|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI  **VIỆN ĐIỆN TỬ - VIỄN THÔNG**  logo_128  **BÁO CÁO TIẾN ĐỘ HỆ ĐIỀU HÀNH**  **Đề tài :**  **Lập trình device driver (cho chuột USB) trên Linux**  **Giảng viên hướng dẫn: TS. Phạm Doãn Tĩnh**  Nhóm sinh viên thực hiện: 10   |  |  | | --- | --- | | **Họ và tên** | **MSSV** | | Nguyễn Đức Long | 20182657 | | Vũ Tiến Thịnh | 20182809 | | Đinh Duy Thành | 20182787 | | Đỗ Thành Đạt | 20182411 | | Nguyễn Văn Minh | 20182688 |   Hà Nội, 1-2022 |

**LỜI NÓI ĐẦU**

Chúng em xin gửi lời cảm ơn đến thầy Phạm Doãn Tĩnh đã hướng dẫn nhóm em trong kì học này, giúp chúng em tìm hiểu về hệ điều hành và các cơ chế hoạt động của hệ điều hành. Với vốn kiến thức còn hạn chế và thời gian có hạn, bài tập lớn của chúng em chắc chắn còn nhiều sai sót, chúng em rất mong nhận được sự góp ý từ thầy để bài tập lớn của chúng em được hoàn thiện hơn.

**Chúng em xin chân thành cảm ơn!**

PHÂN CÔNG CÔNG VIỆC

|  |  |  |  |
| --- | --- | --- | --- |
| Ngày | Công việc | Người thực hiện | Mức độ hoàn thiện |
| 9/12/2021 | Tìm hiểu về Linux Kernel driver | Đinh Duy Thành | 30% |
| Tìm hiểu về Linux driver | Nguyễn Văn Minh |
| Vũ Tiến Thịnh |
| Tìm hiểu về Character driver | Đỗ Thành Đạt |
| Nguyễn Đức Long |
| Soạn thảo báo cáo | Nguyễn Đức Long |
| 20/12/2021 | Tìm hiểu về các thành phần cơ bản của USB | Đỗ Thành Đạt | 70% |
| Tìm hiểu về các mô hình các giao thức USB | Nguyễn Đức Long |
| Tìm hiểu về quy trình viết USB driver | Nguyễn Đức Long  Vũ Tiến Thịnh  Đỗ Thành Đạt  Nguyễn Văn Minh  Đinh Duy Thành |
| Soạn thảo báo cáo | Vũ Tiến Thịnh |
| Làm Slide | Đinh Duy Thành  Nguyễn Văn Minh |

\*Ban đầu đề tài của nhóm em là lập lịch các thuật toán trên Linux chưa chạm đến kernel nên sau đó nhóm em đã thay đổi đề tài nên ngày bắt đầu tìm hiểu đề tài này từ 9/12/2021

**MỤC LỤC**

[PHÂN CÔNG CÔNG VIỆC ii](#_Toc93565672)

[DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT i](#_Toc93565673)

[CHƯƠNG 1. TỔNG QUAN VỀ LINUX DRIVER 1](#_Toc93565674)

[1.1 Giới thiệu về Linux driver 1](#_Toc93565675)

[1.1.1 Linux driver 1](#_Toc93565676)

[1.1.2 Linux device driver 4](#_Toc93565677)

[1.2 LINUX KERNEL MODULE 5](#_Toc93565678)

[1.2.1 Linux kernel module 5](#_Toc93565679)

[1.3 CHARACTER DRIVER 7](#_Toc93565680)

[1.3.1 Device file (device node) 7](#_Toc93565681)

[1.3.2 Device number 9](#_Toc93565682)

[1.3.3 Cấu trúc của character driver 10](#_Toc93565683)

[CHƯƠNG 2. TỔNG QUAN VỀ GIAO THỨC USB 11](#_Toc93565684)

[2.1 Các thành phần cơ bản của giao thức USB 11](#_Toc93565685)

[2.1.1 Endpoint 11](#_Toc93565686)

[2.1.2 Interface 13](#_Toc93565687)

[2.1.3 Configuration 14](#_Toc93565688)

[2.1.4 Sysfs 14](#_Toc93565689)

[2.2 Mô hình giao thức USB 14](#_Toc93565690)

[2.2.1 Chuẩn tín hiệu 14](#_Toc93565691)

[2.2.2 Mô hình mạng 15](#_Toc93565692)

[2.2.3 Kịch bản hoạt động 16](#_Toc93565693)

[2.2.4 Chế độ truyền 17](#_Toc93565694)

[2.2.5 Host view 17](#_Toc93565695)

[2.2.6 Device view 18](#_Toc93565696)

[CHƯƠNG 3. QUI TRÌNH VIẾT USB DEVICE DRIVER 19](#_Toc93565697)

[3.1 Tìm hiểu về thiết bị usb muốn giao tiếp (usb device) 19](#_Toc93565698)

[3.2 Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver 20](#_Toc93565699)

[3.3 Khai báo cấu trúc dữ liệu liên quan tới thiết bị 20](#_Toc93565700)

[3.4 Đăng kí và hủy đăng kí USB Device Driver 21](#_Toc93565701)

[3.5 Hàm thăm dò thiết bị (probe) 23](#_Toc93565702)

[3.5.1 Truy nhập các thông tin về Endpoint 23](#_Toc93565703)

[3.5.2 Lưu lại các thông tin đã truy nhập và cấp phát 24](#_Toc93565704)

[3.5.3 Đăng kí lớp thiết bị 24](#_Toc93565705)

[3.6 Hàm ngắt kết nối thiết bị 27](#_Toc93565706)

[3.7 Các hàm mở / đọc / ghi thiết bị 27](#_Toc93565707)

[3.7.1 Hành động mở tệp tin thiết bị 27](#_Toc93565708)

[3.7.2 Hành động đọc / ghi 28](#_Toc93565709)

[CHƯƠNG 4. KẾT LUẬN 29](#_Toc93565710)

[TÀI LIỆU THAM KHẢO 30](#_Toc93565711)

[PHỤ LỤC I 31](#_Toc93565712)

[PHỤ LỤC II 34](#_Toc93565713)

# DANH MỤC KÝ HIỆU VÀ CHỮ VIẾT TẮT

|  |  |
| --- | --- |
| CPU | Central Processing Unit – Bộ xử lý trung tâm |
| I/O | Input/Output |
| RAM | Random Access Memory |
| ROM | Read Only Memory |
| RTC | Real-Time control |

# TỔNG QUAN VỀ LINUX DRIVER

## Giới thiệu về Linux driver

### Linux driver

Driver là một phần mềm, gồm các lệnh, hướng dẫn CPU cách tương tác với thiết bị. Các thiết bị có thể là chuột, bàn phím, ổ cứng, card mạng, loa, màn hình,… Tuy nhiên, các thiết bị này không được nối trực tiếp với CPU, bởi vì:

* Hệ thống có nhiều thiết bị, nhưng số lượng chân của CPU hữu hạn.
* Tốc độ làm việc của các thiết bị thấp hơn nhiều so với CPU.

Chính vì vậy, các thiết bị kết nối với CPU thông qua một thiết bị khác, gọi là bộ điều khiển (device controller). Có nhiều cách để phân loại bộ điều khiển:

* Phân loại theo chức năng:

+ Hard disk controller: trợ giúp CPU điều khiển ổ cứng.

+ Graphic controller: trợ giúp CPU điều khiển các thiết bị hiển thị.

+ Keyboard controller: trợ giúp CPU điều khiển bàn phím.

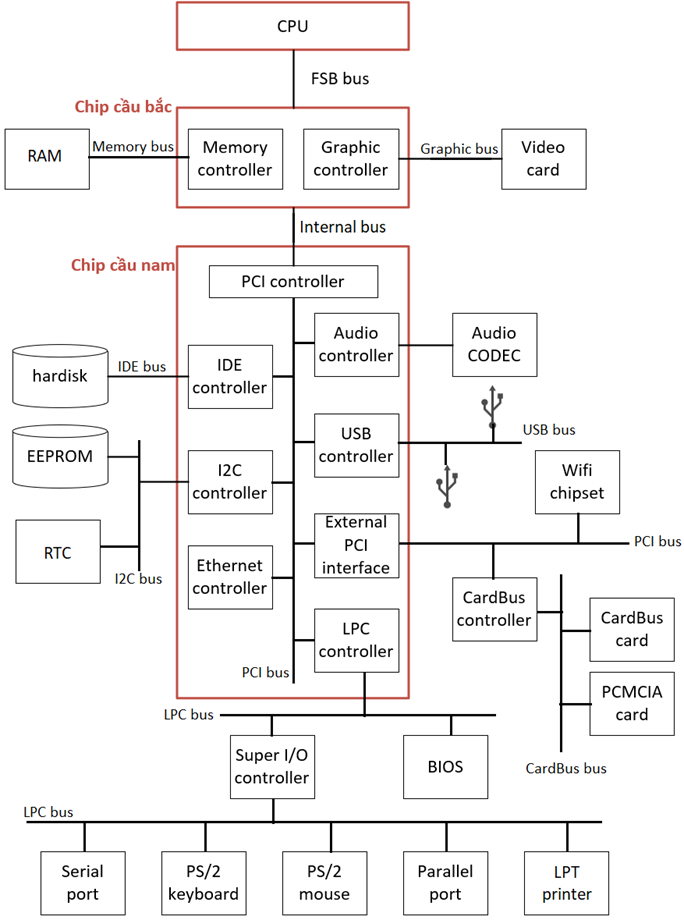
* Phân loại theo kỹ thuật giao tiếp với các thiết bị:

+ PCI controller: hỗ trợ CPU giao tiếp với các thiết bị khác, mà các thiết bị này được thiết kế theo chuẩn PCI.

+ USB controller: hỗ trợ CPU giao tiếp với các thiết bị khác, mà các thiết bị này được thiết kế theo chuẩn USB.

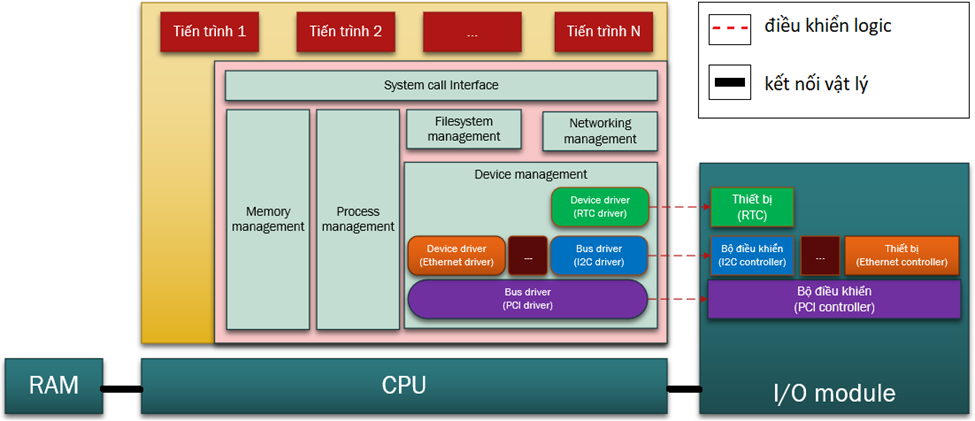
+ I2C controller: hỗ trợ CPU giao tiếp với các thiết bị khác, mà các thiết bị này được thiết kế theo chuẩn I2C.

