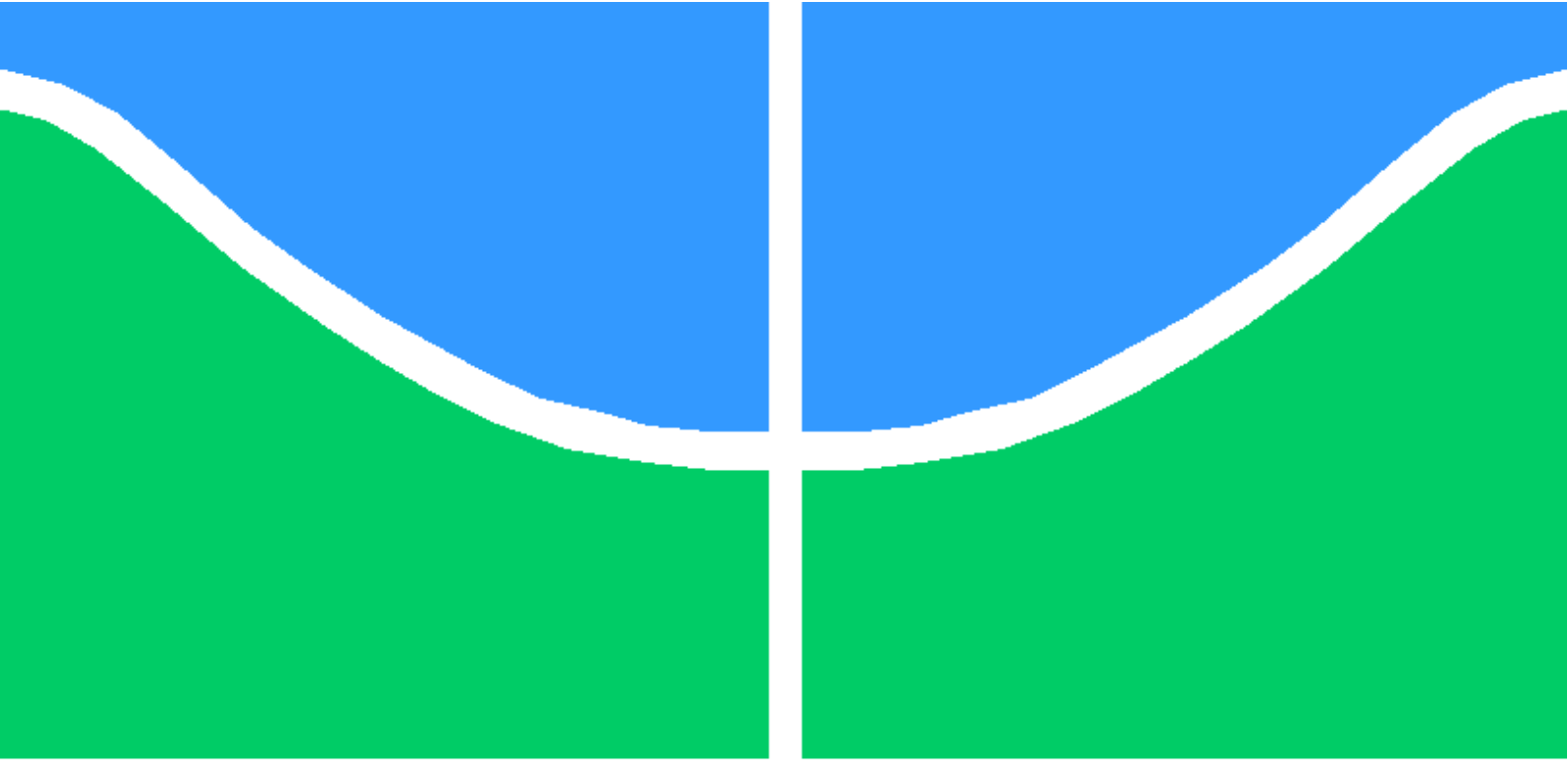


TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UMA INTERFACE USB PARA AQUISIÇÃO DE DADOS DE UM ARRANJO DE MICROFONES: APLICAÇÃO EM PRÓTESE AUDITIVA

Marcello Gurgel Sasaki

Otávio Viegas Caixeta

Brasília, dezembro de 2006

UNIVERSIDADE DE BRASÍLIA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE UMA INTERFACE USB PARA AQUISIÇÃO DE DADOS DE UM ARRANJO DE MICROFONES: APLICAÇÃO EM PRÓTESE AUDITIVA

Marcello Gurgel Sasaki

Otávio Viegas Caixeta

Relatório submetido ao Departamento de Engenharia Elétrica
da Faculdade de Tecnologia da Universidade de Brasília como
requisito parcial para obtenção do grau de Engenheiro Eletricista.

Banca Examinadora

Prof. Ricardo Zelenovsky, Doutor, UnB/ENE (Orientador) _____

Prof. Leonardo R. A. X. Menezes, Ph.D., UnB/ENE _____

Prof. Alexandre Zaghetto, Mestre, UnB/ENE _____

FICHA CATALOGRÁFICA

SASAKI, MARCELLO GURGEL
CAIXETA, OTÁVIO VIEGAS
Desenvolvimento de uma interface USB para aquisição de dados de um arranjo de microfones: aplicação em Prótese Auditiva . [Distrito Federal] 2006.
xvii, 134p.,210 x 197 mm (ENE/FT/UnB, Engenheiro Eletricista,2006)
Monografia de Graduação - Universidade de Brasília.
Faculdade de Tecnologia.
Departamento de Engenharia Elétrica.
1. USB 2. Firmware
3. Áudio 4. Microfone
I. ENE/FT/UnB II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

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.

CESSÃO DE DIREITOS

NOME DOS AUTORES: Marcello Gurgel Sasaki, Otávio Viegas Caixeta.

TÍTULO: Desenvolvimento de uma interface USB para aquisição de dados de um arranjo de microfones: aplicação em Prótese Auditiva .

GRAU / ANO: Engenheiro Eletricista / 2006

Marcello Gurgel Sasaki
SHIS QI 17, conj. 12, casa 10 - Lago Sul
71645-120 Brasília, DF - Brasil

Otávio Viegas Caixeta
SQN 406, bl. E, apto. 205 - Asa Norte
70847-050 Brasília, DF - Brasil

Dedicatórias

Eu dedico esse trabalho aos meus pais por confiarem em mim incondicionalmente. Que este seja um símbolo de sucesso não só meu, mas principalmente deles.

Marcello Gurgel Sasaki

A meus pais. Obrigado por nunca medirem esforços em me ensinar o que é certo e convencer-me a confiar em mim mesmo. Não saberia chegar onde estou sem vocês me mostrando o caminho..

Otávio Viegas Caixeta

Agradecimentos

Primeiramente eu agradeço a Deus que sempre se mostrou presente em minha vida.

A minha família, meu pai, Eduardo Wagner, que nunca me deixou faltar nada, deixando o caminho aberto para que eu seguisse o que era de minha vontade e vocação. Minha mãe, Vera Lúcia, que fez de tudo ao seu alcance para me estimular a crescer e buscar o conhecimento. Meus irmãos, Melissa e Wagner, com quem pude contar sempre. Minha avó, Bernadette, que sempre foi um exemplo de perseverança e força. Meus tios e primos que torceram sempre por mim.

Aos meus colegas com os quais conviví nos últimos cinco anos e se tornaram amigos para sempre. Izumi, minha fiel escudeira de todos os momentos, os companheiros de projeto, Ana Ravena, Francisco e meu co-autor Otávio. Os amigos, Fernanda, Bianchi, Samuel, Marcos, Thompson, Maria Clara, Luíza, Branquinho, Ewerton, Rogério, Solino, Artur, Utida, entre outros pelos momentos de descontração em meio às horas inacabáveis de estudo.

Aos professores e mestres por todo o conhecimento depositado em mim, em especial ao Prof. Ricardo Zelenovsky, orientador e fonte inesgotável de conhecimento. Ao Prof. Leonardo de Menezes pela primeira oportunidade acadêmica. Aos colaboradores de plantão do GPDS Edson Mintsu e Tiago Alves que nunca deixaram uma pergunta sem resposta. E aos colegas Carlos Vinícius e Raphael Hideki pela ajuda neste projeto.

A todas as pessoas que me apoiaram uma hora ou outra durante esses anos contribuindo para minha atual sanidade.

Marcello Gurgel Sasaki

Agradeço a minha família por todo o amor que recebi. A minha mãe, por todo seu carinho e por conseguir me ensinar que o mundo vai muito além do que eu vejo. A meu pai, por sua responsabilidade e dedicação em tudo o que se propõe a fazer. Você me inspira. A meu irmão, por sua paciência e amizade incondicionais.

A meus colegas de engenharia, por trazerem alegria nos momentos mais cansativos e desafiadores de minha vida. Nunca esquecerei as noites em claro estudando e as conversas intermináveis na procrastinação do estudo. Izumi, Lulis, Clara, Ana, Féfis, Marcão, TJ, Samuca, Pombo e tantos outros que tornam impossível dar a todos os espaço que aqui merecem. Obrigado por fazer da elétrica meu segundo lar.

A meus mestres, pelo exemplo de coerência e paixão pelo trabalho. Particularmente ao professor Ricardo Zelenovsky, por toda a orientação e apoio nos becos sem saída. Seu bom humor é contagiante.

A minha namorada, Marcela, por todo o apoio nas horas difíceis e por me fazer me esforçar sempre mais. Você me faz uma pessoa melhor.

Um obrigado também aos companheiros de GPDS, sempre presentes para elucidar qualquer dúvidas e aliviar a tensão. Em particular gostaria de agradecer ao Mintsu, Thiago Alves, Raphael Hideki e Carlos Vinicius por suas contribuições valiosas ao presente trabalho.

E um agradecimento especial a meu co-autor Sasaki, amigo sempre surpreendente, e meus parceiros de projeto Chico e Ana, por toda a dedicação e ânimo nos projetos que enfrentamos. Sem vocês esse projeto seria impossível.

Otávio Viegas Caixeta

RESUMO

O barramento USB se tornou o padrão em conexões entre computador e periféricos. A implementação de uma interface USB para um arranjo de microfones propicia uma oportunidade adequada de estudo desta tecnologia.

O presente trabalho apresenta conceitos básicos de USB e programação de microcontroladores. São também estudados recursos da tecnologia USB específicos para dispositivos que lidam com áudio, bem como sua aplicação no desenvolvimento de um *firmware* de captação sonora para o computador e uma interface gráfica para o tratamento dos sinais.

O resultado deste trabalho em conjunto com o resultado do projeto de um *front-end* de 8 microfones [1] formam um ambiente suficiente para captação e tratamento de sinais sonoros na faixa de frequência da voz humana. Depois de captados pelos 8 microfones presentes no *front-end*, os sinais de áudio são digitalizados e transferidos ao PC pelo barramento USB. No PC, os sinais são tratados por rotinas em Matlab.

O ambiente é todo controlado por estas rotinas, que têm o objetivo de receber os sinais capturados pelos microfones e estimar a direção de chegada deste conjunto de amostras. Para a estimação da direção de chegada são utilizados os métodos DS, CAPON, MUSIC e SPRIT [1].

ABSTRACT

The Universal Serial Bus has become the more pervasive technology to connect peripherals to a PC. The implementation of a USB interface for a microphone array presents itself as a suitable opportunity for studying this technology.

The present work yields basic USB and microcontroller programming concepts. Audio-specific USB functionality is also presented, as well as its application on firmware for audio capture peripherals.

The results achieved in this work combined with those of an 8-microphone front-end [1], create an adequate workstation to record and process sound signals at the human voice frequency band. After being captured by the front-end's 8 mics, the audio signal is digitalized and sent to the PC through the USB port. Finally, Matlab routines process the sampled data on the PC.

The process is completely managed by the afore-mentioned routines, whose objective is to receive the signal captured by the mics and estimate it's DOA (Direction Of Arrival). DS, CAPON, MUSIC and SPRIT methods are used to estimate the DOA [1].

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIMBOLOS

Símbolos Gregos

δ	atraso	[ms]

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*

1 INTRODUÇÃO

1.1 EVOLUÇÃO DAS INTERFACES DIGITAIS

Desde o início da era digital, com o advento dos primeiros computadores, estudou-se a possibilidade de interconexão entre sistemas digitais. Com o surgimento do micro-computador e o conceito de computador pessoal (PC) essa possibilidade se tornou uma necessidade. O computador pessoal precisa de periféricos para interfaceamento com o mundo real, inclusive com o próprio usuário. Entre outros exemplos de periféricos estão os dispositivos de interface humana tais como monitor, mouse e teclado.

Para o PC original foram definidos dois padrões de comunicação: serial e paralelo. Nestes padrões, a conexão física de um único periférico é feita através de um cabo ligado a uma porta paralela ou serial do PC.

A primeira porta paralela do PC utilizava sinal unidirecional no sentido PC $\rightarrow$ periférico, mas em 1980 foram adicionados mais 8 sinais para comunicação na direção contrária. Nas versões seguintes, esforços foram feitos para tornar a comunicação mais eficiente através do uso de técnicas de controle de fluxo e compressão de dados.

Já o sinal serial funcionava com bits sendo transmitidos em série por apenas um par de fios. O CI responsável por esta comunicação era chamado de UART (*Universal Asynchronous Receiver-Transmitter*) e seu desenvolvimento definiu a evolução da porta serial. As primeiras UARTs suportavam taxas abaixo de 9600 bps (aprox. 9600 bits por segundo). Um marco na evolução da comunicação serial foi a inclusão de uma FIFO (*First In, First Out*) na UART, pois isso possibilitou a transmissão serial de dados sem a total atenção do processador.

Tanto a porta paralela quanto a serial atingiram seu ponto máximo de evolução. No entanto, ainda se sentia a necessidade de mais banda e mais portas de expansão. Com essa idéia em mente, em 1994 um grupo de sete empresas líderes da indústria de computadores fundou um consórcio para a criação de um novo padrão de conexão de periféricos. Nascia o barramento *Universal Serial Bus*, o USB. A intensão era de que a USB substituísse as portas serial, paralela, PS/2 e de jogos. Com o tempo, não só essas

tecnologias foram substituídas como ainda surgiram outras aplicações bastante criativas para o barramento USB, tornando-o de fato um tipo universal de conexão.

Essa história de sucesso se deve em parte à especificação minuciosa do padrão USB. Na maioria dos projetos de interconexão com o PC, a interface USB é a que oferece mais vantagens. Além de ser o atual padrão para conexão de periféricos ao computador, a interface USB oferece alta taxa de transmissão de dados, até 480 Mbps, permite *hot swap*, ou seja, conexão e configuração sem a necessidade de reiniciar o computador, utiliza um protocolo de comunicação robusto com detecção e supressão automática de erros e ainda permite a alimentação do periférico pelo próprio barramento.

O barramento USB se tornou bastante versátil e pode ser aplicado a praticamente qualquer dispositivo. Algumas funções como áudio, vídeo, transferência e armazenamento de arquivos foram previstas. Muitas outras podem ser implementadas à medida que forem criadas.

1.2 PROJETO PAI

O arranjo de sensores em paralelo junto a uma unidade digital de processamento constitui uma ferramenta poderosa de processamento de sinais. Atualmente, diversas pesquisas são feitas usando arranjos de microfones em paralelo para aplicações com voz humana. Essa configuração além de aumentar a sensibilidade possibilita a seletividade espacial, melhorando assim a qualidade da onda sonora captada.

O presente trabalho visa a implementação da interface USB entre um arranjo de microfones e um computador PC. Esta tarefa faz parte de um projeto maior chamado de projeto PAI - Prótese Auditiva Inteligente. O objetivo deste projeto é a estimação da direção de chegada da voz humana através do processamento do sinal obtido pelo arranjo de microfones. Como este processamento exige um grande esforço computacional, foi decidido que os sinais dos microfones seriam digitalizados por um microcontrolador e enviados para o PC, onde deles seriam estimados a direção de chegada.

No projeto PAI foi especificado um arranjo composto por oito microfones, cada um com banda de 5 kHz, suficiente para registrar claramente a voz humana. A amostragem do sinal seria feita a 10 kHz de forma a satisfazer o teorema da amostragem de Nyquist. Para enviar o sinal ao PC, optou-se pelo uso de um microcontrolador com arquitetura ARM. Essa arquitetura foi escolhida por sua universalidade e facilidade

de emprego. O microcontrolador será a ponte entre o arranjo de microfones e o computador, utilizando o barramento USB como meio físico para a transferência dos dados captados pelos microfones. [1]

O projeto PAI foi dividido em duas frentes paralelas de trabalho: o desenvolvimento de um *hardware front-end* de captura dos sinais analógicos e o de um *software* de captura e tratamento digital dos sinais.

O *hardware* consiste no projeto de um dispositivo para a captura de sinais sonoros utilizando o arranjo de microfones em paralelo. O projeto do *front-end* envolve não só os sensores sonoros bem como a filtragem, pré-amplificação e entrega dos sinais analógicos adquiridos ao módulo de conversão analógico/digital.

A frente de *software*, contemplada neste trabalho, ficou responsável por digitalizar os sinais e enviá-los em tempo hábil para o PC através do barramento USB. Também é responsabilidade da frente de *software* o recebimento dos dados pelo computador e sua entrega às rotinas que estimam a direção de chegada.

1.3 OBJETIVOS

Este sub-projeto tem como um dos objetivos desenvolver um *firmware* para a plataforma ARM escrito em linguagem C que satisfaça a especificação USB 2.0. Após a implementação da interface USB o *firmware* deve receber os sinais digitalizados de oito microfones, realizar um processamento mínimo dos sinais e em seguida transferí-los para o computador, onde será feita a maior parte do processamento.

É também tarefa deste projeto desenvolver uma plataforma, residente no computador, que realize o recebimento dos sinais amostrados e estime a direção de chegada utilizando os programas já desenvolvidos.

1.4 CONTEÚDO

Inicialmente, no capítulo 2, será feita uma introdução às características do barramento USB. Nas seções seguintes são apresentadas a especificação para o desenvolvimento de um dispositivo USB 2.0 (capítulo 3) e as possibilidades de funções que esses dispositivos podem agregar (capítulos 4 e 5). Enfim as características da arquitetura ARM são discutidas no capítulo 6, seguidas pelo desenvolvimento e resultados no capítulo

7 e as conclusões do trabalho nocapítulo 8.

2 O BARRAMENTO USB

2.1 ARQUITETURA

A arquitetura USB é composta basicamente por três elementos bem definidos: o hospedeiro, ou *host*, o dispositivo, ou *device*, e a interconexão entre eles. Na interconexão é definida a topologia do barramento USB, os modelos de transferência de dados e o agendamento de transferências.

2.1.1 Física

O barramento foi definido no formato mestre-escravo, onde o *host* é o mestre e detém total controle sobre o barramento. Somente o *host* pode iniciar uma transferência, o que torna o desenvolvimento dos dispositivos mais simples, pois estes apenas devem obedecer aos pedidos feito pelo *host*. Assim sendo, todas as transferências são referenciadas ao *host*. O fluxo de dados saindo dos dispositivos em direção ao controlador *host* é chamado de *upstream*, enquanto que o fluxo de dados no sentido contrário é chamado de *downstream*.

Esta simplicidade no projeto do dispositivo vem ao custo de uma maior complexidade no *software* do *host*. Entre as tarefas por ele acumuladas estão o controle e gerenciamento de todas as transferências, a alimentação dos dispositivos a ele conectados e verificação constante da conexão de novos dispositivos.

No barramento pode existir apenas um *host* que é definido por um *host controller*. A este *host controller* está integrado um *root hub* que proporciona uma ou mais portas. A topologia do barramento USB pode ser composta por até 127 dispositivos, entre *hubs* e funções. Os *hubs* aumentam a quantidade de dispositivos que podem ser conectados ao barramento adicionando mais portas. No entanto, é definido pela especificação que um dispositivo pode ser conectado com no máximo 5 níveis de *hubs* entre ele e o *root hub*.

Três velocidades são especificadas para o barramento USB, *low-speed* de 1.5 Mb/s para dispositivos que não necessitam de muita banda, *full-speed* de 12 Mb/s e *high-speed* de 480 Mb/s. Esta última velocidade foi definida pela versão 2.0 da especificação USB.

2.1.2 Lógica

Um PC *host* cria dados para, ou consome dados do mundo real. Os dispositivos USB são responsáveis pela transdução desses dados. Devido à grande variedade de funções que podem ser empregadas, uma estrutura lógica bem definida foi especificada para o caminho desses dados. A estrutura lógica de conexão USB pode ser vista na figura 2.1.

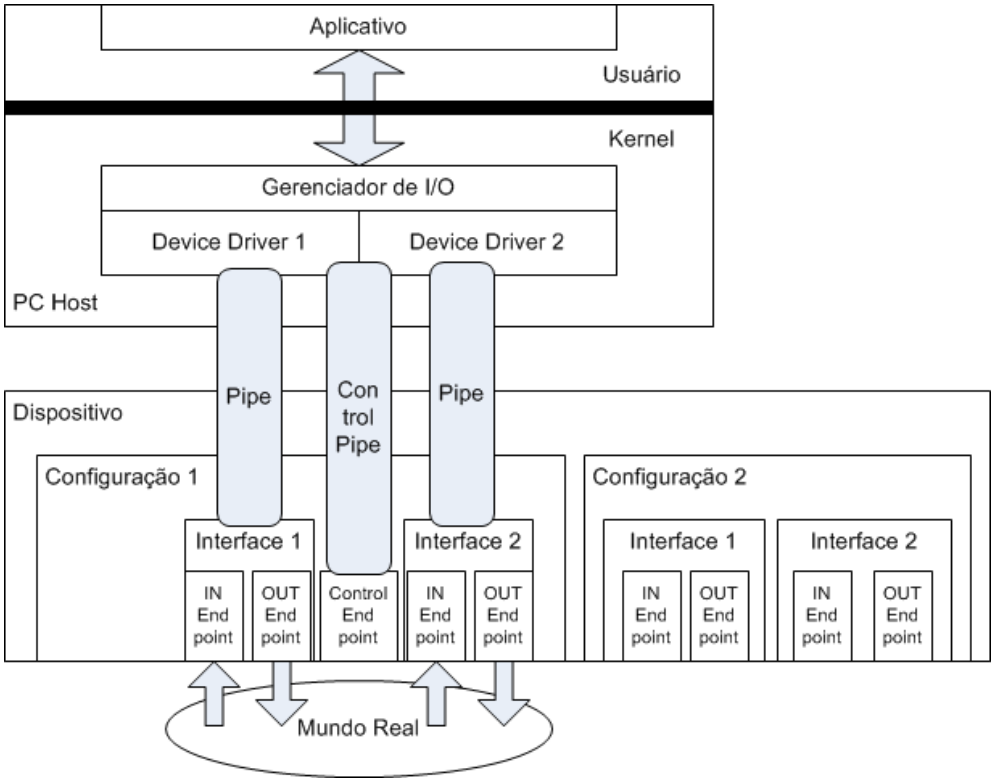


Figura 2.1: Arquitetura lógica da conexão USB

O termo *endpoint* é utilizado para descrever um ponte de entrada ou saída de dados de um sistema USB. O conjunto de um ou mais *endpoints* que implementam uma conexão com o mundo real é chamado de interface. Cada interface está associada a um *device driver* no *host* através de somente um *pipe*, ou canal. Um dispositivo pode ter mais de uma interface e todas elas funcionam em paralelo, cada uma com seu respectivo *driver*. Um conjunto de interfaces pertence a uma configuração, e somente uma configuração pode estar ativa num dado momento.

2.2 BARRAMENTO FÍSICO

O barramento físico USB é composto por quatro condutores: V_{bus}, *D*+, *D*- e *GND*. Os dados são transmitidos de forma diferencial pelos condutores *D*+ e *D*-. O sinal V_{bus} é responsável por fornecer alimentação aos dispositivos. Essa alimentação tem um limite máximo de 100 mA e, devido à sua praticidade, ajudou a tornar o padrão USB muito popular.

Tabela 2.1: Pinagem dos cabos USB

Pino	**Sinal**	**Cor**
1	V_{bus} (5V ou 3V3)	vermelho
2	*D*-	branco
3	*D*+	verde
4	GND	preto

Os cabos USB possuem especificações distintas para os terminais de *upstream* (conector tipo A) e *downstream* (conector tipo B). Há um outro padrão para conectores *downstream*, que foi aprovado pelo *USB Implementers Forum* (USB-IF) somente após a versão 1.1 da especificação USB. Este padrão, usado principalmente para dispositivos menores, é chamado de mini-B. Os cabos USB podem ter um comprimento máximo de 5 metros. Os conectores podem ser visto na Figura 2.2.

(a) Conector tipo A (*upstream*) (b) Conector tipo B (*downstream*)

Figura 2.2: Tipos de conectores USB

O cabo USB deve ser blindado com impedância de 90 Ω entre os condutores diferenciais *D*- e *D*+. Tanto os dispositivos quanto os *hubs* devem possuir resistores de *pull-up* ou *pull-down* conectados a esses condutores. Nos *hubs*, tanto *D*- quanto *D*+ são conectados a resistores de *pull-down* para garantir o estado D0 quando não há nenhum dispositivo ou *hub* conectado à porta. Nos dispositivos, a configuração dos resistores de *pull-up* define sua velocidade. Um resistor conectado a *D*- indica operação a 1,5 Mb/s enquanto que se estiver conectado a *D*+ indica operação a 12 Mb/s. Os dispositivos *high-speed* são inicialmente configurados como *full-speed* e durante a enumeração negociam a velocidade de 480 Mb/s com o *host*.

