appPIO (setControlText texto "Aguarde...") proc
    # nome = toString (reverse (takeWhile ((<>) '\\') (reverse [x \\ x <-: caminho2])))
    # (a,b,proc) = Execute ((juntaListaStringPath(init(pathDividido)))+++"timidity-con.exe -s 44100 -Ow "+++ "\""+++ caminho2 +++ "\"") proc
    # proc = appPIO (setControlText texto "Concluído com sucesso:--> O arquivo Wave esta salvo na mesma pasta que o arquivo Midi") proc
    = proc
```

*figura 4.18 –Rotina da função **"midiToWave"***

A seguir será apresentado o fluxograma desta função:

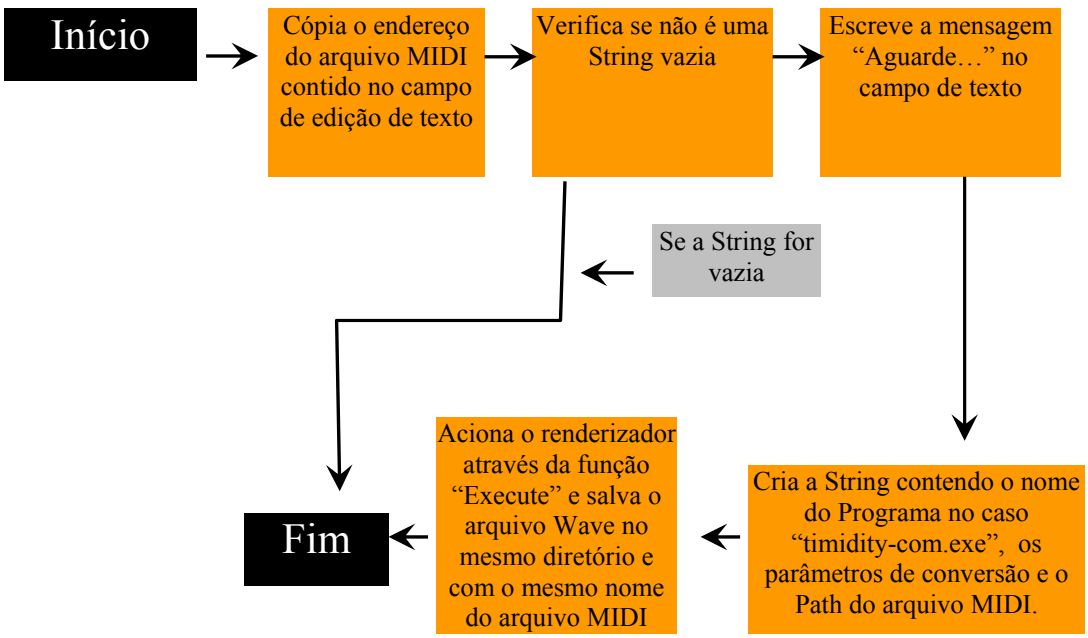


*figura 4.19 –Fluxograma da função **"midiToWave"***

● função ***"PlayMidi"*** → Esta função toca o arquivo MIDI original utilizando os timbres do sintetizador instalado no sistema do usuário. Seu argumento de entrada é uma String contendo o endereço (Path) do arquivo MIDI em questão. Ela chama outra função chamada ***"PlayMid"*** que por sua vez chama a função ***"OSPlayMusic"*** contida nas bibliotecas fornecidas com o programa. Sua rotina é a seguinte:

```
PlayMidi proc
# (Just dial, proc) = accPIO(getWindow wd1) proc
# (se,Just caminho2) = getControlText endereco dial
  | not se = proc
  | (caminho2 == "") = appPIO (setControlText texto "É preciso abrir o arquivo Midi antes de converter") proc
# (_, proc) = OSStopMusic proc
# (_, proc) = stopWav proc
# (_, proc) = playMid caminho2 proc
= proc
```

figura 4.20 –Fluxograma da função ***"PlayMidi"***

O fluxograma desta função é o seguinte:

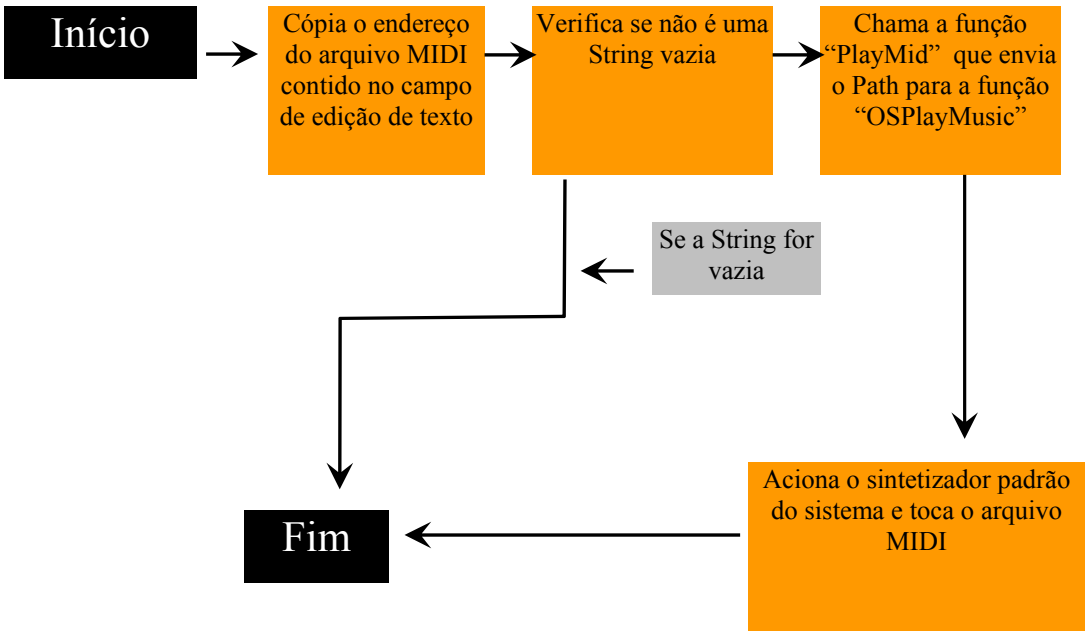


figura 4.21 –Fluxograma da função ***"PlayMidi"***

● função ***"PlayWave"*** → Esta função toca o arquivo WAVE renderizado pelo Timidity++. Seu argumento de entrada é o endereço (Path) do arquivo WAVE. Como este arquivo é salvo no mesmo diretório e com o mesmo nome do arquivo MIDI, basta substituir os 3 (três) últimos caracteres "mid" para "wav"da String contida no campo de

edição de texto e inseri-la como argumento de entrada na função ***"PlayWav"*** contida na biblioteca Wave.icl fornecida com o programa.

```
PlayWave proc
# (Just dial, proc) = accPIO(getWindow wd1) proc
# (se,Just caminho2) = getControlText endereco dial
 | not se = proc
 | (caminho2 == "") = appPIO (setControlText texto "É preciso abrir o arquivo Midi antes de converter") proc
# (_, proc) = OSStopMusic proc
# (_, proc) = stopWav proc
# caminhoWave = (caminho2%(0,((size caminho2)-4)))+++"wav"
# (_, proc) = playWav caminhoWave proc
= proc
```

figura 4.22 –Rotina da função ***"PlayWave"***

O Fluxograma desta função é o seguinte:

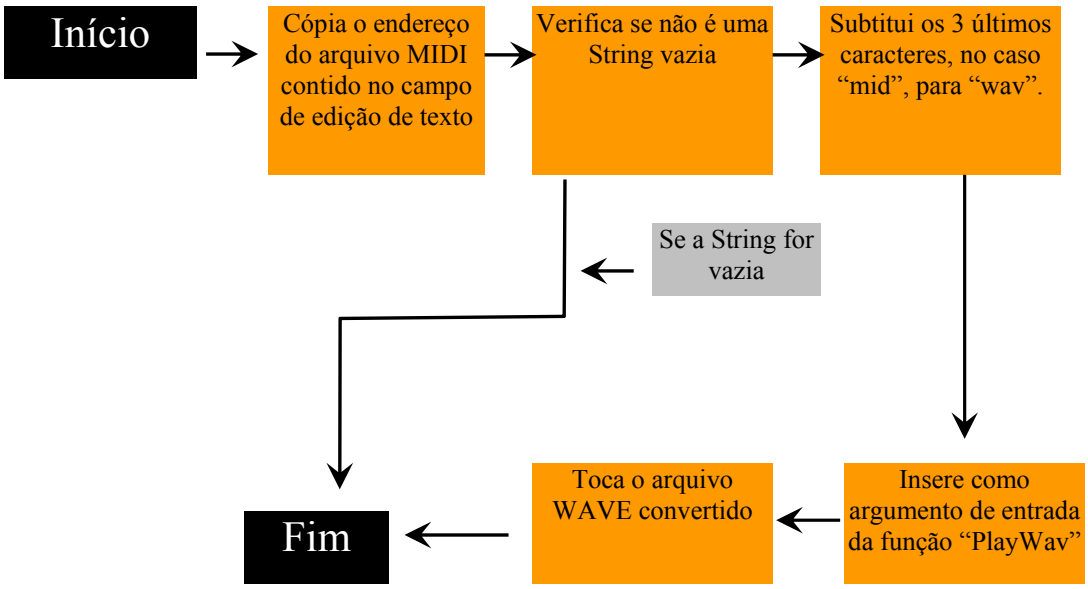


figura 4.23 –Fluxograma da função ***"PlayWave"***

- função ***"StopMidiWave"*** → Esta função para a execução dos arquivos MIDI e WAVE. Elas chamam as funções "stopWave" e "OSStopMusic" simultaneamente. Estas funções estão contidas nas bibliotecas fornecidas com o programa.

```
StopMidiWave proc
# (_, proc) = stopWav proc
# (_, proc) = OSStopMusic proc
=proc
```

figura 4.24 –Rotina da função ***"StopMidiWave".***

4.7 – Implementação de uma interface gráfica para um violão virtual com botões animados.

A implementação deste aplicativo tem como objetivos:

- Apresentar os conceitos básicos para o desenvolvimento de uma interface multimídia contendo botões interativos que simulem o aperto de um botão no braço do violão.
- Apresentar técnicas de plotagem de figuras no arcabouço, que serão utilizadas também na plotagem das partituras do aplicativo final.

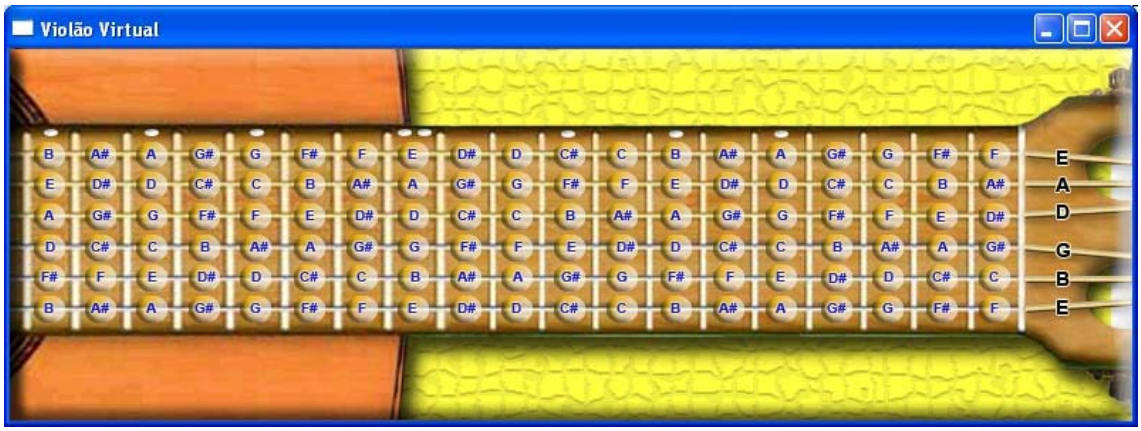


figura 4.25 –Interface gráfica do violão virtual

Este aplicativo oferece ao usuário uma interface interativa que fica monitorando o mouse de forma que quando ele passa sobre um dos botões, este reage dando a impressão de que o botão está sendo apertado.

figura 4.26 – Basta passar o mouse sobre os botões para eles interagirem

O conceito que está por tráz desta técnica é a utilização de arquivos bmps (*bitmap files*) que devem ser mostrados repetidamente. Esses arquivos criam (ao movimento ou clique

do mouse) a ilusão de se pressionar a tecla, sobrepondo-a por um arquivo bmp com o desenho do botão afundado.

Neste aplicativo, é utilizada a rotina de rastreamento da posição do mouse com relação ao arcabouço da janela e a *plotagem* de bitmaps. Deste modo é criada uma animação que simula o movimento de apertar o braço do violão. Quando o mouse é apertado o botão pisca e é acionado um arquivo sonoro (MIDI), equivalente ao som da nota musical pressionada, simulando a ação de se tocar o violão.

figura 4.27 – Ao clique do mouse o botão pisca e é emitido o som da nota correspondente

Para isso foi utilizada uma interface SDI por ser uma interface mais elaborada, permitindo-se incluir bitmap de fundo, menus, barra de ferramentas e outros recursos adicionais.

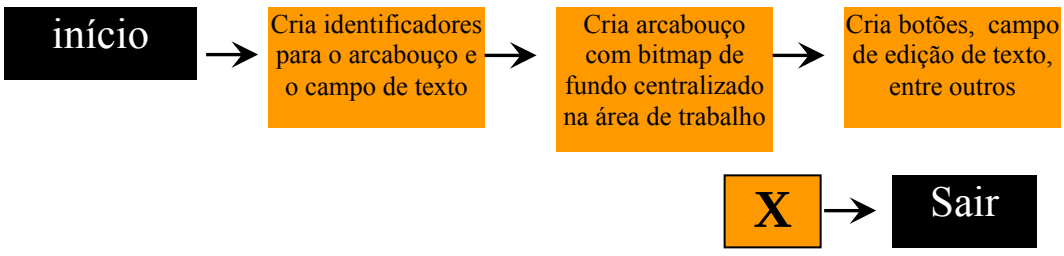


figura 4.28 – Fluxograma da interface SDI

O código de criação da interface SDI está apresentado logo em seguida:

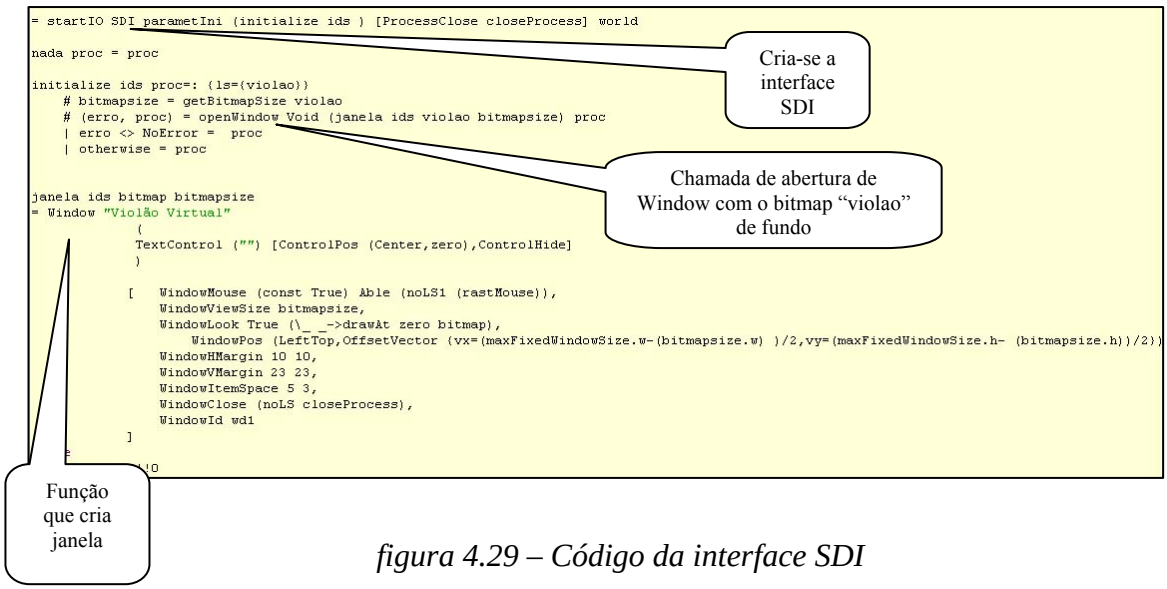


figura 4.29 – Código da interface SDI

Os bitmaps que serão utilizados na criação desta interface são carregados no registro e aberto na inicialização do programa, deste modo evita-se um *bug* do Sistema Operacional Windows que ocorre ao se tentar abrir o mesmo arquivo repetidamente, mais de vinte vezes. A única desvantagem é que a máquina deverá possuir uma memória RAM que permita se armazenar todos os bitmaps abertos.

```
:: Registro =   {
                 notas          :: [{#Char}]
                ,nota           :: Int
                ,notaAnt        :: {#Char}
                ,violao         :: (Bitmap)
                ,botC           :: (Bitmap)
                ,botCs          :: (Bitmap)
                ,botD           :: (Bitmap)
                ,botDs          :: (Bitmap)
                ,botE           :: (Bitmap)
                ,botF           :: (Bitmap)
                ,botFs          :: (Bitmap)
                ,botG           :: (Bitmap)
                ,botGs          :: (Bitmap)
                ,botA           :: (Bitmap)
                ,botAs          :: (Bitmap)
                ,botB           :: (Bitmap)
                ,botC1          :: (Bitmap)
                ,botCs1         :: (Bitmap)
                ,botD1          :: (Bitmap)
                ,botDs1         :: (Bitmap)
                ,botE1          :: (Bitmap)
                ,botF1          :: (Bitmap)
                ,botFs1         :: (Bitmap)
                ,botG1          :: (Bitmap)
                ,botGs1         :: (Bitmap)
                ,botA1          :: (Bitmap)
                ,botAs1         :: (Bitmap)
                ,botB1          :: (Bitmap)
                ,bot6solta      :: (Bitmap)
                ,bot5solta      :: (Bitmap)
                ,bot4solta      :: (Bitmap)
                ,bot3solta      :: (Bitmap)
                ,bot2solta      :: (Bitmap)
                ,bot1solta      :: (Bitmap)
                ,botSoltaDown   :: (Bitmap)
                }
```

figura 4.30 – Código da criação dos registros para armazenar os bitmaps

```
52 Start world
53
54 # (maybeBitmap,world)    = openBitmap "fundoViolao.bmp" world
55 # bitmapViolao           = (fromJust maybeBitmap)
56 # bitmapsize             = getBitmapSize (snd3 bitmapViolao)
57 # (maybeBitmap,world)    = openBitmap "botC.bmp" world
58 # bitmapbotC             = (fromJust maybeBitmap)
59 # (maybeBitmap,world)    = openBitmap "botCs.bmp" world
60 # bitmapbotCs            = (fromJust maybeBitmap)
61 # (maybeBitmap,world)    = openBitmap "botD.bmp" world
62 # bitmapbotD             = (fromJust maybeBitmap)
63 # (maybeBitmap,world)    = openBitmap "botDs.bmp" world
64 # bitmapbotDs            = (fromJust maybeBitmap)
65 # (maybeBitmap,world)    = openBitmap "botE.bmp" world
```

figura 4.31 – Código da abertura dos bitmaps na inicialização

```
126 # parametIni =  {
127                    notas          = [""]
128                   ,nota           = 0
129                   ,notaAnt        = ""
130                   ,violao         = bitmapViolao
131                   ,botC           = bitmapbotC
132                   ,botCs          = bitmapbotCs
133                   ,botD           = bitmapbotD
134                   ,botDs          = bitmapbotDs
135                   ,botE           = bitmapbotE
136                   ,botF           = bitmapbotF
137                   ,botFs          = bitmapbotFs
138                   ,botG           = bitmapbotG
139                   ,botGs          = bitmapbotGs
140                   ,botA           = bitmapbotA
141                   ,botAs          = bitmapbotAs
142                   ,botB           = bitmapbotB
143                   ,botC1          = bitmapbotC1
144                   ,botCs1         = bitmapbotCs1
145                   ,botD1          = bitmapbotD1
146                   ,botDs1         = bitmapbotDs1
147                   ,botE1          = bitmapbotE1
148                   ,botF1          = bitmapbotF1
149                   ,botFs1         = bitmapbotFs1
150                   ,botG1          = bitmapbotG1
151                   ,botGs1         = bitmapbotGs1
152                   ,botA1          = bitmapbotA1
153                   ,botAs1         = bitmapbotAs1
154                   ,botB1          = bitmapbotB1
155                   ,bot6solta      = bitmapbot6solta
156                   ,bot5solta      = bitmapbot5solta
157                   ,bot4solta      = bitmapbot4solta
158                   ,bot3solta      = bitmapbot3solta
159                   ,bot2solta      = bitmapbot2solta
160                   ,bot1solta      = bitmapbot1solta
161                   ,botSoltaDown   = bitmapbotSoltaDown
162               }
163
```

figura 4.32 – Código do armazenamento dos bitmaps no registro na inicialização

A lógica do programa é a seguinte:

● O aplicativo ativa, na inicialização, o função de rastreamento do mouse contida na biblioteca StdIO do CLEAN.
● A cada movimento do mouse são criadas variáveis que armazenam suas coordenadas X e Y (em pixel) e são verificadas se estão dentro de alguma região de algum botão.
● Se a região for uma região de algum dos botões, é acionada a função de plotar um novo bitmap na região em questão, dando a impressão de que o botão está sendo afundado.

● Se o mouse for apertado dentro desta região, é acionada, novamente, a função de plotar o bitmap, porém o figura será a do botão piscando e simultaneamente executa-se a função "PlayMidi" para tocar o arquivo MIDI correspondente ao som da nota em questão.
● Quando o botão do mouse é souto, é plotado o bitmap com a figura do botão afundado e o ciclo se repete a medida que o mouse vai passeando entre os outro botões.

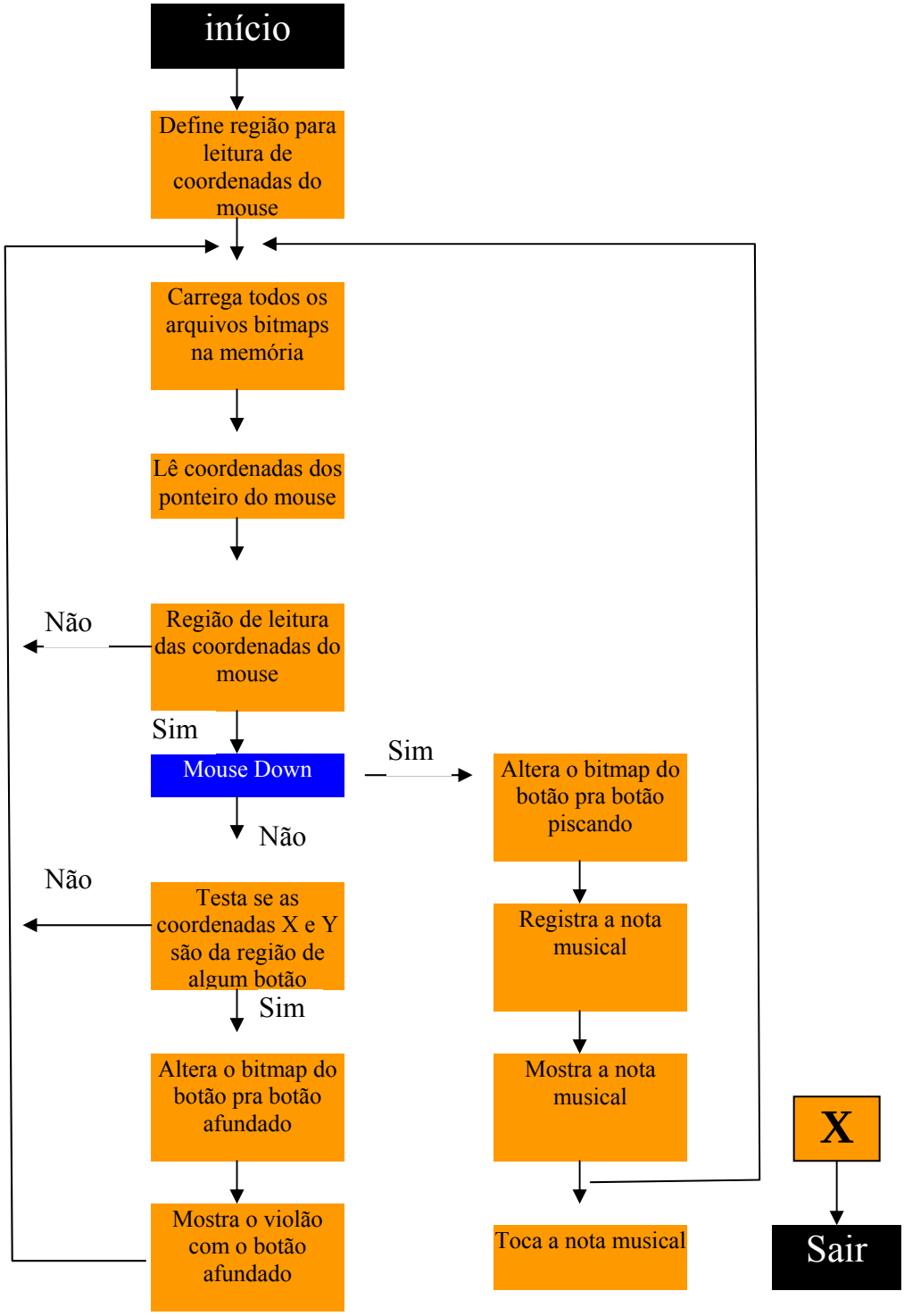


figura 4.33 – Fluxograma do programa ***"Violão Virtual"***

O código da função de rastrear o mouse é o seguinte:

