Rafael Rodrigues Machado

Desenvolvimento das Fundações para Acessibilidade em Ambiente pré-OS

Sorocaba, SP

19 de Dezembro de 2018

Rafael Rodrigues Machado

Desenvolvimento das Fundações para Acessibilidade em Ambiente pré-OS

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCC-So) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Linha de pesquisa: Redes de Computadores e Engenharia de Software.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCC-So

Orientador: Prof. Dr. Gustavo M. D. Vieira

Sorocaba, SP

19 de Dezembro de 2018

Machado, Rafael Rodrigues

Desenvolvimento das Fundações para Acessibilidade em Ambiente pré-OS / Rafael Rodrigues Machado. -- 2018.
109 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, campus Sorocaba, Sorocaba
Orientador: Prof. Dr. Gustavo M. D. Vieira
Banca examinadora: Prof. Dr. Rodolfo Jardim deAzevedo, Profa. Dra. Yeda Regina Venturine
Bibliografia

1. BIOS. 2. UEFI. 3. Acessibilidade. I. Orientador. II. Universidade Federal de São Carlos. III. Título.

Ficha catalográfica elaborada pelo Programa de Geração Automática da Secretaria Geral de Informática (SIn).
DADOS FORNECIDOS PELO(A) AUTOR(A)
Bibliotecário(a) Responsável: Maria Aparecida de Lourdes Mariano – CRB/8 6979

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Rafael Rodrigues Machado, realizada em 19/12/2018:

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

Prof. Dr. Rodolfo Jardim de Azevedo
UNICAMP

Profa. Dra. Yeda Regina Venturini
UFSCar

Dedico este trabalho a meus pais, Luis e Elisabete, e ao meu irmão, Juliano, que me ensinaram tudo o que as escolas não tem a obrigação de ensinar;
Também a minha madrinha, Marlene, e meus avós, Antônio, Ezequiel e Maria, que me proporcionaram tantos bons momentos durante a infância;
A minha esposa, Evelin, que teve grande paciência e me apoiou constantemente durante essa jornada, e meu filho Henrique, que tanto me ensina;

Agradecimentos

Agradeço,

ao Prof. Dr. Gustavo M. D. Vieira por ter aceito a mentoria deste trabalho.

a meus mestres por todas as lições ensinadas dentro e fora dos tatames.
a vida por todas oportunidades e experiências que me trouxeram até aqui.

"Vida é crescimento.
Se paramos de crescer técnica e espiritualmente,
somos tão bons como se estivéssemos mortos.
(A arte da paz, Morihei Ueshiba (Fundador do Aikido))

Resumo

Pessoas portadoras de algum tipo de deficiência enfrentam um nível de dificuldade elevado para tarefas corriqueiras. Essa dificuldade se dá unicamente pelo fato de que em muitos casos a acessibilidade não ter sido considerada necessária quando dada tarefa foi elaborada. Um exemplo deste cenário é a configuração de BIOS de computadores, que não apresentam nenhuma opção que considere a existência de pessoas com algum tipo de necessidade mais específica, como por exemplo leitores de tela no caso de deficientes visuais. Com foco nessa necessidade, a proposta deste trabalho é a validação de que é possível tornar o ambiente pré sistema operacional acessível às pessoas com deficiência visual por meio do desenvolvimento do suporte a áudio em ambiente pré sistema operacional, a ser executado em equipamentos que possuam *BIOS* compatível com a especificação UEFI, e placas de áudio compatíveis com a especificação *High Definition Audio*. Como resultado foi disponibilizada para a comunidade uma prova de conceito de funcionamento de placas de áudio em ambiente pré-OS. Com a existência desse código base acredita-se que será possível aos responsáveis pela especificação UEFI iniciarem discussões relacionadas a criação da abstração para dispositivos de multimídia.

Palavras-chaves: Acessibilidade. Arquitetura de Computadores. BIOS. UEFI. *High Defnition Audio*.

Abstract

People with some kind of disability face a high level of difficulty for everyday tasks. This difficulty is due solely to the fact that in many cases accessibility was not considered necessary when the task was created. An example of this scenario is the BIOS configuration of computers, which do not present any option that considers the existence of people with some more specific need, such as screen readers in the case of visual impairement. Focusing on this need, this work validates that it is possible to make the pre-operating system's environment accessible to visually impaired people by developing the audio support in this environment, to be executed on systems that have BIOS compatible with the UEFI specification, and audio cards compatible with the High Definition Audio specification. As a result, a proof of concept for the operation of audio cards in a pre-OS environment has been made available to the community. With the existence of this code base it is believed that it will be possible for those responsible for the UEFI specification to start discussions related to creating the abstraction for multimedia devices.

Key-words: Accessibility. BIOS. Computers Architecture. UEFI. High Definition Audio.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACPI	*Advances Control and Power Interface*
BDS	*Boot Device Selection*
BIOS	*Basic Input and Output System*
DMA	*Direct Memory Access*
DXE	*Driver Execution Environment*
HDA	*High Definition Audio*
ONU	Organização das Nações Unidas
PCIe	*Peripheral Component Interconnect Express*
PEI	*Pre-EFI Initialization*
PWM	*Pulse Width Modulation*
RAID	*Redundant Array of Independent Disks*
ROM	*Read Only Memory*
SEC	*Security*
UEFI	*Unified Extensible Firmware Interface*

Sumário

Introdução

Nos últimos anos, a palavra inclusão tem sido amplamente utilizada para denotar a ideia de que todas as pessoas são iguais perante a sociedade e, independente de alguma necessidade específica, devem ter as mesmas oportunidades que as demais. Com o intuito garantir estes direitos às pessoas com necessidades especiais, a Organização das Nações Unidas criou a "Convenção Sobre os Direitos das Pessoas com Deficiência" (HENDRICKS, 2007). Essa convenção tem por finalidade fazer com que a humanidade deixe de ver as pessoas com necessidades especiais como seres incapazes e dignos de caridade, passando a tratá-los como membros capazes de ter uma participação ativa na sociedade com direitos e deveres, como pode ser encontrado no prefácio da supracitada convenção, nos itens "e" e "n" apresentados a seguir:

"e) Reconhecendo que a deficiência é um conceito em evolução e que a deficiência resulta da interação entre pessoas com deficiência e as barreiras devidas às atitudes e ao ambiente que impedem a plena e efetiva participação dessas pessoas na sociedade em igualdade de oportunidades com as demais pessoas..."

"n) Reconhecendo a importância, para as pessoas com deficiência, de sua autonomia e independência individuais, inclusive da liberdade para fazer as próprias escolhas,"

Tais itens estão fortemente ligados ao conceito de acessibilidade, que tem como meta criar as formas necessárias para que as pessoas, independentemente de terem algum tipo de necessidade especial ou não, consigam realizar uma dada tarefa com a maior qualidade possível. Para que isso seja possível foi criado o conceito de "Usabilidade Universal", que visa permitir que mais de 90% das pessoas consigam utilizar uma determinada tecnologia sem grandes dificuldades (MEISELWITZ; WENTZ; LAZAR, 2009) (SHNEIDERMAN, 2000).

Para que tal conceito pudesse ser aplicado às atividades a serem realizadas em ambientes computacionais um novo tipo de software foi criado, chamado leitor de tela (do inglês *screen reader*). Um dos propulsores desta tecnologia, criado pela empresa IBM na década de 90, foi o AccessDos, o qual permitia a acessibilidade no sistema operacional DOS aos deficientes visuais, e que foi distribuído livremente a seus clientes (VANDERHEIDEN, 2008), abrindo assim um novo horizonte às tecnologias assistivas, com a futura criação de diversos outros leitores de tela. Desta forma, através de comandos sonoros, os deficientes visuais poderiam se orientar no ambiente virtual em que estão executando uma determinada tarefa, podendo assim realizar a mesma de forma natural e eficiente.

Sendo assim, a participação de pessoas com deficiência visual na área de tecnologia da informação passou a aumentar cada vez mais, chegando a situação atual, na qual

existem pessoas com deficiência visual atuando em atividades de desenvolvimento de software, administração de redes, gerenciamento de projetos e manutenção de computadores (MACHADO, 2016). Dentre as tarefas mencionadas, existe a possibilidade de que tal atividade exija a alteração das configurações de uma porção de software conhecida como BIOS. Este é o software responsável pela inicialização dos computadores, além de ser o ponto de entrada para configuração de opções avançadas existentes nas plataformas. A possibilidade de alteração de tais configurações é algo primordial na execução de atividades mais complexas, e que até o momento não levou em consideração as necessidades relacionadas a acessibilidade apresentadas por estes profissionais.

A proposta apresentada por essa dissertação foi motivada devido a uma postagem em um fórum de discussões relacionadas ao desenvolvimento de BIOS. Este fórum, denominado edk2 (TIANOCORE, 2018a), é o centralizador de discussões relacionadas a tecnologia UEFI, caracterizada como uma evolução dos BIOS e definido em especificação, no qual seus membros são os responsáveis pelo desenvolvimento de um BIOS *open source* chamado Tianocore (TIANOCORE, 2018c), apresentado como uma implementação de BIOS compatível com a especificação UEFI.

Dentre as diversas discussões apresentadas nesse fórum, no dia 11 de abril de 2014, foi postado por Matthew Bradley um questionamento sobre a possibilidade de desenvolvimento de um leitor de tela para que o mesmo pudesse fazer alterações no BIOS de seu computador, nesse caso mudanças em configurações, sem a necessidade de ajuda (como apresentado no Apêndice A). Durante a discussão um dos membros da comunidade, bastante respeitado visto seu grande conhecimento, fez sua contribuição dizendo que se os leitores de tela fossem unicamente componentes relacionados a software, seu desenvolvimento seria difícil devido a inexistência de uma abstração para as funcionalidades de áudio nos BIOS. Desta forma, mesmo sendo os leitores de tela um conceito com mais de 20 anos desde sua definição, atualmente inexistem BIOS acessíveis aos deficientes visuais.

Com estas informações, foram realizadas verificações nas implementações existentes no projeto Tianocore e constatou-se a inexistência de um *driver* para dispositivos de áudio, bem como a inexistência de qualquer referência a componentes de áudio dentre os protocolos definidos na especificação UEFI (UEFI Forum, 2014).

Visando um melhor entendimento do número de pessoas que seriam beneficiadas caso um leitor de tela para ambiente pré-OS fosse desenvolvido, foi realizada uma pesquisa que contou com a participação de membros de um grupo de desenvolvedores brasileiros, composto por pessoas com deficiência visual (MACHADO, 2016). Nesta pesquisa obteve-se como resultado a confirmação de 16 membros do grupo que comprariam um computador que tivesse suporte a acessibilidade em ambiente pré-OS. Apesar de não representar uma grande parcela da população, esse resultado demonstra o interesse de algumas pessoas na proposta do projeto. Esse pode ser um grupo pequeno, mas digno de respeito e admiração.

Vale ressaltar que não foram realizadas pesquisas em outros países, porém acredita-se que mais pessoas poderiam ser auxiliadas pela solução proposta quando considerando o cenário mundial, a ser lembrado que o ponto inicial dessa pesquisa foi iniciado por um cidadão não residente no Brasil.

Para o desenvolvimento de um leitor de tela, seja em ambiente pré-OS ou não, diversos componentes de software precisam trabalhar em conjunto, sendo o início dessa cadeia um *driver* de áudio. Tal componente é necessário para que a controladora de áudio, e o codec sejam capazes de receber as informações e dados necessário para a reprodução de um determinado som, responsável pela orientação do usuário durante a realização de seus afazeres. Sendo assim, constatou-se que a primeira barreira a ser ultrapassada para a inclusão de acessibilidade em ambiente pré-OS com foco em deficientes visuais, seria a possibilidade de realização de operações relacionadas a áudio. Tarefa essa exercida pelos *drivers* de áudio dos sistemas operacionais, porém inexistentes em ambiente pré-OS.

Esta dissertação de mestrado disponibiliza para a comunidade uma prova de conceito de funcionamento de placas de áudio em ambiente pré-OS. Com a existência desse código base acredita-se que será possível aos responsáveis pela especificação UEFI iniciarem discussões relacionadas a criação da abstração para dispositivos de multimídia. Abstração essa inexistente até o momento, criando desta forma as fundações necessárias para o desenvolvimento futuro de um leitor de tela para ambiente pré-OS.

Os resultados desta dissertação apresentam-se na forma de um código de referência capaz de processar *streams* de áudio em ambiente pré-OS, sendo tais funções encapsuladas em uma biblioteca. Esta biblioteca será disponibilizada a comunidade edk2 posteriormente sob licença BSD, permitindo assim sua utilização por parte dos interessados sem que obrigações legais impactem a evolução de tal conceito. Devido ao fato de grande parte dos BIOS existentes serem de código proprietário, a criação do projeto apresentado com licença GPL poderia acarretar em dificuldade de adoção por parte dos fabricantes de BIOS, sendo este o motivo da escolha da licença BSD para o projeto.

Visando a eliminação ou ao menos a redução de incompatibilidades, este trabalho limitou seu escopo a dispositivos de áudio que seguem a especificação *Intel High Definition Audio* (INTEL®, 2010), localizadas em dispositivos os quais seu firmware (entende-se BIOS nesse caso), siga a especificação UEFI (UEFI Forum, 2014).

As seções desta dissertação estão organizadas da seguinte forma. Seções 1 e 1.1 contendo detalhes relacionados as motivações as quais resultaram no desenvolvimento desta dissertação de mestrado. Seções 1.2, 1.3, 1.4 e 1.5 contendo explicações detalhadas relacionadas às tecnologias, as quais se fazem necessárias para o correto entendimento do projeto apresentado. Dentre as explicações são apresentadas algumas informações relacionadas à arquitetura de computadores, permitindo assim o correto entendimento do processo de inicialização de uma máquina moderna, além de ser apresentado em detalhes

a arquitetura HDA e a tecnologia UEFI. Seções 2, 2.1 contendo informações relacionadas ao detalhamento do projeto apresentado bem como sua arquitetura. Seções 2.2 e 2.3 nas quais são apresentados os resultados obtidos.

1 Fundamentos

Neste capítulo serão abordados conhecimentos necessários para o correto entendimento do trabalho, através de explicações técnicas detalhadas sobre o funcionamento das tecnologias e conceitos utilizados no mesmo.

1.1 Leitores de Tela

As tecnologias assistivas com foco em pessoas com deficiência visual começam muito antes da década de 90, como apresentado anteriormente. No ano de 1966, um invento bastante útil foi patenteado, denominado Optacon (GOLDISH; TAYLOR, 1974; LINVILL, 1966), o qual tinha a finalidade de converter imagens em sinais táteis, como apresentado na Figura 1. Tal avanço permitiu a independência dos deficientes visuais, que a partir deste momento, não precisariam mais de ajuda para ler a informações apresentadas em um monitor, e poderiam também ler livros que não estivessem em Braile, visto que o Optacon permitia que todo tipo de figura fosse convertido em um sinal tátil, não se limitando a telas e monitores.

Figura 1: Optacon. Fonte: (LINVILL, 1966)

Seu funcionamento se dava através da conversão da luz refletida por uma dada figura, a qual era analisada através de uma cabeça de leitura conectada a uma célula com pontos vibratórios. A figura convertida era então apresentada ao usuário através de vibrações em alguns dos pontos independentes existentes na célula braile na qual os dedos do usuário estariam posicionados. Tal entendimento pode ser obtido através da patente a qual descreve o funcionamento do Optacon em sua seção de reivindicações, como apresentado a seguir, após tradução para o idioma português:

> *Aparelho para converter uma figura óptica numa figura vibratória que pode ser sentida via tato compreendendo uma pluralidade de palhetas piezoeléctricas vibratórias separadas e dispostas numa matriz, ... Com uma amplitude representativa da luz de uma região diferente da referida figura, ... (LINVILL, 1966).*

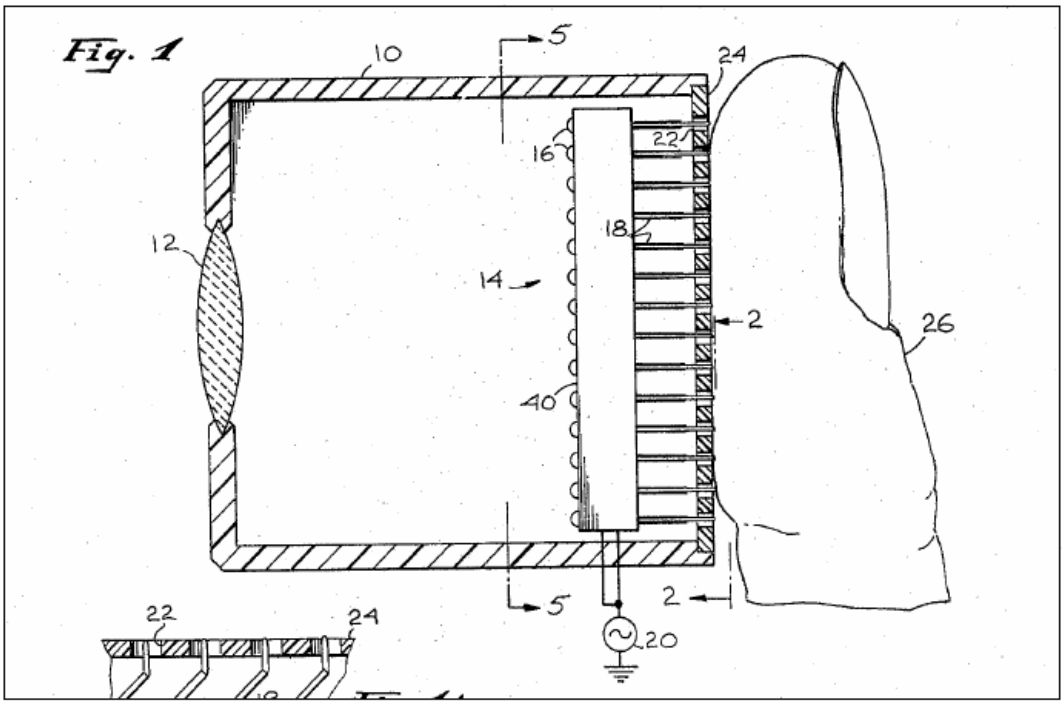

Figura 2: Ponto de contato Optacon. Fonte: (LINVILL, 1966)

Porém como visto, este equipamento não era integrado aos computadores daquela época, fazendo assim com que os deficientes visuais tivessem que adquirir um equipamento adicional para ter acesso às informações. Considerando o fato de que naquela época a disponibilidade de computadores pessoais ainda era muito pequena, a possibilidade de acesso às informações através dos mesmos ainda se apresentada como restrita a um pequeno número de pessoas. Nos dias de hoje, após vários avanços tecnológicos e a disponibilização de computadores de bolso na forma de celulares e *tablets*, foram criadas várias ferramentas com foco em acessibilidade para deficientes visuais, sem que seja necessária a aquisição de

equipamentos externos. Dentre as ferramentas podem ser destacados os leitores de tela (ALLEN et al., 1981).

A função deste tipo de ferramenta é guiar o usuário com deficiência visual descrevendo o ambiente em que se encontra, bem como todas as ações executadas, e os alertas apresentados pelo sistema operacional. Isso permite que um usuário consiga realizar todas as tarefas possíveis dentro do ambiente digital no qual estiver trabalhando. Estas tarefas incluem a leitura de documentos, acesso a *e-mails* e qualquer outro tipo de atividade, incluindo tarefas avançadas como desenvolvimento de softwares e alteração de configurações do sistema operacional.

A Figura 3 apresenta o funcionamento em alto nível de um leitor de tela.

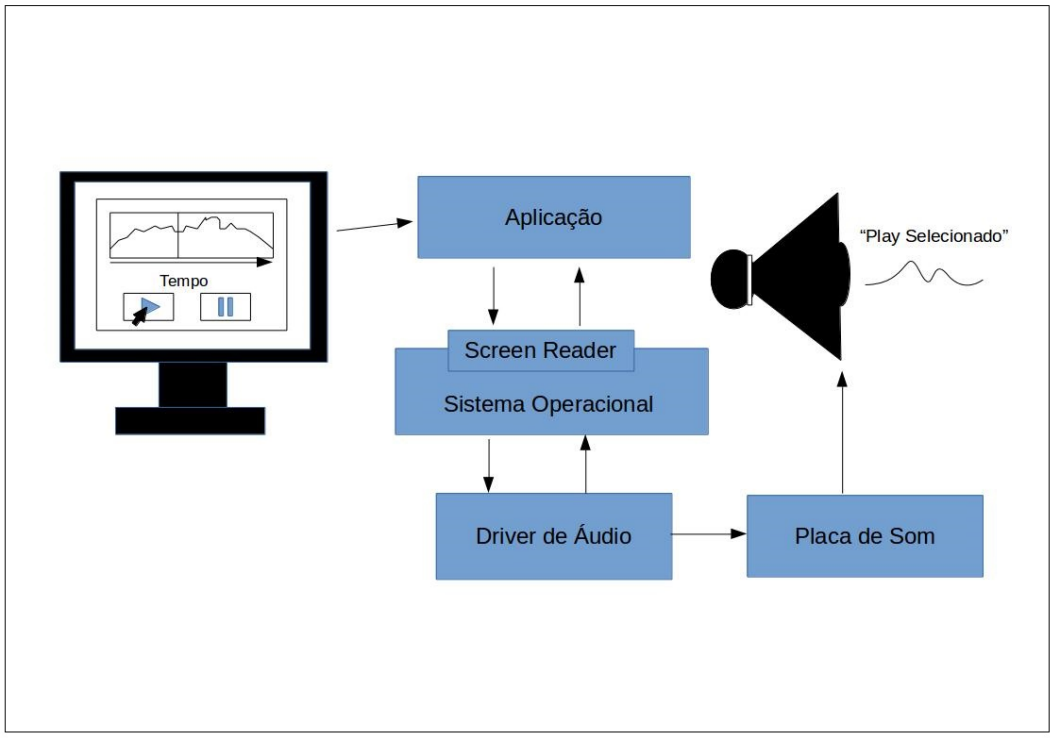

Figura 3: Funcionamento de um leitor de tela

A figura representa o funcionamento de um leitor de tela hipotético no momento em que um determinado componente existente na interface gráfica de um tocador de músicas recebe foco. Nesse momento o leitor de tela faz uso de diversas possíveis APIs disponibilizadas pelos sistemas operacionais, como é o caso da Microsoft Active Accessibility nos sistemas operacionais Windows, ou a iaccessible2 no caso de sistemas operacionais Linux. Possibilitando assim que eventos em interfaces gráficas sejam notificados, permitindo que em seguida as informações do componente atualmente em foco sejam enviadas a um sintetizador de voz, que será responsável por criar a onda sonora que contém as informações do componente em formato de áudio. Esta informação em formato sonoro é posteriormente enviada ao *driver* de áudio que tem a responsabilidade de enviar tal informação até a

controladora de áudio, de forma que a mesma possa realizar a conversão destes sinais sonoros digitais em pulsos analógicos que serão finalmente reproduzidos pelos alto-falantes.

Desta forma é observada a elevada complexidade da conversão das informações existentes em uma dada aplicação em um formato acessível, tarefa esta realizada pelos leitores de tela, trabalhando em conjunto com o sistema operacional, APIs de acessibilidade, sintetizadores de voz, *drivers* de áudio e o *hardware*.

1.1.1 NVDA

Dentre os leitores de tela mais utilizado atualmente destaca-se o NVDA. Este software foi inicialmente desenvolvido por Michael Curran e James Teh, dois deficientes visuais que se conheceram ainda na infância, e que depois de muitos anos se juntaram para melhorar a acessibilidade em computadores à portadores de deficiência visual. Essa junção deu origem ao leitor de tela NVDA, sigla para *NonVisual Desktop Access*, atualmente utilizado por várias pessoas ao redor do mundo em mais de 120 países e cuja tradução esta disponível em mais de 43 idiomas (NVACCESS, 2015). Por tratar-se de um software *open source*, vários voluntários fazem com que sua evolução continue de forma independente e focada na usabilidade e inclusão.

Para melhor entendimento do funcionamento de um leitor de tela, será realizada a descrição dos códigos existentes no repositório do projeto NVDA, permitindo assim o entendimento da interação entre os diversos componentes existentes no mesmo. Esta descrição tem como objetivo apresentar os passos necessários para que um dado áudio possa ser enviado, a partir de um leitor de tela, nesse caso o NVDA, até seu destino final, ou seja a saída de áudio disponível no equipamento. Diferentemente dos leitores de tela existentes no passado, os quais precisavam atuar de forma intrusiva às operações realizadas pelos sistemas operacionais através da interceptação de eventos de teclado e tratamento das informações apresentadas pelas controladoras de vídeo, nos sistemas operacionais atuais existe uma arquitetura pensada no conceito de acessibilidade. No caso dos sistemas operacionais *Windows*, tal arquitetura é denominada MSAA (sigla para *Microsoft Active Acessibility*), que tem a função de exportar informações para aplicações externas ao sistema operacional com foco em acessibilidade. Tais funcionalidades podem ser utilizadas pelas aplicações através da utilização das funções disponibilizadas pela DLL Oleacc.ddl (MICROSOFT®, 2018b).

