{Trang nhận xét cho người hướng dẫn tại công ty}

{Trang nhận xét cho người hướng dẫn tại trường}

{Trang nhận xét cho người phản biện}

**TÓM TẮT**

**Tên đề tài:**

THIẾT KẾ VÀ THỰC THI DRIVER VÀ GIAO THỨC CHO HỆ THỐNG

MÔ PHỎNG MẠNG CAN

**Sinh viên thực hiện:**

1. Hồ Thăng Huy

Số thẻ SV: 106120113 - Lớp: 12DT3

1. Đặng Trung Nguyên

Số thẻ SV: 106120067 - Lớp: 12DT2

1. Huỳnh Vũ Tiến

Số thẻ SV: 106120206 - Lớp 12DT4

**Tóm tắt nội dung:**

Đề tài : **“**THIẾT KẾ VÀ THỰC THI: THIẾT BỊ, DRIVER VÀ GIAO THỨC CHO HỆ THỐNG MÔ PHỎNG MẠNG CAN”

Nội dung đề tài này thiết kế và thực thi giao thức truyền nhận dữ liệu giữa kit DK-TM4C123G với PC sử dụng USB driver loại CDC-ACM (Communication Device Class- Abstract Control Model) trên hệ điều hành linux, truyền nhận dữ liệu giữa kit DK-TM4C123G và bus CAN và thiết kế giao thức sử dụng biểu đồ tuần tự (sequence diagram) và thực thi trên nền tảng Qt Framework sử dụng ngôn ngữ lập trình C/C++.

|  |  |
| --- | --- |
| ĐẠI HỌC ĐÀ NẴNG  **TRƯỜNG ĐẠI HỌC BÁCH KHOA**  KHOA ĐIỆN TỬ - VIỄN THÔNG | **CỘNG HÒA XÃ HÔI CHỦ NGHĨA VIỆT NAM**  Độc lập - Tự do - Hạnh phúc |

**NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TT | Họ tên sinh viên | Số thẻ SV | Lớp | Ngành |
| 1 | Hồ Thăng Huy | 106120113 | 12DT3 | Điện Tử - Viễn Thông |
| 2 | Đặng Trung Nguyên | 106120067 | 12DT2 | Điện Tử - Viễn Thông |
| 3 | Huỳnh Vũ Tiến | 106120206 | 12DT4 | Điện Tử - Viễn Thông |

1. *Tên đề tài đồ án:*

“**THIẾT KẾ VÀ THỰC THI : THIẾT BỊ , DRIVER VÀ GIAO THỨC CHO HỆ THỐNG MÔ PHỎNG MẠNG CAN**”

1. *Đề tài thuộc diện:*  *Có ký kết thỏa thuận sở hữu trí tuệ đối với kết quả thực hiện*
2. *Nội dung các phần :*

|  |  |  |
| --- | --- | --- |
| TT | Họ tên sinh viên | Nội dung |
| 1 | Hồ Thăng Huy | Nghiên cứu và thực thi cdc-acm linux driver |
| 2 | Đặng Trung Nguyên | Thiết kế giao thức giữa PC và board DK-TM4C123G  Và thực thi giao thức trên PC |
| 3 | Huỳnh Vũ Tiến | Tìm hiều giao thức CAN  Nghiên Cứu và thực thi giao thức trên board DK-TM4C123G |

1. *Ngày giao nhiệm vụ đồ án: ……../……./201…..*
2. *Ngày hoàn thành đồ án: ……../……./201…..*

|  |  |
| --- | --- |
|  | *Đà Nẵng, ngày tháng năm 201* |
| **Trưởng Bộ môn**………………………. | **Người hướng dẫn** |

**LỜI CẢM ƠN**

**Kính gởi:**

* **Trường Đại học Bách khoa Đà Nẵng**
* **Công ty FPT Software Đà Nẵng**

Đầu tiên chúng tôi xin chân thành cảm ơn quý thầy cô trong khoa Điện tử - Viễn thông, trường Đại học Bách khoa Đà Nẵng đã tận tình truyền đạt kiến thức trong suốt thời gian học tập tại trường. Với vốn kiến thức quý báu mà chúng tôi tiếp thu được, đã giúp chúng tôi có nên tảng để thực hiện đồ án này.

Xin chân thành cảm ơn công ty FPT Software Đà Nẵng đã tạo điều kiện chúng chúng tôi có môi trường để học tập và làm việc tại quý công ty trong khoãng thời gian 4 tháng vừa qua để thực hiện đồ án này. Quý công ty đã cho chúng tôi, môi trường học tập và làm việc cũng như hỗ trợ rất nhiều cho chúng tôi trong thời gian qua.

Chúng tôi xin chân thành cảm ơn TS. Bùi Thị Minh Tú và thầy Hồ Viết Việt – giảng viên hướng dẫn tại trường Đại Học Bách Khoa Đà Nẵng đã quan tâm tạo điều kiện tốt cho chúng tôi hoàn thành đồ án. Chúng tôi xin chân thành cảm ơn anh Nguyễn Văn Thành – Kỹ sư phần mềm tại FPT Software Đà Nẵng đã giành thơi gian và tận tình giúp đỡ chúng tôi trong quá trình thực hiện đề tài này!

Xin chân thành cảm ơn !!!!

**LỜI CAM ĐOAN**

**Kính gửi:**

Hội đồng bảo vệ đồ án tốt nghiệp khoa Điện tử - Viễn thông, Trường Đại học Bách Khoa-Đại học Đà Nẵng

Chúng tôi gồm:

1. Hồ Thăng Huy – Lớp 12DT3
2. Đặng Trung Nguyên – Lớp 12DT2
3. Huỳnh Vũ Tiến – Lớp 12DT4

Hiện đang là sinh viên Khoa Điện tử -Viễn thông, Trường Đại học Bách Khoa –Đại học Đà Nẵng.

Chúng tôi xin cam đoan nội dung của Đồ án này không phải là bản sao chép của bất cứ Đồ án hoặc Công trình đã có từ trước. Nếu vi phạm chúng tôi xin chịu mọi hình thức kỷ luật của Khoa.

Đà Nẵng, ngày , tháng 05, năm 2017

Sinh viên thực hiện Sinh viên thực hiện Sinh viên thực hiện

Hồ Thăng Huy Huỳnh Vũ Tiến Đặng Trung Nguyên

# 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 BUS ). Chính vì vậy, việc xây dựng nên một phần mềm có thể mô phỏng hoạt động như một hệ thống CAN bus 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 tôi 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 “THIẾT KẾ VÀ THỰC THI DRIVER VÀ GIAO THỨC CHO HỆ THỐNG MÔ PHỎNG CAN BUS”. Phạm vi đề tài đề cập đến việc kiểm tra và sử dụng driver USB CDC-ACM trên hệ điều hành linux để kết nối truyền nhận dữ liệu đến kit TM4C123. Và thực hiện thiết kế thi công một số giao thức quan trọng trong hệ thống mô phỏng mạng CAN . Theo đó, đồ án sẽ được tổ chức thành 3 chương như sau:

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

Chương 2: Cơ sở lý thuyết

Chương 3: Kết quả thực hiện và đánh giá

MỤC LỤC

[LỜI MỞ ĐẦU 1](#_Toc482881600)

[MỤC LỤC 2](#_Toc482881601)

[DANH MỤC HÌNH 5](#_Toc482881602)

[DANH MỤC CÁC CỤM TỪ VIẾT TẮT 7](#_Toc482881603)

[**CHƯƠNG 1 : TỔNG QUAN VỀ HỆ THỐNG MÔ PHỎNG CAN BUS** 8](#_Toc482881604)

[**1.1.** **Giới thiệu chương** 8](#_Toc482881605)

[**1.2.** **Mô hình giao tiếp của hệ thống** 8](#_Toc482881606)

[**1.3.** **Phần mềm CAN** 9](#_Toc482881607)

[1.3.1. Phần mềm cơ sở dữ liệu CAN 9](#_Toc482881608)

[1.3.2. Phần mềm mô phỏng CAN 10](#_Toc482881609)

[**1.4.** **CAN Firmware** 10](#_Toc482881610)

[1.4.1. Linux USB driver 10](#_Toc482881611)

[1.4.2. Giao thức truyền dữ liệu (Communication Protocol) 12](#_Toc482881612)

[**1.5.** **Phần thiết bị sử dụng trong dự án:** 12](#_Toc482881613)

[**1.6.** **Kết luận chương** 13](#_Toc482881614)

[**CHƯƠNG 2 : CƠ SỞ LÝ THUYẾT VÀ MÔI TRƯƠNG THỰC HIỆN** 14](#_Toc482881615)

[**2.1.** **Giới thiệu chương** 14](#_Toc482881616)

[**2.2.** **Môi trường hệ điều hành Linux** 14](#_Toc482881617)

[**2.3.** **Qt framework:** 15](#_Toc482881618)

[**2.4.** **Lý thuyết Driver của thiết bị trên hệ điều hành Linux:** 17](#_Toc482881619)

[2.4.1. Nhiệm vụ của Driver của thiết bị: 17](#_Toc482881620)

[2.4.2. Mô hình phân lớp theo chiều học 18](#_Toc482881621)

[2.4.3. Các lớp Device và mô đun 19](#_Toc482881622)

[2.4.4. Character Driver 20](#_Toc482881623)

[2.4.4.1. Giới thiệu Character Driver 20](#_Toc482881624)

[2.4.4.2. Số hiệu file thiết bị 20](#_Toc482881625)

[2.4.4.3. Phương thức mở và đóng 22](#_Toc482881626)

[2.4.4.4. Phương thức đọc và ghi 22](#_Toc482881627)

[2.4.5. TTY Driver 22](#_Toc482881628)

[2.4.5.1. Hàm mở 23](#_Toc482881629)

[2.4.5.2. Hàm đóng 23](#_Toc482881630)

[2.4.5.3. Hàm ghi 23](#_Toc482881631)

[2.4.6. USB Driver 23](#_Toc482881632)

[2.4.6.1. Quá trình nhận dạng thiết bị USB trên Linux 23](#_Toc482881633)

[2.4.6.2. Giải mã thông tin về thiết bị USB 25](#_Toc482881634)

[2.4.6.3. Điểm cuối USB (USB endpoint) 26](#_Toc482881635)

[2.4.6.4. Giao diện USB (USB interface) 27](#_Toc482881636)

[2.4.6.5. Cấu hình USB (USB configuration) 27](#_Toc482881637)

[**2.5.** **Cơ sở lý thuyết về giao thức CAN** 28](#_Toc482881638)

[2.5.1. Tổng quan về giao thức CAN 28](#_Toc482881639)

[2.5.2. Standard CAN – Extended CAN 29](#_Toc482881640)

[2.5.3. Cơ chế trọng tài (Arbitration) 30](#_Toc482881641)

[**2.6.** **Kết luận chương** 33](#_Toc482881642)

[**CHƯƠNG 3 : THỰC HIỆN VÀ ĐÁNH GIÁ KẾT QUẢ** 34](#_Toc482881643)

[**3.1.** **Giới thiệu chương:** 34](#_Toc482881644)

[**3.2.** **USB Driver** 34](#_Toc482881645)

[3.2.1. Tìm hiểu về thiết bị USB muốn giao tiếp 34](#_Toc482881646)

[3.2.2. Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver 37](#_Toc482881647)

[3.2.3. Đăng ký và hủy đăng ký USB Driver 37](#_Toc482881648)

[3.2.4. Hàm thăm dò thiết bị (probe) 38](#_Toc482881649)

[3.2.5. Hàm ngắt kết nối thiết bị 39](#_Toc482881650)

[3.2.6. Hàm mở/đọc/ghi thiết bị 39](#_Toc482881651)

[3.2.7. Cài đặt USB Driver 40](#_Toc482881652)

[**3.3.** **Giao thức truyền dữ liệu** 42](#_Toc482881653)

[3.3.1. Khung truyền cơ bản 42](#_Toc482881654)

[3.3.2. Khung truyền USB: 44](#_Toc482881655)

[3.3.3. Các struct sử dụng khi thực thi các trường hợp trên: 46](#_Toc482881656)

[3.3.4. Biểu đồ tuần tự: 47](#_Toc482881657)

[3.3.4.1 Tiến trình kiểm tra hoạt động thiết bị 47](#_Toc482881658)

[3.3.4.2 Tiến trình lấy cấu hình thiết bị 47](#_Toc482881659)

[3.3.4.3 Tiến trình mô phỏng 49](#_Toc482881660)