```
    rastMouse  x proc=: {ls={notaAnt,violao}}
    # pos = separaCoordenada (toString (drop 11 [y \\ y <-: (toString x)]))
    # listaCoordenadas = StringTokens  testaDigito pos
    # posX =toInt( listaCoordenadas!!0)
    # posY =toInt( listaCoordenadas!!1)
    # (notaAnt,bmp,xX,yY,nota,popupNota,proc) = testaPosicaoSemClique posX posY proc
    # (notaAnt,bmp1,xX1,yY1,nota,popupNota,proc) = testaPosicaoComClique posX posY proc
    |((posX<16)||(posX>757)||(posY<65)||(posY>193))||(nota=="")
        # proc = appPIO(appWindowPicture wd1 (drawAt {x=0,y=0} violao)) proc
        = {ls = {proc.ls& notaAnt=nota}, io = proc.io}
    |((toString (take 10 [y \\ y <-: (toString x)])) == "(MouseDown")
      # proc = appPIO(appWindowPicture wd1 (drawAt {x=xX1,y=yY1} bmp1)) proc
      # (_, proc) = playMid ("guitar\\"+++nota+++".mid") proc
      = {ls = {proc.ls& notaAnt=nota}, io = proc.io}
    |((toString (take 8 [y \\ y <-: (toString x)])) == "(MouseUp")
        # proc = appPIO(appWindowPicture wd1 (drawAt {x=xX,y=yY} bmp)) proc
        = {ls = {proc.ls& notaAnt=nota}, io = proc.io}
    | (notaAnt == nota) = proc
    | nota<>""
        # proc = appPIO(appWindowPicture wd1 (drawAt {x=0,y=0} violao)) proc
        # proc = appPIO(appWindowPicture wd1 (drawAt {x=xX,y=yY} bmp)) proc
        = {ls = {proc.ls& notaAnt=nota}, io = proc.io}
    = {ls = {proc.ls& notaAnt=nota}, io = proc.io}
```

figura 4.34 – Código da função de rastrear o mouse

A funções encarregadas de fornecer as coordenadas corretas para plotagem de cada botão correspondente a cada nota do violão são ***"testaPosicaoSemClique"*** e a ***"testaPosicaoComClique"***. Elas retornam principamente o bitmap a ser plotado, as coordenadas da plotagem e a nota a ser tocada pela função ***"playMid"***.

Veja a seguir um trecho do código destas funções, ao qual são testadas as posições da corda 6 do violão. A mesma lógica é aplicada aos botões das outras cordas.

```
testaPosicaoSemClique posX posY  proc=: {ls=(notaAnt,violao, botC, botCs, botD, botDs, botE, botF, botFs, botG, botGs, botA, botAs, botB, bot

// corda 6
| (posX>(666+deltaX) && posX <(687+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botF, (666+deltaX),(1+deltaY),"fa3",0,proc)
| (posX>(629+deltaX) && posX <(650+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botFs,(629+deltaX),(1+deltaY),"fa#3",0,proc)
| (posX>(591+deltaX) && posX <(612+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botG, (591+deltaX),(1+deltaY),"sol3",0,proc)
| (posX>(555+deltaX) && posX <(576+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botGs,(555+deltaX),(1+deltaY),"sol#3",0,proc
| (posX>(516+deltaX) && posX <(537+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botA, (516+deltaX),(1+deltaY),"la3",0,proc)
| (posX>(479+deltaX) && posX <(500+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botAs,(479+deltaX),(1+deltaY),"la#3",0,proc)
| (posX>(442+deltaX) && posX <(463+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botB, (442+deltaX),(1+deltaY),"si3",0,proc)
| (posX>(404+deltaX) && posX <(425+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botC, (404+deltaX),(1+deltaY),"do4",0,proc)
| (posX>(367+deltaX) && posX <(388+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botCs,(367+deltaX),(1+deltaY),"do#4",0,proc)
| (posX>(330+deltaX) && posX <(351+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botD, (330+deltaX),(1+deltaY),"re4",0,proc)
| (posX>(294+deltaX) && posX <(315+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botDs,(294+deltaX),(1+deltaY),"re#4",0,proc)
| (posX>(257+deltaX) && posX <(278+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botE, (257+deltaX),(1+deltaY),"mi4",0,proc)
| (posX>(221+deltaX) && posX <(242+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botF, (221+deltaX),(1+deltaY),"fa4",0,proc)
| (posX>(184+deltaX) && posX <(205+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botFs,(184+deltaX),(1+deltaY),"fa#4",0,proc)
| (posX>(146+deltaX) && posX <(167+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botG, (146+deltaX),(1+deltaY),"sol4",0,proc)
| (posX>(110+deltaX) && posX <(131+deltaX))  && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botGs,(110+deltaX),(1+deltaY),"sol#4",0,proc
| (posX>(73+deltaX)  && posX <(94+deltaX))   && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botA, (73+deltaX), (1+deltaY),"la4",0,proc)
| (posX>(37+deltaX)  && posX <(58+deltaX))   && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botAs,(37+deltaX), (1+deltaY),"la#4",0,proc)
| (posX>(1+deltaX)   && posX <(22+deltaX))   && (posY>(1+deltaY) &&posY<(19+deltaY)) =  (notaAnt,botB, (1+deltaX),  (1+deltaY),"si4",0,proc)
```

figura 4.35 – Trecho do código da função ***"testaPosiçãoSemClique"***

O executável deste programa e seu código completo encontra-se no CD fornecido com esta dissertação.

4.8 – Implementação de um compilador TEXTO → MIDI

A implementação deste aplicativo tem como objetivos:

- Apresentar os conceitos básicos para o desenvolvimento de um compilador onde o usuário entre com dados em um campo de texto e a partir dos mesmos é gerado um arquivo SMF.

- Apresentar técnicas de separar *Tokens*[5] de uma *String* ou de um arquivo texto para posterior manipulação e geração de mensagens MIDI.

- Apresentar a função ***"geraMidiF1"*** contida na biblioteca "geraMidiF1.icl" desenvolvida inicialmente por CAMARGO (2007) e aprimorada neste trabalho.

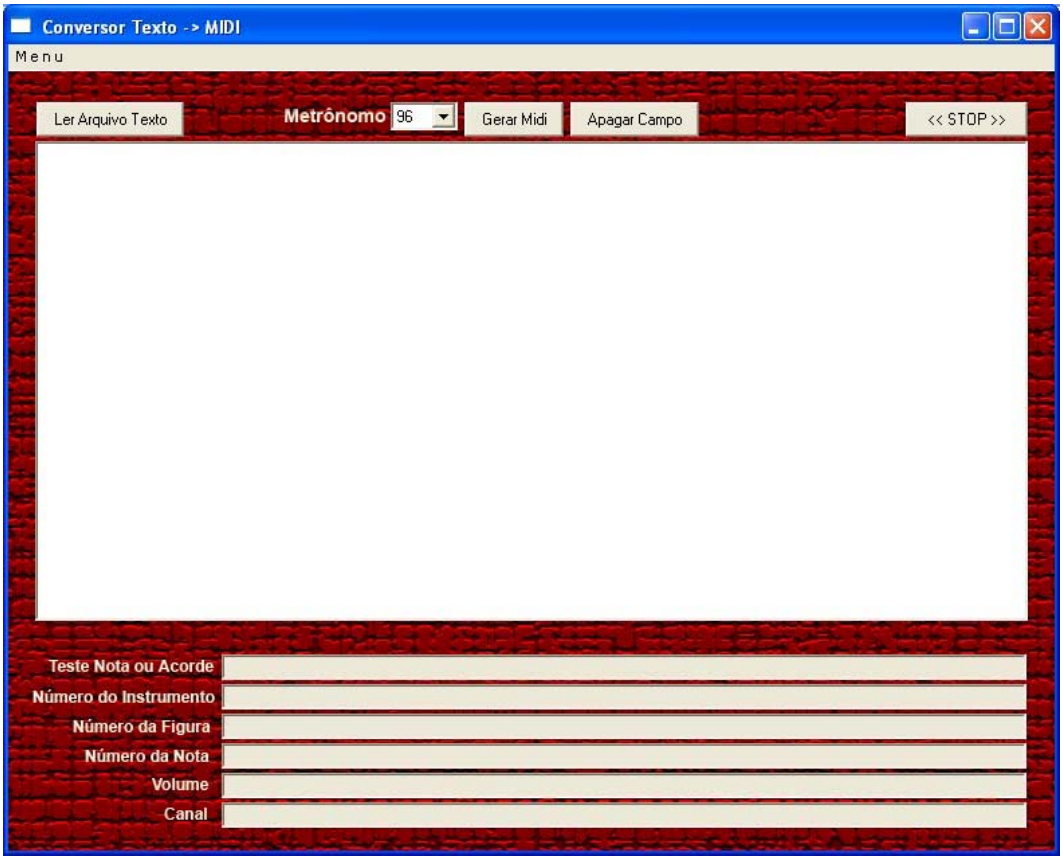


figura 4.36 – Interface gráfica do aplicativo ***"Compilador Texto → MIDI"***

[5] **Token** em computação é um segmento de texto ou símbolo que pode ser manipulado por um parser, que fornece um significado ao texto; em outras palavras, é um conjunto de caracteres (de um alfabeto, por exemplo) com um significado coletivo. http://pt.wikipedia.org/wiki/Token

Este aplicativo oferece ao usuário a possibilidade de criar seqüências MIDI diretamente de um campo de edição de texto, com uma sintaxe simples e fácil de assimilar. Veja a seguir como é esta sintaxe.

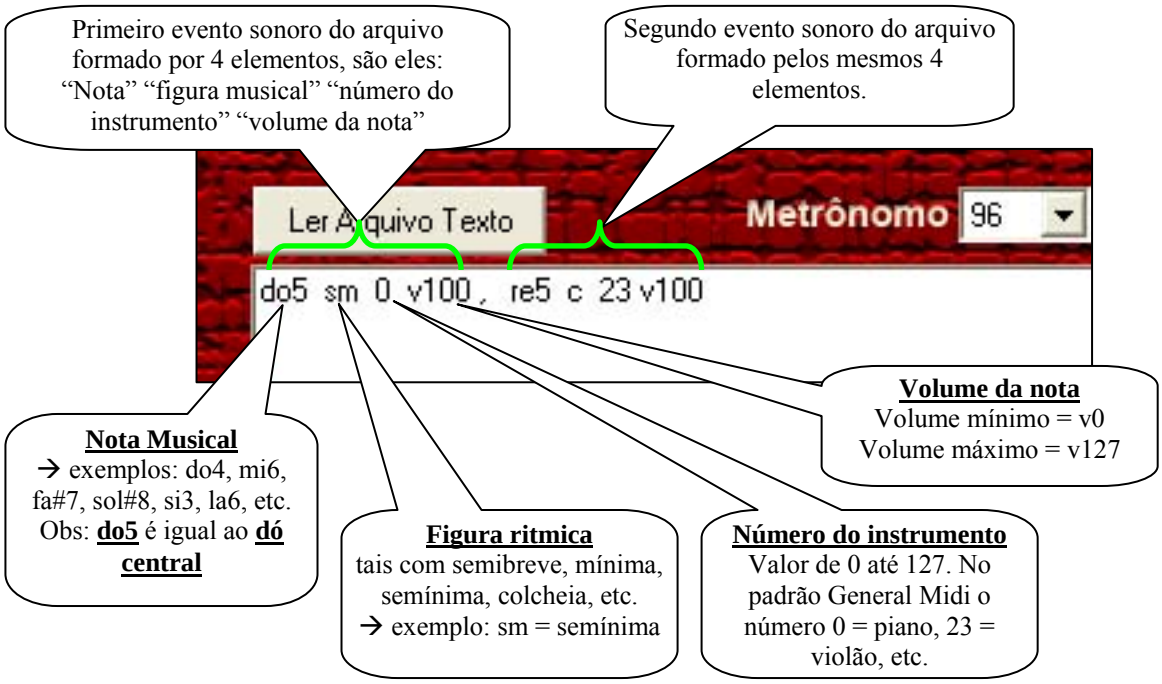


*figura 4.37 – Sintaxe utilizada no **"Compilador Texto → MIDI"***

Cada nota musical ou pausa é criada a partir de 4 (quatro) argumentos que devem ser colocados na seqüência apresentada na figura acima que é a seguinte:

→ "nota musical" "figura rítmica" "número do instrumento" "volume"

- ***"nota musical"*** – Qualquer uma das 128 notas presentes no padrão MIDI. A codificação ficou da seguinte forma:

```
notas =
  ["do0", "do#0", "re0", "re#0", "mi0", "fa0", "fa#0", "sol0", "sol#0", "la0", "la#0", "si0",
   "do1", "do#1", "re1", "re#1", "mi1", "fa1", "fa#1", "sol1", "sol#1", "la1", "la#1", "si1",
   "do2", "do#2", "re2", "re#2", "mi2", "fa2", "fa#2", "sol2", "sol#2", "la2", "la#2", "si2",
   "do3", "do#3", "re3", "re#3", "mi3", "fa3", "fa#3", "sol3", "sol#3", "la3", "la#3", "si3",
   "do4", "do#4", "re4", "re#4", "mi4", "fa4", "fa#4", "sol4", "sol#4", "la4", "la#4", "si4",
   "do5", "do#5", "re5", "re#5", "mi5", "fa5", "fa#5", "sol5", "sol#5", "la5", "la#5", "si5",
   "do6", "do#6", "re6", "re#6", "mi6", "fa6", "fa#6", "sol6", "sol#6", "la6", "la#6", "si6",
   "do7", "do#7", "re7", "re#7", "mi0", "fa7", "fa#7", "sol7", "sol#7", "la7", "la#7", "si7",
   "do8", "do#8", "re8", "re#8", "mi8", "fa8", "fa#8", "sol8", "sol#8", "la8", "la#8", "si8",
   "do9", "do#9", "re9", "re#9", "mi9", "fa9", "fa#9", "sol9", "sol#9", "la9", "la#9", "si9",
   "do10", "do#10", "re10", "re#10", "mi10", "fa10", "fa#10", "sol10"]
```

figura 4.38 – Codificação das notas musicais

(Obs: do5 = dó central)

No caso da pausa basta entrar com a letra ***"p"*** ao invés de entrar com o código da nota.

● ***"figura rítmica"*** – São as figuras musicais que representam a duração das notas, tais como semibreve, mínima, semínima, colcheia, semicolcheia, fusa e semifusa. Deste modo a codificação ficou da seguinte forma:

sb	→ *Semibreve*	**sbp**	→ *Semibreve pontuada*	**sbt**	→ *Semibreve tercina*
m	→ *Mínima*	**mp**	→ *Mínima pontuada*	**mt**	→ *Mínima tercina*
sm	→ *Semínima*	**smp**	→ *Semínima pontuada*	**smt**	→ *Semínima tercina*
c	→ *Colcheia*	**cp**	→ *Colcheia pontuada*	**ct**	→ *Colcheia tercina*
sc	→ *Semicolcheia*	**scp**	→ *Semicolcheia pontuada*	**sct**	→ *Semicolcheia tercina*
f	→ *Fusa*	**fp**	→ *Fusa pontuada*	**ft**	→ *Fusa tercina*
sf	→ *Semifusa*	**sfp**	→ *Semifusa pontuada*	**sft**	→ *Semifusa tercina*

Tabela 4.1 – Codificação das figuras rítmicas

● ***"número do instrumento"*** – É um numero inteiro que varia de 0 até 127 sendo que cada um deles representa um dos instrumentos do padrão General MIDI. Por exemplo: O número 57 é o Trombone, o número 73 é a Flauta, o número 0 (zero) é o piano, e assim por diante. Para ver a numeração de todos os instrumentos disponíveis basta consultar a *"Tabela 1 – Instrumentos do padrão GM"* presente no capítulo 2 desta dissertação.

● ***"volume"*** – É o volume da nota musical que pode variar de 0 (zero) até 127, para isto basta acrescentar um ***" v "*** na frete no valor do volume, por exemplo: ***"v64"*** significa volume no valor 64, ***"v127"*** significa volume 127 (volume máximo), e assim por diante.

Cada evento de nota é separado por uma vírgula, deste modo elas são tocadas em seqüência. Quando o usuário deseja que os eventos de notas sejam tocados simultaneamente, formando um acorde, basta não separá-los com a vírgula. Veja exemplo a seguir:

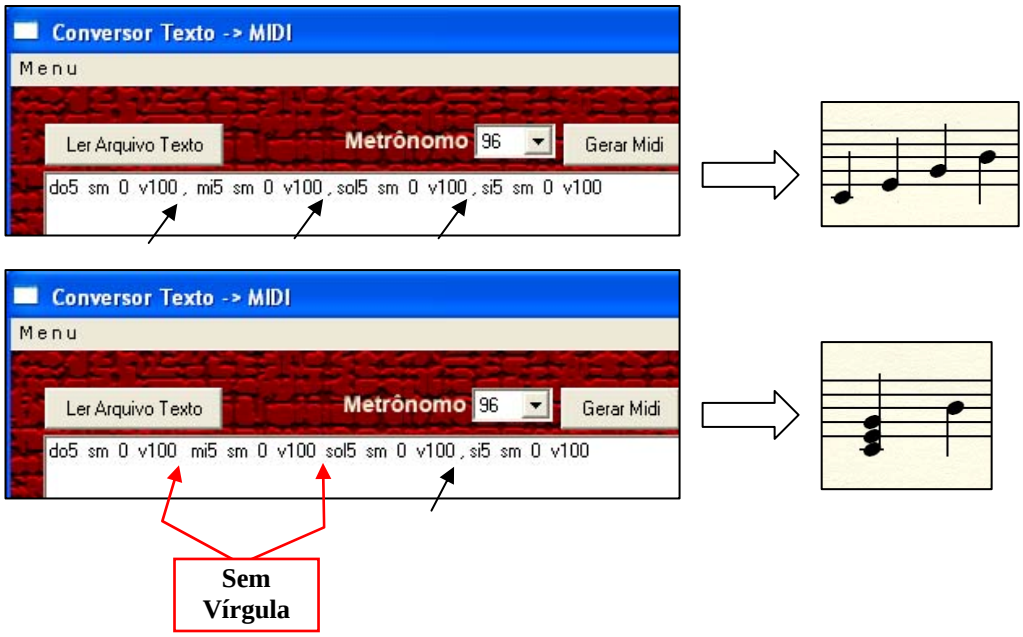


figura 4.39 – Criação de notas melódicas ou acordes

Veja a seguir o que significa cada elemento da interface gráfica.

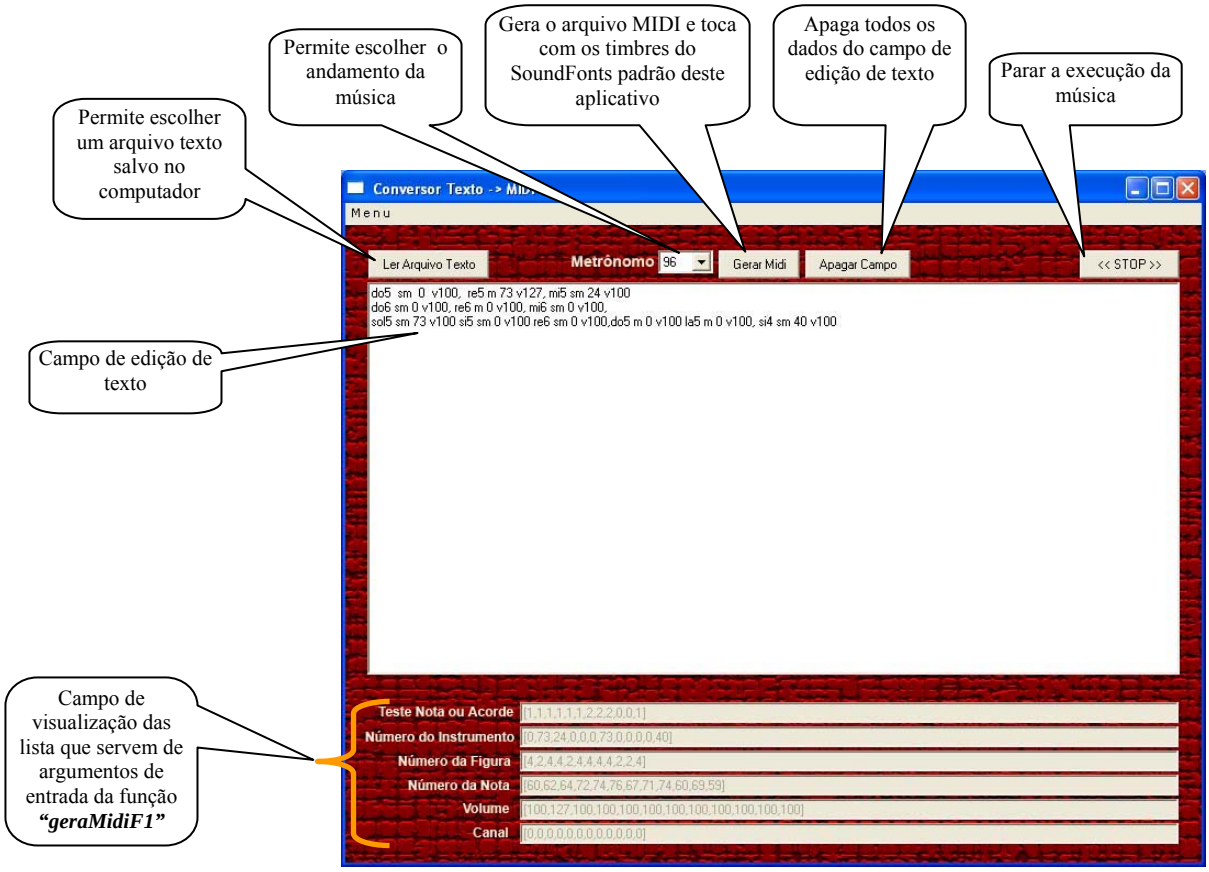


figura 4.40 – função dos elementos da interface gráfica do aplicativo ***"Compilador Texto → MIDI"***

O processo de implementação desta interface é o mesmo já apresentado nos outros aplicativos ao longo deste capítulo. Sendo assim abordaremos apenas a implementação das funções utilizadas neste programa que são as seguintes:

- função ***"abrirTxt"*** → Esta função é chamada quando o botão ***"Ler arquivo Texto"*** é acionado. Ela tem como argumento de entrada o estado local e o processo. Ela executa outra função chamada ***"AbrirArquivoTextoExplproc"*** que está contida na biblioteca ***"StdArqNovo.icl"*** importada pelo aplicativo e fornecida junto com o mesmo. Esta função abre a janela do Explorer do Windows para o usuário escolher um arquivo texto, posteriormente ele lê o conteúdo e grava na variável ***"conteudo"***. Posteriormente ele armazena este conteúdo na forma de String no campo de edição de texto da interface gráfica identificado como ***"t1"*** . O código desta função é o seguinte:

```
abrirTxt (el,proc)
#(conteudo,end,proc)=AbrirArquivoTextoExplproc proc
# proc= appPIO (setControlText t1 conteudo) proc
=(el,proc)
```

figura 4.41 – Código da função ***"abrirTxt"***

Veja a seguir o fluxograma deste função.

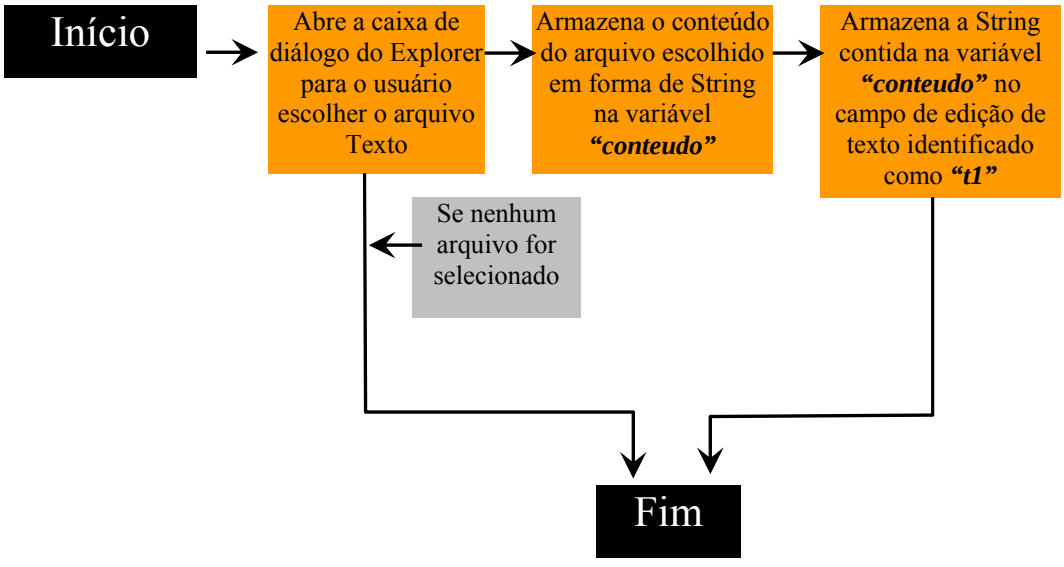


figura 4.42 – Fluxograma da função ***"abrirTxt"***

● função ***"GerarMidi"*** → Esta função é chamada quando o botão ***"Gerar Midi"*** é acionado. É ela que realiza o processo de converter o texto em um arquivo SMF.

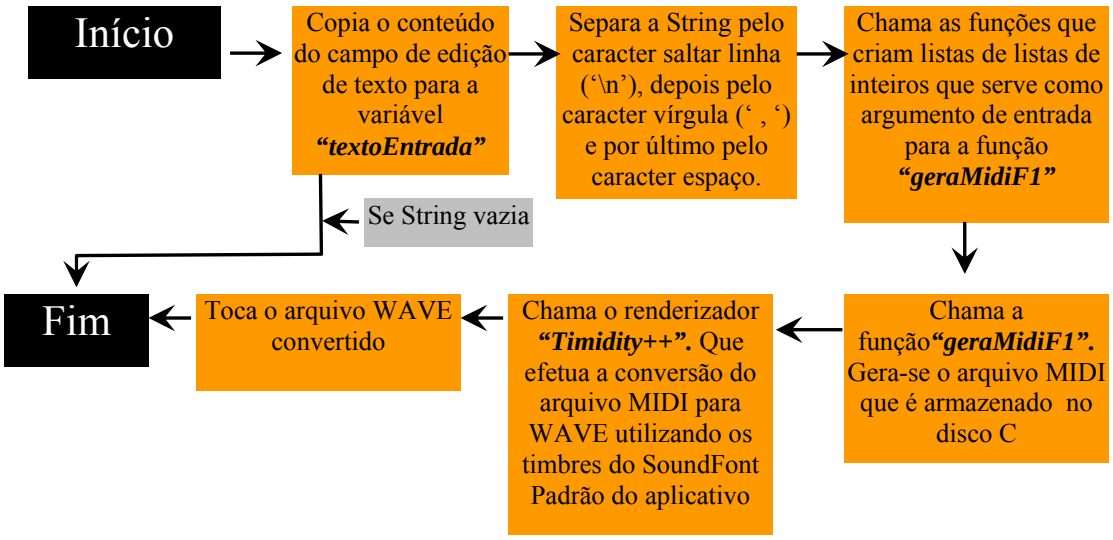


figura 4.43 – Fluxograma da função ***"GerarMidi"***

```
GerarMidi (el,proc)
# (_, proc) = OSStopMusic proc
# (Just dial,proc) = accPIO(getWindow wd1) proc
# (_,Just metr) = getControlText tMet dial
# (_,Just textoEntrada) = getControlText t1 dial
|textoEntrada=="" = (el,proc)

#listaTexto1 = StringTokens testaBarraN textoEntrada
#listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
#listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
#listaTexto4 = flatten (flatten listaTexto3)

#notaAcordeF1 = listaNotaAcorde listaTexto3
#notaAcordeF1String = [(toString x) \\ x<-notaAcordeF1]
#proc =appPIO (setControlText tNotArc  (CastingLstrStr notaAcordeF1String))proc

#instrumentoF1 = listaInstrumentos listaTexto4
#instrumentoF1String = [(toString x) \\ x<-instrumentoF1]
#proc =appPIO (setControlText tInst  (CastingLstrStr instrumentoF1String))proc

#figuraF1 = listaFiguras listaTexto4
#figuraF1String = [(toString x) \\ x<-figuraF1]
#proc =appPIO (setControlText tFig  (CastingLstrStr figuraF1String))proc

#notaF1 = listaNotas listaTexto4
#notaF1String = [(toString x) \\ x<-notaF1]
#proc =appPIO (setControlText tNot  (CastingLstrStr notaF1String))proc

#volumeF1 = listaVolumes listaTexto4
#volumeF1String = [(toString x) \\ x<-volumeF1]
#proc =appPIO (setControlText tVol  (CastingLstrStr volumeF1String))proc

#canalF1 = listaCanal listaTexto4
#canalF1String = [(toString x) \\ x<-canalF1]
#proc =appPIO (setControlText tCan  (CastingLstrStr canalF1String))proc

#listaF1 = [notaAcordeF1,instrumentoF1,figuraF1,notaF1,volumeF1,canalF1]

# musicaListaF1 = [ [[ 96,4,4,0,(toInt metr)]],listaF1]
# (listaMIDI,proc) = gerar2 musicaListaF1 proc
=(el,proc)
```

figura 4.44 – Código da função ***"GerarMidi"***

Esta conversão é realizada da seguinte forma:

1 – Armazena-se a String contida no campo de edição de texto na variável ***"textoEntrada"***.

2 – Separa-se a String pelo caracter de "saltar linha" ('\n') e armazena o resultado na variável ***"listaTexto1".*** Para realizar esta separação foi utilizada a função ***"StringTokens"*** contida da biblioteca ***StdArqNovo***. Esta função aplica, mapeia, uma função *booleana* de teste de caractere em cada caractere de uma *string*. Sendo assim foi criada uma função *booleana* que testa se o caracter é de "saltar linha" ('\n') chamada ***"testaBarraN"*** Enquanto esta função, no caso ***"testaBarraN"*** devolve falso (*False*) um *token* é formado, o mesmo é reconhecido quando a função *booleana* retorna verdade (*True*), no caso, quando encontra um separador. Feito isto o separador é eliminado (quantos existirem em seqüência) e o reconhecimento de um novo *token* é iniciado, repetindo-se o processo até chegar no fim da String. Deste modo a função retorna uma lista de Strings → [{#Char}]. Veja o exemplo a seguir:

```
module testes1

import StdEnv, StdIO, StdArqNovo

Start = StringTokens testaBarraN "Carlos \n Menezes \n Junior"

press any key to exit
["Carlos "," Menezes "," Junior"]
```

figura 4.45 – exemplo de implementação da função ***"StringTokens"***

3 – Separa-se as Strings contidas na variável ***"listaTexto1"*** pelo caracter "virgula" (' , ') mapeando a função ***"StringTokens"*** em cada String da lista . Deste modo tem-se uma lista de lista de String → [[{#Char}]]. O resultado é armazenado na variável ***"listaTexto2"***. Veja o exemplo a seguir:

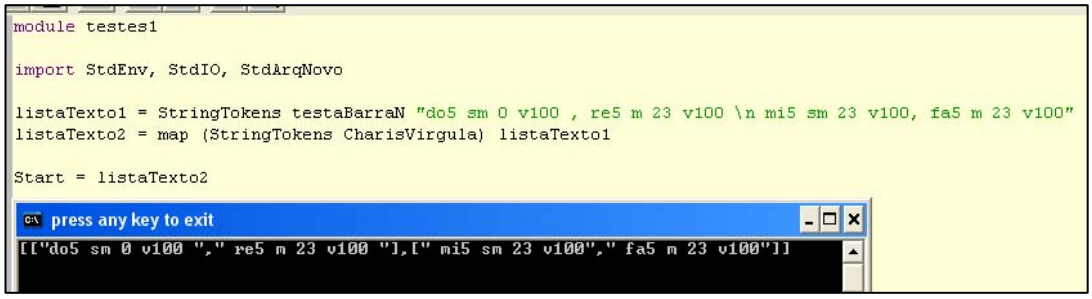


*figura 4.46 – exemplo de implementação da função **"StringTokens"** em Strings contidas em uma lista*

4 – Separa-se as Strings contidas na variável ***"listaTexto2"*** pelo caracter "espaço" (' ') mapeando a função ***"StringTokens"*** dentro da lista de lista de Strings. Deste modo tem-se uma lista de lista de lista de String → [[[{#Char}]]]. Sendo assim cada lista representa um evento de nota contendo os 4 (quatro) argumentos já apresentados anteriormente, por exemplo → [[["do5", "sm", "0", "v100"], ["re5", "m", "23", "v127"]]]. O resultado é armazenado na variável ***"listaTexto3"***. Veja exemplo abaixo:

*figura 4.47 – exemplo de implementação da função **"StringTokens"** em Strings contidas em uma lista de lista*

5 – Cria-se uma lista de Strings onde cada elemento é uma das Strings contidas na variável ***"listaTexto3"***. Para isto utilizou-se a função ***"flatten"*** que junta lista de lista em uma lista única. O resultado é armazenado na variável ***"listaTexto4"***. Veja exemplo a seguir:

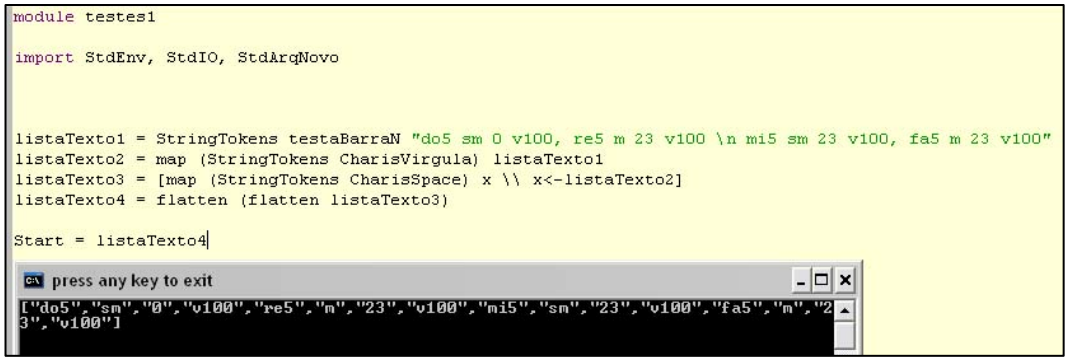


*figura 4.48 – lista de Strings contida na váriável **"listaTexto4"***

6 – Aplica-se as funções que criam as listas que servem de argumento de entrada para a função ***"geraMidiF1"*** que será apresentada com detalhes mais adiante. Estas funções são as seguintes:

- ***"listaNotaAcorde"*** → Testa se é nota melódica ou acorde e cria a lista com a codificação padronizada para ser utilizada com a função ***"geraMidiF1"***. Para isto ela analisa os elementos da variável ***"listaTexto3"***. Quando o tamanho da lista que representa o evento de notas é igual a 4, é interpretado como nota melódica. Quando o tamanho da lista é 8, 12, 16, 20 ou 24 é interpretado como sendo notas de acorde.
- ***"listaInstrumentos"*** → Pega os elementos que representam o número dos instrumentos contidos na variável ***"listaTexto4"*** e cria a lista com a codificação padronizada para ser utilizada com a função ***"geraMidiF1"***. Para isto é utilizada uma expressão Zermelo-Frankel que pega sempre o terceiro elemento de cada evento de nota e os converte nos números correspondentes.
- ***"listaFiguras"*** → Pega os elementos que representam o tipo de figura rítmica contidos na variável ***"listaTexto4"*** e cria a lista com a codificação padronizada para ser utilizada com a função ***"geraMidiF1"***. Para isto é utilizada uma expressão Zermelo-Frankel que pega sempre o segundo elemento de cada evento de nota e os converte nos números correspondentes.
- ***"listaNotas"*** → Pega os elementos que representam a nota musical contidos na variável ***"listaTexto4"*** e cria a lista com a codificação padronizada para ser utilizada com a função ***"geraMidiF1"***. Para isto é

utilizada uma expressão Zermelo-Frankel que pega sempre o primeiro elemento de cada evento de nota e os converte nos números correspondentes.

● ***"listaVolumes"*** → Pega os elementos que representam a nota musical contidos na variável ***"listaTexto4"*** e cria a lista com a codificação padronizada para ser utilizada com a função ***"geraMidiF1"***. Para isto é utilizada uma expressão Zermelo-Frankel que pega sempre o primeiro elemento de cada evento de nota e os converte nos números correspondentes.

● ***"listaCanal"*** → Cria uma lista com os canais MIDI ao qual cada nota será executadas. No caso deste trabalho todas as notas utilizarão o canal 0 (zero).

O código destas funções é apresentado a seguir:

```
//lista com código 1 caso seja nota, 0 ou 2 se for acorde // entra com listaTexto3 [[string]] : exemplo ->[["do5", "m", "0","v100"],["re5",...]]
listaNotaAcorde x
#lt= [[(length y)/4] \\ y<-(flatten x)]
= listaNotaAcordeAux lt [] 0

listaNotaAcordeAux lt nlt num
|lt==[] = nlt
|num==2 = listaNotaAcordeAux (tl lt) (nlt++(converteDois (hd lt))) 0
|otherwise = listaNotaAcordeAux (tl lt) (nlt++(converteZero (hd lt))) 2

converteZero x
|x==[1] = [1]
|otherwise = [y \\ y<-[0..0], i<-[1,2..(hd x)]]

converteDois x
|x==[1] = [1]
|otherwise = [y \\ y<-[2..2], i<-[1,2..(hd x)]]
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
//lista com código do instrumento // entra com listaTexto4 [string] : exemplo ->["do5", "m", "0","v100","re5",...]
listaInstrumentoString x= [(x!!i) \\ i<-[2,6..((length x)-1)]]
listaInstrumentos x= map toInt (listaInstrumentoString x)
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
//lista com código das figuras // entra com listaTexto4 [string] : exemplo ->["do5", "m", "0","v100","re5",...]
listaFigurasString x= [(x!!i) \\ i<-[1,5..((length x)-1)]]
listaFiguras x= map codigoFigura (listaFigurasString x)
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
//lista com codigo das notas // entra com listaTexto4 [string] : exemplo ->["do5", "m", "0","v100","re5",...]
listaNotasString x= [(x!!i) \\ i<-[0,4..((length x)-1)]]
listaNotas x= map codigoNota (listaNotasString x)
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
//lista com código do volume // entra com listaTexto4 [string] : exemplo ->["do5", "m", "0","v100","re5",...]
listaVolumeString x= [(x!!i):=(0,'0') \\ i<-[3,7..((length x)-1)]]
listaVolumes x= map toInt (listaVolumeString x)
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
//lista com código do canal, a princípio tudo ficará no canal 0 // entra com listaTexto4 [string] : exemplo ->["do5", "m", "0","v100","re5",...]
listaCanal x=  [y \\ y<-[0..0], i<-[0,4..((length x)-1)]]
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

figura 4.49 – Código das funções ***"listaNotaAcorde"*** , ***"listaInstrumentos"*** , ***"listaFiguras"*** , ***"listaNotas"*** , ***"listaVolumes"*** , ***"listaCanal"*** .

7 – Chama-se a função ***"gerar2"*** tendo como argumento de entrada a variável ***"musicaListaF1"*** contendo todas as listas necessárias para a criação do SMF, inclusive o número do metrônomo contido no popup e convertido para número inteiro. Por sua vez esta função chama outra função, a ***"geraMidiF1"*** que gera o arquivo MIDI e salva no disco C com o nome ***"defaultGeraMidiF1.mid".***

8 – Chama-se o renderizador ***"Timidity++"*** que renderiza o arquivo "***defaultGeraMidiF1.mid"*** utilizando o SoundFonts padrão do programa. O arquivo é salvo no disco C com o nome "***defaultGeraMidiF1.wav".*** Logo em seguida este arquivo sonoro é executado para que o usuário possa ouvir a música.

```
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
gerar2 mus proc

# listaMIDI = geraMidiF1 mus
# arqSalvar = map toChar listaMIDI
# vetorSalva = {x\\x<-arqSalvar}
# (_,file3, proc)= fopen "C:\\defaultGeraMidiF1.mid" FWriteData proc
# file3 = fwrites vetorSalva file3
# (_, proc)= fclose file3 proc
# (_, proc) = OSStopMusic proc
# (_, proc) = stopWav proc

# (a,b,proc) = Execute ((juntaListaStringPath(init(pathDividido)))+++"timidity-con.exe -s 44100 -Ow \"C:\\defaultGeraMidiF1.mid\"") proc
# (_, proc) = playWav "C:\\defaultGeraMidiF1.wav" proc

= (listaMIDI,proc)
//xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

*figura 4.50 – Código das funções **"gerar2"***

4.8.1 – A Função *"geraMidiF1"*

Esta função está contida na biblioteca ***"geraMidiF1.icl"*** desenvolvida por CAMARGO (2007) e aprimorada neste trabalho. Ela é utilizada para geração de arquivos SMF formato 1 em baixo nível. Segundo CAMARGO (2007) ela possui as seguintes potencialidades:

1. Facilita ao usuário entrar com dados de listas, aderentes ao programador e à manipulação por uma linguagem funcional, onde cada lista é um track a ser implementado pelo programa (cada lista é uma lista similar à lista manipulada pelo programa gera MIDI formato 0);
2. Trata automaticamente notas solos e notas em acordes, colocando delta times de forma a efetivar o estado da nota (solo ou acorde);
3. Trata pausas de figuras musicais;

4. Trata notas solos e acordes de forma mais completa, permitindo partir de um acorde e iniciar um novo acorde, acrescentando mais um identificador na lista de status da nota;
5. Calcula o meta evento de tempo dado o valor do metrônomo e a ppq;
6. Monta uma lista contendo o arquivo MIDI já com o cabeçalho, os *tracks* com seu contador de Bytes, fim de *track* e *label*.

Neste trabalho foram introduzidos alguns recursos na função geraMIDIF1, são eles:
- Capacidade de gerar todas as figuras rítmicas com ponto de aumento[6] tais como, mínima pontuada (𝅗𝅥.) , semínima pontuada (♩.), colcheia pontuada (♪.), etc.
- Capacidade de gerar tercinas[7].
- Capacidade de calcular o delta time de forma independente da formula de compasso, tornando-o mais eficiente.

O fluxograma, a seguir, ilustra a lógica de implementação da biblioteca ***geraMIDIF1***.

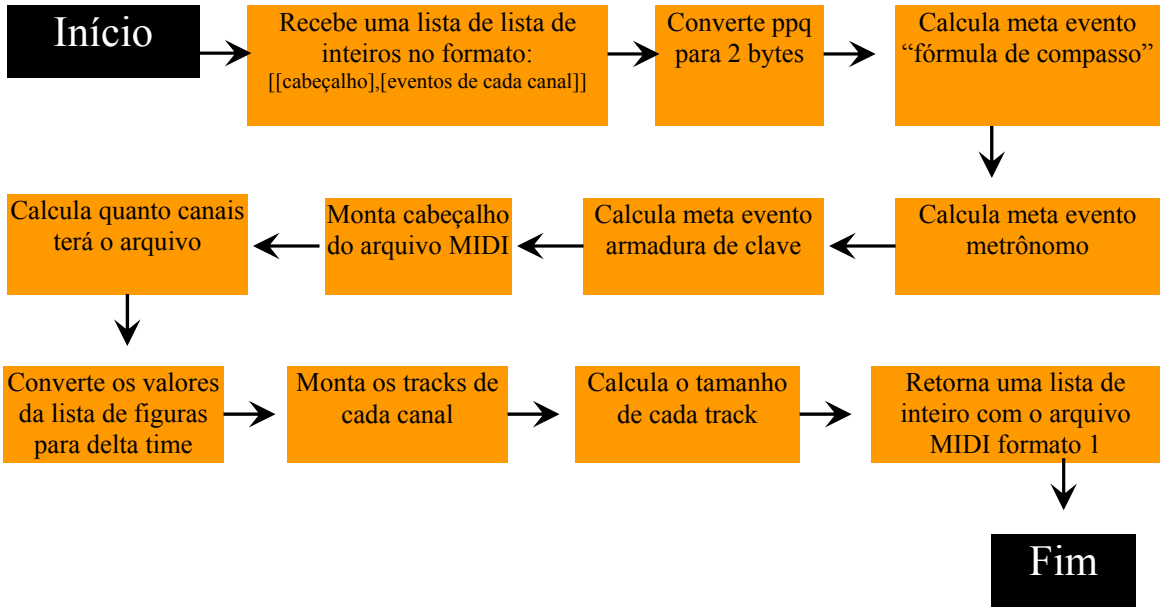


Figura 4.51 - Fluxograma da função ***geraMIDIF1***

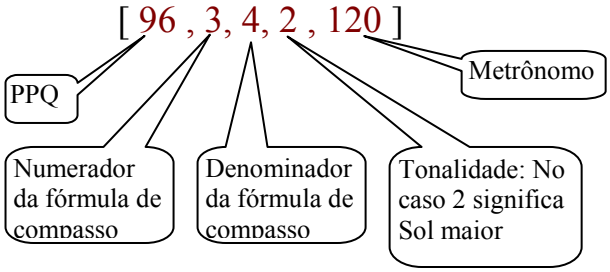
[6] Ponto de aumento é um sinal que, colocado à direita de uma nota ou pausa, aumenta metade de sua duração.
[7] Quiáltera de três notas – conjunto de três figuras rítmicas iguais que valem por dois da mesma espécie.

A função ***geraMidiF1*** possui 1 (um) argumento: uma lista de lista de listas. Nesta estão contidos o cabeçalho e os tracks da seguinte forma:

[[cabeçalho] , [Track 1] , [Track 2] … [Track n]]

Onde:

● [cabeçalho] – Estão contidas as informações tais como PPQ, formulas de compasso, tonalidade e metrônomo. Veja exemplo a seguir:



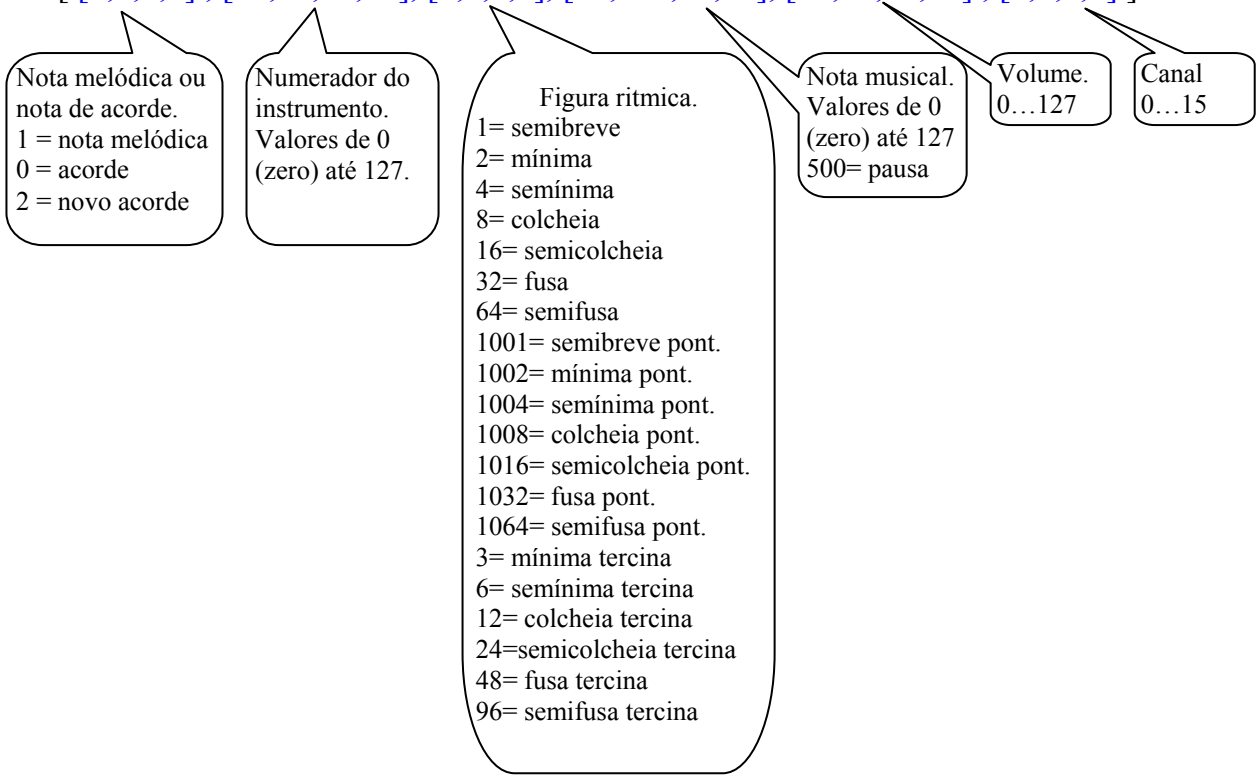
No caso da tonalidade, o código de cada uma delas é o seguinte:

0	→ Dó maior	**1**	→ Lá menor
2	→ Sol maior	**3**	→ Mi menor
4	→ Re maior	**5**	→ Si menor
6	→ La maior	**7**	→ Fá # menor
8	→ Mi maior	**9**	→ Dó # menor
10	→ Si maior	**11**	→ Sol # menor
12	→ Fa # maior	**13**	→ Re # menor
14	→ Dó # maior	**15**	→ La # menor
16	→ Fá maior	**17**	→ Re menor
18	→ Si b maior	**19**	→ Sol menor
20	→ Mi b maior	**21**	→ Dó menor
22	→ La b maior	**23**	→ Fá menor
24	→ Re b maior	**25**	→ Si b menor
26	→ Sol b maior	**27**	→ Mi b menor
28	→ Do b maior	**29**	→ La b menor

Tabela 4.2 – código utilizado para identificar a tonalidade

● [Track] – Estão contidas as listas com as seguintes informações de eventos de notas conforme exemplo a seguir:

[[1,1,0,0] , [23,23,23,23], [4,4,8,8], [60,500,62,65], [90,90,90,90] , [0,0,0,0]]



Veja a seguir o trecho de código da função ***geraMidiF1.***