Hình 1 là sơ đồ khối của hệ thống máy tính cá nhân, sử dụng bộ xử lý của Intel. Từ hình này, ta cũng thấy rằng, thiết bị (device) và bộ điều khiển (device controller) chỉ mang tính chất tương đối. Ví dụ, trên PCI bus, PCI controller đóng vai trò là bộ điều khiển, còn I2C controller đóng vai trò là thiết bị. Nhưng trên I2C bus, thì I2C controller lại là bộ điều khiển, còn EEPROM và RTC đóng vai trò là thiết bị.



Hình 1 Sơ đồ khối của máy tính cá nhân sử dụng bộ xử lý Intel

Đứng từ góc độ của CPU, bộ điều khiển cũng chỉ là một thiết bị. Do đó, cần có driver hướng dẫn CPU làm việc với bộ điều khiển. Driver này được gọi là bus driver. Còn driver hướng dẫn CPU làm việc với thiết bị thì được gọi là device driver (hình 2).



Hình 2 Mối liên hệ giữa driver, bộ điều khiển và thiết bị dưới góc độ thực thi

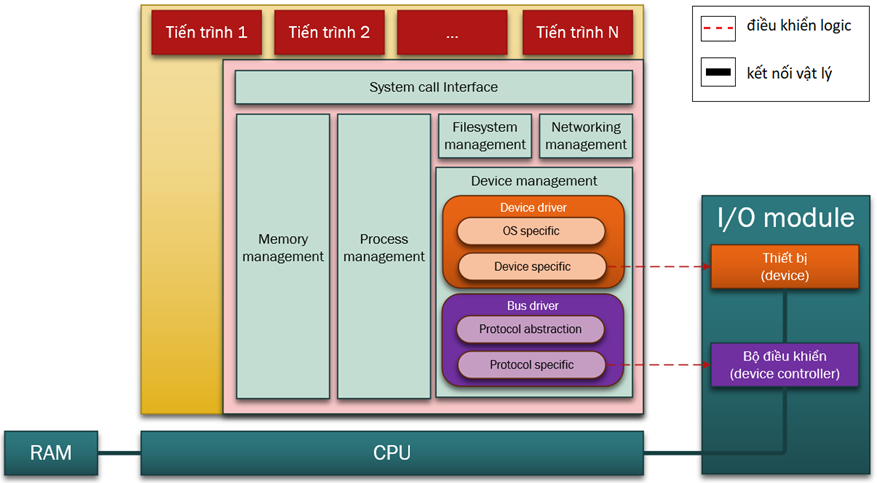
Để thuận tiện cho quá trình phát triển, bus driver và device driver được chia làm 2 thành phần (hình 3).

Device driver gồm 2 phần:

* Thành phần OS specific. Thành phần này cung cấp cho hệ điều hành các dịch vụ đọc/ghi dữ liệu của thiết bị. Điều này cho phép chúng ta xây dựng hệ điều hành độc lập với cấu trúc của thiết bị.
* Thành phần device specific. Thành phần này chứa các lệnh hướng dẫn CPU điều khiển thiết bị, giám sát thiết bị, trao đổi dữ liệu với thiết bị. Chúng ta sử dụng datasheet của thiết bị để xây dựng thành phần này. Datasheet là một tài liệu được cung cấp bởi nhà sản xuất thiết bị. Nó mô tả sơ đồ khối chức năng, nguyên lý hoạt động, hiệu suất hoạt động, đặc tính điện của thiết bị và đặc biệt là bản đồ thanh ghi (register map).

Bus driver gồm 2 phần:

* Thành phần protocol abstraction. Thành phần này che giấu đi sự phức tạp của các giao thức trên bus, cung cấp các dịch vụ cho device driver sử dụng. Ví dụ như đọc/ghi một thanh ghi nào đó của thiết bị.
* Thành phần protocol specific. Thành phần này chứa các lệnh hướng dẫn CPU làm việc với bộ điều khiển, giúp đọc/ghi dữ liệu trên bus. Nó cũng được xây dựng dựa trên datasheet của bộ điều khiển.



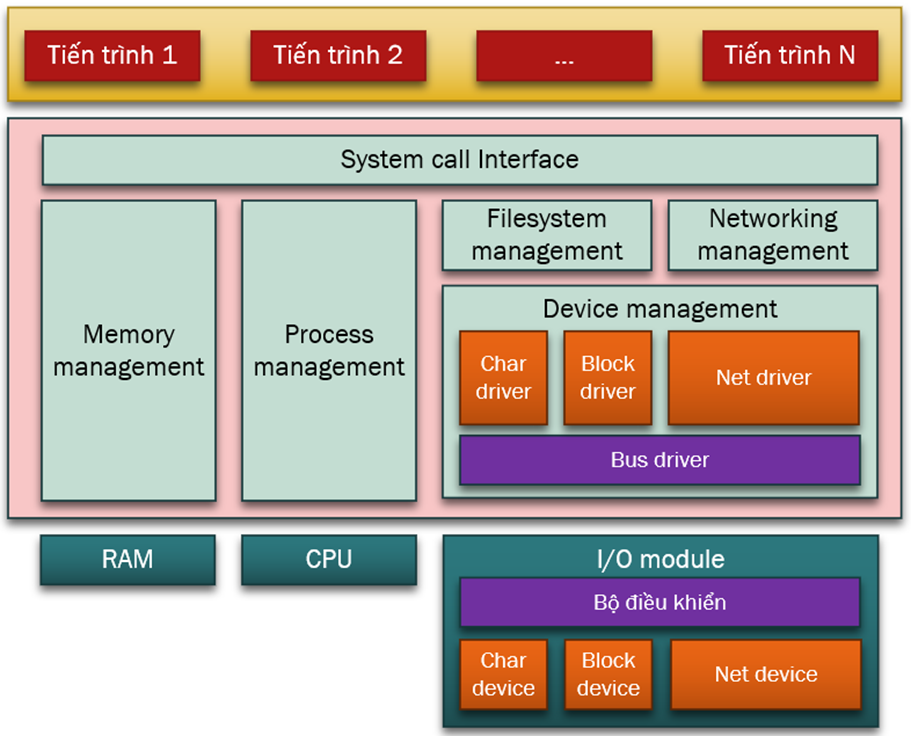
Hình 3 Các thành phần của bus driver và device driver

### Linux device driver

Dựa vào lượng dữ liệu mỗi lần thiết bị trao đổi với CPU, thiết bị được chia làm 3 loại:

* Character device: lượng dữ liệu nhỏ nhất mà CPU và thiết bị trao đổi với nhau là 1 byte. Ví dụ về các thiết bị thuộc loại này là chuột, bàn phím, loa,…
* Block device: lượng dữ liệu nhỏ nhất mà CPU và thiết bị trao đổi với nhau là một khối, gồm nhiều byte (ví dụ 1 khối gồm 512 byte). Thông thường, block device là các thiết bị lưu trữ, như ổ cứng chẳng hạn.
* Network device: lượng dữ liệu nhỏ nhất mà CPU và thiết bị trao đổi với nhau là một gói tin, gồm nhiều byte. Gói tin có kích thước không cố định. Thông thường, network device là các thiết bị mạng, như NIC card, Wifi chip.

Tương ứng với ba loại thiết bị trên, chúng ta có ba loại device driver: character driver, block driver, network driver.



Hình 4 Phân loại device driver

## LINUX KERNEL MODULE

### Linux kernel module

Linux kernel module là một file với tên mở rộng là (.ko). Nó sẽ được lắp vào hoặc tháo ra khỏi kernel khi cần thiết. Chính vì vậy, nó còn có một tên gọi khác là loadable kernel module. Một trong những kiểu loadable kernel module phổ biến đó là driver.

Việc thiết kế driver theo kiểu loadable module mang lại 3 lợi ích:

* Giúp giảm kích thước kernel. Do đó, giảm sự lãng phí bộ nhớ và giảm thời gian khởi động hệ thống.
* Không phải biên dịch lại kernel khi thêm mới driver hoặc khi thay đổi driver.
* Không cần phải khởi động lại hệ thống khi thêm mới driver. Trong khi đối với Windows, mỗi khi cài thêm driver, ta phải khởi động lại hệ thống, điều này không thích hợp với các máy server.

Phần lớn các driver đều là các loadable kernel module, nhưng không phải là tất cả. Vẫn có một số driver được tích hợp luôn vào trong kernel, đặc biệt là các bus driver. Chúng được gọi là built-in driver. Các device driver thường sẽ là các loadable kernel module.

Ngược lại, không phải loadable kernel module nào cũng là driver, ví dụ kvm.ko là loadable kernel module nhưng không phải là driver. Trên thực tế, loadable kernel module được chia làm 3 loại chính: device driver, system call và file system.

Trong khóa học này, các thuật ngữ module, device driver, loadable kernel module, Linux kernel module, loadable module, kernel module đều được hiểu là một.

Khi cần một module nhưng nó lại chưa có trong kernel space, kernel sẽ đưa module ấy vào. Quá trình này có thể diễn ra một cách tự động, với trình tự sau:

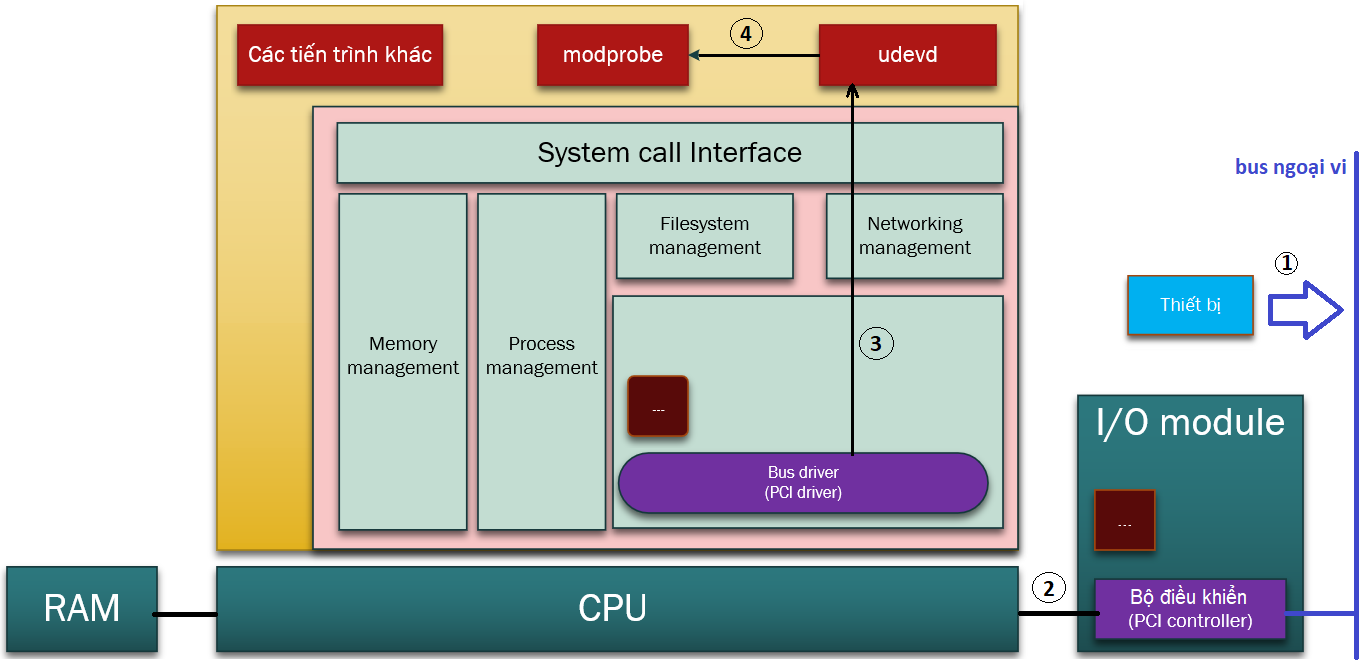
Bước 1: Kernel kích hoạt tiến trình modprobe cùng với tham số truyền vào là tên của module (ví dụ xxx.ko).