Valores típicos para os resistores são de 15 kΩ para os de *pull-down* e de 1,5 kΩ para os de *pull-up*.

2.3 PROTOCOLO DE COMUNICAÇÃO

Conforme dito anteriormente, os sinais *D*+ e *D*- são responsáveis pela comunicação serial no barramento. Para tanto, a especificação USB define 3 estados lógicos como vistos na Tabela 2.2.

Tabela 2.2: Estados lógicos do barramento USB

Estado			
Low Speed	**High/Full Speed**	***D-***	***D+***
K	J	baixo	alto
J	K	alto	baixo
SE0	SE0	baixo	baixo

Existe ainda um estado chamado *idle*, ou ocioso, que representa a ausência de atividade no barramento. Para dispositivos *low-speed* ou *full-speed*, este estado é equivalente ao estado J.

A comunicação entre o *host* e um dispositivo USB é feita através de transações compostas por pacotes. O primeiro pacote é sempre enviado pelo *host*. Dependendo do tipo de transação, os pacotes seguintes podem ser iniciados tanto por um *hub*, quanto por uma função [2]. Os pacotes são constituídos de pelo menos três campos: um de sincronização, chamado de SYNC, um de identificação do pacote chamado de PID (*Packet IDentificator*) e o EOP (*End Of Packet*) indicando o fim do pacote, definido por dois SE0. Entre o campo de PID e o campo EOP pode haver um campo de dados de no máximo 8 bytes para um dispositivo *low-speed*, 1023 para um *full-speed* e 1024 bytes para um *high-speed*.

O campo SYNC é uma sequência KJKJKJKK de 8 bits para dispositivos *low/full-speed*; já para o *high-speed*, o SYNC é formado de 15 transições KJ mais o KK no final, totalizando 32 bits. O PID identifica o tipo de pacote, sendo ele composto por quatro bits identificadores seguidos pelo seu complemento, totalizando 8 bits. Os tipos de pacotes e seus respectivos PID estão listado na Tabela 2.3.

Para a transmissão de dados é utilizado a codificação NRZI (*Non Return to Zero Inverted*). Nesta codificação, o 0 é representado pela inversão de estado nas linhas *D*- e *D*+, ou seja, de J para K ou vice-versa. Já 1 é representado mantendo-se os estados. Este esquema pode ser visto na Figura 2.3.

Tabela 2.3: Tipos de pacotes USB 2.0

PID	pacote	tipo
0101	SOF	token
1101	SETUP	
1001	IN	
0001	OUT	
0011	DATA0	data
1011	DATA1	
0111	DATA2	
1111	MDATA	
0010	ACK	handshake
1010	NACK	
1110	STALL	
0110	NYET	
1100	PRE	special
1100	ERR	
1000	SPLIT	
0100	PING	
0000	(reservado)	(reservado)

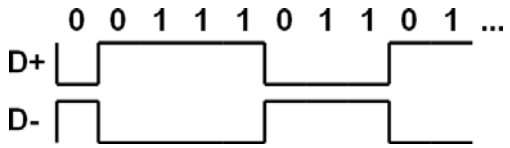


Figura 2.3: Exemplo de codificação NRZI

Visto que o CLOCK do barramento é transmitido através das transições das linhas *D*- e *D*+, a especificação USB determinou a inserção de um bit de *stuffing*, de forma que as linhas não permanecessem muito tempo em um mesmo estado durante a transmissão de uma sequência de 1's. Assim, a cada seis bits 1, um 0 é enviado de forma redundante. Um exemplo pode ser visto na Figura 2.4

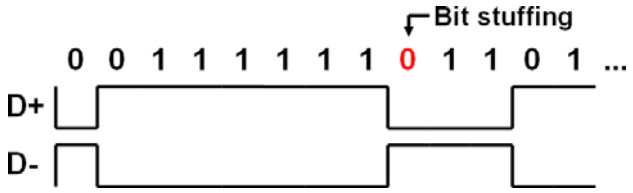


Figura 2.4: Exemplo de codificação NRZI com bit *stuffing*

2.4 TIPOS DE PACOTES

A seguir será explicado um pouco do que significa cada pacote e como eles são usados para organizar as transações pelo barramento USB.

2.4.1 Start Of Frame - SOF

Este pacote é enviado pelo host a cada 1 ms com a única finalidade de sincronismo. Ele é enviado a todos os dispositivos conectados ao barramento, ou seja, não tem distinção de endereço. Este pacote marca o início de um *frame* (ou quadro). No caso de um *root hub high-speed*, um quadro é dividido em 8 sub-quadros. Isto é feito usando SOFs adicionais. Sendo assim, cada sub-quadro tem uma duração de 125 μs. A Figura 2.5 mostra esse esquema de quadros e sub-quadros.

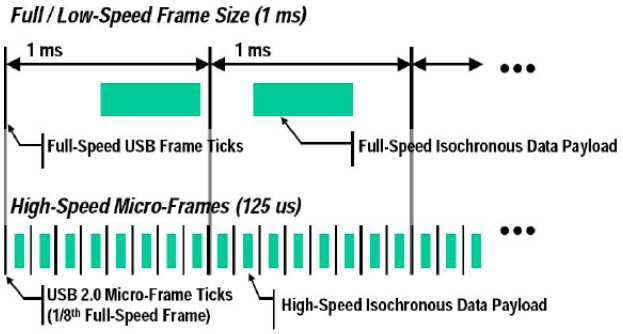


Figura 2.5: Esquema de frames e sub-frames do protocolo USB

No campo de dados desse pacote é enviado um contador crescente de quadros de 11 bits com mais 5 bits de CRC (*Cyclic Redundancy Check*) para checagem de erros durante a transmissão.

Tabela 2.4: Pacote de inicio de quadro - SOF

KJKJ..KK	8 bits	11 bits	5 bits	2 x SE0
SYNC	PID	contador de quadros	CRC	EOP

O CRC é gerado pelo transmissor apenas com os bits do campo de dados anterior ao CRC. Quando o pacote chega ao receptor, este cria um CRC a partir dos bits recebidos e, caso não seja igual ao CRC recebido, rejeita o pacote.

2.4.2 Pacotes Token - SETUP, IN e OUT

Os pacotes SETUP, IN e OUT são pacotes *token* enviados por *hubs* para indicar qual dispositivo vai participar da transação. O pacote SETUP é sempre endereçado ao *endpoint* 0 de um dispositivo, indicando que o *host* quer enviar algum comando de controle. Ele deve ser atendido o quanto antes, interrompendo qualquer outra transação. Pacotes IN e OUT são endereçados a qualquer *endpoint* de um dispositivo e indicam a direção do próximo pacote; IN é de leitura da função e OUT de escrita da função. O campo de dados do pacote é formado por 7 bits de endereçamento do dispositivo, 4 bits de identificação do *endpoint* e mais 5 bits de CRC.

Tabela 2.5: Pacotes de configuração - SETUP, IN e OUT

KJKJ..KK	**8 bits**	**7 bits**	**4 bits**	**5 bits**	**2 x SE0**
SYNC	PID	ADDR	ENDP	CRC	EOP

2.4.3 Pacotes de Dados - DATA0, DATA1, DATA2 e MDATA

Os pacotes de dados são enviados logo após um pacote de *token*, do *host* para o dispositivo se o pacote de *token* foi um SETUP ou um OUT e no sentido contrário se o *token* foi do tipo IN. O formato básico do campo de dados desse pacote é constituído pela informação, que varia de 0 a 1024 bytes mais 16 bits de CRC.

Tabela 2.6: Pacotes de dados - DATA0, DATA1, DATA2 e MDATA

KJKJ..KK	**8 bits**	**0 a 1.023 x 8 bits**	**16 bits**	**2 x SE0**
SYNC	PID	informação	CRC16	EOP

De forma a garantir a sincronização e sequência dos pacotes de dados, o protocolo USB utiliza um mecanismo chamado *Data Toggle* em alguns tipos de transmissão. Quando a informação a ser enviada ultrapassa o limite de 1024 bytes do pacote, ela é separada em vários pacotes que são mandados em sequência, alternando entre DATA0 e DATA1. No receptor os dados são validados apenas se a sequência for respeitada. Se ocorrer algum erro e dois pacotes DATA0 ou DATA1 forem recebidos em sequência, o receptor indica um erro na transmissão. Na especificação 2.0 foram adicionados os pacotes DATA2 e

MDATA, utilizados apenas para transferências do tipo *isochronous* em *high-speed*.

2.4.4 Pacotes de *Handshake* - ACK, NAK, STALL, NYET

Os pacotes de *handshake* são utilizados para negociar o estado da transação. Eles vão sempre na direção oposta à do último pacote enviado. Seu campo de dados é nulo.

Tabela 2.7: Pacotes de handshake - ACK, NAK, STALL e NYET

KJKJ..KK	**8 bits**	**2 x SE0**
SYNC	PID	EOP

ACK indica o recebimento com sucesso do pacote de dados ou de um *token*. NAK indica que o receptor por algum motivo não está apto a receber o pacote anterior. STALL indica que o receptor detectou um erro na transmissão, como um pedido inválido. O pacote NYET, implementado na versão 2.0 da especificação, é uma resposta ao pacote PING enviada pelo *host*. Esse pacote será melhor explicado mais adiante.

2.4.5 Pacotes Especiais - PRE, ERR, SPLIT e PING

A maioria dos pacotes dessa categoria é utilizada pelo *host* para se comunicar com *hubs*. O pacote PRE é utilizado pelos *hubs* para indicar uma transação com um dispositivo *low-speed*. Os pacotes ERR e SPLIT são utilizados em transações com *hubs* para diminuir a armazenagem de dados pelo *hub*, sendo o primeiro utilizado para indicar um erro nessa operação.

O pacote PING foi especificado na versão 2.0 para tornar o uso do barramento mais eficiente para dispositivos *high-speed*. Ao invés do *host* mandar sempre um pacote de dados OUT com o risco de receber um NAK, é enviado um pacote PING, que por ser bem menor utiliza menos banda do barramento. Caso o dispositivo esteja inapto a receber o pacote de dados ele deve responder com um NYET. O *host* então vai continuar enviando pacotes PING até que o dispositivo responda com um ACK, quando então o pacote de dados será finalmente transmitido.

2.5 TIPOS DE TRANSAÇÕES

Todas as transações USB são compostas pelos pacotes abordados na seção anterior. Uma transação deve ocorrer dentro de um mesmo *quadro* (ou *microquadro*) e quando iniciada não pode ser interrompida por outras transações até o seu término. As transações são compostas por três estágios. O primeiro estágio é sempre efetuado pelo *host* . Nele é enviado um dos pacotes *token* com as configurações da transação. No segundo estágio é feita a transferência de dados na direção especificada pelo pacote *token* . No terceiro estágio são enviados os pacotes de *handshake* confirmando o recebimento dos dados. A direção do segundo e do terceiro estágio são sempre opostas. Uma transferência pode conter uma ou mais transações.

De acordo com a especificação USB, são definidos quatro tipos de transferências: *Interrupt* (interrupção), *Bulk* (volumosa), *Isochronous* (isócrona) e *Control* (controle)[2]. Cada tipo de transferência é definida para diferentes tipos de dados que podem ser transmitidos. As principais características dos tipos de transferência podem ser vistos na Tabela 2.8.

Tabela 2.8: Tipos de transferências USB e suas características

		tamanho máximo		
tipo	**caracteristicas**	**LS**	**FS**	**HS**
Interrupt	tempo e integridade	8	64	3072
Bulk	integridade	-	64	512
Isochronous	tempo	-	1023	3072
Control	tempo e integridade	8	64	64

Os parâmetros para classificar os tipos de transferência são tempo e integridade dos dados. Tipos de transferências onde o tempo é fator crítico são aquelas em que os dados devem ser entregues a uma taxa definida e qualquer informação corrompida durante a transmissão perde sua importância, ou seja, são descartadas. Como exemplo têm-se os sinais de áudio e vídeo. No entanto, transferências onde a maior importância está na integridade dos dados entregues utilizam algumas técnicas de detecção e correção de erros, como o esquema de *acknoledgment* e retransmissão de pacotes. Assim, a taxa de informação pode não ser constante.

A seguir serão detalhadas as características de cada tipo de transferência.

2.5.1 Interrupt

As transações do tipo *interrupt* são recomendadas para transmissões que requerem tanto uma certa taxa quanto acurácia nos dados transmitidos. Apesar do nome, ela é feita de forma que o *host* use um *polling* periódico para checar por novos dados. Quando o *buffer* de envio ou de recepção deve receber novos dados, aquele *endpoint* ativa uma *flag* indicando o pedido de transmissão. Estas *flags* são verificadas pelo *host*. Os dados são então enviados ou recebidos a uma taxa baixa, usando todos os artifícios de tratamento de erros disponíveis. O *host* garante no máximo uma transmissão do tipo *interrupt* por quadro. Geralmente, os dispositivos que utilizam esse tipo de transferência utilizam uma taxa com período de alguns milisegundos.

A taxa de *polling* é pré-configurada e agendada pelo controlador *host*. Dentro dessa taxa o *host* inicia a transação enviando um pacote *token* do tipo IN ou OUT e o dispositivo deve responder de acordo. No caso da transação ser do tipo IN, o dispositivo envia um pacote de dados e espera um ACK do *host*, mas se não houver nenhum dado novo para envio ele apenas responde com um NACK. No caso da transação ser do tipo OUT o dispositivo deve estar preparado para receber um pacote de dados do *host* e, após o recebimento, deve responder com um ACK, ou NAK caso ocorra algum erro durante a transação. O dispositivo ainda pode responder com um STALL caso algum erro grave aconteça.

2.5.2 Bulk

A transação do tipo *bulk* utiliza exatamente os mesmo pacotes das transações do tipo *interrupt*. A diferença está no agendamento. O *host* aloca todo o espaço não utilizado de um *quadro* (ou *sub-quadro*) para as transações do tipo *bulk*. Essas transações têm como característica a acurácia dos dados. Logo, são utilizados pacotes de dados do tipo DATA0 e DATA1 para sincronização e checagem de erros. Os pacotes podem ser validados com ACKs ou descartados caso os CRC sejam diferentes. Neste caso é enviado um NAK e o pacote de dados deve ser retransmitido. Em dispositivos *high-speed* podem ser utilizados os pacotes do tipo PING e NYET para tornar a transferência mais eficiente e fazer melhor uso do barramento.

2.5.3 Isochronous

Às transações do tipo *isochronous* são garantidas um pedaço de cada *quadro* (ou *sub-quadro*). Nessas transações os pacotes de *handshake* não são utilizados, pois um dado atrasado é tão inútil quanto dado nenhum [3]. Os dados são transmitidos em pacotes do tipo DATA0 que podem conter de 0 a 1023 bytes. Quando um dispositivo que utilize esse tipo de transação é conectado ao barramento, o *host* deve garantir a banda requerida pelo dispositivo. A banda requerida será garantida em cada *quadro (sub-quadro)*, porém não será garantida a sua posição dentro do mesmo, podendo haver *jitter*, ou pequenas variações na taxa de entrega dos dados. Caso essa banda não esteja disponível o dispositivo não será inicializado e a conexão não será estabelecida.

Para dispositivos *full-speed* , que requerem uma banda larga, podem ser agendadas até três transações *isochronous*. Nesse caso obtém-se uma banda de 53 MB/s. Pacotes *token* adicionais são utilizados para a checagem de erros nas três transações. Esse procedimento é feito entre o *host* e o *hub* e é totalmente transparente do ponto de vista do dispositivo.

2.5.4 Control

Transações do tipo *control* são geralmente endereçadas ao *endpoint* 0 e carregam os pedidos pré-definidos do protocolo USB, chamados de *device requests*. Essas transações são as mais complexas, pois devem ser tratadas imediatamente e não toleram a existência de erros. Por isso, grande esforço foi colocado na especificação desse tipo de transação.

Uma transação do tipo *control* tem três fases. A primeira, chamada de *setup*, consiste em um pacote do tipo SETUP, um pacote de dados de tamanho fixo igual a 8 bytes e um pacote de *handshake*. A segunda fase, a de dados, é opcional. Ela consiste em um ou mais pacotes de dados, todos eles precedidos de um pacote de IN ou OUT e seguidos de um pacote de *handshake*. O final dessa fase é sinalizado com um pacote curto, ou seja, que não utiliza o tamanho máximo do pacote. A última fase, chamada de *Status*, é utilizada para confirmar a recepção da fase de *setup* ou de dados. Essa fase é composta de um pacote IN ou OUT, dependendo da direção da fase anterior, um pacote de dados de tamanho igual a zero e um pacote de ACK. Caso a fase anterior não seja validada pelo dispositivo, esta fase consistirá em um pacote do tipo

IN e um NAK ou STALL.

Na fase de *setup* é sempre enviado um pacote do tipo DATA0 de 8 bytes. Os dados desse pacote são pré-definidos pela especificação USB. São 5 parâmetros utilizados para definir o tipo de pedido: *bmRequestType*, *bRequest*, *wValue*, *wIndex* e *wLength*.

O primeiro parâmetro, *bmRequestType*, usa 8 bits para indicar a direção da próxima transação, o tipo de pedido e o destinatário do pedido. O segundo parâmetro, *bRequest*, indica qual informação o *host* está pedindo com 1 byte. Os próximos três parâmetros são de 2 bytes e são utilizados como suporte para os dados trocados. O parâmetro *wValue* é utilizando quando apenas um byte ou palavra é requerida. *wIndex* é utilizado quando um único índice é requerido. O parâmetro *wLength*, quando maior que um, indica o tamanho da próxima transferência de dados. Além dos pedidos padrões, cada classe tem seus pedidos específicos e outros ainda podem ser criados pelo próprio fabricante. Esses pedidos serão melhor explorados mais adiante.

3 ENUMERAÇÃO

A proposta do padrão USB foi a de criar um barramento de expansão para qualquer tipo de dispositivo, de forma que a conexão de novos dispositivos aconteça sem a intervenção do usuário. Para isso, o conceito de *plug and play* foi expandido e aplicado a todos os dispositivos USB. Dessa forma o usuário não precisa se preocupar com configurações de endereços, portas, IRQs ou DMAs. O processo de enumeração é realizado quando um dispositivo novo é detectado. Esse processo se encarrega das configurações, inclusive da escolha e instalação do *driver* correto para o dispositivo.

3.1 ETAPAS DA ENUMERAÇÃO

A enumeração se inicia com a detecção de um novo dispositivo. Este trabalho é realizado pelo *hub*, seja ele embarcado no controlador *host* ou um dispositivo físico conectado a uma das portas do primeiro. A partir do momento em que o *host* é iniciado, este faz um *polling* periódico no *root hub* a fim de descobrir se algum dispositivo novo foi conectado ao barramento. Todos os *hubs* devem responder a esse *polling* indicando os dispositivos detectados por eles ou simplesmente enviar um NAK caso nada tenha mudado desde o último pedido.

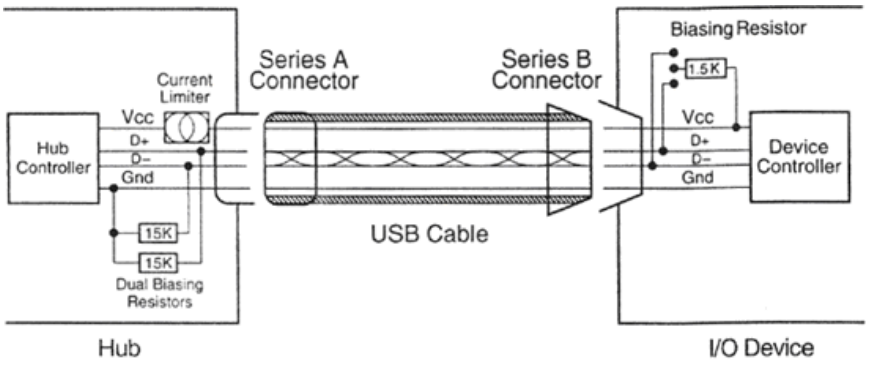


Figura 3.1: Detalhes da conexão do cabo USB

Como dito anteriormente, os *hubs* têm resistores de *pull-down* de cerca de 15 kΩ, conectados às linhas *D-* e *D+* de cada uma de suas portas, o que mantém as tensões nessas linhas próximas de zero. Quando um

dispositivo é conectado, a tensão em uma das linhas se eleva para próximo de V_{bus}. Isso acontece por que os dispositivos têm resistores de *pull-up* de 1,5 kΩ ligados à linha *D-* (*low-speed* 1,5 Mb/s) ou à linha *D+* (*full-speed* 12 Mb/s), detalhes podem ser vistos na Figura 3.1. O divisor resistivo faz com que o nível de tensão na linha fique mais próximo de V_{bus}. O *hub*, ao sentir essa mudança de nível informa ao *host* , no próximo *polling* , que um novo dispositivo foi conectado ao barramento.

Antes de conectado, o dispositivo se encontra no estado *unattached*. Ao ser conectado ao *hub,* o dispositivo passa para o estado *attached*. Como o *hub* já se encontra configurado e em operação, o dispositivo logo passa para o estado *powered*. O *hub* então atualiza o estado do dispositivo num registrador de status e espera pedidos do *host*. Nesse momento se inicia o processo de enumeração totalmente controlado pelo *host*. Este vai enviar pedidos tanto para o *hub* ao qual o dispositivo foi conectado quanto para o próprio dispositivo, afim de conhecê-lo e configurá-lo apropriadamente.

O seguinte processo exemplifica os pedidos feito por um *host* em plataforma Windows. De forma a ser compatível com qualquer plataforma, o dispositivo deve estar preparado para receber os pedidos em qualquer ordem e a qualquer momento, respondendo apropriadamente a cada um.

1. Request: Get_Port_Status. Para: *hub*.

   O *host* detecta um novo dispositivo.

2. Request: Clear_Port_Feature(C_PORT_CONNECTION). Para: *hub*.

   Limpa a *flag* indicando uma mudança de estado na porta do *hub*.

3. Request: Set_Port_Feature(PORT_RESET). Para: *hub*.

   O *hub* responde mandando um *reset* para o dispositivo. O *hub* mantém o *reset* por 10 ms e então habilita a porta. O PC *host* vai detectar essa mudança no próximo período de *polling*. É durante esse tempo de *reset* que o *hub* configura a velocidade do dispositivo. Se detectar uma tensão alta em *D-*, configura o dispositivo em *low-speed*. Se detectar nível alto em *D+*, configura como *full-speed*. Se o dispositivo suportar *high-speed*, ele primeiramente é configurado como *full-speed* e dentro desses 10 ms de *reset* ele manda um sequência de KJKJ... em alta velocidade. Se o *hub* suportar *high-speed* ele vai detectar essa sequência e responder ao dispositivo, configurando-o como *high-speed*. Caso contrário, o *hub* não consegue detectar essa sequência de KJKJ... e não responderá, fazendo com

que o dispositivo permaneça configurado como *full-speed*.

4. Request: Get_Port_Status. Para: *hub*.

   O *host* espera até que o dispositivo tenha retornado do estado de *reset*.

5. Request: Clear_Port_Feature(C_PORT_RESET). Para: hub.

   Limpa a *flag* no registrador do hub. Nesse ponto o dispositivo está alimentado e já foi reiniciado, logo ele se encontra no estado *default*. O dispositivo a partir de agora vai responder aos pedidos enviados ao endereço 0 (endereço padrão). Como o processo de enumeração ocorre de forma exclusiva apenas um dispositivo responderá pelo endereço padrão.

6. Request: Get_Device_Descriptor. Para: dispositivo.

   O dispositivo responde enviando o *device descriptor*, ou descritor do dispositivo.

7. Request: Set_Address. Para: dispositivo.

   O PC *host* aloca um endereço para o dispositivo. A partir desse momento todos os pedidos para esse dispositivo serão enviados para esse endereço. O dispositivo deve guardar o endereço e responder a todos os pedidos endereçados a ele.

8. Request: Get_Device_Descriptor. Para: dispositivo.

   O PC *host* repete este pedido para o novo endereço como verificação. Ele deve obter exatamente a mesma resposta obtida no pedido 6.

9. Request: Get_Configuration_Descriptor. Para: dispositivo.

   O PC *host* começa a coletar maiores informações sobre o dispositivo, suas configurações, interfaces e *endpoints*. Nesse ponto pode ser necessário intervenção do usuário, mas geralmente não é o caso.

10. Seleção do driver.

    O PC *host* inicia a busca por *drivers* para o dispositivo. Primeiramente, ele tenta encontrar um arquivo .INF com VendorID e ProductID equivalentes aos coletados durante a enumeração. Não encontrando, ele vai então analisar se o dispositivo se encaixa em uma classe específica e, caso consiga, vai carregar os *drivers* USB padrão. Caso não tenha obtido sucesso em nenhuma destas tentativas o usuário será perguntado pelos *drivers* do dispositivo.

11. Request: Set_Configuration. Para: dispositivo.

    O dispositivo agora se encontra configurado e operacional, passando para o estado *configured*.

3.2 PC REQUESTS PADRÕES

PC *Requests* são todos os pedidos enviados pelo *host* por um canal de controle, normalmente pelo *endpoint* 0 que é obrigatório em todo dispositivo. Este canal de controle está sempre aberto e operacional. Nessa seção serão explorados todos os *Standard PC Requests*, que são os pedidos mínimos ao qual um dispositivo deve responder, listados na Tabela 3.1.

Os pedidos são totalmente definidos por dois parâmetros, o *bmRequestType* e o *bRequest*. O parâmetro *bmRequestType* é mapeado bit a bit da seguinte maneira: o bit 7 define a direção da fase de dados que segue a fase de *setup*, os bits 6..5 definem o grupo ao qual o pedido pertence e os bits 4..0 definem o destinatário do pedido. Os possíveis valores podem ser vistos na Figura 3.2, valores não mostrados são reservados.