```
 9 geraMidiF1 :: [ [ [Int]] ]-> [Int]
10 geraMidiF1 musica = [77,84,104,100,0,0,0,6,0,1,0,length musica]++
11                      ppqCabPrinc ++
12                      trackMetaEvento ++
13                      (processaCanais musica)++
14                      [72,67,74,38,76,86,76]
15 where
16   cab                = (hd (musica!!0))
17   ppqCabPrinc        = (transInt2B (toInt(hd(hd (musica!!0)))))
18   fCompasso          = formulaCompasso (toInt(cab!!1)) (toInt(cab!!2))
19   byteMetronomo      = metronomoBytes (toInt(cab!!4))
20   metaEventoMetronomo = [0,255,81,3] ++ byteMetronomo
21   metaEventoKeySign  = mostraTonalidade (toInt(cab!!3))
22   cabe2              = [77,84,114,107]
23   cabe3              = fCompasso
24   cabe4              = metaEventoKeySign
25   cabe5              = metaEventoMetronomo
26   nbytes             = (length cabe3) + (length cabe4) + (length cabe5) + (length fimArq)
27   fimArq             = [0,255,47,0]
28   trackMetaEvento    = cabe2 ++ (transInt4B(nbytes)) ++cabe3++cabe4++cabe5++fimArq
```

*Figura 4.52 – Função **geraMidiF1***

Com isso encerra-se aqui este capítulo que se encarregou de mostrar como utilizar alguns recursos que a linguagem Clean oferece para manipulação dos elementos do domínio musical. Foram apresentadas técnicas para implementação de aplicativos voltados para converter MIDI para Wave utilizando SoundFonts, interfaces interativas utilizando bitmaps e compiladores Texto para MIDI.

Capítulo 5

Implementação do editor MIDI para violão com articulação humanizada nota a nota

Neste capítulo serão apresentadas as técnicas de implementação do editor MIDI para violão com articulação humanizada nota a nota em linguagem funcional Clean, foco deste trabalho. Este aplicativo oferece ao usuário um conjunto de recursos que permite a criação e edição de seqüências MIDI para violão de uma maneira prática e direta. Estes recursos são os seguintes:

- Entrar com as notas através do braço do violão, facilitando sua edição.
- Escolher entre 30 trimbes e articulações com qualidade profissional tais como violão de nylon acústico, violão de nylon elétrico, violão de aço, mordentes, aporjaturas, arrastados, entre outros.
- Entrar com notas melódicas ou montar acordes em qualquer disposição.
- Visualização das notas em pentagrama com recurso de Scroll para direita e esquerda.
- Opção de modificar timbres em notas específicas de um acorde.
- Utilização de Toolbars para escolha das figuras rítmicas
- Edição em campo de texto, permitindo ao usuário criar as seqüências musicais sem utilizar o mouse.
- Ouvir as seqüências e modificá-las em qualquer ponto.
- Salvar as seqüências em formato Texto. Desta forma o usuário pode armazenar suas seqüências em um arquivo extremamente compacto, facilitando o transporte e

o compartilhamneto pela internet, inclusive sendo totalmente viável entre usuários que não possuem banda larga.

- Abrir as seqüências salvas em formato texto
- Salvar as seqüências diretamente em formato WAVE.
- Desfazer (Undo) e refazer (Redo).

5.1 – A interface gráfica e seus componentes

Buscou-se criar uma interface gráfica que apresente os recursos citados acima de uma forma direta e de fácil visualização. Veja na figura 1 a interface do programa e na figura 2 a discriminação de suas partes componentes.

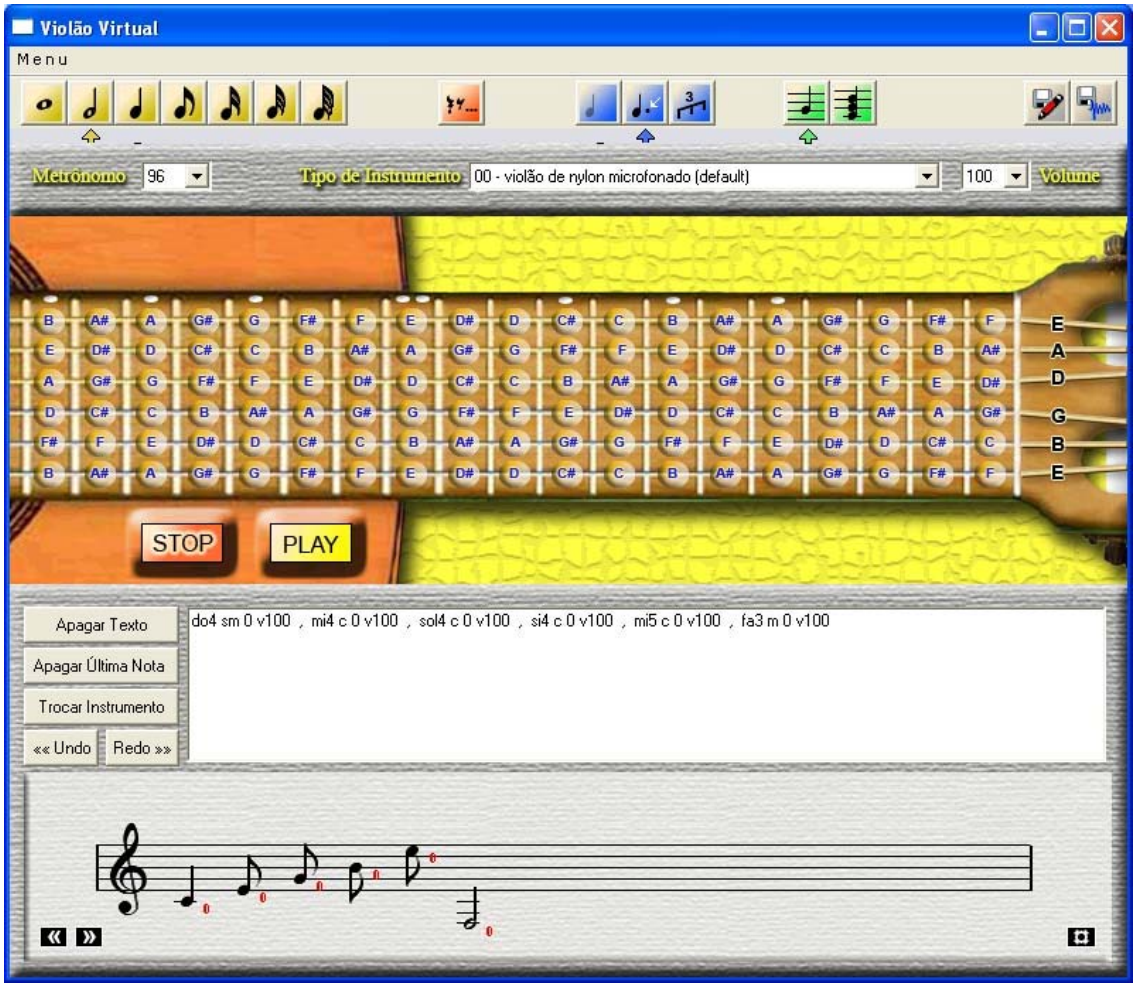


figura 5.1 - Interface do programa

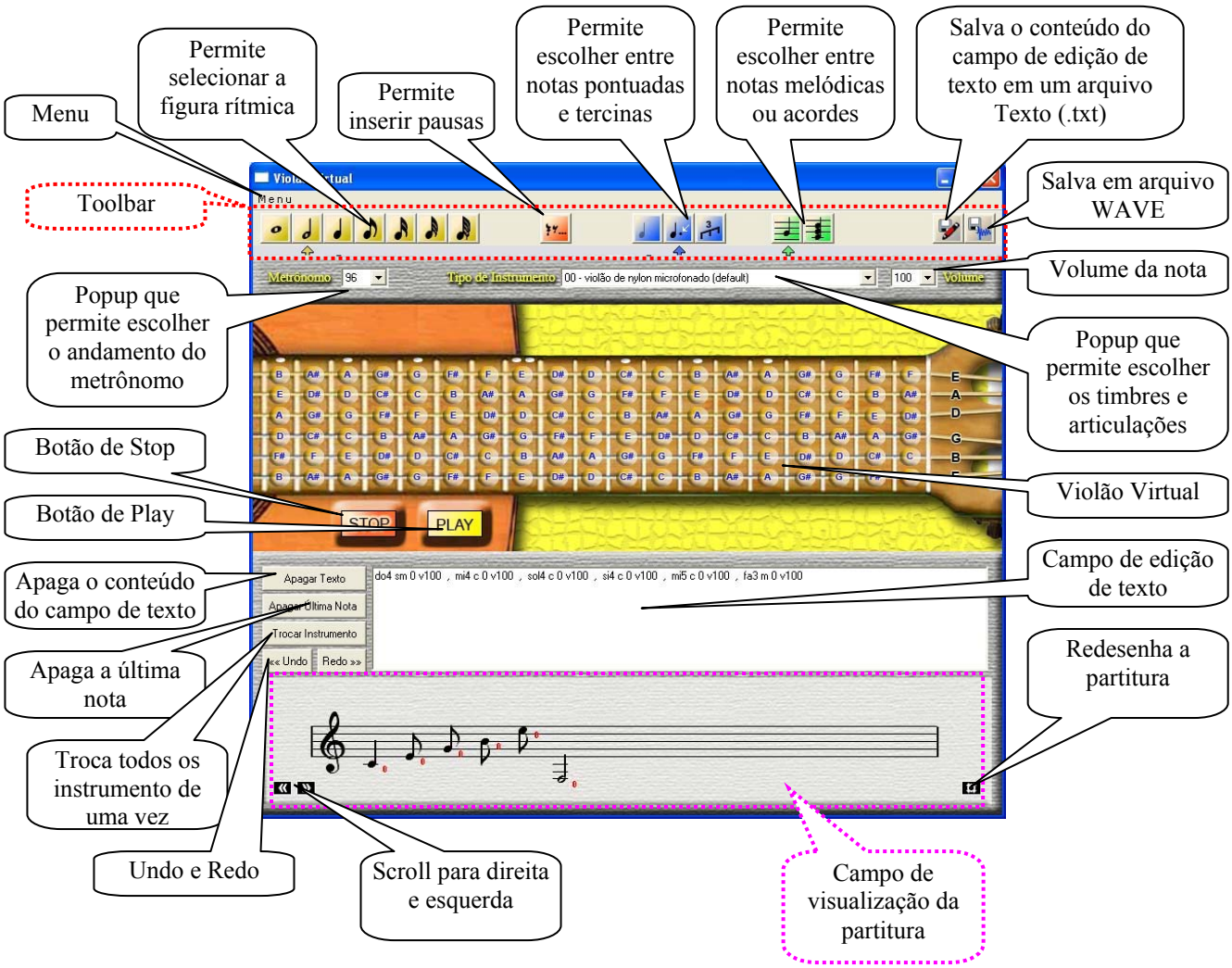


figura 5.2 – Componentes da Interface do programa

5.1.1 – Descrição de cada componente do aplicativo

1-Menu

Pode-se escolher entre abrir um arquivo texto ou sair do programa.

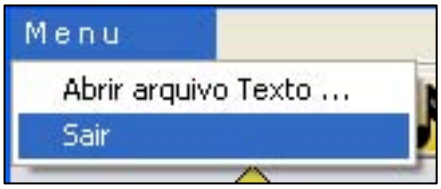


figura 5.3 – opções do Menu

O código que cria o menu é o seguinte:

```
354 initialize ids proc=: (ls=(violao))
355     #(certo1, arq1, proc) = fopen "violoes_mestrado.sf2" FReadData proc
356     #(conteudo1, arq1) = freads arq1 70000000
357     #(certo1, proc) = fclose arq1 proc
358     # (ok,file, proc)= fopen ("c:\\soundfont1.sf2") FWriteData proc
359     | not ok = proc
360     # file = fwrites conteudo1 file
361     # (_, proc)= fclose file proc
362
363     # bitmapsize = getBitmapSize violao
364     # (erro, proc) = openWindow Void (janela ids violao bitmapsize) proc
365     | erro <> NoError =  proc
366
367     # (erro, proc) = openMenu Void (Opcoes ids) proc
368     | erro <> NoError = openNotice (Notice ["Erro ao abrir o menu do Programa"] (NoticeButton "OK" id) []) proc
369     | otherwise = proc
370
371 where
372         Opcoes ids = Menu "&M e n u          "
373             ((MenuItem "&Abrir arquivo Texto ..."       [MenuFunction (noLS (abrirTextoMenu ids))])
374             :+:
375             (MenuItem "&Sair" [MenuFunction close])) []
376             where
377                 close (el,proc)
378                 # proc = openNotice (Notice ["Deseja realmente sair?"]
379                                 (NoticeButton "&Sim" (noLS closeProcess))
380                                 [NoticeButton "&Não" id]) proc
381                 =(el,proc)
```

figura 5.4 – Código que cria o menu

2-Escolha da figura rítmica

Ao clicar na figura rítmica desejada a seta logo abaixo indicará que esta figura foi selecionada e será acrescentada assim que o usuário clicar em uma das notas do Violão Virtual.

figura 5.5 – Figuras rítmicas

Estes botões estão inseridos no Toolbar (ou barra de ferramentas), assim como o botão de pausas, os botões de figuras pontuadas e tercinas, o botão de salvar texto e o botão de salvar WAVE. O código que cria o Toolbar é apresentado a seguir:

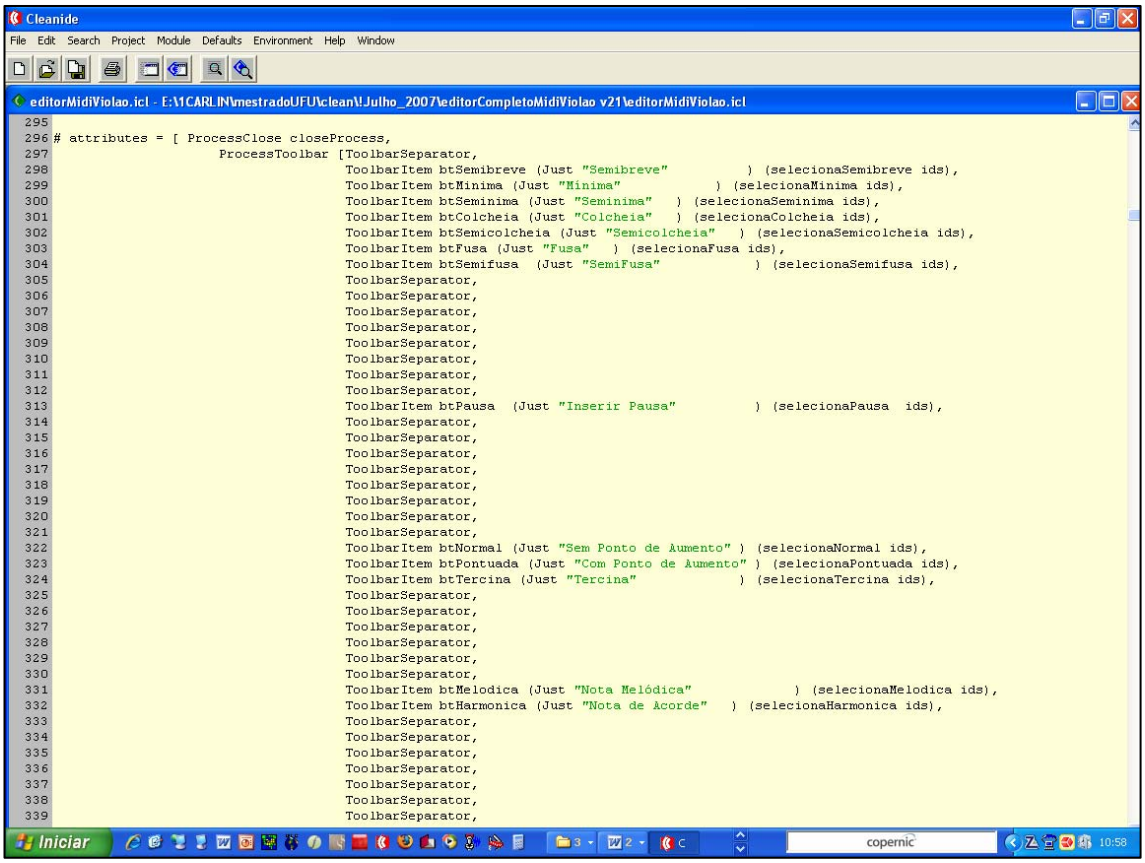


figura 5.6 – Código que cria o Toolbar

Para inserir os bitmaps nestes botões é preciso criar uma variável global no registro e importar o respectivo bitmap para esta variável, pois quando é criado um item no Toolbar é solicitado como um dos argumentos de entrada um bitmap. O processo de criação de Toolbar está ricamente detalhado no trabalho de CAMARGO (2007) fornecido como material de apoio contido no CD, não sendo necessário inseri-lo no corpo desta dissertação.

3-Inserir pausa

Ao clicar no botão inserir pausa o aplicativo insere a pausa correspondente a figura rítmica selecionada.

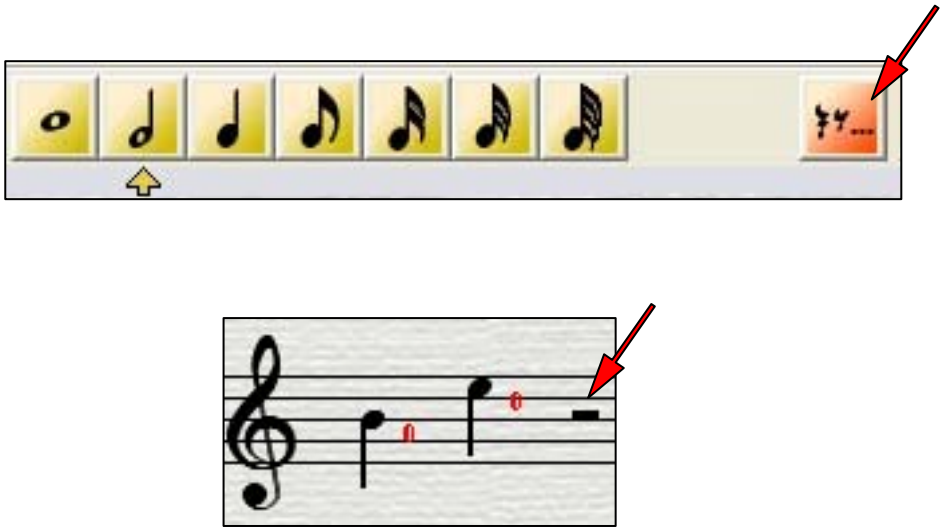


figura 5.7 – Botão inserir pausa, neste caso inserindo uma pausa de mínima

A função que insere a pausa chama-se ***"selecionaPausa"***. O código é apresentado a seguir.

