TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UMA INTERFACE USB PARA PROTÓTIPO DE UMA PRÓTESE AUDITIVA INTELIGENTE - PAI

Jorge Augusto Vieira Lima

Rafael Oliveira de Castro Alves

Brasília, junho de 2007

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UMA INTERFACE USB PARA PROTÓTIPO DE UMA PRÓTESE AUDITIVA INTELIGENTE - PAI

Jorge Augusto Vieira Lima

Rafael Oliveira de Castro Alves

Relatório submetido ao Departamento de Engenharia Elétrica como requisito parcial para obtenção do grau de Engenheiro Eletricista

Banca Examinadora

Prof. Ricardo Zelenovsky, ENE/UnB _____
Orientador

Prof. Felipe da Silva Pereira, ENE/UnB _____
Examinador interno

Eng. Giuler Alberto _____
Examinador externo

Dedicatórias

Dedico este trabalho a todas as pessoas que fizeram e ainda fazem parte da minha vida e foram importantes em minha formação, tanto profissional quanto de caráter.

Rafael Oliveira de Castro Alves

Dedico aos meus pais pelo apoio incondicional e pelo grande investimento em meu estudos..

Jorge Augusto Vieira Lima

Agradecimentos

Agradeço a minha família, que sempre esteve presente na minha vida e nunca deixou me faltar nada, aos meus amigos, que ao longo dos últimos cinco anos influenciaram a pessoa que sou hoje, e aos professores, que foram uma fonte de conhecimento da qual tive pleno acesso. Pois só com a ajuda desses consegui concluir esta árdua tarefa.

Jorge Augusto Vieira Lima

Agradeço às pessoas que mais me apoiaram, do começo ao final do curso: aos professores, que além de se mostrarem profissionais exemplares, tornaram-se amigos e conselheiros no momento em que precisei; ao meu pai, em quem me espelho para ser sempre humilde e trabalhador; à minha mãe, por ser tão confiante no meu trabalho e me animar nos momentos em que senti mais dificuldade; meus irmãos, por sempre tomarem conta do caçula; aos colegas de curso, por realizarem a caminhada junto comigo e, por fim, aos meus amigos, que estiveram sempre presentes, proporcionando momentos de descontração e diversão que me motivaram e deram energia para que eu conseguisse completar com sucesso esta etapa.

Rafael Oliveira de Castro Alves

RESUMO

O presente texto apresenta o desenvolvimento de uma interface USB utilizando um microprocessador ARM. A finalidade de tal interface é o envio de oito canais de áudio obtidos através de um arranjo de microfones do protótipo de uma prótese auditiva inteligente, para teste de algoritmos de identificação de direção de chegada e filtragem espacial de som. Primeiro são apresentados resumos da especificação USB e das funcionalidades da arquitetura do processador utilizado. Em seguida são mostrados e discutidos os problemas encontrados, as soluções utilizadas e os resultados obtidos.

ABSTRACT

The present document contains the development of a USB interface using an ARM microprocessor. The purpose of such interface is to send eight audio channels acquired from a microphone array of a smart auditory prosthesis, so direction of approach algorithms and space sound filtering could be tested. First it presents summaries of the USB specification and the used ARM processor architecture features. Following, the found problems as well as the used solutions and results obtained are shown and discussed.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Siglas

AC	AudioControl
ADC	Analog to Digital Converter
AIC	Audio Interface Class
ARM	Advanced RISC Machine
AS	AudioStreaming
CI	Circuito Integrado
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
FIFO	First In,FirstOut
FINATEC	Fundação de Empreendimentos Científicos e Tecnológicos
FFT	Fast Fourier Transform
FU	Feature Unit
IRQ	Interrupt Request
IT	Input Terminal
LSB	Least Significant Byte
MCK	Main CLOCK
MIPS	Millions of Instructions Per Second
MPEG	Moving Picture Experts Group
MSB	Most Significant Byte
MU	Mixing Unit
NRZI	Non Return to Zero Inverted
OT	Output Terminal
PC	Personal Computer
PCM	Pulse Code Modulation
PLL	Phase-Locked Loop
PU	Processing Unit
RAM	Random Access Memory
RISC	Reduced Instruction Set Complex
SU	Selector Unit
TC	Timer/Counter
U(S)ART	Universal (Synchronous)/Asyncrhonous Receiver-Transmitter
UDP	USB Device Port
USB	Universal Serial Bus
USB-IF	USB Implementers Forum
XU	Extension Unit

Capítulo 1

Introdução

Este capítulo apresenta a principal motivação do presente trabalho

1.1 O projeto PAI

Próteses auditivas são pequenos dispositivos auriculares que têm como finalidade amplificar o som para melhorar a percepção auditiva do deficiente. Entretanto, as próteses possuem uma falha: ao amplificar indiscriminadamente todos os sons, ela gera para o usuário uma dificuldade de separar o que é interessante do que é apenas ruído. O cérebro humano, ao receber a informação captada pelos dois ouvidos, automaticamente indica a direção de cada som, permitindo à pessoa se concentrar no que quer ouvir. Inspirado nisso, surgiu a idéia de se produzir uma prótese auditiva inteligente que a partir de um arranjo de microfones e utilizando técnicas de processamento de sinal, faria uma filtragem espacial, enfatizando os sons vindos de uma direção desejada.

1.1.1 O protótipo

Para o desenvolvimento da PAI, se propôs a construção de um protótipo que faria a captação do som através do arranjo de 8 microfones, a pré-amplificação e filtragem desses sinais e seu envio para um PC, onde seriam feitos testes de algoritmos para detecção da direção de chegada do som e filtragem espacial. O diagrama de blocos representando o protótipo está na figura 1.1.

A construção desse protótipo foi iniciada por duas equipes distintas, uma encarregada do hardware de pré-amplificação e filtragem, e a outra responsável pela programação do ARM que realizaria a interface com o PC. Após seis meses de trabalho, todo o hardware havia sido construído e uma versão do programa estava pronta. Entretanto, o protótipo ainda não estava funcional.

1.2 Definição do problema

Após toda a construção do hardware, os testes mostraram que o sinal entregue ao ARM não estava satisfatório, seja do ponto de vista do nível no qual as tensões se encontravam, seja do ponto de vista das freqüências. Assim sendo, fazia-se necessária a revisão do projeto para detecção e solução das deficiências.

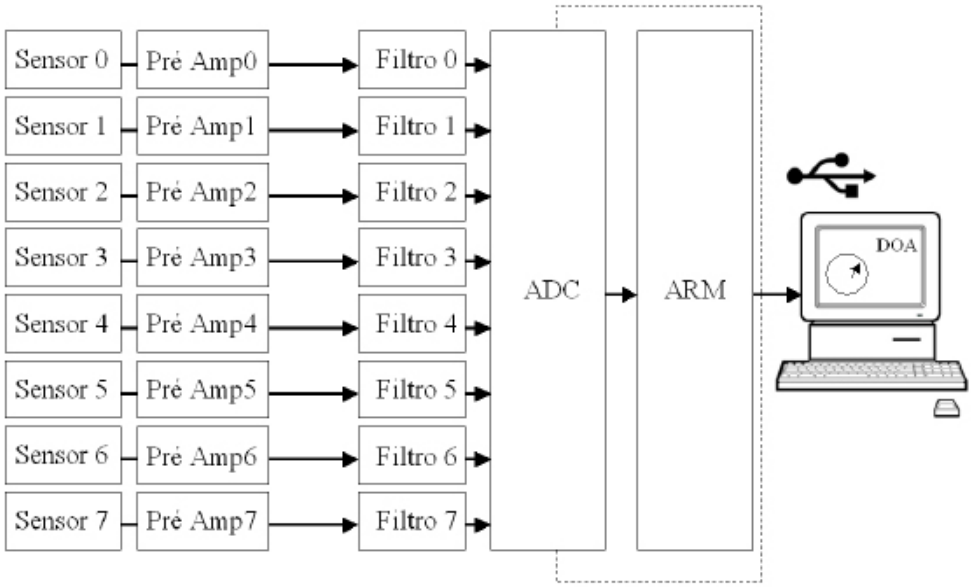

Figura 1.1: Diagrama de Blocos do Hardware do protótipo

Quanto à interface com o PC, escolheu-se utilizar a porta USB, pela velocidade e praticidade disponibilizada pela mesma. Apesar do ARM possuir um módulo USB integrado, aparentemente existem limitações não informadas pelo fabricante e por isso não se conseguia transmitir com a velocidade necessária. Conseguiu-se transmitir apenas quatro canais. Dessa forma, mal pode se trabalhar com os algoritmos de detecção.

1.3 Objetivos do projeto

A equipe responsável pelo presente projeto estabeleceu como meta conseguir entregar ao PC todos os oito canais de áudio, de forma a possibilitar o início dos testes e desenvolvimento dos algoritmos de detecção de direção de chegada.

Capítulo 2

Arquitetura ARM e AT91SAM7S

2.1 Introdução

O kit de desenvolvimento AT91SAM7S256-EK contém um microcontrolador AT91SAM7S256 e traz grandes facilidades para o projeto do firmware do microcontrolador, já que permite alimentação da placa e gravação da memória de programa através da interface USB, disponibiliza fácil acesso a todos os pinos do controlador de I/O do AT91 além de leds e botões para interface com o usuário, facilitando o teste e debug do programa.

2.2 Características

O microcontrolador presente no kit integra um processador ARM de 32-bits com arquitetura RISC de alto desempenho, instruções de 16-bits e 256 Kbytes de memória flash interna de alta velocidade. Existem duas fontes de *clock*: um circuito RC de 3 a 20 MHz e um PLL. O fabricante garante o funcionamento correto do processador a até 50 MHz.

Existem ainda vários periféricos integrados, como Controlador Avançado de Interrupção (AIC), três tipos diferentes de contadores, 32 linhas de I/O programáveis multiplexadas com até dois periféricos, interfaces USB e SPI com os respectivos controladores e um conversor A/D de oito canais e 10 bits. A seguir será dada uma descrição mais completa dos módulos utilizados do projeto: AIC, Temporizador/Contador (TC), Conversor A/D e controladores USB e SPI.

2.3 Controlador Avançado de Interrupção (AIC)

O Controlador Avançado de Interrupção possui oito níveis de prioridade, podendo-se manipular até 32 fontes de interrupção entre internas (provenientes dos periféricos do *chip*) e externas (disparadas pelos pinos do controlador), as quais são individualmente mascaráveis. Todas as interrupções podem ser programadas como sensível a nível ou a flanco, sendo que para as externas pode-se determinar ainda se o nível ou flanco será positivo ou negativo.

Cada fonte de interrupção pode ser programada independentemente no que diz respeito a seu tipo de ativação e prioridade através do registrador AIC_SMR (*Source Mode Register*). Deve-se, ainda, ativar ou desativar as fontes através dos registradores AIC_IECR (*Interrupt Enable Command Register*) e AIC_IDCR (*Interrupt Disable Command Register*) respectivamente e escrever o endereço do manipulador da interrupção no AIC_SVR correto.

A qualquer instante, verificar o estado da máscara de interrupção através da leitura do AIC_IMR (*Interrupt Mask Register*).

2.4 Temporizador/Contador (TC)

O temporizador/contador do AT91SAM7S256 inclui 3 canais idênticos de 16 bits. Cada canal pode ser programado independentemente para realizar uma grande variedade de tarefas como medição de frequência e intervalos, contagem de eventos, geração de pulsos e modulação por largura de pulso (PWM). Cada um dos canais controla um sinal de interrupção conectado ao AIC. Além disso, existem dois registradores globais que permitem iniciar os 3 canais simultaneamente com a mesma função e definir a entrada externa do *clock* de cada canal, permitindo que eles sejam encadeados.

O valor do contador (TC_CV) é incrementado a cada batida do *clock* e cada canal pode selecionar independentemente entre 5 sinais de *clock* internos, que são frações do *clock* mestre do *chip*, e 3 externos. Os sinais de *clock* interno são:

- TIMER_*clock*1 = MCK/2;
- TIMER_*clock*2 = MCK/8;
- TIMER_*clock*3 = MCK/32;
- TIMER_*clock*4 = MCK/128;
- TIMER_*clock*5 = MCK/1024.

Os *clock*s podem ser ativado/desativados e ligados/desligados através de vários registradores ou eventos diferentes. Com o *clock* desativado, as ações de ligar e desligar não surtem efeito.

Também estão presentes em cada canal dois sinais (TIOA e TIOB) cujas funções variam de acordo com o modo de operação do TC. Os modos de operação são:

- Modo captura - realiza medição de sinais;
- Modo forma de onda - realiza geração de onda.

O modo de captura funciona capturando os valores do *timer* em diferentes momentos pré-determinados, de acordo com os sinais presentes nas entradas TIOA e TIOB. Esses valores do TC são gravados em dois registradores (RA e RB).

2.4.1 O Modo Forma de Onda

No modo forma de onda o TC gera sinais nos pinos TIOA e TIOB. Comparando-se o valor do contador com o dos registradores RA e RB, determina-se o nível do pino respectivo. O registrador RC também pode ser utilizado para controlar o comportamento desses dois pinos. Assim, existem 4 formas distintas de operação dentro do modo forma de onda. Em cada uma delas, o valor do contador é reiniciado em situações distintas, mas em todos os casos TC_CV pode ser reiniciado por um evento ou gatilho externo. Essas formas são selecionadas pelo campo WAVSEL do registrador de modo de canal (TC_CMR).

2.4.1.1 Caso WAVSEL = 10

Neste modo, TC_CV é incrementado de 0 até o valor de RC, quando é reiniciado e a contagem recomeça. A figura 2.1 exemplifica isto.

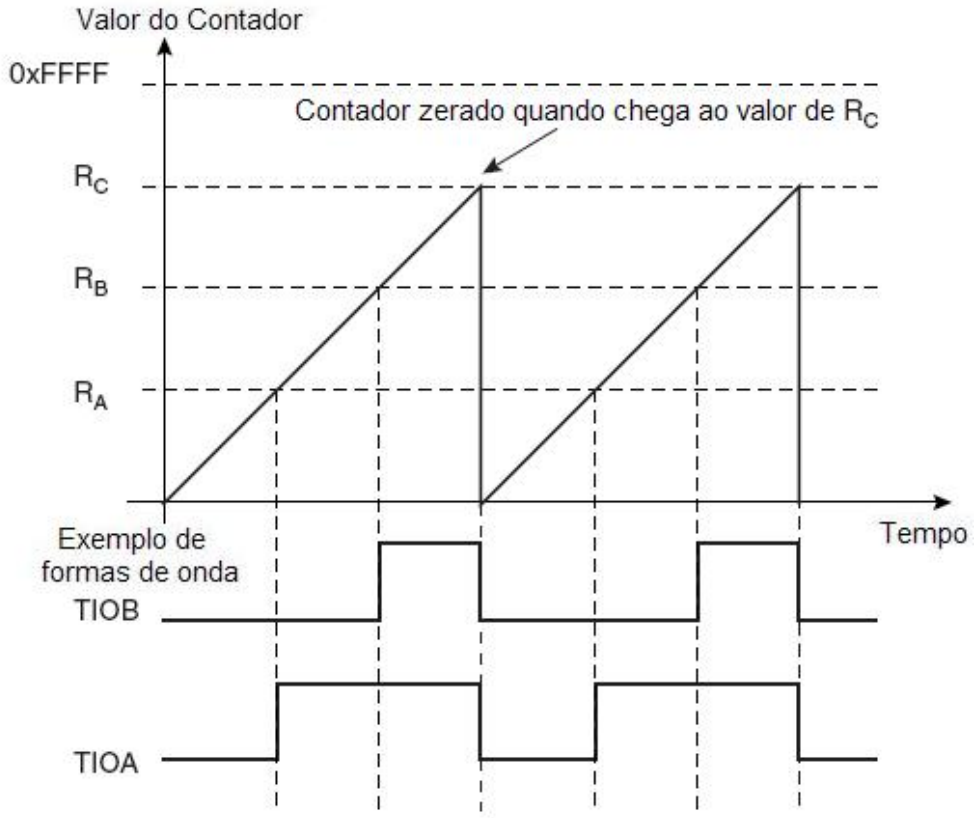

Figura 2.1: Comportamento do contador com WAVSEL = 10

2.5 Conversor Analógico/Digital (ADC)

O ADC implementado no *chip* consiste de um conversor por aproximações sucessivas com resolução de até 10 bits. A conversão vai de 0V até o valor da tensão no pino ADVREF. Ele também contém um multiplexador analógico de 8 canais.