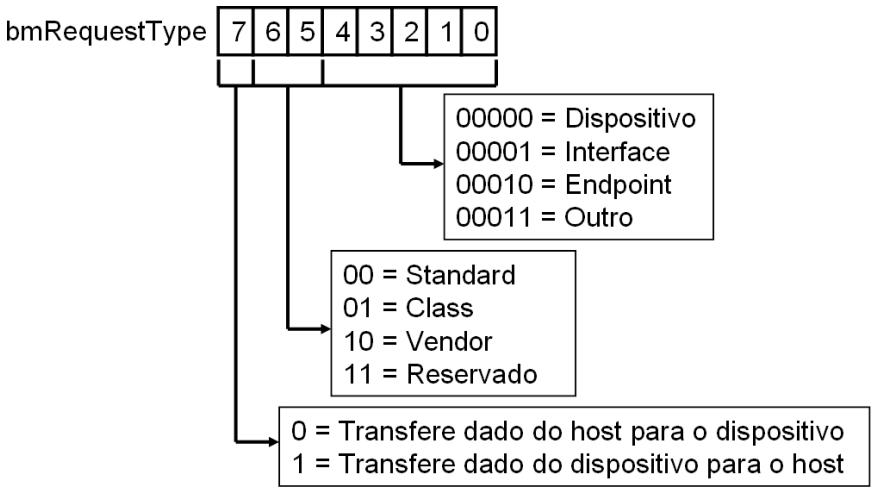


Figura 3.2: Mapeamento dos bits do parâmetro bmRequest

Já o parâmetro *bRequest* é um valor correspondente ao pedido, como pode ser visto na tabela 3.1.

Devido à sua importância, os pedidos padrões serão explicados um a um. Vale notar que a USB representa variáveis de mais de um byte na forma *little-endian*. Dessa forma, o primeiro byte representado é o menos significativo (LSB, *Less Significant Byte*), seguido pelo byte mais significativo (MSB, *Most Significant Byte*).

Tabela 3.1: Identificadores dos Standard PC Requests

bRequest	Requests
0	GET_STATUS
1	CLEAR_FEATURE
2	Reservado
3	SET_FEATURE
4	Reservado
5	SET_ADDRESS
6	GET_DESCRIPTOR
7	SET_DESCRIPTOR
8	GET_CONFIGURATION
9	SET_CONFIGURATION
10	GET_INTERFACE
11	SET_INTERFACE
12	SYNCH_FRAME

3.2.1 Get Status

Esse pedido retorna o atual status do destinatário especificado.

• *bmRequestType*: 10000000b (dispositivo), 10000001b (interface), 10000010b (*endpoint*).

• *bRequest*: GET_STATUS.

• *wValue*: zero.

• *wIndex*: especifica a interface ou *endpoint*. É igual a zero se for destinada ao dispositivo.

• *wLength*: dois.

• Dados: status do dispositivo, interface, ou *endpoint*. Este campo é mapeado bit a bit e seus valores dependem do destinatário do pedido. Para o dispositivo, o bit 0 indica se este está atualmente alimentado externamente (1) ou pelo barramento (0), e o bit 1 indica se o dispositivo está atualmente habilitado para *remote wakeup* (1). O valor padrão para esse bit é 0, ou desabilitado. O valor do bit 1 pode ser alterado com os pedido SET_FEATURE e CLEAR_FEATURE. Todos os outros bits são reservados e devem ser iguais a zero. Para interface nenhum bit é especificado, todos são reservados e devem ser igual a zero. Para *endpoint*, somente o bit 0 é especificado, ele indica se o *endpoint* está inoperante (*halt*) (1), ou em operação normal (0). Esse bit pode ser modificado com os pedidos SET_FEATURE e CLEAR_FEATURE.

Estados válidos: *addressed* e *configured*. Caso o pedido especifique uma interface ou *endpoint* que não exista o dispositivo responde com STALL.

3.2.2 Clear Feature

Este pedido é utilizado para desativar uma *feature*, ou função, do dispositivo, interface ou *endpoint*.

- *bmRequestType*: 00000000b (dispositivo), 00000001b (interface), 00000010b (*endpoint*).
- *bRequest*: CLEAR_FEATURE.
- *wValue*: *feature*, ou função. Os valores possíveis de função dependem do *bmRequestType*. Se for para *endpoint*, tem apenas a função ENDPOINT_HALT (0) que coloca um *endpoint* em estado inoperante. Para o dispositivo, tem duas funções, DEVICE_REMOTE_WAKEUP (1) que habilita o dispositivo a retirar o *host* do estado suspenso, e TEST_MODE (2) que ativa a bateria de testes definidas na especificação USB. Para interface não tem nenhuma função definida.
- *wIndex*: indica qual interface ou *endpoint* é o destinatário. No caso em que o dispositivo é o destinatário, o valor é igual a zero.
- *wLength*: zero.
- Dados: vazio.

Estados válidos: *addressed*: somente para o *endpoint* 0; *configured*.

Em caso de erro: o dispositivo deve responder com um STALL.

3.2.3 Set Feature

Este pedido é utilizado para habilitar uma certa função.

- *bmRequestType*: 00000000b (dispositivo), 00000001b (interface), 00000010b (*endpoint*).
- *bRequest*: SET_FEATURE.

• *wValue*: *feature*, ou função. Os valores possíveis dependem do *bmRequestType*. Se for para *endpoint*, há apenas a função ENDPOINT_HALT (0), que coloca um *endpoint* em estado inoperante. Para o dispositivo há duas funções, DEVICE_REMOTE_WAKEUP (1), que habilita o dispositivo a retirar o *host* do estado suspenso e TEST_MODE (2), que ativa a bateria de testes definidas na especificação USB. Para interface não há nenhuma função definida.

• *wIndex*: o MSB especifica o tipo de teste caso a função seja TEST_MODE. Os seletores de TEST_MODE válidos são listados na Tabela 3.2, caso contrário ele é igual a zero. O LSB indica o número da interface ou *endpoint*. Se o pedido for para o dispositivo esse valor é igual a zero.

• *wLength*: zero.

• Dados: vazio.

Tabela 3.2: Seletores de modo de teste

valor	**descrição**
00H	Reservado
01H	Test_J
02H	Test_K
03H	Test_SE0_NAK
04H	Test_Packet
05H	Test_Force_Enable
06H-3FH	Reservado para seletores de teste padrão
3FH-BFH	Reservado
C0H-FFH	Reservado para modos de teste específicos do fabricante

Estados válidos: *default*, o dispositivo deve aceitar apenas SET_FEATURE para TEST_MODE nesse estado; *addressed*, o dispositivo deve aceitar esse pedido apenas para interface e *endpoint* zero, caso contrário responde com STALL; *configured*.

3.2.4 Set Address

Este pedido configura o endereço que será utilizado para todos os acessos futuros ao dispositivo.

• *bmRequestType*: 00000000b.

• *bRequest*: SET_ADDRESS.

• *wValue*: endereço do dispositivo. Todos os futuros acessos a esse dispositivo serão direcionados a esse endereço.

• *wIndex*: zero.

• *wLength*: zero.

• Dados: vazio.

Estados válidos: *default*, o dispositivo deve mudar para o estado *addressed* somente se o endereço for diferente de zero, caso contrário permanece nesse estado; *addressed*, se o endereço for igual a zero o dispositivo deve mudar para o estado *default*, caso contrário deve permanecer nesse estado e passar a utilizar o endereço recebido.

3.2.5 Get Descriptor

Este pedido é utilizado para recuperar um descritor do dispositivo.

• *bmRequestType*: 10000000b.

• *bRequest*: GET_DESCRIPTOR.

• *wValue*: o MSB especifica o tipo de descritor e o LSB especifica o índice do descritor apenas para descritor de configuração ou de *string*. O índice é utilizado quando um dispositivo possui mais de um descritor de configuração ou de *string*, caso contrário ele sempre será igual a zero.

• *wIndex*: especifica o LanguageID no caso de descritor de *string*, caso contrário é igual a zero.

• *wLength*: especifica o numero de bytes a ser enviado. Se o descritor for maior que o valor desse campo apenas o inicio do descritor deve ser enviado. Caso o tamanho do descritor seja menor, a transação deve ser terminada com um pacote curto (de tamanho menor que o máximo), ou de tamanho zero.

• Dados: descritor.

Três tipos de descritores são requisitados pelo *host*, DEVICE (também DEVICE_QUALIFIER), CONFIGURATION (também OTHER_SPEED_CONFIGURATION) e STRING. Todo dispositivo deve forne-

cer o descritor de dispositivo e ao menos um descritor de configuração. Um pedido do descritor do tipo CONFIGURATION deve retornar os descritores de configuração e de todas as interfaces e *endpoints* que pertencem àquela configuração.

Os descritores que fazem parte de uma mesma configuração devem seguir a seguinte ordem hierárquica: logo após o descritor de configuração, é declarado o descritor da primeira interface, seguido por descritores específicos de interfaces que a estendem. Então são enviados os descritores dos *endpoints*, que pertencem à interface, e os descritores específicos que os estendem. Se houverem outras interfaces na mesma configuração, essas devem seguir o descritor do último *endpoint* da interface anterior.

Estados válidos: *Default*, *addressed* e *configured*.

3.2.6 Set Descriptor

Este pedido é opcional e é utilizado para alterar um descritor já existente ou adicionar um novo descritor.

- *bmRequestType*: 00000000b.
- *bRequest*: SET_DESCRIPTOR.
- *wValue*: O MSB especifica o tipo de descritor e o LSB especifica o índice do descritor apenas para descritor de configuração ou de *string*. O índice é utilizado quando um dispositivo possui mais de um descritor de configuração ou de *string*, caso contrário ele sempre será igual a zero.
- *wIndex*: especifica o LanguageID no caso de descritor de *string*, caso contrário é igual a zero.
- *wLength*: especifica o número de bytes a ser enviado do *host* para o dispositivo.
- Dados: descritor. Os tipos válidos de descritores são apenas *device*, *configuration* e *string*.

Estados válidos: *addressed* e *configured*. Se o dispositivo não suportar esse tipo de pedido deve responder com STALL.

3.2.7 Get Configuration

Este pedido é utilizado para recuperar a configuração atual do dispositivo.

- *bmRequestType*: 10000000b.
- *bRequest*: GET_CONFIGURATION.
- *wValue*: zero.
- *wIndex*: zero.
- *wLength*: um.
- Dados: valor de configuração.

Se o dispositivo retornar o valor zero, ele não está configurado.

Estados válidos: *addressed*: retorna zero; *configured*: retorna o número da configuração atual, deve ser diferente de zero.

Em caso de erro: responder com um STALL.

3.2.8 Set Configuration

Este pedido habilita uma configuração do dispositivo.

- *bmRequestType*: 00000000b.
- *bRequest*: SET_CONFIGURATION.
- *wValue*: o número da configuração a ser utilizada é colocado no byte menos significativo desse campo. Esse valor deve ser igual a zero ou outro valor válido especificado nos descritores de configuração. Se o valor for igual a zero, o dispositivo deve mudar para o estado *addressed*.
- *wIndex*: zero.
- *wLength*: zero.
- Dados: vazio.

Estados válidos: *addressed*, se a configuração especificada é a zero, o dispositivo permanece nesse estado. Se for especificada uma configuração válida diferente de zero, o dispositivo é configurado e deve mudar para o estado *configured*, caso contrário o dispositivo deve responder com STALL; *configured*, caso a configuração zero seja especificada o dispositivo deve mudar para o estado *addressed*. Se for especificada uma configuração válida diferente de zero, o dispositivo deve ser configurado de acordo e permanece nesse estado, caso contrário o dispositivo responde com STALL.

3.2.9 Get Interface

Esse pedido é apenas utilizado em dispositivos que suportam mais de uma configuração na mesma interface, ele retorna o valor da função alternativa da interface especificada (*alternate setting*). Essas funções de interface são mutualmente exclusivas e definem dois modos de operação da mesma interface.

- *bmRequestType*: 10000001b.
- *bRequest*: GET_INTERFACE.
- *wValue*: zero.
- *wIndex*: especifica a interface, caso ela não existo o dispositivo responde com STALL.
- *wLength*: um.
- Dados: função alternativa da interface.

Estados válidos: *configured*. Se esse pedido for feito ao dispositivo enquanto ele se encontra no estado *addressed*, o dispositivo responde com STALL.

3.2.10 Set Interface

Este pedido permite ao *host* selecionar uma função de interface para a interface especificada.

- *bmRequestType*: 00000001b.
- *bRequest*: SET_INTERFACE.
- *wValue*: função alternativa de interface.

- *wIndex*: especifica a interface, caso ela não exista o dispositivo responde com STALL.
- *wLength*: zero.
- Dados: vazio.

Estados válidos: *configured*. O dispositivo deve responder com STALL se estiver no estado *addressed*.

3.2.11 Synch Frame

Este pedido é usado para marcar e depois avisar o quadro de sincronização de um *endpoint*.

- *bmRequestType*: 10000010b.
- *bRequest*: SYNCH_FRAME.
- *wValue*: zero.
- *wIndex*: especifica o *endpoint*. O *endpoint* deve estar configurado para transferências do tipo *isochronous*
- *wLength*: dois.
- Dados: número do quadro.

Estados válidos: *configured*. O dispositivo deve responder com STALL se estiver no estado *addressed*.

3.3 DESCRITORES

Descritores, ou *descriptors*, são estruturas de dados que contém todos os parâmetros do dispositivo USB de uma forma organizada e específica. Os descritores padrões são definidos pela especificação USB. Descritores também podem ser definidos por uma classe ou até mesmo pelo fabricante do dispositivo. Nessa seção serão explicados os descritores padrões da especificação USB.

Um dispositivo deve ter obrigatoriamente pelo menos dois descritores: DEVICE e CONFIGURATION. Outros descritores são adicionados de acordo com as características de operação do dispositivo. Existem ainda descritores opcionais de STRING, que fornecem informações humanamente inteligíveis.

Todos os descritores padrões começam com dois parâmetros de 1 byte denominados *bLength* e *bDescriptorType*. O primeiro indica o tamanho do descritor e o segundo o tipo do descritor. Os tipos de descritores padrões estão definidos na Tabela 3.3.

Tabela 3.3: Tipos de Descritores

bDescriptorType	Tipo de Descritor
1	DEVICE
2	CONFIGURATION
3	STRING
4	INTERFACE
5	ENDPOINT
6	DEVICE_QUALIFIER
7	OTHER_SPEED_CONFIGURATION
8	INTERFACE_POWER

3.3.1 Dispositivo

O descritor de dispositivo (DEVICE) descreve informações gerais sobre o dispositivo USB. Os parâmetros são válidos para todas as configurações do dispositivo. Em um dispositivo só pode haver um descritor do tipo DEVICE. Para o caso de dispositivos *high-speed* que têm parâmetros diferentes para funcionamento em *full-speed* e *high-speed*, além do descritor do tipo DEVICE também deve haver o descritor DEVICE_QUALIFIER. Os campos do descritor do tipo DEVICE podem ser vistos na Tabela 3.4 e os do descritor do tipo DEVICE_QUALIFIER na Tabela 3.5.

Os dois primeiros campos, como já foi dito, existem em todos os descritores padrão

O campo bcdUSB guarda a versão da especificação USB que o dispositivo implementa. O valor é escrito no formato BCD (*Binary Coded Decimal*) da seguinte forma: 0xJJMN, onde JJ representa o numero da versão, M o número da sub-versão e N o número da sub-sub-versão (JJ.M.N). A versão 2.0 da especificação USB é representada da seguinte maneira 0x0200.

Os campos *bDeviceClass*, *bDeviceSubClass* e *bDeviceProtocol* definem a classe, subclasse e protocolo que o dispositivo se encaixa. Normalmente o valor desses campos é igual a zero, visto que esses parâmetros geralmente são configurados no âmbito da interface nos seus respectivos descritores.

O campo *bMaxPacketSize0* define o tamanho do *buffer* do *endpoint* 0 em bytes. Esse valor deve ser 8

Tabela 3.4: Descritor de dispositivo - DEVICE

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (18)
1	*bDescriptorType*	1	Constante (DEVICE)
2	*bcdUSB*	2	BCD
4	*bDeviceClass*	1	Classe
5	*bDeviceSubClass*	1	SubClasse
6	*bDeviceProtocol*	1	Protocolo
7	*bMaxPacketSize0*	1	Número
8	*idVendor*	2	ID
10	*idProduct*	2	ID
12	*bcdDevice*	2	BCD
14	*iManufacturer*	1	Índice
15	*iProduct*	1	Índice
16	*iSerialNumber*	1	Índice
17	*bNumConfigurations*	1	Número

ou um múltiplo de 8 e não pode ser maior que o especificado para um *endpoint* do tipo *control*. Ver Tabela 2.8.

No campo *idVendor* é colocado um número único de até 2 bytes correspondente ao identificador do fabricante, que é disponibilizado pelo USB-IF. Maiores detalhes podem ser obtidos no site do USB-IF [4].

O campo *idProduct* guarda um número de até 2 bytes escolhido pelo fabricante para ser o número de identificação do produto implementado. Esse número deve ser único para cada produto de um determinado fabricante.

O campo *bcdDevice* representa a versão de *firmware* do produto em BCD, ele é definido pelo próprio desenvolvedor.

Nos campos *iManufacturer*, *iProduct* e *iSerialNumber* são colocados índices para os descritores do tipo STRING para os nomes do fabricante, do produto e do numero de série do produto, respectivamente.

O último campo, *bNumConfigurations*, informa quantas configurações (descritores do tipo CONFIGURATION) o dispositivo possui. Esse número é no mínimo igual a um, pois todo dispositivo deve ter ao menos uma configuração.

O descritor do tipo DEVICE_QUALIFIER é utilizado para configurar dispositivos *high-speed* e inclui quase todos os parâmetros do descritor do tipo DEVICE.

Tabela 3.5: Segundo descritor para dispositivos *high-speed* - DEVICE_QUALIFIER

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número
1	*bDescriptorType*	1	Constante (DEVICE_QUALIFIER)
2	*bcdUSB*	2	BCD
4	*bDeviceClass*	1	Classe
5	*bDeviceSubClass*	1	SubClasse
6	*bDeviceProtocol*	1	Protocolo
7	*bMaxPacketSize0*	1	Número
8	*bNumConfigurations*	1	Número
9	*bReserved*	1	Zero

3.3.2 Configuração

O descritor do tipo CONFIGURATION tem informações sobre configurações específicas do dispositivo. Um dispositivo pode ter várias configurações, cada configuração é descrita por um descritor CONFIGURATION e pelos descritores de interface e *endpoint* associados a ela. Quando o descritor CONFIGURATION é solicitado pelo *host*, ele é enviado junto com todos os descritores associados. Os parâmetros do descritor CONFIGURATION estão listados na tabela 3.6.

Tabela 3.6: Descritor de configuração - CONFIGURATION

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (9)
1	*bDescriptorType*	1	Constante (CONFIGURATION)
2	*wTotalLength*	2	Número
4	*bNumInterfaces*	1	Número
5	*bConfigurationValue*	1	Número
6	*iConfiguration*	1	Índice
7	*bmAttributes*	1	Bit Mapeado
8	*bMaxPower*	1	mA

O campo *wTotalLength* informa o numero total de bytes da configuração, incluindo os descritores de configuração, de sua(s) interface(s) e *endpoint(s)* e incluindo os descritores específicos de classes ou os definidos pelo fabricante. Esse campo é importante para a devida configuração do dispositivo durante a fase de enumeração. Se a configuração completa for maior ou menor que a especificada nesse campo, um erro será levantado e o dispositivo não será configurado.

O campo *bNumInterfaces* informa o número de interfaces associadas a essa configuração.

O campo *bConfigurationValue* guarda um valor que identifica essa configuração. Esse valor deve ser único para o mesmo dispositivo. Ele é utilizado pelo pedido SET_CONFIGURATION para selecionar essa configuração.

O campo *iConfiguration* é um índice para um descritor do tipo STRING que descreve essa configuração.

O campo *bmAttributes* é mapeado bit a bit da seguinte maneira: o bit 7 é reservado e por razões históricas deve ser igual a 1, se o bit 6 for igual a 1 indica que o dispositivo tem alimentação própria (externa ao barramento), o bit 5 indica se o dispositivo suporta *remote wakeup* (1) ou não (0), os bits de 4 a 0 também são reservados mas devem ser colocados em 0.

O campo *bMaxPower* informa a metade da corrente em mA de que o dispositivo irá necessitar. Esse valor é limitado a 250 que equivale a 500 mA, o máximo que o barramento pode prover a um dispositivo. Se esse valor for diferente de zero e o bit 6 do campo *bmAttributes* for igual a 1, quer dizer que o dispositivo é alimentado tanto pelo barramento quanto por uma fonte externa.

Assim como existe o descritor do tipo DEVICE_QUALIFIER para o dispositivo *high-speed*, existe um descritor de configuração para o dispositivo que está operando em *high-speed* denominado OTHER_SPEED_CONFIGURATION. Esse descritor tem a estrutura idêntica ao descritor do tipo CONFIGURATION porém o campo *bDescriptorType* é igual a OTHER_SPEED_CONFIGURATION.

3.3.3 Interface

Interfaces de uma configuração são descritas com o uso de descritores do tipo INTERFACE. Os parâmetros desse descritor podem ser vistos na Tabela 3.7. O descritor INTERFACE é sempre enviado junto com o descritor CONFIGURATION, ele não pode ser acessado individualmente. Uma interface pode incluir funções alternativas de forma que seus parâmetros possam ser alterados depois que o dispositivo já foi configurado. A função alternativa padrão de uma interface é a zero. O pedido SET_INTERFACE é utilizado para selecionar outra função alternativa da interface. Para cada função alternativa é necessário um descritor de interface e dos *endpoints* associados a ela.

O campo *bInterfaceNumber* identifica a interface dentro de uma configuração. Funções alternativas de

Tabela 3.7: Descritor de interface - INTERFACE

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (9)
1	*bDescriptorType*	1	Constante (INTERFACE)
2	*bInterfaceNumber*	1	Número
3	*bAlternateSetting*	1	Número
4	*bNumEndpoints*	1	Número
5	*bInterfaceClass*	1	Classe
6	*bInterfaceSubClass*	1	SubClasse
7	*bInterfaceProtocol*	1	Protocolo
8	*iInterface*	1	Índice

uma mesma interface têm o mesmo valor para esse campo.

O campo *bAlternateSetting* identifica as funções alternativas de uma interface, a padrão que deve sempre existir, é igual a zero.

O campo *bNumEndpoints* informa o número de descritores de *endpoints* que segue esse descritor, ou seja, o número de *endpoints* associados a essa função específica dessa interface. Caso a interface utilize apenas o *endpoint* 0, nenhum descritor do tipo ENDPOINT é necessário, assim o valor desse campo é igual a zero.

Os campos *bInterfaceClass*, *bInterfaceSubClass* e *bInterfaceProtocol* indicam em que classe, subclasse e protocolo USB essa interface se enquadra. Os valores válidos aqui são os das classes aprovadas pelo USB-IF ou 0xFF que é para uma classe definida pelo desenvolvedor do dispositivo. Todos os outros valores são reservados. Maiores detalhes sobre as classes existentes podem ser obtidos no site da USB-IF.

Por fim, o campo *iInterface* informa o índice para uma descrição da função da interface contida num descritor do tipo STRING.

3.3.4 Endpoint

Cada *endpoint* de uma interface, com exceção do *endpoint* 0, tem seu descritor do tipo ENDPOINT. Eles são utilizados para fornecer informações para que o *host* determine a largura de banda necessária para cada *endpoint*. Estes descritores são enviados junto com o descritor da configuração a qual pertence a interface a que ele está associado, nunca individualmente. A Tabela 3.8 mostra os parâmetros desse

descritor.

Tabela 3.8: Descritor de *endpoint* - ENDPOINT

Offset	**Campo**	**Bytes**	**Valor**
0	*bLength*	1	Número (7)
1	*bDescriptorType*	1	Constante (ENDPOINT)
2	*bEndpointAddress*	1	*Endpoint*
3	*bmAttributes*	1	Bit Mapeado
4	*wMaxPacketSize*	2	Número
6	*bInterval*	1	Número

O campo *bEndpointAddress* define o endereço do *endpoint* dentro do dispositivo. Esse campo é mapeado da seguinte maneira: o bit 7 define a direção do fluxo de dados, IN (1) ou OUT (0), os bits 6 a 4 são reservados e devem ser desativados, os bits 3 a 0 definem o número do *endpoint*. Dois *endpoints* de direções opostas podem ter o mesmo número contanto que o valor do campo seja único.

O campo *bmAttributes* é mapeado bit a bit e define atributos do *endpoint*. Os bits 7 e 6 são reservados e devem ser desativados. Os bits 5 e 4 definem o uso funcional do *endpoint*: dados (00b), retorno (01b), dados de retorno implícito (10b). o Valor 11b é reservado. Os bits 3 e 2 definem o tipo de sincronização para transferências do tipo *isochronous*, logo são apenas utilizadas se o *endpoint* utilizar esse tipo de transferência, devendo ser configurado como sem sincronização (00b) para outros tipos de transferência. Outras definições possíveis para esses bits são assíncrono (01b), adaptativo (10b) e síncrono (11b). Os bits 1 e 0 definem o tipo de transferência: *control* (00b), *isochronous* (01b), *bulk* (10b) e *interrupt* (11b).

