

**BỘ GIÁO DỤC & ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH  
KHOA ĐIỆN - ĐIỆN TỬ  
BỘ MÔN ĐIỆN TỬ-CÔNG NGHIỆP**



# **ĐỒ ÁN TỐT NGHIỆP**

**NGÀNH KỸ THUẬT ĐIỆN - ĐIỆN TỬ**

**ĐỀ TÀI:**

**NGHIÊN CỨU VÀ BIÊN SOẠN**

**GIÁO TRÌNH KIT KM9260**

**TRÊN NỀN LINUX.**

**GVHD: ThS Nguyễn Đình Phú**

**SVTH: Nguyễn Tấn Như;**

**MSSV: 07101080.**

**Tp. Hồ Chí Minh - 2/2012**

**BỘ GIÁO DỤC & ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH  
KHOA ĐIỆN – ĐIỆN TỬ  
BỘ MÔN ĐIỆN TỬ-CÔNG NGHIỆP**  
-----❧\*★\*❧-----

# **ĐỒ ÁN TỐT NGHIỆP**

**NGÀNH KỸ THUẬT ĐIỆN - ĐIỆN TỬ**

**ĐỀ TÀI:**

**NGHIÊN CỨU VÀ BIÊN SOẠN  
GIÁO TRÌNH KIT KM9260  
TRÊN NỀN LINUX.**

**GVHD: ThS Nguyễn Đình Phú;**

**SVTH: Nguyễn Tấn Như;**

**Lớp: 071012B, Hệ chính qui;**

**MSSV: 07101080.**

**Tp. Hồ Chí Minh - 2/2012**

# PHẦN I

# GIỚI THIỆU

**KHOA ĐIỆN - ĐIỆN TỬ**  
**NGÀNH KỸ THUẬT ĐIỆN TỬ**

**NHIỆM VỤ ĐỒ ÁN TỐT NGHIỆP**

Họ và tên: **NGUYỄN TẤN NHƯ** MSSV: **07101080**

Lớp: **071012B**  
Ngành: **KỸ THUẬT ĐIỆN - ĐIỆN TỬ**  
Hệ: **ĐẠI HỌC CHÍNH QUY**  
Niên khóa: **2007– 2012**

1. Tên đề bài:  
**“NGHIÊN CỨU VÀ BIÊN SOẠN GIÁO TRÌNH KIT NHÚNG KM9260  
TRÊN NỀN LINUX”**

2. Các số liệu ban đầu:

3. Nội dung các phần thuyết minh:

4. Các bản vẽ đồ thị:

5. Giáo viên hướng dẫn: **ThS Nguyễn Đình Phú**

6. Ngày giao nhiệm vụ: .....

7. Ngày hoàn thành nhiệm vụ: .....

**Thông qua bộ môn**  
**Ngày ... tháng ... năm 2012**  
**Chủ nhiệm bộ môn**

## LỜI MỞ ĐẦU

Thế giới ngày nay với khoa học kỹ thuật phát triển mạnh mẽ cuộc sống con người ngày càng được phát triển tốt hơn. Khoa học kỹ thuật đem lại nhiều tiện ích thiết thực hơn cho cuộc sống con người. Góp phần to lớn trong quá trình phát triển của khoa học kỹ thuật là sự phát triển mạnh mẽ của vi xử lý. Từ bộ vi xử lý đầu tiên Intel 4004 được sản xuất bởi công ty Intel vào năm 1971, đến nay ngành công nghiệp vi xử lý đã phát triển vượt bậc và đa dạng với nhiều loại như: 8951, PIC, AVR, ARM, Pentium, Core i7,....

Cùng với sự phát triển đa dạng về chủng loại thì tài nguyên của vi xử lý cũng được nâng cao. Các vi xử lý ngày nay cung cấp cho người dùng một nguồn tài nguyên rộng lớn và phong phú. Có thể đáp ứng được nhiều yêu cầu khác nhau trong thực tế. Để giúp cho người dùng sử dụng hiệu quả và triệt để các tài nguyên này thì hệ thống nhúng ra đời. Hệ thống nhúng (Embedded system) là một thuật ngữ để chỉ một hệ thống có khả năng tự trị được nhúng vào trong một môi trường hay một hệ thống mẹ. Đó là các hệ thống tích hợp cả phần cứng và phần mềm phục vụ các bài toán chuyên dụng trong nhiều lĩnh vực công nghiệp, tự động hoá điều khiển, quan trắc và truyền tin. Với sự ra đời của hệ thống nhúng thì vi xử lý ngày càng được ứng dụng rộng rãi trong đời sống cũng như trong công nghiệp vì khả năng xử lý nhanh, đa dạng, tiết kiệm năng lượng và độ ổn định của hệ thống nhúng.

Những năm gần đây, sự năng động và tích cực hội nhập quốc tế đã đem về hơi thở mới cho Việt Nam về mọi mặt: kinh tế, xã hội, văn hóa, nghệ thuật ... Lĩnh vực kỹ thuật nói chung và kỹ thuật điện tử nói riêng cũng có những thay đổi theo chiều hướng tích cực. Bên cạnh việc áp dụng những kỹ thuật mới (chủ yếu mua từ nước ngoài) vào sản xuất, nhiều công ty ở Việt Nam đã chú trọng đến việc phát triển đội ngũ R&D (Research And Development) để tự chế tạo sản phẩm hoàn thiện cung ứng cho thị trường. Một trong những sản phẩm đó là kit KM9260 là một kit nhúng được tích hợp cao trên nền vi điều khiển AT91SAM9260.

Tuy hệ thống nhúng rất phổ biến trên toàn thế giới và là hướng phát triển của ngành Điện tử sau này nhưng hiện nay ở Việt Nam độ ngũ kỹ sư hiểu biết về hệ thống nhúng còn rất hạn chế không đáp ứng được nhu cầu nhân lực trong lĩnh vực này. Trước tình

hình thiếu nhân lực như thế này, trường Đại Học Sư Phạm Kỹ Thuật Thành Phố Hồ Chí Minh với tư cách là một trong những trường sư phạm kỹ thuật đứng đầu của Việt Nam đã nghiên cứu về lĩnh vực hệ thống nhúng và sẽ đưa vào hệ thống môn học đào tạo trong tương lai gần nhất.

Vì vậy việc biên soạn giáo trình về hệ thống nhúng là một yêu cầu cần thiết trong thời điểm hiện tại cũng như trong tương lai. Nhận thấy được nhu cầu cấp thiết đó nên sinh viên thực hiện đã chọn đề tài: **“NGHIÊN CỨU VÀ BIÊN SOẠN GIÁO TRÌNH KIT NHÚNG KM9260 TRÊN NỀN LINUX”** để làm đồ án tốt nghiệp cho mình.

Những kiến thức, năng lực đạt được trong quá trình học tập ở trường sẽ được đánh giá qua đợt bảo vệ đồ án cuối khóa. Vì vậy sinh viên thực hiện đề tài cố gắng tận dụng những kiến thức đã học ở trường cùng với sự tìm tòi, nghiên cứu cùng với sự hướng dẫn tận tình của Giáo viên hướng dẫn cùng Thầy/Cô thuộc Khoa Điện-Điện Tử để có thể hoàn thành tốt đồ án này.

Mặc dù sinh viên thực hiện đề tài đã cố gắng hoàn thành nhiệm vụ đề tài đặt ra và đúng thời hạn nhưng chắc chắn sẽ không tránh khỏi những thiếu sót, mong quý Thầy/Cô và các bạn sinh viên thông cảm. Sinh viên thực hiện đề tài mong nhận được những ý kiến đóng góp của quý Thầy/Cô và các bạn sinh viên.

*TP.HCM, Ngày tháng năm 2012*

**Sinh viên thực hiện đề tài**

***Nguyễn Tấn Như***

## LỜI CẢM ƠN

Lời đầu tiên, sinh viên thực hiện đề tài xin được phép chân thành gửi lời cảm ơn đến thầy Nguyễn Đình Phú, giáo viên hướng dẫn đề tài, đã định hướng và trao đổi những kinh nghiệm quý báu để em thực hiện những nội dung trong đề tài này cách hoàn chỉnh.

Kể đến, em cũng xin tỏ lòng biết ơn đến thầy Nguyễn Tấn Thịnh, cựu sinh viên của trường Đại Học Sư Phạm Kỹ Thuật, đã giúp em có được những kiến thức rất cơ bản có vai trò là nền tảng để giúp em phát triển những nội dung trong đề tài.

Em cũng xin trân trọng cảm ơn các thầy cô trong trường Đại Học Sư Phạm Kỹ Thuật đã tận tình truyền đạt những kiến thức và tình yêu nghề để em có sự đam mê nghiên cứu khám phá những kiến thức mới trong ngành.

Cuối cùng em xin dâng lời cảm ơn đến Chúa Giêsu, cha, mẹ và những người thân trong gia đình, bạn bè, ... đã tạo điều kiện thuận lợi về tinh thần và vật chất giúp em hoàn thành đề tài này.

*TP.HCM, Ngày    tháng    năm 2012*

**Sinh viên thực hiện đề tài**

***Nguyễn Tấn Như***

# MỤC LỤC

<b><i>NỘI DUNG</i></b>	<b><i>TRANG</i></b>
<b>PHẦN A: GIỚI THIỆU .....</b>	
Trang bìa .....	i
Nhiệm vụ đồ án .....	ii
Lời mở đầu .....	iii
Lời cảm ơn .....	iv
Mục lục .....	v
Liệt kê hình vẽ .....	ix
Liệt kê bảng .....	xii
 <b>PHẦN B: NỘI DUNG .....</b>	
 <b>CHƯƠNG I: DẪN NHẬP..... 1</b>	
1.1. Đặt vấn đề .....	2
1.2. Lý do chọn đề tài. ....	2
1.3. Đối tượng nghiên cứu .....	3
1.4. Giới hạn đề tài .....	3
1.5. Dàn ý nghiên cứu .....	3
1.6. Tình hình nghiên cứu .....	4
1.7. Ý nghĩa thực tiễn. ....	5
 <b>CHƯƠNG V:</b>	
<b>KẾT QUẢ NGHIÊN CỨU-KẾT LUẬN-HƯỚNG PHÁT TRIỂN..... 116</b>	
5.1 Kết quả nghiên cứu. ....	117
5.2 Kết luận .....	117
5.3 Hướng phát triển .....	118
 <b>PHẦN C: PHỤ LỤC..... 119</b>	



## LIỆT KÊ HÌNH VẼ

Hình	Trang
Hình 2.1: Sơ đồ khối tổng quát VN8-01 .....	6
Hình 2.2: Sơ đồ chân VN8-01 .....	7
Hình 2.3: CPU xử lý 5 giai đoạn.....	10
Hình 2.4: Kiến trúc đường ống 5 tầng .....	11
Hình 2.5: 1 lệnh đơn cần 5 xung clock .....	11
Hình 2.6: Tổ chức bộ nhớ chương trình trong VN8-01 .....	12
Hình 2.7 Hoạt động của PC đối với lệnh thường và lệnh rẽ nhánh .....	13
Hình 2.8: Cấu trúc của 1 lệnh đơn .....	14
Hình 2.9: Các nguồn ngắt của VN8-01 .....	17
Hình 2.10: Hoạt động của Stack và thanh ghi PC.....	18
Hình 2.11: Cấu trúc thanh ghi INTCON.....	19
Hình 2.12: Cấu trúc thanh ghi PIR1 .....	20
Hình 2.13: Cấu trúc thanh ghi PIE1 .....	21
Hình 2.14: Cấu trúc thanh ghi OPTION .....	22
Hình 2.15: Clock ngõ vào cho các bộ Timer 0,1,2 .....	22
Hình 2.16: Sơ đồ khối của Timer0.....	25

Hình 2.17: Sơ đồ khối bộ định thời 1(Timer1) .....	29
Hình 2.18: Cấu trúc thanh ghi T1CON .....	29
Hình 2.19: Sơ đồ khối của Timer2 .....	32
Hình 2.20: Cấu trúc thanh ghi T2CON .....	32
Hình 2.21: Thiết kế hoạt động cho chức năng WDT .....	34
Hình 2.22: Sơ đồ khối Watchdog-Timer.....	35
Hình 2.23: Chế độ hoạt động của CPP và nguồn Timer .....	39
Hình 2.24: Cấu trúc thanh ghi CCPCON .....	39
Hình 2.25: Cấu trúc thanh ghi PIR1 .....	40
Hình 2.26: Vị trí bit CCPIE.....	40
Hình 2.27: quan hệ giữa CK bồn phân (Thigh) và CK xung (Tcycle) .....	42
Hình 2.28: Sơ đồ khối Capture của ngõ vào CCPI .....	43
Hình 2.29: Sơ đồ khối Compare.....	45
Hình 2.30: Sơ đồ khối PWM.....	46
Hình 2.31: Thay đổi CKNV khi CK xung cố định .....	47
Hình 2.32: Cấu tạo bộ truyền USART .....	50
Hình 2.33: cấu trúc thanh ghi TXSTA.....	51
Hình 2.34: Cấu trúc thanh ghi RCSTA .....	52
Hình 2.35: vị trí cờ ngắt nhận .....	53
Hình 2.36: Vị trí bit cờ ngắt .....	54
Hình 2.37: Quan hệ giữa clock lấy mẫu và clock truyền dữ liệu.....	55
Hình 2.38: Clock khi SPBRG = 0x00 .....	56
Hình 2.39: Giảm đồ xung truyền dữ liệu từng bit.....	57
Hình 2.40: Lấy mẫu dữ liệu ứng với BaudRate = $F_{osc}/(64x(X+1))$ .....	58
Hình 2.41: Lấy mẫu dữ liệu ứng với Baud Rate = $f_{osc}/(16x(X+1))$ .....	58
Hình 2.42: Lấy mẫu dữ liệu ứng với Baud Rate = $f_{osc}/(8x(X+1))$ .....	58
Hình 2.43: Cấu tạo của bộ truyền USART .....	60
Hình 2.44: Truyền bất đồng bộ 1 khung dữ liệu dạng truyền đơn lẻ.....	61
Hình 2.45: Truyền bất đồng bộ Back-to-Back.....	61
Hình 2.46 Cấu tạo bộ nhận bất đồng bộ USART.....	62
Hình 2.47: Quá trình truyền đồng bộ Master .....	65
Hình 2.48: Chế độ nhận đồng bộ Master (nhận 1 lần).....	67

Hình 2.49: Hoạt động truyền trong chế độ Standby (đồng bộ Slave).....	69
Hình 2.50: Hoạt động nhận trong chế độ Standby (đồng bộ Slave) .....	70
Hình 3.1: Giản đồ thời gian thể hiện tín hiệu xung reset và xung presence .....	74
Hình 3.2: Giản đồ trình tự thời gian của xung reset & presence .....	74
Hình 3.3: Giản đồ thời gian cho VN8-01 ghi tín hiệu mức 1 & mức 0 .....	75
Hình 3.4: Giản đồ thời gian để VN8-01 đọc tín hiệu mức 1 & 0.....	75
Hình 3.5: Dạng đóng gói TO-92 .....	77
Hình 3.6: Dạng đóng gói SO.....	77
Hình 3.7: Mô tả chi tiết giá trị bit trong bộ nhớ lưu giá trị chuyển đổi .....	78
Hình 3.8: Thể hiện cấu trúc bộ nhớ DS18B20.....	79
Hình 3.9: Thanh ghi cấu hình độ phân giải (byte4) .....	80
Hình 3.10: Nguồn cung cấp qua đường tín hiệu .....	83
Hình 3.11: DS18B20 với nguồn cung cấp độc lập.....	83
Hình 3.12: Kết nối phân cứng các thiết bị giao tiếp I2C .....	85
Hình 3.13: Quan hệ truyền/nhận dữ liệu giữa thiết bị chủ/tớ .....	85
Hình 3.14: Giản đồ thể hiện tín hiệu xung Start và Stop .....	87
Hình 3.15: Quá trình truyền 1bit dữ liệu.....	87
Hình 3.16: Quá trình truyền 8bit dữ liệu và tín hiệu đáp ứng Ack .....	88
Hình 3.17: Tín hiệu đáp ứng ACK từ thiết bị nhận .....	88
Hình 3.18: Lưu đồ thuật toán truyền nhận dữ liệu .....	89
Hình 3.19: Cấu trúc byte dữ liệu đầu tiên .....	89
Hình 3.20: Quá trình truyền dữ liệu .....	90
Hình 3.21: Ghi dữ liệu từ chủ đến tớ với chiều được xác định.....	91
Hình 3.22: Thiết bị chủ đọc dữ liệu từ thiết bị tớ.....	91
Hình 3.23: Thiết bị chủ kết hợp đọc & ghi dữ liệu trong quá trình truyền thông.....	92
Hình 3.24: Dạng đóng gói DIP của DS1307.....	92
Hình 3.25: Tổ chức bộ nhớ trong DS1307 .....	93
Hình 3.26: Tổ chức bên trong của các thanh ghi thời gian .....	94
Hình 4.1: Sơ đồ khối tổng quát Kit thí nghiệm VN8-01 .....	99
Hình 4.2: Sơ đồ chân của 74HC573.....	102
Hình 4.3: Sơ đồ chân IC 74HC595 .....	103
Hình 4.5: Sơ đồ chân 74HC138 .....	106

Hình 4.7: Sơ đồ chân của LCD16*2 .....	107
Hình 4.8: Sơ đồ chân của 74HC541.....	108
Hình 4.10 : Sơ đồ chân của 74HC151.....	111
Hình 4.11 : Sơ đồ chân của IC 74HC07.....	112
Hình 4.12 : Sơ đồ chân của IC 74HC08.....	112
Hình 4.14 : sơ đồ chân của ADC0809 .....	113

## LIỆT KÊ BẢNG

<b>Bảng</b>	<b>Trang</b>
Bảng 2.2: Vùng giá trị bộ nhớ Ram/bank.....	14
Bảng 2.3: Mô tả thanh ghi INTCON.....	19
Bảng 2.4: mô tả thanh ghi PIR1.....	20
Bảng 2.5: Mô tả thanh ghi PIE1.....	21
Bảng 2.6: Mô tả thanh ghi OPTION.....	22
Bảng 2.6: Mô tả thanh ghi OPTION.....	25
Bảng 2.7: Mô tả thanh ghi T1CON.....	29
Bảng 2.8: Mô tả thanh ghi T2CON.....	32
Bảng 2.9: Mô tả thanh ghi CCPCON.....	39

Bảng 2.10: Mô tả nội dung bit cờ ngắt.....	40
Bảng 2.11: Mô tả chức năng bit CCPIE.....	40
Bảng 2.12: Mô tả chức năng thanh ghi TXSTA.....	51
Bảng 2.13: Mô tả chức năng thanh ghi RCSTA.....	52
Bảng 2.14: Mô tả chức năng của cờ ngắt.....	54
Bảng 2.15: Mô tả chức năng bit cờ ngắt.....	54
Bảng 2.16: Công thức tính tốc độ Baud.....	55
Bảng 2.17: Các bước thiết lập bộ truyền bất đồng bộ.....	60
Bảng 2.18: Các bước thiết lập bộ nhận bất đồng bộ.....	62
Bảng 2.19 Các bước thiết lập cho bộ truyền MASTER.....	64
Bảng 2.20: Các bước cấu hình bộ nhận đồng bộ Master.....	66
Bảng 2.21: Các bước để thiết lập một chế độ truyền đồng bộ SLAVE .....	67
Bảng 2.22: Các bước cấu hình chế độ nhận đồng bộ Slave.....	69
Bảng 3.1: Mối quan hệ giữa nhiệt độ và gt lưu trong bộ nhớ độ phân giải 12bits...	79
Bảng 3.2: Giá trị cấu hình tương ứng với từng độ phân giải.....	80
Bảng 4.1 : Mô tả chức năng chân của GLCD.....	109

PHẦN II

# NỘI DUNG

# CHƯƠNG I

# DẪN NHẬP

### **1.1. ĐẶT VẤN ĐỀ**

Trên thế giới hệ thống nhúng (embedded system) đã xuất hiện và có mặt trên thị trường vào những năm 1960, cho đến nay đã hơn nửa thế kỷ hình thành và phát triển. Với những ưu điểm mà các hệ thống khác không có như tính gọn nhẹ, đáp ứng nhanh, độ tin cậy cao, và quan trọng là khả năng linh hoạt trong nhiều ứng dụng tạo điều kiện thuận lợi cho sản xuất hàng loạt dẫn đến giá thành giảm, ... Hệ thống nhúng được sử dụng đa dạng trong mọi lĩnh vực như: dân dụng, quân sự, y tế,... Doanh thu mà các hệ thống nhúng mang lại là vô cùng lớn. Theo thống kê của một hãng nghiên cứu của Canada thì đến năm 2009, tổng thị trường của các hệ thống nhúng toàn cầu đạt 88 tỷ USD. Trong đó phần cứng đạt 78 tỷ và phần mềm đạt 3.5 tỷ. Trong tương lai lượng doanh thu này sẽ còn tăng lên đáng kể. Lĩnh vực này đã và đang là một cơ hội và thách thức vô cùng lớn cho các doanh nghiệp trong và ngoài Việt Nam.

Tại Việt Nam, việc bắt tay vào nghiên cứu và phát triển phần mềm cũng như phần cứng hệ thống nhúng chỉ mới bắt đầu trong vài năm trở lại đây và được xem là một “cơ hội vàng” cho các công ty điện tử nghiên cứu và phát triển ứng dụng vào các sản phẩm công nghệ cao. Nhiều công ty xem việc phát triển phần cứng và phần mềm hệ thống nhúng là “đích nhắm trong tương lai” và không ngừng đầu tư hợp tác với các đối tác nước ngoài để tiếp thu và phát triển. Trong bối cảnh đất nước ta đang trên đường hội nhập phát triển, việc đầu tư mở các nhà máy công xưởng chuyên về sản xuất hệ thống nhúng và tìm kiếm đối tác lớn từ nước ngoài có kinh nghiệm trong lĩnh vực này là không khó. Thế nhưng liệu chúng ta có đủ yếu tố con người để làm việc trong những môi trường này hay không?

Hiện nay đội ngũ nhân lực của nước ta trong lĩnh vực thiết kế hệ thống nhúng còn hạn chế, chưa có một kiến thức đủ rộng và sâu để có thể hợp tác phát. Việt Nam bước đầu đã có những chương trình hợp tác với các hãng điện tử lớn như Toshiba, Samsung, Panasonic, ... tuy nhiên những chương trình như thế còn rất hạn chế và không có một định hướng chiến lược chung. Việt Nam cần phải đẩy mạnh hơn nữa vấn đề định hướng nghiên cứu và phát triển cho ngành hệ thống nhúng từ trong các trường đại học và trung tâm nghiên cứu, cũng như trang bị những kiến thức về lý thuyết và thực hành cho những sinh viên trẻ, đáp ứng nhu cầu ngày càng cao của nhà tuyển dụng. Muốn vậy vấn đề đặt ra là phải có một chương trình được thiết kế chuyên sâu cả về lý thuyết và thực hành trong các cơ sở đào tạo chính quy để định hướng ban đầu cho các sinh viên



theo học tại trường có một niềm đam mê lĩnh vực lập trình nhúng đầy tiềm năng này, tạo một nền tảng vững chắc cho sự phát triển bền vững và nhanh chóng trong tương lai.

Một trong những yếu tố quan trọng nhất quyết định đến chất lượng thành công của chương trình đào tạo là tài liệu hướng dẫn thực hiện chương trình đào tạo đó. Vì thế việc biên soạn một giáo trình đầy đủ, chuyên sâu, giảng dạy song hành giữa lý thuyết và thực hành để người học không chỉ nắm vững nguyên lý hoạt động mà còn có thể thao tác điều khiển vào những ứng dụng thực tế của hệ thống nhúng, từ đó tự mình có thể nghiên cứu phát triển trong các môi trường chuyên nghiệp sau này là một nhu cầu cấp thiết.

### 1.2. LÝ DO CHỌN ĐỀ TÀI

Tên đề tài của nhóm là “**NGHIÊN CỨU VÀ BIÊN SOẠN GIÁO TRÌNH KIT NHÚNG KM9260 TRÊN NỀN LINUX**”.

Khi một họ CHIP vi điều khiển mới được ra đời, cùng với những tính năng vượt trội hơn so với các vi điều khiển khác, nhà sản xuất thường cho ra đời các kit nhúng sử dụng ngay những CHIP này để làm nổi bật những tính năng mà nó hỗ trợ. Kit nghiệm KM9260 là một loại như thế. Được xây dựng dựa vào cấu trúc hệ thống nhúng at91sam9260-ek của hãng Atmel, kit được tích hợp rất nhiều ngoại vi thích hợp với những đặc điểm của hệ thống nhúng để nhóm chúng em chọn làm đề tài nghiên cứu của mình.

Việc lựa chọn một kit cụ thể để viết giáo trình hướng dẫn lập trình ứng dụng trong hệ thống nhúng được quyết định bởi ba lý do sau:

- *Thứ nhất*, đa số các hệ thống nhúng (bao gồm phần cứng và phần mềm) được xây dựng theo một chuẩn nhất định. Nghiên cứu một đối tượng cụ thể trong chuẩn này có nghĩa rằng tất cả các đối tượng khác cũng đang được nghiên cứu. Tiếp cận theo hướng cụ thể sẽ giúp cho người học có nhiều kinh nghiệm lập trình từ thực tiễn, rút ra kiến thức tổng quát từ những thao tác được hướng dẫn, ... tạo nền tảng cho việc nghiên cứu các hệ thống khác. Nếu tiếp cận theo hướng chung nhất, mặc dù người học có thể hiểu rõ nguyên tắc hoạt động của hệ thống nhưng sẽ không thể áp dụng vào bất kỳ một hệ thống thực tế nào.
- *Thứ hai*, độ tin cậy cao, giá thành và thời gian thiết kế giảm. Đa số các kit thí nghiệm được nghiên cứu và kiểm tra qua nhiều công đoạn trước khi xuất hiện trên thị trường đại trà vì thế hệ thống hoạt động với độ ổn định cao. Do sản xuất với số lượng lớn nên chi phí thiết kế sẽ giảm, thời gian thực hiện được rút ngắn.

- Thứ ba, các kit thí nghiệm được sản xuất với rất nhiều ngoại vi khác nhau tích hợp trên một board mạch, nhiều chân cắm mở rộng, ... thích hợp cho việc ứng dụng kết hợp với các kit thí nghiệm hiện có tránh lãng phí trong quá trình chuyển giao thay đổi công nghệ.

Một hệ thống nhúng có thể chạy trên nhiều hệ điều hành khác nhau như WinCE, Android, Linux, ...nhưng nhóm chúng em chọn hệ điều hành Linux để nghiên cứu và triển khai các bài thí nghiệm trong giáo trình vì hệ điều hành này có nhiều ưu điểm mà các hệ điều hành khác không có.

- Ưu điểm đầu tiên là Linux có mã nguồn mở, chúng ta có thể thao tác chỉnh sửa mã nguồn này để phù hợp với hệ thống. Hiện nay Linux hỗ trợ cho rất nhiều dòng vi điều khiển khác nhau như: ARM, AVR, ...nên khi nghiên cứu linux chúng ta có thể dễ dàng làm việc trên nhiều hệ thống khác.
- Linux được xây dựng hoàn toàn bằng ngôn ngữ lập trình C. Đây là ngôn ngữ thông dụng và phổ biến. Người học khi nghiên cứu sẽ dễ dàng tiếp thu được thuật toán chương trình điều khiển và nguyên lý làm việc của hệ thống.
- Có rất nhiều sách hay và thông tin bàn luận về hệ điều hành này trên internet. Do đó khi nghiên cứu người học có thể tìm hiểu trên mọi kênh thông tin để nắm vững nội dung của bài học.
- Và còn rất nhiều ưu điểm khác, những ưu điểm này sẽ được người học nhận ra trong quá trình sử dụng.

Với những lý do trên nhóm chúng em chọn đề tài : **"NGHIÊN CỨU VÀ BIÊN SOẠN GIÁO TRÌNH KIT NHÚNG KM9260 TRÊN NỀN LINUX"** làm đề tài tốt nghiệp của mình. Với mục đích làm tài liệu nghiên cứu cho các bạn sinh viên mong muốn tìm hiểu thế giới lập trình nhúng đầy thú vị, góp phần làm cho môn học này được phát triển rộng rãi trong đội ngũ kỹ sư trẻ của nước ta, nâng cao chất lượng của nguồn nhân lực.

### 1.3. ĐỐI TƯỢNG NGHIÊN CỨU

Đề tài này tập trung nghiên cứu chủ yếu là phần mềm hệ thống nhúng linux trên kit ARM KM9260. Những kiến thức về hệ điều hành nhúng bao gồm driver và application được hiện thực hóa thông qua các bài thực hành trên kit thí nghiệm. Các bài thực hành được xây dựng từ đơn giản đến nâng cao điều khiển các thiết bị ngoại vi được tích hợp

trên kit để người học có thể hiểu và ứng dụng vào các dự án lớn hơn ngoài thực tế.

Do đây là tài liệu hướng dẫn lập trình hệ thống nhúng dựa trên nền hệ điều hành linux nên những kiến thức về lập trình hợp ngữ, những chương trình không mang tính chất hệ điều hành không được đề cập và áp dụng trong những bài thực hành của giáo trình.

### 1.4. GIỚI HẠN ĐỀ TÀI

Nội dung của đề tài (giáo trình) bao gồm 3 vấn đề:

- *Lập trình hệ thống nhúng căn bản:*

Trình bày những kiến căn bản nhất về hệ thống nhúng, làm cơ sở để người học tiếp cận phần lập trình hệ thống nhúng nâng cao. Bao gồm lý thuyết về hệ điều hành Linux, lý thuyết về hệ thống nhúng, hướng dẫn các phần mềm hỗ trợ trong quá trình sử dụng kit nhúng, hướng dẫn nạp các phần mềm hệ thống nhúng và biên dịch phần mềm ứng dụng.

- Phần lý thuyết về hệ điều hành Linux sẽ trình bày các kiến thức căn bản của Linux: phân vùng đĩa trong Linux, cách truy xuất phân vùng trong Linux, các thư mục hệ thống trong Linux, màn hình Terminal. Ngoài ra phần này còn trình bày các thao tác cơ bản trong Linux như: khởi động lại hệ thống, tắt hệ thống, tạo thư mục tập tin, sao chép thư mục tập tin, di chuyển thư mục tập tin, phân quyền quản lý tập tin thư mục, .... Cuối cùng phần này trình bày trình soạn thảo trong Linux là trình soạn thảo VI.
- Phần lý thuyết hệ thống nhúng sẽ trình bày về phần cứng của kit nhúng KM9260, trình bày về các phần mềm hệ thống của kit nhúng bao gồm Romcode, Bootstrapcode, U-boot, Kernel, Rootfs.
- Phần hướng dẫn các phần mềm hỗ trợ trong quá trình sử dụng kit nhúng sẽ trình bày các phần mềm cần thiết như: Vmware Workstation, Samba, Putty, tftpd32, SSH SECURE SHELL CLIENT.
- Và phần cuối cùng sẽ hướng dẫn nạp các phần mềm hệ thống vào các vùng nhớ tương ứng trên kit để kit có thể chạy hệ điều hành nhúng thành công. Phần này còn trình bày cách biên dịch, chạy một chương trình ứng dụng và cách biên dịch, cài đặt một driver.

- *Lập trình hệ thống nhúng nâng cao:*

Bao gồm kiến thức về mối quan hệ giữa hai lớp user và kernel trong hệ thống phần mềm nhúng.

- Đối với lớp user, đề tài cung cấp những thông tin các hàm hỗ trợ trong quản lý thời gian thực của hệ thống, các lệnh hỗ trợ lập trình đa tuyến và tiến trình. Cho người học có một tầm nhìn tổng quát trong phương pháp lập trình bằng hệ điều hành. Đề tài không đi sâu tìm hiểu vấn đề xung đột trong truy xuất dữ liệu giữa các tuyến và các tiến trình với nhau mà chỉ nêu những kiến thức đủ để thực hiện các bài tập ví dụ trong đề tài.
- Đối với lớp kernel, đề tài cung cấp những thông tin chủ yếu về driver, vai trò, vị trí và phân loại driver trong hệ thống phần mềm nhúng. Đi sâu vào tìm hiểu character device driver, các hàm giao diện, các bước hoàn chỉnh để viết và đưa một driver (loại character) hoạt động trong hệ thống. Bên cạnh đó đề tài giáo trình còn trình bày các hàm thao tác trong gpio, quản lý và truy xuất thời gian thực trong kernel làm cơ sở cho điều khiển các thiết bị ngoại vi.
- Thực hành giao tiếp các thiết bị ngoại vi:  
Bao gồm nhiều bài thực hành khác nhau, mỗi bài sẽ điều khiển một loại thiết bị ngoại vi. Đề tài sẽ cung cấp mã chương trình ví dụ cho cả hai phần driver và application cho mỗi bài ví dụ. Các thiết bị ngoại vi là những linh kiện hiển thị đơn giản, các linh kiện đo lường, điều khiển; các chuẩn giao tiếp như UART, I2C, 1-Wire.

Do hạn chế về thời gian nghiên cứu nên đề tài chỉ đưa ra các chương trình ví dụ nhỏ dưới dạng module chưa kết hợp thành một dự án lớn để có thể hiểu hết sức mạnh mà một hệ thống nhúng có thể làm. Đề tài chỉ nhằm cung cấp cho người đọc những kiến thức lập trình nhúng căn bản và cụ thể nhất để từ đó làm cơ sở để lập trình những ứng dụng lớn hơn không phải chỉ một mà là tập hợp những kỹ sư khác nhau cùng nghiên cứu phát triển.

## 1.5. DÀN Ý NGHIÊN CỨU

### 1.5.1. Lập trình nhúng căn bản:

- *Giới thiệu về vi điều khiển AT91SAM9260*: cung cấp cho người học các kiến thức cần thiết về vi điều khiển phục vụ cho việc nghiên cứu sử dụng kit nhúng. Phần này bao gồm:
  - Họ vi điều khiển ARM.
  - Vi điều khiển AT91SAM9260:
    - Đặc điểm chính

- Sơ đồ chân của AT91SAM9260
- Nguồn cấp cho AT91SAM9260
- Bảng đồ vùng nhớ của AT91SAM9260
- *Lý thuyết về hệ điều hành Linux:* Cung cấp cho người học các kiến thức cơ bản về Linux để ứng dụng cho hệ điều hành nhúng. Phần này bao gồm:
  - Kiến thức về Linux
  - Các thao tác cơ bản trên hệ điều hành Linux
  - Trình soạn thảo VI
- *Lý thuyết về hệ thống nhúng:* Cung cấp cho người học các kiến thức ban đầu về hệ thống nhúng. Phần này bao gồm:
  - Phần cứng hệ thống nhúng trên kit km9260
  - Phần mềm hệ thống nhúng trên kit km9260
- *Các phần mềm hỗ trợ trong quá trình sử dụng kit:* Hướng dẫn các thao tác sử dụng phần mềm và chức năng của từng phần mềm. Phần này bao gồm:
  - Chương trình máy tính ảo Vmware Workstation
  - Chương trình Samba
  - Chương trình Putty
  - Chương trình tftpd32
  - Chương trình SSH SECURE SHELL CLIENT
- *Hướng dẫn nạp phần mềm hệ thống và biên dịch phần mềm ứng dụng cho kit:* Phần này bao gồm:
  - Trình biên dịch chéo Cross Toolchains
  - Các bước biên dịch Kernel
  - Chỉnh sửa Kernel
  - Các biến môi trường và lệnh cơ bản của U-boot
  - Cài đặt cho hệ thống
  - Các bước biên dịch Driver và cài đặt Driver
  - Các bước biên dịch chương trình ứng dụng và chạy chương trình ứng dụng

### 1.5.2. Lập trình nhúng nâng cao:

- *Lập trình user application:* Cung cấp cho người học những kiến thức cần thiết để có thể viết một chương trình ứng dụng chạy trong lớp user. Phần này bao gồm những nội dung:

- *Chương trình helloworld*: Đây là chương trình đơn giản, cung cấp kiến thức về các hàm đơn giản trong C (hàm printf(), hàm exit()) và khắc phục những lỗi xảy ra trong quá trình biên dịch chương trình user sao cho có thể chạy ổn định.
- *Trì hoãn thời gian trong user*: Cung cấp cho người học những kiến thức về trì hoãn thời gian trong lớp user có đơn vị từ giây đến nano giây. Người học được thao tác với hàm định thời gian tác động alarm(), các hàm truy xuất và thao tác thông tin thời gian thực của hệ thống linux.
- *Lập trình đa tiến trình trong user*: Người học được làm quen với khái niệm tiến trình; cách quản lý tiến trình; các hàm tạo lập, thay thế và nhân đôi tiến trình; ... trong linux.
- *Lập trình đa tuyến trong user*: Trình bày các vấn đề về tuyến; phân biệt giữa tuyến và tiến trình; Các hàm tạo lập, đồng bộ hóa hoạt động; Những ưu và nhược điểm so với tiến trình; ... trong linux.
- *Lập trình trong lớp kernel driver*: Cung cấp cho người học những kiến thức chuyên sâu về driver loại character theo đó họ có thể tự mình viết và đọc hiểu các driver khác ngoài thực tế. Phần này bao gồm những nội dung sau:
  - *Hệ thống nhúng mối quan hệ giữa driver và application trong phần mềm hệ thống nhúng*: Nhắc lại những kiến thức về định nghĩa; cấu trúc và vai trò từng thành phần trong hệ thống nhúng. So sánh về vai trò, nêu mối quan hệ giữa driver và application trong hệ thống nhúng phục vụ cho việc phân phối nhiệm vụ thực thi cho hai thành phần này trong hệ thống.
  - *Các loại driver nhận dạng từng loại driver trong linux*: Xác định vai trò của driver; phân loại; cách quản lý driver; các hàm tương tác với số định danh driver trong linux;
  - *Character device driver*: Cung cấp những kiến thức đầy đủ về character device driver bao gồm: Định nghĩa, cách thức tạo số định danh thiết bị, cấu trúc và các hàm để đăng ký character driver vào hệ thống linux.
  - *Các giao diện chuẩn trong driver*: Trình bày cấu trúc, giải thích các tham số lệnh của giao diện hàm trong driver linux như read(), write() và ioctl() phục vụ cho việc viết hoàn chỉnh một character device driver;
  - *Các bước để viết một character driver*: Căn cứ vào những kiến thức lý

thuyết được trình bày trong những bài trước. Bài này rút ra những bước cần thiết cần phải tiến hành để lập trình thành công một character driver. Nhắc lại thao tác biên dịch driver trong linux.

- *Helloworld driver*: Đưa ra ví dụ minh họa lập trình một character driver hoàn chỉnh mang tên helloworld.ko. Driver này sử dụng tất cả những giao diện hàm đã được giới thiệu đồng thời còn lập trình sẵn một chương trình user application liên kết giữa người dùng với driver, sử dụng tất cả những chức năng mà nó hỗ trợ.
- *Các hàm trong GPIO*: Giới thiệu cho người học những kiến thức về điều khiển các cổng vào ra gpio. Bao gồm quy định số chân, các hàm khởi tạo, truy xuất các chân trong CHIP vi điều khiển.
- *Trì hoãn thời gian trong kernel*: Cũng tương tự như trong phần lập trình user, bài này cung cấp những kiến thức về quản lý thời gian trong kernel; các hàm truy xuất thời gian thực; thao tác khởi tạo ngắt dùng timer trong kernel.

### 1.5.3. Lập trình giao tiếp phần cứng:

Bao gồm những bài thực hành riêng lẻ, được sắp xếp theo thứ tự từ dễ đến khó. Mỗi bài sẽ điều khiển một hoặc nhiều thiết bị ngoại vi tùy theo yêu cầu của bài toán. Hầu hết đều ứng dụng những kiến thức đã học trong phần lập trình user application và lập trình kernel driver. Các bài thực hành bao gồm:

- Điều khiển LED đơn;
- Điều khiển 2 LED 7 đoạn rời;
- Điều khiển nhiều LED 7 đoạn bằng phương pháp quét;
- Điều khiển LCD 16x2;
- Giao tiếp điều khiển GPIO ngõ vào đếm xung;
- Điều khiển LED ma trận;
- **Điều khiển ADC0809;**
- Điều khiển module I2C tích hợp trên CHIP AT91SAM9260;
- Điều khiển module ADC tích hợp trên CHIP AT91SAM9260;
- **Điều khiển module UART tích hợp trên CHIP AT91SAM9260;**
- (...);

### **1.6. TÌNH HÌNH NGHIÊN CỨU**

*Trong nước:*

Hiện nay đã có rất nhiều giáo trình nghiên cứu sâu vào nguyên lý hoạt động của hệ thống nhúng cả về phần cứng lẫn phần mềm. Những quyển giáo trình này chủ yếu trình bày những lý thuyết chung chung không đi cụ thể vào một hệ thống nào. Chính vì thế sau khi nghiên cứu xong, người học khó có thể áp dụng ngay vào thực tế mà đòi hỏi phải có một quá trình nghiên cứu chuyên sâu về hệ thống đó.

Trên các kênh truyền thông cũng có nhiều diễn đàn bàn luận về đề tài lập trình phần mềm hệ thống nhúng trên kit, nhưng đa số nhỏ lẻ và không theo trình tự logic từ dễ đến khó. Điều này khiến cho người mới bắt đầu muốn nghiên cứu và viết được một ứng dụng phải trải qua quá trình thử sai tổng hợp từ nhiều nguồn thông tin khác nhau. Thậm chí còn có thể đi sai hướng nghiên cứu.

*Ngoài nước:*

Bản về lập trình hệ thống nhúng, đã có nhiều sách xuất bản từ rất lâu. Nhưng nội dung được trình bày rất phong phú, người đọc cần phải có kiến thức chuyên môn sâu, nhất là kiến thức trong lĩnh vực ngoại ngữ chuyên ngành mới có thể đọc và lĩnh hội hết kiến thức bên trong. Điều này đòi hỏi sinh viên nghiên cứu phải có khả năng tìm hiểu và vốn ngoại ngữ dồi dào.

Các giáo trình đã có đa số trình bày những ví dụ tổng quát theo hướng tất cả các hệ thống đều có thể thực thi. Chưa có những ví dụ về lập trình phần cứng điều khiển các thiết bị ngoại vi cụ thể.

Đề tài được nghiên cứu nhằm khắc phục những nhược điểm trên. Nghiên cứu một đối tượng cụ thể sẽ giúp cho người học ứng dụng ngay những gì đã lĩnh hội vào thực tiễn. Từ đó khắc sâu lý thuyết, có thể áp dụng vào những hệ thống tương tự.

### **1.7. Ý NGHĨA THỰC TIỄN.**

Lập trình hệ thống nhúng là một lĩnh vực mới có triển vọng của nước ta trong những năm gần đây. Thúc đẩy sự phát triển trong lĩnh vực này ngoài việc thu hút đầu tư thì việc phát triển đội ngũ kỹ sư có kinh nghiệm là điều quan trọng nhất.

Đề tài được biên soạn với những bài thực hành xen kẽ lý thuyết được sắp xếp theo từng mức độ giúp cho người học không chỉ nắm vững lý thuyết mà còn có thể thao tác



điều khiển trên mạch phần cứng. Vì đa số các hệ thống nhúng hoạt động trên nền hệ điều hành linux đều theo một chuẩn chung nên khi tiếp xúc với các hệ thống khác người học có thể tự mình nghiên cứu tìm hiểu.

Đề tài được thiết kế theo kit KM9260, là kit thí nghiệm lập trình nhúng được tích hợp nhiều ngoại vi và các chân giao tiếp vào ra, thuận lợi cho việc kết nối với các bộ thí nghiệm khác hoặc lắp đặt vào các bộ phận chuyên biệt. Điều này làm tiết kiệm chi phí thiết kế, tận dụng những bộ thí nghiệm hiện có, ứng dụng lập trình thành công có thể áp dụng vào dự án nhanh chóng.

Đề tài khi được áp dụng vào chương trình giảng dạy sẽ góp phần làm phổ biến thêm môn lập trình hệ thống nhúng cho những sinh viên đang theo học tại các trường đại học. Giúp họ có được những kiến thức nền tảng, định hướng đam mê ban đầu để tiếp tục nghiên cứu trong các môi trường chuyên nghiệp sau này.

## CHƯƠNG II

# LẬP TRÌNH NHÚNG CĂN BẢN

# Lời đầu chương

Lập trình nhúng là một vấn đề tương đối khó, đòi hỏi người nghiên cứu học tập phải có sự kiên nhẫn và lòng đam mê thật sự mới có thể hiểu và tiến xa trong lĩnh vực này. Với mục đích định hướng cho người học có sự đam mê ban đầu trong môn lập trình hệ thống nhúng, nhóm thực hiện đề tài đã trình bày một cách chi tiết những nội dung cốt yếu và cần thiết sao cho người học có thể ứng dụng ngay những gì đã học vào thực tế, tránh trường hợp lý thuyết suông quá tổng quát, nhằm tạo sự hứng thú trong quá trình học tập.

Trong chương IILập trình nhúng căn bản, như tên gọi của nó, chương này trình bày những kiến thức rất căn bản về phần cứng hệ thống nhúng có liên quan đến kit thí nghiệm KM9260 như: Vi điều khiển AT91SAM9260, cấu trúc các linh kiện, hệ điều hành nhúng linux, các phần mềm hỗ trợ lập trình, ...

Để nghiên cứu những nội dung trong chương này đạt hiệu quả cao nhất, người học phải có những kiến thức và kỹ năng nền tảng sau:

- Kiến thức tổng quát về kỹ thuật số, lý thuyết vi xử lý, hiểu và sử dụng thành thạo một loại vi xử lý đơn giản (chẳng hạn 89x51, PIC, ...).
- Sử dụng thành thạo những thao tác cơ bản trong PC tương đương trình độ A.
- Kiến thức tổng quát về lý thuyết hệ thống nhúng, hệ điều hành, ...

Chương này được trình bày theo cấu trúc từng bài học, mỗi bài học có thể ngắn hoặc dài tùy theo nội dung kiến thức trình bày. Nội dung kiến thức trong mỗi bài là một vấn đề riêng biệt nhằm giúp cho người học lĩnh hội một cách chắc chắn vai trò của bài học trong quá trình tìm hiểu nghiên cứu lập trình chương trình ứng dụng trong hệ thống nhúng.

*Cấu trúc các bài học bao gồm:*

- **Bài 1:** Tổng quan về vi điều khiển AT91SAM9260;

Bài này cung cấp cho người học những kiến thức về vi điều khiển ARM, sơ đồ chân, bản đồ vùng nhớ của vi điều khiển sử dụng trong kit KM9260. Đây là những kiến thức quan trọng, giúp cho người học hiểu được nguyên lý kết nối phần cứng, cách sử dụng và khai báo địa chỉ vùng nhớ trong phần mềm, ... được đề cập trong những bài học khác.

- **Bài 2:** Một số thao tác cơ bản trên hệ điều hành Linux;

Hệ điều hành Linux được trình bày trong bài này là một hệ điều hành dành cho máy tính PC và cũng có một số đặc điểm giống với hệ điều hành chạy trên hệ thống nhúng. Mục đích là cung cấp cho người học những kiến thức về quản lý bộ

nhớ, các thao tác từ đơn giản đến phức tạp trong môi trường shell của hệ điều hành Linux và một số các phần mềm hỗ trợ biên soạn chương trình ứng dụng khác.

- **Bài 3:** Hệ thống phần cứng và phần mềm nhúng trong kit KM9260;

Bài này giới thiệu một số vấn đề về cấu trúc phần cứng và phần mềm trong kit KM9260 nhằm giúp cho người học hiểu rõ tên, vị trí, vai trò của từng linh kiện, sơ đồ chân kết nối, ... bên cạnh đó là nguyên lý và vai trò hoạt động của từng thành phần trong hệ thống phần mềm nhúng, ... Những kiến thức này có vai trò quan trọng trong lập trình giao tiếp phần cứng và sửa chữa hệ thống trong quá trình hoạt động nếu phát sinh ra lỗi.

- **Bài 4:** Phần mềm hỗ trợ lập trình nhúng;

Cung cấp cho người học kỹ năng sử dụng những chương trình hỗ trợ quá trình lập trình nhúng như: Chương trình máy tính ảo, chương trình hỗ trợ nạp chương trình thông qua cổng USB, thiết bị hiển thị đầu cuối qua cổng COM, phần mềm giao tiếp mạng. Các phần mềm này hỗ trợ việc đăng nhập, lập trình, biên dịch, và thực thi chương trình ứng dụng trong kit.

- **Bài 5:** Thao tác trên phần mềm hệ thống nhúng KM9260;

Sau khi nghiên cứu Bài 4, người học sẽ thực hành sử dụng những thao tác đã được học để thao tác biên dịch và nạp các thành phần trong hệ thống phần mềm nhúng bao gồm rootfilesystem, uboot, kernel, driver và phần mềm ứng dụng để các thành phần này có thể chạy ổn định trên phần cứng hệ thống nhúng kit KM9260.

Những vấn đề được trình bày trong chương này không nhằm giải thích chuyên sâu về nguyên lý hoạt động của hệ thống nhúng mà cung cấp cho người học những kiến thức cụ thể có liên quan đến kit KM9260, phục vụ cho việc lập trình các ứng dụng thực tế giao tiếp giữa kit với các thiết bị ngoại vi trong chương sau.

Do tính riêng biệt trong từng bài học, nên người học có thể bỏ qua một số bài khi cảm thấy đã nắm vững các nội dung được giới thiệu trong mỗi bài học. Hoặc có thể làm tài liệu tham khảo khi chưa nắm vững một nội dung nào đó.

Sau đây chúng tôi sẽ trình bày lần lượt từng nội dung đã được liệt kê.

## **BÀI 1**

# **TỔNG QUAN VỀ VI ĐIỀU KHIỂN AT91SAM9260**

Thành phần quang trọng nhất cấu tạo nên một hệ thống nhúng chính là vi xử lý, vi điều khiển. Vi xử lý, vi điều khiển quyết định đến tính năng, hiệu quả ứng dụng của hệ thống nhúng. Vì vậy trước khi nghiên cứu về hệ thống nhúng thì chúng ta phải có một kiến thức căn bản về vi xử lý của hệ thống nhúng.

Kit nhúng KM9260 được xây dựng dựa trên nền vi điều khiển AT91SAM9260. Là vi điều khiển 32 bit thuộc họ ARM9 tiết kiệm năng lượng.

Trong phần này sẽ trình bày sơ lược các kiến thức về AT91SAM9260. Vì lập trình cho hệ thống nhúng không đặt nặng kiến thức về cấu trúc của vi xử lý. Nên trong phần này chỉ trình bày những kiến thức cần thiết cho quá trình sử dụng kit sau này như: sơ đồ chân của AT91SAM9260, nguồn cung cấp cho AT91SAM9260, bản đồ quản lý vùng nhớ của AT91SAM9260.

Trong vi điều khiển có tích hợp nhiều module ngoại vi khác nhau, kiến thức về từng module sẽ được trình bày trong mỗi bài thí nghiệm với các thiết bị ngoại vi khác nhau.

## I. Họ vi điều khiển ARM:

Cấu trúc ARM (viết tắt từ tên gốc là **Acorn RISC Machine**) là một loại cấu trúc vi xử lý 32bit kiểu RISC được sử dụng rộng rãi trong các thiết kế nhúng. Do có đặc điểm tiết kiệm năng lượng, các bộ CPU ARM chiếm ưu thế trong các sản phẩm điện tử di động, mà với các sản phẩm này việc tiêu tán công suất thấp là một mục tiêu thiết kế quan trọng hàng đầu.

Ngày nay, hơn 75% CPU nhúng 32bit là thuộc họ ARM, điều này khiến ARM trở thành cấu trúc 32bit được sản xuất nhiều nhất trên thế giới. CPU ARM được tìm thấy khắp nơi trong các sản phẩm thương mại điện tử, từ thiết bị cầm tay (PDA, điện thoại di động, máy đa phương tiện, máy trò chơi cầm tay, và máy tính cầm tay) cho đến các thiết bị ngoại vi máy tính (ổ đĩa cứng, bộ định tuyến để bàn.) Một nhánh nổi tiếng của họ ARM là các vi xử lý Xscale của Intel.

Việc thiết kế ARM được bắt đầu từ năm 1983 trong một dự án phát triển của công ty máy tính Acorn.

Nhóm thiết kế, dẫn đầu bởi Roger Wilson và Steve Furber, bắt đầu phát triển một bộ vi xử lý có nhiều điểm tương đồng với Kỹ thuật MOS 6502 tiên tiến. Acorn đã từng sản xuất nhiều máy tính dựa trên 6502, vì vậy việc tạo ra một chip như vậy là một bước tiến đáng kể của công ty này.

Nhóm thiết kế hoàn thành việc phát triển mẫu gọi là ARM1 vào năm 1985, và vào năm sau, nhóm hoàn thành sản phẩm “thực” gọi là ARM2. ARM2 có tuyến dữ liệu 32bit, không gian địa chỉ 26bit tức cho phép quản lý đến 64 Mbyte địa chỉ và 16 thanh ghi 32bit. Một trong những thanh ghi này đóng vai trò là bộ đếm chương trình với 6 bit cao nhất và 2 bit thấp nhất lưu giữ các cờ trạng thái của bộ vi xử lý.

Có thể nói ARM2 là bộ vi xử lý 32bit khả dụng đơn giản nhất trên thế giới, với chỉ gồm 30.000 transistor (so với bộ vi xử lý lâu hơn bốn năm của Motorola là 68000 với khoảng 68.000 transistor). Sự đơn giản như vậy có được nhờ ARM không có vi chương trình (mà chiếm khoảng 1/4 đến 1/3 trong 68000) và cũng giống như hầu hết các CPU vào thời đó, không hề chứa cache. Sự đơn giản này đưa đến đặc điểm tiêu thụ công suất thấp của ARM. Thế hệ sau ARM3 được tạo ra với 4KB cache và có chức năng được cải thiện tốt hơn nữa.

Vào những năm cuối thập niên 80, hãng máy tính Apple Computer bắt đầu hợp tác với Acorn để phát triển các thế hệ lõi ARM mới. Công việc này trở nên quan trọng đến nỗi Acorn nâng nhóm thiết kế trở thành một công ty mới gọi là Advanced RISC Machines. Vì lý do đó bạn thường được giải thích ARM là chữ viết tắt của Advanced RISC

Machines thay vì Acorn RISC Machine. Advanced RISC Machines trở thành công ty ARM Limited khi công ty này được đưa ra sàn chứng khoán London và NASDAQ năm 1998.

Kết quả sự hợp tác này là ARM6. Mẫu đầu tiên được công bố vào năm 1991 và Apple đã sử dụng bộ vi xử lý ARM 610 dựa trên ARM6 làm cơ sở cho PDA hiệu Apple Newton. Vào năm 1994, Acorn dùng ARM 610 làm CPU trong các máy vi tính RiscPC của họ.

Trải qua nhiều thế hệ nhưng lõi ARM gần như không thay đổi kích thước. ARM2 có 30.000 transistors trong khi ARM6 chỉ tăng lên đến 35.000. Ý tưởng của nhà sản xuất lõi ARM là sao cho người sử dụng có thể ghép lõi ARM với một số bộ phận tùy chọn nào đó để tạo ra một CPU hoàn chỉnh, một loại CPU mà có thể tạo ra trên những nhà máy sản xuất bán dẫn cũ và vẫn tiếp tục tạo ra được sản phẩm với nhiều tính năng mà giá thành vẫn thấp.

Thế hệ thành công nhất có lẽ là ARM7TDMI với hàng trăm triệu lõi được sử dụng trong các máy điện thoại di động, hệ thống video game cầm tay, và Sega Dreamcast. Trong khi công ty ARM chỉ tập trung vào việc bán lõi IP, cũng có một số giấy phép tạo ra bộ vi điều khiển dựa trên lõi này.

Dreamcast đưa ra bộ vi xử lý SH4 mà chỉ mượn một số ý tưởng từ ARM (tiêu tán công suất thấp, tập lệnh gọn ...) nhưng phần còn lại thì khác với ARM. Dreamcast cũng tạo ra một chip xử lý âm thanh được thiết kế bởi Yamaha với lõi ARM7. Bên cạnh đó, Gameboy Advance của Nintendo, dùng ARM7 TDMI ở tần số 16,78 MHz.

Hãng DEC cũng bán giấy phép về lõi cấu trúc ARM (đôi khi chúng ta có thể bị nhầm lẫn vì họ cũng sản xuất ra DEC Alpha) và sản xuất ra thế hệ Strong ARM. Hoạt động ở tần số 233 MHz mà CPU này chỉ tiêu tốn khoảng 1 watt công suất (những đời sau còn tiêu tốn ít công suất hơn nữa). Sau những kiện tụng, Intel cũng được chấp nhận sản xuất ARM và Intel đã nắm lấy cơ hội này để bổ sung vào thế hệ già cỗi i960 của họ bằng Strong ARM. Từ đó, Intel đã phát triển cho chính họ một sản phẩm chức năng cao gọi tên là Xscale.

Bảng 2.1: Các họ vi xử lý ARM

Họ	Lõi	Tốc độ xử lý tối đa (MHz)	Kiến trúc
<b>ARM7TDMI</b>	ARM7TDMIS	16.8 MHz	V4T
	ARM710T	40 MHz	
	ARM720T	59.8 MHz	
	ARM740T		
	ARM7EJS		
<b>ARM9TDMI</b>	ARM9TDMI		V4T
	ARM920T	180 MHz	
	ARM922T		
	ARM940T		
<b>ARM9E</b>	ARM946ES		V5TE Và V5TEJ
	ARM966ES		
	ARM968ES		
	ARM926EJS	180 MHz	
	ARM996HS		
<b>ARM10E</b>	ARM1020E		V5TE Và V5TEJ
	ARM1022E		
	ARM1026EJS		
<b>ARM11</b>	ARM1136J(F)S		V6
	ARM1156T2(F)S		V6T2
	ARM1176JZ(F)S		V6Z
	ARM11 MPCore		V6
<b>Cortex</b>	CortexA8	1 GHz	V7A
	CortexR4	600 MHz	V7M
	CortexM3	100MHz	V7R
<b>XScale</b>	80200/IOP310/IOP315		
	80219		
	IOP321		



	IOP33x		
	PXA210/PXA250		
	PXA255	400 MHz	
	PXA26x		
	PXA27x	624 MHz	
	PXA800(E)F		
	Monahans	1.25 GHz	
	PXA900		
	IXC1100		
	IXP2400/IXP2800		
	IXP2850		
	IXP2325/IXP2350		
	IXP42x		
	IXP460/IXP465		

Để đạt được một thiết kế gọn, đơn giản và nhanh, các nhà thiết kế ARM xây dựng nó theo kiểu nổi cứng không có vi chương trình, giống với bộ vi xử lý 8bit 6502 đã từng được dùng trong các máy vi tính trước đó của hãng Acorn.

Cấu trúc ARM bao gồm các đặc tính của RISC như sau:

- Cấu trúc nạp/lưu trữ.
- Không cho phép truy xuất bộ nhớ không thẳng hàng (bây giờ đã cho phép trong lõi Arm v6).
- Tập lệnh trực giao.
- File thanh ghi lớn gồm  $16 \times 32\text{bit}$ .
- Chiều dài mã máy cố định là 32 bit để dễ giải mã và thực hiện pipeline, để đạt được điều này phải chấp nhận giảm mật độ mã máy.
- Hầu hết các lệnh đều thực hiện trong vòng một chu kỳ đơn.

So với các bộ vi xử lý cùng thời như Intel 80286 và Motorola 68020, trong ARM có một số tính chất khá độc đáo như sau:

- Hầu hết tất cả các lệnh đều cho phép thực thi có điều kiện, điều này làm giảm việc phải viết các tiêu đề rẽ nhánh cũng như bù cho việc không có một bộ dự đoán rẽ nhánh.

- Trong các lệnh số học, để chỉ ra điều kiện thực hiện, người lập trình chỉ cần sửa mã điều kiện.
- Có một thanh ghi dịch đóng thùng 32bit mà có thể sử dụng với chức năng hoàn hảo với hầu hết các lệnh số học và việc tính toán địa chỉ.
- Có các kiểu định địa chỉ theo chỉ số rất mạnh.
- Có hệ thống con thực hiện ngắt hai mức ưu tiên đơn giản nhưng rất nhanh, kèm theo cho phép chuyển từng nhóm thanh ghi.

## **II. Vi điều khiển AT91SAM9260:**

### **1. Đặc điểm chính:**

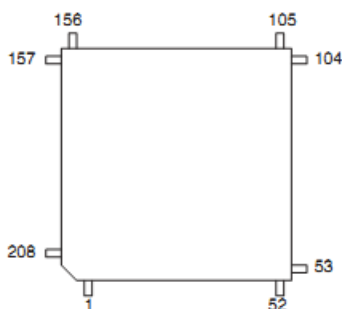
AT91SAM9260 có lõi là bộ vi xử lý ARM926EJS thuộc họ ARM9E hỗ trợ tập lệnh Thumb (tập lệnh nén từ 32 bit xuống 16 bit) và ARM và có thể hoạt động với tốc độ 180Mhz.

- Bộ nhớ:
  - Giao diện bus 32 bit hỗ trợ 4 bank SDRAM, bộ nhớ tĩnh, CompactFlash, NandFlash.
  - Có 2 SRAM được tích hợp bên trong chip hoạt động ở tốc độ cao. Mỗi SRAM có dung lượng 4 Kbyte.
  - Có 1 ROM bên trong chip có dung lượng 32 Kbyte chứa chương trình hệ thống phục vụ cho việc khởi động hệ thống nhúng do nhà sản xuất sẵn,
- Ngoại vi:
  - Bên trong chip tích hợp 4 kênh ADC 10 bit.
  - Có 2 bộ truyền dữ liệu không đồng bộ UART (Universal Asynchronous Receive/Transmitter)
  - Có 4 bộ truyền dữ liệu không đồng bộ/đồng bộ USART (Universal Synchronous Asynchronous Receive/Transmitter).
  - Một giao diện truyền nhận dữ liệu theo chuẩn I2C (chuẩn hai dây).
  - Một bộ điều khiển nối tiếp đồng bộ.
  - Hai giao tiếp SPI hỗ trợ hai chế độ Master/Slaver.
  - Một giao diện SD/MMC (dùng để đọc thẻ nhớ).
  - Một bộ điều khiển 10/100 Mbps Ethernet MAC.
  - Một bộ điều khiển truyền nhận USB và một bộ truyền nhận USB Device.
  - Ba bộ Timer/Counter.
- Hệ thống:

- Có 22 kênh DMA (Direction Memory Access).
- Khởi động từ NAND Flash, SDCard, DataFlash hoặc Serial DataFlash.
- Có bộ điều khiển Reset.
- Ma trận bus AHB 6 lớp 32 bit hoạt động ở tần số 90Mhz.
- Một PLL (bộ nhân tần số) cho hệ thống và một cho USB.
- Hai tín hiệu xung đồng bộ ngoài có thể lập trình được.
- Có bộ điều khiển ngắt cao cấp và sửa lỗi.
- Tích hợp bộ dao động nội RC.
- Cho phép chọn tần số tiết kiệm năng lượng 32,768 Khz và bộ dao động chính 3 đến 20 Mhz.
- Cho phép cài đặt chế độ timer, watchdog timer, realtime timer.
- Khối xuất nhập:
- Ba bộ điều khiển Input/Output song song 32 bit.
- Có 96 đường Input/Output đa mục đích.

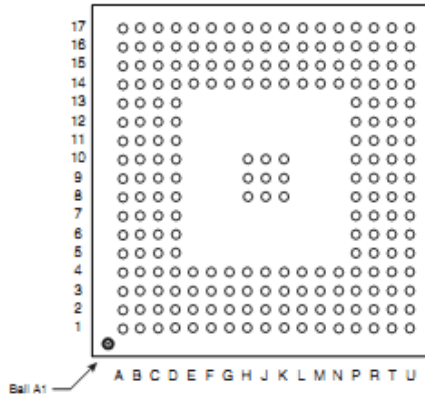
## 2. Sơ đồ chân của AT91SAM9260:

- AT91SAM9260 có bề ngoài hình vuông, có hai dạng sơ đồ chân: một dạng 208 chân và một dạng 217 chân.
- Dạng 208 chân: Các chân được đưa ra xung quanh của gói nhựa bọc lõi và được đặt tên theo số từ 1 đến 208. Hình dạng được minh họa như sau:



*Hình 2.1: Sơ đồ chân của AT91SAM9260 208 chân*

- Dạng 217 chân: các chân được đưa ra ở phía dưới của gói nhựa bọc lõi. Hình dạng được minh họa như sau:



*Hình 2.2: Sơ đồ chân của A91SAM9260 217 chân*

Vị trí của các chân này được xác định bằng cách ghép các chữ cái của hàng với các số của cột và chân gần với chấm đen làm dấu trên thân của chip là chân A1.

Vì số lượng chân của chip rất lớn nên trong giáo trình này không trình bày cụ thể chức năng của từng chân. Nếu người học muốn tìm hiểu thêm về sơ đồ chân của AT92SAM9260 thì có thể tham khảo Datasheet của AT91SAM9260.

### **3. Nguồn cấp cho AT91SAM9260:**

Đặc tính tiết kiệm năng lượng là một trong những thế mạnh của họ vi xử lý ARM. Để tiết kiệm được năng lượng, bên cạnh một kiến trúc lõi tiết kiệm năng lượng thì vi điều khiển còn được thiết kế có nhiều nguồn cấp với các mức điện áp khác nhau.

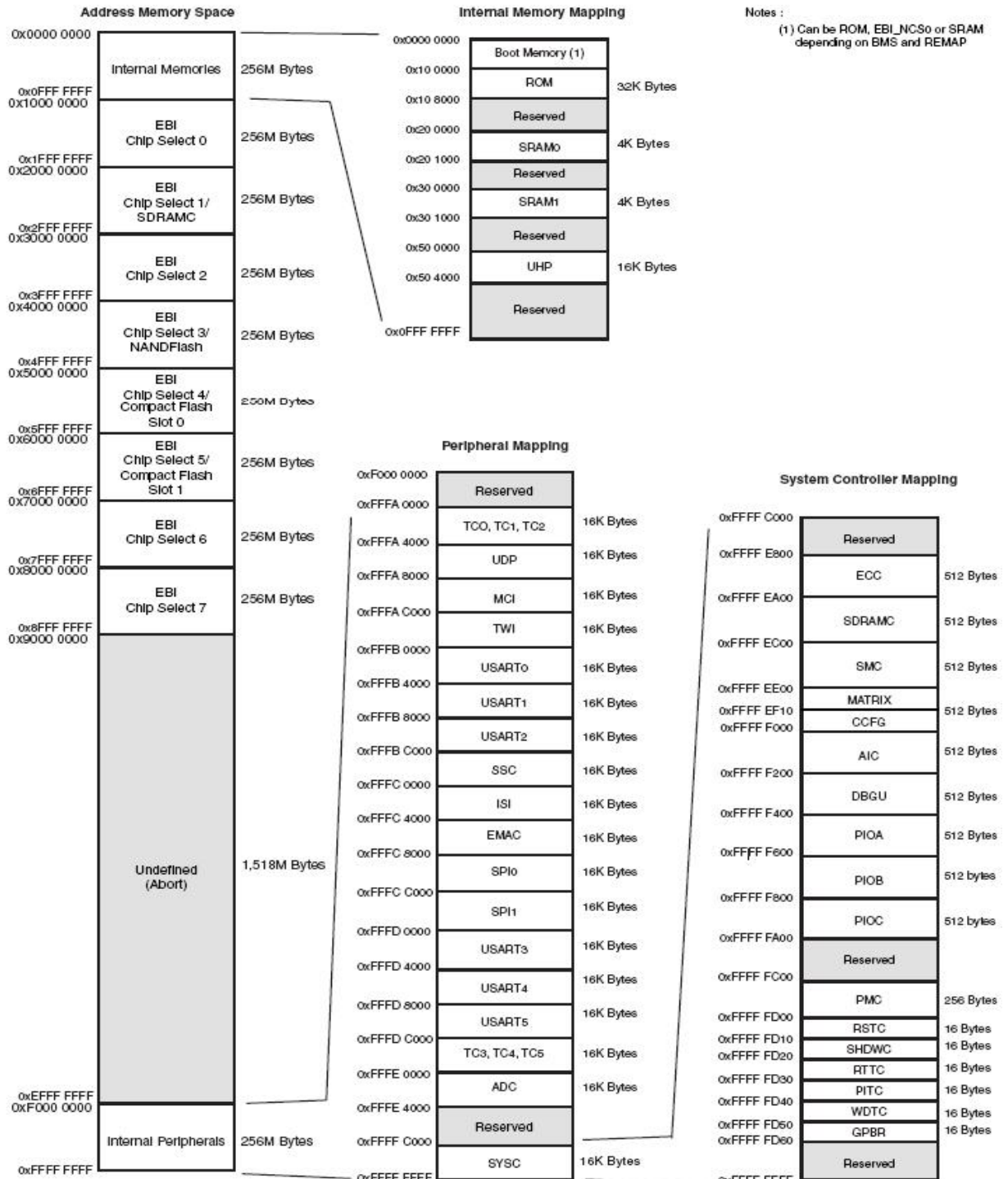
AT91SAM9260 cần có hai nguồn cấp với các mức điện áp 1,8V và 3,3V. Hai nguồn cấp này được cung cấp tới các chân nguồn của chip để có thể cấp nguồn cho từng modul tích hợp trong chip. Cụ thể như sau:

- Chân VDDCORE: chân này cấp nguồn để nuôi lõi vi xử lý với điện áp là 1,8V.
- Chân VDDIOM: cung cấp nguồn cho bộ giao tiếp Input/Output mở rộng. Điện áp cấp đến chân này là 1,8 V hoặc 3,3 V được chọn bởi phần mềm.
- Chân VDDIOP0: cung cấp cho bộ giao tiếp Input/Output ngoại vi (USB, Micro SD) với điện áp 3,3V.
- Chân VDDDBU: cung cấp điện áp cho bộ dao động chậm (dao động nội RC và thạch anh 32,768 Khz) với điện áp 1,8V.
- Chân VDDPLL: cung cấp điện áp cho bộ dao động chính và bộ nhân tần số với điện áp là 1,8V.
- Chân VDDANA: cung cấp điện áp cho bộ ADC với điện áp 3,3V.

### **4. Bảng đồ vùng nhớ của AT91SAM9260:**

AT91SAM9260 có thể quản lý một vùng nhớ lên đến 4Gbyte địa chỉ, vùng nhớ này được phân thành 16 bank với mỗi bank là 256Mbyte địa chỉ. Mỗi bank này sẽ quản lý bộ

nhớ của một thiết bị nhớ nhất định hoặc một vùng nhớ nhất định. Mặc dù thiết bị nhớ hoặc vùng nhớ không chứa dữ liệu hoặc không được dùng trong hệ thống, nhưng AT91SAM9260 vẫn để dành vùng nhớ đó cho thiết bị nhớ hoặc vùng nhớ của nó. Cụ thể sẽ được trình bày trong hình sau:



Hình 2.3: Bảng đồ quản lý vùng nhớ của AT91SAM9260

Theo hình trên thì chúng ta có thể thấy bảng đồ vùng nhớ của AT91SAM9260 được chia ra thành nhiều bank để quản lý. Bank trên cùng là bank 0, tiếp theo là bank 1, cho

đến cuối cùng là bank 15. Các địa chỉ nằm ở bên trái các cột chính là các địa chỉ vật lý. Bên phải các cột là dung lượng của vùng nhớ tương ứng.

Chúng ta có thể thấy rằng mỗi thiết bị nhớ hoặc vùng nhớ đều được vi xử lý quản lý trong một khoảng địa chỉ nhất định. Nếu thiết bị nhớ được quản lý bởi bank 1 thì ô nhớ đầu tiên của thiết bị nhớ đó (có địa chỉ là 0x00000000) sẽ được vi xử lý đọc với địa chỉ là 0x10000000, ô nhớ thứ hai (có địa chỉ 0x00000001) sẽ được đọc với địa chỉ là 0x10000001. Tương tự thì ô nhớ cuối cùng của thiết bị nhớ đó (có địa chỉ 0x0FFFFFFF) được vi xử lý đọc với địa chỉ là 0x1FFFFFFF. Như vậy, chúng ta có thể thấy địa chỉ thực tế của ô nhớ được vi xử lý đọc với một địa chỉ hoàn toàn khác. Để thuận tiện cho việc tìm hiểu sau này và tránh nhầm lẫn về địa chỉ chúng ta sẽ tìm hiểu về các loại địa chỉ:

Địa chỉ vật lý: là địa chỉ mà vi xử lý gán cho vùng nhớ cần được quản lý. Theo hình trên thì địa chỉ vật lý chính là các địa chỉ nằm ở bên trái các cột.

Địa chỉ đoạn: là địa chỉ vật lý tại ô nhớ đầu tiên của vùng nhớ. Theo hình trên thì địa chỉ đoạn của bank 1 là 0x10000000.

Địa chỉ offset: là khoảng cách tính từ ô nhớ đang xét đến địa chỉ đoạn. Như vậy địa chỉ offset có thể hiểu là địa chỉ thực tế của thiết bị nhớ. Địa chỉ offset của ô nhớ thứ nhất của thiết bị nhớ là 0x00000000, của ô nhớ thứ hai là 0x00000001 và của ô nhớ cuối cùng là 0x0FFFFFFF.