Bước 2: Tiến trình modprobe kiểm tra file /lib/modules/<kernel-version>/modules.dep xem xxx.ko có phụ thuộc vào module nào khác không. Giả sử xxx.ko phụ thuộc vào module yyy.ko.

Bước 3: Tiến trình modprobe sẽ kích hoạt tiến trình insmod để đưa các module phụ thuộc vào trước (yyy.ko), rồi mới tới module cần thiết (xxx.ko).

Như vậy, các module được đưa vào kernel space dưới sự giúp đỡ của tiến trình modprobe. Có 3 cách để kernel kích hoạt tiến trình modprobe:

* Cách 1 là sử dụng kmod. Đây là một thành phần của Linux kernel, hoạt động trong kernel space. Khi một thành phần nào đó của kernel cần đưa một module vào trong kernel space, nó sẽ truyền tên module cho hàm request\_module của kmod. Hàm request\_module sẽ gọi hàm call\_usermodehelper\_setup để sinh ra tiến trình modprobe. Các bạn có thể tham khảo mã nguồn của kmod tại /kernel/kmod.c.
* Cách 2 là sử dụng udevd (hình 1). Đây là một tiến trình hoạt động trong user space. Nếu một thiết bị cắm vào hệ thống máy tính, thì điện trở trên bus ngoại vi (ví dụ PCI bus hoặc USB bus) sẽ thay đổi và bộ điều khiển (controller) sẽ biết điều này. Khi đó, bus driver sẽ gửi một bản tin lên cho tiến trình udevd. Bản tin này chứa thông tin về thiết bị. Tiến trình udevd sẽ tra cứu file /lib/modules/<kernel-version>/modules.alias để tìm ra driver nào tương thích với thiết bị. Sau đó, udevd sinh ra tiến trình modprobe.



Hình 5 Minh họa quá trình kích hoạt modprobe bằng udevd

## CHARACTER DRIVER

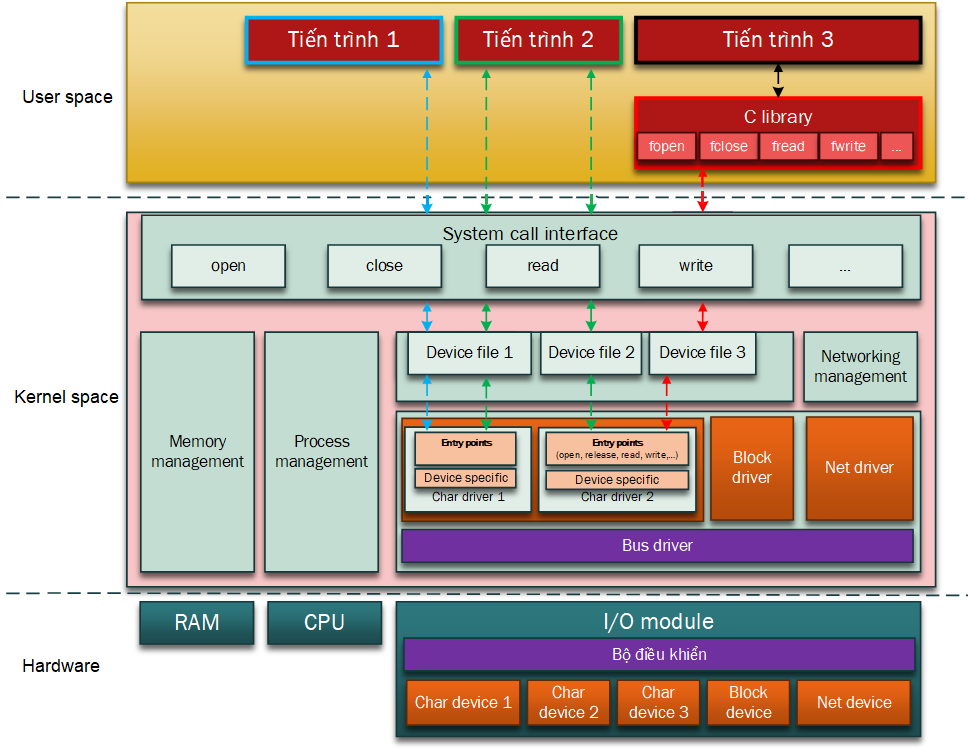
### Device file (device node)

Trong Linux kernel, file được chia làm 6 loại: file thông thường (regular file), thư mục (directory), pipe, socket, symbolic link và device file (còn gọi device node). Với việc tạo ra device file, Linux kernel đã đánh lừa các tiến trình rằng, các char/block device cũng chỉ là các file thông thường. Do đó, các tiến trình sẽ nghĩ rằng, đọc/ghi dữ liệu từ thiết bị cũng giống như đọc/ghi dữ liệu từ file thông thường. Hình 1 biểu diễn cách một tiến trình đọc/ghi dữ liệu từ thiết bị:

* Đầu tiên, tiến trình gọi system call để tương tác với device file. Ngoài ra, tiến trình cũng có thể gọi system call một cách gián tiếp thông qua các library call của thư viện trung gian.
* Tiếp theo, kernel sẽ gọi một entry point của device driver tương ứng với device file. Thực chất, mỗi entry point là một hàm của device driver. Lúc lắp device driver vào kernel, device driver sẽ đăng ký các hàm này với kernel, để kernel biết hàm nào làm gì. Thông thường, sẽ có sự tương ứng 1 – 1 giữa system call của kernel và entry point của device driver.
* Cuối cùng, device driver sẽ hướng dẫn CPU giao tiếp với thiết bị.

Từ hình 1, chúng ta nhận thấy rằng:

* Một device driver có thể ứng với một hoặc nhiều device file.
* Một device driver có thể điều khiển một hoặc nhiều thiết bị.
* Một device file có thể được sử dụng bởi nhiều tiến trình.
* Một tiến trình có thể cần dùng nhiều device file.



Hình 6 Kết nối giữa tiến trình và character device trong Linux

Kernel sử dụng device number để biết device driver nào tương ứng với device file. Device number là một bộ gồm hai số: major number và minor number:

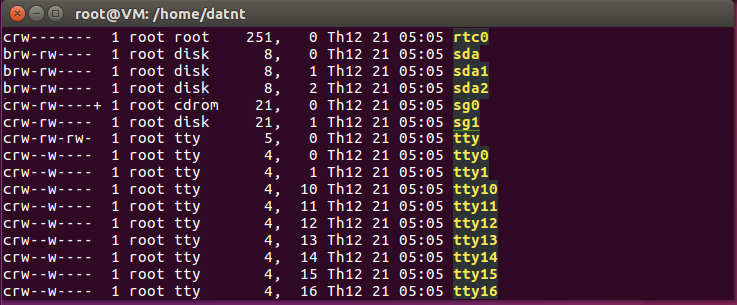
* Major number giúp kernel nhận biết device driver nào tương ứng với device file.
* Minor number giúp device driver nhận biết nó sẽ phải điều khiển thiết bị nào, nếu như device driver đó đang điều khiển nhiều thiết bị.

Khi nhiều tiến trình cùng dùng một device file, thì rất dễ xảy ra tranh chấp. Hiện tượng này gọi là race condition. Các giải pháp cho vấn đề này sẽ được trình bày trong chương Giải pháp chống race condition.

Tuy nhiên, chính những giải pháp chống race condition lại khiến nảy sinh thêm một vấn đề khác. Khi trong hệ thống có nhiều tiến trình, mỗi tiến trình cùng dùng đồng thời nhiều device file khác nhau, thì hệ thống có thể bị rơi vào tình trạng deadlock.

### Device number

Khi gõ lệnh “ls -l /dev” trên terminal, ta sẽ thu được kết quả tương tự như hình 2. Tại cột thứ 1, ký tự bắt đầu có thể là “c” (character device), hoặc “b” (block device). Cột thứ 10 (cột cuối cùng) là tên của device file (mặc định có màu vàng). Cột thứ 5 là major number, còn cột thứ 6 là minor number của device file.



Hình 7 Một số device file trong thư mục /dev

Trong ví dụ ở hình 7, các device file sda, sda1 và sda2 đại diện cho ba phân vùng của ổ cứng. Chúng có cùng major number bằng 8, nghĩa là ba device file này cùng tương ứng với một device driver. Điều này cũng có nghĩa là, device driver này chịu trách nhiệm điều khiển cả ba phân vùng này của ổ cứng. Tuy nhiên, minor number của chúng lại khác nhau (0, 1, 2).

Giả sử, khi tiến trình đọc/ghi vào device file sda2, thì kernel sẽ xác định được major number là 8 và minor number là 2. Từ giá trị major number là 8, kernel sẽ biết được rằng hard disk driver tương ứng với device file này. Sau đó, kernel sẽ truyền đạt yêu cầu đọc/ghi thiết bị cùng với minor number là 2 cho disk driver. Từ giá trị minor number, disk driver sẽ biết rằng nó phải đọc/ghi dữ liệu từ phân vùng thứ 2 trên ổ cứng.

Khi lắp device driver vào kernel, hàm khởi tạo của device driver sẽ đăng kí với Linux kernel các số device number, chính là các cặp <major, minor>. Các cặp này có cùng major number, còn các minor number tạo thành một dải số tự nhiên liên tiếp. Ví dụ, lúc khởi tạo, tty driver đăng ký 64 device number có cùng major number bằng 4, còn các minor thuộc dải [0, 63]. Giá trị minor number thường bắt đầu từ 0, nhưng không phải là bắt buộc.

### Cấu trúc của character driver

Tương tự như các device driver khác, character driver (gọi tắt là char driver) gồm 2 phần:

Phần OS specific gồm các nhóm hàm sau:

* Hàm khởi tạo. Hàm này chịu trách nhiệm:

+ Yêu cầu kernel cấp phát device number.

+ Yêu cầu kernel tạo device file.

+ Yêu cầu kernel cấp phát bộ nhớ cho các cấu trúc dữ liệu của driver và khởi tạo chúng.

+ Yêu cầu khởi tạo thiết bị vật lý.

+ Đăng ký các hàm entry point với kernel.

+ Đăng ký hàm xử lý ngắt.

* Hàm kết thúc. Hàm này làm ngược lại những gì hàm khởi tạo đã làm.
* Các hàm entry point. Ví dụ open(), release(), read(), write(), ioctl(), mmap()…

Phần device specific gồm các nhóm hàm sau:

* Nhóm các hàm khởi tạo/giải phóng thiết bị.
* Nhóm các hàm đọc/ghi vào các thanh ghi của thiết bị.

+ Đọc/ghi các thanh ghi dữ liệu.

+ Lấy thông tin từ các thanh ghi trạng thái.

+ Thiết lập lệnh cho các thanh ghi điều khiển.

* Nhóm các hàm xử lý ngắt.

# TỔNG QUAN VỀ GIAO THỨC USB

Một thiết bị giao tiếp bằng chuẩn USB thực ra rất phức tạp. Nhưng may mắn là hệ điều hành Linux cung cấp cho ta những thư viện hỗ trợ khá đầy đủ, và điều đó đã giúp tiết kiệm được rất nhiều thời gian, công sức cũng như chi phí xây dựng một thiết bị như thế. Đa phần sự phức tạp của USB được hỗ trợ bởi USB Core của Linux.

## Các thành phần cơ bản của giao thức USB

### Endpoint

Là một buffer dùng để lưu trữ dữ liệu. Dữ liệu tại Endpoint có thể là dữ liệu vừa nhận được hoặc dữ liệu chờ để chuyển đi. USB Endpoint truyền dữ liệu theo một hướng duy nhất, từ host đến device (gọi là OUT Endpoint) hoặc ngược lại (IN Endpoint).