No campo *wMaxPacketSize* é colocado o tamanho máximo do pacote que esse *endpoint* é capaz de tratar quando esta configuração é selecionada. Para *endpoint isochronous* esse valor é utilizado para a alocação de banda por *quadro* ou *microquadro*. Para todos os *endpoints* os bits 10 a 0 são utilizados para especificar o tamanho máximo do pacote em bytes. Para *endpoints high-speed* do tipo *isochronous* e *interrupt* os bits 12 e 11 especificam o número de oportunidades de transações adicionais por *microquadro*. Os bits 15 a 13 são reservados e devem ser desativados.

O campo *bInterval* para *endpoints* do tipo *interrupt* informam o intervalo em número de (*micro*)quadros entre um *polling* e outro. Para *endpoints full/low-speed* esse valor deve estar na faixa de 1 a 255 e para *full-speed* a faixa é de 1 a 16, pois esse número é utilizado na expressão $2^{bInterval-1}$ para calcular o período.

Para *endpoints* do tipo *isochronous* o valor desse campo deve estar no intervalo de 1 a 16. O valor desse campo é utilizado na expressão $2^{bInterval-1}$ para calcular o valor do período. Para *endpoints high-speed* do tipo *bulk/control* OUT o valor desse campo especifica a taxa máxima de NAK do *endpoint*. Nesse caso o valor deve estar na faixa de 1 a 255. Maiores detalhes sobre a descrição desses períodos podem ser encontrados no capítulo 5 da especificação USB 2.0.

3.3.5 Texto

Para representar textos legíveis para o usuário, dispositivos USB utilizam descritores do tipo STRING. Como mencionado anteriormente, esses descritores são opcionais. Caso não se utilize nenhum, todos os índices para texto mencionados nos outros descritores devem ser iguais a zero. Os descritores do tipo STRING utilizam a codificação UNICODE (definido por *The Unicode Standard, Worldwide Character Encoding, Version 3.0*) para criar textos que serão mostrados para os usuários. Os textos de um dispositivo USB podem estar disponíveis em vários idiomas que são definidos pelo LANGID (*Language IDentification*). Quando o *host* solicita um texto ele também especifica o idioma requisitado.

Ao requisitar os descritores do tipo STRING o *host* primeiramente pede o texto com índice igual a zero. Este é um descritor que lista as opções de idioma disponíveis no dispositivo. O descritor STRING de índice 0 tem tamanho variável, pois depende do número de idiomas disponíveis, seus parâmetros pode ser vistos na Tabela 3.9. Os descritores que contém os textos de fato devem ser definidos em sequência, seus parâmetros podem ser vistos na Tabela 3.10.

Tabela 3.9: Descritor de *string* 0 - STRING

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (2A+2)
1	*bDescriptorType*	1	Constante (STRING)
2	*wLANGID[0]*	2	Número
...	*wLANGID[...]*	2	Número
N	*wLANGID[A-1]*	2	Número

Os campos *wLANGID[n]* contém os códigos de 16 bits chamados LANGID de cada idioma suportado pelo dispositivo. A lista completa de códigos pode ser encontrada no *site* do USB-IF.

O campo *bString* é de tamanho variável, pois depende do tamanho do texto. A codificação UNICODE,

Tabela 3.10: Descritor de *string* - STRING

Offset	**Campo**	**Bytes**	**Valor**
0	*bLength*	1	Número (B+2)
1	*bDescriptorType*	1	Constante (STRING)
2	*bString*	B	Número

utilizada para escrever os textos pode ser encontrada em [5].

4 CLASSES

Seguindo a filosofia USB de universalizar e tornar mais simples a instalação de dispositivos no computador, o barramento USB deve oferecer suporte aos mais diversos tipos de dispositivos. Muitos destes possuem características semelhantes, podendo ser divididos em grupos: impressoras, *pen-drives* , mouses, máquinas fotográficas, etc. Assim, faz sentido definir especificações para estes grupos, de forma que eles se comuniquem uniformemente com o *host.* Estes "grupos" são as classes, que oferecem diversas vantagens tanto ao usuário quanto aos desenvolvedores[7].

Os sistemas operacionais modernos já têm inclusos *drivers* para a maior parte das classes definidas pelo USB-IF. Isto libera os fabricantes de terem que criar *drivers* específicos para seus produtos, desde que eles se enquadrem bem em uma ou mais classes. Os consumidores por sua vez são poupados de processos interativos de instalação, utilizando CDs que ficam obsoletos quando se instala um sistema operacional mais novo no computador. Basta conectar o dispositivo a uma porta USB disponível que a instalação se dá de forma automática e transparente para o usuário, com o dispositivo funcionando dentro de alguns segundos.

Quando o dispositivo se enquadra em uma classe mas apresenta características não suportadas por nenhuma das classes padronizadas, o fabricante tem a opção de desenvolver apenas um *driver* complementar, que oferece suporte somente a essas características.

Assim como a especificação USB, cada classe define seus descritores e pedidos específicos, bem como o formato dos dados trocados entre *host* e dispositivo. Afim de possibilitar *drivers* mais apropriados às especificidades de cada tipo de aparelho, as classes podem ser divididas em subclasses com seus próprios descritores e pedidos. Estas subclasses são parte integrante da especificação das classes.

A seguir estão relacionadas algumas das principais classes e suas características. Uma lista completa pode ser obtida em [8].

4.1 HID (*HUMAN INTERFACE DEVICES*)

Embora a especificação da classe HID tenha sido claramente feita com dispositivos de interação humana tais como teclados ou mouses em mente, a verdade é que ela não serve exclusivamente a estes propósitos. Um dispositivo HID deve apenas fornecer ou consumir pequenas quantidades de dados a taxas inconstantes.[3] Como a resposta humana é lentíssima quando comparada à velocidade de um computador moderno, esta definição é suficiente para a maior parte dos equipamentos de interação humana encontrados. Bons exemplos de dispositivos HID sem interface humana são termômetros digitais, leitores de código de barra e multímetros.

Os dispositivos HID têm taxas de transferência de até 800 Bps em *low-speed*, 64 KBps em *full-speed* ou 24,576 MBps em *high-speed* [9]. Estas transferências se dão por meio dos chamados *Reports*. Eles são blocos de dados sem restrição de tamanho, com um formato definido pelo *Report Descriptor*, um descritor específico da classe HID.

4.2 MASS STORAGE

A classe de *mass storage* é direcionada para os dispositivos de armazenamento como *pen-drives*, disquetes ou outro qualquer que tenha a habilidade de armazenar dados e disponibilizar seu posterior acesso.

Esta classe define duas especificações possíveis no que tange o modo de transferência de dados: *Bulk-only* ou *CBI (Control/Bulk/Interrupt)*. Uma destas deve ser implementada para que que o dispositivo seja compatível com a especificação, mas não é necessário que ambos os modos de transferência sejam suportados. Sua diferença básica está explícita nos nomes: enquanto a CBI necessita de acesso a *endpoints* que suportem *Control, Bulk* e *Interrupt* para realizar suas transferências de comandos, dados e *status*, a implementação *Bulk-only* necessita apenas de um *endpoint Bulk*.

4.3 VIDEO

As transferências de vídeo para as quais esta classe foi idealizada são em tempo real (*streaming*). Logo, o modo de transferência de dados usado é o *isochronous*, o único que garante uma determinada banda, negociada na fase de enumeração. É também o único modo que não permite correção de erros na transmissão, uma vez que a requisição do reenvio dos dados faria com que estes chegassem atrasados e a sincronia é condição necessária para que os dados tenham alguma validade. Transações em *Bulk* também são suportadas, mas são menos frequentes.

Diferentes dispositivos de vídeo têm diferentes funcionalidades (uma câmera pode apresentar controle de resolução e de *backlight*, enquanto outra mais simples pode ter resolução fixa, sem *backlight*, por exemplo). Assim, os dispositivos são divididos em blocos modulares, chamados de terminais e unidades. Os terminais são as fontes e sorvedouros de dados (no caso de uma câmera, a fonte seria o mecanismo de captura e o sorvedouro a conexão USB com o *host*). As unidades são representações dos elementos responsáveis por alterar as diferentes características do vídeo (como a resolução ou o *backlight*) sobre as quais o dispositivo têm controle. Os descritores específicos da classe se encarregam de listar os terminais, as unidades e as conexões entre eles, criando uma descrição em *software* do *hardware* do dispositivo USB conectado ao *host*.

4.4 AUDIO

A classe de áudio é bastante similar à classe de vídeo: ambas foram projetadas para trabalhar com grandes quantidades de dados sendo transmitidos com severos requerimentos de tempo mas tolerância considerável a erros. A diferença básica é que no caso do vídeo, a quantidade de dados é muito maior. Outra diferença importante é que a especificação da classe de áudio não suporta transferências *Bulk*.

Por se tratar da classe utilizada no presente projeto, uma maior atenção é devida a esta classe. Portanto, em vez de discorrer rapidamente sobre suas características aqui, foi reservada a ela um capítulo.

4.5 HUB

Esta classe é bastante particular em vários aspectos. O *hub*, como visto no capítulo 2, provém e gerencia as portas utilizadas pelo PC para se comunicar com os dispositivos a ele conectados, inclusive outros *hubs*. Sua especificação não é um documento separado, mas está inclusa na própria especificação da USB.

5 AUDIO CLASS

5.1 ABRANGÊNCIA

> "'A especificação para a classe de dispositivos de áudio se aplica a todos os aparelhos ou funções embutidas em aparelhos multifuncionais que sejam usados para manipular áudio, voz e funcionalidades relacionadas ao som. Isso inclui tanto dados de áudio (analógicos ou digitais) quanto funcionalidades usadas para controlar diretamente o ambiente de áudio, tal como controle de volume ou de tom. A classe de dispositivos de áudio não inclui funcionalidade para operar mecanismos de transporte relacionados à reprodução do áudio tais como *decks* de fita ou controles de *drives* de CD-ROM.'"[10]

Não analisaremos as características da classe de áudio no que diz respeito ao transporte e tratamento de *streams* MIDI, uma vez que estas extrapolam os objetivos do presente estudo.

5.2 CARACTERÍSTICAS

5.2.1 Interfaces

Ao se criar a especificação USB da classe áudio, considerou-se que em diversos dispositivos, o áudio é apenas uma de diversas funções. Um bom exemplo é o *drive* de CD-ROM, que pode comportar áudio, vídeo e armazenamento de dados entre outras funções. Por tanto, convencionou-se que a funcionalidade de áudio esteja no nível de interface na hierarquia do funcionamento dos dispositivos, como pode ser visto na seção 2.1. Logo, o áudio em um dispositivo USB é tratado como um conjunto de interfaces relacionadas entre si, que juntas permitem o funcionamento do dispositivo.

Para permitir este arranjo, as interfaces de áudio foram divididas em dois tipos: interfaces AudioControl (AC) e AudioStreaming (AS). As interfaces AudioControl são usadas para acessar os controles das funções ativas, tal como variar o volume ou o tom de um sinal sendo reproduzido por um dispositivo USB. Já as interfaces AudioStreaming são usadas para transportar o sinal de áudio propriamente dito. Cada interface

possui diversas características e é apropriadamente detalhada em descritores específicos da classe de áudio.

Um conjunto de interfaces usado para gerenciar uma funcionalidade de áudio do dispositivo é chamado de *Audio Interface Class* (AIC). Cada AIC deve ter apenas uma interface AudioControl, sendo esta a condição necessária para que o dispositivo seja considerado como compatível com a especificação de áudio. Apesar disso, a vasta maioria dos dispositivos existentes implementam uma ou mais interfaces AudioStreaming em sua AIC, possibilitando uma troca de sinal de áudio com o PC. Um mesmo dispositivo pode ter várias AICs funcionando simultanea e independentemente.

5.2.2 Sincronia

Sistemas de áudio em tempo real tais como dispositivos USB de áudio são extremamente dependentes de sincronia. De fato, um atraso no envio de informação poderia fazer com que esta se tornasse completamente inútil para determinados equipamentos, uma vez que a distorção de sua reprodução ou gravação extemporânea seria tão ou mais prejudicial para a qualidade do sinal do que sua simples omissão.

Não menos importante é a sincronia entre diferentes canais de um mesmo sinal, principalmente quando tratamos de áudio 3D. Por exemplo, o presente projeto tem por objetivo possibilitar a estimação da direção de chegada de um sinal de voz humana captado simultâneamente por 8 microfones. Para tal objetivo, as relações de fase do sinal são tratadas matematicamente pelo PC que funciona de *host* para a comunicação USB desenvolvida. Tal estimação seria inválida caso houvesse um erro nas informações de fase do sinal, ou seja, se os canais estivessem dessincronizados entre si.

Para tratar dos problemas de sincronia entre canais de um mesmo sinal, a especificação exige que cada interface AudioStreaming de um AIC declare em seus descritores seu atraso (δ) interno, em ms. Além disso, os canais devem seguir sempre uma mesma ordem dentro de um quadro, de forma que possa ser garantido que a primeira amostra gravada por um dispositivo após o início do envio do quadro (α) seja a primeira amostra enviada no quadro $(\alpha + \delta)$ e assim por diante (no caso de uma comunicação *full-speed*, com um novo quadro a cada milissegundo).

Os *endpoints* utilizados para o transporte de áudio segundo a especificação desta classe são do tipo *isochronous*. Para tais *endpoints*, a especificação USB define três tipos de possíveis sincronias. Elas serão

brevemente descritas nesta seção.

- **Assíncrono**: *endpoints* assíncronos produzem ou consomem dados em uma frequência fixa determinada por um CLOCK externo à USB ou um CLOCK interno ao dispositivo. Neste modo, não se pode sincronizar usando um início de quadro (SOF) ou outro sinal do barramento USB.

- **Síncrono**: neste modo, os *endpoints* podem ser controlados através da sincronização com o SOF. Eles podem fazer com que sua frequência de amostragem se baseie no CLOCK de 1 kHz do SOF ou controlar a taxa de geração do SOF de modo que suas taxas sejam a mesma.

- **Adaptativo**: no modo adaptativo, os *endpoints isochronous* de áudio podem operar em qualquer frequência de um determinado intervalo. Dispositivos operando neste modo devem implementar processos internos que os permita adequar sua taxa à imposta a sua interface.

5.2.3 Topologia Funcional

Na especificação da classe de áudio USB, foi considerada a grande diversidade de recursos existentes que podem fazer parte de um sistema de áudio. De forma a caracterizar adequadamente tais recursos, o *firmware* de cada dispositivo contém uma descrição em blocos funcionais de seus recursos. Estes blocos funcionais têm suas conexões e características explícitas em descritores apropriados, montando um circuito lógico equivalente à implementação real no *hardware* do dispositivo.

Estes blocos são divididos em duas categorias, a saber: terminais e unidades. Embora não seja obrigatório que um AIC tenha nenhuma unidade, todo AIC deve ter pelo menos dois terminais para que seja um dispositivo funcional de áudio.

Os terminais e unidades comunicam-se entre si através de pinos de entrada e saída. Qualquer pino de entrada ou saída pode conduzir apenas um *cluster* de canais de áudio. Estes *clusters* são compostos por *m* canais de áudio independentes, *m* variando entre 1 e 255. Todos os canais deste *cluster* dividem as mesmas características entre si, tal como taxa de amostragem, resolução da amostra e formato da amostra (PCM, MPEG1, etc). Apesar disso, tais canais são completamente independentes.

5.2.3.1 Terminais

Terminais são os pontos onde o dispositivo troca seus dados de áudio com o mundo exterior. Existem dois tipos de terminais: os de entrada e os de saída. Os terminais de entrada recebem o sinal de áudio do dispositivo. Bons exemplos deste tipo de terminal são um microfone, no caso de um dispositivo que fornece áudio ao *host*, ou um *endpoint* USB, no caso de um dispositivo que reproduz áudio enviado pelo *host*. Já os terminais de saída fazem o contrário, eliminando o sinal do dispositivo. Bons exemplos são caixas de som ou *endpoints* USB. Embora um dispositivo de áudio USB deva ter ao menos um terminal de entrada e outro de saída, não há restrições para terminais extras.

(a) Terminal de Entrada (b) Terminal de Saída

Figura 5.1: Símbolos dos terminais

Já as unidades são as representações lógicas das funções de tratamento de um sinal de áudio. A especificação da classe define cinco tipos de unidades: Unidade de Mixagem, Unidade Seletora, Unidade de Características, Unidade de Processamento e Unidade de Extensão.

5.2.3.2 Unidade de Mixagem (MU)

A MU é um bloco com n canais de entrada dsitribuídos em um ou mais pinos e m canais de saída em um único pino. Ela tem a capacidade de de misturar estes canais de qualquer forma, podendo ainda estabelecer algumas limitações, como conexões permanentes que independem de controle do *host* e conexões impossíveis.

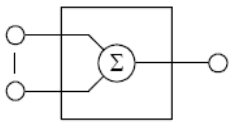


Figura 5.2: Símbolo da Unidade de Mixagem

5.2.3.3 Unidade Seletora (SU)

A unidade seletora é um bloco funcional com n entradas e uma saída. Sua função é selecionar o sinal de apenas um dos n pinos de entrada e levá-lo ao pino de saída. Todos os pinos de entrada devem ter o mesmo número de canais em seu sinal.

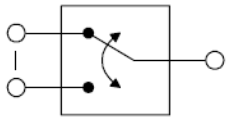


Figura 5.3: Símbolo da Unidade Seletora

5.2.3.4 Unidade de Características (FU)

A FU é uma unidade básica de processamento multicanal. Ela tem controle independente sobre cada canal e simultâneo sobre todos eles, podendo atuar sobre as seguintes características:

- Volume
- Controle de "mudo"
- Controle de tom (graves, médios e agudos)
- Equalizador gráfico
- Controle automático de ganho
- Atraso (*delay*)
- Função *Bass Boost*
- Controle de *Loudness*

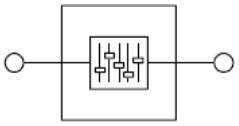


Figura 5.4: Símbolo da Unidade de Características

5.2.3.5 Unidade de Processamento (PU)

Esta unidade é um bloco funcional com múltiplas entradas e uma única saída, oferecendo funções mais avançadas, embora comuns, de processamento de sinais. Cada uma destas funções dá origem a um tipo específico de PU, sendo ao todo seis:

- PU *Up/Down Mix*
- PU *Dolby Prologic*
- PU *3D Stereo Extender*
- PU de Reverberação
- *Chorus* PU
- PU Compressor de Banda Dinâmica

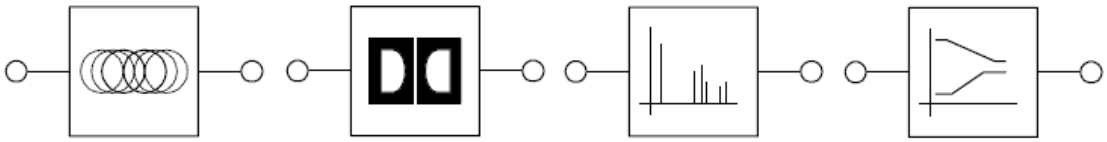


Figura 5.5: Símbolos das PU *3D Stereo*, *Dolby Prologic*, *Chorus* e Compressor de Banda Dinâmica

5.2.3.6 Unidade de Extensão (XU)

A unidade de extensão (XU) é a solução para permitir que os fabricantes possam continuar a incluir novas funcionalidades em seus aparelhos sem que seja necessária uma nova versão da *Audio Class* para isso. As XU podem ter várias entradas, mas apenas um pino de saída.

Embora um *driver* de áudio genérico não possa indentificar ou manipular a funcionalidade contida em uma XU, ele terá ciência de sua presença.

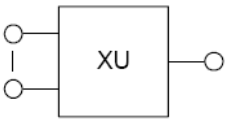


Figura 5.6: Símbolo da Unidade de Extensão

5.3 DESCRITORES

Além dos descritores padrões definidos pela especificação USB, a especificação da classe de áudio define uma série de descritores específicos da classe, bem como define alguns valores padrões para os descritores de uso geral. Estes descritores têm a função de permitir que o *host* possa ter uma compreensão total sobre o *hardware*, de forma a funcionar corretamente e poder utilizar completamente seus recursos.

Entre os descritores que não possuem uma versão específica, estão o descritor de dispositivo e o descritor de configuração. Isto porque as funcionalidades de áudio pertencem à camada da interface, e não à de dispositivo. Um dispositivo de áudio deve portanto usar os descritores padrões de dispositivo e configuração, mas preenchendo os campos que identificam a classe, subclasse e protocolo do dispositivo com 0, o que fará com que o *software* de enumeração do *host* vá até a camada de interface para descobrir estes atributos.

Os descritores específicos da classe de áudio estão divididos em quatro grupos: descritores da interface AudioControl, descritores do *endpoint* AudioControl, descritores da interface AudioStreaming e descritores do *endpoint* AudioStreaming. Como a interface AudioControl usa o *endpoint* 0, não há nenhum descritor padrão para este *endpoint*, seja ele específico da classe ou padrão.

5.3.1 Descritores da interface AudioControl (AC)

Os descritores desta interface contém toda a informação necessária para caracterizar completamente a função de áudio correspondente. O descritor padrão da interface AC descreve a própria interface, conforme descrito no padrão USB. Já o descritor da interface AC específico da classe de áudio é formado por uma concatenação de diversos "‘sub-descritores"’ que detalham o funcionamento interno e as funcionalidades contidas na interface.

O primeiro dos "‘sub-descritores"’ é um cabeçalho que informa o tamanho completo do descritor da interface AC específica, bem como lista todas as interfaces AS e MIDIStreaming controladas por esta interface AC no dispositivo.

O cabeçalho é seguido pelos descritores de todos os terminais e unidades presentes no dispositivo. Cada unidade ou terminal recebe um número de identificação (ID) único de um byte em seu descritor, o

Tabela 5.1: Descritor de cabeçalho da Interface AC

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (8+n)
1	*bDescriptorType*	1	Constante (CS_INTERFACE)
2	*bDescriptorSubtype*	1	Constante (HEADER)
3	*bcdADC*	2	BCD
5	*wTotalLength*	2	Número
7	*bInCollection*	1	Número
8	*baInterfaceNr(1)*	1	Número
...	...	...	...
8+(n-1)	*baInterfaceNr(n)*	1	Número

que limita o número de unidades e terminais endereçáveis por uma interface AC a 255, uma vez que o valor 0 é reservado para ID indefinida. Não importa a ordem em que estes descritores estejam, uma vez que as relações físicas entre as unidades e terminais são especificadas dentro de cada descritor através dos IDs.

Há um tipo de descritor para cada tipo de terminal ou unidade. Há ainda um último tipo de descritor, o descritor de interface associada, que indica a relação entre um terminal ou unidade e uma interface externa à função de áudio. Este descritor deve seguir imediatamente o descritor da unidade ou terminal ao qual se refere.

5.3.1.1 Terminais

Os descritores dos terminais de entrada e saída têm muitas similaridades, dada sua natureza complementar. Entretanto, o do terminal de entrada possui mais campos, pois representa a entrada de um sinal desconhecido no sistema, enquanto as características do sinal de saída já podem ser derivadas das do terminal de entrada e as unidades percorridas.

Ambos os terminais possuem campos caracterizando o tipo de terminal de acordo com uma lista que consta do anexo A da especificação. Assim, o terminal de entrada pode ser um *endpoint isochrounous* ou um arranjo de microfone, entre outras possibilidades, e o terminal de saída pode ser um conjunto de alto-falantes ou um *endpoint isochoronous*, por exemplo. Ambos os descritores também compartilham um campo que indicam qual o terminal complementar associado a ele (um terminal de saída para o de entrada e vice-versa), além de diversos campos que identificam o descritor e o terminal, como a ID e uma string.

Tabela 5.2: Descritor de terminal de entrada

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (12)
1	*bDescriptorType*	1	Constante (CS_INTERFACE)
2	*bDescriptorSubtype*	1	Constante (INPUT_TERMINAL)
3	*bTerminalID*	1	Constante
4	*wTerminalType*	2	Constante
6	*bAssocTerminal*	1	Constante
7	*bNrChannels*	1	Número
8	*wChannelConfig*	2	Bitmap
10	*iChannelNames*	1	Índice
11	*iTerminal*	1	Índice

Nos campos específicos do terminal de entrada, temos um campo para o número de canais presentes no sinal, bem como sua configuração espacial e seus nomes. O fato do campo que define o número de canais ter apenas um byte efetivamente limita a 255 o número de canais por agrupamento nesta classe. A configuração espacial é um campo de dois bytes que dá suporte a doze posições pré-definidas, como centro, *surround* esquerdo ou canal direito, sendo ideal para aplicações de som em que a posição de um canal deve ser observada, como um *home theater*. Para utilizar este campo, devemos habilitar os bits correspondentes aos canais existentes no sinal, multiplexando estes canais obrigatóriamente na ordem dos bits correspondentes. Caso um ou mais canais do terminal de entrada não estejam entre as posições pré-definidas, não se deve habilitar nenhum bit para eles, eles devem simplesmente ser multiplexados após o último canal com a posição especificada.

Tabela 5.3: Descritor de terminal de saída

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (12)
1	*bDescriptorType*	1	Constante (CS_INTERFACE)
2	*bDescriptorSubtype*	1	Constante (INPUT_TERMINAL)
3	*bTerminalID*	1	Constante
4	*wTerminalType*	2	Constante
6	*bAssocTerminal*	1	Constante
7	*bSourceID*	1	Constante
8	*iTerminal*	1	Índice

Nos campos específicos do terminal de saída, é digno de nota apenas o campo *bSourceID*, que está presente em qualquer descritor de unidade e informa que terminal ou unidade está conectado à entrada

deste, sendo assim o responsável pela identificação da topologia da função de áudio.

