UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA

TRABALHO DE GRADUAÇÃO

ESTUDO DO PROCESSO DE DESENVOLVIMENTO DE DEVICE DRIVERS

Autor

Daniel Novais Leite {dnl@cin.ufpe.br}

Orientador

Profº Sergio Vanderlei Cavalcante {svc@cin.ufpe.br}

Recife, Março de 2005.

Resumo

O processo de desenvolvimento de Device Drivers é historicamente pouco popular e documentado, feito principalmente pelas empresas fabricantes de hardware e focados no Sistema Operacional Windows. O desenvolvimento de tais drivers para outros Sistemas Operacionais é muitas vezes esquecido, ou então fica a cargo dos próprios usuários que utilizam técnicas de engenharia reversa e a pouca documentação que conseguem. Este trabalho propõe-se a fazer um estudo sobre as tecnologias existentes, as ferramentas para a criação de drivers e a documentação disponível. Inclui também um estudo de caso de um driver desenvolvido pelo Centro de Informática.

Palavras Chave: Device Drivers, Windows Driver Model, Módulos do Kernel e Arquitetura de Sistemas Operacionais.

Agradecimentos

Este trabalho tem um valor especial para mim, pois ele representa uma transição na minha vida, o término da graduação e o início da vida profissional. Além disso, durante nove meses estive envolvido em um projeto de pesquisa e desenvolvimento nesta área. Um projeto de desenvolvimento de device drivers, e apesar de muitas dúvidas e inseguranças, conseguimos concluí-lo.

Agradeço aos meus familiares, por todo o suporte que me foi dado durante toda a minha vida. Em especial a minha mãe, Ana Maria Novais Leite, por todos os conselhos e estímulos.

Agradeço aos meus colegas de estágio, Orlando Campos, Edgard Lima, Bruno Bourbon e Gilberto Alves, por terem me proporcionado uma ótima experiência durante o projeto e muito companheirismo. Este trabalho também representa o sucesso de nosso trabalho e dedicação.

Agradeço a minha namorada Mariana Dantas de Paula, por todo carinho dedicado a mim e por ter me ajudado a lidar com minhas inseguranças.

Agradeço aos meus professores, em especial Ricardo Menezes Campello, pelas suas ótimas aulas e sua amizade.

Agradeço também a AVCIn, pelas horas descontraídas que me ajudaram a suportar o estresse durante todo o curso. Em especial a Vinício Tavares, Leonardo Arcanjo, Leonardo de Paula e Daniel Marinho.

Por fim, agradeço a meus amigos pela companhia, e em especial a Börje Karlsson, Betuca, Flus, Castor e Ulisses, por sempre terem me incentivado a fazer o curso de Ciências da Computação.

Assinaturas

_____

Aluno: Daniel Novais Leite

_____

Orientador: Sergio Vanderlei Cavalcante

Índice

Índice de Figuras

Índice de Tabelas

1. INTRODUÇÃO

O desenvolvimento de *device drivers*[1] sempre foi considerado uma tarefa bastante complexa, a qual exige um grande conhecimento dos detalhes do funcionamento de sistemas operacionais e uma boa experiência em programação. Poucos eram aqueles que se aventuravam neste campo e eram capazes de realizar tal trabalho. Esta complexidade foi um dos fatores que contribuiu para o abandono que esta área sofreu durante vários anos. Poucas pesquisas foram feitas e ainda menos foi publicado.

Hoje em dia os ventos começaram a soprar no sentido contrário. Com a enorme quantidade de novos hardwares disponíveis, os drivers estão tomando um papel mais importante na indústria da informática O mercado os está enxergando com novos olhos e os profissionais da área estão sendo bastante requisitados. A demanda está cada vez maior, entretanto os desenvolvedores capacitados a realizar tal tarefa ainda são pouco numerosos.

Para suprir essa deficiência, diversas empresas estão investindo em ferramentas, tecnologia e mão de obra. A documentação também não foi esquecida e está sendo organizada e publicada, na forma de livros, artigos de pesquisa, sites na internet e dentro de kits de desenvolvimento.

Este trabalho faz um estudo sobre os device drivers e seu processo de desenvolvimento. Será feita uma introdução aos conceitos básicos de drivers, história e a tecnologia existente atualmente. Também será apresentada uma análise das ferramentas utilizadas para o desenvolvimento e a documentação disponível. Em seguida será descrito um estudo de caso de desenvolvimento de um device driver. Por fim, serão apresentados alguns trabalhos futuros e também uma breve conclusão a respeito do processo atual de desenvolvimento.

[1] Device Drivers são programas responsáveis pela interface entre os aplicativos de usuário e os dispositivos eletrônicos ligados ao computador. Mais detalhes na seção 2.3

O objeto do estudo de caso deste trabalho, tanto o device driver e seu desenvolvimento, é o resultado de um convênio entre o Centro de Informática da UFPE e a empresa Waytec Tecnologia em Comunicação Ltda, de acordo com os termos da lei nº 10176/01, a chamada "Lei da Informática".

2. CONCEITOS BÁSICOS

2.1. Sistemas Operacionais

Um computador moderno consiste de processador, memória, disco rígido, teclado, monitor e outros dispositivos de entrada e saída de dados. Estes componentes eletrônicos que formam o computador são chamados de **hardware**. Eles fornecem a capacidade de processar, armazenar, transmitir e visualizar dados. O hardware utiliza sinais elétricos para representar a informação.

Todos os recursos oferecidos pelo hardware são gerenciados por um programa: o **Sistema Operacional (SO)**. O Sistema Operacional faz a configuração do hardware e controla o acesso dos programas à memória, processador, disco rígido e outros dispositivos. Ele funciona como uma camada de abstração que esconde os detalhes do funcionamento do hardware para que os programas (**Software**) possam usufruir de seus recursos utilizando uma interface comum. O SO coordena este acesso, evitando que um hardware seja requisitado por mais de um processo simultaneamente. O SO também faz a proteção de dados, não deixando que um processo altere as informações de outros processos, evitando assim a corrupção dos programas. Além disto, o SO é responsável pela interface com o usuário, permitindo que este carregue seus programas na memória, além de controlar as entradas do teclado, mouse e outros dispositivos de entrada.

Ao se ligar um computador, um software básico que está gravado de forma permanente no computador(**BIOS**) faz pequenos testes no sistema; rotinas básicas de inicialização, detecção e configuração de hardware. Depois entrega o controle ao Sistema Operacional da máquina. Este, por sua vez, cuida das demais funções de inicialização do sistema, como a interpretação dos dados no disco rígido em estruturas bem definidas, a configuração e inicialização dos serviços de rede, o inicialização dos programas responsáveis pelo gerenciamento dos demais equipamentos ligados ao computador, inicialização do display gráfico e finalmente, a criação de um ambiente de interação com o usuário e a disponibilização de seus recursos para que este possa rodar seus programas.

Os sistemas operacionais mais conhecidos atualmente são:

- Microsoft Windows: Nas versões 98, Me, 2000 e XP: São os sistemas operacionais comerciais, presentes na grande maioria dos computadores pessoais da atualidade.
- Unix: A maioria dos sistemas Unix tem código aberto, apenas algumas versões são vendidas comercialmente. Existem várias versões do Unix, pois existem várias empresas e grupos que criam sua própria distribuição do Unix, sendo que todas obedecem a um padrão de compatibilidade. O Unix é um sistema utilizado em várias plataformas, e em geral, estações de trabalho.
- Linux: É a versão do Unix para PCs. Como no Unix, tem diversas distribuições diferentes, mas todas têm em comum um núcleo (**Kernel**), desenvolvido por uma organização de desenvolvedores.
- Mac OS: É o sistema operacional dos computadores da Apple, o Macintosh. As novas versões, chamadas Mac OS X, têm o kernel baseado no Unix.

2.1.1. Sistemas Operacionais Monolíticos

São sistemas operacionais sem uma estrutura bem definida. Neste caso o sistema operacional é basicamente um conjunto de procedimentos, os quais podem invocar uns aos outros quando necessário. Famosos por serem confusos e sem estrutura definida, estes sistemas são de difícil manutenção, pois o código bastante compartilhado causa confusão e faz com que o programador tenha que rastrear os efeitos de qualquer mudança, além de tornar mais complexa a compreensão de quem não está familiarizado com o código.

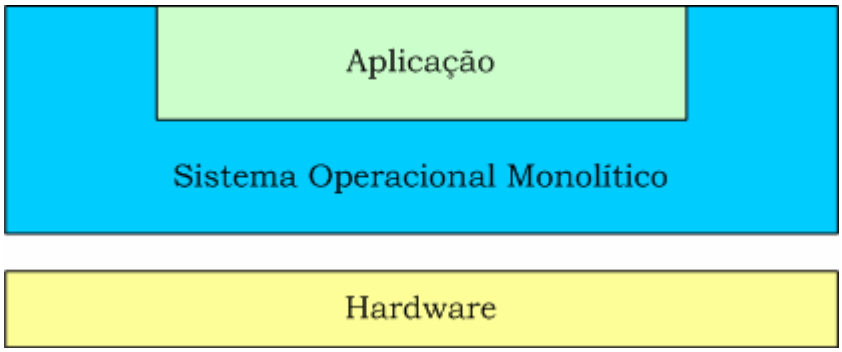

Figura 1: Sistema Operacional Monolítico

Este modelo foi bastante utilizados nos SOs mais antigos, pois tem uma arquitetura simples e de fácil planejamento. Entretanto foi deixado de lado nos SOs comerciais modernos porque dificulta muito o desenvolvimento de sistemas mais complexos e que necessitam de uma grande equipe de desenvolvedores. São bem comuns em sistemas operacionais de menor complexidade, como os SOs embarcados em dispositivos eletrônicos.

2.1.2. Sistemas Operacionais em Camadas

São os sucessores dos sistemas monolíticos. Criados para facilitar o desenvolvimento e manutenção de sistemas complexos. São formados por várias camadas organizadas de forma hierárquica, onde as camadas mais internas têm acesso direto com o hardware, enquanto as camadas mais externas são camadas de interação com o usuário. A abstração vai aumentando de acordo com a distância das camadas ao núcleo. Esta organização facilita a programação, pois os programas estão em uma camada de maior abstração e por isto devem acessar apenas funções de bibliotecas padrão.

Cada camada tem tarefas bem definidas e devem acessar as outras camadas através de uma interface. Desta forma cada camada pode ser desenvolvida independentemente por diferentes equipes, seguindo uma boa documentação. Cada camada também pode ser dividida em subcamadas e componentes com funções bastante específicas.

O sistema operacional também faz uso desta organização para implementar questões de prioridade e segurança. Os programas das camadas mais internas têm maior acesso ao sistema e maior prioridade na fila de tarefas, enquanto que existe uma restrição aos recursos disponíveis para os programas de usuário e sua prioridade é inferior, pois estão em uma camada de abstração mais alta.