No NVDA, a utilização desta infraestrutura pode ser confirmada devido a existência do arquivo oleacc.py, o qual apresenta o seguinte trecho de código:

```
from comtypes import *
from comtypes.automation import *
import comtypes.client
import winKernel
```

```
import winUser
# Include functions from oleacc.dll in the module namespace.
m=comtypes.client.GetModule('oleacc.dll')
```

Para a conversão das informações fornecidas pelo sistema operacional em informações sonoras, o NVDA precisa primeiro tratá-las, tarefa esta realizada por um projeto denominado eSpeak (DUDDINGTON, 1995). O eSpeak é uma implementação *open source* de um sintetizados de voz, responsável por converter uma cadeia de caracteres em fonemas, os quais são convertido em um *stream* de áudio. Este processo de conversão de informações em formato texto para formato sonoro se dá através da função speak, disponível ao NVDA através da um *driver* de sintetização de voz (localizado em /nvda/source/synthDrivers/espeak.py), conforme apresentado no trecho de código a seguir:

```
def _speak(text):
        global isSpeaking
        uniqueID=c_int()
        isSpeaking = True
        # eSpeak can only process compound emojis
        #when using a UTF8 encoding
        text=text.encode('utf8',errors='ignore')
        flags = espeakCHARS_UTF8 | espeakSSML | espeakPHONEMES
        return espeakDLL.espeak_Synth(text,0,0,0,0,flags,
                byref(uniqueID),0)
```

Como apresentado, a chamada da função se dá através da utilização da variável espeakDLL, a qual no caso do NVDA aponta para uma dll obtida através da compilação do projeto espeak mencionado anteriormente. Internamente o eSpeak realiza a conversão das cadeias de texto em fonemas através da função text_decoder_decode_string_multibyte, que possui a seguinte implementação:

```
espeak_ng_STATUS
text_decoder_decode_string_multibyte(espeak_ng_TEXT_DECODER *decoder,
                                     const void *input,
                                     espeak_ng_ENCODING encoding,
                                     int flags)
{
        switch (flags & 7)
        {
        case espeakCHARS_WCHAR:
                return text_decoder_decode_wstring(decoder,
                           (const wchar_t *)input, -1);
        case espeakCHARS_AUTO:
                return text_decoder_decode_string_auto(decoder,
                           (const char *)input, -1, encoding);
        case espeakCHARS_UTF8:
                return text_decoder_decode_string(decoder,
                           (const char *)input, -1,
```

```
                        ESPEAKNG_ENCODING_UTF_8);
        case espeakCHARS_8BIT:
                return text_decoder_decode_string(decoder,
                            (const char *)input, -1, encoding);
        case espeakCHARS_16BIT:
                return text_decoder_decode_string(decoder,
                            (const char *)input, -1,
                            ESPEAKNG_ENCODING_ISO_10646_UCS_2);
        default:
                return ENS_UNKNOWN_TEXT_ENCODING;
        }
}
```

As chamadas existentes nesta função tem como objetivo preencher a estrutura *decoder* com as informações necessários para que posteriormente os fonemas sejam convertidos em sinais sonoros, conversão esta realizada através da função Wavegen(), disponível na biblioteca espeak, a qual se utiliza das informações processadas anteriormente para criação das ondas sonoras a serem enviadas posteriormente ao dispositivo de áudio.

Após a criação das sequências de bytes os quais representam um áudio a ser reproduzido por uma voz sintetizada, é realizado o envio de tal cadeia de bytes ao dispositivo de áudio. Esta ação é realizada através da criação de um evento que tem a finalidade de disparar a ação sonora, como apresentado na função a seguir (no momento em que a função dispatch_audio é executada):

```
static int create_events(short *outbuf, int length,
                         espeak_EVENT *event_list)
{
        int finished;
        int i = 0;

        // The audio data are written to the output device.
        // The list of events in event_list (index: event_list_ix)
        //is read:
        // Each event is declared to the "event" object which
        //stores them internally.
        // The event object is responsible of calling the external
        //callback
        // as soon as the relevant audio sample is played.

        do { // for each event
                espeak_EVENT *event;
                if (event_list_ix == 0)
                        event = NULL;
                else
                        event = event_list + i;
                finished = dispatch_audio((short *)outbuf, length,
```

```
                                        event);
                length = 0; // the wave data are played once.
                i++;
        } while ((i < event_list_ix) && !finished);
        return finished;
}
```

Esta função, dispatch_audio, por sua vez realiza o envio da cadeia de bytes a placa de som, através da função audio_object_write():

```
#ifdef HAVE_PCAUDIOLIB_AUDIO_H
        if (outbuf && length && a_wave_can_be_played) {
                int error = audio_object_write(my_audio,
                                               (char *)outbuf,
                                               2*length);
                if (error != 0)
                        fprintf(stderr, "error: %s\n",
                                audio_object_strerror(my_audio, error));
        }
#endif
```

Como pode ser verificado no trecho de código apresentado, o envio das informações ao dispositivo de áudio depende da utilização de outra biblioteca adicional, denominada pcaudiolib (DUDDINGTON, 2016) que cria um ponto de acesso até o dispositivo de áudio através da função create_audio_device_object, apresentada a seguir:

```
struct audio_object *
create_audio_device_object(const char *device,
                           const char *application_name,
                           const char *description)
{
        struct audio_object *object;
#if defined(_WIN32) || defined(_WIN64)
        if ((object = create_xaudio2_object(device, application_name,
              description)) != NULL)
                return object;
#else
#if defined(__APPLE__)
        if ((object = create_coreaudio_object(device, application_name,
              description)) != NULL)
                return object;

#else
        if ((object = create_pulseaudio_object(device,
                   application_name, description)) != NULL)
                return object;
        if ((object = create_alsa_object(device, application_name,
                                         description)) != NULL)
```

```
                return object;
        if ((object = create_qsa_object(device, application_name,
                                        description)) != NULL)
                return object;
        if ((object = create_oss_object(device, application_name,
                                        description)) != NULL)
                return object;
#endif
#endif
        return NULL;
}
```

Esta função tem como objetivo a criação de um ponto de comunicação entre o *driver* de áudio existente na plataforma e outros aplicativos que desejam fazer uso de tal equipamento, podendo o mesmo ser suportado por sistemas operacionais Apple, Windows e Linux. A Figura 4 apresenta a interação entre as bibliotecas apresentadas nas descrições anteriores.

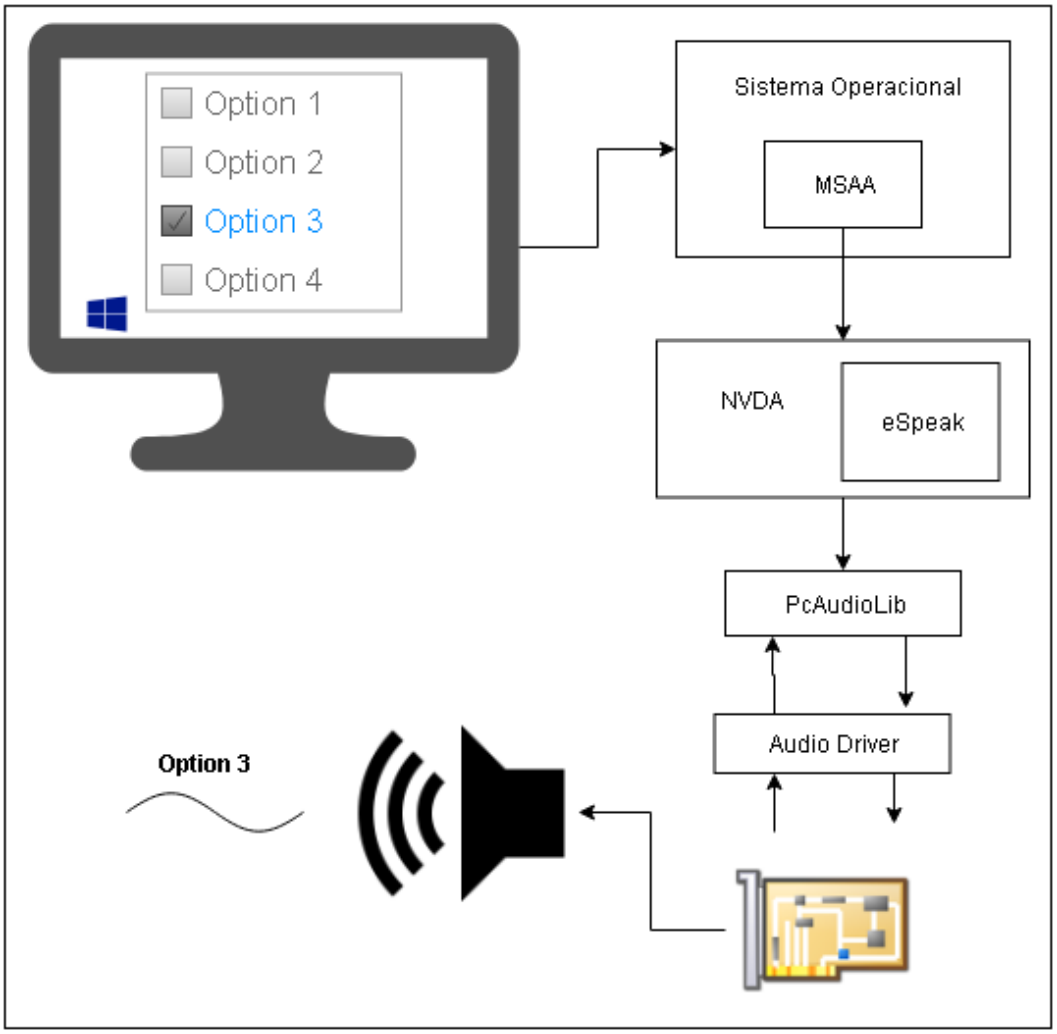

Figura 4: Funcionamento do NVDA

Como apresentado, para que o uso de um leitor de tela seja possível é necessário que os seguintes pré-requisitos sejam satisfeitos no ambiente em questão:

1. Um sistema operacional deve estar instalado (Windows, Linux, Apple)
2. O *driver* da placa de áudio deve estar instalado
3. Alguma forma de comunicação com tal *driver* deve estar disponível (nos sistemas operacionais Windows isso se dá através da interface IXAudio2 (MICROSOFT®, 2018a)
4. Um leitor de tela deve estar instalado (neste caso o NVDA)

1.2 Acessibilidade no ambiente pré-OS

O problema existente é que nem todos os softwares são acessíveis, e um desses softwares é o BIOS dos computadores. O BIOS é responsável por realizar a inicialização básica do equipamento. Esse software é também conhecido como *firmware* devido ao fato de no passado o mesmo não podia ser atualizado, algo que já não é realidade atualmente, porém a associação entre o termo *firmware* e o termo BIOS ainda persiste. Detalhes sobre os BIOS podem ser encontrados na Seção 1.3. Essa falta de acessibilidade ocorre devido ao fato de que, no ambiente pré sistema operacional (pré-OS) não existe um *driver* para a placa de áudio carregado, e portanto não é possível que um leitor de tela e seu sintetizador de vozes enviem informações que posteriormente seriam convertidas em sinais analógicos audíveis.

Considerando o fato de que muitos profissionais com deficiência visual trabalham com a manutenção de computadores, se torna inviável a execução de determinadas tarefas, como por exemplo a configuração de uma placa RAID em um servidor. Outro exemplo bastante simples e que demonstra a necessidade da acessibilidade descrita é o auxilio na seleção do dispositivo de inicialização a ser utilizado. Outra tarefa de grande importância e impossível ao deficiente visual é a realização de atualizações do BIOS em ambiente pré-OS, sendo esta tarefa de extrema importância às organizações no reparo de equipamentos e para garantia da segurança da infraestrutura da empresa, visto que os BIOS tem se tornado grande ponto de atenção por parte dos especialistas em segurança digital.

Estas são algumas dentre várias tarefas que poderiam ser realizas rapidamente quando em ambiente pré-OS, sem a necessidade de instalação de um sistema operacional no equipamento. Porém estas tarefas podem ser executadas unicamente por uma pessoa que não tenha deficiência visual.

Pesquisas apontam que existem no mundo cerca de 285 milhões de pessoas com algum tipo de deficiência visual, dos quais 39 milhões possuem cegueira (World Health

Organization, 2014), portanto a inexistência da acessibilidade nas aplicações pré-OS representa uma clara discriminação para com estas pessoas, pois as mesmas não tem a liberdade de utilizar seus computadores da mesma forma que as pessoas que não possuem dadas condições.

Apesar da necessidade de suporte acessível aos usuários de ambientes pré-OS, infelizmente as empresas não percebem que o gasto necessário para seu desenvolvimento é compensado por um número maior de usuários interessados em seu produto (SHERMAN, 2001) (W3C, 2015). Esse público maior passa a englobar também os parentes das pessoas com necessidades mais específicas, que se tornam novos clientes e passam a confiar e priorizar marcas e produtos que auxiliam seus entes queridos.

1.3 BIOS

Para melhor entendimento da solução proposta, nessa seção são explicados alguns conceitos relacionados aos BIOS, bem como sua função e funcionamento. Desde a década de 60, com o inicio do surgimento dos computadores, os *firmwares* responsáveis por inicializar esses equipamentos eram escritos totalmente em linguagem *Assembly*. A tais *firmwares* foi dado o nome BIOS, do inglês *Basic Input and Output System*. Dentre as tarefas a serem executadas pelo BIOS durante a inicialização de um determinado equipamento podem ser destacadas as seguintes:

1. Limpar memória cache;
2. Detectar memória RAM;
3. Inicializar pilha;
4. Configurar tratadores SMM (*System Management Mode*);
5. Configurar registradores MSR (*Model Specific Registers*);
6. Configurar registradores MTRR (*Memory Type Range Registers*);
7. Inicializar *chipset* (certamente uma das tarefas mais complicadas);
8. Inicialização e enumeração do barramento PCIe;
9. Inicialização e enumeração do barramento USB;
10. Inicializar controladoras de disco (caso exista alguma configuração de RAID, essa tarefa fica ainda mais complicada);
11. Montar a parte dinâmica do *firmware* (Tabelas ACPI));

12. Inicializar *frame buffer* (vbios);
13. Detectar *firmwares* externos e executá-los (*Option* ROM);
14. Verificações de segurança em baixo nível (senha do BIOS, *tamper*);
15. Detectar dispositivos a serem inicializados (*boot device selection*);
16. Mover o processador para o modo protegido (na arquitetura Intel os processadores sempre são iniciados em modo real quando ligados por questões de compatibilidade);
17. Atualização de *microcode* (correção de erros do processador);
18. Criação de tratadores para as interrupções (por exemplo, a Int 15/AX=E820h, que retorna o mapeamento de memória);
19. Outras tarefas especificas de cada plataforma.

As tarefas realizadas pelo BIOS são de extrema importância para que um dado equipamento possa estar funcional, permitindo assim que o *bootloader* e em seguida o sistema operacional sejam executados. Com o passar do tempo, os equipamentos começaram a ficar cada vez mais complexos, com a inclusão de novos tipos de controladores e tecnologias, bem como sua miniaturização. Essa evolução fez com que a complexidade dos *firmwares* aumentasse. Desta forma os *firmwares*, que já não eram softwares triviais, passaram a se tornar um emaranhado de linhas de código em *Assembly*.

Toda esta complexidade passou a ser um problema para as grandes empresas. Dentre os problemas decorrentes dessa complexidade podem ser destacados:

1. Dificuldade em encontrar mão de obra especializada
2. Dificuldade em personalizar *firmwares* para novos produtos
3. Difícil reutilização de código
4. Difícil portabilidade entre plataformas

Para exemplificar melhor todos os fatores apresentados, basta imaginar o que ocorria quando era desenvolvido um novo equipamento por parte de uma determinada empresa. Supondo que este novo equipamento apresentasse alguma nova tecnologia, é aceitável acreditar que a inicialização do *hardware* desse equipamento deveria seguir um fluxo diferente de outras plataformas já existentes, visto que a mesma necessitaria da inclusão de uma nova controladora para gerenciar essa nova tecnologia hipotética.

Outro problema ocorria quando um dado firmware precisava ser portado para uma nova plataforma. Devido ao fato de que cada fabricante desenvolvia seus *firmwares* a seu

modo, as interrupções disponibilizadas para cada arquitetura não eram as mesmas, e com isso seria necessário reescrever cada aplicação para cada plataforma. Adicionalmente, o suporte a múltiplas plataformas utilizando a linguagem *Assembly* é uma tarefa bastante difícil.

O exemplo abaixo apresenta como seria uma simples aplicação *Hello World* em linguagem *Assembly* para arquitetura x86:

```
variable:
   .message   db   "Hello World$"
code:
   mov  ah, 0x9
   mov  dx, offset .message
   int  0x21
   ret
```

Para a simples tarefa de apresentar um texto na tela, é necessário executar a interrupção 0x21, através da subfunção 0x9. O mesmo código, quando desenvolvido para arquitetura ARM, é apresentado a seguir:

```
.global _start
_start:
  MOV R7, #4
  MOV R0, #1
  MOV R2, #12
  LDR R1, =string
  SWI 0
  MOV R7, #1
  SWI 0
  .data
string:
  .ascii "Hello World"
```

1.3.1 BIOS UEFI

Para tentar resolver esses problemas, no ano de 2000 a empresa *Intel Corporation* criou o conceito conhecido como *Extensible Firmware Interface* (EFI), que posteriormente foi adotado por várias empresas e rebatizado para *Unified Extensible Firmware Interface* ou UEFI (UEFI Forum, 2014).

Seguindo esta especificação os fabricantes passariam a implementar uma interface comum de comunicação entre o *firmware* e as aplicações, *drivers* e sistemas operacionais. Com essa padronização uma aplicação *Hello World*, poderia ser implementa de forma portável da seguinte forma:

```
#include <efi.h>
```

```
EFI_STATUS main(EFI_HANDLE ImageHandle,
                EFI_SYSTEM_TABLE *SystemTable)
{
    SystemTable->ConOut->OutputString(SystemTable->ConOut,
                                      L"Hello World\r\n");
    return EFI_SUCCESS;
}
```

Para o suporte a diferentes plataformas detalhes específicos de cada arquitetura são resolvidos em tempo de compilação, deixando a cargo do *firmware* e das ferramentas de compilação todas as complexidades da plataforma.

Juntamente com a especificação UEFI, outra especificação bastante importante para a evolução dos *firmwares* é a especificação PI (*Plataform Initialization*), que define as etapas a serem seguidas durante o processo de *boot*, e os módulos essenciais a serem carregados em cada etapa em um equipamento cujo *firmware* é compatível com estas especificações. As etapas definidas pela especificação PI são apresentadas na Figura 5.

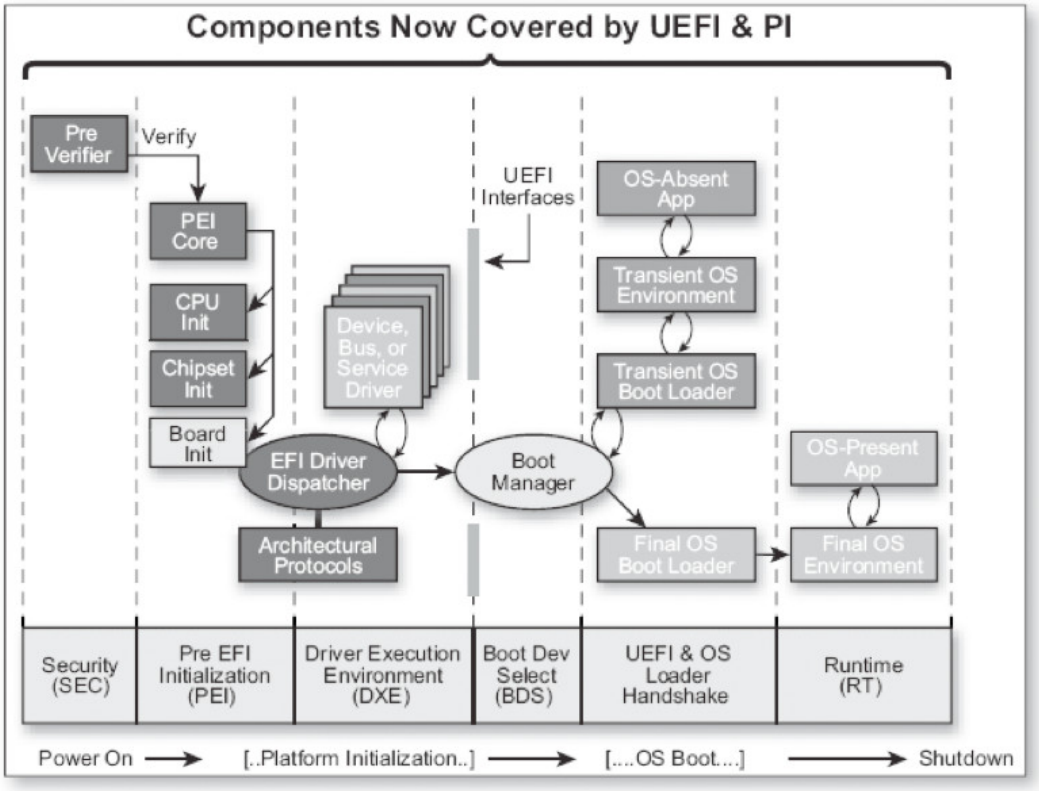

Figura 5: Processo de Inicialização de um Sistema PI/UEFI. Fonte: (VINCENT; MICHAEL; SURESH, 2010)

Cada uma das etapas apresentadas tem uma função bastante específica no processo de *boot*. A Tabela 1 apresenta a função de cada etapa no processo de inicialização do equipamento.

Tabela 1: Atribuições das etapas do processo de boot

Etapa	Atribuições
SEC	- Tratar todos eventos de inicio do sistema - Criar um local temporário a ser utilizado como memória - Garantir requisitos de segurança para a próxima etapa (*root of trust*) - Passar informações importantes para a próxima etapa
PEI	- Inicia memória permanente - Realizar a divisão da memória em blocos para função específica (*memory map*) - Definir locais onde outros *firmwares* podem estar armazenados (*option roms*) - Passar o controle para a próxima etapa
DXE	- Produzir *boot services* - Produzir *runtime services* - Descobrir e executar os *drivers* DXE na ordem correta
BDS	- Implementar politica de *boot* da plataforma (qual dispositivo será iniciado primeiro)

A etapa DXE é o momento no qual os *drivers* serão carregados durante a inicialização de um equipamento com *firmware* compatível com a especificação PI, permitindo assim que componentes e periféricos sejam iniciados corretamente, permitindo que tais periféricos sejam utilizados posteriormente em ambiente pré-OS. Portanto nessa etapa são inicializados por exemplo controladoras SATA, dispositivos *PCI Express*, placas de vídeo entre outros inúmeros tipos de controladoras e periféricos existentes. É neste momento que os *drivers* criam os protocolos definidos pela especificação UEFI, como por exemplo o protocolo EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL (UEFI Forum, 2014, p. 633) criado pelos *drivers* PCIe e que permite a comunicação com estes dispositivos, e os serviços globais denominados *Global Boot Services* (UEFI Forum, 2014, p. 113) definidos pela especificação UEFI e disponíveis em todas as plataformas compatíveis com esta especificação.

Do ponto de nível de acessos a plataforma não existem diferenças entre um *driver* UEFI e uma aplicação UEFI, visto que não existe o conceito de *user mode* e *kernel mode* em ambiente pré-OS.

Outro ponto importante é relacionado a compatibilidade com a especificação PI/UEFI, que não se restringe a notebooks e desktops. Qualquer equipamento pode possuir um firmware que segue essa especificação. É possível encontrar no mercado celulares, tablets e kits de desenvolvimento que possuem firmware compatível com estas especificações, apresentando unicamente algumas alterações necessárias devido a restrições de custo e hardware. Com isso a portabilidade entre equipamentos e plataformas torna-se uma tarefa muito mais simples do que antigamente.

1.3.2 Funcionamento do Chipset

Como apresentado anteriormente, grande parte do trabalho realizado pelo BIOS durante a inicialização de um determinado equipamento consiste na correta configuração das controladoras responsáveis pela comunicação com componentes externos ao processador.

Vale ressaltar, o fato de no momento em que é energizado, o processador não possui conhecimento do hardware o qual irá controlar, motivo esse responsável pela existência do BIOS, visto que são necessárias instruções para a correta inicialização de todo o hardware e componentes externos ao processador. Dentre tais componentes podem ser destacados os dispositivos de armazenamento USB gerenciados através das controladoras AHCI, EHCI e XHCI, dispositivos de armazenamento SATA gerenciados através da controladora SATA, a própria memória RAM gerenciada pela controladora de memória, e dispositivos PCIe controlados pelo *root complex* e *bridges* PCIe (mais informações sobre a arquitetura PCI Express podem ser encontradas na Seção 1.4).