[**3.4.** **Hoạt động của thiết bị** 50](#_Toc482881661)

[3.4.1. Kiểm tra kết nối (CheckAlive) 50](#_Toc482881662)

[3.4.2. Cấu hình cho thiết bị (Configuration) 50](#_Toc482881663)

[3.4.3. Mô phỏng 51](#_Toc482881664)

[**3.5.** **Kết quả thực hiện** 51](#_Toc482881665)

[**3.6.** **Đánh giá và hướng phát triển đề tài:** 55](#_Toc482881666)

[**3.7.** **Kết luận chương** 56](#_Toc482881667)

[KẾT LUẬN 57](#_Toc482881668)

[TÀI LIỆU THAM KHẢO 58](#_Toc482881669)

# DANH MỤC HÌNH

[Hình 1. 1 Mô hình giao tiếp hệ thống 8](#_Toc482862266)

[Hình 1. 2 : Sơ đồ luồng dữ liệu trong dự án Thiết kế hệ thống mô phỏng CAN bus 9](#_Toc482862267)

[Hình 1. 3 Mô hình phần mềm CAN 9](#_Toc482862268)

[Hình 1. 4 Tương tác giữa thiết bị và Driver 11](#_Toc482862269)

[Hình 2. 1 Kiến trúc hệ điều hành Linux. 13](#_Toc482862362)

[Hình 2. 2 : Sơ đồ các khối trong Qt Framework 14](#_Toc482862363)

[Hình 2. 3 Các thành phần của Driver trên Linux 16](#_Toc482862364)

[Hình 2. 4 Kiến trúc tổng quan Linux Kernel 17](#_Toc482862365)

[Hình 2. 5 Tổng quan về character driver trên Linux 18](#_Toc482862366)

[Hình 2. 6 Danh sách thiết bị trong thư mục /dev 19](#_Toc482862367)

[Hình 2. 7 Mô hình biểu diễn tổng quan thiết bị USB 21](#_Toc482862368)

[Hình 2. 8 Hệ thống USB trên Linux 22](#_Toc482862369)

[Hình 2. 9 Xem thông tin các thiết bị USB sử dụng lệnh lsusb 22](#_Toc482862370)

[Hình 2. 10 Tổng quan về hàng đợi 24](#_Toc482862371)

[Hình 2. 11 Tiến trình thêm phần tử vào Queue 25](#_Toc482862372)

[Hình 2. 12 Tiến trình xóa phần tử đầu Queue 25](#_Toc482862373)

[Hình 2. 13 Tiến trình thêm phần tử vào cuối Queue 25](#_Toc482862374)

[Hình 2. 14 Tiến trình xóa phần tử đầu Queue 26](#_Toc482862375)

[Hình 2. 15 Sơ đồ minh họa một ngăn xếp và các hoạt động diễn ra 26](#_Toc482862376)

[Hình 2. 16 Tiến trình thêm phần tử vào Stack 27](#_Toc482862377)

[Hình 2. 17 Tiến trình lấy phần tử từ Stack 27](#_Toc482862378)

[Hình 2. 18. Cấu trúc của bộ đệm vòng 28](#_Toc482862379)

[Hình 2. 19 Thực hiện bộ đệm tuyến tính của bộ đệm vòng 28](#_Toc482862380)

[Hình 2. 20 Tiến trình xử lý trường hợp sử dụng bộ đệm 28](#_Toc482862381)

[Hình 2. 21 :Sơ đồ kết nối của một Bus CAN 29](#_Toc482862382)

[Hình 2. 22: Khung truyền của Standard CAN 30](#_Toc482862383)

[Hình 2. 23 :Extended CAN Frame 31](#_Toc482862384)

[Hình 2. 24 : Phân định trạng thái High và Low trong Bus 31](#_Toc482862385)

[Hình 2. 25 : Cơ chế trọng tài giải quyết xung đột trong CAN Bus 32](#_Toc482862386)

[Hình 2. 26 :Kênh nhận và truyền nhận dạng trạng thái trên Bus 33](#_Toc482862387)

[Hình 3. 1 Quy trình viết USB Driver 30](#_Toc482876508)

[Hình 3. 2 Các thiết bị USB đang kết nối với máy tính 31](#_Toc482876509)

[Hình 3. 3 Thông tin chi tiết về cấu hình USB thiết bị TM4C123G 33](#_Toc482876510)

[Hình 3. 4 Biên dịch bằng sudo make để build driver 36](#_Toc482876511)

[Hình 3. 5 : sử dụng lệnh insmod để cài đặt driver 37](#_Toc482876512)

[Hình 3. 6 sử dụng rmmod để gỡ cài đặt driver và kiểm tra 37](#_Toc482876513)

[Hình 3. 7: Kiểm tra đọc ghi qua file definition 38](#_Toc482876514)

[Hình 3. 8 khung truyền dữ liệu cơ bản 38](#_Toc482876515)

[Hình 3. 9 Bảng giá trị chứa ID của nguồn và đích truyền dữ liệu 39](#_Toc482876516)

[Hình 3. 10 : các loại message và giá trị ID message dung trong giao thức 39](#_Toc482876517)

[Hình 3. 11 Khung truyền dữ liệu chung trên USB bus 40](#_Toc482876518)

[Hình 3. 12 Khung truyền dữ liệu cho trường hợp mô phỏng (simulation) 40](#_Toc482876519)

[Hình 3. 13 Khung truyền dữ liệu cho trường hợp check\_alive 41](#_Toc482876520)

[Hình 3. 14: Khung truyền dữ liệu cho trường hợp nhận và cài đặt cấu hình 41](#_Toc482876521)

[Hình 3. 15: Bảng các struct được dung trong giao thức 42](#_Toc482876522)

[Hình 3. 16: Sơ đồ thiết kế cho quá trình kiểm tra hoạt động thiết bị 42](#_Toc482876523)

[Hình 3. 17: Sơ đồ thiết kế cho việc lấy cấu hình bus CAN 43](#_Toc482876524)

[Hình 3. 18 : Sơ đồ thực hiện quá trình mô phỏng 44](#_Toc482876525)

[Hình 3. 19 Sơ đồ sequence cho nhiệm vụ CheckAlive 45](#_Toc482876526)

[Hình 3. 20 Sơ đồ sequence cho nhiệm vụ cấu hình 45](#_Toc482876527)

[Hình 3. 21 Sơ đồ sequence mô tả hoạt động mô phỏng 46](#_Toc482876528)

[Hình 3. 22: Kết quả mô phỏng quá trình check\_alive thành công 47](#_Toc482876529)

[Hình 3. 23 Kết quả mô phỏng check\_alive thiết bị được ngắt khỏi CAN 48](#_Toc482876530)

[Hình 3. 24 : Kết quá quá trình lấy cấu hình bus CAN 49](#_Toc482876531)

[Hình 3. 25 : kết quả quá trình mô phỏng với dữ liệu nhận từ bus CAN 50](#_Toc482876532)

[Hình 3. 26: Kết quả mô phỏng với dữ liệu gửi từ PC đến bus CAN 51](#_Toc482876533)

# DANH MỤC CÁC CỤM TỪ VIẾT TẮT

ACM : Abstract Control Model

CDC : Communication Device Class

CAN : Controller Area Network

ECU : Electronic Control Unit

EV : Enviroment Variable

ID : Identifier

IDE : Intergrated Drive Electronics

IG : Interactor Generator

USB : Universal Serial Bus

PCI : Peripheral Component Interconnect

SPI : Serial Peripheral Interface

I2C : Inter-Integrated Circuit

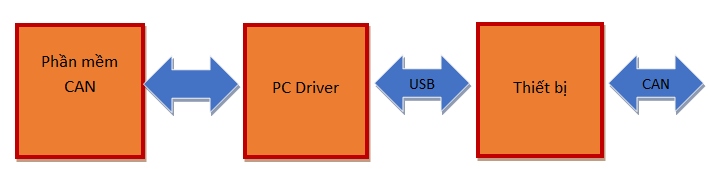
PC : Personal Computer

# CHƯƠNG 1 : TỔNG QUAN VỀ HỆ THỐNG 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**



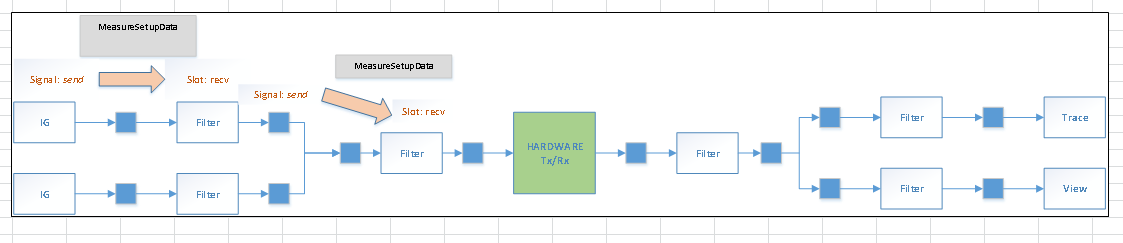
Hình 1. 1 Mô hình giao tiếp hệ thống

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 dựa 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 thiết bị và phần mềm mô phỏng 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.

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.

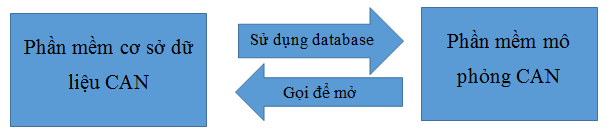


Hình 1. 2 : Sơ đồ luồng dữ liệu trong dự án Thiết kế hệ thống mô phỏng CAN bus

Tín hiệu được truyền đi từ 1 phần mềm mô phỏng CAN trên PC qua các bộ lọc mềm trên PC để chọn các thông điệp (message) cần thiết để truyền đến thiết dựa trên các giao thức tự thiết kế. Các thông điệp được yêu cầu sẽ gửi đi ra cổng CAN của thiết bị đến một thiết bị khác trên mạng CAN . Việc thực hiện thành công tác vụ truyền nhận thông điệp, tại cửa sổ trace sẽ hiện lên các thông điệp đã được gửi đi ra cổng CAN của thiết bị hoặc nhận được từ ngoài cổng CAN để người dùng có thể giám sát

* 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 định dạng Little Endian và Big Endian , MSB và LSB.

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

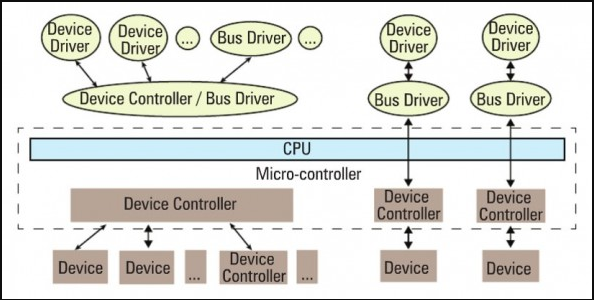
Vai trò chính của phần mềm cơ sở dữ liệu CAN là tạo dựng một cơ sở dữ liệu gồm các node network và các thông điệp gửi đi cũng như nhận về từ chính node đó để cung cấp dữ liệu cho việc mô phỏng ở phần mềm mô phỏng.

* + 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 kiểm tra.
* 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.
* Cửa sổ trace (Trace window) hiển thị các thuộc tính của message như: 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 số 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 nốt thiết bị (device node).
* Phần mềm CAN có thể cấu hình tốc độ Baurd 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 là tập các ngữ cảnh để thực hiện việc truyền nhận dữ liệu khi phần mềm được khởi động, giao thức trong phạm vi đề tài này chỉ đề cập đến các ngữ cảnh đơn giản như : kiểm tra hoạt động kit TM4C123G khi kết nối với CAN bus, truyền nhận dữ liệu chứa cấu hình thiết bị, thực hiện truyện nhận dữ liệu CAN frame khi phần mềm thực hiện chức năng mô phỏng để giám sát dữ liệu từ CAN bus. 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 phải 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.