Có 4 loại endpoint:

* **Controll**: Chuyển quyền kiểm soát thường được sử dụng cho các hoạt động lệnh và trạng thái. Chúng cần thiết để thiết lập một thiết bị USB với tất cả các chức năng liệt kê được thực hiện bằng cách sử dụng chuyển điều khiển. Chúng thường là các gói dữ liệu ngẫu nhiên, được khởi tạo bởi máy chủ lưu trữ và sử dụng cách phân phối nỗ lực nhất. Độ dài gói của quá trình truyền điều khiển trong thiết bị tốc độ thấp phải là 8 byte, thiết bị tốc độ cao cho phép kích thước gói là 8, 16, 32 hoặc 64 byte và thiết bị tốc độ cao phải có kích thước gói là 64 byte. Một chuyển giao quyền kiểm soát có thể có tối đa ba giai đoạn: Setup stage, Data stage and Status stage.
* **Interupt**: Interupt endpoint dùng để truyền dữ liệu dung lượng nhỏ mỗi khi máy chủ truy vấn dữ liệu từ thiết bị. Chúng cũng hay được dùng để gửi dữ liệu giữa các thiết bị với nhau, nhưng lượng dữ liệu truyền đi thường không lớn. Giao thức USB luôn đảm bảo có băng thông dự phòng cho loại endpoint này.
* **Bulk:** Bulk endpoint dùng để truyền dữ liệu có dung lượng lớn. Loại endpoint này có kích thước lớn hơn nhiều so với interupt. Các ví dụ có thể bao gồm lệnh in được gửi đến máy in hoặc hình ảnh được tạo từ máy quét. Truyền hàng loạt cung cấp khả năng sửa lỗi dưới dạng trường CRC16 trên cơ chế tải dữ liệu và phát hiện / truyền lại lỗi, đảm bảo dữ liệu được truyền và nhận mà không có lỗi. Bulk Endpoint sẽ sử dụng băng thông dự phòng chưa được phân bổ trên bus sau khi tất cả các giao dịch khác đã được phân bổ. Nếu bus bận với đẳng thời gian and / or ngắt thì dữ liệu lớn có thể từ từ chảy qua bus. Do đó, Bulk Endpoint chỉ nên được sử dụng cho giao tiếp không nhạy cảm về thời gian vì không có đảm bảo về độ trễ.
* **Isochorous:** Isochorous endpoint dùng để truyền dữ liệu kích thước lớn, nhưng không yêu cầu sự đảm bảo dữ liệu toàn vẹn. Chúng được dùng cho các thiết bị chấp nhận sự mất mát trong truyền dữ liệu, đổi lại sự đảm bảo quá trình được liên tục, như các thiết bị truyền dẫn thời gian thực như video và audio…

Cấu trúc dữ liệu của endpoint được định nghĩa trong struct **usb\_host\_endpoint** của Linux kernel. Trong đó thông tin thực sự của endpoint được chứa trong struct **usb\_endpoint\_descriptor**.

struct usb\_host\_endpoint {

struct usb\_endpoint\_descriptor desc;

struct usb\_ss\_ep\_comp\_descriptor ss\_ep\_comp;

struct list\_head urb\_list;

void \* hcpriv;

struct ep\_device \* ep\_dev;

unsigned char \* extra;

int extralen;

int enabled;

int streams;

};

Các thông số cần quan tâm:

* **bEndpointAddress**: địa chỉ của endpoint, trong đó có 8 bit dùng mã hóa hướng của endpoint là IN hay OUT. Bits 0..3b: Endpoint Number; Bits 4..6b được bảo lưu. Đặt thành 0; Bits 7 Hướng 0 = Ra, 1 = Vào (Bỏ qua đối với Điểm cuối điều khiển).
* **bmAttributes**: Dùng để chỉ định loại Endpoint, có thể là Controll, Interupt, Bulk hoặc Isochronous. Nếu một điểm cuối Isochronous được chỉ định, các thuộc tính bổ sung có thể được chọn như kiểu sử dụng và đồng bộ hóa.
* **wMaxPacketSize**: Kích thước tối đa gói tin mà endpoint có thể chuyển đi. Nếu gói tin cần chuyển lớn hơn giá trị này thì nó sẽ bị chia thành các gói có kích thước tương đương.
* **bInterval**: được sử dụng để chỉ định khoảng thời gian bỏ phiếu của một số lần chuyển nhất định. Các đơn vị được biểu thị bằng khung, do đó điều này tương đương với 1ms đối với thiết bị tốc độ thấp / đầy đủ và 125us đối với thiết bị tốc độ cao.
* **bLenght**: Kích thước bộ mô tả tính bằng byte (7 byte).
* **bDescriptorType**: Bộ mô tả Endpoint (0x05).

### Interface

Tập hợp các Endpoint gọi là Interface. Mỗi interface thể hiện một chức năng cơ bản duy nhất của thiết bị, ví dụ như ổ flash, hay bàn phím. Một thiết bị có thể có nhiều interface. Một interface có nhiều thiết lập, với thiết lập ban đầu được đánh số 0. Thiết lập khác nhau có thể dùng điều khuyển endpoint theo các cách khác nhau…

Interface được định nghĩa trong Linux kernel bằng struct **usb\_interface**.

Các thông số cần quan tâm:

* **bInterfaceNumber**: Cho biết chỉ mục của bộ mô tả giao diện. Điều này phải được dựa trên 0 và tăng lên một lần cho mỗi bộ mô tả giao diện mới.
* **bAlternativeSetting**: có thể được sử dụng để chỉ định các Interface thay thế. Các giao diện thay thế này có thể được chọn với yêu cầu Set Interface.
* **bNumEndpoints**: cho biết số lượng Endpoint được giao diện sử dụng. Giá trị này phải loại trừ điểm cuối bằng 0 và được sử dụng để chỉ ra số lượng bộ mô tả điểm cuối cần tuân theo.
* **struct usb\_host\_interface \*altsetting**: Một mảng chứa các phần tử kiểu interface tương ứng với mỗi thiết lập khác nhau. Mỗi struct usb\_host\_interface chứa một tập hợp các endpoint đã được định nghĩa trong **usb\_host\_endpoint.**
* **unsigned num\_altsetting**: số thiết lập khác nhau của interface, được trỏ đến bởi con trỏ **\*altsetting.**
* **struct usb\_host\_interface \*cur\_altsetting**: con trỏ trỏ đến thiết lập hiện tại của interface.
* **int minor**: Thiết bị trong hệ thống được truy cập bằng tên. Tên thiết bị bao gồm 2 phần là **major** và **minor**. **major** dùng xác định driver cho thiết bị, còn **minor** dùng để xác định chính xác thiết bị đang xét.

### Configuration

**Configuration** là tập hợp các interface. Một thiết bị có thể có nhiều configuration và chuyển đổi qua lại giữa chúng. Chỉ có một configuration được phép sử dụng tại mỗi thời điểm. Linux không thể hỗ trợ các thiết bị sử dụng đồng thời nhiều configuration (tuy nhiên các thiết bị đó rất hiếm).

Tóm lại:

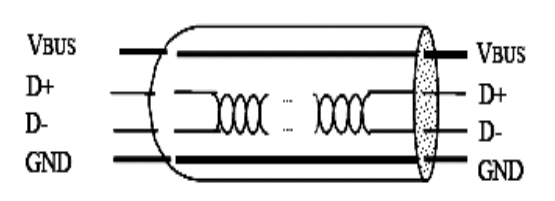
* Một thiết bị có thể có nhiều configuration.
* Một configuration có thể có nhiều interface.
* Một interface có thể có nhiều setting.
* Interface không có hoặc có thể có nhiều endpoint.

### Sysfs

**Sysfs** do tính phức tạp trong cấu trúc vật lý của thiết bị USB, việc mô tả thiết bị trong lập trình cũng không đơn giản. Sysfs được sử dụng như một trình quản lý dữ liệu của Linux giúp việc quản lý những thiết bị như thế này được đơn giản hơn. Mô tả vật lý thiết bị USB cũng như interface của nó được mô tả như những thiết bị độc lập trong sysfs.

## Mô hình giao thức USB

### Chuẩn tín hiệu

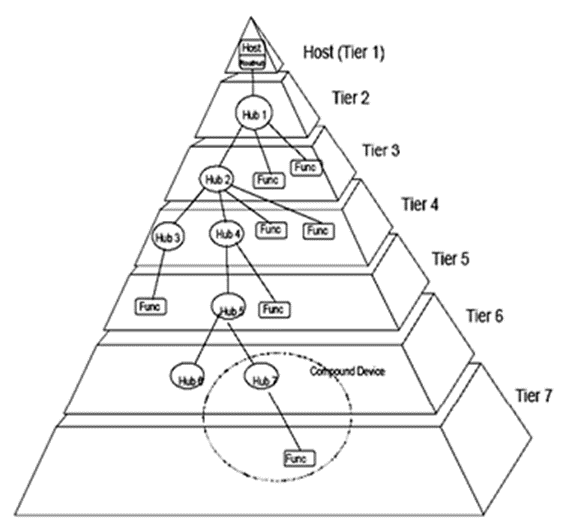


Hình 8: Cable USB

Chuẩn USB sử dụng 4 đường tín hiệu trong đó có 2 đường cấp nguồn DC (VBUS-5V và GND). 2 đường còn lại là một cặp tín hiệu vi sai (D+ và D-) cho phép truyền dữ liệu. Cặp dây tín hiệu này được nối xoắn ở bên trong nên có khả năng chống nhiễu tốt.

Ở phía máy chủ Host thì D+ và D- được nối đất qua các điện trở Rpd còn về phía thiết bị ngoại vi, các đầu dây D+, D- được bảo vệ bởi điện trở cuối (đó chính là các điện trở Rpu. Thiết bị tốc độ cao hoặc toàn tốc cần có điện trở nối +3.3 V cho đầu D+, thiết bị tốc độ thấp cần có điện trở nối lên 3.3 V cho đầu D-. Những điện trở này tạo nên các mức điện thế khác nhau giữa D+ và D- giúp cho máy chủ phát hiện được việc cắm vào hay rút ra của thiết bị cũng như tốc độ truyền dữ liệu của thiết bị.

### Mô hình mạng



Hình 9: Mô hình bus USB

Các thiết bị hoạt động theo chuẩn USB được kết nối với nhau theo đồ hình mạng hình sao phân cấp. Trung tâm của mỗi hình sao này là các Hub. Một Hub ở tại trung tâm của mỗi sao, mỗi đoạn dây là một kết nối từ điểm tới điểm giữa Host và một Hub hoặc một chức năng nào đó, hoặc một Hub nối tới Hub khác hoặc chức năng khác Trong đồ hình như vậy, các thiết bị USB được chia làm 3 loại chính:

* **USB Host**: Chỉ có duy nhất một Host trong 1 hệ thống USB bất kỳ, là thiết bị đóng vai trò điều khiển toàn bộ mạng USB (có thể lên tới tối đa 126 thiết bị). Ví dụ như trên máy tính, USB Host được gắn trên mainboard. Để giao tiếp và điều khiển các USB device, USB Host controller cần được thiết kế tích hợp với USB RootHub (Hub mức cao nhất). Vai trò của thiết bị USB Host:

• Trao đổi dữ liệu với các USB Device

• Điều khiển USB Bus:

+ Quản lý các thiết bị cắm vào hay rút ra khỏi Bus USB qua quá trình điểm danh (Enumeration)

+ Phân xử, quản lý luồng dữ liệu trên Bus, đảm bảo các thiết bị đều có cơ hội trao đổi dữ liệu tùy thuộc vào cấu hình của mỗi thiết bị.