As informações pertinentes às controladoras existentes em um dado modelo de *chipset* podem ser obtidas através de acesso a documentação fornecida pelo fabricante. Vale ressaltar que nem todos os fabricantes fornecem tal documentação e em muitos casos a documentação fornecida não apresenta todos os detalhes necessários para usar os dispositivos corretamente. Porém, o desenvolvimento dessa dissertação de mestrado não necessita de nenhum acordo de confidencialidade, permitindo assim seu desenvolvimento com base em informações abertas ao público.

A Figura 6 apresenta as controladoras existentes em um *chipset*, sendo esse o mesmo modelo de *chipset* presente no equipamento utilizado para validação da prova de conceito objeto desse trabalho.

O *chipset* possui em seu interior controladoras com endereço fixo, facilitando assim a localização dos dispositivos durante o processo de inicialização. Dentre as controladoras, as mais importantes para o desenvolvimento desta pesquisa de mestrado são as controladoras relacionadas a arquitetura PCI Express (compostas pelo *root complex* e os *bridges* e *devices* PCI-e), e a controladora de áudio (que nesse caso segue a especificação *High Definition Audio*).

A complexidade dos *chipsets* é bastante elevada, fato esse maior causador da necessidade do BIOS e seus módulos necessários para a correta inicialização do equipamento antes que seu controle seja cedido ao *bootloader* ou sistema operacional. Vale ressaltar que normalmente os *chipsets* possuem em sua documentação informações valiosas a serem consideradas pelos desenvolvedores de BIOS, como apresentado em (INTEL®, 2005). Nas próximas seções serão apresentadas as tecnologias que trabalham em conjunto com a arquitetura PCIe para o processamento dos sinais sonoros necessários para gravação e reprodução de áudio digital.

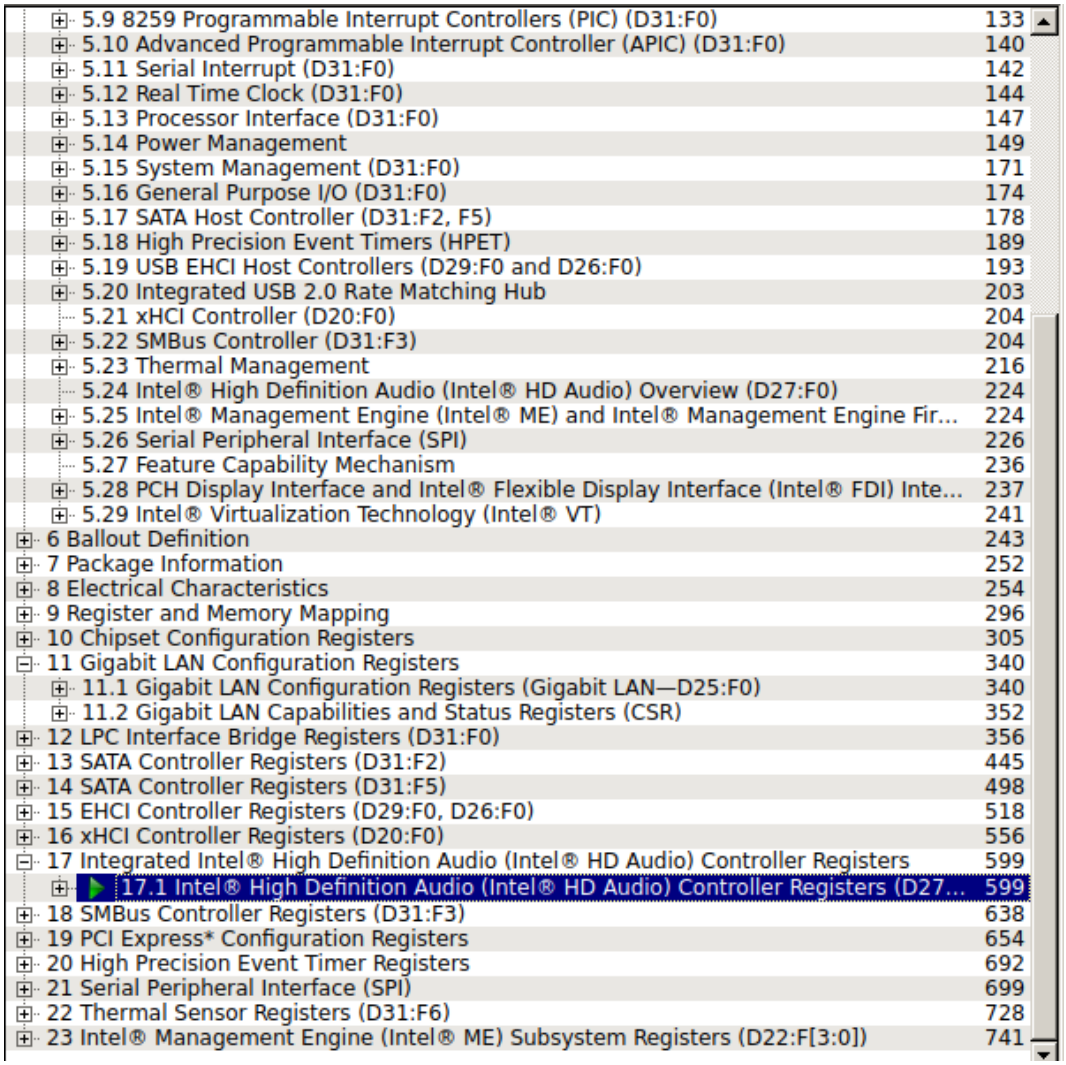

Figura 6: Controladoras integradas ao *chipset*. Fonte: (INTEL®, 2005)

1.4 Arquitetura PCI-e

No início da década de 90, momento no qual a evolução dos computadores pessoais obteve maior impulso, diversas limitações existentes nas arquiteturas de barramentos locais passaram a se tornar barreiras para a evolução dos computadores. Devido a tais limitações diversos modelos de barramentos começaram a ser desenvolvidos pelas empresas envolvidas em tal mercado, como é o caso dos barramentos ISA, EISA, VESA e *Micro Channel*. A fim de eliminar a possibilidade de existência de diversos tipos de barramentos para processadores específicos, em 22 de Julho de 1992, a Intel publicou a versão 1.0 da especificação denominada PCI, sigla para *Peripheral Component Interconnect*, com a finalidade de possibilitar que uma única arquitetura de conexão local passasse a ser utilizadas por todos fabricantes, facilitando assim a evolução dos computadores (ANDERSON; SHANLEY, 1999).

Com o passar do tempo, algumas melhorias foram propostas à arquitetura PCI, se

apresentando desta forma como uma evolução deste conceito, a qual foi denominada PCIe (sigla para *Peripheral Component Interconnect Express*). Desta forma a velocidade de banda de cada linha do barramento passou de 1,6 MBytes/s (12,8 Mbits/s) na especificação PCI 1.0, para 2,5 Gbits/s (ambos se referindo a cada linha de transmissão existentes em um dado barramento) na especificação PCIe 1.0 disponibilizada no ano de 2002 (BUDRUK; ANDERSON; SHANLEY, 2004).

A arquitetura PCIe se utiliza da uma forma de localização dos dispositivos denominada localização geográfica, que se refere ao posicionamento de um dado dispositivo na plataforma em questão. Esta localização se dá através da enumeração dos dispositivos e suas funções durante o processo denominado inicialização da plataforma (*PCIe Enumeration Process*), durante o carregamento dos *drivers* da mesma. Vale ressaltar que o processo de enumeração não se refere a controladoras existentes no próprio *chipset*, visto que tais controladoras possuem uma localização padrão normalmente apresentada em alguma documentação técnica referente a plataforma, sendo tais controladoras localizadas no barramento 0.

Como exemplo podem ser mencionadas controladoras compatíveis com a especificação *Intel High Definition Audio*, as quais obrigatoriamente se localizam no barramento 0, no dispositivo 21, em sua funcao 0 (INTEL®, 2005, p. 13) (localização esta apresentada como 00:1B:00). Desta forma, o processo de enumeração das controladoras existentes no chipset passa a ser uma tarefa mais simples, apesar de diversos outros passos ainda serem necessários durante a inicialização de um *chipset* por completo, como alocação de recursos para os dispositivos, definição de endereços para comunicação, entre outras tarefas específicas para cada tipo de dispositivo.

A arquitetura PCIe define também a forma na qual registradores relacionados às configurações de um dado dispositivo devem ser mapeadas em memória. A esse mapeamento se dá o nome de *configuration space*, e tem papel fundamental para que os *drivers* específicos de um dado dispositivo sejam capazes de detectar e configurar corretamente o mesmo. Dentre as configurações acessíveis nos *configuration spaces* existem algumas de caráter informativo, utilizadas pelos sistemas operacionais para o carregamento dos *drivers* corretos, informações de status do dispositivo, e informações relacionadas aos recursos alocados pelo mesmo durante sua inicialização.

As Figuras 7 e 8 apresentam a definição das informações dos dois tipos básicos de dispositivos definidos pela arquitetura PCI e que foram mantidos na arquitetura PCIe. Esses dois tipos de dispositivo são os do tipo dispositivo (*devices*) sendo os equipamentos com real finalidade de aquisição e processamento de informações, e os do tipo ponte (*bridges*) que atuam como roteadores em uma rede. Esta comparação com infraestruturas de rede auxilia no entendimento da arquitetura PCI na qual os dispositivos seriam os computadores existentes nesta rede hipotética. A localização geográfica se assemelha a

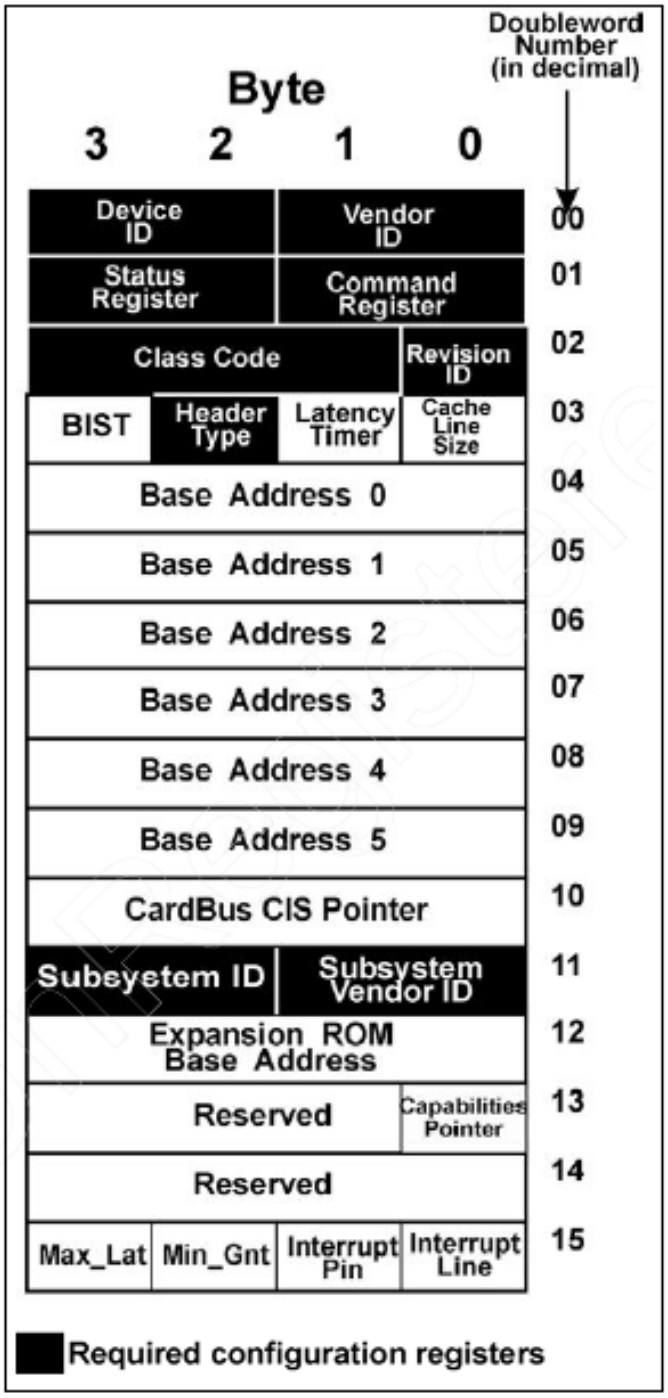

Figura 7: *Configuration Space* de um *device*. Fonte: (BUDRUK; ANDERSON; SHANLEY, 2004, p. 751)

endereços IP, permitindo que as *bridges* saibam o caminho para o qual os pacotes devem ser enviados e de onde vieram.

O conceito de função presente na espeficiação PCIe se refere a uma região de memória com finalidade específica, sendo o número máximo de funções permitidas igual a 8 (representados pelas funções de 0 a 7). Dispositivos com mais de uma função - *multifunction devices* - devem possuir a função número 0 preenchida, podendo as outras funções não

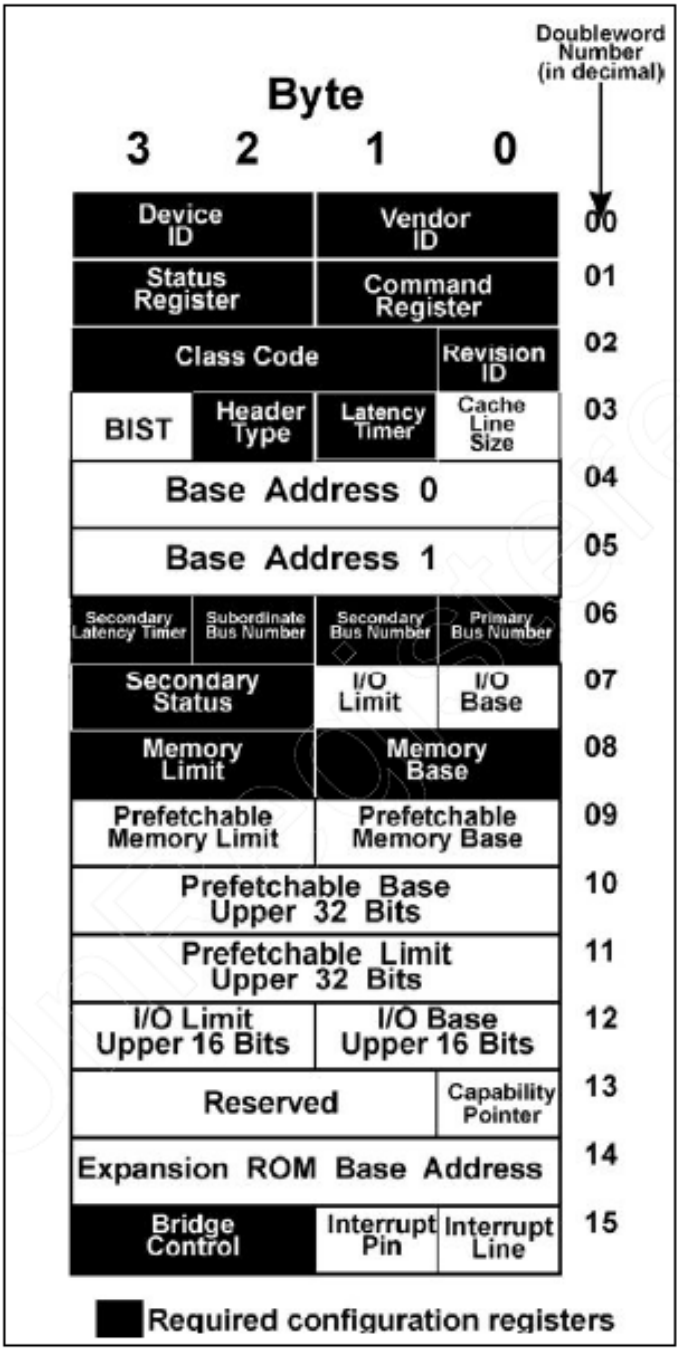

Figura 8: *Configuration Space* de um *bridge*. Fonte: (BUDRUK; ANDERSON; SHANLEY, 2004, p. 752)

estarem ordenadas. No caso de dispositivos *single function*, a função 0 também é a função a estar preenchida.

Como exemplo da necessidade de diversas funções, podem ser apresentadas placas de vídeo com conexão HDMI, nas quais são preenchidas duas funções, sendo uma para informações de áudio e outra para informações de vídeo. A Figura 9 exemplifica um dispositivo *multi function*, localizado no Barramento 3 Device 0 (*Bus 3 Device 0*).

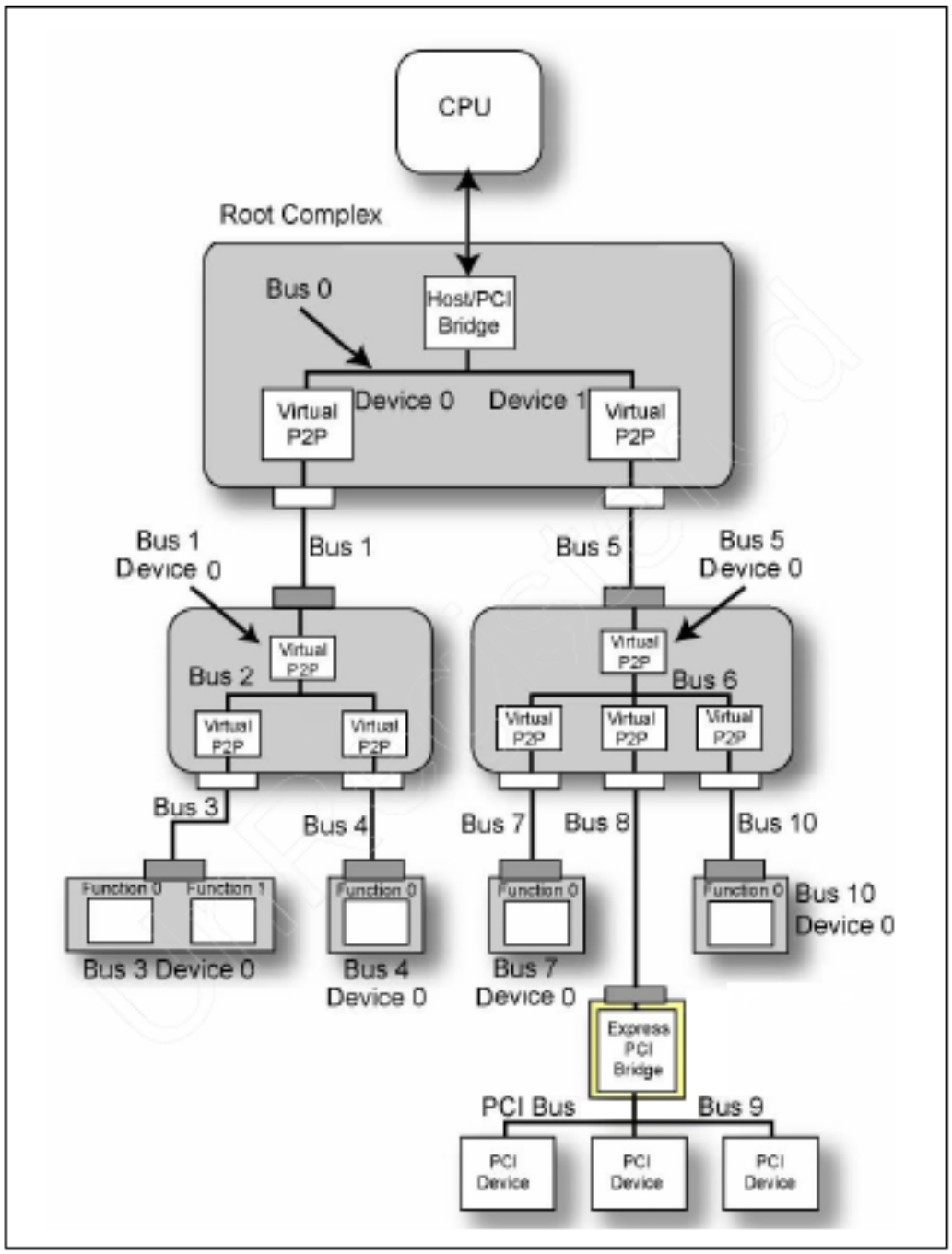

Figura 9: Dispositivos *Single Functions* e *Multi Function*. Fonte: (BUDRUK; ANDERSON; SHANLEY, 2004, p. 749)

Para que um dispositivo possa se comunicar corretamente com o processador, a arquitetura PCIe define um campo existente nos *configuration spaces* denominado BAR (sigla para *Base Address Register*), que possui as informações apresentadas na Figura 10. Neste registrador a informação mais importante para o projeto proposto é o campo denominado *Base Address*, visto que detalhes sobre o processo de enumeração da arquitetura PCIe fugiriam do propósito principal do trabalho apresentado. Neste campo, durante o processo de enumeração é inserido um endereço que representa uma posição

na memória RAM da plataforma, na qual podem ser acessadas informações detalhadas sobre o dispositivo. Tais informações devem seguir um padrão, que varia de acordo com a finalidade do dispositivo. No caso de controladoras compatíveis com a especificação HDA, o mapeamento das informações existentes na região definida pelo BAR é encontrado no documento *Intel High Denifition Audio Programmers Reference Manual* (INTEL®, 2005)

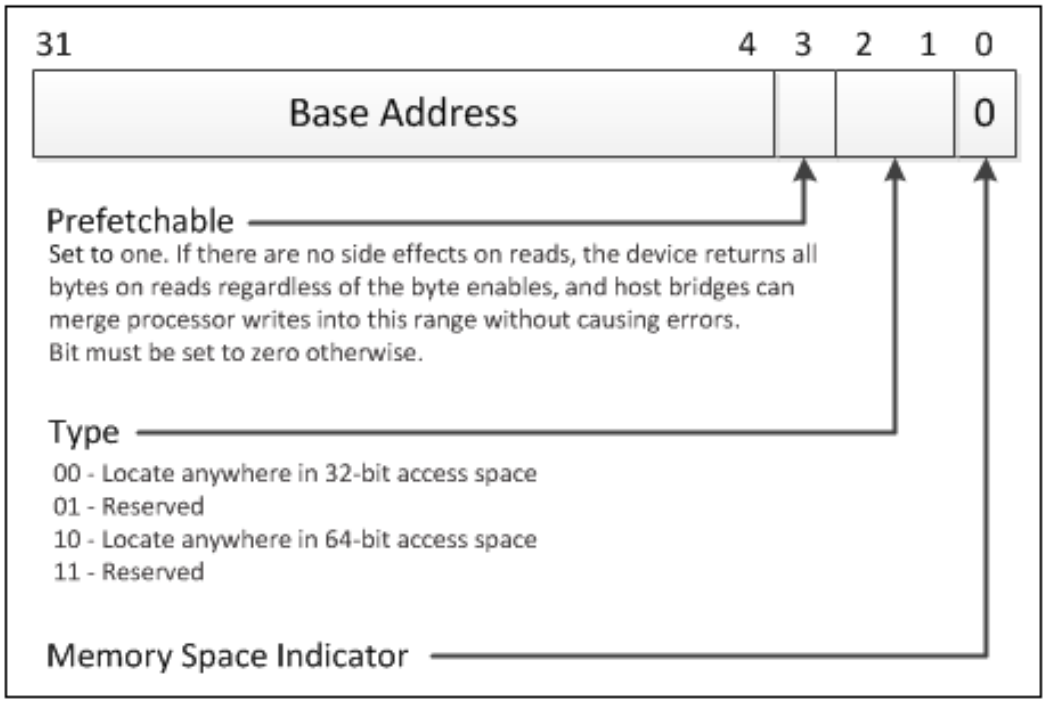

Figura 10: PCIe *Base Address Register*. Fonte: (BUDRUK; ANDERSON; SHANLEY, 2004, p. 796)

A região de memória demarcada pelo BAR, também conhecida como *Memory Mapped Configuration Registers* engloba todos os registradores necessários para configuração e operação da controladora de áudio existente no *chipset*, permitindo assim que seja realizada a configuração da mesma para a correta comunicação com o codec existente na plataforma. Essa configuração é importante para garantir o processamento dos sinais de áudio de forma coordenada, evitando a ocorrência de ruídos em caso de disparidade entre as configurações da controladora e do codec (fato esse de extrema importância para a reprodução do áudio desejado).

Outra informação bastante importante disponível na arquitetura PCIe é utilizada pelos dispositivos para apresentação de erros decorrentes de pacotes malformados, devido a interrupções disparadas por controladoras incapazes de processar uma dada informação, ou devido a falhas em operações de memória (por exemplo leitura ou acesso de endereçamentos não reservados para operações do barramento PCIe). Esta forma de apresentação dos erros detectados pelo barramento e suas controladoras foi originalmente proposta pela arquitetura PCI, e se manteve na arquitetura PCIe, visto que tal mecanismo apresentou

bons resultados. Para tal no próprio *configuration space* dos dispositivos PCIe, o registrador *PCI Status Register*, possui o mapeamento apresentado na Figura 11.