Bank 0 dùng để quản lý bộ nhớ nội của vi xử lý. ROM được quản lý với địa chỉ vật lý từ 0x00100000 đến 0x00108000 với dung lượng là 32Kbyte. Tương tự vùng nhớ SRAM0 cũng được quản lý từ 0x00200000 đến 0x00201000 ..., các vùng nhớ reserved là các vùng nhớ dự trữ.

Từ Bank 1 đến bank 8 dùng để quản lý các thiết bị nhớ bên ngoài. Cụ thể, bank 2 quản lý SDRAMC với chân chọn chip là NCS1. Bank 4 quản lý NandFlash với chân chọn chip là NCS3/NANDCS. Khi vi xử lý truy cập các thiết bị nhớ này thì vi xử lý sẽ đưa tín hiệu chọn vùng nhớ ra các chân NCS0 đến NCS7 chọn chip tương ứng.

Bank 15 quản lý vùng nhớ của các thiết bị ngoại vi.

Các vùng nhớ có màu đậm là các vùng nhớ dự trữ hoặc là chưa sử dụng đến.

### **III. Kết luận:**

AT91SAM9260 là một vi xử lý có tốc độ hoạt động cao 180 Mhz, nhiều ngoại vi, tiết kiệm năng lượng. Vi điều khiển này sử dụng hai nguồn điện áp là 1,8V và 3,3V cho các modun khác nhau. Ngoài ra vi điều khiển này có thể quản lý một vùng nhớ lên đến 4Gbyte và được chia thành nhiều vùng quản lý ứng với các vùng nhớ khác nhau.

Như vậy, sau khi đọc xong chương này người học đã có một nền kiến thức về vi điều khiển AT91SAM9260. Các kiến thức này sẽ hỗ trợ cho quá trình lập trình ứng dụng sau này của người học.

## BÀI 2

# MỘT SỐ THAO TÁC CƠ BẢN TRÊN HỆ ĐIỀU HÀNH LINUX.

## A-TỔNG QUAN VỀ LINUX

Hệ điều hành Linux là một hệ điều hành mã nguồn mở và còn khá non trẻ so với hệ điều hành Window. Nhưng với những ưu điểm của mình: miễn phí, linh hoạt, uyển chuyển, độ an toàn cao, thống nhất trên các hệ thống phần cứng, mã nguồn mở... Linux đã phát triển mạnh mẽ, đa dạng và chiếm được sự tin tưởng của các lập trình viên.

Các hệ thống nhúng hiện nay đa số chạy trên hệ điều hành Linux. Vì vậy muốn ứng dụng được hệ thống nhúng thì điều tiên quyết là người dùng phải biết sử dụng hệ điều hành Linux.

Phần này sẽ trình bày về các khái niệm về phân vùng đĩa, cách truy xuất ổ đĩa trong Linux, giới thiệu các thư mục hệ thống, màn hình terminal, tập lệnh cơ bản của hệ điều hành Linux. Đây là các kiến thức cơ bản đầu tiên người học cần có để bước vào thế giới kỳ diệu của Linux.

### I. Quản lý bộ nhớ trong Linux:

#### 1. Phân vùng đĩa:

Trong các máy vi tính hiện nay, đĩa cứng là thiết bị lưu trữ được sử dụng phổ biến nhất. Đĩa cứng là một loại đĩa từ, ra đời sau đĩa mềm, có dung lượng lớn 32GB, 80GB và với kỹ thuật tiên tiến như hiện nay thì dung lượng của đĩa cứng đã đạt đến đơn vị TB.

Vì dung lượng của đĩa cứng lớn nên để thuận lợi trong việc quản lý nội dung lưu trên đĩa, đồng thời để tăng hiệu suất sử dụng đĩa cứng thì đĩa cứng sẽ được chia thành nhiều phân vùng hay còn gọi là Partition. Có hai loại Partition là Primary Partition và Extended Partition.

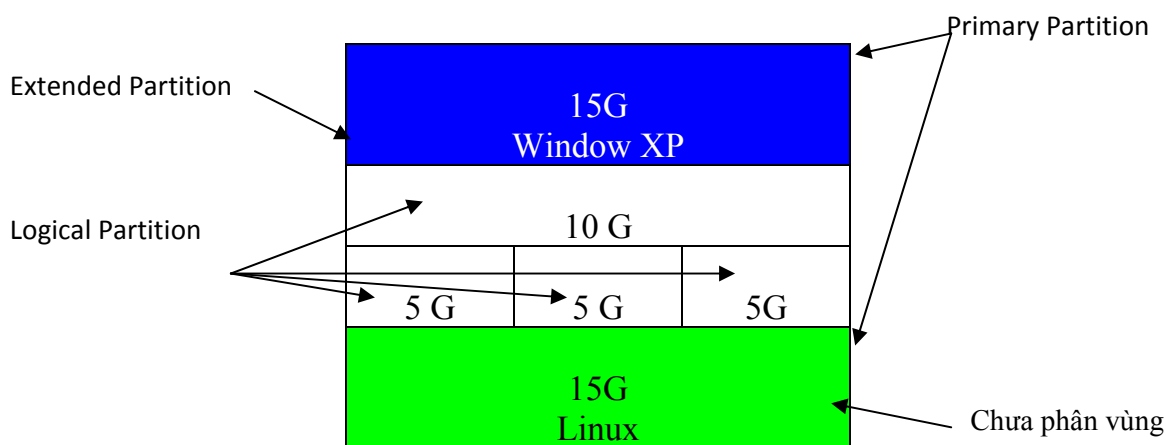
Primary Partition là phân vùng chính, phân vùng này thường được dùng để cài hệ điều hành. Theo nguyên tắc thì trên một đĩa cứng ta chia được tối đa là 4 Primary Partition. Như vậy với một đĩa cứng thì ta chỉ có thể cài tối đa là 4 hệ điều hành.

Lưu ý rằng : ta không thể chia Primary Partition thành các phân vùng nhỏ hơn và Primary Partition ngoài việc dùng để cài đặt hệ điều hành thì ta có thể dùng Primary Partition để lưu trữ dữ liệu.

Extended Partition là phân vùng mở rộng. Thực ra Extended Partition giống như Primary Partition, nhưng với Extended Partition ta có thể chia thành nhiều phân vùng nhỏ



hơn bên trong Extended Partition và được gọi là Logical Partition (Phân vùng logic). Extended Partition thường dùng để lưu trữ dữ liệu của người dùng.



Hình 2.4 : Phân vùng của đĩa cứng

Hình trên minh họa cho việc phân vùng các Partition trên một đĩa cứng 60G.

- Theo hình trên, phân vùng màu xanh dương 15G và màu xanh lá cây 15G là các Primary Partition. Hai phân vùng này chứa hệ điều hành. Mỗi Primary Partition chỉ được chứa một hệ điều hành, hai hệ điều hành có thể tồn tại trên cùng một đĩa cứng nếu chúng được cài đặt trên hai Primary Partition khác nhau và dung lượng đĩa cho phép.
- Phân vùng màu trắng 25G là Extended Partition thường được dùng để lưu trữ dữ liệu của người dùng. Phân vùng này được chia thành 4 phân vùng nhỏ khác là các Logical Partition có dung lượng lần lượt là 10G, 5G, 5G, 5G.
- Phân vùng màu xanh lơ là vùng đĩa chưa được phân vùng. Thông thường đối với các phân vùng này thì hệ thống mặc định là Extended Partition.
- Như vậy đối với đĩa cứng trên thì ta có 2 Primary Partition, 2 Extended Partition, 4 Logical Partition.

Mặc dù ta đã chia đĩa cứng thành các phân vùng, nhưng khi cài hệ điều hành lên đĩa cứng thì mỗi hệ điều hành lại nhận diện và truy xuất các phân vùng này theo cách khác nhau. Ví dụ : hệ điều hành Window thì nhận diện các phân vùng này là các ổ đĩa (đĩa C là Primary Partition, đĩa D,E,... là các Logical Partition) và truy xuất phân vùng này thông qua các ổ đĩa này. Còn đối với hệ điều hành Linux thì nhận diện các phân vùng này là các tập tin và việc truy xuất các phân vùng này cũng thông qua các tập tin này.

Kit KM9260 sử dụng hệ điều hành Linux, nên trong giáo trình này chỉ đề cập đến cách nhận diện và truy xuất phân vùng trên đĩa cứng của Linux.

## 2. Phân vùng của Linux :

Như đã nói ở trên, Linux nhận diện tất cả các loại đĩa và thiết bị là các tập tin nên việc truy xuất các đĩa và thiết bị cũng thông qua các tập tin này.

Đối với ổ đĩa mềm và các phân vùng của đĩa cứng Linux quy ước cách đặt tên như sau : ổ đĩa mềm thứ nhất là fd0, ổ đĩa mềm thứ hai là fd1 .... Đĩa cứng thứ nhất là hda, đĩa cứng thứ hai là hdb .... Nếu có dùng USB thì là sda, sdb, sdc, .... Các phân vùng chính (Primary Partition hay Extended Partition) được đánh số từ 1 đến 4. Các phân vùng Logical Partition được đánh số từ 5 trở lên.

*Ví dụ:* hda1 là phân vùng chính của đĩa cứng thứ nhất, hda2 là phân vùng mở rộng của đĩa cứng thứ nhất. Còn hda5, hda6, hda7 là các phân vùng Logical Partition trong Extended Partition.

Các tập tin truy xuất phân vùng này thường được Linux lưu trong thư mục /dev. Chúng ta lưu ý đến ký tự “/” trong Linux. Linux quy ước ký hiệu “/” là Primary Partition của hệ thống, là phân vùng chứa nhân hệ điều hành Linux. Tất cả các đường dẫn trong Linux đều phải được bắt đầu bởi ký hiệu này. Như vậy, các tập tin truy xuất phân vùng và ổ đĩa nằm ở trong thư mục /dev có nghĩa là thư mục dev nằm trong Primary Partition, các tập tin fd0, fd1, ..., hda0, hda1 nằm trong thư mục dev nhưng khi truy xuất các tập tin này là ta truy xuất các phân vùng tương ứng trên đĩa cứng chứ không phải chỉ truy xuất Primary Partition. Ký hiệu “/” còn được dùng làm dấu phân cách thư mục. Ví dụ: ta có đường dẫn: /home/Nhan/Kernel. Theo đường dẫn này ta thấy thư mục home nằm trong thư mục gốc (Primary Partition) và trong thư mục home có thư mục Nhan, trong thư mục Nhan có thư mục Kernel. Để phân cách giữa các thư mục ta dùng ký hiệu “/”.

## 3. Cách truy xuất ổ đĩa trong Linux:

Trong Linux không có khái niệm ổ đĩa như trong Window. Nên việc truy xuất các phân vùng Linux sử dụng phép gán (mount) tên ổ đĩa với tên một thư mục bất kỳ.

Khi khởi động hệ điều hành, Linux gán cho phân vùng chính (Primary Partition) ký hiệu “/”, các thông tin của phân vùng khác được Linux đặt vào trong thư mục /dev của phân vùng chính. Nếu muốn truy xuất các phân vùng này thì ta thực hiện thao tác kết gán bằng lệnh mount.

*Ví dụ:*

Trong thư mục /dev có các tập tin:

fd0: ổ mềm thứ nhất

hda1: Primary Partition

hda5, hda6, hda7: Logical Partition thứ nhất, thứ hai, thứ 3 nằm trong Extended Partition.

Trong thư mục /home có các thư mục:

diamem  
odiachinh  
odiaphu1  
odiaphu2  
odiaphu3

Muốn truy xuất các phân vùng và ổ đĩa thì ta phải dùng lệnh sau:

```
mount /dev/fd0 /home/diamem  
mount /dev/hda1 /home/odiachinh  
mount /dev/hda5 /home/odiaphu1  
mount /dev/hda6 /home/odiaphu2  
mount /dev/hda7 /home/odiaphu3
```

Lúc này các thao tác sao chép, cắt, dán file vào trong các thư mục diamem, odiachinh, odiaphu1, odiaphu2, odiaphu3 cũng tương đương với việc truy xuất dữ liệu tương ứng vào đĩa mềm, Primary Partition và các Logical Partition.

Khi không muốn sử dụng phân vùng nữa ta có thể dùng lệnh umount để tháo kết gán:

```
umount diamem  
umount odiachinh  
umount odiaphu1  
umount odiaphu2  
umount odiaphu3
```

#### **4. Các thư mục trên Linux:**

Khi cài đặt hệ điều hành xong thì ta sẽ thấy trong Primary Partition có rất nhiều thư mục. Các thư mục này có các chức năng và mục đích sử dụng nhất định. Cơ bản một hệ thống Linux thường có các thư mục sau:

*/bin*: Thư mục này chứa các file chương trình thực thi (dạng nhị phân) và file khởi động của hệ thống.

*/boot*: Các file ảnh (image file) của kernel dùng cho quá trình khởi động thường đặt trong thư mục này.

*/dev*: Thư mục này chứa các file thiết bị. Trong thế giới Linux các thiết bị phần cứng được xem như là các file.

*/ect*: Thư mục này chứa các file cấu hình toàn cục của hệ thống. Có thể có nhiều thư mục con trong thư mục này nhưng nhìn chung chúng chứa các file script để khởi động hay phục vụ cho mục đích cấu hình chương trình trước khi chạy.

*/home*: Thư mục này chứa các thư mục con đại diện cho mỗi user khi đăng nhập. Nơi đây tựa như ngôi nhà của người dùng. Khi người quản trị tạo tài khoản cho bạn, họ cấp cho bạn một thư mục con trong */home*. Bạn hoàn toàn có quyền sao chép, xóa file, tạo thư mục con trong thư mục home của mình mà không ảnh hưởng đến các người dùng khác.

*/lib*: Thư mục này chứa các file thư viện .so hoặc .sa. Các thư viện C và các thư viện liên kết động cần cho chương trình khi chạy và cho toàn hệ thống.

*/lostfound*: Khi hệ thống khởi động hoặc khi chạy chương trình fsck, nếu tìm thấy một chuỗi dữ liệu nào đó bị thất lạc trên đĩa cứng không liên quan đến tập tin, Linux sẽ gộp chúng lại và đặt trong thư mục này để nếu cần bạn có thể đọc và giữ lại dữ liệu bị mất.

*/mnt*: Thư mục này chứa các thư mục kết gán tạm thời đến các ổ đĩa hay thiết bị khác. Bạn có thể thấy trong */mnt* các thư mục con như *cdrom* hoặc *floppy*.

*/sbin*: Thư mục này chứa các file hay chương trình thực thi của hệ thống thường chỉ cho phép sử dụng bởi người quản trị.

*/tmp*: Thư mục này chứa các file tạm mà chương trình sử dụng chỉ trong quá trình chạy. Các file trong thư mục này sẽ được hệ thống dọn dẹp nếu không cần dùng đến nữa.

*/usr*: Thư mục này chứa rất nhiều thư mục con như */usr/bin* hay */usr/sbin*. Một trong những thư mục con quan trọng trong */usr* là */usr/local*, bên trong thư mục local này bạn có đủ các thư mục con tương tự ngoài thư mục gốc như *sbin*, *lib*, *bin*,.... Nếu bạn nâng cấp hệ thống thì các chương trình bạn cài đặt trong *usr/local* vẫn giữ nguyên và bạn không sợ chương trình bị mất mát. Hầu hết các ứng dụng Linux đều thích cài chương trình vào trong */usr/local*.

*/var*: Thư mục này chứa các file biến thiên bất thường như các file dữ liệu đột nhiên tăng kích thước trong một thời gian ngắn sau đó lại giảm kích thước xuống còn rất nhỏ. Điển hình là các file dùng làm hàm đợi chứa dữ liệu cần đưa ra máy in hoặc các hàng đợi chứa mail.

*/usr/include* hoặc */usr/local/include*: chứa các file khai báo hàm (header) cần dùng khi biên dịch chương trình nguồn sử dụng các thư viện của hệ thống.

*/usr/src*: Chứa mã nguồn.

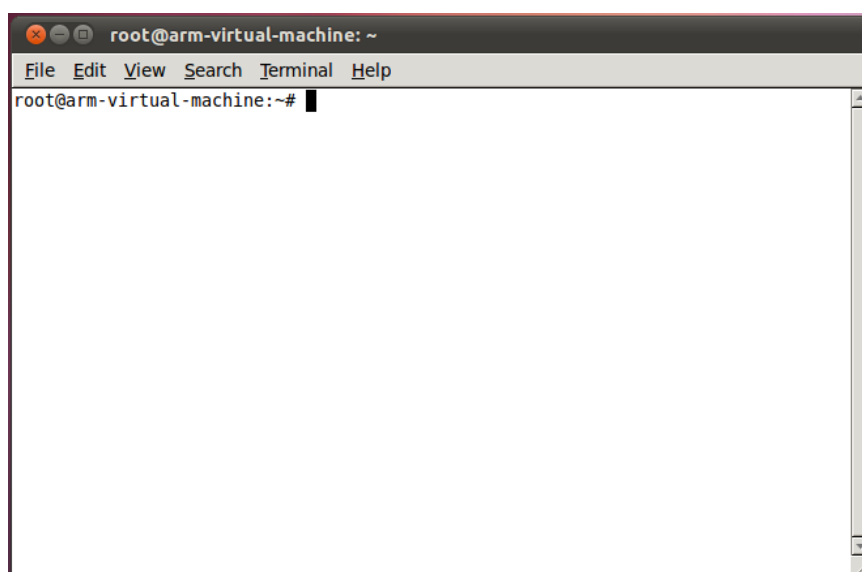
*/usr/man*: Chứa tài liệu hướng dẫn.

## II. Màn hình terminal:

Ngày nay với sự phát triển hiện đại của công nghệ sản xuất chip và sự tiến bộ trong việc lập trình hệ điều hành thì chúng ta đã rất là quen thuộc với các hệ điều hành có hỗ trợ đồ họa như Window XP, Window 7, Linux Ubutu. Với chức năng hỗ trợ đồ họa của mình, các hệ điều hành này giúp cho người sử dụng thuận tiện trong việc thực hiện các thao tác (chỉ cần dùng chuột để thực hiện lệnh) và tạo ra giao diện tương tác thân thiện đối với người dùng. Người dùng không cần phải biết tập lệnh của hệ điều hành mà chỉ cần biết thao tác thực hiện. Ví dụ: để sao chép một tập tin trong Ubutu, người dùng không cần phải biết lệnh sao chép của hệ điều hành là gì, mà chỉ cần biết thao tác sao chép file là được: click chuột phải vào file, chọn copy, sau đó đến vùng cần copy tới, click chuột phải và chọn past into folder.

Tuy nhiên trong thực tế không phải hệ điều hành nào cũng có hỗ trợ đồ họa và không phải hệ thống nào nhúng nào cũng có hỗ trợ đồ họa. Vì vậy mà hầu hết các hệ điều hành hiện nay vẫn còn giữ phương thức giao tiếp với người sử dụng thông qua các dòng lệnh. Tồn tại song song với phương thức giao tiếp qua đồ họa, phương thức giao tiếp thông qua các dòng lệnh phát huy tác dụng trong các hệ thống nhúng như KIT KM9260, các trường hợp sự cố mà chức năng đồ họa không thể hoạt động được.

Trong Linux Ubutu hay các phiên bản Linux có hỗ trợ đồ họa thì phương thức giao tiếp thông qua các dòng lệnh được thực hiện trên màn hình Terminal



*Hình 2.5: Màn hình Terminal của Linux*

KIT KM9260 sử dụng hệ điều hành Linux không có hỗ trợ đồ họa, vì vậy mà việc giao tiếp thông qua các dòng lệnh là hết sức quan trọng. Khi khởi động thì màn hình Terminal của Linux sẽ tự động được kích hoạt.

### III. Tập lệnh cơ bản của Linux:

- *Nhóm lệnh hệ thống:*

Lệnh	Cú pháp	Chức năng
ls	ls <thư mục>	Liệt kê nội dung tập tin thư mục
find	find <tên file>	Tìm file
cp	cp file1 file2	Sao chép file
cat	cat > <tên file> cat <tên file>	Tạo file văn bản mới Hiển thị nội dung file
vi	vi <tên file>	Soạn thảo nội dung file
man	man <từ khóa>	Hướng dẫn về lệnh
mv	mv file1 file2	Đổi tên file, thư mục, di chuyển file
rm	rm file rm -r <tên thư mục>	Xóa file Xóa cây thư mục
mkdir	mkdir <tên thư mục>	Tạo thư mục mới
rmdir	rmdir <tên thư mục>	Xóa thư mục rỗng
clear	clear	Xóa màn hình
cd	cd <đường dẫn>	Di chuyển đến thư mục khác
chmod	chmod <tham số><tên file>	Đổi thuộc tính file, thư mục
uname	uname -r	Xem phiên bản của hệ điều hành
pwd	pwd	Xem đường dẫn thư mục hiện hành
ps	ps	Xem thông tin về tiến trình
kill	kill <số PID >	Hủy tiến trình
gunzip	gunzip <tên file>	Giải nén file
gzip	gzip <tên file>	Nén file
tar	tar <tham số> <tên file>	Nén và giải nén file
env	env	Xem thông tin về biến môi trường
export	export <tên biến>= <nội dung của biến>	Thiết lập biến môi trường
echo	echo <tên biến>	In chuỗi ra màn hình

df	df	Xem dung lượng đĩa
fsck	fsck	Kiểm tra đĩa và hệ thống file

*Bảng 2.2: Các lệnh hệ thống của Linux*

- *Nhóm lệnh quản lý tài khoản đăng nhập:*

Lệnh	Cú pháp	Chức năng
Useradd	user <tên tài khoản>	Thêm tài khoản người dùng
Userdel	userdel <tên tài khoản>	Xóa tài khoản người dùng
Groupadd	groupadd <tên nhóm>	Tạo nhóm mới
Groupdel	groupdel <tên nhóm>	Xóa nhóm

*Bảng 2.3: Các lệnh quản lý tài khoản của Linux*

Cách sử dụng chi tiết các lệnh này sẽ được trình bày kỹ trong phần tiếp theo là: các thao tác cơ bản trên hệ điều hành Linux.

#### **IV. Kết luận:**

Linux quản lý các vùng nhớ bằng các tập tin trong thư mục /dev. Người dùng truy xuất các vùng nhớ này bằng cách kết gán các tập tin này với thư mục bất kỳ trong Linux.

Trong Linux chúng ta thao tác các lệnh thông qua màn hình terminal hoặc chương trình giao diện X-Window. Nhưng thao tác lệnh thông qua màn hình terminal được sử dụng nhiều hơn đối với các hệ thống nhúng không hỗ trợ đồ họa như Kit nhúng KM9260..

Như vậy, sau khi đọc xong phần này người học đã phần nào có khái niệm về hệ điều hành Linux. Các kiến thức này sẽ là nền tảng cho người học trong những phần sau. Trong phần tiếp theo chúng ta sẽ tìm hiểu kỹ hơn về các lệnh trong Linux.

# A-CÁC THAO TÁC CƠ BẢN

Như đã nêu trong phần trước, người dùng thường thao tác với hệ điều hành Linux trên các Kit nhúng thông qua màn hình terminal bằng cách ra lệnh cho hệ điều hành. Vì vậy để dùng được Linux thì chúng ta phải biết được các lệnh trong Linux.

Trong phần này sẽ trình bày các lệnh của Linux hỗ trợ cho các thao tác cơ bản như: liệt kê thư mục tập tin, tạo, xóa thư mục tập tin, sao chép thư mục tập tin, di chuyển và đổi tên thư mục tập tin, kết gán ổ đĩa, thay đổi thư mục hiện hành, ... và một số lệnh của hệ thống..

## I. Nội dung:

### 1. Khởi động lại (reset) hệ thống và tắt hệ thống:

- Để khởi động lại hệ thống ta dùng lệnh:

```
$ reset
```

- Để tắt hệ thống ta dùng lệnh:

```
$ poweroff
```

### 2. Sử dụng tài liệu hướng dẫn man:

Man là một chương trình có sẵn trong Linux có chức năng dò tìm các thông tin mà bạn đăng nhập vào từ dòng lệnh (gọi là khóa hay keyword). Nếu tìm thấy, nội dung chi tiết mô tả về khóa hoặc những thông tin liên quan đến khóa sẽ được hiển thị đầy đủ để bạn tham khảo.

- Bạn có thể gọi trình man theo cú pháp sau:

```
/$ man <keyword>
```

- Để thoát khỏi tài liệu man chúng ta có thể gõ q và nhấn Enter

**Ví dụ:** để tìm hiểu về nội dung và cách sử dụng lệnh ls chúng ta gõ

```
/$ man ls
```

Khi đó tài liệu man về lệnh ls sẽ hiện ra





Hình 2.6: Trình man hướng dẫn lệnh ls

- Trong tài liệu man sẽ liệt kê tên, cú pháp, chức năng của lệnh mà ta cần tìm. Ngoài ra, tài liệu man còn liệt kê ra các tham số có thể đi cùng với lệnh.

**Ví dụ:** như lệnh ls thì ta có thể thấy tài liệu man liệt kê ra hai tham số là -a và -A. Nếu chúng ta dùng lệnh

```
/ $ ls -a
```

Thì chương trình sẽ liệt kê tất cả tập tin và thư mục có trong thư mục gốc và hai dấu . và .. . Còn nếu ta dùng lệnh

```
/ $ ls -A
```

Thì chương trình sẽ liệt kê tất cả các tập tin và thư mục có trong thư mục gốc nhưng sẽ không liệt kê hai dấu . và ..

- Trình man phân các hướng dẫn ra làm thành từng đoạn (session) với những chủ đề khác nhau gồm:

Mục	Tên đề mục	Nội dung
1	User command	Các lệnh thông thường của hệ điều hành
2	System call	Các hàm kernel của hệ thống
3	Subroutines	Các hàm thư viện
4	Devices	Các hàm truy xuất và xử lý file thiết bị
5	File format	Các hàm định dạng file
6	Games	Các lệnh liên quan đến trò chơi
7	Miscell	Các hàm linh tinh
8	Sys. Admin	Các lệnh quản trị hệ thống

*Bảng 2.4: Các nhóm lệnh trình man hỗ trợ*

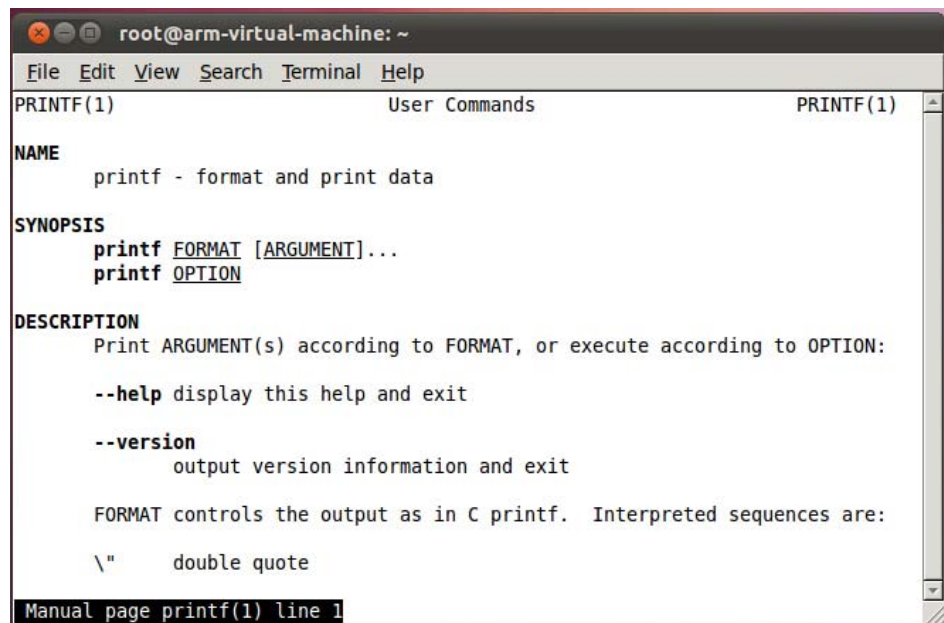
Thông thường nếu chúng ta gọi man và không chỉ định thêm gì cả ngoài từ khóa cần tìm thì man sẽ tìm từ khóa trong phân đoạn 1 (user command). Tuy nhiên chúng ta có thể yêu cầu man tìm trong một phân đoạn khác. Ví dụ từ khóa printf vừa là một lệnh của hệ điều hành vừa là một hàm của C. Nếu chúng ta muốn tìm printf liên quan đến hệ điều hành (user command) chúng ta sẽ gõ:

```
/$ man printf
```

Hoặc

```
/$ man 1 printf
```

Khi đó trình man sẽ liệt kê thông tin của lệnh printf liên quan đến hệ điều hành như sau:



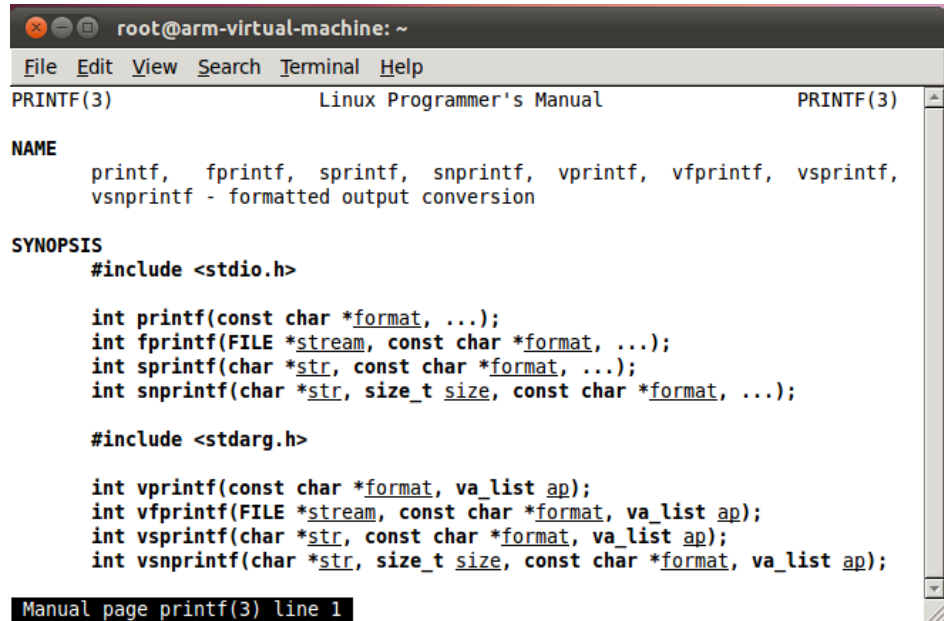
```
root@arm-virtual-machine: ~
File Edit View Search Terminal Help
PRINTF(1) User Commands PRINTF(1)
NAME
    printf - format and print data
SYNOPSIS
    printf FORMAT [ARGUMENT]...
    printf OPTION
DESCRIPTION
    Print ARGUMENT(s) according to FORMAT, or execute according to OPTION:
    --help display this help and exit
    --version output version information and exit
    FORMAT controls the output as in C printf. Interpreted sequences are:
    \" double quote
Manual page printf(1) line 1
```

*Hình 2.7: Trình man hướng dẫn lệnh printf của hệ điều hành*

Còn nếu chúng ta gõ:

```
/$ man 3 printf
```

Thì man sẽ liệt kê các thông tin của printf liên quan đến hàm thư viện C như sau :



```
root@arm-virtual-machine: ~
File Edit View Search Terminal Help
PRINTF(3) Linux Programmer's Manual PRINTF(3)

NAME
printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf,
vsnprintf - formatted output conversion

SYNOPSIS
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>

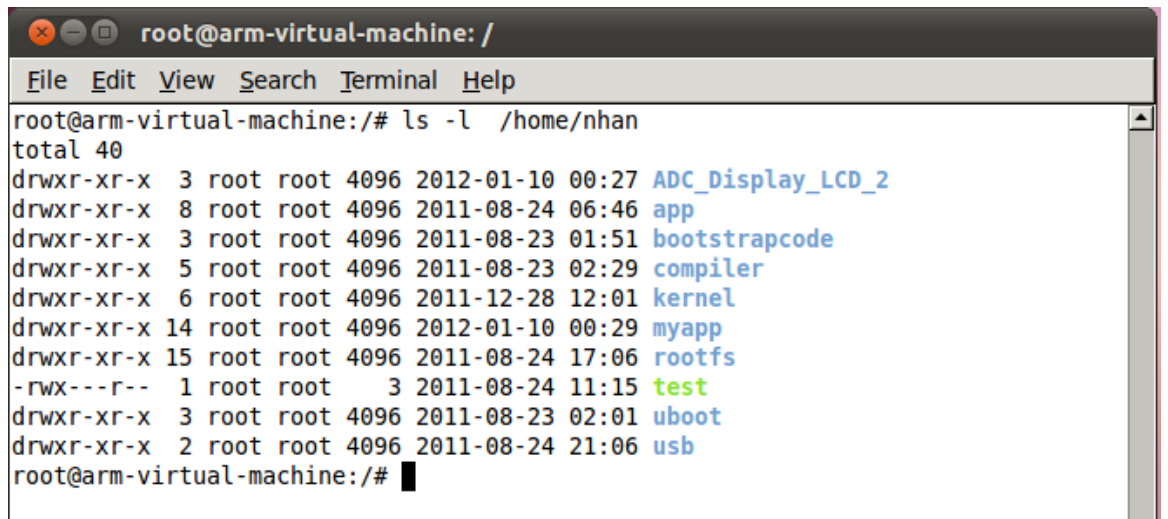
int vprintf(const char *format, va_list ap);
int vfprintf(FILE *stream, const char *format, va_list ap);
int vsprintf(char *str, const char *format, va_list ap);
int vsnprintf(char *str, size_t size, const char *format, va_list ap);

Manual page printf(3) line 1
```

Hình 2.8: Trình man hướng dẫn lệnh printf trong lệnh C

### 3. Liệt kê thư mục tập tin:

- Để liệt kê thư mục con và tập tin trong một thư mục thì ta dùng lệnh ls. Cú pháp của lệnh ls như sau :  
/\$ ls <đường dẫn và tên thư mục cần liệt kê thư mục con và nội dung>
- Ngoài ra lệnh ls còn được hỗ trợ thêm tham số :  
/\$ ls -l <đường dẫn và tên thư mục cần liệt kê thư mục con và nội dung>  
( liệt kê các thư mục con và tập tin theo dạng danh sách chi tiết)



```
root@arm-virtual-machine: /
File Edit View Search Terminal Help
root@arm-virtual-machine:/# ls -l /home/nhan
total 40
drwxr-xr-x 3 root root 4096 2012-01-10 00:27 ADC_Display_LCD_2
drwxr-xr-x 8 root root 4096 2011-08-24 06:46 app
drwxr-xr-x 3 root root 4096 2011-08-23 01:51 bootstrapcode
drwxr-xr-x 5 root root 4096 2011-08-23 02:29 compiler
drwxr-xr-x 6 root root 4096 2011-12-28 12:01 kernel
drwxr-xr-x 14 root root 4096 2012-01-10 00:29 myapp
drwxr-xr-x 15 root root 4096 2011-08-24 17:06 rootfs
-rwx---r-- 1 root root 3 2011-08-24 11:15 test
drwxr-xr-x 3 root root 4096 2011-08-23 02:01 uboot
drwxr-xr-x 2 root root 4096 2011-08-24 21:06 usb
root@arm-virtual-machine:/#
```

Hình 2.9: Lệnh ls liệt kê thư mục tập tin

### 4. Tạo và xóa thư mục tập tin:

- Để tạo thư mục ta dùng lệnh mkdir  
/\$ mkdir <đường dẫn và tên thư mục cần tạo>
- Để xóa thư mục thì ta dùng lệnh rmdir  
/\$ rmdir <đường dẫn và tên thư mục cần xóa>

**Ví dụ:** đang đứng ở ổ đĩa gốc, tạo thư mục /home/nhan/kernel, sau đó xóa lần lượt thư mục kernel và nhan.

Trước tiên là ta phải tạo thư mục nhan trong thư mục home. Vì thư mục home là thư mục có sẵn trong Linux nên ta không cần phải tạo lại nữa. Sau đó ta tạo thư mục kernel nằm trong thư mục nhan và xóa hai thư mục vừa tạo. Trình tự dòng lệnh như sau :

```
/$ mkdir /home/nhan
```

```
/$ mkdir /home/nhan/kernel
```

```
/$ rmdir /home/nhan/kernel
```

```
/$rmdir /home/nhan
```

- Để tạo ra một tập tin thì ta dùng lệnh cat  
/\$ cat > <đường dẫn và tên tập tin muốn tạo>
- Để xóa một tập tin thì ta dùng lệnh rm  
/\$ rm <đường dẫn và tên tập tin muốn xóa>
- Để xem nội dung một tập tin thì ta dùng  
/\$ cat < đường dẫn và tên tập tin muốn xem>

Ví dụ : đứng ở thư mục gốc tạo một tập tin /home/nhan/app/test1, sau đó xóa tập tin này. Thứ tự dòng lệnh như sau :

```
/$ mkdir /home/nhan
```

```
/$mkdir /home/nhan/app
```

```
/$ cat > /home/nhan/app/test1
```

Lúc này chúng ta có thể nhập nội dung cho tập tin test1. Nếu muốn thoát khỏi tập tin thì ta nhấn CtrlD

- Để xem nội dung của tập tin vừa tạo thì ta dùng  
/\$ cat /home/nhan/app/test1

Ta xóa tập tin test1

```
/$ rm /home/nhan/app/test1
```

Lưu ý: trong quá trình tạo thư mục và tập tin thì chúng ta có thể dùng lệnh ls để kiểm tra xem tập tin hay thư mục đã được tạo hay chưa.

## **5. Kết gán ổ đĩa và thư mục:**

Như đã nói ở các phần trước, để truy xuất được các thiết bị lưu trữ hoặc các phân vùng thì chúng ta phải kết gán thiết bị lưu trữ hay phân vùng này với file tương ứng trong Linux. Để làm được việc này thì ta dùng lệnh mount.

- Lệnh mount có cú pháp như sau:

`/$ mount -t vfstype devicefile mdir`

*Trong đó, devicefile là đường dẫn đến file mà Linux nhận diện thiết bị. Tùy chọn -t sẽ kết gán theo kiểu hệ thống file trên thiết bị do vfstype qui định. mdir là đường dẫn kết nối vào hệ thống thư mục của Linux. Hiện nay Linux có thể đọc được rất nhiều hệ thống file, vsftype có thể bao gồm những kiểu hệ thống file thông dụng sau :*

*Msdos : hệ thống file và thư mục theo bảng FAT16, FAT32 của DOS*

*Ntfs: định dạng hệ thống file NTFS của Window NT*

*Ext2: định dạng hệ thống file chuẩn của Linux và Unix*

*Nfs: định dạng hệ thống file truy xuất qua mạng*

*Iso9660: hệ thốn file theo chuẩn ISO*

- Để tháo kết gán ta dùng lệnh umount:

`/$ umount mdir`

### ***Ví dụ1:***

Để kết gán ổ đĩa A trong Linux để đọc các file theo hệ thống bảng FAT của DOS.

Trước tiên là ta tạo thư mục để kết gán

`/$ mkdir /dos`

*Gọi lệnh mount để kết gán thư mục*

`/$ mount -t msdos /dev/fd0 /dos`

*Nếu kết gán thành công thì kể từ bây giờ tất cả những gì chúng ta ghi vào thư mục /dos cũng tương đương như chúng ta ghi vào ổ đĩa A.*

*Để tháo kết gán ta dùng lệnh*

`/$ umount /dos`

### ***Ví dụ 2 :***

Kết gán ổ đĩa CD-ROM vào thư mục /mycdrom

`/$ mkdir /mycdrom`

`/$ mount -t iso9660 /dev/cdrom /mycdrom`

## **6. Thay đổi thư mục hiện hành:**

- Thư mục hiện hành là thư mục mà con trỏ chương trình đang ở đó. Thư mục hiện hành được xác định bởi thư mục ở trước dấu \$ trên dòng lệnh.

### ***Ví dụ:***

- Nếu bắt đầu dòng lệnh là dấu /\$ thư mục hiện hành là thư mục gốc
- Nếu bắt đầu dòng lệnh là /home\$ thì thư mục hiện hành là thư mục /home

- Để thay đổi thư mục hiện hành thì ta dùng lệnh cd. Cú pháp lệnh cd như sau:

`/$ cd <đường dẫn và tên thư mục muốn làm thư mục hiện hành>`

***Ví dụ:***

Ta đang ở thư mục hiện hành là thư mục gốc, ta thay đổi thư mục hiện hành là /home/nhan, giả sử các thư mục đã được tạo sẵn. Ta làm như sau:

```
/$ cd /home/nhan
```

```
/home/nhan$
```

*Ta thấy rằng phần bắt đầu dòng lệnh đã thay đổi /\$ thành /home/nhan\$ nghĩa là thư mục hiện hành đã được thay đổi.*

Để thay đổi thư mục hiện hành là home thì ta làm như sau:

```
/home/nhan$ cd /home
```

Hoặc

```
/home/nhan$ cd ..
```

## **7. Sao chép tập tin và thư mục:**

- Lệnh cp của Linux dùng để sao chép file. Cú pháp của lệnh cp như sau :  
`$ cp <đường dẫn và tên file cần sao chép> <đường dẫn và tên file muốn sao chép đến>`

***Ví dụ :*** trong thư mục /home/nhan/app có tập tin test1, chép tập tin test1 sang thư mục /home/nhan/test với tên test2. Giả sử các thư mục đã được tạo trước đó. Thứ tự các dòng lệnh như sau:

```
/$ cp /home/nhan/app/test1 /home/nhan/test/test2
```

- Lệnh cp còn dùng để sao chép cây thư mục. Để sao chép cây thư mục thì ta thêm tùy chọn -R vào lệnh cp.

***Ví dụ:*** sao chép thư mục /home/nhan/app sang thư mục /home ta làm như sau:

```
/$ cp -R /home/nhan/app /home
```

*Lúc này trong thư mục /home có thêm thư mục app*

## **8. Di chuyển và đổi tên tập tin, thư mục:**

- Chúng ta dùng lệnh mv để di chuyển hoặc đổi tên file. Cú pháp lệnh mv như sau :

```
/$ mv <đường dẫn và tên file cần di chuyển hoặc đổi tên> <đường dẫn và tên file mới>
```

***Ví dụ :***

- Di chuyển file /home/nhan/app/test1 sang thư mục /home. Ta làm như sau:

```
/$ mv /home/nhan/app/test1 /home
```

- Di chuyển file /home/nhan/app/test1 sang thư mục /home với tên test2. Ta làm như sau:

```
/ $ mv /home/nhan/app/test1 /home/test2
```

- Đổi tên file /home/nhan/app/test1 thành /home/nhan/app/test3. Ta làm như sau:

```
/ $ mv /home/nhan/app/test1 /home/nhan/app/test3
```

- Lệnh mv còn dùng để di chuyển thư mục.

**Ví dụ:** để di chuyển thư mục /home/nhan/app ra thư mục /home ta làm như sau:

```
/ $ mv /home/nhan/app/ /home
```

## 9. Phân quyền bảo vệ và truy xuất trên tập tin:

- Một tập tin hay thư mục trong Linux có các thuộc tính sau:

r: chỉ cho đọc

w: cho phép ghi vào tập tin

x: cho phép thực thi chương trình

-: không cho phép

- Đây là các thuộc tính về quyền truy xuất mà hệ điều hành dùng để bảo vệ file và thư mục. Chúng ta có 3 quyền chính trên một file hay thư mục như đã nêu trên. Mặc dù vậy các quyền này được chỉ định cho 3 đối tượng nữa đó là: người sở hữu tập tin (owner), nhóm sở hữu tập tin (group), những người sử dụng tập tin thông thường (other user).
- Tóm lại tập tin của chúng ta được kiểm soát bởi 3 quyền và truy xuất bởi 3 đối tượng khác nhau. Khi chúng ta dùng lệnh

```
/ $ ls -l
```

Thì Linux sẽ liệt kê các thuộc tính của file và thư mục trong thư mục gốc ở đầu các dòng

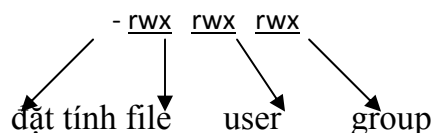
```

root@arm-virtual-machine: /
File Edit View Search Terminal Help
root@arm-virtual-machine: /# ls -l
total 136
drwxr-xr-x  2 root root  4096 2010-12-28 06:02 bin
drwxr-xr-x  3 root root  4096 2010-12-28 06:14 boot
drwxr-xr-x  2 root root  4096 2010-12-23 15:16 cdrom
drwxr-xr-x 17 root root  4440 2011-08-24 03:22 dev
drwxr-xr-x 138 root root 12288 2011-08-24 03:22 etc
-rwxr-xr-x  1 root root  7171 2011-08-23 00:57 helloworld
-rw-r--r--  1 root root    71 2011-08-23 00:57 helloworld.c
drwxr-xr-x  6 root root  4096 2011-08-23 06:29 home
lrwxrwxrwx  1 root root    33 2010-12-28 06:08 initrd.img -> boot/initrd.img
.6.38-10-generic
lrwxrwxrwx  1 root root    33 2010-12-23 15:45 initrd.img.old -> boot/initrd
mg-2.6.35-22-generic
drwxr-xr-x 19 root root 12288 2010-12-28 06:05 lib
drwx----- 2 root root 16384 2010-12-23 15:10 lost+found

```

Hình 2.10: Lệnh ls liệt kê chi tiết thư mục tập tin trong thư mục gốc

- Trên thuộc tính phân quyền của file, đối tượng được chia thành 3 nhóm từ trái sang phải, mỗi nhóm chứa 3 phân quyền đầy đủ như sau:



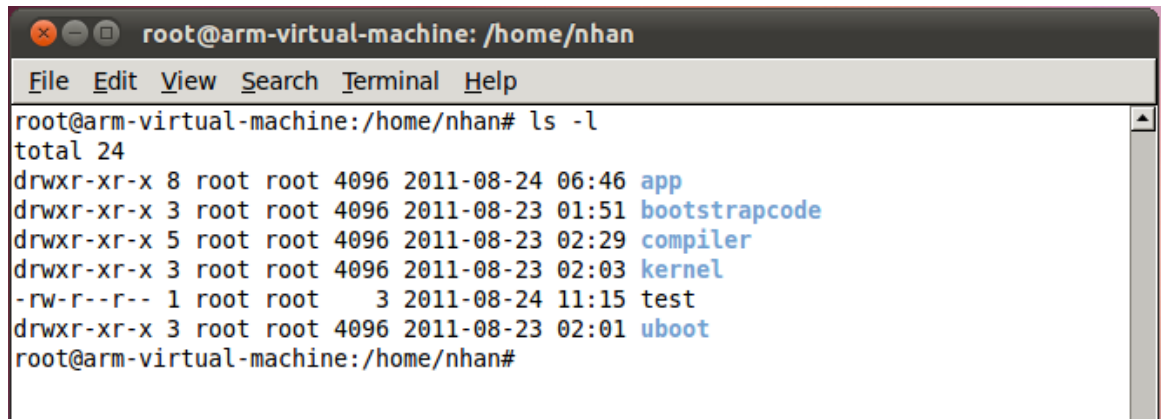
- Còn đầu tiên là *Hình 2.11: thuộc tính phân quyền của tập tin thư mục* nếu là - có nghĩa là tập tin trong thư mục. Trường hợp thư mục thì chúng ta sẽ thấy cờ này ký hiệu là d.
- Còn 3 cờ tiếp theo là các thuộc tính tương ứng với 3 nhóm. Nếu tại vị trí của thuộc tính đó mà có dấu - thì có nghĩa là thuộc tính đó không được kích hoạt cho tập tin hay thư mục này. Còn nếu là chữ thì thuộc tính đó được kích hoạt.
- Chúng ta có thể thay đổi việc kích hoạt hay không kích hoạt các thuộc tính này bằng lệnh chmod. Cú pháp của lệnh chmod như sau:

/ \$ chmod <các thuộc tính> <tên file hoặc thư mục>

**Ví dụ:** ta có thư mục /home/nhan có các file và thư mục được lệnh

/ \$ ls -l liệt kê như sau:

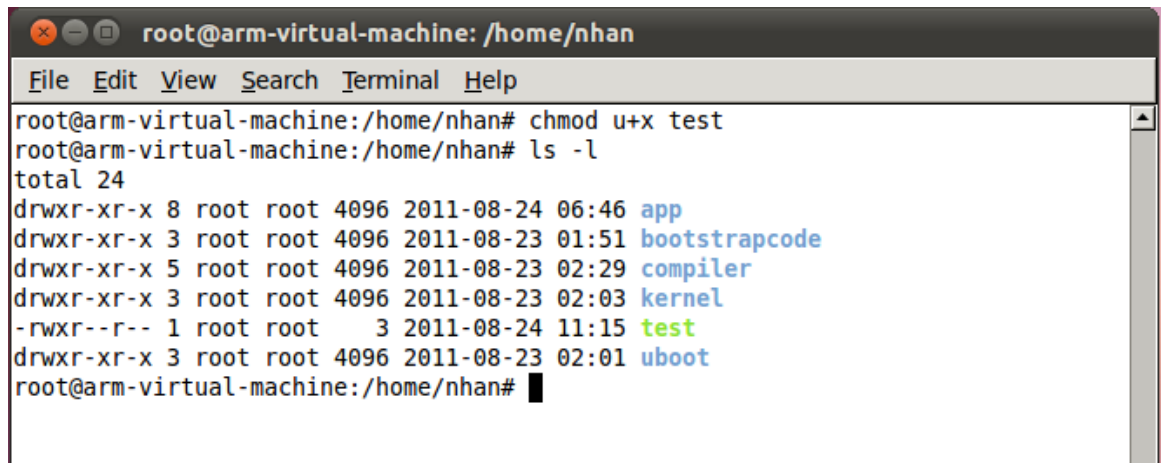




```
root@arm-virtual-machine: /home/nhan
File Edit View Search Terminal Help
root@arm-virtual-machine:/home/nhan# ls -l
total 24
drwxr-xr-x 8 root root 4096 2011-08-24 06:46 app
drwxr-xr-x 3 root root 4096 2011-08-23 01:51 bootstrapcode
drwxr-xr-x 5 root root 4096 2011-08-23 02:29 compiler
drwxr-xr-x 3 root root 4096 2011-08-23 02:03 kernel
-rw-r--r-- 1 root root 3 2011-08-24 11:15 test
drwxr-xr-x 3 root root 4096 2011-08-23 02:01 uboot
root@arm-virtual-machine:/home/nhan#
```

Hình 2.12: Ví dụ lệnh chmod 1

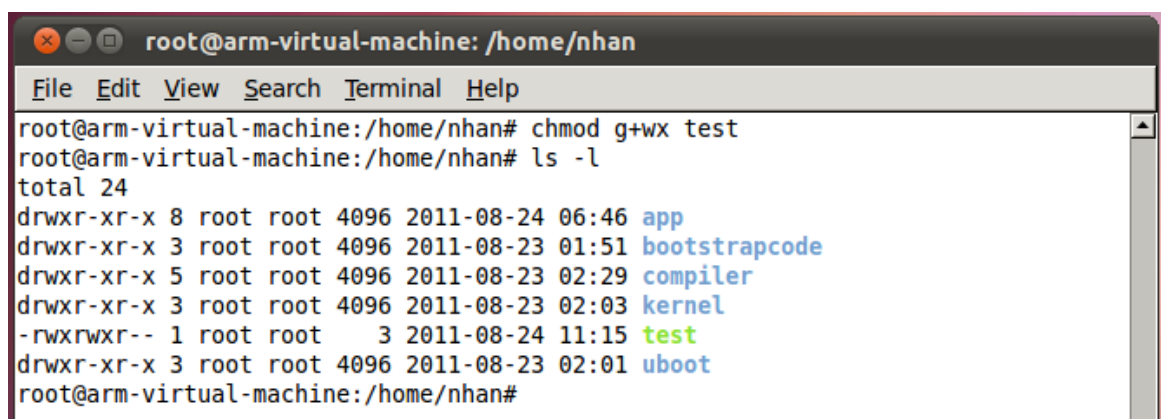
Ta thấy các thư mục sẽ có chữ d ở cở đầu tiên còn tập tin test thì là dấu -. Tập tin test không có thuộc tính x cho nhóm user, group, other, thuộc tính w cho nhóm group và other. Bây giờ chúng ta sẽ thêm thuộc tính x cho nhóm user như sau:



```
root@arm-virtual-machine: /home/nhan
File Edit View Search Terminal Help
root@arm-virtual-machine:/home/nhan# chmod u+x test
root@arm-virtual-machine:/home/nhan# ls -l
total 24
drwxr-xr-x 8 root root 4096 2011-08-24 06:46 app
drwxr-xr-x 3 root root 4096 2011-08-23 01:51 bootstrapcode
drwxr-xr-x 5 root root 4096 2011-08-23 02:29 compiler
drwxr-xr-x 3 root root 4096 2011-08-23 02:03 kernel
-rwxr--r-- 1 root root 3 2011-08-24 11:15 test
drwxr-xr-x 3 root root 4096 2011-08-23 02:01 uboot
root@arm-virtual-machine:/home/nhan#
```

Hình 2.13: Ví dụ lệnh chmod 2

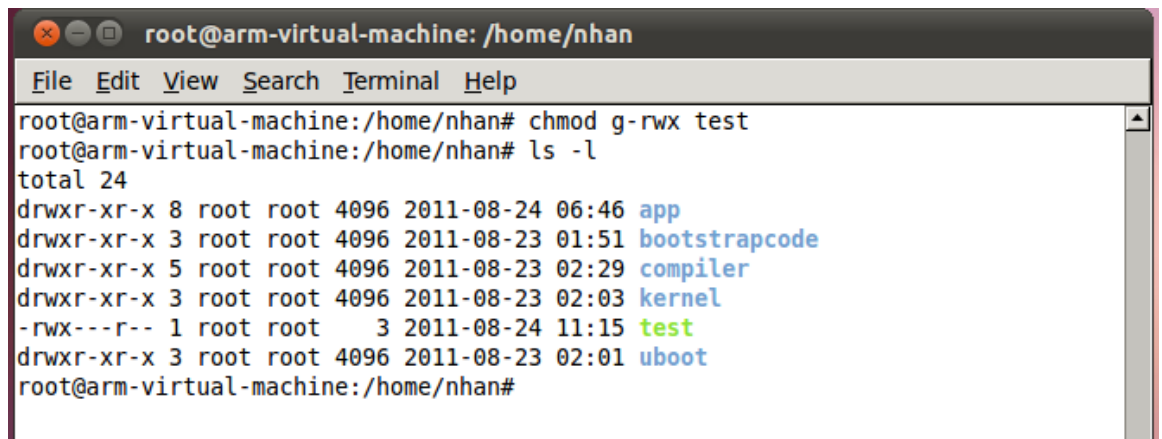
Bây giờ ta thêm thuộc tính w và x cho nhóm group:



```
root@arm-virtual-machine: /home/nhan
File Edit View Search Terminal Help
root@arm-virtual-machine:/home/nhan# chmod g+wx test
root@arm-virtual-machine:/home/nhan# ls -l
total 24
drwxr-xr-x 8 root root 4096 2011-08-24 06:46 app
drwxr-xr-x 3 root root 4096 2011-08-23 01:51 bootstrapcode
drwxr-xr-x 5 root root 4096 2011-08-23 02:29 compiler
drwxr-xr-x 3 root root 4096 2011-08-23 02:03 kernel
-rwxrwxr-- 1 root root 3 2011-08-24 11:15 test
drwxr-xr-x 3 root root 4096 2011-08-23 02:01 uboot
root@arm-virtual-machine:/home/nhan#
```

Hình 2.14: Ví dụ lệnh chmod 3

Ta có thể bỏ kích hoạt tất cả các thuộc tính của group như sau:



```
root@arm-virtual-machine: /home/nhan
File Edit View Search Terminal Help
root@arm-virtual-machine:/home/nhan# chmod g-rwx test
root@arm-virtual-machine:/home/nhan# ls -l
total 24
drwxr-xr-x 8 root root 4096 2011-08-24 06:46 app
drwxr-xr-x 3 root root 4096 2011-08-23 01:51 bootstrapcode
drwxr-xr-x 5 root root 4096 2011-08-23 02:29 compiler
drwxr-xr-x 3 root root 4096 2011-08-23 02:03 kernel
-rwx---r-- 1 root root 3 2011-08-24 11:15 test
drwxr-xr-x 3 root root 4096 2011-08-23 02:01 uboot
root@arm-virtual-machine:/home/nhan#
```

Hình 2.15: Ví dụ lệnh `chmod` 4

## 10. Nén và giải nén tập tin, thư mục:

- Trong Linux chúng ta có thể nén và giải nén các file như \*.gz, \*.bz2, \*.tar
- Để nén và giải nén một file trong Linux chúng ta dùng lệnh tar.
- Ta dùng lệnh tar để giải nén file \*.bz2 như sau :

`/ $ tar -jxvf *.bz2`

*Trong đó :*

tar là tên lệnh

j là tham số dùng khi nén hoặc giải nén file \*.bz2

x là tham số dùng khi giải nén một file

v là tham số yêu cầu hiện chi tiết các thông tin trong quá trình nén hay giải nén file

f có tác dụng chỉ định file cần giải nén hay cần nén là \*.bz2 ở cuối dòng lệnh.

\*.bz2 là file chúng ta cần nén hoặc giải nén.

- Ta dùng lệnh tar để giải nén file \*.gz như sau

`/ $ tar -zxvf *.gz`

*Trong đó:*

z là tham số dùng khi nén hoặc giải nén file \*.gz

- Ta dùng lệnh tar để giải nén file \*.tar như sau:

`/ $ tar -xvf *.tar`

- Ta dùng lệnh tar để xem nội dung một file \*.bz2, \*.gz, \*.tar như sau:

`/ $ tar -tf <tên file>`

*Trong đó:*

t là tham số dùng khi ta muốn xem nội dung file nén.

- Ta dùng lệnh tar để nén file \*.bz2 như sau:

`/ $ tar -jcvf *.bz2 <file1> <file2>`

*Trong đó :*

c là tham số chúng ta dùng khi nén một file.

- Ta dùng lệnh tar để nén file \*.gz như sau:

`/ $ tar -zcvf *.gz <file1> <file2>...`

- Ta dùng lệnh tar để nén file \*.tar như sau :

`/ $ tar -cvf *.tar <file1> <file2>...`

- Ngoài ra, để nén và giải nén file \*.gz chúng ta cũng có thể dùng lệnh gzip và lệnh gunzip :

`/ $ gzip <file cần nén>`

`/ $ gunzip <file *.gz cần giải nén>`

### **11. Biên dịch một chương trình ứng dụng:**

- Khi chúng ta lập trình một ứng dụng bằng ngôn ngữ C trên Linux thì chúng ta phải biên dịch ra file thực thi ứng dụng đó. Trong Linux chúng ta biên dịch một file ứng dụng như sau :

`/ $ gcc <tên file cần biên dịch> -o <tên file muốn xuất ra>`

*Trong đó :*

gcc là tên lệnh dùng để biên dịch một chương trình viết bằng ngôn ngữ C.

-o là tham số yêu cầu chương hệ điều hành tạo ra file thực thi với tên file mà ta muốn được viết ngay sau -o. Nếu chúng ta không dùng tham số này thì hệ điều hành sẽ tự động biên dịch ra file a.out.

### **12. Cài đặt các thông số cho chuẩn truyền nhận dữ liệu Ethernet:**

- Ethernet là chuẩn truyền nhận dữ liệu phổ biến hiện nay. Trong Linux có hỗ trợ các lệnh để người dùng có thể sử dụng chuẩn truyền nhận dữ liệu này.

- Để xem địa chỉ IP của hệ thống ta dùng lệnh :

`$ ifconfig`

- Để cài đặt địa chỉ IP tĩnh cho hệ thống ta dùng lệnh :

`$ ifconfig eth0 <địa chỉ IP tĩnh>`

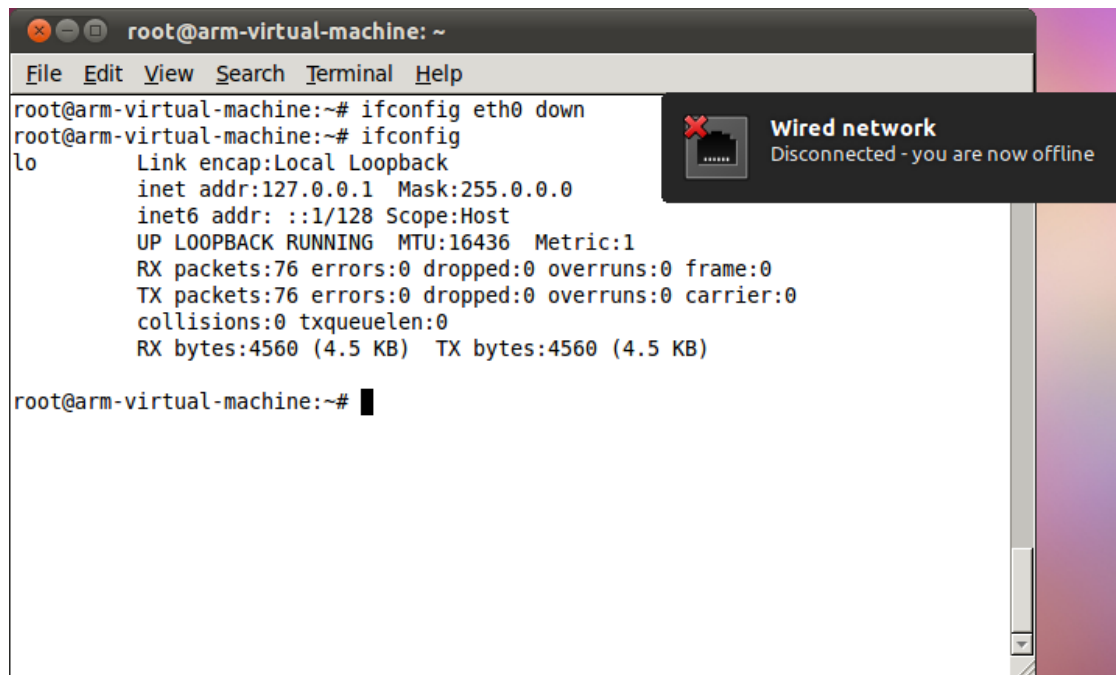
- Để ngừng kích hoạt chế độ chuyển Ethernet ta dùng lệnh :

`$ ifconfig eth0 down`

- Khi ta ngừng kích hoạt chế độ truyền Ethernet thì nếu dùng lệnh

`$ ifconfig`

*Hệ thống sẽ thông báo không có kết nối.*

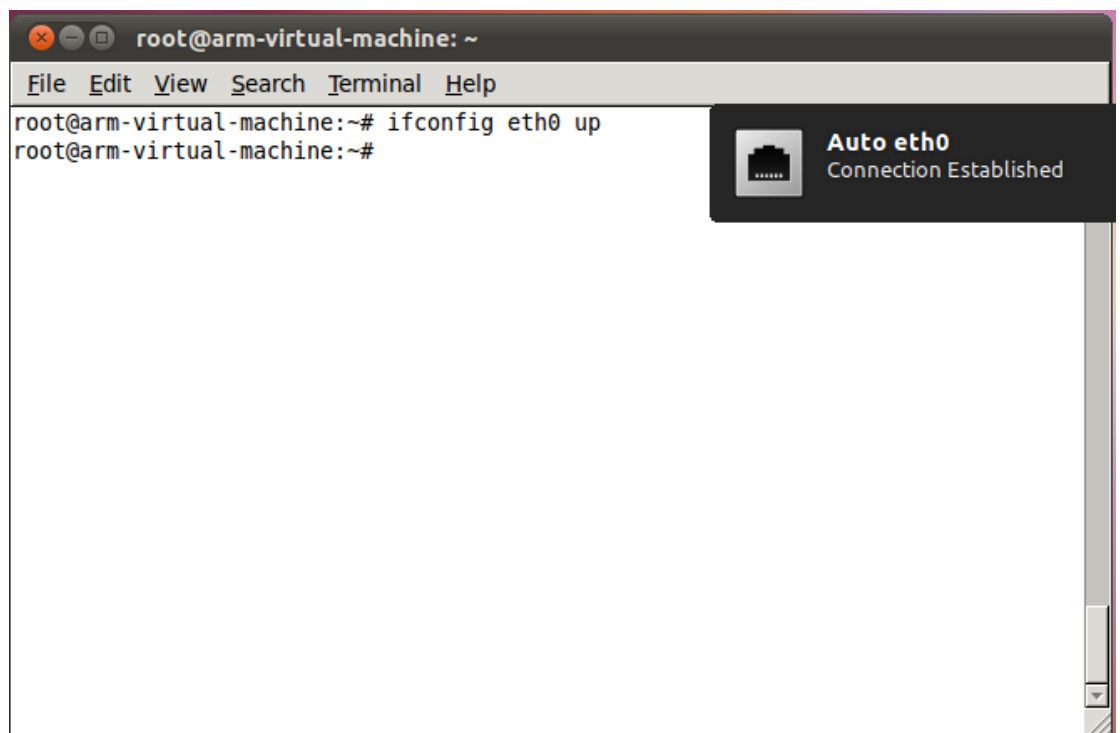


Hình 2.16 : Thông báo cổng Ethernet không có kết nối

- Để kích hoạt chế độ truyền Ethernet ta dùng lệnh :

\$ ifconfig eth0 up

Khi đó hệ thống sẽ thông báo sẵn sàng kết nối



Hình 2.17 : Thông báo cổng Ethernet đã sẵn sàng

- Để kiểm tra hệ thống có kết nối với hệ thống khác hay không ta dùng lệnh :

\$ ping <địa chỉ IP của hệ thống khác>

Nếu thấy có kết nối thì hệ thống sẽ truyền nhận 64 byte dữ liệu liên tục cho máy được kết nối

```
root@arm-virtual-machine: ~  
File Edit View Search Terminal Help  
root@arm-virtual-machine:~# ping 192.168.1.34  
PING 192.168.1.34 (192.168.1.34) 56(84) bytes of data.  
64 bytes from 192.168.1.34: icmp_req=1 ttl=128 time=0.825 ms  
64 bytes from 192.168.1.34: icmp_req=2 ttl=128 time=0.896 ms  
64 bytes from 192.168.1.34: icmp_req=3 ttl=128 time=0.546 ms  
64 bytes from 192.168.1.34: icmp_req=4 ttl=128 time=0.542 ms  
64 bytes from 192.168.1.34: icmp_req=5 ttl=128 time=0.571 ms  
64 bytes from 192.168.1.34: icmp_req=6 ttl=128 time=0.550 ms  
64 bytes from 192.168.1.34: icmp_req=7 ttl=128 time=0.558 ms
```

Hình 2.18 : Hệ thống truyền nhận 64 byte dữ liệu khi thực hiện lệnh ping  
Để ngưng quá trình truyền nhận này chúng ta dùng tổ hợp phím CtrlC.

### 13. Tạo một tài khoản người dùng :

- Để xem chúng ta đang đăng nhập với tài khoản người dùng nào, chúng ta dùng lệnh :

\$ whoami

- Chúng ta chỉ tạo được tài khoản người dùng khi chúng ta hoạt động trong tài khoản root. Để tạo ra một tài khoản mới ta dùng lệnh :

\$ useradd <tên tài khoản muốn tạo>

- Để chuyển sang tài khoản người dùng khác ta dùng lệnh :

\$ su <tên tài khoản người dùng>

**Ví dụ :** ta tạo một tài khoản tên user1 và ta chuyển sang tài khoản user1, rồi ta chuyển sang tài khoản root. Ta làm như sau :

```
root@arm-virtual-machine: ~  
File Edit View Search Terminal Help  
root@arm-virtual-machine:~# useradd user1  
root@arm-virtual-machine:~# su user1  
$ ls  
ls: cannot open directory .: Permission denied  
$ su root  
Password:  
root@arm-virtual-machine:~# ls  
Desktop Documents Downloads Music Pictures Public Templates Videos  
root@arm-virtual-machine:~#
```

Hình 2.19 : Ví dụ về tạo tài khoản người dùng

Ta thấy trong tài khoản user1 ta thực hiện lệnh ls bị lỗi vì các thư mục tập trong thư mục gốc không cho các user khác truy cập. Khi quay trở lại tài khoản root thì chúng ta phải nhập password là root00. Và trong tài khoản root thì ta thực hiện được lệnh ls.

#### 14. Các thao tác với biến môi trường của hệ thống:

- Hệ thống Linux có rất nhiều biến môi trường phục vụ cho quá trình hoạt động của hệ thống. Để xem các biến môi trường của hệ thống ta dùng lệnh sau :

`$ env`

- Để in nội dung của một biến môi trường xác định ra màn hình ta dùng lệnh :

`$ echo $<tên biến môi trường>`

- Để cài đặt biến môi trường ta dùng lệnh sau :

`$ export <tên biến môi trường> = <nội dung biến>`

- Để thêm nội dung vào biến môi trường ta dùng lệnh :

`$ export <tên biến môi trường> = $<tên biến môi trường> <nội dung muốn thêm vào>`

*Sau khi thêm nội dung vào biến môi trường chúng ta có thể dùng lệnh echo để kiểm tra.*

- Gỡ bỏ biến môi trường :

`$ unset <tên biến môi trường>`

## II. Kết luận:

Hệ điều hành Linux hỗ trợ các thao tác cơ bản và cần thiết trong quá trình người dùng tương tác với hệ điều hành. Trong phần này đã trình bày hầu hết các lệnh căn bản nhất cần thiết cho người học sử dụng Linux.

Như vậy, sau khi đọc xong phần này thì người học đã căn bản sử dụng được hệ điều hành Linux. Nhưng trên đây chỉ là những lệnh căn bản nhất của hệ điều hành Linux, Linux còn có rất nhiều lệnh khác mà người soạn giáo trình không thể liệt kê hết được. Tuy nhiên các lệnh mà được trình bày trong giáo trình này cũng đủ để người học thao tác với Linux trong quá trình học sau này.

## B-TRÌNH SOẠN THẢO VI

Nếu chúng ta đã dùng hệ điều hành Window thì chắc hẳn ai cũng quen với trình soạn thảo Notepad. Trong Linux cũng có trình soạn thảo được tích hợp sẵn trong các gói hệ điều hành Linux là trình soạn thảo VI.

Cũng giống như Notepad, trình soạn thảo VI dùng để người dùng soạn dữ liệu, viết chương trình và lưu lại dưới dạng tập tin.

Trong phần này sẽ hướng dẫn cách sử dụng trình soạn thảo VI.

### I. Giới thiệu:

VI là chương trình soạn thảo chuẩn trên các hệ điều hành Unix. Nó là chương trình soạn thảo trực quan, hoạt động dưới 2 chế độ: Chế độ lệnh (command line) và chế độ soạn thảo (input mode)

Để soạn thảo tập tin mới hoặc xem, sửa chữa tập tin cũ ta dùng lệnh:

```
/$ vi <tên tập tin>
```

Khi thực hiện, VI sẽ hiện lên màn hình soạn thảo ở chế độ lệnh. Ở chế độ lệnh, chỉ có thể sử dụng các phím để thực hiện các thao tác như: dịch chuyển con trỏ, lưu dữ liệu, mở tập tin...Do đó, bạn không thể soạn thảo văn bản.

Nếu muốn soạn thảo văn bản, bạn phải chuyển từ chế độ lệnh sang chế độ soạn thảo. Chế độ soạn thảo giúp bạn sử dụng bàn phím để soạn thảo nội dung văn bản.

### II. Các lệnh thao tác trong VI:

#### 1. Chuyển chế độ trong VI:

- Để chuyển sang chế độ soạn thảo chúng ta có thể dùng một trong hai phím sau trên bàn phím:

    i con trỏ sẽ giữ nguyên vị trí.

    a con trỏ sẽ dịch sang phải một ký tự.

- Để chuyển ngược lại mode command ta dùng phím ESC
- Để thực hiện các lệnh trong trình soạn thảo VI thì trình soạn thảo VI phải ở chế độ lệnh

#### 2. Nhóm lệnh di chuyển con trỏ :

    h sang trái 1 ký tự

    e đến ký tự cuối cùng của từ gần nhất bên phải

    w đến ký tự đầu tiên của từ gần nhất bên phải

    b đến ký tự đầu tiên của từ gần nhất bên trái

    k lên 1 dòng

j xuống 1 dòng  
) cuối câu  
( đầu câu  
} đầu đoạn văn  
{ cuối đoạn văn

### 3. Nhóm lệnh xóa:

dw xóa 1 từ  
d^ xóa ký tự từ con trỏ đến đầu dòng  
d\$ xóa ký tự từ con trỏ đến cuối dòng  
3dw xóa 3 từ  
dd xóa dòng hiện hành  
5dd xóa 5 dòng  
x xóa 1 ký tự

Các từ được cách nhau bởi khoảng trắng.

### 4. Nhóm lệnh thay thế:

cw thay thế 1 từ  
3cw thay thế 3 từ  
cc dòng hiện hành  
5cc 5 dòng

### 5. Nhóm lệnh copy, past, undo:

Để copy ta dùng lệnh y và để paste ta dùng lệnh p

y\$ copy từ vị trí hiện tại của cursor đến cuối cùng  
yy copy toàn bộ dòng tại vị trí cursor  
3yy copy 3 dòng liên tiếp  
u Undo lại thao tác trước đó

### 6. Thao tác trên tập tin :

:x lưu và thoát khỏi chế độ soạn thảo  
:wq lưu và thoát khỏi chế độ soạn thảo  
:w lưu vào tập tin mới  
:q thoát nếu ko có thay đổi  
:q! thoát không lưu

- Sau khi gõ các lệnh trên thì nhấn Enter để thực thi lệnh.