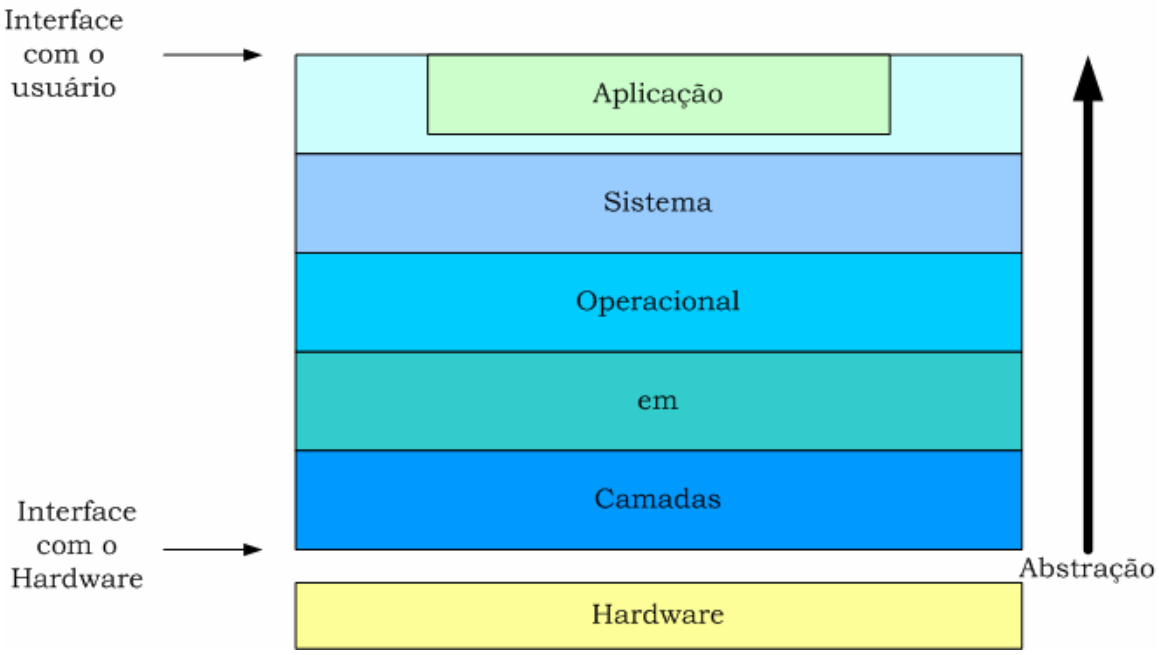

Figura 2: Sistema Operacional em Camadas.

2.2. Máquinas Virtuais

São programas que simulam o ambiente de uma máquina real para outros programas. Os programas que rodam dentro destas máquinas virtuais têm a impressão que estão rodando dentro de máquinas reais específicas, que estão tendo acesso direto ao hardware, entretanto todo o processamento percebido pelo programa é feito pelo software da máquina virtual. A maioria dos programas que rodam em máquinas virtuais pode rodar diretamente em máquinas reais, e em alguns casos, apenas pequenos ajustes são necessários.

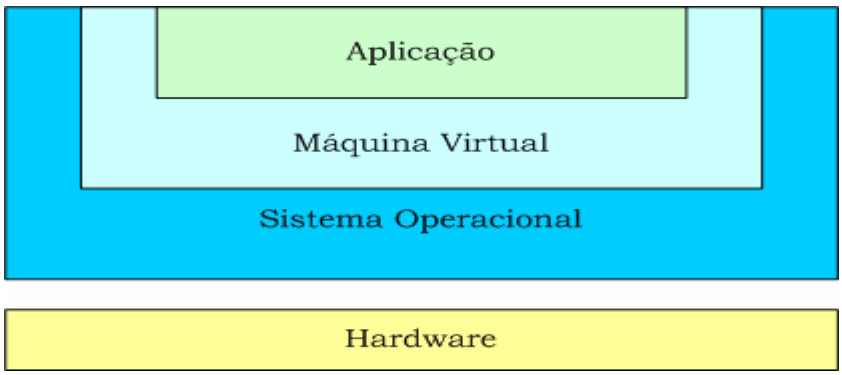

Figura 3: Máquina Virtual

As máquinas virtuais são muito utilizadas para que programas de uma certa plataforma possam rodar em outra plataforma diferente, como também para o desenvolvimento e depuração de programas.

Um bom exemplo de máquina virtual é a máquina virtual Java, que é implementada para diversas plataformas, permitindo assim que o programador desenvolva apenas um código e o execute em diversos ambientes diferentes.

Algumas máquinas virtuais comerciais conhecidas são:

- VMware Workstation da VMware
- Virtual PC da Microsoft.
- Máquina Virtual Java

2.3. O que são Device Drivers?

Para estender as funcionalidades de um computador qualquer, vários equipamentos eletrônicos podem ser ligados a ele, permitindo então que o sistema composto tenha a capacidade de realizar tarefas adicionais, como, por exemplo, imprimir um documento em uma folha de papel ou fazer uma conexão com outro computador através de uma ligação telefônica. Tais equipamentos podem ser impressoras, modems, placas controladoras de áudio, câmeras de vídeo e outros. Estes componentes são equipamentos essencialmente eletrônicos e o computador ao qual o está ligado pode ter controle sobre seu funcionamento ou simplesmente coletar informações. Em geral, um computador comum gerência diversos dispositivos eletrônicos.

Para que um computador consiga trocar informações com algum hardware a ele conectado, é necessária a troca de sinais contendo informações. Também é necessário que tanto o dispositivo quando o computador entendam o mesmo **Protocolo** ("linguagem" específica, a qual as duas partes utilizam para se comunicarem).

Para que a comunicação ocorra, devem existir componentes de cada lado que são responsáveis pelo entendimento, envio e recebimento das informações. Dentro dos

dispositivos eletrônicos o componente que gerencia esta troca de informações com o computador é a chamada **Controladora do dispositivo**. Do outro lado, quem tem esta responsabilidade é o **Device Driver**, que é integrado ao Sistema Operacional e juntos são responsáveis pela interface entre o computador (programas) e o hardware ligado a ele.

Existe uma grande variedade de dispositivos diferentes e de sistemas operacionais. As controladoras podem entender diversos protocolos e cada uma funciona de forma específica. Por outro lado, cada sistema operacional trata das requisições ao hardware de forma diferente e tem métodos próprios para interagir com os device drivers. Para que então esta comunicação entre o computador e o equipamento a ele conectado seja possível, é necessário que existam vários device drivers diferentes para fazer a tradução entre a linguagem de cada controladora e a linguagem do Sistema Operacional ao qual está conectada.

Para ter acesso ao hardware do dispositivo, os device drivers têm que ter acesso privilegiado aos recursos disponibilizados pelo sistema operacional. Tradicionalmente, os drivers são ligados ao SO, tornando-se parte dele. Os drivers executam em um modo de operação que permite ter acesso à funções e variáveis mais internas do kernel, de forma que podem utilizar praticamente qualquer recurso do sistema. Este acesso tão irrestrito, chamado **modo kernel**, traz um pequeno problema de segurança, pois os drivers estão tão estreitamente ligados ao resto do kernel que alguma falha no driver pode comprometer todo o sistema. A robustez fica então a cargo dos seus programadores, os quais têm a responsabilidade de evitar tais falhas.

Para que os drivers possam executar em modo kernel, é preciso que eles sejam ligados ao sistema operacional. Atualmente 3 métodos diferentes são utilizados:

- Recompilar o kernel junto com o device driver e em seguida reinicializar a máquina. Esta forma deixa o device driver ligado estaticamente ao SO. Este tipo de ligação não permite que o driver seja retirado da memória facilmente. É usado em alguns sistemas UNIX.

- Criar uma entrada num arquivo do sistema operacional, a qual aponta para o device driver e indicar a necessidade de carregar aquele arquivo junto ao kernel durante a inicialização da máquina. Vários sistemas Windows utilizaram esta abordagem.
- Inserir o driver junto ao kernel em tempo de execução. Esta abordagem é considerada mais complexa que as anteriores mas está sendo amplamente utilizada hoje em dia pelo SOs mais modernos.Tem a grande vantagem de não precisar reiniciar o computador ao se instalar um novo driver.

Existe também a possibilidade de se desenvolver drivers em **modo usuário**, isto é, acesso ao SO de forma comum ou bastante parecida com a forma com a qual as aplicações de usuários interagem com o sistema. Este modelo não permite grande acesso ao hardware e possui várias restrições. Algumas propostas de melhorias para esta abordagem ainda estão sendo estudadas.

2.4. Visão geral dos Sistemas Operacionais Modernos

2.4.1. Windows XP e 2000

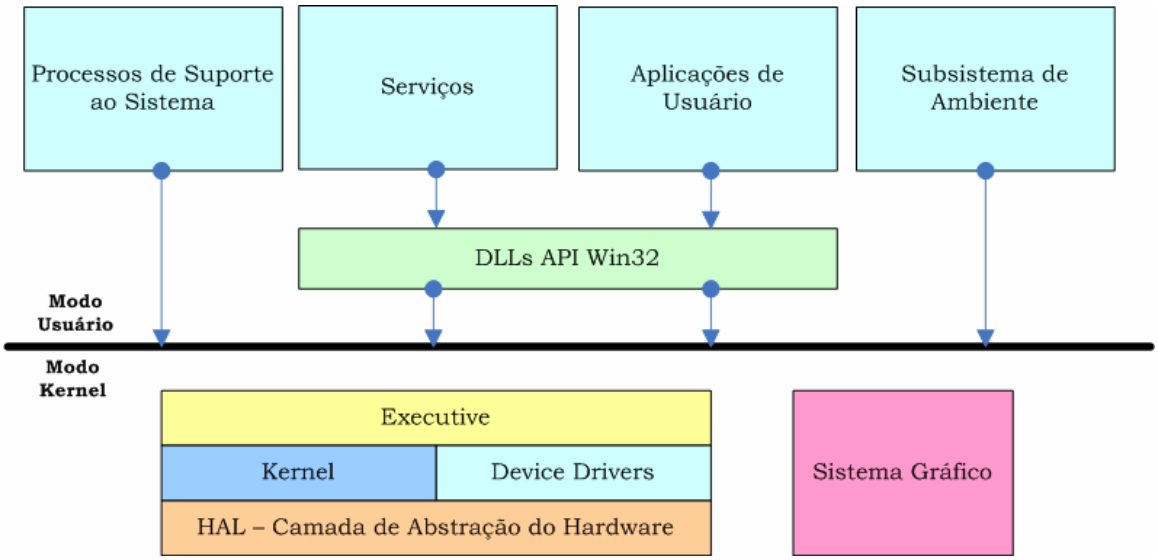

Figura 4: Arquitetura simplificada do Windows XP/2000

Este é um diagrama resumido da arquitetura utilizada nos Windows XP e Windows 2000. As plataformas suportam dois modos de execução: modo usuário e modo kernel. O software que está acima da linha que separa os modos executa em modo usuário, enquanto os componentes abaixo da linha rodam modo kernel.

Os quatro tipos básicos de processos de Usuário são:

- Processos de Suporte ao Sistema: Processos como o login e o gerenciador de sessão, os quais não são inicializados pelo gerenciador de serviços de controle.
- Serviços: Agrupa todos os demais serviços Win32, tais como o Task Scheduler e o Spooler, além dos outros softwares que executam como serviço.
- Aplicações de Usuário: Os programas do usuário.
- Subsistemas de Ambiente: O qual possibilita que as aplicações de usuário acessem os serviços nativos do sistema operacional através de um conjunto de funções.