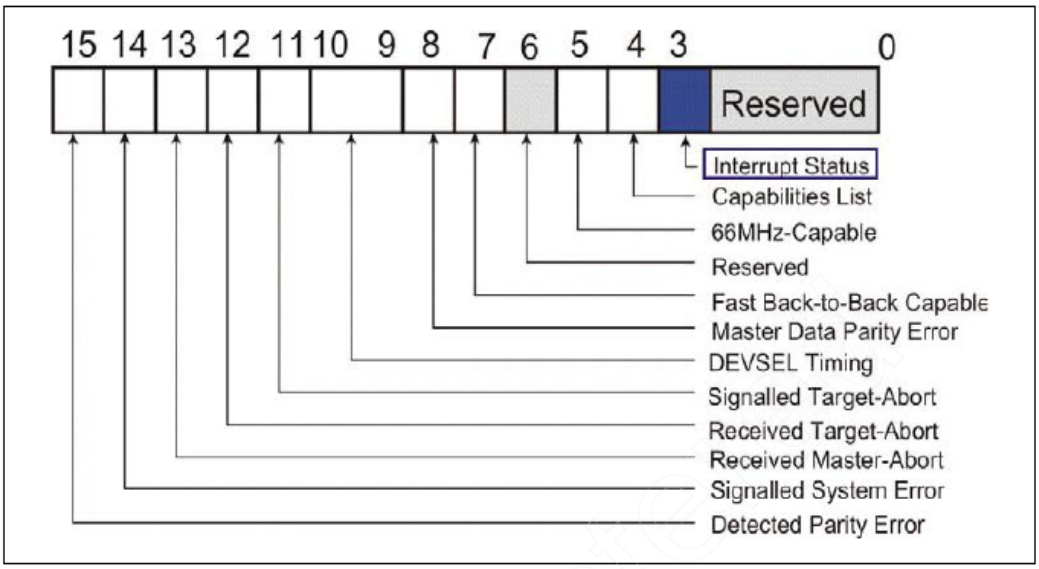

Figura 11: *Status Register*. Fonte: (BUDRUK; ANDERSON; SHANLEY, 2004, p. 347)

Diversos tipos de falhas detectadas no barramento ao processar, enviar ou receber um dado pacote podem ser apresentadas. Para essa dissertação, o bit mais importante neste registrador é o bit de índice 13, denominado *Received Master-Abort*, pois como apresentado a seguir, este bit sempre subia para nível lógico 1 quando um dado stream de áudio era processado.

1.5 Intel High Definition Audio

Reprodução de áudio é uma funcionalidade de entrada e saída essencial para dispositivos pessoais modernos. Para permitir a geração de áudio de alta qualidade, existe uma enorme quantidade de fabricantes de placas de som, dispositivos estes que ficam conectados internamente ou externamente aos computadores, permitindo assim que informações sejam convertidas em impulsos sonoros analógicos que posteriormente são emitidos pelos alto-falantes conectados a placa de som (DAVID JANUS SCOTT, 2006).

Dentre as diversas arquiteturas possíveis para as placas de som, destaca-se uma conhecida como *Intel High Definition Architecture (HDA)*. Esta especificação define um padrão de comunicação a ser seguido pelos fabricantes que possuírem interesse em ser compatíveis com as plataformas Intel existentes até o momento. Dentre outras atribuições, esta especificação define uma interface de comunicação a ser adotada pelos fabricantes de placas de som bem como as formas de interação entre a plataforma e o equipamento de áudio. Objetivo este que fica bem claro na própria especificação, como apresentado no paragrafo a seguir:

Os principais objetivos da especificação *High Definition Audio* são descrever um infra-estrutura para suportar áudio de alta qualidade em um ambiente de PC. A especificação inclui o definição do conjunto de registradores do controlador, a descrição física das interconexões, o modelo de programação dos codecs, e componentes de arquitetura dos codecs(INTEL®, 2010). Desta forma, os dados referentes aos impulsos sonoros são processados através de um trabalho conjunto entre a controladora, responsável pelo gerenciamento dos *streams* de dados a serem processados, e o codec responsável pelo tratamento e conversão do *stream* em impulsos analógicos.

Vale ressaltar que a comunicação entre estes dois componentes é realizada através de um canal dedicado denominado *Intel HD Audio Link*. As informações são trafegadas no *Audio Link* em uma frequência pré definida, que no caso da especificação HDA é de 24 MHz. Os dados trafegados são subdivididos em quadros (denominados *frames* na especificação HDA, exemplificados na Figura 12) com 20833 microssegundos de duração (INTEL®, 2010, p. 21).

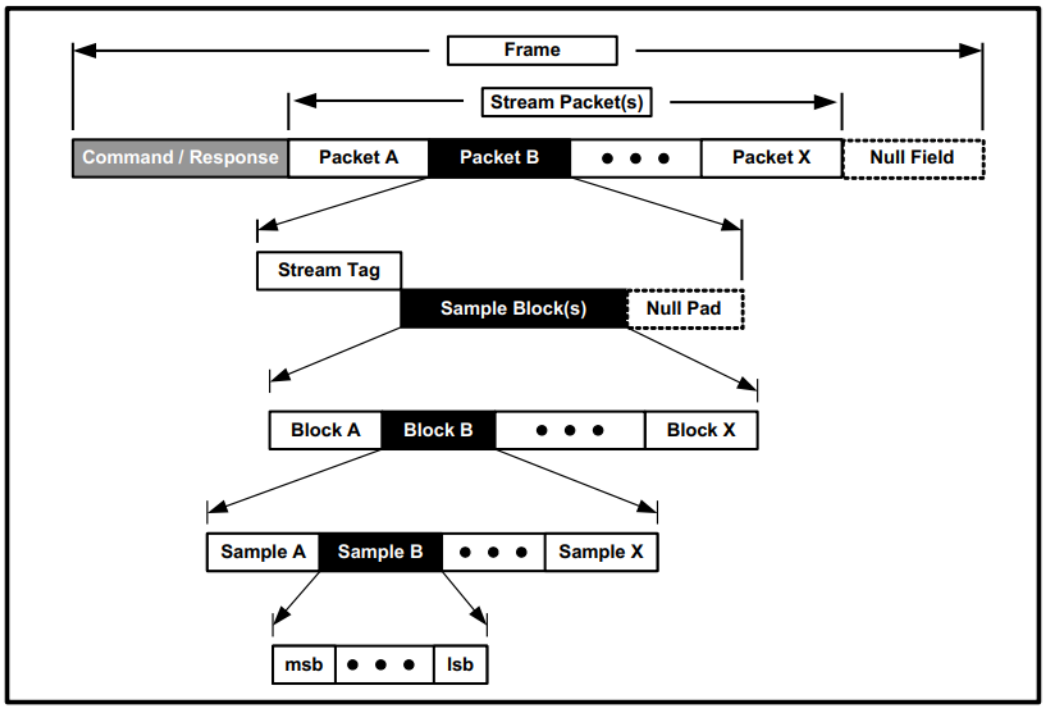

Figura 12: Estrutura de um Frame segundo a especificação HDA. Fonte: (INTEL®, 2010, p. 84)

Outra característica importante das controladoras compatíveis com a especificação HDA, é a existência de controladoras de DMA, responsáveis pela movimentação dos dados a serem processados da memória RAM do equipamento até o codec através do *audio link*, sem que seja necessária a utilização do processador principal da plataforma para tal movimentação. Para que tal ação seja bem sucedida é necessária a correta configuração da controladora, através da gravação de valores em registradores de uso específico, os quais

são detalhados nas especificações do chipset (INTEL®, 2005).

Posteriormente à configuração da controladora HDA, é necessária a configuração do codec e seus *widgets*, que podem ser definidos como componentes com função específica e interconectados aos codecs, com a finalidade de realizar todos os tratamentos necessários aos streams de áudio, permitindo que os mesmos sejam convertidos em sinais analógicos (no caso de streams de saida), ou sinais digitais (no caso de streams de entrada). É importante ressaltar o fato de que *codecs* de fabricantes diferentes podem apresentar detalhes específicos de inicialização, sendo esta liberdade fonte de grande parte da complexidade em se criar um driver genérico que funcione corretamente em qualquer equipamento.

Os *widgets* definidos pela especificação HDA são apresentados a seguir:

1. *Audio Output Converter (AOC)*
2. *Audio Input Converter (AIC)*
3. *Pin (ou Pin Complex)*
4. *Mixer*
5. *Selector*
6. *Power*
7. *Volume Knob*
8. *Beep Generator*

Tais componentes trabalham em conjunto e são apresentados na documentação relacionada ao *codec*, permitindo assim que sejam visualizadas suas interconexões. A Figura 13, extraída da especificação técnica do *codec*, apresenta claramente na forma de diagrama de blocos os *widgets* existentes, bem como sua conexões.

O envio e recebimento de informações dos e para os *widgets* se faz através de comandos denominados verbos (nomeados *verbs* na especificação HDA). Tais *widgets* são utilizados para as mais diversas tarefas, desde configurar o volume de um dado *widget*, até a possibilidade de criar grupos de *streams* a serem processados através de um identificador (streamId). Permitindo assim a realização de processamentos de *streams* de áudio em paralelo (ex.: execução de duas músicas simultaneamente).

As Figuras 14 e 15 apresentam verbos definidos pela especificação HDA, bem como os widgets aos quais os mesmos podem ser enviados. Nestas figuras, itens marcados como R (*Required*) são obrigatórios segundo a especificação, itens marcados como c (*Conditional* são relacionados a existência de funcionalidades opcionais, itens marcados como X são

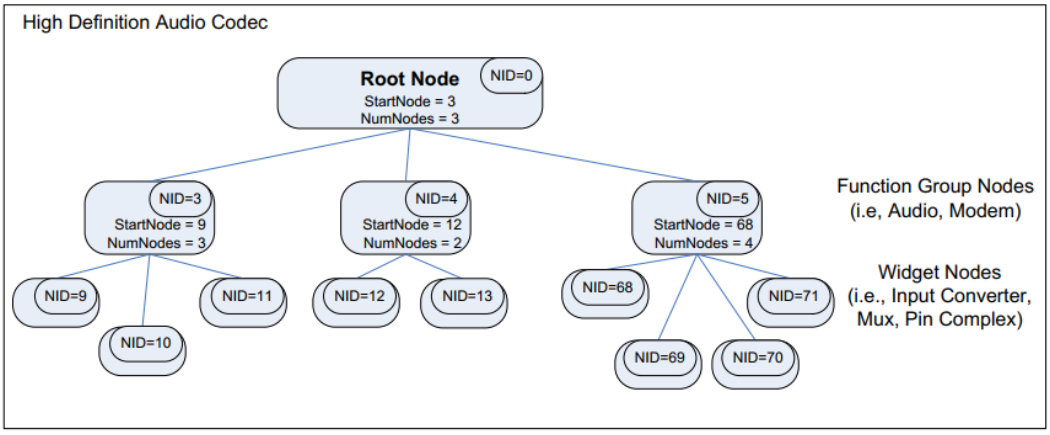

Figura 13: Arquitetura de um Codec segunda a especificação HDA. Fonte: (INTEL®, 2010, p. 129)

relacionados a funcionalidades que exigem suporte de múltiplos canais de entrada, e itens marcados com * são específicos de cadas fabricante.

Vale ressaltar que todos os verbos definidos pela especificação HDA possuem um identificador e um formato de dados a ser utilizado para o envio do comando a ser executado (dados estes denominados *payload*). Como exemplo, o verbo 0x705, utilizado no gerenciamento do modo de energia de alguns tipos de *widgets*, recebe como *payload* o *power state* desejado, como apresentado na Figura 16

A especificação HDA define duas formas as quais os verbos podem ser enviados ao codec. A primeira é através da utilização dos denominados *immediate registers* (INTEL®, 2010, p. 50), e a segunda é através dos *buffers* denominados CORB e RIRB (siglas para *command output ring buffer* e *Response Input Ring Buffer*) (INTEL®, 2010, p. 62).

Os registradores denominados *Immediate Registers* foram criados justamente para facilitar o recebimento e envio de informações aos codecs. Tal tarefa se faz através da realização dos passos descritos a seguir (INTEL®, 2010, p. 50):

1. Escrita de um dado verbo + *payload* no registrador *immediate command output register* (*offset* 60h dentro dos *memory mapped registers*);

2. Escrita do valor 1 no bit de índice 0 (denominado *immediate command busy*), no registrador *immediate command status* (*offset* 68h dentro dos *memory mapped registers*);

3. Verificação seguida dos bits de indice 0 (*immediate command busy*) e indice 1 (*immediate result valid*), os quais devem apresentar os valores 0 e 1 respectivamente, indicando que o comando foi processado, e que um resultado válido está disponível;

Widget / Required Verb Support	Get Code	Set Code	Root Node	Audio Function Group	Modem Function Group	Vendor Defined Function Group	Audio Output Converter	Audio Input Converter	Pin Complex Widget (non Digital Display)	Pin Complex Widget (Digital Display)	Mixer (SumAmp)	Selector (Mux)	Power Widget	Volume Knob	Beep Generator	Vendor Defined Wdiget
Get Parameter	F00		R	R	R	R	R	R	R	R	R	R	R	R	R	R
Connection Select	F01	701						c	c	c		c				*
Get Connection List Entry	F02							R	R	R	R	R	R	R		*
Processing State	F03	##					c	c	c	c		c				*
Coefficient Index	D- -	5- -					c	c	c	c		c				*
Processing Coefficient	C- -	4- -					c	c	c	c		c				*
Amplifier Gain/Mute	B- -	3- -					c	c	c	c	c	c			c	*
Stream Format	A- -	2- -					R	R								*
Digital Converter 1	F0D	70D					c	c								*
Digital Converter 2	F0D	70E					c	c								*
Digital Converter 3	F0D	73E					c	c								*
Digital Converter 4	F0D	73F					c	c								*
Power State	F05	705		R	R	c	c	c	c	c	c	c	R	c	c	c
Channel/Stream ID	F06	706					R	R								*
SDI Select	F04	704					X	X								*
Pin Widget Control	F07	707							R	R						*
Unsolicited Enable	F08	708		c	c	c	c	c	c	c	c	c	c	c		*
Pin Sense	F09	709							c	c						*
EAPD/BTL Enable	F0C	70C							c							*
All GPI Controls	F10 thru F1A	710 thru 71A		c	c											
Beep Generation Control	F0A	70A													R	
Volume Knob Control	F0F	70F												R		
Implementation ID,	F20	720		R	R	R										

Figura 14: Verbos Segundo a especificação HDA. Fonte: (INTEL®, 2010, p. 216)

4. Leitura do registrador “Immediate Response Read” (offset 64h dentro dos Memory Mapped Registers).

Desta forma, um verbo seria enviado ao *codec* e processado no *frame* seguinte, apresentando a resposta para o comando no registrador de *offset* 64h. Tal saída se apresenta bastante simples, porém não é de implementação obrigatória, razão essa para tal abordagem não ser utilizada. O mesmo pode ser detectado nos *drivers* de Linux implementados dentro da arquitetura ALSA (LINUX, 2018)), os quais também não fazem uso dos registradores

Widget / Required Verb Support	Get Code	Set Code	Root Node	Audio Function Group	Modem Function Group	Vendor Defined Function Group	Audio Output Converter	Audio Input Converter	Pin Complex Widget (non Digital Display)	Pin Complex Widget (Digital Display)	Mixer (SumAmp)	Selector (Mux)	Power Widget	Volume Knob	Beep Generator	Vendor Defined Wdiget
Byte 0																
Implementation ID, Byte 1	F20	721		R	R	R										
Implementation ID, Byte 2	F20	722		R	R	R										
Implementation ID, Byte 3	F20	723		R	R	R										
Config Default, Byte 0	F1C	71C							R	R						
Config Default, Byte 1	F1C	71D							R	R						
Config Default, Byte 2	F1C	71E							R	R						
Config Default, Byte 3	F1C	71F							R	R						
Stripe Control	F24	724					c									
Converter Channel Count	F2D	72D					c									
DIP-Size	F2E									R						
ELD Data	F2F									R						
DIP-Index	F30	730								R						
DIP-Data	F31	731								R						
DIP-XmitCtrl	F32	732								R						
Content Protection Control	F33	733								c						
ASP Channel Mapping	F34	734								R						
RESET		7FF		R	R	R										

Figura 15: Verbos Segundo a especificação HDA cont. Fonte: (INTEL®, 2010, p. 217)

de acesso imediato.

A segunda forma de comunicação com o codec se dá através dos *buffers* denominados CORB e RIRB, sendo o CORB o centralizador dos comandos a serem enviados e o RIRB o centralizador das respostas recebidas. Ambos são compostos por um *buffer* circular, conforme apresentado na Figura 17.

Para o envio de comandos através dos registradores CORB e RIRB algumas

	Verb ID	Payload (8 Bits)	Response (32 Bits)
Get	F05h	0	Bits 31:11 are 0 PS-SettingsReset is in bit 10 PS-ClkStopOk is in bit 9 PS-Error is in bit 8 PS-Act is in bits 7:4 PS-Set is in bits 3:0
Set	705h	PS-Set in bits 0:3 bits 4:7 are Reserved and 0	0

Figura 16: Hda *Power State Verb*. Fonte: (INTEL®, 2010, p. 151)

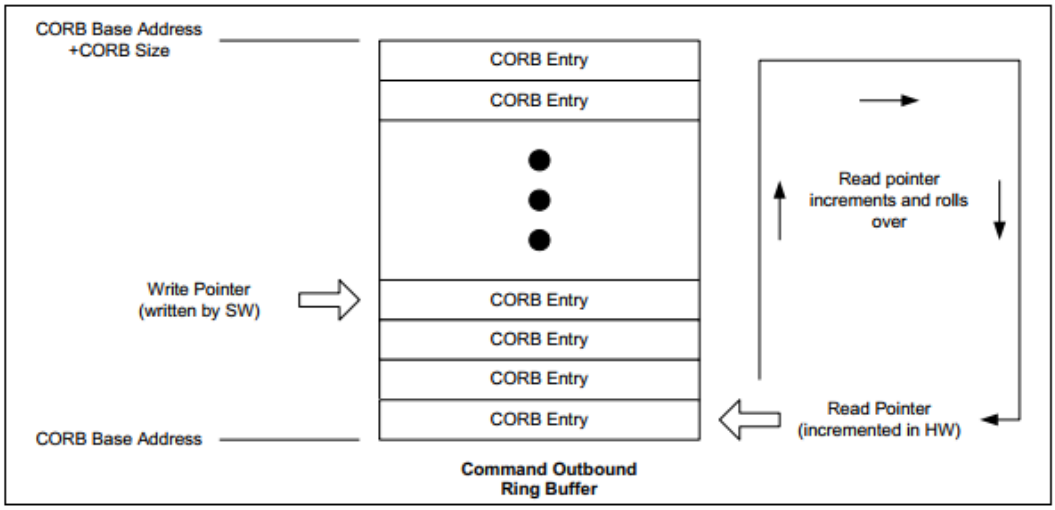

Figura 17: *Ring Buffer CORB*. Fonte: (INTEL®, 2010, p. 63)

configurações adicionais devem ser realizadas durante a inicialização da controladora de áudio. Esta configuração se dá através da alocação de dois buffers, os quais os endereços devem ser escritos nos *memory mapped registers* específicos para cada fim. A Tabela 2 a seguir descreve os registradores necessários para a correta configuração do CORB. No caso dos registradores referentes ao RIRB, o funcionamento é basicamente o mesmo descrito anteriormente, porém considerando *offsets* diferentes, como apresentado na Tabela 3.

Offset	**Nome**
HBAR + 40H	CORBLBASE
HBAR + 44H	CORBUBASE
HBAR + 48H	CORBWP
HBAR + 4CH	CORBCTL

Tabela 2: Registradores CORB

Através da utilização dos registradores CORB e RIRB, o mesmo envio de comandos descrito anteriormente pelos *immediate registers* pode ser alcançado, porém com a diferença que sua implementação é obrigatória nos dispositivos de áudio compatíveis com a especificação HDA. Por essa razão tal abordagem foi selecionada para a implementação

Offset	**Nome**
HBAR + 50H	RIRBLBASE
HBAR + 54H	RIRBUBASE
HBAR + 58H	RIRBWP
HBAR + 5CH	RIRBCTL

Tabela 3: Registradores RIRB

deste projeto, tendo sido a utilização dos *Immediate Registers* realizada somente para fins de entendimento da arquitetura HDA, como apresentado na Seção 1.5.

Sendo assim, estando controladoras DMA configuradas corretamente, bem como o codec e seus *widgets* (e em conformidade com as configurações do barramento PCIe apresentado anteriormente na Seção 1.4), os pacotes com informações a serem processadas podem ser movimentados até o codec através do *audio link* na forma de *frames* de tamanho e formato específico, permitindo que seu processamento seja realizado sem que ocorram conflitos entre os dados movidos pelas controladoras DMA e o codec. Na Seção 2.2 serão apresentadas em detalhes as configurações necessárias para o correto funcionamento do projeto proposto.

2 O Projeto

Nesse capítulo serão apresentados os detalhes técnicos da solução elaborada para satisfazer as motivações apresentadas.

2.1 Arquitetura de Software

Desde sua concepção, o intuito desse projeto de mestrado foi o de criar códigos de forma organizada e de fácil entendimento, facilitando assim seu uso por parte de outras pessoas que tenham interesse tanto no desenvolvimento de aplicações em ambiente pré-OS, quanto para os interessados no funcionamento de dispositivos de áudio segundo o especificado pela Intel.

Este projeto foi arquitetado com o objetivo final de disponibilizar uma biblioteca para ambiente pré-OS que permitisse a realização de ações nos hardwares envolvidos no processamento de áudio, sendo estes componentes a arquitetura PCI-e (Seção 1.4), o *chipset* (Seção 1.3.2) e o codec de áudio (Seção 1.5). A Figura 18 apresenta a arquitetura do projeto, bem como sua integração ao ambiente UEFI existente nas plataformas atuais.

Conforme apresentado na Figura 18, o trabalho realizado teve como objetivo a elaboração de dois componentes complementares. Sendo o primeiro uma biblioteca denominada HdaLib, e o segundo um tocador de áudio minimalista que faz uso da biblioteca mencionada anteriormente.

A biblioteca HdaLib se posiciona dentro da estrutura existente nas plataformas atuais de forma a consumir os protocolos e serviços disponibilizados pela especificação UEFI, mais precisamente, a biblioteca HdaLib faz uso dos servicos denominados *Global Boot Services* (UEFI Forum, 2014, p. 113), e o protocolo EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL (UEFI Forum, 2014, p. 633). Já o tocador *Bene Music Player*, faz uso de todas funcionalidades disponibilizadas pela biblioteca HdaLib, e é explicado em detalhes na Seção 2.3. Detalhes mais aprofundados do funcionamento da biblioteca HdaLib são apresentados na Seção 2.2.

