{Trang trắng này dùng để dán bản Nhận xét của người hướng dẫn, hoặc thay trang  
này bằng Nhận xét của người hướng dẫn}

{Trang trắng này dùng để dán bản Nhận xét của người hướng dẫn, hoặc thay trang  
này bằng Nhận xét của người hướng dẫn}

**TÓM TẮT**

Trong dự án “ Thiết kế và thi công phần mềm mô phỏng CAN bus”, chúng tôi thực hiện nhiệm vụ Lập trình Driver và thiết kế giao thức truyền dữ liệu trong hệ thống mô phỏng CAN bus. Đề tài thực hiện:

* Lập trình Driver dựa trên các hàm được hỗ trợ trong Linux kernel.

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, bao gồm 2 phần quan trọng: giao tiếp với thiết bị (Device-specific) và giao tiếp với hệ điều hành (OS-specific).

* Thiết kế giao thức truyền dữ liệu dựa trên Qt framework

Giao thức truyền dữ liệu giữa cổng software và board TMS123G, vì board đóng vai trò như một node CAN trên mạng CAN bus, phần mềm PC đóng vai trò hiện 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 board và ngược lại.

{ NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP\_mẫu}

# LỜI NÓI ĐẦU

Cộng hoà xã hội chủ nghĩa Việt Nam

Độc lập - Tự do - Hạnh phúc

LỜI CAM ĐOAN

Tôi tên là: Hồ Thăng Huy

Sinh viên lớp 12DT3 khoa Điện Tử - Viễn Thông Đại học Bách Khoa Đà Nẵng.

Tôi tên là: Đặng Trung Nguyên

Sinh viên lớp 12DT2 khoa Điện Tử - Viễn Thông Đại học Bách Khoa Đà Nẵng.

Tôi xin cam đoan đồ án tốt nghiệp với đề tài “ Lập trình Driver và thiết kế giao thức truyền dữ liệu trong hệ thống mô phỏng CAN bus ” của tôi không sao chép từ bất kỳ đồ án hay công trình khoa học đã có trước mà là do tôi tự nghiên cứu dựa trên những tài liệu tham khảo mà tôi đưa ra ở cuối đồ án. Tôi xin chịu hoàn toàn trách nhiệm về nội dung của đồ án và lời cam đoan trên.

Đà Nẵng Ngày 12 Tháng 5 Năm 2017

Người cam đoan:

# MỤC LỤC

LỜI NÓI ĐẦU i

LỜI CAM ĐOAN ii

MỤC LỤC iii

DANH SÁCH CÁC BẢNG, HÌNH VẼ v

DANH SÁCH CÁC KÝ HIỆU, CHỮ VIẾT TẮT vi

CHƯƠNG 1: TỔNG QUAN ĐỀ TÀI 2

1.1. GIỚI THIỆU CHƯƠNG 2

1.2. TỔNG QUAN KIẾN TRÚC HỆ ĐIỀU HÀNH LINUX 2

1.2.1. HẠT NHÂN (KERNEL) 2

1.2.2. SHELL 3

1.2.3. TỔ CHỨC HỆ THỐNG THƯ MỤC TRÊN HỆ ĐIỀU HÀNH LINUX 4

1.2.4. QUẢN LÝ THIẾT BỊ TRÊN HỆ ĐIỀU HÀNH LINUX 5

1.2.6. QUẢN LÝ BỘ NHỚ 7

1.2.7. CÁC CÂU LỆNH CƠ BẢN TRONG LINUX 7

1.3. TỔNG QUAN DỰ ÁN THỰC HIỆN 11

1.3.1. MÔ HÌNH GIAO TIẾP HỆ THỐNG 11

1.3.2. CAN SOFTWARE 12

1.3.3. LINUX USB DRIVER VÀ COMMUNICATION PROTOCOL 13

1.3.4. CAN DEVICE 14

1.4. KẾT LUẬN CHƯƠNG 14

CHƯƠNG 2: LẬP TRÌNH LINUX DRIVER 15

2.1. GIỚI THIỆU CHƯƠNG 15

2.2. LINUX DRIVER 15

2.2.1. NHIỆM VỤ CỦA DEVICE DRIVER 15

2.2.2. MÔ HÌNH PHÂN LỚP THEO CHIỀU DỌC 16

2.2.3. CÁC LỚP DEVICE VÀ MÔ ĐUN 17

2.3. CHARACTER DRIVER 18

2.3.1. SỐ HIỆU FILE THIẾT BỊ (SỐ LỚN, SỐ NHỎ) 18

2.3.2. PHƯƠNG THỨC MỞ VÀ XÓA 19

2.3.3. PHƯƠNG THỨC ĐỌC VÀ GHI 20

2.4. USB DRIVER 20

2.4.1. QUÁ TRÌNH NHẬN DẠNG THIẾT BỊ USB TRÊN LINUX 20

2.4.2. GIẢI MÃ THÔNG TIN VỀ THIẾT BỊ USB 23

2.4.3. USB ENDPOINT 24

2.4.4. USB INTERFACE 25

2.4.5. USB CONFIGURATION 25

2.5. TTY DRIVER 25

2.5.1. HÀM MỞ 25

2.5.2. HÀM ĐÓNG 25

2.5.3. HÀM GHI 26

2.6. QUY TRÌNH VIẾT USB LINUX DRIVER 26

2.6.1. KHAI BÁO DANH SÁCH THIẾT BỊ CÓ THỂ ĐƯỢC ĐIỀU KHIỂN BỞI DRIVER 26

2.6.2. KHAI BÁO CẤU TRÚC DỮ LIỆU LIÊN QUAN TỚI THIẾT BỊ 26

2.6.3. ĐĂNG KÝ VÀ HỦY ĐĂNG KÝ USB DEVICE DRIVER 27

2.6.4. HÀM THĂM DÒ THIẾT BỊ (PROBE) 28

2.6.5. HÀM NGẮT KẾT NỐI THIẾT BỊ 28

2.6.6. HÀM MỞ/ ĐỌC/ GHI THIẾT BỊ 29

2.7. KẾT QUẢ THỰC HIỆN 29

2.8. KẾT LUẬN CHƯƠNG 30

CHƯƠNG 3: COMMNICATION PROTOCOL 30

3.1. GIỚI THIỆU CHƯƠNG 30

3.2. TỔNG QUAN LÝ THUYẾT CẤU TRÚC DỮ LIỆU 31

3.2.1. HÀNG ĐỢI (QUEUE) 31

3.2.2. NGĂN XẾP (STACK) 33

3.2.3. RING BUFFER 35

3.2.4. THUẬT TOÁN TÌM KIẾM TUẦN TỰ 36

3.3. CÁC THÀNH PHẦN TRONG COMMUNICATION PROTOCOL 37

3.3.1. COMMON FORMAT 37

3.3.2. USB FRAME 38

3.3.3. CÁC STRUCT SỬ DỤNG 39

3.4. TIẾN TRÌNH THỰC HIỆN (SEQUENCE DIAGRAM) 39

3.5. KẾT QUẢ THỰC HIỆN 42

3.6. KẾT LUẬN CHƯƠNG 42

# DANH SÁCH CÁC BẢNG, HÌNH VẼ

Hình 1.1. Kiến trúc hệ điều hành Linux

Hình 1.2. Bản đồ hạt nhân Linux

Hình 1.3. Hệ thống thư mục trong Linux

Hình 1.4 Mô hình giao tiếp hệ thống

Hình 1.5. Sơ đồ khối hệ thống CAN

Hình 1.6. Tương tác giữa thiết bị và Driver

Hình 2.1. Các thành phần của Driver trên Linux

Hình 2.2. Kiến trúc tổng quan Linux Kernel

Hình 2.3. Tổng quan về Character Driver trên Linux

Hình 2.4. Danh sách thiết bị trong thư mục /dev

Hình 2.5. Mô hình biểu diễn tổng quan thiết bị USB

Hình 2.6. Hệ thống USB trên Linux

Hình 2.7. Xem thông tin các chi tiết USB sử dụng lệnh lsusb

Hình 2.8. Quy trình viết USB Driver

Hình 3.1. Tổng quan về hàng đợi

Hình 3.2 Tiến trình thêm phần tử vào cuối Queue (mảng)

Hình 3.3 Tiến trình xóa phần tử đầu Queue (mảng)

Hình 3.4. Tiến trình thêm phần tử vào cuối Queue (con trỏ)

Hình 3.5. Tiến trình xóa phần tử đầu Queue (con trỏ)

Hình 3.6. Sơ đồ minh họa một ngăn xếp và các hoạt động diễn ra

Hình 3.7. Tiến trình thêm phần tử vào Stack

Hình 3.8. Tiến trình lấy phần tử từ Stack

Hình 3.9. Cấu trúc của bộ đệm vòng

Hình 3.10. Thực hiện bộ đệm tuyến tính của bộ đệm vòng

Hình 3.11. Tiến trình xử lý trường hợp sử dụng bộ đệm

# DANH SÁCH CÁC KÝ HIỆU, CHỮ VIẾT TẮT

**MỞ ĐẦU**

# Chương 1: Tổng quan đề tài

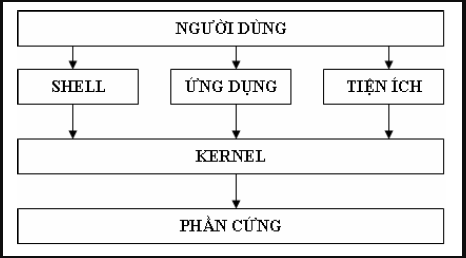
* 1. **Giới thiệu chương**

Dự án được thực hiện trên nền tảng hệ điều hành Linux, nên chương 1 giới thiệu về tổng quan kiến trúc hệ điều hành Linux, giới thiệu tổng quan về toàn bộ đề tài “Thiết kế và thi công phần mềm mô phỏng CAN bus”.

* 1. **Tổng quan kiến trúc 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



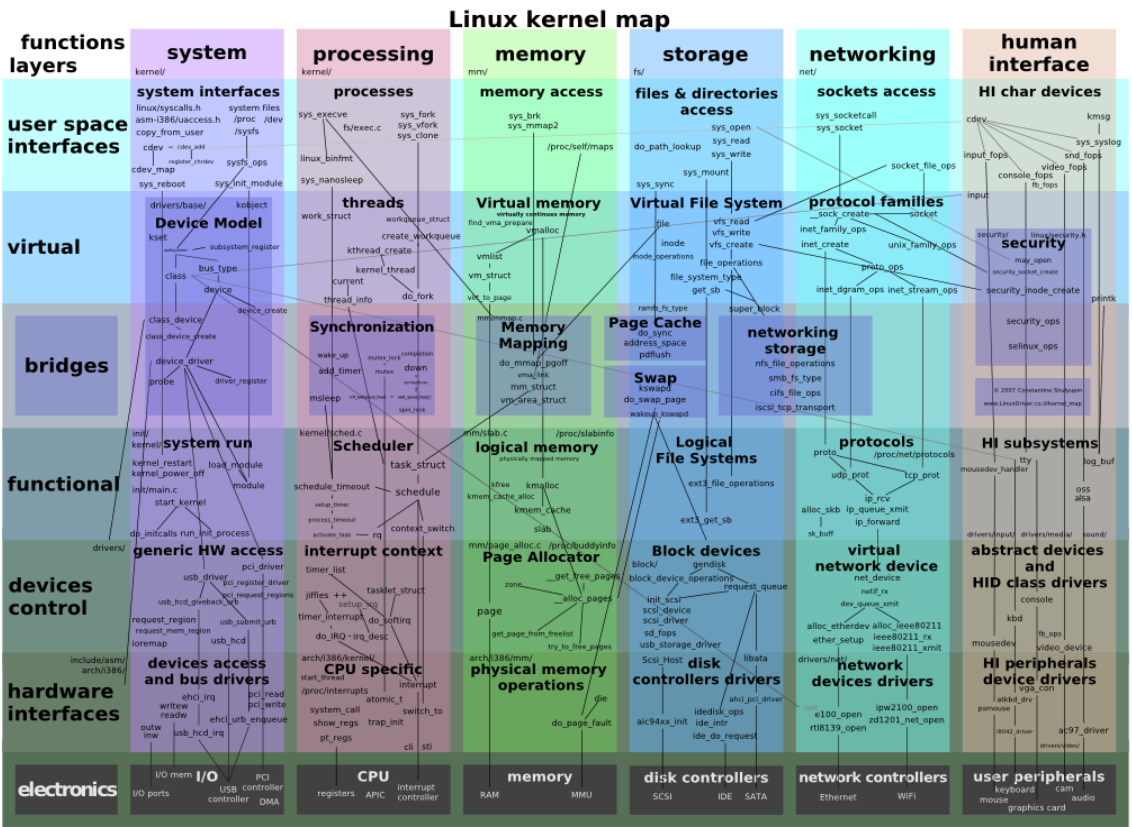
Hình 1.1. Kiến trúc hệ điều hành Linux

* + 1. ***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. Hạt nhân được phát triển không ngừng , thường có 2 phiên bản mới nhất : 1 bản phát triển mới nhất và 1 bản ổn định mới nhất. 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 được xem là trái tim của hệ điều hành Linux, ban đầu được phát triển cho các CPU Intel 80386. Điểm mạnh của loại của loại CPU này là khả năng quản lý 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



Hình 1.2. Bản đồ hạt nhân Linux

* + 1. ***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

Có nhiều loại shell được dùng trong Linux. Điểm quan trọng để phân biệt các shell với nhau là bộ lệnh của mỗi shell

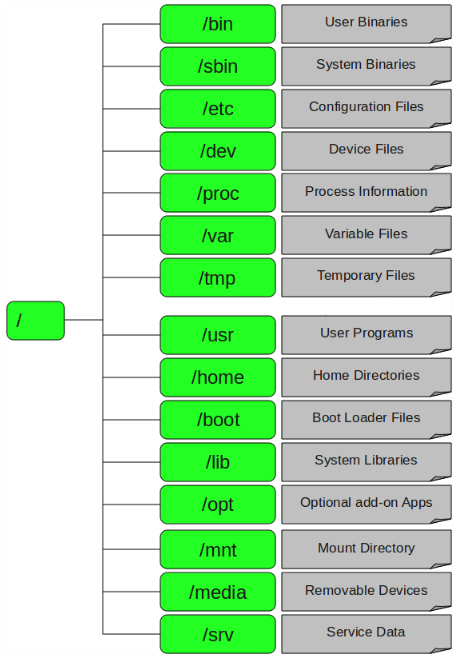
Shell sử dụng trong Linux là GNU Bourne Again Shell (bash). Shell này là shell phát triển từ Bourne Shell, là shell sử dụng chính trong các hệ thống Unix, với nhiều tính năng mới như : Điều khiển các tiến trình , history , tên tập tin dài…

* + 1. ***Tổ chức hệ thống thư mục trên hệ điều hành Linux***

Hệ thống tập tin của Linux và Unix được tổ chức theo một hệ thống phân bậc tương tự cấu trúc của một cây phân cấp. Bậc cao nhất của hệ thống tập tin là thư mục gốc, được ký hiệu bằng gạch chéo “/” (root directory).

Đối với các hệ điều hành Unix và Linux tất các thiết bị kết nối vào máy tính đều được nhận dạng như các tập tin, kể cả những linh kiện như ổ đĩa cứng, các phân vùng đĩa cứng và các ổ USB. Điều này có nghĩa là tất cả các tập tin và thư mục đều nằm dưới thư mục gốc, ngay cả những tập tin biểu tượng cho các ổ đĩa cứng.

Nằm dưới thư mục gốc (/) có một loạt các thư mục quan trọng của hệ thống tập tin được công nhận ở tất cả các bản phân phối Linux khác nhau. Dưới đây là danh sách các thư mục thông thường được nhìn thấy dưới thư mục gốc (/):



Hình 1.3. Hệ thống thư mục trong Linux

* + 1. ***Quản lý thiết bị trên hệ điều hành Linux***

Một trong những mục tiêu chính của hệ điều hành là giúp cho người sử dụng độc lập với thiết bị. Trong Linux, các thiết bị phần cứùng đều được xem như là các tập tin thông thường : chúng có thể được mở, đóng , đọc , ghi, và sử dụng cùng những lời gọi hệ thống để thao tác. Mọi thiết bị trong hệ thống đều được biểu diễn bởi các tập tin thiết bị, ví dụ đĩa IDE đầu tiên của hệ thống được biểu diễn bởi /dev/hda.

Linux ánh xạ tập tin thiết bị vào các lời gọi hệ thống. Linux hỗ trợ ba loại thiết bị là : ký tự, khối và mạng. Có nhiều chương trình điều khiển thiết bị khác nhau trong hạt nhân của Linux nhưng chúng cùng chia sẻ những thuộc tính chung :

* Mã nguồn của hạt nhân : chương trình điều khiển thiết bị là một phần của hạt nhân và cũng như những đoạn mã nguồn khác nếu nó bị lỗi nó có thể đe dọa nghiêm trọng đến hệ thống.
* Chương trình điều khiển thiết bị cung cấp một giao tiếp chuẩn với hạt nhân của Linux. Chúng sử dụng những dịch vụ chuẩn từ hạt nhân này.

Linux sử dụng cơ chế DMA và sử dụng một vector cấu trúc dma\_chan để quản lý các kênh DMA (mỗi kênh có một vector).

Mỗi lớp điều khiển thiết bị : ký tự, khối, mạng cung cấp những giao tiếp chung với hạt nhân. Khi một thiết bị được khởi tạo, thanh ghi điều khiển thiết bị bên trong hạt nhân của Linux được thêm một entry trong vector chrdevs của cấu trúc devive\_struct. Một định danh(id) được xác định cho vector này và là cố định. Mỗi entry trong vector chrdevs, một device\_struct gồm hai phần tử : một con trỏ đến tên của thanh ghi điều khiển thiết bị và một con trỏ tới khối của tập tin thao tác. Khối của tập tin thao tác này là địa chỉ của những thủ tục trong chương trình điều khiển thiết bị mà chúng kiểm soát những thao tác như mở, đóng, đọc, ghi. Nội dung của /proc/devices cho các thiết bị ký tự được lấy từ vector chrdevs.

Các thiết bị khối được truy xuất như những tập tin. Linux quản lý các thanh ghi thiết bị khối bằng vector blkdevs, và tương tự như vector chrdevs mỗi entry của nó là các cấu trúc device\_struct nhưng có hai loại thiết bị khối : SCSI và IDE. Mỗi chương trình điều khiển thiết bị khối phải cung cấp một giao tiếp với vùng đệm cũng như những thao tác tập tin thông thường. Cấu trúc lk\_dev\_struct bao gồm địa chỉ của các thủ tục được yêu cầu và một con trỏ đến những cấu trúc dữ liệu cần thiết, mỗi cấu trúc này sẽ đại diện cho một yêu cầu từ vùng đệm cho những điều khiển đọc hoặc ghi một khối dữ liệu.

* + 1. ***Quản lý tiến trình***

Mỗi tiến trình trong Linux được biểu diễn bằng một cấu trúc dữ liệu task\_struct (task có nghĩa là tiến trình trong Linux). Linux sử dụng task vector để quản lý các con trỏ đến các task\_struct, mặc định là có 512 phần tử. Khi một tiến trình được tạo ra, một task\_struct mới được cấp phát trong bộ nhớ và được thêm vào vector task. Linux hỗ trợ hai loại tiến trình là loại bình thường và loại thời gian thực. Cấu trúc task\_struct gồn những trường như sau : trạng thái (thi hành, chờ, ngưng, lưng chừng), thông tin lập lịch, định danh, thông tin liên lạc giữa các tiến trình, liên kết, định thời gian, hệ thống tập tin, bộ nhớ ảo, ngữ cảnh.

Trong Linux cơ chế đường ống cài đặt sử dụng hai cấu trúc file trỏ đến cùng inode VFS tạm thời xác định một trang vật lý trong bộ nhớ. Khi tiến trình ghi thực hiện việc ghi vào đường ống, các byte sẽ được chép vào trang dữ liệu chia sẻ. Linux phải đồng bộ quá trình truy xuất trong đường ống. Tiến trình ghi sử dụng các hàm thư viện ghi chuẩn.

Các cơ chế socket, semaphores và bộ nhớ chia sẻ của Linux gần giống với hệ thống Unix System V.

* + 1. ***Quản lý bộ nhớ***

Linux hỗ trợ hệ thống chia sẻ bộ nhớ IPC (Inter Process Communication- thông tin liên lạc giữa các tiến trình) của Unix System V. Linux sử dụng bộ nhớ ảo và cơ chế phân trang. Trên hệ thống Alpha AXP một trang có kích thước 8Kb và trên hệ thống Intel x86 một trang có kích thước 4Kb. Chiến lược thay trang được áp dụng là LRU.

Linux có số cấp bảng trang tùy vào các nền khác nhau (Alpha 3, Intel x86 2). Bảng trang sẽ được chuyển thành các tiến trình đặc thù trên các nền khác nhau, điều này giúp cho việc thao tác trên bảng trang không lệ thuộc vào các nền này. Linux sử dụng thuật toán Buddy để thực hiện việc cấp phát và thu hồi các khối của trang. ( 1 khối = 1, 2, hoặc 4 trang...)

* + 1. ***Các câu lệnh cơ bản trong linux***
       1. ***Các câu lệnh về thư mục và file***
* Lệnh cat

Cú pháp: cat file [>|>] [destination file]

Lệnh cat sẽ hiển thị nội dung của một file. Nó thường hữu ích để kiểm tra nội dung của một file bằng sử dụng câu lệnh cat. Đối số mà bạn đưa vào lệnh cat là file bạn muốn xem. Để xem toàn bộ nội dung của một file:

cat name

Lệnh cat cũng có thể trộn nhiều file đang tồn tại vào một file:

cat name1 name2 name3 > allnames

* Lệnh chmod

Cú pháp: chmod [-R] permission-mode file hoặc thư mục

Lệnh chmod dùng để thay đổi quyền truy cập file hoặc thư mục. Ví dụ:

chmod myscript.pl

Để thay đổi quyền của một thư mục và tất cả các file, các thư mục con của thư mục đó sử dụng câu lệnh:

chmod –R 744 public\_html

* Lệnh clear

Xoá màn hình, trả lại dấu chắc dòng lệnh ở phía trên của màn hình

clear

* lệnh cmp

Cú pháp: cmp [ -ls ] file1 file2

Lệnh này so sánh nội dung của hai file. Nếu không có sự khác nhau nào, lệnh cmp sẽ kết thúc một cách yên lặng, tuỳ chọn –l sẽ n ra số byte và các giá trị khác nhau giữa hai file. Tuỳ chọn –s không hiển thị cài gì cả, nó chỉ trả lại trạng thái chỉ ra rằng sự tương đương giữa hai file. Giá trị 0 được trả lại nếu các file giống hệt nhau, giá trị bằng 1 nếu hai file khác nhau và lớn hơn 1 nếu lỗi xuất hiện khi thực hiện câu lệnh.

* Lệnh cp

Cú pháp: cp [ -R ] file\_hoặc\_thư\_mục file\_hoặc\_thư\_mục

Lệnh cp sẽ sao chép một file từ thư mục nguồn đến thư mục đích được đưa vào. Để sao chép toàn bộ các file và các thư mục con bên trong thư mục mong muốn, bạn sử dụng câu lệnh cp với tuỳ chọn –R

* Lệnh file

Cú pháp: file filename

Câu lệnh xác định kiểu của file. Nếu file không phải là file thông thường, kiểu của file được xác định.

* Lệnh find

Câu lệnh find tìm các file và các thư mục.

Cú pháp : find [path] [-type fd] [-name mẫu] [-atime [+-] số\_ngày] [-exec câu\_lệnh {} \;] [-empty].

* Lệnh grep

Cú pháp: grep [–viw] mẫu file

Lệnh grep cho phép bạn tìm kiếm một hoặc nhiều file có các mẫu ký tự đặc biệt. Mỗi dòng của mỗi file chứa các mẫu được hiển thị trên màn hình. Câu lệnh grep hữu ích khi bạn có nhiều file và bạn muốn tìm ra file chứa từ hoặc câu xác định. Sử dụng tuỳ chọn –v, bạn có thể hiển thị các file không chứa một mẫu. Ví dụ, để chọn các dòng trong data.txt không chứa từ “the” ta thực hiện:

grep –vw ‘the’ data.txt

Nếu tuỳ chọn –w không được xác định thì bất kỳ các từ chứa “the” đều phù hợp như “together”. Tuỳ chọn –w được xác định buộc mẫu phải là toàn bộ một từ. Cuối cùng , tuỳ chọn –i bỏ qua sự khác nhau giữa các ký tự chữ hoa và ký tự chữ thường khi tìm kiếm mẫu.

* Lệnh head

Cú pháp: head [-count | -n number] filename

Câu lệnh này sẽ hiển thị vài dòng đầu tiên của một file. Bởi mặc định, 10 dòng đầu

của một file được hiển thị. Tuy nhiên, bạn có thể sử dụng các tuỳ chọn để xác định số dòng hiển thị. Ví dụ:

head -2 doc.txt

sẽ hiển thị hai dòng đầu tiên.

* Lệnh locate

Cú pháp : locate từ\_khoá

Câu lệnh locate tìm đường dẫn đến một file đặc biệt hay một câu lệnh. Lệnh locate sẽ tìm kiếm chính xác hay một phần của chuỗi phù hợp. Ví dụ:

locate foo

kết quả tìm kiếm sẽ đưa ra các file có tên chứa từ khoá ‘foo’ theo đường dẫn tuyệt đối hoặc sẽ không đưa ra kết quả nếu không có tên file như vậy.

* Lệnh ls

Lệnh ls cho phép bạn đưa ra danh sách các file và các thư mục con.

Cú pháp : ls [-1aRl] file\_hoặc\_thư\_mục

Khi sử dụng tuỳ chọn -1 , nó chỉ hiển thị tên file và tên thư mục con của thư mục hiện tại. Khi chọn tuỳ chọn –l, một danh sách các file và thư mục con của thư mục hiện tại được hiển thị với đầy đủ các thông tin về file và thư mục. Tuỳ chọn –a cho phép hiển thị tất cả các file và thư mục (kể cả các file ẩn, tên file bắt đầu bằng dấu chấm) trong thư mục hiện tại. Tuỳ chọn –R sẽ hiển thị tất cả các file và các thư mục con bên trong nó nếu có.

* Lệnh mkdir

Cú pháp: mkdir thư\_mục

Để tạo một thư mục, sử dụng câu lệnh mkdir. Chỉ có 2 giới hạn khi chọn tên thư mục, đó là tên của thư mục có thể lên tới 255 ký tự và tên thư mục có thể chứa bất kỳ ký tự nào trừ ký tự ‘/’.Ví dụ:

mkdir dir1 dir2 dir3

Lệnh trên tạo ra ba thư mục, nằm bên trong thư mục hiện tại.

* Lệnh mv

Cú pháp : mv [-if] file\_nguồn file\_đích

Sử dụng lệnh mv để dịch chuyển hay đổi tên các file hay các thư mục. Câu lệnh thực hiện việc dịch chuyển hay đổi tên phụ thuộc vào file\_đích có là một thư mục hay không.

* Lệnh pwd

Cú pháp: pwd

Câu lệnh này hiển thị tên thư mục hiện tại bao gồm cả đường dẫn tuyệt đối. Ví dụ:

pwd

Trên màn hình hiển thị :

/dev/

* Lệnh rm

Cú pháp: rm [-rif] thư\_mục/file

Để xoá thư mục hoặc file, sử dụng câu lệnh rm. bạn có thể xoá nhiều file sử dụng ký tự đại diện hoặc gõ vào tên các file. Ví dụ:

rm doc1.txt doc2.txt doc3.txt

* Lệnh tail

Cú pháp: tail [-count | -fr] tên\_file

Câu lệnh tail hiển thị phần cuối của một file, mặc định nó sẽ hiển thị 10 dòng cuối

cùng của file. Để hiển thị 50 dòng cuối cùng của file doc.txt, bạn có thể sử dụng câu lệnh:

tail -50 doc.txt

Tuỳ chọn –r sẽ thực hiện công việc ngược lại, mặc định nó sẽ hiển thị tất cả các dòng trừ 10 dòng cuối cùng. Tuỳ chọn –f hữu ích khi đang giám sát một file. Với tuỳ chọn này, tail sẽ chờ cho dữ liệu mới được ghi vào file. Khi dữ liệu mới được thêm vào file, tail sẽ hiển thị dữ liệu lên màn hình. Để dừng lệnh tail khi đang giám sát file, chọn tổ hợp phím Ctrl + C bởi vì lệnh tail không tự dừng được.

* + - 1. ***Các câu lệnh quản lý tiến trình***
* Lệnh bg

Cú pháp: bg

Đưa một tiến trình đang chạy về chạy sau ở hậu cảnh (tiến trình nền)

* Lệnh fg

Cú pháp: fg [%job-number]

Câu lệnh cho phép chuyển một tiến trình nền lên chạy ở trên tiền cảnh

* Lệnh jobs

Cú pháp: jobs

Lệnh này cho phép hiển thị các tiến trình đang chạy

* 1. **Tổng quan dự án thực hiện**
     1. ***Mô hình giao tiếp hệ thống***

CAN Device

PC Driver

CAN Simulator Software

CAN

USB

Hình 1.4. Mô hình giao tiếp hệ thống gồm 3 phần chính

* 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 CAN Device thông qua cổng CAN. CAN Device kết nối với Software, 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.

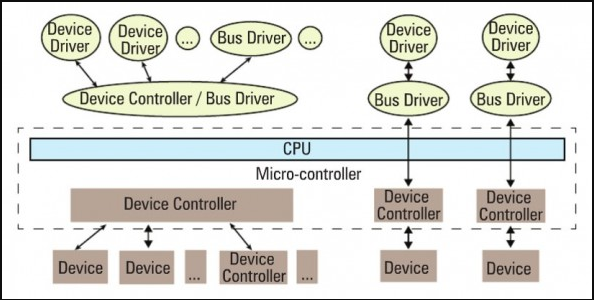


**Hình 1.5. Sơ đồ khối hệ thống CAN**

Sau khi kết nối giữa ECU và Software , 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. ***CAN Software***
       1. ***CAN Database***
       2. ***CAN Simulator***
    2. ***Linux USB Driver và Communication Protocol***
       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 tương tự như vai trò của một phi công hoặc một hệ thống bay tự động được giám sát bởi phi công, 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 1.5. 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 hardware và software .Khi một device 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 Hardware thì Driver sẽ nhận biết được thông điệp, sau đó ghi vào file descriptor tương ứng. Hardware 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 Software.

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

* CAN software có thể đóng/ mở thiết bị thông qua device node.
* CAN software có thể cấu hình baurd rate thông qua file descriptor.
* CAN software có thể gửi/nhận message thông qua file descriptor
  + - 1. ***Communication Protocol***

Giao thức truyền dữ liệu giữa cổng software và board TMS123G, vì board đóng vai trò như một node CAN trên mạng CAN bus, phần mềm PC đóng vai trò hiện 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 board và ngược lại, theo các chức năng mà người giám sát mạng CAN cần đến, ví dụ: lấy cấu hình mạng CAN từ board thông qua cổng USB, hay thực hiện tác vụ mô phỏng hoạt động mạng CAN (simulation),…

* + 1. ***CAN Device***
  1. **Kết luận chương**

Chương 1 đã trình bày tổng quan về toàn bộ đề tài, giúp hiểu rõ hơn về phần mềm mô phỏng CAN bus. Chương tiếp theo sẽ đi sâu vào phân tích Linux Driver.

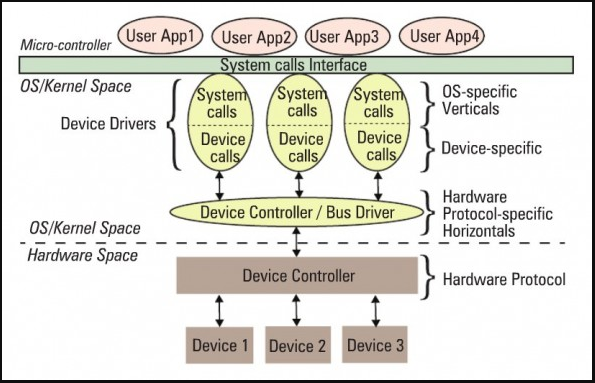
# Chương 2: Lập trình Linux Driver

* 1. **Giới thiệu chương**

Chương 2 trình bày chi tiết về cấu trúc của Linux Driver, đi sâu vào quy trình thiết kế Linux USB Driver.

* 1. **Linux Driver**
     1. ***Nhiệm vụ của Device Driver***

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)



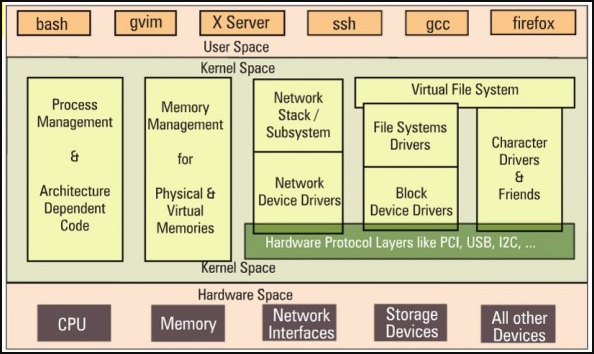
Hình 2.1 Các thành phần của Driver trên Linux

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 dọc***

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.



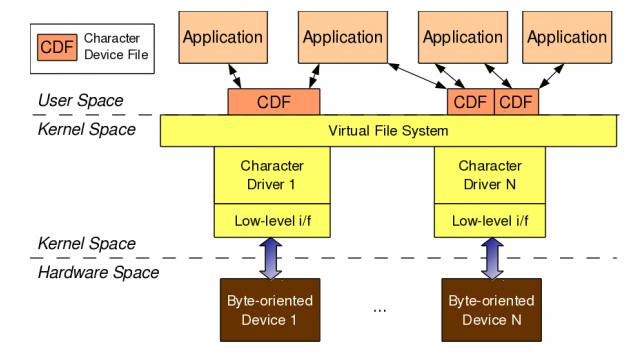
Hình 2.2 Kiến trúc tổng quan Linux Kernel

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**



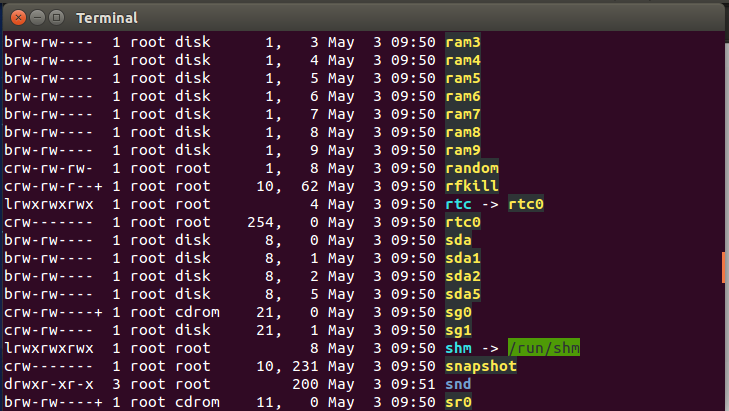
Hình 2.3. Tổng quan về character driver trên Linux

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 (số lớn, số nhỏ)***

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 2.4.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 xóa***
       1. ***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
  + - 1. ***Phương thức xóa***
    1. ***Phương thức đọc và ghi***
       1. ***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).

* + - 1. ***Phương thức ghi***

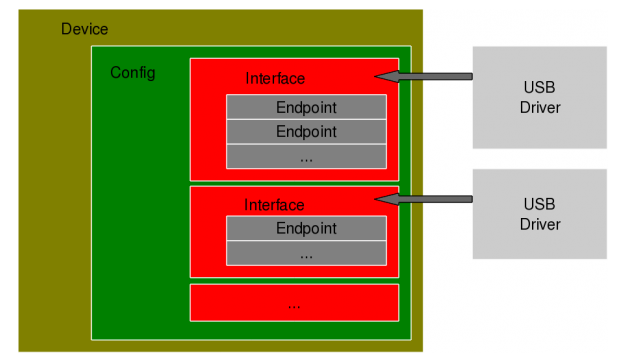
Nếu giá trị bằng biến đếm, số byte yêu cầu đã được chuyển

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>

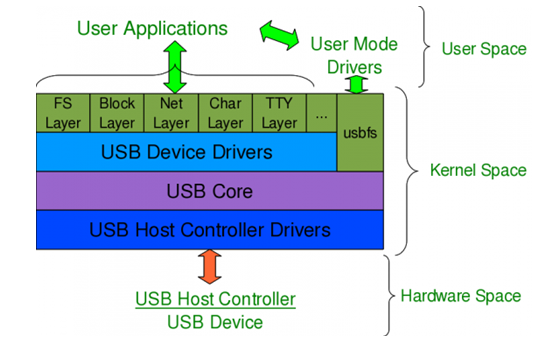
* 1. **USB Driver**
     1. ***Quá trình nhận dạng thiết bị USB trên Linux***



Hình 2.5. Mô hình biểu diễn tổng quan thiết bị USB

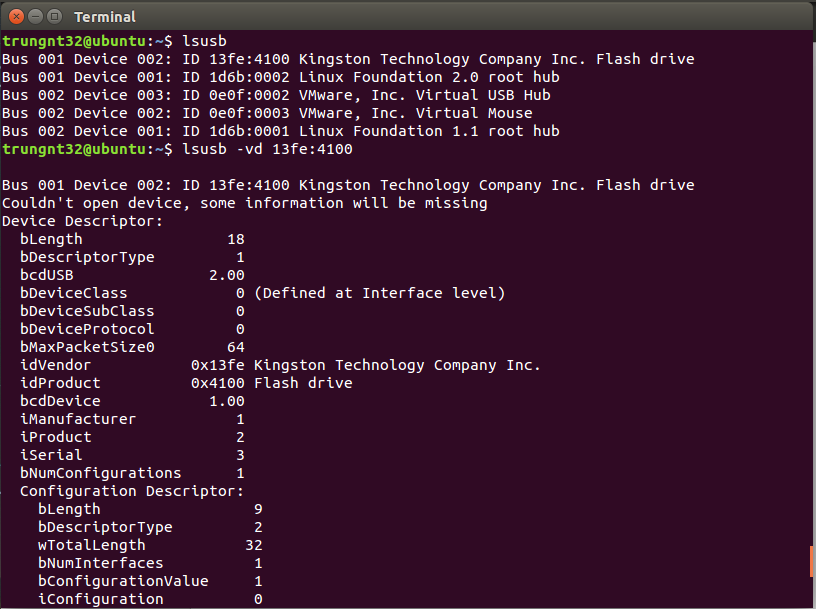
Khi có một thiết bị usb hợp lệ được cắm vào hệ thống Linux, cho dù nó có driver hay không thì nó cũng vẫn được nhận diện (detect) bởi phần cứng ở tầng nhân (kernel space) của hệ thống Linux mà đã được hỗ trợ giao thức usb. Hệ thống có thể làm điều này là bởi vì khả năng của chính bản thân giao thức usb 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 2.6. 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 2.7. 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. ***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. ***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. ***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. **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***

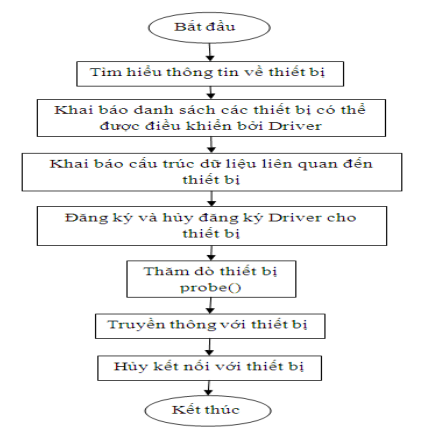
Hàm đóng được gọi bởi lõi TTY khi lệnh đóng được gọi bởi người dùng trên các tập tin xử lý đã được tạo ra trước đó với hàm mở.

* + 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.

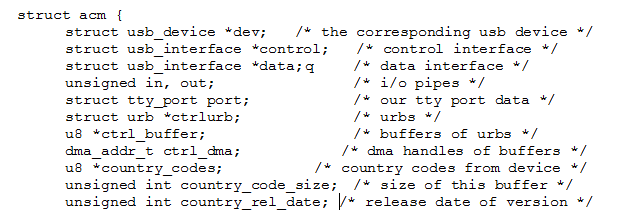
* 1. **Quy trình viết USB Linux Driver**



Hình 2.8. Quy trình viết USB Driver

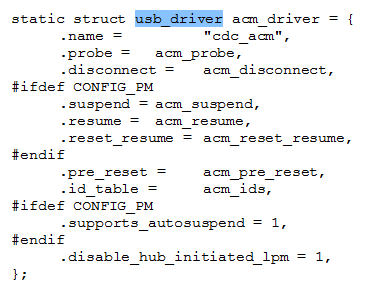
* + 1. ***Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver***
* Macro USB\_DEVICE (vender, product) : tạo cấu trúc usb\_device\_id với Idvendor và Idproduct
* Khai báo cấu trúc usb\_device\_id với USB Core, sử dụng macro MODULE\_DEVICE\_TABLE(usb, usb\_device\_id[]);
  + 1. ***Khai báo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

* + - 1. ***Hành động 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);

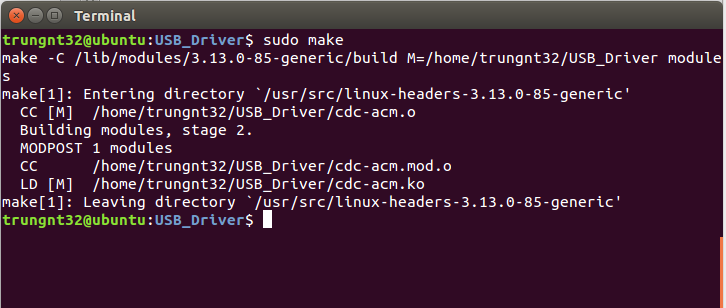
file->private\_data = dev;

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().

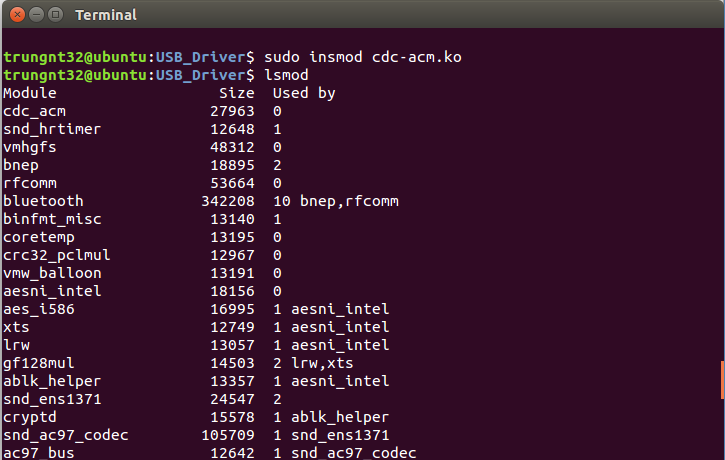
* 1. **Kết quả thực hiện**

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

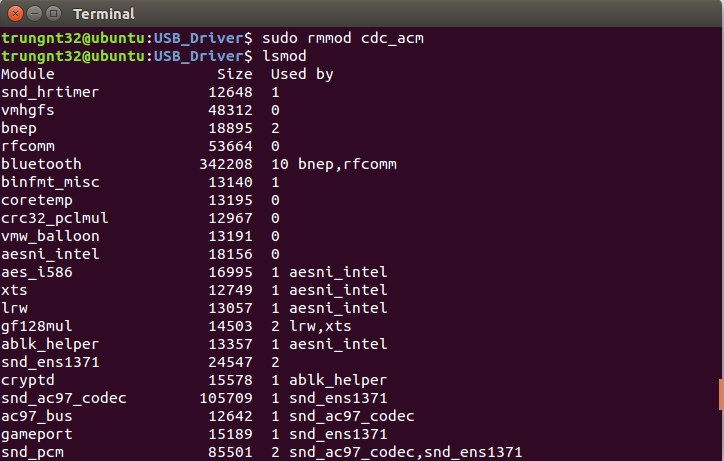
* Biên dịch bằng lệnh make (tạo ra file cdc-acm.ko): sudo make



* 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. **Kết luận chương**

Qua chương 2, chúng ta có thể hiểu được về Linux Driver và có thể tiến hành lập trình Driver cho USB. Chương tiếp theo sẽ tìm hiểu cách tạo các giao thức giúp truyền dữ liệu từ phần mềm sang phần cứng.

# Chương 3: Commnication Protocol

* 1. **Giới thiệu chương**

Phần mềm và phần cứng truyền nhận dữ liệu thông qua các giao thức truyền nhận ( communication protocol). Chương này phân tích quá trình thiết kế và thực hiện các giao thức trên.

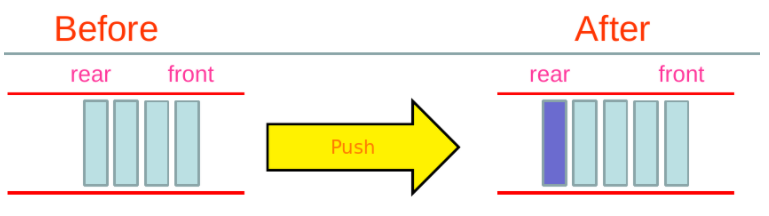
* 1. **Tổng quan lý thuyết cấu trúc dữ liệu**
     1. ***Hàng đợi (Queue)***

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.



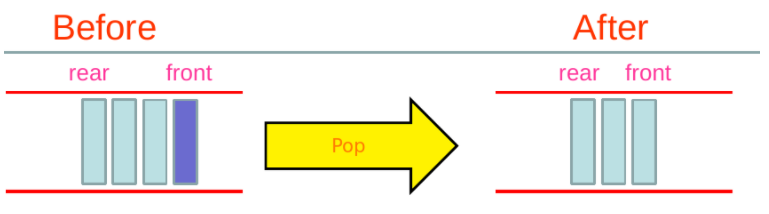
Hình 3.1. Tổng quan về hàng đợi

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



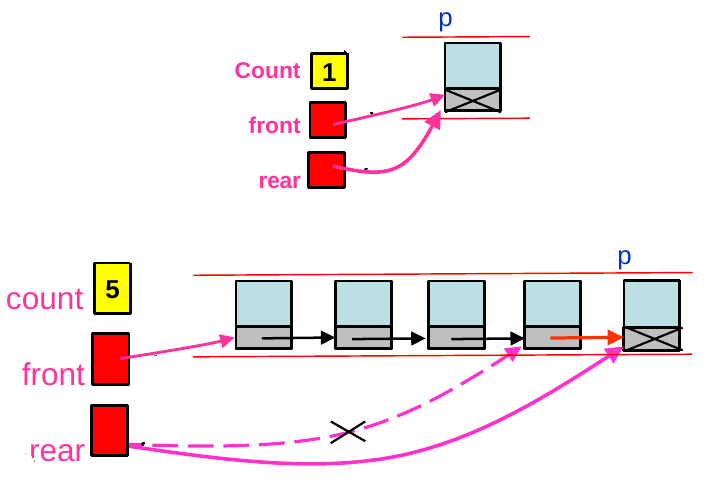
Hình 3.2. Tiến trình thêm phần tử vào Queue

* 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.



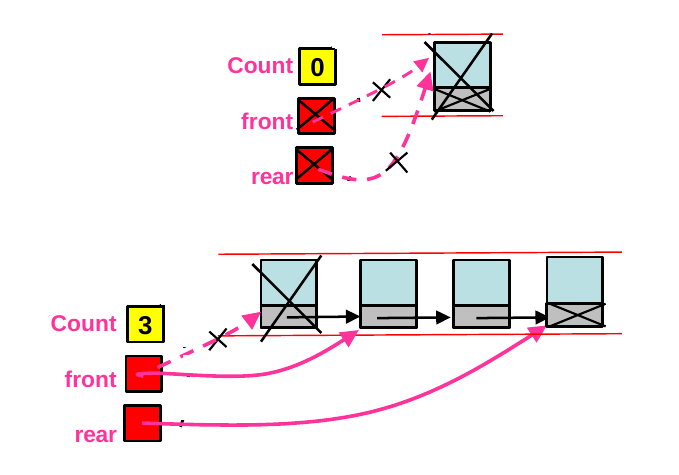
Hình 3.3. Tiến trình xóa phần tử đầu Queue

* 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 3.4. Tiến trình thêm phần tử vào cuối Queue

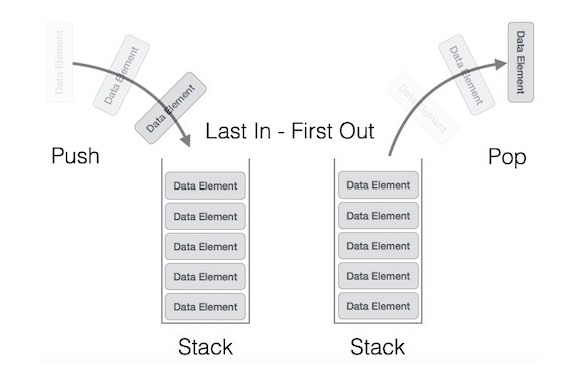
* Xóa phần tử đầu Queue bằng con trỏ: Ta kiểm tra Queue có rỗng không, Nếu không rỗng kiểm tra xem có 1 hay nhiêu hơn 1 phần tử, nếu có 1 phần tử thì ta khởi tạo lại Queue, nếu có nhiều hơn ta cho Front trỏ đến tiếp theo. Giảm count xuống 1.



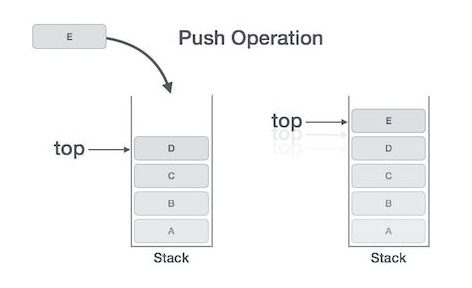
Hình 3.5. Tiến trình xóa phần tử đầu Queue

* + 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.

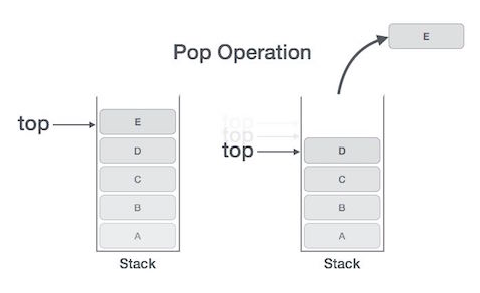
Hình 3.6. Sơ đồ minh họa một ngăn xếp và các hoạt động diễn ra

* 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 3.7. 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.

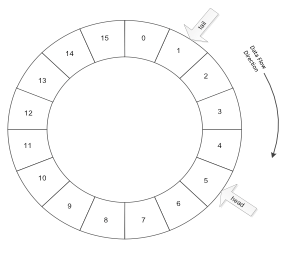


Hình 3.8. Tiến trình lấy phần tử từ Stack

* + 1. ***Ring buffer***

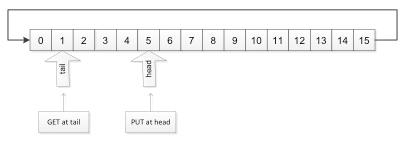
Bộ đệm vòng hữu ích trong việc truyền dữ liệu giữa các quá trình không đồng bộ. Bộ đệm vòng là một hàng đợi vòng (quece), có đặc tính dữ liệu FIFO đầu tiên.

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 3.9. 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 3.10. 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 3.11. Tiến trình xử lý trường hợp sử dụng bộ đệm

* + 1. ***Thuật toán tìm kiếm tuần tự***

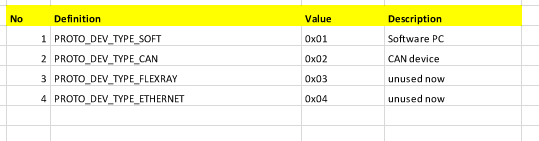
Tìm kiếm là công việc quan trọng đối với các hệ thống tin học và có liên quan mật thiết với quá trình sắp xếp dữ liệu.

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.

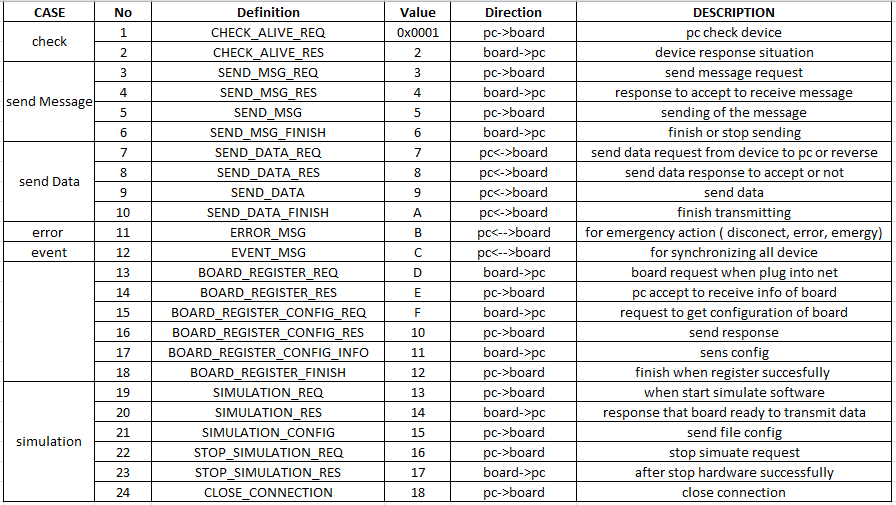
* Bắt đầu từ khoá đầu tiên, lần lượt so sánh khoá x với khoá tương ứng trong dãy.
* Quá trình tìm kiếm kết thúc khi tìm được khoá thoả mãn hoặc đi đến hết dãy hoặc gặp điều kiện dừng vòng lặp.Đây là kĩ thuật tìm kiếm cổ điển nhất trên 1 danh sách chưa được sắp xếp. Nội dung cơ bản của phương pháp này 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. **Các thành phần trong Communication Protocol**
     1. ***Common Format***



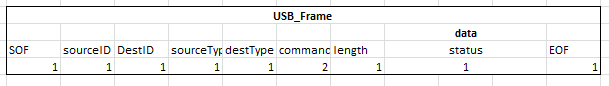
* SourceType / DestType | SourceID / DestID



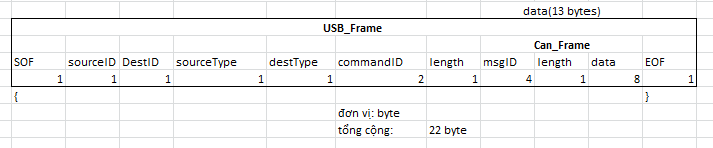
* CommanID ( 2 byte)



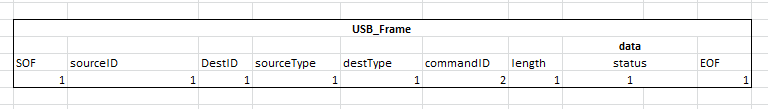
* Data : USB Frame
  + 1. ***USB Frame***



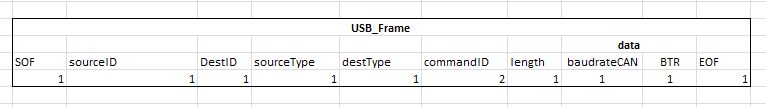
* Trường hợp dành cho mô phỏng (Simulator)



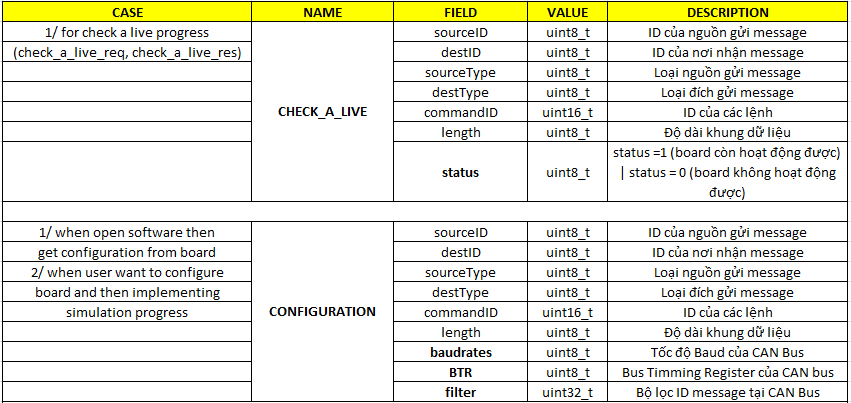
* Trường hợp dành cho kiểm tra thiết bị (Checkalive)

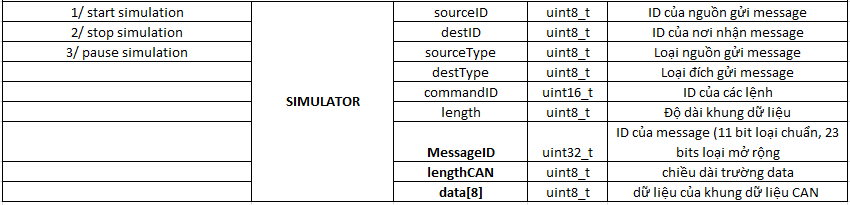


* Trường hợp dành cho cấu hình (Configuration)

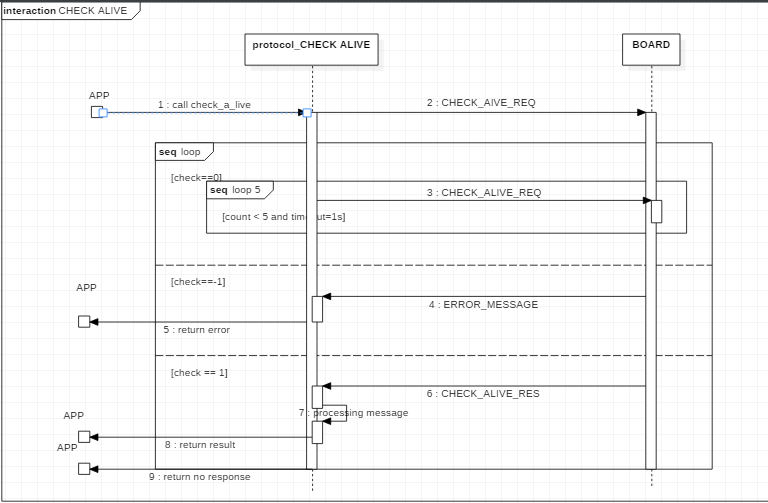


* + 1. ***Các struct sử dụng***

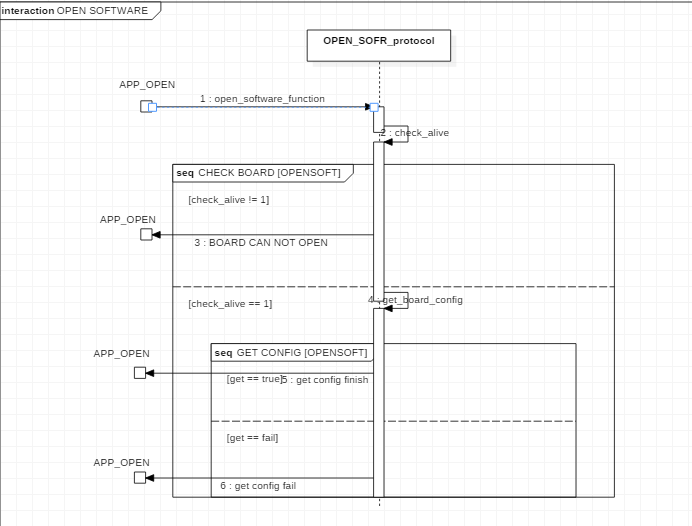




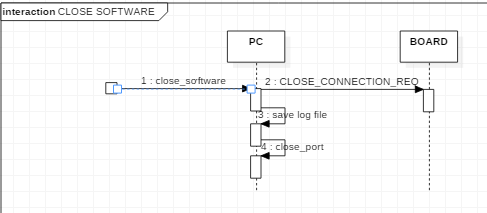
* 1. **Tiến trình thực hiện (Sequence Diagram)**
* Checkalive



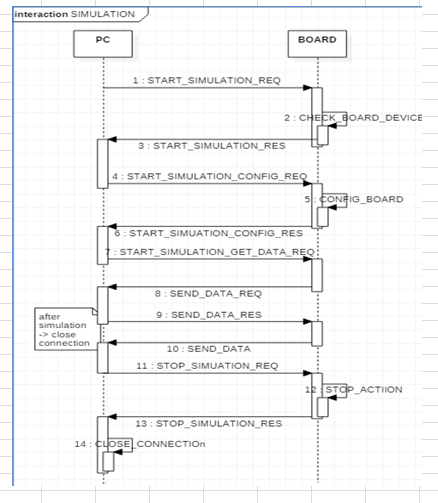
* Opensoft



* Closesoft



* Simulation



* 1. **Kết quả thực hiện**
* **Test CheckAlive**: gửi xuống cổng , respone 0,1
* **Test Simulation**:
* Gửi xuống thiết bị một array gồm 100 phần tử, sau đó board gửi lên lại array đó, trong đó có 91 phần tử là đúng CAN Frame, 9 phần tử còn lại không đúng nên không lưu vào bộ đệm. Hiển thị bộ nhớ đệm nhận dc 91 phần tử.
* Gửi từng CAN Frame trong Ring Buffer gồm 100 phần tử vào bộ đệm thì lập tức gửi từng CAN Frame xuống thiết bị và sau khi thiết bị nhận dc sẽ gửi lại CAN Frame lại PC và Frame này lưu vào vị trí tiếp theo trong Ring Buffer, cho đến vị trí 100 thì lưu lại ở vị trí 0.



* 1. **Kết luận chương**

Chương 3 đi vào thiết kế giao thức truyền dữ liệu, giúp hiểu được cách dữ liệu truyền từ PC sang phần cứng

**TÀI LIỆU THAM KHẢO**