Dentro desta arquitetura, as aplicações de usuário não chamam diretamente os serviços nativos do Sistema Operacional. As aplicações acessam tais serviços através das bibliotecas ligadas dinamicamente (DLL) da API Win32, as quais traduzem as funções documentadas em alguma chamada interna de um serviço do sistema.

Os componentes do modo Kernel:

- Executive: Os serviços básicos do Sistema Operacional, como gerenciamento de memória, segurança, E/S e a comunicação entre-processos.
- Kernel: Funções de baixo-nível do sistema operacional, como o gerenciamento de *threads*, envio de interrupções e exceções, sincronização em multiprocessamento e outros. Também provê um conjunto de rotinas e objetos básicos utilizados para implementar construções de nível mais alto.
- Device Drivers: Contém tanto drivers para equipamentos eletrônicos de E/S como drivers de sistema de arquivos e rede.
- Camada de Abstração de Hardware (HAL): Uma camada de software que isola o kernel, device drivers e o resto do executive das diferenças de plataformas específicas.
- Sistema Gráfico: Implementa as funções da interface gráfica com o usuário (GUI).

2.4.2. Exemplo de requisição de E/S no Windows XP/2000

Quando um programa de usuário quer ler algum dado de um dispositivo, o programa deve usar uma API, como por exemplo a função ReadFile() da API Win32. Um módulo do sub-sistema, a biblioteca KERNEL32.DLL implementa esta API invocando a função da API nativa NtReadFile().

A função NtReadFile faz parte do componente do sistema chamado I/O Manager. A função cria uma estrutura de dados chamada I/O Request Packet (IRP), que é repassada para algum device driver. O driver, caso tenha que realmente acessar o hardware, pode fazer uma leitura direcionada a uma porta de E/S ou a algum registro de memória implementado pelo dispositivo. Mesmo podendo acessar diretamente o hardware, os drivers também podem usar as facilidades da HAL. Esta camada de abstração usa métodos dependentes de plataforma para ler os bytes de alguma porta de E/S. No caso de computadores x86 o HAL usa a instrução IN.

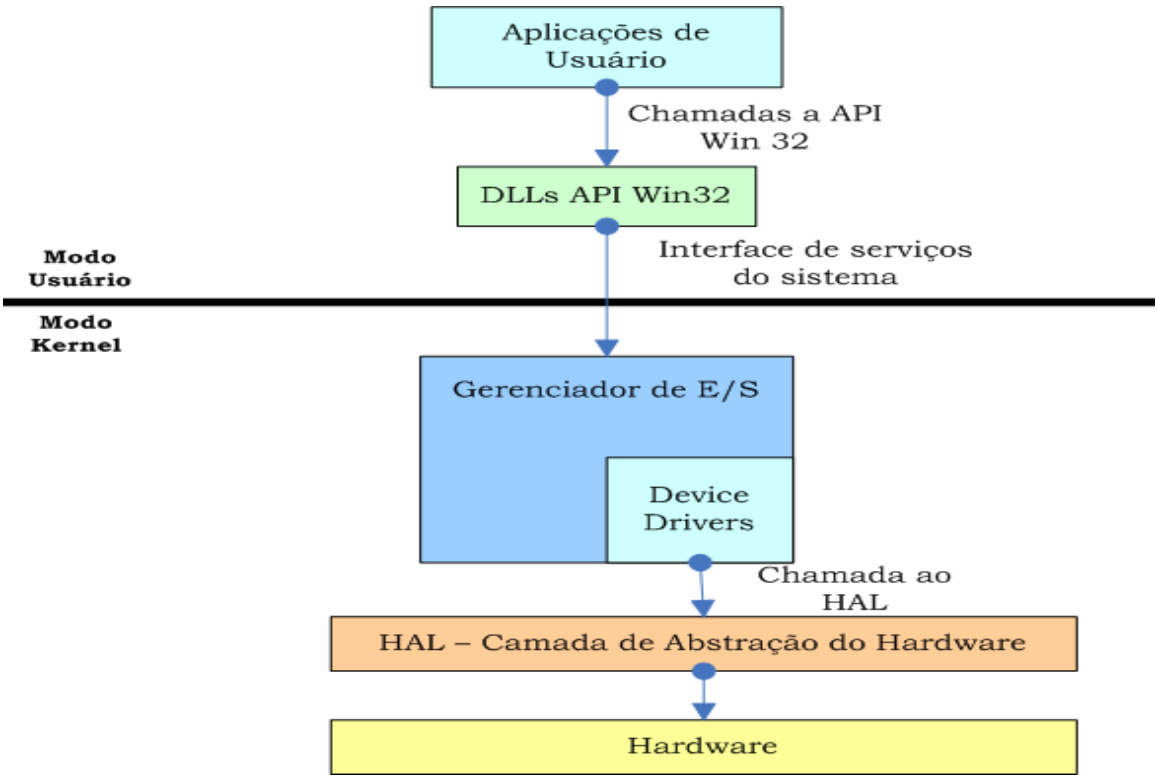

Figura 5:Requisição de E/S no Windows XP/2000

Ao terminar a operação de E/S, o driver completa a IRP chamando uma rotina particular em modo kernel, a qual a finaliza e permite que a aplicação a qual estava esperando volte à execução.

As rotinas do chamado I/O Manager operam em modo kernel para servir a necessidade das aplicações interagirem com o dispositivo. Todas elas fazem a validação de seus parâmetros, evitando assim que programas de usuários acessem dados que não poderiam.

2.4.3. Windows 98/Me

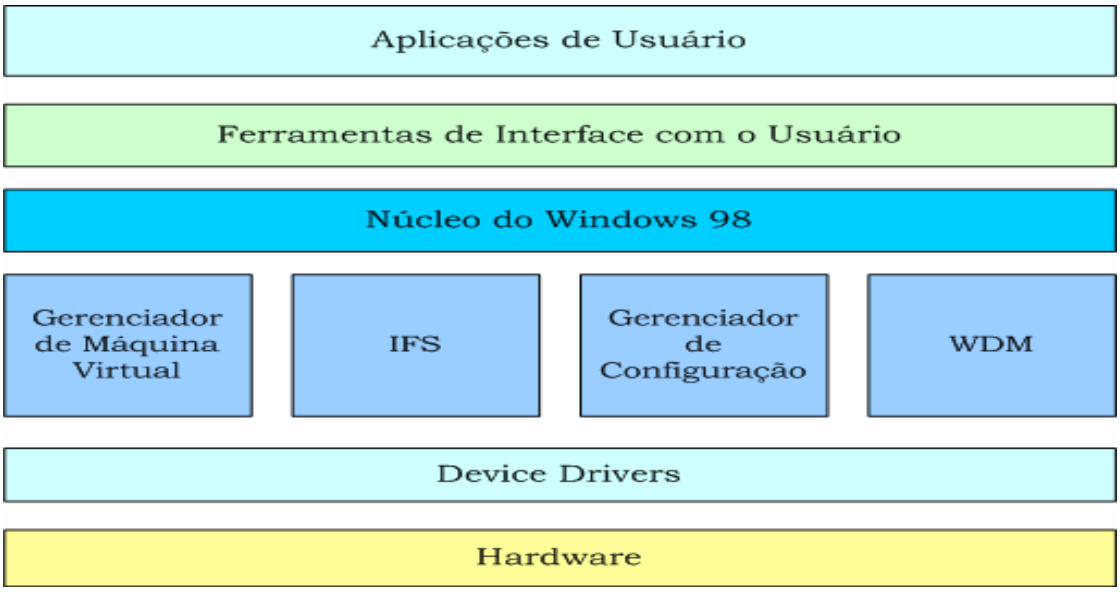

Figura 6: Arquitetura simplificada do Windows 98/ME

A figura 4 é uma visão geral da arquitetura presente no Windows 98 e Windows ME. Nela estão definidos seus principais componentes. Ambos SOs têm uma estrutura muita parecida. O Windows 98/Me tem suporte a aplicações Win32, Win16 e MS-DOS, utilizando máquinas virtuais para executá-las. O Windows 98/ME também provê um conjunto de ferramentas que fazem interface com o usuário e unifica o acesso a informação.

O núcleo do sistema é composto por 3 subsistemas: User, Kernel e uma interface com dispositivos Gráficos (GDI). Cada um destes componentes contém um par de DLLs (uma com 32 e outra com 16 bits) que provêem os serviços para as aplicações de usuários.

- User: O componente gerencia as entradas do teclado, mouse e outros dispositivos de E/S para a interface com o usuário. O componente também é responsável pela interação com o driver de som, temporizador e portas de comunicação.
- O Kernel provê as funcionalidades básicas do sistema operacional, como serviços de E/S para arquivos, gerenciamento de memória, gerenciamento de tarefas, threads e tratamento de exceções. Quando um usuário quer iniciar uma aplicação, o kernel carrega na memória o executável e as DLLs necessárias. O kernel provê serviços para aplicações 16 e 32 bits usando um processo de tradução, convertendo aplicações 16bits em equivalentes 32bits.
- O componente GDI é um sistema gráfico que gerencia o que vai ser mostrado na tela. Também provê suporte para impressoras e outros dispositivos de saída.

Abaixo do núcleo do sistema existem alguns componentes que fazem o gerenciamento de detalhes específicos:

- Gerenciador de WDM: Dá suporte aos drivers WDM no Windows 98/ME adicionando uma nova camada à arquitetura VxD herdada das versões antigas do Windows.
- Gerenciador de Configuração: Provê o suporte à tecnologia Plug and Play
- Gerenciador de Sistemas de Arquivos Instaláveis (IFS): Provê suporte à vários tipos de sistemas de arquivos.
- Gerenciador de Máquinas Virtuais: Gerencia os recursos necessários para cada processo rodando no sistema. O gerenciador de Máquinas Virtuais cria e mantém os ambientes nos quais as aplicações e processos do sistema rodam.

