ESCOLA POLITÉCNICA
DE PERNAMBUCO

d.s.c. UPE
Departamento de Sistemas e Computação

IMPLEMENTAÇÃO DO *UNIVERSEAL HOST CONTROLLER INTERFACE* (UHCI) PARA O MEMTEST86+

Trabalho de Conclusão de Curso

Engenharia da Computação

Rômulo de Barros Correia Jales
Orientador: Prof. Sérgio Campello

Rômulo de Barros Correia Jales

IMPLEMENTAÇÃO DO *UNIVERSEAL HOST CONTROLLER INTERFACE* (UHCI) PARA O MEMTEST86+6+

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Orientador:
Dr. Sérgio Campello Oliveira

DEPARTAMENTO DE SISTEMAS E COMPUTAÇÃO
ESCOLA POLITÉCNICA DE PERNAMBUCO
UNIVERSIDADE DE PERNAMBUCO

Recife, Novembro de 2009.

Dedicatória

à minha esposa, Camila, que me incentiva e me compreende.

Agradecimentos

Agradeço aos meus pais pelo dom vida.

A minha mãe por ser a maior responsável pela grande formação educacional que recebi e recebo. Sem seu esforço nada teria acontecido.

A minha esposa Camila, que é meu anjo da guarda, estrela guia e grande amiga.

Ao professor Sérgio Campello pela paciência e coragem em me orientar.

E a todos os meus amigos.

Resumo

O Memtest86+ é um programa largamente utilizado para realizar testes de memória RAM. Entre suas funções destaca-se a monitoração e controle remotos dos testes, via interface RS232. A interface RS232, está em desuso em detrimento dos sistemas USB, estes, mais versáteis e fáceis de usar. Para o Memtest86+ continuar com funções de operação remotas é necessário implementar o suporte a USB. A implementação do suporte USB envolve três tarefas: quanto ao controle do *hardware* USB; ao *software* de interface entre a aplicação e o *hardware USB* e por último o *driver* dos dispositivos. O controle do hardware consiste na tarefa primordial, e as outras duas dela dependentes. O estudo do controle de *hardware* mostrou que há cinco especificações a seguir: UHCI, OHCI, EHCI, XHCI e WUSB; cada um deles descreve um *hardware* diferente. O UHCI e o OCHI implementam a versão 1.1 do controlador USB; a EHCI implementa a versão 2.0 e o XHCI, a 3.0. O WUSB é uma abordagem sem fio de sistemas USB. Escolheu-se implementar o UHCI por sugestão da comunidade de desenvolvimento de sistemas operacionais, pela presença do controlador no *hardware* do computador no qual a solução foi desenvolvida e por ser implementada pelo virtualizador QEMU. Este trabalho descreve a especificação UHCI e mostra uma possível solução deste controlador. O resultado,porém, não se mostrou estável. Apresentou um problema no tempo de transição não realiza a comunicação pretendida via USB. Concluiu-se por a duas possíveis causas a serem exploradas a partir desta experimentação.

Abstract

Memtest86 + is a program widely used for testing RAM. One of the features of it is to monitor and control the tests remotely via RS232 interface. However, the RS232 interface is in disuse at the expense of USB systems, which are more versatile and easy to use. So to continue the Memtest86 + functions with remote operation is necessary to implement the USB support. The implementation of USB support involves three tasks: implement the control of the USB hardware, implement the software interface between the application and the USB hardware and finally implement the device driver. Being the first major task, which without it makes sense to implement the other two. The study of the control hardware showed that five following specifications: UHCI, OHCI, EHCI, XHCI and WUSB. Each one describes a different hardware. The OCHI and UHCI implement version 1.1 of the USB controller, already implements the EHCI version 2.0 and XHCI implements the 3.0. The approach is WUSB wireless USB systems. Initially we chose to implement UHCI at the suggestion of community development of operating systems, the presence of the driver in the computer hardware for which the solution was developed and being implemented by virtualizer QEMU. This paper then describes the UHCI and shows a possible solution of this controller. However the solution was not stable. Presented a problem in the time period ending not to perform a communication via USB. We present two possible causes that can be further investigated by someone who will use the solution described here.

Sumário

1. Introdução....1
1.1 USB....2
1.1.1 Sistemas USB....3
1.1.2 Hospedeiros....3
1.2 Memtest86+ & USB....5
2. Controlador UHCI....7
2.1 Visão Geral dos Controladores de Hospedeiro USB....7
2.1.1 Requisitos Gerais dos Controladores....7
2.2 UHCI....8
2.2.1. Tipos de transmissão suportada pelo UHCI....9
2.2.2 Estruturas de Dados....10
2.2.3 Escalonador....15
2.2.3 Meio físico....16
2.2.4 Registradores....16
3. Implementação do HCD do UHCI....23
3.1 Considerações Iniciais....23
3.2 Inicialização do UHCI....23
3.2.1 Modo do Processador....23
3.2.2 Visão Geral....23
3.2.3 Procurar Endereço Base do Controlador....24
3.2.4 Reiniciar o controlador....26
3.2.5 Reiniciar Portas....27
3.2.6 Instalar Endereço Base da Lista de Quadros....28
3.2.7 Ativar o Controlador....28
3.3 Controle de Transmissão de Dados....29
4. Resultados....32
4.1 Simulação da Inicialização....32
4.2 Simulação de uma Aplicação....34
5. Conclusões e Trabalhos Futuros....37
Apêndice A – Códigos Fonte....40

Índice de Figuras

Figura 1: Memtest86+ em execução.....1
Figura 2: Pilha USB.....5
Figura 3: Exemplo de relacionamento das estruturas de dados do UHCI.....15
Figura 4: Passos para iniciar o UHCI.....24
Figura 5: Escalonador implementado.....29
Figura 6: Resultado parcial de simulação para sistema sem controlador.....33
Figura 7: Resultado parcial de simulação para sistema com controlador.....33
Figura 9: Inicialização da lista de quadros.....33
Figura 10: Controlador reiniciado e operando.....34

Índice de tabelas

Tabela 1: Velocidade de cada revisão USB..2
Tabela 2: Organização de FL..10
Tabela 3: Organização do TD..11
Tabela 4: Organização do campo controle e estado do TD..11
Tabela 5: Indicar do tipo de erro do TD..13
Tabela 6: Organização do token..13
Tabela 7: Registradores UHCI..17
Tabela 8: Bits do registrador USBCMD..18
Tabela 9: Bits do registrador USBSTS..19
Tabela 10: Bits do registrador USBINTR..20
Tabela 11: Organização do registrador PORTSRC..22
Tabela 12: Cabeçalho do espaço de configuração PCI..25
Tabela 13: Relação entre os atributos da estrutura de dados DADOS_TRANSFERENCIA_UHCI e os campos do TD..31
Tabela 14: estado do registro de controle de um TD após execução..35
Tabela 15: Campo STATUS analisado detalhadamente..35

ESCOLA POLITÉCNICA DE PERNAMBUCO

Tabela de Símbolos e Siglas

USB – *Universal Serial Bus* (Barramento Serial Universal)

RAM – *Random Access Memory* (Memória de Acesso Randômico)

RS232 - *Recommended Standard* 232 (Padrão Recomendado 232)

EIA – *Eletronics Industries Association* (Associação das Industrias de Eletrônicos)

PXE – *Preboot Execution Environment* (Ambiente de Execução de Pre-boot)

BIOS – *Basic Input/Output System* (Sistema básico de entrada/saída)

DMI – *Direct Memory Interface* (Interface de acesso Direto a Memória)

ECC – *Error Correcting Code* (Código de Correção de Erro)

SPD - *Serial Presence Detect* (Deteção Serial de Presença)

ASCII - *American Standard Code for Information Interchange* (Código Padrão Amaricando para Intercambio de Informação)

UHCI – *Universal Host Controller Interface* (Interface Universal de Controle do Hospedeiro)

EHCI – *Enhaced Host Controller Interface* (Interface de Controle do

OHCI – *Open Host Controller Interface* (Interface Aberto de Controle do Hospedeiro)

xHCI – *Extensible Host Controller Interface* (Interface Estendida de Controle do Hospedeiro)

API – *Application Programming Interface* (Interface de Programação de Aplicativos)

HCD – *Host Controller Driver* (Aplicação controladora do Controlador de Hospedeiro)

PCI - *Peripheral Component Interconnect* (Componente de Interconexão de Periféricos)

TD – *Transfer Descriptor* (Descritor de transferência)

QH – *Queue Head* (Cabeça de Fila)

BIT – *Binary Digit* (Digito Binário)

1. Introdução

O Memtest86+[1] é um programa que permite o teste de memória RAM sem a necessidade de equipamentos especiais. Pode ser executado no próprio computador no qual o módulo de memória está operando ao tempo que permite uma rápida detecção de falhas nas memórias a um baixo custo financeiro. A Figura 1 mostra a tela do Memtest86+ em execução.

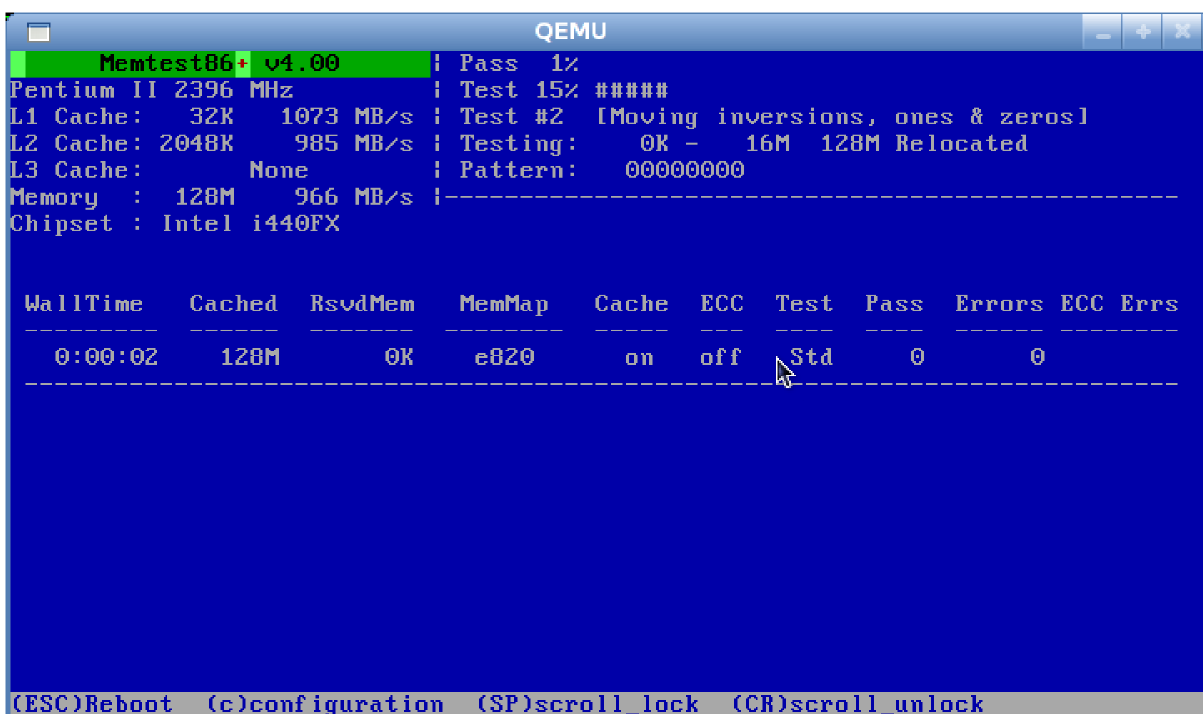

Figura 1: Memtest86+ em execução.

O Memtest86+ também oferece o controle e monitoração remota dos testes. Utilizando a interface serial RS232[2] pode-se testar vários computadores e dar diagnósticos remota e simultaneamente,condição que empresta ganho de tempo e menor custo.