* 1. **Phần thiết bị sử dụng trong dự án:**

Thiết bị được sử dụng trong đề tài này là kit TM4C123G , thiết bị này sử dụng cổng CAN để kết nối với trên đường truyền CAN bus . Công việc chính của CAN Device là tạo một node CAN có thể tùy chỉnh được tốc độ của node để có thể kết nối được với nhiều Bus CAN khác nhau. Node có thể kiểm tra được kết nối giữa CAN Device và Bus, gửi và nhận dữ liệu từ Bus. Để thực hiện việc này chúng ta sử dụng 2 giao thức là CAN và USB. Nhiệm vụ chính của thiết bị trong hai giao thức trên là :

* Giao tiếp USB nhận và gởi message để giao tiếp giữa board và PC
* Giao Tiếp CAN để nhận CAN Frame từ CAN Bus và gởi dữ liệu thu được từ CAN Frame lên PC.
* Kiểm tra thiết bị đã được kết nối với CAN Bus hay chưa, trả kết quả lại cho PC.
* Cấu hình được tốc độ của thiết bị để có thể kết nối được với Bus CAN bất kỳ.
  1. **Kết luận chương**

Qua chương 1, chúng ta đã có được cái nhìn tổng quát về các phần trong dự án “thiết kế và thi công hệ thống mô phỏng CAN bus”, hiểu được chức năng của phần mềm mô phỏng CAN trong việc giám sát tín hiệu trên CAN bus. Chương này còn giúp chúng ta nắm bắt được quy trình thiết kế một hệ thống mô phỏng mạng CAN gồm ba phần: phần mềm mô phỏng giúp người dung dễ quan sát hoạt động các tín hiệu truyền trên mạng CAN, phần các quy tắc giao thức truyền dữ liệu giúp định hình quá trình truyền dữ liệu giữa PC và kit TM4C123G tạo nên các khung truyền dữ liệu với các ngữ cảnh khác nhau, phần thiết bị được kết nối trên mạng CAN tạo nên cầu nối cho việc truyền dữ liệu đi trên bus CAN và ngược lại. Qua đó giúp ta nắm bắt được sự cần thiết của hệ thống mô phỏng mạng CAN bus trong ngành công nghệ hiện nay

# CHƯƠNG 2 : CƠ SỞ LÝ THUYẾT VÀ MÔI TRƯƠNG THỰC HIỆN

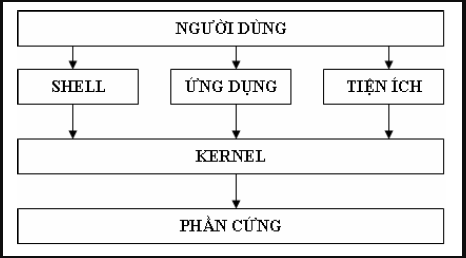
* 1. **Giới thiệu chương**

Linux là hệ điều hành có thể quản lý bộ nhớ rất tốt, do đó 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)

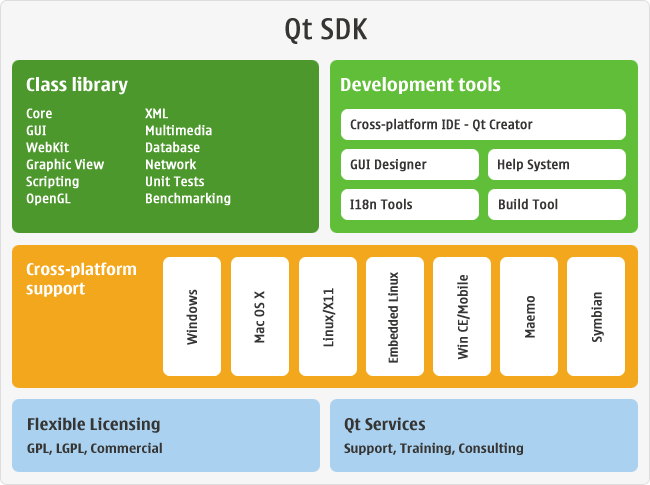
Là trung tâm điều khiển của hệ điều hành Linux , chứa các mã nguồn điều khiển hoạt động của toàn bộ hệ thống. Kernel được thiết kế theo module, do vậy kích thước rất nhỏ. Kernel chỉ tải bộ phận cần thiết lên bộ nhớ , các bộ phận khác được tải lên nếu có yêu cầu sử dụng. Nhờ vậy , so với các hệ điều hành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



Hình 2. 2 : Sơ đồ các khối trong Qt Framework

Qt là một framework thích hợp cho việc viết các ứng dụng chạy trên các nền tảng khác nhau, với các phần cứng khác nhau. Qt hỗ trợ nhiều thư viện có tính linh động cao, cho các mục đích khác nhau như : QSerialPort, QTimer, QObject,…

Trong đồ án này, chúng ta đề cập đến một số lớp (class) hay dùng trong Qt như:

* Lớp QSerialPort: Đây là thư viện hỗ trợ các hàm giúp kết nối với phần cứng thông qua đường truyền nối tiếp (serial). QSerialPort cung cấp một số hàm quan trọng như :

+ *setPortName(const QString &name):*

giúp mở cổng kết nối với phần cứng, ví dụ: trên nền tảng linux, với driver USB CDC, sử dụng setPort(“/dev/ttyACM”) với ttyACM là tệp mô tả thiết bị được tạo ra khi quá trình driver tạo ra khi phần cứng được kết nối vào PC. Trên hệ điều hành windown, sử dụng setPort(“COM”), với COM là cổng kết nối thiết bị trên windown

+ *setParity(QSerialPort::NoParity):*

Cài đặt bit parity để kiểm tra tại cổng serial. Nếu cài đặt thành công, sẽ return giá trị về true, hoặc ngược lại sẽ return về fail

+ *setBaudRate(QSerialPort::Baud115200, QSerialPort::AllDirections)*

Cài đặt tốc độ Baud giữa PC và thiết bị

+ *setStopBits(QSerialPort::OneStop)*

Cài đặt bit dừng, mỗi khi truyền dữ liệu

*+ open(QIODevice::ReadWrite)*

Cài đặt kiểu phương thức tác động lên phần cứng kết nối với PC, nếu sử dụng mode ReadWrite, tương ứng với khả năng có thể ghi và đọc lên thiết bị.

Nếu sử dụng mode Read hoặc Write, chỉ sự dụng được một trong hai phương thức, chỉ đọc hoặc chỉ ghi lên thiết bị được kết nối với PC

* Lớp QTimer : Đây là lớp quản lý định thời cho các chức năng được sử dụng trên các lớp được khởi tạo theo kiểu QTimer hoặc thừa kế từ lớp QTimer.

QTimer được sử dụng trong đồ án này như một ngắt trong quá trình gửi dữ liệu từ PC xuống thiết bị . Một số phương thức phổ biến được sử dụng trong lơp QTimer như sau:

+ *void start (std::chrono::milliseconds msec)*

Khởi tạo một timer, với giá trị được truyền vào theo đơn vị mili giây. Ví dụ start(1000) : khởi tạo một timer với ngắt một giây, sau một giây, sẽ phát ra một tín hiệu timeout() , để thông báo kết thúc một chu kỳ 1s, bắt đầu một chu kỳ mới.

+ *void start()*

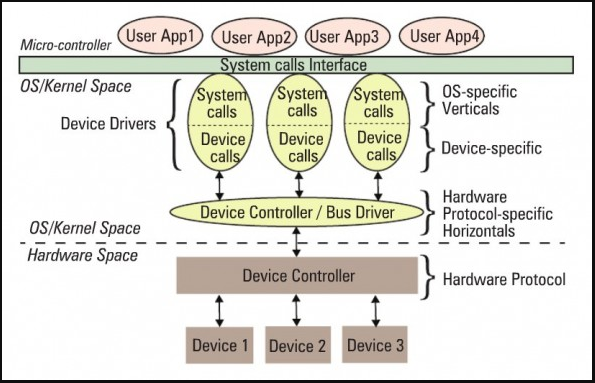
Khởi tạo một định thời không có tham số truyền vào, giá trị mặc địn NULL.

+ *void stop()*

Dừng một đối tượng QTimer.

* Lớp QObject : Đây là lớp cha của mọi lớp trong Qt Framework. Việc khởi tạo một đối tượng thừa kế lớp QObject giúp ta sử dụng được các phương thức connect (Object1, signal1,Object2,slot2) . Phương thức connect() giúp kết nối Signals và Slots giữa các đối tượng, khi đối tượng Object1 phát đi signal1, đối tượng Object2 sẽ bắt được slot2 để để thực thi. Việc phát đi signal và bắt lại slot ở hai đối tượng khác nhau là đặc trưng rất quan trọng trên nền tảng Qt (Qt framework).
  1. **Lý thuyết Driver của thiết bị trên hệ điều hành Linux:**
     1. Nhiệm vụ của Driver của thiết bị:

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).



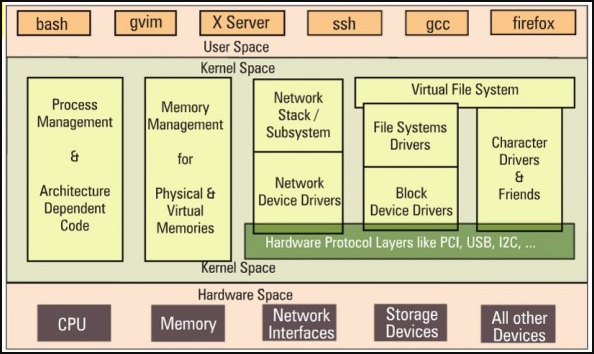
Hình 2. 3 Các thành phần của Driver trên Linux

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.



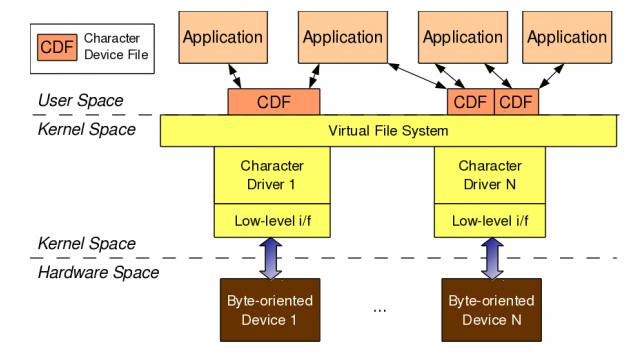
Hình 2. 4 Kiến trúc tổng quan Linux Kernel

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



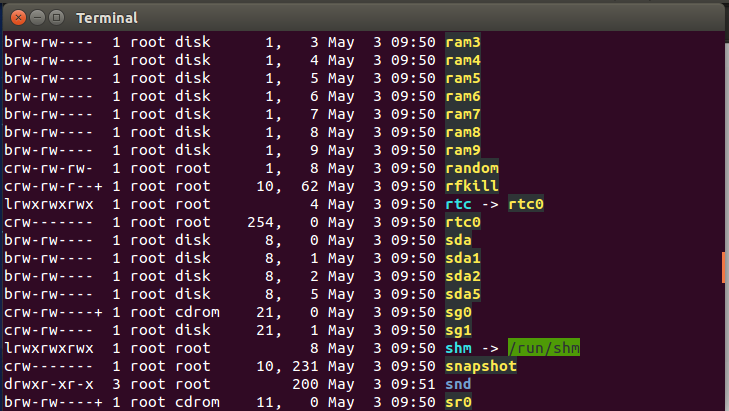
Hình 2. 5 Tổng quan về character driver trên Linux

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. 6 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

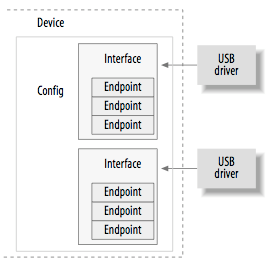
Hàm đóng được gọi bởi lõi TTY khi lệnh đóng được gọi bởi người dùng trên các tập tin xử lý đã được tạo ra trước đó với hàm mở.

* + - 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.

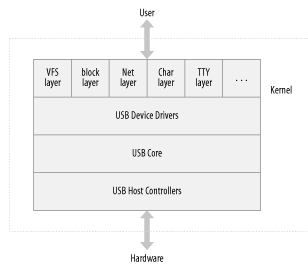
* + 1. USB Driver
       1. Quá trình nhận dạng thiết bị USB trên Linux



Hình 2. 7 Mô hình biểu diễn tổng quan thiết bị USB

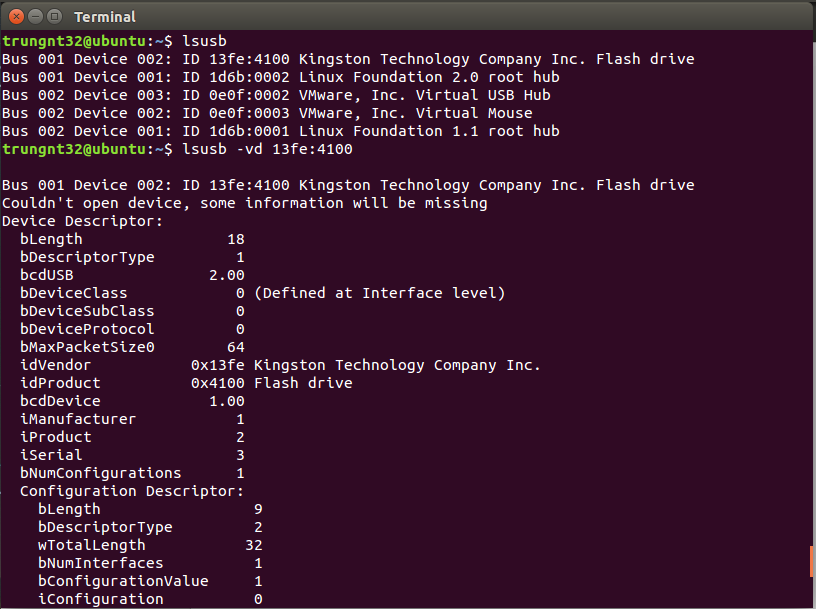
Khi có một thiết bị usb hợp lệ được cắm vào hệ thống Linux, cho dù nó có driver hay không thì nó cũng vẫn được nhận diện (detect) bởi phần cứng ở tầng nhân (kernel space) của hệ thống Linux mà đã được hỗ trợ giao thức usb. Hệ thống có thể làm điều này là bởi vì khả năng của chính bản thân giao thức usb 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. 8 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. 9 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.

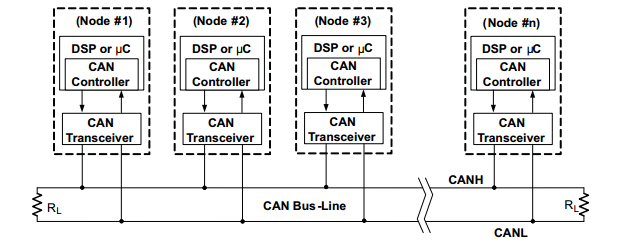
* + - 1. Cấu hình USB (USB configuration)

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. **Cơ sở lý thuyết về giao thức CAN**
     1. Tổng quan về giao thức CAN

Controller Area Network (CAN) là một giao thức truyền thông bus phát triển bởi Bosch (một nhà sản xuất thiết bị điện tại Đức) vào đầu những năm 1980. Sau đó, CAN đã được chuẩn hóa theo tiêu chuẩn ISO-11898 ,ISO-11519 và nó được sử dụng như là giao thức chuẩn để giao tiếp kết nối mạng trong ngành công nghiệp ôtô.

CAN được tổ chức International Standization Organization (ISO) định nghĩa là một chuẩn truyền thông chuyên cho ngành công nghiệp Ô tô, nhằm thay thế cho việc đi dây phức tạp bằng Bus chỉ có hai dây và các thiết bị trên bus giao tiếp với nhau mà không cần host computer.



Hình 2. 21 :Sơ đồ kết nối của một Bus CAN

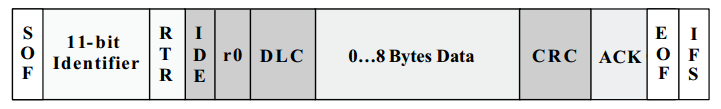
CAN được thiết kế là một mạng đa chủ, mỗi thiết bị đều có thể gởi và nhận tín hiệu. Không giống như các loại mạng truyền thống, CAN không gởi một gói tin lớn từ điểm này đến điểm kia mà nó chia mà nó gởi nhiều gói tin nhỏ theo dạng broadcast đến tất cả các Node trong Bus. Các Node trên Bus bắt các gói tin và dựa và trường ID trong gói tin nhận được để xử lý hay loại bỏ gói tin.

Khi mạng Ethernet phát hiện một vụ xung đột, các nút đang gửi dữ liệu chỉ đơn giản là dừng truyền và đợi một khoảng thời gian ngẫu nhiên trước khi cố gắng để gửi lại một lần nữa. Tuy nhiên, đối với giao thức CAN có một chúc khác biệt đó là các vấn đề xung đột được giải quyết bằng cách sử dụng các nguyên tắc của trọng tài, nơi chỉ có các node có mức độ ưu tiên cao nhất được trao quyền để gửi dữ liệu của nó

* + 1. Standard CAN – Extended CAN

Hiện này chúng ta có hai chuẩn CAN là Standard CAN và Extended CAN, với Externed CAN là chuẩn CAN phiên bản cải tiến của Standard CAN.

* + - 1. Standard CAN

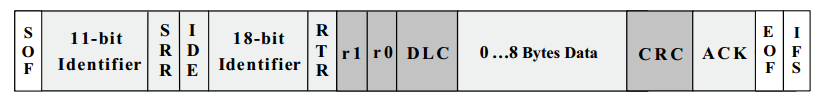


Hình 2. 22: Khung truyền của Standard CAN

*Bảng 2.1: Bảng ký hiệu các trường của Standard CAN*

|  |  |  |
| --- | --- | --- |
| Tên | Độ dài | Mục đích |
| SOF | 1 bit | Bit bắt đầu của khung truyền |
| Identifier(ID) | 11 bit | Chứa ID của message |
| RTR | 1 bit | Phân biệt data frame hay remote frame |
| IDE | 1 | Phân biệt loại extended hay standard |
| R0 | 1 | Bit dành riêng.(chỉ dành để sủa đổi tiêu chuẩn trong tương lai) |
| DLC | 4 | Độ dài của trường data đơn vị (byte) |
| Data | 0-64 bit | Dữ liệu của message |
| CRC | 16 bit | Mã kiểu tra lỗi CRC |
| ACK | 2 bit | “Every node receiving an accurate message overwrites this recessive bit in the original message with a dominate bit, indicating an error-free message has been sent. Should a receiving node detect an error and leave this bit recessive, it discards the message and the sending node repeats the message after rearbitration. In this way, each node acknowledges (ACK) the integrity of its data. ACK is 2 bits, one is the acknowledgment bit and the second is a delimiter” ([1]-trang số 3) |
| EOF | 1 bit | Kết thúc của Frame |
| IFS | 7 bit | Interframe Space (chứa thời gian yêu cầu từ controller để di chuyển khung truyền nhận được vào vị trí thích hợp trong bộ đệm ) |

* + - 1. Extended CAN



Hình 2. 23 :Extended CAN Frame

Khác biệt cơ bản nhất giữa Standard CAN và Extended CAN là trường ID 29 bit.

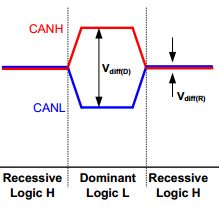
* + 1. Cơ chế trọng tài (Arbitration)
       1. Bài toán xung đột trong bus

Vì CAN là một mạng đa chủ, mỗi thiết bị trong mạng đều có thể gởi và truyền tín hiệu, nên sẽ có khả năng cùng một thời điểm sẽ có nhiều hơn một thiết bị có nhu cầu sử dụng đường truyền. Khi điều đó xảy ra thì xung đột sẽ xảy ra trên đường truyền. Vì thế chúng ta cần một cơ chế để có thể giải quyết được xung đột trên.

Để giải quyết xung đột trên, CAN sử dụng cơ chế trọng tài (Arbitration) để giải quyết vấn đề này.

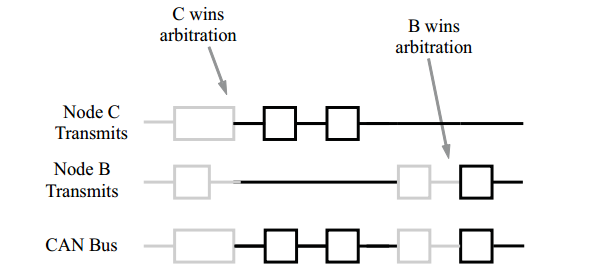
* + - 1. Cách giải quyết

Với một các mạng bình thường thì mức logic cao sẽ tương ứng với mức 1, và múc logic thấp là mức 0 – tuy nhiên điều này không chính xác trong CAN. Trong CAN, mức logic cao sẽ là Recessive và múc thấp là Dominant.



Hình 2. 24 : Phân định trạng thái High và Low trong Bus

Như đã nêu ở trên, khi có nhiều hơn một thiết bị trên Bus yêu cầu đường truyền thì trên Bus sẽ xảy ra xung đột. Vì vậy trong CAN sẽ phân cấp trường ID trong các message, việc phân cấp này sẽ giúp chúng có các ID đươc ưu tiên trong Bus. Nếu hai thiết bị cùng truyền đi hai message thì message có trường ID với mức ưu tiên cao hơn sẽ được gởi truyền đi trong Bus.

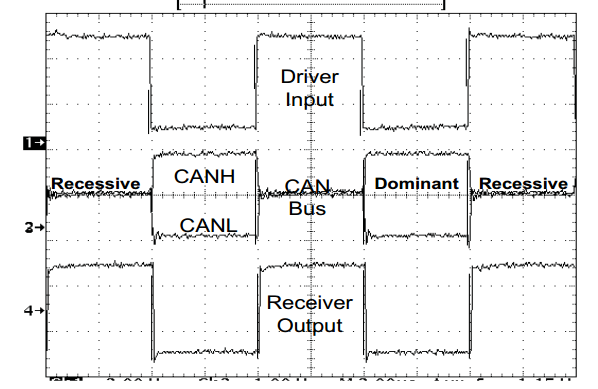


Hình 2. 25 : Cơ chế trọng tài giải quyết xung đột trong CAN Bus

Chính nhờ sự ưu tiên khác nhau của các message mà chúng ta có thể gởi nhiều message trong cùng một lúc tuy nhiên chỉ có message có múc ưu tiên cao nhất là được truyền trong Bus. Vì vậy trong một Bus, các ID có mức ưu tiên cao sẽ được để dành để đánh số ID cho các message có yêu cầu quan trọng trọng hệ thống, ví dụ như hệ phanh, ga, túi khí, …. Còn các hệ thống có độ ưu tiên thấp như hệ thống đa phương tiện, điều hòa, … sẽ được nhận mức ưu tiên cho ID thấp hơn các hệ thống quan trọng khác.

* + - 1. Nhận dạng dữ liệu trong bus

Trong hoạt động của CAN Bus thì một lúc chỉ có một node được gởi dữ liệu và tất cả các node còn lại đều nhận gói dữ liệu đó. Chúng dựa vào trường ID để để nhận CAN\_Frame.



Hình 2. 26 :Kênh nhận và truyền nhận dạng trạng thái trên Bus

* 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 cũng như giao thức CAN. 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

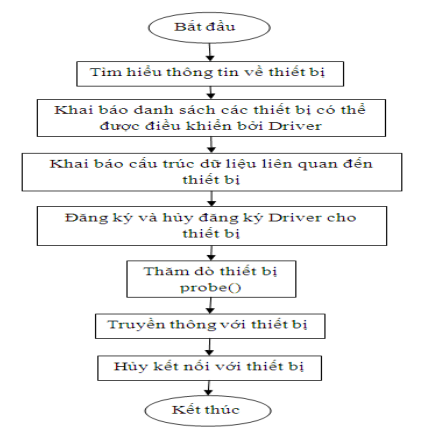
# CHƯƠNG 3 : THỰC HIỆN VÀ ĐÁNH GIÁ KẾT QUẢ

* 1. **Giới thiệu chương:**

Chương này giới thiệu về cách thức thực hiện một driver trên linux và mô tả các ngữ cảnh được sử dụng trong việc thiết kế giao thức truyền dữ liệu giữa PC và kit TM4C123G sử dụng biểu đồ tuần tự. Bên cạnh đó chương còn trình bày một số hình vẽ thể hiện kết quả tiến trình thực thi giao thức và driver

* 1. **USB Driver**

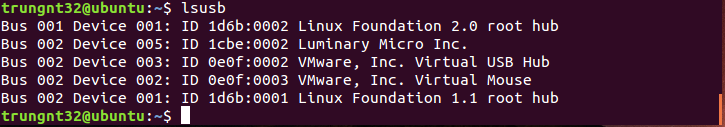
Quy trình viết USB Driver



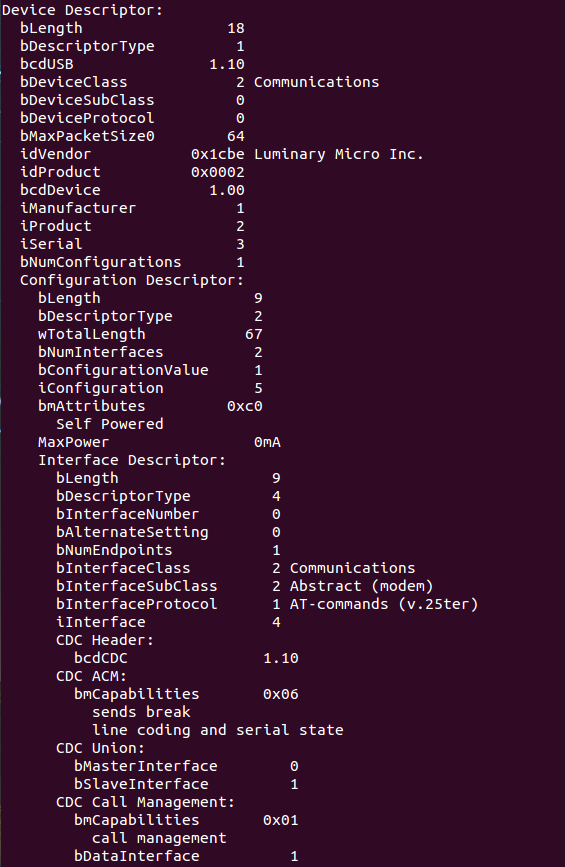
Hình 3. 1 Quy trình viết USB Driver

* + 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.



Hình 3. 2 Các thiết bị USB đang kết nối với máy tính



Hình 3. 3 Thông tin chi tiết về cấu hình USB thiết bị TM4C123G

* + 1. Khai báo danh sách các thiết bị có thể được điều khiển bởi Driver
* macro USB\_DEVICE (idVendor, idProduct) : tạo cấu trúc usb\_device\_id với idVendor và idProduct
* macro MODULE\_DEVICE\_TABLE(usb, usb\_device\_id[]) : Khai báo cấu trúc usb\_device\_id với USB Core.

static const struct usb\_device\_id acm\_ids[] = {

{USB\_DEVICE(0x1cbe,0x0002),

.driver\_info = NO\_UNION\_NORMAL,}

};

MODULE\_DEVICE\_TABLE(usb, acm\_ids);

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

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

Tạo 3 hoạt động cần có để khởi tạo struct usb\_driver

Static struct usb\_driver acm\_driver = {

.name = “cdc\_acm”,

.probe = acm\_probe,

.disconnect = acm\_disconnect,

};

Trong đó sử dụng:

* const char\* name : tên của Driver
* 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. Để hủy đăng kí một USB Device Driver ra khỏi hệ thống, ta sử dụng hàm usb\_deregister(struct usb\_driver &);

Đăng ký struct usb\_driver với USB core:

static init \_\_init acm\_init(void)

{

retval = usb\_register(&acm\_driver);

if (retval) {

tty\_unregister\_driver(acm\_tty\_driver);

put\_tty\_driver(acm\_tty\_driver);

return retval;

}

return 0;

}

Hủy đăng ký usb\_driver với USB core:

static void \_\_exit acm\_exit(void)

{

usb deregister(&acm\_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);

Khi ngắt kết nối thực hiện công việc :

* Lấy lại các cấu trúc dữ liệu hiện có và kết nối với cấu trúc usb\_interface sau đó:

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. Đây là một cấu trúc dữ liệu trong không gian hạt 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 như sau:

dev = usb\_get\_intfdata(interface);

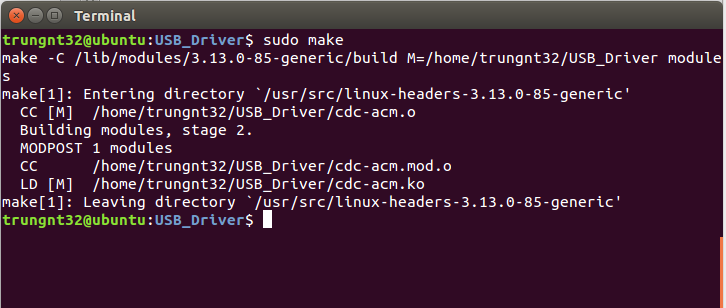
file->private\_data = dev;

Cấu trúc file được 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

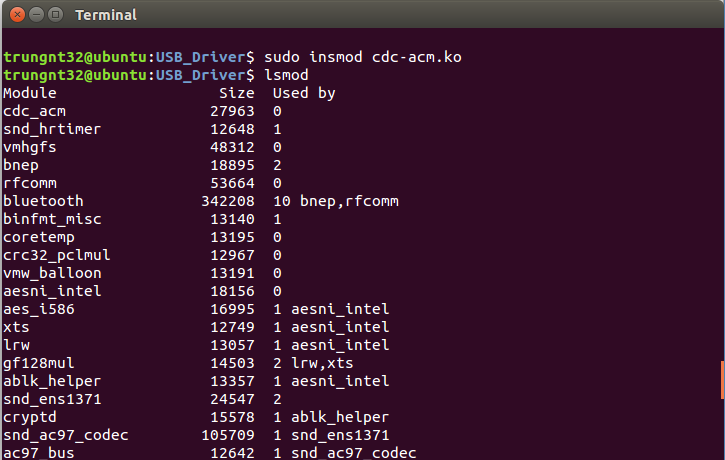
Thực hiện biên dịch driver, cài đặt, gỡ bỏ và kiểm tra driver cho hệ thống Linux:

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

**

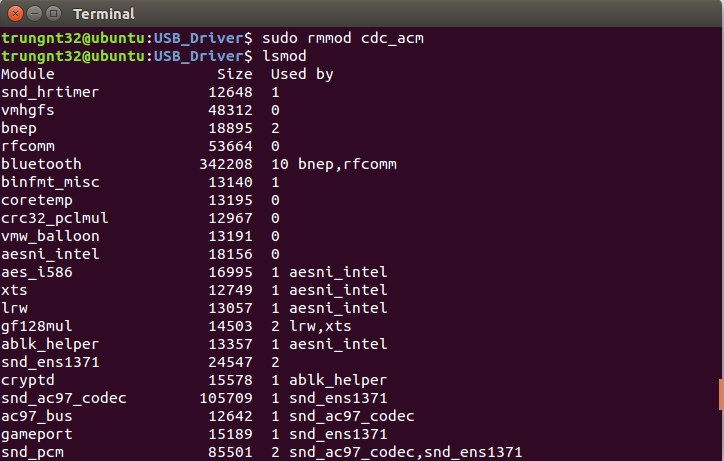
Hình 3. 4 Biên dịch bằng sudo make để build driver

* Nạp driver sử dụng insmod: sudo insmod cdc-acm.ko và kiểm tra, liệt kê tất cả mô đun trong hệ thống bằng lệnh lsmod



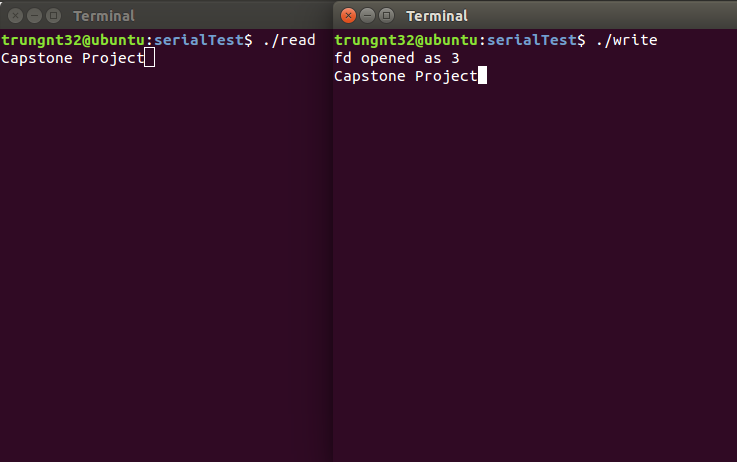
Hình 3. 5 : Sử dụng lệnh insmod để cài đặt driver

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



Hình 3. 6 Sử dụng rmmod để gỡ cài đặt driver và kiểm tra

Viết chương trình ghi và đọc thông qua file definition



Hình 3. 7: Kiểm tra đọc ghi qua file definition

* 1. **Giao thức truyền dữ liệu**

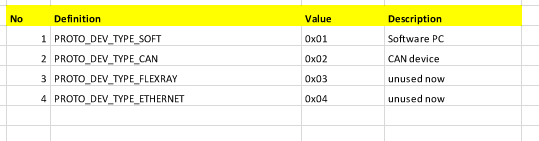
Giao thức truyền dữ liệu cho hệ thống mô phỏng mạng CAN trong đồ án này, là thiết kế các ngữ cảnh truyền dữ liệu từ PC đến thiết bị TMS123G và ngược lại. Một số ngữ cảnh được đề cập trong đồ án này bao gồm các tiến trình:

* check\_alive() kiểm tra thiết bị có khả năng truyền nhận dữ liệu ra cổng CAN .
* get\_configuration() : gửi yêu cầu từ PC xuống thiết bị để nhận về cấu hình tại cổng CAN được kết nối tại thiết bị.
* simulation() : thực hiện truyền dữ liệu từ PC xuống thiết bị và nhận về dữ liệu CAN Frame từ cổng CAN gửi đến thiết bị và thiết bị gửi lên PC
  + 1. Khung truyền cơ bản



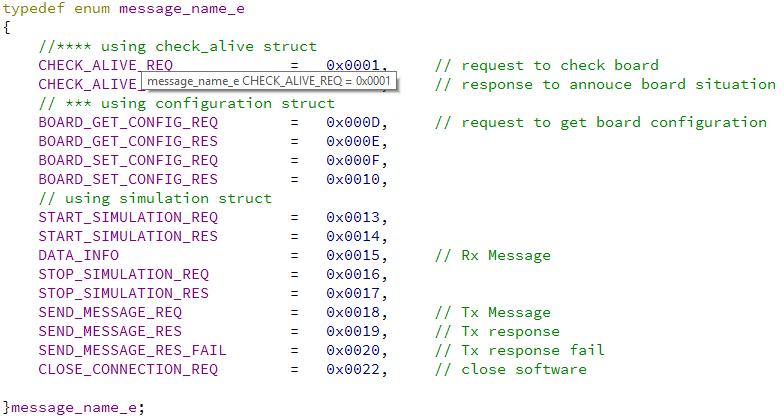
Hình 3. 8 khung truyền dữ liệu cơ bản

* Source ID : Chứa giá trị ID loại nguồn gửi đi
* Dest ID : chưa giá trị ID của loại đích nhận được



Hình 3. 9 Bảng giá trị chứa ID của nguồn và đích truyền dữ liệu

* CommandID : Chứa giá trị của ID các message được truyền đi trên đường truyền USB giữa thiết bị và PC



Hình 3. 10 : các loại message và giá trị ID message dung trong giao thức

**CHECK\_ALIVE\_REQ** : message được truyền từ PC đến thiết bị để yêu cầu kiểm tra khả năng hoạt động của board

**CHECK\_ALIVE\_RES** : message được truyền từ thiết bị đến PC để thông báo trạng thái hoạt động của thiết bị ( giá trị trong trường status = 1: thiết bị còn hoạt động, status = 0 : thiết bị không có khả năng kết nối đến cổng CAN)

**BOARD\_GET\_CONFIG\_REQ** : message được truyền từ PC đến thiết bị để yêu cầu thiết bị gửi cấu hình từ cổng CAN đến PC

**BOARD\_GET\_CONFIG\_RES** : message được truyền từ thiết bị đến PC để thông báo cấu hình CAN được kết nối với thiết bị

**BOARD\_SET\_CONFIG\_REQ** : message được truyền từ thiết bị đến PC để cài đặt cấu hình CAN được kết nối với thiết bị

**BOARD\_SET\_CONFIG\_RES** : message được truyền từ thiết bị đến PC để thông báo việc cài đặt thành công

**START\_SIMULATION\_REQ** : message được truyền từ PC đến thiết bị để thông báo bắt đầu quá trình mô phỏng

**START\_SIMULATION\_RES** : message được truyền từ thiết bị đến PC để thông báo thiết bị đã sẵn sàng cho việc mô phỏng

**DATA\_INFO** : message được truyền từ thiết bị đến PC để gửi dữ liệu được truyền từ một thiết bị khác ngoài cổng CAN đến PC

**STOP\_SIMULATION\_REQ** : message được truyền từ PC đến thiết bị để dừng quá trình mô phỏng

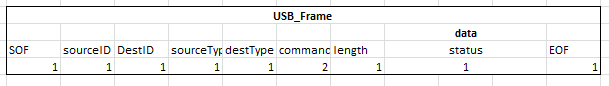
**STOP\_SIMULATION\_RES** : message được truyền từ thiết bị đến PC thông báo thiết bị đã dừng mô phỏng

**SEND\_MESSAGE\_REQ** : message được truyền từ PC đến thiết bị để gửi dữ liệu phát ra từ PC đến cổng CAN

**SEND\_MESSAGE\_RES** : message được truyền từ thiết bị đến PC để thông báo giá trị CAN Frame trong message SEND\_MESSAGE\_REQ đã được gửi ra cổng CAN thành công

**SEND\_MESSAGE\_RES\_FAIL** : message được truyền từ thiết bị đến PC để thông báo giá trị CAN Frame trong message SEND\_MESSAGE\_REQ đã được gửi ra cổng CAN thất bại

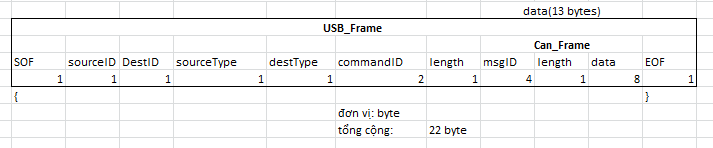
* + 1. Khung truyền USB:



Hình 3. 11 Khung truyền dữ liệu chung trên USB bus

USB\_Frame là định dạng khung struct chung được truyền trên bus USB giữa PC và thiết bị, tùy theo từng ngữ cảnh, trường data trong USB\_Frame sẽ thay đổi theo. Một số khung struct USB\_Frame riêng được đề cập như sau:

* Trường hợp dành cho mô phỏng (Simulator)



Hình 3. 12 Khung truyền dữ liệu cho trường hợp mô phỏng (simulation)

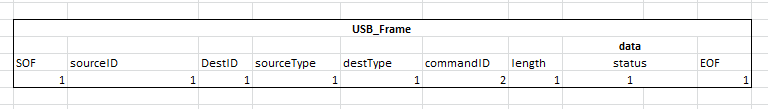
Giá trị data trong USB\_Frame trong trường hợp simulator là một CAN\_Frame bao gồm các giá trị:

msgID : chứa giá trị ID của message CAN được truyền đi trên bus CAN

Data : chứa dữ liệu trong mỗi bus CAN, giá trị của trường data là 8 bytes và thường được hiển thị theo kiểu thập lục phân tại giao diện

Length : chứa chiều dài của CAN\_Frame

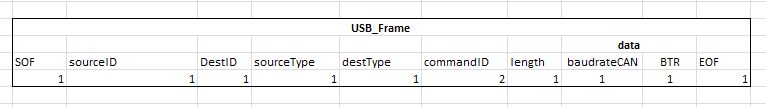
* Trường hợp dành cho kiểm tra thiết bị (Checkalive)



Hình 3. 13 Khung truyền dữ liệu cho trường hợp check\_alive

Trường data trong USB\_Frame trong trường hợp check\_alive là 1 giá trị status 1 byte, chưa giá trị 1 hoặc 0 (1 ứng với trường hợp thiết bị còn hoạt động, có khả năng gửi ra cổng CAN và 0 : ngược lại thiết bị không có khả năng truyền nhận dữ liệu từ CAN bus )

* Trường hợp dành cho cấu hình (Configuration)



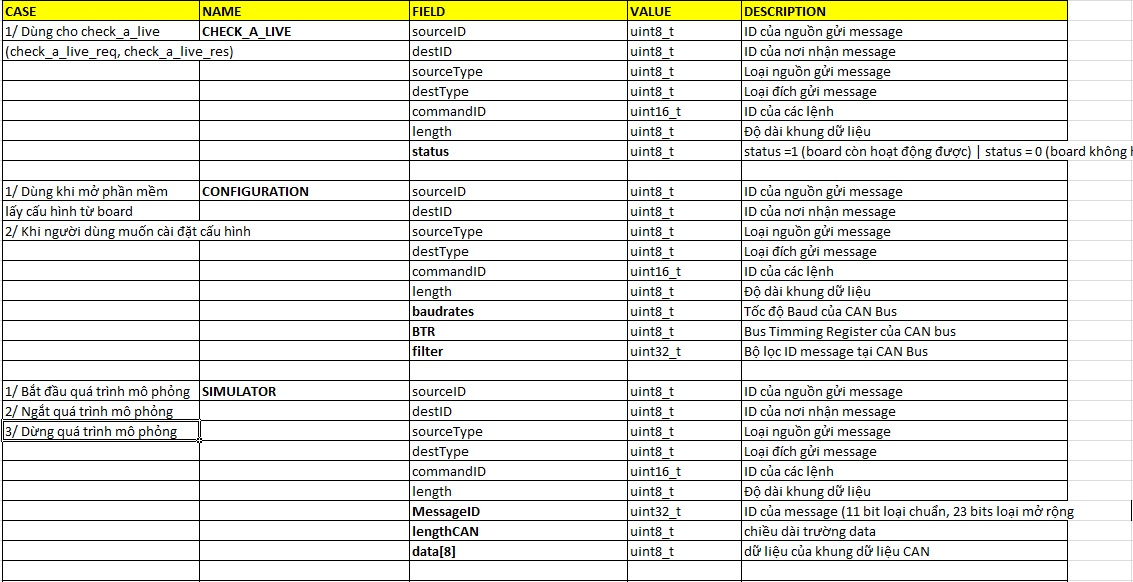
Hình 3. 14: Khung truyền dữ liệu cho trường hợp nhận và cài đặt cấu hình

Giá trị data trong USB\_Frame trong trường hợp simulator là một CAN\_Frame bao gồm các giá trị:

baudRate : chứa giá trị tốc độ Baud của message CAN được truyền đi trên bus CAN

BTR : Bus Timing Register

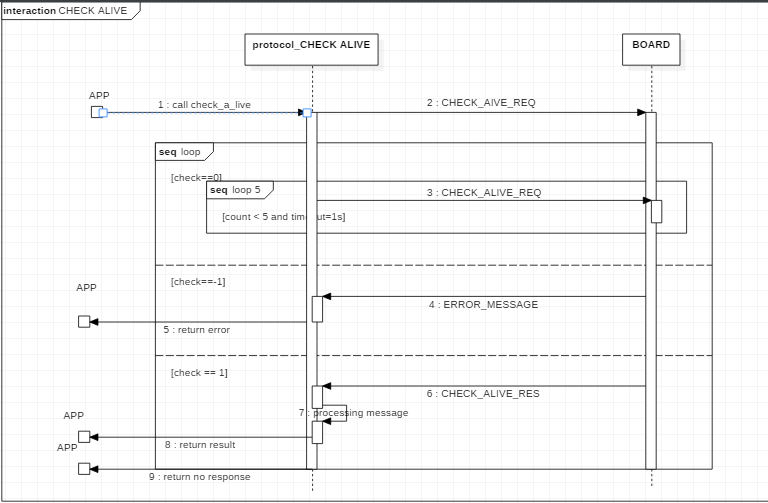
* + 1. Các struct sử dụng khi thực thi các trường hợp trên:



Hình 3. 15: Bảng các struct được dung trong giao thức

* + 1. Biểu đồ tuần tự:

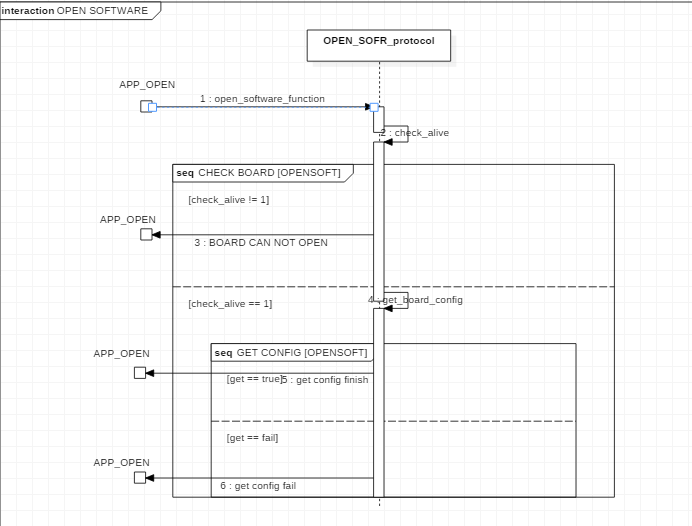
3.3.4.1 Tiến trình kiểm tra hoạt động thiết bị



Hình 3. 16: Sơ đồ thiết kế cho quá trình kiểm tra hoạt động thiết bị

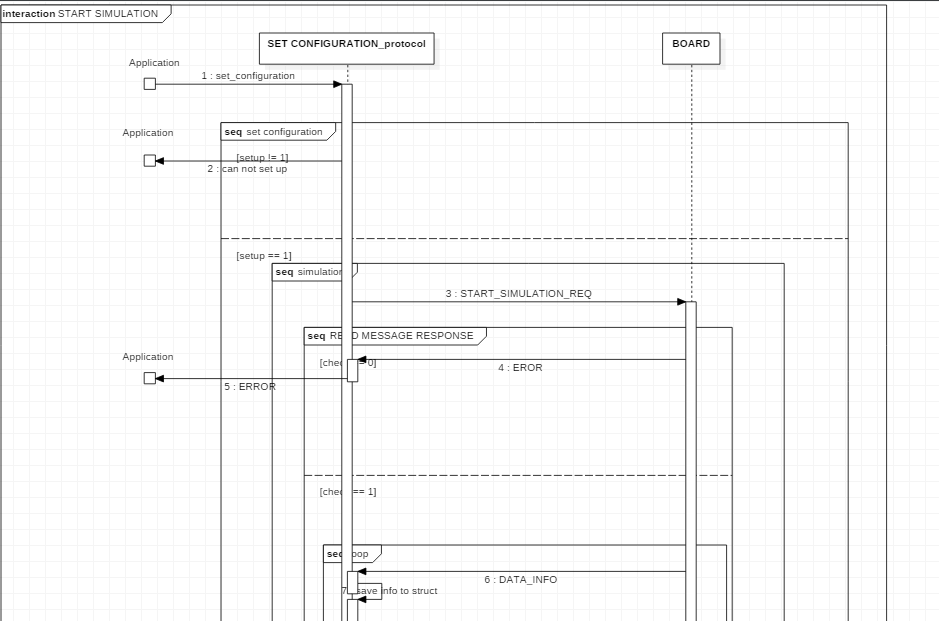
**Giải thích tiến trình :** PC gửi message CHECK\_ALIVE\_REQ để yêu cầu kiểm tra kết nối của thiết bị với bus CAN. Nếu thiết bị hoạt động bình thường, sẽ gửi lại message CHECK\_ALIVE\_RES với trường status chứa giá trị 1, nếu thiết bị chưa kết nối với bus CAN, sẽ gửi lại với giá trị status là 0. Nếu quá trình gửi message CHECK\_ALVIE\_REQ không được đáp ứng, sẽ liên tục gửi 5 lần, và trả về thông báo lỗi cho người dùng

3.3.4.2 Tiến trình lấy cấu hình thiết bị



Hình 3. 17: Sơ đồ thiết kế cho việc lấy cấu hình bus CAN

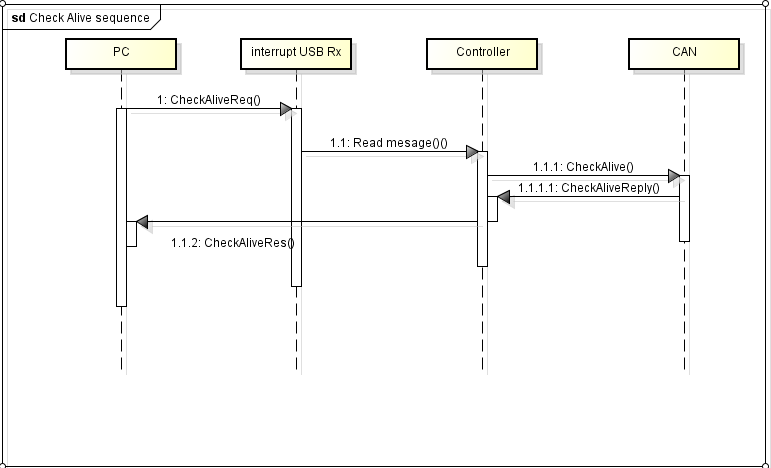
3.3.4.3 Tiến trình mô phỏng



Hình 3. 18 : Sơ đồ thực hiện quá trình mô phỏng

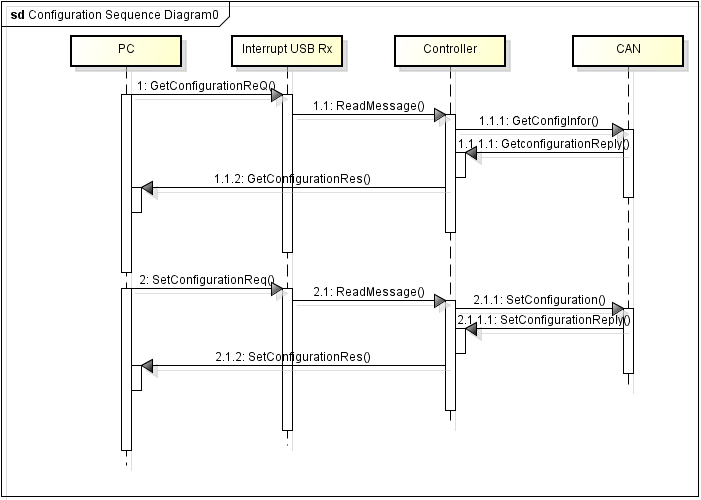
**Giải thích tiến trình:** Gửi message START\_SIMULATION\_REQ để thông báo cho thiết bị chuẩn bị nhận dữ liệu từ cổng CAN. Nếu thiết bị sẵn sàng cho tiến trình mô phỏng, sẽ gửi message START\_SIMULATION\_RES về cho PC . Các dữ liệu đến từ một thiết bị khác trên cổng CAN sẽ mang giá trị commandID DATA\_INFO . Khi PC muốn truyền tín hiệu đến bus CAN sẽ truyền dưới message SEND\_MESSAGE\_REQ và nhận lại SEND\_MESSAGE\_RES nếu việc gửi thành công, và nhận SEND\_MESSAGE\_RES\_FAIL nếu thiết bị không thể gửi dữ liệu ra bus CAN

* 1. **Hoạt động của thiết bị**
     1. Kiểm tra kết nối (CheckAlive)



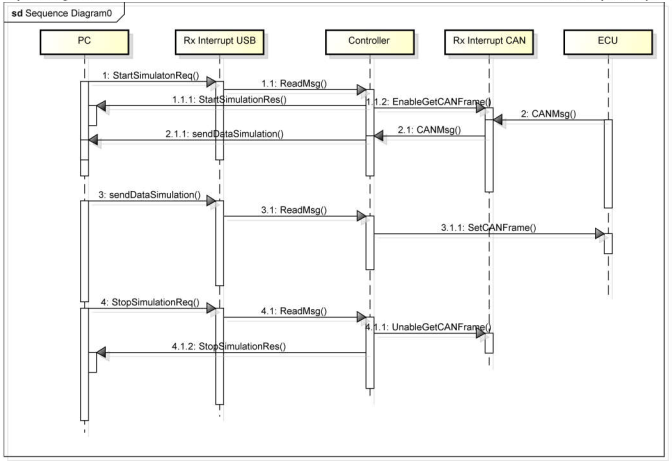
Hình 3. 19 Sơ đồ sequence cho nhiệm vụ CheckAlive

* + 1. Cấu hình cho thiết bị (Configuration)



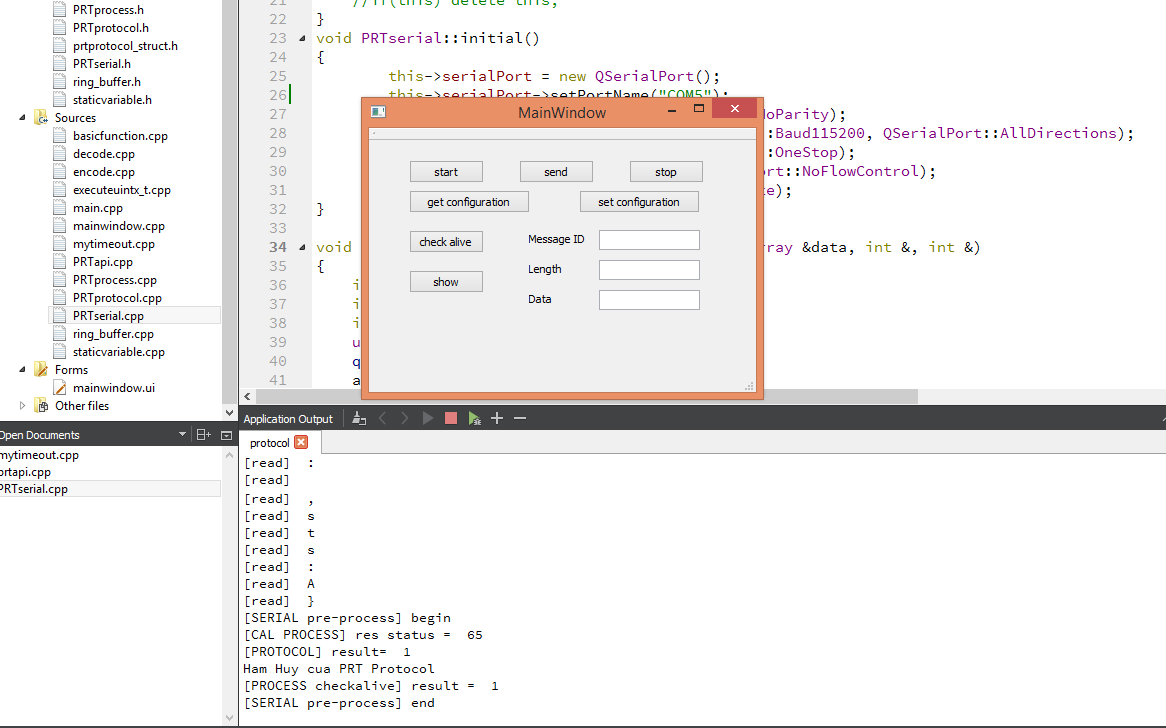
Hình 3. 20 Sơ đồ sequence cho nhiệm vụ cấu hình

* + 1. Mô phỏng



Hình 3. 21 Sơ đồ sequence mô tả hoạt động mô phỏng

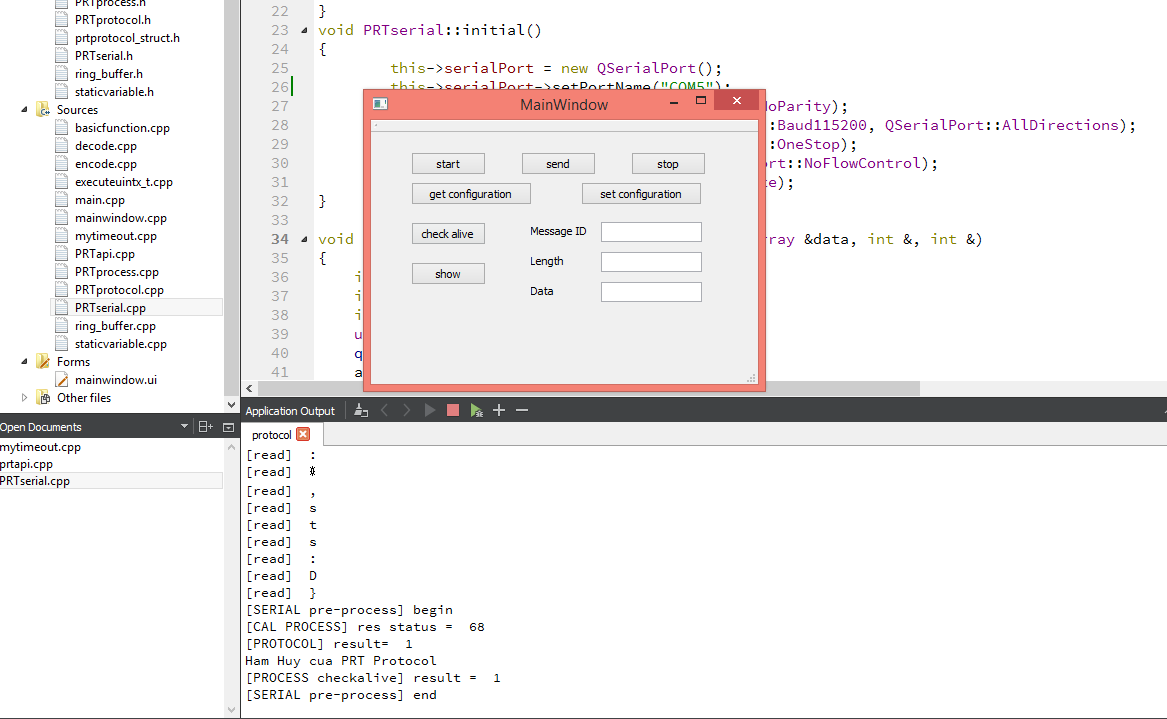
* 1. **Kết quả thực hiện**



Hình 3. 22: Kết quả mô phỏng quá trình check\_alive thành công

**Nhận xét:**

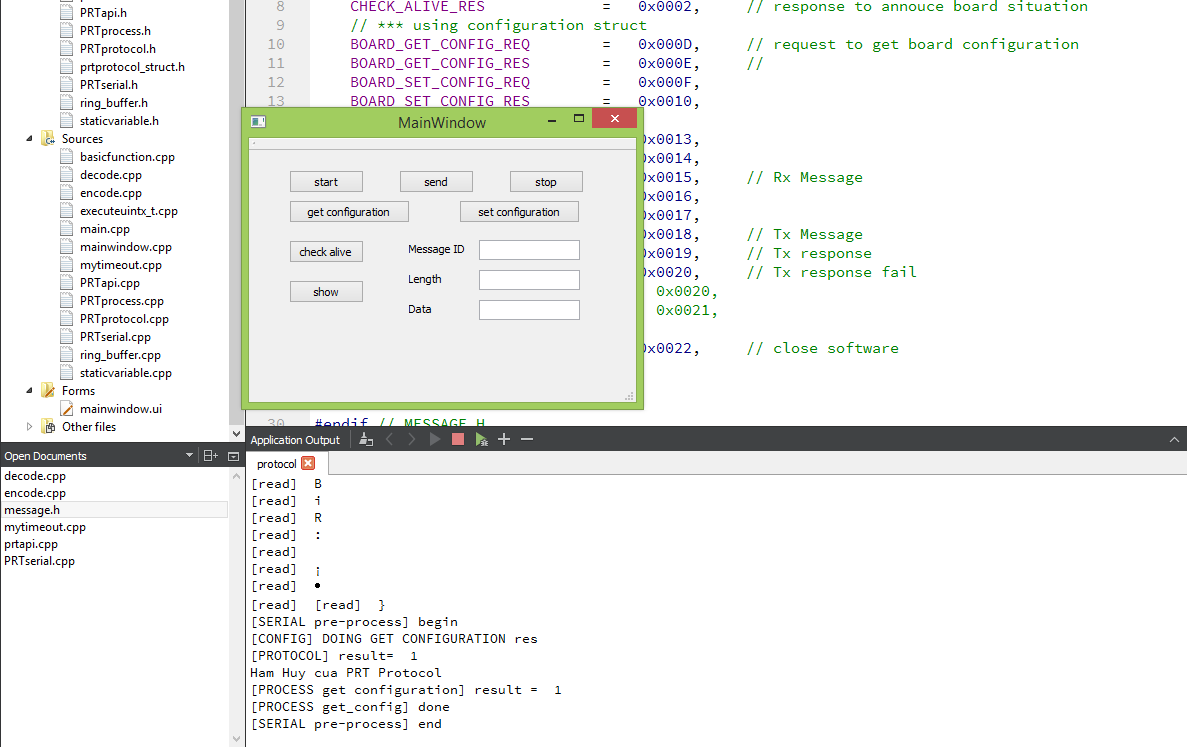
Khi gửi message CHECK\_ALIVE\_REQ từ PC đến thiết bị, thiết bị thực hiện kiểm tra sự hoạt động của cổng CAN , sau đó truyền message CHECK\_ALIVE\_RES về PC. Giá trị [CAL PROCESS] res satus = 65 (A theo mã ASCII ) thông báo thiết bị còn hoạt động



Hình 3. 23 Kết quả mô phỏng check\_alive thiết bị được ngắt khỏi CAN

**Nhận xét:**

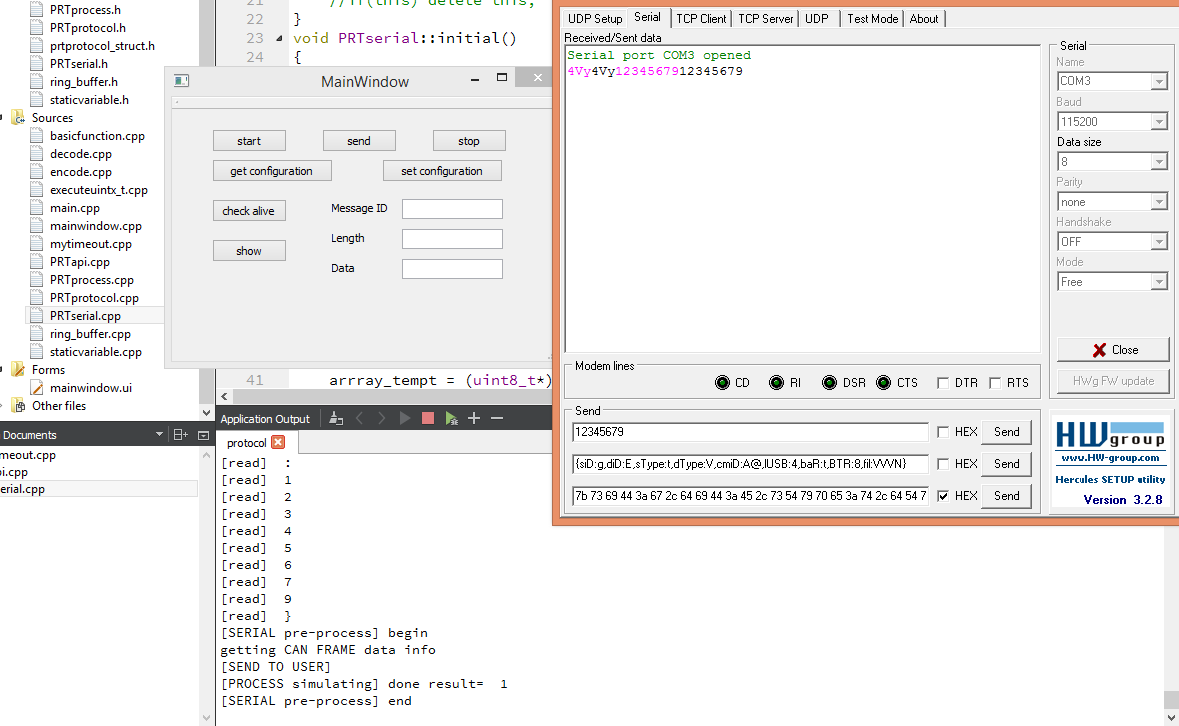
Khi gửi message CHECK\_ALIVE\_REQ từ PC đến thiết bị, thiết bị thực hiện kiểm tra sự hoạt động của cổng CAN , sau đó truyền message CHECK\_ALIVE\_RES về PC. Giá trị [CAL PROCESS] res satus = 68 (D theo mã ASCII ) thông báo thiết bị không có khả năng gửi nhận từ cổng CAN



Hình 3. 24 : Kết quá quá trình lấy cấu hình bus CAN

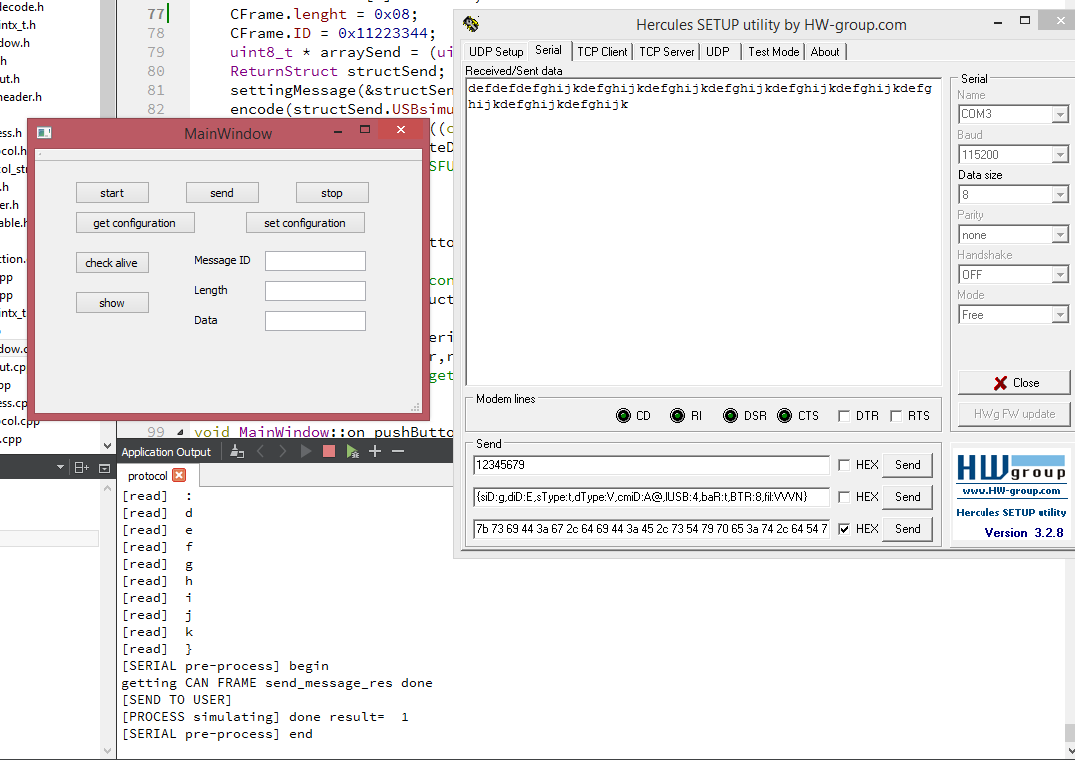
**Nhận xét:**

Khi gửi message BOARD\_GET\_CONFIG\_REQ từ PC đến thiết bị, thiết bị thực hiện kiểm tra sự hoạt động của cổng CAN , sau đó truyền message BOARD\_GET\_CONFIG\_RES về PC. Giá trị [PROTOCOL] result = 1 cho thấy quá trình truyền nhận message GET\_CONFIG đã hoàn thành. Giá trị [CONFIG DOING GET CONFIGURATION] res cho thấy message BOARD\_GET\_CONFIG\_RES đã nhận được tại PC và các giá trị nhận được từ quá trình truyền được in ra trong quá trình trường [READ] . Tuy quá trình lấy cấu hình dữ liệu đã hoàn thành, nhưng hạn chế về mặt tốc độ truyền nhận và chưa sử lý được các dữ liệu trống trong các struct lỗi được truyền đi.



Hình 3. 25 : kết quả quá trình mô phỏng với dữ liệu nhận từ bus CAN

**Nhận xét:** Khi bắt đầu tiến trình mô phỏng, phần mềm PC sẽ liên tục nhận message từ một giá trị bên ngoài cổng CAN gửi đến thiết bị và thiết bị truyền đến PC. Các giá trị ‘12345679’ được gửi từ một PC khác đến một thiết bị CAN, ra cổng CAN, thiết bị được kết nối trực tiếp tới PC sẽ nhận giá trị từ cổng CAN sau đó gửi đến PC để hiện thị. Các giá trị nhận được được hiển thị ra ở trường [READ] . thông báo getting CAN FRAME data info cho thấy PC đã nhận dược một message DATA\_INFO từ cổng CAN và hiện thị ra giá trị ‘12345679’.



Hình 3. 26: Kết quả mô phỏng với dữ liệu gửi từ PC đến bus CAN

**Nhận xét:** Khi bắt đầu tiến trình mô phỏng. Các giá trị ‘defghijk’ được gửi từ PC đến thiết bị CAN,từ thiết bị gửi ra cổng CAN, thiết bị khác được kết nối với bus CAN sẽ nhận giá trị từ cổng CAN sau đó hiển thị ra trên phần mềm giá trị đã gửi đi ‘defghijk’. Các giá trị nhận được được hiển thị ra nhờ phần mềm test. thông báo “getting CAN FRAME send message\_res done” cho thấy PC đã nhận dược một message SEND\_MESSAGE\_RES từ thiết bị nối gửi về, sau khi thiết bị đó đã gửi giá trị CAN frame trong message SEND\_MESSAGE\_REQ ra cổng CAN thành công

* 1. **Đánh giá và hướng phát triển đề tài:**

Qua hơn 4 tháng thực hiện đề tài, chúng tôi đã đạt được một số kết quả nhất định trong việc thiết kế giao thức và truyền nhận dữ liệu theo giao thức trên: đã thực hiện được các tác vụ truyền nhận CAN Frame, lấy thông số cấu hình, song vẫn còn hạn chế trong tốc độ truyền và một số lỗi khi thực hiện các tác vụ trùng lặp trên giao diện

Trong thời gian có hạn, đề tài vẫn chưa khắc phục được các hạn chế trên, trong tương lai sẽ có thể phát triển đến việc xây dựng giao thức cho đa người dung, đa phần mềm , và mở rộng ra tương tác với một số mạng khác như mạng Flexray, mạng Ethernet…

* 1. **Kết luận chương**

Qua nội dung đã trình bày trong chương 3, cho chúng ta cái nhìn tổng quát về quy trình viết driver, cách xây dựng driver trên hệ điều hành linux và cách kết nối dữ liệu từ PC đến kit TM4C123G . Bên cạnh đó, chương 3 còn mô tả thành công một số tiến trình truyền nhận dữ liệu trong hệ thống mô phỏng mạng CAN bus như tiến trình kiểm tra hoạt động của kit DK-TM4C123G khi kết nối với cổng CAN bus và tiến trình mô phỏng, nhận dữ liệu CAN frame trên bus CAN và hiển thị ra giao diện để người dung có thể giám sát mạng CAN .

# KẾT LUẬN

Tuy nhiên vẫn còn hạn chế trong tốc độ truyền nhận dữ liệu do đặc tính của driver và việc khởi tạo quá trình truyền nhận tại PC, nhưng đề tài đã thành công trong việc thực hiện truyền nhận dữ liệu từ PC đến kit TM4C123G thông qua driver USB CDC-ACM sử dụng phần mềm mô phỏng mạng CAN. Ngoài ra, đề tài còn thiết kế và thực hiện thành công một số giao thức quan trọng trên hệ thống mô phỏng CAN bus, với các chức năng chính là việc kiểm tra khả năng hoạt động của thiết bị nối với cổng CAN, nhận và gửi thông tin cài đặt cấu hình đến bus CAN, thực hiện được tiến trình mô phỏng, nhận dữ liệu tới từ cổng CAN và xuất dữ liệu ra PC để người dùng giám sát, cũng như gửi thành công dữ liệu từ PC đến thiết bị ra cổng CAN. Việc thực hiện thành công một số giao thức quan trọng như đã đề cập ở trên, sẽ tạo tiền đề quan trọng cho việc thực hiện các chức năng mở rộng trong giao thức sau này, như hỗ trợ đa người dùng trên thiết bị, hoặc thực hiện mô phỏng các dạng sóng trong tín hiệu truyền đi từ PC đến bus CAN và ngược lại. Qua nội dung đã trình bày trong đề tài nói trên, chúng tôi hy vòn mang đến cho người đọc một cái nhìn tổng quan về hệ thống mô phỏng mạng CAN và tính cấp thiết của việc sử dụng phần mềm mô phỏng CAN bus để giám sát tiến trình hoạt động của mạng CAN trong ngành công nghệ nói chung và đặc biệt là trong ngành sản xuất và phát triển các thiết bị, ứng dụng trên ô tô.

Trong quá trình thực hiện đề tài, không tránh khỏi những thiếu sót nhất định. Kính mong quý đọc giả đóng góp để chúng tôi có thể hoàn thành đồ án tốt hơn nữa.

Xin chân thành cảm ơn!

Nhóm Sinh Viên Thực Hiện

# TÀI LIỆU THAM KHẢO

[1] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, “Linux Device Driver, 3rd Edition”, O’Reilly, 2005.

[2] Daniel P. Bovet, Marco Cesati, “Understanding the Linux Kernel, 3rd Edition”, O’Reilly, 2005

**PHỤ LỤC 1**

**PHỤ LỤC 2**