2.4.4. Exemplo de requisição de E/S no Windows 98/ME

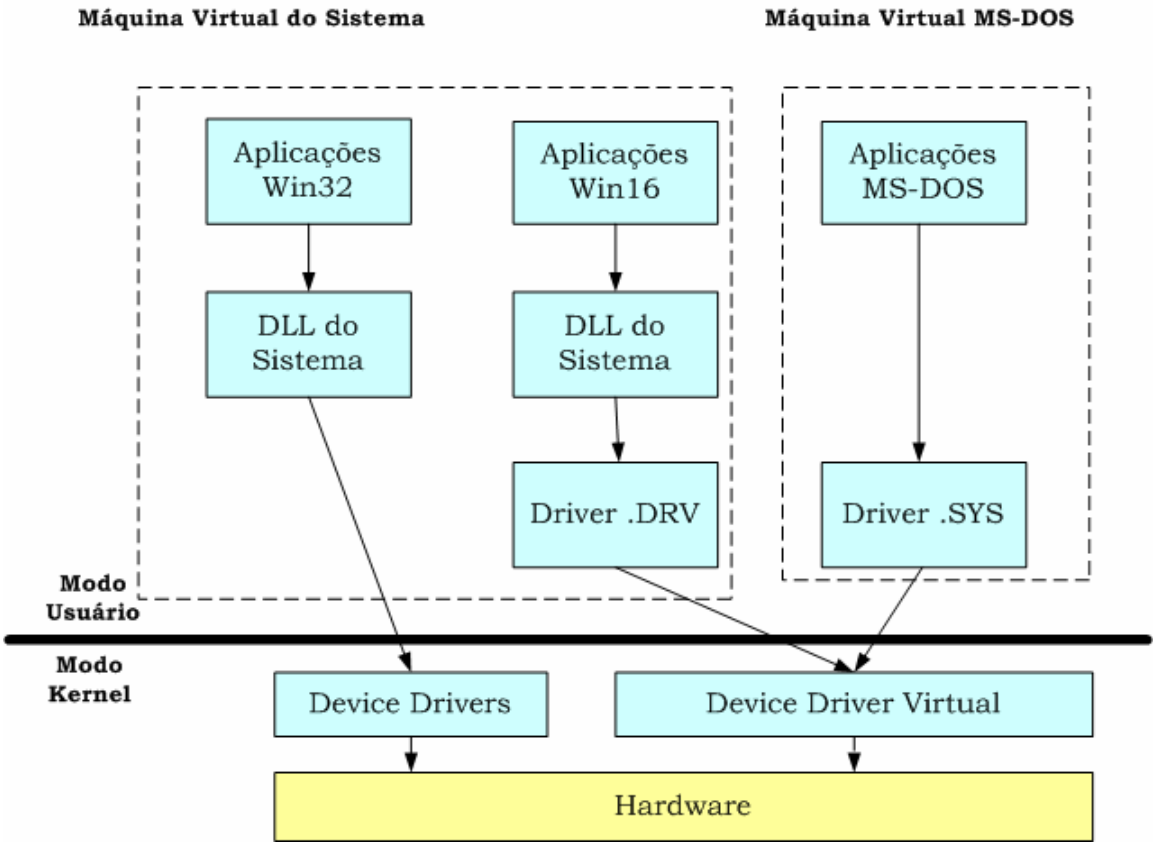

Figura 7: Requisição de E/S no Windows 98/ME

No Windows 98/ME, apesar de todas os programas rodarem dentro de máquinas virtuais, existe uma diferença de como as aplicações Win32, Win16 e MS-DOS rodam.

A parte esquerda da figura 5 mostra o caso das aplicações Win32. O executável de 32 bits chama uma função da API Win32, como ReadFile(), que é implementada por alguma DLL do sistema, como o KERNEL32.DLL. A DLL valida os parâmetros e então acessa em o driver requisitado em modo kernel.

Já as aplicações Win16, mostradas na coluna do meio, e as aplicações MS-DOS, mostradas na coluna à direta, chamam direta ou indiretamente os serviços de um driver modo usuário. As aplicações Win16, chamam uma DLL do sistema, que por sua vez repassa a requisição de E/S para um driver modo usuário .DRV. As aplicações MS-DOS por sua vez, acessam diretamente o driver .SYS em sua máquina virtual. Em ambos os casos, o

sistema intercepta as requisições de E/S feitas pelos drivers das máquinas virtuais e simula o acesso ao hardware para evitar que haja mais de um acesso ao hardware ao mesmo tempo.

2.4.5. Linux (Kernel 2.4)

Um sistema Unix pode ser visto como uma pirâmide, onde cada camada tem uma função específica e se comunica apenas com as camadas imediatamente adjacentes.

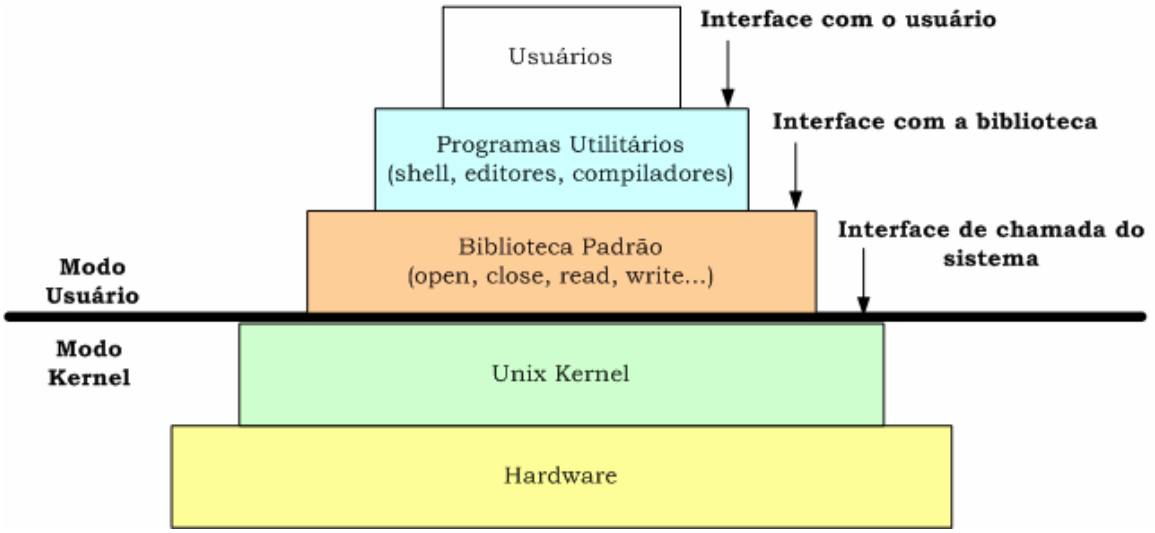

FIGURA 8: CAMADAS DO UNIX

Na base da pirâmide estão os elementos do hardware.

Logo acima do hardware fica o kernel do Unix, o qual compreende o gerenciador de memória, gerenciador de processos, os arquivos de sistema e mais algumas funcionalidades. Esta camada provê uma interface para as chamadas do sistema.

Na terceira camada, fica a biblioteca padrão. Uma coleção de funções básicas que são chamadas pelos programas para acessarem as camadas inferiores através de chamadas do sistema.

Acima disto fica uma camada que serve de interface com o usuário(topo da pirâmide). Esta quarta camada é composta pelo shell, compiladores, editores e programas, os quais acessam o resto do sistema através da interface da biblioteca padrão.

A última camada representa os usuários. Estes são os atores que vão acessar o sistema, através do shell ou interface gráfica. Eles vão carregar programas de usuário ou utilizar as ferramentas disponibilizadas pelo sistema, como editores e compiladores.

A forma de acessar os dispositivos de E/S no Unix foi projetada para ser bastante simples. O sistema integra o hardware ao sistema de arquivos, e o as operações de leitura e escrita são feitas da mesma forma utilizada para arquivos normais, utilizando as mesmas funções do sistema. Tais arquivos são chamados arquivos especiais. Cada equipamento tem um arquivo associado a ele, geralmente localizado em "**/dev**". Este arquivo também está associado a um módulo do kernel (o qual faz o papel de device driver).

Quando uma aplicação deseja fazer uma leitura em algum dispositivo, ela utiliza as funções da biblioteca padrão (Open, Close, Read, Write..) que por sua vez, aplica estes métodos ao arquivo associado. Quando isto é feito, o Sistema Operacional procura o módulo responsável pelo arquivo e executa a função correspondente que foi pedida pela aplicação. Quando a operação é terminada, o módulo informa o término ao Sistema Operacional, o qual retorna o resultado através do retorno da função da biblioteca padrão. Assim a aplicação tem a impressão de estar lendo um arquivo normal ao invés de estar lendo de um hardware externo.

2.5. Histórico dos Device Drivers

Durante os primeiros anos da era dos computadores digitais, os device drivers não existiam na forma em que os conhecemos hoje. Cada programador escrevia suas próprias funções para acessar os dispositivos de entrada e saída (E/S) da máquina para a qual estava codificando. Tarefas comuns como ler dados de alguma fita magnética ou imprimir a saída dos programas eram sempre reescritas nos diferentes programas. Os programadores tinham então rotinas prontas, que poderiam copiar dentro de diversos programas para fazer tais tarefas.

Foi então que a idéia de reuso destes pedaços de software começou a aparecer. Os programadores notaram que estavam duplicando o esforço, reescrevendo várias vezes o mesmo código e acharam que seria uma boa alternativa escrever pequenos softwares

especiais com essas funcionalidades que seriam compartilhados entre todos. Estes softwares ficaram conhecidos como drivers. Alguns programadores da época ainda resistiram ao processo de compartilhamento, pois acreditavam que conseguiam fazer rotinas melhores, mais rápidas ou mais eficientes, entretanto apesar desta pequena resistência a idéia de compartilhamento dos drivers prevaleceu.

Os primeiros drivers escritos de forma compartilhada, eram aplicações em linguagem da máquina, e suas funções eram documentadas, para que os diversos programadores pudessem utilizá-las (mesmo que essa documentação estivesse dentro do código). Os drivers eram muito variados pois sua implementação era muito ligada à arquitetura para a qual tinha sido planejado. Com a diversidade de máquinas utilizadas e a forma como os drivers eram construídos, era praticamente impossível o reuso dos mesmos drivers para diferentes plataformas.

Em 1973, um dos primeiros passos para o reuso do software de baixo nível foi dado. O kernel do UNIX foi reescrito em linguagem C, o que o tornou o primeiro sistema operacional facilmente portável entre diversas plataformas.

Com o advento dos computadores pessoais, começou um avanço no desenvolvimento dos drivers. Os primeiros PCs tinham processadores Intel com 640KB de memória, endereçada diretamente com endereços de 20 bits. Os processadores tinham apenas um modo de operação, chamado "real mode". A arquitetura incluía o conceito de slots de expansão, onde os usuários poderiam colocar placas compradas isoladamente. Essas placas geralmente vinham com instruções de como ajustar os chaveadores para pequenas mudanças na configuração.

Os device drivers dessa época, eram programas de 16 bits, escritos em linguagem assembly e baseados na instrução INT para comunicação com a BIOS e com os serviços do sistema MS-DOS.

Tempos depois, a IBM lançou os PCs da classe AT, baseados no processador Intel 80286. Este processador adicionou um modo de operação chamado modo protegido, onde os programas poderiam endereçar até 16MB de memória principal e extendida, usando endereços de 24-bit. Apesar disto, o MS-DOS continuou sendo um sistema operacional *real mode*.

Enquanto os processadores evoluíam, avanços também foram feitos na área de sistemas operacionais. A Microsoft que criou o ambiente gráfico Windows, teve também que desenvolver uma coleção de Windows Drivers para dispositivos comuns como display, teclado e mouse. Estes drivers eram arquivos executáveis com a extensão .DRV e assim como os drivers do MS-DOS, eram escritos principalmente em linguagem assembly.

Ao lançar uma versão do Windows com modo protegido a Microsoft modificou os drivers *real mode* .DRV para executarem no modo protegido. Mas o hardware que não fosse padrão do Windows (display, teclado e mouse) continuaria a ser gerenciado pelos drivers *real mode* do MS-DOS.