Os canais podem ser ativados ou desativados independentemente, e após um comando de início o ADC

converte todos os canais ativos, iniciando pelo canal mais baixo. O comando de início pode ser dado por software ou por um sinal externo, existindo inclusive a opção de se configurar o temporizador/contador para iniciar as conversões. Após cada conversão o ADC ativa uma *flag* que indica o final da conversão e grava seu valor em dois registradores, um geral da última conversão e um específico do canal. Caso o registrador de dado do canal não seja lido antes da próxima conversão daquele canal, o *flag* de *overrun error* é ativado.

2.6 Porta de Dispositivo USB (UDP)

Integrado ao *chip* existe um controlador de transações USB chamado de USB Device Port (UDP). O UDP está de acordo com a especificação de USB V2.0 *full-speed*, que suporta uma velocidade máxima nominal de até 12 Mbps. Todos os tipos de transação USB (controle, *bulk*, isócrona e *interrupt*) são atendidas por pelo menos um dos quatro endpoints disponíveis no UDP. As linhas de transmissão diferencial da USB (D+ e D-) são controladas exclusivamente pelo UDP, sendo que o firmware e o processador não têm acesso direto a elas. Desta forma, o UDP se encarrega de fazer toda a sinalização, sincronização e verificação do estado das transferências independente do processador. Cabe ao processador apenas disponibilizar os dados a serem transferidos no momento correto. Toda a configuração e o controle do UDP são feitos através de registradores.

Existem 4 endpoints no UDP, cada um permitindo alguns tipos de transferência, como mostrado na tabela 2.1

Tabela 2.1: Descrição dos Endpoints do UDP

Número do endpoint	**Mnemônico**	**Dois bancos**	**Tamanho Máx. do endpoint**	**Tipo de endpoint**
0	EP0	não	8	Controle/Bulk/Interrupt
1	EP1	sim	64	Bulk/Isócrono/Interrupt
3	EP2	sim	64	Bulk/Isócrono/Interrupt
3	EP3	não	64	Controle/Bulk/Interrupt

O tamanho de cada endpoint indica quantos bytes podem ser escritos em sua FIFO antes de cada envio. As FIFOS do endpoint 1 e endpoint 2 possuem dois bancos independentes. Isto é necessário para as transmissões do tipo isócrona, que garantem uma banda determinada. Nestes casos, enquanto o *firmware* escreve em um banco, o UDP lê e transmite o outro, ou enquanto o *firware* lê um banco, o UDP recebe dados e os escreve no outro, como mostrado na figura 2.2

O UDP sinaliza para o processador o recebimento ou a transmissão correta de pacotes através de *flags* do registrado UDP_CSR do respectivo endpoint. Dessa forma, o processador sabe quando ler os dados ou disponibilizar novos pacotes para transmissão. Da mesma forma, o *firmware* sinaliza para o UDP que um novo pacote está pronto para transmissão através de *flags* de registradores.

A figura 2.3 mostra o diagrama de tempo de uma transação Data IN para um endpoint com dois bancos na FIFO, o que foi utilizado no projeto.

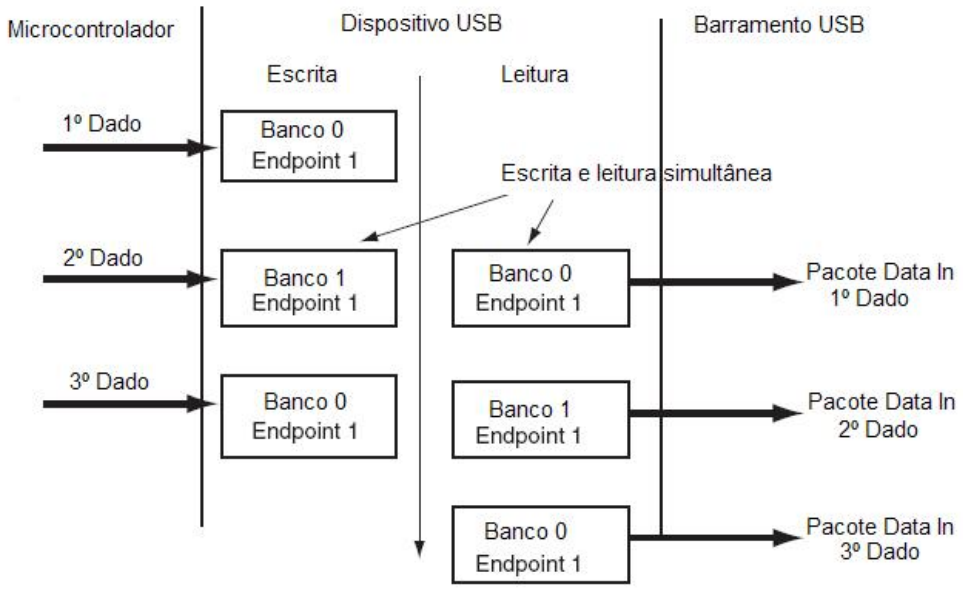

Figura 2.2: Esquema de *Ping-pong* para transferência isócrona

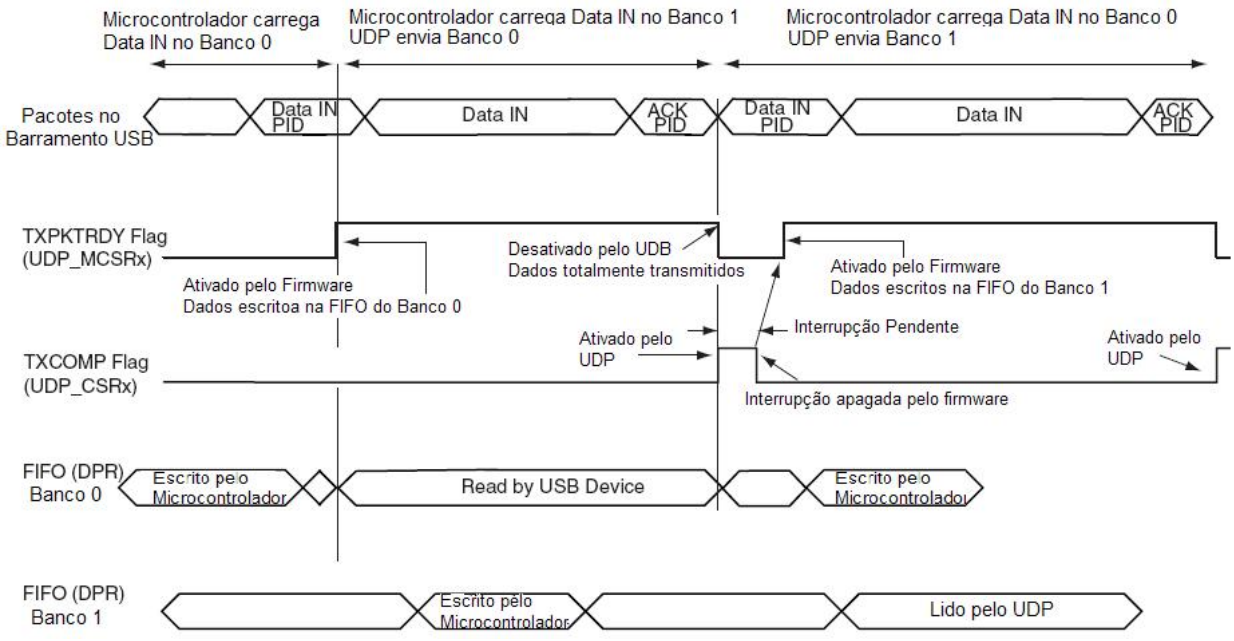

Figura 2.3: Diagrama de tempo dos processos de escrita e leitura nos bancos

Capítulo 3

Barramento USB

3.1 Introdução

Como o próprio nome diz, o padrão USB (*Universal Serial Bus*) foi desenvolvido para ser um padrão de transferência serial sólido e de aplicabilidade o mais variável possível. Suas principais características são:

- Configuração Automática;
- Conectável com o computador em operação;
- Confiabilidade (detecção de erros);
- Velocidade;
- Versatilidade;
- Baixo consumo de energia.

Essas, entre outras, tornaram o USB uma das formas mais usadas para projeto de periféricos hoje em dia e motivaram sua escolha para a interface do projeto atual com o PC.

3.2 Visão Geral

O padrão USB atende a três velocidades:

- *High Speed* - 480Mbits/s (USBv2.0);
- *Full Speed* - 12Mbits/s (USBv1.1);
- *Low Speed* - 1.5Mbits/s (USBv1.0).

O USB (Barramento Serial Universal) é um padrão onde toda a transmissão é controlada por apenas um dos lados da transmissão, o *host*. A especificação em si não prevê meios para que haja um arranjo com mais de um *host*. O USB usa uma topologia do tipo estrela, similar a usada em redes do tipo *ethernet*. Isso impõe que em algum lugar nesse sistema exista um *hub*. Esse tipo de topologia tem algumas vantagens além de apenas conectar os dispositivos. Primeiramente, a alimentação para cada dispositivo pode ser monitorada e até desligada caso ocorra uma sobrecorrente, assim não comprometendo os outros dispositivos conectados ao barramento. Dispositivos High, Full e Low Speed são todos suportados, pois o *hub* faz uma filtragem das velocidades. Até 127 dispositivos podem ser conectados a um único barramento USB e a qualquer tempo, mesmo com o PC ligado e já com o sistema operacional rodando.

USB, como o nome sugere, é um barramento serial. Ele usa 4 fios, sendo que dois são para alimentação (+5V e GND). Os dois fios restantes são um par trançado que transmitem os dados de maneira diferencial (D+ e D-). Ele usa um esquema de codificação NRZI (*Non Return to Zero Invert*) para enviar dados, e esses dados contêm um campo *sync* para sincronizar o *clock* do *host* e do dispositivo. USB suporta *plu'n'play* com carregamento e descarregamento automático dos *drivers*. O usuário apenas conecta o dispositivo no barramento. O *host* irá detectar essa adição, "interrogar"o novo dispositivo inserido e carregar o *driver* apropriado, tudo isso ocorrendo instantaneamente. O carregamento do *driver* apropriado é feito utilizando a combinação PID/VID (*Product ID* e *Vendor ID*) que são passados ao *host* em parte do "interrogatório"citado acima. O usuário final não tem que se preocupar com nenhum termo técnico como IRQ, porta, endereço , etc. Quando o usuário não quiser mais usar o dispositivo ele deve apenas desconectá-lo diretamente e o *host* irá detectar sua ausência e automaticamente descarregar o *driver*.

Outro ponto interessante do USB são seus modos de transferência. Ele suporta transferências do tipo controle, *interrupt*, *bulk* e isócrona. Mais a frente serão citadas os outros tipos de transferência, mas é bom ter em mente que a transferência isócrona permite que um dispositivo assegure uma determinada largura de banda com uma latência garantida. Isso é ideal para aplicações em audio e vídeo onde precisa-se de aquisição de dados em tempo real, pois qualquer atraso na transmissão significa perda de dados. Cada tipo de transferência citado possui *trade-offs* entre detecção e correção de erros, latência garantida de largura de banda.

3.2.1 Identificação da Velocidade do Dispositivo

Um dispositivo USB deve indicar sua velocidade colocando um resistor de *pull-up* no pino D+ para indicar que é *full speed* ou então no pino D- para indicar que é *low speed*. A figura 3.1 ilustra um dispositivo *full speed*. Esses resistores de *pull-up* também servirão para que o *host* ou o *hub* detectem a presença de um dispostivo conectado em sua porta. Sem um resistor de *pull-up* supõe-se que não há nada conectado ao barramento. Alguns componentes têm esse resistor integrados no silício, assim podendo ser ligados e desligados pelo próprio *firmware*, outros precisam de um resistor externo.

O fato de não ter sido mencionada a forma como um dispositivo *high speed* faz para se identificar é porque ele se conecta como se fosse um dispositivo *full speed*. Depois de ser reconhecido ele fará um procedimento para se identificar como *high speed*, e abrirá esse tipo de conexão caso o *hub* suporte. Se o dispositivo estiver operando em modo *high speed* então o resistor de *pull up* será removido para balancear a linha.

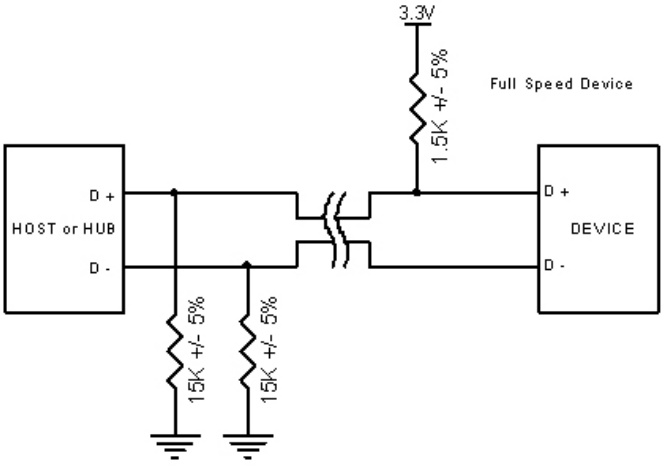

Figura 3.1: Exemplo de ligação de um dispositivo *Full-Speed*

3.3 Protocolos USB

Cada transação USB consiste de:

3.3.1 Pacote Token (Cabeçalho definindo o que virá a seguir)

Existem três tipos de pacotes token,

- IN: Informa o dispositivo USB que o *host* requisita informações;
- OUT: Informa o dispositivo USB que o *host* quer enviar informações;
- Setup: Usado para iniciar transferências de controle.

3.3.2 Pacote Data (É opcional e contém o dado)

Exitem dois tipos de pacotes Data, cada um capaz de transmitir até 1024 bytes de dados,

- Data 0;
- Data 1.

O modo *high speed* define mais dois tipos de pacote data, DATA2 e MDATA.

3.3.3 Pacote Status (Usado para garantir que os pacotes foram entregues e para prover um meio de correção de erros)

- **Pacotes Handshake**
  - **ACK**: Informa que o pacote foi recebido com sucesso;

  - **NAK**: Informa que o dispositivo não pode enviar ou receber dados;
  - **STALL**: O dispositivo se encontra em um estado que requer intervenção do *host*.

- **Pacotes Start of Frame** : Consiste em um número de *frame* de 11 bits que é enviado pelo *host* a cada 1ms em um barramento *full speed* ou a cada 125us em um barramento *high speed*.

3.3.4 Campos de um Pacote USB

Os pacotes USB consistem nos seguinte campos:

Sync: Todo pacote deve começar com o campo sync. Esse campo tem 8 bits em dispositivos *low* e *full speed* e 32 bits em dispositivos *high speed* e é usado para sincronizar o clock do *host* e do dispositivo. Os últimos dois bits indicam onde o campo PID começa.

PID: PID significa *Packet ID* (identificação do pacote). Esse campo é utilizado para identificar o tipo de pacote que está sendo enviado. Existem quatro bits para o PID, no entanto para garantir que ele será recebido corretamente os quatro bits são complementados e repetidos, obtendo assim um PID de 8 bits. A tabela 3.1 mostra os valores possíveis para o PID:

Tabela 3.1: Valores possíveis para campo PID

Grupo	Valor PID	Identificador de Pacote
Token	0001	OUT Token
	1001	IN Token
	0101	SOF Token
	1101	SETUP Token
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK Handshake
	1010	NAK Handshake
	1110	STALL Handshake
	0110	NYET (No Response Yet)
Special	1100	PREamble
	1100	ERR
	1000	Split
	0100	Ping

ADDR: Esse campo identifica o endereço do dispositivo para o qual o pacote está destinado. Como tem 7 bits o máximo de dispositivos suportados é 127. O endereço 0 não é válido pois qualquer dispositivo que ainda não tem endereço deve responder a pacotes enviados para o endereço 0.

ENDP: Possui 4 bits, permitindo assim até 16 *endpoints*. Dispositivos *low speed* só podem utilizar 4 *endpoints* no máximo (já incluindo o *pipe* padrão).

CRC: Verificação de redundância cíclica, é aplicada nos dados transportados pelo pacote. Pacote Token tem um CRC de 5 bits enquanto pacotes Data têm um CRC de 16 bits.

EOP: Fim de pacote.

3.4 Nomenclatura USB

Nos itens a seguir serão descritas definições importantes para entender o funcionamento do protocolo USB.