A interface RS232 foi padronizada em 1969 pela EIA e conta com muitas aplicações associadas. Atualmente há uma tendência em substituir esta interface pela USB,tanto que os novos computadores poucos contém a interface RS232,especialmente se *Notebooks ou netbooks* A USB é famosa pela grande quantidade de dispositivos e aplicativos que a utilizam, pela facilidade de um usuário não-técnico em reconhecer a entrada além da grande maioria dos sistemas

operacionais a reconhecerem.

1.1 USB

USB, ou *Universal Serial Bus* (Barramento Serial Universal), é uma interface física de comunicação desenvolvida por um consórcio de empresas em 1994 com o intuito de simplificar tanto o manuseio por parte dos usuários quanto as interfaces de comunicação. Provendo uma interface extensível que permite conectar dispositivos sem a necessidade de desligar o computador além de ter baixo custo.

Desde o lançamento da USB a especificação passou por 3 revisões. A primeira em 1998 corrigiu pequenos erros da especificação original, sendo intitulada versão 1.1[3]. Em 2000 foi lançada a revisão 2.0 que, além de aumentar a velocidade de conexão trouxe, a possibilidade de fabricantes especificarem seus produtos sem a necessidade de pagar *royalties.* Já a versão 3.0, liberada em 2008, tem como grande vantagem em relação as anteriores a velocidade de conexão chegando a 4.8 Gb/s. A Tabela 1 mostra as diferenças de velocidade para cada revisão.

Tabela 1: Velocidade de cada revisão USB.

Revisão	Velocidade máxima (Mb/s)
1.1	12
2.0	480
3.0	4800

Mesmo passados mais de 10 anos do lançamento do padrão e sofrido 3 revisões o padrão físico e a pinagem dos conectores não mudaram. Inclusive as versões mais novas mantêm a compatibilidade com as anteriores. Isto é uma vantagem da interface, pois um equipamento mais antigo pode continuar operando em sistemas que trazem o padrão mais recente.

1.1.1 Sistemas USB

Sistemas USB são constituídos por três elementos básicos:

1. Dispositivos

2. Interconectores
3. Hospedeiros

Dispositivos são os equipamentos que se conectam ao hospedeiro e provê uma função específica ao mesmo ou ao sistema como um todo caso seja um *hub,* que tem a função de expandir o número de interfaces de conexão do sistema USB. Exemplos de dispositivos são *webcams, mouses,* teclados ou seja tudo aquilo que observamos no nosso cotidiano que se ligam aos nossos computadores via USB. É possível ter até 127 dispositivos presentes num sistema USB.

Já os interconectores provêm o meio físico para interligar os dispositivos e o hospedeiro. O sistema USB tem topologia estrela em camadas onde o *host* está sempre no topo, isolado, e pode ter até 7 sub-camadas. A presença de um *hub* numa determinada camada N indica que o sistema terá N+1 camadas. Como o hospedeiro aloca um *hub* virtual todo sistema USB possui no mínimo duas camadas.

Por fim, os hospedeiros, são os sistemas ao qual todo o sistema USB está conectado. Apenas um hospedeiro é permitido a sistemas USB. Como o Memtest86+ executa num computador tipicamente classificado como hospedeiro detalharemos esse elemento do sistema USB na próxima seção.

1.1.2 Hospedeiros

Todo sistema USB possui um único hospedeiro que é controlado por um dispositivo denominado *Host Controller* (HC) (controle do hospedeiro). Em geral cada hospedeiro traz apenas um único HC, caso tenha mais de um, cada controlador determina um sistema USB isoladamente.

Cada versão da especificação USB possui a descrição funcional desses controladores. Para USB 1.1 há duas especificações:

- **UHCI – *Universal Host Controller Interface*.[4] – Interface de Controle do Hospedeiro Universal**: desenvolvido pela Intel em 1996. Para um fabricante de controladores desenvolver sob essa especificação é necessário pagar o licenciamento da tecnologia. É usado apenas pela INTEL, dona do padrão, e VIA, única licitada.

- **OHCI – *Open Host Controller Interface*[5]. - Interface de Controle do Hospedeiro Aberto**: desenvolvido pela *Microsoft*, *Compaq* (atualmente *Hewlett-Packard*) e *National Semiconductor* em 1999. Como o nome sugere o padrão é aberto a todos aqueles que tem o interesse em desenvolver usando a especificação 1.1 do USB.

Para a revisão 2.0:

- **EHCI - *Enhanced Host Controller Interface*[6] - Interface de Controle do Hospedeiro Melhorado:** descreve o controlador especificado pela versão USB 2.0, lançado em 2002

E para 3.0:

- **xHCI - *Extensible Host Controller [7]* - Interface de Controle do Hospedeiro Extensivo**: o mais novo da família implementa o controlador desenvolvido para o USB 3.0 e liberado em 2009.

Existe ainda um quinto controlador para interface sem fio, denominado *wireless USB* (WUSB), que é baseado na versão 1.1 da especificação do USB.

Apesar do UHCI e OHCI descreverem o controlador de hospedeiro da especificação USB 1.1, o HCD de cada um não são compatíveis entre si. Isto significa que para cada HC é necessário uma solução específica. Por exemplo: caso um sistema operacional tenha desenvolvido uma solução para OHCI e se este for executado num computador que traz controlador UCHI o sistema não reconhecerá o controlador presente, e vice-versa. Assim, sistemas operacionais de uso geral, como *Linux* ou *Windows* trazem em seu núcleo ambas as soluções tanto para o UHCI quanto para o OHCI. Por sua vez, o EHCI, e XHCI, não tem esse problema.

Para acessar o HC é necessário implementar o *Host Controller Driver* – HCD, que é um conjunto de rotinas, ou seja *software,* que controlam o HC. Para só então as aplicações que usam a USB poderem acessar de fato os dispositivos. Esse modelo de acesso é descrito como arquitetura em camadas, ou simplesmente pilha, uma camada só acessa o seu imediato inferior ou superior. A Figura 2 mostra essa organização.

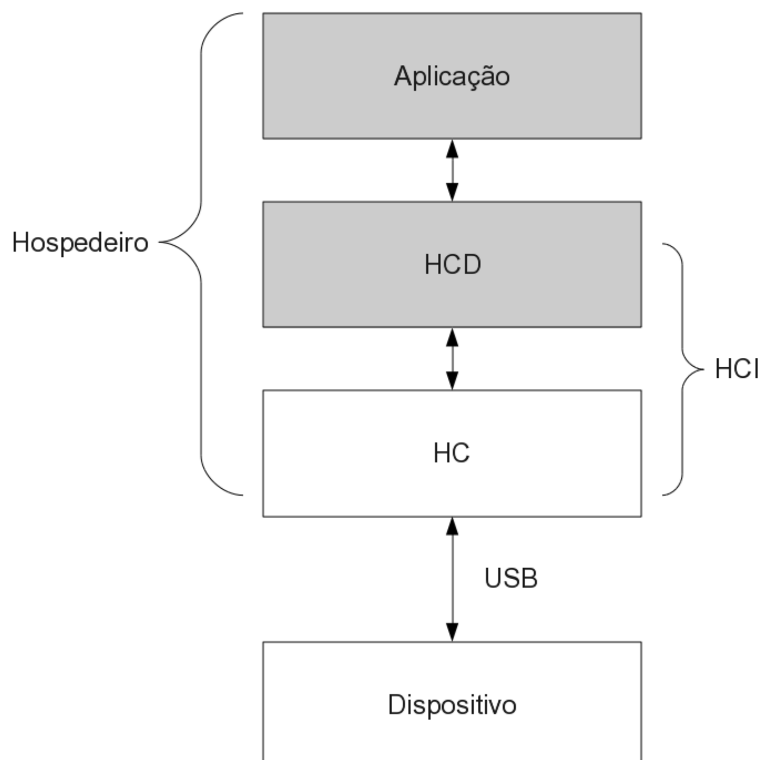

Figura 2: Pilha USB.

Os quadrados em cinza na Figura 2 indicam que é um elemento implementado em software e os em branco em hardware. A junção do HC e o HCD dá o *Host Controller Interface* (HCI), que pode ser: UHCI, OHCI, EHCI, XHCI, já descritos anteriormente. A seta na Figura 2 nomeada USB aludi aos interconectores.

1.2 Memtest86+ & USB

O objetivo inicial deste trabalho era portar o Memtest86+ para ter acesso a USB e permitir que todo o controle e monitoramento remoto, antes acessível apenas via RS232 fosse feita também pela USB.

Após estudo do Memtest86+ e conforme descrito brevemente na seção anterior as etapas necessárias para realizar o objetivo do trabalho são:

1. Implementar o HCD;
2. Implementar o controlador do dispositivo USB para comunicação;
3. Implementar rotinas de controle do Memtest86+ via USB.

Relembrando, implementar a parte *software* da pilha USB de acordo com a Figura 2.

Como a primeira etapa é desenvolver o HCD foi necessário estabelecer um ponto de partida escolhendo um dos 4 disponíveis. Foi escolhido o UHCI pelas seguintes razões:

- O computador onde estava sendo desenvolvido trazia um HC compatível com UHCI.
- O virtualizador QEMU[8] implementa, unicamente, o UHCI.
- A opinião de desenvolvedores experientes recomendaram o UHCI[9].

Consequentemente trabalhando com a especificação USB 1.1.

Este trabalho traz todas as etapas necessárias para implementar o UHCI no Memtest86+, o parte do software do HCI da Figura 2. E de maneira genérica, como implementar o UHCI para qualquer sistema. As demais etapas: como implementar o *device driver* de um dispositivo e a aplicação não serão descritas dada a complexidade em desenvolver o UHCI conforme se mostrará no decorrer do texto e devido ao tempo hábil para escrita deste documento.

2. Controlador UHCI

2.1 Visão Geral dos Controladores de Hospedeiro USB

A função primordial de um controlador de hospedeiros USB é prover uma interface física e lógica entre o dispositivo e hospedeiro, tendo como objetivos específicos realizar as seguintes tarefas[3]:

- Detectar uma remoção e um encaixe de dispositivos USB no hospedeiro;
- Gerenciar o controle de fluxo entre o dispositivo e o hospedeiro;
- Gerenciar o controle de dados entre o hospedeiro e o dispositivo;
- Gerar estatísticas de dados e estados;
- Controlar a interface física entre o hospedeiro e o dispositivo inclusive o fornecimento de energia;

2.1.1 Requisitos Gerais dos Controladores

Os controladores tem por requisito uma série de itens que devem ser tratados ou no âmbito da implementação do *hardware* ou no *software* incluindo de maneira mais objetiva as tarefas citadas na seção anterior:

1. **Controle de Estado:** controlar os estados de operação do controlador. Este pode assumir vários estados como: executando, parado, adormecido ou desativado. É função do controlador chavear entre os estados.
2. **Montagem e Desmontagem de Pacotes:** os dados oriundos da memória do sistema precisam passar por um tratamento para então serem enviados pela interface física. E quando o controlador recebe as informações precisa reconstruí-las de forma que a memória do sistema consiga armazenar a informação.
3. **Gerador de Quadros:** é responsabilidade do controlador compartilhar o tempo de uso do meio físico por meio de quadros.

4. **Processamento de Dados:** processar as requisições de transmissão de dados de e para o controlador.

5. **Tratamento de Erro de Transmissão:** notificar o controle quando erros acontecerem e que tipos de erro.

6. **Acionamento Remoto:** é função do controlador ser capaz de por o estado do barramento em modo de espera e quando vier solicitação externas pô-lo em modo ativo.

7. ***Hub:*** na especificação de sistemas USB o hospedeiro contém um *hub.* O controlador precisa conter esse *hub.* Uma função do *hub* é reconfigurar as portas quando forem detectados eventos que indiquem uma desconexão ou uma conexão.

8. **Fornecer API para as Aplicações:** a Figura 2 mostra que as aplicações não acessam diretamente o *hardware* do controlador. É implementado uma camada de *software,* o HCD, que abstrai o controlador para as aplicações. Para isto é necessário desenvolver uma API.

2.2 UHCI

O UHCI é uma implementação do controlador de hospedeiros desenvolvida pela Intel na especificação 1.1 do USB. É dividido em duas partes: uma implementada em *hardware*, o HC, e outra em software o HCD. A função primordial é mover dados entre os dispositivos e o hospedeiro.

Esse controlador foi desenvolvido visando as seguintes funcionalidades:

- **Facilidade de Uso:** o UHCI foi desenvolvido visando facilitar sua implementação em *hardware* deixando que parte dos controles sejam feitos usando o *software.*

- **Minimização de Custo:** como o *hardware* é simples ele tem baixo custo de produção possuindo um preço mais baixo em relação a outros controladores. Contudo essa simplicidade não causa impacto na performance, já que a parte complexa de tratar os dados é feita no *software*.

- **Flexibilidade:** o UHCI pode ser implementado em uma grande quantidade de

produtos desde aplicações dedicadas até computadores de uso geral.

- **Redução de Complexidade de *Hardware:*** pensado em deixar a implementação do *hardware* simples o UHCI definiu as seguintes diretivas:
  - Estruturas de dados são desenvolvidos de forma simples. Todos os ponteiros são implementados em *software* e a única operação feita em *hardware* é copiar os ponteiros de ligação.
  - Os tipo de dados que descrevem as transmissões têm a mesma forma.
  - O registrador de configuração pode ser implementado usando o espaço de registro da PCI, implicando em prevenção de conflitos de interrupção.
  - O controlador usa um escalonador para realizar o controle apropriado para o dispositivo.
- **Aumento na Performance do Sistema:** o UHCI maximiza o uso dos conectores USB reduzindo o impacto em outras interfaces; O tempo de espera de um quadro é de 1-ms. E o UHCI minimiza o tráfico com a memória do sistema. Já que só há um tipo de dado de transferência.

Podemos concluir, que o UHCI transfere a complexidade do sistema para o software. O impacto dessa abordagem é o aumento significativo do tempo de trabalho para desenvolver o HCD desse controlador.

2.2.1. Tipos de transmissão suportada pelo UHCI

O UHCI suporta os quatro tipos de transmissão entre um dispositivo e um hospedeiro, a saber:

1. **Transmissão Isócrona:** é caracterizada pela taxa de transferência entre o hospedeiro e o dispositivo. Um dispositivo típico que necessita deste tipo de transferência é um microfone pois os dados precisam chegar rapidamente e a uma taxa constante à aplicação cliente.
2. **Transmissão de Interrupção:** é caracterizado por ter apenas uma direção de envio: do dispositivo para o controlador. Uma pequena quantidade de dados é enviada para o controlador e este gera uma interrupção na aplicação cliente.

Um exemplo típico é o teclado.

3. **Transmissão de Controle:** este tipo de transmissão é usado pelo controlador para obter o estado, definir a configuração e controlar o dispositivo. Todos os dispositivos estão sujeitos a esse tipo de transmissão.

4. **Transmissão de Massa:** essa transmissão provê garantia de entrega dos dados. É usada em dispositivos que necessitam mover grandes quantidades de dados, como um *pendrive*.

2.2.2 Estruturas de Dados

O UHCI provê estruturas de dados que permitem realizar a transmissão de dados entre o hospedeiro e o dispositivo, a saber:

1. **Lista de Quadros (FL, do inglês *Frame List*);**
2. **Descritores de Transferência (TD, *Transfer Descriptor*);**
3. **Filas (QH, *Queue Head*);**

Lista de Quadros (FL):

As listas de quadros são elementos de um vetor de até 1024 posições. Cada FL aponta para o endereço de um TD ou QH a ser processado pelo controlador. Cada elemento tem um tamanho de 32 *bits* divididos em 5 partes descritos pela Tabela 2. O *hardware* do UHCI não escreve dados nessa lista, apenas lê, a escrita é feita pelo *software* do controlador.

Tabela 2: Organização de FL.

bit	31-4	3	2	1	0
Tipo	Endereço	0	0	Q	T

Os *bits* 31 a 4 contêm o endereço do elemento a ser processado pelo controlador. Os *bits* 3 e 2 são reservados. O *bit* 1 determina que tipo de estrutura de dados será processado, se é um QH ou TD, caso contenha 0 será um TD, caso 1 um QH. E o *bit* 0 determina se o valor contido no campo endereço é válido ou não. Caso o valor seja 1 indica que o campo é válido, caso 0 não é válido.

Descritores de Transferência (TD):

Os descritores são estruturas que contêm as instruções de controle da transmissão e o endereço da massa de dados a transmitir ou receber do controlador. Podendo um descritor não ter massa de dados a transmitir, *nullpacket*. O TD é uma estrutura que contém 16 *bytes* divididos em quatro campos cada um sendo de 32 *bits*: ponteiro para próximo elemento a processar, controle e estado, *token* e por fim o campo que contém o endereço da massa de dados. O descritor necessita que os dados estejam alinhados, ou seja, não podem estar segmentados. Como tem 16 *bytes* de diferença o controlador acessa cada campo adicionando ao endereço um deslocamento. Conforme a tabela abaixo:

Tabela 3: Organização do TD.

Deslocamento	**Campo**
00h	Próximo elemento
04h	Controle e Estado
08h	*token*
0Ch	Endereço da massa de dados

O formato e a função do campo contendo o ponteiro para o próximo elemento é idêntico ao FL. Desta forma, se mantém a regularidade das instruções para o controlador, implicando num projeto de *hardware* mais simples.

Já o campo controle e estado determina como o TD será tradada pelo controlador e também o estado do TD após a execução do mesmo pelo controlador. A Tabela 4 descreve a organização deste campo.

Tabela 4: Organização do campo controle e estado do TD.

bit	31, 30	29	28, 27	26	25	24	23-16	15 - 11	10 - 0
função	R	SPD	C ERR	LS	ISO	IOC	Status	R	ActLen

Os *bits,* 31, 30 e 15 à 11 marcados com R por que são reservados e não são controlados, sempre contendo o valor 0. O *bit 29* SPD (*Short Packet Detected)* contém o estado de um pacote sendo recebido. Quando este *bit* é 1 é por que um

pacote com um valor menor que o tamanho máximo de pacote foi recebido. Há duas situações que podem indicar um recebimento deste tipo. A primeira é caso um pacote realmente tenho um tamanho pequeno, a outra é quando houve um erro na transmissão.

Os *bits* 28 e 27 C ERR, contador de erros, contém o número de vezes que um TD pode falhar. De zero (0,0) à 3 (1,1). Caso um erro ocorra o controlador decrementa esse campo. Caso o campo seja zero o controlador inativa o TD. O campo LS (*low speed), bit* 26, informa ao controlador, caso esteja em 1, que deve transferir a velocidade baixa (1,5 Mbps). O próximo campo ISO no *bit 25* determina se o tipo de transmissão é isócrona, valor 1, ou não, 0. A vantagem de usar esse campo que utiliza uma estrutura de dado para encapsular os dados e o controle dos 4 tipos de transmissão existentes. Já o campo IOC *bit* 24, determina que ao fim da transmissão deva-se gerar uma interrupção.

O conjunto de *bits* do campo *status* são escritos pelo controlador para reportar o resultado de operação do TD. O *bit 16* é reservado. Os demais *bits* são descritos abaixo*:*

- **23 – A*ctive:*** quando em 1 o TD está ativo. Quando em 0 pode indicar que um erro ocorreu, sendo necessário verificar os demais *bits* desse campo. Caso não erro indica que o TD foi processado corretamente.
- **22 – *Stalled:*** indica que um erro grave ocorreu. Quando esse *bit* está marcado indica que o TD está inativo. Os *bits* 21, 20, 20 e 18 determina que tipo de erro ocorreu.
- **21 - Erro de *Buffer***: indica que há problemas nos dados a transferir ou sendo transferido. Ocorre por exemplo quando a quantidade de *bytes* transferida é maior ou menor do que a especificada pelo campo correspondente. O *bit stalled* também é marcado.
- **20 – Babble:** é um erro fatal para controlador. Quando este erro é marcado o controlador não executa outros TD. Ocorre quando o controlador recebe dados inesperados, evidenciando um problema de configuração no controle da transição.

- **19 – NAK:** ativa o *bit* quando o controlador recebe um NAK. Quando esse erro ocorre o *bit 18* também é marcado em 1 e o *bit* 23 permanece ativo.

- **18 – Erro de CRC/Timeout:** Quando com valor 1 indica duas situações: a primeira é que o TD está inconsistente e o algoritmo de verificação CRC falhou. A segunda indica que o tempo de vida do pacote expirou, e que houve um problema de comunicação entre o controlador e o dispositivo. Para identificar uma situação da outra verifica-se o valor do *bit* 23, já que para ambos os casos o *bit* 18 conterá o valor 1. Caso o *bit* 23 tenha com valor 0, indica um erro de CRC, como esse erro é um problema estrutural do pacote, estando o TD é marcado inativo.. Caso o estado o TD esteja ativo, *bit* 23 com valor 1, indica que ocorreu um *time out*.

Tabela 5: Indicar do tipo de erro do TD.

Erro	Bit 23	Bit 18
Erro de CRC	0	1
TIMEOUT	1	1

- **17 – *BitStuff:*** Indica que recebeu um sequência de mais de 6 valores 1 numa sequência.

Para finalizar o estudo do campo, os *bits* de 10 à 0 contém o total de *bytes* transferidos.

O terceiro campo do TD é o *token*. Esse campo contém toda a informação de controle do TD, controlado pela aplicação que usa USB. A Tabela 6 descreve a organização deste registro:

Tabela 6: Organização do token.

bit	31,21	20	19	18,15	14,8	7,0
função	MaxLen	R	D	EndPt	Device Adress	PID

Todos os *bits* do registrador *token* informam ao controlador como será a transição. Com exceção do *bit* 20 que é reservado e é sempre lido 0.

- ***Maximum Length*** **(MaxLen), bits 31 à 21:** Este registrador de 10 *bits* informa ao controlador quantos *bytes* serão transmitidos ou recebidos da transição. Apesar do campo poder ser preenchido com um valor de até 2047, o máximo que o controlador suporte é 1024. Valores acima causarão um erro de consistência do TD.
- ***Data Toggle*** **(D), bit 19:** tem a função de sinalizar a aplicação a ordem dos dados. Vide seção 8.6 de [3].
- ***EndPoint*** **(EndPt), *bits* 18 à 15:** este campo de 4 *bits,* estende o endereçamento de um dispositivo. Podendo ter até 16 *endpoints*
- ***Device Address bits 14 à 8*****:** identifica o endereço de dispositivo que serve como origem ou destino da transição.
- ***Packet Identification (*****PID), bits 7 à 0:** esse campo determina que transição ocorrerá, assumindo 3 tipos:
  - *setup:* para configuração do dispositivo, (69h).
  - In: para uma transição que receberá dados, (E1h).
  - out: para transições que escrevem dados no dispositivo, (2Dh).

Por fim, o último campo contém o endereço da região de memória para realizar *buffer*, no qual a massa de dados será escrita ou lida.

Filas (QH):

As filas são estruturas de dados de 16 *bytes*, que organizam os descritores dos tipos de transmissão de massa, de controle ou de interrupção, ou seja TD apontados por uma fila deve ter o valor do campo ISO em 0.

As filas contém dois ponteiros idênticos ao FL. Um para percorrer os TD da própria lista, determinando um movimento vertical e outro ponteiro para indicar a próxima lista a ser processada, determinando um movimento horizontal.

Observe que só falamos em dois campos do tipo FL, totalizando apenas 8 *bytes.* Como o campo deve conter 16 *bytes* , o espaço restante é preenchido com valores que não serão levados em consideração pelo controlador. Mas ao fazer uma busca por um elemento o controlador conta de 0h até 0Fh (15 em hexadecimal), por

que ele processa também um TD que tem 16 *bytes.* Esse detalhe é devido ao compromisso do UHCI em criar um *hardware* simples.

O último campo do TD, contém o endereço de 32 *bits* do *buffer* que contém os dados a serem transferidos, ou a região de memória no qual o controlador irá escrever os valores recebido.

Essas estruturas de dados se relacionam no escalonador. A Figura 3 mostra um exemplo. A lista de quadro aponta para o primeiro descritor, do tipo isócrono, que por sua vez aponta para outro descritor do mesmo tipo. O último descritor isócrono aponta para uma fila. A fila contém dois ponteiros, um para indicar os descritores, do tipo não-isócrono e outro para processar a próxima fila. Os movimentos vertical e horizontal ilustram como o UHCI irá processar os dados, lendo da esquerda para direita no movimento vertical e de cima para baixo no horizontal.

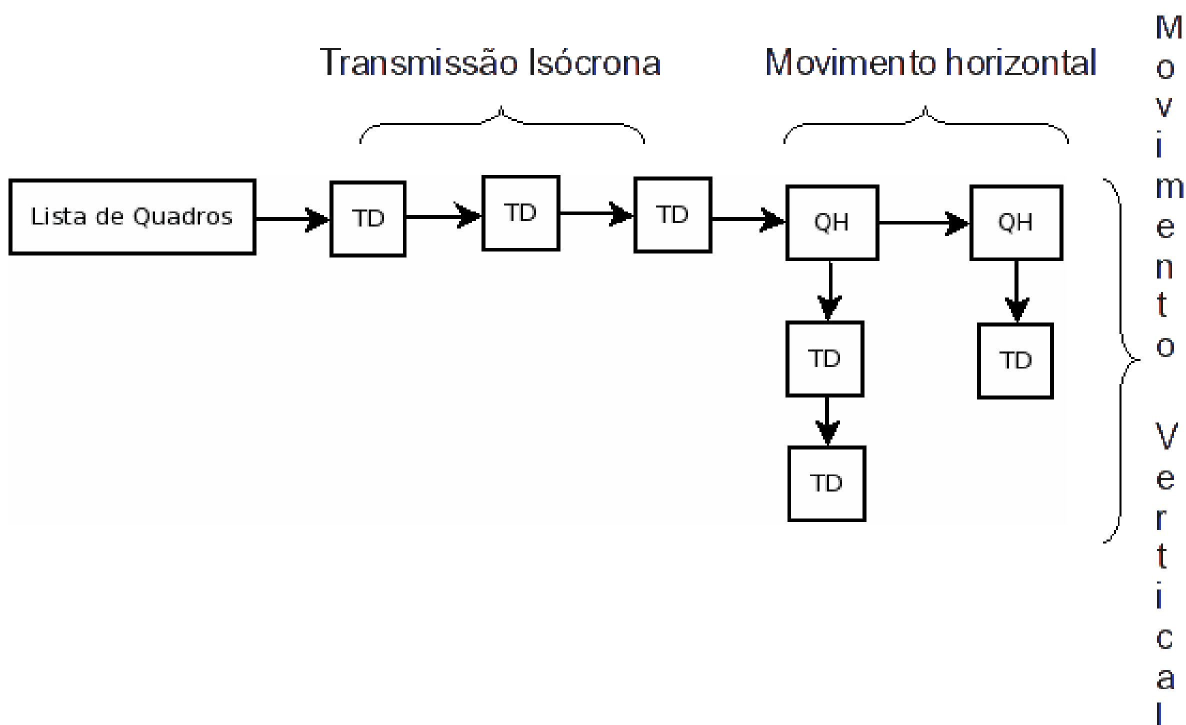

Figura 3: Exemplo de relacionamento das estruturas de dados do UHCI.

2.2.3 Escalonador

A especificação do UHCI descreve um escalonador que visa controlar o compartilhamento de tempo de acesso a interface física por cada tipo de pacote e assim garantir banda para todas as aplicações que acessam a USB dando no mínimo 10% do tempo para transmissões de controle. Esse tipo de transmissão

necessita ser atendida com um rígido esquema de garantia de entrega dos pacotes.

Parte do controle de tempo é executado via *hardware,* que informa ao escalonador se o tempo do quadro do pacote a ser transferido tem tempo suficiente para a transmissão completa. A transmissão será abortada caso o controlador detecte que não há tempo na janela de execução do quadro. Eventualmente o tempo do quadro pode estourar durante a transmissão, neste caso a transmissão é completada e o descritor transmitido é o último elemento do quadro a ser tratado. É importante citar que as informações não são perdidas na memória do sistema. Caso um descritor não venha ser transmitido ele será na próximo vez que o quadro estiver apto a acessar o meio físico, pois o controlador não retira elementos do escalonador, apenas atualiza suas entidades.

2.2.3 Meio físico

O UHCI determina que todos os controladores tenham no mínimo duas portas de interfaceamento de dispositivo e possua um *hub.* Estes provêm o meio físico do controlador, com as ligações elétricas de transmissão de informação e fornecimento de energia aos dispositivos.

2.2.4 Registradores

O UHCI possui 8 registradores de controle e estado do controlador, Tabela 7. Todos eles são acessíveis usando o endereço base (EBASE), somado a um deslocamento específico.

- **USBCMD:** responsável por controlar o UHCI. Escritas neste registrador alteram o estado de configuração do UHCI;
- **USBSTS:** responsável por indicar o estado do controlador;
- **USBINTR:** configura as interrupções do controlador;
- **FRNUM**: registrador que é um contador que indica qual quadro está sendo processado;
- **FLBASEADD:** determina o endereço base da lista de quadros. O valor contido neste registrador indica ao controlador onde encontrar os quadros

para processá-los;

- **SOF**: ajusta o tempo de um quadro USB;
- **PORTSRC1 e PORTSRC2**: contém o endereço base de controle das portas 1 e 2 do UHCI. Caso o controlador tenha mais porta para acessá-las basta incrementar 4 ao endereço da última porta mapeada. Por exemplo: suponha que o UHCI tenha 4 portas. O endereço da porta 3 é EBASE + 14h e da porta 4 EBASE + 18h.

Tabela 7: Registradores UHCI.

Registrador	Mnemônico	Descrição
EBASE + 0h	USBCMD	Registrador de comando
EBASE + 2h	USBSTS	Registrador de estado
EBASE + 4h	USBINTR	Registrador de interrupções
EBASE + 6h	FRNUM	Número do quadro
EBASE + 8h	FRBASEADD	Endereço base da lista de quadros
EBASE + Ch	SOFMOD	Modificador do iniciador de quadros
EBASE + 10h	PORTSRC1	Registrador da porta 1
EBASE + 12h	PORTSRC2	Registrador da porta 2

USBCMD – *USB Command Register:*

O USBCMD é o registrador de 16 *bits* base do UHCI. Dele é possível controlar o *hardware* do controlador. Possui 8 *bits* de controle, sendo do *bit* 15 ao 8 sem função. Enumerados pela Tabela 8.

- ***Max Packet* (MAXP), bit 7:** Tamanho máximo do pacote. Caso seja 1, o pacote tem até 64 *bytes*. Caso 0, 32 *bytes;*
- ***Configure Flag* (CF), bit 6:** função de semáforo para indicar que o controlador saiu do estado de configuração.
- ***Software Debug* (SWDBG), bit 5**: Caso o valor seja 1 indica que o controlador está em modo de depuração. Após cada transição o controlador inativa o *hardware.*
- ***Force Global Reset* (FGR), bit 4:** quando em 1, o controlador manda uma

mensagem em *broadcasting* indicando para os dispositivos saírem do modo suspenso.

- ***Enter Global Suspend Mode* (EGSM), bit 3:** quando em 1 o controlador entra em modo suspenso. Do qual nenhuma transição é executada.
- ***Global Reset* (GRESET), bit 2:** quando em 1, o controlador envia uma mensagem de *broadcasting* para todos os dispositivos reiniciarem, inclusive o próprio controlador.
- ***Host Controller Reset* (HCRESET), bit 1:** quando em 1, o controlador reinicia.
- ***Run/Stop* (RS), bit 0**: quando em 1, o controlador está em execução. Quando em 0 controlador está parado.

Tabela 8: Bits do registrador USBCMD.

bit	**Mnemônico**
15,8	reservado
7	MAXP
6	CF
5	SWDBG
4	FGR
3	EGSM
2	GRESET
1	HCRESET
0	RS

USBSTS – *USB Status Register:*

O USBSTS, deslocado de 4h do endereço base, registrador de 16 *bits* contém o estado do *hardware* e interrupções pendentes. Sendo 6 *bits* de registro, os *bits* 15 à 6 são reservados. Enumerados na Tabela 9.

- ***Host Controller Halted (*HCHalted), *bit* 5:** indica que o controlador está parado. Pode ser devido a mudança para 0 no *bit* RS do USBCMD ou resultado de um erro.
- ***Host Controller Process Error*, *bit* 4*:*** quando em 1 indica que um erro de origem de *software* ocorreu. Como um TD mal formado ou algum campo mal preenchido.
- ***Host System Error*, *bit* 3*:*** quando em 1*,* indica um erro oriundo do *hardware*.
- ***Resume Detect*, *bit* 2:** quando em 1**,** e o controlador está no modo suspenso, *bit* 3, EGSM do USBCMD está em 1, indica que um solicita que controlador fique ativo. Caso o EGSM esteja em 0, essa solicitação não é válida.
- ***USB Error Interrupt, bit 1***: quando em 1, indica que uma transição ocasionou um erro. Caso o TD da transição tenha o bit IOC ativo, o bit 0 do registrado USBSTS fica em 0.
- ***USB Interrupt (USBINT), bit 0:*** quando em 1, um TD com o *bit* IOC ativo terminou um transição. Indicando uma interrupção pendente.

Tabela 9: Bits do registrador USBSTS.

bit	**Mnemônico**
15,6	reservado
5	HCHalted
4	*Host Controller Process Error*
3	*Host System Error*
2	*Resume Detect*
1	*USB Error Interrupt*
0	USBINT

USBINTR – *USB Interrupt Enable Register:*

Esse registrador, acessado somando 8h ao endereço base, habilita e desabilita as interrupções do controlador. Possui 16 bits sendo apenas 4 para

controle, o restante (15 até 4) reservados. Descritos na Tabela 10.

- ***Short Packet Interrupt Enable, bit* 3**: habilita, quando em 1, interrupção quando detecta um pacote recebido cujo o tamanho é inferior ao número de bytes máximo. Essa interrupção é útil para indicar o fim de uma tranmissão longa.
- ***Interrupt On Complete (IOC), bit 2:*** habilita interrupção quando um TD com o *bit* IOC ativo transiciona com sucesso.
- ***Resume Interrupt Enable, bit 1:*** quando ativo, permite ao controlador informar o pedido de reinício de um dispositivo.
- ***Timeout/CRC Interrupt Enable, bit 0:*** o controlador irá sinalizar uma interrupção quando identificar um erro de *timeout* e de verificação do *CRC.*

Tabela 10: Bits do registrador USBINTR.

bit	**Mnemônico**
15,4	reservado
3	*Short Packet Interrupt Enable*
2	*Interrupt On Complete (IOC)*
1	*Resume Interrupt Enable*
0	*Timeout/CRC Interrupt Enable*

FRNUM – *Frame Number Register:*

O registrador FRNUM, estando a 6h do endereço base, é um contador de 11 *bits, q*ue se incrementa a cada quadro de tempo. Quando o *bit* RS está ativo o FRNUM se atualiza, quando RS, está inativo o contador pára, o RS torne a se ativar o FRNUM incrementa do último valor registrado.

FLBASEADD – *Frame List Base Adress:*

O FLBASEADD é um registrador de 32 *bits*, estando a 8h do endereço base. Cujos 20 *bits* mais significativos são usados para armazenar o endereço do primeiro

elemento da lista de quadros (FL). Os 12 *bits* restantes são lido como zero.

O FLBASEADD em junção com o FRNUM determinam o endereço da lista de quadro a ser processada pelo controlador. Ou seja, esses registradores são responsáveis por indicar a posição de memória dos dados para o controlador.

SOF – Start Of Frame:

O SOF é um registrador de 8 *bits,* sendo 7 de controle, para determinar o tempo de execução de cada quadro. O ajuste desse registrador influencia no tempo de incremento de FRNUM.

PORTSRC – *Port Source:*

Os PORTSRC são dois registradores, PORTSRC1 e PORTSRC2, de formato e função idênticas, que estão a 10h e 12h, respectivamente, do endereço base. E tem a função de controlar e indicar o estado das portas de conexão com dispositivos USB do UHCI. Ambos, tem 16 *bits*, sendo 6 reservados: *bits* 15 à 13, 11, 10 e 7. A Tabela 11 exibe a organização do *bits* do registrador.

Caso o controlador possua mais de 2 portas basta adicionar mais 2 ao valor da última porta mapeada.

- **Suspend, bit 12:** determina se uma porta está no modo suspenso.
- ***Port Reset, bt 9:*** quando em 1, reinicia a porta, enviando também um sinal de *reset* ao controlador.
- ***Low Speed Device Attached, bit 8:*** quando em 1, indica que um dispositivo de baixa velocidade está conectada na porta.
- ***Resume Detect, bit 6:*** quando em 1 exibe uma solicitação de reinicio da porta.
- ***Line Status, bits 5 e 4:*** indica os níveis lógicos dos pinos D+ e D- da porta. É usado para detecção de erro e diagnóstico.
- ***Port Enable/Disable Change, bit 3:*** caso em 1 indica uma mudança ou para estado habilitado ou para desabilitado.
- ***Port Enable/Disable, bit 2:*** habilita, quando em 1, e desabilita quando em 0 a porta.

- ***Connect Status Change , bit 1:*** caso em 1 indica uma mudança no estado da conexão da porta.
- ***Current Connect Status* , bit 0**: caso em 1 indica que há um dispositivo na conectado na porta. Quando em 0, não há.

Tabela 11: Organização do registrador PORTSRC.

Bit	descrição
12	Suspend
9	*Port Reset*
8	*Low Speed Device Attached*
6	*Resume Detect*
5 e 4	*Line Status*
3	*Port Enable/Disable Change*
2	*Port Enable/Disable*
1	*Connect Status Change*
0	*Current Connect Status*

3. Implementação do HCD do UHCI

3.1 Considerações Iniciais

A implementação do HCD do UHCI envolve o desenvolvimento de duas grandes tarefas:

1. Controlador do *hardware*
2. Escalonador de pacotes.

A primeira envolve a configuração de registradores e acesso a PCI para obter informações do controlador. Já a segunda trata do desenvolvimento das estruturas de dados e criação dos mecanismos que controlam o fluxo dos dados e o *hardware.*

3.2 Inicialização do UHCI

3.2.1 Modo do Processador

O USB lida com instruções de 32 *bits*, isto implica que é necessário usar sistemas que tenham palavras de 32 *bits* como os que estão baseados em processadores da arquitetura x86, para o qual o Memtest86+ foi projetado para executar os testes de memória.

Processadores 8086 de 32 *bit* ao iniciar, ou sair de uma operação de *reset,* está em modo real (RMODE) que lê apenas palavras de 16 *bits,* mesmo sendo de 32 *bits.* Isso ocorre para manter a compatibilidade com sistemas antigos e legados, já que essa família tem o compromisso de suportar modelos mais antigos[10]. Assim, para poder usar o USB é necessário que o processador leia valores de 32 *bits,* estando no modo protegido (PMODE). Essa operação já está incluída nas rotinas de *BOOT* do Memtest86+.

3.2.2 Visão Geral

Após o *BOOT*, é necessário iniciar o controlador UHCI. Esta tarefa tem por objetivo configurar o controlador para um estado conhecido e passível de controle.

Para tanto é necessário primeiramente localizar o endereço base do controlador, então reiniciar o controlador, posteriormente reiniciar as portas encontradas, endereçar a lista de quadros e por fim ativar o controlador. A Figura 4 mostra esquematicamente essas etapas.

A primeira etapa é necessária para localizar fisicamente o controlador UHCI. A segunda etapa configura o estado do controlador para um ponto inicial conhecido, pois após o *BOOT* o estado do controlador é desconhecido. A terceira etapa tem que reiniciar e ativar as portas USB. A quarta etapa configura o endereço da região de memória reservada para as estruturas de dados que encapsulam os dados que transitaram entrem o dispositivo e o hospedeiro. E por fim, põe o estado do controlador em ativo.

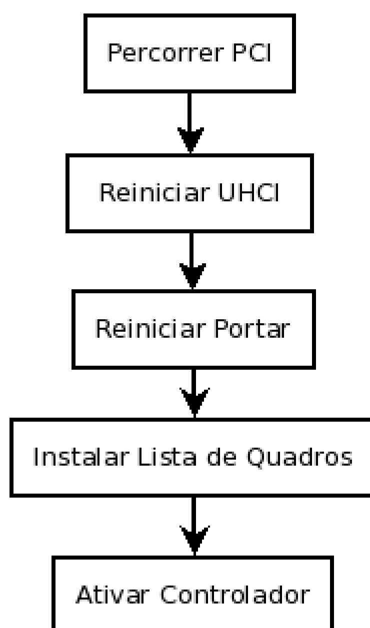

Figura 4: Passos para iniciar o UHCI.

3.2.3 Procurar Endereço Base do Controlador

O controlador UHCI é um dispositivo PCI e a forma de acesso de um dispositivo PCI é pelo seu endereço base, que é um valor que define a região de memória do qual se acessa as funções do controlador. Para obter esse valor é necessário percorrer o espaço de configuração da PCI, detalhado na Tabela 12, que

é preenchido pela BIOS durante o *BOOT[11]*. O Memtest86+ já provê uma rotina de acesso ao espaço de configuração da PCI, a função pci_conf_read. Essa função tem os seguintes parâmetros de entrada: *unsigned bus*, *unsigned dev*, *unsigned fn*, *unsigned reg* e *unsigned len* que especificam o endereço do valor que será retornado pelo parâmetro *unsigned value.*

Tabela 12: Cabeçalho do espaço de configuração PCI.

Registradores	**Bits 32 - 24**	**Bits 23-16**	**Bit 15-8**	**Bits 7-0**
00	Identificador de dispositivo		Identificador de fabricante	
04	Estado		Comando	
08	Código de Classe	Subclasse	Interface de programa	Identificador da revisão
0C	BIST	Tipo de cabeçalho	Tempo de latência	Tamanho da memória *cache*
10	Endereço base 0			
14	Endereço base 1			
18	Endereço base 2			
1C	Endereço base 3			
20	Endereço base 4			
24	Endereço base 5			
28	Ponteiro CIS			
2C	Identificador de subsistema		Identificador do fabricante do subsistema	
30	Endereço da ROM expandida			
34	Reservado			Ponteiro de capacidades
38	Reservado			
3C	Latência Máxima	Minimo garantido	Pino de interrupção	Linha de interrupção

O PCI pode conter até 256 cabeçalhos do espaço de configuração, onde cada pode ter até 32 dispositivos desempenhando 8 funções distintas[11]. Esses parâmetros são passados a função pci_conf_read pelos argumentos *unsigned bus*, *unsigned dev*, *unsigned fn*, respectivamente. O parâmetro *unsigned reg* determina

qual a linha do cabeçalho de configuração da PCI se deseja ler, podendo ser de 0h (linha 1) até 3Ch (linha 16) e o último parâmetro de entrada especifica a quantidade de *bytes* necessários para armazenar o valor lido.

A primeira etapa para obter o endereço base é identificar o controlador UHCI. Esta operação é feita lendo o registrador 8h de um dado cabeçalho PCI. Este campo contém código da classe e esta no espaço limitado ente os *bits* 31-24. Para dispositivos controladores de interface seriais o valor neste campo deve ser: 0Ch[11]. A segunda faixa de *bits* ente o 23-16 identifica a sub-classe, o valor que identifica USB é 3h. O campo seguinte entre 15-8 corresponde a identificação do controlador. Para UHCI o valor é 0h, para EHCI 20h e para OHCI 1h[4],[6],[5]. O último campo corresponde a revisão do dispositivo, que pode ser despresado. Logo, o valor de leitura do registrador 8h de um cabeçalho que corresponde a um dispositivo UHCI deve ser C0300h.

Identificado o controlador obtém-se o endereço base (EBASE) do controlador. Segundo a especificação do UHCI este endereço é encontrado no registrador 0x20, denominado endereço base 4. Com esse valor é possível iniciar a configuração do controlador, pois a partir dele é possível acessar as funções do controlador.

3.2.4 Reiniciar o controlador

A especificação do UHCI determina que para reiniciar o controlador é necessário fazer os seguintes passos:

1. Desabilitar as interrupções, escrevendo zero em todo o registrador USBINTR.
2. Ativar o *bit* 1 *(Host Controller Reset* - HCRESET*)* do registrador USBCMD.
3. Esperar *bit* 6 (*Configure Flag* - CF) do registrador USBCMD ter valor igual a 1.

O Memtest86+ provê as seguinte rotinas que permitem a leitura e escrita de endereços de memória:

- **outl(valor, endereço):** escreve um valor de 32 bits no endereço;
- **outb(valor, endereço):** escreve um *byte* no endereço;
- **outw(valor, endereço):** escreve uma palavra[1] no endereço;

1 A terminologia palavra designa um valor com 16 *bits*.

- **inl(endereço):** lê um valor de 32 bits no endereço;
- **inb(endereço):** lê um *byte* no endereço;
- **inw(endereço):** lê uma palavra no endereço;

como o registrador USBCMD tem 16 *bits* para escrever no HCRESET deve usar a função parametrizada outw(0x1,USBCMD) e ler o conteúdo do endereço CF usa-se inw(USBCMD).

A necessidade de desabilitar as interrupções é prevenir que algum evento modifique o estado do controlador sem a anuência do sistema de controle.

O estado do controlador após o reinicio é parado. Com os registradores contendo o valor padrão. O FRNUM terá zero. As portas estarão desabilitadas, as interrupções desabilitadas e o registrador de estado em zero. Inclusive as portas são reconfiguradas. O único registrador sem valor conhecido é FLBASEADD, que não tem valor padrão predefinido.

3.2.5 Reiniciar Portas

Para reiniciar as portas o UHCI determina os seguintes passos:

1. Escrever 1 no *bit* 9 (*Port Reset*) de todas as portas mapeadas.
2. Ativar a porta escrevendo no bit 2 (*Port Enable/Disable)* do endereço da porta.

Escrever 1 no *bit* 2 da porta pretendida não é suficiente para ativá-la, o controlador só a ativará se ele detectar a presença física de um dispositivo conectado a essa porta. É possível fazer uma verificação lendo o *bit* 0 (Current *Connect Status).*

3.2.6 Instalar Endereço Base da Lista de Quadros

A lista de quadros é um vetor que armazena o endereço das estruturas de dados a serem processadas pelo controlador. O controlador determina qual quadro a ser processador somando o endereço contido no registrador FLBASEADD com o valor do registrador FRNUM, que é um contador incrementado a cada intervalo de tempo predefinido. Assim o controlador sabe qual elemento a processar acessando o endereço de memória determinado por FLBASEADD + FRNUM.

Segundo a descrição do registrador FLBASEADD, que tem 32*bits* de tamanho, somente os 20 *bits* mais significativos são usados no processo de escrita. Os outros 12 *bits* menos significativos são sempre escritos com 0. Isto significa que ao configurar o endereço base da lista de quadros é necessário ter o cuidado em especificar um valor cujo os 12 *bits* menos significativos sejam zero.

A lista de quadros é um vetor com até 1024 posições, já que FRNUM é um registrador de 16 *bits* cujo os 11 menos significativos variam, podendo assumir valores de 0 até $2^{10} – 1$ (1023). O fato de ser um vetor implica que a região de memória ocupada pela lista é continua. Ou seja, além de assegurar ao valor de memória em FLBASEADD atenda os requisitos os próximos 1024 valores deverão ser reservados para a lista de quadro.

Para solucionar os problemas inciamos um ponteiro, que nada mais é que uma variável que aponta para um endereço de memória, com o valor que atenda aos requisitos. Depois percorremos 1024 posições seguintes indexando a um vetor com o mesmo tamanho da lista de quadros. Assim, asseguramos que o registrador FLBASEADD aponta para uma região que será acessada apenas pelo controlador.

3.2.7 Ativar o Controlador

Para ativar o controlador é necessário determinar também o tamanho máximo dos pacotes a trafegarem. Especificamos 64 *bytes* escrevendo 1 no bit 7 (*Max Packet* – MaxP) do registrador USBCMD. Pois assim garantimos maior *throughput* do sistema.

Por fim, para por o controlador em modo ativo escreve-se 1 no *bit* 0 (*Run/Stop* - RS) do registrador USBCMD.

3.3 Controle de Transmissão de Dados

Após inicializar, o UHCI está apto a transmitir informação. É necessário,porém, controlar os dados e as estruturas utilizadas, a entidade responsável por isso é o escalonador.

Implementamos um escalonador simples. Com apenas duas filas. Uma para transmissão de controle e outra de transmissão de dados. A Figura 5 esquematiza o

escalonador implementado.

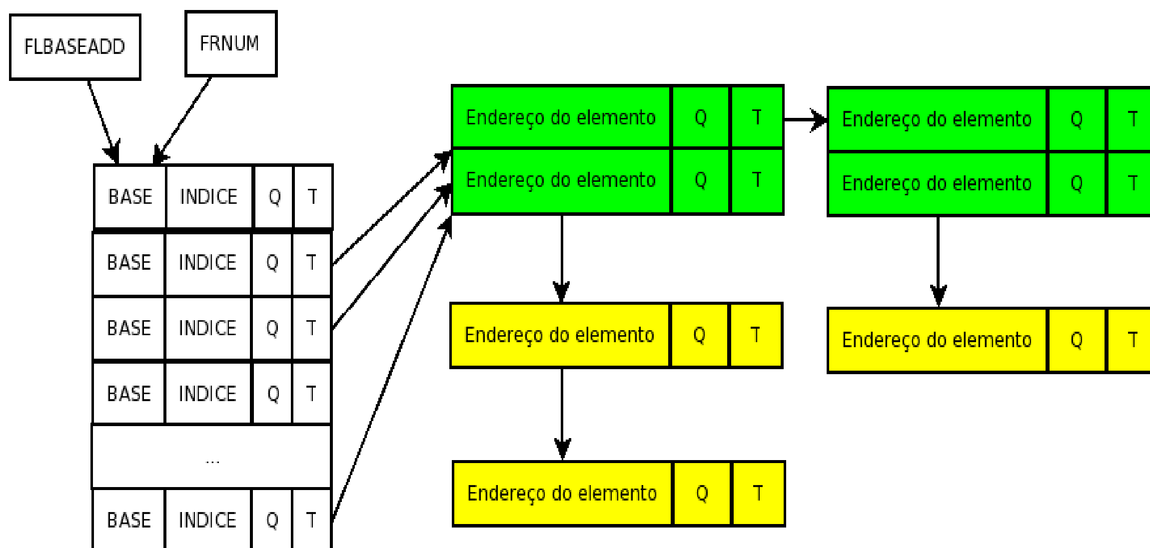

Figura 5: Escalonador implementado

O escalonador opera da seguinte forma:

1. O valor de FLBASEADD e o FRNUM indicam qual a lista a ser processada. Ou seja o índice do vetor da lista de quadros. Sempre apontam para a fila de transmissão de controle. No caso o valor do campo Q é 1, para indicar ao controlador que o próximo elemento é uma fila e o valor de T é 0, pois o endereço composto por BASE e INDICE é válido.

2. O Elemento em verde, é uma fila. Da esquerda para direita, o primeiro representa a fila de controle. O elemento contém dois itens, o primeiro indica o endereço do próximo elemento a ser processado. Caso o campo Q seja 1 indica que o próximo elemento é uma fila, caso 0 é um descritor. No nosso escalonador o campo é 1. Esse primeiro campo representa o movimento horizontal do escalonador. O escalonador só processa o endereço do campo em duas situações: a primeira é caso o próximo item da fila esteja marcado como terminativo ou quando todos os TD da fila forem processados.

3. O segundo elemento da fila contém o endereço do descritor que irá controlar a transmissão. Caso o campo Q seja 1, há um problema de consistência na fila e a operação é abortada, caso seja 0, o campo está preenchido corretamente. Como o escalonador organiza as requisições da aplicação

eventualmente o campo T pode ser marcado. Caso esse campo esteja em 0 significa que há descritores a ser tratados. O controlador então processa o endereço especificado por esse campo. Iniciando o movimento vertical.

4. O controlador enquanto não encontrar um TD terminativo processa o movimento vertical. Caso ache um terminativo, processa realiza o movimento horizontal.

Observe que a fila de controle é sempre executada antes da fila de transmissão de informação. Assim, nosso escalonador assegura que os sinais de controle sempre terão prioridade sobre os demais. Assegurando mais de 10% do tempo de acesso ao barramento para pacotes do tipo de controle.

É função do controlador gerar o TD e o QH coerentemente. Para isso implementamos funções que interpretam as requisições da aplicação. A aplicação, por sua vez, deve informar ao controlador que tipo de transmissão deseja realizar. Assim, o controlador saberá em qual fila e como estruturar as requisições do cliente. Logo, para cada fila foi implementada uma função de construção e configuração associada. Abaixo a assinatura da função para criar um TD para fila de controle.

```
void criarTdControle(struct DADOS_TRANSFERENCIA_UHCI dados)
```

A estrutura de dados para transferência é mostrada abaixo. A aplicação, a camada mais alta da Figura 2, é a responsável por preencher os valores dessa estrutura.

```
struct DADOS_TRANSFERENCIA_UHCI{
dword buffer; /*inerente ao que vai transferir*/
dword quantidade_bytes;
dword velocidade;
dword identificador_processo;
dword controle;
dword endpoint;
dword endereco_dispositivo;
};
```

A Tabela 13 mostra a relação entre os atributos da estrutura de dados e os campos do TD.

Tabela 13: Relação entre os atributos da estrutura de dados DADOS_TRANSFERENCIA_UHCI e os campos do TD.

<table>
<tr><th>Atributos da estrutura</th><th colspan="2">Campos do TD</th></tr>
<tr><td>quantidade_bytes</td><td rowspan="5">token</td><td>MaxLen</td></tr>
<tr><td>identificador_processo</td><td>PID</td></tr>
<tr><td>controle</td><td>DataToggle</td></tr>
<tr><td>endpoint</td><td>EndPt</td></tr>
<tr><td>endereco_dispositivo</td><td>Device Adress</td></tr>
<tr><td>velocidade</td><td>Controle e estado</td><td>LS</td></tr>
<tr><td>buffer</td><td colspan="2">buffer</td></tr>
</table>

Logo, a função critaTDControle lê o parâmetro de entrada dados e segundo as relações da Tabela 13 cria o TD com os valores contidos na estrutura e indexa no escalonador.

4. Resultados

Para simular os resultados utilizamos o QEMU[8], que é um virtualizador que implementa o *hardware* do UHCI.

O programa em execução exibe ao usuário as seguintes mensagens:

- Se existe um controlador UHCI presente;
- O endereço base do UHCI, caso presente;
- Se o controlador foi iniciado com sucesso;
- O estado das portas;
- E o status de execução do escalonador.

O QEMU permite que o usuário controle a presença ou não de um controlador USB. Caso o operador deseje a presença da USB usa-se a sinalização: "-usb". O QEMU inciará o controlador UHCI virtual. É possível especificar os tipo de dispositivos presentes, usando a sinalização "-usbdevice" dispositivo.

4.1 Simulação da Inicialização

Como primeiro teste da solução, simula-se o comportamento da aplicação quanto a presença do controlador. A Figura 6 exibe o resultado, caso não exista controlador.

Neste caso não é localizado no espaço de configuração da PCI um controlador com o valor de classe, sub-classe e interface de programa com o valor 0x000c0300 ou seja, um UHCI.

Já a Figura 7 exibe o resultado haja um controlador disponível, de acordo com a tela do programa foi localizado um controlador no endereço 0000c020h de onde todas as operações de controle do *hardware* do UHCI será baseado. Em seguinte o programa reinicia o *hardware*.

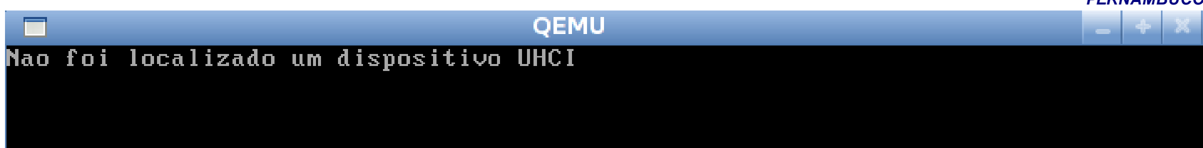

Figura 6: Resultado parcial de simulação para sistema sem controlador.

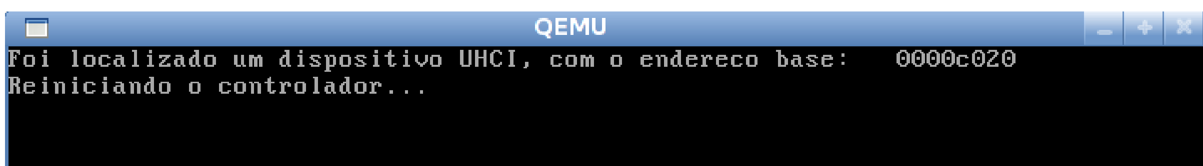

Figura 7: Resultado parcial de simulação para sistema com controlador.

Após reiniciado o controlador é necessário verificar quais portas estão conectadas. A figura exibe o resultado da verificação. Observe que sempre existirá duas portas no mínimo presentes.

Mesmo que ambas as portas estejam desconectadas isso não representa um erro ao sistema. O USB é capaz de lidar com conexões do tipo *plug'n'play,* que permite a execução instantânea do dispositivo associado sem a necessidade de reiniciar o computador.

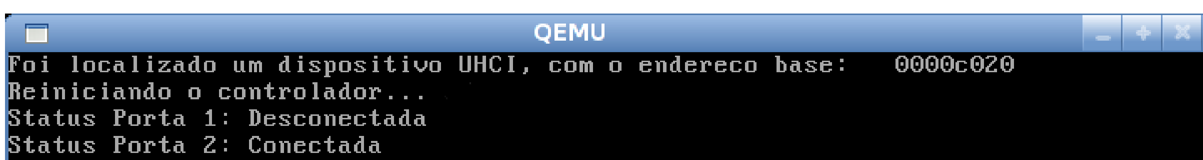

Figura 8: Resultado da verificação do estado de conexão das portas.

O próximo passo é inicializar a lista de quadros, configurando um endereço a FLBASEADD e zerar o contador de quadros o registrador FRNUM. A Figura 9 exibe o resultado desta etapa.

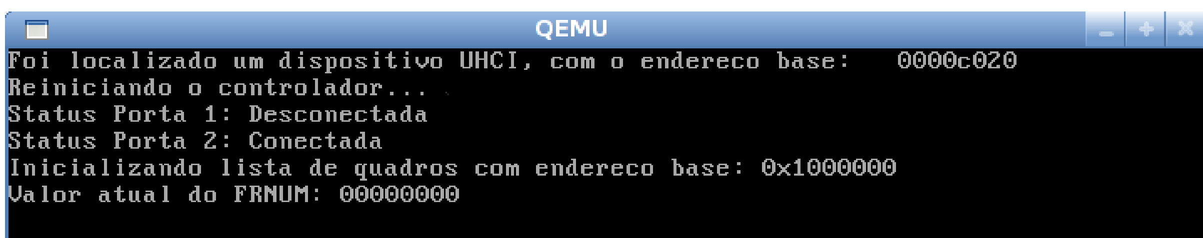

Figura 9: Inicialização da lista de quadros.

Por fim, o controlador é inicializado. Observe que a linha 2 da Tabela 10 apresenta o *status* final da operação de inicialização e o valor de FRNUM já incrementado. Isto indica que o controlador já está em operação.

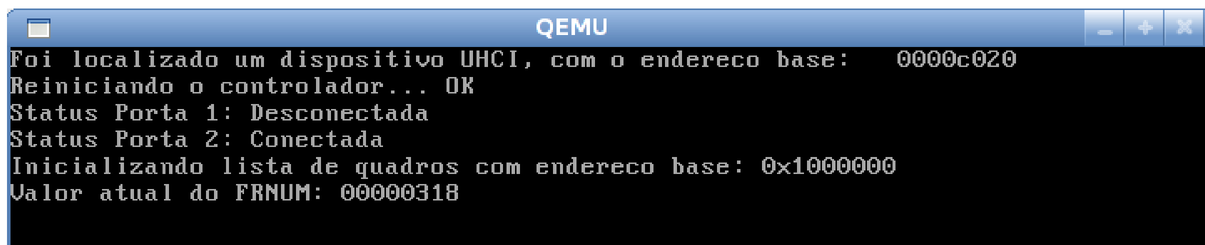

Figura 10: Controlador reiniciado e operando.

4.2 Simulação de uma Aplicação

Para uma aplicação usar o USB, após a inicialização do controlador é necessário seguir o seguinte algorítmo:

1. Obter a descrição do dispositivo;
2. Obter a descrição de configuração do dispositivo;
3. Configurar um endereço para o dispositivo no hospedeiro;
4. Configurar o dispositivo para uso da aplicação.

Segundo [3] para obter a descrição de um dispositivo é necessário usar uma fila de controle e utilizar a seguinte estrutura de dados:

```
struct REQUISITOR{
byte   bmRequestType;
byte   bRequest;
word   wValue;
word   wIndex;
word   wLength;
};
```

Essa estrutura para o contexto do UHCI representa a massa de dados a

transferir. O dispositivo interpreta essa informação e transmite ao controlador as informações do dispositivo.

Para a nossa implementação do UHCI. Devemos fazer:

1. Obter os dados de requisição;
2. Construir um TD para essa requisição;
3. Associar o TD a fila correspondente;
4. Verificar *status* do TD;
5. Processar próxima requisição do cliente.

Em análise depurativa podemos verificar que: o TD é construído com sucesso, enfileirado corretamente e consumido pelo controlador entretanto a análise do status do pacote após operação pelo controlador indica que houve um erro de *timeout, p*ois o valor do campo controle/estado do TD é 0x840000. A Tabela 14 exibe a leitura desse campo.

Tabela 14: estado do registro de controle de um TD após execução.

Campo	Valor lido
SPD	0
C ERR	0
LS	0
ISO	0
IOC	0
STATUS	84
R	0
ACTLEN	0

Analisando detalhadamente o campo STATUS, Tabela 15, evidenciamos que o campo ACTIVE e CRC/TIMEOUT ERRO estão ativos. Isto significa que houve um erro entre o dispositivo e *hardware* do controlador.

A análise primária do erro se mostrou infrutífera, já que não conseguimos contornar o erro satisfatoriamente. No entanto elaboramos 2 hipóteses para o erro e suas possíveis soluções:

Tabela 15: Campo STATUS analisado detalhadamente.

Campo	Valor
Status	84h
Active	1
Stalled	0
Data erro buffer	0
Babbled	0
NAK	0
CRC/TIMEOUT	1
Bitstuff	0
Reservado	0

1. O compilador não reserva a faixa de endereçamento: como os endereços, lista de quadros e das filas são definidos estaticamente, imaginamos que alguma variável na hora de compilação receba o mesmo endereço da lista de quadros ou das filas. Ocasionando inconsistência da informação. A solução adotada é analisar cuidadosamente o mapa de memória gerado e verificar variáveis com endereços sobrepostos. Outra solução é criar um mapa de memória estático para todo o programa e não apenas para algumas variáveis.
2. Má implementação do controlador: é possível que tenhamos implementado alguma estrutura de dados ou controle de forma equivocada. O documento de especificação do UHCI mesmo sendo bem escrito exige uma certa maturidade para o entendimento completo do qual fomos adquirindo ao longo do desenvolvimento do trabalho. É possível que algo ainda esteja errado. Logo a solução, é revisar toda a implementação atrás de alguma falha.

Uma terceira hipótese foi levantada: o UHCI virtual do QEMU está com problemas. Entretanto testando outras soluções que usam USB verificou-se que não há o erro detectado em nossa aplicação utilizando o QEMU. E testando num ambiente real, o erro persistia.

5. Conclusões e Trabalhos Futuros

A implementação de um suporte a USB para um programa como o Memtest86+, não executável sob um sistema operacional, envolve basicamente três tarefas: implementar o controlador do *hardware*, seguindo-se, os *drivers* dos dispositivos e por fim, a interface de aplicação. A primeira tarefa é a mais importante, uma vez que as outra duas camadas dependem do bom funcionamento do controle. Entre os 5 tipos de controladores USB escolheu-se o UHCI para esta implementação, por sugestão da comunidade de desenvolvimento de sistemas operacionais, pela presença do controlador no computador no qual se desenvolveu este trabalho, ainda porque o QEMU implementa esse dispositivo virtualmente.

O UHCI é um controlador cuja complexidade está no *software.* Resta aos desenvolvedores do *hardware* foco exclusivo no desempenho do controlador. Desenvolver o *driver* de controle para um dispositivo UHCI requer habilidade e experiência prévia, pois será necessário lidar com várias estruturas de dados; manipulação direta com *bits* e gerenciamento de memória.

A implementação aqui descrita foi feita de forma mais genérica possível, apesar de dirigida ao Memtest86+. Assim,este trabalho serve-se também como guia para implementação outras, bem como fonte de consulta da especificação e de fonte de código

A rotina de implementação realiza toda a tarefa de inicializar o *hardware,* desde o seu mapeamento no espaço de configuração da PCI, passando pela configuração do modo operacional do controlador, finalizando com a implementação de um escalonador de pacotes. Seu *driver,* atende aos requisitos descritos pela especificação USB 1.1. que restringe principalmente o tempo de acesso ao barramento em pelo meno 10% para pacotes de controle.

Apesar da implementação inicializar o controle do *hardware* com sucesso, quando simulamos uma aplicação a transição não é concluída. As hipóteses possíveis da origem desta falha, pode ser um problema de endereçamento ou da própria implementação em si.

A sugestão de trabalho futuro, já pensando em atingir a motivação original é implementar as outras camadas do sistema USB, inclusive o sistema que permitirá a comunicação de dois hospedeiros.

Por fim, espera-se que este trabalho possa ser continuado com o término do porte do USB para o Memtest86+, finalizando a motivação original e garantindo ao Memtest86+, que é um *software* livre que depende de desenvolvimento colaborativo, manter suas funcionalidades atuais, como o controle e operação remota.

Bibliografia

[1]: **Memtest86+. An Advanced Memory Diagnostic Tool** Disponível em: http://www.memtest.org/ Acesso: 04/11/2009

[2]: Electronics Industries Association, EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange,

[3]: **Universal Serial Bus Especification** Disponível em: http://www.usb.org/developers/docs/usbspec.zip Acesso: 14/12/2009

[4]: **Universal Host Controller Interface (UHCI) Design Guide** Disponível em: http://download.intel.com/technology/usb/UHCI11D.pdf Acesso: 14/12/2009

[5]: **Open Host Controller Interface Specification for USB** Disponível em: http://www.o3one.org/hwdocs/usb/hcir1_0a.pdf Acesso: 14/12/2009

[6]: **Enhanced Host Controller Interface Specification for Universal Serial Bus** Disponível em: http://www.intel.com/technology/usb/download/ehci-r10.pdf Acesso: 14/12/2009

[7]: **Extensible Host Controller Interface** Disponível em: http://www.usb.org/developers/docs/usb_30_spec_052109.zip Acesso: 14/12/2009

[8]: **QEMU. Open source processor Emulator** Disponível em: http://www.qemu.org Acesso: 03/11/2009

[9]: OSDEV.org - enumarating USB device, http://forum.osdev.org/viewtopic.php?f=1&t=17111 Acesso: 14/12/2009

[10]: STALLINGS, Willian , Arquitetura e Organização de Computadores, 5ª edição Prentice Hall

[11]: **PCI Local BusSpecification** Disponível em: http://www.ece.mtu.edu/faculty/btdavis/courses/mtu_ee3173_f04/papers/PCI_22.pdf Acesso: 14/12/2009