Pouco depois dos PCs com processadores 80386 terem sido lançados, a Microsoft apresentou a versão 3.0 do Windows, agora com um novo modo de operação: *Enhanced*, o qual poderia explorar na íntegra as capacidades de memória virtual. Entretanto cada hardware ainda necessitava de um driver *real mode*.

Este novo Windows apresentou um novo problema. O Windows teria que rodar múltiplas instâncias de aplicações MS-DOS e, para isto, o Windows teve que suprir uma máquina virtual para cada aplicação. Contudo as aplicações tentavam acessar diretamente o hardware através de instruções IN e OUT como faziam no MS-DOS, lendo e escrevendo na memória do hardware e gerenciando as interrupções do hardware. Estas aplicações provavelmente iriam entrar em conflito.

Para resolver este problema, a Microsoft introduziu o conceito de device driver virtual, comumente chamados de VxDs. Estes drivers virtuais eram uma camada mediadora entre as aplicações e o hardware, onde eles interceptavam as tentativas de acesso direto ao hardware feitas pelas aplicações e em seguida colocavam o processador em um tipo de "real mode" chamado "virtual 8086" para poder rodar o drivers *real mode* do MS-DOS. Esta estratégia era apenas uma forma de contornar problemas estruturais da arquitetura do Windows 3.0.

Logo em seguida a Microsoft lançou o Windows NT, o qual foi derivado do sistema operacional OS/2 que estava sendo desenvolvido em parceria com a IBM, até que o acordo foi desfeito e a Microsoft resolveu então desenvolvê-lo e transformá-lo no Windows NT. O novo sistema foi desenvolvido com uma nova arquitetura, tendo em vista fazê-lo mais

resistente e seguro que seus antecessores. Os drivers do Windows NT tiveram que usar uma nova tecnologia, bastante diferente da utiliza nos drivers até então produzidos. Estes novos drivers eram escritos em linguagem C, portanto poderiam ser recompilados para novas arquiteturas sem requerer nenhuma mudança no código.

O Windows 3.0 evoluiu para os Windows 3.1 , 3.11 e 95. Este último trouxe algumas novidades, como a configuração automática do Hardware através de Plug and Play. Já os drivers pouco evoluiram, os drivers VxD usados no Windows 95 agora tinham 32 bits e rodavam em modo protegido.

Enquanto isso, o Windows NT chegava a sua versão 4.0, e nova tecnologia de drivers era necessária para estes sistemas. Foi então que a Microsoft desenvolveu a Windows Driver Model (WDM), e a disponibilizou nos seus sistemas operacionais Windows 98, Me, 2000 e XP. A intenção era desenvolver uma tecnologia com a capacidade de escrever um mesmo driver e utilizá-lo sem modificações em todas as novas plataformas. Como o MS-DOS estava presente apenas como opcional, não mais existia a necessidade de se criar drivers real mode compatíveis com ele, logo este tipo de driver praticamente deixou de ser desenvolvido.

Já na plataforma Unix os caminhos mudaram um pouco. Durante algumas décadas, os drivers mantiveram praticamente a mesma tecnologia de quando foram criados. Eles eram ligados estáticamente ao kernel e carregados na memória durante a inicialização. Um das maiores causas desta estagnação foi o fato do Unix rodar principalmente em mini-computadores e estações de trabalho, as quais geralmente não estão ligados com muitos dispositivos.

Com a chegada do Linux, esse cenário mudou. O número de equipamentos disponíveis para a plataforma PC é muito maior que a quantidade disponível para mini-computadores. Para resolver este problema a idéia de escrever os drivers como módulos instaláveis, de forma dinâmica foi desenvolvida. Esses módulos são chamados de módulos do kernel, e podem ser compilados junto com o mesmo ou então ligados ou desligados dinamicamente durante a execução, através das funções INSMOD e RMMOD. Estes drivers devem ser escritos em linguagem C, como o próprio kernel do Unix, obedecer a uma interface padrão e serem compilados para a plataforma específica onde irão funcionar.

2.6. Características dos Drivers Atuais

Os device drivers modernos são escritos principalmente em linguagem C/C++. Eles devem implementar uma interface padrão (**API**) com funções de inicialização, de carga/descarga, registro de funcionalidades, leitura e escrita, gerenciamento de memória. Esta API é definida pelo Sistema Operacional no qual serão inseridos. O SO utiliza esta API para chamar as funções do driver para inicializá-lo, retirá-lo da memória, executar leituras e escritas e outras funções do dispositivo.

Ao rodar em modo kernel, os drivers têm acesso privilegiado ao hardware e as rotinas mais internas do SO. A programação neste ambiente requer um esforço maior quanto a questão de gerenciamento de memória e outros recursos, pois o modo kernel impõe restrições ao acesso a tais componentes.

Por outro lado, as funções de API de modo usuário não estão disponíveis, como também não é possível utilizar os mesmos softwares para depuração de código utilizados no desenvolvimento de aplicações de usuário. Estes fatores contribuem para deixar a programação em modo kernel mais complexa.

Alguns drivers exigem certas funcionalidades que não podem ser implementadas em modo kernel, como por exemplo drivers que necessitam de interface gráfica. Tais drivers possuem então uma arquitetura híbrida, uma parte em modo kernel e outra em modo usuário. A parte em modo kernel faz a comunicação com o hardware e algumas das funções de nível mais baixo, enquanto que a parte em modo usuário faz a interface com o usuário. Ambas partes devem se comunicar para o total funcionamento do driver.

Em alguns casos, os desenvolvedores disponibilizam ferramentas junto com os drivers para que os usuários possam fazer ajustes na configuração e para adequar as funcionalidades aos seus interesses. Estes aplicativos são à parte modo usuário dos drivers, onde na maioria das vezes, são aplicações nativas da plataforma, capazes de se comunicar com o device driver. É possível também a possibilidade de ferramentas de configuração remotas, isto é, programas que rodam em outras máquinas, mas que controlam o funcionamento de drivers remotamente.

2.7. Funcionamento de um Driver

Ao carregar um driver na memória, o Sistema Operacional chama a função padrão de inicialização do driver, e esta por sua vez, deve informar ao SO o tipo de dispositivo que gerencia, as funções que implementa e que podem ser invocadas, além de reservar os recursos necessários para tal tarefa (como por exemplo as portas de entrada/saída do aparelho em questão e sua memória interna).

Ao receber algum evento do hardware, o SO verifica qual driver instalado é responsável pelo dispositivo e se o mesmo têm alguma função para tratar aquele evento. O SO deverá então executar a função definida para aquele evento, a qual poderá utilizar os recursos já previamente alocados durante a inicialização. Caso o driver não tenha uma função específica, pode usar funções de tratamento genéricas, ou então o próprio SO pode responder ao evento recebido ou então ignorá-lo.

Estas funções de tratamento de eventos definem as funcionalidades de cada driver. É o código delas que determina o comportamento do driver. Estas funções podem ser implementadas gradualmente, não necessitando, para o funcionamento do driver, que todas estejam presentes. Esta capacidade é muito útil na etapa de desenvolvimento e testes, pois permite que sejam testadas funções específicas e de forma incremental.

Por final, o driver deve implementar funções de liberação de recursos e de descarga da memória, que são bastante úteis quando se quer testar várias versões do mesmo driver, com a vantagem de não se necessitar reiniciar a máquina.

2.8. Tipos de Device Drivers

Os drivers podem ser de dois tipos: Modo usuário ou Modo Kernel. Alguns drivers podem também ter partes em modo usuário e outra parte rodando em modo kernel. Esta abordagem híbrida é muito comum quando um driver precisa ter alguma aplicação gráfica para interação com usuário ou configuração, como é o caso de drivers de impressoras.

2.8.1. Modo Usuário

Os drivers modo usuário são essencialmente programas que entendem o protocolo usado pela controladora e conseguem acessar os dados do hardware via algum driver já existente.

Um exemplo comum disto são drivers para equipamentos com interface serial ou paralela. O sistema operacional já disponibiliza drivers e métodos que controlam o fluxo de dados em tais portas, logo, os desenvolvedores utilizam tais funções do SO para acessar os dados recebidos pelas portas. Como a funcionalidade de envio e recebimento de dados já está implementada, fica a cargo do driver apenas a compreensão do protocolo e implementar o comportamento do driver. Rodando em modo usuário, o software tem acesso à biblioteca padrão do SO como qualquer aplicação normal de usuário, conseqüentemente a implementação fica muito simplificada.

Os drivers modo usuário tem espaço na pilha ilimitado, pontos de entrada e interface com o sistema bastante diferente dos drivers modo kernel, acesso à biblioteca padrão do SO e uma forma mais fácil de depuração do código, pois podem utilizar *debuggers* de modo usuário.

2.8.2. Windows

Existem vários tipos de driver utilizados nas versões atuais do Microsoft Windows, em especial estão os drivers da Windows Driver Model (WDM) que estão sendo amplamente utilizados. A Windows Driver Model provê um framework para que os drivers criados utilizando esta tecnologia possam rodam em todos os sistemas operacionais que a suportam, assim como os Windows 98/Me/ XP e 2000. A figura abaixo mostra alguns tipos de drivers existentes:

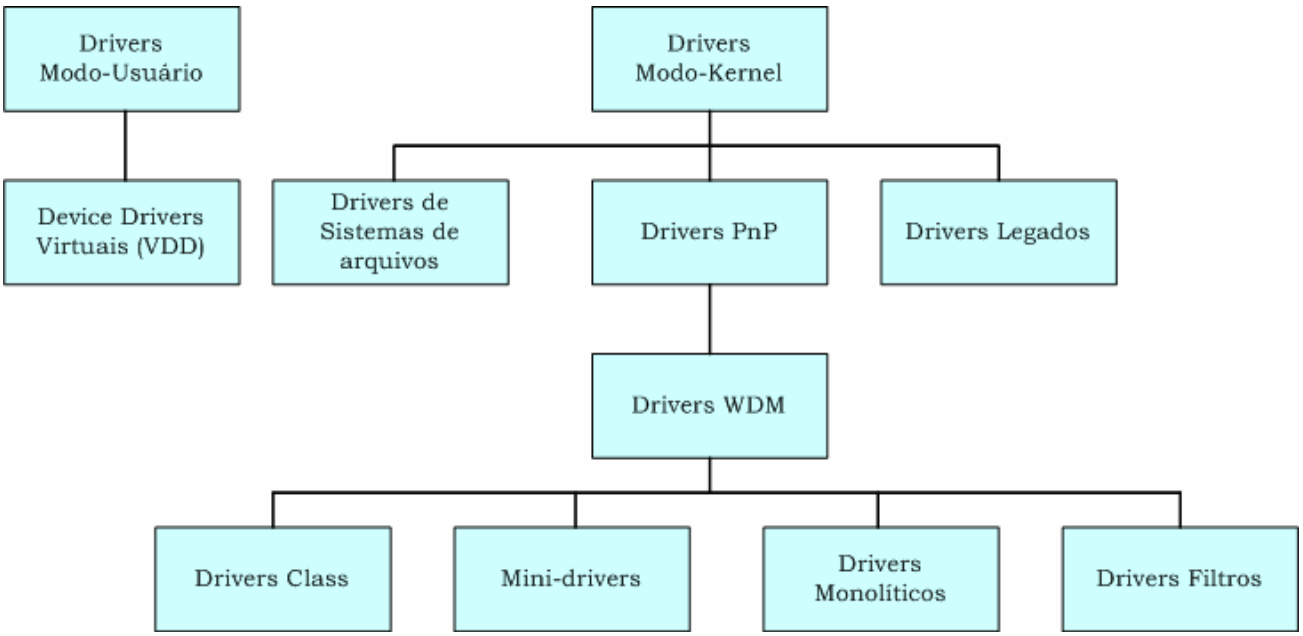

Figura 9: Tipos de Drivers para Windows

Virtual Device Drivers (VDD): São componentes em modo-usuário que permitem aplicações baseadas em MS-DOS acessar o hardware em plataformas Intel x86. Um VDD é baseado na máscara de permissão de E/S para capturar o acesso as portas. Ele simula a operação do equipamento para o qual a aplicação foi originalmente programada possa se comunicar diretamente com o hardware.