```
selecionaPausa  ids  proc =: (ls=(setaVermelha,cinzaPausas,figuraSelec,tipoFigSelec,coordenadaX1))
# (Just dial, proc) = accPIO(getWindow (ids!!3)) proc
# (_,Just textoEntrada) = getControlText (ids!!0) dial
# novaNotaTexto = ", " +++ "p" +++ " " +++ figuraSelec +++ tipoFigSelec +++ " " +++ "0" +++ " " +++ "v0"
# textoSaida = textoEntrada +++ novaNotaTexto
# proc =appPIO (setControlText (ids!!0) textoSaida) proc
# proc = PlotarQuinzeNotas ids proc
= (ls = (proc.ls& coordenadaX1= (coordenadaX1 + 40)), io = proc.io)
```

figura 5.8 – código da função que ***"selecionaPausa"***

Esta função insere a pausa no campo de edição de texto e posteriormente plota a mesma no pentagrama utilizando a função ***"PlotarQuinzeNotas"***. Esta função será abordada com detalhes mais adiante.

4-Selecionar figura pontuada e tercina

Estes botões permitem inserir ponto de aumento nas figuras rítmicas ou transformá-las em tercinas. Veja exemplo abaixo:

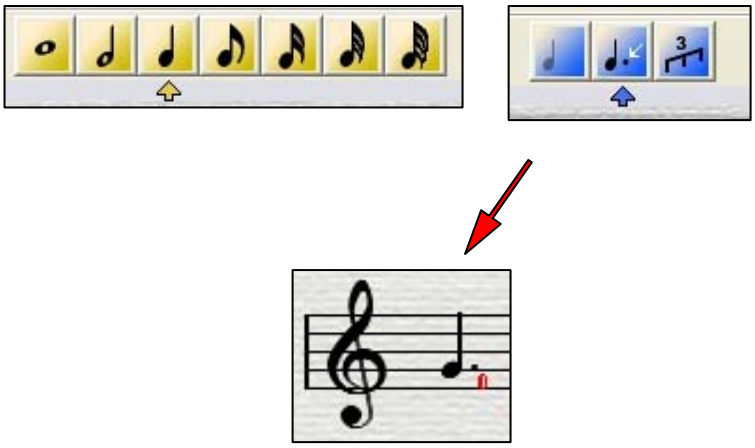


Figura 5.9 – Botão que seleciona se a figura rítmica será normal, pontuada ou tercina

5-Selecionar nota melódica ou nota de acorde

Estes botões permitem selecionar se a nota que estará sendo inserida é nota melódica ou nota de acorde.

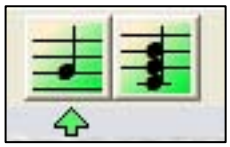


Figura 5.10 – Botão que seleciona se é nota melódica ou nota de acorde

6-Salva em arquivo Texto

Este botão permite salvar o conteúdo do campo de edição de texto em um arquivo texto (.txt).

Figura 5.11 – Botão que salva em arquivo formato texto (.txt)

O código da função que este botão chama é o seguinte:

```
848 //xxxxx FUNÇÃO QUE SALVA O ARQUIVO TEXTO ATRAVES DO EXPLORER
849 salvarTexto ids proc
850 # (Just dial, proc) = accPIO(getWindow (ids!!3)) proc
851 # (_,Just texto) = getControlText (ids!!0) dial
852 # (talvez, proc) = selectOutputFile  "Salvar Arquivo Texto"  "*.txt" proc
853 | isNothing talvez= proc
854 # (_,arquivoTexto, proc)= fopen (fromJust talvez) FWriteData proc
855 # arquivoTexto = fwrites texto arquivoTexto
856 # (_, proc)= fclose arquivoTexto proc
857 = proc
```

Figura 5.12 – Código da função que salva o arquivo Texto

O processo de salvar um arquivo de dados já foi abordado no capítulo 4.

7-Salva em arquivo Wave

Este botão permite salvar o trabalho em arquivo WAVE.

Figura 5.13 – botão que salvar em arquivo WAVE

```
837 //xxxxx FUNÇÃO QUE SALVA O ARQUIVO WAVE ATRAVES DO EXPLORER
838 salvarWave ids proc
839 # (talvez, proc) = selectOutputFile  "Salvar Arquivo Wave"  "*.wav" proc
840 | isNothing talvez= proc
841 # textoBatch = "@echo off"+++ (toChar 13,toChar 10)
842     +++"copy c:\\defaultGeraMidiF1.wav "+++"\""+++(fromJust talvez)+++"\""+++ (toChar 13,toChar 10)+++"exit"
843 # (_,arquivoWave, proc)= fopen ("c:\\salvarWave.bat") FWriteData proc
844 # arquivoWave = fwrites textoBatch arquivoWave
845 # (_, proc)= fclose arquivoWave proc
846 # (a,proc) = launchExecutable2 "c:\\salvarWave.bat" [] proc
847 = proc
```

Figura 5.14 – Código da função que salva o arquivo em formato WAVE

8-Seleciona o número do metrônomo

Este popup permite selecionar o andamento da música através do número do metrônomo, quanto maior o número mais rápido é o andamento.

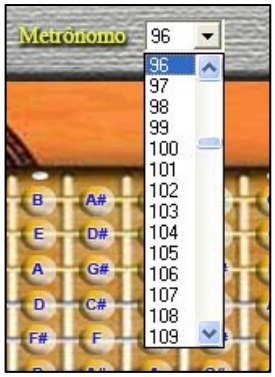


Figura 5.15 – Metrônomo

9-Seleciona o tipo de instrumento e articulação para cada nota

Este popup permite selecionar o timbre e a articulação para cada nota. No total são 30 opções diferentes de sons de violões. Todos os timbres foram gravados em estúdio e editados de forma a oferecer qualidade profissional no padrão de CD (taxa de amostragem 44.100Hz, 16 bit, stereo). Em trabalhos futuros este número pode ser expandido para 127 instrumentos.

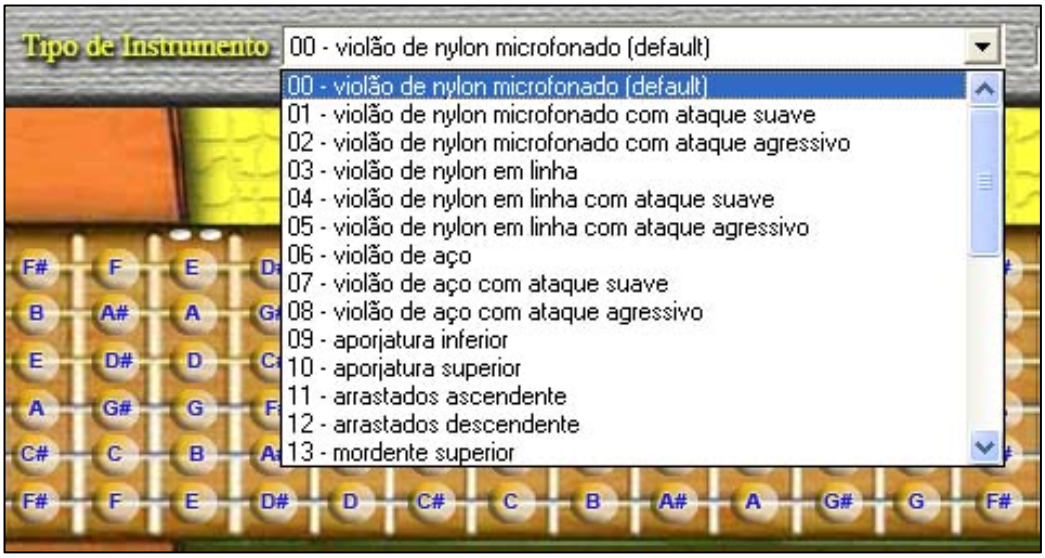


Figura 5.16 – Popup que permite selecionar o tipo de som do instrumento em cada nota

10-Seleciona o volume de cada nota

Este popup permite selecionar o volume (key velocity) de cada nota.

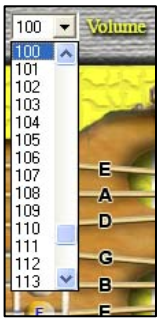


Figura 5.17 – Popup que permite selecionar o volume de cada nota

11-Violão virtual

Com o violão virtual o usuário entra com as notas clicando com o mouse diretamente no braço do violão. Quando o botão de uma das notas é acionado o programa lê as informações quanto ao tipo de figura ritmica selecionana, se é pontuada ou tercina ou normal, se é nota melódica ou de acorde, qual o tipo de instrumento e qual é o volume e então adiciona um evento de nota no campo de edição de texto, acrescenta a nota no pentagrama e toca a mesma com o timbre selecionado.

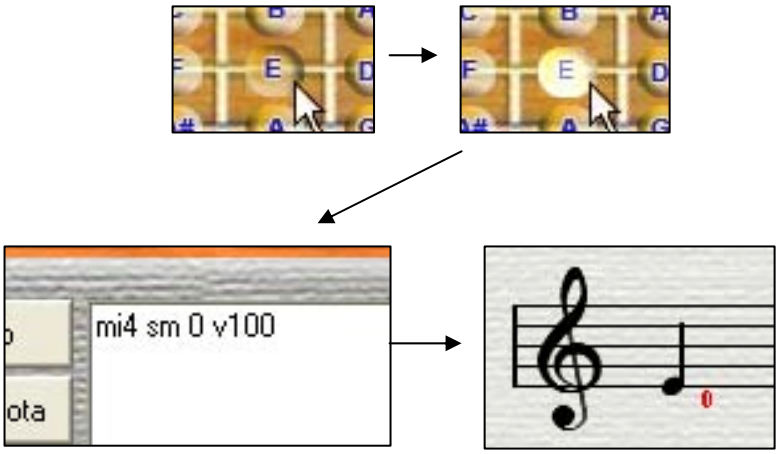


Figura 5.18 –Acrescentando notas através do violão virtual

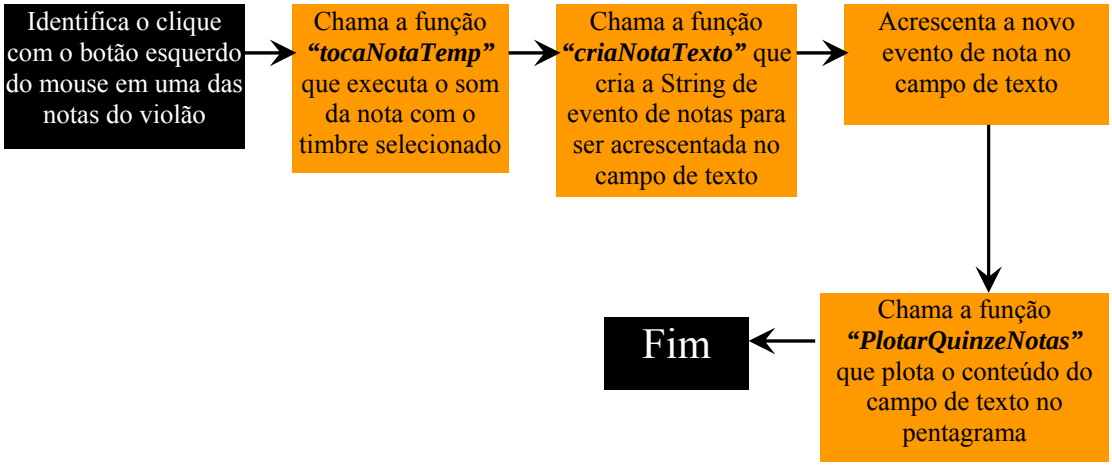


Figura 5.19 –Fluxograma de acrescentar notas utilizando o violão virtual

11- Campo de edição de texto

Com este campo o usuário pode entrar com as notas e editá-las em modo texto. A sintaxe é a mesma apresentada no item 4.8 do capítulo 4.

Figura 5.20 – Campo de edição de texto

13- Botão Apagar Texto

Este botão apaga todo o conteúdo do campo de texto.

Figura 5.21 – botão apagar texto

```
696 //xxxxx FUNÇÃO QUE APAGA O CAMPO DE TEXTO xxxxxxxxx
697     apaga (el,proc =: (ls=(undo,redo,coordenadaX1,stringPentagramaSobra)})
698     # (Just dial,proc) = accPIO(getWindow wd1) proc
699     # (_,Just textoEntrada) = getControlText t1 dial
700     |textoEntrada=="" = (el,proc)
701     # proc= appPIO (setControlText t1 "") proc
702     # undoAtual = [textoEntrada]++undo
703     # proc = PlotarQuinzeNotas ids proc
704     =(el, (ls = (proc.ls& undo=undoAtual, redo=[], coordenadaX1=120, stringPentagramaSobra = []}, io = proc.io} )
```

Figura 5.22 – Código da função que apaga o conteúdo do campo de edição de texto

14- Botão Apaga última nota

Este botão apaga a última nota ou acorde que foi inserido na seqüência musical. Ele chama a função ***"apagaUltima"***.

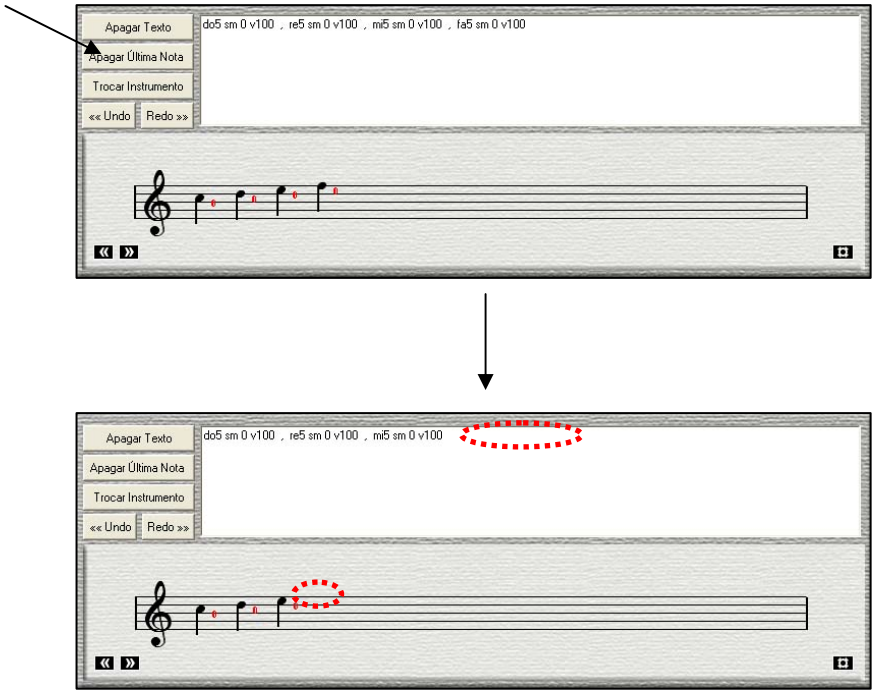


Figura 5.23 – Botão que apaga a última nota

```
626 // função que apaga a última nota
627     ApagaUltima proc =: (ls=(undo,redo,stringPentagramaSobra,stringPentagrama)}
628     # (Just dial,proc) = accPIO(getWindow wd1) proc
629     # (_,Just textoEntrada) = getControlText t1 dial
630     |textoEntrada=="" = proc
631
632     # reversoCortado = (dropWhile ((<>) ',') (reverse [x \\ x <-: textoEntrada]))
633     | reversoCortado == [] = appPIO (setControlText t1 "") proc
634
635     # semUltimaNota = toString (reverse (tl reversoCortado))
636     # proc= appPIO (setControlText t1 semUltimaNota) proc
637     # undoAtual = [textoEntrada]++undo
638     # proc = PlotarQuinzeNotas ids proc
639     = (ls = (proc.ls& undo=undoAtual, redo=[], stringPentagramaSobra = [],stringPentagrama = textoEntrada}, io = proc.io}
```

Figura 5.24 – Código da função ***"apagaUltima"***

15- Botão Trocar instrumento

Este botão troca todos os instrumentos de todas as notas. Ele chama a função ***"mudaInstrumentoGeral"***.

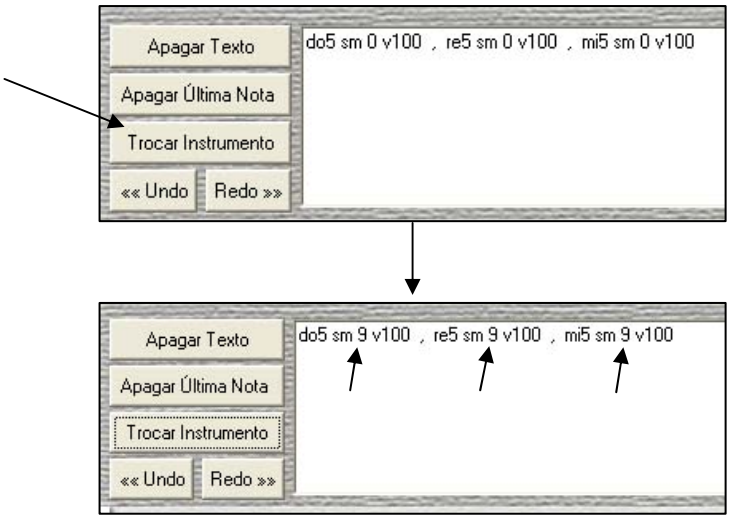


Figura 5.25 – botão que muda o instrumento de todas as notas

```
612 // função que muda todos os instrumentos
613     mudaInstrumentoGeral proc =: (ls=(undo,redo,stringPentagramaSobra,stringPentagrama))
614     # (Just dial,proc) = accPIO(getWindow wd1) proc
615     # (_,Just textoEntrada) = getControlText t1 dial
616     |textoEntrada=="" = proc
617
618     # (_,Just novoInstrumento) = getControlText popInstrum dial
619     # stringNovoInstrumento = mudaInstrumentos textoEntrada (novoInstrumento%(0,1))
620     # undoAtual = [textoEntrada]++undo
621     # proc= appPIO (setControlText t1 stringNovoInstrumento) proc
622     # proc = PlotarQuinzeNotas ids proc
623     = (ls = (proc.ls& undo=undoAtual, redo=[], stringPentagramaSobra = [],stringPentagrama = textoEntrada), io = proc.io)
624
```

Figura 5.26 – Código da função ***"mudaInstrumentoGeral"***

16- Botões Undo e Redo

Com estes botões é possível desfazer as últimas ações ou refazê-las. Eles chamam as funções ***"undoFunc"*** e ***"redoFunc"*** respectivamente.

Figura 5.27 – Botões Undo e Redo

```
587 // função undo
588     undoFunc proc =: (ls=(undo,redo,stringPentagramaSobra,stringPentagrama))
589     # (Just dial,proc) = accPIO(getWindow wd1) proc
590     # (_,Just textoEntrada) = getControlText t1 dial
591     |undo == [] = proc
592     # volta1 = hd undo
593     # undoAtual = t1 undo
594     # redoAtual = [textoEntrada]++redo
595     # proc= appPIO (setControlText t1 volta1) proc
596     # proc = PlotarQuinzeNotas ids proc
597     = (ls = (proc.ls& undo=undoAtual, redo=redoAtual,stringPentagramaSobra = [],stringPentagrama = textoEntrada), io = proc.io)
598
599 // função redo
600     redoFunc proc =: (ls=(undo,redo,stringPentagramaSobra))
601     # (Just dial,proc) = accPIO(getWindow wd1) proc
602     # (_,Just textoEntrada) = getControlText t1 dial
603     |redo == [] = proc
604     # volta1 = hd redo
605     # undoAtual = [textoEntrada]++undo
606     # redoAtual = t1 redo
607     # proc= appPIO (setControlText t1 volta1) proc
608     # proc = appPIO (setWindowLook (ids!!3) True (False, lookPentagrama)) proc
609     # proc = PlotarQuinzeNotas ids proc
610     = (ls = (proc.ls& undo=undoAtual, redo=redoAtual, stringPentagramaSobra = []), io = proc.io)
```

*Figura 5.28 – Código das funções **"undoFunc"** e **"redoFunc"***

17- Campo de visualização de partitura

Neste campo pode-se visualizar no pentagrama todas as notas que foram inseridas e seus respectivos timbres (representados pelos números de cor vermelha). O máximo de notas que podem ser visualizadas simultaneamente são 15 (quinze). Deste modo quando o usuário quer visualizar as outras notas ele deve utilizar o Scroll que faz a partitura "caminhar" nota a nota para esquerda ou direita. É possível também redesenhar o pentagrama quando isto for necessário.

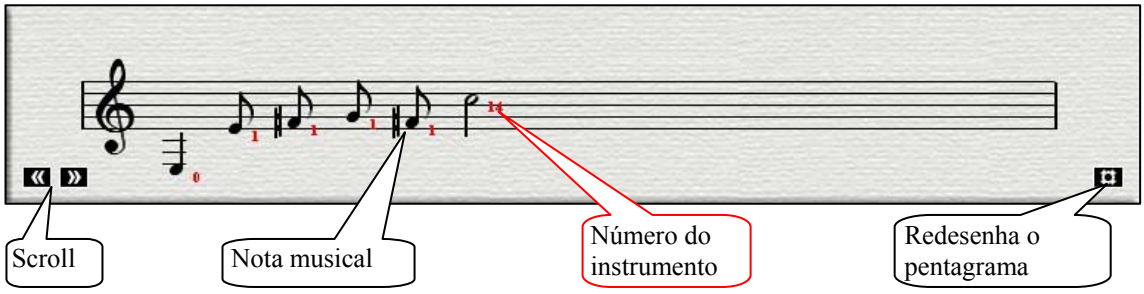


Figura 5.29 – Campo de visualização de partituras

18- Botões Play e Stop

Os botões Play e Stop servem para ouvir a seqüência musical que está sendo criada e parar a execução da mesma, respectivamente.

Figura 5.30 – botões Stop e Play

Quando o play é acionado é chamada a função ***"gerarMidi",*** já abordada no capítulo 4, que cria o arquivo MIDI, armazeno-o como temporário no disco C, converte para WAVE e toca o arquivo.

5.2 – Implementação das funções de manipulação de texto e partitura

Foram criadas algumas funções para manipulação dos dados contidos no campo de edição de texto e para visualização das notas no pentagrama, são elas:

● ***"criaNotaTexto"*** → Esta função cria a String de evento de nota formada pela nota musical, figura rítmica, número do instrumento e volume (exemplo: "do5 sm 2 v90"). Ela tem como argumento de entrada a String "nota" e o processo. Segue abaixo o fluxograma desta função.

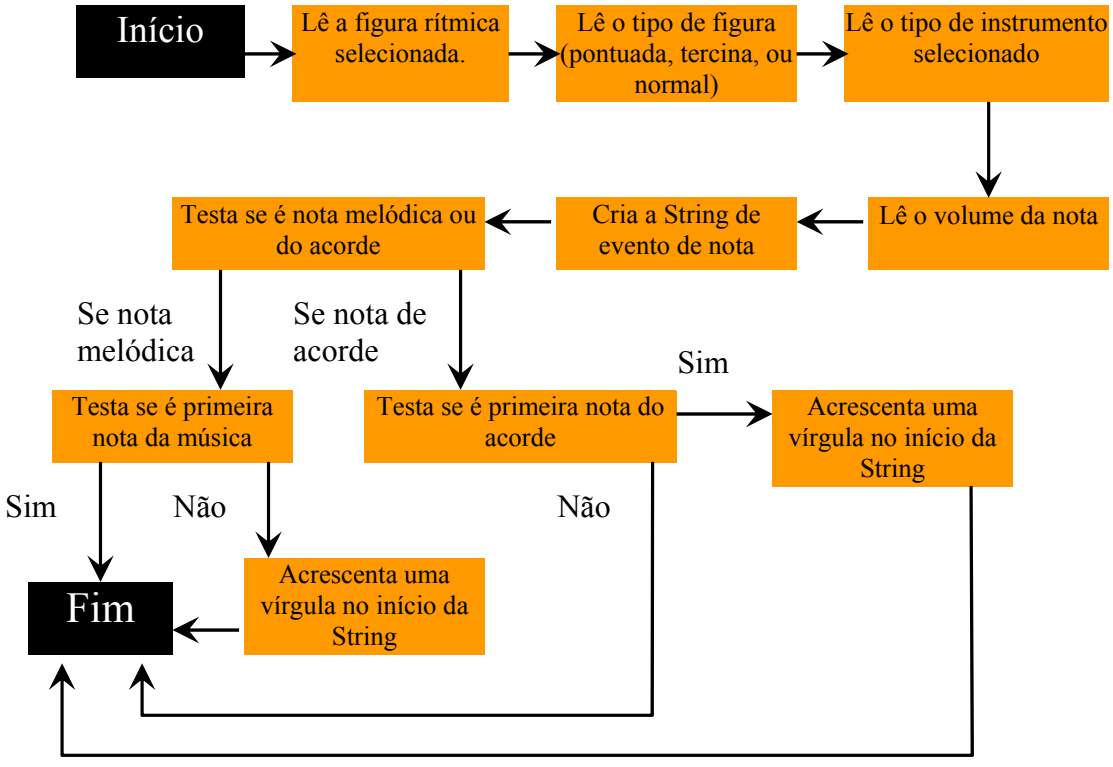


Figura 5.31 – Fluxograma da função ***"criaNotaTexto"***

Esta String é posteriormente concatenada no final da String contida no campo de edição de Texto.