* **USB Device**: là các thiết bị đóng vai trò như các slave giao tiếp với USB Host. Xin lưu ý một điều hết sức quan trọng đó là các thiết bị này hoàn toàn đóng vai trò bị động, không bao giờ được tự ý gửi gói tin lên USB Host hay gửi gói tin giữa các USB Device với nhau, tất cả đều phải thông qua quá trình điều phối của USB Host. Các bạn sẽ hiểu cơ chế này rõ hơn trong phần truyền thông của chuẩn USB. Chức năng của thiết bị USB Device:

• Trao đổi dữ liệu với USB Host.

• Phát hiện gói tin hay yêu cầu từ USB Host theo giao thức USB.

* **USB Hub**: đóng vai trò như các Hub trong mạng Ethernet của chúng ta. Cấp nguồn cho các thiết bị USB.

### Kịch bản hoạt động

Quá trình hoạt động của chuẩn USB có thể được chia làm hai giai đoạn chính:

* **Quá trình điểm danh**: là quá trình USB Host phát hiện các thiết bị cắm vào và rút ra khỏi đường USB Bus. Mỗi khi một thiết bị tham gia vào Bus USB, USB Host sẽ tiến hành đọc các thông tin mô tả (Description) của USB Device, từ đó thiết lập địa chỉ (NodeID) và chế độ hoạt động tương ứng cho thiết bị USB Device. Các địa chỉ sẽ được đánh từ 1->126 nên về lý thuyết, chuẩn USB cho phép kết nối 126 thiết bị vào đường Bus. Khi thiết bị rút ra khỏi đường Bus, địa chỉ này sẽ được thu hồi.
* **Quá trình truyền dữ liệu**: Đứng ở góc độ mức hệ thống, các Interface chính là các dịch vụ khác nhau mà thiết bị đó cung cấp còn các Endpoint chính là các cổng cần thiết cho mỗi dịch vụ. Tương ứng với khái niệm trong kiến trúc TCP/IP, ví dụ giao thức FTP là giao thức sử dụng để truyền file sẽ sử dụng hai cổng 20,21. Trong khi đó giao thức HTTP lại sử dụng port 80, giao thức Telnet sử dụng port 23.

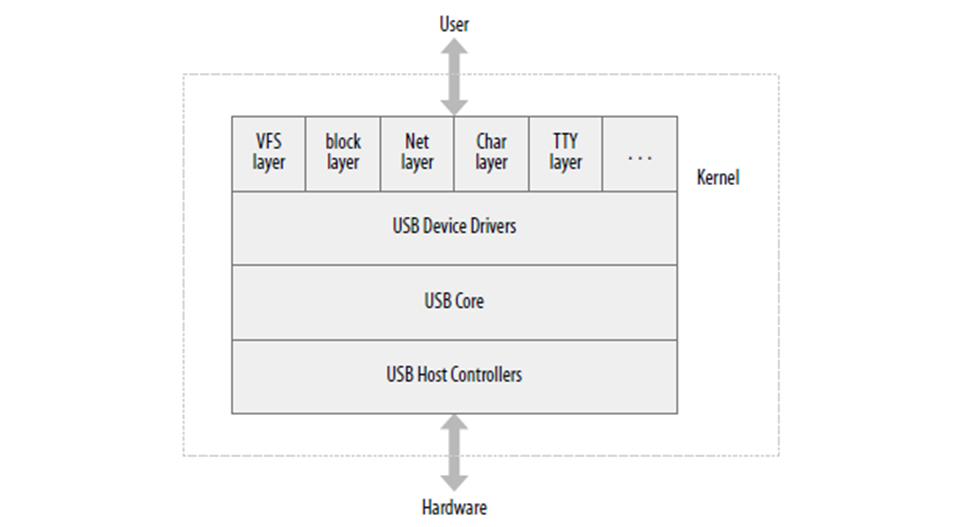
Thực tế các Endpoint cũng như các Port trong chuẩn TCP/IP đóng vai trò như các bộ đệm truyền/nhận dữ liệu. Nhờ việc sử dụng nhiều bộ đệm mà các quá trình truyền thông được tiến hành song song và cho tốc độ cao hơn, bên cạnh đó giúp cho việc phân tách các dịch vụ khác nhau. Với chuẩn USB, các thiết bị được thiết kế với tối đa là 16 Endpoint.

### Chế độ truyền

Chuẩn USB cung cấp cho chúng ta tổng cộng là 4 chế độ truyền, đáp ứng nhiều mục đích khác nhau tùy thuộc vào cơ chế truyền cũng như tốc độ mà người thiết kế mong muốn.

* Truyền điều khiển (Control transfer): là chế độ truyền được tất cả các thiết bị USB hỗ trợ để truyền các thông tin điều khiển với tốc độ tương đối chậm.
* Truyền ngắt (Interrupt transfer): sử dụng cho các thiết bị cần truyền một lượng dữ liệu nhỏ, tuần hoàn theo thời gian ví dụ như chuột, bàn phím. Khi đó, ví dụ cứ 10s một lần USB Host sẽ gửi request xuống và USB Device sẽ trả dữ liệu về cho USB Host (với trường hợp Interrupt In Endpoint).
* Truyền theo khối (Bulk transfer): sử dụng cho các thiết bị cần truyền một lượng dữ liệu lớn, yêu cầu độ chính xác tuyệt đối, không có ràng buộc quá chặt chẽ về thời gian thực ví dụ như thẻ nhớ USB, máy in. Cái này tương tự như giao thức TCP trong mạng Ethernet
* Truyển đẳng thời (Isochronos transfer): sử dụng cho các thiết bị cần truyền một lượng dữ liệu lớn với tốc độ rất nhanh, đảm bảo ràng buộc về thời gian thực tuy nhiên chấp nhận hy sinh độ chính xác ở một mức nhất định như các thiết bị nghe nhạc, xem phim kết nối theo chuẩn USB. Chuẩn này tương tự giao thức UDP trong mạng Ethernet.

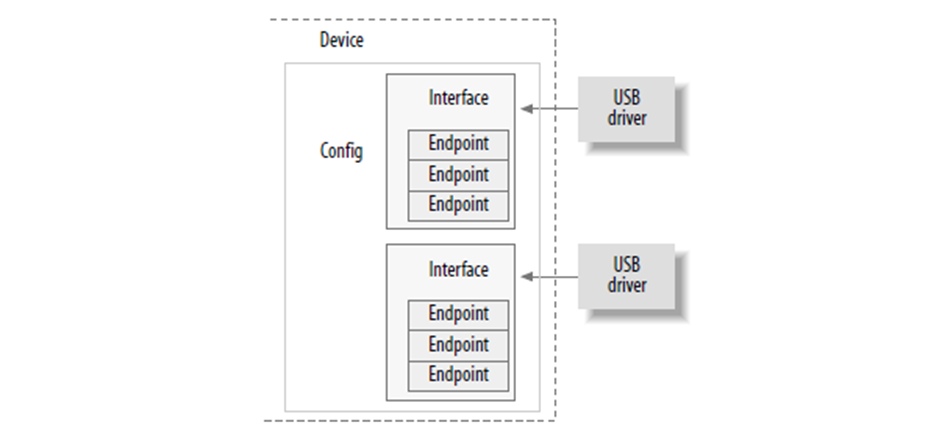
### Host view



Hình 10: USB driver overview

Hệ thống USB trên Linux được phân làm nhiều tầng. Tầng USB Device Drivers nằm giữa 2 tầng: USB Core và các hệ thống con khác nhau của nhân (Kernel Subsystems) như: hệ thống con thiết bị kiểu khối (Block), hệ thống con thiết bị kiểu kí tự (Char), hệ thống con thiết bị đầu vào (Input)... USB Core do nhân hệ điều hành Linux cung cấp, với các giao diện lập trình (Application Programming Interface – API) nhằm hỗ trợ các thiết bị USB và USB Host Controller. Mục tiêu của USB Core là trừu tượng hóa tất cả các phần cứng và các thành phần phụ thuộc thiết bị. Các USB Device Driver sẽ sử dụng lại các dịch vụ của tầng USB Core mà không cần quan tâm tới các nền tảng phần cứng khác nhau. USB Device Driver cũng cần xuất ra các API lên tầng trên. Tùy theo thiết bị có thể được xếp vào các lớp con như lớp vào dữ liệu (Input), lớp âm thanh (Sound), lớp bộ nhớ (memory) mà Driver có thể xuất ra các API khác nhau. Các ứng dụng người dùng khi muốn giao tiếp với thiết bị phần cứng sẽ sử dụng các API này để yêu cầu USB Device Driver thực hiện

### Device view



Hình 11: Mô hình device view

Tất cả các thiết bị USB hợp lệ đều chứa một hoặc nhiều cấu hình. Cấu hình của Thiết bị USB giống như một cấu hình, trong đó cấu hình mặc định là cấu hình thường được sử dụng. Linux chỉ hỗ trợ một cấu hình cho mỗi thiết bị.

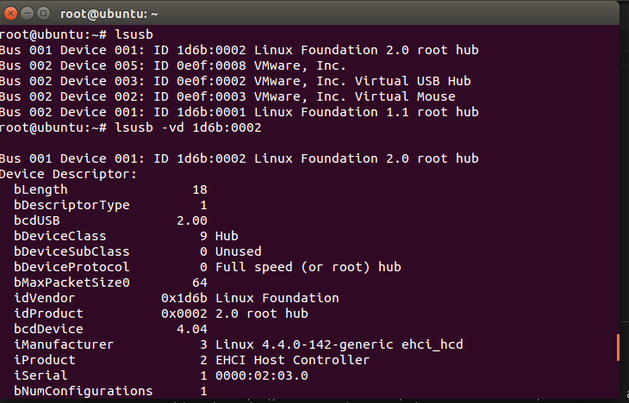
# QUI TRÌNH VIẾT USB DEVICE DRIVER

## Tìm hiểu về thiết bị usb muốn giao tiếp (usb device)

Đầu tiên cần có thông tin về Firmware trên thiết bị. Các thông tin cần thiết bao gồm: idVendor, idProduct, số lượng Configuration, số lượng Interface trong từng Configuration, số lượng và loại Endpoint trong từng Interface.

Nếu chúng ta là người phát triển firmware trên thiết bị thì hiển nhiên biết các thông tin này. Hoặc có thể tìm trong tài liệu đi kèm thiết bị. Hoặc cũng có một vài công cụ giúp xác định thông tin về thiết bị usb như USB Snoopy, USB Control Center (của Cypress) trên Windows, hoặc lệnh lsusb trên Linux.

Trên hệ điều hành Linux ta chỉ việc kết nối thiết bị tới máy tính, chạy lệnh ***lsusb*** trên terminal, tất cả các thiết bị USB đang kết nối với máy tính sẽ được liệt kê ra. Từ đó ta có thể biết được idVendor và idProduct của thiết bị. Tiếp tục, gõ lệnh ***lsusb –vd <idVendor>:<idProduct>*** để hiển thị các thông tin về cấu hình USB của thiết bị. Ta được kêt quả như sau:



## Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver

Khi viết một USB Driver người lập trình viên cần chỉ định xem Driver này sẽ được sử dụng cho các thiết bị hoặc các lớp thiết bị nào. Cấu trúc *usb\_device\_id* cung cấp một kiểu thiết bị, nó có thể là một thiết bị cụ thể cũng có thể là một lớp thiết bị. Có một số macro để khởi tạo cấu trúc này.

Ví dụ sử dụng macro USB\_DEVICE (vendor, product) để tạo ra một cấu trúc *usb\_device\_id* với IDvendor và IDproduct.