### **III. Kết luận:**

VI là trình soạn thảo của Linux, VI có hai chế độ làm việc là chế độ lệnh và chế độ soạn thảo. Các thao tác trên VI rất khác so với các tác của các trình soạn thảo quen thuộc nên làm cho người dùng cảm thấy khó khăn trong việc soạn thảo dữ liệu với VI.

Vì tính phức tạp của trình soạn thảo VI, nên trong thực tế trình soạn thảo VI chỉ dùng vào mục đích soạn thảo tập tin có nội dung ngắn hoặc là để sửa nội dung tập tin trong trường hợp cần sửa một cách nhanh chóng trong hệ điều hành Linux. Còn đối với việc soạn các tập tin có nội dung phức tạp, dài thì các lập trình viên vẫn thường dùng các trình soạn thảo quen thuộc của Window như Notepad, Wordpad, Microsoft Word,....

Như vậy, trong bài này chúng ta đã làm quen với hệ điều hành Linux, một trong những hệ điều hành được sử dụng phổ biến hiện nay. Hiểu biết về Linux sẽ tạo nền tảng cho chúng ta tìm hiểu sâu về hệ thống nhúng.

Trong bài sau chúng ta sẽ tìm hiểu về hệ thống nhúng là kit nhúng KM9260.

**HỆ THỐNG PHẦN CỨNG VÀ PHẦN MỀM  
TRONG KIT NHÚNG KM926**

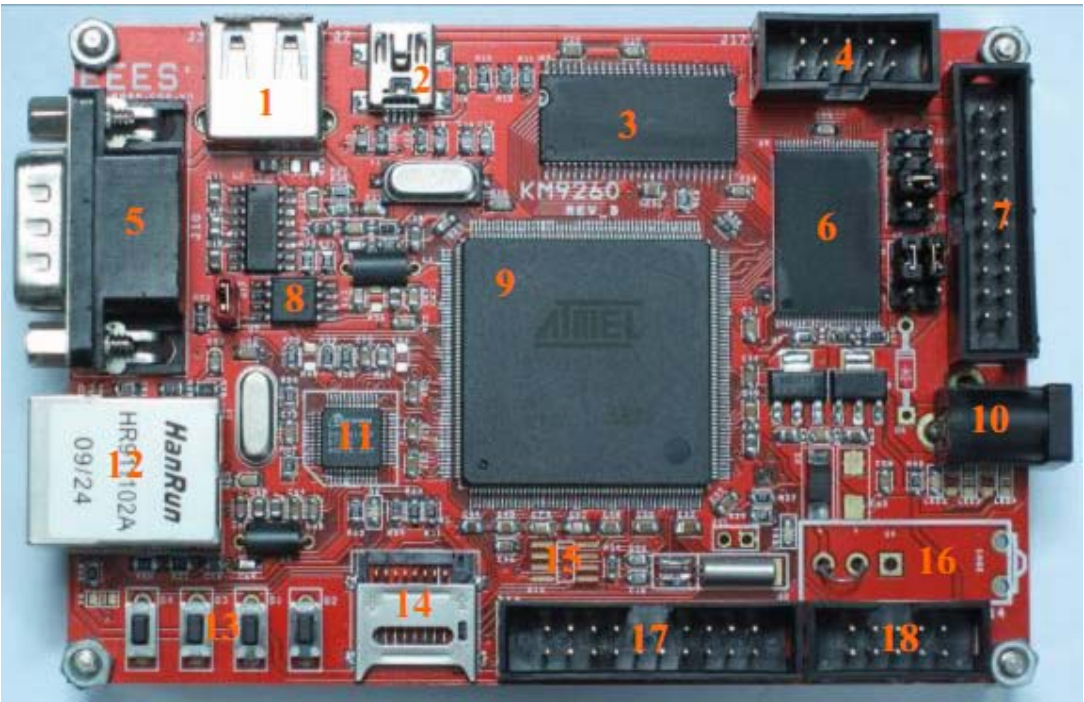
**A-Phần cứng hệ thống nhúng trong kit KM9260:**

Một hệ thống nhúng bao gồm hai phần: phần cứng và phần mềm. Trong đó phần cứng là nền tảng, là điều kiện cần để một hệ thống nhúng có thể tồn tại. Để có thể sử dụng được một hệ thống nhúng thì chúng ta phải hiểu được phần cứng của nó

Trong hệ thống nhúng ngoài vi xử lý là nòng cốt thì bên cạnh đó còn có nhiều linh kiện khác: chip nhớ, các thiết bị ngoại vi, ... Trong phần này chúng ta sẽ tìm hiểu phần cứng của Kit KM9260 gồm những linh kiện nào, chức năng của các linh kiện, vị trí của từng linh kiện trên board.

Vì phần cứng hệ thống nhúng đa số là do các công ty sản xuất thiết kế, thi công nên với tư cách là một người nghiên cứu để sử dụng thì trong giáo trình này chúng ta không đi sâu vào mạch nguyên lý của kit nhúng.

**Hình dạng của board Kit KM9260**



*Hình 2.20 : Hình dạng kit KM9260*

Các thành phần trên board bao gồm:

STT	KÝ HIỆU	TÊN
1	J3	Cổng USB (loại A)
2	J7	Cổng USB Devices (loại B)

3	U2	MT48LC16M16A2, SDRAM 256Mb (32MB) 133Mhz
4	J17	Kết nối mở rộng SCI
5	J10	Cổng truyền dữ liệu nối tiếp BD9
6	U8	K9F2G08UOM, NAND Flash (256MB)
7	J5	Giao tiếp JTAG ICE
8	U9	Serial dataflash (512kB)
9	U1	AT91SAM9260, 16/32 bit ARM926EJ-S 180Mhz
10	J12	Jack nguồn 5VDC
11	U5	DM9161EA, Ethernet 10/100 Full-Duplex
12	RJ1	Cổng Giao tiếp Ethernet (RJ45)
13	S1, S2, S3, S4	Các nút nhấn
14	U10	Khe thẻ nhớ MicroSD
15	U11	I2C EEPROM
16	J12	Công tắc nguồn
17	J16	Kết nối mở rộng Uart, Adc, Twi
18	J14	Kết nối mở rộng SPI

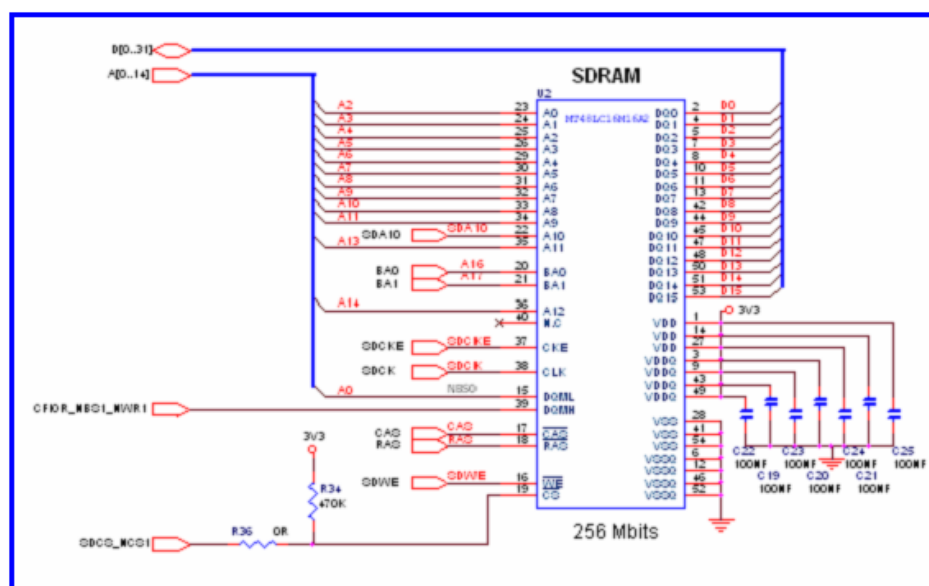
Bảng 2.5: Các linh kiện trên kit KM9260

## I. Cpu (C1):

Sử dụng vi điều khiển AT91SAM9260 đã được giới thiệu trong phần trước.

## II. Bộ nhớ:

### 1. Sdram (U2):



Hình 2.21 : Sơ đồ kết nối SDRAM

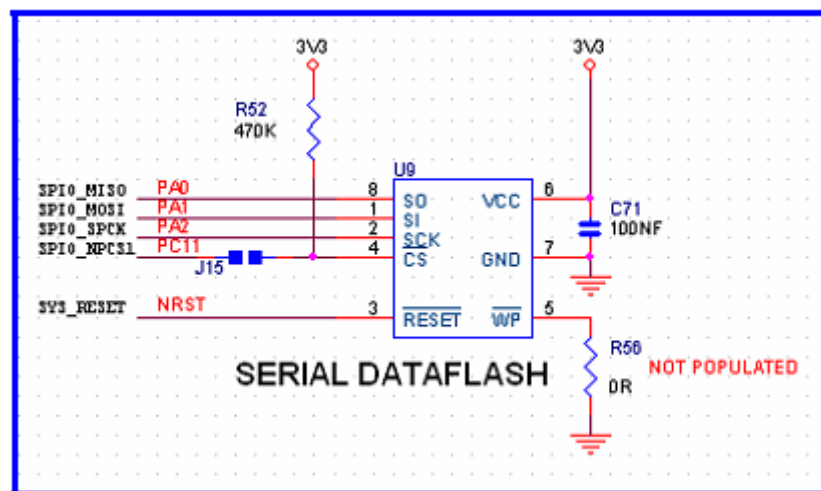
- Bộ nhớ chính sử dụng SDRAM bus 133Mhz, SDRAMC được cấu hình với bus data 16 bit.
- Bảng sau trình bày thông số bảng đồ vùng nhớ của SDRAM trong hệ thống.

<b>Mã số linh kiện</b>	MT48LC16M16A2 TC75
<b>Chân chọn chip</b>	NCS1
<b>Địa chỉ logic</b>	0x20000000
<b>Dung lượng</b>	0x2000000 (32MB)

*Bảng 2.6: Thông số của SDRAM*

- Mã số linh kiện là số được in trên thân của chip, mã số này do nhà sản xuất quy định.
- Chân chọn chip là NCS1 có nghĩa là chân CS của chip được nối với chân SDCS\_NCS1 của vi xử lý.
- Địa chỉ logic là địa chỉ mà vi xử lý gán cho chip tương ứng với NCS1.
- Dung lượng của SDRAM là 32 MB

## 2. Serial dataflash (U9):



*Hình 2.22 : Sơ đồ kết nối Serial DataFlash*

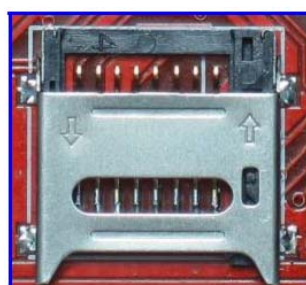
- Board sử dụng chip nhớ serial dataflash kết nối qua đường SPI0 (slot CS1).
- AT91Bootstrap, các biến môi trường (U-Boot's Environment Variables), U-Boot được lưu trữ trong serial dataflash. Các phân vùng chứa các bootloader được thể hiện bởi bảng sau:



<b>Mã số linh kiện</b>	K9F2G08UOM	
<b>Chân chọn chip</b>	NCS3	
<b>Địa chỉ logic</b>	0x40000000	
<b>Dung lượng</b>	0x10000000 (256MB)	
<b>Offset</b>	<b>Phân Vùng</b>	<b>Phần mềm lưu</b>
0x00000000	0	Kernel
0x00300000	1	Rootfs (JFFS2)

*Bảng 2.8: Thông số của NAND FLASH*

#### 4. Thẻ nhớ MicroSD (U10):



*Hình 2.24 : Khe cắm thẻ nhớ Micro SD*

- Do có hỗ trợ microSD, có thể thay thế vai trò của NAND Flash cho việc lưu trữ kernel Linux và rootfs. Phân vùng đầu tiên (first partition) được format theo định dạng FAT, phân vùng thứ 2 được định dạng theo ext2 hoặc ext3 dùng để chứa rootfs. Để load kernel Linux từ MicroSD vào SDRAM đòi hỏi U-Boot phải hỗ trợ mmc sub system command set. MicroSD thường được dùng để boot Linux có rootfs dung lượng lớn, ví dụ như Debian distribution.

### III. Kết nối ngoại vi:

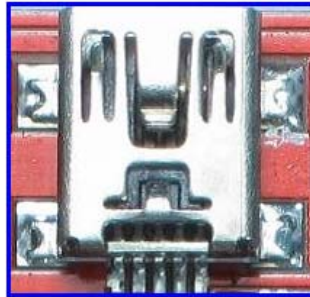
#### 1. Cổng USB (J13):



*Hình 2.25 : Cổng USB*

Cổng USB tốc độ truy xuất 2.0, tương tự MicroSD, hệ thống có thể boot Linux thông qua ổ đĩa di động USB..

## **2. Cổng thiết bị USB (J7):**



*Hình 2.26 : Cổng USB Device*

Với USB device connector, ta có thể biến board nhúng thành các thiết bị USB. MCU AT91SAM9260 cho phép ta truy xuất đến tất cả các vùng nhớ trong hệ thống thông qua chương trình ứng dụng trên máy tính SAMBA, khi đó dây cable USB này được dùng đến.

## **3. Cổng truyền dữ liệu nối tiếp DB9 (J10):**



*Hình 2.27 : Cổng truyền dữ liệu nối tiếp DB9*

AT91SAM9260 có tích hợp cổng RS232. KM9260 dùng cổng RS232 này cho việc hiển thị, xuất nhập với console chính của Linux.

## **4. Cổng kết nối Ethernet (RJ1):**

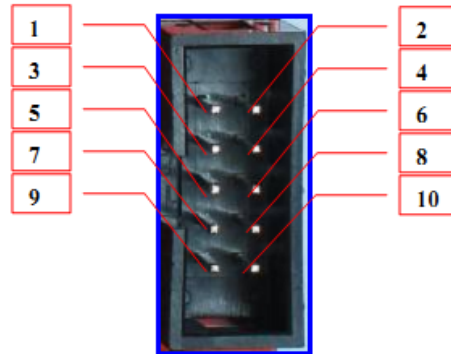


*Hình 2.28 : Cổng kết nối Ethernet*



AT91SAM9260 có thể kết hợp với chip Fast Ethernet PHY DM9161AEP mang lại cho hệ thống tính năng mạnh mẽ về các ứng dụng mạng. KM9260 có thể sử dụng như hệ thống webserver nhúng, sử dụng trong hệ thống thu thập đo lường, điều khiển từ xa...

## 5. Kết nối mở rộng SCI (J17):



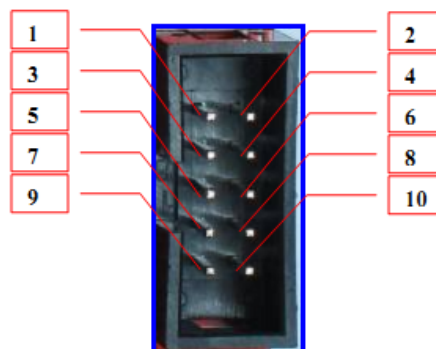
Hình 2.29 : Khe cắm kết nối mở rộng SCI

SỐ CHÂN	TÍN HIỆU	CHÂN VXL	SỐ CHÂN	TÍN HIỆU	CHÂN VXL
1	GND	-	2	5V	-
3	GND	-	4	3V3	-
5	TK0	23 (PB16)	6	TF0	26 (PB17)
7	TD0	27 (PB18)	8	RD0	28 (PB19)
9	RK0	163 (PB20)	10	RF0	164 (PB21)

Bảng 2.9: Sơ đồ chân của khe cắm kết nối mở rộng SCI

Connector cho phép ta có thể mở rộng kết nối với các thiết bị I2S audio codec, truyền nhận dữ liệu 32bit stream (High-speed Continuous Data Stream)...

## 6. Kết nối mở rộng SPI (J14):



Hình 2.30 : Khe cắm kết nối mở rộng SPI

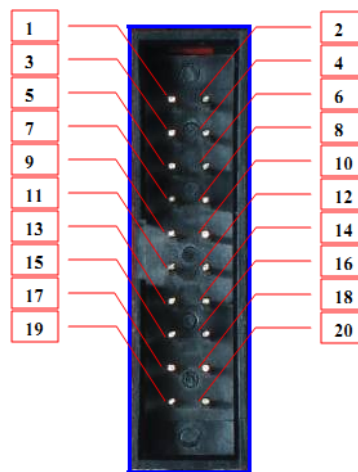


SỐ CHÂN	TÍN HIỆU	CHÂN VXL	SỐ CHÂN	TÍN HIỆU	CHÂN VXL
1	GND	-	2	3V3	-
3	SPI1_MISO	9 (PB0)	4	SPI1_MOSI	10 (PB1)
5	SPI1_SPCK	11 (PB2)	6	SPI1_NPCS0	12 (PB3)
7	SPI1_NPCS1	67 (PC5)	8	SPI1_NPCS2	62 (PC4)
9	GPIO	63 (PC6)	10	GPIO	64 (PC7)

*Bảng 2.10: Sơ đồ chân của khe cắm kết nối mở rộng SPI*

Connector mở rộng cho SPI, bao gồm 3 đường chip select tương ứng cho 3 slot CS0, CS1, CS2.

## 7. Kết nối mở rộng Uart, Adc, Twi (J16):



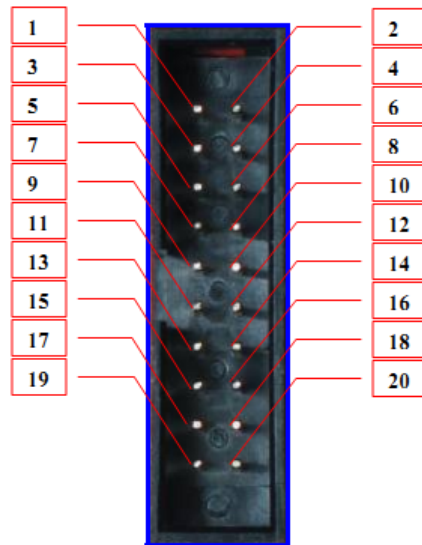
*Hình 31: Khe cắm kết nối mở rộng Uart, Adc, Twi*

SỐ CHÂN	TÍN HIỆU	CHÂN VXL	SỐ CHÂN	TÍN HIỆU	CHÂN VXL
1	5V	-	2	3V3	-
3	AVDD	-	4	GND	-
5	AGND	-	6	VREFP	-
7	AD0	158 (PC0)	8	AD1	159 (PC1)
9	IRQ1	127 (PC15)	10	UART_TXD0	15 (PB4)
11	UART_RXD0	16 (PB5)	12	UART_TXD1	17 (PB6)
13	UART_RXD1	18 (PB7)	14	UART_TXD2	19 (PB8)
15	UART_RXD2	20 (PB9)	16	UART_TXD3	161 (PB10)
17	UART_RXD3	162 (PB11)	18	TWD	208 (PA23)
19	TWCK	1 (PA24)	20	GPIO	23 (PB16)

*Bảng 2.11: Sơ đồ chân của khe cắm kết nối mở rộng Uart, Adc, Twi*

Connector mở rộng cho các cổng giao tiếp serial bao gồm UART, TWI, bộ chuyển đổi ADC, GPIO, ngắt ngoài IRQ1.

#### 8. Giao tiếp JTAG ICE (J5) :



Hình 2.32: Khe cắm giao tiếp JTAG ICE

SỐ CHÂN	TÍN HIỆU	CHÂN VXL	SỐ CHÂN	TÍN HIỆU	CHÂN VXL
1	3V3	-	2	3V3	-
3	NTRST	35	4	GND	-
5	TDI	30	6	GND	-
7	TMS	31	8	GND	-
9	TCK	34	10	GND	-
11	RTCK	37	12	GND	-
13	TDO	29	14	GND	-
15	NRST	36	16	GND	-
17	NC	-	18	GND	-
19	NC	-	20	GND	-

Bảng 2.12: Sơ đồ chân của khe cắm giao tiếp JTAG ICE

Cổng JTAG ICE theo chuẩn 20 pin cho phép nạp chương trình và debug hệ thống.

#### **IV. Nút nhấn:**

##### **1. Nút Reset (S2):**



*Hình 2.33: Nút reset*

Reset hệ thống, tích cực mức thấp.

##### **2. Nút Wake up (S1):**



*Hình 2.34: Nút Wake up*

Nút có tác dụng đánh thức hệ thống từ trạng thái power down.

##### **3. Nút ứng dụng 1 (S3):**



*Hình 2.35: Nút ứng dụng 1*

User button, kết nối với PC15 (chân 127) của AT91SAM9260 MCU. Tích cực mức thấp.

##### **4. Nút ứng dụng 2 (S4):**



Hình 2.36: Nút ứng dụng 2

User button, kết nối với PC8 (pin 61) của AT91SAM9260 MCU. Tích cực mức thấp.

## V. Led hiển thị:

### 1. Led hiển thị trạng thái của hệ thống:

TÊN LED	CHỨC NĂNG	TRẠNG THÁI	Ý NGHĨA
LED1	Shutdown	On	VXL Trạng thái Power down
		Off	VXL không hoạt động
LED2	Power	On	Bật nguồn
		Off	Tắt nguồn
D6	Duplex	On	Ethernet PHY full-duplex
		Off	Ethernet PHY half-duplex

Bảng 2.13: Ý nghĩa của các Led hiển thị trạng thái của hệ thống

### 2. Led ứng dụng:

Ký hiệu là D4, được nối với chân PA6 (chân 185). Tích cực mức thấp.

## VI. Jumper :

### 1. Jumper chọn chip :

KÝ HIỆU	CHỨC NĂNG
J15	Chọn chip Serial dataflash
J13	Chọn chip NAND Flash

Bảng 2.14: Chức năng của J13 và J15



Hình 2.37: Serial DataFlash và Jumper

chọn Serial DataFlash

Jumper chọn chip được thiết kế hỗ trợ cho việc khôi phục boot loader sau khi người dùng ghi chương trình không phù hợp vào vùng Bootstrap hoặc U-Boot của hệ thống

### 2. Jumper chọn cấu hình hệ thống:

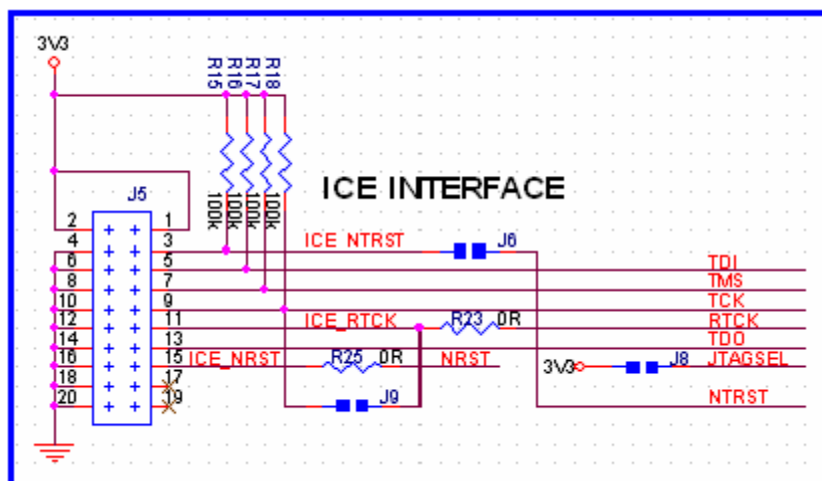
KÝ HIỆU	CHỨC NĂNG	VỊ TRÍ	Ý NGHĨA	MẶC ĐỊNH
J1	Chọn tần số hoạt động	1-2	Dùng dao động nội RC	2-3
		2-3	Dùng thạch anh 32,768Khz	

J2	Chọn nguồn cho vi xử lý	1-2	Dùng nguồn pin	1-2
		2-3	Dùng nguồn chính 1,8 V	

Bảng 2.15 : Chức năng của J1 và J2

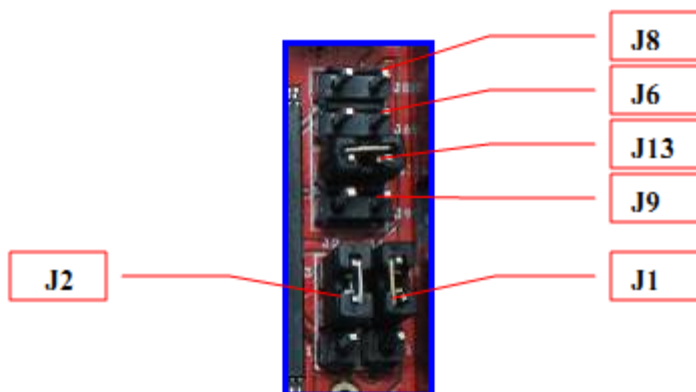
### 3. Các Jumper liên quan đến jtag

- Sơ đồ mạch FTAG INTERFACE bao gồm các jumper liên quan đến JTAG như sau :



Hình 2.38: Sơ đồ kết nối của giao tiếp JTAG ICE

- Vị trí của các Jumper J1, J2, J6, J8, J9, J13 được thể hiện ở hình sau :



Hình 2.39: Các jumper J1, J2, J6, J8, J9, J13

## VII. Tổng kết:

Phần cứng hệ thống nhúng gồm nhiều thành phần : vi xử lý, chip nhớ, ngoại vi, nút nhấn, led báo, jump được kết nối với nhau tạo thành một thể thống nhất. Mỗi thành phần đều có một chức năng riêng.

Để thuận lợi cho việc tìm hiểu sâu hơn về kit KM9260 thì người học sau khi đọc phần này nên cố gắng nhận diện được các linh kiện trên kit, xác định được vị trí, chức năng của từng linh kiện.

Như đã nói ở phần trên, chúng ta sẽ không tìm hiểu sâu về phần cứng của kit KM9260. Thay vào đó chúng ta sẽ tìm hiểu kỹ hơn về các phần mềm hệ thống nhúng trong phần tiếp theo của giáo trình.

## **B-Phần mềm hệ thống nhúng trong kit KM9260:**

CHƯƠNG III

LẬP TRÌNH NHÚNG

NÂNG CAO



## Lời đầu chương

Sau khi nghiên cứu chương II-Lập trình nhúng căn bản chúng ta đã có được những kiến thức cần thiết để tiến sâu hơn vào thế giới lập trình phần mềm hệ thống nhúng Linux đầy thú vị. Trước khi đi vào nghiên cứu những nội dung trong chương III, người học phải có những kiến thức nền tảng sau đây:

- Sử dụng được các lệnh trong ngôn ngữ lập trình C như: if, if...else, if...else if..., switch...case, ...; hiểu và vận dụng được các khái niệm trong ngôn ngữ lập trình C như: Con trỏ, cấu trúc, mảng, ... vào chương trình;
- Hiểu một cách tổng quát các kiến thức cơ sở lý thuyết về hệ điều hành nói chung. Phải giải thích được các khái niệm về tiến trình, tuyến, chia khe thời gian, ...trong hệ điều hành.
- Và một số những kỹ năng cần thiết như biên dịch chương trình ứng dụng, driver viết bằng ngôn ngữ lập trình C trong Linux; Sử dụng thành thạo các lệnh trong môi trường shell của Linux;

Trong chương III-Lập trình nhúng nâng cao chúng ta sẽ được tìm hiểu lý thuyết về hai lớp trong hệ thống phần mềm nhúng là user application và kernel driver cũng như một số những thao tác lập trình để xây dựng chương trình trong hai lớp này.

Phần lập trình nhúng nâng cao được biên soạn để viết hoàn chỉnh các driver và chương trình ứng dụng cho một số những thiết bị ngoại vi phổ biến thường được sử dụng trong việc dạy học như: LED đơn, LED 7 đoạn, LED ma trận, LCD, ADC, ...Do đó những kiến thức được trình bày trong chương này chỉ là những kiến thức cần thiết để phục vụ cho việc lập trình dự án điều khiển các thiết bị ngoại vi trên. Những kiến thức chuyên sâu khác người học sẽ tự tìm hiểu trong những nguồn thông tin sách được chúng tôi giới thiệu trong phần tài liệu tham khảo.

Các bài luyện tập trong quyển sách này bao gồm:

*Phần A: Lập trình user application;*

Phần này trình bày ví dụ về cách viết chương trình ứng dụng trong user, nhắc lại cách biên dịch và thực thi chương trình trong hệ điều hành. Bên cạnh đó các bạn sẽ tiếp xúc với hệ thống thời gian thực trong Linux thông qua các hàm trì hoãn thời gian, các hàm thao tác với thời gian thực, các hàm tạo tín hiệu định thời theo chu kỳ. Chúng ta cũng sẽ

nguyên cứu về lập trình đa tiến trình, đa tuyến về nguyên lý hoạt động và cách sử dụng vào các chương trình ứng dụng thời gian thực.

*Phần B: Căn bản lập trình driver;*

Sau khi lập trình chương trình ứng dụng trong lớp User thành thạo, chúng ta sẽ tiếp tục làm việc với lớp sâu hơn là Driver. Qua đó, người học sẽ hiểu vai trò của Driver, cách phân phối nhiệm vụ giữa Driver và Application sao cho đạt hiệu quả cao nhất. Các lệnh lập trình driver, các thao tác cài đặt và sử dụng driver cũng sẽ được chúng tôi trình bày trong chương này.

Trên đây là nội dung tổng quát của phần II-Lập trình hệ thống nhúng nâng cao. Mỗi bài luyện tập đều bao gồm trình bày lý thuyết và chương trình ví dụ, các chương trình đều được giải thích rõ ràng từng dòng lệnh đồng thời cũng có đưa ra kết quả thực thi chương trình trong hệ thống. Người học không những chỉ đọc mà còn phải kiểm tra kết quả thực thi bằng cách làm lại các chương trình ví dụ được nêu trong từng bài học để có thể hiểu được vai trò của bài học trong giáo trình.

**PHẦN A**

**LẬP TRÌNH USER APPLICATION**

**BÀI 1****CHƯƠNG TRÌNH HELLOWORLD****I. Mở đầu:**

“*Hello world!*” là một chương trình quan trọng và căn bản đầu tiên khi bước vào thế giới của bất kỳ một ngôn ngữ lập trình nào. Mặc dù đơn giản nhưng bài này sẽ cho chúng ta làm quen với các bước lập trình, biên dịch và thực thi một chương trình ứng dụng trong hệ thống Linux. Học bài này chúng ta có dịp ôn lại những thao tác biên dịch chương trình đã được học trong phần lập trình hệ thống nhúng căn bản.

**II. Nội dung:****1. Hàm printf:****a. Mô tả lệnh:**

```
int printf ( const char* format, ..., <parameter_n>, ...);
```

**b. Giải thích chức năng:**

Hàm `printf()` dùng in một thông tin cần hiển thị ra màn hình đầu cuối. Thông thường những thông tin này là thông báo của người lập trình về quá trình xử lý của chương trình đang đi đến đâu, có xảy ra lỗi hay không.

Hàm trả về một số kiểu `integer`. Nếu quá trình hiển thị thành công, hàm trả về một số dương. Ngược lại sẽ trả về số âm.

Tham số `const char* format` là định dạng chuỗi thông tin cần hiển thị.

Tùy theo chuỗi thông tin cần hiển thị, thì `<parameter_n>` là một giá trị thuộc dạng nào đó.

**c. Ví dụ:**

```
#include <stdio.h>

int main() {
    if ( printf("Characters %d: %c, %c, %c, %c\n", 1, 'A', 65, 'B',
        66) == -1) {
        printf("Display error!\n");
    } else {
        printf("Display no error!\n");
    }
}
```

Kết quả xuất ra màn hình như sau:

```
Characters 1: A, A, B, B
```

```
Display no error!
```

**\*\*Chúng ta thấy trong câu lệnh `printf()` đầu tiên, ta xuất chuỗi "Characters ..." ra màn hình. Chuỗi này có định dạng hiển thị ban đầu là một số nguyên (`%d`), tiếp theo là 4 ký tự (`%c`). Mặc dù các tham số không phải lúc nào cũng là nhưng ký tự thuần túy, nhưng hàm `printf` cũng vẫn hiểu là những ký tự-vì hàm này đã được định dạng phải hiểu là một ký tự. Ta ghi 65, hàm sẽ tự động chuyển số này thành ký tự 'A' trong mã ascii. Tương tự cho các tham số khác. Hàm thực thi không có lỗi, vì vậy in ra câu thông báo "Display no error!".**

## 2. Hàm `exit`:

### d. Mô tả lệnh:

```
exit (int n);
```

**\*\*Hàm chứa trong thư viện `stdlib.h`**

### a. Giải thích chức năng:

Hàm `exit(n)` dùng để thoát khỏi một chương trình đang thực thi. Thông thường hàm này dùng cho trường hợp khẩn cấp, người lập trình muốn thoát khỏi chương trình một cách đột ngột khi phát sinh ra một lỗi nào đó đã được dự đoán trước.

Trong trường hợp xảy ra lỗi, người lập trình muốn biết đó là lỗi gì thì sẽ thêm vào thông tin cho `n`. `n` là một số nguyên integer do người lập trình định nghĩa, cho biết thông tin kèm theo khi thực hiện lệnh `exit(n)`;

Lệnh `exit(n)` thường đặt cuối chương trình, giá trị trả về `n` được gán bằng 0, cho biết là chương trình thực thi không bị lỗi.

### b. Ví dụ:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    if ( printf("Characters %d: %c, %c, %c, %c\n", 1, 'A', 65, 'B', 66)
        == -1) {
        printf("Display error!\n");
        exit(1); //Báo lỗi thoát là 1
    }
}
```

```
} else {  
    printf("Display no error!\n");  
}  
  
/*Chương trình thực thi không bị lỗi, trả về mã lỗi là 0*/  
exit (0);  
}
```

### **3. Chương trình helloworld:**

#### **a. Mã chương trình:**

```
/*Khai báo thư viện chuẩn vào ra-do sử dụng lệnh printf*/  
#include <stdio.h>  
#include <stdlib.h>  
  
/*Chương trình chính*/  
int main() {  
    /*In ra chuỗi "Hello world!"*/  
    printf ("Hello world!");  
    /*Thoát khỏi chương trình, trả về mã lỗi là 0*/  
    exit(0);  
}
```

#### **b. Biên dịch và chạy chương trình:**

- Chép chương trình đã hoàn thành vào máy Linux ảo;
- Biên dịch chương trình dùng lệnh sau:

```
arm-none-linux-gnueabi-gcc HelloWorld.c -o HelloWorld
```

*\*\*Lệnh này sử dụng chương trình biên dịch chéo trong chương trình hỗ trợ biên dịch cho ARM. Ý nghĩa của lệnh này là biên dịch một tập tin chương trình có tên là HelloWorld.c, kết quả cho ra một tập tin mã máy có thể chạy được trên ARM có tên là HelloWorld.*

*\*\*Chú ý trước khi sử dụng lệnh này cần phải khai báo biến môi trường PATH trong Linux, sao cho trỏ đến nơi chứa các tập tin thực thi gcc cho ARM.*

- Chép tập tin đã biên dịch HelloWorld vào kit;
- Tiến hành chạy chương trình bằng dòng lệnh sau:

```
./HelloWorld
```

Khi nhấn enter để chạy chương trình, hệ thống Linux sẽ xuất hiện thông báo lỗi như sau:

```
-sh: ./HelloWorld: Permission denied.
```

Nguyên nhân xảy ra lỗi là do tập tin chương trình HelloWorld vừa chép qua không có quyền thực thi trong hệ thống Linux. Để sửa lỗi này chúng ta cần phải cho phép tập tin chương trình HelloWorld thực thi được bằng dòng lệnh sau:

```
chmod 777 HelloWorld
```

Sau đó tiến hành chạy chương trình, chương trình chạy thành công. Lúc này màn hình hiển thị chuỗi ký tự "Hello World!" đã được lập trình trước đó.

```
./HelloWorld  
Hello world!
```

### ***c. Giải thích chương trình:***

Đây là một chương trình đơn giản, nhiệm vụ của chương trình là in ra chuỗi "Hello world!" ra màn hình hiển thị (console).

Phần đầu của chương trình là khai báo hai thư viện chuẩn `<stdio.h>` và `<stdlib.h>` dùng cho các hàm `printf` và `exit`.

Tiếp theo là khai báo và định nghĩa phần chương trình chính `main()`. Hàm `main()` có kiểu dữ liệu trả về là `int` và hàm này không có tham số.

Trong hàm `int main()`, ta sử dụng hai hàm `printf` và `exit`. Hàm `printf` (Định nghĩa trong thư viện `stdio.h`) dùng xuất một chuỗi ký tự được định dạng ra màn hình. Hàm `exit(n)` (định nghĩa trong thư viện `stdlib.h`) dùng thoát chương trình chính và trả về mã lỗi là 0.

### **III. Tổng kết:**

Sau bài học này chúng ta đã biết cách xây dựng cho mình một chương trình ứng dụng Linux đơn giản xuất ký tự ra màn hình hiển thị. Sử dụng thành thạo hơn những thao tác biên dịch chương trình. Chúng ta còn biết cách cho phép một tập tin trong Linux được quyền thực thi để khắc phục lỗi "permission denied" trong hệ thống Linux. Kỹ thuật sử dụng hàm `printf()` rất quan trọng trong việc thông báo lỗi trong quá trình lập trình ứng dụng, phục vụ cho việc gỡ lỗi lập trình. Chúng ta cũng đã biết cách sử dụng hàm `exit(n)` trong việc báo lỗi chương trình.

Trong bài tiếp theo, chúng ta sẽ đi vào tìm hiểu những lệnh về trì hoãn thời gian trong hệ thống Linux. Các lệnh này rất quan trọng trong quá trình lập trình những ứng dụng có liên quan đến định thời chính xác về thời gian, hay nói đúng hơn là những công việc mang tính chu kỳ.



**BÀI 2****TRÌ HOÃN THỜI GIAN  
TRONG USER APPLICATION****I. Kiến thức ban đầu:**

Trì hoãn thời gian (delay), là một trong những thao tác quan trọng nhất trong quá trình lập trình ứng dụng giao tiếp với các thiết bị ngoại vi, lập trình hoạt động mang tính chất chu kỳ. Trong các ứng dụng, chúng ta sử dụng lệnh delay để làm những công việc như: định thời gian cập nhật thông tin về ngày-tháng-năm hiện tại, thông tin về nhiệt độ hiện tại, hay định thời gian kiểm tra tình trạng hoạt động của một bộ phận nào đó trong hệ thống nhằm điều khiển những ngõ vào ra thích hợp. Các thiết bị ngoại vi (như LCD, các thiết bị lưu trữ (ổ đĩa VCD, ổ đĩa cứng, thẻ nhớ, ...)) thông thường có thời gian đáp ứng thấp hơn rất nhiều lần so với tốc độ làm việc của CPU. Nhằm mục đích đồng bộ hóa hoạt động giữa CPU và các thiết bị ngoại vi, một trong những biện pháp đơn giản nhất là trì hoãn hoạt động của CPU, làm cho CPU phải chờ các thiết bị làm việc xong rồi mới tiếp tục công việc tiếp theo.

*\*\*Trong thực tế người ta dùng biện pháp khác hiệu quả hơn rất nhiều. Đó là dùng ngắt điều khiển hoạt động của CPU. Thay vì bắt CPU phải chờ, CPU sẽ tiến hành làm công việc khác. Khi thiết bị ngoại vi thực thi xong nhiệm vụ nó phát sinh ra một tín hiệu báo cho CPU biết để CPU nhận thông tin hay ra lệnh tiếp theo cho thiết bị hoạt động. Vấn đề ngắt sẽ được trình bày trong những phần tiếp theo của quyển sách này.*

Trì hoãn thời gian trong hệ thống Linux khác với trì hoãn thời gian thông thường chúng ta đã hiểu là trong lúc trì hoãn thời gian, CPU không làm gì cả ngoài việc tiêu tốn thời gian trong việc chờ đợi. Trong hệ điều hành Linux nói riêng và các hệ điều hành khác nói chung, điều có một cơ chế quản lý thời gian hoạt động của từng tiến trình và từng tuyến. Nghĩa là các tiến trình và tuyến được cho phép hoạt động song song trong hệ thống. (*\*\*Định nghĩa tiến trình và tuyến sẽ được trình bày trong những bài tiếp theo*). Khi một tiến trình sử dụng lệnh trì hoãn thời gian, CPU không tiêu tốn thời gian hoạt động cho việc chờ đợi, mà nó sẽ thực hiện tiến trình khác, khi thực hiện xong tất cả những tiến trình đó, nó quay trở lại tiến trình cũ có trì hoãn thời gian và kiểm tra xem lệnh trì hoãn thời gian có hoàn thành hay chưa. Nếu chưa hoàn thành, hệ điều hành tiếp

tục thực hiện những tiến trình khác. Nếu hoàn thành, hệ điều hành sẽ thực hiện những lệnh tiếp theo trong tiến trình hiện tại. Quá trình chia khe thời gian vẫn tiếp tục. Hoạt động của hệ điều hành cũng tương tự khi có nhiều tiến trình sử dụng hàm trì hoãn thời gian. Trong hệ điều hành nhúng Linux dùng cho Kit KM9260, lệnh trì hoãn thời gian sử dụng hoạt động của timer định thời. Khi một lệnh trì hoãn thời gian được khởi động, timer sẽ được gán một giá trị cùng với những thông số cài đặt phù hợp sao cho có khả năng trì hoãn đúng bằng khoảng thời gian mong muốn. Timer bắt đầu chạy, khi đạt đến điều kiện định thời, timer sinh ra một ngắt báo cho CPU biết để tiến hành thoát khỏi lệnh khi hoãn hiện tại.

Bài này sẽ giới thiệu cho chúng ta những phương pháp trì hoãn thời gian cơ bản thường được sử dụng trong quá trình lập trình ứng dụng cho một hệ thống nhúng. Các ưu và nhược điểm khi dùng từng phương pháp để người lập trình có thể lựa chọn cho mình phương pháp hiệu quả nhất phù hợp với yêu cầu ứng dụng. Bên cạnh đó chúng ta sẽ được tìm hiểu các lệnh về Realtime trong hệ điều hành Linux. Những kiến thức này có vai trò quan trọng trong việc thiết kế ứng dụng điều khiển LCD sau này và những ứng dụng có liên quan đến thời gian thực khác;

Có hai lớp chúng ta có thể thực hiện những lệnh về thời gian thực, lớp driver và user application. Ở đây trong phần lập trình ứng dụng chúng ta chỉ đề cập những lệnh thời gian thực trong lớp user application. Còn những lệnh thời gian thực trong driver sẽ được đề cập trong phần lập trình driver của quyền sách này.

## **II. Nội dung:**

### **1. Hàm sleep:**

#### **a. Cú pháp:**

```
#include <unistd.h>
void sleep (unsigned int seconds);
hay
unsigned int sleep(unsigned int seconds);
```

#### **b. Giải thích:**

Hàm sleep làm nhiệm vụ trì hoãn một khoảng thời gian, có đơn vị là giây. Trì hoãn ở đây không phải là tiêu tốn thời gian làm việc của CPU trong việc chờ đợi thời gian trì hoãn kết thúc mà hệ điều hành vẫn tiếp tục chia khe cho CPU thực hiện những công việc

khác. Nghĩa là trong lúc trì hoãn, CPU vẫn làm những công việc khác do hệ điều hành phân công thay vì chỉ chờ đợi hết thời gian.

Hàm `sleep` có một đối số và dữ liệu trả về. Đối số `unsigned int seconds` là khoảng thời gian cần trì hoãn, tính bằng đơn vị giây. Hàm `sleep` trả về giá trị 0 khi thời gian trì hoãn đã hết, trả về số giây còn lại khi chưa kết thúc thời gian trì hoãn.

Chúng ta có thể không cần sử dụng dữ liệu trả về của hàm, bằng cách dùng trường hợp thứ nhất `void sleep (unsigned int seconds);`

Hàm `sleep` được dùng trong những trường hợp trì hoãn thời gian không đòi hỏi tính chính xác cao. Thông thường là những khoảng thời gian lớn, vì đơn vị trì hoãn nhỏ nhất là giây.

**c. Ví dụ:**

Đoạn chương trình sau sẽ minh họa cho chúng ta cách sử dụng hàm `sleep()`.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    printf ("Delay in 10s ...\n");
    sleep(10);
    printf ("Delay finished.\n");
    exit(0);
}
```

Chương trình này làm nhiệm vụ in ra chuỗi "Delay in 10s ..." để thông báo rằng hàm `sleep` bắt đầu thực thi. Sau khoảng thời gian trì hoãn 10s, chương trình tiếp tục in ra chuỗi "Delay finished." thông báo rằng hàm `sleep` chấm dứt thực thi.

Chúng ta tiến hành biên dịch chương trình trên nạp vào kit. Kết quả khi thực thi chương trình như sau:

```
Delay in 10s ...
(Chương trình trì hoãn 10s)
Delay finished.
```

**2. Hàm `usleep`:**

**a. Cú pháp:**

```
#include <unistd.h>

void usleep (unsigned long usec);
```

hay

```
int usleep (unsigned long usec);
```

**b. Giải thích:**

Cũng tương tự như hàm `sleep()`, hàm `usleep` cũng làm nhiệm vụ trì hoãn một khoảng thời gian theo yêu cầu của người lập trình. Tuy nhiên có một điểm khác biệt là đơn vị thời gian trì hoãn tính bằng micro giây. Nghĩa là độ phân giải nhỏ hơn 1 triệu lần. Hàm `usleep` ứng dụng trong việc trì hoãn thời gian một cách chính xác cao và không dài (nhỏ hơn 1 giây).

Hàm `usleep()` có hai dạng sử dụng. Một dạng không có giá trị trả về và một dạng có giá trị trả về. Giá trị trả của hàm `usleep` là 0 nếu hoàn tất thời gian trì hoãn, ngược lại sẽ trả về thời gian trì hoãn còn lại. Đối số của hàm `usleep` là một số nguyên không dấu có độ lớn là 4 bytes (dạng unsigned long) chính là số micro giây cần trì hoãn.

**c. Ví dụ:**

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    printf ("Delay in 10us ...\n");
    usleep(10);
    printf ("Delay finished.\n");
    exit(0);
}
```

Chương trình này làm nhiệm vụ in ra chuỗi “Delay in 10us ...” để thông báo rằng hàm `usleep` bắt đầu thực thi. Sau khoảng thời gian trì hoãn 10s, chương trình tiếp tục in ra chuỗi “Delay finished.” thông báo rằng hàm `sleep` chấm dứt thực thi.

Chúng ta tiến hành biên dịch chương trình trên nạp vào kit. Kết quả khi thực thi chương trình như sau:

```
Delay in 10us ...
(Chương trình trì hoãn 10us)
Delay finished.
```

Do thời gian trì hoãn là rất ngắn nên chúng ta chỉ thấy hai dòng chữ xuất hiện cùng một lúc. Nếu muốn nhìn thấy hai dòng chữ trên xuất hiện tuần tự, chúng ta tiến hành thay đổi thông số như sau: `usleep(1000000)`; thì hệ thống sẽ trì hoãn 1s.

### 3. Hàm `nanosleep`:

#### a. Cú pháp:

```
#include <time.h>

int nanosleep (const struct timespec *req, struct timespec *rem);
```

#### b. Giải thích:

Để hiểu cách sử dụng của hàm `nanosleep` trước tiên chúng ta phải hiểu rõ cấu trúc `timespec` trong thư viện `<time.h>`.

Cấu trúc `struct timespec` được Linux định nghĩa như sau:

```
struct timespec {
    __kernel_time_t tv_sec;    /* seconds */
    long            tv_nsec;   /* nanoseconds */
};
```

Cấu trúc `timespec` bao gồm có hai phần tử con, `__kernel_time_t tv_sec` và `long tv_nsec`. `timespec` là cấu trúc thời gian trong Linux, bao gồm có giây (`tv_sec`) và nano giây (`tv_nsec`) của giây hiện tại. Ví dụ để tạo một khoảng thời gian là 1000 nano giây, chúng ta làm các bước như sau:

```
/*Định nghĩa biến thuộc cấu trúc timespec*/
struct timespec time_delay;
/*Gán giá trị thời gian cần trì hoãn cho biến*/
time_delay.tv_sec = 0;
time_delay.tv_nsec = 1000;
```

Như vậy thông qua việc gán giá trị cho các phần tử trong cấu trúc `timespec`, chúng ta đã tạo ra được một khoảng thời gian định thời cần thiết cho việc sử dụng hàm `nanosleep` sau đây.

Bây giờ chúng ta sẽ đi vào tìm hiểu các tham số trong hàm `nanosleep`. Hàm `nanosleep` có hai tham số cần quan tâm. `const struct timespec *req` là con trỏ trỏ đến địa chỉ của cấu trúc `timespec` được định nghĩa trước đó. Theo như các bước làm trước thì chúng ta đã định nghĩa biến `time_delay` có thời gian định thời là 1000 nano

giây. Địa chỉ của biến này sẽ được gán vào đối số thứ nhất của hàm `nanosleep`. Đối số thứ hai của hàm `nanosleep` cũng là một cấu trúc thời gian `timespec` trong Linux, `struct timespec *rem`. `rem` cũng là một con trỏ trỏ đến địa chỉ của một biến cấu trúc `timespec` đã định nghĩa trước đó. `*rem` làm nhiệm vụ lưu giá trị thời gian còn lại khi quá trình trì hoãn chưa kết thúc (kết thúc bị lỗi). Nếu không muốn lưu giá trị thời gian này, chúng ta có thể không cần sử dụng tham số `struct timespec *rem` của `nanosleep` bằng cách khai báo trong tham số thứ hai hằng số `NULL`. Hàm `nanosleep` có giá trị trả về kiểu `int`, trả về giá trị 0 nếu quá trình trì hoãn không có lỗi. Ngược lại, trả về giá trị -1. Trong trường hợp này, giá trị thời gian còn lại sẽ được lưu vào trong biến con trỏ `*rem` nếu đã khai báo trong đối số thứ hai. Người lập trình có thể sử dụng giá trị này để gọi một hàm trì hoãn khác, trì hoãn phần còn lại chưa hoàn thành.

Hàm `nanosleep` có một ưu điểm là thời gian trì hoãn có độ phân giải cao hơn rất nhiều so với hàm `sleep` và `usleep`. Hàm `nanosleep` sử dụng trong những ứng dụng đòi hỏi trì hoãn với độ chính xác rất cao (tính bằng một phần tỷ của giây).

### ***c. Ví dụ:***

Đoạn chương trình sau sẽ giúp cho chúng ta biết cách sử dụng hàm `nanosleep`.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main() {
    /*Khai báo cấu trúc thời gian dùng lưu trữ giá trị khoảng thời gian cần trì hoãn*/
    struct timespec rq, rm;
    /*Cài đặt khoảng thời gian cần trì hoãn*/
    rq.tv_sec = 0;
    rq.tv_nsec = 1000;
    /*In thông báo quá trình trì hoãn bắt đầu*/
    printf ("Delay has just begun ...\n");
    /*Gọi hàm nanosleep trì hoãn một khoảng thời gian đã cài đặt*/
    nanosleep (&rq, &rm);
    /*In thông báo quá trình trì hoãn kết thúc*/
    printf ("Delay has just finished. \n");
}
```

```
/*Kết thúc chương trình*/  
exit (0);  
}
```

Như vậy để sử dụng được hàm `nanosleep` chúng ta phải làm theo những bước sau:

- Khai báo biến `struct timespec` để lưu giá trị khoảng thời gian cần trì hoãn, và biến `struct timespec` để chứa giá trị trả về của hàm `nanosleep`;

```
struct timespec rq, rm;
```

- Gán giá trị thời gian cần trì hoãn;

```
rq.tv_sec = 0;
```

```
rq.tv_nsec = 1000;
```

- Gọi hàm `nanosleep`; Gán địa chỉ của biến vào các con trỏ tham số của hàm;

```
nanosleep (&rq, &rm);
```

Biên dịch và chạy chương trình trên kit, màn hình hiển thị kết quả như sau:

```
Delay has just begun ...
```

```
(Chương trình trì hoãn 1000ns)
```

```
Delay finished.
```

Do thời gian trì hoãn rất nhỏ nên hai chuỗi thông tin xuất hiện gần như đồng thời. Có thể tăng thời gian trì hoãn bằng cách thay đổi thông số của `tv_sec` và `tv_nsec`, chẳng hạn như:

```
req.tv_sec = 1;
```

```
req.tv_nsec = 0;
```

thì thời gian trì hoãn sẽ tăng lên 1s, lúc này chúng ta có thể thấy 2 dòng thông tin xuất hiện tuần tự.

#### **4. Hàm `alarm`:**

##### **a. Cú pháp:**

```
#include <unistd.h>
```

```
unsigned int alarm (unsigned int seconds);
```

##### **b. Giải thích:**

Hàm `alarm()` tạo một tín hiệu có tên là `SIGALRM` sau một khoảng thời gian là `seconds` giây, được quy định bởi người lập trình. Nếu giá trị `seconds` bằng 0 thì hàm `alarm` bị vô hiệu hóa.

Hàm alarm thường được dùng kết hợp với hàm xử lý tín hiệu ngắt, hàm này làm nhiệm vụ thực hiện những công việc do người lập trình định nghĩa khi tín hiệu ngắt xảy ra. Thông thường chúng ta sử dụng hàm sigaction() hay signal() để thực hiện nhiệm vụ trên.

**c. Ví dụ:**

Đoạn chương trình sau đây sẽ cho chúng ta biết cách tạo một khoảng thời gian định thời cho tín hiệu SIGALRM tác động:

```
alarm (30); /*Sau một khoảng thời gian là 30s, thì tín hiệu SIGALRM sẽ tác
động tích cực, công việc của chúng ta là xây dựng một hàm thực thi khi có tín hiệu tác
động*/
```

```
/*Sau khi tín hiệu SIGALRM tác động thì cần phải tạo lại định thời cho tín hiệu này
bằng cách gọi lại hàm SIGALRM*/
```

Hàm alarm() dùng để định thời tạo ra một tín hiệu ngắt. Thông thường hàm này không dùng như một hàm riêng biệt mà nó phải kết hợp với những hàm khác. Đó là những hàm xử lý tín hiệu ngắt.

**5. Kiến thức căn bản về đồng hồ thời gian thực trong hệ thống Linux:**

Thời gian thực trong hệ điều hành mang một ý nghĩa rất rộng. Trong phần này chúng ta nói đến thời gian thực với ngụ ý là đồng hồ thời gian thực trong hệ thống Linux. Trong phần này chúng ta sẽ nghiên cứu những lệnh truy xuất những thông tin về thời gian trong hệ thống (bao gồm ngày, tháng, năm, giờ, phút, giây, ...);

Những hàm truy xuất về thời gian được định nghĩa trong thư viện <time.h>. Các hàm trong thư viện <time.h> bao gồm:

- Hàm thao tác với thời gian thực;
  - o clock: Trả về khoảng thời gian (đơn vị là ticks) kể từ khi chương trình thực thi;
  - o difftime: Trả về khoảng cách thời gian giữa hai thời điểm;
  - o time: Trả về thời gian hiện tại của hệ thống;
- Hàm chuyển đổi thời gian:
  - o asctime: chuyển đổi kiểu cấu trúc thời gian tm thành chuỗi thời gian;
  - o ctime: chuyển đổi kiểu dữ liệu thời gian time\_t thành chuỗi thời gian;



- `gmtime`: chuyển đổi kiểu dữ liệu thời gian `time_t` thành cấu trúc thời gian `tm` theo giờ UTC;
- `mktime`: Chuyển đổi cấu trúc thời gian `tm` thành loại dữ liệu thời gian `time_t`;
- `localtime`: Chuyển đổi loại dữ liệu thời gian `time_t` thành cấu trúc thời gian `tm` theo thời gian cục bộ;
- `strftime`: Chuyển cấu trúc thời gian `struct tm` thành chuỗi theo định dạng của người dùng;
- Các hằng số xây dựng sẵn bao gồm:
  - `CLOCKS_PER_SEC`: là khoảng thời gian 1s tính bằng đơn vị ticks;
  - `NULL`: Con trỏ rỗng;
- Các loại dữ liệu thời gian chuẩn:
  - `clock_t`: Loại dữ liệu thời gian dạng xung ticks;
  - `size_t`;
  - `time_t`: kiểu dữ liệu thời gian;
  - `struct tm`: cấu trúc thời gian;

Chúng ta sẽ đi sâu tìm hiểu từng nội dung đã liệt kê trên đây:

## **5.1. Các loại dữ liệu thời gian chuẩn:**

### **5.1.1. *clock\_t*:**

Theo thư viện `<time.h>` thì `clock_t` được định nghĩa như sau:

```
typedef __kernel_clock_t      clock_t;  
/*Trong thư viện <types.h>*/
```

Mà `__kernel_clock_t` được định nghĩa:

```
typedef long      __kernel_clock_t;  
/*Trong thư viện <posix_types.h>*/
```

Như vậy, `clock_t` thực chất là một kiểu số nguyên `long`. Nhiệm vụ của biến này là lưu những giá trị khoảng thời gian (tính bằng ticks) do các hàm thao tác trên thời gian trả về.

### **5.1.2. *size\_t*:**

Theo thư viện `<time.h>` thì `size_t` được định nghĩa như sau:

```
typedef __kernel_size_t size_t;
```

*/\*Trong thư viện <types.h>\*/*

mà `__kernel_size_t` được định nghĩa:

```
typedef unsigned long    __kernel_size_t;
```

*/\*Trong thư viện <posix\_types.h>\*/*

Như vậy, `size_t` thực chất là một kiểu số nguyên `unsigned long`. Nhiệm vụ của biến này là lưu những giá trị kích thước của một biến nào đó, do hàm `sizeof`, `maxsize`, ... trả về.

### **5.1.3. `time_t`:**

Kiểu dữ liệu `time_t` được định nghĩa trong thư viện `<time.h>` như sau:

```
typedef __kernel_time_t  time_t;
```

*/\*Trong thư viện <types.h>\*/*

mà `__kernel_time_t` được định nghĩa:

```
typedef long    __kernel_time_t;
```

*/\*Trong thư viện <posix\_types.h>\*/*

Như vậy, `size_t` thực chất là một kiểu số nguyên `long`. Nhiệm vụ của loại biến này là lưu giá trị thời gian được trả về bởi các hàm về thời gian như `time`, `difftime`, ...

**5.1.4. struct tm:**

Trong thư viện <time.h> struct tm được định nghĩa như sau:

```
struct tm {  
    /*  
     * Số giây trong một phút  
     * giá trị nằm trong khoảng từ 0 đến 59  
     */  
    int tm_sec;  
    /* Số phút trong một giờ, có giá trị từ 0 đến 59 */  
    int tm_min;  
    /* Số giờ trong một ngày có giá trị từ 0 đến 23 */  
    int tm_hour;  
    /* Số ngày trong một tháng, giá trị trong khoảng 0 đến 31 */  
    int tm_mday;  
    /* Số tháng trong một năm, giá trị nằm trong khoảng 0 đến 11 */  
    int tm_mon;  
    /* Số năm kể từ năm 1900 */  
    long tm_year;  
    /* Số ngày trong tuần, có giá trị từ 0 đến 6 */  
    int tm_wday;  
    /* Số ngày trong một năm, tính từ ngày 1 tháng 1, có giá trị từ 0 đến 365 */  
    int tm_yday;  
};
```

Chúng ta thấy cấu trúc struct tm là một cấu trúc xây dựng sẵn với tất cả những thông tin liên quan đến thời gian thực: Năm, tháng, ngày, giờ, phút, giây, ngày trong tuần, ... Cấu trúc này đúng làm kiểu dữ liệu trả về cho các hàm thao tác với thời gian. Chúng ta sẽ nghiên cứu ngay trong phần sau của chương.

**5.2. Các hằng số xây dựng sẵn trong <time.h>:****5.2.1. CLOCKS\_PER\_SEC:**

Đây là một hằng số được định nghĩa sẵn quy định số ticks trong một giây. Như vậy trong một giây sẽ có CLOCKS\_PER\_SEC ticks. Chúng ta dùng hằng số này để quy đổi

ra giây giá trị trả về của hàm `clock()`. Hay dùng làm hằng số qui đổi ra giây của tham số của một số hàm trì hoãn thời gian tính bằng ticks.

### 5.2.2. *NULL:*

`NULL` là một hằng số quy định một con trỏ rỗng, hay nói cách khác con trỏ này không trỏ đến bất kỳ một đối tượng nào cả. Hằng số này sử dụng trong những trường hợp khi muốn vô hiệu hóa một tham số nào đó của hàm (khi không muốn sử dụng một tham số của hàm nào đó).

*\*\*Trên đây chúng ta đã tìm hiểu những loại dữ liệu, hằng số quan trọng của thời gian thực trong Linux. Sau đây là những hàm sử dụng những loại dữ liệu này. Mỗi hàm chúng ta sẽ có một ví dụ minh họa cách sử dụng, các bạn cần phải đọc hiểu và biên dịch chạy trên hệ thống Linux nhằm nắm vững thêm nguyên lý hoạt động của chúng. Từ đó, sẽ có thể ứng dụng những hàm này trong những ứng dụng khác liên quan đến thời gian thực.*

## 5.3. Các hàm thao tác với thời gian thực:

### 5.3.1. *clock:*

#### a. *Cú pháp:*

```
#include <time.h>
clock_t clock (void);
```

#### b. *Giải thích:*

Hàm `clock` có kiểu dữ liệu trả về là `clock_t` và không có tham số.

Nhiệm vụ của hàm `clock` là trả về khoảng thời gian tính từ khi chương trình chứa hàm `clock` thực thi cho đến khi hàm `clock` được gọi.

Chương trình ví dụ sau sẽ cho chúng ta biết rõ cách sử dụng hàm này hơn.

#### c. *Chương trình ví dụ:*

Mục đích của chương trình này là tạo ra một chương trình trì hoãn. Thời gian trì hoãn tính bằng giây do người lập trình quy định.

```
/*chương trình trì hoãn thời gian ứng dụng hàm clock*/
/*Khai báo thư viện dùng cho hàm printf*/
#include <stdio.h>
/*Khai báo thư viện dùng cho các hàm thời gian*/
```

```
#include <time.h>

/*Tạo hàm wait trì hoãn thời gian. Hàm này có tham số là seconds giây cần trì hoãn,
không có giá trị trả về */
void wait (int seconds) {
    /*Khai báo biến loại clock_t để lưu dữ liệu kiểu ticks*/
    clock_t endwait;

    /*Chọn khoảng thời gian kết thúc là thời gian hiện tại, cộng với khoảng thời gian là
    một giây*seconds*/
    endwait = clock () + seconds * CLOCKS_PER_SEC;

    /*Chờ cho đến khi thời gian hiện tại lớn hơn thời gian kết thúc thì thoát*/
    while (clock () < endwait) {}
}

/*Chương trình chính ứng dụng hàm wait(seconds)*/
int main () {
    /*Khai báo biến int n để lưu giá trị đếm xuống*/
    int n;

    /*In câu thông báo bắt đầu đếm xuống*/
    printf ("Starting countdown...\n");

    /*Vòng lặp for đếm xuống*/
    for (n = 10; n>0; n--) {

        /*In giá trị hiện tại của n để theo dõi*/
        printf ("%d\n", n);

        /*Gọi hàm wait đã được lập trình trước đó trì hoãn một khoảng thời gian là 1 giây*/
        wait(1);
    }

    /*Sau khi thoát khỏi vòng lặp, in ra câu thông báo kết thúc*/
    printf ("FIRE!!!\n");

    /*Trả về giá trị 0 cho hàm main, để thông báo là không có lỗi xảy ra*/
    return 0;
}
```

Ý nghĩa từng dòng lệnh đã được giải thích rất kỹ trong chương trình. Chúng ta tiến hành biên dịch và sao chép vào kit. Sau khi tiến hành thực thi chương trình chúng ta sẽ có kết quả như sau:

```
./Time_0
Starting countdown...
10
9
8
7
6
5
4
3
2
1
FIRE!!!
```

Như vậy chúng ta đã thực hiện thành công chương trình trì hoãn thời gian sử dụng hàm clock để lấy thời gian hiện tại tính từ lúc chương trình thực thi và sử dụng hằng số CLOCKS\_PER\_SEC. Tuy nhiên mục đích của chương trình trên không phải là để tạo ra một cách trì hoãn thời gian khác mà chỉ giúp chúng ta hiểu rõ cách sử dụng hàm clock. Thực ra, trì hoãn thời gian theo cách này không hiệu quả cho những ứng dụng lớn, vì tiêu tốn thời gian hoạt động của CPU vô ích.

### **5.3.2. *difftime:***

#### **a. *Cú pháp:***

```
#Include <time.h>
double difftime (time_t time2, time_t time1);
```

#### **b. *Giải thích:***

Hàm difftime có hai đối số kiểu time\_t. Đối số thứ nhất time\_t time2 là thời điểm ban đầu, đối số thứ hai time\_t time1 là thời điểm lúc sau. Hàm difftime có kiểu dữ liệu trả về là số thực dạng double là độ chênh lệch thời gian tính bằng giây giữa hai thời điểm time2 và time1.

#### **c. *Chương trình ví dụ:***

Chương trình này làm nhiệm vụ tính toán thời gian nhập dữ liệu của người dùng, sử dụng hàm difftime để tìm sự chênh lệch thời gian giữa hai lần cập nhật (trước và sau khi nhập dữ liệu).

```
/*chương trình ví dụ hàm difftime*/
/*Khai báo thư viện dùng cho hàm printf và hàm gets*/
#include <stdio.h>
/*Khai báo thư viện dùng cho các hàm về thời gian*/
#include <time.h>
/*Chương trình chính*/
int main (void) {
    /*Khai báo biến start: lưu thời điểm ban đầu, end: lưu thời điểm lúc sau thuộc kiểu dữ liệu time_t*/
    time_t start, end;
    /*Khai báo biến lưu tên người dùng nhập vào, chứa tối đa 256 ký tự*/
    char szInput [256];
    /*Khai báo biến số thực lưu giá trị khoảng thời gian khác biệt của hai thời điểm start và end*/

    /*Cập nhật thời gian hiện tại, trước khi người dùng nhập thông tin. Sử dụng hàm time(), là hàm trả về thời gian hiện tại của hệ thống, kiểu dữ liệu trả về là con trỏ time_t*/
    time (&start);
    /*In ra thông báo yêu cầu người sử dụng nhập thông tin cho hệ thống.*/
    printf ("Please, enter your name: ");
    /*Yêu cầu nhập vào biến mảng szInput*/
    gets (szInput);
    /*Cập nhật thời gian hiện tại, sau khi người dùng đã nhập thông tin. Sử dụng hàm time(time_t *time_val) để lấy thời gian hiện tại*/
    time (&end);
    /*Sử dụng hàm difftime để tính thời gian chênh lệch giữa start và end*/
    dif = difftime (end, start);
    /*In ra thông báo đã nhận được thông tin người dùng*/
    printf ("Hi %s. \n", szInput);
    /*In ra thông báo khoảng thời gian tiêu tốn khi đợi nhập dữ liệu*/
    printf ("It took you %.2lf seconds to type your name. \n", dif);
    /*Trả về 0 cho hàm main báo rằng quá trình thực thi không bị lỗi.*/
```

```
return 0;
}
```

Tiến hành biên dịch và chạy chương trình trên kit, chúng ta có kết quả như sau:

```
./time_difftime_linux
Please, enter your name: Nguyen Tan Nhu
Hi Nguyen Tan Nhu.
It took you 5.00 seconds to type your name.
```

Trước khi gọi hàm `gets()`, chúng ta gọi hàm `time()` để lấy về thời gian hiện tại của hệ thống, sau khi người sử dụng nhập xong dữ liệu, ta lại gọi hàm `time()` để cập nhật thời gian lúc sau. Tiếp đến gọi hàm `difftime` để lấy về khoảng thời gian sai biệt giữa hai thời điểm. Phần sau chúng ta sẽ giải thích rõ hơn hoạt động của hàm `time` trong hệ thống linux.

### **5.3.3. *time:***

#### **a. *Cú pháp:***

```
#include <time.h>
time_t time(time_t* timer);
```

#### **b. *Giải thích:***

Hàm `time` dùng để lấy về thời gian hiện tại của hệ thống linux. Hàm `time` có đối số `time_t* time` là con trỏ lưu giá trị trả về của hàm. Đối số này có thể bỏ trống bằng cách điền hằng `NULL` lúc này thời gian hiện tại sẽ là giá trị trả về của hàm.

#### **c. *Chương trình ví dụ:***

Chương trình ví dụ sau đây sẽ cho chúng ta hiểu rõ cách sử dụng hàm `time` (Chúng ta có thể tham khảo ví dụ trên để xem cách sử dụng hàm `time`). Chương trình này sẽ in ra số giờ của hệ thống kể từ ngày 1 tháng 1 năm 1970.

```
/*Chương trình ví dụ hàm time*/
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Bắt đầu chương trình chính*/
int main () {
/*Khai báo biến time_t seconds làm giá trị trả về cho hàm time*/
    time_t seconds;
```



*/\*Sử dụng hàm time lấy về thời gian hiện tại của hệ thống, là số giây kể từ ngày 1 tháng 1 năm 1970\*/*

```
seconds = time (NULL);
```

*/\*In ra thời gian hiện tại\*/*

```
printf ("%ld hours since January 1st, 1970\n", seconds/3600);
```

*/\*Trả về giá trị 0 cho hàm main, thông báo quá trình thực thi không có lỗi\*/*

```
return 0;
```

```
}
```

Biên dịch và thực thi chương trình trên kit, chúng ta có kết quả như sau:

```
./time_time_linux
```

```
366958 hours since January 1, 1970
```

Như vậy hàm `time` làm nhiệm vụ trả về thời gian hiện tại của hệ thống tính bằng giây kể từ ngày 1 tháng 1 năm 1970. Chúng ta có thể chuyển thành dạng ngày tháng năm, giờ phút giây bằng cách sử dụng hàm `gmtime` hay `localtime`.

#### **5.4. Các hàm chuyển đổi thời gian thực:**

##### **5.4.1. *asctime:***

###### **a. *Cú pháp:***

```
#include <time.h>
```

```
char* asctime (const struct tm * timeptr);
```

###### **b. *Giải thích:***

Hàm `asctime` có chức năng biến đổi cấu trúc thời gian dạng `struct tm` thành chuỗi thời gian có dạng `Www Mmm dd hh:mm:ss yyyy`. Trong đó `Www` là ngày trong tuần, `Mmm` là tháng trong năm, `dd` là ngày trong tháng, `hh:mm:ss` là giờ:phút:giây trong ngày, cuối cùng `yyyy` là năm.

Chuỗi thời gian được kết thúc bằng 2 ký tự đặc biệt: ký tự xuống dòng “\n” và ký tự `NULL`.

Hàm có tham số là con trỏ cấu trúc `struct tm * timeptr`, giá trị trả về là con trỏ ký tự `char *`

Ví dụ sau đây sẽ cho chúng ta hiểu rõ cách sử dụng hàm `asctime`.

###### **c. *Chương trình ví dụ:***

Nhiệm vụ của đoạn chương trình này là in ra chuỗi thời gian trả về của hàm `asctime`.

*/\*Chương trình ví dụ hàm asctime\*/*

```
/*Khai báo các thư viện cho các hàm dùng trong chương trình*/
#include <stdio.h>
#include <time.h>

/*Chương trình chính*/
int main () {
/*Khai báo biến time_t rawtime để lưu thời gian hiện tại của hệ thống*/
time_t rawtime;
/*Khai báo biến con trỏ cấu trúc struct tm để lưu thời gian*/
struct tm * timeinfo;
/*Đọc về thời gian hiện tại của hệ thống*/
time ( &rawtime );
/*Chuyển đổi kiểu thời gian time_t thành cấu trúc thời gian struct tm*/
timeinfo = localtime ( &rawtime );
/*In thời gian hệ thống dưới dạng chuỗi*/
printf ("The current date/time is: %s", asctime (timeinfo) );
/*Trả về giá trị 0 cho hàm main để thông báo không có lỗi xảy ra trong quá trình thực thi lệnh*/
return 0;
}
```

Sau khi biên dịch và chạy chương trình trên kit, chúng ta có kết quả sau:

```
./time_asctime_linux
The current date/time is: Sat Nov 12 06:04:50 2011
```

Hàm asctime thường dùng trong những trường hợp cần biến một cấu trúc thời gian thành chuỗi ký tự ascii phục vụ cho việc hiển thị trên các thiết bị đầu cuối, hay trong các tập tin cần quản lý về thời gian.

#### **5.4.2. ctime:**

##### **a. Cú pháp:**

```
#include <time.h>
char * ctime ( const time_t * timer );
```

##### **b. Giải thích:**

Hàm ctime làm nhiệm vụ chuyển thông tin về thời gian dạng time\_t thành thông tin về thời gian dạng chuỗi ký tự mã ascii. Chuỗi thông tin thời gian này có dạng Www

Mmm dd hh:mm:ss yyyy. Trong đó Www là ngày trong tuần, Mmm là tháng trong năm, dd là ngày trong tháng, hh:mm:ss là giờ:phút:giây trong ngày, cuối cùng yyyy là năm.

Chuỗi thời gian được kết thúc bằng 2 ký tự đặc biệt: ký tự xuống dòng “\n” và ký tự NULL.

Hàm ctime có tham số là con trỏ biến thời gian dạng `time_t * timer`, giá trị trả về của hàm là con trỏ ký tự `char *`

Chương trình ví dụ sau đây sẽ cho chúng ta hiểu rõ cách sử dụng hàm ctime.