```
755    criaNotaTexto nota proc =: (ls={undo,redo,figuraSelec,tipoFigSelec,tipoNotaPausaSelec,tipoMelHarmSelec,primeiraNotaAcorde})
756    # (Just dial, proc) = accPIO(getWindow wd1) proc
757    # (_,Just instrumento) = getControlText popInstrum dial
758    # (_,Just texto) = getControlText t1 dial
759    # (_,Just volume) = getControlText popVolume dial
760    # undoAtual = [texto] ++ undo
761    # instrumentoTexto = toString (toInt (instrumento%(0,1)))
762    # volumeTexto = "v" +++ volume
763    # notaTexto = nota +++ " " +++ figuraSelec +++ tipoFigSelec +++ " " +++ instrumentoTexto +++ " " +++ volumeTexto
764    | (tipoMelHarmSelec == 0)&& (primeiraNotaAcorde == 0) && (texto == "")
765        = (notaTexto,{ls = {proc.ls& primeiraNotaAcorde=1,undo=undoAtual}, io = proc.io})
766    | texto == ""
767        = (notaTexto,{ls = {proc.ls& primeiraNotaAcorde=0,undo=undoAtual}, io = proc.io})
768    | otherwise
769
770        | (tipoMelHarmSelec == 0)&& (primeiraNotaAcorde == 0)
771            = (("  ,  " +++ notaTexto),{ls = {proc.ls& primeiraNotaAcorde=1,undo=undoAtual}, io = proc.io})
772        | (tipoMelHarmSelec == 0)&& (primeiraNotaAcorde == 1)
773            = (("   " +++ notaTexto),{ls = {proc.ls& primeiraNotaAcorde=1,undo=undoAtual}, io = proc.io})
774        | (tipoMelHarmSelec == 2)&& (primeiraNotaAcorde == 0)
775            = (("  ,  " +++ notaTexto),{ls = {proc.ls& primeiraNotaAcorde=1,undo=undoAtual}, io = proc.io})
776        | (tipoMelHarmSelec == 2)&& (primeiraNotaAcorde == 1)
777            = (("   " +++ notaTexto),{ls = {proc.ls& primeiraNotaAcorde=1,undo=undoAtual}, io = proc.io})
778        | otherwise
779            = (("  ,  " +++ notaTexto),{ls = {proc.ls& primeiraNotaAcorde=0,undo=undoAtual}, io = proc.io})
780
```

*Figura 5.32 – Código da função **"criaNotaTexto"***

● ***"PlotarQuinzeNotas"*** → Esta função cria a partitura plotando as figuras musicais no pentagrama a partir dos eventos de notas contidos no campo de edição de texto. Veja a seguir o fluxograma desta função.

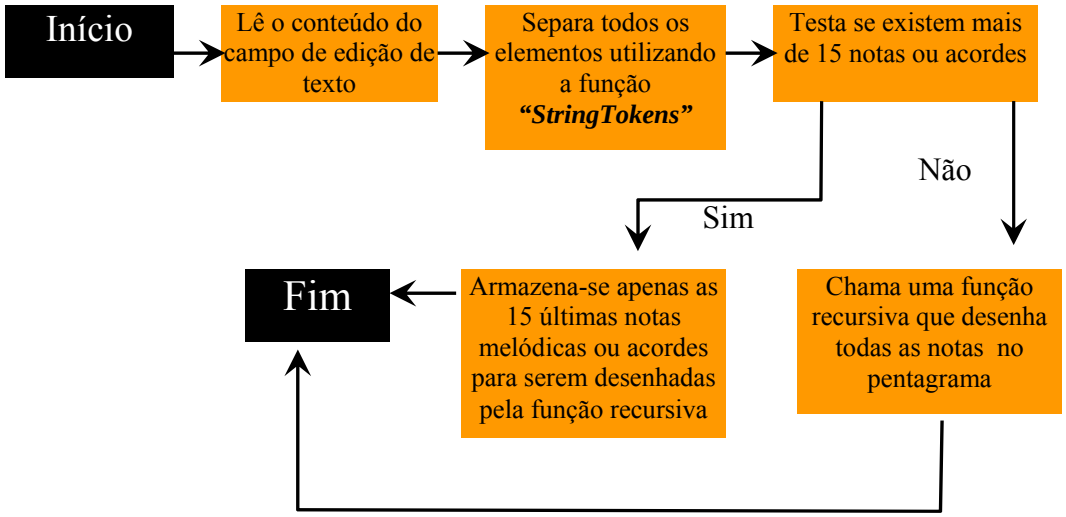


*Figura 5.33 – Fluxograma da função **"PlotarQuinzeNotas"***

O processo de desenhar notas musicais em pentagramas está ricamente detalhado no trabalho de CAMARGO (2007) fornecido como material de apoio contido no CD, não sendo necessário inseri-lo no corpo desta dissertação.

```
2447 PlotarQuinzeNotas ids proc =: (ls=(fundoPentagrama))
2448 # (Just dial, proc) = accPIO(getWindow (ids!!3)) proc
2449 # (_,Just textoEntrada) = getControlText (ids!!0) dial
2450 #listaTexto1 = StringTokens testaBarraN textoEntrada
2451 #listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
2452 #listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
2453 #listaTexto4 = flatten (flatten listaTexto3)
2454 #listaTexto5 = flatten listaTexto3
2455 #listaTexto6 = [x \\ x<-listaTexto5 | x<>[]]
2456 #numeroDeNotasEAcorde = length listaTexto6
2457 #listaTexto3SemVazios = map (removeMember []) listaTexto3
2458 |numeroDeNotasEAcorde > 15
2459     #preLista1 = StringTokens CharisVirgula textoEntrada
2460     #preLista2 = reverse (take 15 (reverse preLista1))
2461     #preLista3 = listaParaString "," preLista2
2462
2463     #listaTexto1 = StringTokens testaBarraN preLista3
2464     #listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
2465     #listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
2466     #listaTexto4 = flatten (flatten listaTexto3)
2467
2468     #notaAcordeF1 = listaNotaAcorde listaTexto3
2469     #notaAcordeF1String = [(toString x) \\ x<-notaAcordeF1]
2470
2471     #instrumentoF1 = listaInstrumentos listaTexto4
2472     #instrumentoF1String = [(toString x) \\ x<-instrumentoF1]
2473
2474     #figuraF1 = listaFiguras listaTexto4
2475     #figuraF1String = [(toString x) \\ x<-figuraF1]
2476
2477     #notaF1 = listaNotasString listaTexto4
2478
2479     #volumeF1 = listaVolumes listaTexto4
2480     #volumeF1String = [(toString x) \\ x<-volumeF1]
2481
2482     #canalF1 = listaCanal listaTexto4
2483     #canalF1String = [(toString x) \\ x<-canalF1]
2485     #listaCoordenadasX1 = geraCoordenadasX1 notaAcordeF1 120 40
2486     # proc = appPIO(appWindowPicture (ids!!3) (drawAt (x=10,y=455) fundoPentagrama)) proc
2487     # proc = atualizaRolaDirAux ids notaF1 figuraF1 listaCoordenadasX1 notaAcordeF1 instrumentoF1 proc
2488
2489     = proc
2490
2491 |otherwise
2492     #notaAcordeF1 = listaNotaAcorde listaTexto3
2493     #notaAcordeF1String = [(toString x) \\ x<-notaAcordeF1]
2494
2495
2496     #instrumentoF1 = listaInstrumentos listaTexto4
2497     #instrumentoF1String = [(toString x) \\ x<-instrumentoF1]
2498
2499
2500     #figuraF1 = listaFiguras listaTexto4
2501     #figuraF1String = [(toString x) \\ x<-figuraF1]
2502
2503     #notaF1 = listaNotasString listaTexto4
2504
2505     #volumeF1 = listaVolumes listaTexto4
2506     #volumeF1String = [(toString x) \\ x<-volumeF1]
2507
2508     #canalF1 = listaCanal listaTexto4
2509     #canalF1String = [(toString x) \\ x<-canalF1]
2510
2511     #listaCoordenadasX1 = geraCoordenadasX1 notaAcordeF1 120 40
2512     # proc = appPIO(appWindowPicture (ids!!3) (drawAt (x=10,y=455) fundoPentagrama)) proc
2513     # proc = atualizaRolaDirAux ids notaF1 figuraF1 listaCoordenadasX1 notaAcordeF1 instrumentoF1 proc
2514
2515
2516     = proc
```

*Figura 5.34 – Código da função **"PlotarQuinzeNotas"***

● ***"RolarEsquerda"*** → Esta função é chamada quando o botão de Scroll para a esquerda é acionado. Ela faz com que as notas que estão à esquerda da última nota visualizada no pentagrama passem a ser visíveis. Deste modo as outras são deslocadas para a direita. A implementação desta função é semelhante a ***"PlotarQuinzeNotas"***, a única diferença é que ela pega os 15 (quinze) eventos contidos na lista utilizada na função

"PlotarQuinzeNotas" deslocados de um índice a menos, ou seja, se numa seqüência musical contendo 22 (vinte) notas visualiza-se as 15 (quinze) últimas (da oitava até a vigésima segunda), a função ***"RolarEsquerda"*** plota as 15 (quinze) penúltimas (da sétima a vigésima primeira). Se chamada novamente, são plotadas as 15 (quinze) antepenúltimas (da sexta a vigésima), e assim por diante.

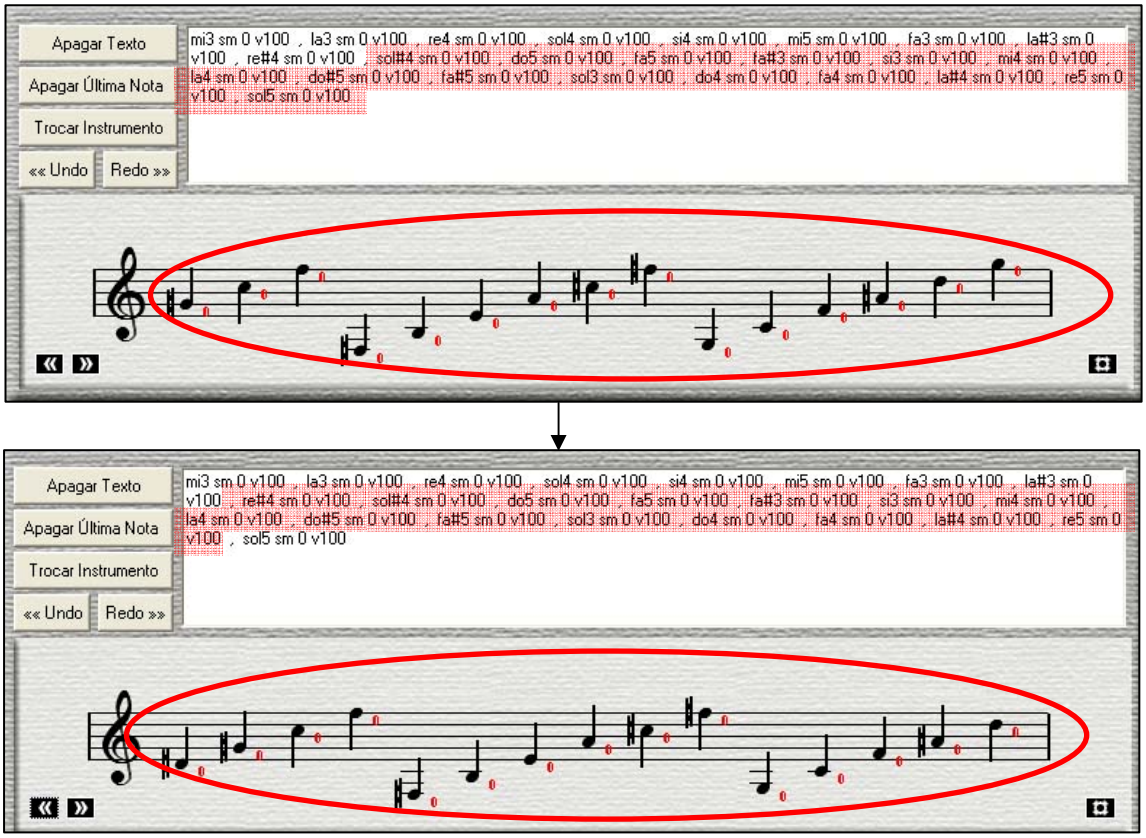


Figura 5.35 – função ***"RolarEsquerda"***

```
2520 RolarEsquerda ids proc =: (ls={fundoPentagrama, stringPentagrama, stringPentagramaSobra})
2521 # (Just dial, proc) = accPIO(getWindow (ids!!3)) proc
2522 # (_,Just textoEntrada) = getControlText (ids!!0) dial
2523
2524 #listaTexto1 = StringTokens testaBarraN textoEntrada
2525 #listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
2526 #listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
2527 #listaTexto4 = flatten (flatten listaTexto3)
2528
2529 #listaTexto5 = flatten listaTexto3
2530 #listaTexto6 = [x \\ x<-listaTexto5 | x<>[]]
2531 #numeroDeNotasEAcorde = length listaTexto6
2532 #listaTexto3SemVazios = map (removeMember []) listaTexto3
2533
2534 |numeroDeNotasEAcorde > 15
2535     #textoEntrada = stringPentagrama
2536     #preLista1 = StringTokens CharisVirgula textoEntrada
2537     #preLista2 = reverse (take 15 (drop 1 (reverse preLista1)))
2538     #preLista3 = listaParaString "," preLista2
2539
2540     #stringPentagramaNovo = listaParaString "," (init preLista1)
2541
2542     #listaTexto1 = StringTokens testaBarraN preLista3
2543     #listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
2544     #listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
2545     #listaTexto4 = flatten (flatten listaTexto3)
2546
2547     #listaTexto5 = flatten listaTexto3
2548     #listaTexto6 = [x \\ x<-listaTexto5 | x<>[]]
2549     #numeroDeNotasEAcordeRegistro = length listaTexto6
2550     |numeroDeNotasEAcordeRegistro < 15 = proc
2551
2552     #notaAcordeF1 = listaNotaAcorde listaTexto3
2553     #notaAcordeF1String = [(toString x) \\ x<-notaAcordeF1]
2554
2555     #instrumentoF1 = listaInstrumentos listaTexto4
2556     #instrumentoF1String = [(toString x) \\ x<-instrumentoF1]
```

Figura 5.36 – Trecho do código da função ***"RolarEsquerda"***

● ***"RolarDireita"*** → Esta função é chamada quando o botão de Scroll para a direita é acionado. Ela tem a mesma lógica de implementação da função ***"RolarEsquerda"***, só que ao invés de deslocar à esquerda da lista, desloca à direita.