Sau khi đã tạo ra một cấu trúc *usb\_device\_id* ta cần phải khai báo cấu trúc này với USB Core, để làm việc này ta sử dụng macro

MODULE\_DEVICE\_TABLE(usb, usb\_device\_id[]);

Trong đó usb\_device\_id[] là một mảng các cấu trúc usb\_device\_id đã khởi tạo trước đó.

Ví dụ:

static struct usb\_device\_id mydev\_table[] = {

     { USB\_DEVICE (0x2179, 0x1989) }, { }

};

MODULE\_DEVICE\_TABLE(usb, mydev\_table);

## Khai báo cấu trúc dữ liệu liên quan tới thiết bị

Trong quá trình thăm dò thiết bị, có rất nhiều thông tin về thiết bị ta truy nhập được và cần phải lưu lại để sử dụng về sau. Các thông tin cần thiết bao gồm: thiết bị cụ thể (được xác định bởi cấu trúc **usb\_device**), thông tin các Configuration, Interface, Endpoint của thiết bị. Tùy theo đặc điểm của từng phần cứng cụ thể mà ta định nghĩa một cấu trúc dữ liệu phù hợp.

Ví dụ: Thiết bị có 1 Configuration, Configuration đó có 1 Interface, Interface có 2 Endpoint là Bulk IN và Bulk OUT, ta có thể định nghĩa một cấu trúc dữ liệu như sau:

//cấu trúc dữ liệu lưu trữ các thông tin về thiết bị,

//endpoint, bộ đệm

struct usb\_mydevice {

//con trỏ tới cấu trúc mô tả thiết bị

     struct usb\_device         \*udev;

     struct usb\_interface      \*interface;

//bộ đệm dữ liệu vào

     unsigned char              \*bulk\_in\_buffer;

//kích thước bộ đệm dữ liệu vào

     size\_t                    bulk\_in\_size;

//địa chỉ 2 Endpoint Bulk IN và OUT

     \_\_u8                      bulk\_in\_endpointAddr;

     \_\_u8                      bulk\_out\_endpointAddr;

};

## Đăng kí và hủy đăng kí USB Device Driver

Để tầng USB Core có thể nhận ra Driver, lập trình viên cần phải đăng kí Driver đó. Sử dụng hàm sau để làm việc này:

usb\_register(struct usb\_driver &);

Hàm này thường được gọi trong hàm khởi tạo mô-đun Driver đang viết.Tham số cần truyền cho hàm này là một con trỏ tới cấu trúc usb\_driver. Cấu trúc này bao gồm các thông tin về Driver đang viết. Cấu trúc này được định nghĩa như sau:

struct usb\_driver {

     const char \* name;

     int (\* probe) (struct usb\_interface \*intf,

const struct usb\_device\_id \*id);

     void (\* disconnect) (struct usb\_interface \*intf);

     int (\* ioctl) (struct usb\_interface \*intf,

unsigned int code, void \*buf);

     int (\* suspend) (struct usb\_interface \*intf,

pm\_message\_t message);

     int (\* resume) (struct usb\_interface \*intf);

     int (\* reset\_resume) (struct usb\_interface \*intf);

     int (\* pre\_reset) (struct usb\_interface \*intf);

     int (\* post\_reset) (struct usb\_interface \*intf);

     const struct usb\_device\_id \* id\_table;

     struct usb\_dynids dynids;

     struct usbdrv\_wrap drvwrap;

     unsigned int no\_dynamic\_id:1;

     unsigned int supports\_autosuspend:1;

};

Các thông tin quan trọng gồm:

* const char\* name : tên của Driver
* const struct usb\_device\_id\* id\_table : con trỏ tới bảng chứa các thiết bị sẽ được điều khiển bởi Driver này và đã được khai báo bằng macro MODULE\_DEVICE\_TABLE().
* int (\*probe) (struct usb\_interface \*intf, const struct usb\_device\_id\* id) : đây là một tham số rất quan trọng. Tham số này là một con trỏ tới một hàm (hàm thăm dò), hàm này sẽ được gọi khi thiết bị được kết nối tới hệ thống. Trong hàm này ta sẽ thực hiện các công việc quan trọng như xác định các Endpoint, cấp phát bộ nhớ…
* void (\*disconnect) (struct usb\_interface\* intf) : Một con trỏ tới một hàm (hàm ngắt kết nối), hàm này sẽ được gọi khi thiết bị được gỡ bỏ ra khỏi hệ thống. Trong hàm này lập trình viên cần phải thực hiện công tác dọn dẹp hệ thống như giải phóng bộ nhớ, hủy các công việc đang dở dang...

Để hủy đăng kí một USB Device Driver ra khỏi hệ thống, ta sử dụng hàm sau:

usb\_deregister(struct usb\_driver &);

Hàm này thường được gọi trong hàm kết thúc mô-đun Driver đang viết.

## Hàm thăm dò thiết bị (probe)

Khi thiết bị mới được kết nối tới hệ thống, nếu Driver được chỉ định cho điều khiển thiết bị đó thì hàm thăm dò của Driver sẽ được gọi. USB Core truyền tới hàm thăm dò một con trỏ tới cấu trúc usb\_interface mô tả Interface được chọn trên thiết bị.

Nguyên mẫu hàm thăm dò như sau:

int (\*probe) (struct usb\_interface \*intf, const struct usb\_device\_id\* id);

Trong hàm thăm dò, Driver cần thực hiện một số công việc sau:

* Lấy ra địa chỉ các Endpoint cần dùng, lấy ra kích thước các bộ đệm cho thiết bị
* Cấp phát bộ đệm
* Lưu lại các thông tin (địa chỉ Endpoint, kích thước bộ đệm, địa chỉ bộ đệm…)
* Đăng kí lớp thiết bị cho Driver

### Truy nhập các thông tin về Endpoint

Đoạn mã sau thực hiện việc lấy các thông tin của một Endpoint Bulk IN và một Endpoint Bulk OUT:

iface\_desc = interface->cur\_altsetting;

for (i = 0; i < iface\_desc->desc.bNumEndpoints; ++i) {

     endpoint = &iface\_desc->endpoint[i].desc;

     //bulk\_in

     if (!dev->bulk\_in\_endpointAddr &&

usb\_endpoint\_is\_bulk\_in(endpoint)) {

           buffer\_size = le16\_to\_cpu(endpoint->wMaxPacketSize);

           dev->bulk\_in\_size = buffer\_size;

           dev->bulk\_in\_endpointAddr = endpoint->bEndpointAddress;

           dev->bulk\_in\_buffer = kmalloc(buffer\_size, GFP\_KERNEL);

           if (!dev->bulk\_in\_buffer) {

                err("Could not allocate bulk\_in\_buffer");

                goto error;

           }

     }

     //bulk\_out

     if (!dev->bulk\_out\_endpointAddr &&

           usb\_endpoint\_is\_bulk\_out(endpoint)) {

           dev->bulk\_out\_endpointAddr = endpoint->bEndpointAddress;

     }

}

### Lưu lại các thông tin đã truy nhập và cấp phát

Để lưu trữ các dữ liệu liên quan tới thiết bị nhằm mục đích sử dụng sau, ta sử dụng hàm sau:

usb\_set\_intfdata(intf, dev);

Trong đó dev là con trỏ tới cấu trúc dữ liệu ta cần lưu trữ. (cấu trúc dữ liệu giao tiếp với thiết bị được người lập trình khai báo thích hợp với dữ liệu thiết bị cung cấp)

### Đăng kí lớp thiết bị

Như ta đã thấy trong mô hình phân lớp của hệ thống USB trên Linux, phía trên lớp USB Device Driver là lớp Các hệ thống con nhân Linux (hay Các lớp thiết bị), bao gồm: Char, Block, TTY, Mem… Các thiết bị USB có thể được đối xử theo một trong các hệ thống con đó, chẳng hạn như chuột, bàn phím có thể coi như thuộc lớp Input. Tức là các ứng dụng người dùng sẽ sử dụng các API lớp Input để thực hiện giao tiếp với chuột, bàn phím. Do đó Driver cũng cần đăng kí thực hiện đăng kí lớp thiết bị mà nó quản lý. Tùy theo từng lớp có các hàm đăng kí khác nhau:

* Nếu Driver muốn đối xử với thiết bị như kiểu Input, ta sử dụng hàm:

input\_register\_device(struct input\_dev\* )

* Nếu Driver muốn đối xử với thiết bị như một thiết bị kiểu kí tự thông thường ta sử dụng hàm:

usb\_register\_dev(structusb\_interface\*,structusb\_class\_driver\* )

* Tham số đầu tiên cần truyền cho hàm **usb\_register\_dev** là con trỏ tới cấu trúc chứa thông tin về Interface được chọn trên thiết bị, tham số thứ hai là con trỏ tới cấu trúc **usb\_class\_driver**. Cấu trúc này có một trường quan trọng là một con trỏ tới cấu trúc **file\_operations** định nghĩa các thao tác trên thiết bị được Driver hỗ trợ. Hai cấu trúc này được định nghĩa như dưới đây:

struct file\_operations {

     struct module \*owner;

     loff\_t (\*llseek) (struct file \*, loff\_t, int);

     ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);

     ssize\_t (\*write) (struct file \*,

const char \*, size\_t, loff\_t \*);

     int (\*readdir) (struct file \*, void \*, filldir\_t);

     unsigned int (\*poll) (struct file \*,

struct poll\_table\_struct \*);

     int (\*ioctl) (struct inode \*,

struct file \*, unsigned int, unsigned long);

     int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

     int (\*open) (struct inode \*, struct file \*);

     int (\*flush) (struct file \*);

     int (\*release) (struct inode \*, struct file \*);

     int (\*fsync) (struct file \*, struct dentry \*, int datasync);

     int (\*fasync) (int, struct file \*, int);

     int (\*lock) (struct file \*, int, struct file\_lock \*);

     ssize\_t (\*readv) (struct file \*, const struct iovec \*,

                               unsigned long, loff\_t \*);

     ssize\_t (\*writev) (struct file \*, const struct iovec \*,

                                unsigned long, loff\_t \*);

     ssize\_t (\*sendpage) (struct file \*, struct page \*, int,

                                size\_t, loff\_t \*, int);

     unsigned long (\*get\_unmapped\_area)(struct file \*,

unsigned long, unsigned long,

unsigned long,unsigned long);

};

Trong cấu trúc **file\_operations** các trường thông tin chính là các con trỏ hàm. Các con trỏ này trỏ tới các hàm tương ứng với các thao tác trên tệp tin như: mở, đọc, ghi, lọc, xóa… Chúng ta cần cài đặt các hàm này để khi có hành động tương ứng trên tệp tin, hàm của chúng ta sẽ thực thi các công việc cần thiết như: chuẩn bị dữ liệu, chuyển dữ liệu từ không gian nhận sang không gian người dùng và ngược lại, giải phóng dữ liệu…

## Hàm ngắt kết nối thiết bị

Khi thiết bị được gỡ bỏ ra khỏi hệ thống, hàm ngắt kết nối được gọi. Nguyên mẫu hàm như sau:

**void (\*disconnect) (struct usb\_interface\* intf);**

Trong hàm disconnect cần thực hiện hai công việc sau:

* Hủy các dữ liệu về thiết bị đã lưu trữ từ hàm thăm dò, để làm điều này ta sẽ thiết lập dữ liệu NULL cho interface intf:
* usb\_set\_intfdata(intf, NULL);
* Hủy đăng kí lớp thiết bị:
* usb\_deregister\_dev(struct usb\_interface\* , struct usb\_class\_driver\* );

## Các hàm mở / đọc / ghi thiết bị

### Hành động mở tệp tin thiết bị

**static int mydevice\_open(struct inode \*inode, struct file \*file);**

Hành động này có tác dụng chuẩn bị cho các hành động đọc, ghi sau đó. Trong hàm này ta sử dụng hàm **usb\_get\_intfdata()** để lấy ra các thông tin liên quan tới thiết bị đã lưu trữ từ hàm **probe()** (bằng hàm **usb\_set\_intfdata**), và thiết lập dữ liệu này cho cấu trúc file.

dev = usb\_get\_intfdata(interface);

file->private\_data = dev;

Cấu trúc **file** (được định nghĩa trong *<linux/fs.h*>) là một cấu trúc rất quan trọng trong Driver. Chúng ta cần chú ý rằng đây là một cấu trúc dữ liệu trong không gian nhân và cấu trúc này sẽ không liên quan gì tới con trỏ FILE\* trong thư viện của ngôn ngữ C trong không gian người dùng. Cấu trúc **file** thể hiện một tệp tin đang mở trong hệ thống Linux. Khi một tệp tin được mở, một thể hiện của cấu trúc này được tạo ra và liên kết với tệp tin đó. Khi bất kỳ hàm nào (đọc, ghi…) thao tác trên tệp tin, thể hiện của cấu trúc này sẽ được truyền cho hàm đó.