5.3.1.2 Unidades

Os descritores das unidades possuem todas as informações necessárias para caracterizá-las, permitindo que o *host* tenha uma visão completa de suas capacidades de processamento de sinal, e possa enviar pedidos válidos para efetuar suas funções.

A unidade seletora, por exemplo, tem campos em seu descritor para informar quantos pinos de entrada ela possui, mais campos para informar qual o último bloco funcional conectado a cada um destes pinos. O descritor da unidade de mixagem possui todas estas informações e um campo para controlar a posição no espaço dos canais, como no terminal de entrada. Tem ainda um campo para indicar quais dos controles de mixagem são programáveis.

O descritor da Unidade de Características indica para cada canal quais controles próprios desta unidade são suportados. Já a unidade processadora, dada suas múltiplas naturezas, tem um descritor comum que funciona como um cabeçalho e um descritor para cada uma das funções que ela pode implementar, como unidade *Dolby Prologic* ou de reverberação. O cabeçalho traz todas as informações básicas sobre a unidade, como que tipo de função ela implementa, seu número de pinos de entrada, de onde vem cada pino, o número de canais no pino de saída e sua configuração espacial. Já os descritores específicos trazem as informações relevantes para o uso de cada uma de suas funções. Finalmente, a unidade de extensão possui os mesmos campos que o cabeçalho da PU, à exceção do campo que indica qual função esta implementa, que é substituído pela informação de um código do fabricante para a unidade.

5.3.2 Descritores da interface AudioStreaming (AS)

O descritor de interface padrão desta interface é idêntico ao descrito na especificação USB. Já o descritor de interface AS específico desta classe traz três informações novas: o ID do terminal ao qual esta interface está ligada, o atraso em número de quadros introduzido pelo circuito do dispositivo e o formato do sinal de áudio portado por esta interface.

A informação sobre o formato de áudio é particularmente importante, pois é ela quem informa ao *host*

Tabela 5.4: Descritor de Interface AS específico da classe de áudio

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (7)
1	*bDescriptorType*	1	Constante (CS_INTERFACE)
2	*bDescriptorSubtype*	1	Constante (AS_GENERAL)
3	*bTerminalLink*	1	Constante
4	*bDelay*	1	Número
5	*wFormatTag*	1	Número

em qual formato deve estar o sinal de áudio na comunicação com o *endpoint*. A especificação define inúmeros tipos de formato em um documento separado, *USB Data Formats*,que é considerado parte da especificação. Estes formatos são tão variados entre si quanto o MPEG2 7.1 e o PCM mono de 8 bits. Cada formato tem um descritor específico que deve seguir o descritor da interface AS. Ele traz para o *host* informações específicas do formato de áudio utilizado pelo *endpoint*. Este descritor também é descrito no documento *USB Data Formats*.

Certos formatos podem precisar de informações ainda mais específicas sobre a comunicação do USB com a conexão que a interface representa. Para estes formatos, existe um descritor específico do formato, cujos detalhes bem como indicação dos formatos de áudio em que é necessário estão também presentes no documento *USB Data Formats*.

5.3.3 Descritores do *endpoint* AudioStreaming (AS)

Estes descritores são responsáveis por fornecer tanto informações de como os sinais de áudio se relacionam com a função de áudio como informações específicas do *endpoint* utilizado. A classe de áudio é a única classe a alterar o descritor padrão de endpoint, expandindo-o em 2 campos. Estes campos são o *bRefresh*, que deve ser igual a 0, e o *bSynchAddress*, que informa o endereço do *endpoint* utilizado para sincronização caso o *endpoint* assim exija. Caso o *endpoint* não exija, este campo também deve ser igual a 0.

Já o descritor do *endpoint isochronous* de áudio específico da classe possui três campos com informações relevantes, sendo os outros constantes usados apenas para identificar o descritor e seu tamanho. O campo *bmAttributes* informa se o *endpoint* suporta ou não controles de taxa de amostragem e tom, bem

Tabela 5.5: Descritor de *endpoint isochronous* expandido

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (9)
1	*bDescriptorType*	1	Constante (ENDPOINT)
2	*bEndpointAddress*	1	*Endpoint*
3	*bmAttributes*	1	Bit Mapeado
4	*wMaxPacketSize*	2	Número
6	*bInterval*	1	Número
7	*bRefresh*	1	Número (0)
8	*bSynchAddress*	1	*Endpoint*

como se este endpoint apenas suporta pacotes do tamanho máximo por ele informado no descritor anterior. Já o campo *bLockedDelayUnits* indica a unidade (milissegundos ou amostras PCM decodificadas) utilizada pelo campo seguinte, *wLockDelay* para especificar o tempo que o *endpoint* gasta para ajustar-se à fonte de CLOCK. Estes dois campos só são usados em *endpoints* síncronos ou adaptativos, pois nos assíncronos o CLOCK é gerado no próprio dispositivo e é independente. Neste caso, estes campos devem ser iguais a 0.

Tabela 5.6: Descritor de *endpoint isochronous* específico da classe de áudio

Offset	Campo	Bytes	Valor
0	*bLength*	1	Número (9)
1	*bDescriptorType*	1	Constante (CS_ENDPOINT)
2	*bDescriptorSubType*	1	Constante (EP_GENERAL)
3	*bmAttributes*	1	Bit Mapeado
4	*bLockDelayUnits*	1	Número
5	*wLockDelay*	n	Número (depende de *bLockDelayUnits*)

Há um último descritor, de estrutura idêntica ao do descritor padrão do *endpoint* AS, chamado de descritor padrão de sincronia do *endpoint* AS *isochronous* de áudio. Ele só é utilizado quando há um *endpoint* OUT adaptativo ou um *endpoint* IN assíncrono, e serve para fornecer informações sobre o dispositivo de ressincronização. Este descritor dá um valor ao campo *bRefresh* deixado em 0 no descritor padrão do *endpoint*. Este valor indica a taxa em que o *endpoint* fornece informações sobre a sincronia. Esta taxa deve ser uma potência de 2, sendo que só o expoente é retornado e este deve ser um valor entre 1 (2 ms) e 9 (512 ms).

5.4 REQUESTS

A classe de áudio, além de suportar todos os pedidos padrões requeridos pelo padrão USB, utiliza pedidos específicos da classe. A maioria desses pedidos específicos são usados para ler ou configurar controles de áudio. Neste caso eles se encaixam em um de dois grupos distintos.

Os pedidos para AudioControl manipulam atributos dos controles individuais de cada unidade. Neles estão contidas informações suficientes para se distinguir o destinatário do pedido.

Os pedidos para *endpoints* AudioStreaming controlam parâmetros das interfaces ou *endpoints*. Eles são direcionados para as interfaces ou *endpoint isochronous* ao qual o controle pertence.

Mais outros dois pedidos específicos são suportados pela classe de aúdio. Pedido de memória (*Memory Request*), podem ser direcionados a qualquer entidade de uma função de áudio para controle genérico de seus atributos, através de uma interface mapeada na memória. Já o pedido de *status* (*Get Status*) é um pedido genérico pra uma entidade da interface AudioControl ou da interface AudioStreaming que não manipula nenhum controle.

A principio todos os pedidos específicos são opcionais e, como já especificado, todos os pedidos não suportados devem ser respondidos com um STALL. Porém se um pedido do tipo SET é suportado, o pedido do tipo GET associado também deve ser suportado.

Os pedidos do tipo SET são utilizados para configurar um parâmetro de um controle dentro de uma entidade da função de áudio, como ajustar um controle de volume, ou ainda de um espaço de memória associado a uma entidade. Já os do tipo GET servem para verificar um destes parâmetros, como descobrir qual o volume máximo possível ou qual a resolução do controlador de volume. Os pedidos podem ser sobre um parâmetro atual, o mínimo de um parâmetro, o máximo de um parâmetro, um parâmetro de resolução ou ainda de um parâmetro definido na memória.

- *bmRequestType*: 00100001b (interfaces de áudio), 00100010b (*endpoint*) no caso do SET. Para o GET, 10100001b e 10100010b respectivamente.

- *bRequest*: SET_CUR (parâmetro atual), SET_MIN (parâmetro mínimo), SET_MAX (parâmetro máximo), SET_RES (parâmetro de resolução), SET_MEM (parâmetro definido na

memória).

• *wValue*: o valor desse campo depende do valor em *wIndex* e depende da entidade à qual o pedido é endereçada. Na maioria dos casos esse campo contém o seletor de controle no MSB.

• *wIndex*: especifica a interface ou *endpoint* endereçados no LSB e a ID da entidade ou zero no MSB

• *wLength*: especifica o tamanho do bloco de dados.

• Dados: bloco de dados com o parâmetro desejado. Varia para cada remetente do pedido.

6 ARQUITETURA ARM E AT91SAM7S

6.1 INTRODUÇÃO

O kit de desenvolvimento utilizado, AT91SAM7S256-EK , teve vários bons motivos para ser escolhido. O primeiro é que este é o kit recomendado pelo fabricante do processador ARM, a ATMEL. Além disso, ao contrário da maior parte dos kits para este processador, este kit possui diversas funcionalidades para facilitar o desenvolvimento do *software*: leds e botões para *debug* e implementação de partes interativas do código, interface serial para *debug*, a maior das memórias internas disponíveis em kits e grande número de pinos de I/O. Um último mas não menos importante motivo é que foi conseguida junto à FINATEC uma verba para ser gasta com o kit de desenvolvimento de nosso projeto, e esta verba era suficiente para comprar esta placa. Assim, não tínhamos motivos para comprar uma placa mais barata e menos adequada.

6.2 PRINCIPAIS CARACTERÍSTICAS

O microcontrolador utilizado no presente projeto é o AT91SAM7S, que possui uma unidade lógico-aritmética com arquitetura RISC de 32 bits. O ARM é o líder em MIPS/Watt no mercado, com capacidade para realizar até 49,5 milhões de instruções por segundo no modelo utilizado. O microcontrolador conta com 64 kbytes de memória RAM de alta velocidade e 256 kbytes de memória Flash de alta velocidade, ambas com capacidade de acesso em um único ciclo. O chip possui duas fontes de CLOCK, sendo um circuito RC com uma faixa de operação entre 3 e 20 MHz e um PLL, sendo que o fabricante garante a operação correta do microcontrolador até 55 MHz.

O controlador de interrupções do AT91SAM7S nos permite mascarar cada interrupção individualmente e selecionar entre 8 níveis de prioridade para cada uma. Todos os 32 pinos de E/S que o ARM possui têm a capacidade de ativar interrupções por borda. Estes pinos carregam até 3 sinais multiplexados, controlados pelo *Parallel Input/Output Controller* (PIO), que tem controle também sobre resistores de *pull-up* ou modo de dreno aberto para cada um dos pinos. Inúmeros periféricos ainda compõem o microcontrolador, entre eles:

- Unidade Serial de *debug*
- Timer de intervalos periódicos com contador de intervalos de 12 bits
- Timer de tempo real, que conta em segundos
- Timer/Counter de 16 bits com três canais independentes
- Porta USB 2.0 *full-speed* integrada, utilizando FIFOs de 328 bytes para os *endpoints*
- Duas portas USART com taxas independentes
- Interface SPI
- Conversor ADC de 8 canais com resolução de 10 bits
- Suporte a ICE JTAG para *debug*

Nas seções seguintes um maior destaque será dado a três módulos que tiveram uma participação mais significativa no presente projeto: o Conversor Análogo/Digital (ADC), o Timer/Counter (TC) e o controlador USB (UDP).

6.3 CONVERSOR ANALÓGICO/DIGITAL (ADC)

O AT91SAM7S possui um módulo ADC integrado, que recebe como entradas 8 portas analógicas multiplexadas. Este ADC opera utilizando um Registrador de Aproximações Sucessivas (SAR) de 10 bits, e pode operar com palavras de 10 ou 8 bits. As conversões usam a faixa entre o terra do kit de desenvolvimento e a tensão do pino ADVREF como referência para a conversão. Tanto a duração de uma única conversão quanto o tempo de inicialização do *Sample and Hold* e do SAR são reguláveis por *software*.

Dos 8 pinos de entrada analógica do ADC, 4 são de uso exclusivo deste módulo e 4 estão multiplexados com outras. Para utilizar estes últimos 4 canais é necessário além de programar o ADC para receber seu sinal, configurar também o PIO para tratá-los como entradas analógicas.

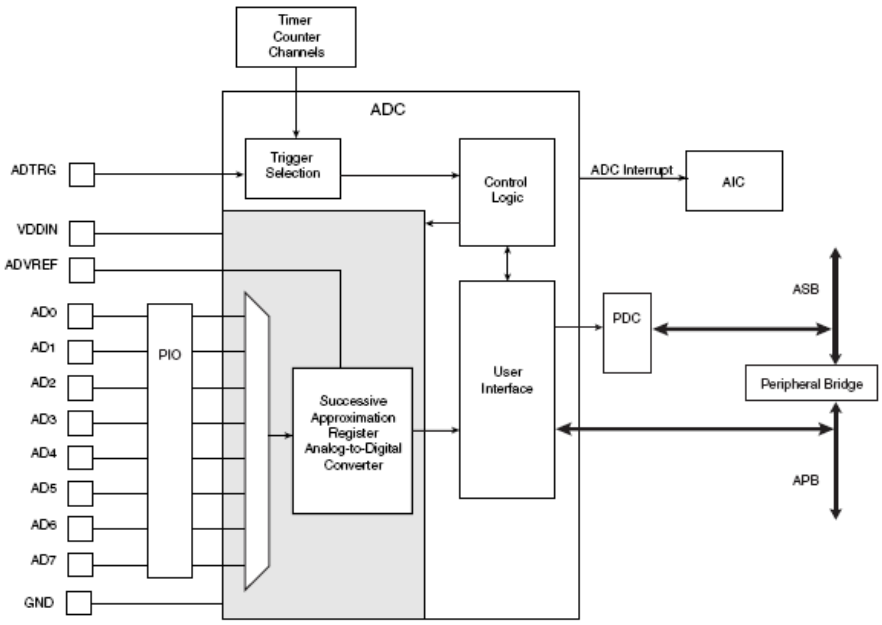


Figura 6.1: Diagrama de Blocos ADC

O ADC pode ser programado para converter a cada ciclo qualquer combinação de entradas, convertendo todas as entradas programadas após receber um único comando de início de conversão. Estas conversões podem ser iniciadas tanto por *software* quanto por um sinal externo, sendo que microcontrolador possibilita o uso de um sinal do módulo Timer/Counter como sinal externo. O sequenciador de conversões então começa a partir do menor canal a converter todos os canais, sempre em ordem crescente. Quando uma amostra é convertida, o ADC pode ser programado para acionar uma interrupção ou apenas habilitar uma *flag* que indica que a conversão acabou. Caso a amostra não seja acessada até o término da próxima conversão, uma *flag* de erro será levantada. Uma operação de DMA (acesso direto à memória) ainda pode ser usada para copiar a amostra. Neste caso, a *flag* que indica que a conversão que terminou serve para gatilhar um pedido de DMA que faz a transferência automaticamente.

6.4 TIMER/COUNTER (TC)

Há 3 canais Timer/Counter de 16 bits idênticos no AT91SAM7S: TC0, TC1 e TC2. Cada um deles pode realizar independentemente uma ampla gama de funções como medição de frequência, contagem de eventos ou intervalos e geração de pulsos. Cada TC pode escolher entre três fontes de CLOCK, além de possuir dois pinos de entrada e saída (TIOA e TIOB) que podem ter suas funções configuradas pelo

usuário. Estes pinos estão multiplexados com outras funções no PIO, e portanto devem ser configurados para o TC antes de serem usados. Cada TC tem também a habilidade de gerar interrupções no processador. Os TCs funcionam com um contador de 16 bits incrementado na borda de subida do CLOCK ou pino de E/S selecionado. Este contador pode ter seu valor lido em tempo real. Quando o contador atinge seu valor máximo e reinicia sua contagem, ele habilita uma *flag*. O contador pode ainda ser reiniciado por um gatilho, que pode ser tanto de *software* quanto de *hardware*. No caso de *hardware*, configura-se um dos pinos de E/S para este fim. Já no caso de *software*, o gatilho pode vir de duas formas: o contador pode ser reiniciado ao alcançar o valor de RC, que pode ser configurado pelo usuário, ou quando o *software* força um sinal de SYNC, o que reinicia simultâneamente todos os TCs.

Existem dois modos distintos de operação para os TCs: o modo de captura, que serve para medir sinais, e o modo de forma de onda, que gera uma onda.

6.4.1 Modo de Captura

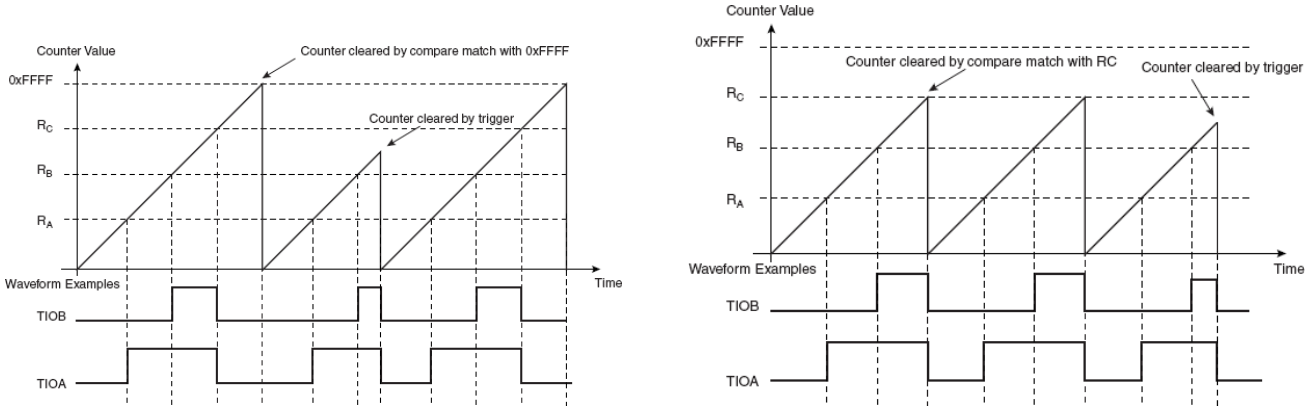
No modo de captura os dois pinos de E/S são configurados pra entrada. Os TCs podem ser programados para copiar o conteúdo do contador para um registrador específico quando ocorre um evento programável em uma das entradas, o que pode fornecer informações sobre a frequência e quantidade das oscilações destas entradas. Uma destas entradas pode ainda ser utilizada como gatilho para reiniciar a contagem.

6.4.2 Modo de forma de onda

Neste modo, TIOA é sempre saída e TIOB pode ser um gatilho externo ou uma saída, sendo que há dois registradores para controlar o comportamento de cada um dos pinos independentemente (RA para o TIOA e RB para o TIOB), no caso do pino estar configurado como saída. Assim, dependendo de o contador estar acima ou abaixo do valor configurado no registrador correspondente e da configuração selecionada, o sinal do pino será alto ou baixo.

Há quatro comportamentos distintos para um TC operando em modo de forma de onda, todos podendo utilizar gatilhos ou não. Estes comportamentos são selecionados através do campo WAVSEL presente em um dos registradores deste módulo. Para WAVSEL = 00, o TC sempre conta de 0 até atingir o valor

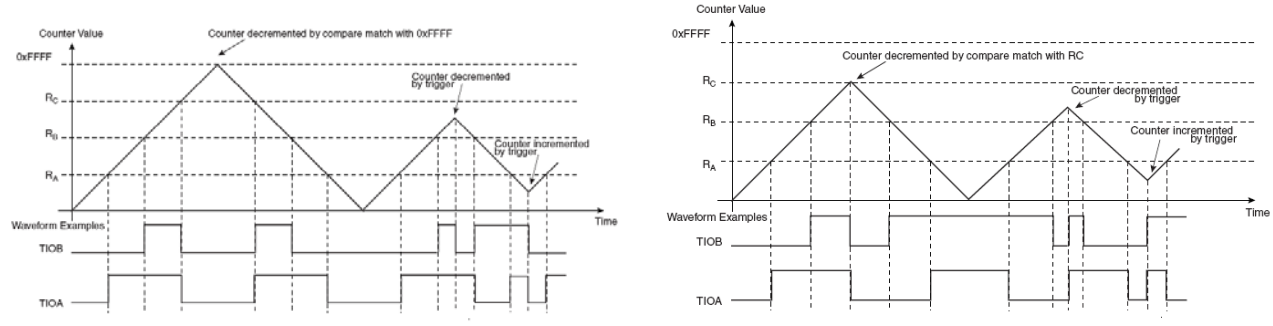
máximo ou ser gatilhado, e então reinicia a contagem como pode ser visto na figura 6.2. O modo WAVSEL = 10 é semelhante, mas em vez do TC incrementar o contador até estourar sua contagem, ele é reiniciado ao alcançar RC.



(a) WAVSEL = 00 (b) WAVSEL = 10

Figura 6.2: Formas de onda de um TC no modo UP

No caso de WAVSEL = 01, o contador do TC também é incrementado até alcançar o valor máximo, mas em vez de ser reiniciado, ele começa a ser decrementado até alcançar 0, onde voltará a ser incrementado. Este modo pode ser visto em 6.3, bem como WAVSEL = 11. Vale notar que um gatilhamento neste modo terá o efeito de mudar o sentido da onda, e não reiniciá-la como nos casos descritos acima. Já em WAVSEL = 11, em vez de mudar de sentido ao alcançar o valor máximo, ele o faz apenas até o valor de RC. Assim, um apanhado dos quatro casos nos mostra que o bit mais significativo de WAVSEL é usado para configurar entre incrementar até RC (1) ou até estourar (0), e o bit menos significativo para configurar como modo de subida e descida (1) ou apenas subida da onda (0).



(a) WAVSEL = 01 (b) WAVSEL = 11

Figura 6.3: Formas de onda de um TC no modo UP/DOWN

6.5 PORTA DE DISPOSITIVO USB (UDP)

Os microcontroladores da família AT91SAM7S a partir do modelo 64 têm um módulo de controle da porta USB chamado de *USB Device Port*, ou simplesmente UDP. O UDP é compatível com a especificação USB V2.0 para dispositivos *full-speed*. O módulo possui quatro *endpoints* que podem ser configurados em um dos 4 tipos de transferências USB. Esses *endpoints* podem estar associados a um ou dois bancos da memória RAM de porta dupla.

Toda a interface do *firmware* com o UDP é feita através de seus registradores, porém podem ser usadas duas das linhas da porta paralela para verificação de V_{bus} e controle do resistor de *pull-up* conectado à linha $D+$. O módulo também possui uma linha de interrupção conectada ao controlador de interrupção do microcontrolador - *Advanced Interrupt Controller* (AIC).

6.5.1 Endpoints

Os quatro *endpoints* são denominados EP0, EP1, EP2 e EP3 respectivamente. EP0 é normalmente utilizado para o canal de controle obrigatório e necessário durante o processo de enumeração. EP1 e EP2 são os únicos que suportam transferências do tipo *isochronous* por serem os únicos a suportarem *buffer* com dois bancos de memória. As características completas se encontram na Tabela 6.1.

Tabela 6.1: Características dos *endpoints* do UDP

Endpoint	Banco Dual	Tamanho Máximo do Endpoint	Tipo
0 (EP0)	Não	8	*Control/Bulk/Interrupt*
1 (EP1)	Sim	64	*Bulk/Iso/Interrupt*
2 (EP2)	Sim	64	*Bulk/Iso/Interrupt*
3 (EP3)	Não	64	*Control/Bulk/Interrupt*

Cada *endpoint* pode ser configurado para fazer um tipo de transferência USB. Em todos os tipos de transferência é suportada a detecção de erros. Outras características suportadas para cada tipo de transferência podem ser conferidas na tabela 6.2.

Tabela 6.2: Características das transferências suportadas pelo UDP

Transferência	Direção	Tamanhos Suportados do Endpoint	Retransmissão
Control	Bidirecional	8, 16, 32, 64	Automática
Isochronous	Unidirecional	64	Não
Interrupt	Unidirecional	8, 16, 32, 64	Sim
Bulk	Unidirecional	8, 16, 32, 64	Sim

6.5.2 Transferência de Dados

Como explorado em capítulos anteriores, as transferências através do barramento USB são feitas de transações. Três tipos de transações são necessárias: SETUP, DATA IN e DATA OUT. Transações do tipo SETUP são apenas utilizadas em transferências do tipo *control*. DATA IN e DATA OUT são utilizada para transferência de dados do dispositivo para o *host* e do *host* para o dispositivo respectivamente.

Para transferências do tipo *isochronous* é obrigatório o uso de dois bancos de memória. Os dois bancos são usados para transferência com atributo *ping-pong*. Nesse tipo de transferência, enquanto um banco é lido/escrito pelo *firmware*, o outro banco é escrito/lido pelo periférico USB para transmissão pelo barramento. Isso garante a taxa de transmissão de 1 MBps. A Figura 6.4 exemplifica uma transferência DATA IN com atributo *ping-pong*.