### c. *Chương trình ví dụ:*

Tương tự như chương trình ví dụ trên, chương trình này cũng lấy thời gian hiện tại của hệ thống chuyển sang chuỗi thông tin về thời gian cuối cùng in ra màn hình hiển thị. Nhưng lần này chúng ta sử dụng hàm ctime.

```
/*Chương trình ví dụ hàm ctime*/
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Chương trình chính*/
int main () {
    /*Khai báo biến kiểu thời gian time_t lưu thời gian hiện tại của hệ thống*/
    time_t rawtime;
    /*Đọc về giá trị thời gian hiện tại của hệ thống*/
    time ( &rawtime );
    /*In chuỗi thông tin thời gian ra màn hình*/
    printf ( "The current local time is: %s", ctime (&rawtime) );
    /*Trả về giá trị 0 cho hàm main, thông báo rằng quá trình thực thi lệnh không có lỗi*/
    return 0;
}
```

Biên dịch và chạy chương trình trên kit, chúng ta có kết quả như sau:

```
./time_ctime_linux
The current local time is: Sat Nov 12 07:03:28 2011
```

Kết quả hiển thị cũng tương tự như hàm asctime. Nhưng hàm ctime có nhiều ưu điểm hơn hàm asctime. Chúng ta không cần phải chuyển thông tin thời gian dạng `time_t`

thành dạng `struct tm` rồi thực hiện chuyển đổi sang thông tin thời gian dạng chuỗi. Hàm `ctime` sẽ chuyển trực tiếp `time_t` thành chuỗi thời gian.

### **5.4.3. *gmtime:***

#### **a. *Cú pháp:***

```
#include <time.h>

struct tm * gmtime ( const time_t * timer);
```

#### **b. *Giải thích:***

Hàm `gmtime` làm nhiệm vụ chuyển đổi kiểu thời gian dạng `time_t` thành cấu trúc thời gian `struct tm` phục vụ cho các hàm thao tác với thời gian thực khác.

Hàm có tham số là con trỏ thời gian kiểu `time_t`. Dữ liệu trả về là con trỏ cấu trúc thời gian dạng `struct tm`.

#### **c. *Chương trình ví dụ:***

Nhiệm vụ của chương trình ví dụ này là in ra giá trị thời gian giữa các múi giờ khác nhau, sử dụng hàm `gmtime` để trả về cấu trúc thời gian `struct tm`.

```
/*Chương trình ví dụ sử dụng hàm gmtime*/
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*Khai báo các hằng số dùng trong quá trình lập trình*/
#define MST (-7)
#define UTC (0)
#define CCT (+8)
/*Chương trình chính*/
int main () {
    /*Khai báo biến time_t để lưu thời gian hiện tại của hệ thống*/
    time_t rawtime;
    /*Khai báo biến con trỏ struct tm chứa dữ liệu trả về sau khi chuyển đổi */
    struct tm * ptm;
    /*Cập nhật giá trị thời gian hiện tại*/
    time (&rawtime);
    /*Chuyển sang cấu trúc thời gian tm*/
    ptm = gmtime ( &rawtime );
    /*In các giá trị thời gian ra màn hình*/
```

```
puts ("Current time around the world");
printf ("Phoenix, AZ (U.S.): %2d:%2d\n", (ptm->tm_hour+MST)%24,
ptm->tm_min);
printf ("Reykjavik (Iceland): %2d:%2d\n", (ptm->tm_hour+UTC)%24,
ptm->tm_min);
printf ("Beijing (China), AZ (U.S.): %2d:%2d\n", (ptm-
>tm_hour+CCT)%24, ptm->tm_min);
/*Trả về giá trị 0 cho hàm main*/
return 0;
}
```

Biên dịch và chạy chương trình trên kit, ta có kết quả sau:

```
./time_gmtime_linux
Current time around the World:
Phoenix, AZ (U.S.) : -5:20
Reykjavik (Iceland) : 2:20
Beijing (China) : 10:20
```

#### **5.4.4. mktime:**

##### **a. Cú pháp:**

```
#include <time.h>
time_t mktime ( struct tm * timeptr );
```

##### **b. Giải thích:**

Nhiệm vụ của hàm mktime là chuyển thông tin thời gian dạng cấu trúc struct tm thành thông tin thời gian dạng time\_t; Bên cạnh đó những thông tin có liên quan trong cấu trúc struct tm chưa được cập nhật cũng sẽ được thay đổi phù hợp với ngày tháng năm đã thay đổi. Cụ thể như thế nào sẽ được trình bày trong ví dụ sử dụng hàm mktime.

Hàm mktime có tham số là con trỏ cấu trúc struct tm \* là thời gian muốn chuyển đổi. Giá trị trả về là thời gian dạng time\_t.

Sau đây là ví dụ về cách sử dụng hàm mktime.

##### **c. Chương trình ví dụ:**

Nhiệm vụ của chương trình ví dụ này, người sử dụng sẽ nhập thông tin về ngày tháng năm, chương trình sẽ cho biết ngày đó là thứ mấy.

```
/*Chương trình ví dụ hàm mktime*/
/*Khai báo các thư viện cho các hàm sử dụng trong chương trình*/
```

```
#include <stdio.h>
#include <time.h>
/*Chương trình chính*/
int main () {
    /*Khai báo biến kiểu time_t lưu thông tin thời gian*/
    time_t rawtime;
    /*Khai báo biến cấu trúc lưu cấu trúc thời gian cần chuyển đổi*/
    struct tm * timeinfo;
    /*Khai báo biến lưu các ngày trong tuần*/
    char * weekday[] = { "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"};
    /*Khai báo các số nguyên lưu thông tin nhập từ người dùng*/
    int year, month, day;
    /*Yêu cầu nhập thông tin từ người dùng*/
    printf ("Enter year: "); scanf ("%d",&year);
    printf ("Enter month: "); scanf ("%d",&month);
    printf ("Enter day: "); scanf ("%d",&day);
    /*Cập nhật thời gian hiện tại*/
    time (&rawtime );
    /*Chuyển thành cấu trúc thời gian*/
    timeinfo = localtime ( &rawtime );
    /*Cập nhật thời gian do người dùng định nghĩa*/
    timeinfo->tm_year = year - 1900;
    timeinfo->tm_mon = month - 1;
    timeinfo->tm_mday = day;
    /*Khi gọi hàm mktime thì những thông tin còn lại như tm_wday sẽ được hệ thống tự động cập nhật*/
    mktime ( timeinfo );
    printf ( "That day is a %s. \n", weekday[timeinfo->wday]);
    /*Trả về giá trị 0 cho hàm main, thông báo không có lỗi xảy ra trong quá trình thực thi lệnh*/
    return 0;
}
```

Khi biên dịch và chạy chương trình, chúng ta sẽ có kết quả như sau:

```
./time_mktime
Enter year: 2011
Enter month: 11
Enter day: 17
That day is a Thursday.
```

#### **5.4.5. localtime:**

##### **a. Cú pháp:**

```
#include <time.h>
struct tm * localtime ( const time_t * timer );
```

##### **b. Giải thích:**

Hàm localtime dùng chuyển đổi thông tin thời gian dạng `time_t` thành thông tin thời gian dạng `struct tm`.

Hàm localtime có tham số là con trỏ đến biến thời gian `const time_t timer` cần chuyển đổi.

Giá trị trả về của hàm là con trỏ cấu trúc `struct tm *`.

##### **c. Chương trình ví dụ:**

Sau đây là chương trình ví dụ sử dụng hàm `localtime`. Chương trình này sẽ cập nhật thời gian hiện tại dạng `time_t`, chuyển sang cấu trúc `struct tm`, cuối cùng in ra màn hình dưới dạng chuỗi thời gian bằng hàm `asctime`.

```
/*Chương trình ví dụ hàm localtime*/
/*Khai báo các thư viện cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <time.h>
/*chương trình chính*/
int main () {
    /*Khai báo biến thời gian dạng time_t để lưu thời gian hiện tại*/
    time_t rawtime;
    /*Khai báo con trỏ cấu trúc thời gian struct tm lưu thông tin thời gian sau khi chuyển đổi*/
    struct tm * timeinfo;
    /*Cập nhật thông tin thời gian hiện tại*/
    time ( &rawtime);
    timeinfo = localtime ( &rawtime );
```

```
/*Chuyển đổi cấu trúc thời gian thành chuỗi sau đó in ra màn hình*/
printf ("Current local time and date: %s", asctime ( timeinfo ));
/*Trả về giá trị 0, thông báo quá trình thực thi lệnh không bị lỗi*/
return 0;
}
```

Sau khi biên dịch và thực thi chương trình, chúng ta có kết quả sau:

```
./time_localtime_linux
Current local time and date: Thu Nov 17 11:54:42 2011
```

#### **5.4.6. *strftime:***

##### **a. *Cú pháp:***

```
#include <time.h>

size_t strftime ( char * ptr, size_t maxsize, const char *
format, const struct tm * timeptr );
```

##### **b. *Giải thích:***

Hàm `strftime` làm nhiệm vụ chuyển thông tin thời gian dạng cấu trúc `struct tm` thành thông tin thời gian dạng chuỗi có định dạng do người lập trình định nghĩa.

Hàm `strftime` có 4 tham số: `char * ptr`, là con trỏ ký tự dùng để lưu chuỗi thông tin thời gian trả về của hàm sau khi đã định dạng; `size_t maxsize`, là kích thước tối đa của chuỗi thông tin thời gian trả về; `const char * format`, là định dạng chuỗi thông tin thời gian mong muốn (tương tự như phân định dạng thông tin xuất ra trong hàm `printf`); cuối cùng, `const struct tm * timeptr`, là con trỏ cấu trúc nguồn cần chuyển đổi.

Các định dạng được quy định trong bảng sau:

<b>Ký hiệu</b>	<b>Được thay thế bằng</b>	<b>Ví dụ</b>
%a	Tên viết tắt của ngày trong tuần;	Mon
%A	Tên đầy đủ của ngày trong tuần;	Monday
%b	Tên viết tắt của tháng;	Aug
%B	Tên đầy đủ của tháng;	August
%c	Chuỗi ngày giờ chuẩn	Thu Aug 07:55:02
%d	Ngày của tháng (01-31)	17
%H	Giờ trong ngày (00-23)	23
%I	Giờ trong ngày (01-12)	12



%j	Ngày trong năm (001-366)	245
%m	Tháng trong năm (01-12)	08
%M	Phút trong giờ (00-59)	45
%p	AM hoặc PM	AM
%S	Giây trong phút (00-61)	02
%U	Số tuần trong năm (00-53) với Chúa nhật là ngày đầu tiên trong tuần;	34
%w	Số ngày trong tuần (0-6), Chúa nhật: 0	4
%W	Số tuần trong năm (00-53) với thứ Hai là ngày đầu tiên trong tuần;	34
%x	Chuỗi ngày tháng năm (dd/mm/yy)	18/11/1 1
%X	Chuỗi giờ phút giây (HH:MM:SS)	07:32:3 4
%y	Hai số cuối của năm	11
%Y	Số Năm hoàn chỉnh	2011
%Z	Tên múi giờ	CDT
%%	Ký tự đặc biệt	%

Hàm `strftime` có giá trị trả về là một số kiểu `size_t`. Bằng 0 nếu quá trình chuyển đổi xảy ra lỗi (do kích thước quy định không đủ,...). Bằng một số dương kích thước chuỗi ký tự đã chép vào con trỏ `char * ptr` nếu quá trình chuyển đổi thành công;

Chương trình ví dụ sau sẽ giúp cho chúng ta hiểu rõ cách sử dụng hàm `strftime`.

**c. Chương trình ví dụ:**

Chương trình ví dụ này có nhiệm vụ in ra chuỗi thông tin về giờ phút giây hiện tại theo dạng 12H, có thêm AM và PM phía sau.

*/\*Chương trình ví dụ hàm strftime\*/*

*/\*Khai báo các thư viện cho các hàm sử dụng trong chương trình\*/*

```
#include <stdio.h>
```

```
#include <time.h>
```

*/\*Chương trình chính\*/*

```
int main () {
```

```
/*Khai báo biến kiểu time_t để lưu thời gian hiện tại của hệ thống*/
time_t rawtime;

/*Khai báo con trỏ cấu trúc thời gian làm tham số cho hàm strftime*/
struct tm * timeinfo;

/*Khai báo vùng nhớ có kích thước 80 ký tự lưu chuỗi thông tin thời gian*/
char buffer [80];

/*Đọc thời gian hiện tại của hệ thống*/
time (&rawtime);

/*Chuyển thông tin thời gian dạng time_t thành thông tin thời gian dạng cấu trúc struct tm, theo thời gian cục bộ*/
timeinfo = localtime ( &rawtime );

/*Thực hiện chuyển đổi cấu trúc thời gian struct tm thành chuỗi thời gian theo định dạng người dùng*/
strftime (buffer, 80, "Now it's %I:%M%p.", timeinfo);

/*Trả về giá trị 0 thông báo quá trình thực thi lệnh không xảy ra lỗi*/
return 0;
}
```

Biên dịch và chạy chương trình chúng ta có kết quả sau:

```
./time_strftime
Now it's 08:02AM.
```

Như vậy đến đây chúng ta có thể tạo ra một chuỗi thông tin về thời gian với định dạng linh động, phù hợp với mục đích người dùng, thay vì sử dụng định dạng có sẵn của những hàm chuyển đổi thời gian khác.

### **III. Tổng kết:**

Trì hoãn thời gian là một trong những vấn đề quan trọng trong lập trình ứng dụng điều khiển các thiết bị ngoại vi. Trong chương này chúng ta đã tìm hiểu những lệnh căn bản nhất về trì hoãn thời gian. Tùy theo từng ứng dụng cụ thể, người lập trình sẽ chọn cho mình cách trì hoãn thời gian thích hợp sao cho không làm ảnh hưởng nhiều đến tính thời gian thực của chương trình. Chúng ta cũng đã tìm hiểu lệnh alarm tạo lập định thời cho tín hiệu SIGALRM, những lệnh thao tác với thời gian thực trong hệ thống linux được định nghĩa trong thư viện time.h.

Trong bài sau, chúng ta sẽ bước vào những vấn đề lý thú của hệ điều hành Linux nói riêng và các hệ điều hành thời gian thực khác nói chung, đó là các hàm thao tác với tiến trình và tuyến phục vụ cho việc thiết kế các hệ thống thời gian thực một cách dễ dàng hơn.

**BÀI 3****LẬP TRÌNH ĐA TIẾN TRÌNH  
TRONG USER APPLICATION****I. Kiến thức ban đầu:****a. Định nghĩa tiến trình trong Linux:**

Trong hầu hết các hệ điều hành đa nhiệm tựa UNIX như Linux, thì khả năng xử lý đồng thời nhiều tiến trình là một trong những đặc điểm quan trọng. Như vậy tiến trình được định nghĩa như thế nào trong hệ thống Linux.

Giả sử khi chúng ta thực hiện câu lệnh `ls -l` thì Linux sẽ tiến hành liệt kê tất cả những thông tin về tập tin và thư mục có trong thư mục hiện hành. Như vậy hệ điều hành Linux đã thực hiện một tiến trình. Cùng một lúc hai người sử dụng lệnh `ls -l` (chẳng hạn một người đăng nhập trực tiếp và một người đăng nhập qua mạng) hệ điều hành cũng sẽ đáp tất cả những yêu cầu trên. Như vậy ta nói rằng hệ điều hành đã thực hiện song song hai tiến trình. Thậm chí ngay cả một chương trình, cũng có thể có nhiều tiến trình cùng một lúc xảy ra. Ví dụ như chương trình ghi dữ liệu vào đĩa CD, có rất nhiều tiến trình xảy ra cùng một lúc, một tiến trình làm nhiệm vụ chép nội dung dữ liệu từ đĩa cứng sang đĩa CD và một tiến trình tính toán thời gian còn lại để hoàn thành công việc hiển thị trên thanh trạng thái ... Tiến trình và chương trình là hai định nghĩa hoàn toàn khác nhau, nhiều khi cũng có thể coi là một. Khi chương trình có duy nhất một tiến trình xử lý thì lúc đó tiến trình và chương trình là một. Nhưng khi có nhiều tiến trình cùng thực hiện để hoàn thành một chương trình thì lúc đó tiến trình chỉ là một phần của chương trình.

Mỗi một tiến trình trong Linux đều được gán cho một con số đặc trưng duy nhất. Số này gọi là định danh của tiến trình, PID (Process Identification Number). PID dùng để phân biệt giữa các tiến trình cùng tồn tại song song trong hệ thống. Mỗi một tiến trình được cấp phát một vùng nhớ lưu trữ những thông tin cần thiết có liên quan đến tiến trình. Các tiến trình được điều phối xoay vòng thực hiện bởi hệ thống phân phối thời gian của hệ điều hành. Các tiến trình được trao đổi thông tin với nhau thông qua cơ chế giao tiếp giữa các tiến trình (IPC-Inter Process Communication). Các cơ chế giao tiếp giữa các tiến trình bao gồm những kỹ thuật như đường ống, socket, message, chuyển hướng xuất nhập,

hay phát sinh tín hiệu ... (Chúng ta sẽ đi sâu tìm hiểu những kỹ thuật này trong những bài sau).

*Như vậy, tiến trình được hiểu là một thực thể điều khiển đoạn mã lệnh có riêng một không gian địa chỉ vùng nhớ, có ngăn xếp riêng lẻ, có bảng chữ các số mô tả tập tin được mở cùng tiến trình và đặc biệt là có một định danh PID (Process Identify) duy nhất trong toàn bộ hệ thống vào thời điểm tiến trình đang chạy. (Theo định nghĩa trong sách lập trình Linux của nhà xuất bản Minh Khai)*

### **b. Cấu trúc tiến trình:**

Phần trên chúng ta đã khái quát được những kiến thức căn bản nhất trong một tiến trình. Trong phần này chúng ta sẽ đi sâu vào tìm hiểu cấu trúc của tiến trình, cách xem thông tin của các tiến trình đang chạy trong hệ thống Linux và ý nghĩa của những thông tin đó.

Như định nghĩa trong phần trên mỗi tiến trình được cấp phát một vùng nhớ riêng để lưu những thông số cần thiết. Những thông tin đó là gì?

PID
Code
Data
Library
FileDes

Thông tin đầu tiên là PID (Process Identify). Thông tin này là số định danh cho mỗi tiến trình. Mỗi tiến trình đang chạy trong hệ thống đều có một định danh riêng biệt không trùng lặp với bất kỳ tiến trình nào. Số PID được hệ điều hành cung cấp một cách tự động. PID là một số nguyên dương từ 2 đến 32768. Riêng tiến trình `init` là tiến trình quản lý và tạo ra mọi tiến trình con khác được hệ điều hành gán cho PID là 1. Đó là nguyên nhân tại sao PID của các tiến trình là một số nguyên dương bắt đầu từ 2 đến 32768.

Thông tin thứ hai là Code. Code là vùng nhớ chứa tập tin chương trình thực thi chứa trong đĩa cứng được hệ điều hành nạp vào vùng nhớ (cụ thể ở đây là RAM). Ví dụ khi chúng ta đánh lệnh `ls -l` thì hệ điều hành sẽ chép mã lệnh thực thi của lệnh `ls` trong thư mục bin của `Root File System` vào RAM nội nói đúng hơn là chép vào vùng nhớ

Code của tiến trình. Từ đó hệ điều hành sẽ chia khe thời gian thực thi đoạn chương trình trình này.

Thông tin thứ ba là Library. Library là thư viện chứa những đoạn mã dùng chung cho các tiến trình hay chỉ dùng riêng cho một tiến trình nào đó. Các thư viện dùng chung gọi là các thư viện chuẩn tương tự như thư viện DLLs trong hệ điều hành Windows. Những thư viện dùng riêng có thể là những thư viện do người lập trình định nghĩa trong quá trình xây dựng chương trình (bao gồm thư viện liên kết động và thư viện liên kết chuẩn).

Thông tin thứ tư là FileDes. FileDes là viết tắt của từ tiếng Anh File Description tạm dịch là bảng mô tả tập tin. Như tên gọi của nó, bảng thông tin mô tả tập tin chứa các số mô tả tập tin được mở trong quá trình thực thi chương trình. Mỗi một tập tin trong hệ thống Linux khi được mở điều chứa một số mô tả tập tin duy nhất, bao gồm những thông tin như đường dẫn, cách thức truy cập tập tin (chỉ đọc, chỉ ghi, hay cả hai thao tác đọc và ghi, ...).

Để xem những thông tin trên, chúng ta thực hiện lệnh shell sau:

```
ps -af
```

Sau khi thực hiện xong chúng ta sẽ có kết quả như sau:

```
root@:/home/arm/project/application# ps -af
UID          PID  PPID  C STIME TTY          TIME CMD
root         2731   2600  0 21:55 pts/2    00:00:00 ps -af
```

Cột thứ nhất UID là tên của người dùng đã gọi tiến trình. PID là số định danh mà hệ thống cung cấp cho tiến trình. PPID là số định danh của tiến trình cha đã gọi tiến trình PID. C là CPU thực hiện tiến trình. STIME là thời điểm tiến trình được đưa vào sử dụng. TIME là thời gian chiếm dụng CPU của tiến trình. TTY là terminal ảo nơi gọi thực thi tiến trình. CMD là chi tiết dòng lệnh được triệu gọi thực thi.

Ngoài ra chúng ta cũng có thể dùng lệnh `ps -ax` để xem các tiến trình đang thực thi của toàn bộ hệ thống.

## II. Nội dung:

Phần này chúng ta sẽ tìm hiểu những hàm thao tác với tiến trình. Những hàm này sẽ giúp rất nhiều trong việc thiết kế một hệ thống hoạt động đa tiến trình sao cho hiệu quả nhất. Mỗi hàm sẽ bao gồm 3 phần thông tin. Thứ nhất là cú pháp sử dụng hàm, thứ hai là ý nghĩa các tham số trong hàm, thứ ba là chương trình ví dụ cách sử dụng hàm. Để có thể ứng dụng ngay những kiến thức đã học vào thực tế, chúng ta phải tiến hành đọc hiểu mã lệnh, sau đó lập trình trên máy tính, biên dịch và chạy chương trình, quan sát so sánh với dự đoán ban đầu. Những hàm liệt kê sẽ theo thứ tự từ dễ đến khó, mang tính chất kế thừa thông tin của nhau. Vì thế nên chúng ta phải nghiên cứu theo trình tự của bài học nếu các bạn là người mới bắt đầu.

### 1. Hàm `system()`:

#### a. Cú pháp:

```
#include <stdlib.h>

int system ( const char *cmdstr);
```

#### b. Giải thích:

Hàm `system` sẽ thực thi chuỗi lệnh `cmdstr` do người lập trình nhập vào. Chuỗi lệnh nhập vào `cmdstr` là lệnh trong môi trường shell, chẳng hạn `ls -l` hay `cd <directory> ...`. Hàm `system` khi được gọi trong một tiến trình, nó sẽ tiến hành tạo và chạy một tiến trình khác (là tiến trình chứa trong tham số) khi thực hiện xong tiến trình này thì hệ thống mới tiếp phân công thực hiện những lệnh tiếp theo sau hàm `system`. Nếu tiến trình trong hàm `system` được gán cho hoạt động hậu tiến trình thì những lệnh theo sau hàm `system` vẫn thực hiện song song với tiến trình đó. Chúng ta sẽ làm rõ hơn trong phần chương trình ví dụ.

Hàm `system` có tham số `const char *cmdstr` là chuỗi lệnh người lập trình muốn thực thi.

Hàm `system` có giá trị trả về kiểu `int`. Trả về giá trị lỗi 127 nếu như không khởi động được shell, mã lỗi -1 nếu như gặp các lỗi khác. Những trường hợp còn lại là mã lỗi do lệnh trong tham số `cmdstr` trả về.

#### c. Chương trình ví dụ:

Viết chương trình in ra 10 lần chuỗi `helloworld` như sau:

```
/*chương trình helloworld system_helloworld.c*/

#include <stdio.h>
```

```
int main (void) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("Hello world %d!\n", i);
        sleep(1);
    }
}
```

Biên dịch và lưu tập tin kết xuất với tên `system_helloworld`;

```
arm-none-linux-gnueabi-gcc system_helloworld.c -o system_helloworld
```

Viết chương trình khai báo sử dụng hàm `system` gọi thực thi tiến trình `system_helloworld` đã biên dịch:

```
/*Chương trình ứng dụng hàm system trường hợp 1 system_main_1.c*/
#include <stdio.h>

/*Khai báo thư viện sử dụng hàm system*/
#include <stdlib.h>

int main (void) {
    /*In thông báo hàm system bắt đầu thực thi*/
    printf ( "System has called a process... \n");

    /*Thực thi hàm system gọi tiến trình system_helloworld đã được biên dịch trước đó*/
    system ( "../system_helloworld" );

    /*Thông báo tiến trình trong hàm system thực thi xong*/
    printf ( "Done.\n");
    return 0;
}
```

Biên dịch và lưu tập tin kết xuất với tên `system_main`;

```
arm-none-linux-gnueabi-gcc system_main_1.c -o system_main_1
```

Chạy chương trình `system_main_1` vừa biên dịch trên kit, ta có kết quả sau:

```
./system_main_1
System has called a process...
Hello world 0!
Hello world 1!
Hello world 2!
Hello world 3!
Hello world 4!
```



```
Hello world 5!
Hello world 6!
Hello world 7!
Hello world 8!
Hello world 9!
Done.
```

Giải thích chương trình:

Ở chương trình `system_main_1.c`, hàm `system()` gọi tiến trình `./system_helloworld` chương trình này có nhiệm vụ in ra 10 lần chuỗi `"Hello world"`. Chương trình chứa hàm `system` đợi cho đến khi thực hiện xong việc in 10 lần chuỗi `"Hello world"` sau đó mới thực hiện tiếp lệnh phía sau hàm `system`. Như chúng ta thấy, chuỗi `"Done"` được in phía sau cùng tất cả các dòng thông tin.

Trong trường hợp thứ hai, chúng ta sử dụng hàm `system` để gọi một tiến trình chạy hậu cảnh. Nghĩa là lúc này, những câu lệnh phía sau hàm `system` sẽ thực thi song song với tiến trình được gọi bởi hàm `system`.

Viết chương trình ví dụ sau:

```
/*Chương trình ứng dụng hàm system trường hợp 2 system_main_2.c*/
#include <stdio.h>

/*Khai báo thư viện sử dụng hàm system*/
#include <stdlib.h>

int main (void) {
    /*In thông báo hàm system bắt đầu thực thi*/
    printf ( "System has called a process... \n");

    /*Thực thi hàm system gọi tiến trình system_helloworld đã được biên dịch trước đó*/
    system ( "./system_helloworld &" );

    /*Thông báo tiến trình trong hàm system thực thi xong*/
    printf ( "Done.\n");
    return 0;
}
```

Tiến hành biên dịch chương trình:

```
arm-none-linux-gnueabi-gcc system_main_2.c -o system_main_2
```

Chạy chương trình `system_main_2` vừa biên dịch trên kit, ta có kết quả sau:

```
./system_main_2
System has called a process...
Done.
# Hello world 0!
Hello world 1!
Hello world 2!
Hello world 3!
Hello world 4!
Hello world 5!
Hello world 6!
Hello world 7!
Hello world 8!
Hello world 9!
```

Cũng tương tự trong chương trình `system_main_1.c` nhưng trong hàm `system` ta gọi tiến trình chạy hậu cảnh `system ( "./system_helloworld &" );` kết quả là chuỗi thông tin “Done” được in ra ngay phía sau dòng thông báo “System has called a process...” mà không phải nằm tại vị trí cuối như trường hợp 1. Hơn nữa cũng trong trường hợp này, chúng ta thấy lệnh trong hàm `system` thực thi chậm hơn lệnh phía sau hàm `system`. Đây cũng là một yếu điểm lớn nhất khi triệu gọi tiến trình bằng hàm `system` đó là thời gian thực thi chương trình chậm vì `system` gọi lại shell của hệ thống và chuyển giao trách nhiệm thực thi chương trình cho lệnh shell.

Trong các phần sau, chúng ta sẽ học cách triệu gọi một tiến trình thực thi sao cho ít tốn thời gian hơn.

## **2. Các dạng hàm exec:**

Các hàm tạo tiến trình dạng như `system()` thông thường tiêu tốn thời gian hoạt động của hệ thống nhất. Mỗi một tiến trình khi khởi tạo và thực thi, hệ điều hành sẽ cấp phát riêng một không gian vùng nhớ cho tiến trình đó sử dụng. Khi tiến trình A đang hoạt động, gọi hàm `system()` để tạo một tiến trình B. Thì hệ điều hành sẽ cấp phát một vùng nhớ cho tiến trình B hoạt động, điều này làm thời gian thực hiện của tiến trình B trở nên chậm hơn. Để giải quyết trường hợp này, hệ điều hành Linux cung cấp cho chúng ta một số hàm dạng `exec` có thể thực hiện tạo lập một tiến trình mới nhanh chóng bằng cách tạo

một không gian tiến trình mới từ không gian của tiến trình hiện có. Nghĩa là tiến trình hiện có sẽ nhường không gian của mình cho tiến trình mới tạo ra, tiến trình cũ sẽ không hoạt động nữa. Sau đây chúng ta sẽ nghiên cứu hàm `execlp()` phục vụ cho việc thay thế tiến trình.

### a. Cú pháp:

```
#include <stdio.h>

int execlp ( const char *file, const char *arg, ..., NULL);
```

### b. Giải thích:

Hàm `execlp ()` có nhiệm vụ thay thế không gian của tiến trình cũ (tiến trình triệu gọi hàm) bằng một tiến trình mới. Tiến trình mới tạo ra có ID là ID của tiến trình cũ nhưng mã lệnh thực thi, cấu trúc tập tin, thư viện là của tiến trình mới. Tiến trình cũ sẽ bị mất không gian thực thi, tất nhiên sẽ không còn thực thi trong hệ thống nữa.

Hàm `execlp` có các tham số: `const char *file` là đường dẫn đến tên tập tin chương trình thực thi trên đĩa. Nếu tham số này để trống, hệ điều hành sẽ tìm kiếm trong những đường dẫn khác chứa trong biến môi trường `PATH`; các tham số khác dạng `const char *arg` là những tên lệnh, tham số lệnh thực thi trong môi trường `shell`. Chẳng hạn, nếu chúng ta muốn thực thi lệnh `ps -ax` trong môi trường `shell` thì tham số thứ hai của lệnh là `"ps"`, tham số thứ ba là `"-ax"`. Tham số kết thúc là `NULL`.

Hàm này có giá trị trả về là một số `int`. Hàm trả về mã lỗi 127 khi không triệu gọi được `shell` thực thi, -1 khi không thực thi được chương trình, các mã lỗi còn lại do chương trình được gọi thực thi trả về.

### c. Chương trình ví dụ:

Thiết kế chương trình thực hiện công việc sau: in ra chuỗi `"Hello world"` 10 lần. Mỗi lần in ra cách nhau 1 giây.

Chương trình mẫu mang tên là: `execlp_helloworld.c`

*/\*Chương trình execlp\_helloworld.c \*/*

```
#include <stdio.h>

int main (void) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("Hello world %d!\n", i);
        sleep(1);
    }
}
```

```
}  
return 0;  
}
```

Thực hiện biên dịch chương trình với tập tin ngõ ra là `execlp_helloworld`.

```
arm-none-linux-gnueabi-gcc execlp_helloworld.c -o execlp_helloworld
```

Thiết kế chương trình sử dụng hàm `execlp` để gọi chương trình `execlp_helloworld` vừa biên dịch mang tên `execlp_main.c`.

```
/*Chương trình execlp_main.c*/  
/*Khai báo thư viện cho các lệnh sử dụng trong chương trình*/  
#include <stdio.h>  
#include <stdlib.h>  
/*Chương trình chính*/  
int main() {  
/*Thông báo hàm execlp bắt đầu thực thi*/  
    printf ("Execlp has just start ...");  
/*Tiến hành gọi chương trình execlp_helloworld từ thư mục hiện tại*/  
    execlp ("execlp_helloworld", "execlp_helloworld", NULL);  
/*In ra thông báo kết thúc chương trình execlp_main, chúng ta sẽ không thấy lệnh này thực thi*/  
    printf ("Done.");  
/*Trả về giá trị 0 cho hàm main, thông báo quá trình thực thi chương trình không có lỗi*/  
    return 0;  
}
```

Biên dịch chương trình `execlp_main.c` với tập tin ngõ ra là `execlp_main`, lưu vào cùng thư mục với chương trình `execlp_helloworld`:

```
arm-none-linux-gnueabi-gcc execlp_main.c -o execlp_main
```

Tiến hành chạy chương trình chúng ta có kết quả sau:

```
./execlp_main  
Execlp has just start ...
```

**\*\*Chúng ta thấy chương trình `execlp_helloworld` không thể thực thi được. Nguyên nhân vì khi gọi chương trình `"execlp_helloworld"` hệ điều hành sẽ tiến hành kiểm tra trong biến môi trường `PATH` đường dẫn chứa chương trình này. Mà chương trình**

"execvp\_helloworld" không chứa trong bất kỳ đường dẫn nào trong biến môi trường mà chỉ chứa trong thư mục hiện tại. Để chương trình gọi được chương trình "execvp\_helloworld" chúng ta phải thêm đường dẫn thư mục hiện hành vào biến môi trường bằng lệnh sau: `export PATH=$PATH:. trong shell.`

Khi đó tiến hành chạy lại chương trình `execvp_main` thì chương trình `execvp_helloworld` sẽ được gọi thực thi. Sau đây là kết quả:

```
./execvp_main
Execvp has just start ...
Hello world 0!
Hello world 1!
Hello world 2!
Hello world 3!
Hello world 4!
Hello world 5!
Hello world 6!
Hello world 7!
Hello world 8!
Hello world 9!
```

Do câu lệnh `execvp ("execvp_helloworld", "execvp_helloworld", NULL);` tiến hành thay thế không gian thực thi của tiến trình hiện tại bằng tiến trình mới `execvp_helloworld`. Vì thế, tất cả những lệnh phía sau hàm `execvp` đều không thực thi. Chúng ta thấy chuỗi "Done." không thể nào in ra màn hình được.

Đôi khi thay thế tiến trình đang chạy bằng một tiến trình khác lại không tốt. Giả sử chúng ta muốn quay lại chương trình chính khi chương trình mới tạo thành thực thi xong thì sao. Hệ điều hành Linux cung cấp cho chúng ta những lệnh có khả năng làm được những công việc trên.

### 3. Hàm `fork()`:

#### a. Cú pháp:

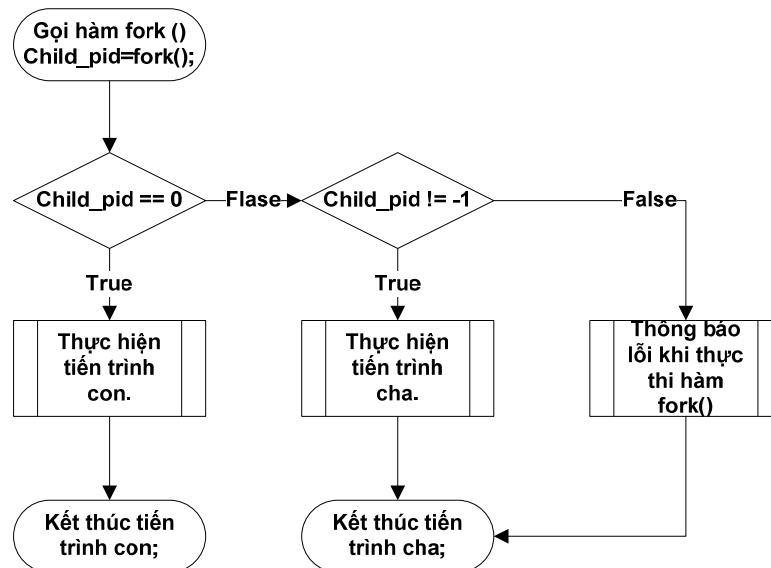
```
#include <sys/types.h>
#include <unistd.h>

pid_t fork();
```

#### b. Giải thích:

Hàm `fork()` cũng nằm trong nhóm hàm tạo tiến trình mới trong tiến trình đang thực thi. Hàm `fork()` làm nhiệm vụ tạo một tiến trình mới là con của tiến trình đang thực thi. Nghĩa là toàn bộ không gian của tiến trình cha sẽ được sao chép qua không gian của tiến trình con, chỉ khác nhau ở mã lệnh thực thi và các giá trị ID của chúng. Tiến trình con có hai thông số ID, PPID là ID của tiến trình cha, và PID là ID của tiến trình con này.

Hàm `fork()` sẽ quay trở về thời điểm gọi hai lần. Lần thứ nhất, hàm `fork()` trả về giá trị 0 để báo hiệu những lệnh tiếp theo sau là thuộc về tiến trình con. Lần thứ hai, hàm `fork()` trả về một giá trị khác 0. Giá trị này chính là ID của tiến trình con mới vừa tạo thành. Hàm `fork()` trả về giá trị -1 khi quá trình tạo lập tiến trình con xảy ra lỗi. Hàm `fork()` không có tham số. Chúng ta có thể dùng đặc điểm trên để tiến hành tạo lập một tiến trình mới ngay trong một tiến trình hiện tại theo lưu đồ sau:



Với lưu đồ trên chúng ta có thể thiết kế một đoạn chương trình chuẩn thao tác và sử dụng hàm `fork()` như sau:

```
pid_t    child_pid;
child_pid = fork();
switch (child_pid) {
    case -1:    printf ("Cannot create child process.");
                /*Thông báo lỗi cho người dùng khi không tạo lập được tiến trình
                con*/
                break;

    case 0:    printf ("This is the child process.");
                /*Những lệnh thuộc về tiến trình con*/
```

```
        break;
default:  printf ("This is the parent process.");
        /*Những lệnh thuộc về tiến trình cha*/
        break;
}
```

Sau đây là chương trình ví dụ minh họa cách sử dụng hàm `fork()` để tạo lập tiến trình.

### **c. Chương trình ví dụ:**

Chương trình ví dụ sau đây sẽ làm nhiệm vụ tạo lập một tiến trình con bên trong tiến trình cha đang thực thi. Tiến trình cha sẽ in ra chuỗi "Parent process: Hello" 5 lần. Trong khi đó tiến trình con cũng in ra chuỗi thông tin "Child process: Hello" nhưng với 10 lần. Mỗi lần cách nhau 1 giây.

Thực hiện thiết kế chương trình với những yêu cầu trên:

```
/*Chương trình ví dụ hàm fork() fork_main.c*/
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h> //Thư viện sử dụng cho hàm printf();
#include <unistd.h> //Thư viện sử dụng cho hàm fork();
#include <sys/types.h> //Thư viện sử dụng cho kiểu pid_t;
/*Chương trình chính*/
int main(void) {
    /*Khai báo biến lưu trữ id cho tiến trình con*/
    pid_t  child_pid;
    /*Khai báo các biến sử dụng đếm số lần in chuỗi ký tự*/
    int n, m;
    /*Thực hiện chia tiến trình bằng hàm fork()*/
    /*Chương trình sẽ quy lại đây hai lần*/
    child_pid = fork();
    /*Tách tiến trình bằng cách kiểm tra child_pid*/
    switch (child_pid) {
        /*Khi chương trình không thể phân chia tiến trình con*/
        case -1: printf ("Cannot create process.");
                return -1;
    }
```

```
    /*Những lệnh trong trường hợp này thuộc về tiến trình con*/
    case 0: printf ("This is the child");
            for (n = 0; n < 10; n++) {
                printf ("Child process: Hello %d\n", n);
                sleep (1);
            }
            return 0;

    /*Những lệnh trong trường hợp này thuộc về tiến trình cha*/
    default:  printf ("This is the parent");
              for (m=0; m < 5; m++) {
                printf ("Parent process: Hello %d\n", m);
                sleep(1);
              }
              break;
    }
    return 0;
}
```

Thực hiện biên dịch chương trình với tập tin chương trình kết xuất là `fork_main`.

```
arm-none-linux-gnueabi-gcc fork_main.c -o fork_main
```

Chép tập tin chương trình `fork_main` vào kit và tiến hành chạy chương trình. Chúng ta có kết quả sau:

```
./fork_main
This is the parentParent process: Hello 0
This is the childChild process: Hello 0
Parent process: Hello 1
Child process: Hello 1
Parent process: Hello 2
Child process: Hello 2
Parent process: Hello 3
Child process: Hello 3
Parent process: Hello 4
Child process: Hello 4
# Child process: Hello 5
Child process: Hello 6
Child process: Hello 7
Child process: Hello 8
```



```
Child process: Hello 9
```

Với kết quả trên, chúng ta thấy tiến trình cha và tiến trình con chạy đồng thời với nhau và trình tự xuất ra chuỗi thông tin. Hai tiến trình này được hệ điều hành chia thời gian thực hiện đồng thời, độc lập với nhau. Do đó khi tiến trình cha kết thúc, tiến trình con vẫn tiếp tục thực hiện. Chúng ta có thể kiểm tra sự thực thi của hai tiến trình trên bằng lệnh shell sau.

```
./fork_main & ps -af
```

```
[1] 3264
```

```
This is the parentParent process: Hello 0
```

```
This is the childChild process: Hello 0
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	3264	3173	0	19:28	pts/1	00:00:00	./fork_main
root	3265	3173	8	19:28	pts/1	00:00:00	ps -af
root	3266	3264	0	19:28	pts/1	00:00:00	./fork_main

Chúng ta thấy, tiến trình `fork_main` được gọi hai lần trong hệ thống. Lần thứ nhất là tiến trình cha, số ID là 3264. Lần thứ 2 là tiến trình con với số PPID là 3264 cùng với số PID của tiến trình cha, và số PID là 3266.

Đôi khi chúng ta muốn tiến trình cha trong lúc thực thi khi cần thực hiện một nhiệm vụ cụ thể sẽ gọi một tiến trình con khác thực hiện thay và chờ tiến trình con kết thúc tiến trình cha mới làm việc tiếp. Trong trường hợp trên thì yêu cầu này không thực hiện được. Tiến trình cha thậm chí kết thúc tiến trình con vẫn tiếp tục thực hiện. Hiện tượng này gọi là hiện tượng tiến trình con bị bỏ rơi. Chúng ta không muốn trường hợp này xảy ra. Hệ thống Linux cung cấp cho chúng ta hàm `wait` để thực hiện công việc này.

#### 4. Hàm `wait()`:

##### a. Cú pháp:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait ( int  &stat_loc );
```

##### b. Giải thích:

Như trên đã đề cập, tiến trình cha và tiến trình con đòi hỏi phải có sự đồng bộ nhau trong việc xử lý thông tin. Tiến trình cha phải đợi tiến trình con kết thúc mới tiến hành làm những công việc tiếp theo, sử dụng kết quả của tiến trình con. Hàm `wait()` được sử dụng với mục đích này.

Hàm `wait()` có tham số `int &stat_loc` là con trỏ để lưu trạng thái của tiến trình con khi thực hiện xong. Hàm `wait()` trả về là số ID của tiến trình con hoàn thành.

Chương trình sau sẽ cho chúng ta hiểu rõ cách sử dụng hàm `wait()`.

### **c. Chương trình ví dụ:**

Cũng giống như chương trình ví dụ trước, chương trình này cũng tạo ra hai tiến trình, tiến trình cha và tiến trình con. Nhưng lần này tiến trình cha sẽ đợi tiến trình con kết thúc sau đó mới thực hiện tiếp công việc của mình khi sử dụng hàm `wait()`.

```
/*Chương trình ví dụ hàm wait() wait_main.c*/
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình*/

#include <stdio.h> //Thư viện sử dụng cho hàm printf();
#include <unistd.h> //Thư viện sử dụng cho hàm fork();
#include <sys/types.h> //Thư viện sử dụng cho kiểu pid_t;
#include <sys/wait.h> //Thư viện sử dụng cho hàm wait;
/*Chương trình chính*/

int main(void) {
    /*Khai báo biến lưu trữ id cho tiến trình con*/
    pid_t child_pid;
    /*Khai báo các biến sử dụng đếm số lần in chuỗi ký tự*/
    int n, m;
    /*Khai báo biến lưu trạng thái tiến trình con khi thực hiện xong*/
    int child_status;
    /*Thực hiện chia tiến trình bằng hàm fork()*/
    /*Chương trình sẽ quy lại đây hai lần*/
    child_pid = fork();
    /*Tách tiến trình bằng cách kiểm tra child_pid*/
    switch (child_pid) {
        /*Khi chương trình không thể phân chia tiến trình con*/
        case -1: printf ("Cannot create process.");
                return -1;
        /*Những lệnh trong trường hợp này thuộc về tiến trình con*/
        case 0: printf ("This is the child");
                for (n = 0; n < 10; n++) {
```

```
        printf ("Child process: Hello %d\n", n);
        sleep (1);
    }
    return 0;

/*Những lệnh trong trường hợp này thuộc về tiến trình cha*/
default:  printf ("This is the parent");
/*Chờ tiến trình con kết thúc*/

        wait (&child_status);
        for (m=0; m < 5; m++) {
            printf ("Parent process: Hello %d\n", m);
            sleep(1);
        }
        break;
    }
return 0;
}
```

Biên dịch chương trình và lưu tên chương trình đã biên dịch thành `wait_main`  
`arm-none-linux-gnueabi-gcc wait_main.c -o wait_main`

Chạy chương trình chúng ta có kết quả sau:

```
./wait_main
This is the parent
This is the child
Child process: Hello 0
Child process: Hello 1
Child process: Hello 2
Child process: Hello 3
Child process: Hello 4
Child process: Hello 5
Child process: Hello 6
Child process: Hello 7
Child process: Hello 8
Child process: Hello 9
Parent process: Hello 0
Parent process: Hello 1
Parent process: Hello 2
Parent process: Hello 3
```

Parent process: Hello 4

Như vậy chúng ta thấy rõ ràng, tiến trình con thực hiện xong ghi 10 lần chuỗi "Hello world" thì tiến trình cha mới thực hiện tiếp công việc của mình ghi 5 lần chuỗi "Hello world" tiếp theo.

Trong hàm `wait (&child_status);` có xuất hiện biến con trả `&child_status`, biến này kết hợp với một số macro trong thư viện `<sys/wait.h>` sẽ cho chúng ta những thông tin thực hiện tiến trình con. Chi tiết về các macro này, các bạn hãy tham khảo trong thư viện `<sys/wait.h>` trong hệ thống linux.

### III. Kết luận:

Trong bài này chúng ta đã học những định nghĩa rất căn bản về tiến trình. Tiến trình là một trong những kỹ thuật lập trình hiệu quả trong lập trình ứng dụng chạy trên hệ điều hành nhúng. Để hiểu và thao tác thành thạo với tiến trình chúng ta cũng đã nghiên cứu những hàm như tạo lập tiến trình, thay thế tiến trình và nhân đôi tiến trình. Bên cạnh đó chúng ta cũng tham khảo cách sử dụng hàm `wait` trong việc đồng bộ hóa hoạt động của các tiến trình với nhau.

*(Thêm giới hạn trong bài này là không trình bày trao đổi thông tin giữa ác tiến trình, tranh chấp dữ liệu thực thi của các tiến trình với nhau).*

Tiến trình là một kỹ thuật hiệu quả nhưng chưa phải là hiệu quả nhất về thời gian thực thi. Vì hệ điều hành còn phải tốn thời gian khởi tạo không gian thực thi cho mỗi tiến trình mới tạo ra. Đối với những ứng dụng đòi hỏi thời gian thực thi nhanh thì tiến trình không thể giải quyết được. Linux cung cấp cho chúng ta một kỹ thuật lập trình mới hiệu quả hơn đó là kỹ thuật lập trình đa tuyến mà chúng ta sẽ nghiên cứu ngay trong bài sau đây.

**BÀI 4**

## **LẬP TRÌNH ĐA TUYẾN TRONG USER APPLICATION**

**I. Kiến thức ban đầu về tuyến trong Linux:**

Trong bài trước chúng ta đã học những khái niệm cơ bản nhất về tiến trình trong Linux. Trong hệ thống Linux, các tiến trình có một không gian vùng nhớ riêng hoạt động độc lập nhau, xử lý một nhiệm vụ cụ thể trong chương trình. Chúng ta có thể tạo lập một tiến trình mới từ tiến trình đang thực thi bằng các hàm thao tác với tiến trình. Tuy nhiên chi phí tạo lập một tiến trình mới rất lớn đặc biệt là thời gian thực thi. Vì hệ điều hành phải tạo lập một không gian vùng nhớ cho chương trình mới, hơn nữa việc giao tiếp giữa các tiến trình rất phức tạp (chúng ta sẽ nghiên cứu trong các bài sau). Hệ điều hành Linux cung cấp cho chúng ta một kỹ thuật lập trình mới ít tốn thời gian hơn và hiệu quả hơn trong lập trình ứng dụng. Đó là kỹ thuật lập trình đa tuyến.

Tuyến là một thuật ngữ được dịch từ tuyến. Một số sách khác còn gọi là tiểu trình hay luồng. Tuyến được hiểu như là bộ phận cấu thành một tiến trình. Có cùng không gian vùng nhớ với tiến trình, có khả năng sử dụng những biến cục bộ được định nghĩa trong tiến trình. Nếu như trong hệ điều hành có nhiều tiến trình cùng hoạt động song song thì trong một tiến trình cũng có nhiều tuyến hoạt động song song nhau chia sẻ không gian vùng nhớ của tiến trình đó. Tiến trình và tuyến nhiều khi cũng tương tự nhau. Nếu như trong tiến trình có nhiều tuyến cùng hoạt động song song thì tuyến là một bộ phận cấu thành tiến trình. Nếu như trong tiến trình chỉ có một tuyến thì tuyến và tiến trình là một. Như vậy đến đây chúng ta có thể hiểu hệ điều hành Linux vừa làm nhiệm vụ chia khe thời gian thực hiện tiến trình, vừa làm nhiệm vụ chia khe thực hiện tuyến trong tiến trình.

Trong bài này chúng ta sẽ nghiên cứu những hàm sử dụng trong lập trình đa tuyến:

- Các hàm tạo lập và hủy tuyến;
- Các hàm chờ đợi tuyến thực thi nhằm thực hiện đồng bộ trình tự thời gian thực thi lệnh giữa các tuyến;

**II. Nội dung:****1. Hàm `pthread_create()`:****a. Cú pháp:**

```
#include <pthread.h>
```

```
int pthread_create (    pthread_t * thread,  
                        pthread_attr_t * attr,  
                        void * (*start_routine) (*void),  
                        void *arg );
```

**b. Giải thích:**

Hàm `pthread_create ()` được dùng để tạo một tuyến mới từ tiến trình đang chạy. Tuyến mới được tạo thành có thuộc tính quy định bởi tham số `pthread_attr_t * attr`, nếu tham số này có giá trị `NULL` thì tuyến mới tạo ra sẽ có thuộc tính mặc định (Chúng ta không đi sâu vào các thuộc tính này do đó thông thường nên sử dụng `NULL`). Khi tuyến được tạo ra thành công, hệ điều hành sẽ tự động cung cấp một ID lưu trong tham số `pthread_t * thread` phục vụ cho mục đích quản lý.

Tuyến mới tạo ra có tập lệnh thực thi chứa trong chương trình con quy định trong tham số `void * (*start_routine) (*void)`. Chương trình con này có tham số chứa trong `void *arg`.

Để thoát khỏi tuyến đang thực thi một cách đột ngột và trả về mã lỗi, chúng ta dùng hàm `pthread_exit()`. Hàm này tương tự như `exit ()` trong tuyến chính `main()`. Nếu `pthread_exit()` hay `exit ()` đặt cuối tuyến thì nhiệm vụ của chúng là trả về giá trị thông báo cho người lập trình biết trạng thái thực thi lệnh của chương trình.

Đoạn mã lệnh sau sẽ cho chúng ta hiểu rõ cách sử dụng hàm `pthread_create ()` và xem cách hàm `pthread_create ()` hoạt động trong thực tế.

**c. Chương trình ví dụ:**

Trong chương trình ví dụ sau đây, chúng ta tạo ra một chương trình con. Chương trình con này in ra chuỗi thông tin được cung cấp từ tham số của nó. Chương trình con này được cả hai tuyến sử dụng để in ra thông tin của mình.

Chúng ta tiến hành lập trình theo yêu cầu trên. Sau đây là chương trình mẫu:

```
/*Chương trình ví dụ hàm pthread_create () pthread_create.c*/  
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình */  
#include <stdio.h> //Cho hàm printf ();  
#include <pthread.h> // Cho hàm pthread_create();  
/*Chương trình con in ra thông tin từ tiến trình triệu gọi*/
```

```
void * do_thread ( void * data) {  
    /*Khai báo bộ đếm in dữ liệu ra màn hình*/  
    int i;  
    /*Chép dữ liệu cung cấp bởi chương trình triệu gọi vào vùng nhớ trong hàm kiểu  
    int*/  
    int me = (int*)data;  
    /*In chuỗi thông tin ra màn hình 5 lần cho mỗi tiến trình triệu gọi*/  
    for ( i=0; i<5; i++) {  
        /*%d đầu tiên là số cho biết tiến trình nào đang triệu gọi, %d thứ hai là số  
        lần in thông tin ra màn hình*/  
        printf (" '%d' -Got '%d' \n", me, i);  
        /*Trì hoãn một khoảng thời gian là 1s*/  
        sleep (1);  
    }  
    /*Thoát khỏi tuyến được triệu gọi, không xuất ra dữ liệu thông báo*/  
    pthread_exit(NULL);  
}  
/*Tuyến chính triệu gọi chương trình con do_thread*/  
int main (void) {  
    /*Khai báo biến lưu mã lỗi trả về bởi hàm pthread_create()*/  
    int thr_id;  
    /*Khai báo biến lưu ID cho tuyến tạo ra*/  
    pthread_t p_thread;  
    /*Biến lưu định danh của tuyến */  
    int a = 1;  
    int b = 2;  
    /* Sử dụng hàm pthread_create() tạo một tuyến mới  
    Tuyến mới có ID lưu vào p_thread, mã lỗi trả về thr_id, hàm thực thi là do_thread,  
    tham số là a*/  
    thr_id = pthread_create ( &p_thread, NULL, do_thread, (void *)a);  
    /*Gọi hàm do_thread() trong tuyến chính, tham số là b*/  
    do_thread ((void*)b);  
    /*Trả về giá trị 0 thông báo quá trình thực thi lệnh không có lỗi*/
```

```
    return 0;
}
```

Tiến hành biên dịch chương trình lưu với tên là `pthread_create` bằng lệnh sau:

**\*\*Chú ý** đây là chương trình có sử dụng đa tuyến nên chúng ta phải biên dịch có sự hỗ trợ của thư viện đa tuyến chứ không biên dịch chương trình giống như cách thông thường.

```
arm-none-linux-gnueabi-gcc pthread_create.c -o pthread_create -  
lpthread
```

Chạy chương trình trên hệ thống chúng ta có kết quả sau:

```
./pthread_create  
'2' -Got '0'  
'1' -Got '0'  
'2' -Got '1'  
'1' -Got '1'  
'2' -Got '2'  
'1' -Got '2'  
'2' -Got '3'  
'1' -Got '3'  
'2' -Got '4'  
'1' -Got '4'
```

Hai tuyến, một xuất phát từ tuyến chính `main()`, và một xuất phát từ hàm `pthread_create`, chạy song song nhau, cùng in ra màn hình chuỗi thông tin. Trong mỗi lần in tuyến ‘2’ luôn thực hiện trước tuyến ‘1’, nguyên nhân là vì tuyến ‘2’ được gọi thực thi trước tiên trong hệ thống. Chương trình con `do_thread` mặc dù chỉ có một nhưng có khả năng chạy đồng thời trên nhiều tiến trình. Thực ra mỗi thời điểm CPU chỉ thực thi một tuyến theo sự phân chia thời gian của hệ điều hành do đó không xảy ra sự xung đột trong việc sử dụng hàm.

## **2. Hàm `pthread_join()`:**

Trong thực tế, chúng ta cần thực hiện đồng bộ hóa hoạt động của giữa các tuyến với nhau. Giả sử tuyến chính triệu gọi một tuyến phụ thực hiện một công việc nào đó, tuyến chính chờ tuyến phụ thực hiện xong, trả về giá trị nào đó thì tuyến chính mới thực hiện những công việc tiếp theo. Linux cũng cung cấp cho chúng ta hàm `pthread_join()` để thực hiện những công việc trên.



**a. Cú pháp:**

```
#include <pthread.h>

int pthread_join ( pthread_t th, void *thread_return);
```

**b. Giải thích:**

Hàm `pthread_join ()` cung cấp cho chúng ta một công cụ đồng bộ hóa trình tự hoạt động của các tuyến.

Hàm `pthread_join ()` có hai tham số. Tham số thứ nhất `pthread_t th` là ID của tuyến muốn đợi. Tham số thứ hai `void *thread_return` là giá trị tuyến trả về từ hàm `pthread_exit(data)` khi được sử dụng cuối chương trình.

Hàm `pthread_join ()` có giá trị trả về là `int`, trả về 0 nếu như hàm thực thi thành công, trả về giá trị khác 0 là mã lỗi thực thi hàm.

**c. Chương trình ví dụ:**

Chương trình ví dụ sau cho chúng ta hiểu được cách sử dụng hàm `pthread_join()` trong chương trình. Chương trình `thread_join.c` có hai tuyến tạo thành. Tuyến chính tạo ra tuyến phụ gọi chương trình con in ra chuỗi thông tin cần thiết và chờ cho tuyến phụ kết thúc, tuyến chính tiếp tục thực hiện in thông báo tuyến phụ đã kết thúc và thoát chương trình thực thi.

Mã chương trình mẫu:

```
/*Chương trình ví dụ cách sử dụng hàm pthread_join, chương trình thread_join.c */
/*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình*/
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

/*Khai báo biến toàn cục sử dụng làm dữ liệu chung cho cả hai tuyến*/
/*Các tuyến nằm trong cùng một tiến trình có thể sử dụng chung biến này*/
char message[] = "Hello world!";

/*Chương trình con dùng cho tạo lập tuyến*/
void * do_thread (void *data) {
    /*Biến dùng cho việc đếm thông tin xuất ra màn hình*/
    int i;

    /*Nạp thông tin trong tham số hàm cho bộ nhớ trong tuyến*/
```

```
int me = (int *)data;
/*In ra thông báo tuyến đang thực thi*/
printf ("Thread function is executing ...\n");
/*In ra giá trị của biến message trước khi bị thay đổi*/
printf ("Thread data is %s\n", message);
/*Trì hoãn in các thông tin thí nghiệm cho người dùng quan sát*/
for ( i=0; i<5; i++) {
    printf (" '%d' got '%d'\n", me, i);
    sleep (1);
}
/*Thay đổi nội dung của biến toàn cục message*/
strcpy(message, "Goodbye!");
/*Xuất ra thông tin trả về cho tham số của hàm pthread_join */
pthread_exit("Thanks for your using me!");
}
/*Tuyến chính*/
int main(void) {
/*Khai báo biến lưu ID của tuyến tạo ra*/
pthread_t a_thread;
/*Biến int lưu mã lỗi của chương trình*/
int res;
/*Định danh cho tuyến*/
int a = 1;
/*Khai báo con trỏ trả dữ liệu về khi đợi tuyến hoàn thành xong nhiệm vụ*/
void * thread_result;
/*Thực hiện tạo tuyến mới từ tuyến chính với những thuộc tính mặc định,
Hàm thực thi là do_thread, tham số cho hàm là biến int a*/
res = pthread_create (&a_thread, NULL, do_thread, (int *)a);
/*Thông báo khi có lỗi xuất hiện*/
if (res != 0) {
    perror ("Thread created error.");
    exit (EXIT_FAILURE);
}
/*In thông báo nạp một tuyến vào danh sách đợi*/
```

```
printf ("Wait for thread to finish ...\n");  
/*Tiến hành nạp tuyến vào danh sách đợi*/  
res = pthread_join(a_thread, &thread_result);  
/*Thông báo mã lỗi trong quá trình thực thi hàm pthread_join*/  
if (res != 0) {  
    perror ("Thread wait error.");  
    exit (EXIT_FAILURE);  
}  
/*Thông báo tuyến phụ đã hoàn thành nhiệm vụ, xuất kết quả trả về*/  
printf ("Thread completed, it returned %s\n", (char  
)thread_result);  
/*In ra biến message sau khi đã bị thay đổi bởi tuyến phụ*/  
printf ("Message is now: %s\n", message);  
/*Trả về giá trị 0 cho tuyến main() thông báo quá trình thực thi không có lỗi*/  
return 0;  
}
```

Biên dịch chương trình lưu với tên thread\_join bằng dòng lệnh sau:

```
arm-none-linux-gnueabi-gcc thread_join.c -o thread_join -lpthread
```

Tiến hành chạy chương trình chúng ta có kết quả sau:

```
./thread_join  
Wait for thread to finish ...  
Thread function is executing ...  
Thread data is Hello world!  
'1' got '0'  
'1' got '1'  
'1' got '2'  
'1' got '3'  
'1' got '4'  
Thread completed, it returned Thanks for your using me!  
Message is now: Goodbye!
```

Chúng ta có thể tạo ra nhiều tuyến hoạt động cùng một lúc và tuyến chính cũng có thể chờ đợi nhiều tuyến cùng một lúc. Bằng cách nạp các tuyến cần chờ đợi vào danh sách chờ đợi khi dùng hàm pthread\_join(), tuyến nào được nạp vào hàng đợi trước sẽ được hoàn thành trước nhất nếu các tuyến đó có cùng thời gian thực thi.

Các tuyến có thể sử dụng chung các biến toàn cục đã định nghĩa, đây cũng là một ưu điểm của lập trình đa tuyến so với lập trình đa tiến trình. Tuy nhiên lập trình đa tuyến có một yếu điểm là tính không bền vững trong quá trình thực thi lệnh. Trong quá trình thực thi một tiến trình có nhiều tuyến chạy đồng thời, nếu có một tuyến bị lỗi thì toàn bộ tiến trình sẽ bị lỗi. Còn lập trình đa tiến trình thì không như thế, khi một tiến trình bị lỗi, các tiến trình khác điều hoạt động bình thường. Chúng ta phải biết tận dụng cả hai ưu điểm của hai phương pháp lập trình trên. Đồng thời khắc phục nhược điểm của chúng. Bằng việc phân công phù hợp nhiệm vụ của từng tác vụ trong khi lập trình.

### **III. Kết Luận:**

Đến đây chúng ta đã biết cách khởi tạo một tuyến phụ chạy song song với một tuyến chính đang thực thi bằng hàm `pthread_create()`. Chúng ta cũng đã biết cách đồng bộ hóa hoạt động giữa các tuyến với nhau, tuyến chính có thể thực hiện đợi cho một hay nhiều tuyến khác thực hiện xong việc xử lý thông tin sau đó mới tiếp tục công việc xử lý của mình, bằng cách dùng hàm `pthread_join()`.

Đến đây chúng ta đã kết thúc phần lập trình ứng dụng trong lớp user application của phần mềm hệ thống nhúng. Trong phần sau chúng ta sẽ nghiên cứu cách viết chương trình trong một môi trường khác là kernel driver, đây là một lớp trung gian giao tiếp giữa hệ điều hành và phần cứng hệ thống.

**PHẦN B**

**CĂN BẢN LẬP TRÌNH DRIVER**

**BÀI 1****DRIVER VÀ APPLICATION  
TRONG HỆ THỐNG NHÚNG****I. Khái quát về hệ thống nhúng:**

Hệ thống nhúng (embedded system) được ứng dụng rất nhiều trong cuộc sống ngày nay. Theo định nghĩa, hệ thống nhúng là một hệ thống xử lý và điều khiển những tác vụ đặc trưng trong một hệ thống lớn với yêu cầu tốc độ xử lý thông tin và độ tin cậy rất cao. Nó bao gồm phần cứng và phần mềm cùng phối hợp hoạt động với nhau, tùy thuộc vào tài nguyên phần cứng mà hệ thống sẽ có phần mềm điều khiển phù hợp. Đôi khi chúng ta thường nhầm lẫn hệ thống nhúng với máy tính cá nhân. Hệ thống nhúng cũng bao gồm phần cứng (CPU, RAM, ROM, USB, ...) và phần mềm (Application, Driver, Operate System, ...). Thế nhưng khác với máy tính cá nhân, các thành phần này đã được rút gọn, thay đổi cho phù hợp với một mục đích nhất định của ứng dụng sao cho tối ưu hóa thời gian thực hiện đáp ứng yêu cầu về thời gian thực (Real-time) theo từng mức độ.

Bài này sẽ đi sâu vào tìm hiểu cấu trúc bên trong phần mềm của hệ thống nhúng nhằm mục đích hiểu được vai trò của driver và application, phân phối nhiệm vụ hoạt động cho hai lớp này sao cho đạt hiệu quả cao nhất về thời gian.

**II. Cấu trúc của hệ thống nhúng:**

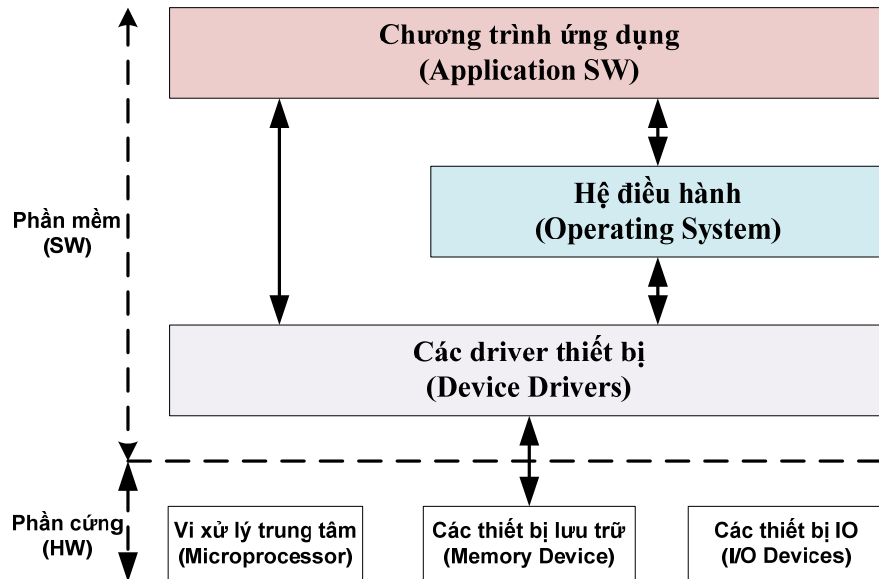
Hệ thống nhúng thông thường bao gồm những thành phần sau: Phần cứng: Bộ vi xử lý trung tâm, bộ nhớ, các thiết bị vào ra; Phần mềm: Các Driver cho thiết bị, Hệ điều hành và các chương trình ứng dụng.

Mối liên hệ giữa các thành phần được minh họa trong sơ đồ hình x\_y.

Thành phần thứ nhất trong hệ thống nhúng là phần cứng. Đây là thành phần quan trọng nhất trong hệ thống. Làm nhiệm vụ thực tế hóa những dòng lệnh từ phần mềm yêu cầu. Phần cứng của hệ thống nhúng thường bao gồm những thành phần chính sau:

- Bộ xử lý trung tâm, làm nhiệm vụ tính toán thực thi các mã lệnh được yêu cầu, được xem như bộ não của toàn hệ thống. Các bộ xử lý trong hệ thống nhúng, không giống như hệ thống máy vi tính cá nhân là những con vi xử lý mạnh chuyên về xử lý dữ liệu, là những dòng vi điều khiển mạnh, được tích hợp sẵn các module ngoại vi giúp cho việc thực thi lệnh của hệ thống được thực hiện nhanh chóng hơn. Hơn nữa tập lệnh của vi

điều khiển cũng trở nên gọn nhẹ hơn, ít tốn dung lượng vùng nhớ hơn phù hợp với đặc điểm của hệ thống nhúng. Với những vi điều khiển đã tích hợp sẵn những ngoại vi mạnh, đa dạng thì kích thước mạch phần cứng trong quá trình thi công sẽ giảm rất nhiều. Đây là ưu điểm của hệ thống nhúng so với các hệ thống đa nhiệm khác.



- Thành phần thứ hai là các thiết bị lưu trữ: Các thiết bị lưu trữ bao gồm có RAM, NAND Flash, NOR Flash, ... mặc dù bên trong vi điều khiển đã tích hợp sẵn ROM và RAM, nhưng những vùng nhớ này chỉ là tạm thời, dung lượng của chúng rất nhỏ, giúp cho việc thực thi những lệnh cũ nhanh hơn. Để lưu trữ những mã lệnh lớn như: Kernel, Rootfs, hay Application thì đòi hỏi phải có những thiết bị lưu trữ lớn hơn. RAM làm nhiệm vụ chứa chương trình thực thi một cách tạm thời. Khi một chương trình được triệu gọi, mã lệnh của chương trình được chép từ các thiết bị lưu trữ khác vào RAM, từ đây từng câu lệnh được biên dịch sẽ lần lượt đi vào vùng nhớ cache bên trong vi xử lý để thực thi. Các loại ROM như NAND Flash, NOR Flash, ... thường có dung lượng lớn nhất trong hệ thống nhúng, dùng để chứa những chương trình lớn (hệ điều hành, rootfs, bootstrapcode, ... ) lâu dài để sử dụng trong những mục đích khác nhau khi người dùng (hệ điều hành và user) cần sử dụng đến. Chúng tương tự như ổ đĩa cứng trong máy tính cá nhân.

- Các thiết bị vào ra: Đây là những module được tích hợp sẵn bên trong vi điều khiển. Chúng có thể là ADC module, Ethernet module, USB module, ... các thiết bị này có vai trò giao tiếp giữa hệ thống với môi trường bên ngoài.

Thành phần quan trọng thứ hai trong một hệ thống nhúng là phần mềm. Phần mềm của hệ thống nhúng thay đổi theo cấu trúc phần cứng. Hệ thống chỉ hoạt động hiệu quả khi phần mềm và phần cứng có sự tương thích nhau. Đi từ thấp lên cao thông thường phần mềm hệ thống nhúng bao gồm các lớp sau: Driver thiết bị, hệ điều hành, chương trình ứng dụng.

- Các driver thiết bị (device driver): Đây là những phần mềm được viết sẵn để trực tiếp điều khiển phần cứng hệ thống nhúng. Mỗi một hệ thống nhúng được cấu tạo từ những phần cứng khác nhau, những vi điều khiển với những tập lệnh khác nhau, những module khác nhau của các hãng khác nhau có cơ chế giao tiếp khác nhau, device driver làm nhiệm vụ chuẩn hóa thành những thư viện chung (có mã lệnh giống nhau), phục vụ cho hệ điều hành và người viết chương trình lập trình dễ dàng hơn. Chẳng hạn, nhiều hệ thống có giao thức truy xuất dữ liệu khác nhau, nhưng device driver sẽ quy về 2 hàm duy nhất mang tên read và write để đọc và nhập thông tin cho hệ thống xử lý. Để phân biệt giữa các thiết bị với nhau, device driver sẽ cung cấp một ID duy nhất cho thiết bị đó nhằm mục đích thuận tiện cho việc quản lý. \*\*Device driver sẽ được trình bày rất rõ trong những bài khác.

- Hệ điều hành: Đây cũng là một phần mềm trong hệ thống nhúng, nhiệm vụ của nó là quản lý tài nguyên hệ thống. Bao gồm quản lý tiến trình, thời gian thực, truy xuất vùng nhớ ảo và vùng nhớ vật lý, các giao thức mạng, ...

- Chương trình ứng dụng: Các chương trình ứng dụng là do người dùng lập trình. Thông thường trong hệ thống nhúng, công việc lập trình và biên dịch không nằm trên chính hệ thống đó. Ngược lại được nằm trên một hệ thống đa nhiệm khác, quá trình này gọi là biên dịch chéo (cross-compile). Sau khi biên dịch xong, chương trình đã biên dịch được chép vào bên trong ROM lưu trữ phục vụ cho quá trình sử dụng sau này. Các chương trình sẽ sử dụng những dịch vụ bên trong hệ điều hành (tạo tiến trình, tạo tuyến, trì hoãn thời gian, ...) và những hàm được định nghĩa trong device driver (giao tiếp thiết bị đầu cuối, truy xuất IO, ...) để tác động đến phần cứng của hệ thống.

*\*\*Quyển sách này chủ yếu trình bày sâu về phần mềm hệ thống nhúng. Trong phần đầu chúng ta đã nghiên cứu sơ lược về cách lập trình ứng dụng, làm thế nào để trì hoãn*



*thời gian, tạo tiến trình, tạo tuyến, ... Phần này sẽ đi sâu vào lớp cuối cùng trong phần mềm là Device driver.*

### III. Mối quan hệ giữa Device drivers và Applications:

Application (chương trình ứng dụng) và Device driver (Driver thiết bị) có những điểm giống và khác nhau. Tiêu chí để so sánh dựa vào nguyên lý hoạt động, vị trí, vai trò của từng loại trong hệ thống nhúng.

Application và Device driver khác nhau căn bản ở những điểm sau:

Về cách thức mỗi loại hoạt động, đa số các Application đơn nhiệm vừa và nhỏ hoạt động xuyên suốt từ câu lệnh đầu tiên cho đến câu lệnh kết thúc một cách tuần tự kể từ khi được gọi từ người sử dụng. Trong khi đó, Device driver thì hoàn toàn khác, chúng được lập trình với theo dạng từng module, nhằm mục đích phục vụ cho việc thực hiện một thao tác của Application được gọn nhẹ và dễ dàng hơn. Mỗi module có một hay nhiều chức năng riêng, được lập trình cho một thiết bị đặc trưng và được cài đặt sẵn trên hệ điều hành để sẵn sàng hoạt động khi được gọi. Sau khi được gọi, module sẽ thực thi và kết thúc ngay lập tức. Một cách khái quát, chúng ta có thể xem: Nếu Application là chương trình phục vụ người dùng, thì Device driver là chương trình phục vụ Application. Nghĩa là Application là người dùng của Device driver.