### Hành động đọc / ghi

#### Hành động đọc

static ssize\_t mydevice\_read(struct file \*file, char \_\_user \*buffer, size\_t count, loff\_t \*ppos);

Hàm này sẽ thực hiện hai công việc ngược lại với hàm đọc. Đầu tiên nó phải chuyển dữ liệu từ không gian người dùng sang không gian nhân hệ điều hành, sau đó nó cần đóng gói dữ liệu này trong các URB và xác nhận các URB này tới tầng USB Core. Để copy dữ liệu người dùng vào không gian nhân, ta sử dụng hàm:

unsigned long copy\_to\_user(void \_\_user \* to, const void \* from, unsigned long size);

#### Hành động ghi

static ssize\_t my\_device  write(struct file \*file, const char \_\_user \*buffer, size\_t count, loff\_t \*ppos);

Hàm này sẽ thực hiện hai công việc ngược lại với hàm đọc. Đầu tiên nó phải chuyển dữ liệu từ không gian người dùng sang không gian nhân hệ điều hành, sau đó nó cần đóng gói dữ liệu này trong các URB và xác nhận các URB này tới tầng USB Core. Để copy dữ liệu người dùng vào không gian nhân, ta sử dụng hàm:

unsigned long copy\_from\_user(void \* to, const void \_\_user \* from, unsigned long size);

# KẾT LUẬN

Nhóm em đã thực hiện thành đề tài lập trình device driver (cho chuột usb) trên linux. Đây là một bài tập tương đối phức tạp, đòi hỏi nhiều thời gian, công sức để nghiên cứu thực hiện. Qua bài tập lớn nay đã giúp nhóm em hiểu sâu hơn về hệ điều hành linux. Nhóm em một lần nữa cảm ơn thầy Phạm Doãn Tĩnh đã hướng dẫn chỉ dạy trên lớp giúp chúng em hoàn thành bài tập lớn này.

Ngoài ra qua bài tập lớn này, chúng em cũng gặp không ít khó khăn như thời gian, tài liệu cũng như kiến thức hạn hẹp nên còn nhiều sai sót. Mong thầy góp ý để bài tập lớn này được hoàn thiện hơn.

Em xin chân thành cảm ơn!

TÀI LIỆU THAM KHẢO

[1] <https://sites.google.com/site/embedded247/embedded_system/usbprotocol/>

[2] <https://www.linuxtoday.com/blog/linux-mouse-drivers/>

[3] <http://matthias.vallentin.net/blog/2007/04/writing-a-linux-kernel-driver-for-an-unknown-usb-device/>

PHỤ LỤC I

|  |
| --- |
| #include <linux/module.h>  #include <linux/fs.h>  #include <linux/string.h>  #include <linux/fcntl.h>  #include <linux/unistd.h>  #include <linux/buffer\_head.h>  #include <asm/segment.h>  #include <asm/uaccess.h>  MODULE\_LICENSE**(**"GPL"**);** // avoid kernel taint warning  MODULE\_DESCRIPTION**(**"Mouse Driver"**);**  MODULE\_AUTHOR**(**"Long"**);**  static int times **=** 0**;**  char info\_buff**[**4**];**  int k**,** btn\_left**,** btn\_right**,** btn\_middle**;**  struct file **\***filehandle1**,** **\***filehandle2**;**  static int dev\_open**(**struct inode **\*,** struct file **\*);**  static int dev\_rls**(**struct inode **\*,** struct file **\*);**  static ssize\_t dev\_read**(**struct file **\*,** char **\*,** size\_t**,** loff\_t **\*);**  static ssize\_t dev\_write**(**struct file **\*,** const char **\*,** size\_t**,** loff\_t **\*);**  /\*  Callbacks required to implement a device driver  \*/  static struct file\_operations fops **=**  **{**  **.**read **=** dev\_read**,**  **.**open **=** dev\_open**,**  **.**write **=** dev\_write**,**  **.**release **=** dev\_rls**,**  **};**  /\*  The following functions help in reading/writing to files  within a Linux kernel module. File I/O should be avoided when possible.  Call VFS level functions instead of the syscall handler directly.  \*/  struct file**\*** file\_open**(**const char**\*** path**,** int flags**,** int rights**)**  **{**  struct file**\*** filp **=** **NULL;**  mm\_segment\_t oldfs**;**  int err **=** 0**;**  oldfs **=** get\_fs**();**  set\_fs**(**get\_ds**());**  filp **=** filp\_open**(**path**,** flags**,** rights**);**  set\_fs**(**oldfs**);**  **if** **(**IS\_ERR**(**filp**))**  **{**  err **=** PTR\_ERR**(**filp**);**  **return** **NULL;**  **}**  **return** filp**;**  **}**  void file\_close**(**struct file**\*** file**)**  **{**  filp\_close**(**file**,** **NULL);**  **}**  int file\_read**(**struct file**\*** file**,** unsigned long long offset**,** unsigned char**\*** data**,** unsigned int size**)**  **{**  mm\_segment\_t oldfs**;**  int ret**;**  oldfs **=** get\_fs**();**  set\_fs**(**get\_ds**());**  ret **=** vfs\_read**(**file**,** data**,** size**,** **&**offset**);**  set\_fs**(**oldfs**);**  **return** ret**;**  **}**  int file\_write**(**struct file**\*** file**,** unsigned long long offset**,** unsigned char**\*** data**,** unsigned int size**)**  **{**  mm\_segment\_t oldfs**;**  int ret**;**  oldfs **=** get\_fs**();**  set\_fs**(**get\_ds**());**  ret **=** vfs\_write**(**file**,** data**,** size**,** **&**offset**);**  set\_fs**(**oldfs**);**  **return** ret**;**  **}**  /\*  Called when module is loaded  \*/  int init\_module**(**void**)**  **{**  // register driver with major number 45 and name mousedev  // returns a non-negative number on success  int t **=** register\_chrdev**(**45**,** "mousedev"**,** **&**fops**);**  **if** **(**t **<** 0**)** printk**(**KERN\_ALERT "Device registration failed ...\n"**);**  **else** printk**(**KERN\_ALERT "Device registered ...\n"**);**  **return** t**;**  **}**  /\*  Called when module is unloaded  \*/  void cleanup\_module**(**void**)**  **{**  unregister\_chrdev**(**45**,** "mousedev"**);**  **}**  /\*  Called when system call 'open' is done on the device file  \*/  static int dev\_open**(**struct inode **\***inod**,** struct file **\***fil**)**  **{**  times**++;**  printk**(**KERN\_ALERT "Device mouse by Phu Duc opened %d times\n"**,** times**);**  **return** 0**;**  **}**  /\*  Called when system call 'read' is done on the device file  \*/  static ssize\_t dev\_read**(**struct file **\***filp**,** char **\***buff**,** size\_t len**,** loff\_t **\***off**)**  **{**  filehandle1 **=** file\_open**(**"/home/long/Documents/Code/mouse-driver/info"**,** 0**,** 0**);**  file\_read**((**struct file**\*)**filehandle1**,** 0**,** info\_buff**,** 1**);**  filehandle2 **=** file\_open**(**"/dev/input/mice"**,** 0**,** 0**);**  file\_read**(**filehandle2**,** 0**,** buff**,** 1**);**  btn\_left **=** buff**[**0**]** **&** 0x1**;**  btn\_right **=** buff**[**0**]** **&** 0x2**;**  btn\_middle **=** buff**[**0**]** **&** 0x04**;**  int info **=**0**;**  info **=** info\_buff**[**0**]-**'0'**;**  **if** **(**btn\_left **>** 0**)**  **{**  printk**(**"\nLeft click!"**);**  info**++;**  printk**(**info\_buff**);**  **}**  **else** **if** **(**btn\_right **>** 0**)**  **{**  printk**(**"\nRight click!"**);**  info**--;**  printk**(**info\_buff**);**  **}**  **else**  **{**  // printk("\nMiddle click!");  // //do nothing  // printk(info\_buff);  **}**  info\_buff**[**0**]** **=** info **+** '0'**;**  info\_buff**[**1**]** **=** info **+** '8'**;**  file\_close**((**struct file**\*)**filehandle1**);**  filehandle1 **=** file\_open**(**"/home/long/Documents/Code/mouse-driver/info"**,** 1**,** 0**);**  file\_write**((**struct file**\*)**filehandle1**,** 0**,** info\_buff**,** 2**);**  file\_close**((**struct file**\*)**filehandle1**);**  **return** 0**;**  **}**  /\*  Called when system call 'write' is done on the device file  \*/  static ssize\_t dev\_write**(**struct file **\***filp**,** const char **\***buff**,** size\_t len**,** loff\_t **\***off**)**  **{**  **return** 0**;**  **}**  /\*  Called when system call 'close' is done on the device file  \*/  static int dev\_rls**(**struct inode **\***inod**,** struct file **\***fil**)**  **{**  printk**(**KERN\_ALERT "Device closed.\n"**);**  **return** 0**;**  **}** |

