Đặt vấn đề

Lịch sử hướng giải quyết

Phạm vi của đề tài

Phương pháp nghiên cứu

**Lời mở đầu**

Cùng với sự phát triển không ngừng của khoa học công nghệ, công nghiệp ô tô cũng đã và đang được hoàn thiện nhằm mục đích đem đến cho người dùng những trải nghiệm ngày càng thoải mái và tiện nghi hơn. Sự ra đời của các công nghệ, thiết bị hiện đại như cảnh báo chệch làn đường, cảnh báo tiền va chạm, hệ thống hỗ trợ phanh và điều khiển động cơ bánh lái…đã mang lại cho người dùng những chuyến hành trình ngày càng thú vị. Tuy nhiên, để áp dụng những công nghệ mới này lên trên một sản phẩm xe ô tô thực tế để đưa ra ngoài thị trường thì từ khâu kiểm thử, vận hành cho đến phát triển các ứng dụng về sau thì đây cũng là một khó khăn đối với các nhà nghiên cứu, cũng như sản xuất ô tô công nghiệp. Việc sử dụng cả hệ thống thực tế chỉ để kiểm thử với một ECU (Electronic Control Unit) mới nhằm hướng tới sự tương thích giữa các ECU với nhau có thể gây ra sự phức tạp cho toàn bộ hệ thống CAN BUS (Controller Area Network ). Chính vì vậy, việc xây dựng nên một phần mềm có thể mô phỏng hoạt động như một hệ thống hoàn chỉnh nhằm thay thế các ECU thực tế là thật sự cần thiết. Nó sẽ giúp cho việc phát triển thêm ứng dụng đơn giản hơn, với độ tin cậy và hiệu quả cao.

Hiểu được tầm quan trọng của nó, nhóm chúng em đã đăng ký tham gia Đồ án tốt nghiệp dưới hình thức Capstone Project cùng FPT Software với đề tài “THIẾT KẾ VÀ THI CÔNG HỆ THỐNG MÔ PHỎNG CAN BUS” . Với nhiệm vụ thực hiện việc liên lạc giữa PC và thiết bị, nên nhóm em đã chọn tên đồ án là “LẬP TRÌNH LINUX DRIVER VÀ THIẾT KẾ GIAO THỨC TRUYỀN DỮ LIỆU”. Đồ án thực hiện dựa trên .Theo đó, đồ án sẽ được tổ chức thành 3 chương như sau:

**Chương 1: Tổng quan đề tài**

**Chương 2: Lập trình Linux Driver**

**Chương 3: Thiết kế giao thức truyền dữ liệu**

Chương 1: Tổng quan hệ thống phần mềm mô phỏng CAN bus

* 1. Giới thiệu chương

Phần mềm mô phỏng CAN bus là một công cụ hữu ích cho quá trình nghiên cứu và thử nghiệm các bộ phận trong mạng CAN. Hệ thống phần mềm mô phỏng CAN bus bao gồm 3 phần: phần mềm CAN, PC Driver và thiết bị. Chương 1 đi vào giới thiệu tổng quan về toàn bộ hệ thống, giúp chúng ta có cái nhìn khái quát về toàn bộ dự án mà nhóm đang thực hiện.

* 1. Mô hình giao tiếp của hệ thống

Phần mềm CAN

PC Driver

Thiết bị

USB

CAN

Hình 1.1. Mô hình giao tiếp hệ thống

Để có thể kiểm tra được ECU có hoạt động đúng yêu cầu thiết kế hay không, ta sẽ xây dựng mô hình kiểm tra như trên , trong đó ý tưởng chính dự trên việc giả lập một hệ thống mạng CAN kết nối giữa các ECU thực tế để kiểm tra bằng cách truyền các thông điệp yêu cầu ECU thực hiện và nhận lại các thông điệp phản hồi từ ECU, sau đó xử lý nội dung của phản hồi và hiển thị lên màn hình thống kê một cách trực quan với độ chính xác cao.

ECU cần kiểm tra sẽ được kết nối với thiết bị thông qua cổng CAN. Thiết bị kết nối với phần mềm CAN, nơi chứa hệ thống giả lập gồm các ECU (Electronic Control Unit) mô phỏng hệ thống thực tế thông qua cổng USB nhờ PC driver .PC driver đóng vai trò cầu nối giao tiếp giữa hardward và software thông qua cơ chế truyền (ghi) và nhận (đọc) tin từ các file descriptor.

Dưới đây là sơ đồ khối mô hình kết nối khi sử dụng , sơ đồ khối mô tả chung của đề tài.

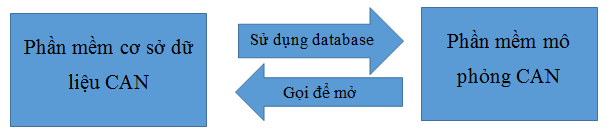


Hình 1.2. Sơ đồ khối hệ thống CAN

Sau khi kết nối giữa ECU và phần mềm CAN , mô hình hệ thống sẽ gồm nhiều ECU được kết nối với nhau thông qua CAN BUS như là một hệ thống CAN thực tế hoàn chỉnh.

* 1. Phần mềm CAN

Phần mềm CAN bao gồm : Phần mềm cơ sở dữ liệu CAN và phần mềm mô phỏng CAN



Hình 1.3. Mô hình phần mềm CAN

* + 1. Phần mềm cơ sở dữ liệu CAN

Phần mềm cơ sở dữ liệu CAN là phần mềm độc lập quản lí cơ sở dữ liệu của hệ thống, lưu giữ các thông điệp (message) và các tín hiệu (signal), hỗ trợ các loại format Little Endian và Big Endian , MSB và LSB.

Các chức năng chính của phần mềm cơ sở dữ liệu CAN là: Tạo mới database và mở 1 database có sẵn.

Phần mềm cơ sở dữ liệu CAN hỗ trợ các chức năng:

* Thêm/sửa/xóa các thông điệp.
* Thêm/ sửa/xóa các tín hiệu trong các thông điệp.
* Thêm/sửa/xóa các nốt mạng (node network) và các biến môi trường EV (environment variable).

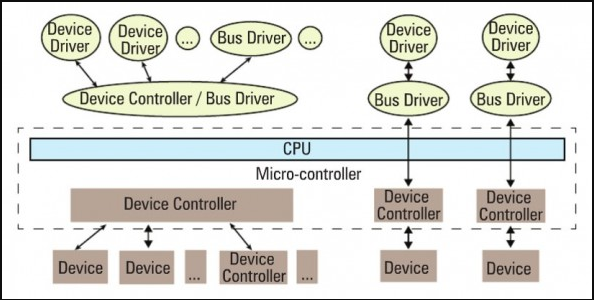
Vai trò chính của phần mềm cơ sở dữ liệu CAN là tạo dựng một cơ sở dữ liệu gồm các node network và các thông điệp gửi đi cũng như nhận về từ chính node đó để cung cấp dữ liệu cho việc mô phỏng ở simulator software.