3.4.1 Funções USB

Podem ser vistas como dispositivos USB que têm uma capacidade ou uma função, como uma impressora, *pen drive*, *scanner* ou outro periférico qualquer que se ligue à porta USB.

3.4.2 Endpoints

Podem ser descritos como fontes ou sorvedouros de dados. Como o barramento é centrado no *host* (assim os *endpoints* não podem controlar transações) eles ocorrem no final do canal de comunicações com a função USB. Todos dispositivos devem suportar o *endpoint 0*, pois esse é o *endpoint* que recebe todos os requerimentos de controle e estado do dispositivo durante a enumeração e enquanto o dispositivo está operacional no barramento.

3.4.3 Enumeração

É o processo de determinar qual dispositivo acabou de ser conectado ao barramento e quais parâmetros ele requisita, tais como potência que irá consumir, número e tipo de *endpoints*, classe do dispositivo, etc. O *host* então irá designar um endereço para o dispositivo permitindo assim que ele transmita dados pelo barramento.

3.4.4 Pipes

Enquanto o dispositivo envia e recebe dados em uma série de *endpoints*, o *software* instalado no *host* transfere dados através dos *pipes*. O *pipe* é uma conexão lógica entre o *host* e o(s) *endpoint*(s).

3.5 Tipos de Transferência/Endpoint

A especificação USB define quatro tipos de transferência/*endpoint*:

- Controle
- *interrupt*

- Isócrona
- *bulk*

3.5.1 Transferências de Controle

São tipicamente utilizadas para operações de comando e de estado. Elas são essenciais para configurar um dispositivo USB com todas as funções de enumeração sendo feitas usando transferências de controle.

Uma transferência de controle pode ter até três estágios.

- O estágio de *setup* é onde a requisição é enviada. Se a função receber os dados com sucesso (CRC, PID estarem corretos), ela responderá com um ACK, caso contrário, ela ignora os dados e não manda nenhum pacote de *handshake*;
- O estágio de dados (que é opcional) consiste em uma ou múltiplas transferências IN ou OUT;
- O estágio de estado reporta o estado do pedido como um todo e varia de acordo com a direção da transferência. Reportar o estado é sempre feito pelo dispositivo.

3.5.2 Transferências do tipo *interrupt*

- Latência Garantida;
- Stream Pipe - Unidirecional;
- Detecção de erro e nova tentativa.

3.5.3 Transferências do tipo *bulk*

- Usada para transferir grande quantidade de dados;
- Detecção de erro por CRC com garantia de entrega;
- Não garante largura de banda ou latência mínima;
- *Stream Pipe* - Unidirecional;
- Apenas modos *full* e *high speed*.

3.5.4 Transferências do tipo Isócrona

Esse tipo de transferência ocorre periodicamente. Elas costumam ter conteúdo sensível ao tempo, como uma transmissão de áudio ou vídeo em tempo real. Se houvesse um atraso ou uma tentativa de retransmissão de um dado não recebido em uma transmissão de áudio em tempo real então seria ouvido sons atrasados e a sincronia seria perdida. No entanto, se um pacote é perdido de vez em quando é muito mais difícil de ser notado pelo ouvinte. Algumas características são:

- Acesso garantido à uma determinada largura de banda no barramento USB;
- Latência limitada;
- Stream Pipe - Unidirecional;
- Detecção de erro por CRC, mas sem nova tentativa de transmissão ou garantia de entrega;
- Apenas modos full e high speed;
- Sem alternância entre pacotes DATA0 e DATA1.

O tamanho máximo do pacote de dado que pode ser transmitido é especificado no descritor do *endpoint* a ser usado nessa transmissão. Isso pode ser até um máximo de 1023 bytes para um dispositivo *full speed* e 1024 bytes para um dispositivo *high speed*. Como esse tamanho máximo vai afetar os requerimentos de largura de banda do barramento, é interessante especificar um tamanho razoável. Se é necessário usar um tamanho máximo grande, é vantajoso especificar uma série de interfaces alternativas com tamanho máximos diferentes, pois se durante a enumeração o *host* não puder habilitar o seu tamanho máximo desejado devido à restrições de largura de banda disponível, o dispositivo ainda terá outra alternativa além de falhar completamente nessa etapa.

Os dados que estão sendo transmitidos em um *enpoint* isócrono podem ser menores que o tamanho negociado anteriormente e podem variar de tamanho de transação para transação. Transações isócronas não possuem um estágio de *handshake* e não podem reportar erros ou as condições STALL e HALT.

3.5.5 Gerenciando a Largura de Banda

O *host* é responsável por gerenciar a largura de banda do barramento. Isso é feito durante a enumeração quando *endpoint* do tipo isócrono e *interrupt* são configurados e durante a operação do barramento. A especificação coloca alguns limites no barramento, permitindo não mais do que 90% de cada frame ser alocado para transferências periódicas (*interrupt* e isócrona) em um barramento *full speed*. Em barramentos *high speed* essa limitação se reduz a não mais que 80% de um microframe pode ser alocado para transferências periódicas.

Então pode notar-se que em um barramento altamente saturado por transferências periódicas, os 10% restantes serão para transferências de controle e assim que essas tiverem sido alocadas então as transferências *bulk* terão o que restou do barramento.

3.6 Descritores USB

Todos dispositivos USB têm uma hierarquia de descritores que descrevem para o *host* informações como o que o dispositivo é, quem o fez, qual a versão USB ele suporta, de quantas maneiras ele pode ser configurado, o número de *endpoints* e seus tipos, etc.

Os descritores USB mais comuns são:

- Descritor de dispositivo;

- Descritor de configuração;
- Descritor de interface;
- Descritor de endpoint;
- Descritor de.

Cada descritor consiste em uma série de campos, e todos eles usam prefixos para indicar o formato do conteúdo dos dados naquele campo. Esses prefixos são: b = byte , w = word (2 bytes), bm = bit map, bcd = binary coded decimal (binário codificado em decimal), i = index, id = identificador.

3.6.1 Descritor de Dispositivo

Representa o dispositivo como um todo. Assim cada dispositivo USB pode ter apenas um descritor desse tipo. Ele especifica alguns informações básicas sobre o dispositivo como qual versão da USB o dispositivo suporta, as identificações do produto e do fabricante (que são utilizados para carregar o *driver* apropriado), o número de configurações possíveis (isso também indica o número de blocos com descritores de configuração existirão).

3.6.2 Descritores de Configuração

Embora os dispositivos USB possam ter várias configurações, a maioria deles é simples e possui somente uma. O descritor de configuração especifica valores como a potência total que esta configuração usa, se o dispositivo tem alimentação própria ou será alimentado pelo barramento e o número de interfaces existentes nessa configuração. Quando um dispositivo é enumerado o *host* obtem os descritores e decide qual configuração será utilizada, podendo utilizar apenas uma configuração de cada vez.

3.6.3 Descritores de Interface

O descritor de interface pode ser visto como um cabeçalho ou grupamento de *endpoints* em um conjunto funcional exercendo uma única função. Por exemplo, pode-se ter um dispositivo multifuncional com funções de *fax*, *scanner* e impressora. Teríamos três descritores de interface. O primeiro seria responsável pela função *fax*, o segundo seria responsável pela função *scanner* e o terceiro pela função impressora. Diferentemente dos descritores de configuração, não existem limitações em ter mais de uma interface ativa ao mesmo tempo.

3.6.4 Descritor de Endpoint

Cada descritor de *endpoint* é usado para especificar o tipo de transferência, direção, intervalo de polling e o tamanho máximo do pacote. O *endpoint* zero é sempre considerado um *endpoint* de controle e assim não precisa de descritor, apenas seu tamanho máximo de pacote é declarado, mas isso é feito do descritor de dispositivo. O *host* usará as informações obtidas por esses descritores para determinar a alocação de largura de banda no barramento.

3.6.5 Descritores de String

São usados para dar informações legíveis em linguagem humana. São opcionais e caso não sejam usados qualquer campo que seja um índice para um descritor de *string* deve ser colocado em zero para indicar que não há nenhuma *string* associada.

3.7 O Pacote de Setup

Todo dispositivo USB deve responder a pacotes de *setup* no *pipe* padrão. Os pacotes de *setup* são usados para detecção e configuração do dispositivo e possuem algumas funções, como dar um endereço ao dispositivo, requisitar um descritor ou checar o estado de um *endpoint*.

3.8 Requisições Padrão

São comuns a todos os dispositivos USB e serão detalhadas a seguir. Requisições de classe são comuns para classes de *drivers*. Por exemplo, todos os dispositivos que sejam da classe áudio terão uma lista de requisições específicas dessa classe.

A tabela 3.2 resume as onze requisições padrão da USB. Após a tabela há uma descrição breve sobre cada requisição. Todo dispositivo deve responder a todas estas requisições (mesmo que a resposta seja apenas um STALL).

Tabela 3.2: Requisições Padrão

Número da requisição (bRequest)	**Requisição (estágio de Dados)**	**Fonte de dados**	**Recipiente**	**wValue**	**wIndex**	**wLength**	**Dado (no estágio de Dados)**
00h	Get_Status	Dispositivo	dispositivo, inteface, *endpoint*.	0	dispositivo interface, ou *endpoint*.	2	status
01h	Clear_Feature	Não há estágio de dados	dispositivo inteface, *endpoint*.	feature	dispositivo interface, ou *endpoint*.	-	-
03h	Set_Feature	Não há estágio de dados	dispositivo inteface, *endpoint*.	feature	dispositivo interface, ou *endpoint*.	-	-
05h	Set_Address	Não há estágio de dados	dispositivo	Endereço do dispositivo	0	-	-
06h	Get_Descriptor	Dispositivo	dispositivo	Tipo do descritor e índice	dispositivo ou language ID	Tamanho do descritor	descritor
07h	Set_Descriptor	*host*	dispositivo	Tipo do descritor e índice	dispositivo ou language ID	Tamanho do descritor	descritor
08h	Get_Configuration	Dispositivo	dispositivo	0	dispositivo	1	configuração
09h	Set_Configuration	Não há estágio de dados	dispositivo	configuração	dispositivo	-	-
0Ah	Get_Interface	Dispositivo	Interface	0	Interface	1	*alternate setting*
0Bh	Set_Interface	Não há estágio de dados	interface	"*alternate setting*"	interface	-	-
0Ch	Synch_Frame	Dispositivo	*endpoint*	0	*endpoint*	2	Número do Frame

3.8.1 Get Status

Usado para: O *host* requisita o status de uma interface ou de um *endpoint*.

Comportamento quando ocorre algum erro: O dispositivo retorna um STALL caso a interface ou o *endpoint* não exista.

3.8.2 Clear Feature

Usado para: O *host* requisita que uma característica de uma interface ou de um *endpoint* seja desabilitada.

Comportamento quando ocorre algum erro: Se a interface ou o *endpoint* não existir ou se a característica não puder ser desabilitada o dispositivo responde com um STALL.

3.8.3 Set Feature

Usado para: O *host* requisita que uma característica de uma interface ou de um *endpoint* seja habilitada.

Comportamento quando ocorre algum erro: Se a interface ou o *endpoint* não existir o dispositivo responde com um STALL.

3.8.4 Set Address

Usado para: O *host* especifica um endereço para utilizar em futuras comunicações com o dispositivo.

Comportamento quando ocorre algum erro: não especificado.

3.8.5 Get Descriptor

Usado para: O *host* requisita um descritor específico.

Comportamento quando ocorre algum erro: Quando o dispositivo recebe um pedido que não suporta deve retornar um STALL.

3.8.6 Set Descriptor

Usado para: O *host* adiciona um descritor ou atualiza um que já existe.

Comportamento quando ocorre algum erro: Quando o dispositivo recebe um pedido que não suporta de responder com um STALL.

3.8.7 Get Configuration

Usado para: O *host* requisita o valor da configuração que está ativa.

Comportamento quando ocorre algum erro: não especificado.

3.8.8 Set Configuration

Usado para: O *host* requisita que o dispositivo use a configuração especificada.

Comportamento quando ocorre algum erro: se wValue não for igual a zero ou a configuração não for suportada pelo dispositivo este deverá retornar um STALL.

3.8.9 Get Interface

Usado para: Em dispositivos que possuem interfaces mutuamente exclusivas (alternativas de uma mesma interface), o *host* requisita a alternativa que está ativa no momento.

Comportamento quando ocorre algum erro: Se a interface não existir o dispositivo deve retornar um STALL.

3.8.10 Set Interface

Usado para: Em dispositivos que possuem interfaces mutuamente exclusivas (alternativas de uma mesma interface), o *host* requisita que a alternativa escolhida seja utilizada.

Comportamento quando ocorre algum erro: Se o dispositivo suporta apenas uma interface padrão ou se alternativa escolhida não existe o dispositivo deve retornar um STALL.

3.8.11 Synch Frame

Usado para: O dispositivo escolhe e reporta um *frame* de sincronização para o *endpoint*.

Comportamento quando ocorre algum erro: Quando o dispositivo recebe um pedido que não suporta deve responder com um STALL.

Capítulo 4

A Classe de Audio

4.1 Introdução

A classe Audio serve para dispositivos que enviam ou recebem informação de áudio, que pode conter voz, música e outros tipos de sim. Dispositivos de áudio podem realizar tanto transações isócronas para fluxos de áudio quanto transmissões *bulk*, no caso de informação codificada utilizando o protocolo MIDI. Os mecanismos de transporte de áudio definidos para essa classe são o mais padronizados possível, visando garantir a interoperabilidade para diferentes fabricantes e mantendo os drivers genéricos. Uma função áudio é considerada uma caixa fechada com mecanismos definidos de comunicação com o mundo exterior. Estes mecanismos são as interfaces de áudio.

Cada função deve conter uma única interface de controle de áudio (*AudioControl* - AC) e pode ter nenhuma ou mais interfaces de fluxo de áudio (*AudioStreaming* - AS). A interface AudioControl serve para acessar os controles e comandos da função de áudio enquanto a *AudioStreaming* serve para transportar fluxos de áudio para dentro e para fora da função. O conjunto de uma *AudioControl* e as várias *AudioStremings* é conhecido como uma coleção de interfaces de áudio (*Audio Interface Collection* - AIC). O diagrama da figura 4.1 mostra o arranjo de uma coleção.

As funcionalidades relacionadas com o controle de parâmetros que influenciam diretamente a percepção do áudio (como volume e equalização) estão localizadas no retângulo central e são acessadas exclusivamente pela interface AudioControl. Fluxos de entrada e saída de áudio são manipulados por interfaces AudioStreaming distintas.

4.2 Interface AudioStreaming

As interfaces *AudioStreaming* realizam a transmissão de fluxos de áudio entre o host e o dispositivo. Cada interface *AudioStreaming* pode ser associada a no máximo um *endpoint* isócrono. Uma interface *AudioStreaming* pode ter ajustes alternativos que podem ser usados para mudar características da interface e do *endpoint* associado a ela, como o tamanho do *subframe* ou o número de canais transmitidos. Para cada *endpoint* isócrono definido em qualquer das interfaces *AudioStreaming* deve haver um terminal de entrada ou saída definido na função de áudio. O host precisa da informação dos descritores do *endpoint*,

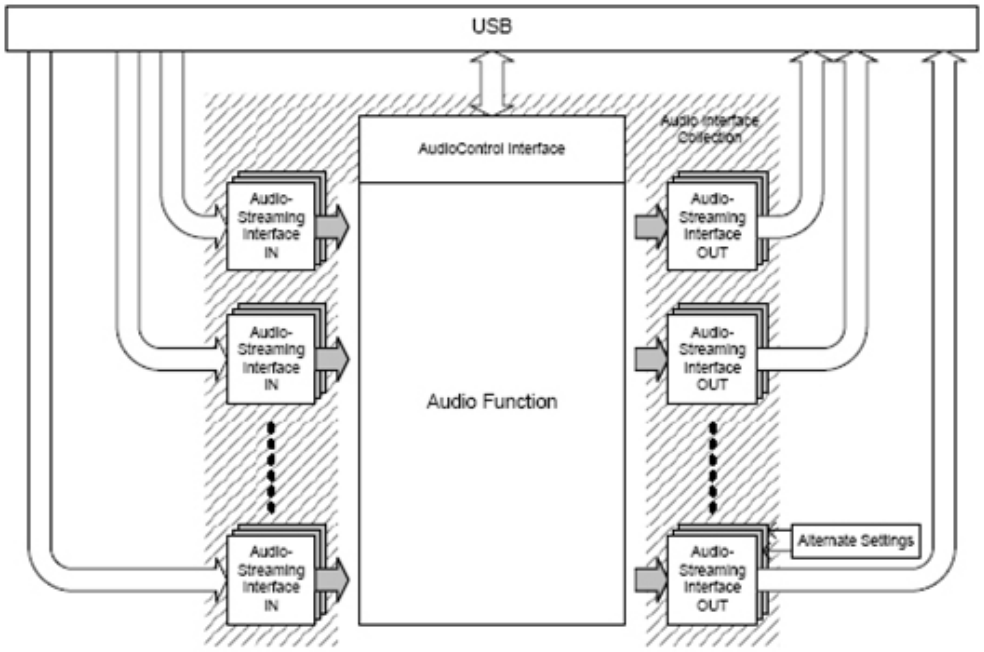

Figura 4.1: Visão Global da Função de Audio

do terminal e da interface para entender completamente as conexões.