2.1.1 Ambiente Utilizado

Inicialmente, a proposta deste trabalho previa a utilização de ambientes de simulação. O trabalho se daria através do desenvolvimento da prova de conceito proposta com a utilização de um ambiente virtualizado, que no caso de desenvolvimento de aplicações e *drivers* para ambiente pré-OS ocorre com a utilização de um subprojeto existente no

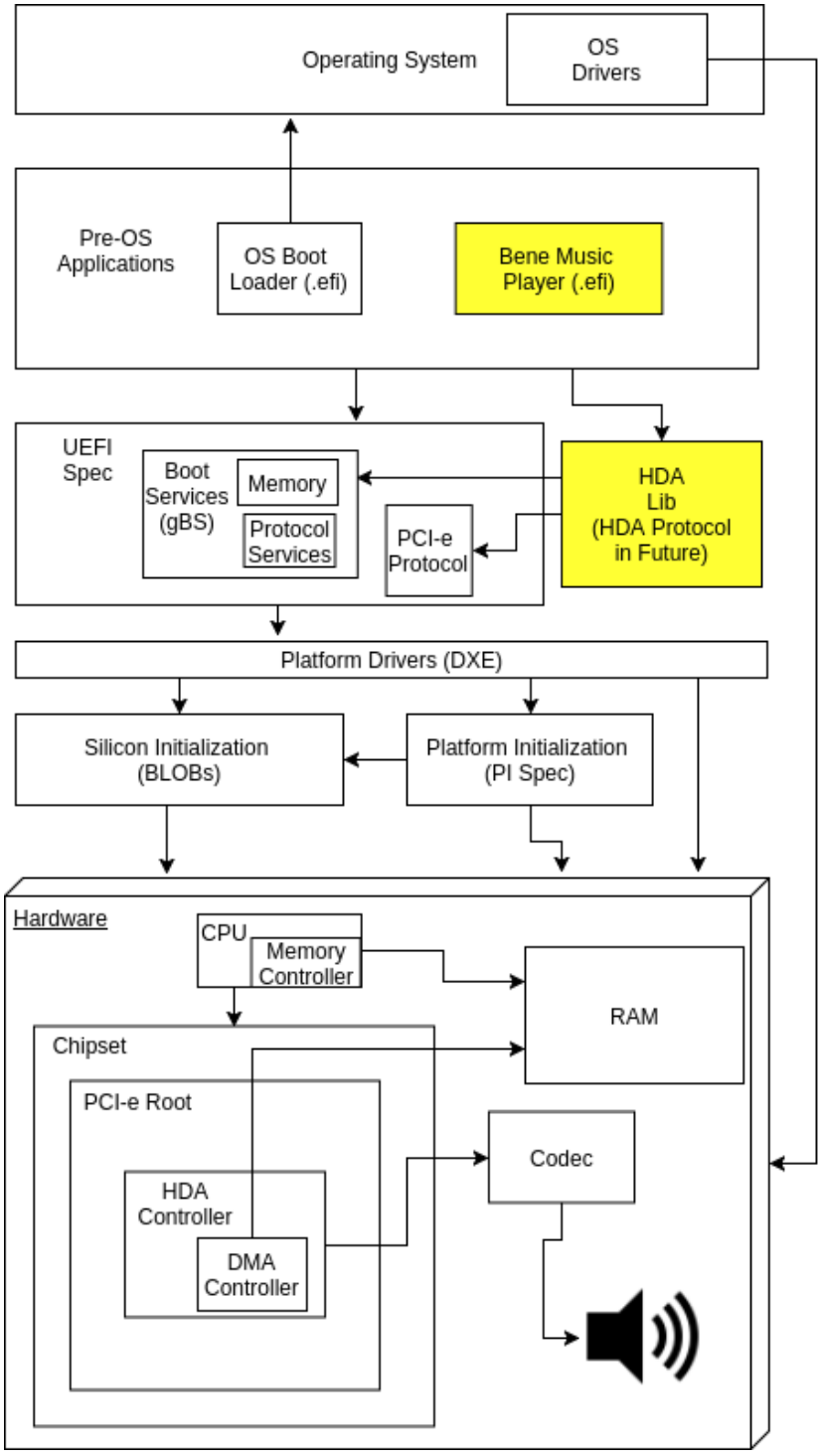

Figura 18: Arquitetura do Projeto

framework edk2, denominado OVMF (Open Virtual Machine Firmware) (TIANOCORE, 2018b).

O projeto OVMF consiste em um BIOS open-source compatível com a especificação UEFI, o qual pode ser executado nos emuladores de CPU QEMU (QEMU, 2018) e KVM (LINUX, 2018). Considerando essa possibilidade, os testes iniciais se deram na forma de detectar se a placa de som existente no equipamento hospedeiro do ambiente virtualizado

poderia ser acessada através do OVMF. O resultado desse experimento demonstrou que o OVMF não era capaz de reconhecer a controladora de áudio como apresentado nas especificações de chipsets Intel. Controladoras de áudio compatíveis com tal especificação devem ser enumeradas na localização geográfica 00:27:00, como apresentado pelo trecho a seguir extraido da especificação HDA (INTEL®, 2005, p. 13):

> *The Intel HD Audio controller resides in PCI Device 27, Function 0 on bus 0. This funcition contains a set of DMA engined that are used to move samples of digitally encoded data between system memory and external codecs.*

O dispositivo não foi detectado através do comando "pci", comando este disponibilizado pelo UEFI Shell (UEFI Forum, 2016) existente no ambiente shell disponibilizado pelo OVMF e acessível através do QEMU. Tal comportamento se manteve mesmo sendo executado o QEMU com o parâmetro "-hwsound hda", que segundo sua documentação seria o comando responsável por permitir que o equipamento hospedeiro (ambiente virtualizado nesse caso), tivesse acesso ao hardware de áudio.

Desta forma, foi escolhida a abordagem na qual o desenvolvimento do projeto descrito seria realizado em um equipamento real, evitando assim problemas relacionados a incompatibilidade com ambientes virtualizados, e possíveis implementações parciais de funcionalidades. Detalhes relacionados ao equipamento utilizado no decorrer deste trabalho se apresentam na Tabela 4.

Componente	**Descrição**
Fabricante	ASUS
Modelo	PU401L
Ano de Fabricação	2014
Processador	Intel Core i7 4500U
Fabricante do BIOS	American Megatrends International (AMI)
Versão do BIOS	208 (2.15.1236)
Fabricante do Codec	Conexant
Modelo do Codec	CX20752

Tabela 4: Equipamento utilizado

Para o correto entendimento da estrutura interna do *codec* existente neste equipamento, se fez necessária a consulta de informações no *datasheet* do mesmo. Esta documentação apresenta o diagrama de blocos do codec, permitindo assim o entendimento dos *widgets* disponíveis, sua localização e interconexões, conforme apresentado na Figura 19.

Desta forma, apresenta-se um equipamento no qual encontra-se um codec compatível com a especificação HDA, sendo as informações relacionadas a este codec de conhecimento público, não sendo necessária a solicitação de informações confidenciais ao fabricante.

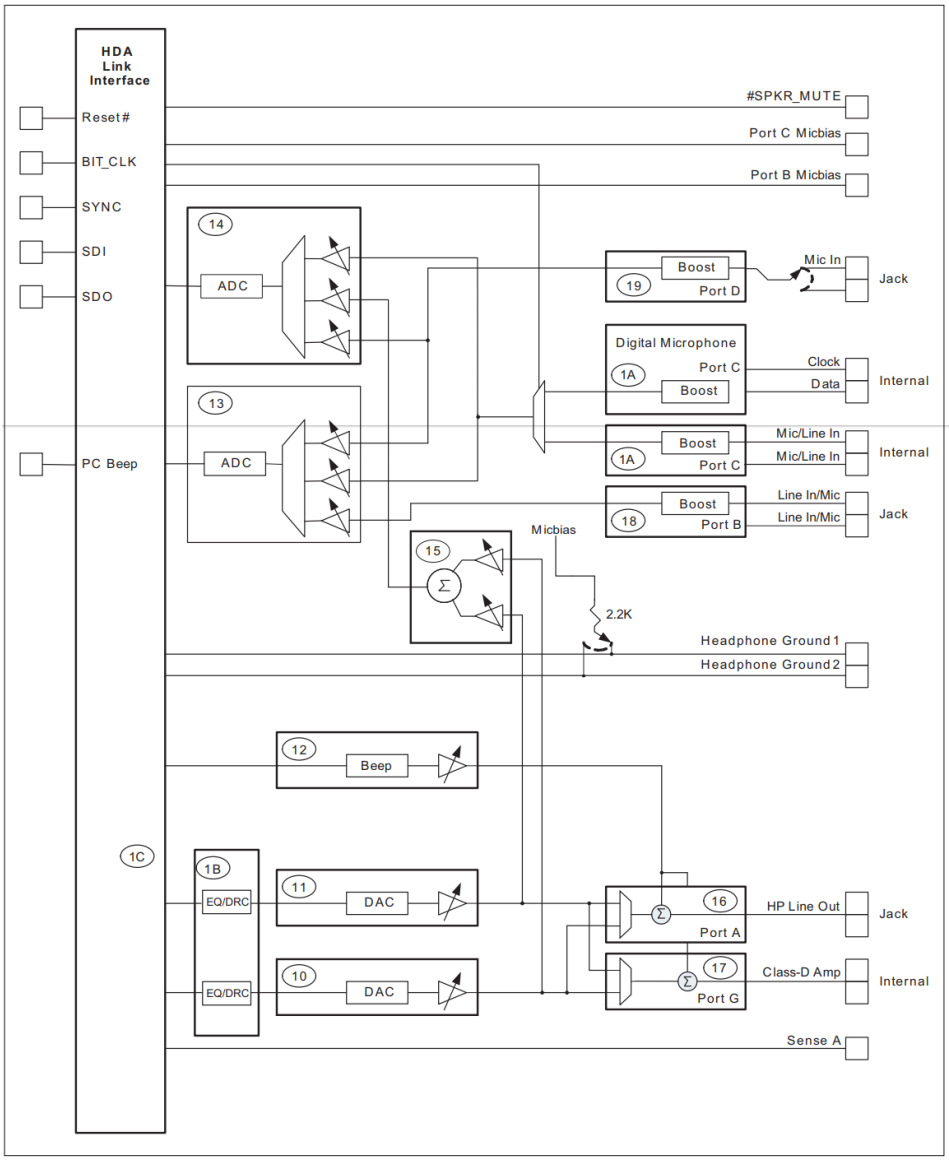

Figura 19: Diagrama de blocos codec Conexant CX20752 rev 1. Fonte: (CONEXANT®, 2015, p. 5)

2.2 HdaLib

Inicialmente, esse projeto visava fazer uso de conceitos existentes na arquitetura ALSA (*Advanced Linux Sound Architechture*), porém sua complexidade se mostrou uma desvantagem, dado o tempo necessário para seu entendimento. Adicionalmente, o fato de os códigos referentes a arquitetura ALSA seguirem a licença GLP e LGPL impediriam sua utilização por parte de projetos que não sigam tais licenças, como é o caso do projeto apresentado neste documento, o qual segue a licença BSD.

Sendo assim a arquitetura ALSA foi utilizada somente como fonte de consulta em pontos específicos, como por exemplo no entendimento das etapas necessárias para configuração do barramento PCIe. Desta forma, a arquitetura proposta não segue a

organização da arquitetura ALSA, sendo a mesma apoiada unicamente nas especificações HDA e UEFI, almejando maior simplicidade focando na facilidade de seu entendimento. Conforme será apresentado no procedimento utilizado para a comunicação básica com o codec existentes na plataforma.

A arquitetura HDA define que codecs compatíveis com a mesma disponibilizem um widget denominado *Beep Generator*, que pode ser utilizado para geração de tons. Como definido na especificação, o mesmo se dá através do envio do *verb Beep Generation Control* (INTEL®, 2010, p. 174), existente no codec sob identificação 0x13 (Figura 19).

Porém, para que o processamento do comando seja executado se faz necessária a correta configuração do codec com relação a seu estado de alimentação, tarefa esta realizada através do envio do verbo *Power State* (INTEL®, 2010, p. 151) responsável pela alimentação do codec. Desta forma, para geração de um simples tom, em DÓ Maior, através do widget *Beep Generator*, as seguintes chamadas de função são necessários:

```
#define HDA_VRB_GET_POWER_STATE = 0xF05;
#define DO (48000/261)

//Inicializa alimentacao do codec (inicialmente em power
//state standby)
GetCodecData8BitPayload(&PcieDeviceConfigSpace, 0x0,
                        0x1, HDA_VRB_SET_POWER_STATE,
                        0x0,  &Response);

//Make a beep
GetCodecData8BitPayload(&PcieDeviceConfigSpace, 0x0,
                      0x13,
                      HDA_VRB_SET_BEEP_GENERATION_CONTROL,
                      DO,  &Response);
```

Para comunicação com o codec a função GetCodecData8BitPayload deve ser acionada. Tal função apresenta a seguinte implementação, sendo esta implementação parte da biblioteca HdaLib desenvolvida para este projeto:

```
EFI_STATUS GetCodecData8BitPayload (
  PCI_HDA_REGION* PcieDeviceConfigSpace,
  UINT8 CodecAddress, UINT8 NodeId,
  HDA_VERB Verb, UINT8 VerbPayload,
  UINT32 *Response)
{
  HDA_COMMAND_FIELD_8BIT_PAYLOAD Comand;
  UINT32 VerbToSend = 0;

  Comand.CAd = CodecAddress;
  Comand.NID = NodeId;
  Comand.VerbIdent = Verb;
```

```
  Comand.VerbPayload = VerbPayload;

  CopyMem(&VerbToSend, &Comand, sizeof(UINT32));

  //write verb to immediate register ICOI
  WriteControllerRegister(PcieDeviceConfigSpace,
                  HDA_OFFSET_ICOI, VerbToSend, 1, 1, sizeof(UINT32));

  //start verb processing
  WriteControllerRegister(PcieDeviceConfigSpace,
                      HDA_OFFSET_ICIS,
                      HDA_START_PROCESSING_IMMEDIATE_COMMAND,
                      1, 1, sizeof(UINT32));

  //wait processing
  do {
          ReadControllerRegister(PcieDeviceConfigSpace,
                                 HDA_OFFSET_ICIS, Response,
                                 1, 1, sizeof(UINT32));
  } while ((*Response & 0x1) == 0x1);

  //read result
  ReadControllerRegister(PcieDeviceConfigSpace,
                         HDA_OFFSET_ICII, Response,
                         1, 1, sizeof(UINT32));

  return EFI_SUCCESS;
}
```

Esta função representa a implementação do algoritmo para envio e recebimento de informações para os codecs como apresentado na Seção 1.5, sendo este envio e recebimento realizados através dos registradores de acesso imediato (*Immediate Registers*) - implementação esta apresentada unicamente com caráter informativo, visto que tais registradores não são obrigatórios segundo a especificação HDA -, os quais se localizam no *controller register space* da controladora HDA existente no *chipset*, registradores estes acessados através das funções apresentadas a seguir, nas quais se faz uso do protocolo PciIoProtocol, definido pela UEFI Spec, como apresentado anteriormente na Seção 2.1.

```
EFI_STATUS WriteControllerRegister (
        PCI_HDA_REGION* PcieDeviceConfigSpace,
        UINT64 Offset,
        VOID* Value,
        UINTN Count,
        UINT8 BarIndex,
        EFI_PCI_IO_PROTOCOL_WIDTH Width)
{
```

```
        EFI_STATUS Status = EFI_SUCCESS;
        UINTN VariableWidth = 0;

        switch (Width){
                case EfiPciWidthUint32:
                        VariableWidth = sizeof(UINT32);
                        break;
                case EfiPciWidthUint16:
                        VariableWidth = sizeof(UINT16);
                        break;
                case EfiPciWidthUint8:
                        VariableWidth = sizeof(UINT8);
                        break;
                default:
                        Status = EFI_INVALID_PARAMETER;
                        goto FINISH;
        }

        VOID* ReadValue = AllocateZeroPool(VariableWidth);

        Status = HdaPciIoProtocol->Mem.Write(HdaPciIoProtocol,
                                             Width,
                                             BarIndex,
                                             Offset,
                                             Count,
                                             Value);

         //this read is needed to guarantee that the write
         //transaction
         //was done correctly, as described at tthe UEFI Driver
         //Writer's guide, page 349
         Status = HdaPciIoProtocol->Mem.Read(HdaPciIoProtocol,
                                              Width,
                                              BarIndex,
                                              Offset,
                                              Count,
                                              ReadValue);

         FreePool(ReadValue);

FINISH:

         return Status;
}

EFI_STATUS ReadControllerRegister (
        PCI_HDA_REGION* PcieDeviceConfigSpace,
        UINT64 Offset,
```

```
        VOID* Value,
        UINTN Count,
        EFI_PCI_IO_PROTOCOL_WIDTH Width)
{

        EFI_STATUS Status = EFI_SUCCESS;

    //fix base addres
    UINT64 HdaControllerBar = (PcieDeviceConfigSpace->HDBARL
                               & 0xFFFFFFF0 );

    Status = RootBridgePciIoProtocol->Mem.Read(RootBridgePciIoProtocol,
                                               Width,
                                               HdaControllerBar
                                                         + Offset,
                                               Count,
                                               Value);
        return Status;
}
```

Durante este trabalho, foi constatado que operações relacionadas a DMA não poderiam ser realizadas com a utilização dos registradores de acesso imediato. Tal informação foi posteriormente detectada na especificação HDA da seguinte forma (INTEL®, 2010, p. 50):

> *These registers can be implemented in platforms not suited for DMA command operations. If implemented, these registers must not be used at the same time as the CORB and RIRB command/response mechanisms, as the operations will conflict.*

O fato de operações relacionadas a DMA não poderem ser realizadas através dos registradores de uso imediato criou a necessidade da utilização dos registradores CORB e RIRB no lugar dos registradores de uso imediato. Esta necessidade elevou a complexidade do projeto, visto que os registradores CORB e RIRB se utilizam de buffers circulares, e criam a necessidade de inicialização de registradores adicionais para controle dos ponteiros conforme apresentado na Seção 1.5. As implementações relacionadas aos *immediate registers* foram mantidas para auxiliar em processos de depuração e devido a sua simplicidade, também útil para o entendimento do funcionamento da biblioteca desenvolvida.

Desta forma, a biblioteca desenvolvida apresenta abstrações que permitem acesso ao *PCIe configuration space* da controladora HDA, e aos registradores da controladora HDA. Funcionalidades estas reponsáveis pela comunicação com o codec e que permitem o acesso às controladoras DMA existentes na controladora HDA, além de funcionalidades

relacionadas a inicialização da controladora, apresentadas na Seção 2.2.1, e alocação dos buffers necessários para o processamento de *streams* de áudio, apresentadas na Seção 2.2.2.

2.2.1 Inicialização

Uma das tarefas necessárias para a inicialização da plataforma é a configuração denominada *No Snooping*, sendo este conceito proveniente da especificação PCI e mantido na especificação PCIe. O registrador no qual se encontra o bit responsável pela flag denominada *No Snooping Enabled*, é o registrador Device Control Register, apresentado na Figura 20.

DEVC—Device Control Register (Intel® High Definition Audio Controller—D27:F0)

Address Offset:	78h–79h	Attribute:	R/W, RO
Default Value:	0800h	Size:	16 bits

Bit	Description
15	Reserved
14:12	Max Read Request Size — RO. Hardwired to 0 enabling 128B maximum read request size.
11	**No Snoop Enable (NSNPEN)** — R/W. 0 = The Intel® High Definition Audio controller will not set the No Snoop bit. In this case, isochronous transfers will not use VC1 (VCi) even if it is enabled since VC1 is never snooped. Isochronous transfers will use VC0. 1 = The Intel High Definition Audio controller is permitted to set the No Snoop bit in the Requester Attributes of a bus master transaction. In this case, VC0 or VC1 may be used for isochronous transfers. **NOTE:** This bit is not reset on $D3_{HOT}$ to D0 transition; however, it is reset by PLTRST#.
10	Auxiliary Power Enable — RO. Hardwired to 0, indicating that Intel High Definition Audio device does not draw AUX power.
9	Phantom Function Enable — RO. Hardwired to 0 disabling phantom functions.
8	Extended Tag Field Enable — RO. Hardwired to 0 enabling 5-bit tag.
7:5	Max Payload Size — RO. Hardwired to 0 indicating 128B.
4	Enable Relaxed Ordering — RO. Hardwired to 0 disabling relaxed ordering.
3	Unsupported Request Reporting Enable — RO. Not implemented. Hardwired to 0.
2	Fatal Error Reporting Enable — RO. Not implemented. Hardwired to 0.
1	Non-Fatal Error Reporting Enable — RO. Not implemented. Hardwired to 0.
0	Correctable Error Reporting Enable — RO. Not implemented. Hardwired to 0.

Figura 20: Registrador *Device Control*. Fonte: (INTEL®, 2005, p. 29)

No *Device Control Register* a posição de número 11 contém o bit referente a configuração de *No Snooping*. Quando este bit é inicializado com valor 1, o *Root Complex* - apresentado na Seção 1.4 - passa a não permitir que informações escritas no *configuration space* da controladora sejam enviadas para memória cache do processador, garantindo desta forma que operações de leitura e escrita sejam realizadas diretamente na posição de memória desejada. Na biblioteca HdaLib, o acesso a essa configuração se dá através da função apresentada a seguir:

```
EFI_STATUS EnablePcieNoSnoop (PCI_HDA_REGION* PcieDeviceConfigSpace)
{
        UINT16 DeviceControlRegister = 0;
        EFI_STATUS Status = EFI_SUCCESS;

        //Read the command register from PCIe config space
        Status = ReadControllerPcieConfiguration(PcieDeviceConfigSpace,
                                        HDA_OFFSET_PCIE_DEVCTL,
                                        (VOID*) &DeviceControlRegister,
                                        1,
                                        EfiPciWidthUint16);

        Print(L"DeviceControlRegister: %x\r\n", DeviceControlRegister);

        //Set DisableInterrupt bit
        DeviceControlRegister += (1 << 11);

        Print(L"DeviceControlRegister: %x\r\n", DeviceControlRegister);

        WriteControllerPcieConfiguration(PcieDeviceConfigSpace,
                                          HDA_OFFSET_PCIE_DEVCTL,
                                          (VOID*) &DeviceControlRegister,
                                          1,
                                          EfiPciWidthUint16);

        return Status;
}
```

Outra funcionalidade relacionada a configuração da plataforma se dá através da configuração do registrador *Trafic Class Select Register*, apresentado na Figura 21, no qual o valor padrão a ser inicializado é o valor 0 sendo este valor definido pela Intel sem o fornecimento de maiores detalhes, para que os pacotes tenham prioridade quando trafegados no barramento. Tal ação pode ser realizada através da biblioteca HdaLib, com a chamada da seguinte função:

```
    UINT8 WriteValue = 0;
        WriteControllerRegister(PcieDeviceConfigSpace,
                                HDA_OFFSET_PCIE_TCSEL,
                                (VOID*) &WriteValue,
                                1,
                                0,
                                EfiPciWidthUint8);
```

Durante o desenvolvimento deste trabalho, um problema detectado foi a sinalização de uma falha no barramento PCIe através da *flag Master Abort* existente no registrador *Status Register* (localizado no *configuration space* da controladora de áudio), registrador este apresentado na Figura 11. Esta sinalização se apresenta sempre que a controladora

TCSEL—Traffic Class Select Register (Intel® High Definition Audio Controller—D27:F0)

Address Offset:	44h	Attribute:	R/W
Default Value:	00h	Size:	8 bits

This register assigned the value to be placed in the TC field. CORB and RIRB data will always be assigned TC0.

Bit	Description
7:3	Reserved.
2:0	**Intel® High Definition Audio Traffic Class Assignment (TCSEL)**— R/W. This register assigns the value to be placed in the Traffic Class field for input data, output data, and buffer descriptor transactions. 000 = TC0 001 = TC1 010 = TC2 011 = TC3 100 = TC4 101 = TC5 110 = TC6 111 = TC7 **NOTE:** These bits are not reset on $D3_{HOT}$ to D0 transition; however, they are reset by PLTRST#.

Figura 21: Registrador *Traffic Class Select Register*. Fonte: (INTEL®, 2005, p. 23)

de DMA responsável pelo processamento dos comandos existentes no buffer alocado para o registrador CORB era ativada. Em vista deste problema, duas possíveis falhas de implementação foram consideradas. Sendo esta a inicialização incorreta da controladora e do codec, ou erros de implementação dos ponteiros relacionados aos registradores CORB e RIRB. A fim de garantir a correta inicialização das controladoras, foi realizado um profundo estudo do processo de inicialização do driver de áudio para controladores compatíveis com a especificação HDA existentes na arquitetura ALSA (Advanced Linux Sound Architecture).

Outra funcionalidade disponibilizada pela biblioteca HdaLib é a desativação das interrupções *legacy* existentes na arquitetura PCIe. Neste processo foram realizados *dumps* das regiões de memória nas quais se encontravam a controladora HDA quando sendo executado no equipamento uma versão inicializável do sistema operacional Ubuntu em sua versão 18.04. Esta comparação resultou na detecção de inconsistência na configuração da controladora quando operando em sistema operacional Linux e ambiente pré-OS. Tais divergências se deram quanto a forma na qual as interrupções estavam configuradas na controladora, tarefa esta realizada pelo registrador *PCI Command Register* em seu bit número 10, de acordo com os detalhes apresentados na Figura 22.

Este bit, segundo a especificação HDA, precisa estar setado com valor 1 para que as interrupções sejam geradas corretamente, configuração esta não realizada adequadamente pelo firmware, sendo uma das responsáveis pelas falhas apresentadas no momento em que o processamento dos comandos através dos registradores CORB e RIRB era iniciado. Devido

PCICMD—PCI Command Register (Intel® High Definition Audio Controller—D27:F0)

Offset Address: 04h–05h | Attribute: R/W, RO
Default Value: 0000h | Size: 16 bits

Bit	Description
15:11	Reserved
10	**Interrupt Disable (ID)** — R/W. 0= The INTx# signals may be asserted. 1= The Intel® High Definition Audio controller's INTx# signal will be de-asserted **NOTE:** This bit does not affect the generation of MSIs.
9	Fast Back to Back Enable (FBE) — RO. Not implemented. Hardwired to 0.
8	SERR# Enable (SERR_EN) — R/W. SERR# is not generated by the ICH7 Intel High Definition Audio Controller.
7	Wait Cycle Control (WCC) — RO. Not implemented. Hardwired to 0.
6	Parity Error Response (PER) — RO. Not implemented. Hardwired to 0.
5	VGA Palette Snoop (VPS). Not implemented. Hardwired to 0.
4	Memory Write and Invalidate Enable (MWIE) — RO. Not implemented. Hardwired to 0.
3	Special Cycle Enable (SCE). Not implemented. Hardwired to 0.
2	**Bus Master Enable (BME)** — R/W. Controls standard PCI Express* bus mastering capabilities for Memory and I/O, reads and writes. Note that this bit also controls MSI generation since MSIs are essentially Memory writes. 0 = Disable 1 = Enable
1	**Memory Space Enable (MSE)** — R/W. Enables memory space addresses to the Intel High Definition Audio controller. 0 = Disable 1 = Enable
0	I/O Space Enable (IOSE)—RO. Hardwired to 0 since the Intel High Definition Audio controller does not implement I/O space.