Một điểm khác biệt giữa Application và Device driver là vấn đề an toàn khi thực thi tác vụ. Nếu một Application chỉ đơn giản thực thi và kết thúc, thì công việc của Device driver phức tạp hơn nhiều. Bên cạnh việc thực thi những lệnh được lập trình nó còn phải đảm bảo an toàn cho hệ thống khi không còn hoạt động. Nói cách khác, trước khi kết thúc, Device driver phải khôi phục trạng thái trước đó của hệ thống trả lại tài nguyên cho các Device driver khác sử dụng khi cần, tránh tình trạng xung đột phần cứng.

Một Application có thể thực thi những lệnh mà không cần định nghĩa trước đó, các lệnh này chứa trong thư viện liên kết động của hệ điều hành. Khi viết chương trình cho Application, chúng ta sẽ tiết kiệm được thời gian, cho ra sản phẩm nhanh hơn. Trong khi đó, Device driver muốn sử dụng lệnh nào thì đòi hỏi phải định nghĩa trước đó. Việc định nghĩa này được thực hiện khi chúng ta dùng khai báo `#include <linux/library.h>`, những thư viện này phải thực sự tồn tại, nghĩa là còn ở dạng mã lệnh C chưa biên dịch. Các thư viện này chứa trong hệ thống mã nguồn của hệ điều hành trước khi được biên dịch.

Một chương trình Application đang thực thi nếu phát sinh một lỗi thì không còn hoạt động được nữa. Trong khi đó, khi một tác vụ trong module bị lỗi, nó chỉ ảnh hưởng đến câu lệnh gọi mà thôi (nghĩa là kết quả truy xuất sẽ không đúng) các lệnh tiếp theo sau vẫn có thể tiếp tục thực thi. Thông thường lúc này chúng ta sẽ thoát khỏi chương trình bằng lệnh `exit(n)`, để đảm bảo dữ liệu xử lý là chính xác.

Chúng ta có hai thuật ngữ mới, user space (không gian người dùng) và kernel space (không gian kernel). Không gian ở đây chúng ta nên hiểu là không gian bộ nhớ ảo, do hệ thống Linux định nghĩa và quản lý. Các chương trình ứng dụng Application được thực thi trong user space, còn những Device driver khi được biên dịch thành tập tin ko sẽ được lưu trữ trong kernel space. Kernel space và User space liên hệ nhau thông qua hệ điều hành (operating system).

Trong khi hầu hết các lệnh trong từng tiến trình và tuyến được thực hiện tuần tự nhau, kết thúc lệnh này rồi mới thực hiện lệnh tiếp theo, trong user space; Thì các module trong device driver có thể cùng một lúc phục vụ đồng thời nhiều tiến trình, tuyến. Do đó Device driver khi lập trình phải đảm bảo giải quyết được vấn đề này tránh tình trạng xung đột vùng nhớ, phần cứng trong quá trình thực thi.

#### **IV. Kết luận:**

Chúng ta đã tìm hiểu sơ lược về cấu trúc tổng quát trong hệ thống nhúng, hiểu được vai trò chức năng của từng thành phần. Bên cạnh đó chúng ta cũng đã phân biệt được những đặc điểm khác nhau giữa chương trình trong user space và Device Driver trong kernel space. Những kiến thức này rất quan trọng khi bước vào lập trình driver cho thiết bị. Chúng ta phải biết phân công nhiệm vụ giữa user application và kernel driver sao cho đạt hiệu quả cao nhất.

Bài tiếp theo chúng ta sẽ đi vào tìm hiểu các loại driver trong hệ thống Linux, cách nhận dạng từng loại, cũng như các thao tác cần thiết khi làm việc với driver.

## **BÀI 2**

# **PHÂN LOẠI VÀ NHẬN DẠNG DRIVER TRONG LINUX**

### **I. Tổng quan về Device Driver:**

Một trong những mục đích quan trọng nhất khi sử dụng hệ điều hành trong hệ thống nhúng là làm sao cho người sử dụng không nhận biết được sự khác nhau giữa các loại phần cứng trong quá trình điều khiển. Nghĩa là hệ điều hành sẽ quy những thao tác điều khiển khác nhau của nhiều loại phần cứng khác nhau thành một thao tác điều khiển chung duy nhất. Ví dụ như, hệ điều hành quy định tất cả những ổ đĩa, thiết bị vào ra, thiết bị mạng điều dưới dạng tập tin và thư mục. Việc khởi động hay tắt thiết bị chỉ đơn giản là đóng hay mở tập tin (thư mục) đó còn sau khi thao tác đóng hay mở hệ điều hành làm gì đó là công việc của device driver.

Trong một hệ thống nhúng, không phải chỉ có CPU mới có thể xử lý thông tin mà tất cả những thiết bị phần cứng đều có một cơ cấu điều khiển được lập trình sẵn, đặc trưng cho từng thiết bị. Mỗi một thẻ nhớ, USB, chuột, USB Camera, ... đều là những hệ thống nhúng độc lập, chúng có từng nhiệm vụ riêng, đảm trách một công việc xử lý thu thập thông tin cụ thể. Mỗi bộ điều khiển của các thiết bị đó đều chứa những thanh ghi lệnh và thanh ghi trạng thái. Và để điều khiển được thì chúng ta phải cung cấp những số nhị phân cần thiết vào thanh ghi lệnh, đọc thanh ghi trạng thái cho biết trạng thái thực hiện. Tương tự khi muốn thu thập dữ liệu, chúng ta phải cung cấp những mã cần thiết, theo những bước cần thiết do nhà sản xuất quy định. Thay vì phải làm những công việc nhàm chán đó, chúng ta sẽ giao cho device driver đảm trách. Device driver thực chất là những hàm được lập trình sẵn, nạp vào hệ điều hành. Có ngõ vào là những giao diện chung, ngõ ra là những thao tác riêng biệt điều khiển từng thiết bị của device driver đó.

Linux cung cấp cho chúng ta 3 loại device driver: Character driver, block driver, và network driver. Character driver hoạt động theo nguyên tắc truy xuất dữ liệu không có vùng nhớ đệm, nghĩa là thông tin sẽ di chuyển lập tức từ nơi gửi đến nơi nhận theo từng byte. Block driver thì khác, hoạt động theo cơ chế truy xuất dữ liệu theo vùng nhớ đệm. Có hai vùng nhớ đệm, đệm ngõ vào và đệm ngõ ra. Dữ liệu trước khi di chuyển vào (ra) hệ thống phải chứa trong vùng nhớ đệm, cho đến khi vùng nhớ đệm đầy thì mới được phép xuất (nhập). Nghĩa là dữ liệu di chuyển theo từng khối. Network driver hoạt động theo một cách riêng dạng socket mạng, chủ yếu dùng trong truyền nhận dữ liệu từ xa giữa các máy với nhau trong mạng cục bộ hay internet bằng các giao thức mạng phổ biến.

**\*\***Trong suốt phần này chúng ta chủ yếu nghiên cứu về character driver. Mục tiêu là có thể tự mình thiết kế một character driver đơn giản;

## II. Các đặc điểm của device driver trong hệ điều hành Linux:

Chúng ta đã biết như thế nào là device driver, đặc điểm của từng loại device driver. Thế nhưng các loại driver này được hệ điều hành Linux quản lý như thế nào?

Bất kỳ một thiết bị nào trong hệ điều hành Linux cũng được quản lý thông qua tập tin và thư mục hệ thống. Chúng được gọi là các tập tin thiết bị hay là các tập tin hệ thống. Những tập tin này đều chứa trong thư mục /dev. Trong thư mục /dev chúng ta thực hiện lệnh `ls -l`, hệ thống sẽ cho ra kết quả sau:

```
crw-rw-rw-  1 root    root      1,   3 Apr 11  2002 null
crw-----  1 root    root     10,   1 Apr 11  2002 psaux
crw-----  1 root    root      4,   1 Oct 28 03:04 tty1
crw-rw-rw-  1 root    tty      4,  64 Apr 11  2002 ttys0
crw-rw----  1 root    uucp      4,  65 Apr 11  2002 ttyS1
crw--w----  1 vcsa    tty       7,   1 Apr 11  2002 vcs1
crw--w----  1 vcsa    tty     7, 129 Apr 11  2002 vcsa1
crw-rw-rw-  1 root    root      1,   5 Apr 11  2002 zero
...
```

Cột thứ nhất cho chúng ta thông tin về loại device driver. Theo thông tin trên thì tất cả đều là character driver vì những ký tự đầu tiên đều là “c”, tương tự nếu là block driver thì ký tự đầu là “b”. Chúng ta chú ý đến cột thứ 4 và 5, tại đây có hai thông tin cách nhau bằng dấu “,” hai số này được gọi là Major và Minor. Mỗi thiết bị trong hệ điều hành đều có một số 32 bits riêng biệt để quản lý. Số này được chia thành hai thông tin, thông tin thứ nhất là Major number. Major number là số có 12 bit, dùng để phân biệt từng nhóm thiết bị với nhau, hay nói cách khác những thiết bị cùng loại sẽ có chung một số Major. Các thiết bị cùng loại có cùng số Major được phân biệt nhau thông qua thông tin thứ hai là số Minor. Số Minor là số có chiều dài 20 bit. Với hai số Major và Minor, tổng cộng hệ điều hành có thể quản lý số thiết bị tối đa là  $2^{12} \times 2^{20}$  tương đương với  $2^{32}$ .

Trong lập trình driver, đôi khi chúng ta muốn thao tác với hai thông tin Major và Minor numbers. Kernel cung cấp cho chúng ta những hàm rất hữu ích để thực hiện những công việc này. Sau đây là một số hàm tiêu biểu:

```
#include <linux/types.h>
#include <linux/kdev_t.h>
int MAJOR (dev_t dev);
```

```
int MINOR (dev_t dev);  
dev_t MKDEV (int major, int minor);
```

Trước khi sử dụng những hàm này, chúng ta phải khai báo thư viện phù hợp cho chúng. thư viện <linux/types.h> chứa định nghĩa kiểu dữ liệu dev\_t, biến kiểu này dùng để chứa số định danh cho thiết bị. Thư viện <linux/kdev\_t.h> chứa định nghĩa cho những hàm MAJOR(), MINOR(), MKDEV, ...

Hàm MAJOR (dev\_t dev) dùng để tách số Major của thiết bị dev\_t dev và lưu vào một biến kiểu int;

Hàm MINOR (dev\_t dev) dùng để tách số Minor của thiết bị dev\_t dev và lưu vào một biến kiểu int;

Hàm MKDEV (int major, int minor) dùng để tạo thành một số định danh thiết bị kiểu dev\_t từ hai số int major, và int minor;

Đối với kernel 2.6 trở đi thì số device driver dev\_t có 32 bit. Nhưng đối với những kernel đời sau đó thì dev\_t có 16 bit.

### **III. Kết luận:**

Trong bài này chúng ta đã đi vào tìm hiểu một cách khái quát vai trò ý nghĩa của từng loại device driver trong hệ thống Linux, mỗi device driver đều có những ưu và nhược điểm riêng và đóng góp một phần để làm cho hệ thống chạy ổn định. Chúng ta cũng đã biết cách thức quản lý thông tin thiết bị của Linux thông qua thư mục /dev, mỗi thiết bị trong Linux đều có một số định danh, tùy vào từng hệ thống mà số này có bao nhiêu bit, số định danh có thể được tạo thành từ hai số riêng biệt Major và Minor bằng hàm MKDEV() hoặc có thể tách riêng một số định danh dev\_t thành hai số Major và Minor bằng hai hàm MAJOR() và MINOR (). Những hàm này rất quan trọng trong lập trình driver.

Trong giới hạn về thời gian, quyển sách này chỉ trình bày cho các bạn cách lập trình một character driver. Trên cơ sở đó các bạn sẽ tự mình tìm hiểu cách lập trình cho các loại driver khác. Bài sau chúng ta sẽ tìm hiểu sâu hơn về character driver, cấu trúc dữ liệu bên trong, các hàm thao tác khởi tạo character device, ...

**BÀI 3****CHARACTER DEVICE DRIVER****I. Tổng quan character device driver:**

Character device driver là một trong 3 loại driver trong hệ thống Linux. Đây là driver dễ và phổ biến nhất trong các ứng dụng giao tiếp vừa và nhỏ đối với lập trình nhúng. Character driver và các loại driver khác đều được hệ điều hành quản lý dưới dạng tập tin và thư mục. Hệ điều hành sử dụng các hàm truy xuất tập tin chuẩn để giao tiếp trao đổi thông tin giữa người lập trình và thiết bị do driver điều khiển. Chẳng hạn những hàm như `read`, `write`, `open`, `release`, `close`, ... được dùng chung cho tất cả các character driver, những hàm này còn được gọi là giao diện điều khiển giữa hệ điều hành (được người lập trình ra lệnh) và device driver (được hệ điều hành ra lệnh). Hoạt động bên trong giao diện này là những thao tác của từng device driver đặc trưng cho từng thiết bị đó. Công việc lập trình các thao tác này gọi là lập trình driver.

Một character driver muốn cài đặt và hoạt động bình thường thì phải trải qua nhiều bước lập trình. Đầu tiên là đăng ký số định danh cho driver, số định danh là số mà hệ điều hành linux cung cấp cho mỗi driver để quản lý. Tiếp theo, mô tả tập lệnh mà driver hỗ trợ, chúng ta có thể xem tập lệnh là những thao tác hoạt động bên trong của driver dùng để điều khiển một thiết bị vật lý. Sau khi đã mô tả tập lệnh, chúng ta sẽ liên kết các tập lệnh này với các giao diện chuẩn mà hệ điều hành linux hỗ trợ, nhằm mục đích giao tiếp giữa hệ điều hành và các thiết bị ngoại vi vật lý mà driver điều khiển. Tiếp theo chúng ta định nghĩa liên kết các giao diện này với cấu trúc mô tả tập tin khi thiết bị được mở. Cuối cùng chúng ta thực hiện cài đặt driver thiết bị vào hệ thống thư mục tập tin linux, thông thường nằm trong thư mục `/dev`.

Trong phần này chúng ta sẽ tìm hiểu một cách chi tiết các bước lập trình driver đã nêu.

**II. Số định danh character driver:**

Thế nào là số định danh, đặc điểm và vai trò của số định danh của character driver cũng hoàn toàn tương tự như device driver khác mà chúng ta đã nghiên cứu rất kỹ trong bài trước. Chúng ta cũng đã biết những hàm rất quan trọng để thao tác với số định danh

này. Ở đây không nhắc lại mà thêm vào đó là làm thế nào để tạo lập một số định danh cho thiết bị nào đó mà không sinh ra lỗi

### ***a. Xác định số định danh hợp lệ cho thiết bị mới theo cách thông thường:***

Một trong những công việc quan trọng đầu tiên cần phải làm trong driver là xác định số định danh cho thiết bị. Có hai thông tin cần xác định là số *Major* và số *Minor*.

Trước hết chúng ta phải biết số định danh nào còn trống trong hệ thống chưa được các thiết bị khác sử dụng. Thông tin về số định danh được linux sử dụng chứa trong tập tin *Documentation/devices.txt*. Tập tin này chứa số định danh, tên thiết bị, thời gian tạo lập, loại thiết bị, ... đã được linux sử dụng hay sẽ dùng cho những mục đích đặc biệt nào đó. Đọc nội dung trong tập tin này, chúng ta sẽ tìm được số định danh phù hợp, và công việc tiếp theo là đăng ký số định danh đó vào linux.

Linux kernel cung cấp cho chúng ta một hàm dùng để đăng ký số định danh cho thiết bị, hàm đó là:

```
#include <linux/fs.h>

int register_chrdev_region (dev_t first, unsigned int count, char
*name);
```

Để sử dụng được hàm, chúng ta phải khai báo thư viện `<linux/fs.h>`. Tham số thứ nhất `dev_t first` là số định danh thiết bị đầu tiên muốn đăng ký với số Major là số hợp lệ chưa được sử dụng, Minor thông thường cho bằng 0. Tham số thứ hai `unsigned int count` là số thiết bị muốn đăng ký, chẳng hạn muốn đăng ký 1 thiết bị thì ta nhập 1, lúc này chỉ có một thiết bị mang số định danh là `dev_t first` được đăng ký. Tham số thứ ba `char *name` là tên thiết bị muốn đăng ký.

Hàm `register_chrdev_region ()` trả về giá trị kiểu `int` là 0 nếu quá trình đăng ký thành công. Và trả về số mã lỗi âm khi quá trình đăng ký không thành công.

Tất cả những thông tin khi đăng ký thành công sẽ được hệ điều hành chứa trong tập tin `/proc/devices` và `sysfs` khi quá trình cài đặt thiết bị kết thúc.

Cách đăng ký số định danh trên có một nhược điểm lớn là chỉ áp dụng khi người đăng ký đồng thời là người lập trình nên driver đó vì thế họ sẽ biết rõ số định danh nào là còn trống. Khi driver được sử dụng trên những máy tính khác, thì số định danh được chọn có thể bị trùng với các driver khác. Vì thế việc lựa chọn một số định danh động là cần thiết.

Vì số định danh động sẽ không trùng với bất kỳ số định danh nào tồn tại trong hệ thống.

Ví dụ, nếu muốn đăng ký một character driver có tên là "lcd\_dev", số lượng là 1, số Major đầu tiên là 2, chúng ta tiến hành khai báo hàm như sau:

```
/*Khai báo biến lưu trữ mã lỗi trả về của hàm*/
int res;
/*Thực hiện đăng ký thiết bị cho hệ thống*/
res = register_chrdev_region (2, 1, "lcd_dev");
if (res < 0) {
    printk ("Register device error!");
    exit (1);
}
```

### ***b. Xác định số định danh cho thiết bị theo cách ngẫu nhiên:***

Linux cung cấp cho chúng ta một hàm đăng ký số định danh động cho driver thiết bị mới.

```
#include <linux/fs.h>
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
unsigned int count, char *name);
```

Cũng tương tự như hàm `register_chrdev_region ()`, hàm `alloc_chrdev_region ()` cũng làm nhiệm vụ đăng ký định danh cho một thiết bị mới. Nhưng có một điểm khác biệt là số Major trong định danh không còn cố định nữa, số này do hệ thống linux tự động cấp vì thế sẽ không trùng với bất kỳ số định danh nào khác đã tồn tại.

Tham số thứ nhất của hàm, `dev_t *dev`, là con trỏ kiểu `dev_t` dùng để lưu trữ số định danh đầu tiên trong khoảng định danh được cấp nếu hàm thực hiện thành công;

Tham số thứ hai, `unsigned int first minor`, là số Minor đầu tiên của khoảng định danh muốn cấp;

Tham số thứ ba, `unsigned int count`, là số lượng định danh muốn cấp, tính từ số Major được cấp động và số Minor `unsigned int first minor`;



Tham số thứ tư, `char *name`, là tên của driver thiết bị muốn đăng ký.

Ví dụ khi muốn đăng ký thiết bị tên `"lcd_dev"`, số Minor đầu tiên là 0, số thiết bị muốn đăng ký là 1, số định danh khi tạo ra được lưu vào biến `dev_t dev_id`. Lúc này hàm `alloc_chrdev_region ()` khai báo như sau:

```
/*Khai báo biến dev_t để lưu giá trị định danh đầu tiên trả về của hàm*/
dev_t dev_id;
/*Khai báo biến lưu mã lỗi trả về của hệ thống*/
int res;
/*Thực hiện đăng ký thiết bị với định danh động*/
res = alloc_chrdev_region (&dev_id, 0, 1, "lcd_dev");
/*Kiểm tra mã lỗi trả về*/
if ( res < 0) {
    printk ("Allocate devices error!");
    return res;
}
```

Tuy nhiên việc đăng ký số định danh động cho thiết bị đôi khi cũng có nhiều bất lợi. Giả sử số định danh của thiết bị cần được sử dụng cho những mục đích khác, vì thế số định danh luôn thay đổi khi mỗi lần cài đặt driver sẽ sinh ra lỗi trong quá trình thực thi lệnh. Để kết hợp ưu điểm của 2 phương pháp, chúng ta sẽ đăng ký driver thiết bị theo cách sau:

```
/*Khai báo các biến cần thiết*/
int lcd_major; //Biến lưu trữ số Major
int lcd_minor; //Biến lưu trữ số Minor
dev_t dev_id; //Biến lưu trữ số định danh thiết bị
int result; //Biến lưu mã lỗi
/*Nếu số Major hợp lệ, đã tồn tại*/
if (lcd_major) {
    dev = MKDEV(lcd_major, lcd_minor); //Tạo số định danh
/*Đăng ký thiết bị với số định danh cố định*/
    result = register_chrdev_region (dev, lcd_nr_devs,
    "lcd_dev");
```

```
    } else {  
        /*Nếu số Major chưa tồn tại, thực hiện tìm kiếm số Major động*/  
        result = alloc_chrdev_region(&dev_id,    lcd_minor,    lcd_nr_devs,  
        "lcd_dev");  
        /*Cập nhật lại số Major động cần sử dụng trong những lần sau*/  
        lcd_major = MAJOR (dev_id);  
    }  
    /*Kiểm tra kết quả thực thi của hai lệnh trên*/  
    if (result < 0) {  
        printk(KERN_WARNING "lcd: can't get major %d\n", lcd_major);  
        return result;  
    }
```

Như vậy ta có thể cập nhật lại số định danh động khi vừa tạo ra để sử dụng cho những chương trình liên quan bằng kỹ thuật như trong đoạn mã lệnh trên.

Character driver bao gồm có nhiều thành phần, đăng ký số định danh chỉ là một trong những thành phần đó. Bên cạnh số định danh character driver còn có những bộ phận như: Cấu trúc dữ liệu (data structure) được gọi là `file_operation`, cấu trúc này chứa những tập lệnh được người lập trình driver định nghĩa; Cấu trúc mô tả tập tin (file) chứa những thông tin cơ bản của tập tin thiết bị; Cấu trúc tập tin chứa thông tin quản lý tập tin thiết bị trong hệ thống linux.

Phần tiếp theo chúng ta sẽ tìm hiểu cách gán các hành vi cho character device driver thông qua việc thao tác với `file_operations`.

### **III. Cấu trúc lệnh của character driver:**

Cấu trúc lệnh của character driver (`file_operations`) là một cấu trúc dùng để liên kết những hàm chứa các thao tác của driver điều khiển thiết bị với những hàm chuẩn trong hệ điều hành giúp giao tiếp giữa người lập trình ứng dụng với thiết bị vật lý. Cấu trúc `file_operation` được định nghĩa trong thư viện `<linux/fs.h>`. Mỗi một tập tin thiết bị được mở trong hệ điều hành linux đều được hệ điều hành dành cho một vùng nhớ mô tả cấu trúc tập tin, trong cấu trúc tập tin có rất nhiều thông tin liên quan phục vụ cho việc thao tác với tập tin đó (chúng ta sẽ nghiên cứu kỹ trong phần sau). Một trong những thông tin này là `file_operations`, dùng mô tả những hàm mà driver thiết bị đang được mở

hỗ trợ. Có thể nói một cách khác mỗi tập tin thiết bị trong hệ thống linux tương tự như một vật thể và *file\_operation* là những công dụng của vật thể đó.

Cấu trúc *file\_operations* là một thành phần trong cấu trúc *file\_structure* khi tập tin thiết bị được mở. Mỗi thành phần trong *file\_operations* bao gồm những lệnh căn bản theo chuẩn do hệ điều hành định nghĩa, nhưng những lệnh này chưa được định nghĩa thao tác cụ thể, đây là nhiệm vụ của người lập trình driver. Chúng ta phải liên kết những thao tác muốn lập trình với những dạng hàm chuẩn này.

Sau đây chúng ta sẽ tìm hiểu một số những thành phần quan trọng trong cấu trúc *file\_structure*:

```
struct module *owner
```

Đây không phải là một lệnh trong driver mà chỉ con trỏ cho biết tên driver nào quản lý những lệnh được liên kết. Thông tin này được thiết lập thông qua macro *THIS\_MODULE* định nghĩa trong thư viện `<linux/module.h>`.

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Hàm chuẩn này dùng để yêu cầu nhận dữ liệu từ thiết bị vật lý. Nhận dữ liệu như thế nào sẽ do người lập trình quyết định, phù hợp với quy định của từng thiết bị. Tham số thứ nhất, `struct file *`, là con trỏ đến cấu trúc tập tin đang mở trong hệ điều hành, dùng để phân biệt thiết bị này với thiết bị khác. Tham số thứ hai, `char __user *`, là con trỏ được khai báo trong user space, chứa thông tin đọc được từ thiết bị. Tham số thứ ba, `size_t` là kích thước dữ liệu muốn đọc (tính bằng byte). Tham số thứ tư, `loff_t *`, là con trỏ chỉ vị trí dữ liệu trong thiết bị cần đọc về, nếu để trống thì mặc định là vị trí đầu tiên. Hàm có giá trị trả về là kích thước dữ liệu đọc về thành công.

```
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

Hàm này dùng để ghi thông tin của người dùng vào thiết bị vật lý. Các thao tác ghi cụ thể sẽ do người lập trình quyết định tùy theo từng thiết bị phần cứng. Tham số thứ nhất, `struct file *`, là con trỏ đến cấu trúc tập tin đang mở trong hệ điều hành, dùng để phân biệt thiết bị này với thiết bị khác khi sử dụng nhiều thiết bị. Tham số thứ hai, `char __user *`, là con trỏ được khai báo trong user space, chứa thông tin muốn ghi từ người sử dụng. Tham số thứ ba, `size_t` là kích thước dữ liệu muốn ghi (tính bằng byte). Tham số thứ tư, `loff_t *`, là con trỏ chỉ địa dữ liệu trong thiết bị cần ghi thông tin, nếu để

trông thì mặc định là vị trí đầu tiên. Hàm có giá trị trả về là kích thước dữ liệu ghi thành công.

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

Đây là hàm luôn được thực thi khi thao tác với driver. Hàm được gọi khi ta sử dụng lệnh mở tập tin driver thiết bị sử dụng. Chúng ta không cần thiết phải lập trình thao tác cho lệnh này. Trong cấu trúc lệnh có thể đặt giá trị `NULL`, như vậy khi đó driver sẽ không được cảnh báo khi thiết bị được mở. Mặc dù không quan trọng nhưng chúng ta nên khai báo lệnh `open_device` trong chương trình để sử dụng mã lỗi trả về khi cần thiết.

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

Hàm chuẩn này được thực thi khi driver thiết bị không còn sử dụng, thoát khỏi hệ thống linux. Cũng tương tự như hàm `open`, hàm `release` có thể không cần khai báo trong cấu trúc tập lệnh `file_operation`. Tuy nhiên để thuận lợi trong quá trình lập trình, chúng ta nên khai báo hàm `release_device` trong driver để có thể trả về mã lỗi nếu cần thiết.

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

`ioctl` là một hàm rất mạnh trong cấu trúc tập lệnh `file_operations`. Hàm này có thể tích hợp nhiều hàm khác do người lập trình driver định nghĩa. Những hàm khác nhau được phân biệt thông qua các tham số của hàm `ioctl`. Tham số thứ nhất, `struct inode *`, là cấu trúc tập tin trong hệ thống thư mục linux (chúng ta sẽ nghiên cứu trong phần sau). Tham số thứ hai, `struct file *`, là cấu trúc tập tin đang mở trong hệ thống linux. Tham số thứ ba, `unsigned int`, là số `unsigned int` phân biệt những lệnh khác nhau, có thể gọi đây là số định danh lệnh. Tham số thứ ba, dạng `unsigned long`, là tham số của hàm tương ứng với số định danh lệnh. Chúng ta sẽ nghiên cứu sâu các sử dụng hàm trong những bài sau.

Sau đây là một ví dụ cho thấy cách gán chức năng cho các hàm sử dụng trong tập lệnh của character device driver.

```
/*Khai báo cấu trúc lệnh cho driver*/  
struct file_operations lcd_fops = {  
/*Tên của module sở hữu tập lệnh này*/  
    .owner =      THIS_MODULE,  
/*Gán lệnh đọc lcd_read vào hàm chuẩn read*/
```

```
.read =      lcd_read,
/*Gán hàm ghi dữ liệu vào hàm chuẩn write*/
.write =     lcd_write,
/*Gán hàm lcd_ioctl vào hàm chuẩn ioctl*/
.ioctl =     lcd_ioctl,
/*Gán hàm khởi động thiết bị vào hàm chuẩn, có thể đặt giá trị NULL*/
.open =      lcd_open,
/*Gán hàm thoát thiết bị vào hàm chuẩn, có thể đặt giá trị NULL*/
.release =   lcd_release,
};
```

Tiếp theo chúng ta sẽ nghiên cứu cấu trúc khác lớn hơn trong character device driver. Cấu trúc này chứa những thông tin thao tác tập tin cần thiết khi tập tin đang mở trong đó có cấu trúc tập lệnh *file\_operations*.

#### **IV. Cấu trúc mô tả tập tin của character driver:**

Cấu trúc mô tả tập tin (*file\_structure*), định nghĩa trong thư viện `<linux/fs.h>` là cấu trúc quan trọng thứ hai trong *character device driver*. Cấu trúc này không xuất hiện trong hệ thống thư mục tập tin của Linux. Mà chỉ xuất hiện khi tập tin được mở, sử dụng trong hệ thống. Khi một tập tin được mở, linux sẽ cung cấp một không gian vùng nhớ lưu trữ những thông tin quan trọng phục vụ cho quá trình lập trình sử dụng tập tin. Những thông tin đó là:

```
mode_t f_mode;
```

Thông tin này quy định chế độ truy xuất tập tin thiết bị. Một tập tin khi được mở trong hệ thống sẽ có thể chỉ được phép đọc, chỉ được phép ghi, hay cả hai bằng cách sử dụng các bit cờ `FMODE_READ` và `FMODE_WRITE`. Chúng ta nên kiểm tra chế độ truy xuất của tập tin thiết bị khi sử dụng hàm `ioctl` hay `open`. Nhưng khi sử dụng hàm `read` và `write` thì không cần thiết. Vì trước khi thực thi các hàm này hệ thống sẽ tự động kiểm tra các cờ hợp lệ hay không, nếu không hệ thống sẽ bỏ qua không thực thi.

```
loff_t f_pos;
```

Là thông tin lưu vị trí truy cập tập tin, phục vụ cho thao tác `read` và `write`. Đây là số có 64 bits, khả năng truy xuất rất rộng. Người lập trình driver có thể tham khảo thông tin

này để biết vị trí hiện tại của con trỏ truy cập tập tin. Tuy nhiên nên hạn chế thay đổi thông tin này. Để thay đổi thông tin này, chúng ta có thể thay đổi trực tiếp bằng cách thay đổi tham số `filp` -> `f_pos` hoặc có thể sử dụng những hàm chuẩn trong linux.

```
unsigned int f_flags;
```

Đây là những cờ thể hiện chế độ truy cập tập tin, bao gồm những giá trị có thể như, `O_RDONLY`, `O_NONBLOCK`, `O_SYNC` trong những thông tin này thì `O_NONBLOCK` được sử dụng nhiều nhất để kiểm tra lệnh thực hiện có phải là lệnh truy xuất theo block hay không. Còn những thông tin truy xuất khác thông thường được kiểm tra thông qua `f_mode`. Các định nghĩa cho giá trị bit cờ chứa trong thư viện `<linux/fcntl.h>`

```
struct file_operations *f_op;
```

Thông tin này chứa định nghĩa các tập lệnh tương ứng của từng tập tin thiết bị. Thông tin này đã được giải thích rõ trong phần trên.

```
void *private_data;
```

Đây là con trỏ đến vùng nhớ dành riêng cho người sử dụng driver. Vùng nhớ này được xóa khi tập tin được mở, nhưng vẫn tồn tại khi tập tin được đóng, vì thế chúng ta phải tiến hành giải phóng vùng nhớ này trước khi thoát.

```
struct dentry *f_dentry;
```

Chứa thông tin về tập tin nguồn được mở, mỗi tập tin được mở trong hệ thống linux điều bắt nguồn từ một tập tin nào đó lưu trong bộ nhớ. Người viết driver thường dùng thông tin này hơn là thông tin về `i_node` của thiết bị để quản lý vị trí tập tin thiết bị được mở trong hệ thống.

Trong thực tế một tập tin tổng quát được mở trong hệ thống có thể có nhiều hơn những thông tin nêu trên. Nhưng đối với tập tin driver thì những thông tin đó không cần thiết. Tất cả những driver điều thao tác trên cơ sở những cấu trúc tập tin được xây dựng sẵn.

### V. Cấu trúc tập tin của character driver:

Cấu trúc tập tin (*inode structure*) được kernel sử dụng để đặc trưng cho một tập tin driver thiết bị. Cấu trúc này hoàn toàn khác với cấu trúc file structure được giải thích trong phần trước, điều này có nghĩa là có thể có nhiều *file structure* biểu thị cấu trúc tập tin đang mở nhưng tất cả những *file structure* này đều có nguồn gốc từ một *inode structure* duy nhất.

Kernel dùng cấu trúc file structure này để biểu diễn một tập tin thiết bị trong cấu trúc hệ thống của mình (hay nói cụ thể hơn là cấu trúc cây thư mục). Chúng ta có thể mở tập tin này với nhiều chế độ truy xuất khác nhau, mỗi chế độ truy xuất sẽ tương đương với một cấu trúc *file structure*. Cấu trúc *inode structure* chứa rất nhiều thông tin về tập tin thiết bị, trong công việc lập trình driver chúng ta chỉ quan tâm đến những thông tin sau đây:

```
dev_t i_rdev;
```

Mỗi một cấu trúc inode structure đại diện cho một tập tin thiết bị, thông tin này trong inode structure chứa số định danh thiết bị mà chúng ta đã tạo trong phần trước.

```
struct cdev *i_cdev;
```

`struct cdev` là kiểu cấu trúc lưu trữ thông tin của một tập tin lưu trữ trong kernel.

Và thông tin `i_cdev` là con trỏ đến cấu trúc này.

Linux cung cấp cho chúng ta hai hàm chuẩn để tìm số định danh Major và Minor của thiết bị biểu thị bằng inode structure. Hai hàm đó là:

```
unsigned int iminor(struct inode *inode);
```

```
unsigned int imajor(struct inode *inode);
```

Chúng ta nên dùng hàm này để lấy thông tin về số định danh thiết bị bên cạnh việc truy cập trực tiếp thông tin `dev_t i_rdev` trong cấu trúc inode structure.

### VI. Cài đặt character device driver vào hệ thống Linux:

Sau khi đăng ký số định danh thiết bị, thiết lập liên kết với *file\_operations*, gán *file\_operations* với *file\_structure*, và định nghĩa một *inode\_structure*, chúng ta đã hoàn thành cơ bản một *character device driver*. Công việc cuối cùng là tiến hành cài đặt *character driver* này vào hệ điều hành và sử dụng. Phần này sẽ trình bày cho chúng ta các bước để cài đặt những cấu trúc trên vào *kernel* trở thành một *character driver* hoàn chỉnh.

Những hàm được giới thiệu sau đây được định nghĩa trong thư viện `<linux/cdev.h>`.

Trước khi cài đặt thông tin vào kernel chúng ta phải khai báo cho kernel dành ra một không gian vùng nhớ riêng, chuẩn bị cho quá trình cài đặt. Có hai cách thực hiện công việc này.

- **Cách 1:** Khai báo cấu trúc trước khi định nghĩa thông tin.

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->ops = &my_fops;
...
```

(Tương tự cho những trường khác)

Đầu tiên chúng ta khai báo con trỏ cấu trúc dạng `struct cdev` để chứa con trỏ `struct cdev` trống do hàm `cdev_alloc()` trả về. Tiếp theo, định nghĩa từng thông tin liên quan cho cấu trúc vừa tạo ra. Chẳng hạn, trong câu lệnh trên, chúng ta cập nhật con trỏ cấu trúc lệnh cho `cdev` này bằng lệnh gán cơ bản `my_cdev->ops = &my_fops;` trong đó `&my_fops` là cấu trúc lệnh đã được tạo thành từ trước được gán vào trường `ops` của cấu trúc `my_cdev`.

- **Cách 2:** Thực hiện định nghĩa thông tin trước khi khai báo cho kernel.

```
struct cdev *my_cdev;
my_cdev->ops = &my_ops;
...
(Tiếp theo cho những trường khác)
```

```
cdev_init ( my_cdev, my_ops);
```

Như vậy, sau khi định nghĩa xong các trường cần thiết cho cấu trúc `struct cdev`, chúng ta tiến hành gọi hàm `cdev_init ()`; để thông báo cho kernel dành ra một vùng nhớ riêng lưu trữ `cdev` mới vừa tạo ra.

Cấu trúc của hàm `cdev_init ()` như sau:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

Với `struct cdev *cdev` là con trỏ cấu trúc lưu thông tin driver đã được khai báo, `struct file_operations *fops` là cấu trúc tập lệnh của driver.

Sau khi khai báo cho kernel dành một vùng nhớ lưu trữ cấu trúc driver, công việc cuối cùng là triệu gọi hàm `cdev_add ()` để cài đặt cấu trúc này vào kernel. Cấu trúc của hàm `cdev_add` như sau:

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Trong đó, `struct cdev *dev` là con trỏ cấu trúc driver cần cài đặt. `dev_t num` là số định danh thiết bị đầu tiên muốn cài đặt vào hệ thống, số định danh này được xác định phù hợp với hệ thống thông qua các hàm đặc trưng. Tham số cuối cùng, `unsigned int count`, là số thiết bị muốn cài đặt vào kernel.



Hàm này sẽ trả về giá trị 0 nếu quá trình cài đặt driver vào kernel thành công. Ngược lại sẽ trả về mã lỗi âm. Kernel chỉ có thể gọi được những hàm trong driver khi nó được cài đặt thành công vào hệ thống.

Đôi khi chúng ta muốn tháo bỏ cấu trúc device driver ra khỏi hệ thống, linux cung cấp cho chúng ta hàm `cdev_del()` để thực hiện công việc này. Cấu trúc của hàm như sau:

```
void cdev_del(struct cdev *dev);
```

Với `cdev *dev` là cấu trúc driver muốn tháo bỏ, khi driver được tháo bỏ, thì kernel không thể sử dụng những hàm định nghĩa trong kernel được nữa.

### VII. Tổng kết:

Trên đây chúng ta đã tìm hiểu rất kỹ những thành phần cấu tạo nên một character driver do hệ thống linux định nghĩa và quản lý. Chúng ta cũng đã biết cách kết nối những trường có liên quan trong từng cấu trúc với những hàm được định nghĩa và cách cài đặt những cấu trúc đó vào kernel.

Để tìm hiểu hơn về nguyên tắc giao tiếp giữa driver và user, trong bài sau chúng ta sẽ bàn về các giao diện chuẩn trong cấu trúc lệnh của character device driver, đó là các hàm `read()`, `write()` và `ioctl()`;

**BÀI 4****CÁC GIAO DIỆN HÀM TRONG DRIVER****I. Tổng quan về giao diện trong cấu trúc lệnh `file_operations`:**

Cấu trúc lệnh `file_operations` là một trong 3 cấu trúc quan trọng của *character device driver*, bao gồm một số những hàm chuẩn giúp giao tiếp giữa chương trình ứng dụng trong *user* và *driver* trong *kernel*. Chương trình ứng dụng chứa những yêu cầu thực thi từ người dùng chạy trong môi trường *user(user space)*. *Driver* chứa những hàm chức năng điều khiển thiết bị vật lý được lập trình sẵn chạy trong môi trường *kernel(kernel space)*. Để những yêu cầu từ *user* được nhận ra và điều khiển được phần thiết bị vật lý thì cần phải có một giao diện điều khiển. Những giao diện điều khiển này thực chất là những hàm `read()`, `write()`, và `ioctl()`, ... được linux định nghĩa sẵn, bản thân nó không có chức năng cụ thể, chức năng cụ thể được định nghĩa bởi người lập trình *driver*. Chẳng hạn, hàm `read()` có chức năng tổng quát là đọc thông tin từ driver đến một vùng nhớ đệm trong *user* (của một chương trình ứng dụng đang chạy), nhưng *driver* muốn lấy được thông tin cung cấp cho *user* thì phải qua nhiều thao tác điều khiển các thanh ghi lệnh thanh ghi dữ liệu của thiết bị vật lý được viết bởi người lập trình *driver*, dữ liệu thu được không thể truyền qua *user* một cách trực tiếp mà phải thông qua các hàm hỗ trợ trong giao diện khác như `copy_to_user()`.

Trong phần này, chúng ta sẽ tìm hiểu nguyên lý hoạt động của những giao diện này, làm cơ sở để viết hoàn chỉnh một *character driver* trong những bài sau.

**II. Giao diện `read()` & `write()`:**

Do hàm `read()` và `write()` nói chung điều có cùng một nhiệm vụ là trao đổi thông tin qua lại giữa *user (application)* và *kernel (driver)*. Nếu hàm `read()` dùng để chép dữ liệu từ *driver* qua *application*, thì ngược lại hàm `write()` dùng để chép dữ liệu từ *application* sang *driver*. Hơn nữa hai hàm này điều có những tham số tương tự nhau. Vì thế chúng ta sẽ nghiên cứu hai hàm này cùng lúc.

**a. Cấu trúc lệnh:**

```
ssize_t read(struct file *filp, char __user *buff, size_t count,
loff_t *offp);

ssize_t write(struct file *filp, const char __user *buff, size_t
count, loff_t *offp);
```

### b. Giải thích:

Trong cả hai hàm trên, `filp` là con trỏ tập tin thiết bị muốn truy xuất dữ liệu; `count` là kích thước muốn truy xuất tính bằng đơn vị byte. `buff` là con trỏ đến ô nhớ khai báo trong *application* để lưu dữ liệu đọc về trong trường hợp là hàm `read()`, trong trường hợp là hàm `write()` thì chính là con trỏ đến vùng nhớ trống cần ghi dữ liệu đến. Cuối cùng `offp` là con trỏ dạng *long offset* lưu vị trí con trỏ hiện tại của tập tin đang được truy cập. Sau đây chúng ta sẽ tìm hiểu kỹ hơn từng tham số trong hàm.

Con trỏ `const char __user *buff` trong hàm `read(...)` và `write(...)` điều là những con trỏ đến một vùng nhớ trong user space do người lập trình khai báo để lưu dữ liệu truy xuất từ *driver* thiết bị. Hơn nữa *driver* hoạt động trong môi trường *kernel*. Giữa *user space* và *kernel space* có những nguyên tắc quản lý bộ nhớ khác nhau. Do đó, *driver*, hoạt động trong môi trường *kernel*, không thể giao tiếp với con trỏ này, hoạt động trong môi trường *user*, một cách trực tiếp mà phải thông qua một số hàm đặc biệt.

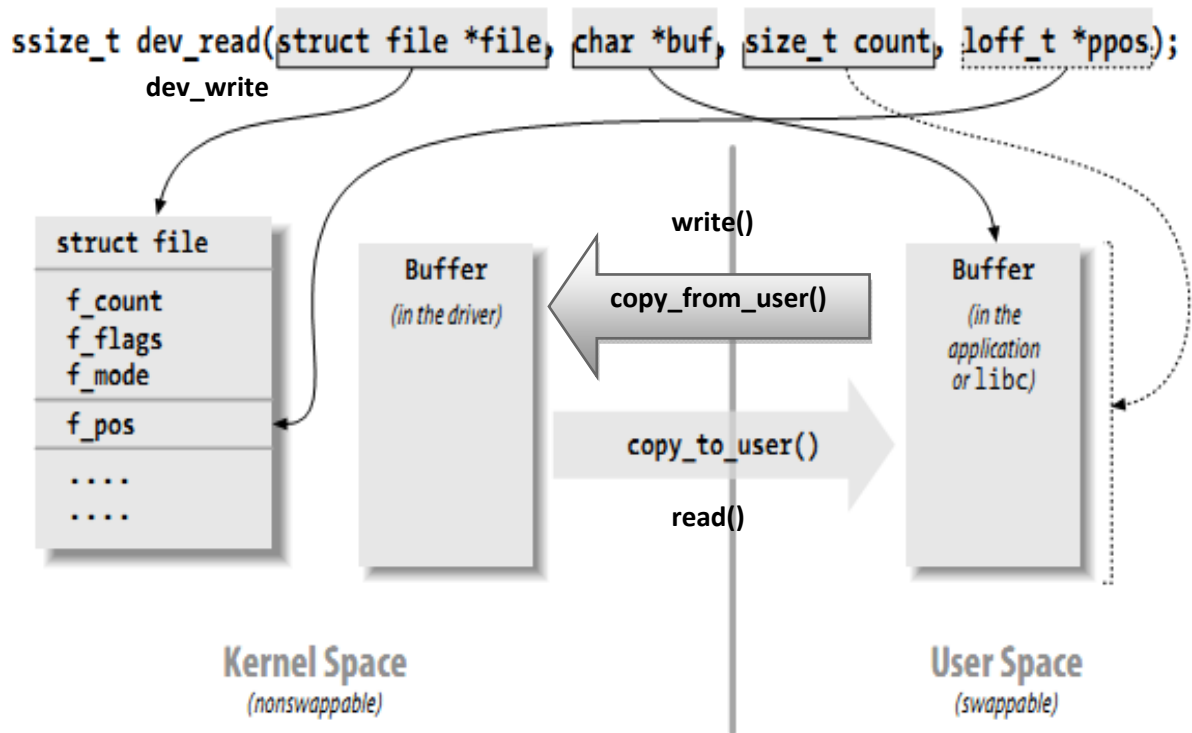
Những hàm giao tiếp này được linux định nghĩa trong thư viện `<asm/uaccess.h>`. Các hàm này là:

- `unsigned long copy_to_user (void __user *to, const void *from, unsigned long count);` Công dụng của hàm là chép dữ liệu từ *kernel space* sang *user space*; Trong đó, `void __user *to` là con trỏ đến vùng nhớ trong *user space* lưu dữ liệu chép từ *kernel space*; `const void *from` là con trỏ chứa dữ liệu trong *kernel space* muốn chép qua *user space*; `unsigned long count` là số bytes dữ liệu muốn chép.
- `unsigned long copy_from_user (void *to, const void __user *from, unsigned long count);` Công dụng của hàm là chép dữ liệu từ *user space* sang *kernel space*; Trong đó, `void *to` là con trỏ đến vùng nhớ trong *kernel space* lưu dữ liệu chép từ *user space*; `const void __user *from` là con trỏ chứa dữ liệu trong *user space* muốn chép qua *kernel space*; `unsigned long count` là số bytes dữ liệu muốn chép.

Hai hàm điều có giá trị trả về kiểu `unsigned long`. Trước khi truyền dữ liệu chúng kiểm tra giá trị con trỏ vùng nhớ có hợp lệ hay không. Nếu không hợp lệ, thì quá trình truyền dữ liệu không thể thực hiện và hàm sẽ trả về mã lỗi `-EFAULT`. Nếu trong quá trình truyền dữ liệu, giá trị con trỏ cập nhật bị lỗi, thì quá trình truyền dừng ngay tại thời điểm

đó, và như thế chỉ một phần dữ liệu được chép. Phần dữ liệu truyền thành công sẽ được thông báo trong giá trị trả về dưới dạng số byte. Chúng ta căn cứ vào giá trị này để truyền lại thông tin nếu cần thiết. Nếu không có lỗi xảy ra, hàm sẽ trả về giá trị 0.

Chúng ta sẽ minh họa nguyên tắc hoạt động của hàm `read()` và `write()` thông qua sơ đồ hình x\_y:



Chúng ta thấy trong sơ đồ trên có hai không gian vùng nhớ đệm, vùng nhớ đệm bên *user space* và vùng nhớ đệm bên *kernel space*. Chương trình trong *user space* gọi hàm `read()` hay hàm `write()` thì driver chạy trong *kernel space* sẽ tiến hành thực hiện những thao tác chép dữ liệu bằng cách sử dụng hai hàm `copy_to_user()` hay `copy_from_user()` để giao tiếp trao đổi thông tin giữa hai vùng đệm này. Đồng thời cập nhật những thông tin cần thiết phục vụ theo dõi quá trình truyền dữ liệu. Công việc này được thực hiện bởi người lập trình driver.

### **III. Giao diện `ioctl()`:**

Trên đây chúng ta đã tìm hiểu hai giao diện `read()` và `write()` dùng trao đổi dữ liệu qua lại giữa *user space* và *kernel space*, hay nói đúng hơn là giữa driver thiết bị vật lý và chương trình ứng dụng. Thế nhưng trong thực tế, ngoài việc trao đổi dữ liệu, một driver thiết bị còn phải thực hiện nhiều công việc khác ví dụ: cập nhật các thông số giao tiếp (tốc độ *baund*, số bits dữ liệu truyền nhận, ...) của thiết bị vật lý, điều khiển trực tiếp các thanh ghi lệnh phần cứng, đóng mở cổng giao tiếp, ... Linux cung cấp cho một giao diện chuẩn khác phục vụ cho tất cả các thao tác điều khiển trên, đó là giao diện `ioctl()`.

Như vậy, `ioctl()` có thể thực hiện những chức năng của hàm `read()` và `write()`, thế nhưng chỉ hiệu quả trong trường hợp số lượng dữ liệu cần truyền không cao, mang tính chất rời rạc. Trong trường hợp dữ liệu truyền theo từng khối, liên tục thì sử dụng hàm `read()` và `write()` là một lựa chọn khôn ngoan.

#### **a. Cấu trúc lệnh:**

Hàm `ioctl()` có cấu trúc như sau:

```
#include <linux/ioctl.h>

int (*ioctl) (struct inode *inode, struct file *filp, unsigned int
cmd, unsigned long arg);
```

#### **b. Giải thích:**

Như trên đã phân tích, tất cả những giao diện trong *character device driver* đều không có những chức năng cụ thể. Những tham số lệnh chỉ mang tính tổng quát, người lập trình driver phải hiểu rõ ý nghĩa từng tham số sau đó sẽ định nghĩa cụ thể chức năng từng tham số này này để đưa vào sử dụng tùy theo yêu cầu trong thực tế. Sau đây chúng ta sẽ tìm hiểu các tham số trong hàm `ioctl`.

- `struct inode *inode` và `struct file *filp` là con trỏ inode structure và *file structure* tương ứng với từng thiết bị được mở trong chương trình. Hai tham số này đều chứa trong số mô tả tập tin (*file descriptor*) gọi tắt là *fd*, được trả về khi thực hiện mở thiết bị thành công bởi hàm `open()`. (*\*\*Theo nguyên tắc, trước khi thực hiện giao tiếp với các thiết bị trong hệ thống linux, chúng ta phải truy cập đến cấu trúc inode của tập tin thiết bị trên bộ nhớ, kích hoạt khởi động thiết bị bằng hàm `open()`, hàm này sẽ quy định chế độ truy xuất tập tin thiết bị này, sau khi kích hoạt thành công hàm `open()` sẽ trả về số mô tả tập tin *fd*, mang tất cả thông tin của thiết bị đang được mở trong hệ thống*).

- `unsigned int cmd`, là số `unsigned int` do người lập trình driver quy định. Mỗi lệnh trong `ioctl` được phân biệt nhau thông qua số `cmd` này.

Số `unsigned int cmd` có 16 bits được chia thành 2 phần. Phần đầu tiên có 8 bits MSBs được gọi là số “*magic*” dùng phân biệt lệnh của thiết bị này với thiết bị khác. Phần thứ hai có 8 bit LSBs dùng để phân biệt những lệnh khác nhau của cùng một thiết bị. Số định danh lệnh này tương tự như số định danh thiết bị, phải được định nghĩa duy nhất trong hệ thống linux. Để biết được số định danh lệnh nào còn trống, chúng ta sẽ tìm trong tập tin *Documentation/ioctl-number.txt*. Tập tin này chứa thông tin về cách sử dụng hàm `ioctl()` và thông tin về số định danh lệnh đã được sử dụng trong hệ thống linux. Nhiệm vụ của chúng ta là đối chiếu so sánh để loại trừ những số định danh này, tìm ra số định danh trống cho thiết bị của mình. Sau khi tìm ra khoảng số định danh lệnh cho mình, chúng ta nên ghi rõ khoảng định danh đó vào tập tin *ioctl-number.txt* để sau này thuận tiện cho việc lập trình driver mới.

Do hàm `ioctl()` tồn tại trong cả hai môi trường, giao diện trong *user space* và những thao tác lệnh hoạt động trong *kernel space* nên các số định danh lệnh này phải quy định chung trong cả hai môi trường. Đối với các chương trình *application* và *driver* có sử dụng giao diện `ioctl`, việc đầu tiên trong đoạn mã chương trình là định nghĩa số định danh lệnh, mỗi số định danh lệnh có một chế độ truy xuất dữ liệu khác nhau. Những chế độ này được giải thích trong tập tin tài liệu *Documentation/ioctl-number.txt*.

Số định danh thiết bị được định nghĩa theo dạng thức sau:

*/\*Định nghĩa số magic cho số định danh lệnh là 'B' trong bảng mã ascii là 66 thập phân\*/*

*/\*Phân định nghĩa số định danh thiết bị được đặt đầu mỗi chương trình hoặc bên trong một <tập tin.h> thư viện\*/*

```
#define IOC_GPIODEV_MAGIC 'B'
```

*/\*Định nghĩa lệnh thứ nhất tên GPIO\_GET, với mã số lệnh trong thiết bị là 10, chế độ truy xuất là \_IO\*/*

```
#define GPIO_GET _IO(IOC_GPIODEV_MAGIC, 10)
```

```
#define GPIO_SET _IO(IOC_GPIODEV_MAGIC, 11)
```

```
#define GPIO_CLEAR _IO(IOC_GPIODEV_MAGIC, 12)
```

```
#define GPIO_DIR_IN _IO(IOC_GPIODEV_MAGIC, 13)
```

```
#define GPIO_DIR_OUT _IO(IOC_GPIODEV_MAGIC, 14)
```

Cấu trúc định nghĩa số định danh lệnh có dạng:

<chế độ truy xuất lệnh>(<số magic thiết bị>, <số mã lệnh thiết bị>)

- <chế độ truy xuất lệnh>: `ioctl` có 4 trạng thái lệnh: (Quy định bởi 2 bits trạng thái)

\_IO lệnh `ioctl` không có tham số;

\_IOW lệnh `ioctl` chứa tham số dùng để nhận dữ liệu từ *user* (tương đương chức năng của hàm `copy_from_user()`);

\_IOR lệnh `ioctl` chứa tham số dùng để gửi dữ liệu sang *user* (tương đương chức năng của hàm `copy_to_user()`);

\_IOWR lệnh `ioctl` chứa tham số dùng cho hai nhiệm vụ, đọc và ghi dữ liệu của *user*.

- <số magic thiết bị>: Là số đặc trưng cho từng thiết bị do người dùng định nghĩa.

- <số mã lệnh thiết bị>: Là số đặc trưng cho từng lệnh trong mỗi thiết bị.

Căn cứ vào số định danh lệnh này, người lập trình *driver* sẽ lựa chọn lệnh nào sẽ được thực thi khi *user* gọi hàm `ioctl`. Công việc lựa chọn lệnh được thực hiện bằng cấu trúc `switch...case...` như sau:

Đây là đoạn chương trình mẫu cho hàm `ioctl()` được khai báo sử dụng trong driver `gpio_dev` mang tên `gpio_ioctl`;

```
static int
```



```
gpio_ioctl(struct inode * inode, struct file * file, unsigned int
cmd, unsigned long arg)
{
    int retval = 0;
    /*Thực hiện lựa chọn lệnh thực thi bằng lệnh switch với tham số lựa chọn là số định
danh lệnh cmd*/
    switch (cmd)
    {
        case GPIO_GET:
            /*Hàm thao tác cho lệnh GPIO_GET*/
            break;
        case GPIO_SET:
            /*Hàm thao tác cho lệnh GPIO_SET*/
            break;
        case GPIO_CLEAR:
            /*Hàm thao tác cho lệnh GPIO_CLEAR*/
            break;
        case GPIO_DIR_IN:
            /*Hàm thao tác cho lệnh GPIO_DIR_IN*/
            break;
        case GPIO_DIR_OUT:
            /*Hàm thao tác cho lệnh GPIO_DIR_OUT*/
            break;
        default:
            /*Trả về mã lỗi trong trường hợp không có lệnh thực thi phù hợp*/
            retval = -EINVAL;
            break;
    }
    return retval;
}
```

- unsigned long arg, là tham số lệnh tương ứng với số định danh lệnh, tham số này có vai trò là bộ nhớ đệm chứa thông tin từ user hay thông tin đến user tùy theo chế độ của lệnh được định nghĩa ở đầu chương trình driver. Trong trường hợp lệnh không có tham số, chúng ta không cần thiết phải sử dụng tham số này trong user, chương trình

trong `ioctl` vẫn thực thi mà không có lỗi. Trong trường hợp hàm cần nhiều tham số, chúng ta khai báo sử dụng tham số này như là một con trỏ dữ liệu.

### IV. Tổng kết:

Như vậy với những giao diện chính trong character device như `read()`, `write()` và `ioctl()`, `open()`, ... chúng ta có thể giao tiếp giữa *user* và *kernel* rất thuận lợi. Điều này cũng có nghĩa rằng: Giữa người sử dụng và thiết bị vật lý có thể giao tiếp trao đổi thông tin với nhau dễ dàng thông qua hoạt động của driver với sự hỗ trợ của giao diện. Bên cạnh những giao diện trên, *linux* còn hỗ trợ rất nhiều giao diện khác, ứng dụng của những giao diện này không nằm trong các driver được nghiên cứu trong giáo trình cho nên chúng sẽ không được đề cập đến. Đối với những driver lớn, sẽ có rất nhiều giao diện khác nhau xuất hiện, nhiệm vụ của chúng ta là tìm hiểu nguyên lý hoạt động của giao diện đó dựa vào những kiến thức đã học. Đây cũng chính là mục đích cuối cùng mà nhóm biên tập muốn thực hiện trong cuốn giáo trình này.

Trước khi bước vào viết một driver hoàn chỉnh, chúng ta sẽ tìm hiểu các cấu trúc để viết một driver đơn giản trong bài sau.

**BÀI 5****TRÌNH TỰ VIẾT  
CHARACTER DEVICE DRIVER**

Lập trình driver là một trong những công việc quan trọng mà một người lập trình hệ thống nhúng cần phải nắm vững. Để một chương trình ứng dụng hoạt động tối ưu, người lập trình phải biết phân công nhiệm vụ giữa *driver* và *application* sao cho hợp lý. Trong một ứng dụng điều khiển, nếu *driver* đảm trách nhiều chức năng thì chương trình trong *application* sẽ trở nên đơn giản, nhưng bù lại tài nguyên phần cứng sẽ dễ bị xung đột, không thuận lợi cho các *driver* khác dùng chung tài nguyên. Ngược lại nếu *driver* chỉ thực hiện những công việc rất đơn giản, thì chương trình trong *application* trở nên phức tạp, việc chuyển qua lại giữa các tiến trình trở nên khó khăn. Việc phân chia nhiệm vụ này đòi hỏi phải có nhiều kinh nghiệm lập trình thực tế với hệ thống nhúng mới có thể đem lại hiệu quả tối ưu cho hệ thống.

Với những kiến thức tổng quát đã trình bày trong những phần trước về *driver* mà chủ yếu là *character device driver*, chúng ta đã có những khái niệm ban đầu về vai trò của *driver* trong hệ thống nhúng, cách thức điều khiển thiết bị vật lý, các giao diện giao tiếp thông tin giữa *user space* và *kernel space*, ... trong phần này chúng ta sẽ đi vào các bước cụ thể để viết hoàn chỉnh một *character device driver*.

Công việc phát triển một ứng dụng điều khiển mới thường tiến hành theo những bước sau:

- Tìm hiểu nhu cầu điều khiển ngoài thực tế: Từ hoạt động thực tiễn hay đơn đặt hàng xác định yêu cầu, thời gian thực hiện, giá cả chi phí, ...
- Phân tích yêu cầu điều khiển: Từ những yêu cầu giới hạn về thời gian, chi phí, chất lượng, người lập trình tìm ra phương pháp tối ưu hóa hệ thống, lựa chọn công nghệ phù hợp, ...; Lập danh sách các yêu cầu cần thực hiện trong hệ thống điều khiển;
- Phân công nhiệm vụ điều khiển giữa *application* và *driver* để hệ thống hoạt động tối ưu;
- Lập trình *driver* theo những yêu cầu được lập;
- Lập trình *application* sử dụng *driver* hoàn tất chương trình điều khiển;

- Chạy kiểm tra độ tin cậy hệ thống;
- Giao cho khách hàng sử dụng, bảo trì sử chữa khắc phục lỗi khi thực thi;

Chúng ta đang tập trung nghiên cứu bước lập trình *driver*, khi đã có những yêu cầu cụ thể. Trong bước này, có rất nhiều cách thực hiện, tuy nhiên nhìn chung điều có những thao tác căn bản sau:

1. Viết lưu đồ hoặc máy trạng thái thực thi cho *driver*;
2. Lập trình mã lệnh;
3. Biên dịch *driver*;
4. Cài đặt vào hệ thống linux;
5. Viết chương trình ứng dụng kiểm tra hoạt động *driver*;
6. Nếu đạt yêu cầu gán cố định vào cấu trúc *kernel* linux, biên dịch lại nhân để *driver* hoạt động lâu dài trong hệ thống; Nếu không tiến hành chỉnh sửa, kiểm tra cho đến khi hoàn thiện;

Trong đó các bước 1, 5, 6 chúng ta đã có dịp nghiên cứu trong phần I lập trình nhúng căn bản hoặc trong các tài liệu chuyên ngành khác. Phần này chúng ta sẽ tìm hiểu chi tiết các bước 2, 3, và 4;

### **I. Lập trình mã lệnh trong *character driver*:**

Có rất nhiều cách viết *character driver* nhưng cũng giống như chương trình ứng dụng *application*, cũng có những cấu trúc chung, chuẩn, từ đó sẽ tùy biến theo từng yêu cầu cụ thể. Ở đây, chúng ta sẽ tìm hiểu hai dạng cấu trúc: Cấu trúc dạng 1, và cấu trúc dạng 2;

*Cấu trúc dạng 1*: Bao gồm những thao tác cơ bản nhất, nhưng đa số những thao tác kiểm tra lỗi, lấy số định danh lệnh, số định danh thiết bị, ... người lập trình *driver* phải tự mình thực hiện. Có một ưu điểm là người lập trình có thể tùy biến theo yêu cầu của ứng dụng, có khả năng phát triển *driver*. Nhưng bù lại, thời gian thực hiện hoàn chỉnh *driver* để có thể chạy trong linux lâu hơn.

*Cấu trúc dạng 2*: Những thao tác cơ bản trong việc thành lập các thông số cho *driver* như: Lấy số định danh lệnh, thiết bị, khai báo cài đặt *driver* vào hệ thống, ... được gói gọn trong một câu lệnh duy nhất. Ưu điểm là thời gian thực hiện hoàn chỉnh một *driver* nhanh hơn, đơn giản hơn, các dịch vụ kiểm tra lỗi được hỗ trợ sẵn, giúp tránh xảy ra lỗi trong quá trình sử dụng. Thế nhưng việc tùy biến của người dùng nằm trong một giới hạn cho phép.

Thông thường chúng ta nên dùng cấu trúc dạng 2 cho những *driver* đơn giản, và những khả năng trong cấu trúc này hầu hết cũng tương tự như trong cấu trúc dạng 1, cũng đủ cho chúng ta thực hiện tất cả những thao tác điều khiển khác nhau.

Sau đây chúng ta sẽ tìm hiểu chi tiết từng cấu trúc để có cái nhìn cụ thể hơn về những ưu và nhược điểm nêu trên. Các chương trình *driver* cũng có cấu trúc tương tự như các chương trình trong C, chỉ khác là chúng có thêm những hàm chuẩn định nghĩa riêng cho việc giao tiếp điều khiển giữa *user space* và *kernel space* và còn nhiều đặc điểm khác nữa, sẽ được chúng ta đề cập trong mỗi phần cấu trúc;

**a. Cấu trúc chương trình character device driver dạng 1:**

*/\*Bước 1: Khai báo thư viện cho các hàm dùng trong chương trình, thư viện dùng trong kernel space khác với thư viện dùng trong user space. Thư viện trong driver là những hàm, biến, hằng số, ... được định nghĩa sẵn và hầu hết được lưu trong thư mục linux/ trong cấu trúc mã nguồn của linux. Do đó khi khai báo, chúng ta phải chỉ rõ đường dẫn đến thư mục này \*/*

```
#include <linux/module.h>           //Thư viện thứ nhất trong thư mục linux/
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
.....
```

*/\*Bước 2: Định nghĩa những hằng số cần dùng trong chương trình driver\*/*

```
#define ...
...
```

*/\*Bước 3: Khai báo số định danh thiết bị (Device number); Cấu trúc (file structure); Cấu trúc (file operations); \*/*

```
int                device_devno; //Khai báo biến lưu số định danh thiết bị
struct cdev        device_cdev; //Khai báo biến lưu cấu trúc tập tin
struct file_operations device_fops; //Khai báo biến lưu cấu trúc lệnh
```

*/\*Bước 4: Khai báo những biến, cấu trúc cần dùng trong chương trình driver\*/*

```
unsigned int variable_0;           //Những biến này được định nghĩa tương tự
trong
unsigned long variable_1;          //mã lệnh C, và C++
struct clk *device_clock_0;        //
```

....

*/\*Bước 5: Khai báo, định nghĩa những hàm con, chương trình con được sử dụng trong driver\*/*

*/\*Chương trình con thứ nhất, có hai tham số parameter\_0 và parameter\_1 cùng kiểu int\*/*

```
void sub_program_0 (int parameter_0, int parameter_1) {
```

*/\*Những lệnh thao tác cho chương trình con\*/*

```
    lệnh_0;
```

```
    lệnh_1;
```

```
    lệnh_2;
```

```
}
```

*/\*Hàm thứ nhất, trả về kiểu int, hai tham số con trở parameter\_0 và parameter\_1 kiểu int\*/*

```
int func_0 (int *parameter_0, int *parameter_1) {
```

*/\*Các thao tác lệnh cho hàm\*/*

```
    lệnh_0;
```

```
    lệnh_1;
```

```
    lệnh_2;
```

```
}
```

*/\*Bước 6: Định nghĩa hàm init cho driver, hàm init là hàm được thực thi đầu tiên khi thực hiện cài đặt driver vào hệ thống linux bằng lệnh shell insmod driver\_name.ko\*/*

*/\*Hàm init có một vai trò quan trọng trong lập trình driver. Ban đầu hàm sẽ gọi thực thi các hàm xác định số định danh thiết bị; Cập nhật các thông số của cấu trúc tập tin, cấu trúc lệnh; Đăng ký các cấu trúc vào hệ thống linux; Khởi tạo các thông số điều khiển ban đầu cho các thiết bị ngoại vi mà driver điều khiển\*/*

*/\*Hàm init có kiểu dữ liệu trả về dạng int, static trong trường hợp này nghĩa là hàm init chỉ dùng riêng cho driver sở hữu\*/*

```
static int __init at91adc_init (void) {
```

*/\*Khai báo biến lưu giá trị trả về của hàm\*/*

```
    int ret;
```

*/\*Xác định số định danh động cho thiết bị, tránh trường hợp bị trùng khi cài đặt với các driver thiết bị khác đã có trong hệ thống\*/*

```
ret = alloc_chrdev_region ( &device_devno, //Chỉ đến địa chỉ biến lưu số
                                //định danh đầu tiên tìm được;
                                0, //Số Minor đầu tiên yêu cầu;
                                1, //Số thiết bị yêu cầu;
                                "device_name" ); //Tên thiết bị muốn đăng
                                ký;

/*Kiểm tra mã lỗi khi thực thi hàm alloc_chrdev_region()*/
if (ret < 0) {
    //In thông báo lỗi khi thực hiện xác định số định danh thiết bị;
    printk (KERN_INFO "Device_name: Device number allocate
fail");
    ret = -ENODEV; //Trả về mã lỗi cho biến ret;
    return ret; //Trả về mã lỗi cho hàm init;
}

/*Khởi tạo thiết bị với số định danh đã xác định, yêu cầu hệ thống dành một vùng
nhớ lưu driver thiết bị sắp được tạo ra;*/
cdev_init ( &device_cdev, //Trỏ đến cấu trúc tập tin đã được định nghĩa;
            device_devno, //Số định danh thiết bị đã được xác định;
            1 ); //Số thiết bị muốn đăng ký vào hệ thống;

/*Chập nhật những thông tin cần thiết cho cấu trúc tập tin vừa được hệ thống
dành ra*/
device_cdev.owner = THIS_MODULE; /*Cập nhật người sở hữu cấu trúc tập
tin*/
device_cdev.ops = &device_fops; /*Cập nhật cấu trúc lệnh đã được định
nghĩa*/
/*Đăng ký thiết bị vào hệ thống */
ret = cdev_add ( &device_cdev, //Trỏ đến cấu trúc tập tin đã được khởi tạo;
                device_devno, //Số định danh thiết bị đã được xác định;
                1 ); //Số thiết bị muốn đăng ký;

/*Kiểm tra lỗi trong quá trình đăng ký thiết bị vào hệ thống*/
```

```
if (ret < 0)
{
    //In thông báo khi lỗi xuất hiện
    printk(KERN_INFO "Device: Device number allocation
    failed\n");
    ret = -ENODEV; //Trả về mã lỗi cho biến;
    return ret; //Trả về mã lỗi cho hàm;
}

/*Thực hiện các công việc khởi tạo thiết bị vật lý do driver điều khiển*/

...

//Trả về mã lỗi 0 khi quá trình thực thi hàm init không có lỗi;
return 0;
}
```

*/\*Bước 7: Khai báo và định nghĩa hàm exit, hàm exit là hàm được thực hiện ngay trước khi driver được tháo gỡ khỏi hệ thống bằng lệnh shell rmmod device.ko; Những tác vụ bên trong hàm exit được lập trình nhằm khôi phục lại trạng thái hệ thống trước khi cài đặt driver. Chẳng hạn như giải phóng vùng nhớ, timer, vô hiệu hóa các nguồn phát sinh ngắt, ...\*/*

```
static void __exit device_exit (void) {
    /*Các lệnh giải phóng vùng nhớ sử dụng trong driver*/
    ...
    /*Các lệnh giải phóng nguồn ngắt, timer, ...*/
    ...
    /*Tháo thiết bị ra khỏi hệ thống bằng hàm*/
    unregister_chrdev_region( device_devno, /*Số định danh đầu tiên nhóm
    thiết bị;*/
        1); //Số thiết bị cần tháo gỡ;
    /*In thông báo driver thiết bị đã được tháo ra khỏi hệ thống*/
    printk(KERN_INFO "device: Unloaded module\n");
}
```



*/\*Bước 8: Khai báo hàm open, đây là hàm được thực thi ngay sau khi lệnh open trong user space thực thi. Những lệnh chứa trong hàm này thông thường dùng cập nhật dữ liệu ban đầu cho chương trình driver, khởi tạo môi trường hoạt động ban đầu để hệ thống làm việc ổn định\*/*

```
static int device_open (struct inode *inode, struct file *filp)
{
    /*Nơi lập trình các lệnh khởi tạo hệ thống*/
    return 0;
}
```

*/\*Bước 9: Khai báo và định nghĩa hàm release, đây là hàm được thực thi ngay trước khi driver bị đóng bởi hàm close trong user space\*/*

```
static int device_release (struct inode *inode, struct file *filp)
{
    /*Nơi thực thi những lệnh trước khi driver bị đóng, không còn sử dụng*/
    return 0;
}
```

*/\*Bước 10: Khai báo các hàm read(), write(), ioctl() trong hệ thống khi cần sử dụng\*/*

*/\*Hàm read() đọc thông tin từ driver qua chương trình trong user\*/*

```
static ssize_t device_read (struct file *filp, char __iomem *buf,
size_t bufsz, loff_t *f_pos) {
    /*Định nghĩa các tác vụ hoạt động trong hàm read, để truy cập lấy thông tin từ thiết bị vật lý*/
    ...
}
```

*/\*Hàm write() ghi thông tin từ user qua driver\*/*

```
static ssize_t device_write (struct file *filp, char __iomem *buf,
size_t bufsz, loff_t *f_pos) {
    /*Định nghĩa các tác vụ hoạt động trong hàm write(), để truy cập lấy thông tin từ chương trình ứng dụng trong user*/
    ...
}
```

*/\*Hàm ioctl định nghĩa các trạng thái lệnh thực thi theo số định danh lệnh từ chương trình ứng dụng bên user\*/*

*static int*

```
gpio_ioctl (struct inode * inode, struct file * file, unsigned int  
cmd, unsigned long arg) {
```

*/\*Khai báo biến lưu mã lỗi trả về cho hàm giao diện ioctl()\*/*

```
int retval = 0;
```

*/\*Chia trường hợp thực thi lệnh\*/*

```
switch (cmd) {
```

```
    case LENH_0:
```

*/\*Lệnh 0 được thực thi\*/*

```
    break;
```

```
    case LENH_1:
```

*/\*Lệnh 1 được thực thi\*/*

```
    break;
```

```
    default:
```

*/\*Có thể thông báo, không có lệnh nào để thực thi\*/*

```
    retval = -EINVAL;
```

```
    break;
```

```
}
```

```
    return retval;
```

```
}
```

*/\*Bước 11: Gán các con trỏ hàm giao diện chuẩn vào cấu trúc lệnh file structure\*/*

```
struct file_operations gpio_fops = {
```

```
    .read      = device_read,
```

```
    .write     = device_write,
```

```
    .ioctl     = device_ioctl,
```

```
    .open      = device_open,
```

```
    .release   = device_release,
```

```
};
```

*/\*Bước 12: Gán các hàm exit và init vào hệ thống driver\*/*

```
module_init (device_init); /*Hàm device_init() thực hiện khi driver được cài đặt*/
```

```
module_exit (device_exit); /*Hàm device_exit() thực hiện khi driver được gỡ bỏ*/
```

*/\*Bước 13: Khai báo các thông tin liên quan đến driver\*/*

```
MODULE_AUTHOR("Tên người viết driver");
```

```
MODULE_DESCRIPTION("Mô tả nhiệm vụ của driver");
```

```
MODULE_LICENSE("GPL"); //Bản quyền của driver là GPL
```

***b. Cấu trúc chương trình character device driver dạng 2:***

Cấu trúc chương trình *character device driver* dạng 2 cũng tương tự như dạng 1, nhưng điểm khác biệt là kỹ thuật tạo và đăng ký thiết bị mới vào hệ thống. Để thuận tiện cho việc tham khảo chúng tôi không ngại nhắc lại những bước tương tự và giải thích những điều mới khi tiếp xúc với câu lệnh.