4.3 Interface AudioControl

A interface de controle de áudio é o canal por onde o *host* envia todos os comandos e solicitações que dizem respeito às alterações dos parâmetros do som. Ela é a responsável por interpretar os pacotes de controle recebidos do *host* e indicar para a função de áudio o que deve ser feito. Para que o *host* saiba quais as funcionalidades existentes no dispositivo, este deve informar a sua topologia para o *host*. Como o dispositivo de áudio utilizado no projeto apenas fornece o sinal, sem permitir maiores manipulações, ele é composto apenas de um terminal de entrada conectado a um terminal de saída.

4.3.1 Terminal de Entrada

O terminal entrada serve para representar uma fonte de informação de áudio para a função. É a interface entre as unidades e o mundo externo. Caso a fonte de dados seja um *endpoint* OUT, o *host* usa as informações do descritor do *endpoint*, do descritor da interface AudioStreaming e do descritor do Terminal de Entrada para entender completamente as funcionalidades do terminal. Geralmente, o sinal de áudio chega ao dispositivo com alguma forma de codificação, como PCM ou MPEG.

4.3.2 Terminal de Saída

O terminal de saída é um sorvedouro da informação de áudio presente dentro da função. Ele conecta as unidades internas ao mundo exterior, servindo como uma saída para os canais de áudio ainda não codi-

ficados, antes que eles possam ser transformados em um fluxo de áudio a ser transmitido ou utilizado. Os canais de áudio internos à função se conectam ao terminal de saída por um único pino. Um Terminal de Saída não necessariamente representa um *endpoint* IN. Ele pode ser uma caixa de som ou um conector RCA, por exemplo.

Capítulo 5

Implementação

Neste capítulo estão descritas as etapas do projeto bem como as soluções utilizadas

5.1 Visão Geral

5.1.1 O Hardware

O primeiro passo da implementação consistiu no teste da placa de aquisição de dados. Logo de princípio notou-se problemas de alimentação tanto das placas em si quanto em amplificadores operacionais utilizados nos filtros e no somador de tensão responsável pelo nível DC do sinal. Além disso, os níveis DC de cada linha tinham um comportamento estranho, existindo em algumas linhas mas não em outras, variando entre 0 e 5 V, aparecendo e desaparecendo dependendo se os microfones estivessem ou não conectados. Havia também vários pontos na placa com soldas refeitas e fios ligando pontos não conectados. Foi decidido então pela construção de uma nova placa. Desta vez ela seria produzida por uma empresa especializada ao invés de usar o laboratório de circuitos impressos da UnB, visando minimizar erros de construção e roteamento e obter um resultado mais preciso. Esta seria única, pois o arranjo de duas placas sobrepostas utilizado até então impossibilitava a maioria das medições e testes na placa inferior, que era exatamente a que estava com o funcionamento um pouco melhor. As mudanças na nova placa foram:

- O estágio responsável pelo nível DC do sinal de saída;
- *Slots* para o ADC de 12 bits, prevendo dois possíveis encapsulamentos;
- Pinos de saída para comunicação com um possível DSP e conversor D/A.

Foi feito então o *layout* e roteamento da nova placa, e estes foram enviados à empresa responsável pela confecção. Como a confecção leva algum tempo, passou-se à próxima etapa do projeto: o *firmware* do microcontrolador para transferência de dados para o PC.

5.1.2 O Software

Após análise do código já existente, o qual não conseguia transmitir na velocidade necessária, enumeramos algumas possíveis causas para o problema:

- A não utilização dos dois bancos disponíveis no *endpoint* 1 da forma descrita no *datasheet*. Até então o procedimento consistia em preencher um *buffer* inteiro, transferí-lo para a FIFO do *endpoint*, e liberar a transmissão. Mas enquanto esta ocorria, o segundo banco não era preenchido.
- Durante a escrita dos dados na FIFO a amostragem do sinal pelo ADC não ocorria.

5.2 Soluções Testadas

As primeiras modificações no *firmware* visavam sanar tais problemas. Foi desenvolvido um novo algoritmo para controlar a aquisição do dado, a escrita deste nos bancos do UDP e a transferência via USB. Com este novo algoritmo, os dois bancos estavam sendo utilizados de forma correta e a amostragem, desencadeada por uma interrupção, ocorria a qualquer momento. Foi observado que a transferência de dados após tais mudanças ocorria de forma ligeiramente mais ágil. Entretanto, continuavam-se perdendo dados quando se utilizava mais de quatro canais. Após algumas tentativas de se reduzir o código e a rotina ao máximo, utilizando o mínimo de comparações e mantendo apenas os blocos estritamente necessários, ainda persistia o problema. Então um pino do microcontrolador passou a ser colocado no nível lógico '1' no início de cada transmissão e em nível '0' ao final da mesma. Assim pode-se medir a duração da transmissão com um osciloscópio. O resultado é mostrado na figura 5.1.

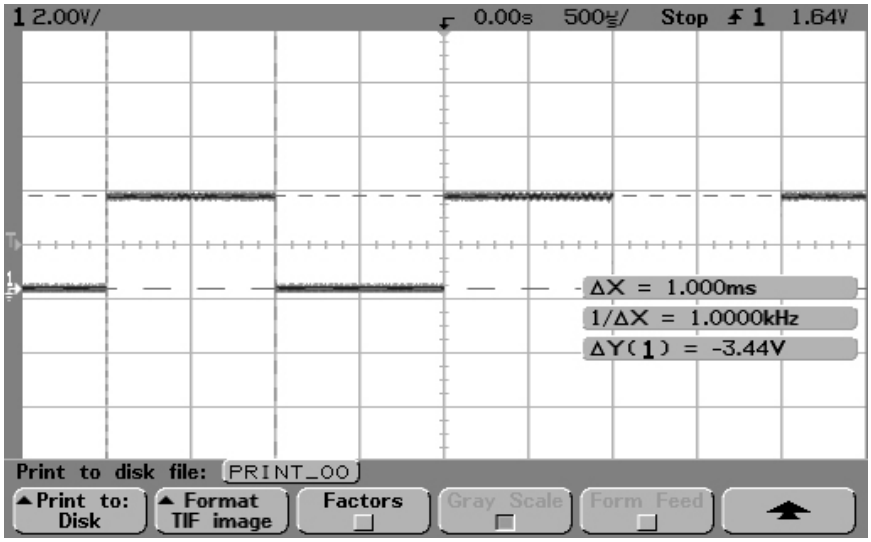

Figura 5.1: Tempo da transferência

Como se observa, cada transmissão levava aproximadamente 1 ms. Como o tamanho máximo dos *endpoints* presentes no AT91SAM7S é 64 bytes, temos uma taxa de 64 KBps, que condiz com os resultados, pois utilizando 4 canais do ADC, gera-se exatamente esse valor. A partir daí, quantos mais canais se adicionava, maior era o número de dados perdidos.

Partiu-se então para uma abordagem diferente, verificando os descritores para descobrir se o problema não estava na configuração do dispositivo USB. Após constatar que os descritores estavam corretos, o próximo passo foi tentar utilizar duas interfaces do dispositivo para transmissão. A especificação USB prevê tal caso. Para isso devem-se acrescentar mais um descritor de interface e um de *endpoint* (associado a essa nova interface) e algumas mudanças nos descritores já existentes (tais como o número de interfaces

utilizadas, o tamanho total em bytes dos descritores, etc.). Agora um novo descritor deveria ser inserido, o descritor de associação de interfaces, pois teríamos duas interfaces realizando a mesma função. Quando o dispositivo foi conectado com esses novos descritores o PC o identificou como sendo dois dispositivos distintos de áudio e ambos com problemas na instalação do *driver*.

Atribuímos o ocorrido à uma questão de drivers, pois os *drivers* genéricos do Windows para dispositivos de áudio não cobririam tal situação. Assim seria necessária a escrita de um *driver* específico para esse dispositivo. Essa idéia foi abandonada pois esse processo requereria muito tempo e conhecimento e fugiria muito do escopo do projeto, além de não se ter certeza de que resolveria o problema, porque havia a desconfiança de que a placa não suportaria a velocidade desejada.

Surgiu então a idéia de utilizar algum tipo de compactação ou codificação para usar menos bits por amostra, uma vez que cada amostra tinha no máximo 12 bits (para o caso de se usar um A/D externo) e gastávamos 16 bits para transmitir cada uma. Uma das soluções seria usar as codificações lei A ou lei u, já que essas são vastamente utilizadas em telefonia digital e já estavam previstas como tipo de codificação para classe áudio da especificação USB. Outra solução apresentada seria usar a modulação delta, transmitindo a primeira amostra com 16 bits e todas as amostras seguintes seriam apenas de 8 bits e serviriam de incremento para o valor anterior. Uma desvantagem desse padrão é que não está especificado na classe áudio, dificultando assim uma possível implementação em tempo real.

Pela simplicidade e praticamente nenhuma distorção para esta aplicação o padrão com modulação delta foi escolhido. Dessa forma os 8 canais são transmitidos utilizando a velocidade disponível.

5.2.1 O novo *firmware*

O *firmware* desenvolvido para o microcontrolador é dividido em quatro blocos funcionais:

- Principal (loop infinito);
- Temporizador;
- Conversor Analógico/Digital;
- Dispositivo USB (UDP).

5.2.1.1 O Módulo Principal

O programa principal serve para inicializar todos os periféricos utilizados na placa. Ele executa as rotinas para configurar os pinos de I/O como entradas (no caso do ADC) ou saída (para controle do *sample and holder*); o modo de funcionamento do temporizador, de forma a gerar as interrupções na freqüência de amostragem; ativa o conversor analógico/digital e configura periférico USB do microcontrolador. Após toda a inicialização, o programa entra em um *loop* infinito onde verifica se o UDP está configurado e se a interface é a operacional. Após tais verificações, no caso da FIFO estar completamente cheia, o programa informa tal situação para o UDP que inicia a transferência do pacote USB. O fluxograma na figura 5.2 representa esse procedimento

5.2.1.2 O Módulo do Temporizador (TIMER)

A função de inicialização do temporizador, chamada pela rotina principal, habilita o canal zero do temporizador com a seguinte configuração:

- O *clock* selecionado como 1/8 do clock da placa (MCK/8);
- Modo forma de onda (WAVEFORM);
- Contador reiniciado quando atinge o valor de RC (WAVSEL = 10b);
- Interrupção gerada no reinício do contador.

Como o *clock* principal da placa é de 48 MHz, temos que o contador será incrementado a uma freqüência de 6 MHz (período de 0,1667 us). Ajustando-se o valor de RC para 749 teremos um período aproximado de 0,125 ms e portanto as interrupções serão geradas a uma freqüência de 8 kHz.

A rotina de tratamento de interrupção apenas limpa a *flag* que gerou a interrupção e verifica se o ADC não está no meio de um grupo de conversões. Nesse caso, ele chama a rotina que inicializa a obtenção das amostras, e ativa o *flag* que indica conversões em curso. Caso contrário, simplesmente ignora o pedido de interrupção.

5.2.1.3 O Módulo do Conversor Analógico/Digital (ADC)

Primeiramente, a função de configuração do ADC o habilita para funcionar da seguinte maneira:

- Disparo por *software*;
- Resolução de 10 bits;
- Modo normal (sem desativação para economia de energia);
- *Clock* de 4 MHz para conversões;
- Tempo de inicialização de 18 us;
- Tempo de inicialização do *sample and holder* de 1 us.

As conversões são iniciadas pela rotina executada a cada interrupção. Nela, o ADC é disparado e executa a conversão em sequência de cada canal habilitado. Após o término de cada conversão, seu valor é guardado e tratado da seguinte forma:

Caso seja o primeiro grupo de dados, ele os envia completos (10 bits) através da USB. A partir de então, ele compara o valor da amostra atual de cada canal com o valor da amostra anterior, enviando apenas o incremento. Este pode assumir valores entre -127 e 127, pois é de 8 bits. Caso o módulo da diferença seja maior que isso, trunca-se o incremento no máximo possível (+127 ou -127) e o valor guardado como amostra anterior será atualizado para ficar condizente com o incremento aplicado, minimizando erros. O procedimento está ilustrado na figura 5.3.

5.2.1.4 O Módulo do Dispositivo USB (UDP)

Neste módulo estão definidos todos os descritores que serão usados durante a enumeração. Ele contém ainda a rotina de habilitação do UDP, ajustando seu *clock* e o resistor de *pull-up* correto. Além disso, estão definidas nele a rotina que verifica se o periférico está configurado, a rotina responsável pela enumeração e a rotina responsável pela transmissão. As duas últimas são as mais importantes e serão explicadas abaixo.

5.2.1.5 Enumeração

A rotina de enumeração é chamada quando se verifica uma transferência de controle, feita através do *endpoint* 0. Ela identifica o tipo de pedido enviado pelo *host*, interpretando os dados, e responde adequadamente. Seu fluxograma é mostrado na figura 5.4.

5.2.1.6 Transferência

A rotina de transferência, executada sempre que se preenche a FIFO, simplesmente verifica se a última transferência foi concluída e indica para o barramento que um novo pacote de dados está pronto, como mostrado na figura 5.5