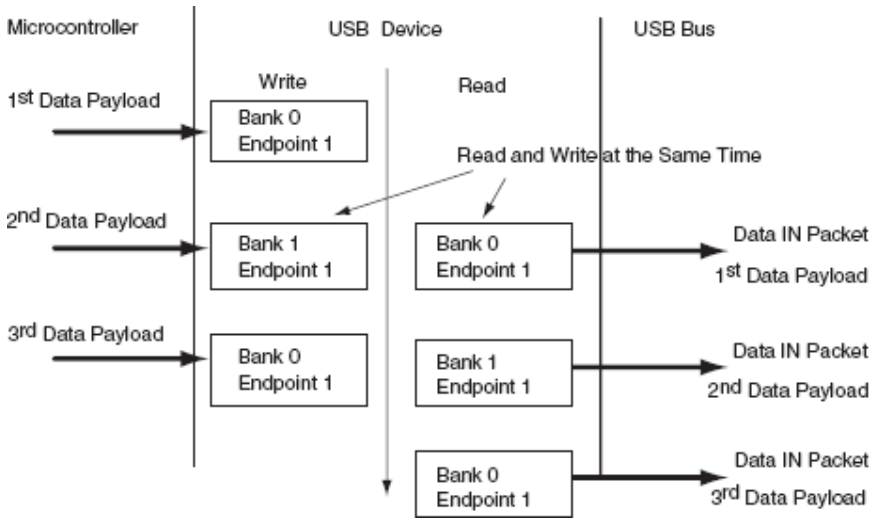


Figura 6.4: Transferência com atributo *ping-pong*

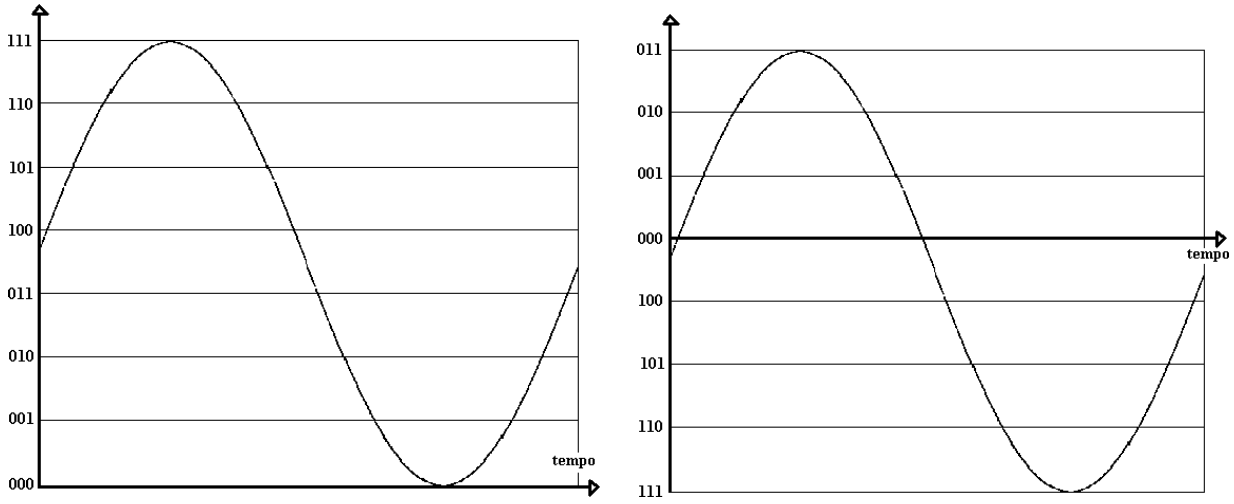
7 IMPLEMENTAÇÃO

7.1 O PROJETO

Ao iniciar o projeto, um dos primeiros passos foi definir como se daria a transmissão dos dados adquiridos pelo ARM para o PC. Como o desenvolvimento de um *device driver* específico para nossa aplicação seria muito trabalhoso, sendo provavelmente inviável dentro do tempo para a conclusão do projeto, buscamos verificar qual classe USB mais se adequaria às nossas necessidades. Não surpreendentemente, descobriu-se que a classe de áudio era perfeita para satisfazer a todos os requisitos do projeto, tendo comunicação de baixa latência, alta velocidade e garantia de banda para entrega de dados. Além disso, sua especificação dá suporte a múltiplos microfones e define uma estrutura multicanal de simples implementação. Mais ainda, esta classe possibilita o uso da própria interface de som do PC, o que permite que seja utilizado *software* já existente para gravar o sinal, como o gravador de som do Windows.

Mas para poder dispensar o desenvolvimento de um *software* específico, foi necessária a adequação do projeto aos padrões da indústria. O ADC presente no kit ARM tem resolução máxima de 10 bits, o que foi definido como o tamanho da palavra usada no sinal. Entretanto, não foi encontrado nenhum *software* de gravação de áudio que suportasse resolução diferente de 8 ou 16 bits. A solução encontrada foi copiar as amostras de 10 bits em palavras de 16 bits. O formato do sinal também teve que ser alterado: o sinal convertido pelo ADC era compostos de valores entre 0 e o máximo da resolução configurada, enquanto o PCM tem como máximos 1 e -1. Mas esta conversão pôde ser realizada com grande facilidade: o PCM interpreta como 1 o maior valor positivo de uma palavra e -1 o maior valor negativo. Além disso, o PCM define que no caso de resoluções entre 8 e 16 bits os bits de sinal estarão nos bits mais significativos da palavra e os que restarem serão deixados em 0. Para transformar uma variável de *unsigned* para *signed*, basta fazer com que seu valor médio seja 0. Como o computador trabalha com complemento de dois, o procedimento para fazer esta conversão é subtrair a amostra a ser convertida de uma palavra com apenas o bit mais significativo igual a 1, ou seja, subtrair de metade do valor máximo da palavra. Esse procedimento está exemplificado na figura 7.1, onde vemos as duas formas de representação para um mesmo sinal. Nesse caso, a representação *signed* pode ser derivada subtraindo-se $100b$ de cada amostra correspondente

unsigned.



(a) Representação *unsigned* (b) Representação *signed*

Figura 7.1: Comparação entre representação *signed* e *unsigned* de uma variável

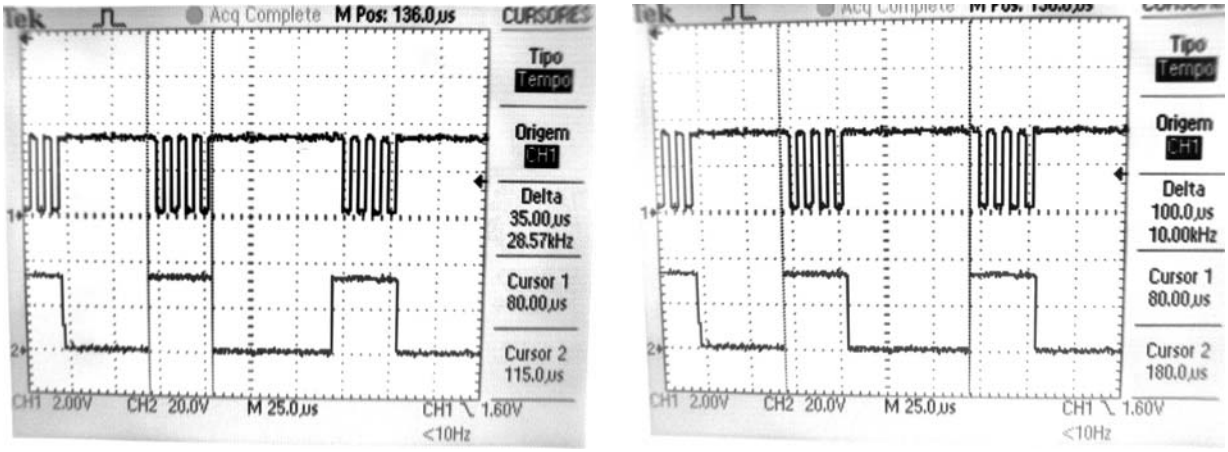
O *software* escolhido para a aquisição do sinal foi o *matlab*, utilizando uma biblioteca pronta chamada *pa_wavrecord*, que pode ser encontrada em [11]. Esta biblioteca oferece três grandes vantagens em relação aos *softwares* convencionais como o gravador de som do *Windows*. A primeira é que ela possui uma taxa de amostragem configurável pelo usuário com precisão de 1 Hz, enquanto os outros programas oferecem apenas algumas poucas opções pré-estabelecidas, entre as quais nunca consta 10 kHz. O segundo motivo é que podemos também configurar o número de canais de gravação e a maioria dos programas só grava nos padrões comerciais (mono, stereo, 5.1). O último motivo é que a biblioteca não retorna um arquivo "*wave*"gravado, mas sim uma matriz de tamanho mxn, onde as amostras estão dispostas nas m linhas de cada uma das n colunas, que representam os diferentes canais. Este formato faz com que seja mais simples tratar as amostras, e o *matlab* tem por padrão funções tanto para reproduzir o áudio disposto nestas matrizes quanto convertê-los em arquivos "*wave*". Infelizmente, quando começaram os testes de aquisição de sinal, descobriu-se que toda a interface multicanal fornecida pelo *pa_wavrecord* e pela classe de áudio não poderia ser utilizada. Por motivos até agora não identificados, só foi possível adquirir os primeiros dois canais do sinal. Inicialmente, desconfiou-se do *driver* de gravação de áudio padrão do *Windows* e descobriu-se que ele realmente só suporta até dois canais. Embora o *firmware* de nosso dispositivo USB estivesse configurado para 8 canais, o *driver* informava que o dispositivo possuía apenas 2 canais. Como o *pa_wavrecord* dá a possibilidade de escolher qual dos *drivers* de gravação do *Windows* deve ser utilizado

para amostrar o sinal, resolveu-se utilizar o *driver* do DirectX. O DirectX é um produto da Microsoft projetado principalmente para otimizar a experiência dos jogos de computador. Ele contém diversos *drivers* de vídeo e áudio, entre outros. Quando este *driver* foi utilizado, verificou-se a correta informação que o dispositivo tinha 8 canais de áudio. Entretanto, embora ele também funcionasse adequadamente para gravar os dois primeiros canais, quando tentou-se gravar qualquer canal a partir do terceiro, uma matriz mxn contendo apenas zeros foi recebida. Ou seja, embora fosse possível amostrar o canal 1 ou o 2 ou ambos, não era possível amostrar nenhuma combinação que contivesse qualquer outro canal.

A solução então encontrada foi multiplexar ainda no ARM os 8 canais do ADC em um único canal de áudio com taxa de amostragem igual a 80 kHz e demultiplexá-la após adquirida pelo PC, sendo reproduzida a 10 kHz. Os testes neste novo módulo apresentaram uma nova limitação: as transferências cuja banda total ultrapassava os 32 kHz apresentavam comportamento estranho, gravando por mais tempo que o programado e ficando ininteligíveis quando reproduzidas. Apenas podia se distinguir um ruído que lembrava uma reprodução acelerada da gravação.

Diversas hipóteses foram levantadas, assim como experimentos a fim de confirmá-las. Considerou-se que o erro poderia ser um problema com o funcionamento do ADC, que não estaria convertendo as amostras com a presteza necessária para suprir a demanda por dados da USB. Foram inseridos então no código do módulo ADC instruções para variar o estado da saída de determinados pinos da porta paralela do microcontrolador, de sorte que quando um grupo de conversões terminasse e acabasse seria possível visualizar com um osciloscópio, bem como a duração de cada uma das oito conversões de um grupo. Mas, como evidenciado pelo resultado nas figuras 7.2 (a) e (b), os tempos eram suficientes. Nestas figuras, o sinal de baixo representa a duração de um grupo de conversões e é levantado quando a rotina de conversões é iniciada e abaixada quando esta termina. Já o sinal de cima muda de alto para baixo toda vez que a conversão de um canal acaba. O tempo necessário para digitalizar uma amostra medido no osciloscópio era de 4 μs, a não ser para a primeira amostra que levava 1 μs a mais para iniciar o *sample and hold*. Isto era um tempo muito abaixo do necessário, com toda a rotina levando 35 μs, ou 35% do tempo entre duas conversões. Verificou-se também que a rotina era iniciada exatamente a cada 100 μs, descartando uma segunda hipótese que considerava que talvez a taxa de amostragem estivesse incorreta.

Uma nova hipótese foi levantada de que o problema talvez fosse causado porque a cópia das amostras



(a) Duração de um grupo de conversões

(b) Tempo entre grupos de conversões

Figura 7.2: Temporização do módulo ADC

no *buffer* para a FIFO da UDP estivesse demorando demais, dessincronizando o envio dos dados. Novas alterações no *firmware* foram feitas de modo a possibilitar a medição no osciloscópio da duração desta etapa, bem como sua relação com a amostragem do ADC. Ao observar as ondas, verificou-se que o tempo total de envio era de 60 μs e era iniciado logo após o fim de dez grupos de amostragem do ADC, ou seja, 1 ms. Desta forma, não havia também atraso nenhum causado devido a uma demora excessiva no envio.

Um novo teste foi então proposto. Após cada amostra feita pelo ADC, escreveríamos sobre a variável que guardava seu resultado um número i, que seria então enviado no lugar da amostra. Este número seria então incrementado para a próxima amostra, gerando uma sequência de -16384 a 16383, já que a variável era um *signed* de 16 bits. Este teste tinha natureza apenas eliminatória. Caso a transmissão funcionasse perfeitamente, o problema deveria estar no módulo ADC, pois se suas amostras sobrescritas não eram corrompidas no envio, suas amostras também não o seriam. O problema poderia estar então em qualquer lugar.

O resultado de uma destas transferências pode ser vista na figura 7.3. Lá, nota-se que após cada grupo de 16 números recebidos corretamente perdia-se uma sequência de números. Esta sequência perdida alternava de tamanho, podendo ser de tamanho 16 até 112.

Verificou-se assim que para taxas de transferência acima de 32 kHz menos da metade da informação enviada era recebida. Além da regularidade do erro, outro fator muito intrigante era que todas as sequências de dados perdidos ou enviados tinha um tamanho múltiplo de 16 e, portanto, múltiplo do número de

	A	B	C	D	E	F	G
1	112	144	272	304	432	464	592
2	113	145	273	305	433	465	593
3	114	146	274	306	434	466	594
4	115	147	275	307	435	467	595
5	116	148	276	308	436	468	596
6	117	149	277	309	437	469	597
7	118	150	278	310	438	470	598
8	119	151	279	311	439	471	599
9	120	152	280	312	440	472	600
10	121	153	281	313	441	473	601
11	122	154	282	314	442	474	602
12	123	155	283	315	443	475	603
13	124	156	284	316	444	476	604
14	125	157	285	317	445	477	605
15	126	158	286	318	446	478	606
16	127	159	287	319	447	479	607
17	(+16)	(+112)	(+16)	(+112)	(+16)	(+112)	(...)

Figura 7.3: Resultados de teste de envio sequencial de números a 80 ksamples/s

microfones amostrados.

Tendo exaurido as possibilidades consideradas e com o tempo para finalizar o projeto já chegando a seu término, os testes foram abandonados em prol da conclusão da monografia. Então, durante a escrita do capítulo sobre a UDP, verificou-se a existência de informações aparentemente conflitantes no *datasheet* utilizado [12]: o *endpoint isochronous* era anunciado como tendo 64 bytes de tamanho em uma tabela, mas capacidade para até 1023 bytes em outra. Isto levou à procura de uma nova versão do documento. Foi encontrada uma atualização de novembro de 2006 [13], que trazia a informação de forma mais clara. A capacidade suportada pela UDP do AT91SAM7S é de fato de apenas 64 bytes, sendo 1023 o máximo definido pela especificação USB 2.0 para um dispositivo *full-speed*, tal como a UDP.

Como o código era baseado no pressuposto de que o *endpoint* teria tamanho 1023, essa possibilidade não havia sido levada em conta. O que ocorria é que o código ajustava o tamanho do *buffer* conforme a taxa de envio utilizada, tendo sempre o tamanho do pacote a ser enviado, uma vez que não se pretendia enviar com este código informações a uma taxa de mais que 1023 bytes por segundo. Para uma amostragem de 32 kHz, o *buffer* tinha $2\frac{bytes}{amostra} \times 32\frac{amostras}{ms} = 64\frac{bytes}{ms}$, o máximo suportado pelo *endpoint* para um único pacote.

Embora haja diversas possibilidades para se sobrepor este obstáculo, como enviar mais de um pacote por milissegundo ou utilizar um *endpoint* extra na transmissão, o tempo para o desenvolvimento do projeto já havia chegado ao fim quando todas estas conclusões foram feitas, deixando a tarefa da solução deste

problema para uma continuação deste trabalho.

7.2 RESULTADOS

Os resultados obtidos neste projeto restringem-se a gravações de até 4 canais com 8 kHz por canal. Não foi possível alcançar uma taxa maior que 32 kHz graças aos contratempos descritos na seção 7.1. Também não foi possível amostrar sinais com mais de 4 canais, pois a montagem de uma das placas necessárias para esta tarefa não foi concluída em tempo hábil.

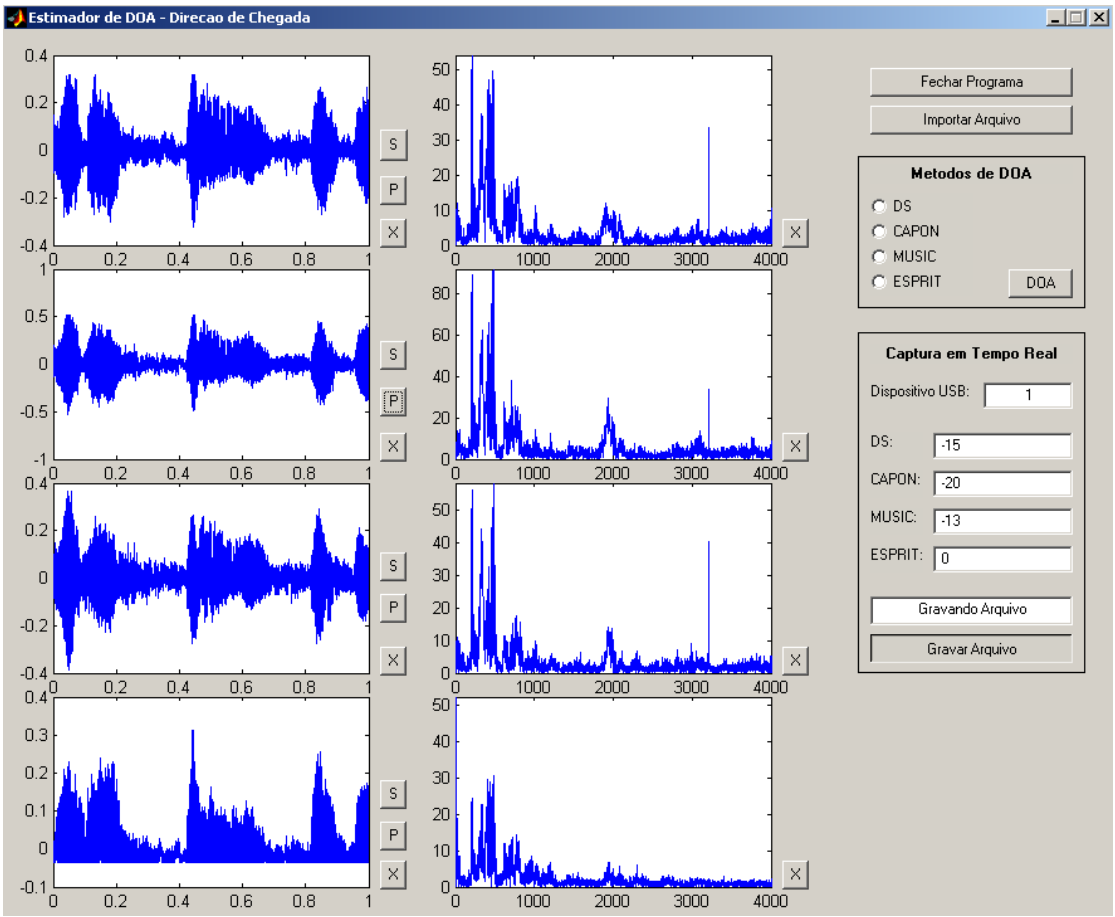


Figura 7.4: Interface desenvolvida para utilização do ambiente

Em conjunto com o grupo de *hardware* uma interface gráfica foi desenvolvida para controlar e apresentar os resultados do processo de captação e análise das amostras. Esta interface controla as diversas rotinas do *matlab* que incluem a gravação, tratamento e estimação de direção de chegada do sinal. Como podemos ver pela figura 7.4, além da estimação da direção de chegada pelos diferentes métodos (DS, CA-

PON, MUSIC e SPRIT), também são apresentados a forma de onda e a FFT de cada canal. A interface fornece ainda o recurso da reprodução independente dos canais de áudio. As amostras da figura 7.4 foram capturadas utilizando o dispositivo desenvolvido neste projeto.

Devido a imprecisões na calibragem do *hardware*, os sinais capturados por diferentes microfones apresentam diferentes componentes DC. Para resolver este problema, um filtro digital foi implementado. Este filtro calcula a média de cada canal e atua sobre cada amostra, forçando uma média 0 a todos os canais. Entretanto, como podemos ver no quarto canal da figura 7.4, essa solução não é eficaz quando a componente DC é próxima dos níveis de saturação. Para contornar esse problema faz-se necessária uma calibragem adequada do *hardware*.

7.3 FIRMWARE

O *firmware* foi projetado de forma modular, onde cada módulo é responsável por uma parte do programa. Para entender os módulo é apresentado um diagrama simplificado do projeto PAI na Figura 7.5.

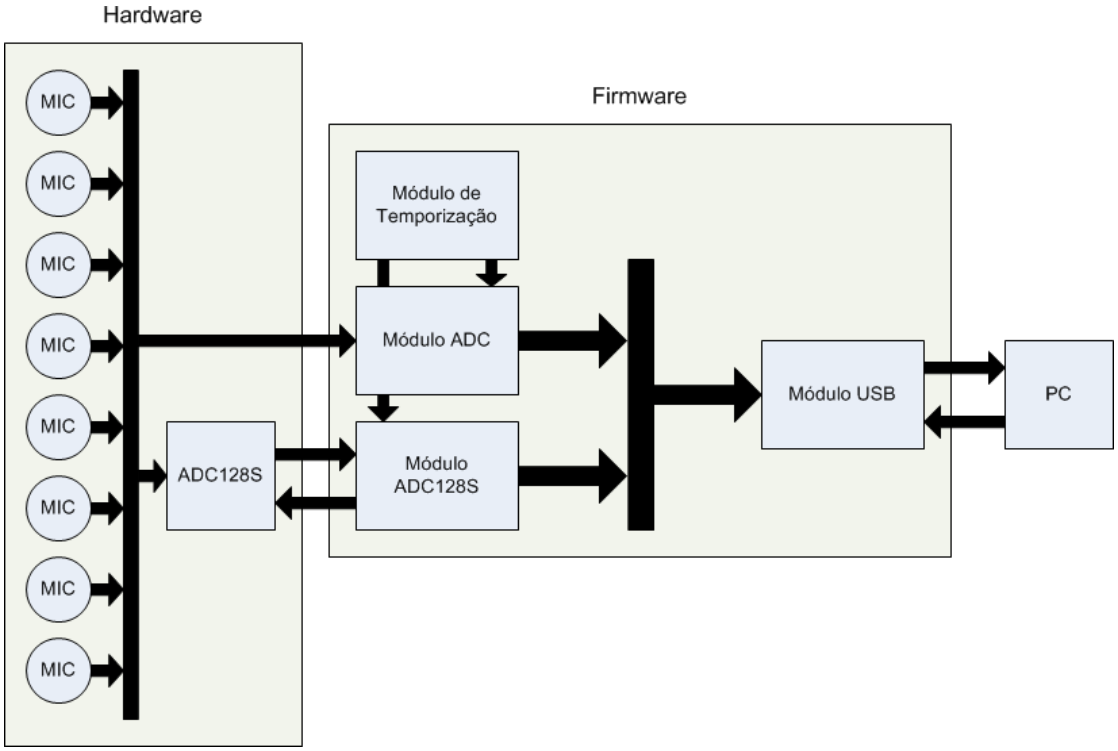


Figura 7.5: Diagrama simplificado do Projeto PAI

Todo o código foi escrito em C no compilador *IAR Embedded Workbench* que está contido no DVD

de documentação do kit ARM. Ele foi escolhido por vários motivos: simplicidade de uso, familiaridade de um dos membros do grupo, o fato de ser gratuito para programas de até 32kB (muito acima do tamanho do *firmware*, que é de aproximadamente 3kB) e, principalmente, a presença de exemplos de código escrito para ele no DVD do kit. Na verdade, antes de decidirmos finalmente pelo IAR, havíamos cogitado usar uma versão do gcc para ARM, mas percebemos que o IAR era mais adequado.

7.3.1 Estrutura de Arquivos

De forma a manter o código organizado o *firmware* foi dividido em arquivos de cabeçalho e arquivos de módulos.

- Arquivos de cabeçalho

  `globalVar.h`: nesse arquivo são declaradas todas as variáveis globais. Esse arquivo deve ser incluído apenas no main.c.

  Os próximos arquivos devem ser incluídos em todos os módulos na ordem que são apresentados aqui:

  `mapa.h`: nesse arquivo são feitas as definições de constantes e macros.

  `functions.h`: nesse arquivo são declarados os protótipos de todas as funções globais.

  `externVar.h`: nesse arquivo todas as variáveis globais, declaradas no arquivo `globalVar.h`, são declaradas como externas.

- Arquivos de módulos

  `main.c`: Este é o módulo principal do *firmware*, responsável pela interação entre todos os outros módulos do *firmware*.

  `adcModulo.c`: esse módulo é responsável pelas rotinas do ADC interno do AT91SAM7S256.

  `adc128s.c`: esse módulo é responsável pelas rotinas de acesso ao ADC externo pelo barramento SPI [1].

  `usbModulo.c`: esse módulo é responsável pelas rotinas de comunicação com o barramento USB.

  `timerModulo.c`: esse módulo configura a temporização e suas interrupções.