```
2578 RolarDireita ids proc =: {ls={ fundoPentagrama, stringPentagrama, stringPentagramaSobra}}
2579 # (Just dial, proc) = accPIO(getWindow (ids!!3)) proc
2580 # (_,Just textoEntrada) = getControlText (ids!!0) dial
2581
2582 #listaTexto1 = StringTokens testaBarraN textoEntrada
2583 #listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
2584 #listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
2585 #listaTexto4 = flatten (flatten listaTexto3)
2586
2587 #listaTexto5 = flatten listaTexto3
2588 #listaTexto6 = [x \\ x<-listaTexto5 | x<>[]]
2589 #numeroDeNotasEAcorde = length listaTexto6
2590 #listaTexto3SemVazios = map (removeMember []) listaTexto3
2591
2592 |(numeroDeNotasEAcorde > 15)&&(stringPentagramaSobra <> [])
2593     #textoEntrada = stringPentagrama
2594     #preLista1 = (StringTokens CharisVirgula textoEntrada)++[(hd stringPentagramaSobra)]
2595
2596     #preLista1String = listaParaString "," preLista1
2597
2598     #preLista2 = reverse (take 15 (reverse preLista1))
2599     #preLista3 = listaParaString "," preLista2
2600
2601     #stringPentagramaNovo = preLista1String
2602
2603     #listaTexto1 = StringTokens testaBarraN preLista3
2604     #listaTexto2 = map (StringTokens CharisVirgula) listaTexto1
2605     #listaTexto3 = [map (StringTokens CharisSpace) x \\ x<-listaTexto2]
2606     #listaTexto4 = flatten (flatten listaTexto3)
2607
2608     #listaTexto5 = flatten listaTexto3
2609     #listaTexto6 = [x \\ x<-listaTexto5 | x<>[]]
2610     #numeroDeNotasEAcordeRegistro = length listaTexto6
2611     |numeroDeNotasEAcordeRegistro < 15 = proc
2612
2613     #notaAcordeF1 = listaNotaAcorde listaTexto3
2614     #notaAcordeF1String = [(toString x) \\ x<-notaAcordeF1]
2615
2616     #instrumentoF1 = listaInstrumentos listaTexto4
2617     #instrumentoF1String = [(toString x) \\ x<-instrumentoF1]
```

Figura 5.37 – Trecho do código da função ***"RolarDireita"***

Deste modo encerra-se aqui este capítulo que apresentou todos os recursos desenvolvidos na implementação deste aplicativo e as potencialidades que este programa oferece. O código completo deste software está contido no CD que é fornecido com esta dissertação.

Capítulo 6

Estudo de caso e análise comparativa com outros softwares semelhantes

Como estudo de caso optou-se por criar alguns exemplos musicais utilizando o editor MIDI para violão com articulação humanizada e compará-los com alguns softwares que oferecem recursos para realizar a mesma tarefa. Os programas escolhidos para fazer tal comparação foram dois, o Finale 2006 e o Sonar 6. Justifica-se a escolha dos mesmos por serem considerados por um grande número de profissionais os principais softwares de edição e manipulação de arquivos MIDI do mercado de estúdio digital na atualidade. O Finale 2006 tem como foco a produção de partituras e o Sonar 6 tem como foco a produção musical integrando MIDI e Áudio, porém ambos oferecem recursos de criar seqüências MIDI com timbres baseados em síntese por wavetable, além de renderiza-las para arquivos WAVE.

O resultado sonoro dos exemplos gerados pelos três aplicativos (O Editor MIDI pra violão, O Finale 2006 e o Sonar 6) foram salvos em CD e mostrados a 5 profissionais da área musical, sendo 3 deles com pós-graduação na área de Computer Music (2 mestres e 1 doutor), 1 produtor musical e dono de estúdio e 1 professor de violão. Após a audição das faixas, cada uma delas gerada por um dos aplicativo, foi perguntado qual das três faixas eles achavam que se aproximava mais da sonoridade de execução humana. Não foi informado aos participantes quais aplicativos geraram os exemplos musicais. Todos os 5 profissionais escolheram a faixa que foi gerada pelo editor MIDI para violão. Os exemplos também foram mostrados para 5 pessoas que não têm formação musical, porém são deleitantes com relação a música. O resultado foi o mesmo. Veja abaixo as partituras dos exemplos musicais gerados pelos três softwares.

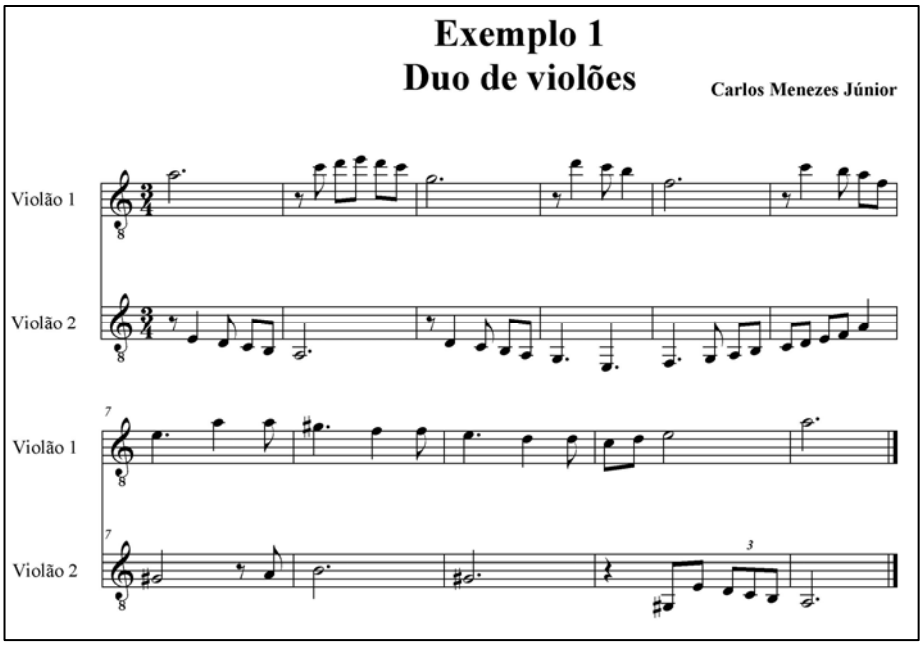


Figura 6.1 – Duo de violões

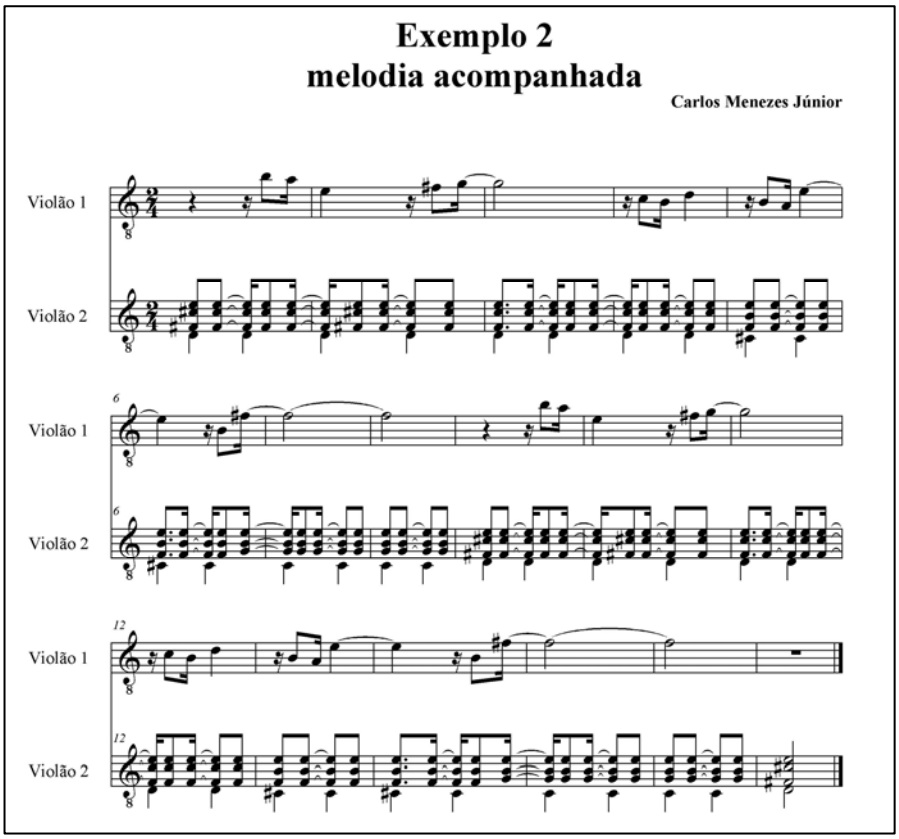


Figura 6.2 – Violão 1 executa a melodia, violão 2 executa os acordes

6.1 – Análise comparativa dos recursos oferecidos pelos três softwares

● ***Finale 2006*** → Este software oferece um grande número de feramentas necessárias para criação e edição de arquivos MIDI. Ele também trabalha com síntese por wavetable e possui um vasto banco de instrumentos, porém, no caso do violão não são oferecidas muitas opções de timbres, restringindo-se apenas àqueles previstos no padrão GM (General MIDI). Este programa possui um recurso interessante chamado "Human Playback" que procura humanizar a execução musical a partir de estilos musicais, porém exige um grau de conhecimento avançado e não é oferecido articulações humanizadas nota a nota.

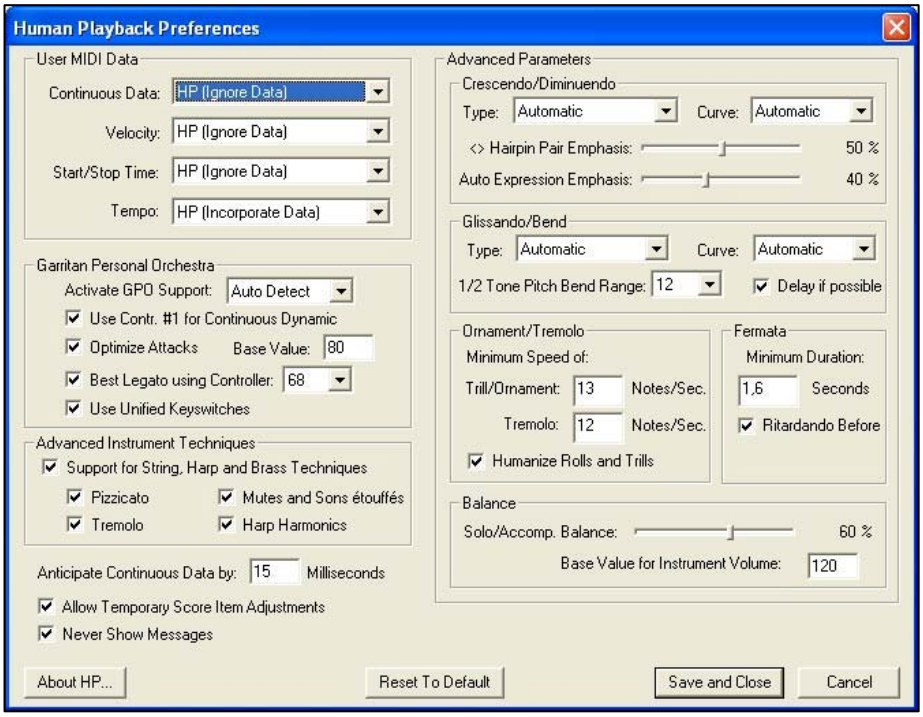


Figura 6.3 – Ferramenta "Human Playback" oferecida pelo Finale 2006.

● ***Sonar 6*** → Este software é considerado um dos principais editores MIDI. Ele disponibiliza praticamente todas as ferramentas necessárias à produção musical. Ele também trabalha com síntese por wavetable, porém assim como o Finale 2006, no caso do violão não são oferecidas muitas opções de timbres, restringindo-se apenas àqueles previstos no padrão GM (General MIDI). Ele não possui nenhum recurso de

humanização automática mas oferece plugins de instrumentos virtuais com timbres acústicos criados a partir de instrumentos reais, porém voltados para bateria. Não é oferecido nenhum instrumento virtual especificamente voltado para violão.

Figura 6.4 – Bateria virtual oferecida pelo Sonar 6.

● ***Editor MIDI para violão*** → Os recursos oferecidos pelo software desenvolvido nesta pesquisa que não são encontrados nos demais softwares apresentados acima são:

- Timbres de violões com articulações humanizadas nota a nota com qualidade acústica profissional.
- Opção de entrar com as notas através de um violão virtual interativo.
- Opção de edição em modo texto com sintaxe simples e salvar o conteúdo do trabalho em arquivos "*.txt".

Capítulo 7

Conclusão e trabalhos futuros

Este trabalho teve como foco desenvolver técnicas de implementação de um sistema voltado para edição de arquivos MIDI de violão onde cada nota pode ser configurada para executar um tipo de articulação humanizada, oferecendo ao músico, seja ele profissional ou não, mais um intrumento de criação artística baseada em sistemas digitais. Para tanto pesquisou-se soluções aplicadas ao domínio musical na área de computação sônica. As técnicas de programação apresentadas neste trabalho podem servir como material de referência para futuros pesquisadores que queiram desenvolver aplicativos voltados para manipulação e geração de conteúdos na área musical.

O resultado sonoro que este aplicativo oferece é bastante satisfatório, com avaliação positiva por parte dos profissionais que ouviram exemplos musicais produzidos com este programa, conforme demostrado no capítulo 6.

Os objetivos propostos foram cumpridos, a saber:

1) - Criar rotinas em linguagem funcional Clean voltadas para manipulação de arquivos SMF

Conforme foi demonstrado no capítulo 4, foram criadas algumas rotinas para manipulação de arquivos SMF que proporcionaram a edição dos eventos MIDI, viabilizando a implementação do aplicativo proposto nesta pesquisa.

2) - Desenvolver funções e implementar exemplos que integrem Programas em Clean com renderizadores de arquivos MIDI para Wave já consagrados e que utilizem bancos SoundFonts tais como o TiMidity++.

Conforme foi demonstrado no capítulo 4 com a implementação do conversor MIDI→WAVE e a apresentação da solução encontrada para integrar o renderizador TiMidity++ com programas feitos em Clean.

3) - Apresentar os conceitos e implementar exemplo de criação de GUI (Graphical User Interface) voltados para aplicativos multimídia em linguagem funcional Clean

Conforme foi demonstrado no capítulo 4 com a implementação do Violão Virtual, integrando técnicas de interfaces visuais com animações sensíveis ao movimento do mouse e respostas sonoras.

4) - Desenvolver técnicas de visualização de partituras em interfaces gráficas.

Conforme foi apresentado no capítulo 5 com o desenvolvimento de funções que convertem informações contidas no campo de edição de texto em partituras e que permitem inserir “Scroll” para direta e esquerda, facilitando a visualização do conteúdo musical.

5) - Desenvolver um compilador Texto -> MIDI.

Conforme foi apresentado no capítulo 4 com a implementação do compilador Texto→MIDI permitindo a criação de sequências musicais através da inserção de dados em um campo de edição de texto e com sintaxe simples e fácil de manipular.

6) - Criar um banco SoundFont editável com articulações humanizadas de violões.

Este arquivo é fornecido juntamente com o programa. Além de permitir o armazenamento de timbres com qualidade profissional, pode-se expandi-los e modifica-los. Sendo assim a utilização da tecnologia SoundFont mostrou-se como sendo uma solução perfeitamente aderente aos propósitos deste trabalho.

7) - Apresentar os conceitos de como criar um banco SoundFont com articulações humanizadas.

Conforme demonstrado no capítulo 3. Deste modo esta dissertação pode servir como material de referência para interessados que queiram desenvolver trabalhos utilizando esta tecnologia.

7.1 - Trabalhos Futuros

1 - Expandir o banco de timbres com mais articulações oferecendo um seqüênciamento mais preciso.

2 – Oferecer a opção de editar as notas através do campo de visualização da partitura.

3 – Disponibilizar a entrada de notas com ligadura de tempo.

4 – Permitir a criação e edição de mais de um violão simultaneamente.

5 – Desenvolver editores MIDI com articulações humanizadas voltadas para outros instrumentos, tais como flauta transversal, violino, clarinete, saxofone, entre outros.

6 – Oferecer a opção de abrir e editar arquivos SMF de violão criados por outros softwares.

7 – Implementar entrada de mensagens MIDI através de um controlador MIDI externo em tempo real.

8 – Oferecer a opção de organizar a visualização da partitura por compassos.

Conclui-se, assim, este trabalho.

Referências Bibliográficas

[1] MACHADO, André Campos. **Tradutor de Arquivos MIDI para Texto Utilizando Linguagem Funcional CLEAN**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2001.

[2] LIMA, Sandra Fernandes de Oliveira. **Um sistema para transposição Automática de Seqüências MIDI baseada em Alcance Vocal**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2006.

[3] LOPES, Guilherme Francisco Lopes. **Desenvolvimento de uma biblioteca e uma calculadora Midi, em linguagem funcional, para análise e manipulação de Arquivos padrão Midi (SMF) formato 0 e 1.** Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2004.

[4] **MIDI Detailed Specification 1.0.** International MIDI Association, 1983, 1991, 1994.

[5] LIMA, Luciano Vieira; RUFINO, Hugo Leonardo Pereira; GOULART, Reane Franco; CURY FILHO, Reny. **Linguagem Funcional CLEAN: uma solução moderna, eficiente e aderente à modelagem de funções e ao ensino da Matemática.** In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, São José do Rio preto, 2003, v.1, p.1-7.

[6] RUFINO, Hugo Leonardo Pereira. **Criação e Utilização de DLLs e Objs em Linguagem Funcional CLEAN e seu Interfaceamento com C**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2004.

[7] COSTA, Antônio Eduardo; **My Clean Book** (2006).

[8] LIMA, Luciano Vieira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Cakewalk SONAR 2.0.** 1a. ed. São Paulo: Editora Érica Ltda, 2002.

[9] LIMA, L.; C.A.L., et al. **Automatic Learning Composition Base don Polyphonic Wave Signals and Musical Histograms.** Tatra Mt Match. Publ. 23, 2001.

[10] COPE, D. **Experiments Musical in Intelligence (EMI): Non-linear Linguistic-Based Composition Interface**, vol., 18, pp. 117-139, 1989.

[11] COPE, D. **Computer Modelling of Musical Intelligence in EMI.** Computer Music Journal, vol. 16, no. 2, Summer, pp. 69-83, 1992.

[12] COPE, D. **Music & Lisp.** OH Expert, March, pp.26-33, 1988.

[13] LIMA, L. V. **Um Sistema de Composição Musical Dirigido por Estilo.** Tese de Doutorado, Universidade de São Paulo, São Paulo - Brasil, 1998.

[14] LIMA, Luciano Vieira. Real time high performance plataform for e-learning with high resolutions áudio, frames and vídeo file. In: **3ª. Conferência do PGL: Partnership in Global Learning "Consolidando Experiências em e-Learning",** FGV, São Paulo – Brasil, 2005.

[15] LIMA, Luciano Vieira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Cakewalk sonar 2.0 : Seqüenciamento e Técnicas de Estúdio Audiodigital.** 2ª. ed. São Paulo: Editora Érica Ltda, 2003.

[16] LIMA, Luciano Vieira; LIMA; Sandra Fernandes de Oliveira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Computação Musical : Arranjo Automático, Conversão de CD em MIDI e MIDI em Áudio utilizando o Band-in-a-Box, Virtual Sound Canvas 3.23 e Akoff Music Composer 2.0.** 1ª. ed. São Paulo: Editora Érica Ltda, 2004.

[17] LIMA, Luciano Vieira; LIMA; MACHADO, André Campos; PINTO, Marília Mazzaro. **Finale 2004: Editoração de Partituras, Composição e Arranjo.** 1ª. ed. São Paulo: Editora Érica Ltda, 2004.

[18] PASCHOAL, Fausto de. **Música no Computador com Sibelius 3 para Windows.** 1ª. ed. São Paulo: Editora Érica Ltda, 2004.

[19] LIMA, Luciano Vieira; MACHADO, André Campos; PINTO, Marília Mazzaro. **Encore 4.5.4 : Editoração de Partituras.** 1a. ed. São Paulo: Editora Érica Ltda, 2003.

[20] LIMA, Luciano Vieira; LIMA; Sandra Fernandes de Oliveira; MACHADO,André Campos; PINTO, Marília Mazzaro.**Sound Forge 8.0: Gravação ao Vivo, Restauração de Sons de LPs e Masterização áudio Digital.** 1ª. ed. São Paulo: Editora Érica Ltda, 2005.

[21] RATTON, Miguel. **Criação de Música e Sons no Computador.** Rio de Janeiro: Editora Campus, 1995.

[22] GROUT, D.; PALISCA, C. **História da Música Ocidental.** Lisboa: Gradiva, 1994.

[23] LIMA, Florêncio de Almeida Lima. **Elementos Fundamentais da Música.** 4ª.ed. Rio de Janeiro,1958.

[24] BAS, **Trattato di Forma Musicale.** Milão: Ricordi, 1964.

[25] ZAMACOIS, Joaquín. **Curso de Formas Musicales.** Cuarta edición. Barcelona: Editorial Labor S.A.,1960.

[26] WINSTON & HORN. **LISP.** 3rd Edition. Addison-Wesley Publishing Company, 1980.

[27] LIMA, L. V. et al. **Editing Standard MIDI File in Functional Language CLEAN.** Diderot 99, Vienna, 1999.

[28] WELLESLEY,Alessandro Barros. **Programação Funcional com Clean 2.1 para Windows.** Uberlândia: Editora Pietro Ltda, 2004.

[29] BIRD, Richard. **Introduction to Functional Programming using Haskell, 2nd edition,** Prentice Hall press,1998.

[30] THOMPSON, Simon. **Haskell: The Craft of Functional Programming,** Second Edition, Addison-Wesley, 507 pages, paperback, 1999.

[31] LIMA, Sandra Fernandes de Oliveira; RUFINO, Hugo Leonardo Pereira; LOPES, Guilherme Francisco; GOULART, Reane Franco; LIMA, Luciano Vieira. **Matemática aplicada à detecção de tessitura vocal e à transposição automática de músicas em arquivos MIDI padrão.** In: CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL, São José do Rio preto, 2003, v.1, p.1-7.

[32] MARTINS, Gláucia Macedo Mendes. **Projeto e Implementação de uma Interface Visual Interativa para Montagens de Conteúdo de Autoria Multimídia.** Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2005.

[33] CAMARGO, Hélcio Júnior. **Desenvolvimento de aplicativos MIDI em linguagem funcional CLEAN**. Dissertação de Mestrado em Engenharia Elétrica, Uberlândia: UFU, 2007.

[34] RATTON,Miguel. **Midi Total.** Rio de Janeiro: Editora Música e Tecnologia, 2005.

[35] RATTON,Miguel. **Dicionário de Áudio e Tecnologia Musical.** Rio de Janeiro: Editora Música e Tecnologia, 2004.

[36] ABDOUNUR, Oscar J. **Matemática e Música : pensamento analógico na construção de significados**. São Paulo, SP: Escrituras Editora, 1999.

[37] MED, Bohumil. **Teoria da Música**. 4.ed., Brasília: Musimed, 1996.

Anexo I

Abaixo são apresentadas as opções dos parâmetros de entrada a partir da linha de comando do DOS para execução do renderizador TiMidity++ versão 2.11.3.

```
TiMidity++ version 2.11.3 (C) 1999-2002 Masanao Izumo <mo@goice.co.jp>
The original version (C) 1995 Tuukka Toivonen <tt@cgs.fi>
TiMidity is free software and comes with ABSOLUTELY NO WARRANTY.

Win32 version by Davide Moretti <dave@rimini.com>
             and Daisuke Aoki <dai@y7.net>

Usage:
  timidity-con [options] filename [...]


Options:
  -A n     Amplify volume by n percent (may cause clipping)
  -a       Enable the antialiasing filter
  -B n,m   Set number of buffer fragments(n), and buffer size(2^m)
  -C n     Set ratio of sampling and control frequencies
  -c file  Read extra configuration file
  -D n     Play drums on channel n
  -E mode  TiMidity sequencer extensional modes:
             mode = w/W : Enable/Disable Modulation wheel.
                    p/P : Enable/Disable Portamento.
                    v/V : Enable/Disable NRPN Vibrato.
                    s/S : Enable/Disable Channel pressure.
                    t/T : Enable/Disable Trace Text Meta Event at playing
                    o/O : Enable/Disable Overlapped voice
                    m<HH>: Define default Manufacture ID <HH> in two hex
                    b<n>: Use tone bank <n> as the default
                    B<n>: Always use tone bank <n>
                    F<args>: For effect.  See below for effect options.
    default: -E wpvSTo
  -e       Increase thread priority (evil) - be careful!
  -F       Disable/Enable fast panning (toggle on/off, default is on)
  -f       Enable fast decay mode (toggle)
  -h       Display this help message
  -I n     Use program n as the default
  -i mode  Select user interface (see below for list)
  -j       Realtime load instrument (toggle on/off)
  -k msec  Specify audio queue time limit to reduce voice
  -L dir   Append dir to search path
  -M name  Specify PCM filename (*.wav or *.aiff) to be played or:
           "auto": Play *.mid.wav or *.mid.aiff
           "none": Disable this feature (default)
  -m msec  Minimum time for a full volume sustained note to decay, 0 disables
  -O mode  Select output mode and format (see below for list)
  -o file  Output to another file (or device/server)  (Use "-" for stdout)
  -P file  Use patch file for all programs
  -p n(a)  Allow n-voice polyphony.  Optional auto polyphony reduction toggle.
  -p a     Toggle automatic polyphony reduction.  Enabled by default.
  -Q n     Ignore channel n
  -q m/n   Specify audio buffer in seconds
             m:Maxmum buffer, n:Filled to start   (default is 5.0/100%)
             (size of 100% equals device buffer size)
  -R n     Pseudo Reveb (set every instrument's release to n ms
             if n=0, n is set to 800(default)
  -s f     Set sampling frequency to f (Hz or kHz)
```

```
  -t code Output text language code:
              code=auto   : Auto conversion by `LANG' environment variable
                            (UNIX only)
                    ascii : Convert unreadable characters to '.'(0x2e)
                    nocnv : No conversion
                    1251  : Convert from windows-1251 to koi8-r
                    euc   : EUC-japan
                    jis   : JIS
                    sjis  : shift JIS
  -T n    Adjust tempo to n%; 120=play MOD files with an NTSC Amiga's timing
  -U      Unload instruments from memory between MIDI files
  -W mode Select WRD interface (see below for list)
  -w mode Windows extensional modes:
              mode=r/R : Enable/Disable rcpcv dll
  -x "configuration-string"
          Read configuration from command line argument
  -Z file Load frequency table

Available WRD interfaces (-W option):
  -Ww     Windows Console WRD tracer
  -Wt     TTY WRD tracer
  -Wd     dumb WRD tracer
  -W-     No WRD trace

Available output modes (-O option):
  -Od     Windows audio driver
  -Ow     RIFF WAVE file
  -Or     Raw waveform data
  -Ou     Sun audio file
  -Oa     AIFF file
  -Ov     Ogg Vorbis
  -Ol     List MIDI event
  -Om     MOD -> MIDI file conversion

Output format options (append to -O? option):
   `8'    8-bit sample width
   `1'    16-bit sample width
   `U'    U-Law encoding
   `A'    A-Law encoding
   `l'    linear encoding
   `M'    monophonic
   `S'    stereo
   `s'    signed output
   `u'    unsigned output
   `x'    byte-swapped output

Available interfaces (-i option):
  -in     ncurses interface
  -id     dumb interface

Interface options (append to -i? option):
   `v'    more verbose (cumulative)
   `q'    quieter (cumulative)
   `t'    trace playing
   `l'    loop playing (some interface ignore this option)
   `r'    randomize file list arguments before playing
   `s'    sorting file list arguments before playing