*/\*Bước 1: Khai báo những thư viện cần thiết cho các lệnh dùng trong chương trình driver\*/*

```
#include <linux/module.h>
```

```
#include <linux/errno.h>
```

```
#include <linux/init.h>
```

```
#include <asm/uaccess.h>
```

```
#include <asm/atomic.h>
```

```
#include <linux/miscdevice.h>
```

*/\*Bước 2: Định nghĩa những hằng số cần dùng trong chương trình driver\*/*

```
#define ...
```

```
...
```

*/\*Bước 3: Khai báo số định danh thiết bị, các biến phục vụ đồng bộ hóa dữ liệu\*/*

*/\*Biến lưu số major của thiết bị\*/*

```
static int dev_major;
```

*/\*Khai báo biến atomic\_t dùng đồng bộ hóa tài nguyên driver thiết bị\*/*

*/\*Cách đồng bộ hóa tài nguyên của thiết bị:*

*Trước khi mở thao tác với driver thiết bị, kỹ thuật atomic sẽ kiểm tra xem thiết bị đã được mở hay chưa, nếu thiết bị đã được mở (nhận biết bằng số counter) thì không cho*

phép truy cập đến thiết bị này, thông báo lỗi cho người sử dụng. Ngược lại, cho phép mở thiết bị, thực thi những câu lệnh tiếp theo trong hệ thống. Cụ thể là:

- Ban đầu cho biến kiểu `atomic_t` bằng 1, nghĩa là thiết bị chưa được mở;
- Khi thiết bị được mở, chúng ta giảm biến kiểu `atomic_t` một đơn vị, lúc này giá trị biến kiểu `atomic_t` bằng 0. Trong quá trình mở, hay đóng điều phải có sự so sánh biến `atomic_t`.
- Chỉ đóng driver thiết bị khi biến kiểu `atomic_t` bằng 1. Chỉ mở khi biến kiểu `atomic_t` bằng 0.
- Đóng thiết bị, thực hiện tăng biến kiểu `atomic_t`. Mở thiết bị, thực hiện giảm biến kiểu `atomic_t`.\*/

```
static atomic_t device_open_cnt = ATOMIC_INIT(1);
```

*/\*Bước 4: Khai báo các biến dùng trong chương trình driver\*/*

...

*/\*Bước 5: Khai báo và định nghĩa các hàm, chương trình con cần sử dụng trong quá trình lập trình driver\*/*

...

*/\*Bước 6: Khai báo và định nghĩa hàm open, là hàm được thực thi khi thực hiện lệnh open() trong user application\*/*

```
static int device_open (struct inode *inode, struct file *file) {
```

*/\*Khai báo biến lưu mã lỗi trả về\*/*

```
int result = 0;
```

*/\*Khai báo biến lưu số minor của thiết bị, đồng thời cập nhật số minor của thiết bị trong hệ thống\*/*

```
unsigned int device_minor = MINOR (inode->i_rdev);
```

*/\*Thực hiện kiểm tra biến `atomic_t` trong quá trình mở thiết bị nhiều lần\*/*

```
if (!atomic_dec_and_test(&device_open_cnt)) {
```

*/\*Tăng biến kiểu `atomic_t`\*/*

```
atomic_inc(&device_open_cnt);
```

*/\*In ra thông báo thiết bị muốn mở đang bận\*/*

```
printk(KERN_ERR DRVNAME ": Device with minor ID %d already in use\n", dev_minor);
```

*/\*Trả về mã lỗi bận\*/*

```
return -EBUSY;
```

```
    }  
  
    /*Thực hiện những công việc tiếp theo nếu mở tập tin thiết bị thành công*/  
    ...  
  
    /*Trả về mã lỗi cuối cùng nếu quá trình thực thi không bị lỗi*/  
    return result;  
}  
  
/*Bước 7: Khai báo và định nghĩa hàm close(), được thực hiện khi thực thi hàm close  
trong user application*/  
static int device_close (struct inode *inode, struct file *file) {  
    /*Thực hiện đồng bộ hóa counter trước khi tăng*/  
    smp_mb_before_atomic_inc();  
    /*Tăng biến đếm atomic khi đóng tập tin thiết bị*/  
    atomic_inc(&device_open_cnt);  
    /*Trả về mã lỗi 0 cho hàm close()*/  
    return 0;  
}  
  
/*Bước 8: Khai báo và cập nhật cấu trúc lệnh file_operations cho thiết bị*/  
struct file_operations device_fops = {  
    /*Gán hàm device_read của thiết bị vào giao diện chuẩn, nếu sử dụng*/  
    .read = device_read,  
    /*Gán hàm device_write của thiết bị vào giao diện chuẩn, nếu sử dụng*/  
    .write = device_write,  
    /*Gán hàm device_open của thiết bị vào giao diện chuẩn open*/  
    .open = device_open,  
    /*Gán hàm device_release của thiết bị vào giao diện chuẩn release*/  
    .release = device_release,  
};  
  
/*Bước 9: Khai báo và cập nhật cấu trúc tập tin thiết bị theo kiểu miscdevice*/  
static struct misdevice device_dev = {  
    /*Thực hiện kỹ thuật lấy số minor động cho thiết bị bằng hằng số định nghĩa sẵn  
    trong thư viện linux/miscdevice.h*/  
    .minor = MISC_DYNAMIC_MINOR,  
    /*Cập nhật tên cho thiết bị*/
```

```
.name = "device",  
/*Cập nhật cấu trúc lệnh đã được định nghĩa*/  
.fops = device_fops,  
};  
  
/*Bước 10: Khai báo định nghĩa hàm init thực hiện khi cài đặt driver vào hệ thống*/  
/*Tương tự như trong dạng 1, tuy nhiên với kỹ thuật mới này, trong hàm init chúng ta  
không cần phải lấy số định danh thiết bị, khởi tạo vùng nhớ cho driver, cài đặt driver  
vào hệ thống. Tất cả những công việc này được thực hiện bởi một hàm duy nhất đó là  
misc_register()*/  
static int __init device_mod_init(void) {  
    /*Những lệnh cần thực hiện khi cài đặt driver vào hệ thống*/  
    ...  
    /*Thực hiện đăng ký driver vào hệ thống, kết hợp trả về mã lỗi*/  
    return    misc_register(&devive_dev);  
}  
  
/*Bước 11: Khai báo và định nghĩa hàm exit()*/  
static void __exit device_mod_exit (void) {  
    /*Thực hiện những công việc khôi phục hệ thống trước khi driver bị tháo gỡ*/  
    ...  
    /*Tháo gỡ thiết bị mang tên là device_dev*/  
    misc_deregister (&device_dev);  
}  
  
/*Bước 12: Gắn các hàm exit và init vào hệ thống driver*/  
module_init (device_mod_init);  
module_exit (device_mod_exit);  
  
/*Bước 13: Cập nhật các thông tin cá nhân cho driver*/  
MODULE_LICENSE("Bản quyền driver thông thường là GPL");  
MODULE_AUTHOR("Tên người viết driver");  
MODULE_DESCRIPTION("Mô tả khái quát thông tin về driver");
```

## **II. Biên dịch driver:**

Sau khi lập trình *driver*, công việc tiếp theo là biên dịch *driver* biến tập tin mã nguồn C thành tập tin ngôn ngữ máy *driver* trước khi cài đặt vào hệ điều hành. Sau đây là các bước biên dịch *driver*.

Biên dịch *driver* có 2 dạng, đầu tiên biên dịch *driver* khi tập tin mã nguồn *driver* thuộc một bộ phận trong cấu trúc mã nguồn mở của *kernel*, khi đó *driver* được biên dịch cùng lúc với *kernel*, cách này chỉ áp dụng khi *driver* đã hoạt động ổn định, hơn nữa chúng ta không cần cài đặt lại *driver* khi khởi động lại hệ thống. Phương pháp thứ hai là biên dịch *driver* khi nó nằm ngoài cấu trúc mã nguồn mở của *kernel*, *driver* có thể được biên dịch trong khi *kernel* đang chạy, ưu điểm của phương pháp này là thời gian thực hiện nhanh, thích hợp cho việc thử nghiệm *driver* mới, thế nhưng mỗi lần hệ thống khởi động lại, *driver* sẽ bị mất do đó phải cài đặt lại khi khởi động. Khi *driver* đã hoạt động ổn định chúng ta mới biên dịch *driver* theo cách 1.

Biên dịch *driver* hoàn toàn khác với biên dịch chương trình ứng dụng. Chương trình ứng dụng có những thư viện chuẩn trong hệ thống *linux*, nên khi biên dịch ta chỉ việc chép tập tin chương trình vào trong một thư mục bất kỳ trong cấu trúc *root file system* và gọi lệnh biên dịch. Nhưng đối với *driver*, những thư viện sử dụng không nằm sẵn trong hệ thống, mà nằm trong cấu trúc mã nguồn mở của *kernel*. Vì thế trước khi biên dịch *driver* chúng ta phải giải nén tập tin mã nguồn mở của *kernel* vào cấu trúc *root file system*. Sau đó tạo tập tin *Makefile* để dùng lệnh *make* trong *shell* biên dịch *driver*. Cấu trúc *Makefile* đã được hướng dẫn kỹ trong phần lập trình hệ thống nhúng căn bản. Trong phần này chúng ta chỉ tạo ra *Makefile* với nội dung cần thiết để có thể biên dịch được *driver*.

Biên dịch *driver* được tiến hành theo các bước sau:

1. Chép tập tin *driver* mã nguồn C vào thư mục nào đó trong cấu trúc *root file system*.
2. Tạo tập tin có tên *Makefile* nằm trong cùng thư mục với tập tin *driver* mã nguồn C. Tập tin *Makefile* có nội dung như sau:

```
/*Thông báo cho trình biên dịch biết loại chip mà driver sẽ cài đặt*/  
export ARCH=arm  
/*Khai báo chương trình biên chéo là những tập tin có tên đầu tiên là arm-  
none-linux-gnueabi-...*/
```

```
export CROSS_COMPILE=arm-none-linux-gnueabi-
```

*/\*Tại đây tập tin mã nguồn driver sẽ được biên dịch thành tập tin .ko, có thể cài đặt vào linux\*/*

```
obj-m += <tên tập tin driver mã nguồn C>.o
```

*/\*Tùy chọn all, thực hiện chuyển đến cấu trúc mã nguồn mở của kernel, tại đây driver sẽ được biên dịch thông qua lệnh modules \*/*

```
all:
```

```
make -C <đường dẫn đến thư mục chứa cấu trúc mã nguồn mở của kernel> M=$(PWD) modules
```

*/\*Tùy chọn clean, thực hiện chuyển đến cấu trúc mã nguồn mở của kernel, thực hiện xóa những tập tin .o, .ko được tạo thành trong lần biên dịch trước\*/*

```
clean:
```

```
make -C / đường dẫn đến thư mục chứa cấu trúc mã nguồn mở của kernel> M=$(PWD) clean
```

3. Tại thư mục chứa mã nguồn *driver*, dùng lệnh *shell*: `make clean all`

Lúc này hệ thống linux sẽ xóa những tập tin có đuôi .o, .ko, ... tạo thành trong những lần biên dịch trước. Tiếp theo, biên dịch tập tin mã nguồn driver thành tập tin .ko, tập tin này có thể được cài đặt vào hệ thống linux thông qua những thao tác sẽ được hướng dẫn trong phần sau.

### **III. Cài đặt driver vào hệ thống linux:**

Trong phần I, chúng ta đã trình bày hai cấu trúc chung để viết hoàn chỉnh một *character device driver*, mỗi cấu trúc đều có những ưu và nhược điểm riêng. Những ưu và nhược điểm này còn được biểu hiện rõ trong việc cài đặt *driver* vào hệ thống.

Sau khi đã biên dịch thành công *driver*, xuất hiện tập tin .ko trong thư mục chứa tập tin mã nguồn, chúng ta dùng những câu lệnh trong *shell* để hoàn tất công đoạn cuối cùng đưa driver vào hoạt động trong hệ thống, tiến hành kiểm tra chức năng. Tùy theo kỹ thuật áp dụng lập trình *driver* mà sẽ có những thao tác cài đặt khác nhau:

#### **1. Cài đặt driver khi áp dụng cấu trúc dạng 1:**

Tiến hành theo các bước sau:

- Di chuyển đến thư mục chứa tập tin .ko vừa biên dịch xong;
- Tại dòng lệnh shell, thực thi: `insmod <tên driver>.ko;`



- Vào tập tin */proc/devices* tìm tên thiết bị vừa cài đặt vào hệ thống, xác định số *Major* và số *Minor* động của thiết bị;
- Sử dụng câu lệnh trong shell: `mknod /dev/<tên thiết bị> c <Số Major> <Số Minor>`, tạo inode trong thư mục */dev/*, làm tập tin thiết bị sử dụng cho những chương trình ứng dụng;

### 2. Cài đặt driver khi áp dụng cấu trúc dạng 2:

- Di chuyển đến thư mục chứa tập tin *.ko* vừa biên dịch xong;
- Tại dòng lệnh *shell*, thực thi lệnh: `insmod <tên driver>.ko`;

Khi đó, cấu trúc inode tự động được tạo ra trong thư mục */dev/* liên kết với số định danh thiết bị lưu trong tập tin */proc/devices* mà không cần phải thông qua những câu lệnh shell khác như trong cách 1. Như vậy, với cấu trúc dạng 2 thời gian cài đặt driver vào hệ thống được rút gọn đáng kể.

### IV. Tổng kết:

Từ những kiến thức lý thuyết nền tảng trong những bài trước, chúng ta đã rút ra được những bước tổng quát để lập trình hoàn chỉnh một *character device driver*. Có nhiều cách để viết ra một *character driver*, trong bài này chỉ đề cập 2 cách căn bản làm nền tảng cho các bạn nghiên cứu thêm những cách khác hiệu quả hơn ngoài thực tế.

Trong bài sau, chúng ta sẽ thực hành viết một *character device driver* mang tên là *helloworld*. Driver này sẽ áp dụng tất cả những kỹ thuật đã được học. Nếu cần thiết, các bạn có thể xem lại lý thuyết cũ trước khi qua bài sau để nắm được những vấn đề cốt lõi trong lập trình *driver*.

**BÀI 6****HELLO WORLD DRIVER****I. Mở đầu:**

Đến đây, chúng ta đã có được những kiến thức gần như đầy đủ để tự mình viết một *character device driver*. Các bạn đã biết thế nào là *character driver*, số định danh lệnh, số định danh thiết bị, cấu trúc *inode*, *file structure*,... cũng như những lệnh khởi tạo và cập nhật chúng. Với những yêu cầu của từng lệnh, chúng ta đã rút ra được hai cấu trúc chung khi muốn viết một *driver* hoàn chỉnh, đã được tìm hiểu trong bài trước. Tùy vào từng cấu trúc mà có các bước cài đặt *driver* khác nhau. Thế nhưng, chúng ta chỉ mới dừng lại các thao tác trên giấy, chưa thực sự lập trình ra được một *driver* nào. Trong bài này, chúng ta sẽ viết một dự án có tên *helloworld* nhằm mục đích thực tế hóa những thao tác lệnh đã được trình bày trong phần *driver*.

Với mục đích là đem những thao tác lệnh đã được học vào thực tế chương trình, dự án này bao gồm có hai thành phần cần hoàn thành đó là *driver* và *Application*. *Driver* trong dự án sẽ áp dụng 3 giao diện chính dùng để trao đổi thông tin dữ liệu và điều khiển qua lại giữa hai lớp *user* và *kernel*, đó là các giao diện hàm *read()*, *write()* và *ioctl()* cùng với các hàm đóng mở *driver* như *open()* hay *close()*. Chương trình ứng dụng (*Application*) sẽ áp dụng những hàm về truy cập tập tin, ... để truy xuất những thông tin trong *driver*, hiển thị cho người dùng. Ngoài ra, *driver* và *application* đều có nhiệm vụ xuất thông tin có liên quan để người lập trình biết môi trường nào đang thực thi.

Theo những yêu cầu trên, đầu tiên chúng ta sẽ phân công tác vụ cho *driver* và *application* trong mục II sau đó lập trình theo những tác vụ đã phân công trong mục III. Người lập trình sẽ biên dịch, cài đặt chương trình vào hệ thống linux, thực thi và quan sát kết quả. Từ đó giải thích rút ra nhận xét.

**II. Phân công tác vụ giữa *driver* và *application*:**

*helloworld* là một dự án bao quát tất cả những kỹ thuật lập trình *driver* được nghiên cứu trong những nội dung trước. Nhiệm vụ chính là dùng hàm *printf()* và *printk()* xuất thông tin ra màn hình hiển thị theo một quy luật phù hợp, từ đó người lập trình có thể hiểu nguyên lý hoạt động của từng hàm sử dụng để áp dụng vào trường hợp khác. *Driver* và *Application* đều có nhiệm vụ riêng, tùy vào từng giao diện sử dụng mà yêu cầu

của từng ví dụ sẽ khác nhau, sao cho toát lên được ý nghĩa cốt lõi của giao diện hàm. Sau đây là chi tiết từng yêu cầu của dự án *helloworld*.

### **II.1. Driver:**

#### *a. Giao diện read():*

Giao diện `read()` được sử dụng để chép thông tin từ *kernel space* sang *user space*. Thông tin được lưu trong *driver* là số khởi tạo ban đầu khi *driver* được mở chứa trong một biến cục bộ. Tại thời điểm gọi giao diện `read()`, thông tin này sẽ được chép qua *user space*, chứa trong biến khai báo trong *user application* thông qua hàm `copy_to_user()`.

Sau khi chép thông tin cho *user*, *driver* thông báo cho người lập trình biết quá trình chép thành công hay không.

#### *b. Giao diện write():*

Giao diện `write()` dùng để chép thông tin từ *user space* qua *kernel space*. Thông tin từ *user* là dữ liệu kiểu số do người dùng nhập vào chuyển qua *kernel* thông qua giao diện `write()` lưu vào một biến trong *kernel*, biến này cũng là nơi lưu thông tin được chuyển sang *user* khi giao diện `read()` được gọi.

Sau khi nhận dữ liệu từ *user*, *driver* thông báo cho người lập trình quá trình chép thành công. Ngược lại sẽ trả về mã lỗi.

#### *c. Giao diện ioctl():*

`ioctl()` là một giao diện hàm đa chức năng, nghĩa là chỉ cần một dạng câu lệnh mà có thể thực hiện được tất cả những chức năng khác. Để thể hiện được những chức năng này của hàm, chúng ta lập trình một ví dụ sau:

Xây dựng giao diện `ioctl()` thành một hàm toán học, thực hiện các phép toán cộng, trừ, nhân, chia. Các tham số được truyền từ *user*, sau khi tính toán xong, kết quả được gửi ngược lại *user*. Các phép toán được lựa chọn thông qua các số định danh lệnh.

### **II.2. Application:**

Chương trình trong *user*, sử dụng kỹ thuật tùy chọn trong hàm `main()` để kiểm tra tất cả những chức năng do chương trình trong *driver* hỗ trợ. Mỗi chức năng sẽ được thực hiện do người sử dụng lựa chọn.

Trước khi đi vào viết chương trình cụ thể, chúng ta sẽ tìm hiểu hệ điều hành linux thực hiện tùy chọn trong hàm `main()` như thế nào:

Hàm `main()` là hàm được thực hiện đầu tiên khi chương trình ứng dụng được gọi thực thi. Từ hàm này, người lập trình sẽ tiến hành tất cả những chức năng như tạo lập tiến trình, tuyến, gọi các chương trình con, ... Thông thường hàm `main()` được khai báo như sau:

```
void main(void) {  
    /*Các lệnh do người lập trình định nghĩa*/  
}
```

Với cách khai báo này thì hàm `main()` không có tham số cũng như không có dữ liệu trả về.

Ngoài ra linux còn hỗ trợ cho người lập trình cách khai báo hàm `main()` khác có dạng như sau:

```
int main (int argc, char **argv) {  
    /*Các lệnh do người lập trình định nghĩa*/  
}
```

Với cách lập trình này hàm `main()` có thể được người sử dụng cung cấp các tham số trong khi gọi thực thi. Cú pháp cung cấp tham số trong câu lệnh `shell` như sau:

```
./<tên chương trình> <tham số 1> <tham số 2> <...> <tham số n>
```

Hàm `main()` có hai tham số:

- Tham số thứ nhất `int argc` là số `int` lưu số lượng tham số khai báo trong câu lệnh `shell` trên, bao gồm cả tham số đầu tiên là tên chương trình chứa hàm `main()`.
- Tham số thứ hai `char **argv` là mảng con trỏ lưu nội dung từng tham số nhập trong câu lệnh gọi chương trình thực thi trong `shell`;

Như vậy tên chương trình chứa hàm `main()` sẽ thuộc về tham số đầu tiên có nội dung lưu trong `argv[0]`. Tương tự `<tham số 1>` là tham số thứ hai chứa trong `argv[1]`, ...Tương tự cho các tham số khác. Tổng quát nếu có `n` tham số trong câu lệnh `shell` thì trong hàm `main` có: `argc = n+1; argv[0], argv[1], argv[2], ..., argv[n]` lưu nội dung của từng tham số.

Trong chương trình ứng dụng *user application* của dự án *helloworld*, hàm `main()` được khai báo có dạng tùy chọn như trên. Khi người dùng nhập: `"add"`, `"sub"`, `"mul"`, `"div"`, `"read"`, `"write"`, thì câu lệnh tương ứng sẽ thực hiện gọi các giao diện hàm cần thiết thực hiện chức năng cộng, trừ, nhân, chia, đọc và ghi được lập trình

trong *dirver*. Thao tác cụ thể sẽ được chúng tôi chú thích trong từng dòng lệnh của *dirver* và *application*.

### **III. Chương trình driver và application:**

#### **A. Chương trình driver:**

```
/*Chương trình driver mang tên helloworld_dev.c*/
/*Khai báo các thư viện cần thiết*/
#include <linux/module.h> /*dùng cho các cấu trúc module*/
#include <linux/errno.h> /*dùng truy xuất mã lỗi*/
#include <linux/init.h> /*dùng cho các macro module_init(), ...*/
#include <asm/uaccess.h> /*dùng cho các hàm copy_to(from)_user*/
#include <asm/atomic.h> /*dùng cho kiểm tra atomic*/
#include <linux/genhd.h> /*dùng cho các giao diện hàm*/
#include <linux/miscdevice.h> /*dùng cho kỹ thuật tạo device tự động*/
/*Khai báo tên cho device và driver*/
#define DRVNAME "helloworld_dev"
#define DEVNAME "helloworld"
/*Khai báo số type cho số định danh lệnh dùng trong ioctl() */
#define HELLOWORLD_DEV_MAGIC 'B' /*Số type là số 66 trong mã ascii*/
/*Số định danh lệnh cho hàm add*/
#define HELLOWORLD_ADD _IOWR(HELLOWORLD_DEV_MAGIC, 10,unsigned long)
/*Số định danh lệnh cho hàm sub*/
#define HELLOWORLD_SUB _IOWR(HELLOWORLD_DEV_MAGIC, 11,unsigned long)
/*Số định danh lệnh cho hàm mul*/
#define HELLOWORLD_MUL _IOWR(HELLOWORLD_DEV_MAGIC, 12,unsigned long)
/*Số định danh lệnh cho hàm div*/
#define HELLOWORLD_DIV _IOWR(HELLOWORLD_DEV_MAGIC, 13,unsigned long)

/*Biến lưu giá trị truyền sang user space khi gọi hàm read() từ user;
hay lưu giá trị nhận được từ user khi hàm write được gọi*/
static char helloworld_dev_number = 34;
/*Khai báo biến đếm atomic và khởi tạo*/
static atomic_t helloworld_open_cnt = ATOMIC_INIT(1);
/*Cấu trúc ioctl() thực hiện các phép toán add, sub, mul và div*/
static int
helloworld_ioctl(struct inode * inode, struct file * file, unsigned
int cmd, unsigned long arg[])
```

```
{    /*Biến lưu mã lỗi trả về của ioctl()*/
    int retval;
    /*Phân biệt các trường hợp lệnh khác nhau*/
    switch (cmd)
    {
    case HELLOWORLD_ADD:
        /*Thông báo quá trình tính toán bắt đầu*/
        printk ("Driver: Start ...\n");
        /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
        arg[2] = arg[0] + arg [1];
        /*In thông báo tính toán kết thúc*/
        printk ("Driver: Calculating is complete\n");
        break;

    case HELLOWORLD_SUB:
        /*Thông báo quá trình tính toán bắt đầu*/
        printk ("Driver: Start ...\n");
        /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
        arg[2] = arg[0] - arg [1];
        /*In thông báo tính toán kết thúc*/
        printk ("Driver: Calculating is complete\n");
        break;

    case HELLOWORLD_MUL:
        /*Thông báo quá trình tính toán bắt đầu*/
        printk ("Driver: Start ...\n");
        /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
        arg[2] = arg[0] * arg [1];
        /*In thông báo tính toán kết thúc*/
        printk ("Driver: Calculating is complete\n");
        break;

    case HELLOWORLD_DIV:
        /*Thông báo quá trình tính toán bắt đầu*/
        printk ("Driver: Start ...\n");
        /*Thực hiện tính toán, chép qua user qua biến arg[2]*/
        arg[2] = arg[0] / arg [1];
```

```
        /*In thông báo tính toán kết thúc*/
        printk ("Driver: Calculating is complete\n");
        break;

default:
        /*Thông báo lỗi cho người sử dụng, không có lệnh hỗ trợ*/
        printk ("Driver: I don't have this operation!\n");
        retval = -EINVAL;
        break;
}

/*Trả về mã lỗi cho hàm ioctl() */
return retval;
}

/*Khai báo và định nghĩa hàm giao diện read*/
static ssize_t helloworld_read (struct file *filp, char __iomem buf[],
size_t bufsz, loff_t *f_pos)
{
        /*Khai báo con trỏ đệm dữ liệu cần truyền qua user */
        int *read_buf;
        /*Biến lưu kích thước đã truyền thành công*/
        int read_size = 0;
        /*Cập nhật con trỏ đệm phát*/
        read_buf = &helloworld_dev_number;
        /*Gọi hàm truyền dữ liệu từ con trỏ đệm phát sang user, kiểm tra
        mã lỗi nếu đúng trả về kích thước phát thành công*/
        if (copy_to_user (buf, read_buf, bufsz) != 0 ) return -EFAULT;
        else read_size = bufsz;
        /*Thông báo số bytes truyền thành công qua user*/
        printk ("Driver: Has sent %d byte(s) to user.\n", read_size);
        /*Trả về số byte dữ liệu truyền thành công qua user*/
        return read_size;
}

/*Định nghĩa giao diện hàm write()*/
static ssize_t helloworld_write (struct file *filp, char __iomem
buf[], size_t bufsz, loff_t *f_pos)
{
        /*Khai báo biến đệm nhận dữ liệu từ user*/
        char write_buf[1];
```

```
/*Khai báo biến lưu kích thước nhận thành công*/
int write_size = 0;
/*Gọi hàm lấy dữ liệu từ user, có kiểm tra lỗi, đúng sẽ trả về số
byte dữ liệu nhận thành công*/
if (copy_from_user (write_buf, buf, bufsize) != 0) return -
EFAULT; else write_size = bufsize;
/*Cập nhật thông tin cho lần đọc tiếp theo*/
helloworld_dev_number = write_buf[0];
/*In ra thông báo số byte dữ liệu nhận thành công từ user*/
printk ("Driver: Has received %d byte(s) from user.\n",
write_size);
/*Trả về số byte nhận thành công cho hàm write()*/
return write_size;
}

/*Khai báo giao diện open()*/
static int
helloworld_open(struct inode *inode, struct file *file) {
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&helloworld_open_cnt)) {
        atomic_inc(&helloworld_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d already in
use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

/*Khai báo giao diện hàm close*/
static int
helloworld_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&helloworld_open_cnt);
}
```



```
    return 0;
}

/*Gán các giao diện hàm vào cấu trúc file operation*/
struct file_operations helloworld_fops = {
    .ioctl      = helloworld_ioctl,
    .read       = helloworld_read,
    .write      = helloworld_write,
    .open       = helloworld_open,
    .release    = helloworld_close,
};

/*Gán cấu trúc file operation vào inode, cập nhật tên thiết bị, hiện
thị trong inode*/
static struct miscdevice helloworld_dev = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = "helloworld",
    .fops       = &helloworld_fops,
};

/*Khai báo và định nghĩa hàm thực hiện lúc cài đặt driver*/
static int __init
helloworld_mod_init(void)
{
    return misc_register(&helloworld_dev);
}

/*Khai báo định nghĩa hàm thực thi lúc tháo gỡ driver khỏi hệ thống*/
static void __exit
helloworld_mod_exit(void)
{
    misc_deregister(&helloworld_dev);
}

/*Gán hàm định nghĩa vào macro init và exit module*/
module_init (helloworld_mod_init);
module_exit (helloworld_mod_exit);

/*Cập nhật thông tin của driver, quyền sở hữu, tên người viết chương
trình*/
MODULE_LICENSE("GPL");
MODULE_AUTHOR("coolwarmboy");
MODULE_DESCRIPTION("Character device for helloworld");
```

### ***B. Chương trình application:***

```
/*Chương trình application mang tên helloworld_app.c*/  
/*Khai báo các thư viện cần thiết cho các hàm sử dụng trong chương trình*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <linux/ioctl.h>  
/*Định nghĩa các số định danh lệnh cho giao diện ioctl, các số định nghĩa này giống như trong driver*/  
#define HELLOWORLD_DEV_MAGIC 'B'  
/*Số định danh lệnh dùng cho lệnh add*/  
#define HELLOWORLD_ADD _IOWR(HELLOWORLD_DEV_MAGIC, 10, unsigned long)  
/*Số định danh lệnh dùng cho lệnh sub*/  
#define HELLOWORLD_SUB _IOWR(HELLOWORLD_DEV_MAGIC, 11, unsigned long)  
/*Số định danh lệnh dùng cho lệnh mul*/  
#define HELLOWORLD_MUL _IOWR(HELLOWORLD_DEV_MAGIC, 12, unsigned long)  
/*Số định danh lệnh dùng cho lệnh div*/  
#define HELLOWORLD_DIV _IOWR(HELLOWORLD_DEV_MAGIC, 13, unsigned long)  
/*Chương trình con in ra hướng dẫn cho người dùng*/  
void  
print_usage()  
{  
    printf("helloworld_app      add|sub|mul|div|read|write      arg_1  
    (arg_2)\n");  
    exit(0);  
}  
/*Hàm main thực thi khi chương trình được gọi từ shell, main khai báo theo dạng tùy chọn tham số*/  
int  
main(int argc, char **argv)  
{  
    /*Biến lưu số file description của driver khi được mở*/  
    int fd;  
    /*Biến lưu mã lỗi trả về cho hàm main*/  
    int ret = 0;
```

```
/*Bộ đệm dữ liệu nhận được từ driver*/
char read_buf[1];
/*Bộ đệm dữ liệu cần truyền sang driver*/
char write_buf[1];
/*Bộ đệm dữ liệu 2 chiều dùng trong hàm giao diện ioctl()*/
unsigned long arg[3];
/*Đầu tiên mở driver cần tương tác, quy định chế độ truy xuất
driver là đọc và ghi*/
if ((fd = open("/dev/helloworld", O_RDWR)) < 0)
{
    /*Nếu có lỗi trong quá trình mở thiết bị, in ra thông báo cho
    người dùng*/
    printf("Error whilst opening /dev/helloworld\n");
    /*Trả về mã lỗi cho hàm main*/
    return fd;
}
/*Liệt kê các trường hợp có thể xảy ra khi giao tiếp với người
dùng*/
if (argc == 4) {
    /*Trong trường hợp là các hàm toán học, dùng ioctl, kiểm
    tra các phép toán và thực hiện tính toán*/
    /*Trong trường hợp là phép toán cộng*/
    if (!strcmp(argv[1], "add")) {
        arg[0] = atoi(argv[2]);
        arg[1] = atoi(argv[3]);
        ioctl (fd, HELLOWORLD_ADD, arg);
        printf ("User: %ld + %ld = %ld\n", arg[0], arg[1],
        arg[2]);
    }
    /*Trong trường hợp là phép trừ*/
    else if (!strcmp(argv[1], "sub")) {
        arg[0] = atoi(argv[2]);
        arg[1] = atoi(argv[3]);
        ioctl (fd, HELLOWORLD_SUB, arg);
        printf ("User: %ld - %ld = %ld\n", arg[0], arg[1],
        arg[2]);
    }
}
```

```
    /*Trong trường hợp là phép nhân*/
    else if (!strcmp(argv[1], "mul")) {
        arg[0] = atoi(argv[2]);
        arg[1] = atoi(argv[3]);
        ioctl (fd, HELLOWORLD_MUL, arg);
        printf ("User: %ld x %ld = %ld\n", arg[0], arg[1],
            arg[2]);
    }
    /*Trong trường hợp là phép chia*/
    else if (!strcmp(argv[1], "div")) {
        arg[0] = atoi(argv[2]);
        arg[1] = atoi(argv[3]);
        ioctl (fd, HELLOWORLD_DIV, arg);
        printf ("User: %ld / %ld = %ld\n", arg[0], arg[1],
            arg[2]);
    }
    /*Trong trường hợp không có lệnh hỗ trợ, in ra hướng dẫn
    cho người dùng*/
    else {
        print_usage();
    }
}

/*Trong trường hợp là lệnh chép thông tin sang driver*/
else if (argc == 3) {
    if (!strcmp(argv[1], "write")) {
        write_buf[0] = atoi(argv[2]);
        printf ("User: has just sent number: %d\n",
            write_buf[0]);
        write(fd, write_buf, 1);
    } else {
        print_usage();
    }
}

/*Trong trường hợp lệnh đọc thông tin từ driver*/
else if (argc == 2) {
    if (!strcmp(argv[1], "read")) {
        read(fd, read_buf, 1);
    }
}
```

```
        printf ("User:  has  just  received  number:  %d\n",
        read_buf[0]);
    } else {
        print_usage();
    }
}
/*In ra hướng dẫn cho người dùng trong trường hợp không có lệnh
nào hỗ trợ*/
else {
    print_usage();
}
}
```

***Biên dịch chương trình driver:***

- Khởi tạo biến môi trường thêm vào đường dẫn đến thư mục chứa công cụ hỗ trợ biên dịch;
- Tạo tập tin có tên Makefile trong thư mục chứa chương trình driver như sau:

```
export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabi-
obj-m += helloworld_dev.o
all:
    #Tùy thuộc vào nơi chứa cấu trúc mã nguồn kernel mà chọn đường
    #dẫn phù hợp cho lệnh make all
    make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) modules
clean:
    #Tùy thuộc vào vị trí chứa cấu trúc mã nguồn kernel mà chọn đường
    #dẫn cho phù hợp với lệnh make clean
    make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) clean
```

- Trở đến thư mục chứa driver thực thi lệnh shell: `make clean all` lúc này tập tin `helloworld_dev.c` sẽ được biên dịch thành `helloworld_dev.ko`;
- Chép tập tin `helloworld.ko` vào kit nhúng cần cài đặt;
- Cài đặt driver vào hệ thống linux bằng lệnh shell: `insmod helloworld_dev.ko`;

Biên dịch chương trình application: Thực hiện theo các bước sau:

- Cài đặt biến môi trường trỏ vào nơi chứa công cụ hỗ trợ biên dịch, (Nếu đã làm trong khi biên dịch driver thì không cần làm bước này);
- Trong thư mục chứa mã nguồn chương trình helloworld\_app.c thực thi lệnh shell:  
`arm-none-linux-gnueabi-gcc helloworld_app.c -o helloworld_app`  
Lúc này tập tin helloworld\_app sẽ được tạo thành nếu chương trình helloworld\_app.c không có lỗi.
- Chép tập tin helloworld\_app đã biên dịch vào kit;
- Thực thi chương trình kiểm tra hoạt động của driver đã lập trình;

Sau đây là kết quả thực thi chương trình trong từng trường hợp:

- ✓ Trường hợp 1: Kiểm tra hoạt động của phép toán add trong driver;

```
./helloworld_app add 100 50
Driver: Start ...
Driver: Calculating is complete
User: 100 + 50 = 150
```

Chúng ta thấy driver đã hoạt động đúng chức năng của mình. Khi chương trình được gọi thực thi, nó gọi hàm ioctl thực hiện truyền hai tham số, kết quả được tính toán trong driver khi tính toán xong, nó gửi dữ liệu qua user. Cuối cùng, user xuất dữ liệu đã tính toán nhận được từ driver ra ngoài màn hình. Tương tự cho các phép toán khác như sau:

- ✓ Trường hợp 2: Kiểm tra hoạt động của phép toán sub trong driver;

```
./helloworld_app sub 100 50
Driver: Start ...
Driver: Calculating is complete
User: 100 - 50 = 50
```

- ✓ Trường hợp 3: Kiểm tra hoạt động của phép toán mul trong driver;

```
./helloworld_app mul 100 50
Driver: Start ...
Driver: Calculating is complete
User: 100 x 50 = 5000
```

- ✓ Trường hợp 3: Kiểm tra hoạt động của phép toán div trong driver;

```
./helloworld_app div 100 50
Driver: Start ...
Driver: Calculating is complete
```

```
User: 100 / 50 = 2
```

✓ Trường hợp 5: Kiểm tra đọc dữ liệu từ driver;

```
./helloworld_app read
```

```
Driver: Has sent 1 byte(s) to user.
```

```
User: has just receive number: 34
```

Khi chương trình application được gọi thực thi, với trường hợp lệnh read, hàm read được gọi thực thi yêu cầu dữ liệu từ driver, driver truyền một số lưu trong vùng nhớ của mình cho user application. User application nhận được thông tin, thông báo cho người dùng.

✓ Trường hợp 6: Kiểm tra ghi dữ liệu vào driver thông qua hàm write:

```
./helloworld_app write 100
```

```
User:Driver: Has received 1 byte(s) from user
```

```
has just sent number: 100
```

Có một vấn đề nảy sinh là, mặc dù thông tin được chuyển qua driver thành công nhưng kết quả in ra xảy ra một lỗi nhỏ về thứ tự in của những dòng thông tin. Nguyên nhân là trong khi user in thông báo, driver đã nhận được dữ liệu và xuất ra thông báo của mình. Vì thế sau khi driver in xong thông báo, user mới tiếp tục thực hiện nhiệm vụ của mình. Như thế chúng ta cần phải chú ý đến vấn đề đồng bộ dữ liệu giữa driver và application để tránh trường hợp này xảy ra.

#### IV. Tổng kết:

Như vậy chúng ta đã hoàn thành công việc viết hoàn chỉnh một *driver* đơn giản dựa vào các bước mẫu trong bài học trước. Các bạn cũng đã hiểu nguyên lý hoạt động của hàm *main* có tham số, cách lựa chọn tham số hoạt động trong từng trường hợp cụ thể theo yêu cầu.

Với những thao tác trên, các bạn có thể tự mình viết những *driver* đơn giản trong xử lý tính toán, xuất nhập thông báo, ... Thế nhưng, trong thực tế, *driver* không phải chỉ dùng trong việc truy xuất những ký tự thông báo. Nhiệm vụ chính của nó là điều khiển các thiết bị phần cứng thông qua các hàm giao tiếp với các cổng vào ra, truy xuất thanh ghi lệnh, dữ liệu, ... của thiết bị, thu thập thông tin lưu vào vùng nhớ đệm trong *driver* chờ chương trình trong *user* truy xuất. Để có thể giao tiếp với các thiết bị phần cứng thông

qua các cổng vào ra, trong bài sau chúng ta sẽ nghiên cứu các hàm giao tiếp gpio do *linux* hỗ trợ sẵn.



**BÀI 7****CÁC HÀM HỖ TRỢ GPIO****I. Tổng quan về GPIO:**

GPIO, viết tắt của cụm từ *General Purpose Input/Output*, là một thư viện phần mềm điều khiển các cổng vào ra tích hợp trên vi điều khiển hay các ngoại vi IO liên kết với vi điều khiển đó. Hầu hết các vi điều khiển đều hỗ trợ thư viện này, giúp cho việc lập trình các cổng vào ra trở nên thuận tiện hơn. Các tập lệnh vào ra và điều khiển, cách quy định số chân, ... hầu hết tương tự nhau so với các loại vi điều khiển khác nhau. Điều này làm tăng tính linh hoạt, giảm thời gian xây dựng hệ thống.

Theo như quy định chuẩn, mỗi một chân IO trên vi điều khiển sẽ tương ứng với một số GPIO của thư viện này. Số GPIO được quy định như sau: Đối với vi điều khiển ARM9260, số cổng vào ra là 3x32 cổng, tương ứng với 3 ports, đó là các Port A, Port B, và Port C. Mỗi chân quy định trong GPIO theo quy luật sau:

**BASEx32+PIN;**

- Trong đó BASE là số cơ sở của Port. Port A có cơ sở là 1, Port B là 2, Port C là 3. PIN là số thứ tự của từng chân trong Port. Chân 0 có giá trị PIN là 32, 1 là 33, ... Ví dụ, chân thứ 2 của Port A có số GPIO là 33; Chân thứ 2 của Port B có số GPIO là 65, ... tương tự cho các chân còn lại trên vi điều khiển. Đối với các vi điều khiển khác có số Port lớn hơn ta chỉ việc tuân theo quy luật trên để tìm số GPIO phù hợp.

GPIO cho các loại vi điều khiển khác nhau đều có chung những tính chất:

- Mỗi chân trong GPIO đều có thể có hai chế độ *input* và *output*, tùy vào loại vi điều khiển mà GPIO đang sử dụng.
- Trong chế độ *input*, các chân GPIO có thể lập trình để trở thành nguồn ngắt hệ thống.
- Và nhiều chức năng khác nữa, trong quyển sách này chúng ta chỉ tìm hiểu những chức năng phổ biến nhất phục vụ giao tiếp với các chân IO trong các chương trình ứng dụng.

Một trong những thao tác đầu tiên để đưa GPIO hoạt động trong hệ thống là xác định GPIO cần dùng bằng hàm:

```
gpio_request(); //Yêu cầu truy xuất chân GPIO
```

Tiếp đến, chúng ta cấu hình chân GPIO là ngõ vào hay ngõ ra bằng hai hàm:

```
int gpio_direction_input(unsigned gpio);  
int gpio_direction_output(unsigned gpio, int value);
```

Công việc cuối cùng là đưa dữ liệu đến chân GPIO, nếu là ngõ ra; hoặc đọc dữ liệu từ chân GPIO, nếu là ngõ vào; ta sử dụng hai hàm sau:

```
int gpio_get_value(unsigned gpio); //Đọc dữ liệu;  
void gpio_set_value(unsigned gpio, int value); //Xuất dữ liệu;
```

Ngoài ra còn có nhiều hàm chức năng khác sẽ được trình bày trong mục sau.

*\*\*Có nhiều cách thao tác với gpio. Hoặc thao tác với giao diện thiết bị trong cấu trúc root file system (đây là những driver đã được lập trình sẵn), việc điều khiển sẽ là thao tác với tập tin và thư mục (lập trình trong user application). Hoặc dùng trực tiếp những lệnh trong mã nguồn kernel, nghĩa là người sử dụng tạo riêng cho mình một driver sử dụng trực tiếp các hàm giao tiếp với gpio sau đó mới viết chương trình ứng dụng điều khiển IO theo những giao diện trong driver hỗ trợ. Nhằm mục đích thuận tiện cho việc điều khiển IO, phần lập trình nhúng nâng cao chỉ trình bày cách thứ hai, điều khiển trực tiếp IO thông qua thư viện gpio.h trong kernel.*

## **II. Các hàm chính trong GPIO:**

### **1. Hàm gpio\_is\_valid():**

Cú pháp hàm như sau:

```
#include <linux/gpio.h>  
int gpio_is_valid (int number);
```

Do thư viện *gpio* dùng chung cho nhiều loại vi điều khiển khác nhau, nên số lượng chân IO của từng loại cũng khác nhau. Cần thiết phải cung cấp một hàm kiểm tra sự tồn tại của chân trong hệ thống. Hàm *gpio\_is\_valid()* làm nhiệm vụ này. Hàm có tham số là *int number* là số chân *gpio* muốn kiểm tra. Hàm trả về giá trị 0 nếu số chân *gpio* cung cấp là hợp lệ, nghĩa là chân *gpio* này tồn tại trong hệ thống mà *gpio* đang hỗ trợ. Ngược lại, nếu chân *gpio* không hợp lệ, hàm trả về mã lỗi âm “-EINVAL”.

Ví dụ, nếu muốn kiểm tra chân gpio 32 hợp lệ hay không, ta dùng đoạn mã lệnh sau:

```
/*Khai báo biến lưu mã lỗi trả về*/  
int ret;  
/*Gọi hàm kiểm tra gpio*/  
ret = gpio_is_valid(32);
```

```
/*Kiểm tra mã lỗi trả về hệ thống*/
if (ret < 0) {
    /*Nếu xảy ra lỗi, in thông báo cho người dùng*/
    printk ("This gpio pin is not valid\n");
    /*Trả về mã lỗi cho hàm gọi*/
    return ret;
}
```

Công việc kiểm tra được thực hiện đầu tiên khi muốn thao tác với một chân nào đó trong gpio. Nếu hợp lệ các lệnh tiếp theo sẽ được thực hiện. Ngược lại, in ra thông báo cho người dùng và thoát ra khỏi driver.

## **2. Hàm `gpio_request()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_request ( unsigned gpio, const char *lable);
```

Sau khi kiểm tra tính hợp lệ của chân *gpio*, công việc tiếp theo là khai báo sử dụng chân gpio đó. *Linux kernel* cung cấp cho chúng ta hàm `gpio_request()` để thực hiện công việc này. Hàm `gpio_request` có hai tham số. Tham số thứ nhất `unsigned gpio` là chân gpio muốn khai báo sử dụng; Tham số thứ hai `const char *lable` là tên muốn đặt cho *gpio*, phục vụ cho quá trình thao tác với chân gpio dễ dàng hơn. Tham số này có thể để trống bằng cách dùng hằng số rỗng `NULL`.

Sau đây là đoạn chương trình mẫu minh họa cách sử dụng hàm `gpio_request()`:

```
/*Những đoạn lệnh trước*/
...
/*Khai báo biến lưu về mã lỗi cho hàm gọi*/
int ret;
/*Gọi hàm gpio_request (), yêu cầu sử dụng chân gpio 32 với tên là "EXPL"*/
ret = gpio_request (32, "EXPL");
/*Kiểm tra mã lỗi trả về*/
if (ret) {
    /*Nếu xảy ra lỗi thì in ra thông báo cho người sử dụng*/
    printk (KERN_WARNING "EXPL: unable to request gpio 32");
}
```

```
/*Trả về mã lỗi cho hàm gọi*/
return ret;
}
/*Nếu không có lỗi, thực thi những đoạn lệnh khác*/
...
```

### **3. Hàm `gpio_free()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
void gpio_free (unsigned gpio);
```

Hàm `gpio_free()` có chức năng ngược lại với hàm `gpio_request()`. Hàm `gpio_free()` dùng giải phóng chân `gpio` nào đó không cần sử dụng cho hệ thống. Hàm này chỉ có một tham số trả về `unsigned gpio` là số `gpio` của chân muốn giải phóng. Hàm không có dữ liệu trả về.

Sau đây là đoạn chương trình ví dụ cách sử dụng hàm `gpio_free`:

```
/*Đoạn chương trình giải phóng chân gpio 32 ra khỏi driver*/
gpio_free(32);
```

### **4. Hàm `gpio_direction_input()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_direction_input (unsigned gpio);
```

Hàm `gpio_direction_input` dùng để cài đặt chế độ giao tiếp cho chân `gpio` là `input`. Nghĩa là chân `gpio` này sẽ nhận dữ liệu từ bên ngoài lưu vào vùng nhớ đệm bên trong. Hàm `gpio_direction_input` có tham số `unsigned gpio` là chân `gpio` muốn cài đặt chế độ `input`. Hàm trả về giá trị 0 nếu quá trình cài đặt thành công. Ngược lại, sẽ trả về mã lỗi âm.

Sau đây là một ví dụ cho hàm `gpio_direction_input()`:

```
/*Hàm cài đặt chân gpio 32 ở chế độ ngõ vào*/
/*Khai báo biến lưu giá trị mã lỗi trả về cho hàm gọi*/
int ret;
/*Cài đặt chân 32 chế độ ngõ vào*/
ret = gpio_direction_input (32);
```

```
/*Kiểm tra lỗi thực thi */
if (ret) {
    /*Nếu có lỗi in ra thông báo cho người dùng*/
    printk (KERN_WARNING "Unable to set input mode");
    /*Trả về mã lỗi cho hàm gọi*/
    return ret;
}
...
```

### **5. Hàm `gpio_direction_output()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_direction_output (unsigned gpio, int value);
```

Hàm `gpio_direction_output` dùng để cài đặt chế độ giao tiếp cho chân *gpio* là *output*. Chân *gpio* sẽ làm nhiệm vụ xuất dữ liệu bên trong chương trình ra ngoài. Hàm `gpio_direction_output()` có hai tham số. Tham số thứ nhất, `unsigned gpio`, là chân *gpio* muốn cài đặt. Tham số thứ hai, `int value`, là giá trị ban đầu của *gpio* khi nó là ngõ ra. Hàm trả về giá trị 0 nếu quá trình cài đặt thành công. Ngược lại, sẽ trả về mã lỗi âm.

Sau đây là một ví dụ cho hàm `gpio_direction_output()`:

```
/*Hàm cài đặt chân gpio 32 ở chế độ ngõ ra, giá trị ban đầu là 1*/
/*Khai báo biến lưu giá trị mã lỗi trả về cho hàm gọi*/
int ret;
/*Cài đặt chân 32 chế độ ngõ vào*/
ret = gpio_direction_output (32, 1);
/*Kiểm tra lỗi thực thi */
if (ret) {
    /*Nếu có lỗi in ra thông báo cho người dùng*/
    printk (KERN_WARNING "Unable to set gpio 32 into output mode");
    /*Trả về mã lỗi cho hàm gọi*/
    return ret;
}
...
```

### **6. Hàm `gpio_get_value()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
int gpio_get_value (unsigned gpio);
```

Khi chân `unsigned gpio` là ngõ vào, hàm `gpio_get_value()` sẽ lấy giá trị tín hiệu của chân này. Hàm có giá trị trả về dạng `int`, bằng 0 nếu ngõ vào mức thấp, khác 0 nếu ngõ vào mức cao.

Sau đây là một ví dụ đọc vào giá trị chân `gpio 32`, kiểm tra và in thông tin ra màn hình:

```
...
if (gpio_get_value(32)) {
    printk ("The input is high value\n");
} else {
    printk ("The input is low value\n");
}
...
```

### **7. Hàm `gpio_set_value()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
void gpio_set_value (unsigned gpio, int value);
```

Ngược lại với hàm `gpio_get_value()`, hàm `gpio_set_value()` có chức năng xuất dữ liệu cho một chân `unsigned gpio` ngõ ra. Với dữ liệu chứa trong tham số `int value`. Bằng 0 nếu muốn xuất ra mức thấp, bằng 1 nếu muốn xuất ra mức cao. Hàm `gpio_set_value()` không có dữ liệu trả về.

Sau đây là một ví dụ xuất mức cao ra chân `gpio 32`:

```
/*Các lệnh khởi tạo gpio 32 là ngõ ra*/
...
/*Xuất mức cao ra chân gpio 32 */
gpio_set_value (32, 1);
/*Các lệnh xử lý tiếp theo*/
...
```

### **8. Hàm `gpio_to_irq()`:**

Cú pháp của hàm này như sau:

```
#include <linux/gpio.h>
```

```
int gpio_to_irq (unsigned gpio);
```

Đôi khi chúng ta muốn cài đặt chế độ ngắt cho một chân nào đó trong *gpio* dùng để thu nhận thông tin đồng bộ từ phần cứng. Hàm `gpio_to_irq()` thực hiện chức năng này. Hàm có tham số `unsigned gpio` là chân *gpio* muốn cài đặt chế độ ngắt. Hàm có giá trị trả về là số IRQ nếu quá trình cài đặt chế độ ngắt thành công. Ngược lại sẽ trả về mã lỗi âm.

Số IRQ được sử dụng cho hàm `request_irq()` khởi tạo ngắt cho hệ thống, hàm sẽ gán số IRQ với hàm xử lý ngắt. Hàm này sẽ thực hiện những thao tác do người lập trình quy định khi xuất hiện ngắt từ tín hiệu ngắt có số định danh ngắt là IRQ. Bên cạnh đó hàm `request_irq()` còn quy định chế độ ngắt cho chân *gpio* là ngắt theo cạnh hay ngắt theo mức, ...

Sau đây là đoạn chương trình ví dụ khởi tạo ngắt từ chân *gpio*.

```
/*Đoạn chương trình khởi tạo ngắt cho chân gpio 70, PC6*/  
/*Khai báo biến lưu mã lỗi trả về hàm gọi*/  
int ret;  
/*Yêu cầu chân gpio, với định danh là IRQ*/  
ret = gpio_request (70, "IRQ");  
/*Kiểm tra mã lỗi trả về*/  
if (ret) {  
/*Thông báo cho người lập trình có lỗi xảy ra*/  
    printk (KERN_ALERT "Unable to request PC6\n");  
}  
/*Cài đặt chế độ kéo lên cho chân gpio*/  
at91_set_GPIO_periph (70, 1);  
/*Cài đặt chân gpio là input*/  
at91_set_gpio_input (70, 1);  
/*Cài đặt chân gpio có chế độ chống lỗi*/  
at91_set_deglitch (70, 1);  
/*Cài đặt gpio thành nguồn ngắt, lưu về số định danh ngắt irq*/  
irq = gpio_to_irq (70); /* Get IRQ number */  
/*Thông báo chân gpio đã khởi tạo ngắt */  
printk (KERN_ALERT "IRQ = %d\n", irq);
```

```
/*Khai báo ngắt chân gpio cho hệ thống*/
ret = request_irq (irq, gpio_irq_handler,
                  IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                  "MyIRQ", NULL);

/*Kiểm tra mã lỗi trả về khi khai báo ngắt*/
if (ret) {
/*Thông báo cho người sử dụng số định danh ngắt không còn trống*/
    printk (KERN_ALERT "IRQ %d is not free\n", irq);
/*Trả về mã lỗi cho hàm gọi */
    return ret;
}
```

Như vậy để tránh lỗi trong quá trình khởi tạo ngắt cho chân gpio, trước tiên chúng ta phải cài đặt những thông số cần thiết cho gpio đó, chẳng hạn như chân gpio phải hợp lệ và ở chế độ ngõ vào.

### **III. Kết luận:**

Trên đây, chúng ta đã sử dụng được những hàm điều khiển các chân gpio ngoại vi để thực hiện chức năng cụ thể của yêu cầu ứng dụng. Kết hợp với những kiến thức trong những bài trước: Giao diện điều khiển liên kết *user space* với *kernel space*, các hàm trì hoãn thời gian trong *user space*, các kỹ thuật lập trình C, ... chúng ta có thể viết được hầu hết tất cả những ứng dụng có liên quan đến các cổng vào ra: chẳng hạn như điều khiển LED, điều khiển LCD, ...

*gpio* không chỉ điều khiển các cổng vào ra trên vi điều khiển, theo yêu cầu trong thực tế, đa số các kit nhúng, số cổng vào ra rất hạn chế, đòi hỏi phải có các chân io mở rộng từ những linh kiện phụ trợ khác, vì thế *gpio* còn hỗ trợ thêm các hàm điều khiển các chân io mở rộng. Vấn đề này sẽ được đề cập trong những tài liệu khác không thuộc phạm vi của giáo trình này.

Trước khi đi vào ứng dụng các hàm thao tác với *gpio* trong điều khiển LED đơn, LCD, ... bài tiếp theo sẽ tìm hiểu thêm về cách trì hoãn thời gian trong *kernel*, với cách trì hoãn thời gian này, chúng ta không cần dùng đến các hàm trì hoãn khác trong *user space* vì lý do yêu cầu đồng bộ hóa phần cứng hệ thống, hay một số trường hợp chúng ta muốn sử dụng delay trong driver gắn liền với những thao tác điều khiển.



## **CÁC HÀM HỖ TRỢ GPIO**

### **I. Sơ lược về thời gian trong kernel:**

Thời gian trong hệ điều hành linux nói riêng và các hệ điều hành khác nói chung điều rất quan trọng. Trong hệ điều hành, những hoạt động điều dựa theo sự tác động mang tính chất chu kỳ. Chẳng hạn như hoạt động chia khe thời gian thực thi, chuyển qua lại giữa các tiến trình với nhau, đồng bộ hóa hoạt động giữa các thiết bị phần cứng có thời gian truy xuất không giống nhau, và nhiều hoạt động quan trọng khác nữa.

Khi làm việc với hệ thống linux, chúng ta cần phân biệt hai khái niệm thời gian: Thời gian tuyệt đối và thời gian tương đối. Thời gian tuyệt đối được hiểu như thời gian thực của hệ thống, là các thông tin như ngày-tháng-năm-giờ-phút-giây và các đơn vị thời gian khác nhỏ hơn, phục vụ cho người sử dụng. Thời gian tương đối được hiểu như những khoảng thời gian được cập nhật không cố định, mang tính chất chu kỳ, mốc thời gian không cố định và thông thường không biết trước. Ví dụ thời gian tuyệt đối là thời điểm trong một ngày, có mốc thời gian tính từ ngày 1 tháng 1 năm 1970 quy định trong các thiết bị thời gian thực. Thời gian tương đối là khoảng thời gian tính từ thời điểm xảy ra một sự kiện nào đó, chẳng hạn định thời tác động một khoảng thời gian sau khi hệ thống có lỗi hoặc cập nhật thông số hiện tại của hệ thống sau mỗi một thời khoảng cố định.

Để có thể kiểm tra, quản lý và thao tác với thời gian một cách chính xác, hệ điều hành phải dựa vào thiết bị thời gian được tích hợp trên hầu hết các vi xử lý đó là *timer*. *Timer* được sử dụng bởi hệ điều hành được gọi là *timer* hệ thống, hệ điều hành cài đặt các thông số thích hợp cho *timer*, quy định khoảng thời gian sinh ra ngắt, khoảng thời gian giữa hai lần xảy ra ngắt liên tiếp được gọi bởi thuật ngữ *tick*, giá trị của *tick* do người sử dụng *driver* quy định. Khi xảy ra ngắt, hệ điều hành linux sẽ cập nhật giá trị của biến *jiffies*, chuyển tiến trình, cập nhật thời gian trong ngày, ...

Trong phần lập trình *user application*, chúng ta đã tìm hiểu những hàm thao tác với thời gian *user space*. Đây là những hàm được hỗ trợ sẵn bởi hệ điều hành, nghĩa là chúng được lập trình để hoạt động ổn định không gây ảnh hưởng đến các tiến trình khác chạy đồng thời. Trong *kernel*, những hàm thao tác với thời gian hoạt động bên ngoài tầm kiểm soát của hệ điều hành, sử dụng trực tiếp tài nguyên phần cứng của hệ thống, cụ thể là thời

gian hoạt động của vi xử lý trung tâm trong vi điều khiển. Vì thế nếu sử dụng không phù hợp thì hệ điều hành sẽ hoạt động không ổn định, xảy ra những lỗi về thời gian thực trong khi thực hiện tác vụ. Nhưng thời gian thực hiện của các hàm này sẽ nhanh hơn, vì không qua trung gian là hệ điều hành, phù hợp với yêu cầu điều khiển của *driver* là nhanh chóng và chính xác.

Trong bài này, đầu tiên chúng ta sẽ tìm hiểu nguyên tắc quản lý thời gian trong *kernel*, sau đó là các hàm tương tác, xử lý thời gian thực, cách sử dụng định thời trong *timer*, và cuối cùng là một số kỹ thuật trì hoãn thời gian trong *kernel*.

## **II. Đơn vị thời gian trong *kernel*:**

Để quản lý chính xác thời gian, hệ điều hành sử dụng bộ định thời *timer* tích hợp sẵn trên vi điều khiển mà nó hoạt động. Bộ định thời (*timer*) được đặt cho một giá trị cố định sao cho có thể sinh ra ngắt theo chu kỳ cho trước. Khoảng thời gian của một chu kỳ ngắt được gọi là *tick*. Trong một giây có khoảng  $N$  *ticks* được hoàn thành. Hay nói cách khác, tốc độ tick là  $N$  Hz. Giá trị  $N$  được định nghĩa bởi người sử dụng trước khi biên dịch *kernel*. Thông thường một số hệ điều hành có giá trị mặc định là  $N = 100$  và đây cũng là giá trị mặc định của hệ điều hành chạy trên kit KM9260.

Giá trị của HZ được định nghĩa như sau:

```
# define USER_HZ 100 /* User interfaces are in "ticks" */
```

và

```
# define HZ 100 /*Internal kernel timer frequency*/
```

trong tập tin `\arch\arm\include\asm\param.h`. Chúng ta thấy, giá trị mặc định của HZ là 100. Có nghĩa là *timer* hệ thống được cài đặt sao cho trong một giây có khoảng 100 lần ngắt xảy ra. Hay nói cách khác, chu kỳ ngắt là 10 ms. Chúng ta cũng chú ý, trong tập tin có hai tham số cần sửa chữa một lúc đó là *USER\_HZ* và *HZ*. Để thuận tiện cho việc chuyển đổi qua lại thời gian giữa *kernel* và *user* thì giá trị của chúng phải giống nhau.

Giá trị *HZ* thay đổi phải phù hợp với ứng dụng mà hệ điều hành đang phục vụ. Làm thế nào để làm được điều này.

Chẳng hạn, giá trị của *HZ* ảnh hưởng rất nhiều đến độ phân giải của những hàm định thời ngắt. Với giá trị  $HZ = 100$ , thời gian lập trình định thời ngắt tối thiểu là 10ms. Với giá trị  $HZ=1000$ , thời gian lập trình định thời ngắt tối thiểu là 1ms. Điều này có nghĩa là giá trị của *HZ* càng lớn càng tốt. Liệu thực sự có phải như thế?

Khi tăng giá trị của *HZ*, điều có những ưu và nhược điểm. Về ưu điểm, tăng giá trị *HZ* làm độ phân giải của thời gian *kernel* tăng lên, như thế một số ứng dụng có liên quan đến thời gian cũng chính xác hơn, ... Về nhược điểm, khi tăng giá trị của *HZ*, thì vi điều khiển thực hiện ngắt nhiều hơn, tiêu tốn thời gian cho việc lưu lưu ngăn xếp, khởi tạo ngắt, ... nhiều hơn, ... do vậy tùy từng ứng dụng cụ thể mà chúng ta thay đổi giá trị *HZ* cho phù hợp để hệ thống hoạt động tối ưu.

### III. jiffies:

*jiffies* là một biến toàn cục được định nghĩa trong thư viện `linux/jiffies.h` để lưu số lượng *ticks* đạt được kể từ khi hệ thống bắt đầu khởi động. Khi xảy ra ngắt timer hệ thống, kernel tiến hành tăng giá trị của *jiffies* lên 1 đơn vị. Như vậy nếu như có *HZ* ticks trong một giây thì *jiffies* sẽ tăng trong một giây là *HZ* đơn vị. Nếu như chúng ta đọc được giá trị *jiffies* hiện tại là *N*, thì thời gian kể từ khi hệ thống khởi động là *N* ticks hay *N/HZ* giây. Đôi khi chúng ta muốn đổi giá trị *jiffies* sang giây và ngược lại ta chỉ việc chia hay nhân giá trị *jiffies* cho (với) *HZ*.

Trong *kernel*, *jiffies* được lưu trữ dưới dạng số nhị phân 32 bits. Là 32 bits có trong số thấp trong tổng số 64 bits của biến *jiffies\_64*. Với chu kỳ *tick* là 10ms thì sau một khoảng thời gian 5.85 tỷ năm đối với biến *jiffies\_64* và 1.36 năm đối với biến *jiffes* mới có thể bị tràn. Xác suất để biến *jiffies\_64* bị tràn là cực kỳ nhỏ và *jiffies* là rất nhỏ. Thế nhưng vẫn có thể xảy ra đối với những ứng dụng đòi hỏi độ tin cậy rất cao. Nếu cần có thể kiểm tra nếu yêu cầu chính xác cao.

Khi thao tác với *jiffies*, *kernel* hỗ trợ cho chúng ta các hàm so sánh thời gian sau, tất cả các hàm này đều được định nghĩa trong thư viện `linux/jiffies.h`:

Đầu tiên là hàm *get\_jiffies\_64()*, với giá trị *jiffies* thì chúng ta có thể đọc trực tiếp theo tên của nó, thế nhưng *jiffies\_64* không thể đọc trực tiếp mà phải thông qua hàm riêng vì giá trị của *jiffies\_64* được chứa trong số 64 bits. Hàm không có tham số, giá trị trả về của hàm là số có 64 bits.

Cuối cùng là các hàm so sánh thời gian theo giá trị của *jiffies*. Các hàm này được định nghĩa trong thư viện `linux/jiffies.h` như sau:

```
#define time_after(unknown,known) ((long)(known)-(long)(unknown)<0)
#define time_before(unknown,known) ((long)(unknown)-(long)(known)<0)
#define time_after_eq(unknown,known) ((long)(unknown)-(long)(known)>= 0)
#define time_before_eq(unknown,known) ((long)(known)-(long)(unknown)>= 0)
```

các hàm này trả về giá trị kiểu `boolean`, tùy theo tên hàm và tham số của hàm. Hàm `time_after(unknown, known)` trả về giá trị đúng nếu `unknown > known`, tương tự cho các hàm khác. Ứng dụng của các hàm này khi chúng ta muốn so sánh hai khoảng thời gian với nhau để thực thi một tác vụ nào đó, chẳng hạn ứng dụng trong trì hoãn thời gian như trong đoạn chương trình sau:

```
/*Đoạn chương trình trì hoãn thời gian 1s dùng jiffies*/  
/*Khai báo biến lưu thời điểm cuối cùng muốn so sánh*/  
unsigned long timeout = jiffies + HZ; //Trì hoãn 1s  
/*Kiểm tra xem giá trị timeout có bị tràn hay không*/  
if (time_after(jiffies, timeout)) {  
    printk("This timeout is overflow\n");  
    return -1;  
}  
/*Thực hiện trì hoãn thời gian nếu không bị tràn*/  
if (time_before(jiffies, timeout)) {  
    /*Do nothing loop to delay*/  
}
```

#### **IV. Thời gian thực trong kernel:**

Trong phần lập trình ứng dụng *user*, chúng ta đã tìm hiểu kỹ về các lệnh xử lý thời gian thực trong thư viện `<time.h>`. Trong *kernel* cũng có những hàm được xây dựng sẵn thao tác với thời gian thực nằm trong thư viện `<linux/time.h>`. Sự khác biệt giữa hai thư viện này là vai trò của chúng trong hệ thống. `<time.h>` chứa trong lớp *user*, giao tiếp với *kernel*, tương tự như các hàm giao diện trung gian giao tiếp giữa người dùng với thời gian thực trong *kernel*, vì thế thư viện chứa những hàm chủ yếu phục vụ cho người dùng; Đối với thư viện `<linux/time.h>` hoạt động trong lớp *kernel*, chứa những hàm được lập trình sẵn phục vụ chủ yếu cho hệ thống *kernel*, giúp người lập trình *driver* quản lý thời gian thực tiện lợi hơn, ví dụ như các hàm xử lý thời gian ngắt, định thời, so sánh thời gian, ... điểm đặc biệt là thời gian thực trong *kernel* chủ yếu quản lý dưới dạng số giây tuyệt đối, không theo dạng ngày tháng năm như trong *user*.

Các kiểu cấu trúc thời gian, ý nghĩa các tham số trong những hàm thời gian đa phần tương tự như trong thư viện `<time.h>` trong *user application* nên chúng sẽ được trình bày sơ lược trong khi giải thích.

**1. Các kiểu, cấu trúc thời gian:**

a. **Cấu trúc timespec:** cấu trúc này được định nghĩa như sau:

```
struct timespec {
    __kernel_time_t  tv_sec;           /* seconds */
    long             tv_nsec;         /* nanoseconds */
};
```

Trong đó, `tv_sec` dùng lưu thông tin của giây; `tv_nsec` dùng lưu thông tin nano giây của giây hiện tại.

b. **Cấu trúc timeval:** Cấu trúc này được định nghĩa như sau:

```
struct timeval {
    __kernel_time_t  tv_sec;           /* seconds */
    __kernel_suseconds_t tv_usec;     /* microseconds */
};
```

Trong đó, `tv_sec` dùng lưu thông tin về số giây; `tv_usec` dùng lưu thông tin về micro giây của giây hiện tại.

c. **Cấu trúc timezone:** Cấu trúc này được định nghĩa như sau:

```
struct timezone {
    int  tz_minuteswest; /* minutes west of Greenwich */
    int  tz_dsttime;     /* type of dst correction */
};
```

Trong đó, `tz_minuteswest` là số phút chênh lệch múi giờ về phía Tây của Greenwich; `tz_dsttime` là tham số điều chỉnh chênh lệch thời gian theo mùa;

**2. Các hàm so sánh thời gian:**

a. **timespec\_equal:** Hàm được định nghĩa như sau:

```
static inline int timespec_equal(const struct timespec *a,
                                const struct timespec *b)
{
    return (a->tv_sec == b->tv_sec) && (a->tv_nsec == b->tv_nsec);
}
```

Nhiệm vụ của hàm là so sánh 2 con trỏ cấu trúc thời gian kiểu `timespec`, `const struct timespec *a` và `const struct timespec *b`. So sánh từng thành phần trong cấu trúc, `tv_sec` và `tv_nsec`, nếu hai thành phần này bằng nhau thì hai cấu trúc thời gian này bằng nhau. Khi đó hàm trả về giá trị *true*, ngược lại trả về giá trị *false*;

**b. *timespec\_compare*:** Hàm được định nghĩa như sau:

```
static inline int timespec_compare(const struct timespec *lhs, const
struct timespec *rhs)
{
    if (lhs->tv_sec < rhs->tv_sec)
        return -1;
    if (lhs->tv_sec > rhs->tv_sec)
        return 1;
    return lhs->tv_nsec - rhs->tv_nsec;
}
```

Nhiệm vụ của hàm là so sánh hai cấu trúc thời gian kiểu `timespec`, `const struct timespec *lhs` và `const struct timespec *rhs`. Kết quả là một trong 3 trường hợp sau:

- Nếu `lhs < rhs` thì trả về giá trị nhỏ hơn 0;
- Nếu `lhs = rhs` thì trả về giá trị bằng 0;
- Nếu `lhs > rhs` thì trả về giá trị lớn hơn 0;

**c. *timeval\_compare*:** Hàm được định nghĩa như sau:

```
static inline int timeval_compare(const struct timeval *lhs, const
struct timeval *rhs)
{
    if (lhs->tv_sec < rhs->tv_sec)
        return -1;
    if (lhs->tv_sec > rhs->tv_sec)
        return 1;
    return lhs->tv_usec - rhs->tv_usec;
}
```

Nhiệm vụ của hàm là so sánh hai cấu trúc thời gian kiểu `timeval`, `const struct timeval *lhs` và `const struct timeval *rhs`. Kết quả trả về là một trong 3 trường hợp:

- Nếu `lhs < rhs` thì trả về giá trị nhỏ hơn 0;
- Nếu `lhs = rhs` thì trả về giá trị bằng 0;
- Nếu `lhs > rhs` thì trả về giá trị lớn hơn 0;

**3. Các phép toán thao tác trên thời gian:**

**a. *mktime*:** Hàm được định nghĩa như sau:

```
extern unsigned long mktime(  
    const unsigned int year, const unsigned int mon,  
    const unsigned int day, const unsigned int hour,  
    const unsigned int min, const unsigned int sec);
```

Nhiệm vụ của hàm là chuyển các thông tin thời gian dạng ngày tháng năm ngày giờ phút giây thành thông tin thời gian dạng giây tính từ thời điểm epoch.

**b. *set\_normalized\_timespec*:** Hàm được định nghĩa như sau:

```
extern void set_normalized_timespec(struct timespec *ts, time_t sec,  
    long nsec);
```

Sau khi chuyển các thông tin ngày tháng năm ... thành số giây tính từ thời điểm epoch bằng cách sử dụng hàm `mktime`, thông tin trả về chưa phải là một cấu trúc thời gian chuẩn có thể xử lý được trong `kernel`. Để biến thành cấu trúc thời gian chuẩn ta sử dụng hàm `set_normalize_timespec` để chuyển số giây dạng `unsigned long` thành cấu trúc thời gian `timespec`.

Hàm có 3 tham số. Tham số thứ nhất, `struct timespec *ts`, là con trỏ trỏ đến cấu trúc `timespec` được định nghĩa trước đó lưu giá trị thông tin thời gian trả về sau khi chuyển đổi xong; Tham số thứ hai, `time_t sec`, là số giây muốn chuyển đổi sang cấu trúc `timespec` (được lưu vào trường `tv_sec`); Tham số thứ ba, `long nsec`, là số nano giây của giây của thời điểm muốn chuyển đổi.

**c. *timespec\_add\_safe*:** Hàm được định nghĩa như sau:

```
extern struct timespec timespec_add_safe(  
    const struct timespec lhs,  
    const struct timespec rhs);
```

Nhiệm vụ của hàm là cộng hai cấu trúc thời gian kiểu `timespec`, `const struct timespec lhs` và `const struct timespec rhs`. Kết quả trả về dạng `timespec` là tổng của hai giá trị thời gian nếu không bị tràn, nếu bị tràn thì hàm sẽ trả về giá trị lớn nhất có thể có của kiểu `timespec`.

**d. *timespec\_sub*:** Hàm được định nghĩa như sau:

```
static inline struct timespec timespec_sub(struct timespec  
    lhs, struct timespec rhs)
```

```
{
    struct timespec ts_delta;
    set_normalized_timespec(&ts_delta, lhs.tv_sec - rhs.tv_sec,
                           lhs.tv_nsec - rhs.tv_nsec);
    return ts_delta;
}
```

Nhiệm vụ của hàm này là trừ hai cấu trúc thời gian kiểu `timespec`, `struct timespec lhs` và `struct timespec rhs`. Hiệu của hai cấu trúc này được trả về dưới dạng `timespec`. Thông thường hàm dùng để tính toán khoảng thời gian giữa hai thời điểm.

**e. `timespec_add_ns`:** Hàm được định nghĩa như sau:

```
static __always_inline void timespec_add_ns(struct timespec *a, u64
ns)
{
    a->tv_sec+=__iter_div_u64_rem(a->tv_nsec+ns, NSEC_PER_SEC, &ns);
    a->tv_nsec = ns;
}
```

Nhiệm vụ của hàm này là cộng thêm một khoảng thời gian tính bằng nano giây vào cấu trúc `timespec` được đưa vào hàm. Hàm có hai tham số. Tham số thứ nhất, `struct timespec *a`, là cấu trúc `timespec` muốn cộng thêm. Tham số thứ hai, `u64 ns`, là số nano giây muốn cộng thêm vào.

#### **4. Các hàm truy xuất thời gian:**

**a. `do_gettimeofday`:** Hàm được định nghĩa như sau:

```
extern void do_gettimeofday(struct timeval *tv);
```

Nhiệm vụ của hàm là lấy về thông tin thời gian hiện tại của hệ thống. Thông tin thời gian hiện tại được trả về cấu trúc dạng `struct timeval *tv` trong tham số của hàm.

**b. `do_settimeofday`:** Hàm được định nghĩa như sau:

```
extern int do_settimeofday(struct timespec *tv);
```

Nhiệm vụ của hàm là cài đặt thông tin thời gian hiện tại cho hệ thống dựa vào cấu trúc thời gian dạng `timespec` chứa trong tham số `struct timespec *tv` của hàm.

**c. `do_sys_settimeofday`:** Hàm được định nghĩa như sau:

```
extern int do_sys_settimeofday(struct timespec *tv, struct timezone
*tz);
```



Nhiệm vụ của hàm là cài đặt thông tin thời gian hiện tại của hệ thống có tính đến sự chênh lệch thời gian về múi giờ dựa vào hai tham số trong hàm. Tham số thứ nhất, `struct timespec *tv`, là thông tin thời gian muốn cập nhật. Tham số thứ hai, `struct timezone *tz`, là cấu trúc lưu thông tin chênh lệch múi giờ muốn cập nhật.

**d. *getboottime*:** Hàm được định nghĩa như sau:

```
extern void getboottime (struct timespec *ts);
```

Nhiệm vụ của hàm là lấy về tổng thời gian từ khi hệ thống khởi động đến thời điểm hiện tại. Thông tin thời gian được lưu về cấu trúc kiểu `timespec`, `struct timespec`.

## **5. Các hàm chuyển đổi thời gian:**

**a. *timespec\_to\_ns*:** Hàm được định nghĩa như sau:

```
static inline s64 timespec_to_ns(const struct timespec *ts)
{
    return ((s64) ts->tv_sec * NSEC_PER_SEC) + ts->tv_nsec;
}
```

Nhiệm vụ của hàm là chuyển cấu trúc thời gian dạng `timespec`, `const struct timespec *ts`, thành số nano giây lưu vào biến 64 bits làm giá trị trả về cho hàm.

**b. *timeval\_to\_ns*:** Hàm được định nghĩa như sau:

```
static inline s64 timeval_to_ns(const struct timeval *tv)
{
    return ((s64) tv->tv_sec * NSEC_PER_SEC) +
        tv->tv_usec * NSEC_PER_USEC;
}
```

Nhiệm vụ của hàm là chuyển đổi cấu trúc thời gian dạng `timeval`, `const struct timeval *tv`, thành số nano giây lưu vào biến 64 bits làm giá trị trả về cho hàm.

**c. *ns\_to\_timespec*:** Hàm được định nghĩa như sau

```
extern struct timespec ns_to_timespec(const s64 nsec);
```

Nhiệm vụ của hàm là chuyển thông tin thời gian dưới dạng nano giây, `const s64 nsec`, thành thông tin thời gian dạng `timespec` làm giá trị trả về cho hàm.

**d. *ns\_to\_timeval*:** Hàm được định nghĩa như sau:

```
extern struct timeval ns_to_timeval(const s64 nsec);
```

Nhiệm vụ của hàm là chuyển thông tin thời gian dạng nano giây, `const s64 nsec`, thành thông tin thời gian dạng `timeval` làm giá trị trả về cho hàm.



**V. Timer và sử dụng ngắt trong timer:****1. Khái quát về timer trong kernel:**

*Timer* là một định nghĩa quen thuộc trong hầu hết tất cả các hệ thống khác nhau. Trong *linux kernel*, *timer* được hiểu là trì hoãn thời gian động để gọi thực thi một hàm nào đó được lập trình trước. Nghĩa là thời điểm bắt đầu trì hoãn không cố định, thông thường được tính từ lúc cài đặt khởi động *timer* trong hệ thống, khoảng thời gian trì hoãn do người lập trình quy định, khi hết thời gian trì hoãn, *timer* sinh ra một ngắt. *Kernel* tạm hoãn hoạt động hiện tại của mình để thực thi hàm ngắt được lập trình trước đó.

Để đưa một *timer* vào hoạt động, chúng ta cần phải thực hiện nhiều thao tác. Đầu tiên phải khởi tạo *timer* vào hệ thống *linux*. Tiếp theo, cài đặt khoảng thời gian mong muốn trì hoãn. Cài đặt hàm thực thi ngắt khi thời gian trì hoãn kết thúc. Khi xảy ra hoạt động ngắt, *timer* sẽ bị vô hiệu hóa. Vì thế, nếu muốn *timer* hoạt động theo chu kỳ cố định chúng ta phải khởi tạo lại *timer* ngay trong chương trình thực thi ngắt. Số lượng *timer* khởi tạo trong *kernel* là không giới hạn, vì đây là một timer mềm không phải là *timer* vật lý, có giới hạn là dung lượng vùng nhớ cho phép. Chúng ta sẽ trình bày cụ thể những bước trên trong phần sau.

**2. Các bước sử dụng timer:**

**\*\*Lưu ý:** Khi sử dụng các hàm trong *timer* chúng ta phải thêm thư viện `<linux/timer.h>` ở đầu chương trình.

**a. Bước 1:** Khai báo biến cấu trúc `timer_list` để lưu *timer* khi khởi tạo

Cấu trúc `timer_list` được *linux kernel* định nghĩa trong thư viện `<linux/timer.h>` như sau:

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    void (*function) (unsigned long);
    unsigned long data;
    struct tvec_t_base_s *base;
}
```

Ta khai báo timer bằng dòng lệnh sau:

```
/*Khai báo một timer có tên là my_timer*/
struct timer_list my_timer;
```

**b. Bước 2:** Khai báo và định nghĩa hàm thực thi ngắt

Hàm thực thi ngắt có dạng như sau:

```
void my_timer_function (unsigned long data) {  
    /*Các thao tác do người lập trình driver quy định*/  
    ...  
    /*Nếu cần thiết có thể khởi tạo lại timer trong phần cuối chương trình*/  
}
```

**c. Bước 3:** Khởi tạo các tham số cho cấu trúc *timer*

Công việc tiếp theo là yêu cầu *kernel* cung cấp một vùng nhớ cho *timer* hoạt động, chúng ta sử dụng câu lệnh:

```
/*Khởi tạo timer vừa khai báo my_timer*/  
init_timer (&my_timer);  
  
Sau khi kernel dành cho timer một vùng nhớ, timer vẫn còn trống chưa được gán những thông số cần thiết, công việc này của người lập trình driver:  
  
/*Khởi tạo giá trị khoảng thời gian muốn trì hoãn, đơn vị tính bằng tick*/  
my_timer.expires = jiffies + delay;  
/*Gán tham số cho hàm thực thi ngắt, nếu không có tham số ta có thể gán một giá trị bất kỳ*/  
my_timer.data = 0;  
/*Gán con trỏ hàm xử lý ngắt vào timer*/  
my_timer.function = my_function;
```

Trong các tham số trên, chúng ta cần chú ý những tham số sau đây:

- `my_timer.expires` đây là giá trị thời gian tương lai có đơn vị là *tick*, tại mọi thời điểm, *timer* so sánh với số *jiffies*. Nếu số *jiffies* bằng với số `my_timer.expires` thì ngắt xảy ra.
- `my_timer.data` là tham số chúng ta muốn đưa vào hàm thực thi ngắt, đôi khi chúng ta muốn khởi tạo các thông số ban đầu cho hàm thực thi ngắt, kế thừa những thông tin đã xử lý từ trước hay từ người dùng.

- `my_timer.function` là trường chứa con trỏ của hàm phục vụ ngắt đã được khởi tạo và định nghĩa trước đó.

**d. Bước 4:** Kích hoạt timer hoạt động trong *kernel*:

Chúng ta thực thi lệnh sau:

```
add_timer (&my_timer);
```

Tham số của hàm `add_timer` là con trỏ của timer được khởi tạo và gán các tham số cần thiết.

Khi *timer* được kích hoạt, hàm thực thi ngắt hoạt động, nó sẽ bị vô hiệu hóa trong chu kỳ tiếp theo. Muốn *timer* tiếp tục hoạt động, chúng ta phải kích hoạt lại bằng câu lệnh sau:

```
/*Kích hoạt timer hoạt động lại cho chu kỳ ngắt tiếp theo*/  
mod_timer (&my_timer, jiffies + new_delay);
```

Nếu muốn xóa timer khỏi hệ thống chúng ta dùng lệnh sau:

```
/*Xóa timer khỏi hệ thống*/  
del_timer (&my_timer);
```

Tuy nhiên lệnh này chỉ thực hiện thành công khi *timer* không còn hoạt động, nghĩa là khi *timer* còn đang chờ chu kỳ ngắt tiếp theo dùng lệnh `del_timer()` sẽ không hiệu quả. Muốn khắc phục lỗi này, chúng ta phải dùng lệnh `del_timer_sync()`, lệnh này sẽ chờ cho đến khi *timer* hoàn thành chu kỳ ngắt gần nhất mới xóa *timer* đó.

**3. Ví dụ:**

Đoạn chương trình sau sẽ định thời xuất thông tin ra màn hình hiển thị với chu kỳ là 1s. Mỗi lần xuất sẽ thay đổi thông tin, thông báo những lần xuất thông tin là khác nhau;

```
/*Khai báo biến cục bộ lưu giá trị muốn xuất ra màn hình, giá trị ban đầu bằng 0*/  
int counter = 0;  
  
/*Khai báo biến timer phục vụ ngắt*/  
struct time_list my_timer;  
  
/*Khai báo định nghĩa hàm phục vụ ngắt*/  
void timer_isr (unsigned long data) {  
  
/*In thông báo cho người dùng*/  
printk ("Driver: Hello, the counter's value is %d\n", counter++);  
}
```

```
/*Cài đặt lại giá trị timer cho lần hoạt động tiếp theo, cài đặt chu kỳ 1 giây*/
mod_timer (&my_timer, jiffies + HZ);
}

/*Thực hiện khởi tạo timer trong khi cài đặt driver vào hệ thống, trong hàm init()*/
static int
__init my_driver_init (void) {
/*Các lệnh khởi tạo khác*/

...

/*Khởi tạo timer hoạt động ngắt*/
/*Khởi tạo timer đã được khai báo*/
init_timer (&my_timer);
/*Cài đặt các thông số cho timer*/
my_timer.expires = jiffies + HZ; //Khởi tạo trì hoãn ban đầu là 1s;
my_timer.data = 0; //Dữ liệu truyền cho hàm ngắt là 0;
my_timer.function = my_function; //Gán hàm thực thi ngắt cho timer.
/*Kích hoạt timer hoạt động trong hệ thống*/
add_timer (&my_timer);
...
}
```

## **VI. Trì hoãn thời gian trong kernel:**

Trì hoãn thời gian là một trong những vấn đề quan trọng trong lập trình *driver* cũng như trong lập trình *application*. Trong phần trước chúng ta đã tìm hiểu những lệnh trì hoãn thời gian từ khoảng thời gian nhỏ tính theo nano giây đến khoảng thời gian lớn hơn tính bằng giây. Tùy thuộc vào yêu cầu chính xác cao hay thấp mà chúng ta áp dụng kỹ thuật trì hoãn thời gian cho phù hợp. *Kernel* cũng hỗ trợ các kỹ thuật trì hoãn khác nhau tùy theo yêu cầu mà áp dụng kỹ thuật nào phù hợp nhất sao cho không ảnh hưởng đến hoạt động của hệ thống.

### **1. Vòng lặp vô tận:**

Kỹ thuật trì hoãn thời gian đầu tiên là vòng lặp *busy loop*. Đây là cách trì hoãn thời gian cổ điển và đơn giản nhất, áp dụng cho tất cả các hệ thống. Kỹ thuật trì hoãn này có dạng như sau:

```
/*Khai báo thời điểm tương lai muốn thực hiện trì hoãn*/  
unsigned long timeout = jiffies + HZ/10; //Trì hoãn 10ms;  
/*Thực hiện vòng lặp vô tận while () trì hoãn thời gian*/  
while (time_before(jiffies, timeout))  
    ;
```

Với cách trì hoãn thời gian trên, chúng ta dùng biến *jiffies* để so sánh với thời điểm tương lai làm điều kiện cho lệnh `while ()` thực hiện vòng lặp. Như vậy chúng ta chỉ có thể trì hoãn một khoảng thời gian đúng bằng một số nguyên lần của *tick*. Hơn nữa, khác với lớp *user*, vòng lặp trong *kernel* không được chia tiến trình thực hiện. Vì thế vòng lặp vô tận trong *kernel* sẽ chiếm hết thời gian làm việc của CPU và như thế các hoạt động khác sẽ không được thực thi, hệ thống sẽ bị ngưng lại tạm thời. Điều này rất nguy hiểm cho các ứng dụng đòi hỏi độ tin cậy cao về thời gian thực. Cách trì hoãn thời gian này rất hiếm khi được sử dụng trong những hệ thống lớn.

Để giải quyết vấn đề này, người ta dùng kỹ thuật chia tiến trình trong lúc thực hiện trì hoãn như sau:

```
while (time_before(jiffies, timeout))  
    schedule(); //Hàm này chứa trong thư viện <linux/sched.h>
```

Trong khi *jiffies* vẫn thỏa mãn điều kiện nhỏ hơn thời điểm đặt trước, *kernel* sẽ chia thời gian thực hiện công việc khác trong hệ thống. Cho đến khi vòng lặp được thoát, những lệnh tiếp theo sẽ tiếp tục thực thi.

Chúng ta cũng có thể áp dụng những lệnh so sánh thời gian thực trong phần trước để thực hiện trì hoãn thời gian với độ chính xác cao hơn.

## **2. Trì hoãn thời gian bằng những hàm hỗ trợ sẵn:**

*Linux kernel* cũng cung cấp cho chúng ta những hàm thực hiện trì hoãn thời gian với độ chính xác cao, thích hợp cho những khoảng thời gian nhỏ. Đó là những hàm:

- `void udelay(unsigned long usec);` Hàm dùng để trì hoãn thời gian có độ phân giải micro giây;
- `void ndelay(unsigned long nsec);` Hàm trì hoãn thời gian có độ phân giải nano giây;
- `void mdelay(unsigned long msec);` Hàm trì hoãn thời gian có độ phân giải mili giây;

*\*\*Các hàm trì hoãn thời gian này chỉ thích hợp cho những khoảng thời gian nhỏ hơn 1 tick trong kernel. Nếu lớn hơn sẽ làm ảnh hưởng đến hoạt động của cả hệ thống vì bản chất đây vẫn là những vòng lặp vô tận, chiếm thời gian hoạt động của CPU.*

### **3. Trì hoãn thời gian bằng hàm `schedule_timeout()`:**

Kỹ thuật này khác với hai kỹ thuật trên, dùng hàm `schedule_timeout()` sẽ làm cho chương trình driver dừng lại tại thời điểm khai báo, rơi vào trạng thái ngủ trong suốt thời gian trì hoãn. Thời gian trì hoãn do người lập trình cài đặt. Để sử dụng hàm này, ta tiến hành các bước sau:

*/\*Cài đặt chương trình driver vào trạng thái ngủ\*/*

```
set_current_state (TASK_INTERRUPTIBLE);
```

*/\*Cài đặt thời gian cho tín hiệu đánh thức chương trình\*/*

```
schedule_timeout (unsigned long time_interval);
```

*\*\*Trong đó `time_interval` là khoảng thời gian tính bằng tick muốn cài đặt tín hiệu đánh thức chương trình đang trong trạng thái ngủ.*

## **VII. Kết luận:**

Trong bài này chúng ta đã tìm hiểu rõ về cách quản lý thời gian trong *kernel*, thế nào là *jiffies*, *HZ*, ... vai trò ý nghĩa của chúng trong duy trì quản thời gian thực cũng như trong trì hoãn thời gian.

Chúng ta cũng đã tìm hiểu các hàm thao tác với thời gian thực trong *kernel*, với những hàm này chúng ta có thể xây dựng các ứng dụng có liên quan đến thời gian thực trong hệ thống.

Có nhiều cách khác nhau để thực hiện trì hoãn trong *kernel* tương tự như trong *user*. Nhưng chúng ta phải biết cách chọn phương pháp phù hợp để không làm ảnh hưởng đến hoạt động của toàn hệ thống.

Đến đây chúng ta có thể bước vào các bài thực hành, viết *driver* và ứng dụng cho một số phần cứng cơ bản trong những bài sau. Hoàn thành những bài này sẽ giúp cho chúng ta có được cái nhìn thực tế hơn về lập trình hệ thống nhúng.



**CHƯƠNG IV**  
**LẬP TRÌNH GIAO TIẾP**  
**NGOẠI VI**

**Lời đầu chương**

Sau khi nghiên cứu những nội dung trong chương III-Lập trình nhúng nâng cao, người học đã có những kiến thức cần thiết về lập trình user application và kernel driver để bắt tay vào viết ứng dụng điều khiển các thiết bị ngoại vi. Để quá trình nghiên cứu đạt hiệu quả cao nhất, trước khi đi vào từng bài thực hành trong chương này người học phải có những kiến thức và kỹ năng sau:

- Sử dụng thành thạo những phần mềm hỗ trợ lập trình nhúng như: SSH, Linux ảo, console putty, ... Tất cả đều được trình bày trong chương II-Lập trình nhúng căn bản.
- Biết cách biên dịch chương trình ứng dụng trong user bằng trình biên dịch gcc trong linux ảo và trực tiếp trong hệ thống linux của kit; Biên dịch chương trình driver bằng tập tin Makefile; Cách cài đặt driver và thực thi ứng dụng trong kit.
- Trình bày được các vấn đề có liên quan đến character device driver. Giải thích được các câu lệnh được sử dụng trong quá trình lập trình application cũng như lập trình driver như: các giao diện hàm, gpio, trì hoãn thời gian, ...

Chương này được trình bày bao gồm những bài tập thực hành riêng biệt nhau, được sắp xếp theo thứ tự từ dễ đến khó. Mỗi bài học sẽ nghiên cứu điều khiển một thiết bị ngoại vi hoặc có thể phối hợp với các thiết bị ngoại vi khác tùy theo yêu cầu điều khiển của bài toán. Nhằm mục đích cho người học ứng dụng ngay những kiến thức đã trong trong các chương trước, từ đó sẽ khắc sâu và áp dụng một cách thành thạo vào các trường hợp trong thực tế.

Các ngoại vi được trình bày trong chương là những module được tích hợp trong CHIP vi điều khiển hoặc được lắp đặt trong các bộ thí nghiệm khác. Các ngoại vi đó là: LED đơn, LED 7 đoạn, LCD, ADC on chip, UART, I2C, ... là những module đơn giản phù hợp với trình độ, giúp cho người học có thể hiểu và vận dụng vào những ứng dụng lớn khác nhanh chóng hơn.

Mỗi bài là một dự án thực hành được cấu trúc thành 3 phần: Phác thảo dự án, thực hiện dự án và kết luận-bài tập. Trong đó:

- Phần phác thảo dự án: Trình bày sơ lược về yêu cầu thực hiện trong dự án. Sau khi khái quát được yêu cầu, sẽ tiến hành phân công nhiệm vụ thực thi giữa hai thành phần user application và kernel driver sau cho dự án được hoạt động tối ưu trong hệ thống.
- Phần thực thi dự án: Bao gồm sơ đồ nguyên lý kết nối các chân gpio với phần cứng; mã chương trình tham khảo của driver và application, mỗi dòng lệnh đều có

những chú thích giúp người học hiểu được quá trình làm việc của chương trình, (mã lệnh chương trình hoàn chỉnh được lưu trong thư mục tham khảo của CD kèm theo đề tài, người học có thể chép vào chương trình soạn thảo và biên dịch thực thi).

- Phần kết luận-bài tập: Phần này sẽ tổng hợp lại những kiến thức kinh nghiệm lập trình mà dự án trình bày, nêu lên những nội dung chính trong bài học tới. Đồng thời chúng tôi cũng đưa ra những bài tập tham khảo giúp cho người học nắm vững kiến thức tạo nền tảng cho dự án mới.

*\*\*Mỗi bài tập thực hành được trình bày mang tính chất kế thừa, do đó người học cần tiến hành theo đúng trình tự được biên soạn để đạt hiệu quả cao nhất.*

## BÀI 1

# GIAO TIẾP ĐIỀU KHIỂN LED ĐƠN

### 1-1- ĐIỀU KHIỂN SÁNG TẮT 1 LED:

#### I. Phác thảo dự án:

Đây là dự án đầu tiên căn bản nhất trong quá trình lập trình điều khiển các thiết bị phần cứng. Người học có thể làm quen với việc điều khiển các chân *gpio* cho các mục đích khác nhau: truy xuất dữ liệu, cài đặt thông số đối với một chân vào ra trong vi điều khiển thông qua *driver* và chương trình ứng dụng. Để hoàn thành được bài này, người học phải có những kiến thức và kỹ năng sau:

- Kiến thức về mối quan hệ giữa *driver* và *application* trong hệ thống nhúng, cũng như việc trao đổi thông tin qua lại dựa vào các giao diện chuẩn;
- Kiến thức về giao diện chuẩn *ioctl* trong giao tiếp giữa *driver* (trong *kernel*) và *application* (trong *user*);
- Kiến thức về *gpio* trong *linux kernel*;
- Lập trình chương trình ứng dụng có sử dụng kỹ thuật hàm *main* có nhiều tham số giao tiếp với người dùng;
- Biên dịch và cài đặt được *driver*, *application* nạp vào hệ thống và thực thi;

*\*\*Tất cả những kiến thức yêu cầu nêu trên điều đã được chúng tôi trình bày kỹ trong những phần trước. Nếu cần người học có thể quay lại tìm hiểu để bước vào nội dung này hiệu quả hơn.*

#### a. Yêu cầu dự án:

Dự án này có yêu cầu là điều khiển thành công 1 led đơn thông qua *driver* và *application*. Người dùng có thể điều khiển led sáng tắt và đọc về trạng thái của một chân *gpio* theo yêu cầu nhập từ dòng lệnh *shell*.

- Đầu tiên, người dùng xác định chế độ vào ra cho chân *gpio* muốn điều khiển.
- Tiếp theo, nếu là chế độ ngõ vào thì sẽ xuất thông tin ra màn hình hiển thị cho biết trạng thái của chân *gpio* là mức thấp hay mức cao. Nếu là chế độ ngõ ra thì người dùng sẽ nhập thông tin *high* hoặc *low* để điều khiển led sáng tắt theo yêu cầu.

**\*\*Lưu ý,** nếu ngõ vào thì người dùng nên kết nối chân *gpio* với một công tắc điều khiển ON-OFF, nếu ngõ ra thì người dùng nên kết nối chân *gpio* với một LED đơn theo kiểu tích cực mức cao.

### **b. Phân công nhiệm vụ:**

- **Driver:** có tên `single_led_dev.c`

Sử dụng kỹ thuật giao diện *ioctl* để nhận lệnh và tham số từ *user application* thực thi điều khiển chân *gpio* theo yêu cầu. *ioctl* có 5 tham số lệnh tương ứng với 5 khả năng mà *driver* có thể phục vụ cho *application*:

- `GPIO_DIR_IN`: Cài đặt chân *gpio* là ngõ vào;
- `GPIO_DIR_OUT`: Cài đặt chân *gpio* là ngõ ra;
- `GPIO_GET`: Lấy dữ liệu mức logic từ chân *gpio* ngõ vào trả về một biến của *user application*;
- `GPIO_SET`: Xuất dữ liệu cho chân *gpio* ngõ ra theo thông tin lấy từ một biến trong *user application* tương ứng sẽ là mức thấp hay mức cao;

- **Application:** có tên `single_led_app.c`

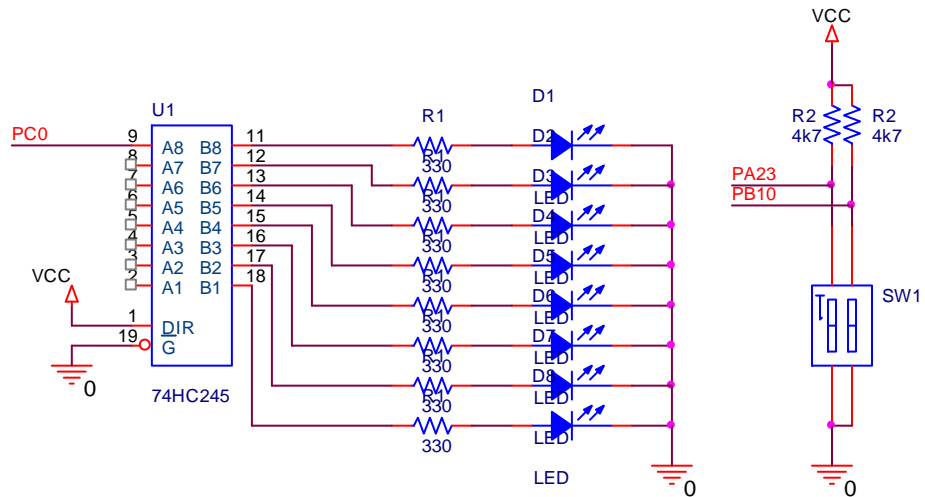
Sử dụng kỹ thuật lập trình hàm *main* có nhiều tham số lựa chọn cho người dùng khả năng điều khiển trên màn hình *shell* trong quá trình thực thi chương trình ứng dụng. Theo đó, chương trình ứng dụng `single_led_app` có những thao tác lệnh sau:

- Đầu tiên người dùng nhập tên chương trình cùng với các tham số mong muốn tương ứng với từng lệnh muốn thực thi.
- Nếu là lệnh `dirin`, người dùng phải cung cấp cho *driver* tham số tiếp theo là số chân *gpio* muốn cài đặt chế độ ngõ vào;
- Nếu là lệnh `dirout`, người dùng phải cung cấp cho *driver* tham số tiếp theo là số chân *gpio* muốn cài đặt chế độ ngõ ra;
- Nếu là lệnh `set`, thông tin tiếp theo phải cung cấp là 1 hoặc 0 và chân *gpio* muốn xuất dữ liệu;
- Nếu là lệnh `get`, thông tin tiếp theo người dùng phải cung cấp là số chân *gpio* muốn lấy dữ liệu. Sau khi lấy dữ liệu, xuất ra màn hình hiển thị thông báo cho người dùng biết.

## **II. Thực hiện:**

### **a. Kết nối phần cứng:**

Thực hiện kết nối phần cứng theo sơ đồ sau:



### b. Chương trình driver:

*/\*Khai báo thư viện cho các hàm sử dụng trong chương trình\*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
```

*/\*Định nghĩa tên driver thiết bị\*/*

```
#define DRVNAME      "single_led_dev"
#define DEVNAME      "single_led"
```

*/\*Định nghĩa số định danh lệnh cho giao diện ioctl\*/*

```
#define IOC_SINGLE_LED_MAGIC  'B'
#define GPIO_GET              _IO(IOC_SINGLE_LED_MAGIC, 10)
#define GPIO_SET              _IO(IOC_SINGLE_LED_MAGIC, 11)
#define GPIO_DIR_IN           _IO(IOC_SINGLE_LED_MAGIC, 12)
#define GPIO_DIR_OUT          _IO(IOC_SINGLE_LED_MAGIC, 13)
```

*/\* Counter is 1, if the device is not opened and zero (or less) if opened. \*/*

```
static atomic_t gpio_open_cnt = ATOMIC_INIT(1);
```

*/\*Khai báo và định nghĩa giao diện ioctl\*/*

```
static int
```

```
gpio_ioctl(struct inode * inode, struct file * file, unsigned int
cmd, unsigned long arg[])
{
```

```

int retval = 0;
/*Kiểm tra số định danh lệnh thực hiện theo yêu cầu*/
switch (cmd)
{
    /*Trong trường hợp là lệnh GPIO_GET*/
    case GPIO_GET:
        /*Lấy thông tin từ chân gpio*/
        retval = gpio_get_value(arg[0]);
        break;

    /*Trong trường hợp là GPIO_SET*/
    case GPIO_SET:
        /*Xuất dữ liệu arg[1] từ user application cho chân gpio arg[0]*/
        gpio_set_value(arg[0], arg[1]);
        break;

    /*Trong trường hợp là lệnh GPIO_DIR_IN*/
    case GPIO_DIR_IN:
        /*Yêu cầu truy xuất chân gpio arg[0]*/
        gpio_request (arg[0], NULL);
        /*Chân gpio arg[0] trong chế độ kéo lên*/
        at91_set_GPIO_periph (arg[0], 1);
        /*Cài đặt chân gpio arg[0] chế độ ngõ vào*/
        gpio_direction_input(arg[0]);
        break;

    /*Trong trường hợp là lệnh GPIO_DIR_OUT*/
    case GPIO_DIR_OUT:
        /*Yêu cầu truy xuất chân gpio arg[0]*/
        gpio_request (arg[0], NULL);
        /*Cài đặt kéo lên cho chân gpio arg[0]*/
        at91_set_GPIO_periph (arg[0], 1);
        /*Cài đặt chế độ ngõ ra cho chân gpio arg[0], giá trị khởi đầu là 0*/
        gpio_direction_output(arg[0], 0);
        break;

    /*Trường hợp không có lệnh thực thì, trả về mã lỗi*/
    default:
        retval = -EINVAL;
        break;
}

```

*/\*Trả về mã lỗi cho ioctl\*/*

```
    return retval;
```

```
}
```

*/\*Khai báo và định nghĩa giao diện open\*/*

```
static int
```

```
gpio_open(struct inode *inode, struct file *file)
```

```
{
```

```
    int result = 0;
```

```
    unsigned int dev_minor = MINOR(inode->i_rdev);
```

```
    if (!atomic_dec_and_test(&gpio_open_cnt)) {
```

```
        atomic_inc(&gpio_open_cnt);
```

```
        printk(KERN_ERR DRVNAME ": Device with minor ID %d already in  
use\n", dev_minor);
```

```
        result = -EBUSY;
```

```
        goto out;
```

```
    }
```

```
out:
```

```
    return result;
```

```
}
```

*/\*Khai báo và định nghĩa giao diện close\*/*

```
static int
```

```
gpio_close(struct inode * inode, struct file * file)
```

```
{
```

```
    smp_mb__before_atomic_inc();
```

```
    atomic_inc(&gpio_open_cnt);
```

```
    return 0;
```

```
}
```

*/\*Gán các giao diện vào file\_operations\*/*

```
struct file_operations gpio_fops = {
```

```
    .ioctl      = gpio_ioctl,
```

```
    .open       = gpio_open,
```

```
    .release    = gpio_close,
```

```
};
```

*/\*Cài đặt các thông số trên vào file\_node\*/*

```
static struct miscdevice gpio_dev = {
```

```
    .minor      = MISC_DYNAMIC_MINOR,
```

```
    .name       = "single_led",
```



```

        .fops          = &gpio_fops,
    };

/*Hàm khởi tạo ban đầu*/
static int __init
gpio_mod_init(void)
{
    return misc_register(&gpio_dev);
}

/*Hàm kết thúc khi tháo gỡ driver ra khỏi hệ thống*/
static void __exit
gpio_mod_exit(void)
{
    misc_deregister(&gpio_dev);
}

/*Gán các hàm khởi tạo init và kết thúc exit vào các macro cần thiết*/
module_init (gpio_mod_init);
module_exit (gpio_mod_exit);

/*Cập nhật các thông tin về chương trình*/
MODULE_LICENSE("GPL");
MODULE_AUTHOR("coolwarmboy");
MODULE_DESCRIPTION("Character device for for generic gpio api");

```

### **c. Chương trình application:**

```

/*Khai báo các thư viện sử dụng trong chương trình*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>

/*Định nghĩa số định danh lệnh sử dụng trong giao diện ioctl*/
#define IOC_SINGLE_LED_MAGIC  'B'
#define GPIO_GET              _IO(IOC_SINGLE_LED_MAGIC, 10)
#define GPIO_SET              _IO(IOC_SINGLE_LED_MAGIC, 11)
#define GPIO_DIR_IN           _IO(IOC_SINGLE_LED_MAGIC, 12)
#define GPIO_DIR_OUT          _IO(IOC_SINGLE_LED_MAGIC, 13)

/*Chương trình con in ra hướng dẫn cho người dùng, khi có lỗi xảy ra*/
void
print_usage()
{

```

```

    printf("single_led_app dirin|dirout|get|set gpio <value>\n");
    exit(0);
}

/*Chương trình chính main() khai báo theo dạng có tham số*/
int
main(int argc, char **argv)
{
    /*Số int lưu trữ số chân gpio*/
    int gpio_pin;

    /*Số mô tả tập tin, được trả về khi mở tập tin thiết bị*/
    int fd;

    /*Bộ nhớ đệm trao đổi dữ liệu qua lại giữa kernel và user trong giao diện ioctl*/
    unsigned long ioctl_buff[2];

    /*Biến trả về mã lỗi trong quá trình thực thi chương trình*/
    int result = 0;

    /*Mở tập tin thiết bị trước khi thao tác*/
    if ((fd = open("/dev/single_led", O_RDWR)) < 0)
    {
        /*In ra thông báo lỗi nếu quá trình mở thiết bị không thành công*/
        printf("Error whilst opening /dev/single_led_dev\n");
        return -1;
    }

    /*Chuyển tham số nhập từ người dùng thành số gpio lưu vào biến gpio_pin*/
    gpio_pin = atoi(argv[2]);

    /*Thông báo cho người dùng đang sử dụng chân gpio*/
    printf("Using gpio pin %d\n", gpio_pin);

    /*So sánh tham số nhập từ người dùng để biết phải thực hiện lệnh nào*/
    /*Trong trường hợp là lệnh "dirin" cài đặt chân gpio là ngõ vào*/
    if (!strcmp(argv[1], "dirin"))
    {
        /*Cập nhật bộ nhớ đệm trước khi chuyển qua kernel*/
        ioctl_buff[0] = gpio_pin;

        /*Sử dụng giao diện ioctl với lệnh GPIO_DIR_IN*/
        ioctl(fd, GPIO_DIR_IN, ioctl_buff);

        /*Trong trường hợp là lệnh "dirout" cài đặt chân gpio là ngõ ra*/
    } else if (!strcmp(argv[1], "dirout"))
    {

```

```

/*Cập nhật cùng nhớ đệm trước khi truyền sang kernel*/
    ioctl_buff[0] = gpio_pin;
    /*Dùng giao diện ioctl với lệnh GPIO_DIR_OUT */
    ioctl(fd, GPIO_DIR_OUT, ioctl_buff);
/*Trong trường hợp là lệnh “get” lấy dữ liệu từ chân gpio*/
    } else if (!strcmp(argv[1], "get"))
    {
        /*Cập nhật vùng nhớ đệm trước khi truyền sang kernel*/
        ioctl_buff[0] = gpio_pin;
        /*Sử dụng ioctl cập nhật thông tin trả về cho user*/
        result = ioctl(fd, GPIO_GET, ioctl_buff);
        /*In thông báo cho người sử dụng biết mức cao hay thấp của chân gpio*/
        printf("Pin %d is %s\n", gpio_pin, (result ? "HIGH" : "LOW"));
    }
/*Trong trường hợp là lệnh “set” xuất thông tin ra chân gpio*/
    } else if (!strcmp(argv[1], "set"))
    {
        /*Kiểm tra lỗi cú pháp*/
        if (argc != 4) print_usage();
        /*Cập nhật thông tin bộ nhớ đệm trước khi truyền cho kernel*/
        /*Cập nhật thông tin về số chân gpio*/
        ioctl_buff[0] = gpio_pin;
        /*Cập nhật thông tin mức muốn xuất ra chân gpio*/
        ioctl_buff[1] = atoi(argv[3]);
        /*Dùng giao diện ioctl để truyền thông điệp cho driver*/
        ioctl(fd, GPIO_SET, ioctl_buff);
    } else print_usage();
    return result;
}

```

#### **d. Biên dịch và thực thi dự án:**

- **Biên dịch driver:**

Trong thư mục chứa tập tin mã nguồn driver, tạo tập tin Makefile có nội dung sau:

```

export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabi-
obj-m += single_led_dev.o
all:

```

*/\*Lưu ý phải đúng đường dẫn đến cấu trúc mã nguồn kernel\*/*

```

make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) modules

```

clean:

*/\*Lưu ý phải đúng đường dẫn đến cấu trúc mã nguồn kernel\*/*

```
make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) clean
```

Biên dịch *driver* bằng lệnh *shell* như sau:

```
make clean all
```

*\*\*lúc này tập tin chương trình driver được tạo thành với tên single\_led\_dev.ko*

- Biên dịch *application*: Bằng lệnh *shell* sau:

```
./arm-none-linux-gnueabi-gcc single_led_app.c -o single_led_app
```

*\*\*Chương trình được biên dịch có tên là single\_led\_app*

- Thực thi chương trình:

➤ Chép *driver* và *application* vào kit;

➤ Cài đặt *driver* bằng lệnh: `insmod single_led_dev.ko`

➤ Thay đổi quyền thực thi cho chương trình *application* bằng lệnh:

```
chmod 777 single_led_app
```

➤ Chạy chương trình và quan sát kết quả:

- Khai báo chân PC0 là ngõ ra:

```
./single_led_app dirout 96
```

```
Using gpio pin 96
```

*(Lúc này led kết nối với PC0 tắt)*

- Xuất dữ liệu mức cao cho PC0:

```
./single_led_app set 96 1
```

```
Using gpio pin 96
```

*(Lúc này ta thấy led nối với chân PC0 sáng lên)*

- Xuất dữ liệu mức thấp cho PC0:

```
./single_led_app set 96 0
```

```
Using gpio pin 96
```

*(Lúc này ta thấy led nối với chân PC0 tắt xuống)*

- Khai báo chân PA23 là ngõ vào:

```
./single_led_app dirin 55
```

```
Using gpio pin 55
```

*\*\*Khi công tắc nối với PA23 ở vị trí ON, chân PA23 nối xuống mass;*

- Lấy dữ liệu vào từ chân PA23

```
./single_led_app get 55
```

```
Using gpio pin 55
```

```
Pin 55 is LOW
```

*\*\* Khi công tắc nối với PA23 ở vị trí OFF, chân PA23 nối lên VCC;*

- Lấy dữ liệu vào từ chân PA23

```
./single_led_app get 55
```

```
Using gpio pin 55
```

```
Pin 55 is HIGH
```

*\*\*Tương tự cho các chân gpio khác;*

### III. Kết luận và bài tập:

#### a. Kết luận:

Phần này các bạn đã nghiên cứu thành công các thao tác truy xuất chân *gpio* đơn lẻ. Kiến thức này sẽ làm nền cho các bài lớn hơn, truy xuất theo port 8 bits, hay điều khiển thiết bị ngoại vi bằng nút nhấn... Chúng ta sẽ tìm hiểu các kỹ thuật lập trình giao tiếp *gpio* khác với nhiều chân *gpio* cùng một lúc trong phần sau.

#### b. Bài tập:

1. Dựa vào các lệnh trong *driver single\_led\_dev.ko* hỗ trợ, hãy viết chương trình *application* cho 1 led sáng tắt với chu kỳ 1s trong 10 lần rồi ngưng.
2. Xây dựng chương trình *application* dựa vào *driver single\_led\_dev.ko* có sẵn để điều khiển 8 LEDS sáng tắt cùng một lúc với chu kỳ 1 s liên tục.
3. Xây dựng *driver* mới dựa vào *driver single\_led\_dev.ko* với yêu cầu: Thêm chức năng set 1 port 8 bit, và tắt 1 port 8 bit. Viết chương trình *application* sử dụng *driver* mới để thực hiện lại yêu cầu của bài 2.

## **1-2- ĐIỀU KHIỂN SÁNG TẮT 8 LED:**

### **I. Phác thảo dự án:**

Dự án này chủ yếu truy xuất chân *gpio* theo chế độ ngõ ra, nhưng điểm khác biệt so với dự án trước là không điều khiển riêng lẻ từng bit mà công việc điều khiển này sẽ do *driver* thực hiện. Phần này sẽ cho chúng ta làm quen với cách điều khiển thông tin theo từng *port 8 bits*. Để việc tiếp thu đạt hiệu quả cao nhất, trước khi nghiên cứu người học phải có những kiến thức và kỹ năng sau:

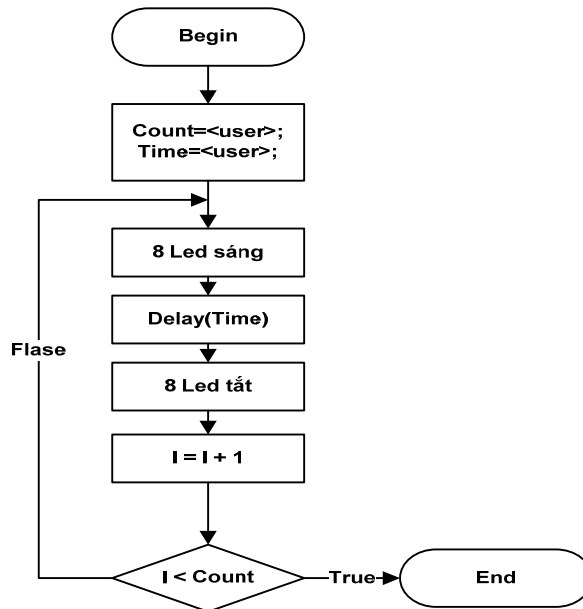
- Kiến thức tổng quát về mối quan hệ giữa *driver* và *application* trong hệ thống nhúng, cũng như việc trao đổi thông tin qua lại dựa vào các giao diện chuẩn;
- Kiến thức về giao diện chuẩn *write* trong giao tiếp giữa *driver* (trong *kernel*) và *application* (trong *user*);
- Kiến thức về *gpio* trong *linux kernel*;
- Lập trình chương trình ứng dụng có sử dụng kỹ thuật hàm *main* có nhiều tham số giao tiếp với người dùng;
- Biên dịch và cài đặt được *driver*, *application* nạp vào hệ thống và thực thi;

*\*\*Tất cả những kiến thức yêu cầu nêu trên điều đã được chúng tôi trình bày kỹ trong những phần trước. Nếu cần người học có thể quay lại tìm hiểu để bước vào nội dung này hiệu quả hơn.*

#### **a. Yêu cầu dự án:**

Yêu cầu của dự án là điều khiển thành công *1 port 8 leds* hoạt động chớp tắt cùng lúc theo chu kỳ và số lần được nhập từ người dùng trong lúc gọi chương trình thực thi. Khi hết nhiệm vụ chương trình sẽ được thoát và chờ lần gọi thực thi tiếp theo.

- Đầu tiên người dùng gọi chương trình *driver*, cung cấp thông tin về thời gian của chu kỳ và số lần nhấp nháy mong muốn;
- Chương trình *application* nhận dữ liệu từ người dùng, tiến hành điều khiển *driver* tác động vào ngõ ra *gpio* làm led sáng tắt theo yêu cầu;
- Lưu đồ điều khiển như sau:



### b. Phân công nhiệm vụ:

- **Driver:** Có tên là `port_led_dev.c`

*Driver* sử dụng giao diện `write()` nhận dữ liệu từ *user application* xuất ra led tương ứng với dữ liệu nhận được. Dữ liệu nhận từ *user application* là một số char có 8 bits. Mỗi bit tương ứng với 1 led cần điều khiển. Nhiệm vụ của *driver* là so sánh tương ứng từng bit trong số char này để quyết định xuất mức cao hay mức thấp cho led ngoại vi. Công việc của *driver* được thực hiện tuần tự như sau:

- Yêu cầu cài đặt các chân ngoại vi là ngõ ra, kéo lên. Công việc này được thực hiện khi thực hiện lệnh cài đặt *driver* vào hệ thống linux;
- Trong giao diện hàm `write()` (nhận dữ liệu từ *user*) thực hiện xuất ra mức cao hoặc mức thấp cho gpio điều khiển led.
- Giải phóng các chân gpio đã được khai báo khi không cần sử dụng, công việc này được thực hiện ngay trước khi tháo bỏ *driver* ra khỏi hệ thống.

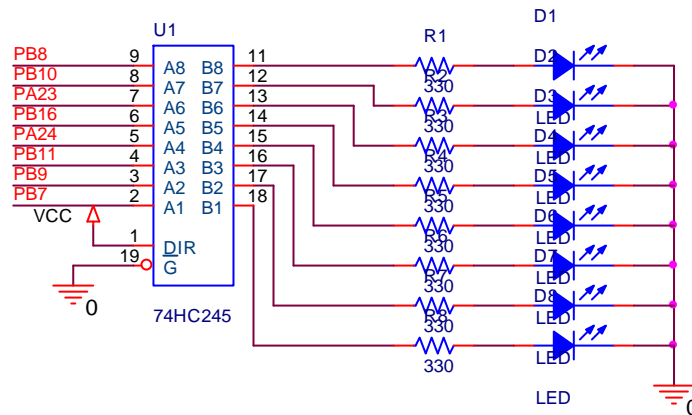
- **Application:** Có tên là `port_led_app.c`

Thực hiện khai báo hàm `main` theo cấu trúc tham số để đáp ứng các yêu cầu khác nhau từ người dùng. Chương trình *application* có hai tham số: Tham số thứ nhất là thời gian tính bằng giây của chu kỳ chớp tắt, tham số thứ hai là số chu kỳ muốn chớp tắt.

Bên cạnh đó, phần này còn lập trình thêm một số chương trình tạo hiệu ứng điều khiển led khác như: 8 led sáng dần tắt dần (Trái qua phải, phải qua trái, ...). Các chức năng này được tổng hợp trong một chương trình *application* duy nhất, người sử dụng sẽ lựa chọn hiệu ứng thông qua các tham số người dùng của hàm `main`.

## II. Thực hiện:

a. **Kết nối phần cứng:** Các bạn thực hiện kết nối phần cứng theo sơ đồ sau:



*\*\*Lưu ý phải đúng số chân đã quy ước.*

b. **Chương trình driver:** port\_led\_dev.c

*/\*Khai báo thư viện cần thiết cho các hàm sử dụng trong chương trình\*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
```

*/\*Đặt tên cho driver thiết bị\*/*

```
#define DRVNAME      "port_led_dev"
#define DEVNAME      "port_led"
```

*/\*Định nghĩa các chân sử dụng tương ứng với chân trong kit hỗ trợ \*/*

*/\*-----Port Control-----\*/*

```
#define P00          AT91_PIN_PB8
#define P01          AT91_PIN_PB10
#define P02          AT91_PIN_PA23
#define P03          AT91_PIN_PB16
#define P04          AT91_PIN_PA24
#define P05          AT91_PIN_PB11
#define P06          AT91_PIN_PB9
#define P07          AT91_PIN_PB7
```

*/\*Định nghĩa port từ các bit đã khai báo\*/*

```
#define P0            (P00|P01|P02|P03|P04|P05|P06|P07)
```



*/\*Khai báo các lệnh set và clear căn bản cho quá trình điều khiển port\*/*

*/\*Basic commands\*/*

```
#define SET_P00()                gpio_set_value(P00,1)
#define SET_P01()                gpio_set_value(P01,1)
#define SET_P02()                gpio_set_value(P02,1)
#define SET_P03()                gpio_set_value(P03,1)
#define SET_P04()                gpio_set_value(P04,1)
#define SET_P05()                gpio_set_value(P05,1)
#define SET_P06()                gpio_set_value(P06,1)
#define SET_P07()                gpio_set_value(P07,1)
```

```
#define CLEAR_P00()              gpio_set_value(P00,0)
#define CLEAR_P01()              gpio_set_value(P01,0)
#define CLEAR_P02()              gpio_set_value(P02,0)
#define CLEAR_P03()              gpio_set_value(P03,0)
#define CLEAR_P04()              gpio_set_value(P04,0)
#define CLEAR_P05()              gpio_set_value(P05,0)
#define CLEAR_P06()              gpio_set_value(P06,0)
#define CLEAR_P07()              gpio_set_value(P07,0)
```

*/\* Counter is 1, if the device is not opened and zero (or less) if opened. \*/*

```
static atomic_t port_led_open_cnt = ATOMIC_INIT(1);
```

*/\*Set và clear các bits trong port tương ứng với dữ liệu 8 bit nhận được\*/*

```
void port_led_write_data_port(char data)
{
    (data&(1<<0)) ? SET_P00() : CLEAR_P00();
    (data&(1<<1)) ? SET_P01() : CLEAR_P01();
    (data&(1<<2)) ? SET_P02() : CLEAR_P02();
    (data&(1<<3)) ? SET_P03() : CLEAR_P03();
    (data&(1<<4)) ? SET_P04() : CLEAR_P04();
    (data&(1<<5)) ? SET_P05() : CLEAR_P05();
    (data&(1<<6)) ? SET_P06() : CLEAR_P06();
    (data&(1<<7)) ? SET_P07() : CLEAR_P07();
}
```

*/\*Giao diện hàm write, nhận dữ liệu từ user để xuất thông tin ra port led\*/*

```
static ssize_t port_led_write (struct file *filp, char __iomem
buf[], size_t bufsize, loff_t *f_pos)
{
```

*/\*Sử dụng hàm xuất dữ liệu ra port led đã định nghĩa\*/*

```
    port_led_write_data_port(buf[0]);
```

```

        return bufsize;
    }
static int
port_led_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&port_led_open_cnt)) {
        atomic_inc(&port_led_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d already in
        use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

static int
port_led_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&port_led_open_cnt);
    return 0;
}

struct file_operations port_led_fops = {
    .write      = port_led_write,
    .open       = port_led_open,
    .release    = port_led_close,
};

static struct miscdevice port_led_dev = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = "port_led",
    .fops       = &port_led_fops,
};

static int __init
port_led_mod_init(void)

```

```

{
    /*Yêu cầu các chân gpio muốn sử dụng*/
    gpio_request (P00, NULL);
    gpio_request (P01, NULL);
    gpio_request (P02, NULL);
    gpio_request (P03, NULL);
    gpio_request (P04, NULL);
    gpio_request (P05, NULL);
    gpio_request (P06, NULL);
    gpio_request (P07, NULL);

    /*Khởi tạo các chân gpio có điện trở kéo lên*/
    at91_set_GPIO_periph (P00, 1);
    at91_set_GPIO_periph (P01, 1);
    at91_set_GPIO_periph (P02, 1);
    at91_set_GPIO_periph (P03, 1);
    at91_set_GPIO_periph (P04, 1);
    at91_set_GPIO_periph (P05, 1);
    at91_set_GPIO_periph (P06, 1);
    at91_set_GPIO_periph (P07, 1);

    /*Khởi tạo các chân gpio có chế độ ngõ ra, giá trị ban đầu là 0*/
    gpio_direction_output(P00, 0);
    gpio_direction_output(P01, 0);
    gpio_direction_output(P02, 0);
    gpio_direction_output(P03, 0);
    gpio_direction_output(P04, 0);
    gpio_direction_output(P05, 0);
    gpio_direction_output(P06, 0);
    gpio_direction_output(P07, 0);
    return misc_register(&port_led_dev);
}

static void __exit
port_led_mod_exit(void)
{
    misc_deregister(&port_led_dev);
}

module_init (port_led_mod_init);
module_exit (port_led_mod_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("coolwarmboy");

```

```
MODULE_DESCRIPTION("Character device for for generic gpio api");
```

**c. Chương trình application:** có tên là port\_led\_app.c

```
/*Khai báo các thư viện cần dùng cho các hàm trong chương trình*/
```

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>
```

```
/*Hàm in ra hướng dẫn thực thi lệnh trong trường hợp người dùng nhập sai cú pháp*/
```

```
void
print_usage()
{
    printf("port_led_app <TimePeriod> <NumberPeriod>\n");
    exit(0);
}
```

```
/*Khai báo hàm main có tham số cho người dùng*/
```

```
int
main(int argc, char **argv)
{
    /*Khai báo số mô tả tập tin cho driver khi được mở*/
    int port_led_fd;
    /*Khai báo vùng nhớ bộ đệm ghi cho giao diện hàm write()*/
    char write_buf[1];
    /*Biến điều khiển và lưu trữ thông tin người dùng*/
    int i, time_period, number_period;
    /*Mở driver và kiểm tra lỗi*/
    if ((port_led_fd = open("/dev/port_led", O_RDWR)) < 0)
    {
        /*Nếu có lỗi thì in ra thông báo và kết thúc chương trình*/
        printf("Error whilst opening /dev/port_led device\n");
        return -1;
    }
    /*Kiểm tra lỗi cú pháp từ người dùng*/
    if (argc != 3) {
        print_usage();
    }
}
```

```

/*Lấy chu kỳ thời gian nhập từ người dùng*/
time_period = atoi(argv[1]);
/*Lấy số chu kỳ mong muốn nhập từ người dùng*/
number_period = atoi(argv[2]);
/*Nạp thông tin cho vùng nhớ đệm*/
write_buf[0] = 0x00;
/*Thực hiện chớp tắt theo đúng số chu kỳ đã đặt*/
for (i=0; i < number_period; i++) {
/*Cập nhật thông tin của vùng nhớ đệm ghi driver*/
    write_buf[0] = ~(write_buf[0]);
/*Ghi thông tin của bộ đệm sang driver để ra port led*/
    write (port_led_fd, write_buf, 1);
/*Trì hoãn thời gian theo đúng chu kỳ người dùng nhập vào*/
    usleep(time_period*500000);
}
/*Trả về giá trị 0 khi không có lỗi xảy ra*/
return 0;
}

```

#### **d. Biên dịch và thực thi dự án:**

- ***Biên dịch driver:***

Tạo tập tin `Makefile` trong cùng thư mục với *driver*. Có nội dung sau:

```

export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabi-
obj-m += port_led_dev.o
all:
/*Lưu ý phải đúng đường dẫn đến cấu trúc mã nguồn kernel*/
make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) modules
clean:
/*Lưu ý phải đúng đường dẫn đến cấu trúc mã nguồn kernel*/
make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD) clean

```

- ***Biên dịch application:***

Trở vào thư mục chứa tập tin chương trình, biên dịch chương trình ứng dụng với lệnh sau:

```

arm-none-linux-gnueabi-gcc port_led_app.c -o port_led_app
**Chương trình biên dịch thành công có tên là: port_led_app

```

- ***Thực thi chương trình:***

Chép *driver* và chương trình vào kit, thực thi và kiểm tra kết quả;

- Cài đặt *driver* vào kit theo lệnh sau:

```
insmod port_led_dev.ko
```

- Thay đổi quyền thực thi cho chương trình ứng dụng:

```
chmod 777 port_led_app
```

- Thực thi và kiểm tra kết quả:

```
./port_led_app 1 10
```

*\*\*Chúng ta thấy 8 led nhấp nháy 10 lần với chu kỳ 1s. Các bạn thay đổi chu kỳ và số lần nhấp nháy quan sát kết quả.*

### III. Kết luận và bài tập:

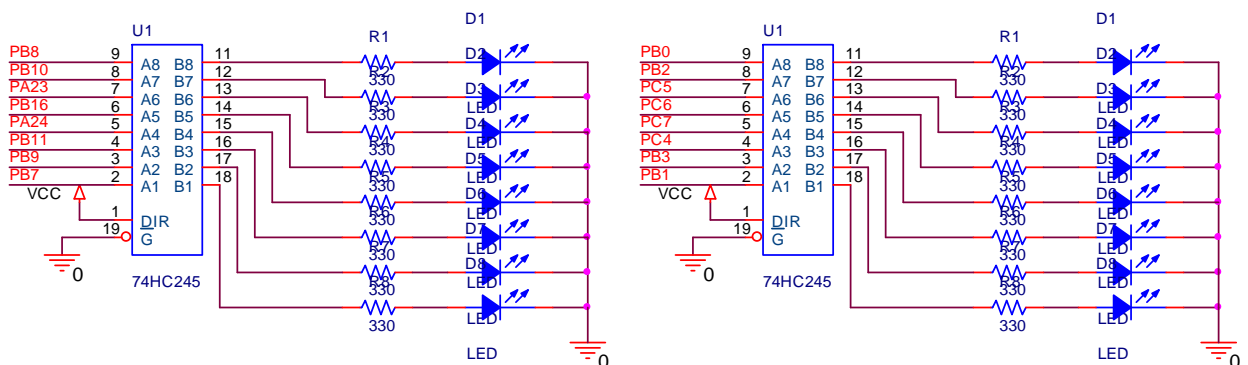
#### a. Kết luận:

Trong bài này chúng ta đã viết xong *driver* điều khiển 8 led đơn trong cùng một lúc tương ứng với dữ liệu nhận được từ *user application*. Chúng ta cũng đã viết một chương trình điều khiển led chớp tắt theo yêu cầu của người dùng. Trong những bài sau, *driver* này sẽ được áp dụng để lập trình các hiệu ứng điều khiển led khác.

*\*\*Do những thao tác biên dịch driver và application đã được chúng tôi trình bày rất kỹ trong phần lập trình hệ thống nhúng căn bản, hơn nữa cũng đã được nhắc lại một cách cụ thể trong những bài đầu tiên của lập trình thực hành điều khiển phân cứng, nên trong những bài tiếp theo sẽ không nhắc lại. Sau khi đã có mã nguồn của driver và application thì công việc còn lại là làm sao cho chúng có thể chạy được trên kit, ... thuộc về người học.*

#### b. Bài tập:

1. Mở rộng *driver* điều khiển 8 LED trên thành *driver* điều khiển 16 LED theo sơ đồ kết nối sau:



Nhiệm vụ của *driver* là nhận dữ liệu có chiều dài 16 bits từ *user application*. Sau đó xuất ra từng LED tương ứng với 16 bits dữ liệu.

2. Viết chương trình điều khiển 16 LEDs này chớp tắt theo yêu cầu của người sử dụng.  
(Về thời gian và số lượng chu kỳ muốn điều khiển).

### **1-3- SÁNG DẪN TẮT DẪN 8 LED:**

#### **I. Phác thảo dự án:**

Dự án này dựa vào *driver* đã có từ bài trước để viết chương trình ứng dụng tạo nhiều hiệu ứng chớp tắt 1 port 8 LEDs khác nhau như: Điều khiển sáng, tắt dần; Sáng dần; Điểm sáng dịch chuyển mất dần; ... Các bài tập này sẽ giúp cho người học làm quen với việc sử dụng *driver* đã xây dựng sẵn vào những chương trình ứng dụng khác nhau để hoàn thành một yêu cầu nào đó trong thực tế.

##### **a. Yêu cầu dự án:**

Dự án bao gồm 2 phần, *driver* giống như của bài lập trình sáng tắt 8 LEDs và applicaiton được lập trình sử dụng *driver* này để tạo ra các hiệu ứng hiển thị LEDs khác nhau. Chương trình *user application* cũng sử dụng kỹ thuật lập trình hàm main có tham số. Người sử dụng phải nhập theo đúng cú pháp để lựa chọn cho mình hiệu ứng LEDs.

Cú pháp đó là: <tên chương trình> <kiểu hiệu ứng> <chu kỳ> <số chu kỳ>

Trong đó:

<Tên chương trình> là tên chương trình đã được biên dịch từ mã nguồn;

<kiểu hiệu ứng> là hiệu ứng hiển thị LED, ở đây có 4 kiểu: *type1*, *type2*, *type3* và *type4*;

<chu kỳ> là khoảng thời gian (tính bằng *ms*) giữa 2 lần thay đổi trạng thái;

<số chu kỳ> là số chu kỳ lặp lại trạng thái.

##### **b. Phân công nhiệm vụ:**

- **Driver:** Có tên là `port_led_dev.c` đã được xây dựng trong bài trước.
- **Application:** Có tên là `1_3_OtherLedControl.c`. Bao gồm những chức năng sau:

##### **1. 8 LEDs sáng dần tắt hết (Dịch trái);**

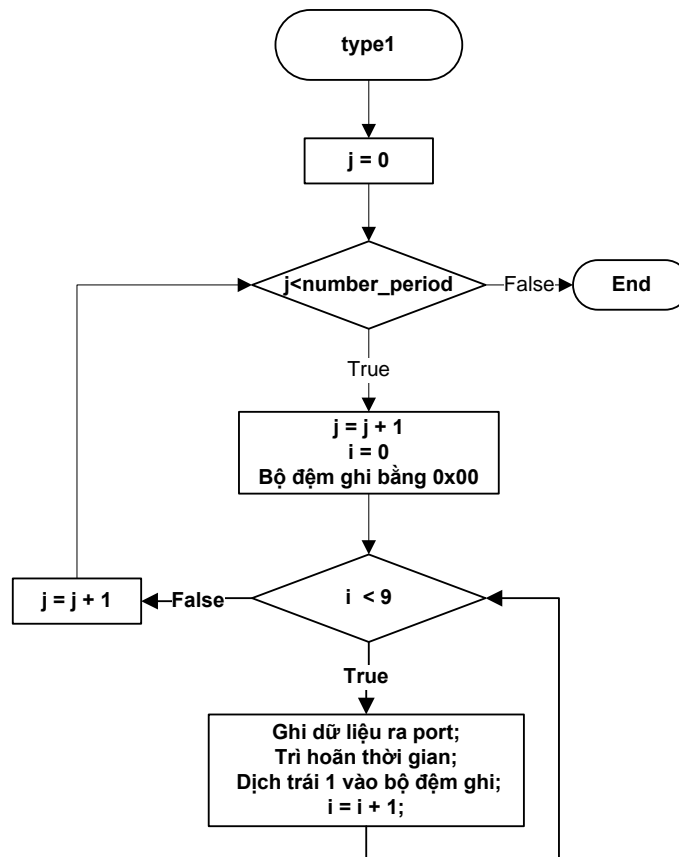
**Cách 1:** Tạo một mảng bao gồm có 9 trạng thái khác nhau của 8 bits sao cho có hiệu ứng sáng dần, sau đó cho 8 led tắt hết. Cứ tiếp tục theo đúng số chu kỳ người sử dụng muốn. Các trạng thái tương ứng như sau:



1: 00000000	4: 00000111	7: 00111111
2: 00000001	5: 00001111	8: 01111111
3: 00000011	6: 00011111	9: 11111111

(Quay lại từ đầu sau khi đã hiển thị hết dữ liệu, mỗi trạng thái đều có trì hoãn thời gian để quan sát được kết quả).

**Cách 2:** Áp dụng lệnh dịch (“<<” và “>>”) trong C để viết bài toán này. Chương trình được viết theo lưu đồ thuật toán sau:



## 2. 8 LEDs sáng dần tắt dần (Dịch trái);

**Cách 1:** Tạo một mảng bao gồm có 15 trạng thái khác nhau của 8 bits sao cho có hiệu ứng sáng dần, sau đó cho 8 led tắt dần. Cứ tiếp tục theo đúng số chu kỳ người sử dụng muốn. Các trạng thái tương ứng như sau:

1. 00000001	2. 00000011	3. 00000111
-------------	-------------	-------------

4. 00001111

5. 00011111

6. 00111111

7. 01111111

8. 11111111

9. 11111110

10. 11111100

11. 11111000

12. 11110000

13. 11100000

14. 11000000

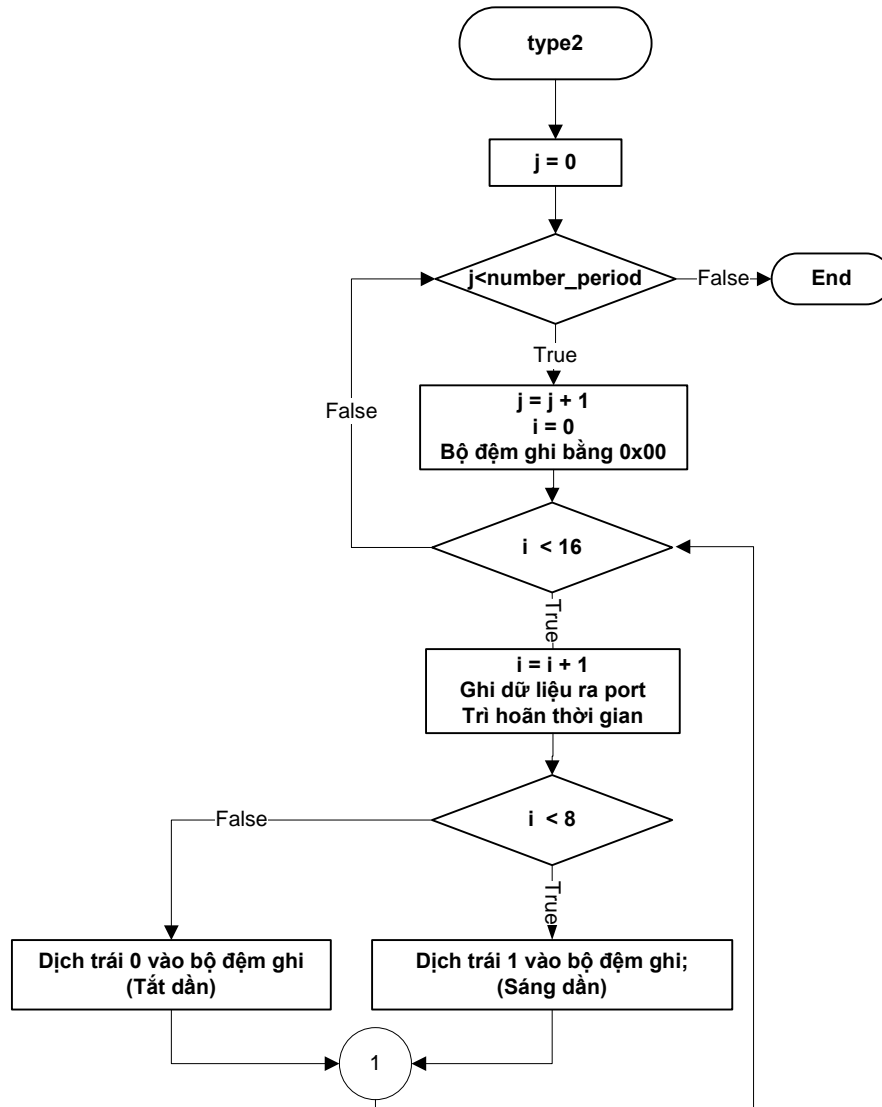
15. 10000000

16. Trạng thái 1.

(Tính là 1 chu kỳ);

(Quay lại từ đầu sau khi đã hiển thị hết dữ liệu, mỗi trạng thái đều có trì hoãn thời gian để quan sát được kết quả).

**Cách 2:** Áp dụng lệnh dịch (“<<” và “>>”) trong C để viết bài toán này. Chương trình được viết theo lưu đồ thuật toán sau:



### 3. 8 LEDs sáng dần:

**Cách 1:** Tạo một mảng bao gồm có 36 trạng thái khác nhau của 8 bits sao cho có hiệu ứng sáng dần. Cứ tiếp tục theo đúng số chu kỳ người sử dụng muốn. Các trạng thái tương ứng như mảng số hex 8 bits sau:

```

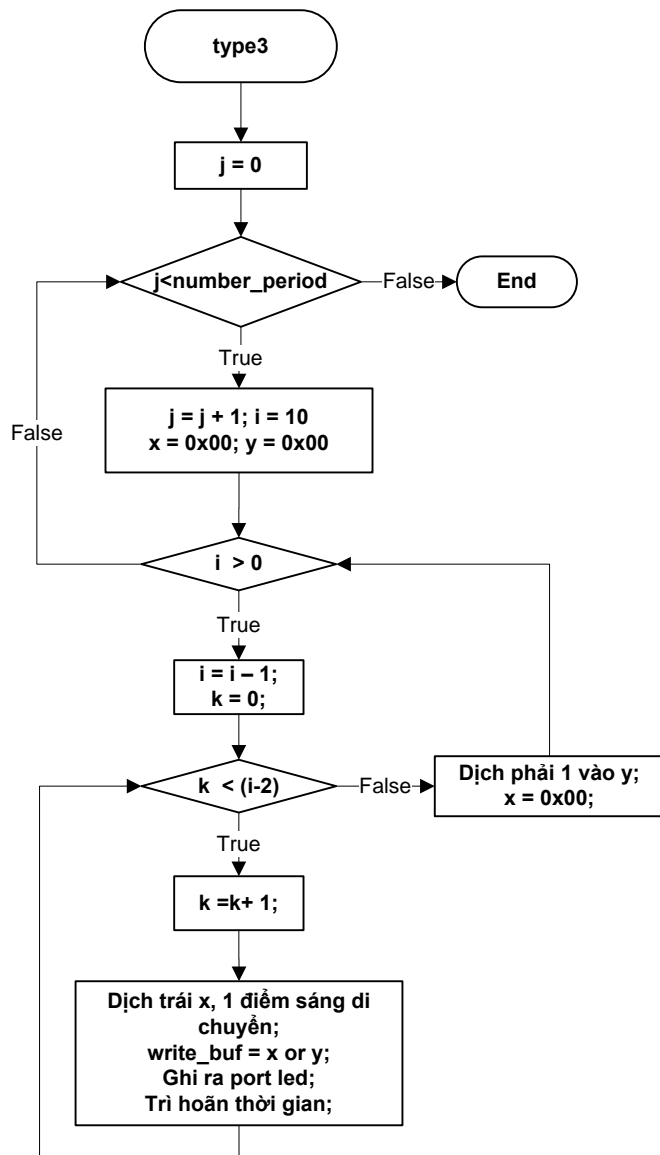
char Data_Display_Type_3[36] = {
    0x00,
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
    0x81, 0x82, 0x84, 0x90, 0xA0, 0xC0,
    0xC1, 0xC2, 0xC4, 0xC8, 0xD0, 0xE0,
    0xE1, 0xE2, 0xE4, 0xE8, 0xF0,

```

```
0xF1, 0xF2, 0xF4, 0xF8,  
0xF9, 0xFA, 0xFC,  
0xFD, 0xFE,  
0xFF  
}
```

*\*\*Mỗi một trạng thái là một giá trị trong mảng, chúng ta chỉ việc xuất các giá trị theo đúng thứ tự 0..35 để đạt được hiệu ứng mong muốn. (Quay lại từ đầu sau khi đã hiển thị hết dữ liệu, mỗi trạng thái đều có trì hoãn thời gian để quan sát được kết quả).*

**Cách 2:** Áp dụng lệnh dịch (“<<” và “>>”) trong C để viết bài toán này. Chương trình được viết theo lưu đồ thuật toán sau:



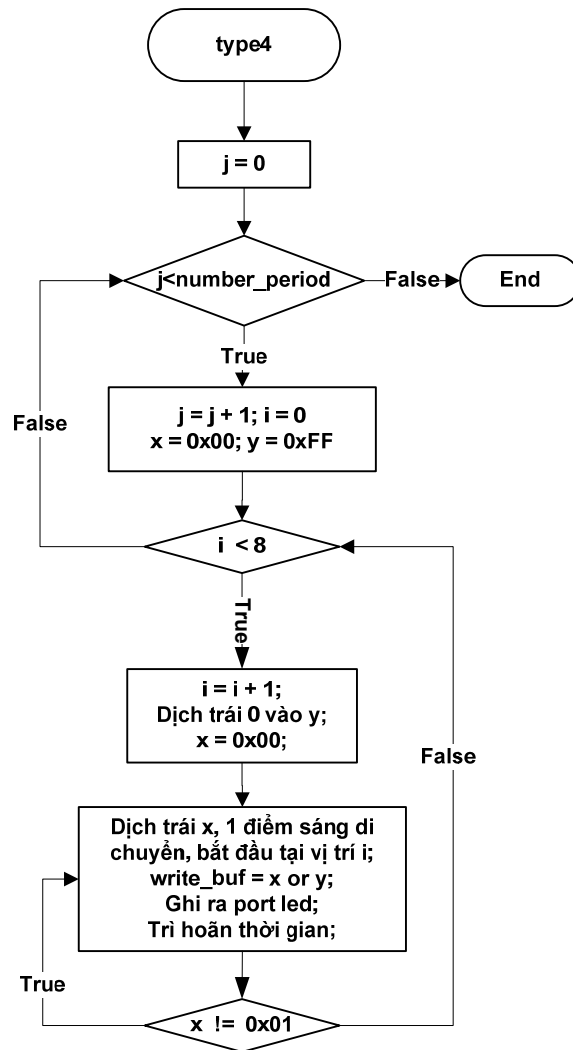
**4. 8 LEDs dịch chuyển mất dần;**

**Cách 1:** Tạo một mảng bao gồm có 36 trạng thái khác nhau của 8 bits sao cho có hiệu ứng dịch chuyển mất dần. Cứ tiếp tục theo đúng số chu kỳ người sử dụng muốn. Các trạng thái tương ứng như mảng số hex 8 bits sau:

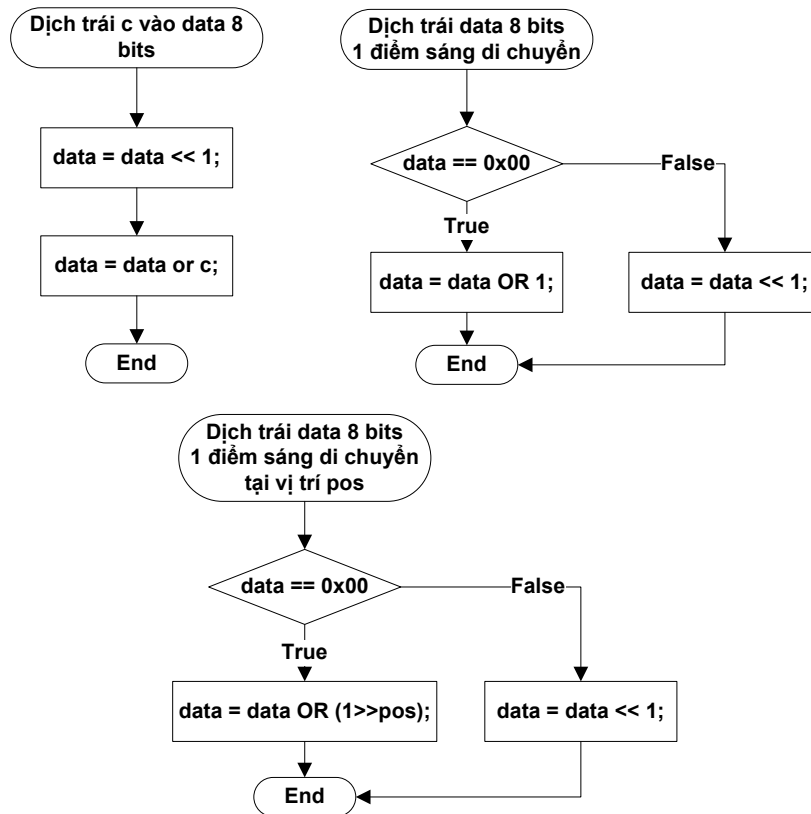
```
char Data_Display_Type_3[36] = {  
    0xFF,  
    0xFE, 0xFD,  
    0xFC, 0xFA, 0xF9,  
    0xF8, 0xF4, 0xF2, 0xF1,  
    0xF0, 0xE8, 0xE4, 0xE2, 0xE1,  
    0xE0, 0xD0, 0xC8, 0xC4, 0xC2, 0xC1,  
    0xC0, 0xA0, 0x90, 0x84, 0x82, 0x81,  
    0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01  
}
```

**\*\*Mỗi một trạng thái là một giá trị trong mảng, chúng ta chỉ việc xuất các giá trị theo đúng thứ tự 0..35 để đạt được hiệu ứng mong muốn.** (Quay lại từ đầu sau khi đã hiển thị hết dữ liệu, mỗi trạng thái đều có trì hoãn thời gian để quan sát được kết quả).