Figura 22: Definição dos BITs segundo a especificação do chipset. Fonte: (INTEL®, 2005, p. 16)

a esta necessidade, a função DisablePcieInterrupts foi adicionada a biblioteca HdaLib, apresentando a seguinte implementação:

```
EFI_STATUS DisablePcieInterrupts (PCI_HDA_REGION* PcieDeviceConfigSpace)
{
        UINT16 CommandRegister = 0;
        EFI_STATUS Status = EFI_SUCCESS;

        //Read the command register from PCIe config space
        Status = ReadControllerPcieConfiguration(PcieDeviceConfigSpace,
                                                 HDA_OFFSET_PCIE_PCICMD,
                                                 &CommandRegister,
                                                 1,
                                                 EfiPciWidthUint16);

        Print(L"Command: %x\r\n", CommandRegister);
```

```
        //Set DisableInterrupt bit
        CommandRegister += (1 << 10);

        Print(L"Command: %x\r\n", CommandRegister);

        WriteControllerPcieConfiguration(PcieDeviceConfigSpace,
                                         HDA_OFFSET_PCIE_PCICMD,
                                         (VOID*) &CommandRegister,
                                         1,
                                         EfiPciWidthUint16);

        return Status;
}
```

Através da utilização das funções apresentadas anteriormente, o processo de inicialização e configuração pode ser realizado, como apresentado na função a seguir:

```
EFI_STATUS InitHdaControllerCodecAndBuffers(
                PCI_HDA_REGION* PcieDeviceConfigSpace,
        HDA_CONTROLLER_REGISTER_SET* ControllerRegisterSet)
{
        UINT32 WriteValue= 0;
        UINT32 Response = 0;

        DisablePcieInterrupts(PcieDeviceConfigSpace);

        EnablePcieNoSnoop(PcieDeviceConfigSpace);

        //Assign traffic priority to TC0
        //(TCSEL -> Traffic Class Select Register)
        WriteValue = 0;
        WriteControllerRegister(PcieDeviceConfigSpace,
                                HDA_OFFSET_PCIE_TCSEL,
                                (VOID*) &WriteValue,
                                1, 0, EfiPciWidthUint8);

        //Allocate the buffers to be used by the
        //CORB and RIRB DMA Engines
        AllocateRIRBBuffer(PcieDeviceConfigSpace);
        AllocateCORBBuffer(PcieDeviceConfigSpace);


        //Turn all nodes on
        SendCommandToAllWidgets8BitPayload(
                                PcieDeviceConfigSpace,
                            HDA_VRB_SET_POWER_STATE,
                            0x0);
```

```
        //Mute output widgets
        GetCodecData8BitPayloadCorbRirb(
                        PcieDeviceConfigSpace, 0x0, 0x10,
                        HDA_VRB_SET_AMPLIFIER_GAIN_MUTE,
                        INITIAL_VOLUME,  &Response);

        GetCodecData8BitPayloadCorbRirb(
                        PcieDeviceConfigSpace, 0x0, 0x11,
                        HDA_VRB_SET_AMPLIFIER_GAIN_MUTE,
                        INITIAL_VOLUME, &Response);

        return Status;
}
```

A função apresentada se inicia com a configuração das interrupções, desativação de cache das informações escritas na controladora HDA (*No Snoop*) e a definição da prioridade dos pacotes gerados no barramento PCIe por parte da controladora de audio. Em seguida é feita a alocação dos buffers necessários para o funcionamento dos registradores CORB e RIRB, conforme explicado mais detalhadamente na Seção 2.2.2.

2.2.2 Gerenciamento de *Buffers*

Outra funcionalidade desenvolvida na biblioteca HdaLib se refere a alocação dos recursos necessários para o correto funcionamento da infra-estrutura de áudio. Para alocação dos buffers para os registradores CORB a seguinte função foi desenvolvida (para o registrador RIRB o comportamento é bastante parecido):

```
EFI_STATUS AllocateCORBBuffer(PCI_HDA_REGION* PcieDeviceConfigSpace)
{

  EFI_STATUS Status = EFI_SUCCESS;
  UINT8* CorbAddressPointer = NULL;
  UINT32 ReadValue32 = 0;
  UINT32 WriteValue = 0;
  UINT8 WriteValue8 = 0;


  //Sets the corb address to a DMA allocated Buffer
  Status = SetupCommonBuffer(&CorbAddressPointer, 1024,
                             &CorbMapping, 16);

  CorbAddress = (UINT64) CorbAddressPointer;

  //Set Corb Lower Address
  WriteValue = (CorbAddress & 0xFFFFFFFF);
  WriteControllerRegister(PcieDeviceConfigSpace,
```

```
                        HDA_OFFSET_CORBLBASE,
                        (VOID*) &WriteValue,
                        1,
                        Bar0,
                        EfiPciWidthUint32);

  //Set Corb Upper Address
  WriteValue = (CorbAddress >> 32);
  WriteControllerRegister(PcieDeviceConfigSpace,
                        HDA_OFFSET_CORBUBASE,
                        (VOID*) &WriteValue ,
                        1,
                        Bar0,
                        EfiPciWidthUint32);

  //set write pointer to 0
  WriteValue = 0;
  WriteControllerRegister(PcieDeviceConfigSpace,
                        HDA_OFFSET_CORBWP,
                        (VOID*) &WriteValue,
                        1,
                        Bar0,
                        EfiPciWidthUint32);

  //Reset read pointer
  WriteValue = 0x8000;
  WriteControllerRegister(PcieDeviceConfigSpace,
                        HDA_OFFSET_CORBRP,
                        (VOID*) &WriteValue,
                        1,
                        Bar0,
                        EfiPciWidthUint32);

  ReadControllerRegister(PcieDeviceConfigSpace,
                        HDA_OFFSET_CORBRP,
                        (VOID*) &ReadValue32,
                        1,
                        EfiPciWidthUint32);

  WriteValue = 0;
  WriteControllerRegister(PcieDeviceConfigSpace,
                        HDA_OFFSET_CORBRP,
                        (VOID*) &WriteValue ,
                        1,
                        Bar0,
                        EfiPciWidthUint32);
```

```
  //start verb processing via CORB.
  WriteValue8 = 0x3;
  WriteControllerRegister(PcieDeviceConfigSpace,
                          HDA_OFFSET_CORBCTL,
                          (VOID*) &WriteValue8,
                          1,
                          Bar0,
                          EfiPciWidthUint8);

  Print(L"CorbAddress: %x\r\n", CorbAddress);
  Print(L"CorbMapping: %x\r\n", CorbMapping);

  return Status;
}
```

Para o correto processamento de comandos através dos registradores CORB/RIRB é obrigatório que o primeiro comando seja posicionado uma posição a frente do ponto inicial do buffer, conforme descrito pela especificação HDA (INTEL®, 2010, p. 63) a seguir:

> *When the CORB is first initialized, WP = 0, so the first command to be sent will be placed at offset (0 + 1) * 4 = 4 bytes, and WP would be updated to be 1.*

Devido ao fato das linhas de entrada e saída de dados da controladora HDA também serem gerenciadas por controladoras DMA, a alocação dos recursos necessários para operação das mesmas se faz necessário. Esta tarefa é realizada pela função apresentada a seguir:

```
EFI_STATUS AllocateStreamsPages(
                PCI_HDA_REGION* PcieDeviceConfigSpace,
                HDA_CONTROLLER_REGISTER_SET* ControllerRegisterSet)
{
        EFI_STATUS Status = EFI_SUCCESS;

        UINT8* BdlAddressPointer = NULL;
        VOID* BdlMapping = NULL;
        UINT64 BdlAddress;

        //These variables are related to the DMAPositionBuffer
        //Used for debugging
        UINT8* DmaAddressPointer = NULL;
        VOID* DmaMapping = NULL;
        UINT64 DmaAddress;

        UINT32 WriteValue = 0;
```

```
//Get the number od output streams, input streams and
//bidirectional streams based on the controllers
//capabilities
    UINTN OssCount =
        (ControllerRegisterSet->GCAP >> 12) & 0xF;;

    UINTN Count = 0;


    for(Count = 0; Count < OssCount; Count++) {
       SetupCommonBuffer(
            &BdlAddressPointer,
                    sizeof(HDA_BUFFER_DESCRIPTOR_LIST),
                    &BdlMapping, 16);

            BdlAddress = (UINT64) BdlAddressPointer;

            //Set Corb Lower Address
            WriteValue = (BdlAddress & 0xFFFFFFFF);
            WriteControllerRegister(
                        PcieDeviceConfigSpace,
                        CALCULATE_OSSN_OFFSET(Count,
                          ControllerRegisterSet->GCAP) +
                        HDA_RELATIVE_OFFSET_SDXBDPL,
                        (VOID*) &WriteValue,
                        1,
                        Bar0,
                        EfiPciWidthUint32);

            //Set Corb Lower Address
            WriteValue = (BdlAddress >> 32);
            WriteControllerRegister(
                        PcieDeviceConfigSpace,
                        CALCULATE_OSSN_OFFSET(Count,
                          ControllerRegisterSet->GCAP) +
                          HDA_RELATIVE_OFFSET_SDXBDPU,
                        (VOID*) &WriteValue ,
                        1,
                        Bar0,
                        EfiPciWidthUint32);
    }

    //Allocate the  DMAPosition Buffer
    //This buffer's size is explained at page 55 of the HDA spec
    SetupCommonBuffer(&DmaAddressPointer,
                     (IssCount + OssCount + BssCount )
```

```
                        * sizeof(UINT64),
                      &DmaMapping, 16);

        DmaAddress = (UINT64) DmaAddressPointer;

        //Set DMA Position Buffer Lower Address
        WriteValue = (DmaAddress & 0xFFFFFFFF);
        WriteValue |= 0x1; //The first bit is related to
                           //enabling the DMAPosition buffer
        WriteControllerRegister(
                    PcieDeviceConfigSpace,
                    HDA_OFFSET_DPIBLBASE,
                    (VOID*) &WriteValue,
                    1,
                    Bar0,
                    EfiPciWidthUint32);

        //Set DMA Position Buffer Lower Address
        WriteValue = (DmaAddress >> 32);
        WriteControllerRegister(
                        PcieDeviceConfigSpace,
                        HDA_OFFSET_DPIBUBASE,
                        (VOID*) &WriteValue ,
                        1,
                        Bar0,
                        EfiPciWidthUint32);

        return Status;
}
```

A função apresentada se inicia com a detecção do número de canais de saída suportados pela controladora HDA existente na plataforma, sendo assim realizada a alocação de uma lista de descritores para cada canal de saída. Adicionalmente é alocado um *buffer* para o *DMAPosition*, permitindo assim que seja realizada a depuração dos *streams* enquanto os mesmos são processados.

A função responsável pela conversão de uma cadeia de bytes obtidas a partir da extração dos pulsos PWM de um arquivo de áudio em um buffer que pode ser processado, processo este descrito na Seção 2.3.1, é apresentada a seguir:

```
EFI_STATUS AllocateResourcesBasedOnFile(
                PCI_HDA_REGION* PcieDeviceConfigSpace,
        HDA_CONTROLLER_REGISTER_SET* ControllerRegisterSet,
                UINTN FileSize,
                UINTN DataAddress)
{
        EFI_STATUS Status = EFI_SUCCESS;
```

```
    UINT32 WriteValue= 0;
    UINT8 WriteValue8= 0;
    UINT16 WriteValue16= 0;
    UINT32 Response = 0;

    VOID* AllignedDataMapping = NULL;

    //Used to split the sound data so the information can be
    //set at several descriptor entries
    UINTN BdlEntriesRequired = 0;
    UINTN BdlEntriesRequiredCurrentEntry = 0;

//Here the Buffer Descriptor List and the DMA
//Position buffers will be allocated and set at
//the controllers configuration space
    Status = AllocateStreamsPages(PcieDeviceConfigSpace,
                          ControllerRegisterSet);


    //Calculating how many entries at BDL are needed
    if(sizeof(SoundData) < 0xFFFFFFFF){
            BdlEntriesRequired = 1;
    } else {
            BdlEntriesRequired = sizeof(SoundData) /
                                        0xFFFFFFFF;

            //in case some extra entry is needed
            if(sizeof(SoundData) % 0xFFFFFFFF > 0) {
                    BdlEntriesRequired += 1;
            }
    }

    for(BdlEntriesRequiredCurrentEntry = 0;
                    BdlEntriesRequiredCurrentEntry < 2;
                    BdlEntriesRequiredCurrentEntry++) {

            //Allocate the buffers
            SetupCommonBuffer(
                &AllignedDataBufferBdlEntry,
                sizeof(SoundData) / BdlEntriesRequired,
                &AllignedDataMapping, 2); //dword aligned

            //Copy the sound data to the aligned buffer
            //so it can be set at the BDL
            CopyMem(AllignedDataBufferBdlEntry,
                            &SoundData,
```

```
                        sizeof(SoundData) /
                        BdlEntriesRequired);

        AddDestriptorListEntryOss0(
                PcieDeviceConfigSpace,
                ControllerRegisterSet,
                (UINT64) AllignedDataBufferBdlEntry,
                sizeof(SoundData) / BdlEntriesRequired,
                BdlEntriesRequiredCurrentEntry,
                BdlEntriesRequiredCurrentEntry+1);

}

//write the cyclic buffer length
//(the sum of all BDLE sizes)
WriteValue = sizeof(SoundData) * 2;
WriteControllerRegister(PcieDeviceConfigSpace,
                      CALCULATE_OSSN_OFFSET(0,
                        ControllerRegisterSet->GCAP) +
                        HDA_RELATIVE_OFFSET_SDXCBL,
                      (VOID*) &WriteValue,
                      1, 0, EfiPciWidthUint32);


//Setup the streamId on the codec nodes
//(StreamId: 1, Channelid: 0)
//Page 160 HDA Spec (we don't have
//support for multiple streams yet. Something like
//the linux sound server should be implemented to
//support that)
WriteValue8 = 0x10;
GetCodecData8BitPayloadCorbRirb(
                    PcieDeviceConfigSpace,
                    0,
                    0x10,
                    HDA_VRB_SET_CHANNEL_STREAM_ID,
                    WriteValue8,
                    &Response);

//Set FIFOSize
WriteValue = 4;
WriteControllerRegister(
                PcieDeviceConfigSpace,
                CALCULATE_OSSN_OFFSET(0,
                  ControllerRegisterSet->GCAP) +
                HDA_RELATIVE_OFFSET_SDXFIFOS,
                (VOID*) &WriteValue,
```

```
                    1,
                    0,
                    EfiPciWidthUint32);

        //Set the stream format at the controller
        //(2 channel, 16 bits, 44.1KHz)
        WriteValue16 = 0x4011;

        //Set stream format at the codec
        GetCodecData16BitPayloadCorbRirb(
                    PcieDeviceConfigSpace, 0x0,
                    0x10, HDA_VRB_SET_STREAM_FORMAT,
                    WriteValue16, &Response);

        //Set stream format at the codec
        GetCodecData16BitPayloadCorbRirb(
                    PcieDeviceConfigSpace, 0x0,
                    0x11, HDA_VRB_SET_STREAM_FORMAT,
                    WriteValue16, &Response);

        WriteControllerRegister(
                    PcieDeviceConfigSpace,
                    CALCULATE_OSSN_OFFSET(0,
                      ControllerRegisterSet->GCAP) +
                    HDA_RELATIVE_OFFSET_SDXFMT,
                    (VOID*) &WriteValue16,
                    1,
                    0,
                    EfiPciWidthUint16);

        return Status;
}
```

A função apresentada tem como objetivo a alocação de um *buffer* que em seguida recebe o conteúdo a ser processado pela controladora HDA. Após a alocação e cópia dos dados para esse *buffer*, o endereço do mesmo é adicionado à lista alocada anteriormente na função. Outro passo importante realizado por essa função é a configuração do *codec*, para que os *widgets* saibam o formato do fluxo de dados a ser processado, sendo este formato o mesmo configurado na controladora HDA.

Desta forma, HdaLib é uma biblioteca capaz de auxiliar na inicialização, gerenciamento e operação da infra-estrutura de áudio em ambiente pré-OS.

2.3 Bene Music Player

Com base em todas informações e conhecimentos adquiridos no decorrer desse projeto, foi desenvolvida uma prova de conceito comprovando a plena capacidade de utilização das funcionalidades disponibilizadas pela arquitetura de áudio ainda em ambiente pré-OS.

Foi construída uma aplicação chamada *Bene Music Player* capaz de realizar a inicialização da controladora de áudio existente no chipset da plataforma, a correta configuração da barramento PCIe, a alocação dos recursos necessário para comunicação entre o codec e o processador, a alocação dos recursos necessário para as controladoras DMA, e por fim o envio de cadeias de bytes contendo pulsos PWM. Estes pulsos são obtidos a partir de arquivos de áudio que devem seguir o formato Linux HeaderLess, o qual contém unicamente os pulsos PWM em seu conteúdo, sem compressão nem informações em qualquer tipo de cabeçalho. Para executar esta tarefa a aplicação utiliza a biblioteca HdaLib também desenvolvida neste trabalho como apresentado na Seção 2.2. Adicionalmente a aplicação possui a capacidade de controle de volume através de envio de *steps* ao codec de áudio e seus *widgets*.

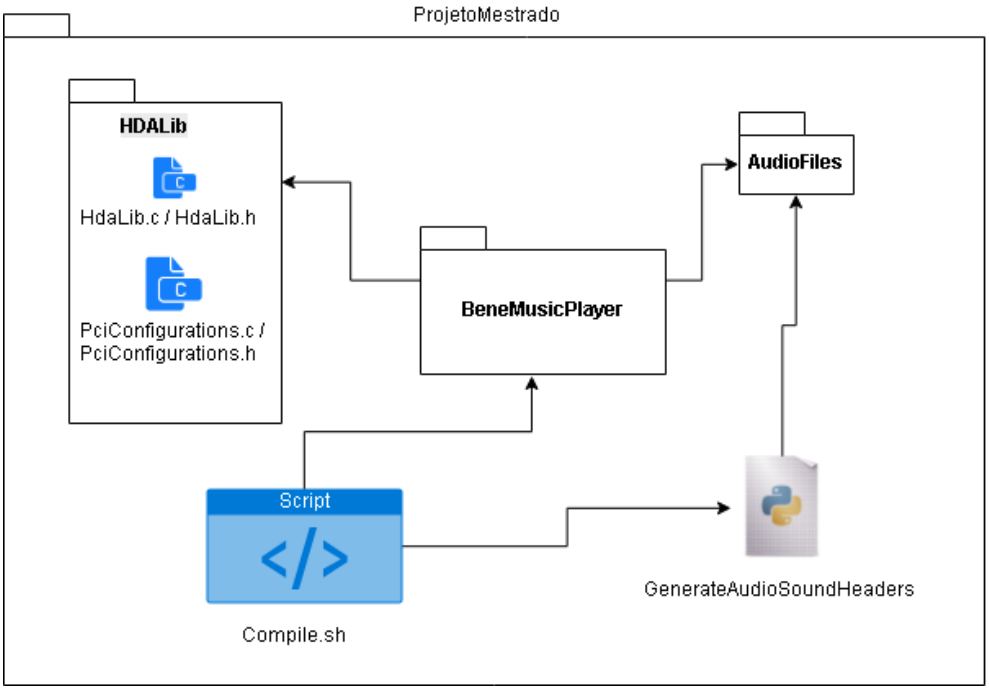

Figura 23: Componentes do Projeto

A Figura 23 apresenta os componentes e suas interações que possibilitam o funcionamento da aplicação *Bene Music Player*. A seguir a descrição da função e conteúdo de cada componente do projeto:

BeneMusicPlayer: Aplicativo para ambiente pré-OS (UEFI), capaz de executar um *stream* de áudio, criado por arquivos de áudio integrados a esta aplicação durante sua

compilação, através da placa de áudio fazendo uso das capacidades disponibilizadas pelo chipset e codec HDA, através da utilização da biblioteca HdaLib.

HdaLib: Biblioteca desenvolvida como o ponto central desse trabalho, na qual existem abstrações para tarefas relacionadas a arquitetura PCI-e, chipset e controladoras/-codecs HDA.

Compile.sh: Script responsável por executar o processo de compilação, no qual serão gerados os arquivos .h com os pulsos PWM dos arquivos de áudio, posteriormente iniciando o processo de compilação do aplicativo com auxilio do *framework* edk2 (Seção 2.3.1).

GenerateAudioSoundFileHeaders.py: Script Python responsável por extrair os pulsos PWM dos arquivos de áudio existentes no projeto, criando assim os arquivos .h a serem integrado ao tocador de áudio. Vale ressaltar que esse script é capaz somente de trabalhar com arquivos no formato linux-header-less, nos quais os pulsos PWM possuem profundidade de 16 bits.

AudioFiles: Esse diretório possui os arquivos de áudio que serão convertidos em arquivos .h durante o processo de compilação. Permitindo assim sua inclusão diretamente no executável final. Esta abordagem foi adotada para evitar a necessidade de criação de um interpretador de formatos de áudio específico. Desta forma, somente os pulsos PWM existentes em um dado arquivo de áudio são incluídos ao projeto.

A função inicial do tocador é apresentada a seguir:

```
EFI_STATUS
EFIAPI
UefiMain (
  EFI_HANDLE        ImageHandle,
  EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS Status = EFI_SUCCESS;

  //Registers from the HDA controller
  HDA_CONTROLLER_REGISTER_SET ControllerRegisterSet;
  PCI_HDA_REGION PcieDeviceConfigSpace;

  UINT32 WriteValue= 0;
  UINT16 CurrentVolume = INITIAL_VOLUME;

  Status = InitHdaLib();

  if(!EFI_ERROR(Status)){

        Status = GetPcieConfigSpace(
```

```
                HDA_BUS, HDA_DEV,
                        HDA_FUNC, &PcieDeviceConfigSpace);

        if(!EFI_ERROR(Status)){
                GetControllerRegisterSet(
                &PcieDeviceConfigSpace,
                                &ControllerRegisterSet);

                InitHdaControllerCodecAndBuffers(
                        &PcieDeviceConfigSpace,
                                &ControllerRegisterSet);


                AllocateResourcesBasedOnFile(
                        &PcieDeviceConfigSpace,
                                &ControllerRegisterSet, 0,0);

                //set the stream id and play
                Print(L"Ready to play");

                //Stream run.
                WriteValue = 0x100002;

                WriteControllerRegister(
                    &PcieDeviceConfigSpace,
                        CALCULATE_OSSN_OFFSET(0,
                            ControllerRegisterSet.GCAP) +
                                HDA_RELATIVE_OFFSET_SDXCTL,
                        (VOID*) &WriteValue,
                        1, 0, EfiPciWidthUint32);
                Print(L"Play done!!!\r\n");
        }
  } else {
          Print(L"failure to init HdaLib\r\n");
  }
  return Status;
}
```

A função apresentada realiza a inicialização da controladora e do codec, e em seguida faz a alocação dos *buffers* que serão utilizados posteriormente pelas controladoras de DMA, sendo estas atividades realizadas através da biblioteca HdaLib. Apresenta-se portanto uma prova de conceito funcional, capaz de processar *streams* de áudio através da utilização da controladora HDA em conjunto com o codec existente na plataforma. Observa-se que nas funções apresentadas diversas macros são utilizadas. Tais macros podem ser encontradas no Apêndice B, no qual existem macros que representam *offsets* definidos pela especificação HDA, bem como macros auxiliares utilizadas para facilitar a

utilização da biblioteca desenvolvida.