PHỤ LỤC II

|  |
| --- |
| #define POLLING\_RATE 250  #define ACCEL\_RECIPROCAL 1  #define SCROLL\_RATIO 5  #define BMAX 900  #include <linux/kernel.h>  #include <linux/slab.h>  #include <linux/module.h>  #include <linux/init.h>  #include <linux/usb/input.h>  #include <linux/hid.h>  #include <linux/fs.h>  #include <linux/string.h>  #include <linux/fcntl.h>  #include <linux/unistd.h>  #include <linux/buffer\_head.h>  #include <asm/segment.h>  #include <asm/uaccess.h>  /\*  \* Version Information  \*/  #define DRIVER\_VERSION "v1.6a"  #define DRIVER\_AUTHOR "Vojtech Pavlik <vojtech@ucw.cz>"  #define DRIVER\_DESC "USB HID Boot Protocol mouse driver with acceleration"  #define DRIVER\_LICENSE "GPL"  MODULE\_AUTHOR**(**DRIVER\_AUTHOR**);**  MODULE\_DESCRIPTION**(**DRIVER\_DESC**);**  MODULE\_LICENSE**(**DRIVER\_LICENSE**);**  // struct file \*filehandle;  // char brightness\_buff[6];  // static int brightness;  struct usb\_mouse  **{**  char name**[**128**];**  char phys**[**64**];**  struct usb\_device **\***usbdev**;**  struct input\_dev **\***dev**;**  // USB Request Block (URB)  struct urb **\***irq**;**  // Data from mouse  signed char **\***data**;**  dma\_addr\_t data\_dma**;**  **};**  static void usb\_mouse\_irq**(**struct urb **\***urb**)**  **{**  struct usb\_mouse **\***mouse **=** urb**->**context**;**  signed char **\***data **=** mouse**->**data**;**  struct input\_dev **\***dev **=** mouse**->**dev**;**  int status**;**  // acceleration happens here  signed int pr **=** POLLING\_RATE**;**  signed int accel\_r **=** ACCEL\_RECIPROCAL **\*** 1000**;**  signed int delta\_x **=** data**[**2**];**  signed int delta\_y **=** data**[**4**];**  signed int rate **=** int\_sqrt**(**delta\_x **\*** delta\_x **+** delta\_y **\*** delta\_y**);**  signed int accel\_x **=** **(**rate **\*** delta\_x**)** **\*** pr **/** accel\_r**;**  signed int accel\_y **=** **(**rate **\*** delta\_y**)** **\*** pr **/** accel\_r**;**  delta\_x **+=** accel\_x**;**  delta\_y **+=** accel\_y**;**  **switch** **(**urb**->**status**)**  **{**  **case** 0**:** /\* success \*/  **break;**  **case** **-**ECONNRESET**:** /\* unlink \*/  **case** **-**ENOENT**:**  **case** **-**ESHUTDOWN**:**  **return;**  /\* -EPIPE: should clear the halt \*/  **default:** /\* error \*/  **goto** resubmit**;**  **}**  input\_report\_key**(**dev**,** BTN\_LEFT**,** data**[**1**]** **&** 0x01**);**  input\_report\_key**(**dev**,** BTN\_RIGHT**,** data**[**1**]** **&** 0x02**);**  input\_report\_key**(**dev**,** BTN\_MIDDLE**,** data**[**1**]** **&** 0x04**);**  input\_report\_key**(**dev**,** BTN\_SIDE**,** data**[**1**]** **&** 0x08**);**  input\_report\_key**(**dev**,** BTN\_EXTRA**,** data**[**1**]** **&** 0x10**);**  input\_report\_rel**(**dev**,** REL\_X**,** delta\_x**);**  input\_report\_rel**(**dev**,** REL\_Y**,** delta\_y**);**  input\_report\_rel**(**dev**,** REL\_WHEEL**,** data**[**6**]<<**SCROLL\_RATIO**);**  input\_sync**(**dev**);**  // filehandle = filp\_open("/home/long/Documents/mousedriver/Info", 0, 0);  // kernel\_read((struct file \*)filehandle, brightness\_buff, 2, 0);  // brightness = (brightness\_buff[0] - '0') \* 10;  // brightness += (brightness\_buff[1] - '0');  // if ((data[1] & 0x01) > 0)  // {  // brightness += 5;  // if (brightness > BMAX)  // brightness = BMAX;  // if (brightness < 500)  // brightness = 500;  // }  // else if ((data[1] & 0x02) > 0)  // {  // brightness -= 5;  // if (brightness > BMAX)  // brightness = BMAX;  // if (brightness < 500)  // brightness = 500;  // }  // brightness\_buff[0] = brightness / 10 + '0';  // brightness\_buff[1] = brightness % 10 + '0';  // filp\_close((struct file \*)filehandle, NULL);  // filehandle = filp\_open("/home/long/Documents/mousedriver/Info", 1, 0);  // kernel\_write((struct file \*)filehandle, brightness\_buff, 2, 0);  // filp\_close((struct file \*)filehandle, NULL);  resubmit**:**  status **=** usb\_submit\_urb**(**urb**,** GFP\_ATOMIC**);**  **if** **(**status**)**  dev\_err**(&**mouse**->**usbdev**->**dev**,**  "can't resubmit intr, %s-%s/input0, status %d\n"**,**  mouse**->**usbdev**->**bus**->**bus\_name**,**  mouse**->**usbdev**->**devpath**,** status**);**  **}**  static int usb\_mouse\_open**(**struct input\_dev **\***dev**)**  **{**  struct usb\_mouse **\***mouse **=** input\_get\_drvdata**(**dev**);**  mouse**->**irq**->**dev **=** mouse**->**usbdev**;**  **if** **(**usb\_submit\_urb**(**mouse**->**irq**,** GFP\_KERNEL**))**  **return** **-**EIO**;**  **return** 0**;**  **}**  static void usb\_mouse\_close**(**struct input\_dev **\***dev**)**  **{**  struct usb\_mouse **\***mouse **=** input\_get\_drvdata**(**dev**);**  usb\_kill\_urb**(**mouse**->**irq**);**  **}**  static int usb\_mouse\_probe**(**struct usb\_interface **\***intf**,** const struct usb\_device\_id **\***id**)**  **{**  struct usb\_device **\***dev **=** interface\_to\_usbdev**(**intf**);**  struct usb\_host\_interface **\***interface**;**  struct usb\_endpoint\_descriptor **\***endpoint**;**  struct usb\_mouse **\***mouse**;**  struct input\_dev **\***input\_dev**;**  int pipe**,** maxp**;**  int error **=** **-**ENOMEM**;**  interface **=** intf**->**cur\_altsetting**;**  // Get endpoint information  **if** **(**interface**->**desc**.**bNumEndpoints **!=** 1**)**  **return** **-**ENODEV**;**  endpoint **=** **&**interface**->**endpoint**[**0**].**desc**;**  **if** **(!**usb\_endpoint\_is\_int\_in**(**endpoint**))**  **return** **-**ENODEV**;**  pipe **=** usb\_rcvintpipe**(**dev**,** endpoint**->**bEndpointAddress**);**  maxp **=** usb\_maxpacket**(**dev**,** pipe**,** usb\_pipeout**(**pipe**));**  // Alloc srtuct usb\_mouse  mouse **=** kzalloc**(sizeof(**struct usb\_mouse**),** GFP\_KERNEL**);**  input\_dev **=** input\_allocate\_device**();**  **if** **(!**mouse **||** **!**input\_dev**)**  **goto** fail1**;**  mouse**->**data **=** usb\_alloc\_coherent**(**dev**,** 8**,** GFP\_ATOMIC**,** **&**mouse**->**data\_dma**);**  **if** **(!**mouse**->**data**)**  **goto** fail1**;**  mouse**->**irq **=** usb\_alloc\_urb**(**0**,** GFP\_KERNEL**);**  **if** **(!**mouse**->**irq**)**  **goto** fail2**;**  mouse**->**usbdev **=** dev**;**  mouse**->**dev **=** input\_dev**;**  **if** **(**dev**->**manufacturer**)**  strlcpy**(**mouse**->**name**,** dev**->**manufacturer**,** **sizeof(**mouse**->**name**));**  **if** **(**dev**->**product**)**  **{**  **if** **(**dev**->**manufacturer**)**  strlcat**(**mouse**->**name**,** " "**,** **sizeof(**mouse**->**name**));**  strlcat**(**mouse**->**name**,** dev**->**product**,** **sizeof(**mouse**->**name**));**  **}**  **if** **(!**strlen**(**mouse**->**name**))**  snprintf**(**mouse**->**name**,** **sizeof(**mouse**->**name**),**  "USB HIDBP Mouse %04x:%04x"**,**  le16\_to\_cpu**(**dev**->**descriptor**.**idVendor**),**  le16\_to\_cpu**(**dev**->**descriptor**.**idProduct**));**  usb\_make\_path**(**dev**,** mouse**->**phys**,** **sizeof(**mouse**->**phys**));**  strlcat**(**mouse**->**phys**,** "/input0"**,** **sizeof(**mouse**->**phys**));**  input\_dev**->**name **=** mouse**->**name**;**  input\_dev**->**phys **=** mouse**->**phys**;**  usb\_to\_input\_id**(**dev**,** **&**input\_dev**->**id**);**  input\_dev**->**dev**.**parent **=** **&**intf**->**dev**;**  input\_dev**->**evbit**[**0**]** **=** BIT\_MASK**(**EV\_KEY**)** **|** BIT\_MASK**(**EV\_REL**);**  input\_dev**->**keybit**[**BIT\_WORD**(**BTN\_MOUSE**)]** **=** BIT\_MASK**(**BTN\_LEFT**)** **|**  BIT\_MASK**(**BTN\_RIGHT**)** **|** BIT\_MASK**(**BTN\_MIDDLE**);**  input\_dev**->**relbit**[**0**]** **=** BIT\_MASK**(**REL\_X**)** **|** BIT\_MASK**(**REL\_Y**);**  input\_dev**->**keybit**[**BIT\_WORD**(**BTN\_MOUSE**)]** **|=** BIT\_MASK**(**BTN\_SIDE**)** **|**  BIT\_MASK**(**BTN\_EXTRA**);**  input\_dev**->**relbit**[**0**]** **|=** BIT\_MASK**(**REL\_WHEEL**);**  input\_set\_drvdata**(**input\_dev**,** mouse**);**  input\_dev**->**open **=** usb\_mouse\_open**;**  input\_dev**->**close **=** usb\_mouse\_close**;**  usb\_fill\_int\_urb**(**mouse**->**irq**,** dev**,** pipe**,** mouse**->**data**,**  **(**maxp **>** 8 **?** 8 **:** maxp**),**  usb\_mouse\_irq**,** mouse**,** endpoint**->**bInterval**);**  mouse**->**irq**->**transfer\_dma **=** mouse**->**data\_dma**;**  mouse**->**irq**->**transfer\_flags **|=** URB\_NO\_TRANSFER\_DMA\_MAP**;**  error **=** input\_register\_device**(**mouse**->**dev**);**  **if** **(**error**)**  **goto** fail3**;**  // Save data & alloc  usb\_set\_intfdata**(**intf**,** mouse**);**  **return** 0**;**  fail3**:**  usb\_free\_urb**(**mouse**->**irq**);**  fail2**:**  usb\_free\_coherent**(**dev**,** 8**,** mouse**->**data**,** mouse**->**data\_dma**);**  fail1**:**  input\_free\_device**(**input\_dev**);**  kfree**(**mouse**);**  **return** error**;**  **}**  static void usb\_mouse\_disconnect**(**struct usb\_interface **\***intf**)**  **{**  struct usb\_mouse **\***mouse **=** usb\_get\_intfdata**(**intf**);**  usb\_set\_intfdata**(**intf**,** **NULL);**  **if** **(**mouse**)**  **{**  usb\_kill\_urb**(**mouse**->**irq**);**  input\_unregister\_device**(**mouse**->**dev**);**  usb\_free\_urb**(**mouse**->**irq**);**  usb\_free\_coherent**(**interface\_to\_usbdev**(**intf**),** 8**,** mouse**->**data**,** mouse**->**data\_dma**);**  kfree**(**mouse**);**  **}**  **}**  static struct usb\_device\_id usb\_mouse\_id\_table**[]** **=** **{**  **{**USB\_INTERFACE\_INFO**(**USB\_INTERFACE\_CLASS\_HID**,** USB\_INTERFACE\_SUBCLASS\_BOOT**,**  USB\_INTERFACE\_PROTOCOL\_MOUSE**)},**  **{}** /\* Terminating entry \*/  **};**  MODULE\_DEVICE\_TABLE**(**usb**,** usb\_mouse\_id\_table**);**  static struct usb\_driver usb\_mouse\_driver **=** **{**  **.**name **=** "leetmouse"**,**  **.**probe **=** usb\_mouse\_probe**,**  **.**disconnect **=** usb\_mouse\_disconnect**,**  **.**id\_table **=** usb\_mouse\_id\_table**,**  **};**  module\_usb\_driver**(**usb\_mouse\_driver**);** |