* + 1. Phần mềm mô phỏng CAN

Phần mềm mô phỏng CAN là một phần mềm độc lập, có nhiệm vụ xử lí việc truyền nhận dữ liệu cũng như phân tích và hiển thị nội dung của các thông điệp, mà định dạng của các thông điệp đó được lấy từ phần mềm cơ sở dữ liệu CAN. Bao gồm các chức năng

* Cấu hình hệ thống, cổng CAN nào đang hoạt động , tốc độ Baud…
* Mở được database vào test model.
* Thêm/xóa được sơ đồ khối kết nối đồng thời cập nhật cấu hình và lưu lại.
* Thêm/xóa được các message vào trong IG. (interactive generator)
* Trong IG có thể cấu hình cycle của từng message.
* Trace window hiển thị các thuộc tính của message: ID, length, direction, data, signal value.
  1. CAN Firmware
     1. Linux USB driver

Driver là một mô-đun trong hệ điều hành có vai trò điều khiển, quản lý, giám sát một thực thể nào đó dưới quyền của nó. Bus driver làm việc với một đường bus, device driver làm việc với một thiết bị (chuột, bàn phím, màn hình, đĩa cứng, camera, …). Có thể lấy ví dụ một thành phần phần cứng có thể được điều khiển bởi một driver hoặc được điều khiển bởi một phần cứng khác mà được quản lý bởi một driver. Trường hợp này, phần cứng có vai trò điều khiển được gọi là một device controller. Bản thân các controller cũng cần driver. Ví dụ: hard disk controller, display controller, audio controller, … quản lý các thiết bị kết nối với chúng, mà nói một cách kỹ thuật hơn đó là các IDE controller, PCI controller, USB controller, SPI controller, I2C controller, … Các khái niệm này được minh họa tổng quan như hình sau:



Hình 1.4. Tương tác giữa thiết bị và Driver

Các device controller thông thường được kết nối với CPU thông qua đường bus (PCI, IDE, USB, SPI, …). Trong vi điều khiển, CPU và các device controller thường được thiết kế trên một chip.

PC Driver có nhiệm vụ giao tiếp giữa phần cứng và phần mềm. Khi một thiết bị được kết nối với PC, driver nhận nhiệm vụ tạo ra vùng nhớ đệm thông qua các file descriptor. Khi ta muốn truyền một thông điệp từ PC xuống phần cứng thì driver sẽ nhận biết được thông điệp, sau đó ghi vào file descriptor tương ứng. Phần cứng sẽ đọc dữ liệu từ file descriptor và thực hiện nội dung thông điệp. Thông điệp phản hồi sẽ được truyền theo chiều ngược lại đến phần mềm.

Driver giúp nhận biết được thiết bị connect, disconnect thông qua cổng USB, đưa ra thông tin, trạng thái của thiết bị.

* Phần mềm CAN có thể đóng/ mở thiết bị thông qua device node.
* Phần mềm CAN có thể cấu hình baurd rate thông qua file descriptor.
* Phần mềm CAN có thể gửi/nhận message thông qua file descriptor
  + 1. Giao thức truyền dữ liệu (Communication Protocol)

Giao thức truyền dữ liệu giữa cổng software và thiết bị TM4C123G, vì thiết bị đóng vai trò như một node CAN trên mạng CAN bus, phần mềm trên PC đóng vai trò hiển thị message để kiểm tra, nên cần thiết kế một chuẩn giao thức gồm nhiều trường hợp khác nhau để hình thành quy luật truyền nhận dữ liệu từ PC xuống thiết bị và ngược lại, theo các chức năng mà người giám sát mạng CAN cần đến, ví dụ: lấy cấu hình mạng CAN từ thiết bị thông qua cổng USB, hay thực hiện tác vụ mô phỏng hoạt động mạng CAN (simulation),…

* 1. CAN Device
  2. Kết luận chương

Qua chương 1, chúng ta đã có được cái nhìn tổng quát về toàn bộ dự án, giúp biết được từng phần trong dự án và chức năng của từng phần. Qua đó chúng ta có thể hiểu được chức năng và sự cần thiết của dự án trong quá trình nghiên cứu và phát triển các thiết bị trong mạng CAN. Chương tiếp theo sẽ đi vào phần cơ sở lý thuyết của đồ án, cung cấp những kiến thức cần có trong quá trình thực hiện.

Chương 2: Cơ sở lý thuyết

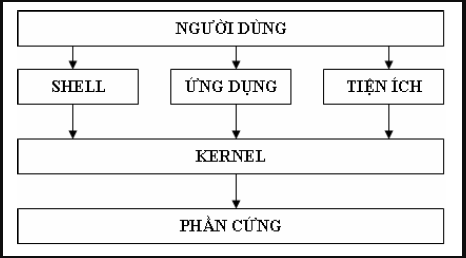
* 1. Giới thiệu chương

Linux là hệ điều hành có thể quản lý bộ nhớ rất tốt, cho nên dự án chọn Linux là môi trường cho phần mềm hoạt động.Lập trình Driver sử dụng các câu lệnh trong Kernel Linux, Kernel chỉ tải các phần cần thiết lên bộ nhớ, các phần khác được tải lên nếu có yêu cầu sử dụng. Với mục đích cung cấp các kiến thức nền tảng để thực hiện đồ án, chương 2 sẽ trình bày sơ lược về kiến trúc hệ điều hành Linux, nhiệm vụ, các lớp thiết bị cơ bản trong Linux Device Driver và các dạng sử dụng bộ nhớ đệm trong quá trình thiết kế giao thức truyền dữ liệu.

* 1. Môi trường hệ điều hành Linux

Kiến trúc của hệ điều hành Linux được chia làm 4 hàng bao gồm:

Người dùng (User) -> Shell, Ứng dụng, Tiện ích -> Nhân Kernel -> Phần cứng.



Hình 2.1. Kiến trúc hệ điều hành Linux.

* Hạt nhân (Kernel)

Là trung tâm điều khiển của hệ điều hành Linux , chứa các mã nguồn điều khiển hoạt động của toàn bộ hệ thống. Kernel được thiết kế theo module, do vậy kích thước rất nhỏ. Kernel chỉ tải bộ phận cần thiết lên bộ nhớ , các bộ phận khác được tải lên nếu có yêu cầu sử dụng. Nhờ vậy , so với các hệ điều hành khác Linux không sử dụng lãng phí bộ nhớ.

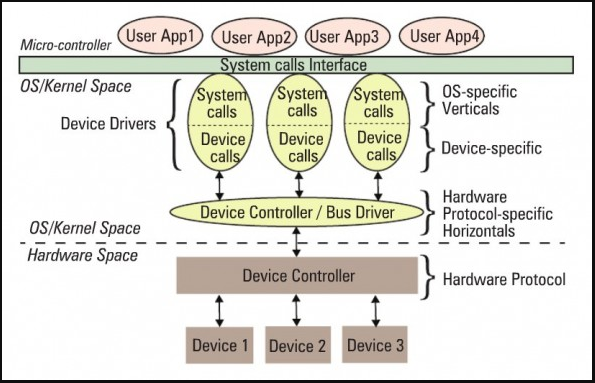
Kernel của Linux có thể truy xuất tới toàn bộ tính năng phần cứng của máy. Yêu cầu của các chương trình cần rất nhiều bộ nhớ , trong khi hệ thống có ít bộ nhớ , hệ điều hành sử dụng không gian đĩa hoán đổi (swap space) để lưu trữ các dữ liệu xử lí của chương trình. Swap space cho phép ghi các trang của bộ nhớ xuất các vị trí dành sẵn trong đĩa và xem nó như phần mở rộng của vùng nhớ chính . Bên cạnh sử dụng swap space, Linux hỗ trợ đặc tính sau :

* Bảo vệ vùng nhớ giữa các tiến trình, điều này không cho phép một tiến trình làm tắt toàn bộ hệ thống.
* Chỉ tải các chương trình khi có yêu cầu.
* Shell

Shell cung cấp tập lệnh cho người dùng thao tác với kernel để thực hiện công việc. Shell đọc các lệnh từ người dùng và xử lí. Ngoài ra shell còn cung cấp một số đặc tính khác như : Chuyển hướng xuất nhập, ngôn ngữ lệnh để tạo các tập tin lệnh tương tự tập tin bat trong DOS.

* 1. Lý thuyết LDD
     1. Nhiệm vụ của LDD

Các bus driver cung cấp giao diện đặc tả cho các giao thức phần cứng tương ứng. Nó nằm ở tầng dưới cùng trong mô hình phân lớp phần mềm của hệ điều hành. Nằm trên nó là các device driver thực sự để vận hành các thiết bị, mang đặc trưng của từng thiết bị xác định. Ngoài ra, mục đích quan trọng của các driver thiết bị là cung cấp một giao diện trừu tường hóa cho người sử dụng, tức là cung cấp một giao diện lên tầng trên của hệ điều hành. Một cách tổng quan, một driver sẽ bao gồm 2 phần quan trọng: giao tiếp với thiết bị (Device-specific) và giao tiếp với hệ điều hành (OS-specific).



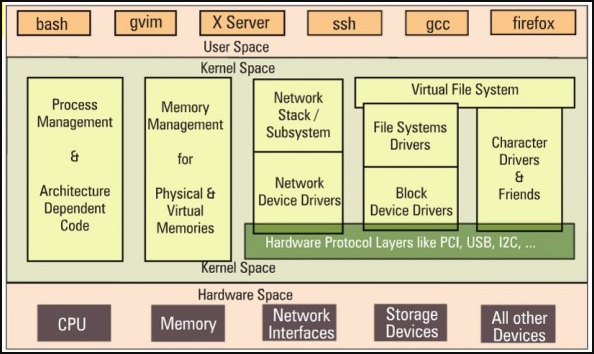
Hình 2.2. Các thành phần của Driver trên Linux

Thành phần giao tiếp với thiết bị (device-specific) của một driver là giống nhau đối với tất cả các hệ điều hành. Nó có thể hiểu và giải mã các thông tin về thiết bị (chi tiết kỹ thuật, kiểu thao tác, hiệu năng, cách lập trình giao tiếp với thiết bị, …)

Thành phần giao tiếp với hệ điều hành (OS-specific) gắn kết chặt chẽ với các cơ chế của hệ điều hành, và do vậy sẽ là khác nhau giữa một driver trên Linux và một driver trên Windows, hoặc MacOS, …

* + 1. Mô hình phân lớp theo chiều học

Trên Linux, device driver cung cấp một giao diện “system call” (giao diện gọi các hàm hệ thống) đến tầng ứng dụng cho người dùng; đây được coi là một ranh giới giữa tầng nhân (kernel space) và tầng người dùng (user space) của Linux. Mô hình phân tầng được chỉ ra như hình vẽ.



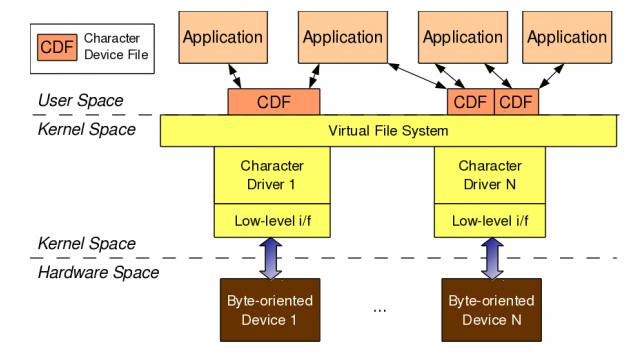
Hình 2.3 Kiến trúc tổng quan Linux Kernel

Tùy thuộc vào đặc trưng của của driver với hệ điều hành, driver trên Linux được phân chia thành 3 loại (phân cấp theo chiều dọc):

* Packet-oriented or the network vertical (driver hướng gói dữ liệu)
* Block-oriented or the storage vertical (driver hướng khối dữ liệu)
* Byte-oriented or the character vertical (driver hướng byte/ký tự)
  + 1. Các lớp Device và mô đun

Có ba loại Device cơ bản, mỗi Module thực hiện một trong các loại này. Các Module được phân loại thành char module, block module, network module.

* Character devices: là loại device có thể được truy cập như một dòng byte( một tập tin), char driver chịu trách nhiện cho quá trình này. Một driver như vậy thường thực hiện ít nhất các lệnh gọi hệ thống như mở, đóng, đọc, ghi.
* Block devices: là loại device có thể lưu trữ một hệ thống tập tin. Trong hầu hết các hệ thống Unix, một Block device chỉ có thể xử lý các hoạt động I/O chuyển một hoặc nhiều khối tin, thường là 512 byte chiều dài
* Network devices: bất kỳ truyền nhận thông qua mạng cũng được thực hiện thông qua một tương tác mạng, tức là một thiết bị có thể trao đổi dữ liệu với các máy chủ khác. Một tương tác mạng chịu trách nhiệm gửi và nhận các gói dữ liệu, được điều khiển bởi hệ thống mạng con trong hạt nhân. Một network driver không biết gì về các kết nối cá nhân, nó chỉ xử lý gói tin.
  + 1. Character Driver
       1. Giới thiệu Character Driver



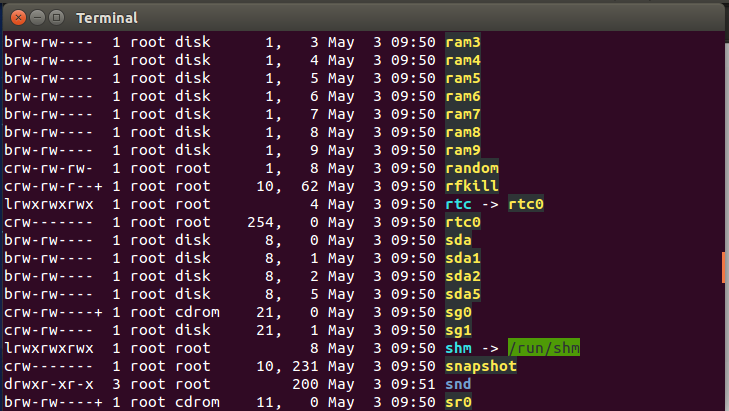
Hình 2.4. Tổng quan về character driver trên Linux

Bất kỳ một ứng dụng nào ở tầng người dùng (user space) muốn thao tác với một thiết bị kiểu character device trong tầng phần cứng (hardware space) sẽ sử dụng character device driver tương ứng trong tầng nhân (kernel space). Việc sử dụng các character driver được thực hiện thông qua các file thiết bị (device files) tương ứng, được liên kết với driver thông qua hệ thống file ảo (virtual file system – VFS). Điều này có nghĩa là các ứng dụng có thể thực hiện các thao tác file thông thường trên các file thiết bị. Các thao tác file này sẽ được VFS diễn giải ra các hàm tương ứng trong driver liên kết với nó. Các hàm này sau đó sẽ thực hiện các truy cập ở mức thấp đến các thiết bị thật sự để đạt được kết quả mong muốn.

* + - 1. Số hiệu file thiết bị

Char device được truy cập thông qua các tên trong hệ thống tập tin. Những tên này được gọi là các tệp đặc biệt hoặc tệp thiết bị hoặc chỉ đơn giản là các nút của cây hệ thống tập tin, chúng thường nằm trong thư mục /dev. Các tập tin đặc biệt của Char device được xác định bởi chữ “c” trong cột đầu tiên của đầu ra của ls –l. Block device cũng xuất hiện trong /dev, nhưng chúng được xác định bởi “b”.

Khi dùng lệnh ls –l trong đường dẫn /dev. Danh sách sau hiển thị một vài thiết bị khi chúng xuất hiện trên một hệ thống điển hình. Số lớn là 1, 10, 254, 8, 21, … Số nhỏ là 3, 4, 5, 6, 7, 8, 9, …



Hình 2.5. Danh sách thiết bị trong thư mục /dev

Số lớn xác định trình điều khiển được liên kết với thiết bị. Ví dụ /dev/ram1 và /dev/ram2 đều được quản lý bởi trình điều khiển 1, trong khi /dev/rtc0 và /dev/sg0 được quản lý bởi trình điều khiển 0. Hạt nhân Linux cho phép nhiều trình điều khiển chia sẻ các số lớn với nhau, nhưng hầu hết các thiết bị vẫn được sắp xếp theo nguyên tắc một số lớn – một trình điều khiển.

Số nhỏ được hạt nhân sử dụng để xác định chính xác thiết bị đang được giới thiệu đến. Có thể lấy một con trỏ trực tiếp tới thiết bị từ hạt nhân hoặc có thểsử dụng con trỏ như một chỉ mục vào một mảng thiết bị có sẵn. Dù bằng cách nào thì hạt nhân hầu như không biết về số nhỏ.

* + - 1. Phương thức mở và đóng
* Phương thức mở

Phương pháp mở được cung cấp cho một trình điều khiển để thực hiện bất kỳ khởi tạo để chuẩn bị cho các hoạt động sau này. Trong hầu hết các trình điều khiển, phương pháp mở thực hiện các tác vụ sau:

* Kiểm tra lỗi cụ thể của thiết bị ( ví dụ như các sự cố về thiết bị không sẵn sàng)
* Khởi tạo các thiết bị nếu nó đang được mở ra lần đầu tiên
* Phương thức đóng

Vai trò của phương pháp đóng là ngược lại với phương pháp mở

* + - 1. Phương thức đọc và ghi
* Phương thức đọc

Trường hợp tối ưu nhất là khi giá trị bằng biến đếm được truyền vào hệ thống thì số số lượng byte dữ liệu yêu cầu đã được chuyển thành công.

Nếu giá trị là dương, nhưng nhỏ hơn đếm, chỉ một phần dữ liệu đã được chuyển.

Nếu giá trị là 0 thì kết thúc tập tin ( và không có dữ liệu nào được đọc)

Giá trị âm có nghĩa là đã có lỗi. Các giá trị tiêu biểu trả về lỗi bao gồm –EINTR (lệnh hệ thống bị gián đoạn) hoặc –EFAULT ( địa chỉ sai).

* Phương thức ghi

Nếu giá trị bằng biến đếm, số byte yêu cầu đã được chuyển

Nếu giá trị dương nhưng nhỏ hơn biến đếm, thì chỉ có một phần dữ liệu được chuyển đi. Chương trình rất có thể sẽ thử ghi phần còn lại của dữ liệu.

Nếu giá trị là 0 thì không có dữ liệu được ghi. Kết quả này không phải là lỗi và không có lý do để trả về mã lỗi.

Giá trị âm có nghĩa là một lỗi đã xảy ra. Các giá trị lỗi được định nghĩa trong <linux/errno.h>

* + 1. TTY Driver
       1. Hàm mở

Hàm mở được gọi bởi TTY Core khi người dùng gọi open trên device node mà tty driver hỗ trợ với một con trỏ đển tty\_struct. Khi hàm mở được gọi, tty driver sẽ lưu dữ liệu trong biến tty\_struct hoặc lưu dữ liệu trong một mảng tĩnh có thể tham chiếu đế số nhỏ (minor) của cổng.

Hàm mở trả lại số âm nếu không thể mở thành công, và trả về 0 để cho biết thành công

* + - 1. Hàm đóng

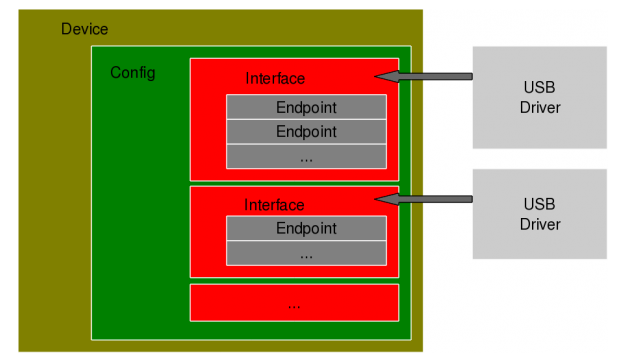
Hàm đóng được gọi bởi lõi TTY khi lệnh đóng được gọi bởi người dùng trên các tập tin xử lý đã được tạo ra trước đó với hàm mở.

* + - 1. Hàm ghi

Hàm ghi được gọi bởi người dùng khi có dữ liệu muốn gửi đến phần cứng. Đầu tiên lõi tty nhận lệnh, và sau đó sẽ chuyển dữ liệu bằng hàm ghi của tty driver, lõi tty thông báo cho tty driver độ lớn của dữ liệu đang được chuyển.

Trong trường hợp không phải tất cả dữ liệu đều được chuyển đi, vì tốc độ và dung lượng bộ nhớ đệm của thiết bị tty, hàm ghi sẽ trả về số lượng dữ liệu đã chuyển đến phần cứng, vì thế có thể kiểm tra được số dữ liệu đã được chuyển đi.

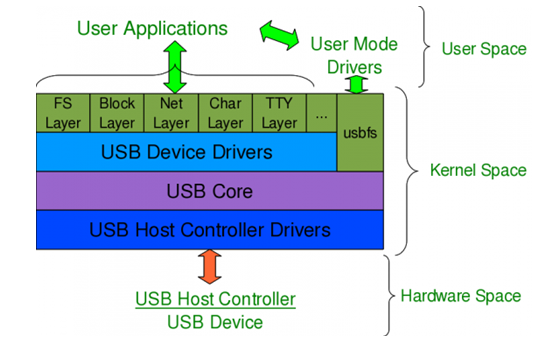
* + 1. USB Driver
       1. Quá trình nhận dạng thiết bị USB trên Linux



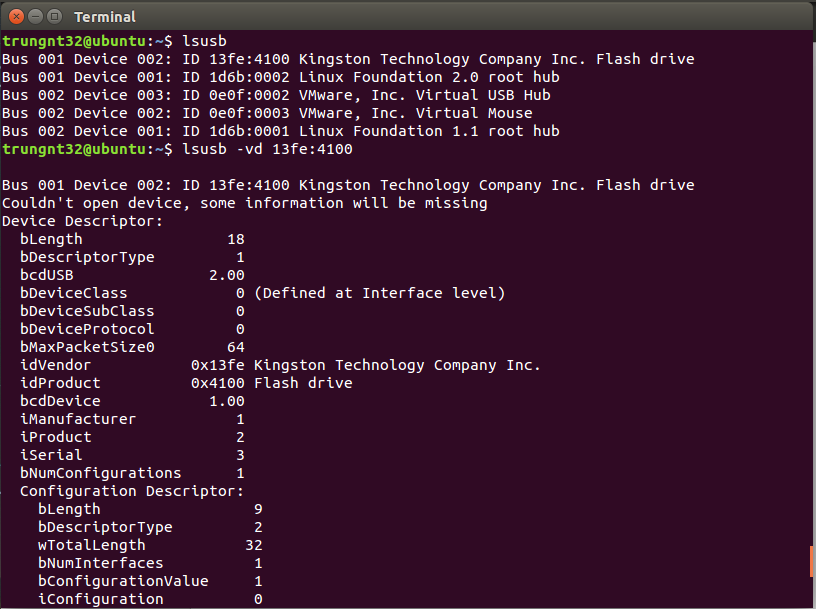
Hình 2.6. Mô hình biểu diễn tổng quan thiết bị USB

Khi có một thiết bị usb hợp lệ được cắm vào hệ thống Linux, cho dù nó có driver hay không thì nó cũng vẫn được nhận diện (detect) bởi phần cứng ở tầng nhân (kernel space) của hệ thống Linux mà đã được hỗ trợ giao thức usb. Hệ thống có thể làm điều này là bởi vì khả năng của chính bản thân giao thức usb đã được thiết kế trong đặc tả của nó. Cụ thể, việc phát hiện ra thiết bị usb cắm vào được thực hiện bởi chip usb host controller (là thiết bị chủ động đường bus của giao thức usb). USB host controller này sẽ thu thập và diễn giải các thông tin ở tầng vật lý (low-level) đến các thông tin đặc tả giao thức USB ở tầng trên (high-level). Các thông tin về thiết bị theo khuôn dạng qui định của giao thức USB lại tiếp tục được đưa vào tầng usb core tổng quát (generic usb core) trong tầng nhân (được điều khiển bởi usbcore driver). Chính điều này giúp cho các thiết bị usb được hệ thống nhận diện ở tầng nhân, mặc dù nó có thể chưa có một driver cụ thể nào cho chức năng của nó.

Sau quá trình diễn ra ở tầng nhân này, sẽ đến nhiệm vụ của các drivers hoặc interfaces hoặc applications (cái mà phụ thuộc vào các bản Linux khác nhau) để tiếp tục nhận dạng ra thiết bị ở tầng người dùng (user space). Hình dưới minh họa cho kiến trúc phân tầng từ trên xuống của hệ thống USB trên Linux

Hình 2.7. Hệ thống USB trên Linux

Dùng lệnh lsusb để liệt kê danh sách thông tin cơ bản của tất cả các thiết bị usb được hệ thống nhận diện. Để xem thông tin chi tiết về 1 thiết bị nào đó, cần sử dụng thêm tham số -vd (<vendor ID>:<product ID>). Minh họa như hình dưới.



Hình 2.8. Xem thông tin các thiết bị USB sử dụng lệnh lsusb

* + - 1. Giải mã thông tin về thiết bị USB

Tất cả các thiết bị USB hợp lệ đều chứa thông tin về một hoặc một vài cấu hình (configurations). Một cấu hình của thiết bị usb giống như một bản hồ sơ về thiết bị đó. Linux chỉ hỗ trợ một cấu hình cho mỗi một thiết bị, do vậy nếu thiết bị có nhiều cấu hình nó sẽ chỉ được hệ thống sử dụng một cái mặc định là cái được sử dụng phổ biến nhất. Với mỗi cấu hình, thiết bị lại có một hoặc một vài giao diện (interfaces). Mỗi một giao diện tương ứng với một chức năng (function) của thiết bị đó. Tức là, số giao diện là bằng với số chức năng của thiết bị usb đó. Vì vậy, không giống như các driver thiết bị đơn chức năng khác, driver cho thiết bị usb thông thường sẽ gắn với một giao diện (chức năng) nào đó của thiết bị usb đấy chứ không thường cho toàn bộ chức năng của thiết bị. Điều đó cũng có nghĩa là thiết bị usb đó có thể có nhiều driver và nhiều giao diện khác nhau có thể cùng chung một driver, nhưng ngược lại mỗi giao diện chỉ có tối đa 1 driver. Tuy nhiên, cũng hoàn toàn là phù hợp nếu thiết bị sử dụng một driver cho tất cả các giao diện của nó.

Mỗi giao diện (interface) lại có một hoặc một vài end-points. Mỗi end-point là một bộ đệm dữ liệu trên thiết bị gắn kết với một đường ống (pipe) logic dùng để truyền thông tin từ hoặc đến thiết bị (giống như “hút” hoặc “bơm” dữ liệu giữa thiết bị và máy host), việc này phụ thuộc vào các chức năng của thiết bị đó. Dựa trên các kiểu thông tin cần trao đổi, có 4 loại end-points (tương ứng với 4 kiểu truyền):

* Control (Kiểu điều khiển)
* Interrrupt (Kiểu ngắt)
* Bulk (Kiểu khối dữ liệu đảm bảo tính chính xác khi truyền)
* Isochronous (Kiểu khối dữ liệu đảm bảo về thời gian truyền)

Theo đặc tả của giao thức USB, tất cả các thiết bị USB hợp lệ đều ngầm định sử dụng end-point 0 để truyền điều khiển (control), đây là end-point duy nhất có 2 chiều (bi-directional).

* + - 1. Điểm cuối USB (USB endpoint)

Phụ thuộc vào các kiểu và thuộc tính của dữ liệu cần trao đổi mà một thiết bị USB có thể có 1 hoặc nhiều endpoints, mỗi endpoint thiết bị có thuộc 1 trong 4 loại sau:

* Control: Là loại endpoint dùng để truyền các thông tin điều khiển. Ví dụ: reset thiết bị, truy vấn thông tin về thiết bị, v.v… Tất cả các thiết bị USB đều có endpoint điều khiển mặc định là endpoint 0.
* Interrupt: Là loại endpoint dùng để truyền một lượng thông tin nhỏ và nhanh, thông thường tối đa là 8 bytes. Ví dụ: truyền dữ liệu cho cổng nối tiếp, truyền dữ liệu với các thiết bị giao tiếp con người (HID – Human Interface Device) như chuột, bàn phím, joystick, v.v…
* Bulk: Là loại endpoint dùng để truyền một lượng lớn dữ liệu nhưng tốc độ chậm hơn, thường cần đảm bảo tính toàn vẹn dữ liệu. Ví dụ: truyền dữ liệu với thiết bị nhớ (mass storage)
* Isochronous: Là loại endpoint dùng để truyền một lượng lớn dữ liệu với băng thông lớn mà không cần đảm bảo tính toàn vẹn dữ liệu. Ví dụ: truyền dữ liệu với các thiết bị ràng buộc về thời gian như audio, video, v.v…

Ngoài ra, tất cả các endpoint cần được xác định hướng “in” hoặc “out” để xác định hướng truyền dữ liệu. Trong đó, “in” là chiều dữ liệu từ thiết bị usb lên máy host và “out” là chiều truyền dữ liệu từ máy host đến thiết bị usb.

Về mặt kỹ thuật, một endpoint được định danh sử dụng số nguyên 8 bit, trong đó bit cao nhất (MSB) sẽ chỉ ra hướng truyền dữ liệu (0 là “out” và 1 là “in”). Riêng endpoint điều khiển (endpoint 0) là có 2 chiều (bi-directional) nên bit MSB được bỏ qua.

* + - 1. Giao diện USB (USB interface)

Nhiều USB endpoint được gói vào các giao diện. Giao diện USB chỉ xử lý một loại kết nối USB , một số thiết bị USB có nhiều giao diện thì cần nhiều loại driver khác nhau cho một phần cứng. Giao diện USB được mô tả trong hạt nhân với cấu trúc usb\_interface struct.

* + - 1. Cấu hình USB (USB configuration)

Nhiều giao diện USB được gói thành các cấu hình. Thiết bị USB có thể có nhiều cấu hình và có thể chuyển đổi giữa các thiết bị để thay đổi trạng thái của thiết bị. Một cấu hình đơn có thể được kích hoạt chỉ tại một thời điểm. Linux mô tả cấu hình USB với cấu trúc usb\_host\_config và mô tả toàn bộ các thiết bị USB với cấu trúc usb\_device.

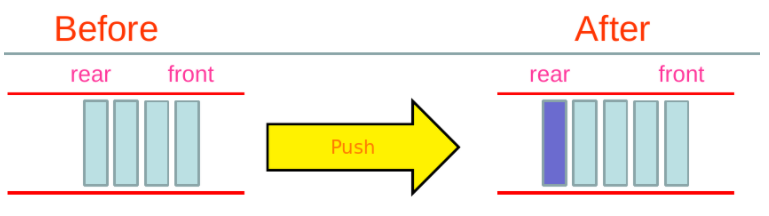
* 1. Lý thuyết giao thức truyền dữ liệu
     1. Hàng đợi (queue)

Hàng đợi (Queue) là một cấu trúc dữ liệu dùng để chứa các đối tượng làm việc theo cơ chế FIFO (First In First Out), nghĩa là “vào trước ra trước”. Trong hàng đợi, các đối tượng có thể được thêm vào hàng đợi bất kỳ lúc nào, nhưng chỉ có đối tượng thêm vào đầu tiên mới được phép lấy ra khỏi hàng đợi. Việc thêm một đối tượng luôn diễn ra ở cuối hàng đợi và một phần tử luôn được lấy ra từ đầu hàng đợi.



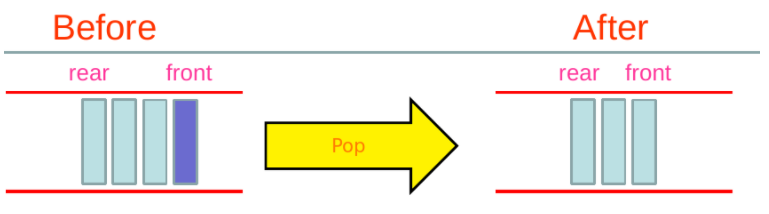
Hình 2.9. Tổng quan về hàng đợi

* Thêm phần tử vào cuối Queue trên mảng: Tăng vị trí của Rear lên 1 và đưa data vào vị trí đó.



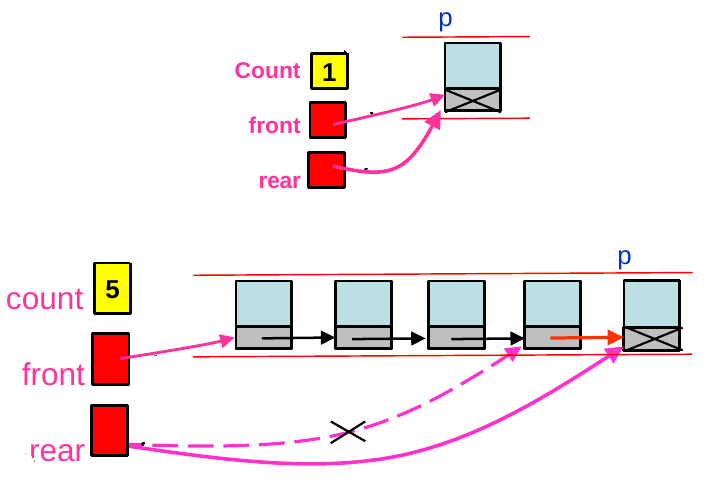
Hình 2.10. Tiến trình thêm phần tử vào Queue

* Xóa phần tử đầu Queue trên mảng: kiểm tra Queue rỗng không, nếu không rỗng ta thực hiện di chuyển các phần tử trong hàng về đầu hàng bằng vòng for (giống như xếp hàng khi mua hàng) sau đó giảm Rear và count.



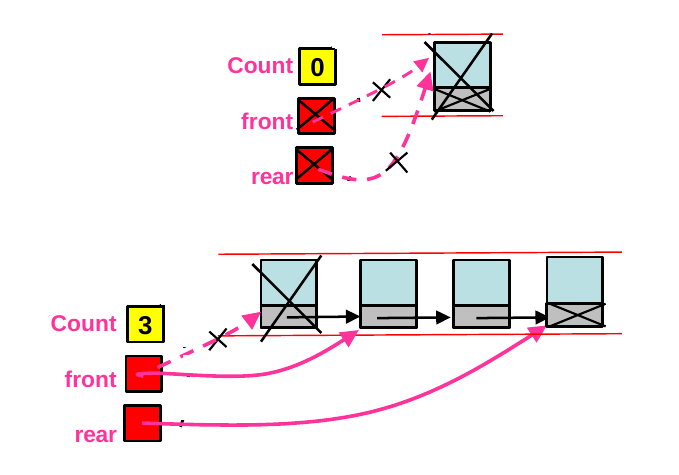
Hình 2.11. Tiến trình xóa phần tử đầu Queue

* Thêm phần tử vào cuối Queue bằng con trỏ: Để thêm phần tử, ta kiểm tra xem hàng có rỗng không, nếu hàng rỗng thì cho cả Front và Rear cùng trỏ về Node P mới tạo chứa phàn tử x cần thêm. Nếu không rỗng ta trỏ Rear->Next về P và Rear trỏ về P. Tăng count lên 1.



Hình 2.12. Tiến trình thêm phần tử vào cuối Queue

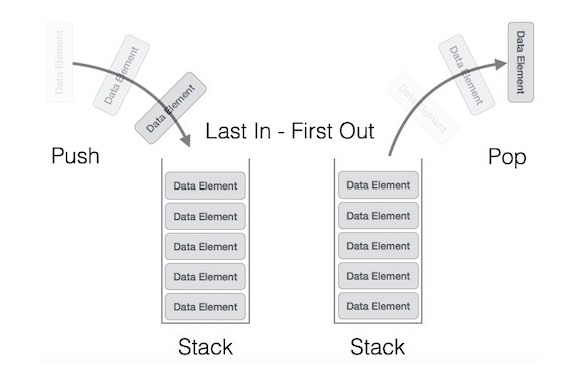
* Xóa phần tử đầu Queue bằng con trỏ: Ta kiểm tra Queue có rỗng không, Nếu không rỗng kiểm tra xem có 1 hay nhiêu hơn 1 phần tử, nếu có 1 phần tử thì ta khởi tạo lại Queue, nếu có nhiều hơn ta cho Front trỏ đến tiếp theo. Giảm count xuống 1.



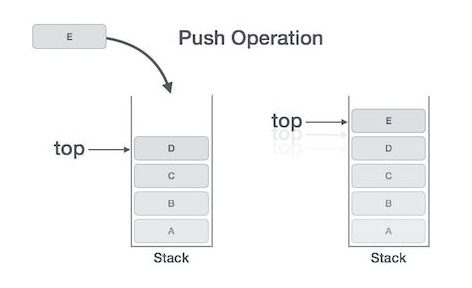
Hình 2.13. Tiến trình xóa phần tử đầu Queue

* + 1. Ngăn xếp (stack)

Một ngăn xếp (Stack) là một cấu trúc dữ liệu trừu tượng (Abstract Data Type – viết tắt là ADT), hầu như được sử dụng trong hầu hết mọi ngôn ngữ lập trình. Ngăn xếp trở thành cấu trúc dữ liệu dạng LIFO (Last-In-First-Out). Ở đây, phần tử được đặt vào (được chèn, được thêm vào) cuối cùng sẽ được truy cập đầu tiên. Trong thuật ngữ ngăn xếp, hoạt động chèn được gọi là hoạt động PUSH và hoạt động xóa được gọi là hoạt động POP.

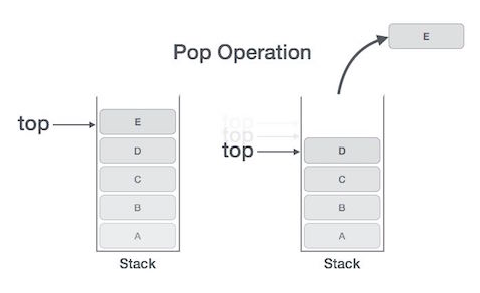
 Hình 2.14. Sơ đồ minh họa một ngăn xếp và các hoạt động diễn ra

* Hoạt động PUSH
* Bước 1: kiểm tra xem ngăn xếp đã đầy hay chưa.
* Bước 2: nếu ngăn xếp là đầy, tiến trình bị lỗi và thoát ra.
* Bước 3: nếu ngăn xếp chưa đầy, tăng top để trỏ tới phần bộ nhớ trống tiếp theo.
* Bước 4: thêm phần tử dữ liệu vào vị trí nơi mà top đang trỏ đến trên ngăn xếp.
* Bước 5: trả về success.



Hình 2.15. Tiến trình thêm phần tử vào Stack

* Hoạt động POP
* Bước 1: kiểm tra xem ngăn xếp là trống hay không.
* Bước 2: nếu ngăn xếp là trống, tiến trình bị lỗi và thoát ra.
* Bước 3: nếu ngăn xếp là không trống, truy cập phần tử dữ liệu tại top đang trỏ tới.
* Bước 4: giảm giá trị của top đi 1.
* Bước 5: trả về success.

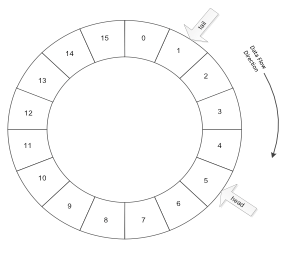


Hình 2.16. Tiến trình lấy phần tử từ Stack

* + 1. Bộ đệm vòng (ring buffer)

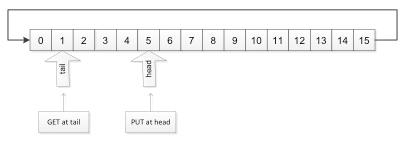
Bộ đệm vòng hữu ích trong việc truyền dữ liệu giữa các quá trình không đồng bộ. Bộ đệm vòng là một hàng đợi vòng (quece), có đặc tính dữ liệu FIFO đầu tiên.

Bộ đệm vòng có hai chỉ số cho phần tử trong bộ đệm. Khoảng cách giữa các chỉ số có thể dao động từ 0 đến tổng số các phần tử trong bộ đệm. Cấu trúc vòng của bộ đệm vòng được thể hiện như sau:



Hình 2.17. Cấu trúc của bộ đệm vòng

Dữ liệu sẽ được đặt ở chỉ số đầu và dữ liệu được đọc từ chỉ số cuối.



Hình 2.18. Thực hiện bộ đệm tuyến tính của bộ đệm vòng

Hàng đợi được sử dụng để tuần tự hóa dữ liệu từ quá trình này sang quá trình khác. Bộ đệm sẽ thu tập dữ liệu để sau đó một quá trình khác có thể lấy dữ liệu để xử lý tiếp



Hình 2.19. Tiến trình xử lý trường hợp sử dụng bộ đệm

* + 1. Thuật toán tìm kiếm tuần tự

Tìm kiếm là công việc quan trọng đối với các hệ thống tin học và có liên quan mật thiết với quá trình sắp xếp dữ liệu.

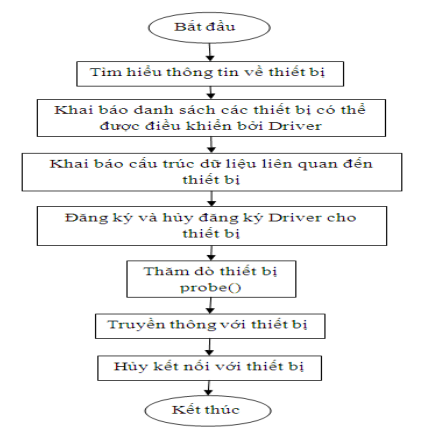
Ý tưởng thuật toán: Xét dãy số cần tìm có n phần tử: a[0], a[1], a[2]... a[n-1]. Giá trị cần tìm là x.