Apesar de também serem chamados Virtual Device Drivers e terem em essência a mesma função de virtualizar o acesso ao hardware, os VDD que estão presentes no Windows XP são diferentes dos VxD presentes nos Windows 98 e Me, pois utilizam tecnologia diferente.

2.8.2.1. Modo Kernel

A maioria dos device drivers roda no chamado Modo Kernel, pois têm acesso a certas estruturas do sistema que os drivers modo usuário não tem acesso.

Dentro dos chamados drivers modo kernel existem várias sub-categorias como drivers do sistema de arquivos, drivers legados e os drivers Plug and Play (PnP).

Drivers de Sistema de Arquivos: São drivers que implementam o modelo padrão de sistema de arquivos nos discos locais ou então sobre conexões de rede.

Drivers Legados: São drivers que controlam diretamente o hardware sem a ajuda de outros drivers. Esta categoria é composta por drivers de versões antigas do Windows NT mas que ainda estão sendo usados no Windows XP e 2000.

Drivers Plug and Play: São drivers que entendem os protocolos PnP. Entre estes estão os drivers WDM, que são o foco deste trabalho para as versões Windows.

Drivers WDM: São drivers PnP que tem a capacidade de entender protocolos de gerenciamento de energia e são compatíveis tanto com os Windows 98/Me quanto com os Windows 2000/XP.

Drivers WDM-Class: São drivers WDM genéricos que gerenciam dispositivos pertencentes a alguma classe de hardware definida pela Microsoft. Não tratam de detalhes específicos dos dispositivos, mas de detalhes comuns a todos os equipamentos daquele tipo. É necessário que sejam usados em conjunto com algum outro driver. Em geral são escritos pela própria Microsoft.

Drivers WDM-MiniDrivers: São drivers específicos para a utilização em conjunto com aos Class Drivers. Os MiniDrivers especificam o comportamento do dispositivo enquanto utilizam Class drivers para funções genéricas de gerenciamento do hardware.

Drivers WDM-Filter: Utilizados para dispositivos que funcionam de forma muito semelhante ao padrão Microsoft. Estes drivers modificam o comportamento de um driver genérico, "filtrando" as operações de E/S para que este se adapte ao funcionamento do dispositivo. Não é muito utilizado pois é difícil conseguir modificar a forma que um driver genérico acessa o hardware.

Drivers WDM-Monolíticos: São drivers que incorporam todas as funcionalidades necessárias para dar suporte ao dispositivo. Estes drivers são "Stand-Alone" e gerenciam sozinhos todo o controle do hardware. Neste tipo de driver, o desenvolvedor necessita implementar todos as funcionalidades desejadas, pois a arquitetura não faz reuso das rotinas já existentes em outros drivers.

2.8.3. Linux

Módulos

Os drivers na plataforma Linux são os chamados módulos do kernel. Estes módulos são arquivos que com o código do device driver, o qual pode ser ligado ou desligado do kernel durante a execução.

Os módulos têm comportamento diferente das aplicações normais. Enquanto as aplicações executam uma tarefa do início ao fim, a função "init_module", de inicialização dos módulos, registra o mesmo junto ao kernel para permitir futuras requisições e logo após termina. A tarefa da função "init_module" é preparar o módulo para receber futuras chamadas as suas funções. Ela registra no kernel quais são as funções do módulo que estão disponíveis para serem chamadas. Assim o SO define qual ou quais funções deve chamar ao receber um pedido de acesso ao dispositivo. Existe também a função "cleanup_module", a qual libera os recursos alocados para o driver durante a "init_module" e também cancela o registro no kernel das funcionalidades do driver. Esta função deve ser chamada quando se está retirando o driver da memória.

Outra diferença das aplicações, é que o módulo não está ligado à biblioteca padrão do Linux. O módulo deve usar funções definidas dentro do próprio kernel. Os módulos também rodam em ambientes diferentes das aplicações de usuário, com acesso à memória e endereçamento diferentes.

O padrão Unix distingue os dispositivos em 3 tipos. Cada módulo geralmente implementa um destes tipos. A divisão de módulos em classes diferentes não é muito rígida, pois permite que os programadores implementem drivers diferentes num mesmo código, mas está não é uma prática de programação recomendável. As três classes de drivers são:

Dispositivos de Caracter: São os dispositivos que podem ser acessados como uma *stream* de bytes (como um arquivo). Os drivers que implementam tais dispositivos são chamados drivers char. Os drivers char geralmente implementam as chamadas do sistema: Open, Close, Read e Write. As portas paralelas (/dev/lp0) e as portas seriais (/dev/ttyS0) são exemplos de dispositivos char. Estes dispositivos são acessados através do sistema de arquivos.

Dispositivos de Blocos: São dispositivos que podem conter um sistema de arquivos, como um disco ou um cartão de memória. Como os dispositivos char, também são acessados através do sistema de arquivos. Os drivers de blocos e os drivers char são muitos parecidos para os usuários, pois a principal diferença é como os dados vão ser gerenciados internamente pelo kernel.

Interfaces de Rede: A interface de rede é responsável de enviar e receber pacotes de dados, encaminhados pelo subsistem de rede do kernel. Estas interfaces não são orientadas à *stream*. Por este motivo a comunicação com tais dispositivos não é feita através do sistema de arquivos como os outros drivers, pois o kernel provê funções relacionadas à transmissão de pacotes.

Existem algumas outras classes de módulos de drivers no Linux, que exploram serviços públicos do kernel para lidar com tipos específicos de dispositivos. Alguns destes dispositivos são: USB, SCSI, FireWire e I2O.

3. FERRAMENTAS PARA DESENVOLVIMENTO

3.1. Windows Driver Development Kit (DDK)

O Windows Driver Development Kit é a solução oficial da Microsoft para o desenvolvimento de device drivers para os sistemas operacionais Windows. O DDK é um conjunto de ferramentas, exemplos, documentação e ambientes que dão suporte ao desenvolvimento de diferentes device drivers.

Infelizmente o DDK não é disponibilizada de forma gratuita. É possível consegui-la encomendando CD-ROM diretamente na home-page da Microsoft Windows Hardware and Driver Central ou então fazer o download caso seja associado ao programa de assinantes da Rede de Desenvolvedores da Microsoft (MSDN).

A documentação do DDK é uma das mais completas fontes sobre desenvolvimento de device drivers existente hoje em dia. Ela inclui informação necessária para se desenvolver, testar, e validar device drivers para a plataforma Windows. Dentre os tópicos abordados estão:

- Instruções para a utilização das ferramentas e dos ambientes de desenvolvimento: A DDK provê informação detalhada sobre utilização, configuração, exemplos e dicas de utilização para cada uma das ferramentas disponíveis no kit.
- Informações detalhadas sobre a arquitetura Windows kernel mode driver. Esta seção é um guia para o desenvolvimento de drivers WDM, ela abrange tópicos como introdução, tipos de drivers WDM, exemplos, além de explicar em detalhes como programar um driver WDM. Esta seção também provê *guidelines* e técnicas de programação genéricas recomendadas pela Microsoft para o desenvolvimento de drivers.
- Uma visão geral do funcionamento e arquitetura dos sistemas Windows, do funcionamento básico de um device driver e conceitos básicos de sistemas operacionais.

- Informação específica sobre como escrever drivers para classes de dispositivos definidas pelo Windows. Para cada classe de dispositivo definida, esta seção contém um guia de planejamento de drivers para a classe, informações básicas sobre as características dos dispositivos da classe, exemplos, guias para implementação de funcionalidades específicas. Também possui subseções com tópicos sobre instalação, depuração, teste e referências dos drivers de cada classe.
- Informações gerais sobre planejamento, construção, depuração, validação, teste e instalação de drivers.

Os exemplos fornecidos pelo DDK podem ser utilizados como esqueleto na construção de device drivers e são muito úteis no durante a aprendizagem.

Em conjunto com toda a documentação, o DDK também provê ferramentas que ajudam no desenvolvimento.

Ferramentas para criação e validação de arquivos do tipo INF:

- GenINF: É uma aplicação gráfica que guia o desenvolvedor pelo processo de criar arquivos do tipo INF.
- ChkINF: É a ferramenta para verificar a sintaxe e a estrutura de arquivos tipo INF.

Ferramentas para gerenciamento de soluções e compilação:

- Build Utility: Ambiente para compilação.
- BinPlace: Gerenciador de soluções.

Ferramentas para testes e validação do driver, entre elas:

- Driver Verifier: Monitora os drivers modo kernel e verifica se o mesmo não está fazendo chamadas ilegais nem causando a deteriorização do sistema. Pode fazer também testes de estresse.
- DevCon: Exibe informações detalhadas sobre o driver e permite carregar, descarregar, instalar, remover e configurar os drivers na máquina local.
- PoolMon: Verifica as alocações de memória e permite saber se existem vazamentos de memória no driver.

- Device Path Exerciser: Testa a segurança e robustez dos drivers, chamando-o através de diversas interfaces de E/S com parâmetros válidos, inválidos e buffers mal formatados.

Ferramentas para depuração:

- WinDbg: Um debugger gráfico para modo usuário ou modo kernel.
- KD: Um programa em linha de comando para depuração modo kernel.
- CDB: Um programa em linha de comando para depuração de drivers modo usuário e programas.

A Microsoft encoraja os desenvolvedores a implementarem seus drivers utilizando a metodologia descrita pelo DDK. Através de um sistema de certificação de performance e qualidade, os testes de compatibilidade de hardware (HTC) dos laboratórios de qualidade de hardware Windows (WHQL), a empresa garante o bom funcionamento de drivers desenvolvidos por terceiros.

Além do DDK, existem alguns outros kits comerciais para desenvolvimento de drivers. Em geral alegam serem mais fáceis de utilizar e menos complexos, sem a necessidade do desenvolvedor saber detalhes do sistema operacional, nem tampouco conhecer os fundamentos da DDK. Alguns destes kits são capazes ainda de produzir device drivers para outros sistemas operacionais.

3.2. Jungo WinDriver e KernelDriver

A Jungo Software Technologies é uma empresa voltada para o desenvolvimento de ferramentas que auxiliam no desenvolvimento de device drivers.

A linha de produtos WinDriver é uma coleção de kits de desenvolvimento de device drivers em modo usuário, planejada para permitir o desenvolvimento de drivers de alta qualidade e performance. Segundo o Jungo Windriver White Paper, não é necessário o conhecimento da Windows DDK nem de detalhes do funcionamento do kernel.

O WinDriver suporta o desenvolvimento de drivers para os barramentos USB, PCI, CardBus, CompactPCI, ISA, PMC, PCI-X, PCI-104 e PMCIA. O desenvolvimento pode ser feito para as plataformas Windows 98/Me/2000/XP/Server 2003, Windows CE, Linux, Solaris e VxWorks.

Segundo a empresa, os drivers desenvolvidos poderão ser utilizados em todos os sistemas operacionais suportados, sem nenhuma mudança no código do driver.

Assim como o DDK, a metodologia de desenvolvimento de driver modo usuário Windriver é acompanhada de uma coleção de ferramentas de desenvolvimento, depuração, testes, documentação e exemplos.

Já a linha de produtos KernelDriver é a contrapartida da Jungo para desenvolvimento de device drivers em modo kernel. O KernelDriver tem características semelhantes ao WinDriver, suportando os mesmos barramentos e plataformas citadas acima. Além disto conta também com uma variedade de ferramentas, documentação e exemplos como na versão para modo usuário.

Ainda na linha de kits de desenvolvimento, vale ressaltar a existência do DriverStudio, da empresa Numega. Semelhante a DDK e aos produtos da Jungo o DriverStudio oferece uma gama de ferramentas, documentação e exemplos para auxiliar o desenvolvimento de drivers. O ponto forte deste kit debugguer SoftICE. Este é conhecido como um poderoso debugguer.

A tabela a seguir demonstra algumas diferenças entre os kits de desenvolvimento:

Empresa	Jungo	Jungo	Numega	Microsoft
Produto	WinDriver	KernelDriver	DriverStudio	DDK
Desenvolvimento	Modo Usuário	Modo Kernel	Modo Kernel	Modo Kernel
Tempo de desenvolvimento	Médio	Médio	Médio	Grande
Curva de Aprendizado	Pequena	Média	Média	Grande
Desenvolve para diversos SOs:	Sim	Sim	Não	Não
Windows	Sim	Sim	Sim	Sim
Linux	Sim	Sim	Não	Não
Solaris	Sim	Não	Não	Não
VxWorks	Sim	Não	Não	Não
Geração de Código	Sim	Sim	Sim	Não
Integração com:				
Visual C++	Sim	Sim	Sim	Não
Borland C++	Sim	Não	Não	Não
Borland Delphi	Sim	Não	Não	Não
Visual Basic	Sim	Não	Não	Não
Suporte a PCI 64 Bit	Sim	Sim	Sim	Sim
Debugger	Não	WinDbg	SoftICE	WinDbg
Diagnóstico de Hardware	Sim	Sim	Não	Não
Segue o padrão Microsoft (WHQL)	Não	Não	Não	Sim

Tabela 1: Diferença entre os Kits de Desenvolvimento.

3.3. Máquinas virtuais

Uma máquina virtual (MV) simula o comportamento de computadores reais, e é utilizada para desenvolvimento e teste de aplicações em diferentes sistemas operacionais.

Uma máquina virtual permite a simulação de diversos sistemas operacionais dentro de uma mesma máquina real, o que resulta em uma imensa capacidade para testes além de ser útil de diversas formas para o desenvolvedor.

Segundo os white papers "Microsoft Virtual PC 2004 Deployment" da Microsoft e "Accelerate Application Development, Testing, and Deployment" da VMware, a máquina virtual representa um ambiente extraordinário para o teste e a depuração de código de aplicações pois facilita a instalação e o gerenciamento de diversos sistemas operacionais na mesma máquina real e permite montar ambientes para testes com várias configurações de memória, espaço em disco, dispositivos conectados.

Outra vantagem é um amplo conhecimento do que está acontecendo com a máquina e evita o risco de danos ao sistema operacional da máquina real. Imagens instantâneas do estado, com informações detalhadas podem ser coletadas facilmente. Também é possível recuperar um estado já gravado, antes de alguma operação arriscada (muito útil quando o resultado desta operação resulta em travamento ou reinicialização do computador.)

3.4. Livros e documentação disponível

Durante anos, pouco foi pesquisado e documentado sobre device drivers. Atualmente este panorama está mudando, pois existe um maior interesse pela área e pesquisas e livros estão sendo publicados cada vez mais.

A Microsoft é uma das empresas que mais estão valorizando este mercado. Para incentivar os desenvolvedores, democratizar e padronizar o processo de desenvolvimento de driver, ela criou a Microsoft Driver Development Kit. Além disto, a Microsoft está estimulando a publicação de livros que tratam do assunto, vários deles publicados através de sua própria editora, a Microsoft Press.

Alguns títulos sobre desenvolvimento de Device Drivers para o SO Windows são:

- ONEY, W., Programming the Microsoft Windows Driver Model, Microsoft Press;
- SOLOMON, D. A.; RUSSINOVICH, M. A. , Inside Microsoft Windows 2000, Microsoft Press;
- BAKER, A.; LOZANO, J. , The Windows 2000 Device Driver Book: A Guide for Programmers, Prentice Hall;
- CANT, C., Writing Windows WDM Device Drivers: Covers Nt 4, Win 98, and Win 2000, CMP Books;

Além disto a Microsoft conta com um laboratório especializado na certificação de drivers, que garante a qualidade dos produtos, o Windows Hardware Quality Labs (WHQL).

Por outro lado, a plataforma Linux não dispõe de tantos avanços. Boa parte da documentação e detalhes do sistema operacional ainda está dentro do código fonte. Isto obriga o desenvolvedor a ser uma espécie de "hacker" para localizar a informação desejada. Felizmente, alguns livros foram publicados abordando este tema:

- RUBINI, A.; CORBET, J., Linux Device Drivers, O'Reilly
- BOVET, D. P.; CESATI, M., Understanding the Linux Kernel, O'Reilly
- POMERANTZ, O. Linux Kernel Module Programming Guide, Ebook
- BECK, M. et al. Linux Kernels Internals, Addison-Wesley

4. ESTUDO DE CASO

4.1. Motivação

Em meados de Março de 2004, um acordo foi feito entre a empresa Waytec Tecnologia em Comunicação Ltda e o Centro de Informática da UFPE, visando o desenvolvimento de Device Drivers para monitores TouchScreen resistivos, fabricados pela própria Waytec. Este convênio foi fundado seguindo os moldes da "Lei da Informática" (lei nº 10176/01), tendo projeto iniciado em Maio de 2004 e com duração até Fevereiro de 2005. Os drivers desenvolvidos deveriam rodar nos Windows 98/ME/2000/XP e no Linux Fedora Core 2, com versões para USB e Porta Serial.

4.2. Requisitos

De acordo com o documento de requisitos elaborado, os device drivers deveriam possuir características similares a outros existentes no mercado, isto é, funcionalidades semelhantes e performance igual ou superior.

O driver deveria ter um instalador automático, ferramentas para configuração e uma aplicação gráfica de calibração, que ajusta as coordenadas dos pontos tocados na malha do monitor touchscreen em pontos desenhados na tela do sistema operacional. Além disto, as seguintes funções deveriam ser implementadas:

- Habilitar/Desabilitar Toque: Esta função indica se os toques na malha resistiva do monitor devem ser mapeados como um mouse ou então apenas ignorados.
- Botão padrão e teclas para o botão secundário: O driver representaria um toque no monitor como um clique de um botão do mouse, configurado como botão padrão. O botão secundário seria representado caso o toque ocorresse junto com o pressionamento de algumas teclas escolhidas.
- Habilitar/Desabilitar Beep: Indica se o computador deve emitir um beep quando o monitor for tocado.

O driver deveria funcionar em 3 modos de operação:

- Modo emulação do mouse: Um clique vai se seguir imediatamente após o usuário tocar a tela, mas o driver enviará continuamente as informações de movimento enviadas pela controladora, até que um sinal indicando que o usuário deixou de tocar a tela seja recebido. Este modo simula a operação de um mouse, e é o único que permite operações como arrastar-e-soltar.
- Modo Ontouch: Um sinal indicando que o botão foi pressionado e logo em seguida liberado é enviado pelo driver ao sistema operacional assim que o usuário toca na malha resistiva. As informações recebidas em seguida pelo driver, devem ser ignoradas até que o usuário pare de tocar na tela.
- Modo OnUntouch: Parecido com o modo Ontouch, mas o driver só enviará o sinal de toque e de liberação assim que o usuário parar de tocar a tela.

Além destes requisitos pedidos pela Waytec, alguns outros requisitos desejáveis foram adicionados pela equipe de desenvolvimento:

- Grande reutilização de código entre os diversos drivers;
- Núcleo do driver independente de plataforma;
- Fácil adaptação do código a outros protocolos;

Estes requisitos tinham em mente a viabilização da re-utilização do grande parte do código para desenvolver drivers para outras controladoras de monitores touchscreen, similares às utilizadas, mas com pequenas mudanças no protocolo utilizado.

4.3. Soluções Encontradas

Por ser um projeto pioneiro para os desenvolvedores, muita pesquisa teve que ser feita antes da fase de planejamento. Apesar disto, várias decisões tiveram que ser modificadas durante o projeto, de acordo com o aumento do conhecimento do problema e por dificuldades de implementação.

Para atender aos diversos requisitos citados acima, utilizamos estas soluções:

- A utilização da linguagem de programação C++. Esta linguagem foi escolhida pois é muito conceituada pela sua ótima performance, pela facilidade de integração com as rotinas do SO e com os drivers modo kernel, que são escritos em C/C++.
- Uma arquitetura híbrida, sendo uma parte do driver em modo usuário e a outra em modo kernel. Esta arquitetura foi utilizada pois o driver precisava se comunicar com uma aplicação gráfica para a calibração e configuração. Além disto, o driver modo kernel dos modelos Linux e Windows são bastante diferentes, e por isso esta parte deveria ser escrita especificamente para cada SO.
- O acesso a funções específicas do SO e das portas de comunicação e feito através de interfaces bem definidas, onde a implementação é dependente de plataforma.
- Os SOs Windows e Linux já possuem um driver específico para o envio e recebimento de dados via Porta Serial. Para minimizar o esforço de implementação, a arquitetura foi planejada para utilizar estes drivers já existentes. Logo, os drivers modo kernel USB foram desenvolvidos com interfaces semelhantes às já existentes da Porta Serial, dando ao resto do sistema uma transparência na forma como o driver se comunica com a controladora.

4.4. Arquitetura Utilizada

A arquitetura encontrada é o resultado de um planejamento prévio para atender aos requisitos do projeto, mas com algumas modificações que foram feitas durante a fase de implementação.