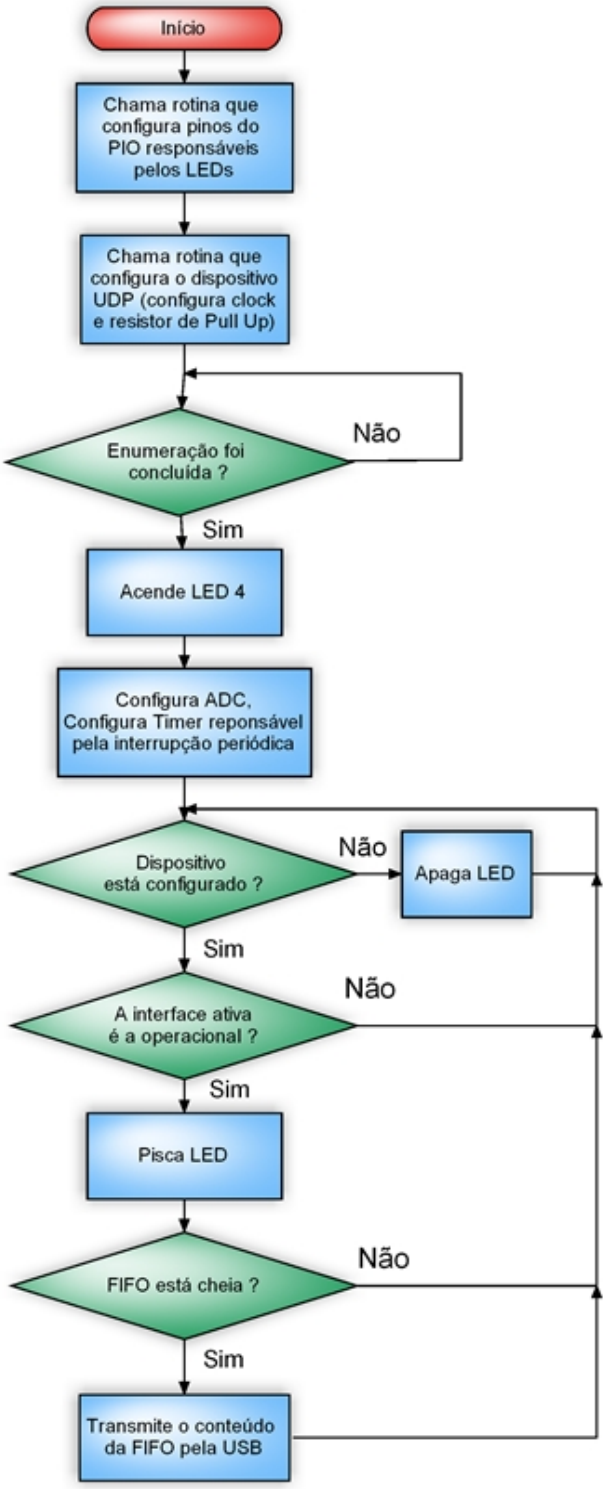

Figura 5.2: Fluxograma do Módulo Principal

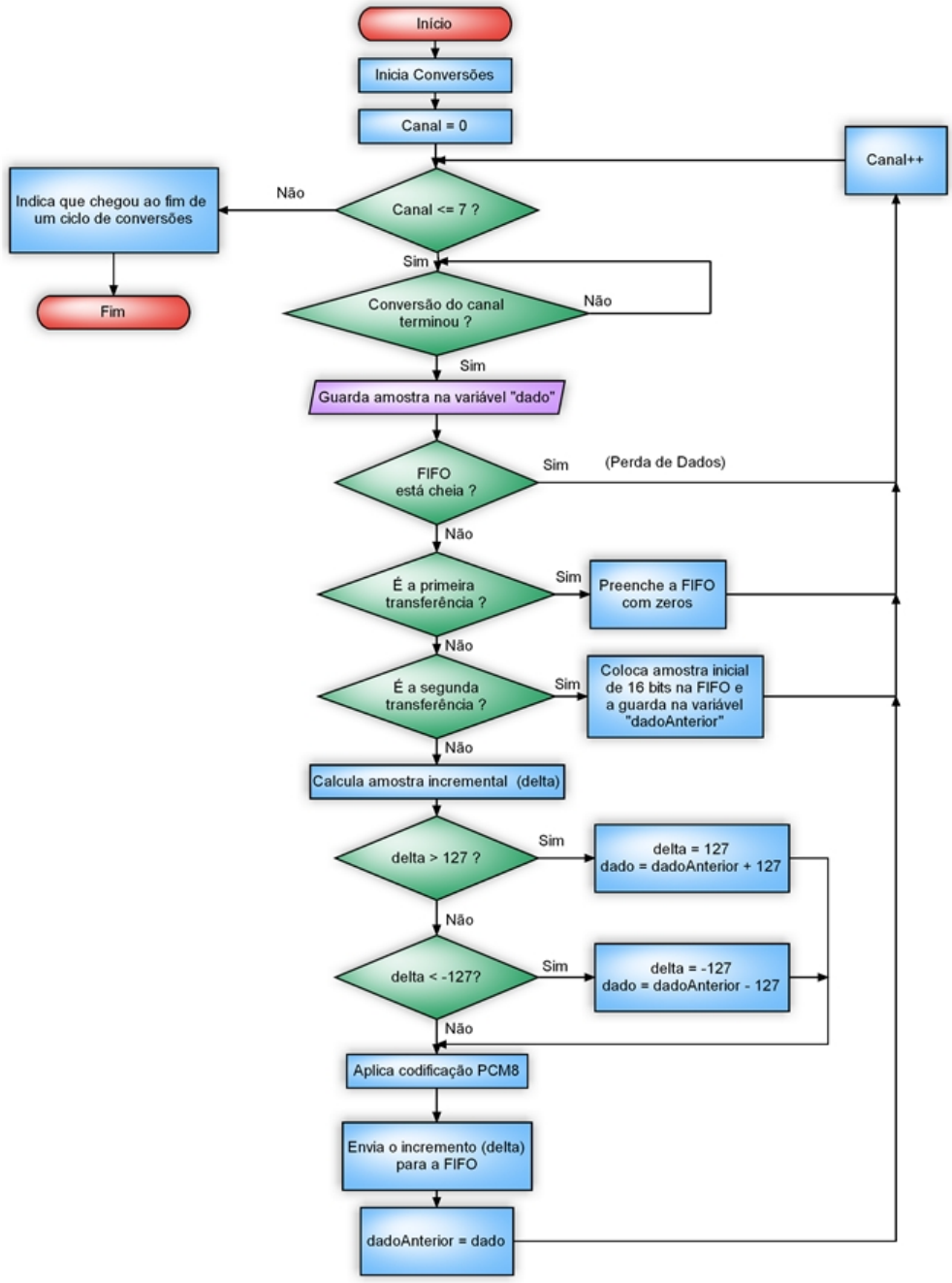

Figura 5.3: Fluxograma da rotina de conversão

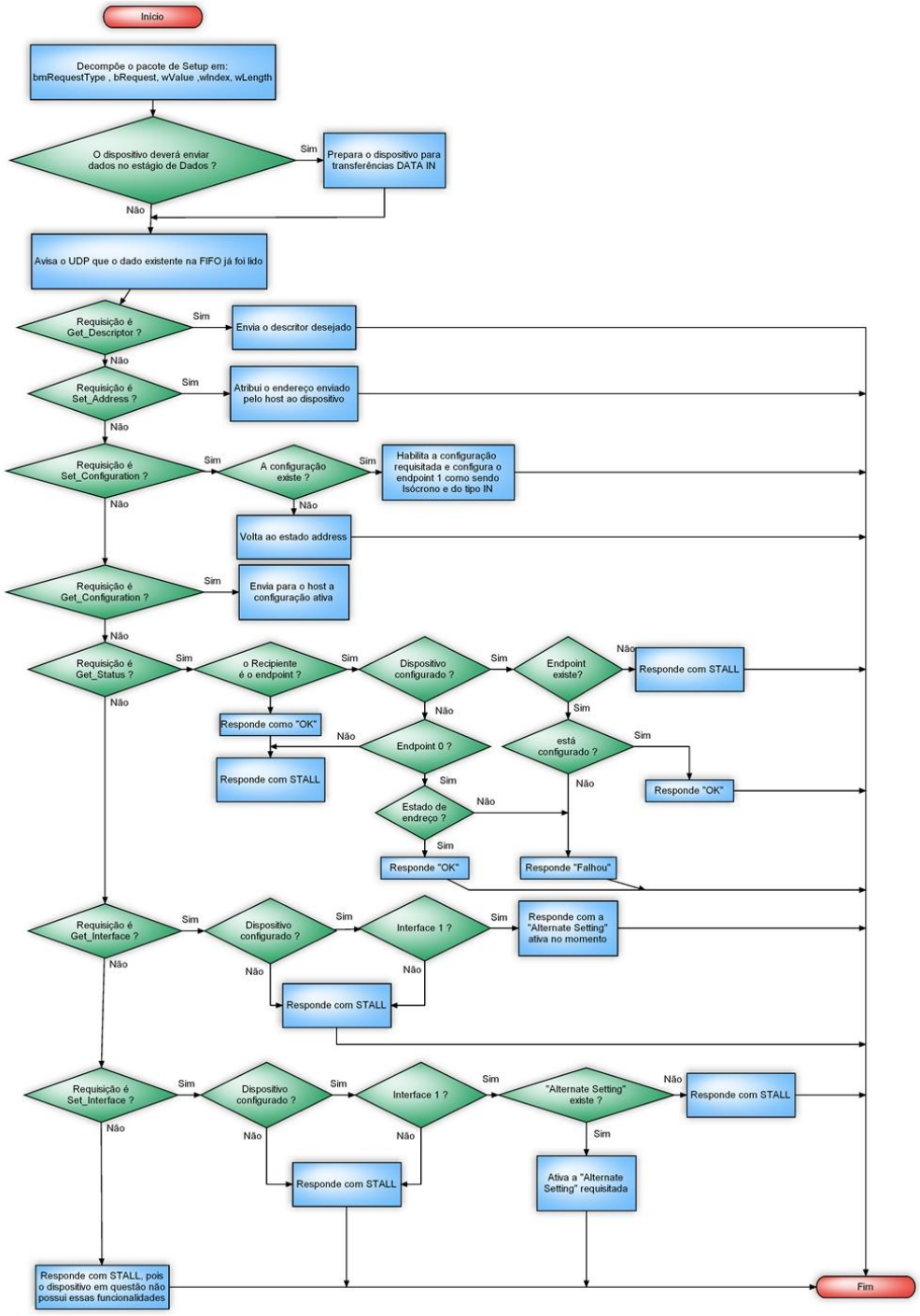

Figura 5.4: Fluxograma da rotina de enumeração

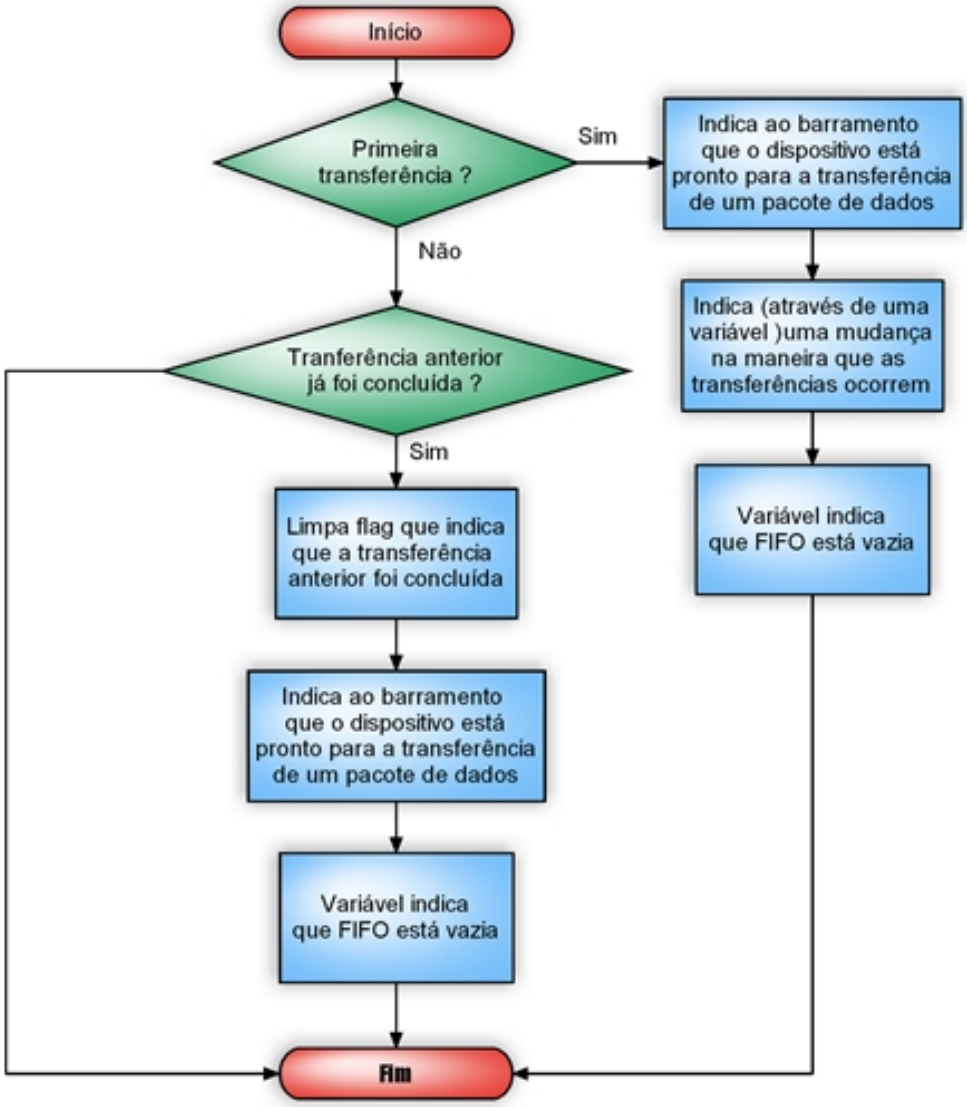

Figura 5.5: Fluxograma da rotina de transferência

Capítulo 6

Resultados

Neste capítulo estão descritos os resultados obtidos com os testes

Com o firmware definitivo, passou-se à fase de testes. O primeiro teste realizado foi para verificar se dessa maneira conseguia-se transmitir dados gerados na mesma velocidade que seriam caso se estivesse transmitindo os oito canais. Para isso, a cada conversão realizada pelo ADC, um contador era incrementado, ao invés de se utilizar o resultado da conversão. Desta forma tinha-se o controle sobre o que se estava enviando e os dados eram gerados independentemente da transmissão. Para se verificar se a velocidade estava suficiente, bastava conferir se nenhum dado estava faltando. Já que a transferência era realizada como se fosse um sinal de áudio PCM8, era necessário a decodificação dos dados recebidos pelo computador. Este teste teve um resultado positivo, pois todos os dados foram recebidos e, após a demultiplexação dos oito "canais", verificava-se que o complemento enviado era de exatamente oito, como esperado. A seguir, passamos a testar com sinais reais nas entradas do ADC. O primeiro teste visava avaliar a reconstrução do sinal com a modulação delta. Utilizando um gerador de funções, aplicamos um sinal senoidal a um único canal do ADC, ajustamos sua freqüência para 500 Hz, sua amplitude para 550 mV e um offset de 850 mV, pois o ADC não trabalha com tensões negativas. O sinal foi reconstruído com sucesso. A forma de onda obtida está na figura 6.1.

Para teste das capacidades do ADC, manteve-se apenas um sinal mas variamos seu offset e sua amplitude. Neste ponto encontramos um problema. A placa utilizada tem como valor de referência para o ADC (ADCVREF) 3,6 V. De acordo com o datasheet do AT91SAM7S256, a conversão seria feita corretamente para valores de tensão entre 0 e ADCVREF. Entretanto, sempre que o valor de pico do sinal passava de 1,5 V os valores obtidos na conversão se comportavam de forma estranha. A forma de onda reconstruída nessa situação foi a mostrada na figura 6.2. Já a de um sinal com o pico ligeiramente abaixo de 1,5 V está na figura 6.3.

Desconfiou-se que o problema pudesse estar na rotina utilizada para a modulação delta. Como nestes testes apenas um canal estava sendo usado, a velocidade era suficiente para se transmitir as amostras completas de 10 bits, sem a necessidade da modulação delta. Este procedimento foi realizado com um sinal que ultrapassava os 1,5 V. Também neste caso o sinal reconstruído teve um comportamento parecido. Tem-se então um problema, pois a placa de amplificação e filtragem do sinal proveniente dos microfones foi projetada para fornecer sinais com um valor DC de 2,5 V. Deve-se alterar o valor dos componentes utilizados para gerar este nível DC para que o pico do sinal fique abaixo de 1,5 V.