* Bắt đầu từ khoá đầu tiên, lần lượt so sánh khoá x với khoá tương ứng trong dãy.
* Quá trình tìm kiếm kết thúc khi tìm được khoá thoả mãn hoặc đi đến hết dãy hoặc gặp điều kiện dừng vòng lặp.Đây là kĩ thuật tìm kiếm cổ điển nhất trên 1 danh sách chưa được sắp xếp. Nội dung cơ bản của phương pháp này là duyệt từ bản ghi thứ nhất cho tới bản ghi cuối cùng, và so sánh lần lượt với khoá X cần tìm, trong quá trình duyệt nếu có bản ghi trùng với giá trị của X thì chúng ta đưa ra vị trí của bản ghi trong dãy, nếu duyệt tới cuối cùng mà không có bản ghi nào trùng giá trị thì quá trình tìm kiếm không thành công.
  1. Kết luận chương

Những kiến thức lý thuyết được nêu trong chương là cơ sở lý thuyết cho quá trình thực hiện đồ án. Qua đó ta có thể hiểu được cấu trúc của Linux Device Driver, các dạng Device Driver và các phương thức cơ bản để có thể thực hiện nhiệm vụ lập trình Linux USB Driver. Thiết kế giao thức truyền dữ liệu có nhiệm vụ truyền nhận dữ liệu từ PC đến thiết bị nên bộ nhớ đệm là vùng nhớ rất cần thiết trong hệ thống, qua chương 2 chúng ta đã biết được các dạng sắp xếp dữ liệu trong các kiểu bộ nhớ đệm khác nhau, từ đó chúng ta sẽ bắt đầu thực hiện và có những đánh giá trong chương 3

Chương 3: Kết quả thực hiện và đánh giá

* 1. Giới thiệu chương
  2. Bảng phân công công việc
  3. USB Driver
     1. Quy trình viết USB Driver



Hình 3.1. Quy trình viết USB Driver

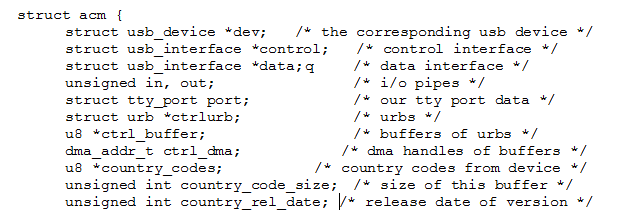
* + - 1. Tìm hiểu về thiết bị USB muốn giao tiếp

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.

Trên hệ điều hành Linux 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 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.

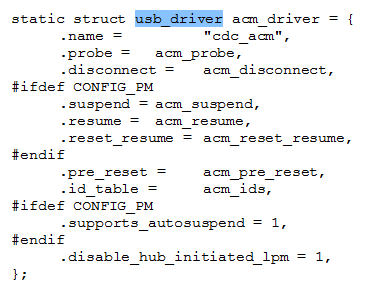
* + - 1. Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver
* Macro USB\_DEVICE (vender, product) : tạo cấu trúc usb\_device\_id với Idvendor và Idproduct
* Khai báo cấu trúc usb\_device\_id với USB Core, sử dụng macro MODULE\_DEVICE\_TABLE(usb, usb\_device\_id[]);
  + - 1. Khai báo cấu trúc dữ liệu liên quan tới thiết bị

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.



* + - 1. Đăng ký và hủy đăng ký USB Device Driver

Để tầng USB Core có thể nhận ra Driver thì cần phải đăng ký Driver. Sử dụng:



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 usb\_deregister(struct usb\_driver &);

* + - 1. 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
  + - 1. 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: 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\* );

* + - 1. Hàm mở/đọc/ghi thiết bị
* Mở tập 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.

* Đọc/ghi

Để đọc ghi dữ liệu từ thiết bị USB ta phải lấy các thông tin từ thiết bị sử dụng hàm usb\_get\_intfdata() và thiết lập dữ liệu cho cấu trúc file nhƣ sau:

dev = usb\_get\_intfdata(interface);

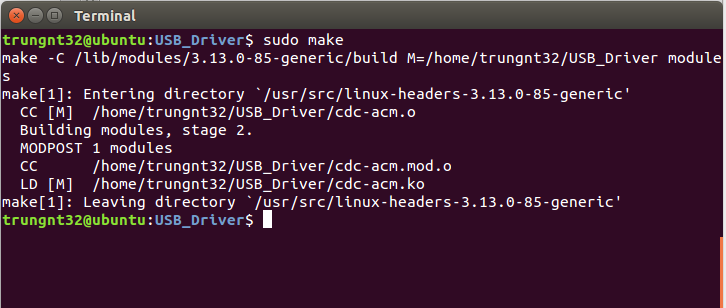
file->private\_data = dev;

Cấu trúc file đƣợc định nghĩa trong <linux/fs.h> là một cấu trúc quan trọng trong không gian nhân của Linux rất cần thiết cho việc viết Driver của USB. Sau đó thực hiện đọc, ghi dữ liệu từ thiết bị sử dụng các hàm read() và write().

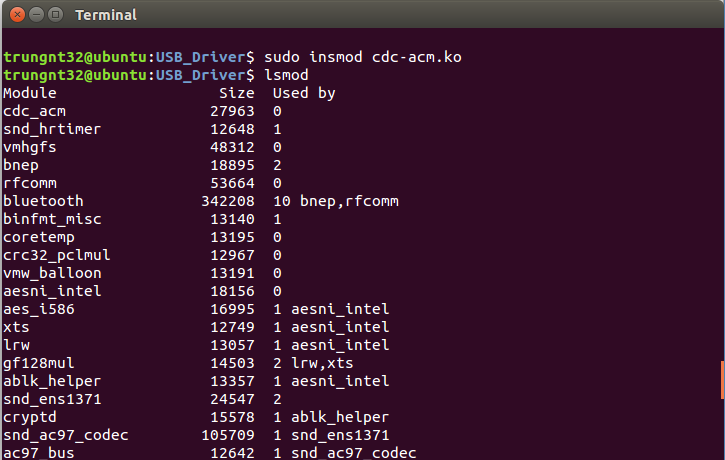
* + 1. Cài đặt USB Driver

Thực hiện biên dịch driver trên (sử dụng Makefile) cho hệ thống Linux sử dụng:

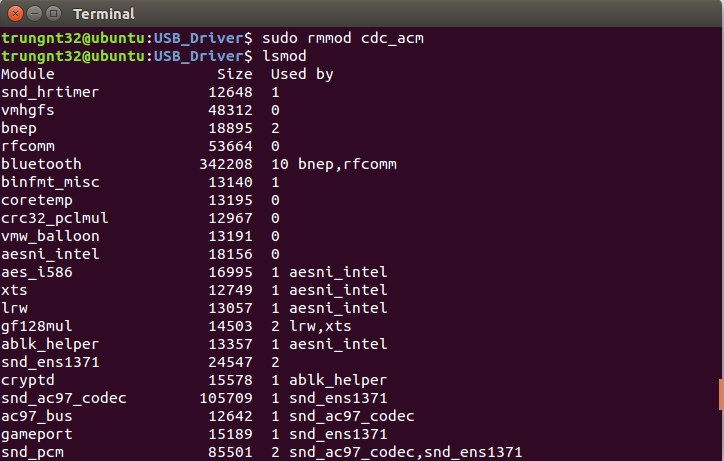
* Biên dịch bằng lệnh make (tạo ra file cdc-acm.ko): sudo make



* Nạp driver sử dụng insmod: sudo insmod cdc-acm.ko và kiểm tra, tra thông tin các mô đun trong hệ thống bằng lệnh lsmod



* Gỡ driver sử dụng lệnh rmmod: sudo rmmod cdc-acm và kiểm tra, tra thông tin các mô đun trong hệ thống bằng lệnh lsmod



* 1. Giao thức truyền dữ liệu
     1. Khung truyền cơ bản
     2. Khung truyền USB
     3. Các struct sử dụng
     4. Sơ đồ tuần tự
     5. Kết quả mô phỏng
  2. Kết luận chương