2.3.1 Pré-Processamento

Durante o processo de compilação da aplicação *Bene Music Player* existe a necessidade de realização de pré-processamento dos arquivos de áudio, processo este necessário para que sejam adicionados aos código do tocador de áudio arquivos contendo os dados a serem enviados para a placa de som. Esse processo se dá através da execução do script denominado GenerateAudioSoundFileHeaders.py, o qual apresenta a seguinte implementação:

```
#!/usr/bin/env python

#This file is used to generate a .h based on a binary file's content
#so it can be accessed as a simple array

#Import the necessary libs
import sys, os.path
import binascii

#loop the files to be processed
for filename in sys.argv[1:]:

    #get the file size so the array can be created with the
    #correct number of elements
    filesize = os.path.getsize(filename)

    #open the file in binary mode
    origfile =  open(filename, "rb")

    #Creates the array with the correct name and size
    output = "static UINT8 SoundData[%d] = {\n" % (filesize)

    #add each byte from the file to an string that will be
    #used later to create the final file
    for i in range(0, filesize):
        stringValue = binascii.hexlify(origfile.read(1))
        output = output + " 0x%s," % stringValue
        if (i % 12) == 11:
            output = output + "\n"
    output = output + "\n};\n"

    #create the .h file with the binary file's content on
    #an integer array
    f = file("%s.h" % filename , "w")
    f.write(output)
    f.close()
```

```
print "Done!"
```

Este script, tem como responsabilidade a criação de um arquivo .h contendo um *array* no qual estão localizados os pulsos PWM que representam um dado som a ser processado pela infra-estrutura de áudio existente no equipamento.

Vale ressaltar que esta abordagem se faz necessária afim de evitar a necessidade do desenvolvimento de um interpretador de outros formatos de áudio mais complexos, como é o caso dos formatos MP3 e WMA, visto que tais interpretadores possuem grande complexidade.

O exemplo a seguir apresenta o conteúdo de um arquivo de áudio no formato *Linux Header-Less*, arquivo este gerado através da utilização de ferramentas de conversão e tratamento de áudio como o Audacity (MAZZONI DOMINIC; DANNENBERG, 1999), a partir de um arquivo em formatos como MP3 e WMA:

```
cb03 ed0f c502 c90f 4ffe 390e 85fa 450d
4bfa ad0c 94fa 280b 17f9 0609 85f9 c306
```

Após ser processado pelo script mencionado anteriormente, resulta no arquivo .h apresentado a seguir:

```
static UINT8 SoundData[32] = {
 0xcb, 0x03, 0xed, 0x0f, 0xc5, 0x02, 0xc9, 0x0f, 0x4f, 0xfe, 0x39, 0x0e,
 0x85, 0xfa, 0x45, 0x0d, 0x4b, 0xfa, 0xad, 0x0c, 0x94, 0xfa, 0x28, 0x0b,
 0x17, 0xf9, 0x06, 0x09, 0x85, 0xf9, 0xc3, 0x06 };
```

Desta forma, é possível a conversão de um arquivo de áudio no formato *Linux Header-Less* em um arquivo .h, permitindo assim que o mesmo seja incluído ao processo de compilação do tocador de áudio.

Conclusão

A existência de funcionalidades pensadas especificamente para pessoas com algum tipo de necessidade especial tem grande impacto na sociedade, pois permitem que um maior número de pessoas tenha acesso às mesmas informações e oportunidades, fazendo com que diferenças na forma de realização de tarefas não sejam uma barreira. A impossibilidade de os deficientes visuais utilizarem os BIOS de seus computadores, devido a falta de um leitor de telas para ambiente pré-OS, claramente vai contra os direitos da pessoa com deficiência, como apresentado no início deste trabalho. O projeto apresentado teve desde sua concepção o objetivo de abrir a primeira porta necessária para que a inclusão de acessibilidade em ambientes pré-OS fosse possível, através da comprovação do funcionamento das controladoras de áudio em ambiente pré-OS. Adicionalmente, foi feita a disponibilização de um código base a ser utilizado pelos interessados em tal funcionalidade no repositório https://github.com/RafaelRMachado/Ms_UefiHda_PreOs_Accessibility.git. Vale ressaltar que o projeto possui caráter experimental, e portanto não foi validado em outras plataformas se não a previamente descrita. Uma possível evolução deste projeto seria a migração do projeto de forma a compor um *driver* DXE, que poderia ser embarcado nas plataformas e posteriormente utilizado em conjunto com um leitor de tela para ambiente pré-OS, sendo tal leitor de tela outro projeto a ser desenvolvido futuramente. Trabalhos relacionados ao desenvolvimento de formas de compressão dos arquivos de áudio, algo não realizado neste trabalho, também poder ser explorados em projetos futuros. Outras melhorias podem ser adicionadas ao código base apresentado podendo esta evolução chegar ao ponto em que outras arquiteturas, como por exemplo arquiteturas ARM, sejam suportadas.

Referências

ALLEN, S. et al. A voice output module developed for a blind programmer. *Journal of Visual Impairment & Blindness*, v. 75, n. 4, p. 157–161, 1981. Citado na página 29.

ANDERSON, D.; SHANLEY, T. *PCI system architecture*. [S.l.]: Addison-Wesley Professional, 1999. Citado na página 42.

BUDRUK, R.; ANDERSON, D.; SHANLEY, T. *PCI express system architecture*. [S.l.]: Addison-Wesley Professional, 2004. Citado 7 vezes nas páginas 15, 43, 44, 45, 46, 47 e 48.

CONEXANT®. *CX20752 Low-Power High Definition Audio CODEC Data Sheet*. 2015. Documento número 004-52DSR02. Citado 2 vezes nas páginas 15 e 60.

DAVID JANUS SCOTT, J. W. R. *High Definition Audio for the Digital Home - Proven Techniques for Getting It Right the First time*. [S.l.]: Intel Press, 2006. Páginas 49-50. Citado na página 48.

DUDDINGTON, J. *eSpeak NG*. 1995. Acessado em: 2018-11-24. Disponível em: <https://github.com/rhdunn/espeak>. Citado na página 31.

DUDDINGTON, R. H. *PC Audio Lib*. 2016. Acessado em: 2018-11-24. Disponível em: <https://github.com/rhdunn/pcaudiolib>. Citado na página 33.

GOLDISH, L. H.; TAYLOR, H. E. The optacon: A valuable device for blind persons. *New Outlook for the Blind*, ERIC, v. 68, n. 2, p. 49–56, 1974. Citado na página 27.

HENDRICKS, A. Un convention on the rights of persons with disabilities. *Eur. J. Health L.*, HeinOnline, v. 14, p. 273, 2007. Citado na página 23.

INTEL®. *Intel I/O Controller Hub 7 (ICH7) /Intel High Definition Audio / AC'97*. 2005. Documento número 307017-001. Disponível em: <http://www.intel.com/content/www/us/en/io/io-controller-hub-7-hd-audio-ac97-manual.html>. Citado 10 vezes nas páginas 15, 41, 42, 43, 47, 50, 59, 65, 67 e 68.

INTEL®. *High Definition Audio Specification*. 2010. Página 16. Citado 10 vezes nas páginas 15, 25, 49, 51, 52, 53, 54, 61, 64 e 72.

LINUX. *HdaController.c*. 2018. Acessado em: 2018-11-24, Linha 891. Disponível em: <https://github.com/torvalds/linux/blob/master/sound/pci/hda/hda_controller.c>. Citado na página 52.

LINUX. *KVM*. 2018. Acessado em: 2018-09-09. Disponível em: <https://www.linux-kvm.org/page/Main_Page>. Citado na página 58.

LINVILL, J. G. *Reading aid for the blind*. Google Patents, 1966. US Patent 3,229,387. Disponível em: <https://www.google.com/patents/US3229387>. Citado 3 vezes nas páginas 15, 27 e 28.

MACHADO, R. R. *Projeto Mestrado - Bios Acessível*. 2016. <https://groups.google.com/d/topic/cegos_programadores/o7uXDaCOCqs/discussion>. Disponível em: <ProjetoMestrado-BiosAcessível>. Citado na página 24.

MAZZONI DOMINIC; DANNENBERG, R. *Audacity*. 1999. Acessado em: 2018-11-29. Disponível em: <https://www.audacityteam.org>. Citado na página 82.

MEISELWITZ, G.; WENTZ, B.; LAZAR, J. Universal usability: Past, present, and future. *Human–Computer Interaction*, v. 3, n. 4, p. 213–333, 2009. Citado na página 23.

MICROSOFT®. *IXAudio2 Interface*. 2018. Acessado em: 2018-11-24. Disponível em: <https://docs.microsoft.com/pt-br/windows/desktop/api/xaudio2/nn-xaudio2-ixaudio2>. Citado na página 35.

MICROSOFT®. *Overview Microsoft Active Accessibility*. 2018. Acessado em: 2018-11-24. Disponível em: <https://docs.microsoft.com/en-us/windows/desktop/winauto/technical-overview>. Citado na página 30.

NVACCESS. *Our Story*. 2015. Acessado em: 2015-09-13. Disponível em: <http://www.nvaccess.org/about/our-story/>. Citado na página 30.

QEMU. *QEMU*. 2018. Acessado em: 2018-09-09. Disponível em: <https://www.qemu.org/>. Citado na página 58.

SHERMAN, P. *Cost-justifying accessibility*. 2001. <https://www.ischool.utexas.edu/~l385t21/AU_WP_Cost_Justifying_Accessibility.pdf>. Acessado em: 01-05-2016. Citado na página 36.

SHNEIDERMAN, B. Universal usability. *Communications of the ACM*, ACM, v. 43, n. 5, p. 84–91, 2000. Citado na página 23.

TIANOCORE. *Lista de E-mail Edk2*. 2018. Acessado em: 2018-11-24. Disponível em: <https://github.com/tianocore/tianocore.github.io/wiki/Mailing-Lists>. Citado na página 24.

TIANOCORE. *OVMF*. 2018. Acessado em: 2018-09-09. Disponível em: <https://github.com/tianocore/tianocore.github.io/wiki/OVMF>. Citado na página 58.

TIANOCORE. *Tianocore.org*. 2018. Acessado em: 2018-08-18. Disponível em: <https://www.tianocore.org/>. Citado na página 24.

UEFI Forum. *Unified Extensible Firmware Interface Specification*. 2014. Citado 5 vezes nas páginas 24, 25, 38, 40 e 57.

UEFI Forum. *UEFI Shell Specification*. 2016. Citado na página 59.

VANDERHEIDEN, G. C. Ubiquitous accessibility, common technology core, and micro assistive technology: Commentary on computers and people with disabilities. *ACM Trans. Access. Comput.*, ACM, New York, NY, USA, v. 1, n. 2, p. 10:1–10:7, out. 2008. ISSN 1936-7228. Disponível em: <http://doi.acm.org/10.1145/1408760.1408764>. Citado na página 23.

VINCENT, Z.; MICHAEL, R.; SURESH, M. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. [S.l.]: Intel Press, 2010. Página 13. Citado 2 vezes nas páginas 15 e 39.

W3C. *Case Study of Accessibility Benefits: Legal & General Group (L&G)*. 2015. Disponível em: <http://www.w3.org/WAI/bcase/legal-and-general-case-study>. Citado na página 36.

World Health Organization. *Visual impairment and blindness*. 2014. Disponível em: <http://www.who.int/mediacentre/factsheets/fs282/en/>. Citado na página 36.

APÊNDICE A – Mail Discussion

Esse apêndice possui a discussão de e-mail iniciada por Matthew Bradley com relação ao possível desenvolvimento de um *Screen Reader* para o BIOS de seu computador. O idioma original foi mantido a fim de evitar perda de sentido devido a tradução. Os e-mails dos envolvidos foram removidos a fim de manter sua privacidade.

On Apr 11, 2014, at 5:55 PM, Matthew Bradley wrote:

Hello, I am a blind computer user. I was told I could try and contact you all to see if something I was looking for was possible. I need to see if there is any way to get a screen reader to work inside the UEFI menus. I know it would probably mean writing a screen reader for the UEFI environment. I just want to see if this is even possible. I was a power user prior to losing my vision. I have found that I am able to get most things done now, even with the loss of my site. I have not, however, found any accessible UEFI firmware. I would like to see if there is any way for us to make this part of the computer accessible also. I know that screen readers are complex, but if I could get it to read it, even one letter at a time, it would allow me to make changes if needed. Any response you could give would be appreciated, even if it is just to say it is not possible.

On Apr 11, 2014, at 23:09 PM, Andrew wrote:

Matthew, Is the screen reader pure software? If so that would difficult in UEFI as there are no multimedia abstractions for sound devices. There are also size constraints with the size of the ROM. If it is support for some kind of hardware device that might be easier to implement. There are already serial friendly text in and text out abstractions in UEFI, and it is easy to support multiple console devices at the same time.

This is the open source site, and it has nothing to do with the UEFI standard that is owned by Unified Extensible Firmware Interface Forum. The UEFI Forum sponsors industry interoperability events so if there was a driver and device to test these events would be the place to do it. The UEFI Forum would also be the place to discuss accessibility APIs, or changes to the UEFI specification to enhance accessibility. The UEFI forum website is http://www.uefi.org.