[1] O módulo ADC128S dependia de um circuito externo. Apesar de ter sido implementado no *firmware*, ele não foi testado devido a falta do circuito de hardware. Esse não foi montado em tempo para a apresentação deste trabalho

7.3.2 Inicialização

Entre os códigos presentes no DVD do kit ARM, estava incluso um código de inicialização de baixo nível da placa. Esse código foi usado, tornando a tarefa de desenvolver o *firmware* muito simples: não foi necessário se preocupar com a inicialização do PLL e das rotinas de tratamento de interrupção, concentrando todo o esforço no programa de aquisição e envio de dados via USB.

7.3.3 Módulo ADC

O módulo ADC tem duas rotinas: uma de inicialização e configuração do ADC e outra que inicia a conversão e transfere o resultado para o *buffer*.

A função `ADC_Init()` configura o registrador ADC_MR (*ADC Mode Register*) do AT91SAM7S256. Primeiro ela define as características da conversão A/D: conversão iniciada por *software*, com resolução de 10 bits por amostra, com tempo de conversão de cada amostra igual a 4 μs. Então, ela habilita os 8 canais analógicos que serão utilizados para captura do sinal dos microfones.

A função `ADC_Convert()` é chamada por uma interrupção que ocorre a cada 100 μs, garantindo uma frequência de 10 kHz em cada canal. Ela inicia a conversão A/D e espera o fim da conversão de cada canal. Quando uma nova amostra é convertida, o resultado é convertido para PCM e transferido para a próxima posição do *buffer*. Ao final é testado se o *buffer* foi completamente preenchido. A variável `bufferFull` sinaliza ao resto do código que o *buffer* está cheio. O *buffer* foi projetado de tal forma que seu tamanho é equivalente a um quadro de 1 ms de amostras de cada canal. O fluxograma da função `ADC_Convert()` pode ser visto na Figura 7.6.

7.3.4 Módulo USB

O módulo USB é responsável por descrever o dispositivo do ponto de vista USB, realizar a devida configuração e então transferir os dados dos microfones para o *host*.

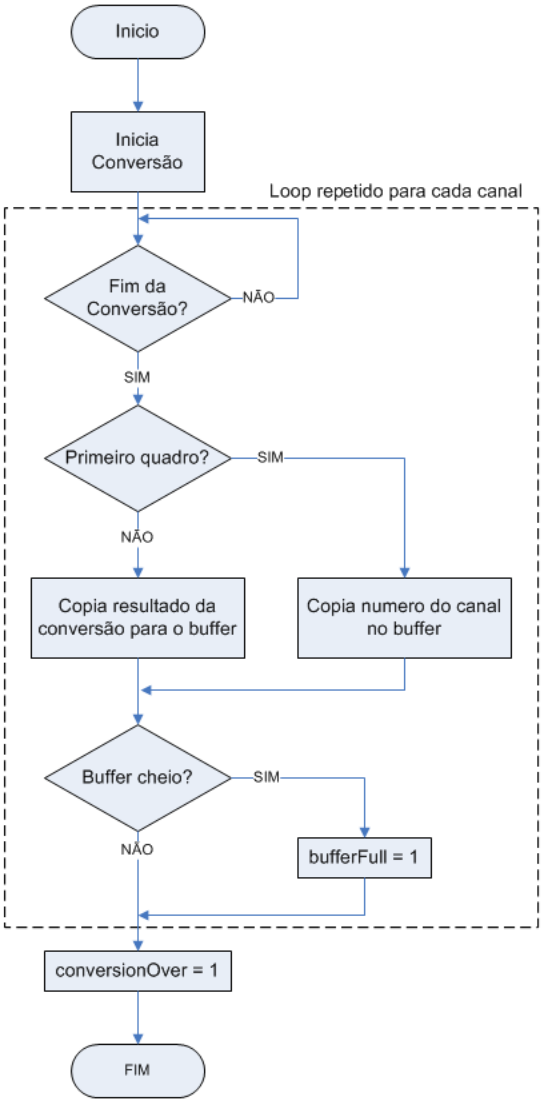


Figura 7.6: Fluxograma da função ADC_Convert()

7.3.4.1 Descrição do Dispositivo USB

No módulo USB são definidos todos os descritores do dispositivo. Para a definição de todos os parâmetros dos descritores faz-se necessário um conhecimento detalhado do funcionamento do *hardware* que será utilizado. O *hardware* do projeto PAI foi descrito simplesmente como um microfone implementando a classe de áudio USB.

O dispositivo modelado possui uma interface do tipo AudioControl (interface 0) e uma interface do tipo AudioStreaming (interface 1). A interface AudioStreaming possui duas funções alternativas. A função alternativa 0 aloca banda igual a zero, pois não possui nenhum *endpoint*, e é selecionada quando o microfone não está em uso. A segunda função alternativa da interface AudioStreaming, possui um *endpoint isoch-*

ronous IN e por isso aloca a banda necessária para a transferência do sinal de áudio do microfone. Essa estrutura é utilizada para que o dispositivo seja configurado mesmo não havendo banda durante o processo de enumeração e também para fazer uso mais eficiente do barramento.

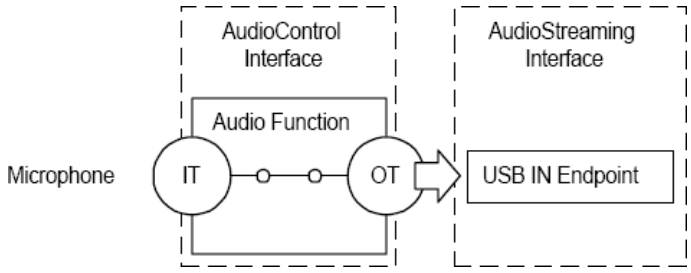


Figura 7.7: Topologia de um microfone USB

A interface AudioControl do dispositivo escolhido não implementa funções por não ter unidades. Ela consiste apenas em um terminal de entrada - o microfone - ligado a um terminal de saída - o *endpoint IN* da interface AudioStreaming. O dispositivo entrega ao *host* um fluxo de dados de áudio com formato PCM de 16-bits a 80kHz [2]. A topologia do dispositivo adotado é representada na Figura 7.7.

Os descritores utilizados para descrever o modelo visto na Figura 7.7 são têm sua hierarquia representada na Figura 7.8.

O primeiro descritor é o DEVICE. Nesta aplicação, ele informa que a função a qual pertence é um dispositivo implementando a especificação 2.0 da USB e apenas uma configuração possível. Ele também é responsável por informar ao *host* que o tamanho máximo de um pacote que pode ser enviado ou recebido pelo *endpoint* 0 deste dispositivo é igual a 8 bytes.

O próximo descritor é o CONFIGURATION da única configuração existente no dispositivo. Ele informa ao *host* que este dispositivo tem duas interfaces (uma AudioControl e uma AudioStreaming) e que pode consumir até 100 mA da porta USB. Este descritor é seguido pelo *AudioControl Interface*, que informa que esta interface implementa a classe de áudio e que utiliza apenas o *endpoint 0*.

O descritor que segue é o primeiro específico da classe de áudio: o *AudioControl Class Header* descreve a versão da especificação como 1.0 e a existência de apenas uma interface AudioControl e uma AudioStreaming. Ele é seguido pelos dois descritores dos terminais.

A partir do próximo, *AudioStreaming Alternative Interface 0*, retornamos aos descritores comum a

[2] 8 canais amostrados a 10 kHz, cada um com 10 bits de sinal seguidos por 6 zeros de *stuffing*.

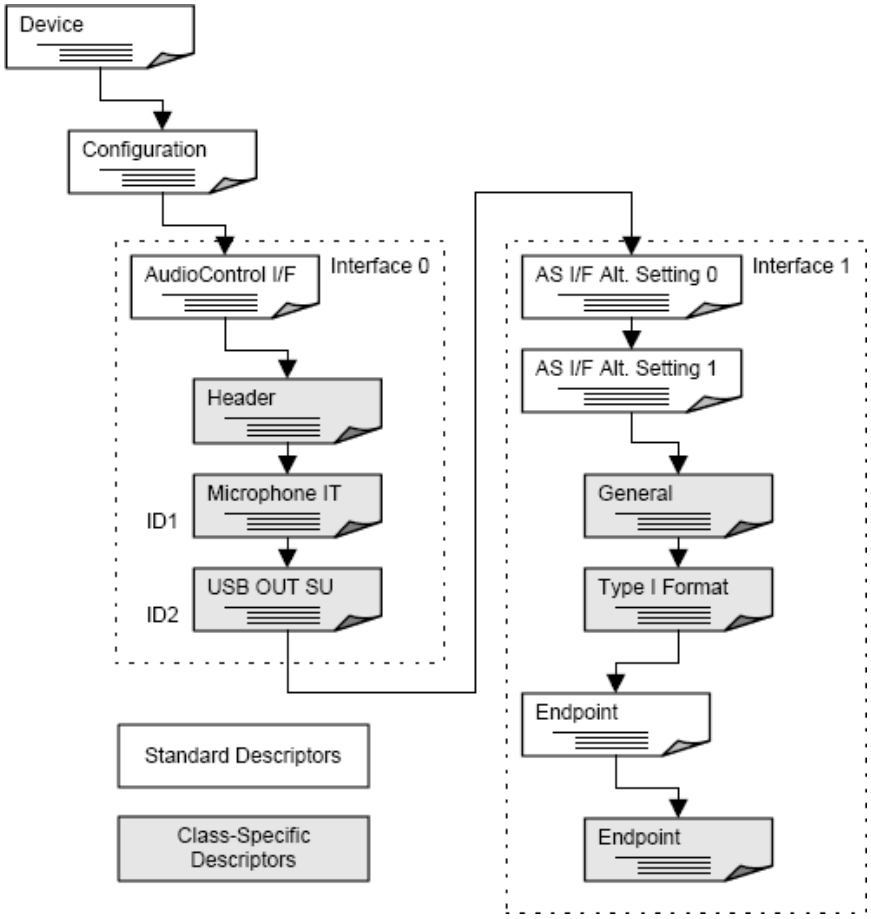


Figura 7.8: Hierarquia de descritores de um microfone USB

todas as classes USB. Ele apenas descreve a interface alternativa 0, que não usa *endpoint* algum e portanto não requer banda do *host* e por isso deve ser usado na enumeração, uma vez que um dispositivo que tenta enumerar requisitando uma banda não disponível no momento não consegue completar sua enumeração. O descritor da interface AudioStreaming alternativa 1 já informa que utiliza o *enpoint* 1, o que denota que o terminal de saída entrega seus dados a ele. Mas isto é expressamente colocado no descritor seguinte, que também nos informa que o atraso da amostra até chegar o *host* é de 1 ms e o formato do som é PCM.

O descritor de tipo de formato informa que há apenas um canal com palavra de 16 bits, sendo que todos eles são utilizados pelo sinal, ou seja, não há bits de *stuffing*. Embora na verdade os bits de *stuffing* existam e sejam 8 canais ao todo, estas informações vão para o *host* pois, como foi descrito em 7.1, o fornecimento correto destas informações implicaria no não-funcionamento da aquisição de dados. O descritor também informa que o dispositivo amostra a 80 kHz.

Finalmente, descritor do *endpoint* 1 configura o modo como assíncrono, com tamanho máximo de

pacote igual a 160 bytes, o que corresponde a um quadro (80 amostras de 2 bytes por milissegundo) e habilita o envio de apenas 1 pacote por quadro.

7.3.4.2 Rotinas

O módulo USB utiliza uma estrutura de dados para interfaceamento com outros módulos do programa. A estrutura _AT91S_AUDIO é definida no arquivo `functions.h` e instanciada no arquivo `globalVar.h` com o nome `Audio` de forma que seja acessível a todos os módulos. Essa estrutura é definida com três variáveis e dois métodos públicos como mostrado abaixo.

```
typedef struct _AT91S_AUDIO
{
    AT91PS_UDP pUdp;
    unsigned char currentConfiguration;
    unsigned char currentSetting;
    // Public Methods:
    unsigned char (*IsConfigured)(struct _AT91S_AUDIO *pAudio);
    unsigned int (*Stream)(struct _AT91S_AUDIO *pAudio, char *pData, unsigned int length);
} AT91S_AUDIO, *AT91PS_AUDIO;
```

A variável `pUdp` indica o endereço base do periférico USB (UDP) dentro do microcontrolador. Isso permite que o mesmo código possa ser utilizado em outros modelos de μControladores da mesma família. A variável `currentConfiguration` indica a atual configuração USB, e a variável `currentSetting` indica a função da interface que está ativa.

O método `IsConfigured(*pAudio)` é responsável pelo processo de enumeração e controle do módulo USB. Ele retorna o valor da configuração atual. Esse método será explorado mais a frente.

O método `Stream(*pAudio,*pData,length)` é responsável por transferir um bloco de dados para a FIFO do *endpoint* AudioStream IN. Esse método também será explorado mais a frente.

A função `USB_Open()` é responsável pela inicialização e configuração do periférico USB. Ela configura a fonte de CLOCK que vai para o periférico USB. Configura os pinos de I/O usados pela USB para habilitar/desabilitar o resistor de *pull-up* ligado a $D+$ e então o habilita para sinalizar que se trata de um dispositivo *full-speed*. Por fim, a estrutura `Audio` é inicializada através da chamada da rotina `USB_AUDIO_Open(pAudio,pUdp)`.

A função `USB_AUDIO_Open(pAudio,pUdp)` é responsável por inicializar as variáveis da estrutura de Audio USB e os métodos que são implementados mais adiante.

O método `IsConfigured(*pAudio)` da estrutura de áudio USB é implementado pela função `AT91F_UDP_IsConfigured(pAudio)`. Essa função, por meio de chamada de outras funções do módulo USB, é responsável pelo tratamento de pacotes de SETUP e configuração do periférico e, portanto, do dispositivo USB. Nela é testado se o dispositivo saiu do estado de RESET ou recebeu algum evento direcionado ao *endpoint* 0. Eventos para o *endpoint* 0 são tratados pela função `AT91F_USB_Enumerate(pAudio)` explorada mais à frente. Por fim, o valor de `currentConfiguration` é retornado pela função. O funcionamento da função `AT91F_UDP_IsConfigured(pAudio)` pode ser visto no fluxograma da Figura 7.9.

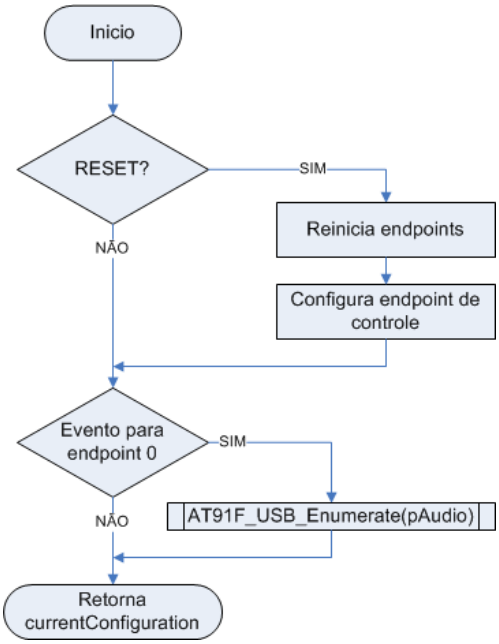


Figura 7.9: Fluxograma da função IsConfigured(pAudio) do módulo USB

O método `Stream(pAudio,*pData,length)` da estrutura de áudio USB é implementado pela função `AT91F_UDP_Stream(pAudio,*pData,length)`. Essa função utiliza o mecanismo de *ping-pong* para a transferência de bloco de dados pela USB. É passado para a função o ponteiro do inicio do bloco de dados e também o tamanho do bloco em número de bytes. A rotina se encarrega de fazer a troca de bancos da FIFO do *endpoint* de áudio.

As próximas funções são privadas, utilizadas somente dentro do módulo USB.

A função `AT91F_USB_SendData(pUdp,*pData,length)` é responsável por enviar blocos de dados pelo *endpoint* 0, ou seja, o *endpoint* de controle. Ela utiliza apenas um banco da FIFO do *endpoint* 0, pois o esquema *ping-pong* não é suportado para transferências do tipo *control*. Para a função são entregues

como parâmetros o endereço de início do bloco de dados e o tamanho do bloco em bytes.

A função `AT91F_USB_SendZlp(pUdp)` tem a rotina para envio de um pacote de dados do tipo IN de tamanho igual a zero pelo *endpoint* de controle.

A função `AT91F_USB_SendStall(pUdp)` gera o envio de um STALL pelo *endpoint* 0.

A função `AT91F_USB_Enumerate(pAudio)` é responsável por tratar o recebimento de pacotes de SETUP pelo *endpoint* de controle. Nessa rotina são implementadas as rotinas de tratamento de cada pedido que possa ser enviado pelo *host*. Essa rotina trabalha com as transferenciais do tipo *control* endereçadas ao *endpoint* 0.

Os dados transferidos durante a fase de dados são armazenados e decodificados para se descobrir que ação o *host* está requisitando do dispositivo. Os pedidos são identificados pelos campos *bmRequestType* e *bRequest*. Cada pedido é então tratado de forma específica configurando o periférico USB do AT91SAM7S256, ou utilizando as funções `AT91F_USB_SendData(pUdp,*pData,length)` para enviar dados ao *host*, ou `AT91F_USB_SendZlp(pUdp)` par indicar o fim da fase de dados, ou `AT91F_USB_SendStall(pUdp)` para enviar um STALL.

7.3.5 Módulo de Temporização

O módulo de temporização consiste em duas funções: `Tim0Init()`, que configura o Timer/Counter 0 e `timer0_irq_handler(void)` que é chamada quando ocorre a interrupção. O Timer/Counter 0 foi escolhido apenas por que consta na documentação do ARM presente no DVD um exemplo onde ele é usado em conjunto com o ADC, pois a mesma função poderia ser realizada por outro recurso do ARM capaz de gerar interrupções periódicas, o *Periodic Interval Timer*.

A rotina `Tim0Init()` inicialmente limpa qualquer configuração existente do *Timer/Counter* 0. Então ela configura a taxa do CLOCK do *Timer/Counter* para ser igual a MCK/8, o que corresponde a 5990,4 kHz. Como é desejada uma taxa de 10 kHz, a frequência atual deve ser dividida por $\frac{5990,4}{10} \cong 599$, o que resulta em uma frequência 0,7 Hz acima de 10 kHz (um erro de $6,7 \times 10^{-5}$). Para isso, o *Timer/Counter* é configurado no modo de contagem WAVSEL = 10 sem *trigger* e RC recebe 599, de forma que cada vez que o *Timer/Counter* contar até 599 ele gerará uma interrupção e reiniciará a contagem. Finalmente é

configurada a interrupção em si, associando-a à função `timer0_irq_handler(void)`, e habilitando a interrupção.

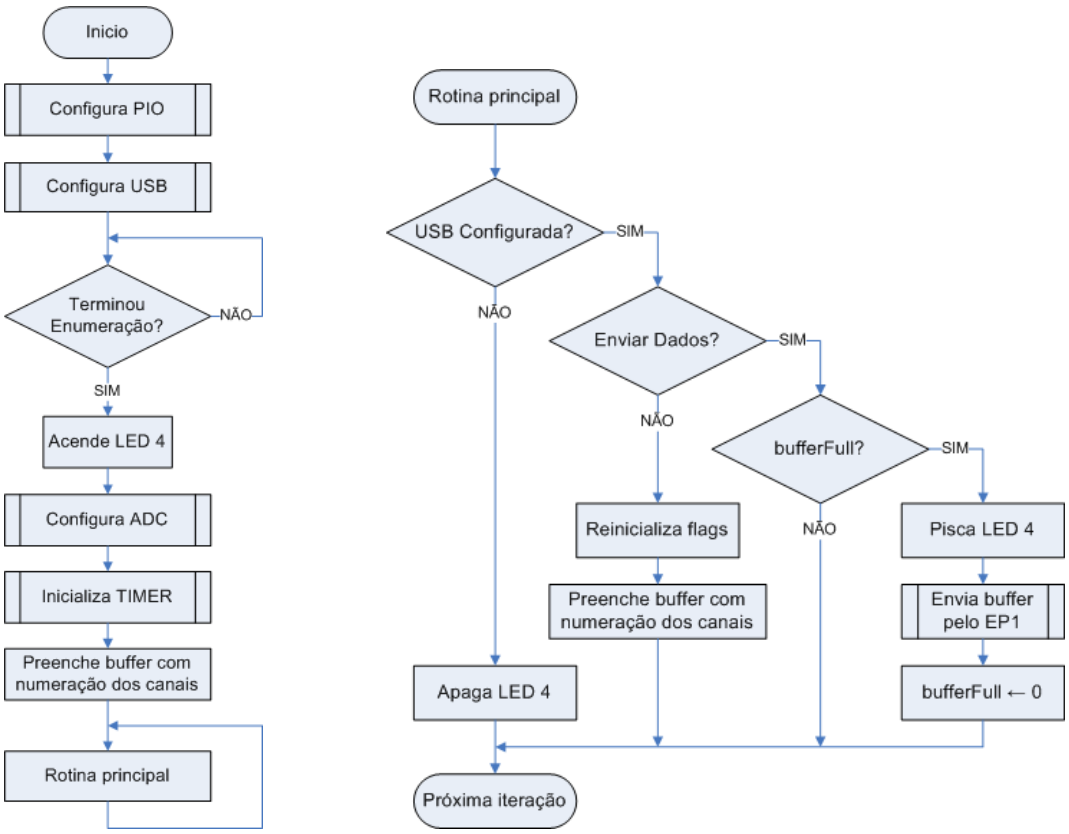
Já a rotina `timer0_irq_handler(void)` apenas chama a rotina `ADC_Convert()` do módulo ADC para converter uma nova leva de amostras, verificando antes se a última conversão já acabou e se o *buffer* não está cheio .

7.3.6 Módulo Principal

O módulo principal funciona como uma espécie de inicialização de alto nível, servindo apenas como ponto de entrada do *firmware*. Inicialmente, é chamada a função `Inicio()`, declarada no próprio main.c, para configurar os LEDs. É chamada então a função `USB_Open()`, do módulo USB, para dar início ao processo de configuração USB descrito na seção 7.3.4.

Quando o módulo USB retorna, o `main.c` entra em um *loop* que apenas espera pelo fim da enumeração. Uma vez que ela ocorra com sucesso, liga-se o LED 4 para indicar que o dispositivo está enumerado e configuram-se o ADC e o TC responsável pela manutenção da taxa de amostragem de 10 kHz.

Com todas as configurações feitas o programa entra em um *loop* infinito onde aguarda que o *host* configure o dispositivo na função alternativa 1, requisitando o envio de um sinal de áudio. Uma vez que isso ocorre o *loop* se encarrega de controlar o *buffer*, inclusive piscando o LED 4 para indicar a velocidade da transmissão e enviando um primeiro quadro apenas com a contagem do canal atual para que o *host* possa se sincronizar e identificar os canais após a demultiplexação. Caso a transmissão pare, o *loop* esvazia o *buffer* e prepara-se pra enviar um novo primeiro pacote quando ocorrer um novo pedido.



(a) Módulo principal

(b) Loop do módulo principal

Figura 7.10: Fluxograma do módulo principal

8 CONCLUSÕES

Este documento é o resultado de seis meses de esforço na criação do *firmware* de um dispositivo USB para levar sinais de áudio em tempo real de um microcontrolador a um computador pessoal. Há ainda uma meta maior permeando o projeto e unindo-o a um segundo grupo de trabalho, que é possibilitar a estimação da direção de chegada de um sinal de voz captado por um arranjo de microfones de baixo custo. [1]

O *firmware* pode ser descrito como a combinação de três partes menores, a saber: a enumeração do dispositivo USB, a comunicação correta entre as partes e a obtenção da taxa necessária de transferência. Apesar do sucesso alcançado nos dois primeiros objetivos, diversos contratempos impediram a concretização do terceiro. Muito tempo foi perdido tentando desenvolver o *firmware* em etapas: inicialmente, o plano era conseguir enumerar o dispositivo para só então criar uma rotina de conversão para prover dados para envio. Quando as tentativas de enumeração foram finalmente abandonadas e a rotina de conversão e preenchimento de *buffer* foi criada, descobriu-se que o motivo que impedia o dispositivo de enumerar é que ele não enumera caso não esteja com a parte de amostragem funcionando corretamente.

Com o dispositivo funcionando adequadamente, iniciaram-se os testes de aquisição de sinal com o PC. Logo nos primeiros testes ficou claro que o Windows não adquiria nenhum canal além do segundo, o que obrigou a uma mudança na arquitetura do *software*, que agora teria que multiplexar toda a informação em um canal. Este canal seria enviado a oito vezes a velocidade de um único canal, ou seja, sem prejuízo de banda por microfone. Mas ao iniciar estes testes, foi verificado que para qualquer taxa acima de 32 kHz sérios problemas de sincronia ocorriam, inviabilizando a amostragem com as características inicialmente especificadas.

Após uma grande procura pela causa do problema descobriu-se que os *endpoints isochronous* do microcontrolador eram de um tamanho distinto do padrão da especificação USB. Este erro era mascarado por uma redação confusa do *datasheet* que só foi modificada em uma revisão lançada em novembro, já a menos de um mês do fim do projeto. Quando a limitação finalmente foi identificada, não havia mais tempo hábil para solucionar o problema e preparar uma rotina de envio na taxa inicialmente proposta.

Ainda assim seria possível fazer a estimação da direção de chegada com as rotinas atuais. Bastaria

utilizar 4 canais amostrados a 8 kHz ou 8 canais a 4 kHz, entre outras tantas alternativas que resultariam em uma taxa de até 32 kHz. A estimação de chegada ainda poderia ser feita, já que o algoritmo utilizado procura por sinais na faixa de 1 kHz. Infelizmente, a necessidade de sincronia entre os membros do grupo responsável pelo *hardware* e os deste grupo, aliados aos atrasos na monografia de ambos os lados impediram tal teste.

Tais percalços não impediram, porém, que se estudassem possíveis soluções para os problemas remanescentes do projeto. Quando esse tiver continuidade, uma das alternativas para aumentar a velocidade da transmissão é utilizar também o *endpoint* 2 do microcontrolador, uma vez que este também suporta transferências *isochronous*. Entretanto, esta solução limitaria a nova taxa ao dobro da atual, ou seja, 64 kHz. Este novo patamar seria uma melhora, mas significaria uma nova limitação que ainda estaria abaixo do inicialmente projetado. Uma solução mais eficaz seria o uso de múltiplo pacotes por quadro, o que, segundo o fabricante do microcontrolador, nos permitiria uma taxa de até 1 MB/s, permitindo uma taxa de 500KHz.

Um outro problema que ficou sem solução é o envio dos canais sem a necessidade de demultiplexação por parte do *host*. Esta tarefa talvez tenha sua resposta na nova versão do *Windows*, o *Windows Vista*, que dará grande importância a arranjos de microfones, inclusive criando um descritor para sua geometria[14]. O *Windows Vista* trará também suporte a elementos da versão 2.0 da classe de áudio [15] e ao modo assíncrono da versão 1.0, que não era suportado pelas versões anteriores do *Windows* [15]. Esta incompatibilidade com o modo assíncrono só foi descoberta no fim do projeto e, como foi o modo implementado no dispositivo, pode explicar os problemas para usar mais de dois canais.

É válido considerar que uma continuação deste projeto ofereceria possibilidades de melhorar significativamente a quantidade de dados transferidos utilizando algum formato de som com compressão suportado pela classe de áudio, ao invés do PCM. Poderia ser interessante também o desenvolvimento de um *driver* específico para esta aplicação, o que poderia possibilitar a estimação da direção de chegada em tempo real. Finalmente, um projeto ainda mais ambicioso seria a de se estimar a DOA no próprio ARM, tarefa dificultada por seu poder de processamento relativamente reduzido e sua incapacidade de trabalhar com ponto flutuante, usado nas rotinas de estimação.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] COSTA, A. R. A. da; GARCIA, F. A. da C. *Desenvolvimento do Hardware para Obtenção de DOA por meio de Arranjo de Sensores: Aplicação em Prótese Auditiva*. 88 p. Monografia (Projeto Final de Graduação em Engenharia Elétrica) — Faculdade de Tecnologia, Universidade de Brasília, Brasília, Dezembro 2006.

[2] ZELENOVSKY, R.; MENDONÇA, A. *PC: Um Guia Prático de Hardware e Interfaceamento*. Rio de Janeiro: MZ Editora Ltda, 2002.

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

[4] SITE do USB - Implementer's Forum. `http://usb.org`.

[5] SITE da UNICODE. `http://www.unicode.com`.

[6] COMPAQ et al. *Universal Serial Bus Specification - Revision 2.0*. Disponível em `http://www.usb.org`: USB Implementers Forum, 2000.

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

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

[9] SALOMÃO, A.; FRIEDMAN, A. de C. *Desenvolvimento de uma interface USB para comunicação com o microcontrolador 8051*. 134 p. Monografia (Projeto Final de Graduação em Engenharia Elétrica) — Faculdade de Tecnologia, Universidade de Brasília, Brasília, Dezembro 2005.

[10] IBM et al. *USB Device Class Definition for Audio Devices - Revision 1.0*. Disponível em `http://www.usb.org/developers/devclass_docs/audio10.pdf`: USB Implementers Forum, 1998.

[11] BIBLIOTECA pa_wavrecord para MATLAB. http://sourceforge.net/projects/pa-wavplay/.

[12] ATMEL. *AT91SAM7S Series Preliminary*. [S.l.], Abril 2006. Disponível em http://www.atmel.com/products/AT91/.

[13] ATMEL. *AT91SAM7S Series Preliminary*. [S.l.], Novembro 2006. Disponível em http://www.atmel.com/products/AT91/.

[14] MICROSOFT. *How to Build and Use Microphone Arrays for Windows Vista*. [S.l.], 2006. Disponível em http://www.microsoft.com/whdc/device/audio/MicArrays_guide.mspx.

[15] MICROSOFT. *Universal Audio Architecture Hardware Design Guidelines*. [S.l.], 2006. Disponível em http://go.microsoft.com/fwlink/?LinkId=50734.

[16] IBM et al. *Universal Serial Bus Device Class Definition for Terminal Types - Release 1.0*. Disponível em http://www.usb.org/developers/devclass_docs/termt10.pdf: USB Implementers Forum, 1998.

[17] IBM et al. *Universal Serial Bus Device Class Definition for Audio Data Formats - Release 1.0*. Disponível em http://www.usb.org/developers/devclass_docs/frmts10.pdf: USB Implementers Forum, 1998.

ANEXOS

I. CÓDIGO FONTE

I.1 GLOBALVAR.H

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI
//*-----------------------------------------------------------------------------
//* Nome do Arquivo     : globalVar.c
//* Objetivo            : declaração e inicializacao de variaveis
//* Criação             : 03/10/2006
//*-----------------------------------------------------------------------------

//*-----------------------------------------------------------------------------
//*         LEDs
//*-----------------------------------------------------------------------------
const int ledMask[4] = {LED1, LED2, LED3, LED4};
unsigned int ledCount1 = 0, ledCount2 = 0, ledCount3 = 0, ledCount4 = 0;

int flagMode = 0;


//*-----------------------------------------------------------------------------------------
//*         ADC Module
//*-----------------------------------------------------------------------------------------
signed short sampleBuffer[BUFFER_SIZE];
char bufferFull = 0;
char adcChannel = 0;
char conversionOver = 1;
unsigned short pIn = 0, pOut = BUFFER_SIZE;
unsigned short contador = 0;
char conversorAtual = 0;


//*-----------------------------------------------------------------------------
//*         USB Module
//*-----------------------------------------------------------------------------
extern const short epSize[4] = { EP0_SIZE, EP1_SIZE, EP2_SIZE, EP3_SIZE };
struct _AT91S_AUDIO Audio;
char firstTransfer = 1;
char packetSent = 1;
```

I.2 MAPA.H

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI
//*-----------------------------------------------------------------------------
//* Nome do Arquivo     : mapa.h
//* Objetivo            : declaração de constantes
//* Criação             : 01/10/2006
//*-----------------------------------------------------------------------------

#include "Board.h"

//#define DEBUG         // Modo DEBUG

/* General Macros */
#define MIN(a, b) (((a) < (b)) ? (a) : (b))

/* Defines Globais */

#define RC_SIZE         599                         // RC Timer Size (10kHz)
//#define RC_SIZE         749                         // RC Timer Size (8kHz)
//#define RC_SIZE         1498                        // RC Timer Size (4kHz)
```

```
#define ADC_RESOLUTION  16                          // resolução do ADC (16)
#define USB_CHANNELS    1                           // 1 a 8 canais de entrada
#define ADC_CHANNELS    4                           // 1 a 8 canais de entrada
#define SAM_FREQ        (ADC_CHANNELS * 10000)      // Sampling frequency in Hz (canais * freq)
#define SAM_1MS         (SAM_FREQ/1000)             // # de amostras em 1ms
#define BUFFER_SIZE     (SAM_1MS * USB_CHANNELS)    // Buffer (1ms de amostras por canal)


//*-----------------------------------------------------------------------------
//*         USB Module
//*-----------------------------------------------------------------------------
/* Basic USB Parameters */
#define VENDOR_ID       0xFFFF
#define PRODUCT_ID      0x0001
#define VERSION         0x0003
#define EP0_SIZE        0x08
#define EP1_SIZE        (0x0002 * BUFFER_SIZE)  // amostras de 1 quadro de 1ms * 2 bytes
#define EP2_SIZE        0x08
#define EP3_SIZE        0x08
#define AUDIO_IN        0x01                    // Endpoint de audio

/* Descriptor types */
#define DESC_TYPE_DEVICE                        0x01
#define DESC_TYPE_CONFIGURATION                 0x02
#define DESC_TYPE_STRING                        0x03
#define DESC_TYPE_INTERFACE                     0x04
#define DESC_TYPE_ENDPOINT                      0x05
#define DESC_TYPE_DEVICE_QUALIFIER              0x06
#define DESC_TYPE_OTHER_SPEED_CONFIGURATION     0x07
#define DESC_TYPE_INTERFACE_POWER               0x08

#define DESC_TYPE_HID                           0x21

/* USB standard request code */
/* Code = bRequest << 8) | bmRequestType */
#define STD_GET_STATUS_ZERO                     0x0080
#define STD_GET_STATUS_INTERFACE                0x0081
#define STD_GET_STATUS_ENDPOINT                 0x0082

#define STD_CLEAR_FEATURE_ZERO                  0x0100
#define STD_CLEAR_FEATURE_INTERFACE             0x0101
#define STD_CLEAR_FEATURE_ENDPOINT              0x0102

#define STD_SET_FEATURE_ZERO                    0x0300
#define STD_SET_FEATURE_INTERFACE               0x0301
#define STD_SET_FEATURE_ENDPOINT                0x0302

#define STD_SET_ADDRESS                         0x0500
#define STD_GET_DESCRIPTOR                      0x0680
#define STD_SET_DESCRIPTOR                      0x0700
#define STD_GET_CONFIGURATION                   0x0880
#define STD_SET_CONFIGURATION                   0x0900
#define STD_GET_INTERFACE                       0x0A81
#define STD_SET_INTERFACE                       0x0B01
#define STD_SYNCH_FRAME                         0x0C82

/* Audio Class Specific Request Code */


//*-----------------------------------------------------------------------------
//*         TIM Module
//*-----------------------------------------------------------------------------
#define TIM0_INT_PRIOR 7

/* Clock Selection */
#define TC_CLKS                 0x7


//*-----------------------------------------------------------------------------
//*         Test Points
//*-----------------------------------------------------------------------------
#define TP0                     AT91C_PIO_PA11
```

```
#define TP1                     AT91C_PIO_PA12
```

I.3 MAIN.C

```
//*----------------------------------------------------------------------------
//*         Projeto PAI
//*----------------------------------------------------------------------------
//* Nome do Arquivo     : main.c
//* Objetivo            : Main aplication
//* Criado              : 01/10/2006
//*----------------------------------------------------------------------------

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

void Inicio(void)
{
    // habilita clock do PIO
    AT91C_BASE_PMC->PMC_PCER = (1 << AT91C_ID_PIOA);

    // COnfigura as linhas de PIO ligadas aos LED1 a LED4
    AT91F_PIO_CfgOutput( AT91C_BASE_PIOA, LED_MASK ) ;

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

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

}

int main(void)
{// Inicio

    Inicio();               // Configura PIO

    USB_Open();             // Configura UDP (USB)

    // Wait for the end of enumeration
    while (!Audio.IsConfigured(&Audio));

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

    ADC_Init();             // Configura ADC

    Tim0Init();             // Configura Timer-Counter 0

    for(int i=0; i < BUFFER_SIZE; i++)
        sampleBuffer[i] = i % ADC_CHANNELS;
    bufferFull = 1;

  // Loop infinito
    for(;;)
    {
        /* Transmissão USB */
        if (Audio.IsConfigured(&Audio)) {
            Audio.BufferReady(&Audio);
            if(Audio.currentSetting && bufferFull) {
                // Pisca LED
                if(ledCount4 == 100)
                    AT91C_BASE_PIOA->PIO_CODR = LED4;
```

```
                if(ledCount4++ == 200) {
                    AT91C_BASE_PIOA->PIO_SODR = LED4;
                    ledCount4 = 0;
                }
#ifdef DEBUG
                AT91C_BASE_PIOA->PIO_SODR = TP0;    // Ativa Ponto de test para debug
#endif //DEBUG
                if (!Audio.Stream(&Audio, (char *) sampleBuffer, (epSize[AUDIO_IN])))
                {
#ifdef DEBUG
                    AT91C_BASE_PIOA->PIO_CODR = TP0;    // Desativa ponto de teste
#endif //DEBUG
                    bufferFull = 0;
                }
            }
            else if(!Audio.currentSetting) {
                AT91C_BASE_PIOA->PIO_CODR = LED4;

                for(int i=0; i < BUFFER_SIZE; i++)
                    sampleBuffer[i] = i % ADC_CHANNELS;

                // Reinicializa as flags
                bufferFull = 1;
                pIn = 0;
                firstTransfer = 1;
                packetSent = 1;
            }
        }
        else
            AT91C_BASE_PIOA->PIO_SODR = LED4;
    }

}// Fim
```

I.4 ADCMODULO.C

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI
//*-----------------------------------------------------------------------------
//* Nome do Arquivo     : adcModulo.c
//* Objetivo            : A/D Converter
//* Criação             : 03/10/2006
//*-----------------------------------------------------------------------------

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

void ADC_Init(void)
{// Inicio
    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)

    AT91C_BASE_ADC->ADC_CHER = 0xF0;       // habilita todos os canais

    AT91F_PIO_CfgPeriph (AT91C_BASE_PIOA, 0 , 0);
}// Fim

void ADC_Convert(void)
{//* Inicio
    AT91C_BASE_ADC->ADC_CR = AT91C_ADC_START;        // inicia conversao
    AT91C_BASE_PIOA->PIO_SODR = TP1;
    flagMode = 0;
```

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

#ifdef DEBUG
        if(flagMode) {
            AT91C_BASE_PIOA->PIO_SODR = TP0;
            flagMode = 0;
        }
        else {
            AT91C_BASE_PIOA->PIO_CODR = TP0;
            flagMode = 1;
        }
#endif

        if(firstTransfer) {
            sampleBuffer[pIn] = pIn % ADC_CHANNELS;
        }
        else {
            // Copia amostra para o buffer, convertendo para PCM:
            sampleBuffer[pIn] =
                (signed short)(((AT91C_BASE_ADC->ADC_LCDR & AT91C_ADC_DATA) << 6) - 0x8000);
        }
        // checa se o ponteiro de entrada no buffer chegou ao final
        if(++pIn == BUFFER_SIZE) {
            bufferFull = 1;
            pIn = 0;
        }
    }
#ifdef DEBUG
    AT91C_BASE_PIOA->PIO_CODR = TP1;
#endif //DEBUG
    conversionOver = 1;
}//* Fim
```

I.5 USBMODULO.C

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI
//*-----------------------------------------------------------------------------
//* Nome do Arquivo     : usbModulo.c
//* Objetivo            : USB Module
//* Creação            : 25/10/2006
//*-----------------------------------------------------------------------------

#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, 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) (AS interface 1)

/* 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 position)
  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
  0x01, 0x00,                           //wFormatTag (Audio Data Format = PCM)

/* Format Type Descriptor */
  0x0B,                                 //bLength
  0x24,                                 //bDescriptorType (CS_INTERFACE)
  0x02,                                 //bDescriptorSubtype (FORMAT_TYPE)
  0x01,                                 //bFormatType (FORMAT_TYPE_I)
  USB_CHANNELS,                         //bNrChannels (# of physical channels)
  0x02,                                 //bSubFrameSize (2 bytes in audio subframe)
  ADC_RESOLUTION,                       //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)
  (char)EP1_SIZE,(EP1_SIZE>>8),         //wMaxPacketSize (# bytes per packet)
  0x01,                                 //bInterval (1 packet per frame)
  0x00,                                 //bRefresh
  0x00,                                 //bSynchAddress (No Endpoint for synchronization)

/* 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
};


void USB_Open(void)
{
    // Configura o divisor do PLL para USBr
    AT91C_BASE_CKGR->CKGR_PLLR |= AT91C_CKGR_USBDIV_1 ;

    // Habilita os clock USB
    AT91C_BASE_PMC->PMC_SCER |= AT91C_PMC_UDP;
    AT91C_BASE_PMC->PMC_PCER |= (1 << AT91C_ID_UDP);

    // Habilita UDP PullUp (USB_DP_PUP)
    AT91F_PIO_CfgOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);
    AT91F_PIO_ClearOutput(AT91C_BASE_PIOA,AT91C_PIO_PA16);

    // Inicia estrutura de 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;
    pAudio->BufferReady = AT91F_USB_BufferReady;
    return pAudio;
}

static unsigned char AT91F_UDP_IsConfigured(AT91PS_AUDIO pAudio)
{
```

```
    AT91PS_UDP pUDP = pAudio->pUdp;
    AT91_REG isr = pUDP->UDP_ISR;

    if (isr & AT91C_UDP_ENDBUSRES) {
        pUDP->UDP_ICR = AT91C_UDP_ENDBUSRES;
        // reinicia todos os 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);
    }
    else if (isr & AT91C_UDP_EPINT0) {
AT91F_USB_Enumerate(pAudio);
    }
    return pAudio->currentConfiguration;
}

static unsigned int AT91F_UDP_Stream(AT91PS_AUDIO pAudio, char *pData, unsigned int length)
{
AT91PS_UDP pUdp = pAudio->pUdp;
unsigned int cpt;

        cpt = MIN(length, epSize[AUDIO_IN]);

        if(firstTransfer) {
            // Envia primeiro pacote
            length -= cpt;
            while (cpt--)
                pUdp->UDP_FDR[AUDIO_IN] = *pData++;
            pUdp->UDP_CSR[AUDIO_IN] |= AT91C_UDP_TXPKTRDY;
            firstTransfer = 0;
        }
        else {
            if(packetSent) {
                // Preenche segundo banco
                length -= cpt;
                while (cpt--)
                    pUdp->UDP_FDR[AUDIO_IN] = *pData++;
                packetSent = 0;
            }

            // Testa se o primeiro banco foi enviado
            if(pUdp->UDP_CSR[AUDIO_IN] & AT91C_UDP_TXCOMP) {
                pUdp->UDP_CSR[AUDIO_IN] &= ~(AT91C_UDP_TXCOMP);
                while (pUdp->UDP_CSR[AUDIO_IN] & AT91C_UDP_TXCOMP);
                packetSent = 1;
                pUdp->UDP_CSR[AUDIO_IN] |= AT91C_UDP_TXPKTRDY;
            }
        }
        return length;
}
static void AT91F_USB_BufferReady (AT91PS_AUDIO pAudio) {
    AT91PS_UDP pUdp = pAudio->pUdp;
    if(pUdp->UDP_CSR[AUDIO_IN] & AT91C_UDP_TXCOMP) {
        pUdp->UDP_CSR[AUDIO_IN] &= ~(AT91C_UDP_TXCOMP);
        while (pUdp->UDP_CSR[AUDIO_IN] & AT91C_UDP_TXCOMP);
        packetSent = 1;
        pUdp->UDP_CSR[AUDIO_IN] |= AT91C_UDP_TXPKTRDY;
    }
}


static void AT91F_USB_SendData(AT91PS_UDP pUdp, const char *pData, unsigned int length)
{
    unsigned int cpt = 0;
    AT91_REG csr;

    do {
        cpt = MIN(length, epSize[0]);
length -= cpt;
```

```
while (cpt--)
            pUdp->UDP_FDR[0] = *pData++;

if (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP) {
            pUdp->UDP_CSR[0] &= ~(AT91C_UDP_TXCOMP);
    while (pUdp->UDP_CSR[0] & AT91C_UDP_TXCOMP);
}

pUdp->UDP_CSR[0] |= AT91C_UDP_TXPKTRDY;
do {
            csr = pUdp->UDP_CSR[0];

    // estagio Data IN interrompido por um Status OUT
    if (csr & AT91C_UDP_RX_DATA_BK0) {
                pUdp->UDP_CSR[0] &= ~(AT91C_UDP_RX_DATA_BK0);
return;
    }
} while ( !(csr & AT91C_UDP_TXCOMP) );

    } while (length);

    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)
{
    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)
{
    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)
{
    AT91PS_UDP pUDP = pAudio->pUdp;
    unsigned char bmRequestType, bRequest;
    unsigned short wValue, wIndex, wLength, wStatus;


    if ( !(pUDP->UDP_CSR[0] & AT91C_UDP_RXSETUP) )
return;

    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);

    if (bmRequestType & 0x80) {
pUDP->UDP_CSR[0] |= AT91C_UDP_DIR;
while ( !(pUDP->UDP_CSR[0] & AT91C_UDP_DIR) );
    }
    pUDP->UDP_CSR[0] &= ~AT91C_UDP_RXSETUP;
    while ( (pUDP->UDP_CSR[0]  & AT91C_UDP_RXSETUP)  );

    // Trata os pedidos padrões da specification USB Rev 2.0
    switch ((bRequest << 8) | bmRequestType)
    {
    case STD_GET_DESCRIPTOR:
```

```
    if (wValue == 0x100)       // Descritor de dispositivo
        AT91F_USB_SendData(pUDP, DeviceDescriptor, MIN(sizeof(DeviceDescriptor), wLength));
    else if (wValue == 0x200)  // Descritor de configuração
        AT91F_USB_SendData(pUDP,
                           ConfigurationDescriptor,
                           MIN(sizeof(ConfigurationDescriptor),
                           wLength));
    else
        AT91F_USB_SendStall(pUDP);
    break;
case STD_SET_ADDRESS:
    AT91F_USB_SendZlp(pUDP);
    pUDP->UDP_FADDR = (AT91C_UDP_FEN | wValue);
    pUDP->UDP_GLBSTATE  = (wValue) ? AT91C_UDP_FADDEN : 0;
    break;
case STD_SET_CONFIGURATION:
    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;
    //pUDP->UDP_CSR[2] = (wValue) ? (AT91C_UDP_EPEDS | AT91C_UDP_EPTYPE_BULK_IN) : 0;
    //pUDP->UDP_CSR[3] = (wValue) ? (AT91C_UDP_EPEDS | AT91C_UDP_EPTYPE_BULK_OUT) : 0;
    break;
case STD_GET_CONFIGURATION:
    AT91F_USB_SendData(pUDP,
                       (char *) &(pAudio->currentConfiguration),
                       sizeof(pAudio->currentConfiguration));
    break;
case STD_GET_STATUS_ZERO:
    wStatus = 0;
    AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
    break;
case STD_GET_STATUS_INTERFACE:
    wStatus = 0;
    AT91F_USB_SendData(pUDP, (char *) &wStatus, sizeof(wStatus));
    break;
case STD_GET_STATUS_ENDPOINT:
    wStatus = 0;
    wIndex &= 0x0F;
    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));
    }
    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
        AT91F_USB_SendStall(pUDP);
    break;
case STD_SET_FEATURE_ZERO:
    AT91F_USB_SendStall(pUDP);
    break;
case STD_SET_FEATURE_INTERFACE:
    AT91F_USB_SendZlp(pUDP);
    break;
case STD_SET_FEATURE_ENDPOINT:
    wIndex &= 0x0F;
    if ((wValue == 0) && wIndex && (wIndex <= 3)) {
        pUDP->UDP_CSR[wIndex] = 0;
        AT91F_USB_SendZlp(pUDP);
    }
    else
    AT91F_USB_SendStall(pUDP);
    break;
case STD_CLEAR_FEATURE_ZERO:
    AT91F_USB_SendStall(pUDP);
    break;
case STD_CLEAR_FEATURE_INTERFACE:
    AT91F_USB_SendZlp(pUDP);
    break;
case STD_CLEAR_FEATURE_ENDPOINT:
    wIndex &= 0x0F;
```

```
        if ((wValue == 0) && wIndex && (wIndex <= 1)) {
            if (wIndex == 1)
                pUDP->UDP_CSR[1] = (AT91C_UDP_EPEDS | AT91C_UDP_EPTYPE_ISO_IN);
            AT91F_USB_SendZlp(pUDP);
        }
        else
            AT91F_USB_SendStall(pUDP);
        break;
    case STD_GET_INTERFACE:
        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:
        if((wIndex == 1) && (wValue <= 1) && (pAudio->currentConfiguration)){
            pAudio->currentSetting = wValue;
            AT91F_USB_SendZlp(pUDP);
            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;
            }
            else if(wValue == 0) {
                pUDP->UDP_CSR[AUDIO_IN] = 0;
            }
        }
        else
            AT91F_USB_SendStall(pUDP);
        break;
    case STD_SET_DESCRIPTOR:
    case STD_SYNCH_FRAME:

    // Trata pedidos especificos da classe de audio
    default:
        AT91F_USB_SendStall(pUDP);
        break;
    }
}
```

I.6 TIMERMODULO.C

```
//*-----------------------------------------------------------------------------
//*         Projeto PAI
//*-----------------------------------------------------------------------------
//* Nome do Arquivo     : timerModulo.c
//* Objetivo            : Interrupção de tempo
//* Criação             : 01/10/2006
//*-----------------------------------------------------------------------------

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

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

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;

#ifdef DEBUG
    if(flagMode) {
```

```
        AT91C_BASE_PIOA->PIO_SODR = TP0;
        flagMode = 0;
    }
    else {
        AT91C_BASE_PIOA->PIO_CODR = TP0;
        flagMode = 1;
    }
#endif

    //* Pisca LED
    if(ledCount1 == 1000)
        AT91C_BASE_PIOA->PIO_CODR = LED1;
    if(ledCount1++ == 2000) {
        AT91C_BASE_PIOA->PIO_SODR = LED1;
        ledCount1 = 0;
    }

    if(conversionOver && !(bufferFull)) {
        conversionOver = 0;
        ADC_Convert();
        AT91C_BASE_PIOA->PIO_CODR = LED2;
    }
    else {
        // Apaga o led 3
        AT91C_BASE_PIOA->PIO_SODR = LED2;
    }
}

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_EDGE_TRIGGERED | 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;
}
```