A núcleo do driver é a classe CController. É ela quem coordena todas as funcionalidades principais, é responsável pela inicialização do dispositivo, recupera as informações de configuração do repositório, realiza os ajustes no modo de operação, responde aos pedidos feitos pela aplicação gráfica e controla todos os eventos enviados ao sistema operacional. Outra classe de grande importância é a CProtocol, a qual faz toda a codificação e decodificação do protocolo utilizado pela controladora do dispositivo. Desta forma, caso a controladora mude o protocolo utilizado para algum outro similar, só séria necessário reescrever esta classe, isoladamente.

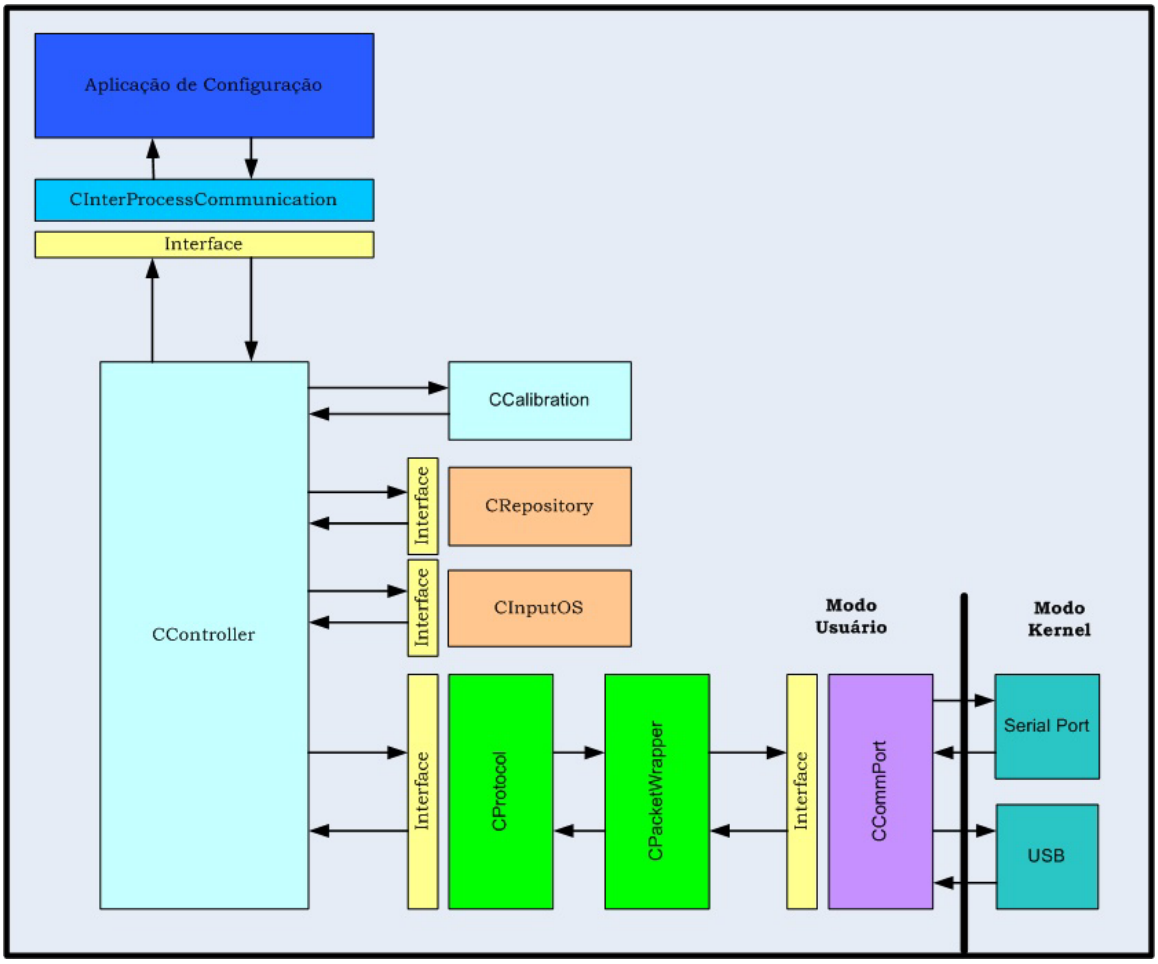

Figura 10: Arquitetura do device driver desenvolvido.

A CController e a CProtocol são as duas maiores classes do driver. Elas foram escritas utilizando apenas funções básicas de bibliotecas padrões, de forma que é possível reutilizá-las em todas as versões do driver.

Como citado anteriormente, os drivers do modo kernel que fazem a conexão com a porta serial já estão implementados tanto na plataforma Windows como no Linux. Estes drivers foram utilizados ao invés de implementar versões próprias, pois são eficientes o suficiente para a aplicação desejada. Por esta razão a os drivers modo kernel da versão USB foram construídos para que funcionassem de forma similar aos drivers de porta serial existentes. Desta forma, poucas mudanças foram necessárias para ajustar o funcionamento do driver da versão serial para a versão USB, e apenas a classe que implementava os detalhes desta comunicação foi alterada.

Como cada SO tem detalhes diferentes quanto ao acesso a arquivos e aos dispositivos de E/S, diversas interfaces foram utilizadas, permitindo assim que apenas as classes mais específicas, as quais fazem a conexão com o hardware ou com SO, tiveram que ser reescritas para cada versão. Estas classes significam uma pequena parte do código. Com a utilização das interfaces, foi possível a utilização de tecnologias diferentes para as mesmas tarefas, como por exemplo, o armazenamento de dados de configuração, a interface com o gerenciador de janelas e a interface com o dispositivo.

A aplicação de gráfica de configuração comunica-se com o device driver através de uma classe que implementa comunicação entre-processos. Esta comunicação só existe nas versões Windows, pois as versões Linux não possuem tal aplicação. A personalização do driver é feita através de arquivos de configuração editados pelo usuário.

4.5. Resultado

Seguindo estas decisões, foi possível implementar um código com uma grande taxa de reutilização. O projeto ficou pronto dentro do prazo, e todas as metas citadas acima, foram alcançadas. Foram implementadas 4 versões do driver:: Windows Serial, Windows USB, Linux Serial e Linux USB

A tabela a seguir mostra algumas pequenas diferenças entre os drivers:

	Windows Serial	Windows USB	Linux Serial	Linux USB
Porcentagem aproximada de código aproveitado	100%	95%	85%	85%
Interface com o dispositivo	Serial	USB	Serial	USB
Armazenamento da configuração	Registro do Windows	Registro do Windows	Arquivo de configuração	Arquivo de configuração
Driver modo kernel	Padrão do Windows	USB	Padrão do Linux	USB
Interface com o gerenciador de janelas	Win32 API	Win32 API	Servidor X	Servidor X

Tabela 2: Diferenças entre os Drivers.

5. CONCLUSÃO E TRABALHOS FUTUROS

5.1. Conclusão

O objetivo deste trabalho foi dar uma noção geral sobre device drivers, analisar o seu processo de desenvolvimento e a metodologia existente. Para isto, foi apresentado o contexto onde os device drivers se inserem, sua história, suas características, suas diferenças e as tecnologias empregadas. Depois desta introdução ao contexto, chegamos a analisar as ferramentas existentes para auxiliar o desenvolvedor, incluindo também a documentação disponível. Finalmente, chegamos ao estudo de caso de um driver desenvolvido para as plataformas Windows e Linux.

Foi concluído então, que o processo de desenvolvimento de device drivers está muito mais evoluído, documentado e organizado do que há uma década. Empresas como a Microsoft estão estimulando o crescimento organizado da área, provendo ferramentas, kits de desenvolvimento, documentação e até um laboratório de certificação de drivers.

Em relação à plataforma Linux, muito ainda se tem que fazer. Falta uma padronização maior, kits de desenvolvimento, ferramentas e uma melhor documentação.

Já o caso de uso, mostrou algumas das decisões tomadas durante o processo de desenvolvimento, de acordo com as dificuldades encontradas no projeto e na implementação. Ainda demonstrou um exemplo interessante de arquitetura que foi utilizada para ambas plataformas, Linux e Windows, com uma alta taxa de reaproveitamento de código.

A tabela a seguir demonstra as diferenças encontradas no suporte oferecido ao desenvolvimento por ambas as plataformas:

Plataforma	Windows	Linux
Tecnologia	WDM	Módulo do Kernel
Ferramentas Específicas	Sim, DDK	Não
Documentação	Boa	Pouca
Livros	Vários	Médio
Padronização	Sim	Pouca
Certificação	Sim	Não
Kits de Desenvolvimento	Sim	Sim

Tabela 3: Suporte ao desenvolvimento.

5.2. Trabalhos Futuros

Entre as propostas para trabalhos futuros, proponho o desenvolvimento de uma metodologia para desenvolvimento de device drivers na plataforma Linux, implementação de ferramentas e uma documentação mais precisa dos detalhes do funcionamento do kernel do Linux. Também seria possível uma proposta de arquiteturas semelhantes àquela vista no caso de estudo, que permitam uma grande reutilização do código de drivers em plataformas diferentes.

6. REFERÊNCIAS

ONEY, W., Programming the Microsoft Windows Driver Model, 2ª Edição, Microsoft Press, 2002.

RUBINI, A.; CORBET, J., Linux Device Drivers, 2ª Edição, O'Reilly, 2001.

TANENBAUM, A. S., Modern Operating Systems, 2ª Edição, Prentice Hall, 2001.

SOLOMON, D. A.; RUSSINOVICH, M. A., Inside Microsoft Windows 2000, 3ª Edição, Microsoft Press, 2000.

BOVET, D. P; CESATI, M. Understanding the Linux Kernel, 2ª Edição, O'Reilly, 2002.

BAKER, A.; LOZANO, J., The Windows 2000 Device Driver Book: A Guide for Programmers,3ª Edição, Prentice Hall, 2000.

CANT, C., Writing Windows WDM Device Drivers: Covers Nt 4, Win 98, and Win 2000, CMP Books, 1999;

POMERANTZ, O. Linux Kernel Module Programming Guide, Ebook, 1999.

BECK, M. et al. Linux Kernels Internals, 2ª Edição ,Addison-Wesley, 1997

MICROSOFT Windows DDK Documentation.

CANT, C., WDM Article. Disponível em: (http://www.phdcc.com/WDMarticle.html) Último acesso em: 07/Mar/2005.

VMWARE; Accelerate Application Development, Testing, and Deployment. Disponível em: (http://www.vmware.com/pdf/dev_test.pdf) Último acesso em: 05/Mar/2005.

MICROSOFT Microsoft Virtual PC 2004 Deployment

Disponível em: (http://download.microsoft.com/download/8/7/6/876af3ca-070a-4846-9b19-bd0389b575fa/Virtual PC 2004 Deployment.doc) Último acesso em: 05/Mar/2005.

JUNGO; WinDriver WhitePaper.

Disponível em: (http://www.jungo.com/support/documentation/wd_wp.pdf)

Último acesso em: 27/Fev/2005.

JUNGO; KernelDriver WhitePaper

Disponível em: (http://www.jungo.com/support/documentation/kd_white_paper.pdf)

Último acesso em: 27/Fev/2005.

Site do Open Systems Resources, Inc. Disponível em: (http://www.osronline.com) Último acesso em: 09/Mar/2005.

GOOGLE, (http://www.google.com).