Effect options:
  -EFdelay=l : Left delay
  -EFdelay=r : Right delay
  -EFdelay=b : Rotate left & right
  -EFdelay=0 : Disabled delay effect
  -EFchorus=0 : Disable MIDI chorus effect control
  -EFchorus=1[,level] : Enable MIDI chorus effect control
                        `level' is optional to specify chorus level [0..127]
  -EFchorus=2[,level] : Surround sound, chorus detuned to a lesser degree.
                        `level' is optional to specify chorus level [0..127]
                        (default)
  -EFreverb=0 : Disable MIDI reverb effect control
  -EFreverb=1[,level] : Enable MIDI reverb effect control
                        `level' is optional to specify reverb level [0..127]
                        This effect is only available in stereo
                        (default)
  -EFreverb=2 : Global reverb effect
  -EFns=n : Enable the n th degree noiseshaping filter. n:[0..4]
            This effect is only available for 8-bit linear encoding
```

Anexo II

Especificação MIDI 1.0

Descritivo em Português elaborado por Miguel Ratton

INTRODUÇÃO

MIDI (Musical Instrument Digital Interface) é um padrão de transmissão serial de dados, que permite a troca de informações entre instrumentos e equipamentos de aplicação musical.

Os equipamentos que implementam funções MIDI, podem não conter todas as funções previstas pelo padrão, mas aquelas implementadas devem seguir estritamente o que está padronizado.

CONVENÇÕES

A representação de números neste texto segue a seguinte convenção:

- Números representados na base hexadecimal são sempre seguidos de um H, como 34H.
- Números representados na base binária são sempre representados precedidos de um $, como $10001011.
- Os demais números estão na base decimal.

Os termos técnicos estão traduzidos, sempre que possível, mas os termos originais, em inglês, foram preservados e são citados, de forma a facilitar a leitura de textos estrangeiros, muito comuns nesta área.

O termo equipamento neste texto significa qualquer dispositivo eletrônico de aplicações musicais: instrumentos musicais, módulos de efeitos sonoros, baterias eletrônicas, seqüenciadores, equipamentos de áudio, etc.

CARACTERÍSTICAS DE HARDWARE

Os equipamentos que implementam o interfaceamento MIDI devem possuir um transmissor ou um receptor, ou ambos, que execute a transmissão/recepção de mensagens no padrão MIDI. Tanto transmissão quanto recepção devem operar à uma taxa de 31.250 bauds (+/- 1%), em modo assíncrono, com um bit de início (start bit), 8 bits de dados(D0 a D7) e um bit de fim (stop bit), perfazendo um total de 10 bits por byte serial, e ocupando um período de 320 micro-segundos.

O hardware do transmissor é composto de um UART (Universal Asynchronous Receiver/Transmitter) para transmissão dos dados seriais, e um acionador (buffer) de saída, capaz de drenar uma corrente de até, 5 mA. A saída deve possuir resistores de proteção na eventualidade de curto-circuito entre os terminais. O conector de saída deve ser aterrado ao equipamento.

O hardware do receptor é composto de um acoplador óptico e um UART para recepção dos dados seriais. O acoplador óptico deve ser capaz de chavear com uma corrente menor ou igual a 5 mA no LED, e os tempos de subida e de descida não devem ultrapassar 2 micro-segundos, sendo recomendados os acopladores ópticos PC-900 (Sharp) e 6N138 (HP). O conector de entrada não deve ser aterrado ao equipamento.

Os conectores do equipamento devem ser do tipo DIN de 5 pinos (fêmea)dispostos em 180 graus, e montados em painel. Os conectores devem ser identificados como MIDI IN (entrada), MIDI OUT (saída) e, opcionalmente, MIDI THRU (repetição da entrada). São recomendados conectores Switchcraft 57 GB5F.

A saída opcional MIDI THRU deve prover uma cópia direta dos dados que entram pelo conector MIDI IN, de forma a permitir a ligação de mais de dois equipamentos em cadeia. No caso de cadeias muito longas (mais de três equipamentos), o padrão

recomenda a utilização de acopladores ópticos mais rápidos, de forma a evitar que o acúmulo de atrasos de subida e descida no acoplador afetem a largura dos pulsos de sinal. O conector MIDI THRU deve ser aterrado ao equipamento.

O cabo de interligação de equipamentos deve possuir dois condutores e mais a blindagem, e não deve ter um comprimento o maior do que 15 metros(50 p,s). Os plugs devem ser do tipo DIN (macho) de 5 pinos em 180 graus, com a blindagem conectada ao pino 2 em ambas as extremidades. É recomendado o plug Switchcraft 05 GM5M. O diagrama abaixo apresenta uma sugestão para os circuitos de saída e de entrada, conforme estabelece o padrão.

FORMATAÇÃO DOS DADOS

A comunicação pelo padrão MIDI utiliza, basicamente, mensagens do tipo multi-bytes contendo um Byte de Status (Status Byte) seguido de um ou dois Bytes de Dados (Data Bytes). As exceções à essa regra são as mensagens de Tempo Real e as de Sistema Exclusivo. A seguir, é apresentada a classificação das mensagens MIDI.
As mensagens MIDI são classificadas em duas categorias principais, que são as Mensagens de Canal (Channel Messages) e as Mensagens de Sistema(System Messages).

- **MENSAGENS DE CANAL** (CHANNEL MESSAGES).

  As Mensagens de Canal são identificadas pelos quatro bits mais significativos do Byte de Status. Os quatro bits menos significativos determinam o número do canal de MIDI (1 a 16) em que a mensagem está sendo transmitida. Isso significa que a mensagem é endereçada ao(s)equipamento(s) sintonizado(s) naquele canal.

  As Mensagens de Canal são subdivididas em dois tipos: Mensagens de Voz (Voice Messages), que controlam as vozes (geradores de som) do equipamento, e são transmitidas pelo Canal de Voz (Voice Channel); e as Mensagens de Modo (Mode Messages), que definem como o equipamento deve responder à recepção das Mensagens de Voz. As Mensagens de Modo são transmitidas pelo Canal Básico (Basic Channel).

Tabela 1 - Mensagens de Voz (Channel Voice Messages)		
BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
$1000nnnn	$0kkkkkkk $0vvvvvvv	Nota Desativada (Note Off) $0vvvvvvv = veloc. que a tecla é solta
$1001nnnn	$0kkkkkkk $0vvvvvvv	Nota Ativada (Note On) $0vvvvvvv = veloc. que a tecla é abaixada $0vvvvvvv = 0 --> Nota Desativada
$1010nnnn	$0kkkkkkk $0vvvvvvv	Pressão na Tecla (Polyph. Aftertouch) $0vvvvvvv = valor da pressão
$1011nnnn	$0ccccccc $0vvvvvvv	Controle (Control Change) $0ccccccc = número do controle (0 a 121) $0vvvvvvv = valor do controle $0ccccccc = 122 a 127: vide Tabela 3
$1100nnnn	$0ppppppp	Mudança de Programa (Program Change) $0ppppppp = número do programa (0 a 127)
$1101nnnn	$0vvvvvvv	Pressão no Teclado (Channel Aftertouch) $0vvvvvvv = valor da pressão
$1110nnnn	$0ggggggg $0hhhhhhh	Variação do Pitch Bend (Pitch Bend Change) Primeiro byte de dados é a parte menos significativa (LSB), e o segundo byte de dados é a parte mais significativa (MSB) (Vide Nota 10)

NOTAS:

1) $nnnn = N-1, onde N é o número do canal, ou seja, $0000 é Canal 1, $0001 é Canal 2, ..., $1111 é Canal 16.

2) $kkkkkkk = número da tecla (nota), que vai de 0 a 127. A nota de número 60 corresponde ao Dó central do teclado.

3) $vvvvvvv = velocidade com que a tecla foi pressionada. A faixa de valores possíveis de velocidade deve obedecer à uma escala logarítmica, do tipo abaixo:

0 .....1.....64.....127

---

off.....ppp.....pp.....p.....mp.....mf.....f.....ff.....ffff

$vvvvvvv=64, caso o teclado não tenha sensibilidade à velocidade. Nota Ativada com vel=0 significa Nota Desativada com vel=64.

4) Toda mensagem de Nota Ativada (Note On) deve ser seguida, em algum momento, de uma mensagem de Nota Desativada para a mesma nota, no mesmo canal.

5) $ccccccc = Número do Controle, conforme a Tabela 3.
6) Enquanto as mensagens enviadas tiverem o mesmo Byte de Status, este pode ser omitido, ficando como Status Corrente (Current Status), até que um novo Byte de Status seja requerido.

7) A sensibilidade à variação do Pitch Bend é determinada no equipamento receptor. O padrão define que a um Pitch Bender indo para a posição central deve enviar uma mensagem com valor 2000H.

Tabela 2 - Mensagens de Modo (Channel Mode Messages)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
$1011nnnn	$01111010	Controle Local (Local Control) $0vvvvvvv=0: desliga Controle

	$0vvvvvv	Local $0vvvvvvv=127: liga Controle Local
$1011nnnn	$01111011 $00000000	Solta Todas as Teclas (All Notes Off)
$1011nnnn	$01111100 $00000000	Desativa Modo Omni (Omni Off) (e solta todas as teclas)
$1011nnnn	$01111101 $00000000	Ativa Modo Omni (Omni On) (e solta todas as teclas)
$1011nnnn	$01111110 $0vvvvvvv	Ativa Modo Monofonico (Mono Mode) (e solta todas as teclas) $0vvvvvv é o número de canais $0vvvvvv=0:o número de canais é igual ao número de vozes do receptor.
$1011nnnn	$01111111 $00000000	Ativa Modo Polifonico (Poly Mode) (e solta todas as teclas)

NOTAS:

1) $nnnn = N-1, onde N é o número do canal, ou seja, $0000 é Canal1, $0001 é Canal 2, ..., $1111 é Canal 16..

2) Com exceção da mensagem de Controle Local, as demais mensagens de Modo funcionam como mensagens de Soltar Teclas, que fazem com que todas as notas que porventura estejam tocando no equipamento receptor sejam comandadas através do Canal Básico para cessar, como se tivessem sido recebidas as respectivas mensagens de Nota Desativada. Entretanto, o padrão não recomenda o seu uso em lugar de mandassem específica de Nota Desativada. No caso do receptor não ter qualquer nota tocando, estas mensagens de Soltar Todas as Teclas devem ser ignoradas.

3) A mensagem de Controle Local é usada opcionalmente para desvincular internamente o teclado dos circuitos geradores de som. Ao receber uma

mensagem de Desativar Controle Local, o equipamento passa a ser comandado apenas via MIDI, e não mais pelo seu teclado. A mensagem de Ativar Controle Local restaura a situação original..

4) O terceiro byte da mensagem de Modo Monofonico especifica o número de canais de MIDI nos quais as Mensagens de Voz serão enviadas.

Tabela 3 - Controles (control change)
atualizado até jan/93

NÚMERO	DESCRIÇÃO DO CONTROLE
0	Seleção de Banco de Programas - MSB (Bank Select)
1	Roda/Alavanca de Modulação - MSB (Modulation Wheel)
2	Controle por Sopro - MSB (Breath Controller)
3	Não definido
4	Pedal MSB (Foot Controller)
5	Tempo do Portamento - MSB (Portamento Time)
6	Entrada de Dados - MSB (Data Entry)
7	Volume Principal - MSB (Main Volume)
8	Equilíbrio - MSB (Balance)
9	Não definido
10	Pan - MSB (Pan)
11	Controle de Expressão - MSB (Expression Controller)
12	Effect Control 1
13	Effect Control 2
14 - 15	Não definidos
16 - 19	Controles de uso Geral 1 a 4 - MSB (General Purpose)
20 - 31	Não definidos
32 - 63	LSB dos Controles 1 a 31
64	Pedal de Sustain (Sustain/Damper Pedal)
65	Liga/Desliga Portamento (Portamento On/Off)
66	Pedal de Sostenuto (Sostenuto Pedal)
67	Pedal Abafador (Soft Pedal)
68	Pedal de Legato
69	Pedal de Sustain 2 (Hold 2)
70	Controle de som 1 (Sound Variation)
71	Controle de som 2 (Timbre/Harmonic Content)
72	Controle de som 3 (Release Time)
73	Controle de som 4 (Attack Time)
74	Controle de som 5 (Brightness)
75 - 79	Controles de som 6 - 10 (ainda não determinados)
80 - 83	Controles de Uso Geral 5 a 8 (General Purpose)
84	Controle de portamento
85 - 90	Não definidos
91	Intensidade do Reverb (Ext. Effect Depth)

92	Profundidade do Tremolo (Tremolo Depth)
93	Profundidade do Chorus (Chorus Depth)
94	Profundidade do Batimento (Celeste Detune Depth)
95	Profundidade do Phaser (Phaser Depth)
96	Incremento (Data Increment)
97	Decremento (Data Decrement)
98	Número de Parâmetro - LSB (Parameter Number)
99	Número de Parâmetro - MSB (Parameter Number)
100	Número de Parâmetro - LSB (Parameter Number)
101	Número de Parâmetro - MSB (Parameter Number)
102 - 119	Não definidos
120 - 127	Reservados para as Mensagens de Modo (vide Tabela 2)

NOTAS:

1) Os controles 0 a 63 são do tipo controle contínuo, e podem assumir valores de até 14 bits. Os controles 64 a 95 são chaves do tipo liga/desliga.

2) Os fabricantes podem alocar novos controles aos números não definidos, desde que forneçam as informações pertinentes, no manual de operação do equipamento.

3) Os controles contínuos são divididos em Bytes Mais significativos(MSB) e Bytes Menos Significativos (LSB). Caso o valor do controle não ultrapasse sete bits, apenas o MSB é transmitido. Caso haja necessidade de representar o valor com maior resolução do que sete bits, são enviados os Bytes MSB e LSB, nessa ordem. Se o valor do controle foi alterado apenas na parte LSB, pode ser transmitido apenas o Byte LSB, sem o envio do MSB.

4) Nos controles do tipo chave liga/desliga, o valor $vvvvvvv = 0 significa desligado (off), enquanto $vvvvvvv = 127 significa ligado (on). Valores de 1 a 126, inclusive, devem ser ignorados.

- **MENSAGENS DE SISTEMA** (SYSTEM MESSAGES)

  As Mensagens de Sistema não utilizam canal de MIDI, e são subdivididas em três tipos:

Mensagens Comuns (System Common Messages), que são comuns a todos os equipamentos de uma cadeia;

Tabela 4 - Mensagens Comuns de Sistema (System Common Messages)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
$11110001	$0nnnddd	Reservada para MIDI Time Code (MTC)
$11110010	$0ggggggg $0hhhhhhh	Ponteiro de Seq. (Song Position Pointer) $0ggggggg=parte menos significativa (LSB) $0hhhhhh=parte mais significativa (MSB)
$11110011	$0sssssss	Seletor de Seqüência (Song Select) $0sssssss=número da seqüência
$11110100	-	Não definido
$11110101	-	Não definido
$11110110	-	Requisita Afinação (Tune Request)
$11110111	-	Fim de Mensagem Exclusiva

NOTAS:

1) O Ponteiro de Seqüência (Song Position Pointer) é um registro interno que mantém o número de tempos (beats) MIDI desde o início da música (1 tempo MIDI é igual a 6 clocks MIDI). Normalmente ele, ajustado para zero, no momento em que o botão de INICIAR (START), pressionado, dando início à execução da seqüência. O ponteiro então é incrementado a cada seis clocks MIDI, até que seja pressionada a tecla PARAR (STOP). Se a tecla CONTINUAR é pressionada, o ponteiro continua a ser incrementado de onde havia parado. O termo seqüência refere-se a qualquer grupo de notas armazenados com seus devidos tempos de ocorrência e de duração. Inclusive seqüências de bateria eletrônica.

2) Seleção de Seqüência (Song Select) especifica qual a seqüência (música) a ser executada quando da ocorrência de uma mensagem de INICIAR.

3) EOX é usado como marca de fim de transmissão de Mensagem Exclusiva.

Mensagens de Tempo Real (System Real-Time Messages), que também são comuns a todos os equipamentos de uma cadeia. Essas mensagens só possuem Byte de Status (nenhum Byte de Dados), e podem ser enviadas a qualquer momento, mesmo entre bytes de uma mensagem com outro Byte de Status. Nesse caso, a Mensagem de Tempo Real deve ser tratada (ou ignorada, se o equipamento não implementar funções de Tempo Real), sem prejuízo da mensagem dentro da qual ela foi inserida;

Tabela 5 - Mensagens de Sistema em Tempo Real (System Real Time)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
$11111000	-	Clock de MIDI (Timing Clock)
$11111001	-	Não definido
$11111010	-	Iniciar (Start)
$11111011	-	Continuar (Continue)
$11111100	-	Parar (Stop)
$11111101	-	Não definido
$11111110	-	Sensor de Atividade (Active Sensing)
$11111111	-	Reset (Reset)

NOTAS:

1) As Mensagens em Tempo Real tem por finalidade sincronizar todos os equipamentos de um sistema operando em tempo real.

2) O padrão permite que elas possam ser enviadas em qualquer momento, mesmo inseridas entre bytes de outras mensagens.

3) O Clock de MIDI o sistema sincronizar-se. Os clocks de MIDI enviados sempre à razão de 24 clocks por semínima.

4) A mensagem de Início (Start) é enviada imediatamente após a tecla de Início ter sido pressionada no equipamento (sequenciador, bateria eletrônica, etc).

5) A mensagem de Sensor Ativo (Active Sensing) é opcional, tanto para transmissores quanto para receptores. O padrão recomenda que essa mensagem seja enviada a cada 300 milisegundos, no máximo, caso não tenha havido o envio de qualquer outra mensagem nesse intervalo de tempo. O receptor ignorar tal mensagem até que receba a primeira, quando então, passar a esperá-la quando não houver recepção de qualquer mensagem MIDI dentro de um intervalo de 300 ms. Passados os 300 ms sem qualquer mensagem nem mensagem de Sensor Ativo, o receptor apagar todas as vozes (all notes off) e retornar à operação normal. Os equipamentos que não implementam Sensor Ativo devem ignorar estas mensagens.

6) A mensagem de Reset serve para Inicializar o sistema, e não deve ser enviada automaticamente, ao ser ligado o instrumento.

Mensagens Exclusivas (System Exclusive Messages), que podem conter qualquer quantidade de Bytes de Dados, além do Byte de Status. Como o número de Bytes de Dados é variável, o fim de uma Mensagem Exclusiva é detectado ao chegar um byte de Fim de Mensagem Exclusiva (EOX, End of Exclusive) ou um novo Byte de Status.

As Mensagens Exclusivas devem incluir em seus Bytes de Dados um Código de Identificação do Fabricante (ID code). Caso o equipamento receptor não reconheça o Código de Identificação, dever ignorar os Bytes de Dados até o fim da mensagem.

O padrão recomenda que os fabricantes divulguem a documentação das suas Mensagens Exclusivas, de forma a permitir que os usuários possam acessar plenamente os equipamentos via MIDI.

Tabela 6 - Mensagens Exclusivas de Sistemas (System Exclusive Messages)

BYTE DE STATUS	BYTE DE DADOS	DESCRIÇÃO
$11110000	$0zzzzzzz : :	Transferência de Dados (Bulk Dump) $0zzzzzzz=identificação Pode ser inserido aqui qualquer número de bytes, para qualquer propósito, desde que tenham sempre o bit mais significativo igual a zero.
$11110111	-	Fim de Mensagem Exclusiva (EOX)

NOTAS:

1) Nenhum Byte de Status ou de Dados deve ser inserido no meio de uma Mensagem Exclusiva, exceto Mensagens de Tempo Real.

2) A Mensagem Exclusiva termina com um EOX ou qualquer outro Byte de Status.

TIPOS DE BYTES

Existem dois tipos de bytes nas mensagens MIDI: os Bytes de Status (Status Bytes) e os Bytes de Dados (Data Bytes).

- BYTE DE STATUS (STATUS BYTE)

O Byte de Status é um número binário de 8 bits, sendo que o bit mais significativo (bit 7, mais à esquerda) tem sempre valor 1. O Byte de Status serve para identificar que categoria e tipo de mensagem está sendo transmitida. Como já foi mencionado anteriormente, excetuando-se os Bytes de Status de Mensagens de Sistema em

Tempo Real, que podem ser inseridos no meio de outras mensagens, um novo Byte de Status sempre indica uma nova mensagem, mesmo que a mensagem anterior não tenha sido completada.

Tabela 7 - Bytes de Status

BYTE DE STATUS	QUANTIDADE DE BYTES DE DADOS	DESCRIÇÃO
		MENSAGENS DE VOZ (Channel Voice)
$1000nnnn	2	Nota Desativada (Note Off)
$1001nnnn	2	Nota Ativada (Note On)
$1010nnnn	2	Pressão na Tecla (Polyph. Aftertouch)
$1011nnnn	2	Controle (Control Change)
$1100nnnn	1	Mudança de Programa (Program Change)
$1101nnnn	1	Pressão no Teclado (Channel Aftertouch)
$1110nnnn	2	Variação do Pitchbend (Pitchbend Change)
$1011nnnn	2	MENSAGENS DE MODO (Channel Mode) Seleciona o Modo de Operação
		MENSAGENS DE SISTEMA (System Messages)
$11110000	*****	Exclusiva (System Exclusive)
$11110sss	0 a 2	Comum (System Common)
$11111ttt	0	Tempo-Real (System Real Time)

- NOTAS:
    1. $nnnn = N-1, onde N é o número do canal, ou seja, $0000 é Canal1, $0001 é Canal 2, ..., $1111 é Canal 16.
    2. ***** = $0iiiiiii, <dados>, EOX.
    3. $0iiiiiii = Identificação do Fabricante/Equipamento.
    4. $sss = $000 a $111 (1 a 7).

5. $ttt: idem.

- STATUS CORRENTE (RUNNING STATUS)

  No caso específico das Mensagens de Voz e de Modo, quando um Byte de Status é recebido e processado, o equipamento receptor deve armazenar aquele Status até que um novo Byte de Status seja recebido. O padrão determina que, se um transmissor tiver que transmitir uma nova mensagem cujo Byte de Status é idêntico ao último transmitido, ele pode (opcionalmente) omiti-lo, transmitindo apenas os Bytes de Dados seguintes.

  O Byte de Status omitido, nesse caso, é considerado como Status Corrente (Running Status). O receptor deve ser capaz de interpretar a transmissão com ou sem Status Corrente. A vantagem desse artifício é que no caso de grande quantidade de mensagens de mesmo Status (como por exemplo, uma série de teclas pressionadas), onde as mensagens ficam menores. O Status Corrente é interrompido sempre que chegar outro Byte de Status, exceto se for um Byte de Status de uma Mensagem de Sistema em Tempo Real, como já foi explicado anteriormente.

- STATUS NÃO IMPLEMENTADOS

  O equipamento receptor deve ignorar os Bytes de Status (e respectivos Bytes de Dados, se houver) que identifiquem funções não implementadas por ele.

- STATUS INDEFINIDOS

O padrão recomenda que os códigos de Bytes de Status ainda não definidos não sejam transmitidos. Além disso, recomenda que se tome todo o cuidado necessário para evitar a transmissão de mensagens espúrias ao ligar ou desligar o equipamento

transmissor. Qualquer Byte de Status não definido deve ser ignorado pelo equipamento receptor, juntamente com os Bytes de Dados subseqüentes.

- BYTES DE DADOS

Exceto nas Mensagens de Sistema em Tempo Real, os Bytes de Status são sempre seguidos por um ou dois Bytes de Dados. Esses bytes são números binários de 8 bits, nos quais o bit mais significativo (bit 7, mais à esquerda) tem sempre valor zero. Os valores que os Bytes de Dados podem assumir, dependendo do Byte de Status a que pertencem, estão definidos nas Tabelas, mais adiante. O transmissor deve sempre enviar o número correto de Bytes de Dados definido pelo padrão para cada Byte de Status. O receptor, por sua vez, só pode processar a mensagem ao recebê-la completamente, isto é, deve esperar a recepção de todos os Bytes de Dados esperados para o Status Corrente. O receptor deve ignorar quaisquer Bytes de Dados recebidos, que não tenham sido precedidos por um Byte de Status válido, exceto no caso do Status Corrente, citado anteriormente.

MODOS DE OPERAÇÃO

Os instrumentos musicais eletrônicos (sintetizadores, samplers, órgãos eletrônicos, etc) possuem dispositivos internos (hardware ou software) responsáveis pela geração das notas (sons). Esses dispositivos, chamados de vozes (voices), são em número limitado (em geral 8, 16 ou 32), e determinam a polifonia do instrumento.

Quando uma tecla é pressionada no teclado do instrumento, o processador interno do mesmo verifica se há alguma voz disponível, isto é, se o instrumento já está tocando o número máximo de notas possível. Se não está ele aloca uma voz disponível à nota correspondente à Nota Ativada. À medida que as teclas são soltas, as vozes vão ficando livres (disponíveis) para serem ocupadas por novas notas. Caso todas as vozes estejam ocupadas, o processador decide, através de um algoritmo predeterminado, como tocar a nota nova. Em geral a voz que está tocando a nota mais antiga passa a ser ocupada pela nova nota, mas pode-se encontrar instrumentos que adotem procedimento diferente.

Dentro do padrão definido na MIDI Specification 1.0, há quatro modos possíveis de se operar as vozes de um instrumento, conforme mostram as a seguir.

Tabela 8 - Modos de Operação (recepção)
modos de recepção para um instrumento ajustado para receber pelo Canal Básico N.

MODO	OPERAÇÃO
Omni On / Poly	O instrumento processa todas as Mensagens de Voz, independente dos Canais de Voz em que elas estão sendo transmitidas, e responde a elas polifonicamente.
Omni On / Mono	O instrumento processa todas as Mensagens de Voz, independente dos Canais de Voz em que elas estão sendo transmitidas, e responde a elas monofonicamente.
Omni Off / Poly	O instrumento só processa as Mensagens de Voz que estão sendo enviadas no Canal de Voz "N", e responde a elas polifonicamente.
Omni Off / Mono	O instrumento só processa as Mensagens de Voz que estão sendo enviadas nos Canais de Voz "N" a "N+M-1", e responde a elas monofonicamente, alocando-as à s vozes 1 a "M", respectivamente. O número de vozes "M" é definido no terceiro byte da Mensagem de Modo.

Tabela 9 - Modos de Operação (transmissão)
modos de transmissão para um instrumento sintonizado no Canal Básico N.

MODO	OPERAÇÃO
Omni On / Poly	O instrumento transmite todas as Mensagens de Voz através do Canal de Voz "N"
Omni On / Mono	O instrumento transmite todas as Mensagens de Voz, para uma voz, através do Canal de Voz "N"
Omni Off / Poly	O instrumento transmite todas as Mensagens de Voz, para todas as vozes, através do Canal de Voz "N".
Omni Off / Mono	O instrumento transmite as Mensagens de Voz das vozes 1 a "M" através dos Canais de Voz "N" a "N+M-1", respectivamente (uma voz por canal).

Um instrumento só pode operar em um único Modo de cada vez, e é recomendado que tanto transmissor quanto receptor(es) estejam operando no mesmo Modo. Caso um instrumento não possa operar no Modo requisitado, ele deve ignorar o comando de mudança de Modo de Operação, ou então utilizar um outro Modo alternativo (normalmente Modo 1).

As Mensagens de Modo só devem ser reconhecidas pelo receptor quando enviadas através do Canal Básico no qual ele está sintonizado, independentemente do seu Modo de Operação corrente. As Mensagens de Voz devem ser recebidas através dos Canais de Voz, conforme estabelecem as Tabelas 8 e 9, dependendo do Modo selecionado. O padrão permite, ainda, que um instrumento seja sintonizado em um ou mais Canais Básicos, por default ou por ação do usuário. Nesse caso, o instrumento se comporta como se fossem vários instrumentos independentes.

CONDIÇÕES INICIAIS

O padrão determina que o instrumento, ao ser ligado, adote as condições iniciais (default) de: Modo 1 de operação, recepção desabilitada de todas as Mensagens de Voz, exceto de Nota Ativada/Nota Desativada (Note On/Note Off) e supressão de quaisquer transmissões espúrias indefinidas.

OBSERVAÇÃO IMPORTANTE:

À época em que foi escrita MIDI Specification 1.0, o Modo 4 era esperado para transmissão/recepção de informações monofônicas através dos canais diferentes de MIDI, em um mesmo instrumento.

Posteriormente, os instrumentos multitimbrais passaram a operar de uma forma não definida em qualquer dos quatro modos, pois podem receber, simultaneamente, mensagens em vários canais diferentes com informações para execução polifônica em cada um desses canais, que são alocadas a uma ou mais vozes por canal. Esse quinto modo tem sido referenciado por muitos fabricantes como Modo Multi.

Tabela 10 - Códigos dos Fabricantes
atualizada até nov/87

Código	Fabricante	Código	Fabricante
00-00-14	Perfect Fretwork	21H	SIEL
01H	Sequential Circuits	22H	Synthaxe
02H	IDP	23H	Stepp _
03H	Voyetra/OctavePlateau	24H	Hohner
04H	Moog	25H	Twister
05H	Passport Designs	26H	Solton
06H	Lexicon	27H	Jellinghaus MS
07H	Kurzwei	28H	Southworth Music
08H	Fender	29H	PPG
0AH	AKG Acoustics	2AH	JEN
0BH	Voyce Music	2BH	SSL
0CH	Waveframe Corp.	2CH	Audio Veritrieb
0DH	ADA	2FH	Elka
0EH	Garfield Electronics	30H	Dynacord
0FH	Ensoniq	40H	Kawai
10H	Oberheim	41H	Roland
11H	Apple Computer	42H	Korg
12H	Grey Matter Response	43H	Yamaha
14H	Palm Tree Instr.	44H	Casio
15H	JL Cooper	46H	Kamyia Studio
16H	Lowrey	47H	Akai
17H	Adams-Smith	48H	Japan Victor
18H	E-Mu Systems	49H	Mesosha
19H	Harmony Systems	7DH	Uso Não Comercial
1AH	ART	7EH	Não Tempo-Real
1BH	Baldwin	7FH	Tempo-Real
1CH	Eventide		
1DH	Inventronics		
1FH	Clarity		

Anexo III

Figuras Musicais.

Usamos 7 figuras que representam 7 valores.

Figura e Pausa	Código	Nome em Português	Nome em Inglês	Valor Proporcional
	1	Semibreve	Hole Note	1
	2	Mínima	Half Note	½
	4	Semínima	Quarter Note	¼
	8	Colcheia	Eigth Note	$^1/_8$
	16	Semicolcheia	16th Note	$^1/_{16}$
	32	Fusa	32th Note	$^1/_{32}$
	64	Semifusa	64th Note	$^1/_{64}$

A semibreve é a figura de maior valor, da qual todas as outras são subdivisões. Então:

- a **semibreve** vale **1** (inteiro)
- a **mínima** vale **1/2** (metade) da semibreve
- a **semínima** vale metade da mínima ou **1/4** da semibreve
- a **colcheia** vale metade da semínima ou **1/8** da semibreve
- a **semicolcheia** vale metade da colcheia ou **1/16** da semibreve
- a **fusa** vale metade da semicolcheia ou **1/32** da semibreve
- a **semifusa** vale metade da fusa ou **1/64** da semibreve

Cada nota musical pode soar um certo intervalo de tempo, dependendo da caixa de ressonância do instrumento. Em uma notação musical, os tempos de cada nota são múltiplos uns dos outros. Assim, pode-se apresentar esta relação conforme a figura abaixo: