Alexandre Heitor Schmidt

ESTUDO DO DESENVOLVIMENTO DE DRIVERS DE DISPOSITIVOS USB PARA O SISTEMA OPERACIONAL LINUX

Lajeado, dezembro de 2007

Alexandre Heitor Schmidt

ESTUDO DO DESENVOLVIMENTO DE DRIVERS DE DISPOSITIVOS USB PARA O SISTEMA OPERACIONAL LINUX

Trabalho de Conclusão de Curso apresentado ao Centro Universitário UNIVATES para obtenção do grau de Bacharel em Engenharia da Computação.

Orientador: Maglan Cristiano Diemer

Lajeado, dezembro de 2007

DEDICATÓRIA

À Palavra.

AGRADECIMENTOS

Ao meu orientador Maglan Cristiano Diemer e ao professor Ronaldo Hüsemann, pela ajuda na escolha do tema, grande disponibilidade e companheirismo.

À minha esposa Camila, pela paciência e apoio incondicionais. Sem ti, jamais teria conseguido.

Aos meus pais, Heitor e Margarete, pelo incentivo e pela infra-estrutura sempre à disposição. Aos meus irmãos Mauro e Leonardo pelas discussões esclarecedoras sobre as normas ABNT *versus* ABNT-Univates.

À família Spiekermann, por compreenderem de forma tão carinhosa minhas ausências.

Aos meus amigos, por terem continuado meus amigos mesmo após longos períodos sem contato, e aos meus colegas formandos de Engenharia da Computação, pela corrente positiva que nos possibilitou chegar até aqui. TEM QUE SER!

RESUMO

Este documento tem o objetivo principal de ser uma referência sobre o desenvolvimento de gerenciadores de dispositivos USB para o *kernel* do sistema operacional Linux. O trabalho traz uma breve introdução sobre a interface USB e suas vantagens e aborda de forma detalhada os diversos conhecimentos necessários para o desenvolvimento de gerenciadores de dispositivos USB no ambiente Linux. Como estudo de caso, apresenta-se o código fonte do módulo *xpad*, um dos gerenciadores de dispositivos do tipo *joystick* USB do *kernel*, com o objetivo de explicar, através de um exemplo prático real, todos passos envolvidos no desenvolvimento deste *driver*.

Palavras-chave: linux*, kernel,* módulos, *drivers,* USB, gerenciadores de dispositivos, xpad, *joystick*, *Human Interface Devices,* subsistema Input, USB *core*.

ABSTRACT

This paper has the main goal of serving as a reference on USB device drivers development for the Linux Operating System's kernel. The paper brings a brief introduction about the USB interface and its advantages and details the necessary knowledge for Linux kernel USB drivers development. As a use case, the source code of the xpad module, one of the Linux USB joystick device drivers, is shown in order to explain by using a real practical example, all the steps involved in the development of such driver.

Keywords: *linux, kernel, modules, device drivers, USB, joystick, Input subsystem Human Interface Devices, USB core.*

SUMÁRIO

RELAÇÃO DE FIGURAS

RELAÇÃO DE LISTAGENS

1 INTRODUÇÃO

No início dos anos 90, o sistema operacional Linux começou a se tornar popular, mas ainda era utilizado por um número reduzido de pessoas que eram, em sua maioria, *hackers* dispostos a gastar seu tempo livre experimentando novidades. Nem mesmo o próprio criador do sistema operacional, Linus Torvalds, acreditava que sua criação tinha um potencial tão grande quando divulgou seu código fonte na Internet (HASAN, 2005).

Conforme o sistema operacional foi se desenvolvendo e se tornando mais maduro, foram surgindo necessidades básicas para que pudesse ser utilizado por usuários comuns, uma vez que a interface até então era puramente textual. Nesta época, surgiram compiladores, editores de texto e diversas outras aplicações que contribuíram para que o desenvolvimento de rotinas internas do sistema, como o próprio *kernel* e os gerenciadores de dispositivos periféricos (ou simplesmente *drivers*), fosse ficando para trás. A carência de gerenciadores de dispositivos acabou por se tornar um dos grandes empecilhos para uma utilização em maior escala do sistema operacional Linux (OLIVEIRA, CARISSIMI e TOSCANI, 2001).

Wirzenius (2003) afirma que os motivos que impediam a utilização de alguns dispositivos no Linux resumiam-se, na maioria dos casos, ao fato de o fabricante não implementar um *driver* compatível ou não fornecer as especificações de seu

dispositivo para que a própria comunidade pudesse desenvolver um *driver*.

Além destas razões, a falta de padronização na forma de comunicação dos periféricos com o computador fazia com que cada dispositivo necessitasse de um *driver* específico para controlá-lo, impossibilitando a utilização de *drivers* genéricos, isto é, que pudessem ser utilizados para controlar mais de um dispositivo.

Este foi um dos motivos pelos quais, em 1994, empresas que detinham grande parte do mercado de periféricos - entre elas a Compaq, Hewlett-Packard, Intel, Microsoft e Philips - reuniram-se para formular e, em 1996, tornar pública a primeira versão do padrão de comunicação USB (*Universal Serial Bus*), que tinha, entre outros objetivos, o de substituir as já obsoletas portas seriais e paralelas dos computadores por um padrão realmente consistente. Devido ao grande sucesso da primeira versão da especificação USB, foi divulgado, em 2000, o padrão USB 2.0, que, desde então, vem sendo adotado por um número cada vez maior de fabricantes de periféricos (USB IMPLEMENTERS FORUM, 2000).

Estas constatações, aliadas à carência de documentações sobre o assunto escritas na língua portuguesa, serviram de motivação para a elaboração deste trabalho, que referencia os procedimentos envolvidos no desenvolvimento de *drivers* de dispositivos USB para o *kernel* do sistema operacional Linux.

Os capítulos iniciais apresentarão os diversos assuntos, cujo estudo se faz necessário para a compreensão do funcionamento de gerenciadores de dispositivos USB no sistema operacional Linux. Inicialmente, no Capítulo 2, será abordada a forma como este sistema operacional implementa o conceito de gerenciadores de dispositivos. Em seguida, no Capítulo 3, será apresentada a especificação USB e a forma como foi implementada no Linux. O Capítulo 4 apresentará a especificação HID, explicando esta subdivisão da especificação USB, sua implementação no Linux e como interage com o subsistema Input. No Capítulo 5, será dada uma breve

explicação sobre como determinados sistemas de arquivos virtuais são utilizados para estabelecer a comunicação entre o sistema operacional e as aplicações. Para finalizar a fundamentação teórica, o Capítulo 6 abordará a implementação de URBs, uma forma de comunicação complementar que visa facilitar a interação com dispositivos USB. O Capítulo 7, com o objetivo de associar teoria e prática, apresentará, através de trechos de código comentados e referências aos capítulos anteriores, o código fonte do *driver xpad* do *kernel* do Linux (apresentado na íntegra no ANEXO I), responsável pelo gerenciamento de diversos modelos de dispositivos USB do tipo j*oystick*. Por fim, no último capítulo, será apresentada uma conclusão, onde serão comentados os resultados obtidos com o desenvolvimento deste trabalho.

2 GERENCIADORES DE DISPOSITIVOS NO SISTEMA OPERACIONAL LINUX

Este capítulo fornece uma visão geral sobre o assunto principal do presente trabalho. Iniciando-se com a apresentação de termos gerais, como a definição de sistema operacional, os assuntos são gradativamente aprofundados até que possam ser apresentadas explicações específicas sobre como o sistema operacional Linux implementa o gerenciamento de dispositivos.

2.1 Sistema operacional

De acordo com Tanenbaum (2003), um sistema computacional moderno é composto por um ou mais processadores, memória principal, dispositivos de armazenamento permanente (discos rígidos), dispositivos periféricos (impressoras, *scanners*, etc.), teclado, monitor, interfaces de rede e outros dispositivos de entrada e saída. Enfim, é um sistema complexo, o que torna extremamente difícil o desenvolvimento de programas capazes de controlar todos estes componentes corretamente. Por esta razão, os computadores fazem uso de um software especial denominado sistema operacional, cujas funções principais concentram-se em

gerenciar os diferentes componentes de hardware e fornecer aos programas do usuário uma interface para interação com este hardware de forma padronizada e mais simples.

Dentre as tarefas pelas quais um sistema operacional é responsável, está o gerenciamento do acesso aos periféricos. Há diferentes formas de um sistema operacional gerenciar esta comunicação, porém a forma de gerenciamento mais comum, utilizada pela grande maioria dos sistemas operacionais, é a apresentada na Figura 1. Neste caso, sempre que um programa necessita algum tipo de operação de entrada e saída, não a solicita diretamente ao hardware, mas sim ao sistema operacional (OLIVEIRA, CARISSIMI e TOSCANI, 2001).

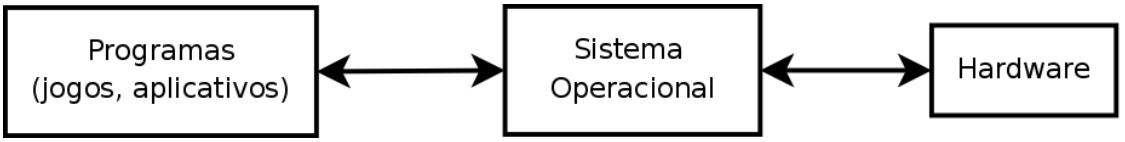

Figura 1: Interação com o hardware através do sistema operacional

Rusling (1999) salienta que o controle dos periféricos é implementado por grande parte dos sistemas operacionais modernos através da utilização de gerenciadores de dispositivos, também conhecidos como *device drivers* ou simplesmente *drivers*.

2.2 *Drivers* de dispositivos

Um *driver* é um programa responsável por traduzir as solicitações feitas pelo sistema operacional, oriundas de usuários ou não, em operações compreensíveis pelo hardware para o qual foi desenvolvido para controlar e vice-e-versa. (TANENBAUM, 2003). A Figura 2 mostra o procedimento de um programa do usuário para acessar os recursos oferecidos pelos dispositivos.

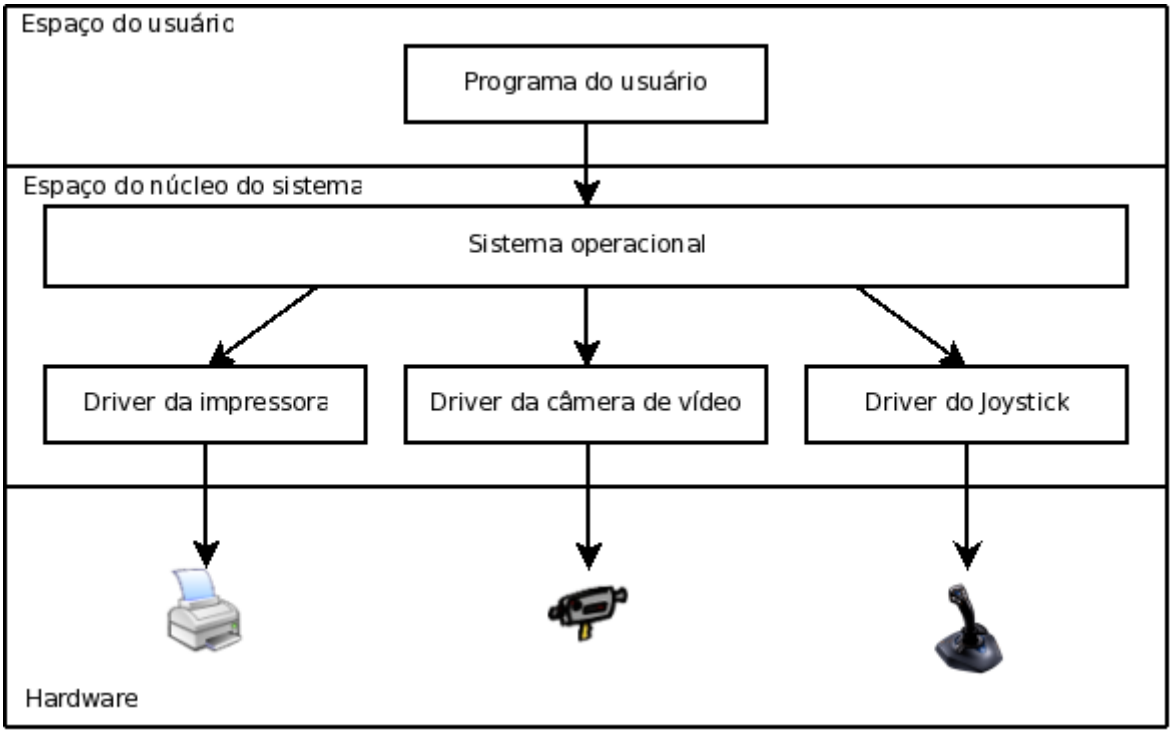

Figura 2: Forma de comunicação de um programa do usuário com o hardware

Como visto anteriormente, a utilização de *drivers* para gerenciar dispositivos de hardware é uma técnica utilizada por praticamente todos os sistemas operacionais mais conhecidos, como Microsoft Windows, MacOSX e Linux. Sendo o Linux o sistema operacional foco deste trabalho, será vista em detalhes a forma como este sistema implementa o conceito de *drivers*.

2.3 O sistema operacional Linux

Oliveira, Carissimi e Toscani (2001) ressaltam que o termo Linux foi originalmente criado em referência apenas ao núcleo do sistema - o *kernel*. Porém, atualmente este termo é utilizado para designar o sistema como um todo, incluindo o conjunto de softwares que o compõe, enquanto o núcleo do sistema passou a ser referenciado pelo nome de *linux kernel*, ou simplesmente *kernel*, que será a nomenclatura utilizada ao longo do trabalho.

O código fonte do *kernel* é distribuído sob a licença *GNU General Public License* (*GPL*) e, como conseqüência disso, pode ser obtido sem custos no *site* do projeto (http://www.kernel.org). Ao ser descompactado, o código fonte se apresenta em uma estrutura lógica de diretórios e esta estrutura é popularmente conhecida pelo nome de árvore do *kernel*.

O fato de a maior parte do código-fonte ser escrito em linguagem C padrão possibilita que o *kernel* seja multi-plataforma, isto é, possa ser compilado de forma transparente em diferentes arquiteturas. Apenas algumas rotinas específicas são escritas na linguagem *Assembly* específica de cada arquitetura e tais rotinas são disponibilizadas no diretório `/arch` da árvore do *kernel*. Assim, no momento da compilação, o *kernel* identifica a arquitetura que está sendo utilizada e, com base nesta informação, compila apenas o código correspondente à esta arquitetura (LINUX KERNEL ORGANIZATION, 2007).

Uma vez compilado, o *kernel* consiste em um grande bloco de código executável responsável por gerenciar as solicitações feitas pelos processos do sistema, como alocação do processador, memória, sistema de arquivos, rede, acesso aos dispositivos de entrada e saída, etc. (CORBET, RUBINI e KROAH-HARTMAN, 2005).

Este gerenciamento pode ser implementado utilizando-se o conceito de *microkernel* ou o conceito de *kernel* monolítico. A implementação por *microkernel* prega que o *kernel* deve conter apenas funções de sincronização, um escalonador de processos e um sistema de comunicação entre processos (*Interprocess Communication - IPC*). Esta abordagem tem a vantagem de tornar o sistema altamente modular, permitindo que somente os recursos necessários sejam carregados, e facilitar sua portabilidade para outras arquiteturas, pelo fato de concentrar todos os componentes que dependem do hardware no próprio *microkernel*. Porém, o custo de processamento para troca de mensagens entre as diferentes camadas do sistema se torna muito elevado, o que, na prática, diminui a performance geral do sistema.

Já um *kernel* monolítico consiste num único bloco de código que integra todas as camadas do sistema operacional. Neste caso, o custo de processamento é extremamente baixo, pois todo o sistema operacional roda em um único processo e isto simplifica fortemente a comunicação entre as diferentes partes do *kernel*. Porém o fato de ter todas as suas funcionalidades encapsuladas em um único programa torna a compreensão e o gerenciamento do código muito difíceis, dificultando a adição de novas funcionalidades (BOVET e CESATI, 2006).

O *kernel* do Linux é dito monolítico, mas, para que contemplasse também as vantagens de um *microkernel* sem penalizar a performance, introduziu-se o conceito de módulos (LOVE, 2005).

2.4 Módulos do *kernel* do Linux

De acordo com Bovet e Cesati (2006), um módulo é um código fonte pré-

compilado - chamado de arquivo objeto ou arquivo binário - que pode ser inserido ou removido do *kernel* em tempo de execução. O objetivo desta funcionalidade é agregar capacidades extras ao sistema sob demanda, como o acesso a determinado sistema de arquivos, à funções de criptografia ou o gerenciamento de dispositivos. Quando utilizados com a finalidade de gerenciar dispositivos periféricos, os módulos são comumente chamados de *drivers*.

2.4.1 Estrutura básica de um módulo

A Listagem 1 apresenta o código-fonte da estrutura básica de um módulo, isto é, as partes que qualquer módulo precisa implementar para que possa ser anexado ao *kernel* (CORBET, RUBINI e KROAH-HARTMAN, 2005).

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  MODULE_LICENSE("Dual BSD/GPL");
4  static int __init inicializacao_do_modulo(void)
5  {
6          printk(KERN_ALERT "Olá, mundo!\n");
7          return 0;
8  }
9  static void __exit remocao_do_modulo(void)
10 {
11         printk(KERN_ALERT "Adeus, mundo cruel!\n");
12 }
13 module_init(inicializacao_do_modulo);
14 module_exit(remocao_do_modulo);
```

Listagem 1: Estrutura básica de um módulo

É conveniente iniciar a explicação do código pelas duas últimas linhas (13 e 14). A *macro* `module_init` informa ao *kernel* qual a função a ser chamada no momento em que o módulo for carregado (no exemplo, a função

`inicializacao_do_modulo` será chamada no momento da inserção do módulo no *kernel*). Da mesma forma, a *macro* `module_exit` associa uma função ao momento da descarga do módulo (no exemplo, a função `remocao_do_modulo` será invocada).

No contexto de gerenciadores de dispositivos, a função de inicialização é responsável por detectar a presença do hardware e preparar o módulo para gerenciar tal dispositivo, fornecendo ao sistema, através de estruturas chamadas funções de registro, informações sobre parâmetros de operação e quais funcionalidades serão oferecidas pelo módulo. A função de descarga, por outro lado, é responsável por liberar os recursos previamente alocados pelo módulo na função de inicialização e é invocada no momento em que o módulo é removido do *kernel*. A carga e a descarga de módulos é realizada por meio dos utilitários `insmod`, `modprobe` e `rmmod`, que serão abordados na Seção 2.4.3.

A função de inicialização será sempre do tipo `static int`, pois deverá retornar ao sistema um valor indicando se o módulo foi carregado com sucesso e, caso não tenha sido, qual o erro que ocorreu. Já a função de remoção será sempre do tipo `static void`, não retornando nenhum valor ao sistema.

As palavras-chaves `__init` e `__exit` funcionam como otimizadores da utilização dos recursos do sistema (em especial a memória) e denotam que as funções que fazem uso delas somente serão utilizadas na inicialização ou na remoção do módulo, respectivamente. Na inicialização do sistema, todas as funções identificadas com `__init` são desalocadas da memória imediatamente após serem executadas, visando aumentar a disponibilidade de recursos. O mesmo ocorre após a carga de um módulo com as funções marcadas com este identificador. O identificador `__exit` é utilizado nos casos em que o *kernel* é compilado sem suporte à remoção de módulos ou quando um módulo é compilado de forma *built-in*, isto é, integrado ao *kernel*. Nestes casos, ao encontrar funções que contenham este

identificador, o *kernel* simplesmente as descarta, uma vez que nunca serão utilizadas.

A *macro* `MODULE_LICENSE` define o tipo de licença utilizada pelo módulo. Caso não utilize uma licença de software compatível com a GPL, o *kernel* exibirá mensagens de advertência no momento em que o módulo for carregado, informando ao usuário que o módulo utilizado pode estar ferindo a licença de uso do *kernel*. A lista de licenças compatíveis pode ser vista no arquivo `/include/linux/license.h` da árvore do *kernel*.

O módulo exemplo da Listagem 1, tanto no momento de sua inicialização quanto no momento da remoção, faz chamadas à função `printk`. Esta é a versão do *kernel* para a função `printf` da biblioteca C padrão e, apesar de semelhante, tem algumas peculiaridades, a exemplo do prefixo `KERN_ALERT`, que define a prioridade de exibição da mensagem. Além disso, a função `printk` é uma versão otimizada da função `printf`, uma vez que o *kernel* não necessita implementar todas as funcionalidades providas por esta função.

O *kernel* implementa suas próprias versões de funções equivalentes às da biblioteca C padrão ao invés de simplesmente incluir esta biblioteca. Há duas razões principais para que esta abordagem seja utilizada. Uma delas é o fato do *kernel* residir em uma região de memória reservada que se comporta de maneira distinta da memória do restante do sistema, o que torna muitas funções da biblioteca padrão incompatíveis. A outra razão está relacionada ao tamanho da biblioteca C padrão. A inclusão dela demandaria a alocação de uma quantidade de memória extra muito grande, o que constituiria um desperdício de memória importante para manter as funções vitais e o desempenho otimizado do núcleo do sistema.

É importante notar que a função `printk` é invocada pelo módulo sem a necessidade da inclusão de algum arquivo que a declare. Isto é possível porque o

módulo, quando carregado, passa a fazer parte do *kernel* e, como conseqüência, tem acesso à qualquer funcionalidade que tenha sido registrada por ele ou por seus módulos. Porém, para que o módulo possa fazer uso destas funcionalidades, é necessário um mecanismo que possibilite a ele descobrir de alguma forma o endereço de memória onde a funcionalidade reside. Isto se torna possível através de uma estrutura chamada tabela de símbolos. A tabela de símbolos atual do sistema pode ser consultada no diretório `/proc/kallsyms` e consiste de uma lista contendo todos os recursos (e respectivos endereços) providos pelo *kernel*.

Por fim, os arquivos `linux/init.h` e `linux/module.h`, incluídos no início do código da Listagem 1, contêm as declarações das *macros* `module_init`, `module_exit`, `MODULE_LICENSE` e diversas outras cujas funcionalidades podem ser consultadas nos próprios arquivos incluídos, presentes na árvore do *kernel*, no subdiretório `/include/linux`.

2.4.2 Tratamento de erros em módulos

Em qualquer programa, deve-se sempre prever possíveis erros no decorrer da execução e tratá-los de forma adequada quando ocorrerem. Em se tratando de desenvolvimento para o *kernel* do sistema operacional, este cuidado deve ser redobrado, uma vez que um erro de execução em um módulo pode afetar outras partes do sistema ou até mesmo deixar o sistema todo em uma condição instável.

A forma mais comum de se tratar erros em módulos é a utilização do comando `goto`. Muitos programadores consideram o uso deste comando inadequado por tornar o código fonte desestruturado, mas ele se mostra muito útil em módulos por facilitar a programação, evitando a identação excessiva.

Conforme pode-se observar na Listagem 2, na inicialização, o módulo tenta registrar suas funcionalidades e, caso ocorra algum problema em uma das tentativas, o fluxo é desviado para o identificador correspondente, o qual tem a responsabilidade de desfazer tudo o que já tiver sido feito até o momento. Desta forma, garante-se que tudo o que foi alocado pelo módulo será desalocado. Cabe lembrar que a ordem de desalocação de recursos deve, na maioria dos casos, ser feita de forma inversa à ordem de alocação, uma vez que podem existir recursos interdependentes.

```
1  static int __init inicializacao(void)
2  {
3      int err;
4
5      /* a fase de registro sempre recebe um
6          ponteiro e um identificador */
7      err = registrar(ptr1, "teste");
8      if ( err ) goto falha_aqui;
9      err = registrar(ptr2, "teste");
10     if ( err ) goto falha_la;
11     err = registrar(ptr3, "teste");
12     if ( err ) goto falha_ali;
13
14     return 0; /* tudo ok */
15
16   falha_ali: cancelar_registro(ptr2, "teste");
17   falha_la: cancelar_registro(ptr1, "teste");
18   falha_aqui: return err; /* propagar o erro */
19 }
```

Listagem 2: Exemplo fictício do tratamento de erros utilizando `goto`

Há ainda outra forma de tratamento de erros, que consiste em manter um histórico de tudo o que foi registrado com sucesso e, caso ocorra algum erro, chamar uma função responsável por desalocar o que tenha sido previamente alocado. Esta solução, no entanto, usualmente requer mais recursos (tanto de processamento quanto de memória) e, por esta razão, não costuma ser utilizada.

A função de inicialização de um módulo sempre retornará um inteiro que poderá assumir o valor `0` (caso nenhum erro tenha ocorrido) ou outro valor

representando um código de erro pertencente às definições declaradas no arquivo `/include/linux/errno.h` da árvore do *kernel*.

2.4.3 Compilação, carga e descarga de um módulo

Conforme visto na Listagem 1, alguns arquivos precisam ser incluídos para que o módulo possa ser compilado. Estes arquivos encontram-se na árvore do *kernel* e, por esta razão, para compilar um módulo, ainda que simples como o apresentado, é necessário dispor de uma versão completa desta árvore[1]. A compilação do módulo pode ser feita sem a necessidade de privilégios especiais, porém apenas superusuários poderão adicionar e remover módulos do *kernel*.

Para realizar a sua própria recompilação e a compilação dos módulos que já vêem presentes na sua árvore de diretórios (no diretório `/drivers`), o *kernel* faz uso da sintaxe do programa *GNU make*, uma ferramenta utilizada para controlar a geração de executáveis a partir dos arquivos fontes de um programa (FREE SOFTWARE FOUNDATION, INC., 2006).

A sintaxe utilizada pelo *kernel* para os chamados *Makefiles* difere um pouco da normalmente utilizada por programas de usuários e aconselha-se a consulta à documentação do *GNU make* para maiores informações.

A utilização de *Makefiles* facilita muito a compilação dos módulos, evitando a necessidade de utilizar comandos complexos a cada vez que se deseja compilá-los.

[1] Na realidade, não é necessário ter um *kernel* completo, mas, devido ao seu tamanho e conseqüente complexidade, é mais conveniente que se tenha uma versão completa disponível ao invés de enfrentar a árdua tarefa de selecionar apenas as partes que seriam realmente necessárias.

A Listagem 3 mostra um exemplo de `Makefile` genérico com o qual é possível compilar um módulo através de uma simples invocação do comando `make`.

```
1 ifneq ($(KERNELRELEASE),)
2       obj-m := olamundo.o
3 else
4       KERNELDIR ?= /lib/modules/$(shell uname -r)/build
5       PWD := $(shell pwd)
6 default:
7       $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
8 endif
```

Listagem 3: Makefile genérico para compilação de módulos

Inicialmente, a variável ambiente `KERNELRELEASE` será testada. Se possuir conteúdo, significa que o `make` foi invocado do próprio sistema de compilação do *kernel*. Isto ocorre, por exemplo, quando o usuário entra com o comando `make modules` para compilar os módulos que já se encontram na árvore do *kernel* por padrão. Se `KERNELRELEASE` não estiver definida, significa que o `make` foi chamado diretamente da linha de comando e, portanto, o sistema de compilação do *kernel* deve ser invocado.

A Listagem 4 mostra a saída típica da invocação do comando `make` para o módulo exemplo, quando compilado com o *kernel* versão 2.6.22-14-generic da distribuição[2] Ubuntu 7.10, que será a versão utilizada para todos os testes efetuados no presente trabalho.

[2] Composição do núcleo (*kernel*) do Linux com uma série de aplicativos.

```
alexsmith@ruapehu:~/modules/hello$ make
make -C /lib/modules/2.6.22-14-generic/build
M=/home/alexsmith/modules/hello modules
make[1]: Entering directory `/usr/src/linux-headers-2.6.22-14-
generic'
  CC [M]  /home/alexsmith/modules/hello/olamundo.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/alexsmith/modules/hello/olamundo.mod.o
  LD [M]  /home/alexsmith/modules/hello/olamundo.ko
make[1]: Leaving directory `/usr/src/linux-headers-2.6.22-14-
generic'
```

Listagem 4: Saída típica do comando `make` *no momento em que é invocado para compilar um módulo*

Cabe salientar que este arquivo `Makefile` tentará compilar o módulo cujo código fonte esteja armazenado num arquivo de nome `olamundo.c`. Sendo assim, para compilar o módulo exemplo apresentado na Listagem 1, é necessário que seu código fonte seja salvo em um arquivo com este mesmo nome e no diretório onde residir o arquivo `Makefile`.

A etapa de compilação gerará um arquivo com o mesmo nome do módulo compilado, porém com a extensão `.ko` (abreviatura de *kernel object*). Este arquivo constitui o código objeto do módulo que pode ser inserido no *kernel* através do comando `insmod` e removido com `rmmod`.

Para a carga de módulos, o Linux dispõe ainda do comando `modprobe`, que realiza a mesma função do comando `insmod`, porém com algumas características que visam facilitar a carga de módulos. Uma destas facilidades é o fato de, ao ser invocado, o comando `modprobe` buscar automaticamente por módulos com o nome informado na árvore do *kernel*, diferentemente do comando `insmod`, que precisa receber o caminho para o módulo como parâmetro. Aconselha-se o leitor a buscar mais informações sobre o comando `modprobe` na *man-page* deste utilitário.

Ao inserir o módulo com o comando `insmod olamundo.ko`, pode-se

verificar se o mesmo foi carregado com o comando `lsmod`, que exibe todos os módulos atualmente inseridos no sistema. Além disso, o módulo exemplo, ao ser inserido ou removido (com o comando `rmmod olamundo.ko`, por exemplo), emite mensagens que são exibidas no terminal e/ou escritas nos *logs* do sistema, dependendo de como o direcionamento de mensagens estiver configurado no sistema. Se forem escritas nos *logs*, é possível visualizá-las com o comando `dmesg`.

2.4.4 Passagem de parâmetros na inicialização

Há diversas ocasiões onde a possibilidade de se passar parâmetros para os módulos se mostra bastante útil, como casos onde um mesmo *driver* é capaz de controlar mais de um dispositivo. Quando isto ocorre, a passagem de parâmetros pode ser utilizada para indicar qual dos dispositivos suportados presentes no sistema o *driver* deverá controlar. Outro exemplo que pode ser citado são os casos onde um dispositivo pode funcionar de várias formas e a configuração a ser utilizada deve ser definida através de parâmetros passados ao módulo.

Os parâmetros podem ser passados tanto pelo comando `insmod` quanto pelo comando `modprobe` e vários são os tipos de dados aceitos. A sintaxe para a passagem de parâmetros é bastante simples:

```
insmod olamundo.ko quantidade=12 modelo="Creative"
```

Ou seja, ao ser inserido, o módulo `olamundo.ko` receberá um parâmetro do tipo inteiro chamado `quantidade` que conterá o valor `12` e outro parâmetro do tipo texto chamado `modelo` que conterá o valor "`Creative`".

Porém é necessário que os módulos informem em seus códigos os nomes dos parâmetros que poderão receber e seus respectivos tipos de dados. Desta forma, no momento da inserção, o sistema é capaz de validar os parâmetros passados, emitindo mensagens de erro caso não tenham sido declarados no módulo ou possuam tipos de dados incompatíveis com os declarados.

O trecho de código da Listagem 5 exemplifica a declaração de dois parâmetros: um inteiro chamado `quantidade` e um *string* chamado `modelo`. A declaração dos parâmetros que um módulo pode receber é feita através da chamada à *macro* `module_param(name, type, perm)` (linhas 3 e 4), declarada no arquivo `/include/linux/moduleparam.h` da árvore do *kernel*. Nesta *macro*, o primeiro parâmetro (`name`) corresponde à variável interna que receberá o valor do parâmetro de mesmo nome passado ao módulo. O segundo parâmetro (`type`) indica o tipo de dados deste parâmetro e o terceiro parâmetro (`perm`) indica a permissão de acesso a este parâmetro. Os tipos e as permissões possíveis são explicados a seguir.

```
1 static int quantidade = 10;
2 static char *modelo = "Um modelo";
3 module_param(quantidade, int, S_IRUGO);
4 module_param(modelo, charp, S_IRUGO);
```

Listagem 5: Declaração de parâmetros que poderão ser passados ao módulo no momento da carga

O trecho de código da Listagem 5 deve ser adicionado ao módulo exemplo (apresentado na Listagem 1) para que o comando `insmod` mostrado anteriormente possa ser executado.

Os tipos de dados que podem ser passados como parâmetro são os seguintes:

`bool` e `invbool`: Tipos complementares que suportam valores booleanos, isto é, `0` e `1`.

`charp`: Ponteiro para parâmetros do tipo texto. No momento da inserção do módulo, a memória necessária para o parâmetro é alocada automaticamente.

`int`, `long`, `short`, `uint`, `ulong`, `ushort`: Tipos básicos de inteiros de vários tamanhos. Os tipos que iniciam com a letra "u" são para inteiros sem sinal (*unsigned*).

É possível também a passagem de vetores como parâmetros, seguindo-se a seguinte sintaxe:

```
module_param_array(nome, tipo, num, perm);
```

Neste caso, `nome` é o identificador do parâmetro, `tipo` define o tipo dos elementos que o vetor irá conter, `num` é um inteiro representando a quantidade de valores que irão compor o vetor e `perm` representa a permissão, que será explicada a seguir.

As *macros* de declaração de parâmetros possuem um campo chamado `perm` que define a permissão de acesso. O valor informado neste campo controla os direitos de acesso ao arquivo correspondente ao parâmetro no sistema de arquivos virtual sysfs e consiste em uma máscara de bits gerada através da combinação dos valores definidos em `include/linux/stat.h`. Os valores mais comuns para `perm` são:

`0`: Indica que o sysfs não exibirá nenhuma informação sobre o parâmetro.

`S_IRUGO`: O parâmetro ficará disponível apenas para leitura para qualquer usuário no diretório padrão onde são disponibilizados os parâmetros dos módulos[3].

`S_IRUGO|S_IWUSR`: O operador "`|`" realiza uma operação lógica do tipo `OR` bit a bit entre os valores de `S_IRUGO` e `S_IWUSR`. Esta operação resulta em uma máscara de bits que indica que o parâmetro poderá ser lido por qualquer usuário e alterado apenas pelo dono do arquivo (neste caso, pelo superusuário do sistema, uma vez que os módulos só podem ser inseridos e removidos por superusuários). Cabe salientar que, se um parâmetro sofrer uma modificação, o módulo que o declarou não receberá nenhuma notificação sobre tal modificação, ficando a cargo dele a detecção de eventuais mudanças.

Na versão do *kernel* utilizada por este trabalho, o arquivo `include/linux/stat.h` não continha nenhuma descrição sobre a função de cada tipo de permissão e a lógica utilizada na nomenclatura das definições. No entanto, tais permissões são bem descritas na *man-page* do comando `stat` do *Linux Programmer's Manual*.

2.5 Dispositivos no Linux

O Linux distingue os dispositivos (e conseqüentemente os módulos que os controlam) em três tipos fundamentais: caractere (*char devices* ou *char modules),* bloco (*block devices* ou *block modules*) ou rede (*network devices* ou *network modules*). Porém esta divisão não é rígida, visto que um programador poderia, por exemplo, implementar vários *drivers* em um único módulo de tamanho maior. No

[3] Geralmente em `/sys/module`.

entanto, programadores que seguem os padrões de programação[4] estabelecidos para o *kernel*, geralmente criam um módulo diferente para cada nova funcionalidade que implementam, pois a decomposição é um elemento chave para escalabilidade e extensibilidade (CORBET, RUBINI e KROAH-HARTMAN, 2005).

A classificação dos dispositivos pode ser mais específica, diferenciando dispositivos pelo tipo de barramento, por exemplo. No entanto, como os dispositivos normalmente são exibidos no sistema segundo a classificação mencionada acima (caractere, bloco ou rede), esta também será a classificação utilizada neste trabalho.

2.5.1 Dispositivos do tipo caractere

Esta classe se refere aos dispositivos cujo acesso é similar ao acesso a um arquivo, ou seja, como se o dispositivo fosse uma seqüência de bytes. A única diferença relevante diz respeito à forma de acesso: nos arquivos, é sempre possível avançar e retroceder, enquanto que, nos dispositivos do tipo caractere, isto nem sempre é possível (como no caso de dispositivos do tipo modem). *Drivers* desenvolvidos para este tipo de dispositivo implementam, na maioria dos casos, ao menos as chamadas de sistema `open`, `close`, `read` e `write`. Como exemplos claros de módulos do tipo caractere, pode-se citar as portas seriais e o *console*. O acesso a este tipo de dispositivos é geralmente feito por meio de entradas no diretório `/dev`, tais como `/dev/tty1` ou `/dev/lp0` (CORBET, RUBINI e KROAH-HARTMAN, 2005).

[4] O principal documento que trata das regras a serem seguidas para programação em *kernel* pode ser consultado em <http://www.kernel.org/doc/Documentation/HOWTO>.

2.5.2 Dispositivos do tipo bloco

Dispositivos de bloco, da mesma forma que dispositivos de caractere, também são acessados por meio de entradas no diretório `/dev`. Este tipo de dispositivo gerencia apenas operações de entrada e saída que transferem um ou mais blocos inteiros de bytes, em geral compostos de 512, 1024, 2048 bytes ou outro valor maior. Exemplos de dispositivos deste tipo são os discos rígidos, que transferem grandes volumes de dados por vez. O Linux, porém, permite que uma aplicação acesse os dispositivos de bloco como se fossem dispositivos de caractere, sendo, portanto, possível ler ou escrever qualquer quantidade de bytes por vez. Esta característica dá ao programador a liberdade de escolha sobre a forma mais conveniente de acessar o dispositivo (CORBET, RUBINI e KROAH-HARTMAN, 2005).

Cabe salientar que a interface de um dispositivo de bloco com o *kernel* é diferente da interface de um dispositivo de caractere, uma vez que a primeira foi desenvolvida com o intuito de lidar com dispositivos que realizam transferências de grandes quantidades de dados, enquanto a segunda possui funções direcionadas à utilização de dispositivos que realizam transferências pequenas. Portanto, mesmo que se possa arbitrar sobre a abordagem utilizada na comunicação com o dispositivo, aconselha-se que a decisão sobre a interface a ser utilizada (bloco ou caractere) seja tomada com base no tipo de dispositivo que será acessado.

2.5.3 Dispositivos do tipo rede

Este tipo de dispositivo também é chamado de interface de rede e pode ser tanto implementado em hardware quanto em software, como a interface *loopback*,

normalmente acessível pelo IP 127.0.0.1. Como a função destes dispositivos é receber e enviar pacotes, eles não são mapeados no diretório `/dev`. Ao invés disso, são referenciados através de uma nomenclatura à parte (como *eth0*, *eth1*, etc.). A forma de comunicação deste tipo de dispositivo é totalmente diferente da dos dois anteriores. Ao invés de utilizar chamadas de sistema do tipo `read` e `write`, são utilizadas chamadas específicas para a transmissão/recepção de pacotes (CORBET, RUBINI e KROAH-HARTMAN, 2005).

2.6 Módulos genéricos

Como dito anteriormente, há um grande esforço para que as características de modularidade sejam preservadas no desenvolvimento de programas que interagem com o *kernel*. Neste sentido, os programadores costumam implementar módulos genéricos de suporte a dispositivos. Para cada um dos padrões SCSI, FireWire e USB, por exemplo, existe no *kernel* um módulo principal onde são disponibilizadas todas as funcionalidades especificadas pelo padrão. Os desenvolvedores de *drivers*, portanto, escrevem submódulos que fazem uso da interface oferecida pelo módulo principal para controlar os dispositivos. Para os dispositivos USB, foco deste trabalho, foi desenvolvido o USB *core* (ou subsistema USB), onde estão implementadas as funções comuns a todos os dispositivos USB. Desta forma, todo programador que queira implementar um *driver* para determinado dispositivo USB necessita implementar apenas as funções específicas daquele dispositivo, deixando o restante a cargo do subsistema (CORBET, RUBINI e KROAH-HARTMAN, 2005).

3 UNIVERSAL SERIAL BUS NO LINUX

O *Universal Serial Bus* (USB), ou Barramento Serial Universal, é um padrão de conexão entre um *host*[1] e uma série de periféricos. Conforme já citado anteriormente, o padrão foi criado inicialmente para substituir uma grande variedade de barramentos lentos (como portas seriais, por exemplo) por um único barramento ao qual qualquer tipo de dispositivo pudesse ser conectado. Inicialmente, na versão 1.1, o protocolo suportava as velocidades *low* (1.5Mbps) e *full* (12Mbps), porém sua revisão mais recente (versão 2.0) suporta conexões do tipo *high*, com velocidades que atingem, em teoria, a marca dos 480Mbps (CORBET, RUBINI e KROAH-HARTMAN, 2005).

Topologicamente, o USB é composto por um *host* mestre, que representa o elo de ligação entre os dispositivos USB e o computador. Um *hub* especial chamado *root hub* (ou *hub* raiz) é conectado a este *host* mestre com o objetivo de permitir a conexão de múltiplos dispositivos USB ao barramento. No *hub* raiz, podem ser conectados mais *hubs* ou dispositivos (também chamados de funções). Este conjunto de dispositivos interconectados é chamado de árvore de conexões USB (USB IMPLEMENTERS FORUM, 2000).

[1] Um *host* é o hardware com o qual o periférico é conectado. No caso de conexões USB, o *host* geralmente é o computador, mas existem periféricos USB capazes de se conectarem uns com os outros, sem a necessidade de um computador.

A Figura 3 mostra como as conexões USB podem ser representadas por níveis (definidos como *Tiers* na especificação USB). Conforme ilustrado, a quantidade máxima de níveis, estabelecida por razões de performance do barramento, é sete. A figura referencia ainda os dispositivos USB do tipo composto (*Compound Device*), que são dispositivos que possuem mais de uma função em um mesmo hardware (como teclados com *mouse* embutido ou *webcams* com microfone, por exemplo). Um dispositivo deste tipo só poderá funcionar se for conectado num nível de número 6 ou inferior, pois do contrário suas funções acabariam ficando num nível superior a 7, o que não é uma condição válida de acordo com a especificação. Por esta razão, o nível 7 admite apenas dispositivos do tipo função.

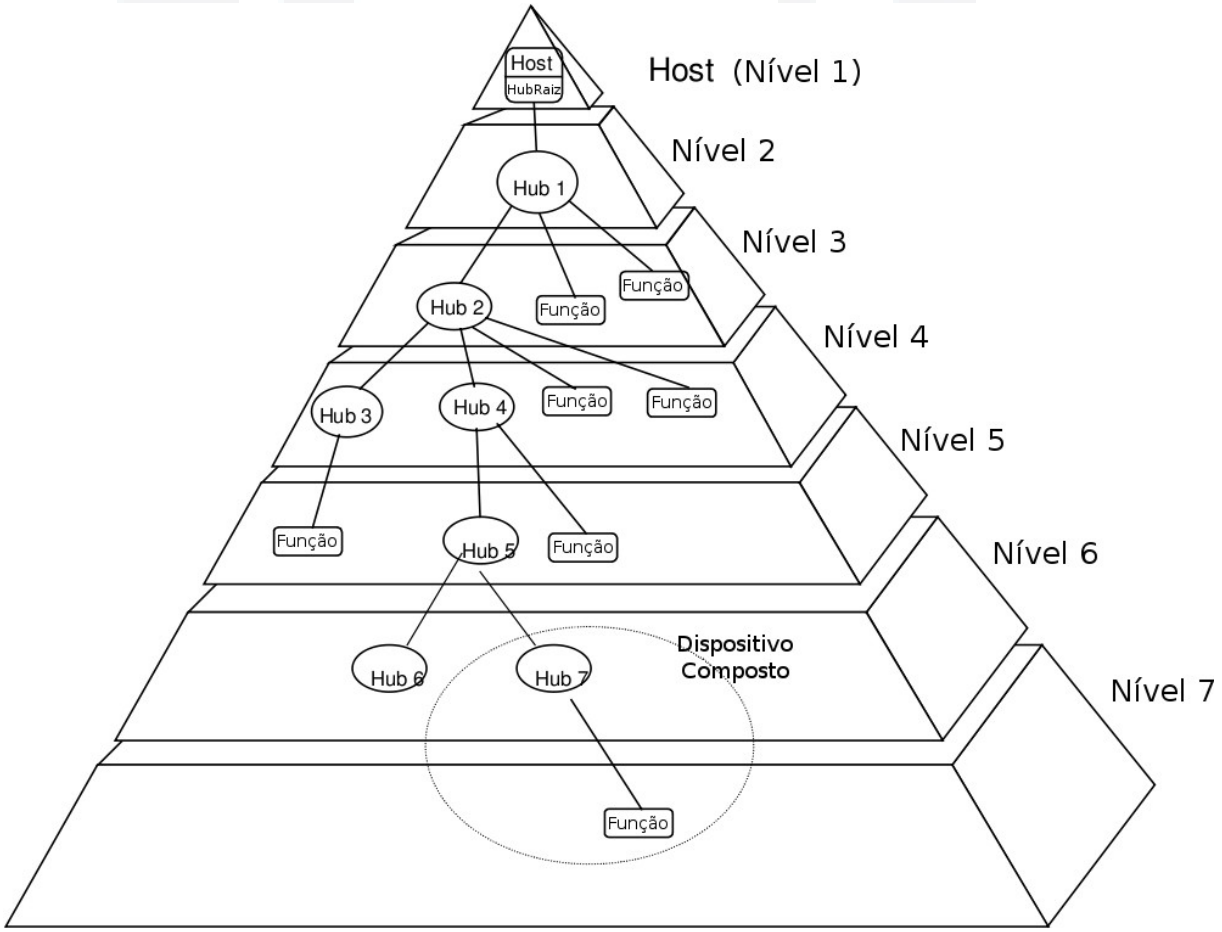

Figura 3: Topologia do Universal Serial Bus

No nível elétrico, a transferência de sinais e tensão de alimentação é feita por meio de um cabo de quatro fios chamados V_{BUS}, D+, D- e GND, conforme a Figura 4.

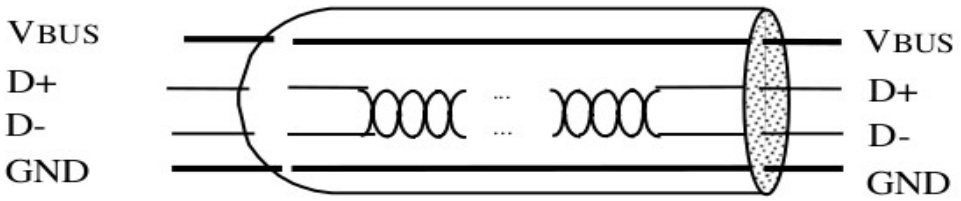

Figura 4: Diagrama representando um cabo USB

Os fios D+ e D- são responsáveis pela transmissão/recepção de dados enquanto que V_{BUS} e GND podem servir para alimentar dispositivos no barramento (V_{BUS} possui a tensão nominal de +5V no *hub* mestre).

Apesar da simplicidade de sua implementação física, o barramento USB tem algumas características muito interessantes, como a possibilidade de alocar uma largura fixa de banda para transferências de dados, provendo assim um suporte eficiente para aplicações de áudio e vídeo. Outra característica interessante é o fato do barramento USB atuar praticamente apenas como um canal de comunicação entre os dispositivos e o computador, independentemente dos dados ou estruturas que estão trafegando (CORBET, RUBINI e KROAH-HARTMAN, 2005).

Na especificação do USB, os dispositivos são classificados em classes e cada classe tem uma forma de comunicação padronizada. Armazenamento, teclado, *mouse*, placas de rede e *joysticks* são exemplos de classes para as quais existe um padrão definido de comunicação. Desta forma, um dispositivo que se encaixe em uma destas classes e se comunique segundo a especificação não necessita de um *driver* específico para funcionar. Já dispositivos de vídeo ou conversores $USB \Leftrightarrow Serial$ geralmente não se enquadram em nenhuma das classes supra-

mencionadas e necessitam de *drivers* específicos.

Todos os dispositivos USB são desenvolvidos segundo a estrutura definida na especificação. A Figura 5 ilustra as subdivisões de um dispositivo USB genérico com os *drivers* acessando tal dispositivo.

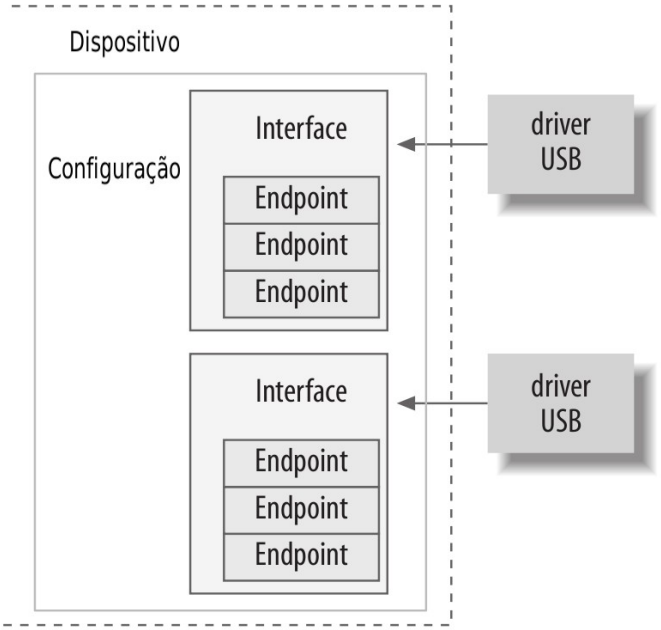

Figura 5: Drivers se conectando às interfaces do dispositivo USB

Cada uma das partes que compõem os dispositivos USB e a forma como cada uma delas foi implementada no sistema operacional Linux será vista a seguir.

3.1 Endpoints

Toda a comunicação entre um dispositivo USB e o USB *host* começa ou

termina em um *endpoint*. Cada dispositivo USB pode conter vários *endpoints*, entretanto a especificação USB exige que qualquer dispositivo contenha ao menos um *endpoint* de número zero, através do qual o dispositivo pode ser configurado (USB IMPLEMENTERS FORUM, 2000).

Quando um dispositivo USB é conectado ao sistema, cada um de seus *endpoints* recebe um endereço único através do qual pode ser acessado após configurado. O fluxo de dados entre *endpoints* e o sistema é sempre unidirecional e, por esta razão, o acesso a um *endpoint* específico é determinado pela combinação do número do *endpoint* com seu endereço e a direção do fluxo de dados, que pode ser do *endpoint* para o sistema (*IN endpoint*) ou do sistema para o *endpoint* (*OUT endpoint*). Cada dispositivo pode possuir, de acordo com a especificação USB, até 15 *endpoints* de entrada e 15 *endpoints* de saída, descontando-se o *endpoint* número zero, utilizado para configuração do dispositivo.

A transmissão de dados entre um *endpoint* e o sistema se dá por meio de uma entidade chamada *pipe*. Em suma, os *pipes* são tipos de transferência de informações e estas transferências podem ocorrer de quatro formas distintas, cada uma com suas peculiaridades e aplicações, conforme descrito a seguir.

3.1.1 Transferências do tipo CONTROL

Este tipo de transferência é utilizado para o acesso às diferentes partes do dispositivo USB com o objetivo de configurá-lo, obter informações sobre ele, enviar comandos ou saber seu estado atual. O tamanho dos pacotes do tipo CONTROL é geralmente pequeno e o barramento USB sempre mantém uma reserva de banda para garantir este tipo de transferência (CORBET, RUBINI e KROAH-HARTMAN,

2005).

3.1.2 Transferências do tipo INTERRUPT

Este tipo de transferência é utilizado para transmissão de pequenas quantidades de dados mediante a solicitação do computador ou do dispositivo em uma taxa fixa. Este é o método de transmissão utilizado por dispositivos como teclados e *mouses*. O nome deste tipo de transmissão tem sua origem no fato de ocorrer em intervalos pré-definidos. Em alguns casos, este tipo de transmissão também é utilizado para controlar o dispositivo. Para transferências do tipo INTERRUPT, o protocolo USB também sempre garante que haverá banda disponível (CORBET, RUBINI e KROAH-HARTMAN, 2005).

3.1.3 Transferências do tipo BULK

Transferências do tipo BULK são utilizadas quando a quantidade de dados a ser transmitida é muito grande (diversas vezes maior do que transferências do tipo INTERRUPT) e não pode haver perda de informações. O protocolo USB não garante que a transferência ocorrerá em um tempo específico e, caso o barramento não disponha de banda suficiente para transmitir o pacote inteiro, o mesmo é enviado/recebido em partes. Impressoras, dispositivos de rede e dispositivos de armazenamento costumam utilizar este tipo de comunicação (CORBET, RUBINI e KROAH-HARTMAN, 2005).

3.1.4 Transferências do tipo ISOCHRONOUS

Dispositivos que necessitam transferir grandes quantidades de dados através de um fluxo constante, mas admitem que nem todos os pacotes sejam entregues (como acontece com câmeras de vídeo ou dispositivos de áudio) utilizam este tipo de transferência (CORBET, RUBINI e KROAH-HARTMAN, 2005).

Transferências do tipo ISOCHRONOUS e INTERRUPT ocorrem de forma periódica (o intervalo entre as transmissões é configurável) e, para que isso seja possível, o USB *core* reserva a largura de banda necessária. Já os tipos CONTROL e BULK são utilizados para transmissões assíncronas, isto é, sempre que o *driver* decide utilizá-los (CORBET, RUBINI e KROAH-HARTMAN, 2005).

3.2 Implementação dos *endpoints*

No *kernel* do Linux, os *endpoints* são acessados através da estrutura `struct usb_host_endpoint` (definida no arquivo `/include/linux/usb.h` da árvore do *kernel*), a qual contém uma subestrutura chamada `struct usb_endpoint_descriptor` (definida no arquivo `/include/linux/usb/ch9.h`[2] da árvore do *kernel*) que aponta para as informações reais do *endpoint*, exatamente no formato especificado pelo dispositivo.

A Listagem 6 mostra esta estrutura como se encontra declarada no arquivo

[2] O arquivo `/include/linux/usb/ch9.h` possui este nome pelo fato de referenciar o Capítulo 9 da especificação USB, que define o *framework* de comunicação com dispositivos USB. Com o objetivo de manter coerência com a especificação USB, as estruturas contidas neste arquivo seguem o padrão da especificação ao invés de seguirem os padrões definidos pelo *kernel*.

/include/linux/usb/ch9.h, porém apenas os campos de interesse para este trabalho são apresentados a seguir.

```
313 struct usb_endpoint_descriptor {
314       __u8  bLength;
315       __u8  bDescriptorType;
316
317       __u8  bEndpointAddress;
318       __u8  bmAttributes;
319       __le16 wMaxPacketSize;
320       __u8  bInterval;
321
322       /* NOTE:  these two are _only_ in audio endpoints. */
323       /* use USB_DT_ENDPOINT*_SIZE in bLength, not sizeof. */
324       __u8  bRefresh;
325       __u8  bSynchAddress;
326 } __attribute__ ((packed));
```

Listagem 6: Estrutura usb_endpoint_descriptor, declarada no arquivo /include/linux/usb/ch9.h

bEndpointAddress: Contém o endereço do *endpoint* e um indicador da direção da transmissão. Dois parâmetros de definição (USB_DIR_OUT e USB_DIR_IN) podem ser utilizados para determinar esta direção através de uma operação do tipo AND bit a bit com este campo.

bmAttributes: Indica o tipo de *endpoint*. Um AND bit a bit com a definição USB_ENDPOINT_XFERTYPE_MASK pode resultar em USB_ENDPOINT_XFER_ISOC, USB_ENDPOINT_XFER_BULK, USB_ENDPOINT_XFER_INT ou USB_ENDPOINT_XFER_CONTROL, indicando se a conexão é do tipo ISOCHRONOUS, BULK, INTERRUPT ou CONTROL, respectivamente.

wMaxPacketSize: Indica o tamanho máximo de cada pacote que o *endpoint* pode gerenciar. Caso a quantidade de bytes a ser transmitida seja superior à especificada neste parâmetro, os dados são divididos em pacotes de tamanho igual ou menor do que wMaxPacketSize.

`bInterval`: Indica o intervalo de tempo entre as interrupções para obtenção de dados do dispositivo. Este campo pode ser definido em unidades de microsegundos ou milisegundos, dependendo da velocidade na qual o dispositivo irá operar.

3.3 Interfaces

Os conjuntos de *endpoints* de um dispositivo USB são sempre vinculados às interfaces. Cada interface de um dispositivo é responsável por gerenciar uma, e apenas uma, conexão lógica do dispositivo, como um *mouse*, um teclado ou uma *stream* de áudio. Os dispositivos USB podem possuir mais de uma interface, como por exemplo uma *webcam* com microfone embutido, que terá uma interface para a transmissão do vídeo e outra para a transmissão do áudio. Pelo fato das interfaces representarem funcionalidades básicas, o padrão para o *kernel* do Linux é de que cada *driver* USB controle uma interface diferente. Assim, para controlar a *webcam* com microfone citada acima, seriam necessários dois *drivers*: um para a interface de áudio e outro para a interface de vídeo.

3.4 Implementação das interfaces

O USB *core* disponibiliza para cada *driver* a estrutura `struct usb_interface`, exibida na Listagem 7, e é a partir desta estrutura que o *driver* gerencia o dispositivo.

```
140 struct usb_interface {
141       /* array of alternate settings for this interface,
142        * stored in no particular order */
143       struct usb_host_interface *altsetting;
144
145       struct usb_host_interface *cur_altsetting;     /* the
currently
146                                  * active alternate setting */
147       unsigned num_altsetting;      /* number of alternate settings
*/
148
149       int minor;              /* minor number this interface is
150                                * bound to */
151       enum usb_interface_condition condition;        /* state of
binding */
152       unsigned is_active:1;         /* the interface is not
suspended */
153       unsigned needs_remote_wakeup:1;     /* driver requires remote
wakeup */
154
155       struct device dev;            /* interface specific device
info */
156       struct device *usb_dev;       /* pointer to the usb class's
device, if any */
157       int pm_usage_cnt;       /* usage counter for autosuspend */
158 };
```

Listagem 7: Estrutura usb_interface, declarada no arquivo `/include/linux/usb.h`

Os campos desta estrutura de maior interesse para o presente trabalho são abordados a seguir:

`struct usb_host_interface *altsetting`: consiste de um vetor contendo todas as configurações possíveis para a interface. Uma interface pode conter várias configurações de seus *endpoints*. Ao serem conectados, os dispositivos utilizam a configuração `0` como padrão.

`unsigned num_altsetting`: armazena a quantidade de configurações disponíveis para a interface. Em outras palavras, o tamanho do vetor de configurações possíveis.

`struct usb_host_interface *cur_altsetting`: um ponteiro para a

configuração de interface que está em uso.

3.5 Configurações

As configurações são utilizadas para permitir que um mesmo dispositivo possa operar de formas diferentes, conforme apropriado. Cada conjunto de interfaces previstas no dispositivo é encapsulado em uma configuração diferente, permitindo (ou, em alguns casos, obrigando) que um mesmo dispositivo funcione com *drivers* diferentes, dependendo da sua configuração ativa. A configuração ativa de um dispositivo não deve ser definida pelo *driver* do dispositivo, mas sim determinada automaticamente com base em informações como potência disponível no barramento (para determinar se a alimentação deverá vir de uma fonte externa ou se pode vir do próprio barramento) ou opção do usuário (o dispositivo pode ter, por exemplo, um botão que, internamente, altere a configuração atual do dispositivo, colocando-o em modo de espera) (CORBET, RUBINI e KROAH-HARTMAN, 2005).

3.6 Implementação das configurações

O Linux permite o acesso às configurações de um dispositivo através da estrutura `struct usb_host_config` e ao dispositivo como um todo através da estrutura `struct usb_device`. Raramente um *driver* necessita acessar a estrutura `struct usb_host_config` e, por esta razão, esta estrutura não será detalhada. Porém, o USB *core* exige que as informações sobre o dispositivo lhe sejam passadas no formato da estrutura `struct usb_device`, entretanto a

estrutura `struct usb_host_config` contém apenas uma lista das interfaces do dispositivo em seu campo `struct usb_interface *interface[USB_MAXINTERFACES]`. Por esta razão, é disponibilizada a função `interface_to_usbdev`, que, como o nome já diz, converte uma `struct usb_interface` em uma `struct usb_device`.

3.7 Declaração dos dispositivos suportados pelo *driver*

Um *driver* pode ser desenvolvido com o intuito de gerenciar diversos dispositivos que funcionem de forma similar, como uma linha de *mouses*, *joysticks* ou teclados. Porém, nem todos os *mouses* possuem as mesmas funcionalidades e, por esta razão, deve-se prover uma forma do *driver* especificar quais os dispositivos que é capaz de controlar. Para suprir esta necessidade, o USB *core* disponibiliza a estrutura `struct usb_driver` (declarada no arquivo `/include/linux/usb.h` da árvore do *kernel* e exibida na Listagem 8), responsável por informar ao USB *core* as funções de interface do módulo. É também esta estrutura que armazena a lista com as características que um dispositivo deve possuir para que o *driver* em questão possa controlá-lo.

```
  833 struct usb_driver {
  834       const char *name;
  835
  836       int (*probe) (struct usb_interface *intf,
  837                     const struct usb_device_id *id);
  838
  839       void (*disconnect) (struct usb_interface *intf);
  840
  841       int (*ioctl) (struct usb_interface *intf, unsigned int code,
  842                     void *buf);
  843
  844       int (*suspend) (struct usb_interface *intf, pm_message_t
message);
  845       int (*resume) (struct usb_interface *intf);
  846
  847       void (*pre_reset) (struct usb_interface *intf);
  848       void (*post_reset) (struct usb_interface *intf);
  849
  850       const struct usb_device_id *id_table;
  851
  852       struct usb_dynids dynids;
  853       struct usbdrv_wrap drvwrap;
  854       unsigned int no_dynamic_id:1;
  855       unsigned int supports_autosuspend:1;
  856 };
```

Listagem 8: Estrutura usb_driver, declarada no arquivo `/include/linux/usb.h`

Os campos mais importantes desta estrutura são comentados a seguir.

`struct module *owner`: É um apontador para o próprio *driver* e deve ser sempre inicializado com o parâmetro de definição `THIS_MODULE`, para permitir que o USB *core* consiga gerenciar o *driver*, não deixando que o mesmo seja, por exemplo, removido em um momento inoportuno.

`const char *name`: É o nome que identifica o *driver* no sistema. Este nome deve ser único, pois corresponde à entrada (arquivo) que será criada em `/sys/bus/usb/drivers` e, por esta razão, não pode já existir quando o *driver* for carregado.

`const struct usb_device_id *id_table`: Aponta para a lista de estruturas `struct usb_device_id`, na qual cada item identifica um dispositivo

suportado para que, no momento da inserção de um novo dispositivo, o USB *core* possa verificar se o *driver* pode ou não controlá-lo, comparando suas características com os dados desta tabela. A estrutura `struct usb_device_id` é vista em detalhes mais adiante.

`int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)`: Este campo define qual função do *driver* deverá ser chamada se o USB *core* achar que este *driver* pode gerenciar o novo dispositivo conectado ao sistema. O parâmetro `struct usb_interface *intf` conterá a interface do dispositivo detectado que o USB *core* acredita que o *driver* seja capaz de controlar. O parâmetro `const struct usb_device_id *id` conterá os dados utilizados pelo USB *core* para ter tomado esta decisão.

`void (*disconnect) (struct usb_interface *intf)`: Este campo define a função do *driver* a ser invocada quando o dispositivo for desconectado.

A Listagem 9 mostra como a `struct usb_driver` deve ser preenchida. Neste exemplo, o nome do *driver* é definido como "`meu_driver`" e, caso o USB *core* entenda que o novo dispositivo conectado ao sistema possa ser controlado por este *driver*, chamará sua função *probe*, definida, neste caso, como `verifica_dispositivo`. No momento em que o dispositivo não estiver mais presente no sistema, a função `desconecta_dispositivo` será chamada, conforme definido no campo `disconnect`.

```
1 static struct usb_driver meu_driver = {
2   .owner = THIS_MODULE,
3   .name = "meu_driver",
4   .id_table = minha_tabela,
5   .probe = verifica_dispositivo,
6   .disconnect = desconecta_dispositivo,
7 };
```

Listagem 9: Exemplo de definição da `struct usb_driver`

O campo `id_table` da estrutura `struct usb_driver` é composto de uma lista de estruturas do tipo `struct usb_device_id`, onde cada item representa um conjunto de informações que identificam os dispositivos suportados pelo *driver*.

Esta estrutura é apresentada na Listagem 10 e seus campos são explicados a seguir.

```
 98 struct usb_device_id {
 99       /* which fields to match against? */
100       __u16       match_flags;
101
102       /* Used for product specific matches; range is inclusive */
103       __u16       idVendor;
104       __u16       idProduct;
105       __u16       bcdDevice_lo;
106       __u16       bcdDevice_hi;
107
108       /* Used for device class matches */
109       __u8        bDeviceClass;
110       __u8        bDeviceSubClass;
111       __u8        bDeviceProtocol;
112
113       /* Used for interface class matches */
114       __u8        bInterfaceClass;
115       __u8        bInterfaceSubClass;
116       __u8        bInterfaceProtocol;
117
118       /* not matched against */
119       kernel_ulong_t    driver_info;
120 };
```

Listagem 10: Estrutura usb_device_id, declarada no arquivo `/include/linux/mod_devicetable.h`

`__u16 match_flags`: Máscara de bits que indica quais campos da estrutura serão utilizados para confrontar os dados obtidos do dispositivo pelo USB *core* com os dados do *driver*. Esta comparação visa determinar quais *drivers* são capazes de controlar o novo dispositivo conectado ao sistema.

`__u16 idVendor`: Número único fornecido pelo USB Forum que identifica o fabricante do hardware.

`__u16 idProduct`: Número único que identifica o produto conectado. Este número pode ser gerado por qualquer fabricante que já possua um número de fabricante fornecido pelo USB Forum.

`__u16 bcdDevice_lo` e `__u16 bcdDevice_hi`: Dois números que definem o intervalo dentro do qual o *driver* pode gerenciar o produto. São utilizados nos casos em que um fabricante disponibiliza um produto em diversas versões, mas nem todas elas funcionam da mesma maneira. O intervalo de versões é fechado, o que significa que, se `bcdDevice_lo` for definido como `1` e `bcdDevice_hi` for definido como `5`, o *driver* poderá controlar dispositivos cuja versão seja `1`, `2`, `3`, `4` ou `5`.

`__u8 bDeviceClass`, `__u8 bDeviceSubClass` e `__u8 bDeviceProtocol`: Definem a classe, a subclasse e o protocolo do dispositivo, respectivamente. Estes valores são definidos pelo USB Forum[3] e um dispositivo que se encaixe em uma combinação dos três deve funcionar conforme a especificação USB para esta combinação. Da mesma forma, um *driver* que suporte uma combinação determinada poderá gerenciar qualquer dispositivo que também se encaixe em tal combinação.

`__u8 bInterfaceClass`, `__u8 bInterfaceSubClass` e `__u8 bInterfaceProtocol`: De forma similar aos parâmetros anteriores, esta combinação define o funcionamento de uma interface específica do dispositivo e também é especificada pelo USB Forum. A lista contendo os códigos válidos para classe, subclasse e protocolo de interfaces é a mesma utilizada para os parâmetros `__u8 bDeviceClass`, `__u8 bDeviceSubClass` e `__u8 bDeviceProtocol`.

`kernel_ulong_t driver_info`: Apesar de não ser possível determinar

[3] A lista contendo os códigos de classe, subclasse e protocolo pode ser consultada em <http://www.usb.org/developers/defined_class>.

que o USB *core* efetue comparações utilizando este campo através da máscara definida em `match_flags`, ele pode ser utilizado para fornecer informações adicionais que o fabricante considere importantes para diferenciação dos dispositivos.

Conforme comentado no início desta seção, um grande número de *drivers* é desenvolvido para ser capaz de controlar mais de um dispositivo. Esta é a razão pela qual o campo `const struct usb_device_id *id_table` da estrutura `struct usb_driver` não é apenas uma entrada, mas sim uma lista de uma ou mais estruturas do tipo `struct usb_device_id`. Visando facilitar a criação desta tabela e evitar que o programador precise definir a máscara `match_flags` para cada entrada, o USB *core* provê, no arquivo `/include/linux/usb.h` da árvore do *kernel*, diversas *macros* que possibilitam a definição das entradas da tabela de uma forma bastante mais simples. Dentre as *macros* mais utilizadas, estão:

`USB_DEVICE(vendor, product)`: Cria uma estrutura `struct usb_device_id` com os códigos do fabricante e do produto especificados e já define `match_flags` para testar apenas estes dois campos da estrutura quando for identificar um dispositivo.

`USB_DEVICE_VER(vendor, product, lo, hi)`: Faz o mesmo que a *macro* anterior, porém testa também o intervalo de versões do dispositivo.

`USB_DEVICE_INFO(class, subclass, protocol)`: Testa apenas os campos classe, subclasse e protocolo do dispositivo.

`USB_INTERFACE_INFO(class, subclass, protocol)`: Testa apenas os campos classe, subclasse e protocolo da interface.

Nos *drivers*, costuma-se utilizar apenas estas *macros* ao invés de definir uma `struct usb_device_id` para cada entrada da tabela. Um exemplo de tabela, identificando apenas um dispositivo (que poderia ser a tabela utilizada na Listagem 9), pode ser vista na Listagem 11.

```
1 static struct usb_device_id minha_tabela [] = {
2   { USB_DEVICE(COD_FABR_MEU_DRIVER, COD_PROD_MEU_DRIVER) },
3   { } /* Terminador da tabela */
4 };
5 MODULE_DEVICE_TABLE(usb, minha_tabela);
```

Listagem 11: Exemplo de tabela de dispositivos que funcionam com este driver

É importante reparar na chamada da *macro* `MODULE_DEVICE_TABLE` que aparece imediatamente após a declaração da tabela. Esta chamada exporta a lista de dispositivos suportados, tornando-a visível globalmente no *kernel* e permitindo que o sistema de detecção de hardware carregue este *driver* quando um dispositivo que se encaixe em uma das entradas da tabela for conectado ao sistema. O primeiro parâmetro desta *macro* deve ser a constante `usb` sempre que a tabela se referir a dispositivos do tipo USB. Isto porque a mesma estrutura de declaração de dispositivos é compartilhada por outros subsistemas, como o subsistema PCI. O segundo parâmetro deve ser a tabela de dispositivos propriamente dita.

3.8 Registro do *driver* no sistema

Uma vez que a estrutura `usb_driver` estiver definida (conforme o exemplo mostrado na Listagem 9), o dispositivo poderá ser registrado no sistema através de uma chamada à função `usb_register`, conforme exemplificado no trecho de código a seguir:

```
1  static int __init usb_meu_driver_init(void)
2  {
3    int result;
4
5    result = usb_register(&meu_driver);
6    if (result < 0)
7            err("Erro %d registrando driver.\n", result);
8
9    return result;
10 }
```

Listagem 12: Exemplo de registro do driver na inicialização do módulo

Não havendo erros, neste ponto o USB *core* toma conhecimento do *driver* e faz uso desta informação a cada novo dispositivo USB conectado ao sistema. Confrontando as informações recebidas pelo dispositivo com as informações sobre os *drivers* existentes, o USB *core* é capaz de decidir se o dispositivo possui um *driver* apto a controlá-lo e, em caso afirmativo, associar este *driver* ao dispositivo.

Conforme citado na seção sobre módulos, qualquer funcionalidade registrada deve ser cancelada, isto é, desalocada da memória quando não for mais necessária. A Listagem 13 mostra um exemplo de código que desaloca o *driver* através da função `usb_deregister` quando a função de saída do módulo é invocada.

```
1 static void __exit usb_meu_driver_exit(void)
2 {
3   usb_deregister(&meu_driver);
4 }
```

Listagem 13: Exemplo de liberação de recursos alocados na descarga do módulo

4 *HUMAN INTERFACE DEVICES* (HID)

O USB é uma arquitetura de comunicação que dá aos computadores pessoais a possibilidade de se conectarem à diversos tipos de dispositivos através de um simples cabo de 4 fios. Estes dispositivos são divididos em classes que visam agrupar dispositivos de funcionalidades semelhantes para que seja possível a definição de um padrão de comunicação para cada classe. Esta padronização facilita o desenvolvimento, tanto de dispositivos de hardware, evitando que cada fabricante tenha que criar um padrão diferente de comunicação, quanto para os sistemas operacionais e desenvolvedores de *drivers*, pois possibilita que um mesmo *driver* controle diversos dispositivos de uma mesma classe (USB IMPLEMENTERS FORUM, 2001).

Dentre as mais utilizadas estão as classes *Display* (que define o padrão de comunicação com dispositivos de exibição, como monitores e LCDs), *Communication* (que define o padrão de comunicação com dispositivos de rede e modems), *Mass Storage* (que define o padrão de comunicação com dispositivos de armazenamento, como discos rígidos e *pen drives*) e *Human Interface*. Esta última, também chamada de *Human Interface Devices* (ou simplesmente HID), é a responsável pelo controle dos dispositivos utilizados pelos usuários para interagir com o computador. Exemplos deste tipo de dispositivo são teclados, *mouses*, painéis de controle e até mesmo outros que interajam com o computador de forma semelhante aos citados, como leitores de código de barras.

É importante notar que a expressão HID foi criada para designar dispositivos de interface humana que utilizem o barramento USB, mas passou a ser utilizada para designar qualquer dispositivo deste tipo, independente do barramento que o mesmo utilize.

Neste trabalho, optou-se por abordar mais especificamente a classe HID pelo fato do dispositivo *joystick*, cujo *driver* será estudado mais adiante, pertencer a esta classe.

No sistema operacional Linux, há todo um subsistema responsável pelo gerenciamento de dispositivos pertencentes à classe HID de dispositivos USB. Este subsistema (que na verdade engloba os subsistemas Input, HID e USB) surgiu não somente com a função de oferecer suporte à dispositivos USB que se enquadrem na classe HID, mas também com a pretensão de substituir grande parte do sistema atual de gerenciamento de dispositivos de entrada (*input devices*) (LINUX KERNEL ORGANIZATION, 2002).

4.1 O subsistema Input

O subsistema Input fornece, da mesma forma que o USB *core,* uma interface que padroniza o acesso aos dispositivos de entrada. Esta interface serve como um meio de comunicação entre os dois grupos distintos de módulos do subsistema: os módulos gerenciadores de dispositivos e os módulos manipuladores de eventos.

Os **módulos gerenciadores de dispositivos** são responsáveis pela comunicação direta com o hardware (através do barramento USB, por exemplo)

para geração de eventos (pressionamento de uma tecla do teclado, movimentação do *mouse*, etc.) a partir das informações enviadas pelos dispositivos que os gerenciadores controlam. Todos os eventos e seus respectivos tipos suportados pelo subsistema Input estão declarados no arquivo `/include/linux/input.h` da árvore do *kernel*.

Os módulos gerenciadores devem declarar em seu código quais os tipos de eventos que suportam e quais os códigos relativos a estes tipos de eventos podem ser gerados. Esta declaração é feita através dos campos da estrutura `struct input_dev`, comentados a seguir.

`evbit`: Máscara de bits que informa quais os eventos suportados pelo *driver*. Pode assumir diversos valores, porém apenas os seguintes são relevantes para o presente trabalho:

`EV_KEY`: Informa que o *driver* poderá interpretar eventos do tipo pressionamento de teclas vindos do dispositivo. Exemplos de fontes para este tipo de evento são o pressionamento de teclas de um teclado ou de botões de um *joystick* ou *mouse*.

`EV_REL`: Informa que o *driver* poderá interpretar eventos do tipo movimentação relativa vindos do dispositivo. A movimentação de um *mouse*, por exemplo, caracteriza este tipo de evento.

`EV_ABS`: Informa que o *driver* poderá interpretar eventos do tipo coordenadas absolutas vindos do dispositivo. A movimentação de eixos de um *joystick* é um exemplo deste tipo de evento.

`keybit`: É uma máscara de bits contendo todos os códigos relativos às

possíveis teclas que o dispositivo pode gerar. Só é válido se o bit correspondente ao evento EV_KEY estiver habilitado em `evbit`.

`relbit`: É uma máscara de bits contendo todos os códigos dos possíveis movimentos relativos que o dispositivo pode gerar, tais como movimentos relativos ao eixo X ou ao eixo Y ou deslocamento do botão de rolagem do *mouse*. Só é válido se o bit correspondente ao evento EV_REL estiver habilitado em `evbit`.

`absbit`: É uma máscara de bits contendo todos os códigos relativos aos possíveis movimentos absolutos que o dispositivo pode gerar, tais como coordenadas nos eixos X, Y ou Z ou outro tipo de informação de valor absoluto. Só é válido se o bit correspondente ao evento EV_ABS estiver definido em `evbit`.

Por se tratarem de máscaras de bits, todos os campos citados podem assumir mais de um valor através do uso de operadores lógicos como o AND ou o OR bit a bit, o que torna possível que o *driver* gerencie mais de um tipo de evento simultaneamente.

Os **módulos manipuladores de eventos**, por sua vez, obtêm os eventos enviados ao subsistema Input e os repassam para onde for necessário através de diversas interfaces que respondem a esses eventos modificando o estado do sistema (imprimindo um caractere na tela ou movimentando o ponteiro do *mouse*, por exemplo).

4.2 O subsistema HID

O subsistema HID consiste na implementação em software da especificação HID e serve como uma camada de integração do subsistema USB com o subsistema Input. Esta integração se dá através de dois módulos: `usbhid` e `hid-input`. O módulo `usbhid` tem a função de interpretar sinais enviados pelo barramento USB e convertê-los para o formato fornecido pela especificação HID. O módulo `hid-input`, por sua vez, converte os sinais no padrão HID para um formato compreendido pelo subsistema Input.

5 SISTEMA DE ARQUIVOS VIRTUAL

Neste capítulo, apresenta-se de forma mais detalhada os sistemas de arquivos virtuais sysfs e udev, através dos quais torna-se possível a interação dos dispositivos de entrada com o espaço de usuário.

O sistema de arquivos virtual (ou *Virtual Filesystem Switch* - VFS) é uma camada de software do *kernel* que consiste em uma interface onde são implementadas funções comuns à maioria dos sistemas de arquivos, como operações de leitura (*read*), escrita (*write*), abertura (*open*) e fechamento (*close*) de arquivos (BOVET e CESATI, 2006).

Os módulos responsáveis por gerenciar sistemas de arquivos específicos fazem uso desta interface para se comunicar com o sistema operacional. Isto permite que o Linux se torne compatível com uma vasta gama de sistemas de arquivos, inclusive com sistemas de arquivos não nativos do sistema operacional, como NTFS e FAT[1].

Além de possibilitar a integração com sistemas de arquivos externos, o sistema de arquivos virtual é utilizado para representar estruturas internas do *kernel*

[1] NTFS e FAT são dois tipos de sistemas de arquivos proprietários utilizados pelo sistema operacional Windows, da Microsoft.

que se comportam de forma semelhante a de um sistema de arquivos. Dois destes sistemas, cuja compreensão é de grande importância para o presente trabalho, são apresentados a seguir.

5.1 O sistema de arquivos virtual sysfs

O sysfs foi concebido para possibilitar que processos do usuário tenham acesso a informações das estruturas internas do *kernel* (conhecidas pelo nome de *kobjects*) de uma forma mais simplificada. Para tanto, o sysfs consiste de um sistema de arquivos virtual residente apenas na memória, no qual cada diretório corresponde a um objeto do *kernel*, cada arquivo contido em um diretório corresponde a um atributo deste objeto e cada *link* simbólico corresponde a uma relação entre os objetos (MOCHEL, 2005).

O sysfs é geralmente montado no diretório `/sys` e sua estrutura básica é representada pela Listagem 14, podendo apresentar algumas variações, dependendo da distribuição Linux que estiver sendo utilizada.

```
root@ruapehu:/sys# tree -L 1
.
|-- block
|-- bus
|-- class
|-- devices
|-- firmware
|-- fs
|-- kernel
|-- module
`-- power
```

Listagem 14: Primeiro nível de diretórios do sistema de arquivos virtual `/sys`

Esta estrutura representa os principais subsistemas que se registraram no sysfs no momento da inicialização do sistema. Cada objeto do *kernel* que for registrado no sysfs terá uma ou mais referências registradas nestes diretórios, de acordo com a classe na qual se enquadre.

Da mesma forma, cada dispositivo USB conectado ao computador recebe uma entrada no sysfs através da qual é possível acessar suas propriedades.

A nomenclatura utilizada pelo sysfs para exibir os dispositivos USB conectados ao sistema segue a sintaxe ilustrada pela Listagem 15.

```
`- sys
 `- devices
  `- pci0000:00
   `- <pci_address>
    `- usb<root_hub>
     `- <root_hub>-<hub_port>...
      `- <root_hub>-<hub_port>...:<config>-<interface>
```

Listagem 15: Sintaxe utilizada pelo sysfs para representar dispositivos USB

Nesta listagem, `<pci_address>` representa o endereço da interface PCI[2] ao qual o *hub* ligado diretamente ao *host* mestre está conectado, `<root_hub>` representa o número do *hub* mestre (atribuído pelo USB *core*) ao qual o dispositivo se conectou, `<hub_port>` representa a porta deste *hub* à qual o dispositivo se conectou, `<config>` representa a configuração em uso e `<interface>` representa a interface em uso. Cabe lembrar que a topologia USB especifica que podem haver diversos *hubs* conectados em seqüência e esta é a razão da presença das reticências logo após `<root_hub>-<hub_port>`, significando que `hub_port` pode ser expandido diversas vezes[3], de acordo com o número de *hubs* conectados uns aos outros em cascata a partir de um mesmo *hub* mestre.

[2] Os endereços de todos os dispositivos de hardware que o sistema foi capaz de mapear podem ser consultados através do comando `lspci`.

[3] Mantendo a restrição de que a árvore de dispositivos não ultrapasse a altura máxima permitida pela especificação USB, conforme a Seção 3.

5.2 O sistema de arquivos udev

No sistema operacional Linux, a comunicação entre aplicações e dispositivos ocorre por meio da escrita e da leitura em arquivos especiais, criados geralmente no diretório `/dev`. Neste diretório, cada dispositivo presente no sistema é representado por um arquivo através do qual pode ser estabelecida uma comunicação. O arquivo `/dev/hda`, por exemplo, é tradicionalmente utilizado para representar o primeiro *drive* IDE do sistema. Em versões mais antigas do *kernel*, esta árvore de dispositivos era controlada por um programa chamado devfs. Entretanto, a partir do momento em que a quantidade de dispositivos aumentou de forma significativa, diversas limitações deste sistema de arquivos começaram a aparecer, forçando o desenvolvimento de um sistema de arquivos mais robusto e dinâmico (KROAH-HARTMANN, 2003).

Foi então que surgiu o sistema de arquivos udev, desenvolvido com o intuito de substituir o já obsoleto sistema de arquivos devfs. A idéia central do udev é oferecer as mesmas características deste sistema, juntamente com diversas melhorias que eram tidas como limitações do devfs.

A Listagem 16 mostra alguns dispositivos mapeados pelo sistema de arquivos udev no diretório `/dev`. No exemplo, `sda` corresponde ao primeiro disco SCSI[4] do sistema e cada partição deste disco é exibida no formato `sda<n>`, sendo n o número da partição. Assim, a primeira partição recebe o nome de `sda1`, a segunda, `sda2` e assim por diante. A adição de um número inteiro para identificar subdivisões de um dispositivo ou diversos dispositivos que utilizem um mesmo *driver* é a metodologia padrão de identificação, conforme pode ser constatado na numeração dos dispositivos de entrada do tipo *mouse* do exemplo (`mouse0`, `mouse1` e `mouse2`).

[4] O suporte a discos rígidos do tipo SATA no Linux é feito através da emulação de um disco SCSI. Por esta razão, a nomenclatura utilizada por dispositivos SATA segue o mesmo padrão de dispositivos SCSI.

```
crw-rw---- 1 root root 13, 32 2007-11-09 12:04 input/mouse0
crw-rw---- 1 root root 13, 33 2007-11-09 14:05 input/mouse1
crw-rw---- 1 root root 13, 34 2007-11-09 14:07 input/mouse2
brw-rw---- 1 root disk  8,  0 2007-11-09 12:05 sda
brw-rw---- 1 root disk  8,  1 2007-11-09 12:05 sda1
brw-rw---- 1 root disk  8,  2 2007-11-09 12:05 sda2
brw-rw---- 1 root disk  8,  3 2007-11-09 12:05 sda3
brw-rw---- 1 root disk  8, 16 2007-11-09 12:05 sdb
brw-rw---- 1 root disk  8, 17 2007-11-09 12:05 sdb1
brw-rw---- 1 root disk  8, 18 2007-11-09 12:05 sdb2
brw-rw---- 1 root disk  8, 19 2007-11-09 14:06 sdb3
```

Listagem 16: Listagem de dispositivos mapeados no diretório `/dev`*, gerada com o comando* `ls -l`

O tipo de dispositivo (caractere ou bloco) também pode ser identificado observando-se a primeira coluna da Listagem 16, onde a letra "c" representa um dispositivo do tipo caractere (*char device*) e a letra "b" representa um dispositivo do tipo bloco (*block device*) (CORBET, RUBINI e KROAH-HARTMAN, 2005).

Além disso, cada dispositivo possui dois números denominados *major* e *minor* que o identificam no sistema. Tais números são exibidos na Listagem 16 imediatamente antes da data de criação do arquivo, separados por vírgula. Os *major numbers* identificam o *driver* associado ao dispositivo, enquanto os *minor numbers* têm a função de identificar cada um dos diferentes dispositivos gerenciados por um mesmo *driver*. No exemplo, `/dev/sda` possui *major number* `8` e *minor number* `0`.

Durante algum tempo, os números *major* e *minor* eram definidos por uma entidade denominada *The Linux Assigned Names and Numbers Authority* (LANANA), que disponibiliza uma lista de todos os dispositivos e seus respectivos números em seu *site* principal <http://lanana.org/docs/device-list/index.html>.

Na Listagem 17 é possível visualizar a seção de dispositivos com *major number* `8` e seus respectivos *minor numbers* permitidos. A seção indica que o primeiro disco SCSI pode ter até 16 partições diferentes (de `sda0` a `sda15`) e o mesmo ocorre para os demais discos que por ventura estejam presentes no

sistema.

```
8 block       SCSI disk devices (0-15)
               0 = /dev/sda          First SCSI disk whole disk
              16 = /dev/sdb          Second SCSI disk whole disk
              32 = /dev/sdc          Third SCSI disk whole disk
                 ...
             240 = /dev/sdp          Sixteenth SCSI disk whole disk

             Partitions are handled in the same way as for IDE
             disks (see major number 3) except that the limit on
             partitions is 15.
```

Listagem 17: Formato do arquivo com as definições de dispositivos disponibilizado pela LANANA

A partir da versão 2.6, é possível que os números *major* sejam obtidos de forma dinâmica, no momento em que o *driver* é carregado. Esta tem sido a forma mais recomendada de utilização na escrita de *drivers*, pois evita problemas gerados pelas metodologias anteriores de alocação estática, tais como a grande quantidade de arquivos no diretório `/dev` ou a limitação no número de dispositivos que podem estar presentes no sistema simultaneamente.

6 USB REQUEST BLOCKS

Para facilitar a comunicação dos *drivers* com os dispositivos, o *kernel* do Linux disponibiliza uma estrutura chamada *USB request block* (URB). A definição desta estrutura (apresentada na Listagem 18) encontra-se, na árvore do *kernel*, no arquivo `include/linux/usb.h` e é através dela que se torna possível o envio e a recepção de dados de forma assíncrona com um dispositivo USB[1].

[1] É possível a comunicação direta com o dispositivo USB sem a utilização de URBs, porém estas estruturas simplificam muito a troca de informações com os dispositivos e, conseqüentemente, facilitam a escrita de *drivers*.

```
 1126 struct urb
 1127 {
 1128       /* private: usb core and host controller only fields in the
urb */
 1129       struct kref kref;       /* reference count of the URB */
 1130       spinlock_t lock;        /* lock for the URB */
 1131       void *hcpriv;                /* private data for host
controller */
 1132       atomic_t use_count;          /* concurrent submissions
counter */
 1133       u8 reject;              /* submissions will fail */
 1134
 1135       /* public: documented fields in the urb that can be used by
drivers */
 1136       struct list_head urb_list;   /* list head for use by the
urb's
 1137                                     * current owner */
 1138       struct usb_device *dev;      /* (in) pointer to associated
device */
 1139       unsigned int pipe;           /* (in) pipe information */
 1140       int status;             /* (return) non-ISO status */
 1141       unsigned int transfer_flags; /* (in) URB_SHORT_NOT_OK
| ...*/
 1142       void *transfer_buffer;       /* (in) associated data buffer
*/
 1143       dma_addr_t transfer_dma;     /* (in) dma addr for
transfer_buffer */
 1144       int transfer_buffer_length;  /* (in) data buffer length */
 1145       int actual_length;           /* (return) actual transfer
length */
 1146       unsigned char *setup_packet; /* (in) setup packet (control
only) */
 1147       dma_addr_t setup_dma;        /* (in) dma addr for
setup_packet */
 1148       int start_frame;        /* (modify) start frame (ISO) */
 1149       int number_of_packets;       /* (in) number of ISO packets
*/
 1150       int interval;                /* (modify) transfer interval
 1151                                     * (INT/ISO) */
 1152       int error_count;        /* (return) number of ISO errors */
 1153       void *context;               /* (in) context for completion
*/
 1154       usb_complete_t complete;     /* (in) completion routine */
 1155       struct usb_iso_packet_descriptor iso_frame_desc[0];
 1156                               /* (in) ISO ONLY */
 1157 };
```

Listagem 18: Estrutura urb, declarada no arquivo `/include/linux/usb.h`

Os campos desta estrutura que terão utilidade para o estudo do dispositivo USB *joystick* (foco de estudo deste trabalho) são apresentados e explicados a seguir:

`struct usb_device *dev`: Um ponteiro para o `usb_device` para o qual o URB será enviado. Esta estrutura deve ser inicializada para que o URB possa ser enviado ao USB *core.*

`unsigned int pipe`: Contém informações sobre o *endpoint* para o qual este URB será enviado. Deve ser inicializado pelo *driver* USB antes de poder ser enviado ao USB *core*. Para definir esta variável, são utilizadas funções de acordo com a direção do tráfego no protocolo USB. Cabe lembrar que cada *endpoint* pode ser de um único tipo. Todas as funções seguem uma nomenclatura padronizada como `usb_{snd|rcv}{ctrl|bulk|int|isoc}pipe(struct usb_device *dev, unsigned int endpoint`. As opções `snd` e `rcv` informam se o *endpoint* será de escrita ou de leitura, respectivamente, enquanto as opções `ctrl`, `bulk`, `int` e `isoc` informam se a transmissão será do tipo CONTROL, BULK, INTERRUPT ou ISOCHRONOUS, respectivamente (vide Seção 3.1). Assim, se `pipe` for definida com a função `usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)`, isto significa que o *endpoint* (indicado no segundo parâmetro da função) do dispositivo USB (indicado no primeiro parâmetro da função) será de escrita (`snd`) e do tipo CONTROL (`ctrl`).

`unsigned int transfer_flags`: Esta variável é uma máscara de bits que define o comportamento do URB. Os valores que ela pode assumir são listados e comentados a seguir.

`URB_SHORT_NOT_OK`: Este bit indica que qualquer dado do tipo `short` que for lido por *endpoints* de entrada deverá ser tratado como um erro pelo USB *core* e é útil apenas para URBs que lêem dados do dispositivo.

`URB_ISO_ASAP`: Se o URB é do tipo ISOCHRONOUS, este bit indica que o *driver* deseja que o URB seja executado o mais brevemente possível e que a variável `start_frame` do URB seja definida no momento da execução. Se este bit

não for definido, o *driver* será o responsável por definir a variável `start_frame` e deverá ser capaz de tomar as medidas necessárias, caso a transferência não ocorra no momento definido.

`URB_NO_TRANSFER_DMA_MAP`: Quando o URB contiver um *buffer*[2] DMA[3] a ser transmitido, este bit pode ser definido para que o USB *core* utilize o *buffer* apontado pela variável `transfer_dma`, ao invés de utilizar o apontado pela variável `transfer_buffer`.

`URB_NO_SETUP_DMA_MAP`: Funciona de forma similar ao bit `URB_NO_TRANSFER_DMA_MAP`, porém é utilizado por URBs do tipo CONTROL que tenham um *buffer* DMA definido. Quando definido, este bit faz com que o USB *core* utilize o *buffer* apontado pela variável `setup_dma`, ao invés de utilizar a variável `setup_packet`.

`URB_ASYNC_UNLINK`: Indica que a função `usb_unlink_urb` (detalhada na Seção 6.1) deverá executar de forma assíncrona, isto é, ao ser chamada, o sistema continuará a execução sem aguardar o retorno da função. Quando este bit não está definido, a função `usb_unlink_urb` só retorna quando o URB tiver sido totalmente desalocado e terminado. Este bit deve ser utilizado com cautela, pois pode causar erros de sincronização difíceis de serem identificados.

`URB_NO_FSBR`: Este bit é utilizado apenas pelo *driver* controlador de *host* USB UHCI e geralmente não é utilizado por outros *drivers*.

`URB_ZERO_PACKET`: Serve para que o USB *core* envie sempre um

[2] Região de memória disponível para armazenamento de dados.
[3] *Direct Memory Access*: Indica que a transferência pode ocorrer diretamente do dispositivo para a memória ou vice-e-versa, sem a necessidade de intervenção da CPU no processo.

pacote de tamanho zero a cada vez que um pacote do tipo BULK terminar de ser enviado. Este bit é necessário para que alguns dispositivos (como vários conversores $Infravermelho \Leftrightarrow USB$) funcionem corretamente.

`URB_NO_INTERRUPT`: Indica que não deve ser gerada uma interrupção ao fim de cada URB transmitido. Este bit deve ser utilizado com cautela e somente quando múltiplos URBs serão transmitidos em seqüência e para o mesmo *endpoint.* O USB *core* faz uso deste bit para efetuar transmissões do tipo DMA.

`void *transfer_buffer`: É um ponteiro para o *buffer* que será utilizado para o envio de dados ao dispositivo (se o URB for de saída (OUT)) ou para a recepção de dados (se o URB for de entrada (IN)). Este *buffer* deve ser alocado com a função `kmalloc` para que o controlador mestre possa acessá-lo corretamente. Para *endpoints* do tipo CONTROL, este *buffer* é utilizado no estágio de transferência dos dados.

`dma_addr_t transfer_dma`: *Buffer* a ser utilizado para transmitir dados ao dispositivo USB utilizando DMA. O controlador mestre tentará utilizar este *buffer* ao invés de `transfer_buffer` sempre que `URB_NO_TRANSFER_DMA_MAP` estiver definido em `transfer_flags`.

`int transfer_buffer_length`: Indica o tamanho do *buffer* usado para transferências. O controlador mestre utiliza o valor deste campo para decidir se os dados transmitidos devem ser quebrados em pacotes menores.

`unsigned char *setup_packet`: É um ponteiro para um pacote de 8 bytes que contém a configuração de um URB do tipo CONTROL e só é válido se o URB for do tipo CONTROL. Neste caso, `setup_packet` sempre é transmitido antes dos dados contidos em `transfer_buffer`.

`dma_addr_t setup_dma`: Se `URB_NO_TRANSFER_DMA_MAP` estiver definido em `transfer_flags`, o controlador mestre enviará o pacote `setup_dma` ao invés de `setup_packet` antes do início de uma transmissão do tipo CONTROL para efetuar a configuração do dispositivo. Da mesma forma que `setup_packet`, este pacote é válido somente para URBs do tipo CONTROL.

`usb_complete_t complete`: Aponta para a função que deve ser disparada pelo USB *core* quando o URB é transferido completamente ou na ocorrência de algum erro. Dentro desta função, o *driver* USB deve inspecionar o URB, liberá-lo ou reiniciar a transferência. O tipo `usb_complete_t` é definido como `typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);`

`void *context`: Em diversos casos, o acesso às informações obtidas na inicialização do *driver* podem ser necessárias em outras funções, como na função *callback* `complete` (acima). Para suprir esta demanda, este campo é um ponteiro para uma estrutura qualquer definida pelo próprio *driver*.

`int actual_length`: Quando o URB é finalizado, esta variável contém o tamanho atual dos dados enviados ou recebidos pelo URB. Para URBs de entrada (IN), este valor deve ser utilizado ao invés do `transfer_buffer_length`, pois os dados recebidos podem ser menores do que o tamanho total do *buffer*.

`int status`: Quando o URB é finalizado, ou enquanto é processado pelo USB *core*, esta variável contém o seu estado. O único momento no qual o *driver* USB pode acessar esta variável com segurança é dentro da função informada em `complete`. Esta restrição existe para prevenir condições de corrida que podem ocorrer enquanto o URB é processado pelo USB *core.* Esta variável pode assumir diversos valores, mas apenas os relevantes para este trabalho são expostos a seguir:

`0`: URB transferido com sucesso.

`-ENOENT`: URB cancelado por uma chamada a `usb_kill_urb`.

`-ECONNRESET`: URB desconectado por uma chamada a `usb_unlink_urb`.

`-EINPROGRESS`: indica que o URB ainda está sendo transmitido.

`-EPROTO`: um erro de bitstuff ocorreu durante a transferência ou o hardware não enviou um pacote de resposta a tempo.

`-EILSEQ`: um erro de CRC ocorreu durante a transferência do URB.

`-EPIPE`: o *endpoint* "travou". Neste caso, desde que o *endpoint* não seja do tipo CONTROL, é possível destravá-lo através de uma chamada a `usb_clear_halt`.

`-ECOMM`: dados recebidos mais rápido do que a capacidade de escrita da memória do sistema. Ocorre somente para URBs de entrada.

`-ENOSR`: não foi possível obter os dados na velocidade necessária para manter a taxa de dados. Ocorre somente para URBs de saída.

`-EOVERFLOW`: ocorre quando o *endpoint* recebe um pacote de tamanho maior do que o especificado para ele.

-EREMOTEIO: ocorre somente se o bit URB_SHORT_NOT_OK estiver definido na variável transfer_flags do URB e significa que não foi recebida toda a quantidade de dados solicitada pelo URB.

-ENODEV: o dispositivo USB não está mais conectado.

-EXDEV: só ocorre em URBs do tipo ISOCHRONOUS e indica que somente parte da transferência foi completada.

-EINVAL: erro grave que geralmente tem como causa a má formulação dos parâmetros de usb_submit_urb.

-ESHUTDOWN: erro severo no *driver* do controlador USB. Ocorre geralmente quando o URB é enviado após a remoção do dispositivo ou quando o controlador foi desabilitado. Também pode ocorrer se a configuração do dispositivo for modificada após o URB já ter sido enviado.

int start_frame: Define ou retorna o número do primeiro quadro a ser utilizado uma transmissão do tipo ISOCHRONOUS.

int interval: Especifica o intervalo entre cada transmissão. Esta variável é válida somente para URBs do tipo INTERRUPT ou ISOCHRONOUS. A unidade utilizada varia de acordo com a velocidade do dispositivo. Para os do tipo *low-speed* e *full-speed*, interval é especificado em quadros, que são equivalentes a milisegundos. Para dispositivos *high-speed*, a unidade utilizada é *microframe* (microquadro), sendo que cada um equivale a 125 microsegundos. Este valor deve ser definido antes do envio do URB ao USB *core.*

`int number_of_packets`: Válido apenas para URBs do tipo ISOCHRONOUS e especifica o número de *buffers* com os quais o URB irá lidar. Deve ser definido antes de o URB ser enviado ao USB *core.*

`int error_count`: Especifica a quantidade de transferências do tipo ISOCHRONOUS nas quais ocorreu algum erro. É definido após a conclusão da transmissão.

`struct usb_iso_packet_descriptor iso_frame_desc[0]`: Válida somente para URBs do tipo ISOCHRONOUS, esta variável é um vetor de estruturas do tipo `usb_iso_packet_descriptor` que definem o URB. Esta estrutura permite que um mesmo URB gerencie diversas transmissões do tipo ISOCHRONOUS simultaneamente. Também pode ser utilizada para obtenção do estado individual de cada transferência.

A estrutura `struct usb_iso_packet_descriptor` contém os seguintes campos:

`unsigned int offset`: inicia em `0` e indica o início dos dados no *buffer* de transferência.

`unsigned int length`: indica o tamanho do *buffer* de transferência.

`unsigned int actual_length`: indica a quantidade de bytes já recebida pelo *buffer*.

`unsigned int status`: indica o estado atual do pacote e corresponde à variável status da estrutura `struct urb`.

Devido à grande complexidade da estrutura `struct urb`, é comum que a inicialização desta estrutura seja realizada através da utilização de alguma das diversas funções de inicialização auxiliares, desenvolvidas de forma a facilitar a utilização de URBs. Tais funções são detalhadas nas seções seguintes.

6.1 Criação e destruição de um URB

A estrutura `struct urb` nunca deve ser utilizada de forma estática nos *drivers*, pois desta forma o USB *core* perderia o controle sobre os URBs existentes no sistema. Ao invés disso, os URBs devem ser alocados por meio de chamadas à função `usb_alloc_urb`, cujo protótipo é o que segue:

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

O primeiro parâmetro (`int iso_packets`) representa o número de pacotes do tipo ISOCHRONOUS que este URB irá transportar. Quando o tipo de transmissão utilizado não for ISOCHRONOUS, este parâmetro deve conter o valor `0`.

O segundo parâmetro (`int mem_flags`) é uma máscara de bits contendo configurações para alocação de memória e segue o mesmo padrão utilizado pela função de alocação de memória do *kernel* `kmalloc`, disponível em `/include/linux/slab.h`.

Se a função conseguir alocar o espaço de memória necessário para o URB e

executar corretamente, será retornado um ponteiro para o novo URB criado. Caso haja alguma falha na execução, a função retorna `NULL`.

Após a utilização do URB, o *driver* deve desalocar a memória utilizada através de uma chamada à função `usb_free_urb`, que tem o seguinte protótipo:

```
void usb_free_urb(struct urb *urb);
```

O argumento da função é um ponteiro para o URB que se deseja liberar. Após chamar esta função, o URB passado como parâmetro é invalidado e, portanto, não pode mais ser acessado.

6.2 Inicialização de um URB

Como diferentes tipos de dispositivos requerem diferentes parâmetros de inicialização e, conseqüentemente, diferentes tipos de URBs, o USB *core* provê *macros* com a finalidade de auxiliar na inicialização de cada tipo de URB. Estas *macros* são definidas no arquivo `/include/linux/usb.h` da árvore do *kernel* e aconselha-se o leitor a examinar este arquivo, uma vez que seu código é amplamente comentado. A seguir, cada uma destas *macros* é apresentada, acompanhada de uma breve explicação sobre seu funcionamento e seus parâmetros.

6.2.1 URBs do tipo INTERRUPT

```
void usb_fill_int_urb(struct urb *urb,
                      struct usb_device *dev,
                      unsigned int pipe,
                      void *transfer_buffer,
                      int buffer_length,
                      usb_complete_t complete_fn,
                      void *context,
                      int interval);
```

Esta função é utilizada para inicializar um URB para transferências do tipo INTERRUPT e seus parâmetros são descritos a seguir:

`struct urb *urb`: Um ponteiro para o URB a ser inicializado.

`struct usb_device *dev`: Indica o dispositivo USB para o qual o URB será enviado.

`unsigned int pipe`: Indica para qual *endpoint* do dispositivo o URB será enviado. Este valor é obtido através das funções `usb_sndintpipe` ou `usb_rcvintpipe`, já apresentadas.

`void *transfer_buffer`: Um ponteiro para o *buffer* de envio ou recepção de dados. Este *buffer* não pode ser estático e deve, portanto, ser criado através de uma chamada a função `kmalloc` ou similar.

`int buffer_length`: Indica o tamanho do *buffer* apontado por `transfer_buffer`.

`usb_complete_t complete_fn`: É um apontador para uma função do tipo

callback que deverá ser chamada quando o URB for completado.

`void *context`: Equivale ao campo `void *context` da estrutura `struct urb`, comentada anteriormente.

`int interval`: Indica o intervalo entre as interrupções, conforme definido na estrutura `struct urb`.

6.2.2 URBs do tipo BULK

```
void usb_fill_bulk_urb(struct urb *urb,
                       struct usb_device *dev,
                       unsigned int pipe,
                       void *transfer_buffer,
                       int buffer_length,
                       usb_complete_t complete_fn,
                       void *context);
```

Salvo pelo fato de não possuir o parâmetro `int interval`, esta função é exatamente igual à função `usb_fill_int_urb`. A razão para a ausência do parâmetro mencionado é óbvia, uma vez que, como já dito, transferências do tipo BULK não ocorrem em intervalos específicos, como ocorre nas transferências do tipo INTERRUPT.

6.2.3 URBs do tipo CONTROL

```
void usb_fill_control_urb(struct urb *urb,
                          struct usb_device *dev,
                          unsigned int pipe,
                          unsigned char *setup_packet,
                          void *transfer_buffer,
                          int buffer_length,
                          usb_complete_t complete_fn,
                          void *context);
```

A inicialização de URBs do tipo CONTROL se dá de forma muito similar à inicialização de URBs do tipo BULK, com uma única diferença: o parâmetro `unsigned char *setup_packet`. Este parâmetro deve ser preenchido com as informações de configuração do dispositivo, conforme definido anteriormente no parâmetro `char *setup_packet` da estrutura `struct urb`.

6.2.4 URBs do tipo ISOCHRONOUS

Para este tipo de URB, não existe uma *macro* auxiliar de inicialização. Por esta razão, URBs deste tipo devem ser inicializados da forma tradicional, isto é, definindo-se manualmente cada um dos campos requeridos pela `struct urb`. Isto, porém, não constitui uma tarefa complexa, conforme ilustra o modelo de inicialização da Listagem 19.

```
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisopipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DSEC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
        urb->iso_frame_desc[j].offset = j;
        urb->iso_frame_desc[j].length = 1;
}
```

Listagem 19: Trecho do código de inicialização de um URB do tipo ISOCHRONOUS (extraído do driver `/drivers/usb/media/konicawc.c`*)*

6.3 Envio de um URB

Após ter sido corretamente criado e inicializado no sistema, o URB pode ser enviado ao USB *core* através da função a seguir:

```
int usb_submit_urb(struct urb *urb, gfp_t mem_flags);
```

O primeiro parâmetro desta função referencia o URB a ser enviado e o segundo, é equivalente ao parâmetro passado a função `kmalloc` e serve para informar ao USB *core* como deve alocar a memória necessária para este envio de URB. Há diversas possibilidades de configuração de `mem_flags`, porém apenas as mais utilizadas são explicadas a seguir.

`GFP_ATOMIC`: Utilizado quando a alocação de memória deverá ocorrer de forma atômica, isto é, sem a possibilidade de algum outro processo interferir durante o processo de alocação. Internamente, significa que o processo não será colocado

em modo *sleep* pelo *kernel* até que a alocação tenha sido concluída.

`GFP_NOIO`: Impede que qualquer tipo de operação de entrada ou saída (I/O) seja realizada até que a alocação termine.

`GFP_KERNEL`: Alocação normal, sem restrições. Quando nenhum dos dois tipos de alocação anteriores for realmente necessário, recomenda-se utilizar `GFP_KERNEL`.

Após o envio do URB, duas condições são possíveis:

1. O envio foi completado com sucesso: neste caso, `usb_submit_urb` retornará o valor `0` e a função *callback* definida no URB será invocada. O campo `status` do URB também assumirá o valor `0`.

2. Algum erro ocorreu: neste caso, `usb_submit_urb` retorna um valor negativo e o código correspondente ao erro ocorrido pode ser consultado no campo `status` do URB.

6.4 Cancelamento de um URB

No caso de desconexão do dispositivo ou por determinação explícita do *driver*, pode ser necessário cancelar um URB enviado. Isto pode ser feito através das seguintes funções:

```
int usb_kill_urb(struct urb *urb);

int usb_unlink_urb(struct urb *urb);
```

Cada uma destas funções, como seu próprio protótipo deixa claro, recebe como parâmetro o URB a ser cancelado.

A diferença básica entre estas duas funções é que `usb_kill_urb` é utilizada de forma geral quando o dispositivo é removido do sistema e executa de forma síncrona, enquanto `usb_unlink_urb` é utilizada para apenas cancelar o URB de forma assíncrona e, para que funcione corretamente, o bit `URB_ASYNC_UNLINK` deve estar definido no URB. A função `usb_kill_urb` é comumente utilizada na função *callback* de desconexão do dispositivo.

7 ESTUDO DE CASO: O *DRIVER* XPAD

Para uma melhor compreensão do desenvolvimento de *drivers* de dispositivos USB para Linux, o presente trabalho aborda as etapas envolvidas na utilização de um novo dispositivo USB que é conectado ao computador (conexão, funcionamento e desconexão). Visando manter o foco no tema principal do trabalho, optou-se por utilizar para o exemplo prático um dispositivo simples e didático. Também era desejável que o dispositivo a ser estudado fosse um periférico de larga utilização e, portanto, conhecido e sem dificuldades de aquisição. Além destas características, o dispositivo já deveria ser compatível com Linux, isto é, já possuir um *driver* que o controlasse. Após uma pesquisa para encontrar o dispositivo que melhor se enquadrava nestes requisitos, optou-se pelo *joystick* USB modelo MD-007, fabricado na China pela empresa japonesa MIDI Japan. A Figura 6 apresenta o dispositivo utilizado.

Figura 6: Joystick USB modelo MIDI MD-007

7.1 Mensagens geradas quando o dispositivo é conectado

Para exibir as mensagens geradas pelo *kernel* do Linux, o sistema provê o comando `dmesg`. Além de permitir ao usuário a visualização das mensagens geradas durante a inicialização do sistema, este comando é muito útil também para a obtenção de informações sobre o comportamento do sistema quando um novo dispositivo é inserido ou removido (LINUX KERNEL ORGANIZATION, 2007).

Ao conectar o *joystick* ao computador, o comando `dmesg` exibe a saída representada na Listagem 20.

```
root@ruapehu:~# dmesg
[ 6179.384000] usb 2-1: new low speed USB device using uhci_hcd and
address 2
[ 6179.568000] usb 2-1: configuration #1 chosen from 1 choice
[ 6179.684000] usbcore: registered new interface driver hiddev
[ 6179.744000] input: LuenKeung Co.,Ltd USB  Joystick
as /class/input/input6
[ 6179.744000] input: USB HID v1.10 Joystick [LuenKeung Co.,Ltd USB
Joystick] on usb-0000:00:1d.0-1
[ 6179.744000] usbcore: registered new interface driver usbhid
[ 6179.744000] drivers/usb/input/hid-core.c: v2.6:USB HID core driver
[ 6179.752000] usbcore: registered new interface driver xpad
[ 6179.752000] drivers/usb/input/xpad.c: driver for Xbox controllers
v0.1.6
```

Listagem 20: Saída do comando `dmesg` *ao conectar o joystick ao computador*

Cabe salientar que a ordem das mensagens geradas para o *log* não é fixa, uma vez que o fato de o *kernel* executar de forma assíncrona faz com que os *drivers* sejam carregados de acordo com a prioridade dada pelo escalonador do sistema.

Na primeira linha da Listagem 20, o identificador `2-1` informa que o dispositivo foi conectado ao *hub* mestre 2, em sua porta número 1. Além disso, trata-se de um dispositivo de baixa velocidade (*low speed*), que será controlado pelo *driver* controlador de *hub* mestre `uhci_hcd`, recebendo o endereço 2 do barramento USB.

A segunda linha diz que o dispositivo dispõe de apenas 1 configuração (*1 choice*) e que ela foi escolhida como configuração atual.

Em seguida, o USB *core* informa que conseguiu registrar com sucesso o *driver* de interfaceamento `hiddev`, ou seja, o dispositivo conectado foi reconhecido pelo USB *core* como sendo do tipo HID e o *driver* de suporte a este tipo de dispositivo foi carregado com sucesso.

Por pertencer à classe HID, o subsistema Input deve ser invocado para interpretar as mensagens enviadas pelo dispositivo ao sistema. Quando invocado, o subsistema Input gera as duas linhas seguintes, dando informações sobre o dispositivo detectado e sobre a localização dele no sistema de arquivos sysfs (no subdiretório `/class/input/input6` do diretório onde o sysfs estiver montado).

Após a carga dos *drivers* necessários para estabelecer a comunicação do subsistema USB com a interface HID e a comunicação desta interface com o subsistema Input, o *driver* que controla o dispositivo pode ser carregado. Isto ocorre nas duas últimas linhas. A penúltima informa que o *driver* identificado pelo USB *core* como apropriado para controlar o novo dispositivo é o xpad, enquanto a última linha dá informações sobre a carga deste *driver.* Neste momento o dispositivo já está pronto para ser utilizado por alguma aplicação.

7.2 O *driver* xpad

Conforme visto na seção anterior, o *joystick* foi reconhecido pelo sistema como um controle Xbox e o *driver* escolhido pelo sistema para controlá-lo foi o xpad, desenvolvido por Marko Friedemann, Oliver Schwartz, Georg Lukas, Thomas Pedley e Edgar Hucek. O código fonte do módulo é composto de dois arquivos (`xpad.h` e `xpad.c`) que residem no diretório `/drivers/input/joystick` da árvore do *kernel* (ANEXO I). O objetivo desta seção é demonstrar a aplicação prática dos assuntos abordados nos capítulos anteriores através da análise de cada parte do código fonte do *driver* xpad.

Para facilitar o estudo do *driver*, as declarações serão explicadas na seqüência em que aparecem nos arquivos, iniciando-se pelo arquivo `xpad.c`.

Inicialmente, são incluídos os arquivos que permitem ao *driver* acessar as funções disponibilizadas pelo *kernel* e as funções necessárias para estabelecer a comunicação com o dispositivo USB, com o USB *core,* com o HID *core* e com o subsistema Input. Em seguida, na linha 65 da Listagem 21, o arquivo `xpad.h` é incluído com a finalidade de declarar as definições e os cabeçalhos das funções que serão utilizadas pelo módulo e que podem eventualmente ser utilizadas por outros módulos. Dentre as entidades declaradas está a estrutura `struct usb_xpad`, que armazena as características particulares do dispositivo xpad. Outra declaração importante é a responsável por armazenar a lista de dispositivos suportados pelo *driver*, que difere um pouco da lista padrão `struct id_table` e é chamada de `struct xpad_device`.

Retornando ao arquivo `xpad.c`, a Listagem 21 ilustra também a declaração de um parâmetro do módulo (vide Seção 2.4.4) chamado `debug`. Este parâmetro é inicializado, por padrão, com o valor `0` e, se definido com valor `1` na carga do módulo, habilita a exibição de detalhes sobre os pacotes recebidos pela função `static void xpad_process_packet(struct usb_xpad *xpad, u16 cmd, unsigned char *data)` do arquivo `xpad.c` (linha 155), facilitando que novos dispositivos que se comuniquem de forma similar possam ser adicionados à lista de dispositivos suportados pelo *driver.*

```
65 #include "xpad.h"
66
67 static unsigned long debug = 0;
68 module_param(debug, ulong, 0444);
69 MODULE_PARM_DESC(debug, "Debugging");
```

Listagem 21: Parâmetro do módulo

A Listagem 22 mostra a lista de dispositivos suportados pelo *driver*. Cada entrada desta lista possui os campos "código do fabricante", "código do produto", "isMat", "nome do dispositivo" e "is360", nesta ordem. Os dois primeiros campos são códigos atribuídos aos dispositivos pelo USB Forum. Os dispositivos que possuem o

campo "isMat" definido como 1 identificam dispositivos do tipo *Dance Pad*, enquanto os que possuem o campo "is360" definido como 1 identificam *joysticks* do tipo Xbox 360. Estes dois campos são necessários porque dispositivos *Dance Pad* ou Xbox 360 possuem algumas características especiais, como ausência de eixos axiais nos *Dance Pads* e a existência de botões adicionais em controles Xbox 360.

```
   71 static const struct xpad_device xpad_device[] = {
   72       /* please keep those ordered wrt. vendor/product ids
   73         vendor, product, isMat, name, is360                  */
   74       { 0x044f, 0x0f07, 0, "Thrustmaster, Inc. Controller", 0},
   75       { 0x045e, 0x0202, 0, "Microsoft Xbox Controller", 0},
   76       { 0x045e, 0x0285, 0, "Microsoft Xbox Controller S", 0},
   77       { 0x045e, 0x0287, 0, "Microsoft Xbox Controller S", 0},
   78       { 0x045e, 0x0289, 0, "Microsoft Xbox Controller S", 0}, /*
microsoft is stupid */
   79       { 0x045e, 0x028e, 0, "Microsoft Xbox 360 Controller", 1},
   80       { 0x046d, 0xca84, 0, "Logitech Xbox Cordless Controller", 0},
   81       { 0x046d, 0xca88, 0, "Logitech Compact Controller for Xbox",
0},
   82       { 0x05fd, 0x1007, 0, "???Mad Catz Controller???", 0}, /*
CHECKME: this seems strange */
   83       { 0x05fd, 0x107a, 0, "InterAct PowerPad Pro", 0},
   84       { 0x0738, 0x4516, 0, "Mad Catz Control Pad", 0},
   85       { 0x0738, 0x4522, 0, "Mad Catz LumiCON", 0},
   86       { 0x0738, 0x4526, 0, "Mad Catz Control Pad Pro", 0},
   87       { 0x0738, 0x4536, 0, "Mad Catz MicroCON", 0},
   88       { 0x0738, 0x4540, 1, "Mad Catz Beat Pad", 0},
   89       { 0x0738, 0x4556, 0, "Mad Catz Lynx Wireless Controller", 0},
   90       { 0x0738, 0x6040, 1, "Mad Catz Beat Pad Pro", 0},
   91       { 0x0c12, 0x8802, 0, "Zeroplus Xbox Controller", 0},
   92       { 0x0c12, 0x8809, 0, "Level Six Xbox DDR Dancepad", 0},
   93       { 0x0c12, 0x8810, 0, "Zeroplus Xbox Controller", 0},
   94       { 0x0c12, 0x9902, 0, "HAMA VibraX - *FAULTY HARDWARE*", 0}, /*
these are broken */
   95       { 0x0e4c, 0x1097, 0, "Radica Gamester Controller", 0},
   96       { 0x0e4c, 0x2390, 0, "Radica Games Jtech Controller", 0},
   97       { 0x0e6f, 0x0003, 0, "Logic3 Freebird wireless Controller", 0},
   98       { 0x0e6f, 0x0005, 0, "Eclipse wireless Controller", 0},
   99       { 0x0e6f, 0x0006, 0, "Edge wireless Controller", 0},
  100       { 0x0e6f, 0x000c, 0, "PELICAN PL-2047", 0},
  101       { 0x0e8f, 0x0201, 0, "SmartJoy Frag Xpad/PS2 adaptor", 0},
  102       { 0x0f30, 0x0202, 0, "Joytech Advanced Controller", 0},
  103       { 0x0f30, 0x8888, 0, "BigBen XBMiniPad Controller", 0},
  104       { 0x102c, 0xff0c, 0, "Joytech Wireless Advanced Controller",
0},
  105       { 0x12ab, 0x8809, 1, "Xbox DDR dancepad", 0},
  106       { 0xffff, 0xffff, 0, "Chinese-made Xbox Controller", 0}, /* WTF
are device IDs for? */
  107       { 0x0000, 0x0000, 0, "nothing detected - FAIL", 0}
  108 };
```

Listagem 22: Lista de dispositivos suportados pelo driver

A lista de dispositivos suportados é utilizada na função `static int xpad_probe(struct usb_interface *intf, const struct usb_device_id *id)` do *driver* (linha 335) para verificar se o dispositivo detectado pelo USB *core* encontra-se de fato na lista de dispositivos suportados.

Dispositivos do tipo HID interagem com o sistema operacional através do envio de códigos que identificam um tipo de interação. Em teclados, este código geralmente diz respeito ao pressionamento de uma tecla, enquanto que em um *mouse*, o código pode significar uma modificação de coordenada ou o pressionamento de um dos botões. No caso do *joystick*, os código gerados são relativos ao pressionamento de um de seus botões ou à movimentação de um de seus eixos. Todos os códigos suportados pelo subsistema Input são definidos no arquivo `/include/linux/input.h`.

Para facilitar a referência aos códigos que os dispositivos suportados pelo *driver* xpad podem gerar, são utilizados vetores contendo todos estes possíveis códigos. A Listagem 23 mostra a declaração destes vetores.

```
  110 static const signed short xpad_btn[] = {
  111       BTN_A, BTN_B, BTN_C, BTN_X, BTN_Y, BTN_Z, /* analogue buttons
*/
  112       BTN_START, BTN_BACK, BTN_THUMBL, BTN_THUMBR,    /*
start/back/sticks */
  113       BTN_0, BTN_1, BTN_2, BTN_3,             /* d-pad as buttons
*/
  114       BTN_TL, BTN_TR,                         /* Button LB/RB */
  115       BTN_MODE,                          /* The big X */
  116       -1                                 /* terminating entry */
  117 };
  118
  119 static const signed short xpad_mat_btn[] = {
  120       BTN_A, BTN_B, BTN_X, BTN_Y,   /* A, B, X, Y */
  121       BTN_START, BTN_BACK,          /* start/back */
  122       BTN_0, BTN_1, BTN_2, BTN_3,   /* directions */
  123       -1                       /* terminating entry */
  124 };
  125
  126 static const signed short xpad_abs[] = {
  127       ABS_X, ABS_Y,           /* left stick */
  128       ABS_RX, ABS_RY,         /* right stick */
  129       ABS_Z, ABS_RZ,          /* triggers left/right */
  130       ABS_HAT0X, ABS_HAT0Y,   /* digital pad */
  131       ABS_HAT1X, ABS_HAT1Y,   /* analogue buttons A + B */
  132       ABS_HAT2X, ABS_HAT2Y,   /* analogue buttons C + X */
  133       ABS_HAT3X, ABS_HAT3Y,   /* analogue buttons Y + Z */
  134       -1                 /* terminating entry */
  135 };
```

Listagem 23: Códigos de botões, teclas e eixos que os dispositivos suportados podem gerar

Os dois primeiros vetores declaram os botões utilizados pelos dispositivos. O primeiro deles, `xpad_btn[]`, contém a lista de botões utilizada pela maior parte dos dispositivos suportados, enquanto o segundo, `xpad_mat_btn[]` contém a lista de botões utilizada pelos dispositivos do tipo *Dance Pad*. O terceiro vetor, `xpad_abs[]`, contém a lista de códigos gerados pelos eixos que os dispositivos suportados podem apresentar. Para facilitar a iteração nos itens dos vetores sem que seja necessário definir um tamanho fixo para eles, o último item de cada vetor recebe o valor `-1`.

A Listagem 24 exibe a declaração das interfaces definidas pelo USB Forum que o *driver* é capaz de gerenciar. Conforme dito anteriormente, esta tabela tem a função de informar ao USB *core* que o *driver* que a declara pode ser invocado para

gerenciar qualquer dispositivo que implemente esta combinação de classe, subclasse e protocolo.

```
  137 static struct usb_device_id xpad_table [] = {
  138       { USB_INTERFACE_INFO('X', 'B', 0) },      /* Xbox USB-IF not
approved class */
  139       { USB_INTERFACE_INFO( 3 ,  0 , 0) },      /* for Joytech
Advanced Controller */
  140       { USB_INTERFACE_INFO( 255 ,  93 , 1) }, /* Xbox 360 */
  141       { }
  142 };
  143
  144 MODULE_DEVICE_TABLE(usb, xpad_table);
```

Listagem 24: Interfaces definidas pelo USB Forum suportadas pelo driver

A primeira entrada da tabela (linha 138) cita a classe `'X'`, equivalente ao valor hexadecimal `58h` e que, conforme diz o comentário, não consta na lista de classes especificadas pelo USB Forum (USB IMPLEMENTERS FORUM, 2006). Desta forma, a subclasse `'B'` (`42h` em hexadecimal) também não consta nesta lista. Mesmo assim, se algum dispositivo com esta combinação for detectado, o *driver* se habilita a controlá-lo.

A segunda entrada da tabela (linha 139) cita a classe `03`, que define dispositivos do tipo HID (vide Seção 4). A especificação HID define somente dois valores significativos de subclasses: valor `0`, significando que não há subclasse e valor `1`, quando se trata de uma subclasse que seja uma interface de inicialização. Os valores restantes (de 2 a 255) são de uso restrito. A especificação HID também define os protocolos que podem ser utilizados: `0`, quando nenhum protocolo será utilizado (utilizado na linha 139), `1`, quando se tratar de um teclado e 2, quando se tratar de um *mouse*. Os valores restantes (de 3 a 255) são de uso restrito.

A próxima entrada da tabela (linha 140) faz referência a dispositivos da classe 255. Segundo o USB Forum, dispositivos pertencentes a esta classe não se enquadram em nenhuma das classes definidas e são consideradas "específicas do

fabricante". Os campos subclasse e protocolo, portanto, são identificadores utilizados pelos fabricantes. No caso do *driver* xpad, o suporte é oferecido aos dispositivos que se enquadrarem na subclasse 93 (5Dh em hexadecimal) e utilizarem o protocolo 1.

Conforme a Seção 2.4.1, há algumas declarações obrigatórias para a criação de módulos para o *kernel*. A linha 536 da Listagem 25 declara como função a ser invocada no momento da inicialização do módulo a função `static int __init usb_xpad_init(void)`, enquanto a linha 537 declara como função a ser invocada no momento da descarga do módulo a função `static void __exit usb_xpad_exit(void)`.

```
510 static struct usb_driver xpad_driver = {
511       .name       = "xpad",
512       .probe            = xpad_probe,
513       .disconnect = xpad_disconnect,
514       .id_table   = xpad_table,
515 };
516
517 /**
518  * driver init entry point
519  */
520 static int __init usb_xpad_init(void)
521 {
522       int result = usb_register(&xpad_driver);
523       if (result == 0)
524             info(DRIVER_DESC " " DRIVER_VERSION);
525       return result;
526 }
527
528 /**
529  * driver exit entry point
530  */
531 static void __exit usb_xpad_exit(void)
532 {
533       usb_deregister(&xpad_driver);
534 }
535
536 module_init(usb_xpad_init);
537 module_exit(usb_xpad_exit);
538
539 MODULE_AUTHOR(DRIVER_AUTHOR);
540 MODULE_DESCRIPTION(DRIVER_DESC);
541 MODULE_LICENSE("GPL");
```

Listagem 25: Declaração das funções de inicialização e descarga do driver

Ambas as funções fazem uso de uma instância da estrutura `struct usb_driver` (abordada na Seção 3.7), responsável por declarar as funções que irão compor a interface do *driver* com o USB *core.* Esta estrutura (linha 510) associa uma função a cada uma das funções de interface do USB *core*. O campo `probe` define `xpad_probe` como a função a ser chamada quando o USB *core* desejar testar se o *driver* pode suportar determinado dispositivo. O campo `disconnect` define `xpad_disconnect` como a função a ser chamada quando o dispositivo que está sendo controlado pelo *driver* for desconectado do sistema. O campo `id_table` declara a tabela que será utilizada pelo USB *core* para decidir se o *driver* pode suportar o novo dispositivo conectado ao sistema.

A função de descarga contém apenas uma chamada para liberação dos recursos alocados pelo *driver* enquanto o mesmo era utilizado pelo sistema.

A função de inicialização tem a incumbência de tentar registrar a estrutura `struct usb_device` (declarada a partir da linha 510) e tomar as atitudes necessárias, caso isso não seja possível. Se tudo correr bem, uma mensagem informando o nome do *driver* e sua versão (ambos declarados no arquivo `xpad.h` como `DRIVER_DESC` e `DRIVER_VERSION`) é enviada ao *log* do sistema.

As linhas 539, 540 e 541, por fim, declaram o autor, uma breve descrição do *driver* e a licença sob a qual o *driver* é disponibilizado. O autor e a descrição são declarados sob a forma de definições, feitas previamente quando o arquivo `xpad.h` é incluído. Estas *macros* para declaração de informações à respeito de um módulo são abordadas na Seção 2.4.1.

7.2.1 Detecção do dispositivo

Quando um novo dispositivo USB é conectado ao sistema, o USB *core* percorre sua lista interna de dispositivos suportados em busca de um *driver* apto a gerenciar tal dispositivo. Se for encontrado, o *driver* em questão deverá possuir uma função responsável por responder à solicitação do USB *core,* informando se tem ou não condições de gerenciar o novo dispositivo. No caso do *driver* xpad, esta função chama-se `xpad_probe`.

A função `xpad_probe` da Listagem 26 recebe, por definição, um parâmetro contendo a interface em uso do dispositivo USB e outro, contendo uma estrutura `struct usb_device_id`, que corresponde a uma das entradas da tabela de dispositivos suportados (no caso, para uma entrada da tabela `xpad_table`) que o USB *core* utilizou para concluir que este *driver* poderia controlar o dispositivo em questão.

```
  335 static int xpad_probe(struct usb_interface *intf, const struct
usb_device_id *id)
  336 {
  337       struct usb_device *udev = interface_to_usbdev(intf);
  338       struct usb_xpad *xpad;
  339       struct input_dev *input_dev;
  340       struct usb_endpoint_descriptor *ep_irq_in;
  341       int i;
  342       int error = -ENOMEM;
  343       int probedDevNum = -1;  /* this takes the index into the known
devices
  344                                  array for the recognized device */
  345
  346       /* try to detect the device we are called for */
  347       for (i = 0; xpad_device[i].idVendor; ++i) {
  348             if ((le16_to_cpu(udev->descriptor.idVendor) ==
xpad_device[i].idVendor) &&
  349                   (le16_to_cpu(udev->descriptor.idProduct) ==
xpad_device[i].idProduct)) {
  350                   probedDevNum = i;
  351                   break;
  352             }
  353       }
  354
  355       /* sanity check, did we recognize this device? if not, fail */
  356       if ((probedDevNum == -1) || (!
xpad_device[probedDevNum].idVendor &&
  357                                    !xpad_device[probedDevNum].idProduct))
  358             return -ENODEV;
```

Listagem 26: Detecção do dispositivo - etapa inicial

Ao ser invocada pelo USB *core*, a função pode apresentar dois tipos de valores de retorno:

Valor negativo: Se ocorrerem erros durante a fase de detecção do dispositivo, ou caso descubra que não pode controlar o dispositivo informado pelo USB *core*, a função `probe` deve retornar um valor negativo (que pode corresponder a um código de erro). Neste caso, o USB *core* continuará a percorrer sua lista de *drivers* em busca de outro módulo capaz de controlar o dispositivo detectado.

Valor `0`: Caso o dispositivo possa ser controlado pelo *driver* e os processos de detecção e configuração do dispositivo forem completados com êxito, a função deverá retornar este valor.

No início da função, as estruturas necessárias para sua execução são declaradas. Em especial, na linha 337, a interface recebida como parâmetro é convertida em uma estrutura `struct usb_device` com a função `interface_to_usbdev`, abordada brevemente na Seção 3.6.

Após inicializar as estruturas auxiliares, o *driver* itera sobre a sua tabela de dispositivos suportados em busca de uma combinação com o código do fabricante e com o código do produto fornecidos pelo USB *core,* conforme ilustrado pela Listagem 27. Ao fim da iteração, a variável `probedDevNum` conterá o índice da tabela correspondente ao dispositivo detectado ou `-1` se nenhum dispositivo da lista possui tal código de fabricante e código do produto. Se isto ocorrer, a função `xpad_probe` retornará o código de erro `-ENODEV` (os códigos de erro válidos são comentados na Seção 6).

```
346       /* try to detect the device we are called for */
347       for (i = 0; xpad_device[i].idVendor; ++i) {
348             if ((le16_to_cpu(udev->descriptor.idVendor) ==
xpad_device[i].idVendor) &&
349                 (le16_to_cpu(udev->descriptor.idProduct) ==
xpad_device[i].idProduct)) {
350                   probedDevNum = i;
351                   break;
352             }
353       }
354
355       /* sanity check, did we recognize this device? if not, fail */
356       if ((probedDevNum == -1) || (!
xpad_device[probedDevNum].idVendor &&
357                                    !xpad_device[probedDevNum].idProduct))
358             return -ENODEV;
```

Listagem 27: Laço para testar os dispositivos suportados

O USB Forum define, na especificação USB, que a transmissão no barramento deverá sempre seguir o padrão de ordenação de bytes *little endian*[1]. Com base nesta informação, o *driver* xpad utiliza, nas linhas 348 e 349 da Listagem

[1] O formato *little endian,* em oposição ao formato *big endian,* indica que a disposição dos bytes em uma cadeia de mais de um byte se dará colocando-se os bytes menos significativos nos endereços menos significativos.

27, a função `le16_to_cpu` para converter os valores enviados pelo dispositivo (em formato *little endian* de 16 bits) para um formato reconhecido pela CPU atual, que pode utilizar uma ordenação diferente.

7.2.2 Inicialização das estruturas auxiliares

Após identificar o dispositivo, a inicialização de algumas estruturas se torna necessária para estabelecer a comunicação com o ele. Conforme pode ser visto na Listagem 28, na linha 360, o *driver* solicita ao *kernel* que reserve uma área de memória capaz de armazenar uma estrutura `struct usb_xpad` e que preencha esta área de memória com zeros[2] para garantir que todos os campos desta estrutura sejam inicializados e não contenham informações inválidas. O *driver* utiliza esta estrutura para armazenar todas as informações que possam vir a ser necessárias durante o funcionamento do dispositivo que controla.

[2] As funções `kmalloc` e `kzalloc` são utilizadas para alocação de memória. A diferença entre elas é que a primeira apenas aloca uma área de memória sem inicializá-la, enquanto a segunda inicializa a área alocada preenchendo-a com zeros.

```
360       xpad = kzalloc(sizeof(struct usb_xpad), GFP_KERNEL);
361       input_dev = input_allocate_device();
362       if (!xpad || !input_dev)
363             goto fail1;
364
365       xpad->idata = usb_buffer_alloc(udev, XPAD_PKT_LEN,
366                                      GFP_ATOMIC, &xpad->idata_dma);
367       if (!xpad->idata)
368             goto fail1;
369
370       /* setup input interrupt pipe (button and axis state) */
371       xpad->irq_in = usb_alloc_urb(0, GFP_KERNEL);
372       if (!xpad->irq_in)
373             goto fail2;
374
375       xpad->udev = udev;
376       xpad->dev = input_dev;
377       xpad->isMat = xpad_device[probedDevNum].isMat;
378       xpad->is360 = xpad_device[probedDevNum].is360;
379       usb_make_path(udev, xpad->phys, sizeof(xpad->phys));
380       strlcat(xpad->phys, "/input0", sizeof(xpad->phys));
```

Listagem 28: Inicialização das estruturas de controle do driver

Em seguida, na linha 361, a função `input_allocate_device` do subsistema Input é chamada para alocar espaço de memória para um novo dispositivo do tipo *input*. Caso alguma das tentativas de alocar memória falhe, o fluxo de execução é desviado para o final da função, onde as estruturas que já tiverem sido alocadas previamente são liberadas. Se não ocorrerem erros, as estruturas alocadas serão inicializadas.

A função `usb_buffer_alloc`, na linha 365, define um *buffer* de tamanho `XPAD_PKT_LEN`[3] bytes que é alocado de maneira atômica, isto é, sem que outros processos possam interferir no processo de alocação. O endereço deste *buffer* é armazenado no campo `idata` da estrutura `struct usb_xpad`. Um endereço para troca de informações no modo DMA também é armazenado no campo `idata_dma`. Em caso de erro na alocação, o fluxo é também desviado para o fim da função e as estruturas alocadas são liberadas.

Na linha 371, um novo URB é alocado e um ponteiro para a área de memória

[3] Constante definida em `xpad.h`.

onde irá residir é armazenado no campo `irq_in` da estrutura `struct xpad`. Após inicializado[4], este URB será o responsável por trazer as informações de estado de botões e eixos do *joystick* até o *driver* para que possam ser interpretadas.

As linhas 375 a 378 simplesmente copiam informações que serão utilizadas posteriormente para a estrutura `struct usb_xpad`, de forma que possam ser facilmente acessadas em outras partes do *driver*.

Em seguida, a função `usb_make_path` define um caminho estável na árvore de dispositivos USB, isto é, que não mudará até que o dispositivo seja fisicamente removido e conectado a outra porta. A função gera o mesmo caminho, mesmo que o sistema seja reiniciado ou que o dispositivo seja desconectado e reconectado à mesma porta.

A Listagem 29 dá continuidade à inicialização da estrutura contida em `input_dev`. Na linha 384, o identificador de dispositivo USB é convertido em uma estrutura compatível com o subsistema Input e a estrutura passa a ser referenciada pelo campo `input_dev->id`.

```
382       input_dev->name = xpad_device[probedDevNum].name;
383       input_dev->phys = xpad->phys;
384       usb_to_input_id(udev, &input_dev->id);
386
387       input_set_drvdata(input_dev, xpad);
388
389       input_dev->open = xpad_open;
390       input_dev->close = xpad_close;
```

Listagem 29: Continuação da inicialização da estrutura `struct input_dev`

A linha 387 atribui o conteúdo de `xpad` a um campo específico de `input_dev`. Desta forma, os dados até então definidos em `xpad` podem ser

[4] A inicialização deste URB é abordada mais adiante na Listagem 31.

acessados de qualquer local onde `input_dev` for visível.

De maneira geral, a utilização dos dispositivos do tipo HID se dá da mesma forma que o acesso a arquivos, isto é, através de instruções `open` (para abrir uma conexão com o dispositivo), `close` (para desalocar os ponteiros que mantinham o dispositivo aberto), `read` (para ler informações geradas pelo dispositivo) e `write` (para definir alguma configuração, como ligar ou desligar um LED, por exemplo). As linhas 389 e 390 definem que as funções `xpad_open` e `xpad_close` do *driver* serão chamadas quando houver uma tentativa de abrir (`open`) ou fechar (`close`) o dispositivo, respectivamente.

7.2.3 Definição dos eventos suportados pelo dispositivo

Nesta etapa, o dispositivo informa ao subsistema Input quais os tipos de eventos que reconhece e quais os códigos relativos a botões ou teclas que poderá gerar.

A Listagem 30 mostra como os eventos são definidos através de iterações sobre os vetores declarados anteriormente (vide Listagem 23). Inicialmente, verifica-se se o dispositivo atual é do tipo *Dance Pad* (linha 398). Em caso positivo, o *driver* informa que serão reconhecidos apenas eventos do tipo EV_KEY e os códigos declarados no vetor `xpad_mat_btn[]` (vide Seção 4.1). Caso não se trate de um *Dance Pad*, os eventos possíveis são definidos, na linha 403, como sendo EV_KEY e EV_ABS. Eventos do tipo EV_KEY representam pressionamentos de teclas ou botões, enquanto eventos do tipo EV_ABS representam movimentações axiais (inexistentes em dispositivos do tipo *Dance Pad*) através de valores absolutos, isto

é, indicam a posição atual de determinado eixo. Em seguida, na linha 405, o vetor `xpad_btn[]` é iterado para que sejam definidos os possíveis códigos de botões gerados pelo dispositivo.

```
398       if (xpad->isMat) {
399             input_dev->evbit[0] = BIT(EV_KEY);
400             for (i = 0; xpad_mat_btn[i] >= 0; ++i)
401                   set_bit(xpad_mat_btn[i], input_dev->keybit);
402       } else {
403             input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
404
405             for (i = 0; xpad_btn[i] >= 0; ++i)
406                   set_bit(xpad_btn[i], input_dev->keybit);
407
408             for (i = 0; xpad_abs[i] >= 0; ++i) {
409
410                   signed short t = xpad_abs[i];
411
412                   set_bit(t, input_dev->absbit);
413
414                   switch (t) {
415                   case ABS_X:
416                   case ABS_Y:
417                   case ABS_RX:
418                   case ABS_RY:      /* the two sticks */
419                         input_set_abs_params(input_dev, t,
420                                     -32768, 32767, 16, 12000);
421                         break;
422                   case ABS_Z: /* left trigger */
423                   case ABS_RZ:      /* right trigger */
424                   case ABS_HAT1X:   /* analogue button A */
425                   case ABS_HAT1Y:   /* analogue button B */
426                   case ABS_HAT2X:   /* analogue button C */
427                   case ABS_HAT2Y:   /* analogue button X */
428                   case ABS_HAT3X:   /* analogue button Y */
429                   case ABS_HAT3Y:   /* analogue button Z */
430                         input_set_abs_params(input_dev, t,
431                                     0, 255, 0, 0);
432                         break;
433                   case ABS_HAT0X:
434                   case ABS_HAT0Y:   /* the d-pad */
435                         input_set_abs_params(input_dev, t,
436                                     -1, 1, 0, 0);
437                         break;
438                   }
439             }
440
441             if (!xpad->is360)
442                   if (xpad_rumble_probe(udev, xpad, ifnum) != 0)
443                         err("could not init rumble");
444       }
```

Listagem 30: Iteração sobre os vetores contendo os códigos de comandos que podem ser emitidos pelo joystick

A última iteração, que inicia na linha 408, ocorre sobre os itens do vetor `xpad_abs[]` e, para este vetor, alguns testes adicionais se fazem necessários. Isto porque este vetor trata de eixos que informam coordenadas absolutas de extremos distintos. Assim, cada grupo de eixos com valores extremos comuns é definido separadamente através da função explicada a seguir:

```
input_set_abs_params(struct input_dev *dev,
                     int axis,
                     int min,
                     int max,
                     int fuzz,
                     int flat)
```

A função define que, para o dispositivo `dev`, o eixo `axis` fornecerá valores no intervalo de `min` a `max` e que cada valor informado poderá apresentar uma imprecisão `fuzz`. O parâmetro `flat` indica o valor que será fornecido pelo dispositivo quando o eixo em questão estiver posicionado em seu centro.

Assim, a linha 419 define que os eixos ABS_X, ABS_Y, ABS_RX e ABS_RY informarão valores no intervalo de `-32768` até `32767`, sendo que cada valor poderá apresentar um desvio de `16` pontos para mais ou para menos e a posição central destes eixos é indicada pelo valor `12000`. A linha 430, por sua vez, define outros eixos cujos valores poderão variar de `0` a `255`. Por fim a linha 435 define eixos que podem apresentar valores variando de `-1` até `1`.

Nas linhas 441 e 442, o *driver* testa se o dispositivo detectado é um Xbox 360 e, em caso negativo, tenta habilitar as funcionalidades de vibração do *joystick*.

7.2.4 Criação do URB

Conforme abordado na Seção 6, a utilização de URBs simplifica de maneira considerável a comunicação com o dispositivo USB. No *driver* xpad, esta também é a forma de comunicação utilizada. A Listagem 31 mostra como o URB responsável por receber os dados enviados pelo dispositivo é criado.

```
447       ep_irq_in = &intf->cur_altsetting->endpoint[0].desc;
448       usb_fill_int_urb(xpad->irq_in, udev,
449                        usb_rcvintpipe(udev, ep_irq_in->bEndpointAddress),
450                        xpad->idata, XPAD_PKT_LEN, xpad_irq_in,
451                        xpad, ep_irq_in->bInterval);
452       xpad->irq_in->transfer_dma = xpad->idata_dma;
453       xpad->irq_in->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

Listagem 31: Declaração do URB para comunicação com o dispositivo

Na linha 448, a função `usb_fill_int_urb` (apresentada na Seção 6.2.1) atribui a `xpad->irq_in` um URB do tipo INTERRUPT configurado para receber informações do dispositivo `udev` (que se refere ao próprio *joystick*) através do *enpoint* número `0` (vide linha 447). O URB utilizará o *buffer* `xpad->idata` e receberá pacotes de tamanho `XPAD_PKT_LEN`. A cada URB que for completado, a função `xpad_irq_in` será chamada para interpretar os dados recebidos. O penúltimo parâmetro atribui ao campo de dados privados do URB o endereço da estrutura `xpad`, para que o *driver* possa acessá-la posteriormente sob demanda na função *callback* `xpad_irq_in`. O último parâmetro configura o URB para que as interrupções aconteçam com a mesma freqüência definida pelo *endpoint* `0`.

A linha 452 informa que o endereço do *buffer* de transferência DMA do URB aponta para o mesmo *buffer* alocado anteriormente em `xpad->idata_dma`.

A linha 453 altera o valor do campo `transfer_flags` habilitando o bit

relativo a `URB_NO_TRANSFER_DMA_MAP`, para que o USB *core* dê preferência à utilização do *buffer* `transfer_dma` ao invés de utilizar `transfer_buffer`.

7.2.5 Registro do dispositivo

Não havendo nenhum erro na inicialização das estruturas, o *driver* pode se registrar no sistema e iniciar o controle do dispositivo. A Listagem 32 mostra o como o *driver* xpad efetua este registro através de uma chamada à função `input_register_device`, passando-lhe como parâmetro a estrutura `xpad->dev`, que neste ponto contém todas as informações necessárias para fazer o gerenciamento do *joystick*.

```
455       error = input_register_device(xpad->dev);
456       if (error)
457             goto fail3;
458
459       usb_set_intfdata(intf, xpad);
```

Listagem 32: Registro do dispositivo no sistema e dados privados do driver

É importante notar que o *driver* se registra no subsistema Input e não no subsistema USB (com a função `usb_register_dev`). Isto ocorre porque o *joystick* é um dispositivo do tipo *input* e os sinais gerados por ele são processados por este subsistema, ainda que o barramento utilizado seja o USB.

Caso algum erro ocorra no processo de registro, o fluxo de código é desviado para o identificador `fail3`, desalocando os recursos previamente alocados.

7.2.6 Dados privados do *driver*

Da mesma forma que ocorreu nas linhas 387 e 451, a linha 459 da Listagem 32 mostra a atribuição dos dados privados do *driver* definidos na estrutura `struct usb_xpad` ao campo reservado para este fim na variável `intf`, recebida pela função `xpad_probe` como parâmetro.

Esta atribuição tem a mesma finalidade das outras duas, isto é, permitir que qualquer função que tenha acesso à interface do dispositivo possa também acessar as propriedades definidas na estrutura `struct usb_xpad`.

7.2.7 Envio de informações ao dispositivo

Por se tratarem de dispositivos de entrada, pode-se pensar que o fluxo de dados neste tipo de dispositivo ocorre sempre em um único sentido: do dispositivo para o computador. Em certos casos, porém, pode ser necessário o envio de informações ao dispositivo para ativar mecanismos de *feedback*, como acionar o gerador de vibrações do *joystick* ou para ligar/desligar o LED que indica se o teclado numérico está ou não habilitado, por exemplo.

Se o tipo de dispositivo conectado for um *joystick* Xbox 360, o *driver* xpad envia, ao reconhecer o dispositivo, um pacote de dados responsável por desligar todos os LEDs indicativos deste *joystick*, conforme mostra a Listagem 33.

```
  461       /* Turn off the LEDs on xpad 360 controllers */
  462       if (xpad->is360) {
  463             char ledcmd[] = {1, 3, 0}; /* The LED-off command for
Xbox-360 controllers */
  464             int j;
  465             usb_bulk_msg(udev, usb_sndintpipe(udev,2), ledcmd, 3, &j,
0);
  466       }
```

Listagem 33: Troca de informações sem a utilização de URBs

O envio deste pacote é feito através de uma chamada à função `usb_bulk_msg`, cuja função é enviar um pacote do tipo BULK a um determinado *endpoint* sem a necessidade de configurar um URB, uma vez que os dados a serem transmitidos neste caso são fixos e não utilizam funções *callback*.

7.2.8 Controle de erros

A última parte da função `xpad_probe` diz respeito ao tratamento de eventuais erros que possam ocorrer durante o processo de detecção do dispositivo. Caso nenhum erro ocorra, o fluxo de execução chegará à linha 468 (mostrada na Listagem 34) e o valor `0`, indicando que nenhum erro ocorreu, será retornado ao USB *core.*

```
  468       return 0;
  469
  470 fail3:      usb_free_urb(xpad->irq_in);
  471 fail2:      usb_buffer_free(udev, XPAD_PKT_LEN, xpad->idata, xpad-
>idata_dma);
  472 fail1:      input_free_device(input_dev);
  473       kfree(xpad);
  474       return error;
  475 }
```

Listagem 34: Controle de erros

Conforme visto nos itens anteriores, cada ocorrência de erro desvia o fluxo de execução para um identificador a partir do qual todas as estruturas previamente alocadas com sucesso são liberadas. A Listagem 34 mostra que a liberação dos recursos sempre ocorre na ordem inversa à ordem de alocação. Conforme já foi dito, esta metodologia facilita o desenvolvimento do *driver,* evitando a necessidade de identações excessivas para controlar o que já foi alocado e tornando o código mais legível.

O retorno da função `xpad_probe`, em caso de erro, é o valor da variável `error`, inicialmente definida com o valor negativo `-ENOMEM` (que indica falta de memória). Ao longo da execução, o valor desta variável vai sendo alterado para o código de erro retornado pelas funções que podem falhar. Desta forma, o valor retornado em caso de erro corresponde sempre ao erro que ocorreu mais recentemente.

7.2.9 Funções open e close

Conforme já comentado na Seção 7.2.2, o acesso aos dispositivos normalmente ocorre da mesma forma que o acesso a arquivos (através de funções `open`, `read`, `write` e `close`). Para permitir esta forma de acesso, os *drivers* associam funções a estas operações (vide linhas 389 e 390 do *driver* xpad). A Listagem 35 e a Listagem 36 descrevem o que ocorre nas funções associadas aos eventos `open` e `close` do *driver,* respectivamente.

```
293 static int xpad_open(struct input_dev *dev)
294 {
295       struct usb_xpad *xpad = dev->private;
296       int status;
297
298       info("opening device");
299
300       xpad->irq_in->dev = xpad->udev;
301       if ((status = usb_submit_urb(xpad->irq_in, GFP_KERNEL))) {
302             err("open input urb failed: %d", status);
303             return -EIO;
304       }
305
306       if(!xpad->is360) {
307             xpad_rumble_open(xpad);
308       }
309
310       return 0;
311 }
```

Listagem 35: Função xpad_open

Como foi definida para ser invocada sempre que uma tentativa de acesso ao dispositivo for feita, a função `xpad_open` tem a incumbência de enviar o URB previamente instanciado pela função `xpad_probe` com o objetivo de iniciar a recepção de eventos oriundos do dispositivo. Isto é feito pela função `usb_submit_urb`.

Na linha 306, o *driver* testa se o dispositivo reconhecido é do tipo Xbox 360 e, caso não seja, a função de ativação do módulo de vibração do dispositivo é também acionada.

```
318 static void xpad_close(struct input_dev *dev)
319 {
320       struct usb_xpad *xpad = input_get_drvdata(dev);
321
322       info("closing device");
323       usb_kill_urb(xpad->irq_in);
324       if(!xpad->is360) {
325             xpad_rumble_close(xpad);
326       }
327 }
```

Listagem 36: Função xpad_close

A função `xpad_close`, por sua vez, tem a finalidade de parar a recepção de informações do dispositivo. Isto é feito através do cancelamento do URB ativo, com a função `usb_kill_urb`. Da mesma forma que a função `xpad_open` iniciava também o módulo de vibração dos *joysticks* que não fossem Xbox 360, a função `xpad_close` deve parar este módulo, como ocorre na linha 325.

Cabe aqui uma observação sobre a linha 320, que é um exemplo de utilização dos dados privados do *driver,* definidos na função `xpad_probe`. Nesta linha, uma nova estrutura `struct usb_xpad` é alocada e passa a apontar para os dados privados do *driver* através de uma chamada à função `input_get_drvdata`.

7.2.10 Processamento das informações recebidas

Uma vez estabelecida a conexão através da função `xpad_open`, o *driver* passa a receber, em intervalos definidos pelo URB do tipo INTERRUPT, pacotes contendo informações sobre eventos gerados pelo dispositivo. Estes pacotes são enviados à função *callback* definida no URB que, no caso do *driver* xpad, corresponde à função `xpad_irq_in`. A Listagem 37 mostra o comportamento desta função *callback*.

```
257 static void xpad_irq_in(struct urb *urb)
258 {
259       struct usb_xpad *xpad = urb->context;
260       int retval;
261
262       switch (urb->status) {
263       case 0:
264             /* success */
265             break;
266       case -ECONNRESET:
267       case -ENOENT:
268       case -ESHUTDOWN:
269             /* this urb is terminated, clean up */
270             dbg("%s - urb shutting down with status: %d",
271                 __FUNCTION__, urb->status);
272             return;
273       default:
274             dbg("%s - nonzero urb status received: %d",
275                 __FUNCTION__, urb->status);
276             goto exit;
277       }
278
279       xpad_process_packet(xpad, 0, xpad->idata);
280
281 exit:
282       retval = usb_submit_urb(urb, GFP_ATOMIC);
283       if (retval)
284             err("%s - usb_submit_urb failed with result %d",
285                 __FUNCTION__, retval);
286 }
```

Listagem 37: Função chamada a cada URB recebido

Uma das primeiras ações a serem tomadas no recebimento de um URB é testar o campo `status` para verificar a possível ocorrência de erros (vide Seção 6 para uma lista dos possíveis códigos de erro). Caso algum erro irrecuperável tenha ocorrido, uma mensagem de erro é gerada e a função retorna sem tomar nenhuma atitude. Caso o erro possa ser contornado, o fluxo de código é desviado para o identificador `exit` e o *driver* tentará enviar o URB novamente. Se o URB for recebido com sucesso, a função `xpad_process_packet` é invocada para identificar os eventos recebidos e informar tais eventos ao subsistema Input. Esta função é mostrada de forma reduzida na Listagem 38, uma vez que o código é bastante repetitivo.

```
  155 static void xpad_process_packet(struct usb_xpad *xpad, u16 cmd,
unsigned char *data)
  156 {
  157       struct input_dev *dev = xpad->dev;
  158       int i;
  159
  160       if(debug) {
  161             printk(KERN_INFO "xpad_debug: data :");
  162             for(i = 0; i < 20; i++) {
  163                   printk("0x%02x ", data[i]);
  164             }
  165             printk("\n");
  166       }
   .
   .
   .
  248       input_sync(dev);
  249 }
```

Listagem 38: Processamento do URB recebido

Na linha 160, o *driver* verifica se o parâmetro `debug` foi informado no momento da carga do módulo. Em caso afirmativo, todos os bytes contidos no *buffer* do URB são exibidos através de chamadas à função `printk`.

O código presente entre as linhas 167 e 249 (ANEXO I) testa todos os eventos que possam ter sido enviados pelo dispositivo. O subsistema Input provê funções específicas para o tratamento de cada tipo de evento, sendo que os protótipos das funções deste tipo utilizadas pelo *driver* são muito semelhantes, como se pode verificar a seguir:

```
input_report_key(struct input_dev *dev,
                 unsigned int code,
                 int value)
```

Esta função relata ao subsistema Input a ocorrência de um evento do tipo `EV_KEY`. O parâmetro `code` contém o código da tecla ou botão e o parâmetro `value` informa o estado atual deste botão.

```
input_report_rel(struct input_dev *dev,
                 unsigned int code,
```

```
                    int value)
```

Esta função relata ao subsistema Input a ocorrência de um evento do tipo EV_REL. O parâmetro code contém o código que identifica onde ocorreu o movimento relativo e o parâmetro value informa qual foi o deslocamento que ocorreu.

```
input_report_abs(struct input_dev *dev,
                 unsigned int code,
                 int value)
```

Esta função relata ao subsistema Input a ocorrência de um evento do tipo EV_ABS. O parâmetro code contém o código que identifica onde ocorreu o movimento absoluto e o parâmetro value informa qual o valor deste movimento.

Por fim, a linha 248 exibe a chamada da função input_sync, que informa ao subsistema Input que todos os eventos possíveis foram reportados. Esta chamada é importante para que as ações detectadas ocorram de maneira simultânea e não separadamente. Este comportamento é especialmente esperado, por exemplo, em movimentações relativas e absolutas, uma vez que, se não fossem sincronizadas, poderiam acontecer primeiro em uma direção e depois na outra, quando o esperado seria uma movimentação diagonal.

7.2.11 Desconexão do dispositivo

A fase de desconexão do dispositivo é invocada quando o USB *core* detecta que o dispositivo não está mais presente no sistema. No caso do *driver* xpad, a função associada a este evento é a função xpad_disconnect, exibida na Listagem 39.

```
483 static void xpad_disconnect(struct usb_interface *intf)
484 {
485       struct usb_xpad *xpad = usb_get_intfdata(intf);
486
487       usb_set_intfdata(intf, NULL);
488       if (xpad) {
489             usb_kill_urb(xpad->irq_in);
490             if(!xpad->is360) {
491                   xpad_rumble_close(xpad);
492             }
493             input_unregister_device(xpad->dev);
494
495             usb_free_urb(xpad->irq_in);
496
497             usb_buffer_free(interface_to_usbdev(intf), XPAD_PKT_LEN,
498                         xpad->idata, xpad->idata_dma);
499
500             if(!xpad->is360) {
501                   xpad_rumble_disconnect(xpad);
502             }
503
504             kfree(xpad);
505       }
506 }
```

Listagem 39: Desconexão do dispositivo USB

O objetivo desta função é liberar os recursos alocados para manter o dispositivo funcionando, parando eventuais URBs que estejam ainda em andamento, liberando os *buffers* alocados e as estruturas previamente registradas.

7.3 Comunicação com o espaço do usuário

Conforme citado no Capítulo 5, a comunicação dos dispositivos com o espaço do usuário - e conseqüentemente com as aplicações - se dá por meio de entradas nos diretórios `/proc`, `/sys` e `/dev` do Linux. Dispositivos do tipo *joystick* são representados no diretório `/dev/input` por arquivos que seguem a nomenclatura `js<n>`, onde `<n>` é um número inteiro que inicia em `0` e cresce seqüencialmente conforme mais *joysticks* são adicionados. Como apenas um

joystick foi utilizado para o estudo de caso, o dispositivo foi associado à entrada `/dev/input/js0` (LINUX KERNEL ORGANIZATION, 2001).

Através de leituras no arquivo `/dev/input/js0`, os eventos gerados pelo *joystick* podem ser capturados na forma especificada pela API de dispositivos do tipo *joystick*, que se resume à estrutura apresentada e explicada a seguir.

```
struct js_event {
    __u32 time;     /* event timestamp in milliseconds */
    __s16 value;    /* value */
    __u8 type;      /* event type */
    __u8 number;    /* axis/button number */
};
```

`time`: Utilizado para ordenar as mensagens enviadas pelo *joystick* de forma cronológica.

`type`: Indica o tipo de evento gerado, podendo assumir os valores `0x01` (representando que um botão foi pressionado ou solto) e `0x02` (indicando que um eixo foi movido). No momento em que a primeira leitura é realizada, o campo `type` assume o valor `0x80`, indicando que as informações que estão sendo lidas representam os valores de estado inicial do *joystick*. No caso de informações de estado inicial de botões, uma operação de `OR` bit a bit é realizada entre os valores `0x80` e `0x01`, fazendo com que o campo `type` assuma o valor `0x81`. O mesmo ocorre quando são obtidos os valores de estado inicial para eixos, porém a operação de `OR` bit a bit ocorre entre os valores `0x80` e `0x02`, resultando no valor `0x82`.

`number`: Representa o número do botão ou eixo do *joystick* que gerou o evento.

`value`: Indica o valor relativo ao evento gerado. O campo `value` pode assumir até $2^{16}=65536$ valores distintos. No entanto, costuma-se utilizar valores `0` e `1` para representar o estado de botões (indicando não pressionamento e pressionamento, respectivamente) e valores variáveis apenas para movimentações axiais.

O código fonte exibido no APÊNDICE I exemplifica a conexão de uma aplicação com o dispositivo através da entrada atribuída a tal dispositivo no diretório `/dev`. A saída fornecida pela aplicação assemelha-se à da Listagem 40.

```
alexsmith@ruapehu:~/modules/xpad$ ./usedev
Timestamp: 5092736 Value:      1 Type: 0x81 Number: 0
Timestamp: 5092736 Value:      1 Type: 0x81 Number: 1
Timestamp: 5092736 Value:      1 Type: 0x81 Number: 2
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 3
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 4
Timestamp: 5092736 Value:      1 Type: 0x81 Number: 5
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 6
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 7
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 8
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 9
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 10
Timestamp: 5092736 Value:      1 Type: 0x81 Number: 11
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 12
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 13
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 14
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 15
Timestamp: 5092736 Value:      0 Type: 0x81 Number: 16
Timestamp: 5092736 Value:      0 Type: 0x82 Number: 0
Timestamp: 5092736 Value:      0 Type: 0x82 Number: 1
Timestamp: 5092736 Value:    258 Type: 0x82 Number: 2
Timestamp: 5092736 Value: -32767 Type: 0x82 Number: 3
Timestamp: 5092736 Value: -32767 Type: 0x82 Number: 4
Timestamp: 5092736 Value:    258 Type: 0x82 Number: 5
Timestamp: 5092736 Value:      0 Type: 0x82 Number: 6
Timestamp: 5092736 Value:      0 Type: 0x82 Number: 7
Timestamp: 5092736 Value:  32767 Type: 0x82 Number: 8
Timestamp: 5092736 Value: -28898 Type: 0x82 Number: 9
Timestamp: 5092736 Value:    258 Type: 0x82 Number: 10
Timestamp: 5092736 Value: -32767 Type: 0x82 Number: 11
Timestamp: 5092736 Value: -32767 Type: 0x82 Number: 12
Timestamp: 5092736 Value:    258 Type: 0x82 Number: 13
```

Listagem 40: Início da comunicação com o dispositivo joystick

As informações de inicialização permitem que a aplicação identifique as

capacidades do dispositivo, tais como número de eixos, número de botões e valores iniciais. Em seguida, ao pressionar um botão ou movimentar um eixo, são gerados eventos, conforme mostrado na Listagem 41.

```
Timestamp: 5426512 Value:      1 Type: 0x01 Number: 3
Timestamp: 5427120 Value:      0 Type: 0x01 Number: 3
Timestamp: 5460992 Value:      0 Type: 0x01 Number: 1
Timestamp: 5461376 Value:      1 Type: 0x01 Number: 1
Timestamp: 5491620 Value:  -5161 Type: 0x02 Number: 10
Timestamp: 5492276 Value: -21674 Type: 0x02 Number: 10
Timestamp: 5492436 Value: -23480 Type: 0x02 Number: 10
Timestamp: 5492916 Value: -32767 Type: 0x02 Number: 10
Timestamp: 5498996 Value: -23480 Type: 0x02 Number: 10
Timestamp: 5499300 Value: -21674 Type: 0x02 Number: 10
Timestamp: 5499540 Value: -19352 Type: 0x02 Number: 10
Timestamp: 5500052 Value:  -9289 Type: 0x02 Number: 10
Timestamp: 5500356 Value:  -1033 Type: 0x02 Number: 10
```

Listagem 41: Exibição de eventos gerados pelo joystick

A listagem mostra que inicialmente foram pressionados dois botões que foram identificados pelos números `3` e `1`. Os eventos gerados são do tipo `JS_EVENT_BUTTON` e pode-se notar que o comportamento dos botões é diferente: para o botão `3`, o pressionamento gera o valor `0` e o não pressionamento gera o valor `1`. O comportamento do botão `1` é exatamente oposto.

Em seguida, outra movimentação foi realizada, desta vez sobre um eixo, conforme indicado pelo valor `0x02`, que corresponde à constante `JS_EVENT_AXIS`. Pode-se notar que o eixo foi movimentado desde seu ponto central até o valor mínimo e, posteriormente, deste valor mínimo até retornar ao seu ponto central.

Percebe-se, portanto, que a interface de comunicação com o dispositivo é extremamente simples e padronizada, facilitando de forma significativa o desenvolvimento de aplicações.

7.4 Mensagens geradas quando o dispositivo é removido

Ao remover-se o dispositivo, a seguinte linha é exibida pelo comando `dmesg`:

```
[ 4901.244000] usb 2-1: USB disconnect, address 2
```

Esta linha informa que houve uma desconexão de um dispositivo USB que estava ligado ao *hub* mestre 2, em sua porta número 1, com endereço 2. Neste instante, todas as estruturas previamente alocadas para garantir o funcionamento do dispositivo são liberadas[5].

[5] Na verdade, a liberação dos recursos alocados deve ser realizada de forma explícita pelos *drivers*, conforme demonstrado na Seção 7.2.11, pois não ocorre automaticamente.

8 CONCLUSÃO

Este trabalho apresentou um estudo sobre os procedimentos envolvidos no desenvolvimento de gerenciadores de dispositivos USB para o *kernel* do sistema operacional Linux.

Conforme se pôde constatar através das pesquisas realizadas, a especificação USB vem sendo adotada por um número cada vez maior de fabricantes de periféricos, consolidando este padrão de comunicação no mercado. Este fato comprova a relevância da pesquisa realizada, que pretende servir como material de apoio para desenvolvedores que desejem conhecer os procedimentos envolvidos na comunicação com dispositivos USB em plataforma Linux.

A realização do trabalho possibilitou também a constatação de que algumas partes do *kernel* do Linux, a exemplo do USB *core*, encontram-se muito bem documentadas. Já outras partes, em especial o subsistema Input, possuem uma documentação deficitária, o que requereu a busca por fontes alternativas de pesquisa e estudos aprofundados do código fonte em si para compreensão de seu funcionamento.

Além disso, os estudos contribuíram para os objetivos pessoais do autor,

possibilitando a ampliação dos conhecimentos relativos ao sistema operacional Linux, destacando-se as áreas de programação em *kernel* e desenvolvimento de gerenciadores de dispositivos.

Considera-se que o objetivo proposto tenha sido atingido. Além disso, o estudo de caso escolhido (*driver* xpad) se mostrou importante para demonstrar a aplicação dos conceitos citados, visto que praticamente todas as etapas envolvidas no desenvolvimento de *drivers* para dispositivos USB, desde detecção, registro e operação até a desconexão do dispositivo, foram devidamente documentadas no referencial teórico.

É importante ressaltar que, por ter sido escrito na língua portuguesa, este documento possibilita a expansão da comunidade brasileira - ainda bastante pequena - de desenvolvedores de *drivers*, considerando que, em muitos casos, a língua pode representar um empecilho significativo para que isto aconteça.

Espera-se, portanto, que a publicação deste trabalho possa significar uma contribuição importante para que ainda mais dispositivos USB passem a ser compatíveis com o sistema operacional Linux, aumentando a usabilidade deste sistema operacional e tornando-o mais popular.

Pretende-se, posteriormente à conclusão deste trabalho, utilizar a base de conhecimento adquirida para unir esforços com a comunidade de desenvolvedores de *drivers* para o *kernel*, de forma a poder contribuir para ampliação do suporte à dispositivos USB.

REFERÊNCIAS

BOVET, Daniel P; CESATI, Marco. **Understanding the Linux Kernel**. 3. ed. Sebastopol, CA, Estados Unidos: O'Reilly Media, Inc., 2006.

CORBET, Jonathan; RUBINI, Alessandro; KROAH-HARTMAN, Greg. **Linux Device Drivers**. 3 ed. Sebastopol, CA, Estados Unidos: O'Reilly Media, Inc., 2005.

Free Software Foundation, Inc. **The GNU Make Manual**. Estados Unidos, out. 2007. Disponível em: <http://www.gnu.org/software/make/manual/html_node/index.html>. Acesso em 17 out. 2007.

HASAN, Ragib. **History of Linux**. Illinois, Estados Unidos, out. 2005. Disponível em: <https://netfiles.uiuc.edu/rhasan/linux/>. Acesso em 10 abr 2007.

KROAH-HARTMANN, Greg. **udev - A Userspace Implementation of devfs**. Canadá, jul. 2003. Disponível em: <http://www.kroah.com/linux/talks/ols_2003_udev_paper/Reprint-Kroah-Hartman-OLS2003.pdf>. Acesso em 9 nov. 2007.

Linux Kernel Organization, Inc. **Joystick API Documentation v1.2**. Estados Unidos, mai. 2001. Disponível em <http://www.kernel.org/doc/Documentation/input/joystick-api.txt>. Acesso em 15 dez. 2007.

Linux Kernel Organization, Inc. **Linux Input drivers v1.0**. Estados Unidos, mai. 2002. Disponível em <http://www.kernel.org/doc/Documentation/input/input.txt>. Acesso em 8 nov. 2007.

Linux Kernel Organization, Inc. **The Linux-USB Host Side API**. Estados Unidos, jun. 2007. Disponível em: <http://kernel.org/doc/htmldocs/usb.html>. Acesso em 25 jul. 2007.

LOVE, Robert. **Linux Kernel Development**. 2. ed. Indianápolis, IN, Estados Unidos: Novel Press, 2005.

MOCHEL, Patrick. **The sysfs filesystem**. Estados Unidos, out. 2005. Disponível em: <http://www.kernel.org/pub/linux/kernel/people/mochel/doc/papers/ols-2005/mochel.pdf>. Acesso em 17 out. 2007.

OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas Operacionais**. 2. ed. Porto Alegre, RS, Brasil: Editora Sagra Luzzatto, 2001.

RUSLING, David A. **The Linux Kernel**. United Kingdom, 1999. Disponível em: <http://www.tldp.org/LDP/tlk/>. Acesso em 24 jun. 2007.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 2. ed. São Paulo, SP, Brasil: Prentice Hall, 2003.

USB Implementers Forum, Inc. **Universal Serial Bus Specification**. Estados Unidos, 2000. Disponível em: <http://www.usb.org/developers/docs/usb_20_040907.zip>. Acesso em 15 mar. 2007.

USB Implementers Forum, Inc. **Device Class Definition for Human Interface Devices (HID)**. Estados Unidos, 2001. Disponível em: <http://www.usb.org/developers/devclass_docs/HID1_11.pdf>. Acesso em 8 nov. 2007.

USB Implementers Forum, Inc. **USB Class Codes**. Estados Unidos, 2006. Disponível em: <http://www.usb.org/developers/defined_class>. Acesso em 25 nov. 2007.

WIRZENIUS, Lars. **Linux: the big picture**. Austrália, abr. 2003. Disponível em: <http://liw.iki.fi/liw/texts/linux-the-big-picture.html>. Acesso em 17 abr. 2007.

APÊNDICE I - CÓDIGO FONTE DE ACESSO AO DISPOSITIVO

Arquivo `usedev.c`, desenvolvido pelo autor para testar o acesso ao dispositivo *joystick* através do arquivo `/dev/input/js0`. Para compilar o código fonte, foi utilizado o compilador gcc, invocado da seguinte forma: `gcc -Wall -std=c99 usedev.c -o usedev`

```
1  /* for printf */
2  #include <stdio.h>
3  /* for open */
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  /* for read, write and close */
8  #include <unistd.h>
9  /* for arch dependent types */
10 #include <asm/types.h>
11
12 /* path to joystick device */
13 #define DEV "/dev/input/js0"
14
15 /* Joystick API structures */
16 #define JS_EVENT_BUTTON         0x01    /* button pressed/released */
17 #define JS_EVENT_AXIS           0x02    /* joystick moved */
18 #define JS_EVENT_INIT           0x80    /* initial state of device */
19
20 struct js_event {
21     __u32 time;     /* event timestamp in milliseconds */
22     __s16 value;    /* value */
23     __u8 type;      /* event type */
24     __u8 number;    /* axis/button number */
25 };
26
27 int main (int argc, char *argv[]) {
28     int f; /* file pointer */
```

```
29      size_t bytes_read = 0;
30
31        struct js_event e;
32      /* Open device for read only access */
33      f = open(DEV, O_RDONLY);
34
35      if ( f < 0 ) {
36          printf("Error %i while trying to open file.\n", f);
37          return 1;
38      }
39
40      while (1) {
41          bytes_read = read(f, &e, sizeof(struct js_event));
42          if ( bytes_read < 0 ) {
43              printf("Read error %i\n", bytes_read);
44              break;
45          }
46
47          printf("Timestamp: %7i Value: %6i Type: 0x%02x Number:
%1i\n", e.time, e.value, e.type, e.number);
48      }
49
50      close(f);
51      return 0;
52  }
```

ANEXO I - CÓDIGO FONTE DO DRIVER XPAD

Arquivo /drivers/input/joystick/xpad.h:

```
 1 /*
 2  * Xbox Controller driver for Linux - v0.1.5
 3  *
 4  *    header file containing ioctl definitions
 5  *
 6  * Copyright (c)  2003  Marko Friedemann <mfr@bmx-chemnitz.de>
 7  *
 8  *
 9  * This program is free software; you can redistribute it and/or
10  * modify it under the terms of the GNU General Public License as
11  * published by the Free Software Foundation; either version 2 of
12  * the License, or (at your option) any later version.
13  *
14  * This program is distributed in the hope that it will be useful,
15  * but WITHOUT ANY WARRANTY; without even the implied warranty of
16  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
17  * GNU General Public License for more details.
18  *
19  * You should have received a copy of the GNU General Public License
20  * along with this program; if not, write to the Free Software
21  * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307 USA
22  */
23
24 #ifndef __XPAD_h
25 #define __XPAD_h
26
27
28 /*********** ioctl stuff, can be used outside of the driver
***********/
29 #define USB_XPAD_IOC_MAGIC    'x'
30
31 #define USB_XPAD_IOCRESET     _IO(  USB_XPAD_IOC_MAGIC, 0 )
```

```
32 #define USB_XPAD_IOCSRUMBLE   _IOW( USB_XPAD_IOC_MAGIC, 3, int )
33 #define USB_XPAD_IOCGRUMBLE   _IOR( USB_XPAD_IOC_MAGIC, 4, int )
34
35 #define USB_XPAD_IOCSIR       _IOW( USB_XPAD_IOC_MAGIC, 5, int )
36 #define USB_XPAD_IOCGIR       _IOR( USB_XPAD_IOC_MAGIC, 6, int )
37
38 #define USB_XPAD_IOC_MAXNR    6
39
40 /************************* driver internals
**************************/
41 #ifdef __KERNEL__
42
43 #include <linux/input.h>
44 #include <linux/circ_buf.h>
45
46 /****************** driver description and version
********************/
47 #define DRIVER_VERSION        "v0.1.6"
48 #define DRIVER_AUTHOR         "Marko Friedemann <mfr@bmx-
chemnitz.de>,\
49  Oliver Schwartz <Oliver.Schwartz@gmx.de>, Georg Lukas <georg@op-
co.de>,\
50  Thomas Pedley <Gentoox@shallax.com>, Edgar Hucek <hostmaster@ed-
soft.at>"
51
52 #define DRIVER_DESC           "driver for Xbox controllers"
53
54 /****************************** constants
******************************/
55 #define XPAD_MAX_DEVICES      4
56 #define XPAD_PKT_LEN          32    /* input packet size */
57 #define XPAD_PKT_LEN_FF       6     /* output packet size - rumble */
58
59 #define XPAD_TX_BUFSIZE       XPAD_PKT_LEN_FF * 8     /* max. 8
requests */
60
61 /************************* the device struct
*************************/
62 struct usb_xpad {
63       struct input_dev *dev;              /* input device interface */
64       struct usb_device *udev;            /* usb device */
65
66       struct urb *irq_in;                 /* urb for int. in report */
67       unsigned char *idata;               /* input data */
68       dma_addr_t idata_dma;
69
70       char phys[65];                      /* physical input dev path
*/
71
72       unsigned char offsetset_compensation;
73       int left_offset_x;
74       int left_offset_y;
75       int right_offset_x;
76       int right_offset_y;
77
78       int isMat;                    /* is this a dancepad/mat? */
79       int is360;                    /* is this a Xbox 360 Controller
*/
80
```

```
   81 #ifdef CONFIG_USB_XPAD_RUMBLE
   82       int rumble_enabled;               /* ioctl can toggle rumble
*/
   83
   84       int ep_out_adr;                   /* number of out endpoint */
   85       unsigned char tx_data[XPAD_PKT_LEN_FF];  /* output data
(rumble) */
   86       int strong_rumble, play_strong;          /* strong rumbling */
   87       int weak_rumble, play_weak;       /* weak rumbling */
   88       struct timer_list rumble_timer;          /* timed urb out retry
*/
   89       wait_queue_head_t wait;           /* wait for URBs on queue */
   90
   91       spinlock_t tx_lock;
   92       struct circ_buf tx;
   93       unsigned char tx_buf[XPAD_TX_BUFSIZE];
   94       long tx_flags[1];                 /* transmit flags */
   95 #endif
   96 };
   97
   98 /* for the list of know devices */
   99 struct xpad_device {
  100       u16 idVendor;
  101       u16 idProduct;
  102       u8  isMat;
  103       char *name;
  104       u8  is360;
  105 };
  106
  107
  108 /************************ rumble function stubs
***********************/
  109 #ifndef CONFIG_USB_XPAD_RUMBLE
  110  #define xpad_rumble_ioctl(dev, cmd, arg) -ENOTTY
  111  #define xpad_rumble_open(xpad) {}
  112  #define xpad_rumble_probe(udev, xpad, ifnum) 0
  113  #define xpad_rumble_close(xpad) {}
  114  #define xpad_rumble_disconnect(xpad) {}
  115 #else /* CONFIG_USB_XPAD_RUMBLE */
  116
  117  #define XPAD_TX_RUNNING     0
  118  #define XPAD_TX_INC(var, n) (var) += n; (var) %= XPAD_TX_BUFSIZE
  119
  120  #ifndef __USB_XPAD_RUMBLE
  121   extern int  xpad_rumble_ioctl(struct input_dev *dev, unsigned int
cmd, unsigned long arg);
  122   extern void xpad_rumble_open(struct usb_xpad *xpad);
  123   extern int  xpad_rumble_probe(struct usb_device *udev, struct
usb_xpad *xpad, unsigned int ifnum);
  124   extern void xpad_rumble_close(struct usb_xpad *xpad);
  125   extern void xpad_rumble_disconnect(struct usb_xpad *xpad);
  126  #endif /* __USB_XPAD_RUMBLE */
  127 #endif /* CONFIG_USB_XPAD_RUMBLE */
  128
  129 #endif /* __KERNEL__ */
  130
  131 #endif /* __XPAD_h */
```

Arquivo /drivers/input/joystick/xpad.c:

```
/*
 * Xbox input device driver for Linux - v0.1.6
 *
 * Copyright (c)  2002 - 2004  Marko Friedemann <mfr@bmx-chemnitz.de>
 *
 *    Contributors:
 *          Vojtech Pavlik <vojtech@suse.sz>,
 *          Oliver Schwartz <Oliver.Schwartz@gmx.de>,
 *          Thomas Pedley <gentoox@shallax.com>,
 *          Steven Toth <steve@toth.demon.co.uk>,
 *          Franz Lehner <franz@caos.at>,
 *          Ivan Hawkes <blackhawk@ivanhawkes.com>
 *          Edgar Hucek <hostmaster@ed-soft.at>
 *          Niklas Lundberg <niklas@jahej.com>
 *
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
02111-1307 USA
 *
 *
 * This driver is based on:
 *  - information from     http://euc.jp/periphs/xbox-
controller.en.html
 *  - the iForce driver    drivers/char/joystick/iforce.c
 *  - the skeleton-driver  drivers/usb/usb-skeleton.c
 *
 * Thanks to:
 *  - ITO Takayuki for providing essential xpad information on his
website
 *  - Vojtech Pavlik     - iforce driver / input subsystem
 *  - Greg Kroah-Hartman - usb-skeleton driver
 *
 * TODO:
 *  - fine tune axes
 *  - NEW: Test right thumb stick Y-axis to see if it needs flipping.
 *  - NEW: get rumble working correctly, fix all the bugs and support
multiple
 *         simultaneous effects
 *  - NEW: split funtionality mouse/joustick into two source files
 *  - NEW: implement /proc interface (toggle mouse/rumble
enable/disable, etc.)
 *  - NEW: implement user space daemon application that handles that
interface
 *
 * History: moved to end of file
 */

```

```
   54 #include <linux/kernel.h>
   55 #include <linux/init.h>
   56 #include <linux/slab.h>
   57 #include <linux/module.h>
   58 #include <linux/smp_lock.h>
   59 #include <linux/usb.h>
   60 #include <linux/version.h>
   61 #include <linux/usb/input.h>
   62 #include <linux/timer.h>
   63 #include <asm/uaccess.h>
   64
   65 #include "xpad.h"
   66
   67 static unsigned long debug = 0;
   68 module_param(debug, ulong, 0444);
   69 MODULE_PARM_DESC(debug, "Debugging");
   70
   71 static const struct xpad_device xpad_device[] = {
   72       /* please keep those ordered wrt. vendor/product ids
   73         vendor, product, isMat, name, is360                */
   74       { 0x044f, 0x0f07, 0, "Thrustmaster, Inc. Controller", 0},
   75       { 0x045e, 0x0202, 0, "Microsoft Xbox Controller", 0},
   76       { 0x045e, 0x0285, 0, "Microsoft Xbox Controller S", 0},
   77       { 0x045e, 0x0287, 0, "Microsoft Xbox Controller S", 0},
   78       { 0x045e, 0x0289, 0, "Microsoft Xbox Controller S", 0}, /*
microsoft is stupid */
   79       { 0x045e, 0x028e, 0, "Microsoft Xbox 360 Controller", 1},
   80       { 0x046d, 0xca84, 0, "Logitech Xbox Cordless Controller", 0},
   81       { 0x046d, 0xca88, 0, "Logitech Compact Controller for Xbox",
0},
   82       { 0x05fd, 0x1007, 0, "???Mad Catz Controller???", 0}, /*
CHECKME: this seems strange */
   83       { 0x05fd, 0x107a, 0, "InterAct PowerPad Pro", 0},
   84       { 0x0738, 0x4516, 0, "Mad Catz Control Pad", 0},
   85       { 0x0738, 0x4522, 0, "Mad Catz LumiCON", 0},
   86       { 0x0738, 0x4526, 0, "Mad Catz Control Pad Pro", 0},
   87       { 0x0738, 0x4536, 0, "Mad Catz MicroCON", 0},
   88       { 0x0738, 0x4540, 1, "Mad Catz Beat Pad", 0},
   89       { 0x0738, 0x4556, 0, "Mad Catz Lynx Wireless Controller", 0},
   90       { 0x0738, 0x6040, 1, "Mad Catz Beat Pad Pro", 0},
   91       { 0x0c12, 0x8802, 0, "Zeroplus Xbox Controller", 0},
   92       { 0x0c12, 0x8809, 0, "Level Six Xbox DDR Dancepad", 0},
   93       { 0x0c12, 0x8810, 0, "Zeroplus Xbox Controller", 0},
   94       { 0x0c12, 0x9902, 0, "HAMA VibraX - *FAULTY HARDWARE*", 0}, /*
these are broken */
   95       { 0x0e4c, 0x1097, 0, "Radica Gamester Controller", 0},
   96       { 0x0e4c, 0x2390, 0, "Radica Games Jtech Controller", 0},
   97       { 0x0e6f, 0x0003, 0, "Logic3 Freebird wireless Controller", 0},
   98       { 0x0e6f, 0x0005, 0, "Eclipse wireless Controller", 0},
   99       { 0x0e6f, 0x0006, 0, "Edge wireless Controller", 0},
  100       { 0x0e6f, 0x000c, 0, "PELICAN PL-2047", 0},
  101       { 0x0e8f, 0x0201, 0, "SmartJoy Frag Xpad/PS2 adaptor", 0},
  102       { 0x0f30, 0x0202, 0, "Joytech Advanced Controller", 0},
  103       { 0x0f30, 0x8888, 0, "BigBen XBMiniPad Controller", 0},
  104       { 0x102c, 0xff0c, 0, "Joytech Wireless Advanced Controller",
0},
  105       { 0x12ab, 0x8809, 1, "Xbox DDR dancepad", 0},
  106       { 0xffff, 0xffff, 0, "Chinese-made Xbox Controller", 0}, /* WTF
are device IDs for? */
```

```
107         { 0x0000, 0x0000, 0, "nothing detected - FAIL", 0}
108 };
109
110 static const signed short xpad_btn[] = {
111         BTN_A, BTN_B, BTN_C, BTN_X, BTN_Y, BTN_Z,/* analogue buttons */
112         BTN_START, BTN_BACK, BTN_THUMBL, BTN_THUMBR,    /*
start/back/sticks */
113         BTN_0, BTN_1, BTN_2, BTN_3,                 /* d-pad as buttons */
114         BTN_TL, BTN_TR,                             /* Button LB/RB */
115         BTN_MODE,                               /* The big X */
116         -1                                      /* terminating entry */
117 };
118
119 static const signed short xpad_mat_btn[] = {
120         BTN_A, BTN_B, BTN_X, BTN_Y,   /* A, B, X, Y */
121         BTN_START, BTN_BACK,          /* start/back */
122         BTN_0, BTN_1, BTN_2, BTN_3,   /* directions */
123         -1                        /* terminating entry */
124 };
125
126 static const signed short xpad_abs[] = {
127         ABS_X, ABS_Y,             /* left stick */
128         ABS_RX, ABS_RY,           /* right stick */
129         ABS_Z, ABS_RZ,            /* triggers left/right */
130         ABS_HAT0X, ABS_HAT0Y,     /* digital pad */
131         ABS_HAT1X, ABS_HAT1Y,     /* analogue buttons A + B */
132         ABS_HAT2X, ABS_HAT2Y,     /* analogue buttons C + X */
133         ABS_HAT3X, ABS_HAT3Y,     /* analogue buttons Y + Z */
134         -1                  /* terminating entry */
135 };
136
137 static struct usb_device_id xpad_table [] = {
138         { USB_INTERFACE_INFO('X', 'B', 0) },      /* Xbox USB-IF not
approved class */
139         { USB_INTERFACE_INFO( 3 ,  0 , 0) },      /* for Joytech
Advanced Controller */
140         { USB_INTERFACE_INFO( 255 ,  93 , 1) }, /* Xbox 360 */
141         { }
142 };
143
144 MODULE_DEVICE_TABLE(usb, xpad_table);
145
146 /**
147  *    xpad_process_packet
148  *
149  *    Completes a request by converting the data into events
150  *    for the input subsystem.
151  *
152  *    The report descriptor was taken from ITO Takayukis website:
153  *     http://euc.jp/periphs/xbox-controller.en.html
154  */
155 static void xpad_process_packet(struct usb_xpad *xpad, u16 cmd,
unsigned char *data)
156 {
157         struct input_dev *dev = xpad->dev;
158         int i;
159
160         if(debug) {
161               printk(KERN_INFO "xpad_debug: data :");
```

```
  162             for(i = 0; i < 20; i++) {
  163                   printk("0x%02x ", data[i]);
  164             }
  165             printk("\n");
  166       }
  167
  168       /* digital pad (button mode) bits (3 2 1 0) (right left down
up) */
  169       input_report_key(dev, BTN_0, (data[2] & 0x01));
  170       input_report_key(dev, BTN_1, (data[2] & 0x08) >> 3);
  171       input_report_key(dev, BTN_2, (data[2] & 0x02) >> 1);
  172       input_report_key(dev, BTN_3, (data[2] & 0x04) >> 2);
  173
  174       /* start and back buttons */
  175       input_report_key(dev, BTN_START, (data[2] & 0x10) >> 4);
  176       input_report_key(dev, BTN_BACK, (data[2] & 0x20) >> 5);
  177
  178       /* stick press left/right */
  179       input_report_key(dev, BTN_THUMBL, (data[2] & 0x40) >> 6);
  180       input_report_key(dev, BTN_THUMBR, data[2] >> 7);
  181
  182       /* buttons A, B, X, Y digital mode */
  183       if(xpad->is360) {
  184             input_report_key(dev, BTN_A, (data[3] & 0x10) >> 4);
  185             input_report_key(dev, BTN_B, (data[3] & 0x20) >> 5);
  186             input_report_key(dev, BTN_X, (data[3] & 0x80) >> 7);
  187             input_report_key(dev, BTN_Y, (data[3] & 0x40) >> 6);
  188             input_report_key(dev, BTN_TL, data[3] & 0x01 );
  189             input_report_key(dev, BTN_TR, (data[3] & 0x02) >> 1);
  190             input_report_key(dev, BTN_MODE, (data[3] & 0x04) >> 2);
  191       } else {
  192             input_report_key(dev, BTN_A, data[4]);
  193             input_report_key(dev, BTN_B, data[5]);
  194             input_report_key(dev, BTN_X, data[6]);
  195             input_report_key(dev, BTN_Y, data[7]);
  196       }
  197
  198       if (xpad->isMat)
  199             return;
  200
  201       /* left stick (Y axis needs to be flipped) */
  202       if(xpad->is360) {
  203             input_report_abs(dev, ABS_X, (__s16)(((__s16)data[7] <<
8) | (__s16)data[6]));
  204             input_report_abs(dev, ABS_Y, ~(__s16)(((__s16)data[9] <<
8) | data[8]));
  205       } else {
  206             input_report_abs(dev, ABS_X, (__s16)(((__s16)data[13] <<
8) | (__s16)data[12]));
  207             input_report_abs(dev, ABS_Y, ~(__s16)(((__s16)data[15] <<
8) | data[14]));
  208       }
  209
  210       /* right stick */
  211       if(xpad->is360) {
  212             input_report_abs(dev, ABS_RX, (__s16)(((__s16)data[13] <<
8) | (__s16)data[12]));
  213             input_report_abs(dev, ABS_RY, (__s16)(((__s16)data[11] <<
8) | (__s16)data[10]));
```

```
214         } else {
215                 input_report_abs(dev, ABS_RX, (__s16)(((__s16)data[17] <<
8) | (__s16)data[16]));
216                 input_report_abs(dev, ABS_RY, (__s16)(((__s16)data[19] <<
8) | (__s16)data[18]));
217         }
218
219         /* triggers left/right */
220         if(xpad->is360) {
221                 input_report_abs(dev, ABS_Z, data[4]);
222                 input_report_abs(dev, ABS_RZ, data[5]);
223         } else {
224                 input_report_abs(dev, ABS_Z, data[10]);
225                 input_report_abs(dev, ABS_RZ, data[11]);
226         }
227
228         if(!xpad->is360) {
229                 /* digital pad (analogue mode): bits (3 2 1 0) (right
left down up) */
230                 input_report_abs(dev, ABS_HAT0X, !!(data[2] & 0x08) - !!
(data[2] & 0x04));
231                 input_report_abs(dev, ABS_HAT0Y, !!(data[2] & 0x01) - !!
(data[2] & 0x02));
232
233                 /* button A, B, X, Y analogue mode */
234                 input_report_abs(dev, ABS_HAT1X, data[4]);
235                 input_report_abs(dev, ABS_HAT1Y, data[5]);
236                 input_report_abs(dev, ABS_HAT2Y, data[6]);
237                 input_report_abs(dev, ABS_HAT3X, data[7]);
238
239                 /* button C (black) digital/analogue mode */
240                 input_report_key(dev, BTN_C, data[8]);
241                 input_report_abs(dev, ABS_HAT2X, data[8]);
242
243                 /* button Z (white) digital/analogue mode */
244                 input_report_key(dev, BTN_Z, data[9]);
245                 input_report_abs(dev, ABS_HAT3Y, data[9]);
246         }
247
248         input_sync(dev);
249 }
250
251 /**
252  *    xpad_irq_in
253  *
254  *    Completion handler for interrupt in transfers (user input).
255  *    Just calls xpad_process_packet which does then emit input
events.
256  */
257 static void xpad_irq_in(struct urb *urb)
258 {
259         struct usb_xpad *xpad = urb->context;
260         int retval;
261
262         switch (urb->status) {
263         case 0:
264                 /* success */
265                 break;
266         case -ECONNRESET:
```

```
267         case -ENOENT:
268         case -ESHUTDOWN:
269                 /* this urb is terminated, clean up */
270                 dbg("%s - urb shutting down with status: %d",
271                     __FUNCTION__, urb->status);
272                 return;
273         default:
274                 dbg("%s - nonzero urb status received: %d",
275                     __FUNCTION__, urb->status);
276                 goto exit;
277         }
278
279         xpad_process_packet(xpad, 0, xpad->idata);
280
281 exit:
282         retval = usb_submit_urb(urb, GFP_ATOMIC);
283         if (retval)
284                 err("%s - usb_submit_urb failed with result %d",
285                     __FUNCTION__, retval);
286 }
287
288 /**
289  *     xpad_open
290  *
291  *     Called when a an application opens the device.
292  */
293 static int xpad_open(struct input_dev *dev)
294 {
295         struct usb_xpad *xpad = dev->private;
296         int status;
297
298         info("opening device");
299
300         xpad->irq_in->dev = xpad->udev;
301         if ((status = usb_submit_urb(xpad->irq_in, GFP_KERNEL))) {
302                 err("open input urb failed: %d", status);
303                 return -EIO;
304         }
305
306         if(!xpad->is360) {
307                 xpad_rumble_open(xpad);
308         }
309
310         return 0;
311 }
312
313 /**
314  *     xpad_close
315  *
316  *     Called when an application closes the device.
317  */
318 static void xpad_close(struct input_dev *dev)
319 {
320         struct usb_xpad *xpad = input_get_drvdata(dev);
321
322         info("closing device");
323         usb_kill_urb(xpad->irq_in);
324         if(!xpad->is360) {
325                 xpad_rumble_close(xpad);
```

```
326         }
327 }
328
329 /**
330  *    xpad_probe
331  *
332  *    Called upon device detection to find a suitable driver.
333  *    Must return NULL when no xpad is found, else setup everything.
334  */
335 static int xpad_probe(struct usb_interface *intf, const struct
usb_device_id *id)
336 {
337         struct usb_device *udev = interface_to_usbdev(intf);
338         struct usb_xpad *xpad;
339         struct input_dev *input_dev;
340         struct usb_endpoint_descriptor *ep_irq_in;
341         int i;
342         int error = -ENOMEM;
343         int probedDevNum = -1;  /* this takes the index into the known
devices
344                                    array for the recognized device */
345
346         /* try to detect the device we are called for */
347         for (i = 0; xpad_device[i].idVendor; ++i) {
348                 if ((le16_to_cpu(udev->descriptor.idVendor) ==
xpad_device[i].idVendor) &&
349                     (le16_to_cpu(udev->descriptor.idProduct) ==
xpad_device[i].idProduct)) {
350                         probedDevNum = i;
351                         break;
352                 }
353         }
354
355         /* sanity check, did we recognize this device? if not, fail */
356         if ((probedDevNum == -1) || (!
xpad_device[probedDevNum].idVendor &&
357                                      !xpad_device[probedDevNum].idProduct))
358                 return -ENODEV;
359
360         xpad = kzalloc(sizeof(struct usb_xpad), GFP_KERNEL);
361         input_dev = input_allocate_device();
362         if (!xpad || !input_dev)
363                 goto fail1;
364
365         xpad->idata = usb_buffer_alloc(udev, XPAD_PKT_LEN,
366                                        GFP_ATOMIC, &xpad->idata_dma);
367         if (!xpad->idata)
368                 goto fail1;
369
370         /* setup input interrupt pipe (button and axis state) */
371         xpad->irq_in = usb_alloc_urb(0, GFP_KERNEL);
372         if (!xpad->irq_in)
373                 goto fail2;
374
375         xpad->udev = udev;
376         xpad->dev = input_dev;
377         xpad->isMat = xpad_device[probedDevNum].isMat;
378         xpad->is360 = xpad_device[probedDevNum].is360;
379         usb_make_path(udev, xpad->phys, sizeof(xpad->phys));
```

```
380         strlcat(xpad->phys, "/input0", sizeof(xpad->phys));
381
382         input_dev->name = xpad_device[probedDevNum].name;
383         input_dev->phys = xpad->phys;
384         usb_to_input_id(udev, &input_dev->id);
385         input_dev->dev.parent = &intf->dev;
386
387         input_set_drvdata(input_dev, xpad);
388
389         input_dev->open = xpad_open;
390         input_dev->close = xpad_close;
391
392         /* this was meant to allow a user space tool on-the-fly
configuration
393            of driver options (rumble on, etc...)
394            yet, Vojtech said this is better done using sysfs (linux
2.6)
395            plus, it needs a patch to the input subsystem */
396 /*      input_dev->ioctl = xpad_ioctl;*/
397
398         if (xpad->isMat) {
399               input_dev->evbit[0] = BIT(EV_KEY);
400               for (i = 0; xpad_mat_btn[i] >= 0; ++i)
401                     set_bit(xpad_mat_btn[i], input_dev->keybit);
402         } else {
403               input_dev->evbit[0] = BIT(EV_KEY) | BIT(EV_ABS);
404
405               for (i = 0; xpad_btn[i] >= 0; ++i)
406                     set_bit(xpad_btn[i], input_dev->keybit);
407
408               for (i = 0; xpad_abs[i] >= 0; ++i) {
409
410                     signed short t = xpad_abs[i];
411
412                     set_bit(t, input_dev->absbit);
413
414                     switch (t) {
415                     case ABS_X:
416                     case ABS_Y:
417                     case ABS_RX:
418                     case ABS_RY:      /* the two sticks */
419                           input_set_abs_params(input_dev, t,
420                                       -32768, 32767, 16, 12000);
421                           break;
422                     case ABS_Z: /* left trigger */
423                     case ABS_RZ:      /* right trigger */
424                     case ABS_HAT1X:   /* analogue button A */
425                     case ABS_HAT1Y:   /* analogue button B */
426                     case ABS_HAT2X:   /* analogue button C */
427                     case ABS_HAT2Y:   /* analogue button X */
428                     case ABS_HAT3X:   /* analogue button Y */
429                     case ABS_HAT3Y:   /* analogue button Z */
430                           input_set_abs_params(input_dev, t,
431                                       0, 255, 0, 0);
432                           break;
433                     case ABS_HAT0X:
434                     case ABS_HAT0Y:   /* the d-pad */
435                           input_set_abs_params(input_dev, t,
436                                       -1, 1, 0, 0);
```

```
437                         break;
438                   }
439             }
440
441             if (!xpad->is360)
442                   if (xpad_rumble_probe(udev, xpad, ifnum) != 0)
443                         err("could not init rumble");
444       }
445
446       /* init input URB for USB INT transfer from device */
447       ep_irq_in = &intf->cur_altsetting->endpoint[0].desc;
448       usb_fill_int_urb(xpad->irq_in, udev,
449                   usb_rcvintpipe(udev, ep_irq_in->bEndpointAddress),
450                   xpad->idata, XPAD_PKT_LEN, xpad_irq_in,
451                   xpad, ep_irq_in->bInterval);
452       xpad->irq_in->transfer_dma = xpad->idata_dma;
453       xpad->irq_in->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
454
455       error = input_register_device(xpad->dev);
456       if (error)
457             goto fail3;
458
459       usb_set_intfdata(intf, xpad);
460
461       /* Turn off the LEDs on xpad 360 controllers */
462       if (xpad->is360) {
463             char ledcmd[] = {1, 3, 0}; /* The LED-off command for
Xbox-360 controllers */
464             int j;
465             usb_bulk_msg(udev, usb_sndintpipe(udev,2), ledcmd, 3, &j,
0);
466       }
467
468       return 0;
469
470 fail3:      usb_free_urb(xpad->irq_in);
471 fail2:      usb_buffer_free(udev, XPAD_PKT_LEN, xpad->idata, xpad-
>idata_dma);
472 fail1:      input_free_device(input_dev);
473       kfree(xpad);
474       return error;
475 }
476
477 /**
478  *    xpad_disconnect
479  *
480  *    Called upon device disconnect to dispose of the structures and
481  *    close the USB connections.
482  */
483 static void xpad_disconnect(struct usb_interface *intf)
484 {
485       struct usb_xpad *xpad = usb_get_intfdata(intf);
486
487       usb_set_intfdata(intf, NULL);
488       if (xpad) {
489             usb_kill_urb(xpad->irq_in);
490             if(!xpad->is360) {
491                   xpad_rumble_close(xpad);
492             }
```

```
493             input_unregister_device(xpad->dev);
494
495             usb_free_urb(xpad->irq_in);
496
497             usb_buffer_free(interface_to_usbdev(intf), XPAD_PKT_LEN,
498                             xpad->idata, xpad->idata_dma);
499
500             if(!xpad->is360) {
501                   xpad_rumble_disconnect(xpad);
502             }
503
504             kfree(xpad);
505       }
506 }
507
508 /****************** Linux driver framework specific stuff
************/
509
510 static struct usb_driver xpad_driver = {
511       .name       = "xpad",
512       .probe            = xpad_probe,
513       .disconnect = xpad_disconnect,
514       .id_table   = xpad_table,
515 };
516
517 /**
518  * driver init entry point
519  */
520 static int __init usb_xpad_init(void)
521 {
522       int result = usb_register(&xpad_driver);
523       if (result == 0)
524             info(DRIVER_DESC " " DRIVER_VERSION);
525       return result;
526 }
527
528 /**
529  * driver exit entry point
530  */
531 static void __exit usb_xpad_exit(void)
532 {
533       usb_deregister(&xpad_driver);
534 }
535
536 module_init(usb_xpad_init);
537 module_exit(usb_xpad_exit);
538
539 MODULE_AUTHOR(DRIVER_AUTHOR);
540 MODULE_DESCRIPTION(DRIVER_DESC);
541 MODULE_LICENSE("GPL");
542
543 /*
544  *  driver history
545  * ----------------
546  *
547  * 2005-11-25 - 0.1.6 : Added Xbox 360 Controller support
548  *
549  * 2005-03-15 - 0.1.5 : Mouse emulation removed.  Deadzones
increased.
```

```
 550  *  - Flipped the Y axis of the left joystick (it was inverted, like
on a
 551  *    flight simulator).
 552  *
 553  * 2003-05-15 - 0.1.2 : ioctls, dynamic mouse/rumble
activation, /proc fs
 554  *  - added some /proc files for informational purposes (readonly
right now)
 555  *  - added init parameters for mouse/rumble activation upon
detection
 556  *  - added dynamic changes to mouse events / rumble effect
generation via
 557  *    ioctls - NOTE: this requires a currently unofficial joydev
patch!
 558  *
 559  * 2003-04-29 - 0.1.1 : minor cleanups, some comments
 560  *  - fixed incorrect handling of unknown devices (please try ir
dongle now)
 561  *  - fixed input URB length (the 256 bytes from 0.1.0 broke
everything for the
 562  *    MS controller as well as my Interact device, set back to 32
(please
 563  *    REPORT problems BEFORE any further changes here, since those
can be fatal)
 564  *  - fixed rumbling for MS controllers (need 6 bytes output report)
 565  *  - dropped kernel-2.5 ifdefs, much more readable now
 566  *  - preparation for major rework under way, stay tuned
 567  *
 568  * 2003-03-25 - 0.1.0 : (Franz) Some Debuggin
 569  *  - Better Handling
 570  *  - X/Y support, Speed differenting
 571  *  - Landing Zone, Dead Zone, Offset kompensation, Zero-
adjustment, .... aso.
 572  *  - Removed Wheel handling in Mouse Emulation .. sensless..
 573  *
 574  * 2003-01-23 - 0.1.0-pre : added mouse emulation and rumble support
 575  *  - can provide mouse emulation (compile time switch)
 576  *    this code has been taken from Oliver Schwartz' xpad-mouse
driver
 577  *  - basic rumble support (compile time switch)        EXPERIMENTAL!
 578  *
 579  * 2002-08-05 - 0.0.6 : added analog button support
 580  *
 581  * 2002-07-17 - 0.0.5 : (Vojtech Pavlik) rework
 582  *  - simplified d-pad handling
 583  *
 584  * 2002-07-16 - 0.0.4 : minor changes, merge with Vojtech's v0.0.3
 585  *  - verified the lack of HID and report descriptors
 586  *  - verified that ALL buttons WORK
 587  *  - fixed d-pad to axes mapping
 588  *
 589  * 2002-07-14 - 0.0.3 : (Vojtech Pavlik) rework
 590  *  - indentation fixes
 591  *  - usb + input init sequence fixes
 592  *
 593  * 2002-07-02 - 0.0.2 : basic working version
 594  *  - all axes and 9 of the 10 buttons work (german InterAct device)
 595  *  - the black button does not work
 596  *
```

```
597  * 2002-06-27 - 0.0.1 : first version, just said "XBOX HID
controller"
598  */
```