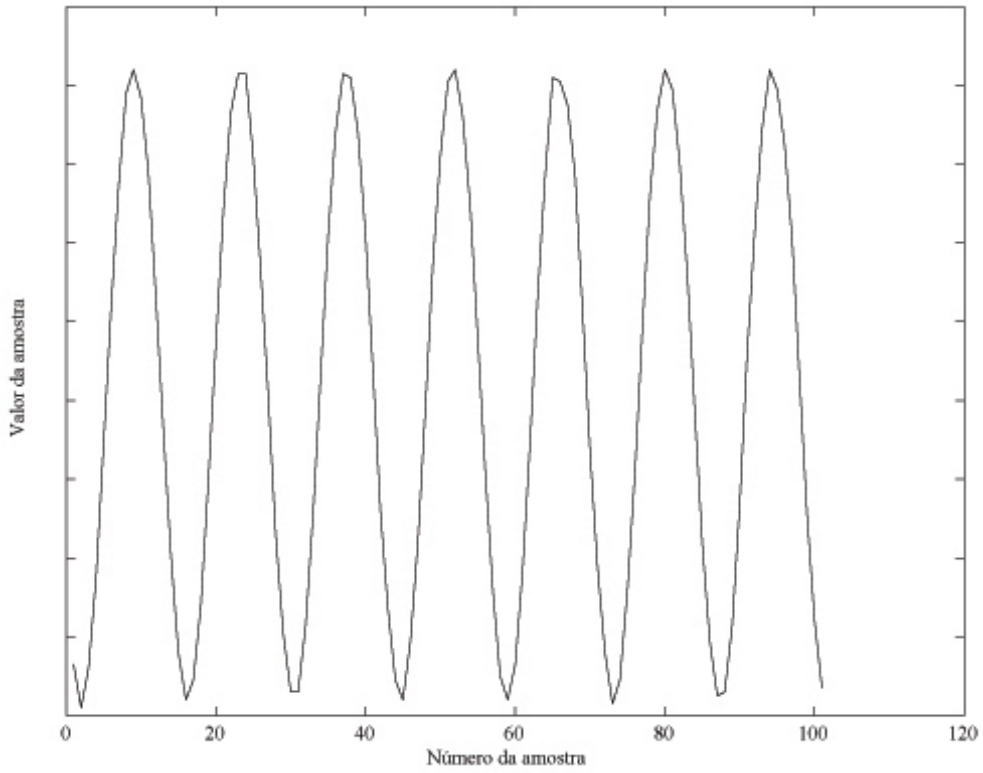

Figura 6.1: Reconstrução bem sucedida do sinal senoidal

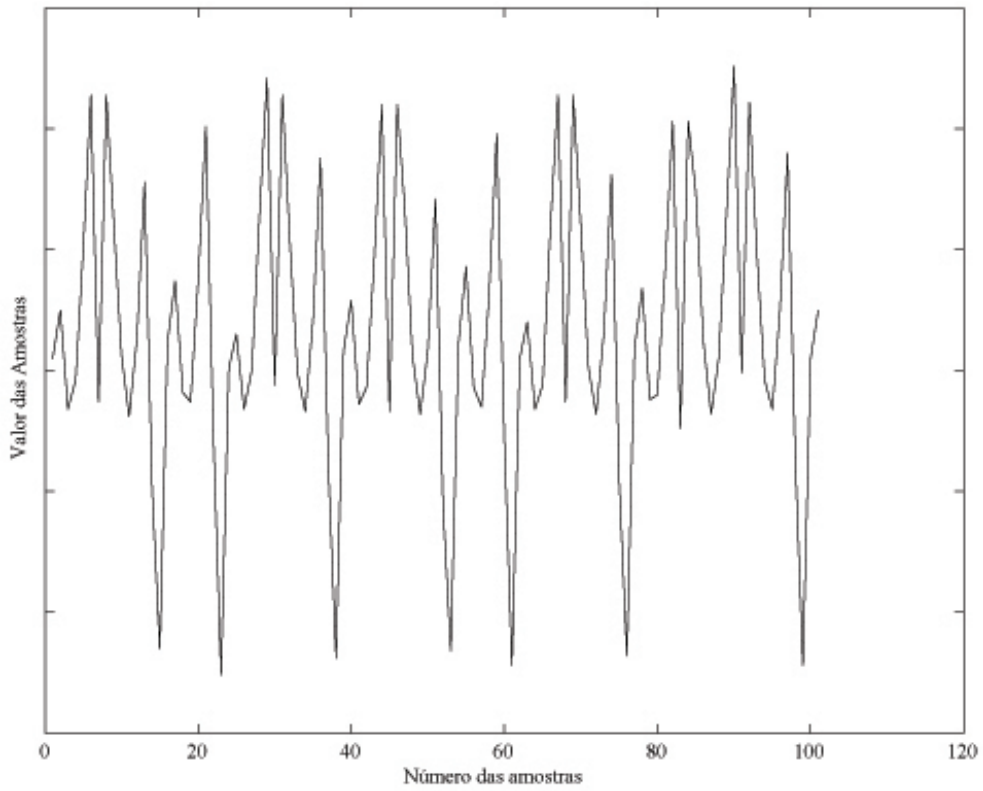

Figura 6.2: Reconstrução de sinal com pico acima de 1,5 V

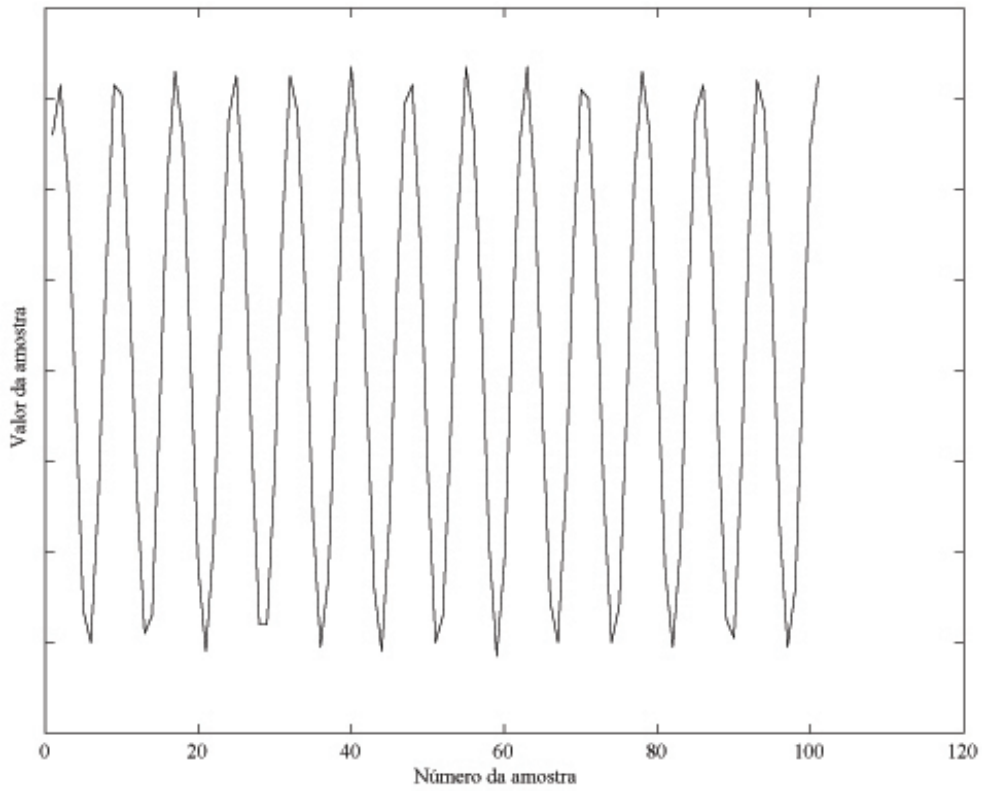

Figura 6.3: Reconstrução de sinal com pico de 1,4 V

O último teste realizado foi de aquisição e transferência de todos os oito canais do ADC. Como a placa dos microfones ainda não estava pronta até a data de escrita deste documento, utilizou-se novamente o gerador de funções ajustado com um sinal senoidal. Este mesmo sinal, com os ajustes nos quais a transferência e reconstrução de um único canal foi realizada com sucesso, foi aplicado simultaneamente aos oito canais. Os oito canais foram reconstruídos de forma correta, obtendo-se sucesso no objetivo de se transmitir toda a informação gerada. As oito formas de onda reconstruídas são apresentadas na figura 6.4.

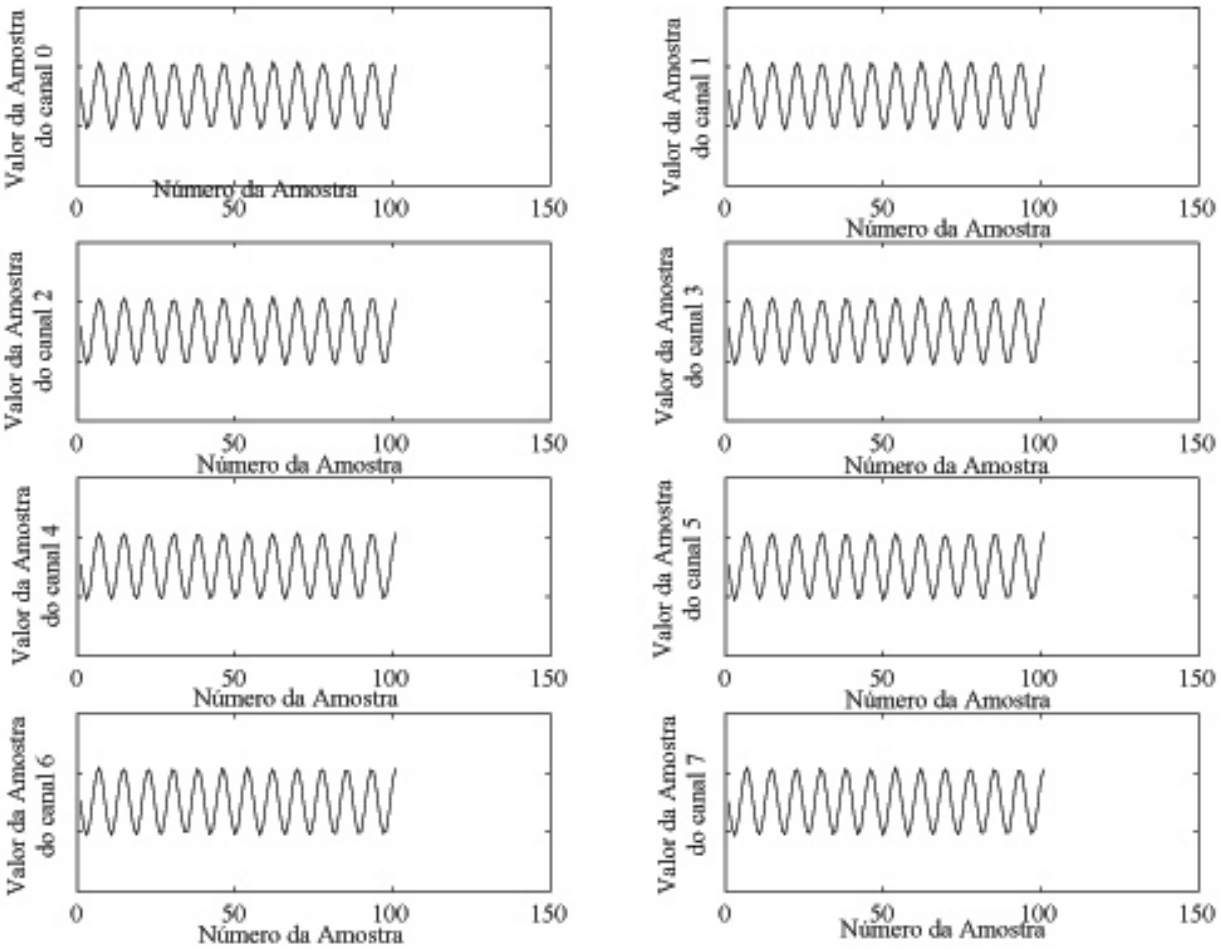

Figura 6.4: Reconstrução dos oito canais simultaneamente

Capítulo 7

Conclusão

Este trabalho é a continuação de outros dois trabalhos realizados por colegas dos autores. Seu objetivo inicial era finalizar um protótipo que fosse capaz de adquirir oito canais de voz através de um arranjo de microfones e transmití-los para um PC através da porta USB, de forma que estes sinais pudessem ser utilizados para testes de algoritmos de identificação de direção de chegada.

Logo no início descobriu-se a dificuldade de entendimento completo de dois trabalhos desenvolvidos durante seis meses por outras quatro pessoas, os quais não acompanhávamos, em um período tão curto. Para possibilitar tal entendimento decidimos trabalhar por etapas.

A primeira etapa consistiu em determinar e solucionar os problemas do *hardware* analógico que fornecia os sinais ao microcontrolador. Estas falhas se mostraram muito mais graves do que o imaginado, existindo inclusive problemas intermitentes. Após identificar algumas falhas graves, tentou-se solucioná-las modificando a própria placa. Infelizmente isso se mostrou ineficaz, quando foi decidido fazer uma nova placa com o mesmo propósito e sanando os problemas. Como o processo de confecção é demorado, passou-se então para a segunda etapa.

A segunda etapa consistiu em identificar a causa e solucionar as falhas do *firmware* desenvolvido para o microcontrolador. Esse não estava transmitindo as informações na velocidade suficiente. De acordo com sua documentação oficial ele tinha tal capacidade. Então começamos a analisar suas funções para aquisição de dados e controle do módulo USB tornando-as o mais eficiente possível. Mesmo assim a velocidade de transmissão continuou aquém do esperado.

Partiu-se então para uma abordagem diferente. Agora o foco seria mudar a maneira como o PC se comunicaria com a placa, através dos descritores USB. Várias tentativas foram efetuadas, tais como a mudança do tipo de codificação, da freqüência de amostragem, do número de interfaces utilizadas e do número de endpoints utilizados. No dois últimos itens citados, o PC falhou em reconhecer adequadamente o dispositivo. Isso foi atribuído ao fato dos *drivers* do Windows responsáveis pelos dispositivos de áudio USB não preverem tal situação.

Por último foi testado o tempo gasto na transmissão de cada pacote. Descobriu-se que esse tempo era grande demais para um pacote de apenas 64 bytes. Surgiu então a desconfiança de que o *hardware* (AT91SAM7S256) não suportava a velocidade necessária. Passou-se a procurar outras soluções para conseguir transmitir todos os dados na velocidade disponível. A solução proposta foi utilizar um sistema

simples de modulação delta, nesse caso cada amostra seria transmitida por apenas 8 bits (com exceção da primeira que teria 16 bits) com perda mínima de informação.

Como até então a confecção da placa não havia sido concluída, passou-se para a etapa dos testes conseguindo com êxito a transmissão de oito canais analógicos simultaneamente. Nessa etapa outro problema foi encontrado, o ADC da placa mostrou-se limitado, não funcionando conforme sua especificação. Com o tempo já escasso não se pôde investigar a fundo tal fato.

No fim, mesmo com tantos problemas, conseguiu-se a principal tarefa a que se propôs: transmitir todos os 8 canais corretamente pelo barramento USB. No entanto, com algumas limitações. Está se utilizando a velocidade máxima disponível, o que faz com que a freqüência de amostragem não possa ser maior que 8 kHz e conseqüentemente os sinais de voz estarão limitados a 4 kHz. Essa freqüência é mais do que suficiente para canais de comunicação, mas ainda se deve averiguar se não haverá algum problema para a estimação da direção de chegada. Neste ponto já é possível se fazer tal averiguação, pois já se consegue entregar a informação para o PC.

7.1 Propostas para Continuação

No ponto em que se encontra o projeto, o próximo passo será a montagem da placa de filtragem para aquisição dos canais de áudio. É comum que tais placas apresentem problemas que precisam ser descobertos e solucionados. Fazem-se necessários vários testes em pontos diferentes do *hardware*.

Possuindo uma placa que forneça os sinais analógicos filtrados e em níveis adequados, o comportamento anômalo encontrado na conversão feita pelo AD integrado no microprocessador deve ser investigado. Após a solução do mesmo, o projeto passará para a fase mais importante que são os testes, implementações e otimizações de algoritmos de direção de chegada, que é o que tornará a prótese auditiva realmente inteligente.

Referências Bibliográficas

[1] AXELSON, J. *USB Complete: Everything You Need to Develop USB Peripherals*, Third Edition. [S.l.]: Lakeview Reasearch LLC, 2005.

[2] HYDE, J. *USB Design by Example: A Practical Guide to Building I/O Devices*, Second Edition. United States of America: Intel Press, 2001.

[3] SITE do USB - ImplementerŠs Forum. http://usb.org.

[4] DOCUMENTAÇÃO das classes USB em versão final. http://www.usb.org/developers/devclass_docs.

[5] ATMEL. *AT91SAM7S Series Preliminary*. [S.l.], Novembro 2006. Disponível em http://www.atmel.com/products/A

[6] SASAKI, MARCELLO GURGEL; CAIXETA, OTÁVIO VIEGAS (2006). *Desenvolvimento de uma interface USB para aquisição de dados de um arranjo de microfones: aplicação em Prótese Auditiva.* Monografia de Graduação, Publicação ENE 02/2006, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 134p.

[7] COSTA, ANA RAVENA ALCÂNTARA DA e GARCIA, FRANCISCO AUGUSTO DA COSTA (2006). *Desenvolvimento do Hardware para Obtenção de DOA por Meio de Arranjo de Sensores: Aplicação em Prótese Auditiva.* Monografia de Graduação, Publicação ENE 02/2006, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 88p.

ANEXOS

I. CÓDIGOS FONTE DO FIRMWARE

Listing I.1: Programa Principal

```
//*----------------------------------------------------------------------------
//*         Projeto PAI - Módulo Principal
//*----------------------------------------------------------------------------
//* File Name           : main.c
//* Object              : Aplicação Principal
//* Creation            : 05/2006
//*----------------------------------------------------------------------------

// Include Standard files
#include "Board.h"
#include "mapa.h"
#include "functions.h"
#include "globalVar.h"

void Inicio(void)
{
    // Habilita o Clock do PIO
    AT91C_BASE_PMC->PMC_PCER = (1 << AT91C_ID_PIOA);

    // Configura os pinos do PIO correspondentes aos LEDs de 1 a 4 para serem saídas
    AT91F_PIO_CfgOutput( AT91C_BASE_PIOA, LED_MASK ) ;

    // Apaga os LEDs
    AT91F_PIO_SetOutput( AT91C_BASE_PIOA, LED_MASK ) ;

    // Configura o PIO11 como saída
    AT91F_PIO_CfgOutput(AT91C_BASE_PIOA,TP0);
    AT91F_PIO_SetOutput(AT91C_BASE_PIOA,TP0);
    // Configura o PIO12 como saída
    AT91F_PIO_CfgOutput(AT91C_BASE_PIOA,TP1);
    AT91F_PIO_SetOutput(AT91C_BASE_PIOA,TP1);

}

//*----------------------------------------------------------------------------
//* Function Name       : Main
//* Object              : Software entry point
//* Input Parameters    : none.
//* Output Parameters   : none.
//*----------------------------------------------------------------------------
int main(void)
{// Begin

    Inicio();               // Configura PIO

    USB_Open();             // Configura UDP (USB)

    // Espera o fim da Enumeração
    while (!Audio.IsConfigured(&Audio));

    // Led4 is switched on
    AT91C_BASE_PIOA->PIO_CODR = LED4;

    ADC_Init();             // Configura ADC

    Tim0Init();             // Configura Timer-Counter 0

   // Loop infinito
    for(;;)
    {
        /* Transmissão USB */
        if (Audio.IsConfigured(&Audio)) {

            if(Audio.currentSetting) {

                // Pisca LED
                if(ledCount4 == 25000)
                    AT91C_BASE_PIOA->PIO_CODR = LED4;
                if(ledCount4++ == 50000) {
                    AT91C_BASE_PIOA->PIO_SODR = LED4;
                    ledCount4 = 0;
                }
```

```
            if (contador==64) {

                Audio.Stream(&Audio);
              }

            }

     }
        else
            AT91C_BASE_PIOA->PIO_SODR = LED4;
    }

}// End
```

Listing I.2: Módulo Temporizador

```
//*-----------------------------------------------------------------------------
//*          Projeto PAI - Módulo do Temporizador
//*-----------------------------------------------------------------------------
//* File Name           : timerModulo.c
//* Object              : Interrupção de tempo
//* Creation            : 05/2006
//*-----------------------------------------------------------------------------

#include "mapa.h"
#include "functions.h"
#include "externVar.h"

//*---------------------------- Interrupt Function ------------------------------

//*-----------------------------------------------------------------------------
//* Function Name       : timer0_irq_handler
//* Object              : C handler interrupt function called by the interrupts
//*                       assembling routine
//* Output Parameters   : start a new round of ADC conversions
//*-----------------------------------------------------------------------------
void timer0_irq_handler(void)
{
    unsigned int dummy;
    //Limpa o registrador de status do timer
    dummy = AT91C_BASE_TC0->TC_SR;
    //Suprime: warning variable "dummy" was set but never used
    dummy = dummy;

    //Pisca LED
    if(ledCount1 == 1000)
        AT91C_BASE_PIOA->PIO_CODR = LED1;
    if(ledCount1++ == 2000) {
        AT91C_BASE_PIOA->PIO_SODR = LED1;
        ledCount1 = 0;
    }
    //Verifica se o A/D não está no meio de uma conversão
    if(conversionOver) {
        conversionOver = 0;
        ADC_Convert();
    }
}

void Tim0Init(void)
{
  unsigned int dummy = 0;
  // Habilita o CLOCK do timer0
  AT91C_BASE_PMC->PMC_PCER = 1<<AT91C_ID_TC0;
  // Desabilita o clock e as suas interrupcoes
  AT91C_BASE_TC0->TC_CCR = AT91C_TC_CLKDIS ;
  AT91C_BASE_TC0->TC_IDR = 0xFFFFFFFF ;
  // Limpa o registrador de status do timer
  dummy = AT91C_BASE_TC0->TC_SR;
  // Supprime: warning variable "dummy" was set but never used
  dummy = dummy;
  // Seleciona o tamanho do contador interno (RC)
  AT91C_BASE_TC0->TC_RC = (int) RC_SIZE;
  // Seleciona o modo de contagem (MCK/8, UP, clear quando = RC, WAVE)
  AT91C_BASE_TC0->TC_CMR = AT91C_TC_CLKS_TIMER_DIV2_CLOCK | AT91C_TC_CPCTRG | AT91C_TC_WAVE ;
  // Habilita TIM0
  AT91C_BASE_TC0->TC_CCR = AT91C_TC_CLKEN | AT91C_TC_SWTRG;

//---------------------------- Configura e habilita a interrupção ---------------------------------

  // Desabilita a interrupcao no controlador de interrupção
```

```
  AT91C_BASE_AIC->AIC_IDCR |= 1 << AT91C_ID_TC0 ;
  // Seta o endereço no vetor de interrupções
  AT91C_BASE_AIC->AIC_SVR[AT91C_ID_TC0] = (int) timer0_irq_handler ;
  // Configura o modo (sensibilidade e prioridade)
  AT91C_BASE_AIC->AIC_SMR[AT91C_ID_TC0] |= AT91C_AIC_SRCTYPE_INT_POSITIVE_EDGE | TIM0_INT_PRIOR;
  // Limpa a flag de interrupção
  AT91C_BASE_AIC->AIC_ICCR |= 1 << AT91C_ID_TC0 ;
  // Habilita a interrupção do timer 0 no AIC
  AT91C_BASE_AIC->AIC_IECR |= 1 << AT91C_ID_TC0;
  // Interrupção de trigger interno (estouro de RC) habilitada
  AT91C_BASE_TC0->TC_IER |= AT91C_TC_CPCS;
}
```

Listing I.3: Módulo ADC

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI - Módulo ADC
//*-----------------------------------------------------------------------------
//* File Name           : adcModulo.c
//* Object              : Módulo do conversor A/D
//* Creation            : 05/2007
//*-----------------------------------------------------------------------------

#include "mapa.h"
#include "functions.h"
#include "externVar.h"

void ADC_Init(void)
{
    AT91C_BASE_ADC->ADC_MR = (  (AT91C_ADC_TRGEN & 0) |             // Desabilita trigger de hardware
                                (AT91C_ADC_TRGSEL & (0<<1)) |       // Seleção de trigger
                                (AT91C_ADC_LOWRES & (0<<4)) |       // Resolução de 10-bit
                                (AT91C_ADC_SLEEP & (0<<5)) |        // Modo normal
                                (AT91C_ADC_PRESCAL & (5<<8)) |      // ADC prescaler = 5(ADCClock = 4MHz)
                                (AT91C_ADC_STARTUP & (3<<16)) |     // Tempo de inicialização do ADC = 8 (18 us)
                                (AT91C_ADC_SHTIM & (8<<24)));       // Tempo de inicialização do SH = 3(1 us)

    // Habilita todos os canais
    AT91C_BASE_ADC->ADC_CHER = 0xFF;

    //Isso faz os pinos do AD virarem entrada
    AT91F_PIO_CfgPeriph (AT91C_BASE_PIOA, 0 , 0);
}

void ADC_Convert(void)
{
    // Inicia conversão
    AT91C_BASE_ADC->ADC_CR = AT91C_ADC_START;

    for(adcChannel = 0; adcChannel < 8; adcChannel++) {
        // Espera fim da conversao
        while(!(AT91C_BASE_ADC->ADC_SR & AT91C_ADC_DRDY));

        // Lê amostra
        dado = (signed short) (AT91C_BASE_ADC->ADC_LCDR & AT91C_ADC_DATA);

        //Verifica se pode escrever na FIFO
        if(contador<64){

          //Verifica se é a primeira transferência
          if(firstTransfer){
            //Preenche a FIFO com 0
             AT91C_BASE_UDP->UDP_FDR[AUDIO_IN] = 0;
             contador++;
          }

          //Verifica se deve enviar as amostras iniciais
          if(secondTransfer){

              //Envia primeira amostra completa para a FIFO
            dado = (unsigned short) dado;
               AT91C_BASE_UDP->UDP_FDR[AUDIO_IN] = (char) dado;
               AT91C_BASE_UDP->UDP_FDR[AUDIO_IN] = (char) (dado >> 8);
               dadoAnterior[adcChannel] = dado;
               contador+=2;

               //Verifica se já enviou os 8 canais.
               if(adcChannel==7)
                 //Sinaliza que as próximas amostras serão de 8 bits (incrementos)
                 secondTransfer=0;
          }
          else if(!(firstTransfer)){
```

```
            //Calcula o incremento
            delta = dado - dadoAnterior[adcChannel];

            //Trunca delta ao valor máximo que o incremento pode ter
            if(delta>127){
              delta = 127;
              //Atualiza o valor do dado para cálculo correto do próximo delta
              dado = dadoAnterior[adcChannel]+127;
            }
            //Trunca delta ao valor mínimo que o incremento pode ter
            if(delta<-127){
              delta = -127;
              //Atualiza o valor do dado para cálculo correto do próximo delta
              dado = dadoAnterior[adcChannel]-127;
            }
            //Faz com que os dados transmitidos sejams sempre positivos
            //(Codificação PCM8)
            if(delta>=0)
              delta = delta+128;
            else{
              delta = delta + 127;
            }
            //Envia incremento para a FIFO
            AT91C_BASE_UDP->UDP_FDR[AUDIO_IN] = (char) delta;
            dadoAnterior[adcChannel] = dado;
            //Incrementa posição da FIFO
            contador++;
          }

        }

    }
    //Indica que o A/D não está mais no meio de uma conversão
    conversionOver = 1;
}
```

Listing I.4: Módulo USB

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI - Módulo USB
//*-----------------------------------------------------------------------------
//* File Name           : usbModulo.c
//* Object              : Controle do Dispositivo USB
//* Creation            : 05/2006
//*-----------------------------------------------------------------------------

#include "mapa.h"
#include "functions.h"
#include "externVar.h"

/* USB Descriptors */
const char DeviceDescriptor[] = {
  0x12,                                 //bLength
  0x01,                                 //bDescriptorType (DEVICE)
  0x00,0x02,                            //bcdUSB (USB 2.0)
  0x00,                                 //bDeviceClass
  0x00,                                 //bDeviceSubClass
  0x00,                                 //bDeviceProtocol
  EP0_SIZE,                             //bMaxPacketSize0 (Endpoint0 Max Size)
  (char)VENDOR_ID,(VENDOR_ID>>8),       //idVendor
  (char)PRODUCT_ID,(PRODUCT_ID>>8),     //idProduct
  (char)VERSION,(VERSION>>8),           //bcdDevice
  0x00,                                 //iManufacturer String
  0x00,                                 //iProduct String
  0x00,                                 //iSerialNumber String
  0x01,                                 //bNumConfigurations
};

const char ConfigurationDescriptor[] = {
  0x09,                                 //bLength
  0x02,                                 //bDescriptorType (CONFIGURATION)
  0X64,0x00,                            //bTotalLength
  0x02,                                 //bNumInterfaces (AudioControl + AudioStreaming)
  0x01,                                 //bConfigurationValue
  0x00,                                 //iConfiguration String
  0x80,                                 //bmAttributes (Bus Powered, not Self Powered, no Remote wakeup capability)
  0x32,                                 //MaxPower (x2 = 100mA)

/* AudioControl Interface */
  0x09,                                 //bLength
  0x04,                                 //bDescriptorType (INTERFACE)
  0x00,                                 //bInterfaceNumber
```

```
  0x00,                           //bAlternateSetting
  0x00,                           //bNumEndpoints (Uses EP0)
  0x01,                           //bInterfaceClass (AUDIO)
  0x01,                           //bInterfaceSubClass (AUDIO_CONTROL)
  0x00,                           //bInterfaceProtocol
  0x00,                           //iInterface String

/* AudioControl Class Interface */
  0x09,                           //bLength
  0x24,                           //bDescriptorType (CS_INTERFACE)
  0x01,                           //bDescriptorSubtype (HEADER)
  0x00, 0x01,                     //bcdADC (Audio Class spec release ver 1.0)
  0x1E, 0x00,                     //wTotalLength (Class Length)
  0x01,                           //bInCollection (1 AudioStreaming Interface)
  0x01,                           //baInterfaceNr(1) (AudioStreaming interface 1 belongs to this AudioControl interface)

/* Input Terminal Descriptor */
  0x0C,                           //bLength
  0x24,                           //bDescriptorType (CS_INTERFACE)
  0x02,                           //bDescriptorSubtype (INPUT_TERMINAL)
  0x01,                           //bTerminalID
  0x05, 0x02,                     //wTerminaltype (Microphone)
  0x00,                           //bAssocTerminal (No terminal associations)
  USB_CHANNELS,                   //bNrChannels (# of logical output channels)
  0x00, 0x00,                     //wChannelConfig (No predefined channel positions configuration)
  0x00,                           //iChannelNames String
  0x00,                           //iTerminal String

/* Output Terminal Descriptor */
  0x09,                           //bLength
  0x24,                           //bDescriptorType (CS_INTERFACE)
  0x03,                           //bDescriptorSubtype (OUTPUT_TERMINAL)
  0x02,                           //bTerminalID
  0x01, 0x01,                     //wTerminalType (USB Streaming)
  0x00,                           //bAssocTerminal (No associations)
  0x01,                           //bSourceID (From input terminal)
  0x00,                           //iTerminal String
  //Audio Class wTotalLength vem de AudioControl Class Interface até aqui

 /* AudioStreaming Interface Zero-bandwidth Alt0 */
  0x09,                           //bLength
  0x04,                           //bDescriptorType (INTERFACE)
  0x01,                           //bInterfaceNumber
  0x00,                           //bAlternateSetting (0)
  0x00,                           //bNumEndpoints (Zero-bandwidth alternate setting)
  0x01,                           //bInterfaceClass (AUDIO)
  0x02,                           //bInterfaceSubclass (AUDIO_STREAMING)
  0x00,                           //bInterfaceProtocol
  0x00,                           //iInterface String

 /* AudioStreaming Interface Operational Alt1 */
  0x09,                           //bLength
  0x04,                           //bDescriptorType (INTERFACE)
  0x01,                           //bInterfaceNumber
  0x01,                           //bAlternateSetting (1)
  0x01,                           //bNumEndpoints (Uses 1 isochoronous endpoint)
  0x01,                           //bInterfaceClass (AUDIO)
  0x02,                           //bInterfaceSubclass (AUDIO_STREAMING)
  0x00,                           //bInterfaceProtocol
  0x00,                           //iInterface String

/* AudioStreaming Class-Specific Interface */
  0x07,                           //bLength
  0x24,                           //bDescriptorType (CS_INTERFACE)
  0x01,                           //bDescriptorSubtype (GENERAL)
  0x02,                           //bTerminalLink (Connected to output terminal)
  0x01,                           //bDelay
  0x02, 0x00,                     //wFormatTag (Audio Data Format = PCM8)

/* Format Type Descriptor */
  0x0B,                           //bLength
  0x24,                           //bDescriptorType (CS_INTERFACE)
  0x02,                           //bDescriptorSubtype (FORMAT_TYPE)
  0x01,                           //bFormatType (FORMAT_TYPE_I)
  USB_CHANNELS,                   //bNrChannels (# of physical channels)
  0x01,                           //bSubFrameSize (2 bytes in audio subframe)
  0x08,                           //bBitResolution (# of bits per sample)
  0x01,                           //bSamFreqType (# of frequencies supported)
  (char)SAM_FREQ,
  (char)(SAM_FREQ>>8),
  (SAM_FREQ>>16),                 //tSamFreq[1]

/* Endpoint Descriptor */
```

```
  0x09,                                 //bLength
  0x05,                                 //bDescriptorType (ENDPOINT)
  0x81,                                 //bEndpointAddress (IN 1) (10000001b)
  0x01,                                 //bmAttributes (Asynchronous Isochronous, not shared) (00000001b)
  0x40,0x00,                            //wMaxPacketSize (# bytes per packet)
  0x01,                                 //bInterval
  0x00,                                 //bRefresh
  0x00,                                 //bSynchAddress (Endpoint for synchronization info: unused)

/* Class Endpoint Descriptor */
  0x07,                                 //bLength
  0x25,                                 //bDescriptorType (CS_ENDPOINT)
  0x01,                                 //bDescriptorSubtype (GENERAL)
  0x00,                                 //bmAttributes (none)
  0x00,                                 //bLockDelayUnits (unused)
  0x00,0x00                             //wLockDelay (unused)
  //Configuration bTotalLength vem de Configuration descriptor até aqui
};

const short StringDescriptor[] = {
    0x0304,                             //bDescriptorType + bLength
    0x0409,                             //Only English strings

/* Manufacturer */
    0x0308,                             //bDescriptorType + bLength
    'U','n','B',

/* Product */
    0x0316,                             //bDescriptorType + bLength
    'M','i','c','r','o','p','h','o','n','e'
};


void USB_Open(void)
{

   // Configura o divisor PLL da USB Set the PLL USB Divider
    AT91C_BASE_CKGR->CKGR_PLLR |= AT91C_CKGR_USBDIV_1 ;
    //Habilita o clock UDPCK de 48MHz e o clock "System Peripheral USB Clock"
    AT91C_BASE_PMC->PMC_SCER |= AT91C_PMC_UDP;
    AT91C_BASE_PMC->PMC_PCER |= (1 << AT91C_ID_UDP);
    //Habilita os resistores de PullUp (USB_DP_PUP)
    AT91F_PIO_CfgOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);
    AT91F_PIO_ClearOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);

    // Inicializa a estrutura Audio
    USB_AUDIO_Open(&Audio, AT91C_BASE_UDP);


}

AT91PS_AUDIO USB_AUDIO_Open(AT91PS_AUDIO pAudio, AT91PS_UDP pUdp)
{
    pAudio->pUdp = pUdp;
    pAudio->currentConfiguration = 0;
    pAudio->currentSetting = 0;
    pAudio->IsConfigured = AT91F_UDP_IsConfigured;
    pAudio->Stream = AT91F_UDP_Stream;
    return pAudio;
}

static unsigned char AT91F_UDP_IsConfigured(AT91PS_AUDIO pAudio)
{
    AT91PS_UDP pUDP = pAudio->pUdp;
    AT91_REG isr = pUDP->UDP_ISR;
    //Verifica no registrador de interrupções se
    //o barramento foi reiniciado
    if (isr & AT91C_UDP_ENDBUSRES) {
        pUDP->UDP_ICR = AT91C_UDP_ENDBUSRES;
        // Reseta todos endpoints
        pUDP->UDP_RSTEP  = (unsigned int) -1;
        pUDP->UDP_RSTEP  = 0;
        // Habilita a função
        pUDP->UDP_FADDR = AT91C_UDP_FEN;
        // Configura endpoint 0
        pUDP->UDP_CSR[0] = (AT91C_UDP_EPEDS | AT91C_UDP_EPTYPE_CTRL);

    }
    //Verifica no registrador de interrupções se
    //a interrupção foi do enpoint 0
    else if (isr & AT91C_UDP_EPINT0) {
        //chama a função que irá enumerar o dispositivo
        AT91F_USB_Enumerate(pAudio);
```

```
    }


    return pAudio->currentConfiguration;
}

static void AT91F_UDP_Stream(AT91PS_AUDIO pAudio)
{
        AT91PS_UDP pUdp = pAudio->pUdp;

        //Caso seja a primeira transfereência, simplesmente notifica dado pronto
        if(firstTransfer){
          pUdp->UDP_CSR[AUDIO_IN] |= AT91C_UDP_TXPKTRDY;
          //marca a aquisição do dado inicial
          secondTransfer=1;
          //marca a mudança da forma de transferencia
          firstTransfer=0;
          //indica que pode escrever na FIFO
          contador=0;
        }

        //Verifica se a última transferencia foi comcluída
        else if((pUdp->UDP_CSR[AUDIO_IN] & AT91C_UDP_TXCOMP)){

                //Limpa o flag de transferência concluída
                pUdp->UDP_CSR[AUDIO_IN] &= ~(AT91C_UDP_TXCOMP);
                //Notifica o dado pronto
                pUdp->UDP_CSR[AUDIO_IN] |= AT91C_UDP_TXPKTRDY;
                //indica que pode escrever na FIFO
                contador = 0;
        }

}



//Envia um dado. Usado durante a enumeração.
static void AT91F_USB_SendData(AT91PS_UDP pUdp, const char *pData, unsigned int length)
{
    unsigned int cpt = 0;
    AT91_REG csr;

    do {

        //Evita estourar a FIFO
        cpt = MIN(length, epSize[0]);
        length -= cpt;

        //Carrega todo o dado a ser enviado na FIFO ou parte dele até que esta esteja cheia
        while (cpt--)
            pUdp->UDP_FDR[0] = *pData++;

        //Verifica se não há transmissões ocorrendo
        if (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) {
            pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
            while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
        }

        //Sinaliza que o dado está pronto para envio
        pUdp->UDP_CSR[0] |= AT91C_UDP_TXPKTRDY;

        do {
            csr = pUdp->UDP_CSR[0];

            // Verifica se a transmissão foi interrompida pelo recebimento de algum pacote
            if (csr & AT91C_UDP_RX_DATA_BK0) {
                pUdp->UDP_CSR[0] &= ~(AT91C_UDP_RX_DATA_BK0);
                return;
            }

            //Espera liberação do barramento
        } while ( !(csr & AT91C_UDP_TXCOMP) );
//Repete até enviar o dado completo
    }while (length);

    //Verifica se o barramento está liberado
    if (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) {
        pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
        while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
    }
}
```

```
void AT91F_USB_SendZlp(AT91PS_UDP pUdp) //Responde com um dado qualquer
//É usado pois toda requisição deve ser respondida, mesmo que não solicite dado.
{
    pUdp->UDP_CSR[0] |= AT91C_UDP_TXPKTRDY;
    while ( !(pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) );
    pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
    while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
}

void AT91F_USB_SendStall(AT91PS_UDP pUdp) //Responde com STALL
{
    pUdp->UDP_CSR[0] |= AT91C_UDP_FORCESTALL;
    while ( !(pUdp->UDP_CSR[0] & AT91C_UDP_ISOERROR) );
    pUdp->UDP_CSR[0] &= ~(AT91C_UDP_FORCESTALL | AT91C_UDP_ISOERROR);
    while (pUdp->UDP_CSR[0] & (AT91C_UDP_FORCESTALL | AT91C_UDP_ISOERROR));
}

static void AT91F_USB_Enumerate(AT91PS_AUDIO pAudio)
// Responsável pela enumeração do dispositivo
// Respondendo a todas requisições
{
    AT91PS_UDP pUDP = pAudio->pUdp;
    unsigned char bmRequestType, bRequest;
    unsigned short wValue, wIndex, wLength, wStatus;



    // Verifica se existe um pacote de Setup na FIFO
    //do endpoint 0
    if ( !(pUDP->UDP_CSR[0] & AT91C_UDP_RXSETUP) ){

        return;
    }


    // Decompõe o pacote de Setup
    bmRequestType = pUDP->UDP_FDR[0];
    bRequest      = pUDP->UDP_FDR[0];
    wValue        = (pUDP->UDP_FDR[0] & 0xFF);
    wValue       |= (pUDP->UDP_FDR[0] << 8);
    wIndex        = (pUDP->UDP_FDR[0] & 0xFF);
    wIndex       |= (pUDP->UDP_FDR[0] << 8);
    wLength       = (pUDP->UDP_FDR[0] & 0xFF);
    wLength      |= (pUDP->UDP_FDR[0] << 8);
    // Verifica a direção da fase de transferência de dados
    if (bmRequestType & 0x80) {
        //Habilita transferências do tipo Data IN
        pUDP->UDP_CSR[0] |= AT91C_UDP_DIR;
        while ( !(pUDP->UDP_CSR[0] & AT91C_UDP_DIR) );
    }
    // Diz para o UDP que já leu o dado da FIFO
    pUDP->UDP_CSR[0] &= ~AT91C_UDP_RXSETUP;
    while ( (pUDP->UDP_CSR[0]  & AT91C_UDP_RXSETUP)  );

    // Verifica o pedido
    switch ((bRequest << 8) | bmRequestType)
    {
    case STD_GET_DESCRIPTOR:
        if (wValue == 0x100)        // Retorna o  Device Descriptor
            AT91F_USB_SendData(pUDP, DeviceDescriptor, MIN(sizeof(DeviceDescriptor), wLength));
        else if (wValue == 0x200)  // Retorna o Configuration Descriptor
            AT91F_USB_SendData(pUDP, ConfigurationDescriptor, MIN(sizeof(ConfigurationDescriptor), wLength));
        else if (wValue == 0x300)  // Retorna o String Descriptor
            AT91F_USB_SendData(pUDP, (char *)StringDescriptor, MIN(sizeof(StringDescriptor), wLength));
        else
            AT91F_USB_SendStall(pUDP); // retorna um STALL
        break;
    case STD_SET_ADDRESS: //Atribui o endereço enviado pelo host ao dispositivo
        AT91F_USB_SendZlp(pUDP);
        pUDP->UDP_FADDR = (AT91C_UDP_FEN | wValue);
        pUDP->UDP_GLBSTATE  = (wValue) ? AT91C_UDP_FADDEN : 0;
        break;
    case STD_SET_CONFIGURATION://Seleciona a cofiguração enviada pelo host e habilita
                               //o endpoint 1 como isócrono do tipo IN
                              //Se não existir configuração, retorna ao estado address
        AT91F_USB_SendZlp(pUDP);
        pAudio->currentConfiguration = wValue;
        pUDP->UDP_GLBSTATE  = (wValue) ? AT91C_UDP_CONFG : AT91C_UDP_FADDEN;
        pUDP->UDP_CSR[AUDIO_IN] = (wValue) ? (AT91C_UDP_EPEDS | AT91C_UDP_EPTYPE_ISO_IN) : 0;
        break;
    case STD_GET_CONFIGURATION: //Envia para o host a configuração ativa
        AT91F_USB_SendData(pUDP, (char *) &(pAudio->currentConfiguration), sizeof(pAudio->currentConfiguration));
        break;
```

```
case STD_GET_STATUS_ZERO: //Envia o status do dispositivo
    wStatus = 0;
    AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
    break;
case STD_GET_STATUS_INTERFACE: //Envia o status da interface
    wStatus = 0;
    AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
    break;
case STD_GET_STATUS_ENDPOINT: //Envia o status do endpoint
    wStatus = 0;
    wIndex &= 0x0F;
    //Verifica se o dispositivo está no estado configurado e o endpoint selecionado
    //realmente existe
    if ((pUDP->UDP_GLBSTATE & AT91C_UDP_CONFG) && (wIndex <= 3)) {
        wStatus = (pUDP->UDP_CSR[wIndex] & AT91C_UDP_EPEDS) ? 0 : 1;
        AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
    }
    //Verifica se o dispositivo está no estado de endereço e o endpoint selecionado
    // é o de controle
    else if ((pUDP->UDP_GLBSTATE & AT91C_UDP_FADDEN) && (wIndex == 0)) {
        wStatus = (pUDP->UDP_CSR[wIndex] & AT91C_UDP_EPEDS) ? 0 : 1;
        AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
    }
    else
        //responde com STALL pois a requisição estava errada
        AT91F_USB_SendStall(pUDP);
    break;
    //Como não há nenhum "feature" em nosso dispositivo, os comandos a seguir são
    //ignorados, pois nunca ocorrerão
case STD_SET_FEATURE_ZERO: //Habilita alguma característica do dispositivo
    AT91F_USB_SendStall(pUDP);
    break;
case STD_SET_FEATURE_INTERFACE: //Habilita alguma característica da interface
    AT91F_USB_SendZlp(pUDP);
    break;
case STD_SET_FEATURE_ENDPOINT: //Habilita alguma característica do endpoint
    AT91F_USB_SendStall(pUDP);
    break;
case STD_CLEAR_FEATURE_ZERO: //Desabilita alguma característica do dispositivo
    AT91F_USB_SendStall(pUDP);
    break;
case STD_CLEAR_FEATURE_INTERFACE: //Desabilita alguma característica da interface
    AT91F_USB_SendZlp(pUDP);
    break;
case STD_CLEAR_FEATURE_ENDPOINT: //Desabilita alguma característica do endpoint
        AT91F_USB_SendStall(pUDP);
    break;
case STD_GET_INTERFACE: //Responde com o "alternate setting" ativo da interface
    //Verifica se o dispositivo está configurado e se a solicitação é para interface 1
    // pois neste caso é a única interface com "alternate settings"
    if(wIndex == 1 && (pAudio->currentConfiguration))
        AT91F_USB_SendData(pUDP, (char *) &(pAudio->currentSetting), MIN((sizeof(pAudio->currentSetting)), wLength));
    else
        AT91F_USB_SendStall(pUDP);
    break;
case STD_SET_INTERFACE: //Ativar um "alternate setting" da interface
    //Verifica se o dispositivo está configurado e se a solicitação é para interface 1
    // pois neste caso é a única interface com "alternate settings"
    if((wIndex == 1) && (wValue <= 1) && (pAudio->currentConfiguration)){
        pAudio->currentSetting = wValue;
        AT91F_USB_SendZlp(pUDP);
        //Caso o "alternate setting" seja 1, ele configura o EP de audio
        if(wValue == 1) {
            pUDP->UDP_RSTEP  = (unsigned int) -1;
            pUDP->UDP_RSTEP = 0;
            pUDP->UDP_CSR[AUDIO_IN] = (wValue) ? (AT91C_UDP_EPEDS | AT91C_UDP_EPTYPE_ISO_IN) : 0;
        }
        //Se o "alternate setting" for 0, ele desabilita o EP de audio
        else if(wValue == 0) {
            pUDP->UDP_CSR[AUDIO_IN] = 0;
        }
    }
    else
        AT91F_USB_SendStall(pUDP);
    break;

//Os outros pedidos não fazem sentido para esse dispositivo e serão todos respondidos
//com STALL, assim como os pedidos específicos da classe audio, já que nosso dispositivo
//não tem funcionalidades extras, apenas envia os sinais de voz.
default:
    AT91F_USB_SendStall(pUDP);
    break;
}
```

II. ROTINAS DO MATLAB

Listing II.1: Rotina para captura dos sinais

```
amostras = 64000
freq = 64000
dispositivo = 6
driver = 'dx'
mics = 8
fs = freq / mics;

%Programa que faz tudo

%Captura sinais
pa_wavrecord(driver)
dispositivo = input('Qual o dispositivo USB? : ');

    %a funcao pa_wavrecord captura o som a aprtir do dispositivo escolhido.
    %Deve-se escolher tambem o numero de canais, de amostras, frequencia de
    %amostragem e o driver, q no caso e DirectX.
input1 = pa_wavrecord(1,1,amostras,freq,dispositivo,driver);
fprintf('Próxima etapa: Decodificar o sinal\n')
pause;

%Desfaz a codificacao PCM8
fprintf('\nDecodificando o sinal\n')
for i = 1:64000
    if input1(i) < 0
        inputx(i)=(input1(i))*128+1;
    else
        inputx(i)=input1(i)*128;
    end
end

inputx = round(inputx);

fprintf('Próxima etapa: Demultiplexar o sinal\n')
pause;

%Demultiplexa o sinal
fprintf('\nDemulplexando o sinal\n')

    %Remontando o valor inicial
j=1;
for i = 1:2:15
    valorInic(j)=(inputx(i+1)+127)*256+(inputx(i)+127);
    j=j+1;
end

canal0(1) = valorInic(1);
canal1(1) = valorInic(2);
canal2(1) = valorInic(3);
canal3(1) = valorInic(4);
canal4(1) = valorInic(5);
canal5(1) = valorInic(6);
canal6(1) = valorInic(7);
canal7(1) = valorInic(8);

    %reconstrucao dos canais
j=2;
for k = 17:8:64000
        canal0(j) = canal0(j-1)+inputx(k);
        canal1(j) = canal1(j-1)+inputx(k+1);
        canal2(j) = canal2(j-1)+inputx(k+2);
        canal3(j) = canal3(j-1)+inputx(k+3);
        canal4(j) = canal4(j-1)+inputx(k+4);
        canal5(j) = canal5(j-1)+inputx(k+5);
        canal6(j) = canal6(j-1)+inputx(k+6);
        canal7(j) = canal7(j-1)+inputx(k+7);
        j=j+1;
    end
```