Apêndice A – Códigos Fonte

Arquivo UHCI.C

```
/*Inicializa o UHCI*/

#include "uhci.h"

#include "io.h"

#include "test.h"

#include "pci.h"


struct UHCI_HOST_STRUCT uhci_dev;

struct UHCI_PORT_STATUS portStatus1;

struct UHCI_PORT_STATUS portStatus2;

struct QH *fila = 0x1008000;

struct TD *primeiro_td  = 0xd60000;

word rec_desc[18]={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

static dword *framelist = ENDERECO_FRAME_LIST;


void imprimirStatus(void){

        int x= inw(uhci_dev.base_adr + FRNUM);

        hprint(5,22,x);

}

void executar_TD(){

        imprimirStatus();

        word x;

        word y;

        fila->primeiro_td = (primeiro_td);

        do {

        y = primeiro_td->control;
```

```
//          hprint(14,0,y);
//          cprint(1,0,"Td em execucao");
            x= inw(uhci_dev.base_adr + FRNUM);
            hprint(5,22,x);
            }while(y!=0x0);
            primeiro_td->link = primeiro_td->link | 0x1;
            imprimirStatus();
}

/*Função que le os dados para transferir e monta a cadeia de TDs necessária para transmissão*/
/*TD é uma estrutura de dados do UHCI*/
void criarTdControle(struct DADOS_TRANSFERENCIA_UHCI dados){
            imprimirStatus();
            dword tam_dados = dados.quantidade_bytes;
            if (dados.quantidade_bytes <= 64){
                        //TD TERMINATIVO
                        primeiro_td->link = 0x5;
            }else{
                        primeiro_td->link = 0x0;
            }
            primeiro_td->control = 0;
            if(dados.velocidade == VELOCIDADE_MAXIMA)
                        primeiro_td->control = primeiro_td->control & (~(1<<26));
            else
                        primeiro_td->control = primeiro_td->control | 0x4000000; //Poe 1 no bit 26

            primeiro_td->control = primeiro_td->control        | 0x800000;  //Poe estado ativo.
//hprint(11,0,primeiro_td->control);
            primeiro_td->token = 0;
```

```
	primeiro_td->token = primeiro_td->token | ((tam_dados-1)<<21); //Tamanho de dados a transmitir ao todo.

	primeiro_td->token = primeiro_td->token | (dados.controle<<19); //Toggle

	primeiro_td->token = primeiro_td->token | dados.identificador_processo;

	primeiro_td->buffer = &dados.buffer;

	imprimirStatus();

}


void obter_pid_vid_dispositivo(void){

	imprimirStatus();

	struct REQUISITOR od;

	struct DADOS_TRANSFERENCIA_UHCI td1,dados;

	struct TD comando,recebe;

	od.bmRequestType	= 0x80; //HOST-TO-DEVICE

	od.bRequest		= 0x0100; //GET_DESCRIPTOR

	od.wValue			= 6;

	od.wIndex			= 0 ;

	od.wLength			= 18;


	td1.buffer				= &od;

	td1.quantidade_bytes		= 8;				//DESCRITOR tem 8 bytes

	td1.velocidade			= VELOCIDADE_MAXIMA;

	td1.identificador_processo = PCKT_ID_SETUP;

	td1.controle				= 0;

	td1.endpoint				= 0;

	td1.endereco_dispositivo		= 0;


	dados.buffer					= &rec_desc;

	dados.quantidade_bytes			= 18;
```

```
        //DESCRITOR tem 18 bytes
         dados.velocidade                        = VELOCIDADE_MAXIMA;
         dados.identificador_processo   = PCKT_ID_IN;
         dados.controle                          = 1;
         dados.endpoint                          = 0;
         dados.endereco_dispositivo     = 0;

        criarTdControle(td1);
        int x;
        x =     (primeiro_td->control>>16);
        x = x & 0xFF;
        executar_TD();
        criarTdControle(dados);
        executar_TD();
        imprimirStatus();
}

void reconfigurar_portas(void){
        outw(PORTSC_RESET,uhci_dev.base_adr+USB_PORT1_OFFSET);
        outw(PORTSC_RESET,uhci_dev.base_adr+USB_PORT2_OFFSET);
}

void iniciar_qh(void){

        fila->qhlp=0x1;
        int i=0;
        for (i=0;i<1024;i++){
                framelist[i] = 0x1;
        }
```

```
        for (i=0;i<1024;i=i+2){
                *(framelist+i*4) = 0;
                *(framelist+i*4) =  (ENDERECO_FILA & 0xfffffffe) |0x2;
        }
}

void instalar_framelist(void){

        cprint(4,0,"Inicializando lista de quadros com endereco base: 0x1000000");
        iniciar_qh();
        /*instala inicio do vetor framelist no controlador*/
        outl(framelist,uhci_dev.base_adr+FLBASEADD);
        /*Determina o SOF padrão de 1ms*/
        //outb(UHCI_SOF_DEFAULT, uhci_dev.base_adr + SOF);
        outb(127, uhci_dev.base_adr + SOF);
        /*Zera numero de frames*/
        outw(0x0, uhci_dev.base_adr + FRNUM);
        cprint(5,0,"Valor atual do FRNUM:");
        int x= inw(uhci_dev.base_adr + FRNUM);
        hprint(5,22,x);
}

void get_ports_status(void){

        if((inw(uhci_dev.base_adr+USB_PORT1_OFFSET)&0x2)==0x0){
                cprint(2,0,"Status Porta 1: Desconectada");
                portStatus1.is_enable = FALSE;
        }else{
                cprint(2,0,"Status Porta 1: Conectada");
```

```
                portStatus1.is_enable = TRUE;
        }
        if((inw(uhci_dev.base_adr+USB_PORT2_OFFSET)&0x2)==0x0){
                cprint(3,0,"Status Porta 2: Desconectada");
                        portStatus2.is_enable = FALSE;
        }else{
                cprint(3,0,"Status Porta 2: Conectada");
                portStatus2.is_enable = TRUE;
        }
}

void reconfigurar_uhci(void){
        /*UHCI RESET: 0x2
        offset: 00
        */
        outw(USBCMD_HCRESET,uhci_dev.base_adr);
        outw(0,uhci_dev.base_adr+USBINTR);
}

void ativar_portas(void){
        outb(0x4,uhci_dev.base_adr+USB_PORT1_OFFSET);
        outb(0x4,uhci_dev.base_adr+USB_PORT2_OFFSET);
}

/*incializa a uhci*/
void iniciar_uhci(void){
        int bus, dev, func;
        char achou= FALSE;
        unsigned long value;
```

```
        for (bus = 0; bus < 255; bus++){
                for (dev = 0; dev < 32; dev++){
                for (func = 0; func < 8; func++){
                                pci_conf_read(bus,dev,func,0x8,0x4,&value);
                                if(value!=0xffffffff){
                /*Se O regitro 08 for 0x0c0300 é UHCI (VIDE PAGINA 19 UHCI Design Guide)*/
                value = (value>>8)&0xffffffff; /*Excluindo Revision ID que não é necessário*/
                                        if(value==0x000c0300)
                                        {
                                /*obtendo registrador contendo endereço base da usb*/

        pci_conf_read(bus,dev,func,0x20,0x4,&uhci_dev.base_adr);
                                                        uhci_dev.base_adr                    =
uhci_dev.base_adr&0xfffffffE;
                                                        achou = TRUE;
                                                break;
                                        }
                                        }
                }
                                if (achou)
                                        break;
                }
                if (achou)
                        break;
        }
        if (achou){
                cprint(0,0,"Foi localizado um dispositivo UHCI, com o endereco base: ");
                hprint(0,59,uhci_dev.base_adr);
                cprint(1,0,"Reiniciando o controlador...");
```

```
                reconfigurar_uhci();
                reconfigurar_portas();
                ativar_portas();
                instalar_framelist();
                /*ativando o controlador, com 64 bytes de tamanho maximo*/
                outw(0x4,uhci_dev.base_adr+USBINTR);
                outw(USBCMD_RS | USBCMD_MAXP | USBCMD_CF,uhci_dev.base_adr);
                cprint(1,29,"OK");
                get_ports_status();
                obter_pid_vid_dispositivo();
//              hprint(12,0,rec_desc[0]);
                while(1){

                }
        }else{
                cprint(0,0,"Nao foi localizado um dispositivo UHCI");
        }
}
```

Arquivo UHCI.h

```
/*USB UHCI
Author: Rômulo Jales
Gets and stes USB PCI controller
*/
typedef unsigned int     dword;
typedef unsigned short  word;
typedef unsigned char   byte;
#define TRUE   1
#define FALSE  0
#define TAMANHO_MX_PCKT 64
```

```
#define ENDERECO_FRAME_LIST 0x1000000
#define ENDERECO_FILA          0x1008000
#define USB_PORT1_OFFSET 0x10
#define USB_PORT2_OFFSET 0x12
#define VELOCIDADE_MAXIMA        0
#define VELOCIDADE_BAIXA  1

/*Identificação do TD TOKEN*/
#define PCKT_ID_OUT 0xe1
#define PCKT_ID_IN     0x69
#define PCKT_ID_SETUP       0x2d

//Código USB-UHCI: 0x0c0300
#define SBC 0x0c //Classe: Serial bus controllers
#define USB 0x03 //Subclasse: usb

#define UHCI 0x00 //Intrest controller
#define OHCI 0x20 //Non Intel or VIA
#define EHCI 0x30 //USB 2.0

/*Comandos*/
#define   USBCMD                0x0000 //offset
#define   USBCMD_RS             0x0001 /*1 - Rodando, 0 - Parado*/
#define   USBCMD_HCRESET 0x0002
#define   USBCMD_GRESET             0x0004
#define   USBCMD_EGSM               0x0008
#define   USBCMD_FGR                0x0010
#define   USBCMD_SWDBG              0x0020
#define   USBCMD_CF                 0x0040
```

```
#if TAMANHO_MX_PCKT == 64
        #define   USBCMD_MAXP       0x0080 /*Caso 1 - 64bytes, 0 - 32bytes */
#else
        #define   USBCMD_MAXP       0x0
#endif

/*STATUS*/
#define   USBSTS                0x0002 //offset
#define   USBSTS_USBINT             0x0001 //0000000000000001
#define   USBSTS_ERROR              0x0002 //0000000000000010
#define   USBSTS_RESUME             0x0004 //0000000000000100
#define   USBSTS_PCI_PROBLEM        0x0008 //0000000000001000
#define   USBSTS_PROC_PROBLEM   0x0010 //0000000000010000
#define   USBSTS_HALTED             0x0020 //0000000000100000

/*Interrupções*/
#define   USBINTR               0x0004
#define   USBINTR_TIMEOUT  0x0001
#define   USBINTR_RESUME   0x0002
#define   USBINTR_IOC               0x0004
#define   USBINTR_SP            0x0008

/*FRAME*/
#define FRNUM                   0x0006
#define FLBASEADD       0x0008
#define SOF                 0x000C

/*PORTA*/
```

ESCOLA POLITÉCNICA DE PERNAMBUCO

```
#define   PORTSC_CONECT_STATUS            0x0001
#define   PORTSC_CONECT_STATUS_CHANGE         0x0002
#define   PORTSC_ENABLE                   0x0004
#define   PORTSC_PEC                      0x0008
#define   PORTSC_DPLUS                    0x0010
#define   PORTSC_DMINUS                   0x0020
#define   PORTSC_RD                 0x0040
#define   PORTSC_RES1                     0x0080
#define   PORTSC_LSDA                     0x0100
#define   PORTSC_RESET                    0x0200
#define   PORTSC_OC                 0x0400
#define   PORTSC_OCC                      0x0800
#define   PORTSC_SUSP                     0x1000
#define   PORTSC_RES2                     0x2000
#define   PORTSC_RES3                     0x4000
#define   PORTSC_RES4                     0x8000

#define UHCI_NUM_FRAMES            1024
#define UHCI_SOF_DEFAULT  64 /*1 milisegundo*/

/*
Espaço de configuração PCI

Registro | bits 31-24  | bits 23-16  | bits 15-8    | bits 7-0
------------------------------------------------------------------------
00       |        Device ID            |     Vendor ID
------------------------------------------------------------------------
04       |        Status               |     Command
------------------------------------------------------------------------
```

```
08      | Class code  | Subclass    | Prog IF     | Revision ID
-----------------------------------------------------------------------
0C      | BIST        | Header type | Latency Timer | Cache Line Size
-----------------------------------------------------------------------
10      |                   Base address #0 (BAR0)
-----------------------------------------------------------------------
14      |                   Base address #1 (BAR1)
-----------------------------------------------------------------------
18      |                   Base address #2 (BAR2)
-----------------------------------------------------------------------
1C      |                   Base address #3 (BAR3)
-----------------------------------------------------------------------
20      |                   Base address #4 (BAR4)
-----------------------------------------------------------------------
24      |                   Base address #5 (BAR5)
-----------------------------------------------------------------------
28      |                   Cardbus CIS Pointer
-----------------------------------------------------------------------
2C      |        Subsystem ID        |       Subsystem Vendor ID
-----------------------------------------------------------------------
30      |        Expansion ROM base address
-----------------------------------------------------------------------
34      |        Reserved                      | Capabilities Pointer
-----------------------------------------------------------------------
38      |        Reserved
-----------------------------------------------------------------------
3C      | Max latency | Min Grant  | Interrupt PIN |      Interrupt Line
-----------------------------------------------------------------------
*/
```

```
struct PCI_DRIVE{
        word pid;
        word vid;
        byte  classCode;
        byte  subClassCode;
        byte  pIf;
        dword   base4;  //De acordo com UHCI specs contem o endereço base!
                        //Se fosse EHCI seria o primeiro != 0;
};
struct UHCI_HOST_STATUS{
};
struct UHCI_HOST_STRUCT{
        dword base_adr;

        dword num_ports;

};
struct UHCI_PORT_STATUS{
        byte is_enable;
        byte is_connected;
};
typedef struct  _deviceDescriptor{
        byte     bLength;
        byte    bDescriptorType;
        word     wbcdUSB;
        byte    bDeviceClass;
        byte    bDeviceSubClass;
        byte    bDeviceProtocol;
        byte    bMaxPacketSize;
```

```
            word   widVendor;
            word   widProduct;
            word   wbcdDevice;
            byte   biManufacture;
            byte   biProduct;
            byte   biSerialNumber;
            byte   bNumConfigurations;
}deviceDescriptorType;
struct REQUISITOR{
        byte   bmRequestType;
        byte   bRequest;
        word   wValue;
        word   wIndex;
        word   wLength;
};
struct DADOS_TRANSFERENCIA_UHCI{
            dword buffer; /*inerente ao que vai transferir*/
            dword quantidade_bytes; /*Quantidade de bytes a transferir do buffer*/
            dword velocidade; /*0 - FULL, 1 - LOW*/
            dword identificador_processo; /*um de PCKT_ID_SETUP PCKT_ID_In PCKT_ID_OUT*/
            dword controle;
            dword endpoint;
            dword endereco_dispositivo;
};
struct QH{
            dword qhlp;              /* Tamanho estrutura: 4 bytes*/
            dword primeiro_td;       /* Tamanho estrutura: 8 bytes*/
            /*Documentação determina que o QH tem que ter 16 bytes*/
            dword foo;               /* Tamanho estrutura: 12 bytes*/
```

```
        dword bar;              /* Tamanho estrutura: 16 bytes*/
};
struct TD{
        dword link;
        dword control;
        dword token;
        dword buffer;
};
void reconfigurar_uhci(void);
void iniciar_uhci(void);
void ativar_portas(void);
void iniciar_qh(void);

#define PORT1_STATUS_LINE 5
#define PORT1_STATUS_COLU 0
#define PORT2_STATUS_LINE 6
#define PORT2_STATUS_COLU 0
```