Thanks,

Andrew

APÊNDICE B – HdaLib.h

Este apêndice tem como objetivo a documentação das macros criadas durante o desenvolvimento do projeto descrito neste documento. Trata-se das definições da biblioteca batizada HdaLib, sendo este arquivo denominado HdaLib.h.

```
#ifndef _HDA_LIB_H
#define _HDA_LIB_H

#include<Uefi.h>
#include <IndustryStandard/Pci.h>
#include <Protocol/PciIo.h>

//#include "PciConfigurations.h"

/** @file
  This Library contains definitions and functions
  needed to work with Intel High Definition compatible
  controllers and codecs.
**/


/* Location of the Intel HDA Controller defined at
 * intel's chipset manuals
 */
#define HDA_BUS 0
#define HDA_DEV 27
#define HDA_FUNC 0


/*
 * As described at HDA Spec, page 57:
 * The Buffer Descriptor List (BDL) is a memory structure that describes
 * the buffers in memory. The BDL is comprised of a series of Buffer
 * Descriptor List Entries. There must be at least two entries in the
 * list, with a maximum of 256 entries. Also, the start of the structure
 * must be aligned on a 128
*/
#define HDA_BUFFER_DESC_LIST_MAX_ENTRIES 256


///
/// HD header defined at the IO-Controller_Hub-7-hd-audio-ac97 manual
/// As described at page 13 of the documentation, the addresses
/// not defined should be treated as reserved
```

```
///
typedef struct {
        UINT16  VID;
        UINT16  DID;
        UINT16  PCICMD;
        UINT16  PCISTS;
        UINT8   RID;
        UINT8   PI;
        UINT8   SC;
        UINT8   BCC;
        UINT8   CLS;
        UINT8   LT;
        UINT8   HEADTYP;
        UINT8   RES0;
        UINT32  HDBARL;
        UINT32  HDBARU;
        UINT8   RESV1[20];
        UINT16  SVID;
        UINT16  SID;
        UINT8   RESV2[4];
        UINT8   CAPPTR;
        UINT8   RESV3[7];
        UINT8   INTLN;
        UINT8   INTPN;
        UINT8   RESV4[2];
        UINT8   HDCTL;
        UINT8   RESV5[3];
        UINT8   TCSEL;
        UINT8   RESV6[8];
        UINT8   DCKSTS;
        UINT8   RESV7[2];
        UINT16  PID;
        UINT16  PC;
        UINT32  PCS;
        UINT8   RESV8[8];
        UINT16  MID;
        UINT16  MMC;
        UINT32  MMLA;
        UINT32  MMUA;
        UINT16  MMD;
        UINT8   RESV9[2];
        UINT16  PXID;
        UINT16  PXC;
        UINT32  DEVCAP;
        UINT16  DEVCTL;
        UINT16  DEVS;
        UINT8   RESV10[132];
```

```
        UINT32  VCCAP;
        UINT32  PVCCAP1;
        UINT32  PVCCAP2;
        UINT16  PVCCTL;
        UINT16  PVCSTS;
        UINT32  VC0CAP;
        UINT32  VC0CTL;
        UINT8   RESV11[2];
        UINT16  VC0STS;
        UINT32  VciCAP;
        UINT32  VciCTL;
        UINT8   RESV12[2];
        UINT16  VciSTS;
        UINT8   RESV13[8];
        UINT32  RCCAP;
        UINT32  ESD;
        UINT8   RESV14[8];
        UINT32  L1DESC;
        UINT8   RESV15[4];
        UINT32  L1ADDL;
        UINT32  L1ADDU;
} PCI_HDA_REGION;


/* Offsets of the registers defined at HDA PCie config space.
 * Details can be found at the Intel I/O Controller Hub 7 (ICH7)/
 * Intel High Definition Audio / AC 97 Programmer's Reference Manual
 * April 2005
*/
#define HDA_OFFSET_PCIE_VID     0x0
#define HDA_OFFSET_PCIE_DID     0x2
#define HDA_OFFSET_PCIE_PCICMD  0x4
#define HDA_OFFSET_PCIE_PCISTS  0x6
#define HDA_OFFSET_PCIE_RID     0x8
#define HDA_OFFSET_PCIE_PI      0x9
#define HDA_OFFSET_PCIE_SCC     0xA
#define HDA_OFFSET_PCIE_BCC     0xB
#define HDA_OFFSET_PCIE_CLS     0xC
#define HDA_OFFSET_PCIE_LT      0xD
#define HDA_OFFSET_PCIE_HEADTYP 0xE
#define HDA_OFFSET_PCIE_HDBARL  0x10
#define HDA_OFFSET_PCIE_HDBARU  0x14
#define HDA_OFFSET_PCIE_SVID    0x2C
#define HDA_OFFSET_PCIE_SID     0x2E
#define HDA_OFFSET_PCIE_CAPPTR  0x34
#define HDA_OFFSET_PCIE_INTLN   0x3C
#define HDA_OFFSET_PCIE_INTPN   0x3D
```

```
#define HDA_OFFSET_PCIE_HDCTL   0x40
#define HDA_OFFSET_PCIE_TCSEL   0x44
#define HDA_OFFSET_PCIE_DCKSTS  0x4D
#define HDA_OFFSET_PCIE_PID     0x50
#define HDA_OFFSET_PCIE_PC      0x52
#define HDA_OFFSET_PCIE_PCS     0x54
#define HDA_OFFSET_PCIE_MID     0x60
#define HDA_OFFSET_PCIE_MMC     0x62
#define HDA_OFFSET_PCIE_MMLA    0x64
#define HDA_OFFSET_PCIE_MMUA    0x68
#define HDA_OFFSET_PCIE_MMD     0x6C
#define HDA_OFFSET_PCIE_PXID    0x70
#define HDA_OFFSET_PCIE_PXC     0x72
#define HDA_OFFSET_PCIE_DEVCAP  0x74
#define HDA_OFFSET_PCIE_DEVCTL  0x78
#define HDA_OFFSET_PCIE_DEVS    0x7A
#define HDA_OFFSET_PCIE_VCCAP   0x100
#define HDA_OFFSET_PCIE_PVCCAP1 0x104
#define HDA_OFFSET_PCIE_PVCCAP2 0x108
#define HDA_OFFSET_PCIE_PVCCTL  0x10C
#define HDA_OFFSET_PCIE_PVCSTS  0x10E
#define HDA_OFFSET_PCIE_VC0CAP  0x110
#define HDA_OFFSET_PCIE_VC0CTL  0x114
#define HDA_OFFSET_PCIE_VC0STS  0x11A
#define HDA_OFFSET_PCIE_VciCAP  0x11C
#define HDA_OFFSET_PCIE_VciCTL  0x120
#define HDA_OFFSET_PCIE_VciSTS  0x126
#define HDA_OFFSET_PCIE_RCCAP   0x130
#define HDA_OFFSET_PCIE_ESD     0x134
#define HDA_OFFSET_PCIE_L1DESC  0x140
#define HDA_OFFSET_PCIE_L1ADDL  0x148
#define HDA_OFFSET_PCIE_L1ADDU  0x14C



/* Offsets of the registers defined at HDA
 * compatible controllers.
 * Details can be found at the page 27 of the
 * HDA Specification Rev 1.0a
*/
#define HDA_OFFSET_GCAP 0x0
#define HDA_OFFSET_VMIN 0x2
#define HDA_OFFSET_VMAJ 0x3
#define HDA_OFFSET_OUTPAY 0x4
#define HDA_OFFSET_INPAY 0x6
#define HDA_OFFSET_GCTL 0x8
#define HDA_OFFSET_WAKEEN 0x0C
```

```
#define HDA_OFFSET_WAKESTS 0x0E
#define HDA_OFFSET_GSTS 0x10
#define HDA_OFFSET_Rsvd 0x12
#define HDA_OFFSET_OUTSTRMPAY 0x18
#define HDA_OFFSET_INSTRMPAY 0x1A
#define HDA_OFFSET_INTCTL 0x20
#define HDA_OFFSET_INTSTS 0x24
#define HDA_OFFSET_WALCLK 0x30
#define HDA_OFFSET_SSYNC 0x38
#define HDA_OFFSET_CORBLBASE 0x40
#define HDA_OFFSET_CORBUBASE 0x44
#define HDA_OFFSET_CORBWP 0x48
#define HDA_OFFSET_CORBRP 0x4A
#define HDA_OFFSET_CORBCTL 0x4C
#define HDA_OFFSET_CORBSTS 0x4D
#define HDA_OFFSET_CORBSIZE 0x4E
#define HDA_OFFSET_RIRBLBASE 0x50
#define HDA_OFFSET_RIRBUBASE 0x54
#define HDA_OFFSET_RIRBWP 0x58
#define HDA_OFFSET_RINTCNT 0x5A
#define HDA_OFFSET_RIRBCTL 0x5C
#define HDA_OFFSET_RIRBSTS 0x5D
#define HDA_OFFSET_RIRBSIZE 0x5E
#define HDA_OFFSET_ICOI 0x60
#define HDA_OFFSET_ICII 0x64
#define HDA_OFFSET_ICIS 0x68
#define HDA_OFFSET_DPIBLBASE 0x70
#define HDA_OFFSET_DPIBUBASE 0x74

//This is the last offset before the descriptors.
//DO NOT change or remove this definition
#define HDA_OFFSET_SD0CTL 0x80
#define HDA_OFFSET_SD0STS 0x83
#define HDA_OFFSET_SD0LPIB 0x84
#define HDA_OFFSET_SD0CBL 0x88
#define HDA_OFFSET_SD0LVI 0x8C
#define HDA_OFFSET_SD0FIFOS 0x90
#define HDA_OFFSET_SD0FMT 0x92
#define HDA_OFFSET_SD0BDPL 0x98
#define HDA_OFFSET_SD0BDPU 0x9C

/*
 * These definitions are relative to each
 * descriptor.
 * Details can be found at the page 27 of the
 * HDA Specification Rev 1.0a
*/
```

```
#define HDA_RELATIVE_OFFSET_SDXCTL 0
#define HDA_RELATIVE_OFFSET_SDXSTS 0x3
#define HDA_RELATIVE_OFFSET_SDXLPIB 0x4
#define HDA_RELATIVE_OFFSET_SDXCBL 0x8
#define HDA_RELATIVE_OFFSET_SDXLVI 0xC
#define HDA_RELATIVE_OFFSET_SDXFIFOS 0x10
#define HDA_RELATIVE_OFFSET_SDXFMT 0x12
#define HDA_RELATIVE_OFFSET_SDXBDPL 0x18
#define HDA_RELATIVE_OFFSET_SDXBDPU 0x1C


//Extract the OSS count from the controller's Global
//capabilities register
#define HDA_OSS_COUNT(GCAP) \
        ((GCAP >> 12) & 0xF)

//Extract the ISS count from the controller's Global
//capabilities register
#define HDA_ISS_COUNT(GCAP) \
        ((GCAP >> 8) & 0xF)

//Extract the BSS count from the controller's Global
//capabilities register
#define HDA_BSS_COUNT(GCAP) \
        ((GCAP >> 3) & 0x1F)


//These macros calculate the offset of the
//stream descriptors as defined at page 27 of the HDA Spec
#define CALCULATE_ISSN_OFFSET(StreamIndex) \
        (HDA_OFFSET_SD0CTL + (StreamIndex * 0x20))

#define CALCULATE_OSSN_OFFSET(StreamIndex,GCAP) (HDA_OFFSET_SD0CTL + \
  (HDA_ISS_COUNT(GCAP) * 0x20 ) + \
  (StreamIndex * 0x20))

#define CALCULATE_BSSN_OFFSET(StreamIndex,GCAP) (HDA_OFFSET_SD0CTL + \
  (HDA_ISS_COUNT(GCAP) * 0x20) + \
  (HDA_OSS_COUNT(GCAP) * 0x20) + \
  (StreamIndex * 0x20))


typedef struct {
        UINT8   StreamReset: 1;
        UINT8   StreamRun: 1;
        UINT8   InterruptOnCompletionEnable: 1;
        UINT8   FIFOErrorInterruptEnable: 1;
```

```
        UINT8   DescriptorErrorInterruptEnable: 1;
        UINT8   Reserved1: 3;
        UINT8   Reserved2;
        UINT8   StrippeControl: 2;
        UINT8   TrafficPriority: 1;
        UINT8   BidirectionalDirectionControl: 1;
        UINT8   StreamNumber: 4;

} HDA_CONTROLLER_STREAM_DESCRIPTOR_CONTROL;


typedef struct {
        UINT8   SD0CTL[3]; //details of this property can be found at the
                           //HDA_CONTROLLER_STREAM_DESCRIPTOR_CONTROL struct
        UINT8   SD0STS;
        UINT32  SD0LPIB;
        UINT32  SD0CBL;
        UINT16  SD0LVI;
        UINT16  Rsvd8;
        UINT16  SD0FIFOS;
        UINT16  SD0FMT;
        UINT32  Rsvd9;
        UINT32  SD0BDPL;
        UINT32  SD0BDPU;
} HDA_CONTROLLER_STREAM_DESCRIPTOR;



/* Structure representing the HDA controller
 * register set. Details can be found at
 * the page 27 of the HDA Specification Rev 1.0a
*/
typedef struct {
        UINT16  GCAP;
        UINT8   VMIN;
        UINT8   VMAJ;
        UINT16  OUTPAY;
        UINT16  INPAY;
        UINT32  GCTL;
        UINT16  WAKEEN;
        UINT16  STATESTS;
        UINT16  GSTS;
        UINT8   Rsvd0[6];
        UINT16  OUTSTRMPAY;
        UINT16  INSTRMPAY;
        UINT32  Rsvd;
        UINT32  INTCTL;
```

```
        UINT32  INTSTS;
        UINT8   Rsvd1[8];
        UINT32  WALCLK;
        UINT32  Rsvd2;
        UINT32  SSYNC;
        UINT32  Rsvd3;
        UINT32  CORBLBASE;
        UINT32  CORBUBASE;
        UINT16  CORBWP;
        UINT16  CORBRP;
        UINT8   CORBCTL;
        UINT8   CORBSTS;
        UINT8   CORBSIZE;
        UINT8   Rsvd4;
        UINT32  RIRBLBASE;
        UINT32  RIRBUBASE;
        UINT16  RIRBWP;
        UINT16  RINTCNT;
        UINT8   RIRBCTL;
        UINT8   RIRBSTS;
        UINT8   RIRBSIZE;
        UINT8   Rsvd5;
        UINT32  ICOI;
        UINT32  ICII;
        UINT16  ICIS;
        UINT8   Rsvd6[6];
        UINT32  DPIBLBASE;
        UINT32  DPIBUBASE;
        UINT8   Rsvd7[8];

        HDA_CONTROLLER_STREAM_DESCRIPTOR* ISS;
        HDA_CONTROLLER_STREAM_DESCRIPTOR* OSS;
        HDA_CONTROLLER_STREAM_DESCRIPTOR* BSS;

} HDA_CONTROLLER_REGISTER_SET;

/*
 * This enumeration contains all verbs
 * that can be executed on widgets present on
 * a HDA compatible codec.
 * Details can be found at the page 218 of the
 * HDA Specification Rev 1.0a
 */
typedef enum {
        HDA_VRB_GET_PARAMETER = 0xF00,
        HDA_VRB_GET_CONNECTION_SELECT = 0xF01,
        HDA_VRB_GET_GET_CONNECTION_LIST_ENTRY = 0xF02,
```

```
HDA_VRB_GET_PROCESSING_STATE = 0xF03,

/*
 * These verbs are widget dependent.
 * We need to find a way to set the correct
 * verb id at runtime
 */
HDA_VRB_GET_COEFFICIENT_INDEX = 0xD,
HDA_VRB_GET_PROCESSING_COEFFICIENT = 0xC,
HDA_VRB_GET_AMPLIFIER_GAIN_MUTE = 0xBA0,
HDA_VRB_GET_STREAM_FORMAT = 0xA,


HDA_VRB_GET_DIGITAL_CONVERTER_1 = 0xF0D,
//HDA_VRB_GET_DIGITAL_CONVERTER_2 = 0xF0D,
//HDA_VRB_GET_DIGITAL_CONVERTER_3 = 0xF0D,
//HDA_VRB_GET_DIGITAL_CONVERTER_4 = 0xF0D,


HDA_VRB_GET_POWER_STATE = 0xF05,
HDA_VRB_GET_CHANNEL_STREAM_ID = 0xF06,
HDA_VRB_GET_SDI_SELECT = 0xF04,
HDA_VRB_GET_PIN_WIDGET_CONTROL = 0xF07,
HDA_VRB_GET_UNSOLICITED_ENABLE = 0xF08,
HDA_VRB_GET_PIN_SENSE = 0xF09,
HDA_VRB_GET_EAPD_BTL_ENABLE = 0xF0C,
HDA_VRB_GET_ALL_GPI_CONTROLS_F10 = 0xF10,
HDA_VRB_GET_ALL_GPI_CONTROLS_F11 = 0xF11,
HDA_VRB_GET_ALL_GPI_CONTROLS_F12 = 0xF12,
HDA_VRB_GET_ALL_GPI_CONTROLS_F13 = 0xF13,
HDA_VRB_GET_ALL_GPI_CONTROLS_F14 = 0xF14,
HDA_VRB_GET_ALL_GPI_CONTROLS_F15 = 0xF15,
HDA_VRB_GET_ALL_GPI_CONTROLS_F16 = 0xF16,
HDA_VRB_GET_ALL_GPI_CONTROLS_F17 = 0xF17,
HDA_VRB_GET_ALL_GPI_CONTROLS_F18 = 0xF18,
HDA_VRB_GET_ALL_GPI_CONTROLS_F19 = 0xF19,
HDA_VRB_GET_ALL_GPI_CONTROLS_F1A = 0xF1A,
HDA_VRB_GET_BEEP_GENERATION_CONTROL = 0xF0A,
HDA_VRB_GET_VOLUME_KNOB_CONTROL = 0xF0F,
HDA_VRB_GET_IMPLEMENTATION_ID_BYTE_0 = 0xF20,
//HDA_VRB_GET_IMPLEMENTATION_ID_BYTE_1 = 0xF20,
//HDA_VRB_GET_IMPLEMENTATION_ID_BYTE_2 = 0xF20,
//HDA_VRB_GET_IMPLEMENTATION_ID_BYTE_3 = 0xF20,
HDA_VRB_GET_CONFIG_DEFAULT_BYTE_0 = 0xF1C,
//HDA_VRB_GET_CONFIG_DEFAULT_BYTE_1 = 0xF1C,
//HDA_VRB_GET_CONFIG_DEFAULT_BYTE_2 = 0xF1C,
//HDA_VRB_GET_CONFIG_DEFAULT_BYTE_3 = 0xF1C,
```

```
HDA_VRB_GET_STRIPE_CONTROL = 0xF24,
HDA_VRB_GET_CONVERTER_CHANNEL_COUNT = 0xF2D,
HDA_VRB_GET_DIP_SIZE = 0xF2E,
HDA_VRB_GET_ELD_DATA = 0xF2F,
HDA_VRB_GET_DIP_INDEX = 0xF30,
HDA_VRB_GET_DIP_DATA = 0xF31,
HDA_VRB_GET_DIP_XMITCTRL = 0xF32,
HDA_VRB_GET_CONTENT_PROTECTION_CONTROL = 0xF33,
HDA_VRB_GET_ASP_CHANNEL_MAPPING = 0xF34,


HDA_VRB_SET_CONNECTION_SELECT = 0x701,

/*
 * These verbs are widget dependent.
 * We need to find a way to set the correct
 * verb id at runtime
 */
HDA_VRB_SET_COEFFICIENT_INDEX = 0x5,
HDA_VRB_SET_PROCESSING_COEFFICIENT = 0x4,
HDA_VRB_SET_AMPLIFIER_GAIN_MUTE = 0x3B0,
HDA_VRB_SET_STREAM_FORMAT = 0x2,

HDA_VRB_SET_DIGITAL_CONVERTER_1 = 0x70D,
HDA_VRB_SET_DIGITAL_CONVERTER_2 = 0x70E,
HDA_VRB_SET_DIGITAL_CONVERTER_3 = 0x73E,
HDA_VRB_SET_DIGITAL_CONVERTER_4 = 0x73F,
HDA_VRB_SET_POWER_STATE = 0x705,
HDA_VRB_SET_CHANNEL_STREAM_ID = 0x706,
HDA_VRB_SET_SDI_SELECT = 0x704,
HDA_VRB_SET_PIN_WIDGET_CONTROL = 0x707,
HDA_VRB_SET_UNSOLICITED_ENABLE = 0x708,
HDA_VRB_SET_PIN_SENSE = 0x709,
HDA_VRB_SET_EAPD_BTL_ENABLE = 0x70C,
HDA_VRB_SET_ALL_GPI_CONTROLS_710 = 0x710,
HDA_VRB_SET_ALL_GPI_CONTROLS_711 = 0x711,
HDA_VRB_SET_ALL_GPI_CONTROLS_712 = 0x712,
HDA_VRB_SET_ALL_GPI_CONTROLS_713 = 0x713,
HDA_VRB_SET_ALL_GPI_CONTROLS_714 = 0x714,
HDA_VRB_SET_ALL_GPI_CONTROLS_715 = 0x715,
HDA_VRB_SET_ALL_GPI_CONTROLS_716 = 0x716,
HDA_VRB_SET_ALL_GPI_CONTROLS_717 = 0x717,
HDA_VRB_SET_ALL_GPI_CONTROLS_718 = 0x718,
HDA_VRB_SET_ALL_GPI_CONTROLS_719 = 0x719,
HDA_VRB_SET_ALL_GPI_CONTROLS_71A = 0x71A,
HDA_VRB_SET_BEEP_GENERATION_CONTROL = 0x70A,
HDA_VRB_SET_VOLUME_KNOB_CONTROL = 0x70F,
```

```
        HDA_VRB_SET_IMPLEMENTATION_ID_BYTE_0 = 0x720,
        HDA_VRB_SET_IMPLEMENTATION_ID_BYTE_1 = 0x721,
        HDA_VRB_SET_IMPLEMENTATION_ID_BYTE_2 = 0x722,
        HDA_VRB_SET_IMPLEMENTATION_ID_BYTE_3 = 0x723,
        HDA_VRB_SET_CONFIG_DEFAULT_BYTE_0 = 0x71C,
        HDA_VRB_SET_CONFIG_DEFAULT_BYTE_1 = 0x71D,
        HDA_VRB_SET_CONFIG_DEFAULT_BYTE_2 = 0x71E,
        HDA_VRB_SET_CONFIG_DEFAULT_BYTE_3 = 0x723,
        HDA_VRB_SET_STRIPE_CONTROL = 0x724,
        HDA_VRB_SET_CONVERTER_CHANNEL_COUNT = 0x72D,
        HDA_VRB_SET_DIP_INDEX = 0x730,
        HDA_VRB_SET_DIP_DATA = 0x731,
        HDA_VRB_SET_DIP_XMITCTRL = 0x732,
        HDA_VRB_SET_CONTENT_PROTECTION_CONTROL = 0x733,
        HDA_VRB_SET_ASP_CHANNEL_MAPPING = 0x734,


        HDA_VRB_SET_RESET = 0x7FF
} HDA_VERB;

/*
 * This macro is used when a Immediate command needs to
 * be started
 */
#define HDA_START_PROCESSING_IMMEDIATE_COMMAND 0x01

/*
 * This macro extracts the total node count from a
 * SubordinateNodeCount property
 */
#define HDA_SUB_NODE_COUNT_TOTAL_NODE(Subordinate) \
        (Subordinate & 0xFF)

/*
 * This macro extracts the total node count from a
 * SubordinateNodeCount property
 */
#define HDA_SUB_NODE_COUNT_START_NODE(Subordinate) \
        ((Subordinate >> 16) & 0xFF)

/*
 * This macro extracts the node type from a
 * FuncttionGroup property
 */
#define HDA_NODE_TYPE(FunctionGroupType) \
        (FunctionGroupType & 0xFF)
```

```
/*
 * This macro is used to check if a given function group
 * can generate unsolicited responses
 */
#define HDA_UNSOLICITED_RESPONSE_CAPABLE(FunctionGroupType) \
        ((FunctionGroupType >> 8) & 0x1)

/*
 * This macro extracts the widget type from the
 * widget capabilities information of a widget
 */
#define HDA_WIDGET_TYPE(WidgetCap) \
        ((WidgetCap >> 20) & 0xF)

/*
 * This structure represents an HDA command
 * with 8 bits payload.
 * Details can be found at the page 141 of the
 * HDA Specification Rev 1.0a
 */
typedef struct {
        UINT32 VerbPayload: 8;
        UINT32 VerbIdent: 12; //the verb
        UINT32 NID: 8; //node id
        UINT32 CAd: 4; //Codec address
        UINT8 Reserved; //Reserved

} HDA_COMMAND_FIELD_8BIT_PAYLOAD;

/*
 * This structure represents an HDA command
 * with 16 bits payload.
 * Details can be found at the page 141 of the
 * HDA Specification Rev 1.0a
 */
typedef struct {
        UINT32 VerbPayload: 16;
        UINT32 VerbIdent: 4; //the verb
        UINT32 NID: 8; //node id
        UINT32 CAd: 4; //Codec address
        UINT8 Reserved; //Reserved

} HDA_COMMAND_FIELD_16BIT_PAYLOAD;


/*********************************************
 * This structure represents the stream format,
```

```
 * as defined at page 58 of the HDA Specification
 * Rev 1.0a
 * This struct is used to configure the widgets
 * Do not confuse with the struct
 * HDA_STREAM_DESCRIPTOR_FORMAT
 ********************************************/
typedef struct {
        UINT16 NumberOfChannels: 4;
        UINT16 BitsPerSample: 3;
        UINT16 Reserved: 1;
        UINT16 SampleBaseRateBaseDivisor: 3;
        UINT16 SampleBaseRateMultiple: 3;
        UINT16 SampleBaseRate: 1;
        UINT16 StreamType: 1;
} HDA_STREAM_FORMAT;

/********************************************
 * This structure represents the stream format,
 * as defined at page 48 of the HDA Specification
 * Rev 1.0a
 * This struct is used to configure the controller
 * Do not confuse with the struct HDA_STREAM_FORMAT
 ********************************************/
typedef struct {
        UINT16 NumberOfChannels: 4;
        UINT16 BitsPerSample: 3;
        UINT16 Reserved: 1;
        UINT16 SampleBaseRateBaseDivisor: 3;
        UINT16 SampleBaseRateMultiple: 3;
        UINT16 SampleBaseRate: 1;
        UINT16 Reserved2: 1;
} HDA_STREAM_DESCRIPTOR_FORMAT;

#define PCM_STRUCT_BITS_PER_SAMPLE_8 0
#define PCM_STRUCT_BITS_PER_SAMPLE_16 1
#define PCM_STRUCT_BITS_PER_SAMPLE_20 2
#define PCM_STRUCT_BITS_PER_SAMPLE_24 3
#define PCM_STRUCT_BITS_PER_SAMPLE_32 4

#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_1 0
#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_2 1
#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_3 2
#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_4 3
#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_5 4
#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_6 5
#define PCM_STRUCT_SAMPLE_BASE_DIV_BY_7 6
```

```
//192 kHz, 176.4 kHz
#define PCM_STRUCT_SAMPLE_BASE_MULTIPLE_X4 3
//144 kHz
#define PCM_STRUCT_SAMPLE_BASE_MULTIPLE_X3 2
//96 kHz, 88.2 kHz, 32 kHz
#define PCM_STRUCT_SAMPLE_BASE_MULTIPLE_X2 1
//48KHz/44.1kHz or less
#define PCM_STRUCT_SAMPLE_BASE_MULTIPLE_48_OR_LESS 0

#define PCM_STRUCT_SAMPLE_BASE_44_1KHZ 1
#define PCM_STRUCT_SAMPLE_BASE_48KHZ 0

#define PCM_STRUCT_TYPE_PCM 0
#define PCM_STRUCT_TYPE_NON_PCM 1

/*********************************************/


/*
 * This enumeration contains all possible parameters
 * to be used when executing a HDA_VRB_GET_PARAMETER (0xF00)
 * call.
 * Details can be found at the page 217 of the
 * HDA Specification Rev 1.0a
 */
typedef enum  {
        HDA_PARAM_VENDOR_ID = 0x00,
        HDA_PARAM_REVISION_ID = 0x02,
        HDA_PARAM_SUBORDINATE_NODE_COUNT = 0x04,
        HDA_PARAM_FUNCTION_GROUP_TYPE = 0x05,
        HDA_PARAM_AUDIO_FUNC_CAP = 0x08,
        HDA_PARAM_AUDIO_WIDGET_CAP = 0x09,
        HDA_PARAM_SAMPLE_SIZE_RATE_CAP = 0x0A,
        HDA_PARAM_STREAM_FORMATS = 0x0B,
        HDA_PARAM_PIN_CAP = 0x0C,
        HDA_PARAM_INPUT_AMP_CAP = 0x0D,
        HDA_PARAM_OUTPUT_AMP_CAP = 0x12,
        HDA_PARAM_CONNECTION_LIST_LENGTH = 0x0E,
        HDA_PARAM_SUPPORTED_POWER_STATES = 0x0F,
        HDA_PARAM_PROCESSING_CAP = 0x10,
        HDA_PARAM_GPIO_COUNT = 0x11,
        HDA_PARAM_VOLUME_KNOB_CAP = 0x13
} HDA_PARAMETER;

/*
 * This enumeration contains all possible parameters
 * widget types present at  the HDA Specification
```

```
 * Rev 1.0a
 */
typedef enum  {
        HDA_WIDGET_TYPE_AUDIO_OUTPUT = 0x0,
        HDA_WIDGET_TYPE_AUDIO_INPUT = 0x1,
        HDA_WIDGET_TYPE_AUDIO_MIXER = 0x2,
        HDA_WIDGET_TYPE_AUDIO_SELECTOR = 0x3,
        HDA_WIDGET_TYPE_AUDIO_PIN_CONPLEX = 0x4,
        HDA_WIDGET_TYPE_AUDIO_POWER = 0x5,
        HDA_WIDGET_TYPE_AUDIO_VOLUME_KNOB = 0x6,
        HDA_WIDGET_TYPE_AUDIO_BEEP_GENERATOR = 0x7,
        HDA_WIDGET_TYPE_AUDIO_VENDOR_DEFINED = 0xF
} HDA_WIDGET_TYPE;

/*
 * This enumeration contains the function group
 * types defined by the HDA Specification
 */
typedef enum  {
        HDA_FUNCTION_GROUP_TYPE_AUDIO = 0x1,
        HDA_FUNCTION_GROUP_TYPE_MODEM = 0x2
} HDA_FUNCTION_GROUP_TYPE;

/*
 * This enumeration contains the possible node types
 * available at a codec.
 */
typedef enum {
        HDA_NODE_ROOT = 0x0,
        HDA_NODE_FUNCTION_GROUP = 0x1,
        HDA_NODE_WIDGET = 0x2,
        HDA_UNKNOWN = 0xFF
} HDA_NODE_TYPE;


/*
 * This can be used when a verb does not need
 * any kind of payload
 */
#define HDA_VRB_EMPTY_PAYLOAD 0

/*
 * This enumeration contains the possible power states a
 * node can present based on the HDA specification.
 *
 * Details can be found at the page 151 of the
 * HDA Specification Rev 1.0a
```

```
 *
 */
typedef enum {
        HDA_POWER_D0 = 0x0,
        HDA_POWER_D1 = 0x1,
        HDA_POWER_D2 = 0x2,
        HDA_POWER_D3 = 0x3,
        HDA_POWER_D3_COLD = 0x4,
} HDA_POWER_STATE;




/*
 * This structure represents a node that can be
 * available at a HDA codec. A node can be a
 * widget or a function group. Depending on the
 * node type some information can be disregarded
 */
struct  Node{
  UINT32 NodeId;
  HDA_NODE_TYPE NodeType;
  HDA_WIDGET_TYPE WidgetType;
  UINT32 VendorId;
  UINT32 RevisionId;
  UINT32 StartingChildNodeAddess;
  UINT32 SubordinateNodeCount;
  UINT32 FunctionGroupType;
  UINT32 FuncCap;
  UINT32 WidgetCap;
  UINT32 SampleSizeRateCap;
  UINT32 StreamFormat;
  UINT32 PinCap;
  UINT32 InputAmpCap;
  UINT32 OutputAmpCap;
  UINT32 ConnectionListLength;
  UINT32 SupportedPowerStates;
  UINT32 ProcessingCap;
  UINT32 GPIOCount;
  UINT32 VolKnobCap;

  //This register information is filed
  //using the F05 verb (GetPowerState)
  UINT32 PowerState;

  //This register information is filed
  //using the 0xB verb (Amplifier Gain/Mute)
```

```
  UINT32 RightGain;
  UINT32 LeftGain;


  ////This register information is filed
  //using the F05 verb (GetStreamId)
  UINT32 ChannelStreamId;



  struct Node *ChildNodes;
};


//Page 142 of the HDA Specification
typedef struct {
        UINT32 Response;
        UINT8 Reserved: 2;
        UINT8 UnSol: 1;
        UINT8 Valid: 1;
        UINT8 UnUsed: 4;

} HDA_RESPONSE_FIELD;


typedef struct {

                UINT64 Address;
                UINT32 Length;
                UINT32 IntrptOnComp: 1;
                UINT32 Resv: 31;

} HDA_BUFFER_DESCRIPTOR_LIST_ENTRY;


typedef struct {
                HDA_BUFFER_DESCRIPTOR_LIST_ENTRY
                        BDLEntry[HDA_BUFFER_DESC_LIST_MAX_ENTRIES];
} HDA_BUFFER_DESCRIPTOR_LIST;

EFI_STATUS InitHdaLib ();


EFI_STATUS AllocateCORBBuffer(PCI_HDA_REGION* PcieDeviceConfigSpace);

EFI_STATUS AllocateRIRBBuffer(PCI_HDA_REGION* PcieDeviceConfigSpace);
```

```
EFI_STATUS FillCodecNode(PCI_HDA_REGION* PcieDeviceConfigSpace,
                                 UINT32 CurrentNodeId,
                                 HDA_NODE_TYPE NodeType,
                                 struct Node *CurrentNode);

EFI_STATUS GetNodeById(struct Node *RootNode,
                               UINT32 NodeIdToSearch,
                               struct Node* NodeDetected);

EFI_STATUS GetCodecTree (PCI_HDA_REGION* PcieDeviceConfigSpace,
               struct Node *RootNode);

EFI_STATUS ReleaseCodecTree (PCI_HDA_REGION* PcieDeviceConfigSpace,
               struct Node *RootNode);

EFI_STATUS GetCodecData8BitPayload (
             PCI_HDA_REGION* PcieDeviceConfigSpace,
                    UINT8 CodecAddress, UINT8 NodeId,
                    HDA_VERB Verb, UINT8 VerbPayload,
                    UINT32 *Response);

EFI_STATUS GetCodecData8BitPayloadCorbRirb (
  PCI_HDA_REGION* PcieDeviceConfigSpace,
  UINT8 CodecAddress, UINT8 NodeId,
  HDA_VERB Verb, UINT8 VerbPayload,
  UINT32 *Response);

EFI_STATUS GetCodecData16BitPayloadCorbRirb (
  PCI_HDA_REGION* PcieDeviceConfigSpace,
  UINT8 CodecAddress, UINT8 NodeId,
  HDA_VERB Verb, UINT16 VerbPayload,
  UINT32 *Response);


EFI_STATUS SendCommandToAllWidgets8BitPayload (
               PCI_HDA_REGION* PcieDeviceConfigSpace,
               HDA_VERB Verb, UINT8 VerbPayload);

EFI_STATUS SendCommandToAllWidgets16BitPayload (
               PCI_HDA_REGION* PcieDeviceConfigSpace,
               HDA_VERB Verb, UINT16 VerbPayload);

EFI_STATUS GetCodecData16BitPayload (
             PCI_HDA_REGION* PcieDeviceConfigSpace,
                    UINT8 CodecAddress, UINT8 NodeId,
                    HDA_VERB Verb, UINT16 VerbPayload,
                    UINT32 *Response);
```

```
UINT32 GetAmplifierGain(PCI_HDA_REGION *PcieDeviceConfigSpace,
                                UINT8 NodeId, BOOLEAN InputOutput,
                                BOOLEAN LeftRight);

EFI_STATUS DisablePcieInterrupts (
               PCI_HDA_REGION* PcieDeviceConfigSpace);

EFI_STATUS EnableMSI (PCI_HDA_REGION* PcieDeviceConfigSpace);

EFI_STATUS EnablePcieNoSnoop (PCI_HDA_REGION* PcieDeviceConfigSpace);

EFI_STATUS AddDestriptorListEntryOss0(
               PCI_HDA_REGION* PcieDeviceConfigSpace,
               HDA_CONTROLLER_REGISTER_SET* ControllerRegisterSet,
               UINT64 DataAddress,
               UINT32 DataLength,
               UINT8 BdlEntryIndex,
               UINT32 SdxLastValidIndex);

EFI_STATUS AllocateStreamsPages(
               PCI_HDA_REGION* PcieDeviceConfigSpace,
               HDA_CONTROLLER_REGISTER_SET* ControllerRegisterSet);

#endif
```