**Cách 2:** Áp dụng lệnh dịch (“<<” và “>>”) trong C để viết bài toán này. Chương trình được viết theo lưu đồ thuật toán sau:

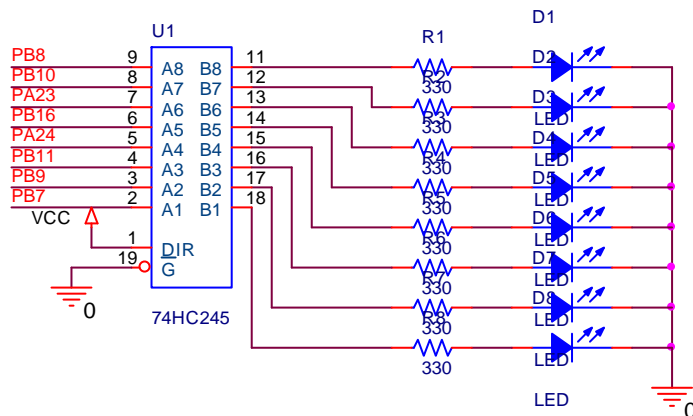


- **Các hàm dịch bit:** Một điểm sáng di chuyển, một điểm sáng di chuyển tại vị trí pos, dịch trái bit c vào data được thực hiện theo những lưu đồ sau:



## II. Thực hiện:

a. **Kết nối phần cứng:** Các bạn thực hiện kết nối phần cứng theo sơ đồ sau:



*\*\*Lưu ý phải đúng số chân đã quy ước.*

b. **Chương trình driver:** Tương tự như bài trước;

c. **Chương trình application:**

**Cách 1:** Điều khiển 8 LEDs bằng cách xuất trình tự các dữ liệu trạng thái ra port;

*/\*Chương trình mang tên 1\_3\_OtherPortControl\_Method\_1.c\*/*

*/\*Khai báo thư viện cần thiết cho các lệnh trong chương trình\*/*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>

/*Dữ liệu trạng thái LEDs cho hiệu ứng 1, 8 LEDs sáng dần và tắt hết*/
char Data_Display_Type_1[9] = {
0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF };

/*Dữ liệu trạng thái LEDs cho hiệu ứng 2, 8 LEDs sáng dần và tắt dần*/
char Data_Display_Type_2[15] = {
0x00, 0x01, 0x03, 0x07, 0x0F, 0x1F, 0x3F, 0x7F, 0xFF,
0xFE, 0xFC, 0xF8, 0xE0, 0xC0, 0x80 };

/*Dữ liệu trạng thái LEDs cho hiệu ứng 3, 8 LEDs sáng dần*/
char Data_Display_Type_3[36] = {
0x00,
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x81, 0x82, 0x84, 0x90, 0xA0, 0xC0,
0xC1, 0xC2, 0xC4, 0xC8, 0xD0, 0xE0,
0xE1, 0xE2, 0xE4, 0xE8, 0xF0,
0xF1, 0xF2, 0xF4, 0xF8,
0xF9, 0xFA, 0xFC,
0xFD, 0xFE,
0xFF
};

/*Dữ liệu trạng thái LEDs cho hiệu ứng 4, 8 LEDs dịch chuyển mát dần*/
char Data_Display_Type_4[36] = {
0xFF,
0xFE, 0xFD,
0xFC, 0xFA, 0xF9,
0xF8, 0xF4, 0xF2, 0xF1,
0xF0, 0xE8, 0xE4, 0xE2, 0xE1,
0xE0, 0xD0, 0xC8, 0xC4, 0xC2, 0xC1,
0xC0, 0xA0, 0x90, 0x84, 0x82, 0x81,
0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01
```



```
};  
/*Biến lưu số mô tả tập tin cho driver khi được mở*/  
int port_led_fd;  
/*Bộ nhớ đệm dữ liệu cần ghi vào driver*/  
char write_buf[1];  
/*Chương trình cho in thông báo hướng dẫn sử dụng khi có lỗi cú pháp từ người dùng*/  
void print_usage()  
{  
printf ( "OtherPortControl <type1|type2|type3|type4>  
<TimePeriod> <NumberPeriod>\n");  
exit(0);  
}  
/*Hàm main được khai báo dạng tham số lựa chọn, để lấy thông tin người dùng*/  
int  
main(int argc, char **argv)  
{  
/*Biến time_period lưu thời gian thay đổi trạng thái;  
biến number_period lưu số chu kỳ cần thực hiện*/  
long int time_period, number_period;  
/*Các biến đếm phục vụ cho truy xuất dữ liệu trạng thái*/  
int j;  
char i;  
/*Mở tập tin thiết bị trước khi thao tác*/  
if ((port_led_fd = open("/dev/port_led", O_RDWR)) < 0)  
{  
printf("Error whilst opening /dev/port_led  
device\n");  
return -1;  
}  
/*Kiểm tra cú pháp nhập từ người dùng*/  
if (argc != 4) {  
print_usage();  
printf("%d\n",argc);  
}
```

```
    }  
    /*Lấy thời gian trì hoãn từ người dùng*/  
    time_period = atoi(argv[2])*500;  
    /*Lấy số chu kỳ cần lặp lại từ người dùng*/  
    number_period = atoi(argv[3]);  
    /*So sánh thực hiện từng kiểu hiệu ứng khác nhau dựa vào tham số  
    nhập từ người dùng*/  
    /*Trường hợp 1: Hiệu ứng 8 LEDs sáng dần tắt hết*/  
    if (!strcmp(argv[1], "type1")) {  
        for (j=0; j < number_period; j++) {  
            for (i=0; i < 9; i++) {  
                /*Cập nhật thanh ghi đếm bằng dữ liệu trạng  
                thái*/  
                write_buf[0] = Data_Display_Type_1[i];  
                /*Xuất thanh ghi đếm sang driver*/  
                write(port_led_fd, write_buf, 1);  
                /*Trì hoãn thời gian bằng time_period (ms)*/  
                usleep(time_period);  
            };  
        };  
        /*Trường hợp 2: Hiệu ứng 8 LEDs sáng dần tắt dần*/  
        /*Phương pháp điều khiển cũng tương tự như trường hợp đầu tiên*/  
        } else if (!strcmp(argv[1], "type2")) {  
            for (j=0; j < number_period; j++) {  
                for (i=0; i < 15; i++) {  
                    write_buf[0] = Data_Display_Type_2[i];  
                    write(port_led_fd, write_buf, 1);  
                    usleep(time_period);  
                };  
            };  
            /*Trường hợp 3: Hiệu ứng 8 LEDs sáng dòn*/  
            } else if (!strcmp(argv[1], "type3")) {  
                for (j=0; j < number_period; j++) {  
                    for (i=0; i < 36; i++) {  
                        write_buf[0] = Data_Display_Type_3[i];
```

```
        write(port_led_fd, write_buf, 1);
        usleep(time_period);
    };
};

/*Trường hợp 4: Hiệu ứng 8 LEDs dịch chuyển mắt dần*/
    } else if (!strcmp(argv[1], "type4")) {
        for (j=0; j < number_period; j++) {
            for (i=0; i < 36; i++) {
                write_buf[0] = Data_Display_Type_4[i];
                write(port_led_fd, write_buf, 1);
                usleep(time_period);
            };
        };

/*In ra thông báo lỗi trong trường hợp không có lệnh nào hỗ trợ*/
        } else {
            print_usage();
        }
    }
    return 0;
}
```

**Cách 2:** Điều khiển 8 led bằng lệnh dịch (>> và <<);

```
/*Chương trình mang tên 1_3_OtherPortControl_Method_2.c*/
/*Khai báo thư viện cần thiết cho các lệnh dùng trong chương trình*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>

/*Chương trình cho in thông báo hướng dẫn sử dụng khi có lỗi cú pháp
từ người dùng*/
void
print_usage()
{
    printf("OtherPortControl <type1|type2|type3|type4>
<TimePeriod> <NumberPeriod>\n");
    exit(0);
}
```

```
}

/*Hàm dịch trái “cờ c” vào data 8 bits*/
char shift_left_1_8bits_c(char data, char c) {
    return ((data<<1) | c);
}

/*Hàm dịch phải “cờ c” vào data 8 bits*/
char shift_right_1_8bits_c(char data, char c) {
    return ((data>>1) | (c<<7));
}

/*Hàm dịch trái 1 điểm sáng di chuyển trong data 8 bits*/
char shift_left_1_lighting_led_8bits(char data) {
    if (data == 0) {
        return (data | 1);
    } else {
        return (data<<1);
    }
}

/*Hàm dịch phải một điểm sáng di chuyển trong data 8 bits*/
char shift_right_1_lighting_led_8bits(char data) {
    if (data == 0) {
        return (data | 0x80);
    } else {
        return (data>>1);
    }
}

/*Hàm dịch trái một điểm sáng di chuyển trong data 8 bits tại vị trí pos
(bit 0: pos=1, bit 1: pos=2, ...*/
char shift_left_1_lighting_at_position_led_8bits(char
data, char pos) {
    if (data == 0) {
        return (data | (1>>pos));
    } else {
        return (data<<1);
    }
}
```

```
/*Hàm dịch phải một điểm sáng di chuyển trong data 8 bits tại vị trí pos
(bit 0: pos=1, bit 1: pos=2, ...*/
char      shift_right_1_lighting_at_position_led_8bits(char
data, char pos) {
    if (data == 0) {
        return (data | (1<<pos));
    } else {
        return (data>>1);
    }
}

/*Chương trình chính main() khai báo dưới dạng tham số nhập từ người
dùng*/
int
main(int argc, char **argv)
{
    /*Biến lưu số mô tả tập tin thiết bị khi nó được mở*/
    int port_led_fd;
    /*Thanh ghi đếm ghi vào driver và các biến phụ thao tác bit*/
    char write_buf[1], x, y;
    /*Biến lưu trữ thời gian và số chu kỳ người dùng mong muốn*/
    long int time_period, number_period;
    /*Các biến đếm hỗ trợ thao tác dữ liệu*/
    int j;
    char i,k,l;
    /*Mở tập tin thiết bị trước khi thao tác*/
    if ((port_led_fd = open("/dev/port_led", O_RDWR)) < 0)
    {
        printf("Error whilst opening /dev/port_led
device\n");
        return -1;
    }
    /*Kiểm tra lỗi cú pháp từ người dùng*/
    if (argc != 4) {
        print_usage();
        printf("%d\n",argc);
    }
}
```

```
}
/*Lấy thời gian chu kỳ từ người dùng*/
time_period = atoi(argv[2])*500;
/*Lấy số chu kỳ mong muốn từ người dùng*/
number_period = atoi(argv[3]);
/*So sánh các trường hợp lệnh nhập từ người dùng*/
/*Trường hợp 1: Hiệu ứng LEDs sáng dần tắt hết*/
if (!strcmp(argv[1], "type1")) {
    for (j=0; j < number_period; j++) {
        write_buf[0] = 0x00;
        for (i=0; i<8; i++) {
            write(port_led_fd, write_buf, 1);
            usleep(time_period);

write_buf[0]=shift_left_1_8bits_c(write_buf[0],1);
        }
    };
/*Trường hợp 2: Hiệu ứng LEDs di chuyển sáng dần sau đó tắt dần*/
} else if (!strcmp(argv[1], "type2")) {
    for (j=0; j < number_period; j++) {
        write_buf[0] = 0x00;
        for (i=0; i<16; i++) {
            write(port_led_fd, write_buf, 1);
            usleep(time_period);
            if (i<8) {

write_buf[0]=shift_left_1_8bits_c(write_buf[0],1);
            } else {

write_buf[0]=shift_left_1_8bits_c(write_buf[0],0);
            }
        }
    };
/*Trường hợp 3: Hiệu ứng LEDs di chuyển sáng dần*/
} else if (!strcmp(argv[1], "type3")) {
    for (j=0; j < number_period; j++) {
```

```
x = 0x00;
y = 0x00;
for (i=10;i>0;i--) {
    for (k=0;k<(i-2);k++) {
        x=shift_left_1_lighting_led_8bits
        (x);
        write_buf[0] = x | y;
        write(port_led_fd, write_buf, 1);
        usleep(time_period);
    }
    y = shift_right_1_8bits_c (y,1);
    x = 0x00;
}
}

/*Trường hợp 4: Hiệu ứng LEDs di chuyển mất dần*/
} else if (!strcmp(argv[1], "type4")) {
    for (j=0; j < number_period; j ++) {
        x = 0x00;
        y = 0xFF;
        for (i=0;i<8;i++) {
            y = shift_left_1_8bits_c (y,0);
            x = 0x00;

            do {
                x=
                shift_right_1_lighting_at_position_led_8bits(x,i);
                write_buf[0] = x | y;
                write(port_led_fd, write_buf, 1);
                usleep(time_period);
            } while (x != 0x01);
        }
    }
} else {
    print_usage();
}
return 0;
}
```

**d. Biên dịch và thực thi dự án:**

- **Biên dịch driver:** Mang tên `port_led_dev.ko`

Chép *driver* đã biên dịch thành công trong bài trước vào kit để chuẩn bị cài đặt vào hệ thống.

- **Biên dịch application:** Mang tên `1_3_Othercontrol_Method_1.c` và `1_3_Othercontrol_Method_2.c`

Trở vào thư mục chứa tập tin chương trình, biên dịch chương trình ứng dụng với lệnh sau:

```
arm-none-linux-gnueabi-gcc      1_3_Othercontrol_Method_1.c      -o
Othercontrol_Method_1
arm-none-linux-gnueabi-gcc      1_3_Othercontrol_Method_2.c      -o
Othercontrol_Method_2
```

**\*\*Chương trình biên dịch thành công có tên là:** `Othercontrol_Method_1` và `Othercontrol_Method_2`. **\*\*\*Lưu ý:** câu lệnh trong shell phải viết trên cùng một dòng.

- **Thực thi chương trình:**

Chép *driver* và chương trình vào kit, thực thi và kiểm tra kết quả;

- Cài đặt *driver* vào kit theo lệnh sau:

```
insmod port_led_dev.ko
```

- Thay đổi quyền thực thi cho chương trình ứng dụng:

```
chmod 777 Othercontrol_Method_1
```

```
chmod 777 Othercontrol_Method_2
```

- Thực thi và kiểm tra kết quả: Thực thi chương trình ứng dụng theo cú pháp được quy định, quan sát và kiểm tra kết quả.

**\*\*Chúng ta thấy 8 led nhấp nháy 10 lần với chu kỳ 1s. Các bạn thay đổi chu kỳ và số lần nhấp nháy quan sát kết quả.**

Như vậy, để tạo ra nhiều hiệu ứng điều khiển LEDs khác nhau, việc đơn giản nhất là chỉ việc tạo ra các trạng thái điều khiển mong muốn, lưu vào một vùng nhớ nào đó, công việc cuối cùng là định thời xuất các trạng thái đó ra LEDs hiển thị. Nhưng cách này chỉ hiệu quả khi số lượng LEDs điều khiển là nhỏ và trạng thái LEDs xuất ra không mang tính quy luật rõ ràng. Đối với những trường hợp: số lượng LEDs điều khiển lớn, hiệu ứng mang tính quy luật thì cách thứ 2 là phù hợp



nhất. Chúng ta phải áp dụng cách nào là tùy trường hợp, sao cho tốn ít dung lượng bộ nhớ và thời gian thực hiện là nhỏ nhất.

### III. Kết luận và bài tập:

#### a. Kết luận:

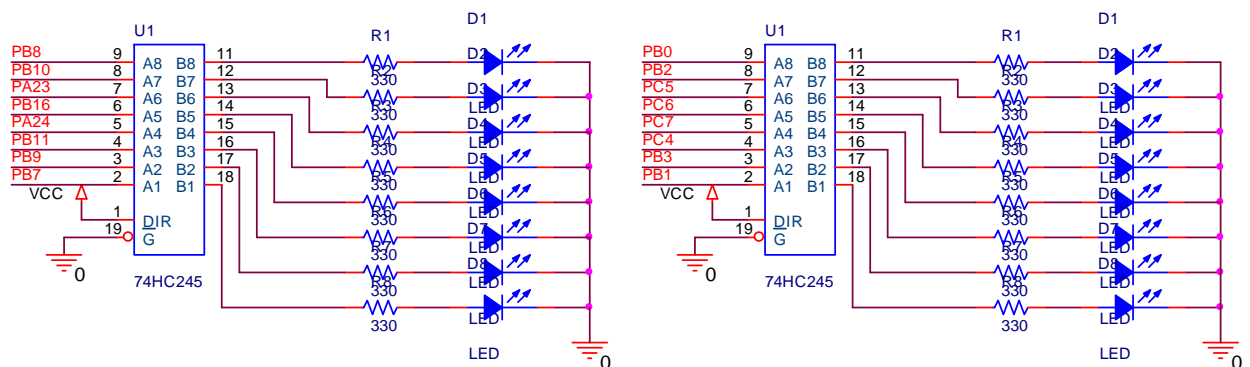
Trong bài này, chúng ta đã viết thành công các chương trình ứng dụng tạo ra nhiều hiệu ứng LEDs khác nhau dựa vào một driver điều khiển LEDs đã có. Đây cũng là mục đích của quyển sách này: Với việc thực hành viết driver và chương trình ứng dụng người học có thể tự mình nghiên cứu một driver có sẵn để viết chương trình ứng dụng phục vụ nhu cầu trong thực tế.

Đến đây chúng ta sẽ tạm ngưng công việc điều khiển LEDs đơn để bước sang một đối tượng khác. Đó là LEDs 7 đoạn, được ứng dụng rất nhiều trong hiển thị thông tin số thập phân.

#### b. Bài tập:

1. Dựa vào driver điều khiển 16 LED đã được hoàn thành trong bài luyện tập trước để viết ứng dụng tạo ra các hiệu ứng điều khiển LEDs tương tự như trong ví dụ trên (nhưng là điều khiển 16 LEDs đơn).

Driver điều khiển theo sơ đồ kết nối sau:



2. Viết chương trình điều khiển 8 LEDs với cùng yêu cầu như ví dụ trên nhưng có chiều thay đổi, chuyển từ dịch trái sang dịch phải, và ngược lại.
3. Kết hợp các hiệu ứng trên thành một chuỗi hiệu ứng liên tục, kế tiếp nhau không cần phải qua tham số chọn lựa hiệu ứng.
4. Hãy viết chương trình điều khiển 8, 16 LEDs sáng dần từ ngoài vào trong và từ trong ra ngoài.

5. Hãy viết chương trình điều khiển 8, 16 LEDs dịch chuyển mất dần từ ngoài vào trong và từ trong ra ngoài.

### 1-4- CÀI ĐẶT THỜI GIAN DÙNG TIMER:

#### I. Phác thảo dự án:

Trong những bài trước, để thực hiện trì hoãn thời gian chúng ta thường dùng kỹ thuật dùng các hàm trì hoãn thời gian hỗ trợ sẵn trong *user application*. Thao tác với thời gian còn có một ứng dụng khác rất quan trọng là định thời gian thực hiện tác vụ. Công việc định thời có thể mang tính chu kỳ hoặc không mang tính chu kỳ. Trong bài này chúng ta sẽ áp dụng kỹ thuật định thời gian mang tính chất chu kỳ để cập nhật điều khiển trạng thái LEDs, thay vì dùng kỹ thuật trì hoãn thời gian thông thường.

Định thời gian trong hệ thống Linux có hai cách thực hiện, định thời trong *kernel driver* (không có sự quản lý của hệ điều hành) và định thời trong *user application* (có sự quản lý của hệ điều hành). Cả hai cách đều được sử dụng trong bài này.

#### a. Yêu cầu dự án:

Nội dung điều khiển LEDs trong bài này không có gì mới, đơn giản vẫn là điều khiển LEDs chớp tắt theo chu kỳ thời gian được quy định bởi người sử dụng khi gọi chương trình ứng dụng thực thi.

Người sử dụng gọi chương trình thực thi theo cú pháp sau:

<tên chương trình> <chu kỳ>

Trong đó:

<tên chương trình> là tên chương trình sau khi được biên dịch;

<chu kỳ> là thời gian tính bằng giây do người sử dụng nhập vào;

#### b. Phân công nhiệm vụ:

##### 1. Timer trong user application:

- *Driver*:

Làm nhiệm vụ nhận dữ liệu 8 bits từ *user application* để xuất ra LEDs hiển thị, tương tự như *driver port\_led\_dev.ko* trong bài trước.

- *Application*:

Nhận thông tin khoảng thời gian cập nhật thay đổi trạng thái LEDs. Thông tin này sẽ được chuyển thành thời gian định thời. *User application* sử dụng hàm `alarm()` và hàm `signal()` để khởi tạo định thời. Hàm `alarm()` làm

nhiệm vụ cài đặt thời gian xuất hiện tín hiệu ngắt SIGALRM. Khi xảy ra tín hiệu ngắt SIGALRM, hàm `signal()` sẽ đón bắt và thực thi hàm phục vụ ngắt được người lập trình quy định.

## 2. Timer trong kernel driver:

- *Driver:*

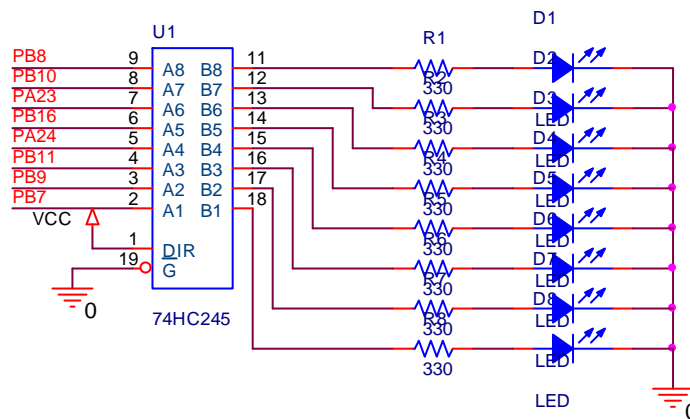
Nhận thông tin khoảng thời gian (tính bằng s) từ người dùng thông qua *user application*, cài đặt định thời thông qua ngắt thời gian dùng *timer* mềm. Đầu tiên *driver* sẽ tiến hành khởi tạo *timer* dựa vào các hàm thao tác với *timer* đã được tìm hiểu trong những bài trước. Sau mỗi lần đến thời gian định thời, *driver* cập nhật trạng thái LEDs. *Driver* được thay đổi thời gian cập nhật khi giao diện hàm `write()` được gọi bởi *user application*.

- *Application:*

Lúc này nhiệm vụ của *user application* rất đơn giản, chỉ dùng để cập nhật thời gian định thời cho *driver* thay đổi trạng thái LEDs hiển thị bằng cách gọi giao diện hàm `write()` mỗi khi chương trình thực thi.

## II. Thực hiện:

Kết nối phần cứng theo sơ đồ sau:



### 1. Timer trong user application:

a. *Chương trình driver:* Là *driver* `port_led_dev.ko` trong bài trước;

b. *Chương trình user application:* có tên `1_4_1_TimerLedControl.c`

*/\*Khai báo thư viện cần dùng cho các hàm trong chương trình\*/*

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
#include <signal.h> //Thư viện cho hàm signal()
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>
/*Biến lưu số mô tả tập tin thiết bị được mở khi thao tác*/
int port_led_fd;
/*Bộ đệm ghi dữ liệu sang driver xuất ra LEDs*/
char write_buf[1];
/*Biến lưu khoảng thời gian định thời*/
long int time_period;
/*Biến lưu trạng thái đảo LEDs*/
int i=0;
/*Hàm in ra hướng dẫn cho người dùng trong trường hợp nhập sai cú pháp*/
void
print_usage()
{
    printf("timer_led_control_app <TimePeriod>\n");
    exit(0);
}
/*Hàm xử lý tín hiệu ngắt SIGALRM được cài đặt vào hàm signal()*/
void catch_alarm(int sig_num) {
    if (i == 0) {
        i = 1;
        /*In thông báo LEDs tắt*/
        printf ("LEDs are off\n");
        /*Cập nhật trạng thái bộ ghi đệm*/
        write_buf[0] = 0x00;
    } else {
        i = 0;
        /*In thông báo LEDs sáng*/
        printf ("LEDs are on\n");
        /*Cập nhật trạng thái bộ nhớ đệm */
    }
}
```

```
        write_buf[0] = 0xFF;
    }
    /*Ghi bộ nhớ đệm sang driver, xuất hiện thị LEDs*/
    write(port_led_fd, write_buf, 1);
    /*Cài đặt lại định thời cho tín hiệu SIGALRM*/
    alarm (time_period);
}
/*Chương trình chính, khai báo dưới dạng tham số cho người dùng lựa
chọn nhập tham số thời gian*/
int
main(int argc, char **argv)
{
    /*Trước khi thao tác cần mở tập tin thiết bị driver*/
    if ((port_led_fd = open("/dev/port_led", O_RDWR)) < 0)
    {
        printf("Error whilst opening /dev/port_led
device\n");
        return -1;
    }
    /*Kiểm tra lỗi cú pháp nhập từ người dùng*/
    if (argc != 2) {
        print_usage();
    }
    /*Lấy về thông tin thời gian nhập từ người dùng*/
    time_period = atoi(argv[1]);
    /*Cài đặt bộ định thời tín hiệu SIGALRM gắn với hàm thực thi ngắt
catch_alarm()*/
    signal (SIGALRM, catch_alarm);
    /*Quy định khoảng thời gian cho bộ định thời*/
    alarm (time_period);
    printf ("Go to death loop ...");
    /*Vòng lặp busy, hoặc có thể làm công việc khác trong khi bộ định
thời đang hoạt động*/
    while (1);
}
```

**c. Biên dịch và thực thi chương trình:**

- *Driver*: Chép *driver* `port_led_control.ko` vào kit chuẩn bị cài đặt;
- *Application*: Biên dịch tập tin chương trình bằng lệnh sau:

```
arm-none-linux-gnueabi-gcc 1_3_TimerLedControl_1.c -o  
TimerLedControl_1
```

- Cài đặt *driver* vào hệ thống bằng lệnh sau: `insmod port_led_control.ko`
- Thực thi chương trình: `./TimerLedControl_1 1`

Chúng ta thấy LEDs sáng tắt theo chu kỳ 1s, đồng thời in ra câu thông báo trong màn hình hiển thị như sau:

```
./TimerLedControl_1 1  
Go to death loop ...  
LEDs are off  
LEDs are on  
LEDs are off  
LEDs are on  
LEDs are off  
...
```

(Chương trình cứ tiếp tục in ra theo chu kỳ 1s)

Chúng ta thấy mặc dù chương trình chính đi vào vòng lặp vô tận (Go to death loop ...) nhưng bộ định thời vẫn còn hoạt động và in ra câu thông báo đồng thời điều khiển LEDs sáng tắt theo chu kỳ 1s cho đến khi người dùng kết thúc chương trình bằng tín hiệu ngắt (bằng cách dùng lệnh `ctrl+C`).

**2. Timer trong driver:****a. Chương trình driver:**

*/\*Khai báo thư viện cho các lệnh trong chương trình\*/*

```
#include <linux/module.h>  
#include <linux/errno.h>  
#include <linux/init.h>  
#include <asm/gpio.h>  
#include <asm/atomic.h>  
#include <asm/atomic.h>
```

```
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <linux/time.h> //Thư viện dùng cho timer
#include <linux/jiffies.h> //Thư viện chứa ticks jiffies
/*Tên tập tin thiết bị*/

#define DRVNAME      "timer_led_dev"
#define DEVNAME      "timer_led"

/*-----Port Control-----*/

/*Định nghĩa các chân thao tác tương ứng với chân gpio*/

#define P00          AT91_PIN_PB8
#define P01          AT91_PIN_PB10
#define P02          AT91_PIN_PA23
#define P03          AT91_PIN_PB16
#define P04          AT91_PIN_PA24
#define P05          AT91_PIN_PB11
#define P06          AT91_PIN_PB9
#define P07          AT91_PIN_PB7
#define P0            (P00|P01|P02|P03|P04|P05|P06|P07)

/*Basic commands*/

/*Định nghĩa các lệnh set và clear PIN căn bản*/

#define SET_P00()      gpio_set_value(P00,1)
#define SET_P01()      gpio_set_value(P01,1)
#define SET_P02()      gpio_set_value(P02,1)
#define SET_P03()      gpio_set_value(P03,1)
#define SET_P04()      gpio_set_value(P04,1)
#define SET_P05()      gpio_set_value(P05,1)
#define SET_P06()      gpio_set_value(P06,1)
#define SET_P07()      gpio_set_value(P07,1)

#define CLEAR_P00()    gpio_set_value(P00,0)
#define CLEAR_P01()    gpio_set_value(P01,0)
#define CLEAR_P02()    gpio_set_value(P02,0)
#define CLEAR_P03()    gpio_set_value(P03,0)
#define CLEAR_P04()    gpio_set_value(P04,0)
```



```
#define CLEAR_P05()                gpio_set_value(P05,0)
#define CLEAR_P06()                gpio_set_value(P06,0)
#define CLEAR_P07()                gpio_set_value(P07,0)
/*Biến sửa lỗi trong quá trình mở thiết bị*/
/* Counter is 1, if the device is not opened and zero (or less) if opened. */
static atomic_t timer_led_open_cnt = ATOMIC_INIT(1);
/*Khai báo biến cấu trúc lưu timer khởi tạo tên my_timer*/
struct timer_list my_timer;
/*Thời gian khởi tạo ngắt cho mỗi chu kỳ là 1s*/
int time_period=1;
/*Biến cập nhật trạng thái LEDs*/
int i=0;

/*Hàm chuyển dữ liệu 8 bit thành trạng thái tương ứng của LEDs*/
void timer_led_write_data_port(char data)
{
    (data&(1<<0)) ? SET_P00() : CLEAR_P00();
    (data&(1<<1)) ? SET_P01() : CLEAR_P01();
    (data&(1<<2)) ? SET_P02() : CLEAR_P02();
    (data&(1<<3)) ? SET_P03() : CLEAR_P03();
    (data&(1<<4)) ? SET_P04() : CLEAR_P04();
    (data&(1<<5)) ? SET_P05() : CLEAR_P05();
    (data&(1<<6)) ? SET_P06() : CLEAR_P06();
    (data&(1<<7)) ? SET_P07() : CLEAR_P07();
}

/*Khai báo và định nghĩa hàm phục vụ ngắt cho timer (my_timer)*/
void my_timer_function (unsigned long data) {
    /*Nếu i = 0 thì cho i = 1 cập nhật LEDs tắt và ngược lại để tạo hiệu
    ứng sáng tắt theo chu kỳ thời gian đã nhập*/
    if (i == 0) {
        i = 1;
        timer_led_write_data_port(0x00);
    } else {
        i = 0;
    }
}
```

```
        timer_led_write_data_port(0xFF);
    }

    /*Chu kỳ chớp tắt phải >= 1 */
    if (time_period==0) {
        time_period = 1;
    }

    /*Cài đặt lại chu kỳ ngắt sau mỗi lần thực thi hàm phục vụ ngắt, tạo hiệu ứng sáng tắt liên tục, với chu kỳ là time_period giây*/
    mod_timer (&my_timer, jiffies + time_period*HZ);
}

/*Giao diện hàm write() làm nhiệm vụ cập nhật thời gian của một chu kỳ ngắt do người sử dụng nhập vào thông qua user application*/
static ssize_t timer_led_write (struct file *filp, char
__iomem buf[], size_t bufsz, loff_t *f_pos)
{
    time_period = buf[0];
    printk ("Timer period has just been set to %ds\n",
time_period);
    return bufsz;
}

/*Hàm thực thi khi thiết bị được mở*/
static int
timer_led_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&timer_led_open_cnt)) {
        atomic_inc(&timer_led_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}
```

```
/*Hàm thực thi khi thiết bị được đóng*/
static int
timer_led_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&timer_led_open_cnt);

    return 0;
}

/*Khai báo và cập nhật tập tin lệnh cho thiết bị*/
struct file_operations timer_led_fops = {
    .write      = timer_led_write,
    .open       = timer_led_open,
    .release    = timer_led_close,
};

/*Gán tập tin lệnh và tên thiết bị vào tập tin thiết bị sẽ được tạo*/
static struct miscdevice timer_led_dev = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = "timer_led",
    .fops       = &timer_led_fops,
};

/*Hàm được thực thi khi cài đặt driver vào hệ thống*/
static int __init
timer_led_mod_init(void)
{
    /*Cài đặt các thông số cho các PIN muốn thao tác là ngõ ra*/
    gpio_request (P00, NULL);
    gpio_request (P01, NULL);
    gpio_request (P02, NULL);
    gpio_request (P03, NULL);
    gpio_request (P04, NULL);
    gpio_request (P05, NULL);
    gpio_request (P06, NULL);
    gpio_request (P07, NULL);

    at91_set_GPIO_periph (P00, 1);
}
```

```
at91_set_GPIO_periph (P01, 1);
at91_set_GPIO_periph (P02, 1);
at91_set_GPIO_periph (P03, 1);
at91_set_GPIO_periph (P04, 1);
at91_set_GPIO_periph (P05, 1);
at91_set_GPIO_periph (P06, 1);
at91_set_GPIO_periph (P07, 1);

gpio_direction_output(P00, 0);
gpio_direction_output(P01, 0);
gpio_direction_output(P02, 0);
gpio_direction_output(P03, 0);
gpio_direction_output(P04, 0);
gpio_direction_output(P05, 0);
gpio_direction_output(P06, 0);
gpio_direction_output(P07, 0);

/*Các thao tác khởi tạo timer*/

/*Yêu cầu kernel dành một vùng nhớ lưu timer sắp được tạo*/
init_timer (&my_timer);

/*Cập nhật thời gian sẽ sinh ra ngắt*/
my_timer.expires = jiffies + time_period*HZ;

/*Cập nhật dữ liệu cho hàm phục vụ ngắt*/
my_timer.data = 0;

/*Gán hàm phục vụ ngắt cho timer*/
my_timer.function = my_timer_function;

/*Cuối cùng kích hoạt timer hoạt động trong hệ thống*/
add_timer (&my_timer);

/*Đăng ký tập tin thiết bị đã được khai báo trong những bước đầu tiên*/
return misc_register(&timer_led_dev);
}

/*Hàm được thực thi khi tháo gỡ thiết bị ra khỏi hệ thống*/
static void __exit
timer_led_mod_exit(void)
{
```

*/\*Trước khi tháo gỡ driver ra khỏi hệ thống, chúng ta phải tháo bỏ timer ra, nếu không timer vẫn chạy mặc dù driver không còn tồn tại trong hệ thống nữa. Dẫn đến phát sinh lỗi trong những lần cài đặt tiếp theo\*/*

```
del_timer_sync(&my_timer);  
  
/*Thực hiện lệnh tháo bỏ driver ra khỏi hệ thống*/  
misc_deregister(&timer_led_dev);  
}
```

```
module_init (timer_led_mod_init);  
module_exit (timer_led_mod_exit);
```

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("coolwarmboy");  
MODULE_DESCRIPTION("Character device for for generic gpio  
api");
```

**b. Chương trình application:**

*/\*Khai báo các thư viện cần dùng trong chương trình\*/*

```
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>
```

*/\*Biến lưu số mô tả tập tin khi thiết bị được mở\*/*

```
int port_led_fd;
```

*/\*Bộ đệm ghi có kích thước 1 byte\*/*

```
char write_buf[1];
```

*/\*Biến lưu thời gian nhập vào từ người dùng\*/*

```
long int time_period;
```

*/\*Hàm in thông báo hướng dẫn\*/*

```
void  
print_usage()  
{  
    printf("timer_led_control_app <TimePeriod>\n");  
    exit(0);  
}
```

```
/*Hàm main() được sử dụng có tham số*/
int
main(int argc, char **argv)
{
    /*Mở và kiểm tra lỗi trong quá trình mở*/
    if ((port_led_fd = open("/dev/timer_led", O_RDWR)) < 0)
    {
        printf("Error whilst opening /dev/port_led
device\n");
        return -1;
    }
    /*Kiểm tra lỗi cú pháp nhập từ người dùng*/
    if (argc != 2) {
        print_usage();
    }
    /*Lưu thông tin thời gian nhập vào từ người dùng*/
    time_period = atoi(argv[1]);
    /*Cập nhật bộ đếm ghi*/
    write_buf[0] = time_period;
    /*Ghi dữ liệu trong bộ đếm sang driver*/
    write (port_led_fd, write_buf, 1);
    /*Trả về 0 trong trường hợp không xảy ra lỗi trong quá trình thực thi
lệnh*/
    return 0;
}
```

**c. Biên dịch và thực thi chương trình:**

Biên dịch chương trình *driver* và *application* tương tự như trong các bài trước, ở đây chúng ta không nhắc lại.

Chép *driver* và *applicaiton* vào kit cài đặt và xem kết quả.

Sau khi cài đặt *driver* vào hệ thống, 8 LEDs sẽ sáng tắt với chu kỳ 1s (như đã lập trình). Chúng ta thay đổi tần số chớp tắt bằng cách chạy chương trình ứng dụng trên, theo cú pháp đã được quy định trong chương trình:

<tên chương trình> <thời gian trì hoãn>

<tên chương trình> là tên chương trình ứng dụng sau khi đã biên dịch;

<thời gian trì hoãn> là thời gian trì hoãn giữa hai lần thay đổi trạng thái;

### III. Kết luận và bài tập:

#### a. Kết luận:

Trong bài này chúng ta đã thực hành thành công cách sử dụng ngắt thời gian trong cả *driver* và *application*. Mỗi cách khác nhau đều có những ưu và nhược điểm riêng trong quá trình sử dụng. Tuy nhiên hai cách này có một điểm yếu chung là thời gian phát sinh ngắt tối thiểu là 1s đối với kỹ thuật “alarm” trong *user application*, 10ms đối với *timer* trong *driver*. Những ứng dụng đòi hỏi thời gian ngắt ngắn hơn thì không còn phù hợp với kỹ thuật này nữa. Chúng ta sẽ tìm hiểu kỹ thuật khác hiệu quả hơn trong những module điều khiển khác.

#### b. Bài tập:

1. Viết chương trình sử dụng kỹ thuật “alarm” để lập trình hiệu ứng led sáng dần, tắt dần; sáng dồn; một điểm sáng dịch chuyển mất dần; Với *driver* đã được lập trình trong bài điều khiển 8 LEDs sáng tắt. Chu kỳ ngắt do người sử dụng quy định trong lúc thực thi.
2. Lập trình tương tự với kỹ thuật ngắt dùng *timer* trong *driver*.

**\*\*Các bài tập này điều dựa vào sơ đồ kết nối phần cứng trong hai ví dụ trên.**

**BÀI 2****GIAO TIẾP ĐIỀU KHIỂN  
LED 7 ĐOẠN RỜI****I. Phác thảo dự án:**

LEDs 7 đoạn là một trong những linh kiện điện tử hiển thị thông tin phổ biến. Các thông tin hiển thị thông thường là các số thập phân, một số trường hợp là số nhị phân, thập lục phân, ... Có nhiều cách hiển thị LEDs 7 đoạn khác nhau, điều khiển trực tiếp thông qua các cổng vào ra, và điều khiển bằng phương pháp quét. Đề đơn giản, trong bài này chúng ta sẽ nghiên cứu cách điều khiển trực tiếp thông qua các cổng xuất mã 7 đoạn ra LEDs thông thường. Làm nền tảng cho phương pháp hiển thị bằng phương pháp quét LEDs trong bài sau.

Những kiến thức về LED 7 đoạn đã được nghiên cứu rất kỹ trong những môn học kỹ thuật số căn bản. Nên sẽ không được nhắc lại trong quyển sách này mà chỉ áp dụng vào hướng dẫn thực hành thao tác với hệ thống nhúng.

**a. Yêu cầu dự án:**

Dự án này sẽ hướng dẫn cách điều khiển 2 LEDs 7 đoạn rời hiển thị số nguyên từ 00 đến 99. Người dùng sẽ nhập 3 tham số: giới hạn 1, giới hạn 2 và chu kỳ đếm (tính bằng ms). Hệ thống sẽ tiến hành đếm bắt đầu từ giới hạn 1 đến giới hạn 2 với tốc độ đếm được quy định. Nếu giới hạn 1 lớn hơn giới hạn 2 thì chương trình sẽ thực hiện đếm xuống. Nếu giới hạn 1 nhỏ hơn giới hạn 2 thì chương trình sẽ thực hiện đếm lên. Nếu giới hạn 1 bằng giới hạn 2 thì hiển thị số giới hạn 1 ra LEDs. Cả hai giới hạn đều nằm trong khoảng từ 00 đến 99. Cú pháp lệnh khi thực thi như sau:

<tên chương trình> <giới hạn 1> <giới hạn 2> <thời gian trì hoãn>

Trong đó:

<tên chương trình> là tên chương trình sau khi đã biên dịch thành công;

<giới hạn 1> là giới hạn đầu tiên do người dùng nhập vào từ 00 đến 99;

<giới hạn 2> là giới hạn sau do người dùng nhập vào có giá trị từ 00 đến 99;

**b. Phân công nhiệm vụ:**

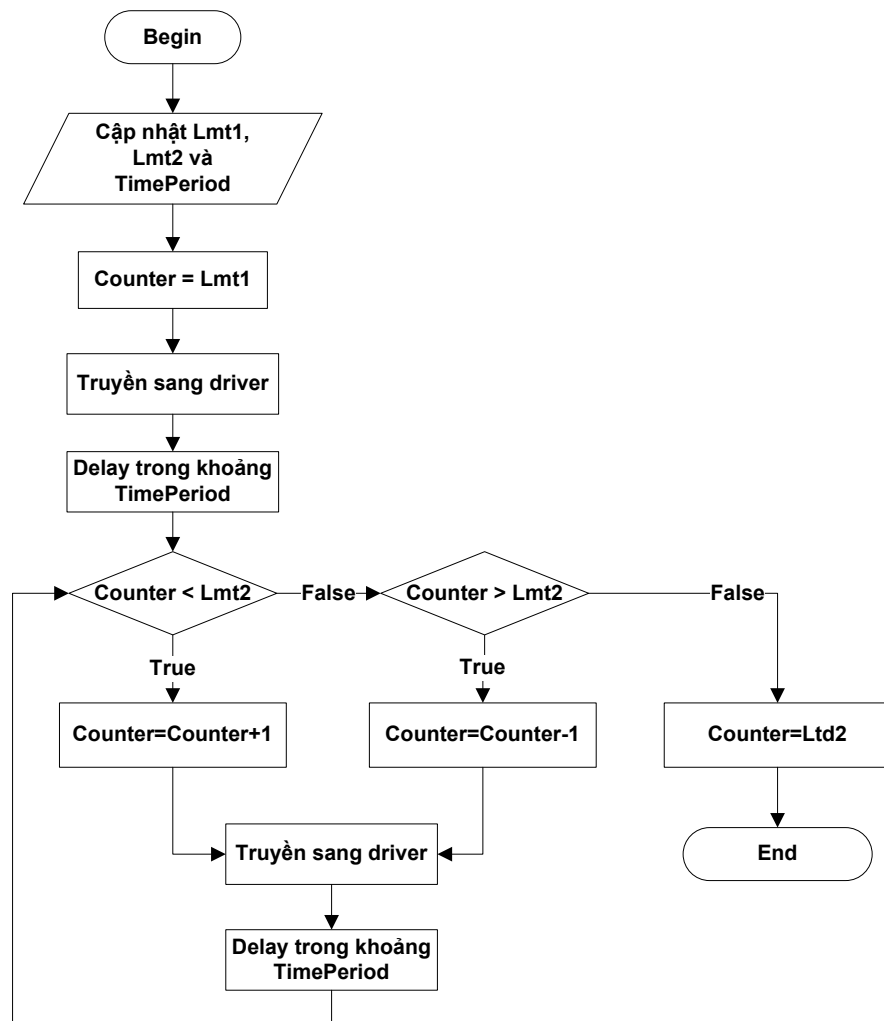
- *Driver:*



Áp dụng giao diện hàm `write()` nhận dữ liệu là số nguyên từ 00 đến 99 từ *user application*, giải mã sang số BCD trước khi chuyển thành mã 7 đoạn để hiển thị ra LEDs.

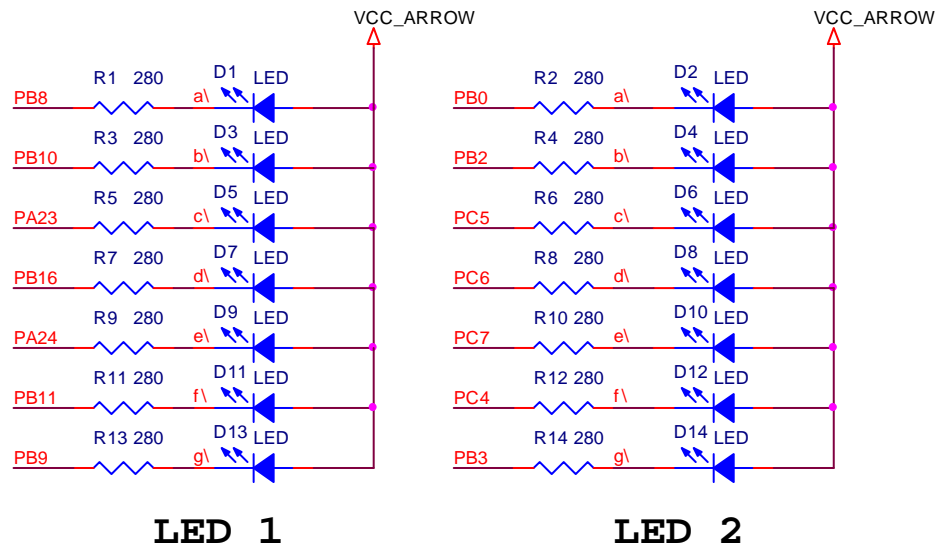
- *Application*:

Nhận các tham số từ người dùng nhập khi gọi chương trình thực thi. Thực hiện đếm lên hay đếm xuống tùy thuộc vào giá trị nhận được từ tham số người dùng. Dùng giao diện hàm `write()` truyền số từ 00 đến 99 cho *driver* hiển thị ra LEDs. Chương trình *application* thực hiện theo lưu đồ sau:



## II. Thực hiện:

1. Kết nối phần cứng theo sơ đồ sau:



## 2. Viết chương trình:

- **Driver:** Tên 2\_Dis\_Seg\_led\_dev.c

```
/*Khai báo thư viện cần dùng cho các hàm trong chương trình*/
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>

/*Tên driver thiết bị*/
#define DRVNAME      "dis_seg_led_dev"
#define DEVNAME      "dis_seg_led"

/*Khai báo các chân LEDs sử dụng tương ứng với chân trong chip*/
/*Khai báo các chân của LEDs thứ nhất (LEDs chục)*/
#define A1            AT91_PIN_PB8
#define B1            AT91_PIN_PB10
#define C1            AT91_PIN_PA23
#define D1            AT91_PIN_PB16
#define E1            AT91_PIN_PA24
#define F1            AT91_PIN_PB11
#define G1            AT91_PIN_PB9

/*Khai báo các chân của LEDs thứ hai (LEDs đơn vị)*/
```

```
#define A2          AT91_PIN_PB0
#define B2          AT91_PIN_PB2
#define C2          AT91_PIN_PC5
#define D2          AT91_PIN_PC6
#define E2          AT91_PIN_PC7
#define F2          AT91_PIN_PC4
#define G2          AT91_PIN_PB3
```

*/\*Các lệnh set và clear bit cho các chân của LEDs 7 đoạn\*/*

*/\*Basic commands\*/*

*/\*Các lệnh cho LEDs 1 \*/*

```
#define SET_A1()      gpio_set_value(A1,1)
#define SET_B1()      gpio_set_value(B1,1)
#define SET_C1()      gpio_set_value(C1,1)
#define SET_D1()      gpio_set_value(D1,1)
#define SET_E1()      gpio_set_value(E1,1)
#define SET_F1()      gpio_set_value(F1,1)
#define SET_G1()      gpio_set_value(G1,1)
```

```
#define CLEAR_A1()    gpio_set_value(A1,0)
#define CLEAR_B1()    gpio_set_value(B1,0)
#define CLEAR_C1()    gpio_set_value(C1,0)
#define CLEAR_D1()    gpio_set_value(D1,0)
#define CLEAR_E1()    gpio_set_value(E1,0)
#define CLEAR_F1()    gpio_set_value(F1,0)
#define CLEAR_G1()    gpio_set_value(G1,0)
```

*/\*Các lệnh cho LEDs 2\*/*

```
#define SET_A2()      gpio_set_value(A2,1)
#define SET_B2()      gpio_set_value(B2,1)
#define SET_C2()      gpio_set_value(C2,1)
#define SET_D2()      gpio_set_value(D2,1)
#define SET_E2()      gpio_set_value(E2,1)
#define SET_F2()      gpio_set_value(F2,1)
#define SET_G2()      gpio_set_value(G2,1)
```

```
#define CLEAR_A2()    gpio_set_value(A2,0)
```

```
#define CLEAR_B2()          gpio_set_value(B2,0)
#define CLEAR_C2()          gpio_set_value(C2,0)
#define CLEAR_D2()          gpio_set_value(D2,0)
#define CLEAR_E2()          gpio_set_value(E2,0)
#define CLEAR_F2()          gpio_set_value(F2,0)
#define CLEAR_G2()          gpio_set_value(G2,0)

/* Counter is 1, if the device is not opened and zero (or
less) if opened. */
static atomic_t dis_seg_led_open_cnt = ATOMIC_INIT(1);
/*Định nghĩa mã LEDs 7 đoạn tích cực mức thấp*/
static int SSC[10] =
{0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
/*Hàm chuyển đổi dữ liệu 8 bit thành các trạng thái LEDs tương ứng trong
LEDs 7 đoạn*/
/*Dành cho LEDs 1*/
void dis_seg_led_write_data_led_1(char data)
{
    (data&(1<<0)) ? SET_A1() : CLEAR_A1();
    (data&(1<<1)) ? SET_B1() : CLEAR_B1();
    (data&(1<<2)) ? SET_C1() : CLEAR_C1();
    (data&(1<<3)) ? SET_D1() : CLEAR_D1();
    (data&(1<<4)) ? SET_E1() : CLEAR_E1();
    (data&(1<<5)) ? SET_F1() : CLEAR_F1();
    (data&(1<<6)) ? SET_G1() : CLEAR_G1();
}
/*Hàm chuyển đổi dành cho LEDs 2*/
void dis_seg_led_write_data_led_2(char data)
{
    (data&(1<<0)) ? SET_A2() : CLEAR_A2();
    (data&(1<<1)) ? SET_B2() : CLEAR_B2();
    (data&(1<<2)) ? SET_C2() : CLEAR_C2();
    (data&(1<<3)) ? SET_D2() : CLEAR_D2();
    (data&(1<<4)) ? SET_E2() : CLEAR_E2();
    (data&(1<<5)) ? SET_F2() : CLEAR_F2();
    (data&(1<<6)) ? SET_G2() : CLEAR_G2();
}
```

```
}

/*Hàm giải mã số nguyên từ 00 đến 99 sang số BCD và viết trạng thái ra
LEDs vật lý*/

void dis_seg_led_decode_and_write_to_led(char data) {
    dis_seg_led_write_data_led_1(SSC[data/10]);
    dis_seg_led_write_data_led_2(SSC[data%10]);
}

/*Giao diện hàm write() nhận số nguyên từ user application hiển thị ra
LEDs 7 đoạn*/

static ssize_t dis_seg_led_write (struct file *filp, char
__iomem buf[], size_t bufsz, loff_t *f_pos)
{
    /*Gọi hàm chuyển đổi dữ liệu ghi ra LEDs*/
    dis_seg_led_decode_and_write_to_led(buf[0]);
    return bufsz;
}

static int
dis_seg_led_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&dis_seg_led_open_cnt)) {
        atomic_inc(&dis_seg_led_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

static int
dis_seg_led_close(struct inode * inode, struct file * file)
{

```

```
smp_mb__before_atomic_inc();
atomic_inc(&dis_seg_led_open_cnt);
return 0;
}

struct file_operations dis_seg_led_fops = {
    .write      = dis_seg_led_write,
    .open       = dis_seg_led_open,
    .release    = dis_seg_led_close,
};

static struct miscdevice dis_seg_led_dev = {
    .minor       = MISC_DYNAMIC_MINOR,
    .name        = "dis_seg_led",
    .fops        = &dis_seg_led_fops,
};

static int __init
dis_seg_led_mod_init(void)
{
    /*Cài đặt các chân gpio sử dụng thành thành ngõ ra*/
    gpio_request (A1, NULL);
    gpio_request (B1, NULL);
    gpio_request (C1, NULL);
    gpio_request (D1, NULL);
    gpio_request (E1, NULL);
    gpio_request (F1, NULL);
    gpio_request (G1, NULL);

    at91_set_GPIO_periph (A1, 1);
    at91_set_GPIO_periph (B1, 1);
    at91_set_GPIO_periph (C1, 1);
    at91_set_GPIO_periph (D1, 1);
    at91_set_GPIO_periph (E1, 1);
    at91_set_GPIO_periph (F1, 1);
    at91_set_GPIO_periph (G1, 1);
}
```

```
        gpio_direction_output(A1, 0);
        gpio_direction_output(B1, 0);
        gpio_direction_output(C1, 0);
        gpio_direction_output(D1, 0);
        gpio_direction_output(E1, 0);
        gpio_direction_output(F1, 0);
        gpio_direction_output(G1, 0);

        gpio_request (A2, NULL);
        gpio_request (B2, NULL);
        gpio_request (C2, NULL);
        gpio_request (D2, NULL);
        gpio_request (E2, NULL);
        gpio_request (F2, NULL);
        gpio_request (G2, NULL);

        at91_set_GPIO_periph (A2, 1);
        at91_set_GPIO_periph (B2, 1);
        at91_set_GPIO_periph (C2, 1);
        at91_set_GPIO_periph (D2, 1);
        at91_set_GPIO_periph (E2, 1);
        at91_set_GPIO_periph (F2, 1);
        at91_set_GPIO_periph (G2, 1);

        gpio_direction_output(A2, 0);
        gpio_direction_output(B2, 0);
        gpio_direction_output(C2, 0);
        gpio_direction_output(D2, 0);
        gpio_direction_output(E2, 0);
        gpio_direction_output(F2, 0);
        gpio_direction_output(G2, 0);
        return misc_register(&dis_seg_led_dev);
    }

    static void __exit
    dis_seg_led_mod_exit(void)
    {
```

```
misc_deregister(&dis_seg_led_dev);  
}  
  
module_init (dis_seg_led_mod_init);  
module_exit (dis_seg_led_mod_exit);  
  
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("coolwarmboy");  
MODULE_DESCRIPTION("Character device for for generic gpio  
api");
```

- **Application:**

```
/*Khai báo thư viện cần thiết cho các lệnh cần dùng trong chương trình*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <linux/ioctl.h>  
  
/*Hàm in ra hướng dẫn khi người dùng nhập sai cú pháp*/  
void  
print_usage()  
{  
    printf("dis_seg_led_app <lmt1> <lmt2> <time_period>\n");  
    exit(0);  
}  
  
/*Chương trình chính, khai báo dạng tham số cho người dùng*/  
int  
main(int argc, char **argv)  
{  
    /*Biến lưu số mô tả tập tin, khi tập tin thiết bị được mở*/  
    int dis_seg_led_fd;  
    /*Bộ đệm ghi*/  
    char write_buf[1];  
    /*Thời gian thay đổi trạng thái do người dùng nhập vào*/  
    long int time_period;
```



```
/*Các biến phục vụ cho quá trình đếm*/
char counter, lmt1, lmt2;
/*Mở tập tin thiết bị trước khi thao tác*/
if ((dis_seg_led_fd = open("/dev/dis_seg_led", O_RDWR))
< 0)
{
    printf("Error whilst opening /dev/dis_seg_led
device\n");
    return -1;
}
/*Kiểm tra lỗi cú pháp từ người dùng*/
if (argc != 4) {
    print_usage();
}
/*Cập nhật các tham số cần thiết từ người dùng*/
lmt1 = atoi(argv[1]);
lmt2 = atoi(argv[2]);
time_period = atoi(argv[3])*1000;
/*Cho counter bằng giới hạn 1 */
counter = lmt1;
/*Cập nhật trạng thái LEDs lần đầu tiên*/
write_buf[0] = counter;
write(dis_seg_led_fd,write_buf,1);
usleep(time_period);
/*Thực hiện đếm theo yêu cầu của thuật toán đã trình bày*/
do {
    if (counter < lmt2) {
        counter++;
    } else {
        counter--;
    }
    write_buf[0] = counter;
    write(dis_seg_led_fd,write_buf,1);
    usleep(time_period);
}
/*Liên tục đếm cho đến khi counter bằng giới hạn 2 */
```

```
    } while (counter != 1mt2);  
    /*Trả về 0 khi không có lỗi trong quá trình thực thi chương trình*/  
    return 0;  
}
```

**3. Biên dịch và thực thi chương trình:**

Biên dịch thực thi và chạy chương trình trên kit, quan sát kết quả.

**III. Kết luận và bài tập:****a. Kết luận:**

Trong phần này chúng ta đã điều khiển thành công module 2 LEDs 7 đoạn rồi để hiển thị số các số nguyên trong khoảng từ 00 đến 99 bằng các chân gpio thông qua giao diện hàm write giao tiếp giữa *application* và *driver*. Trong trường hợp có nhiều LEDs số lượng IO không đủ, thì chúng ta sẽ chuyển sang dùng phương pháp khác tối ưu hơn. Cũng dùng số lượng chân là 16, chúng ta có thể điều khiển 8 LEDs 7 đoạn. Chúng ta sẽ nghiên cứu cách điều khiển LEDs bằng phương pháp quét trong bài sau.

**b. Bài tập:**

- Viết chương trình *driver* để điều khiển 2 LEDs 7 đoạn hiển thị thông tin là số hex trong khoảng từ 00-FF. Sau đó viết chương trình *application* với cùng ý tưởng như trong bài ví dụ trên:
  - Người dùng nhập vào hai tham số: Giới hạn 1 và giới hạn 2, là hai số hex trong khoảng từ 00-FF;
  - Chương trình thực hiện đếm lên hay đếm xuống từ giới hạn 1 đến giới hạn 2;
  - Khoảng thời gian giữa hai lần thay đổi giá trị được quy định bởi người dùng.
- Mở rộng *driver* và *application* trong ví dụ trên để có thể hiển thị số âm. Lúc này giá trị có thể hiển thị trên LEDs có thể nằm trong khoảng từ -9 đến 99;

**BÀI 3****GIAO TIẾP ĐIỀU KHIỂN****LED 7 ĐOẠN BẰNG PHƯƠNG PHÁP QUÉT****I. Phác thảo dự án:**

Bài trước chúng ta đã điều khiển thành công 2 LEDs 7 đoạn dùng phương pháp thông thường là xuất Port điều khiển trực tiếp. Phương pháp này tuy đơn giản nhưng số lượng io điều khiển là rất lớn trong trường hợp có nhiều LEDs cần điều khiển đồng thời. Bài này chúng ta sẽ tìm sử dụng phương pháp quét để điều khiển đồng thời 8 LEDs 7 đoạn hiển thị thông tin.

Phương pháp quét LEDs được chúng ta nghiên cứu rất kỹ trong môn học thực tập vi xử lý 89XX51. Giáo trình này sẽ áp dụng thuật toán này vào lập trình điều khiển trong hệ thống nhúng.

Trong bài này chúng ta sẽ sử dụng kỹ thuật ngắt thời gian mới, dùng timer vật lý được tích hợp sẵn trong vi điều khiển. Do timer được sử dụng trong chương trình liên quan đến những chức năng của hệ điều hành nên chúng ta phải tự lập trình chức năng này thay vì dùng những hàm được hỗ trợ sẵn.

**a. Yêu cầu dự án:**

Dự án này điều khiển 8 LEDs 7 đoạn bằng phương pháp quét. Với phương pháp điều khiển này, chúng ta sẽ lập trình hiển thị nhiều hiệu ứng khác nhau:

- Hiển thị số “07101080” ra 8 LEDs;
- Đếm hiển thị từ XX đến YY với chu kỳ Z do người dùng quy định;
- Đếm giờ phút giây hiển thị 8 LEDs;

**b. Phân công nhiệm vụ:**

➤ *Driver*: Trong dự án này chúng ta sử dụng chung một *driver*. *Driver* này có những đặc điểm sau:

- *Driver* dùng phương pháp ngắt thời gian timer 0, đây là timer vật lý tích hợp trong vi điều khiển. Tạo chu kỳ ngắt 1ms, khi đến thời điểm ngắt, chương trình sẽ cập nhật thay đổi trạng thái quét LEDs hiển thị theo dữ liệu đã được lưu trữ trong bộ nhớ đệm.
- *Driver* sử dụng hai giao diện hàm: `ioctl()` và `write()`. Trong đó:

- Giao diện hàm `write()` nhận mảng số nguyên trong khoảng từ 0 đến 9 bao gồm 8 phần tử tương ứng với 8 LEDs 7 đoạn cần hiển thị. Hàm sẽ cập nhật dữ liệu trong bộ nhớ đệm hiển thị ngay sau khi nhận được mảng thông tin từ người dùng. Như vậy nếu *user application* muốn hiển thị bất kỳ số nào ra LEDs thì chỉ cần gán số đó vào bộ đệm ghi, sau đó gọi giao diện hàm `write()` chuyển thông tin từ bộ đệm ghi trong *user* sang bộ đệm nhận trong *driver* để hiển thị ra LEDs.

*\*\*Bên cạnh các số từ 0 đến 9, driver còn hỗ trợ hiển thị thêm các ký tự đặt biệt, được quy định bởi những mã số khác nhau: Ký tự “-” được quy định bởi mã số 10, ký tự “\_” được quy định bởi mã số 11, ký tự “ ” (khoảng trắng) được quy định bởi mã số 12, và một số ký tự khác nếu muốn chúng ta có thể thêm vào.*

- Giao diện hàm `ioctl()` thực hiện nhiều chức năng khác nhau:
  1. Trì hoãn thời gian với độ phân giải 10ms. Người dùng gọi giao diện `ioctl()` với tham số lệnh `SWEEP_LED_DELAY` và khoảng thời gian trì hoãn tương ứng để thực hiện trì hoãn theo yêu cầu.
  2. Nhận số nguyên (00000000 đến 99999999) từ *user application* giải mã hiển thị ra LEDs 7 đoạn. Người dùng gọi giao diện `ioctl()` với tham số lệnh `SWEEP_LED_NUMBER_DISPLAY` và số nguyên muốn hiển thị. Chương trình *driver* sẽ nhận thông tin này, giải mã và ghi vào bộ nhớ đệm hiển thị trong *driver* để quét ra LEDs.
  3. Nhận 3 số nguyên giờ, phút, giây từ *user application*, giải mã và ghi vào bộ đệm hiển thị ra LEDs dưới dạng “HH-MM-SS”. Chức năng này có tham số lệnh là `SWEEP_LED_TIME_DISPLAY`.

➤ *Application:*

Trong dự án này chúng ta sẽ xây dựng một chương trình *application* tổng hợp thực hiện tất cả các hiệu ứng điều khiển LEDs 7 đoạn trên. *Application* sẽ được xây dựng dựa vào cấu trúc hàm `main()` có nhiều tham số để người dùng nhập tham số thực thi chương trình. Chương trình sẽ nhận tham số này, lựa chọn trường hợp và đáp ứng đúng yêu cầu. Cụ thể từng chức năng như sau:

- Hiển thị dãy số “01234567” ra 8 LEDs 7 đoạn:

Người dùng nhập dòng lệnh theo cú pháp sau:

```
./<Tên chương trình> display_number
```

Trong đó:

<Tên chương trình> là tên chương trình *application* sau khi biên dịch;

displaynumber là tham số yêu cầu xuất dãy số;

Người lập trình chỉ cần khai báo bộ đếm ghi là một mảng bao gồm 8 thành phần, mỗi phần tử tương đương với 1 LED. Cập nhật bộ đếm ghi là các số từ 0 đến 7. Cuối cùng gọi giao diện hàm write() để ghi bộ đếm ghi sang bộ đếm nhận trong driver hiển thị ra LEDs.

- Đếm hiển thị từ XX đến YY với chu kỳ T:

Người dùng nhập dòng lệnh shell theo cú pháp:

```
./<Tên chương trình> count_number XX YY T
```

Trong đó:

XX: Là số giới hạn 1;

YY: Là số giới hạn 2;

T: Là chu kỳ (đơn vị là 10ms) thay đổi trạng thái.

Chương trình sẽ đếm từ XX đến YY với chu kỳ đếm là Tx10ms. Mỗi lần cập nhật trạng thái chương trình sẽ gọi giao diện hàm ioctl() để yêu cầu driver giải mã hiển thị LEDs.

- Đếm giờ phút giây hiển thị 8 LEDs 7 đoạn:

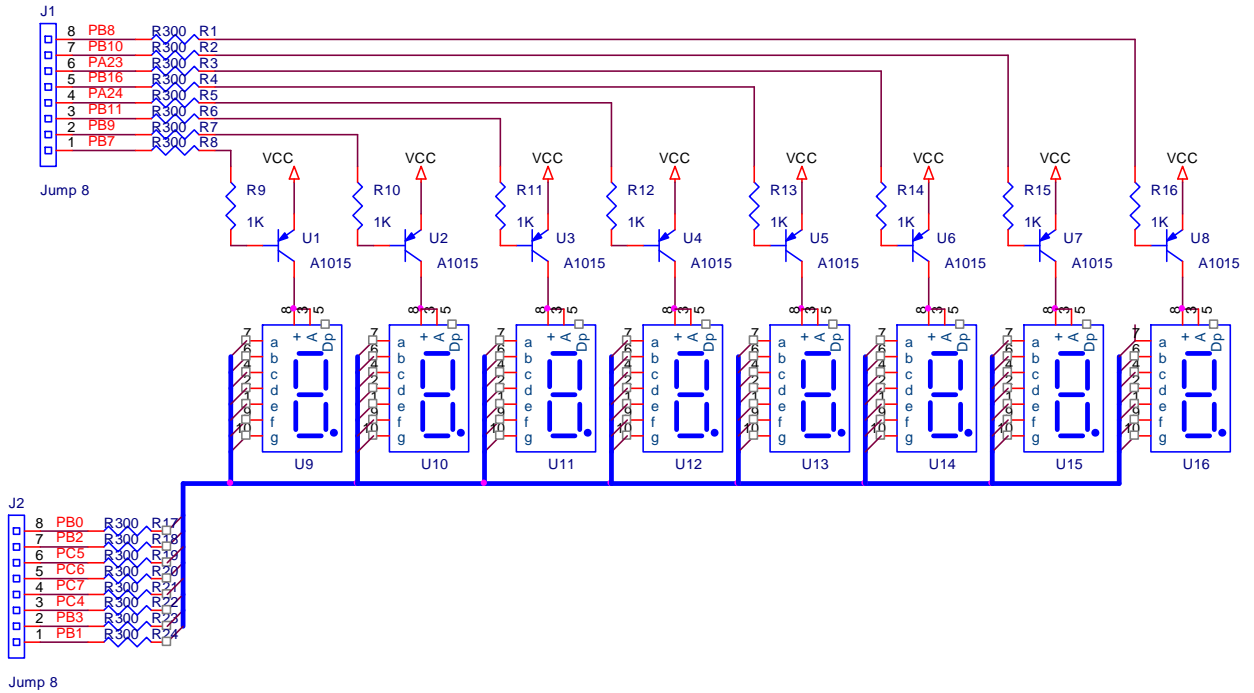
Người dùng nhập dòng lệnh shell theo cú pháp:

```
./<Tên chương trình> count_time HH MM SS
```

Chương trình thực hiện đếm giờ phút giây bắt đầu, sau mỗi chu kỳ 1s thông tin này sẽ được truyền qua driver. Tại đây driver sẽ giải mã và hiển thị ra LEDs 7 đoạn theo dạng “HH-MM-SS”.

## II. Thực hiện:

### 1. Kết nối phần cứng theo sơ đồ sau:



## 2. Chương trình driver:

*/\* Khai báo thư viện cần thiết cho các lệnh cần dùng trong chương trình \*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <mach/at91_tc.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>
```

*/\* Các thư viện hỗ trợ cho ngắt và trì hoãn thời gian \*/*

```
#include <linux/interrupt.h> //Chứa hàm hỗ trợ khai báo ngắt
#include <linux/clock.h> //Chứa hàm khởi tạo clock
#include <linux/irq.h>
#include <linux/time.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/delay.h>
```

*/\* Tên driver thiết bị \*/*

```
#define DRVNAME "sweep_seg_led_dev"
```

```
#define DEVNAME          "sweep_seg_led"

/*-----Port Control-----*/

/*Khai báo các chân điều khiển LEDs 7 đoạn*/
/*Các chân điều khiển tích cực cho LEDs*/

#define P00              AT91_PIN_PB8
#define P01              AT91_PIN_PB10
#define P02              AT91_PIN_PA23
#define P03              AT91_PIN_PB16
#define P04              AT91_PIN_PA24
#define P05              AT91_PIN_PB11
#define P06              AT91_PIN_PB9
#define P07              AT91_PIN_PB7

/*Các chân điều khiển truyền dữ liệu cho LEDs tích cực hiển thị*/

#define A                AT91_PIN_PB0
#define B                AT91_PIN_PB2
#define C                AT91_PIN_PC5
#define D                AT91_PIN_PC6
#define E                AT91_PIN_PC7
#define F                AT91_PIN_PC4
#define G                AT91_PIN_PB3

/*Các lệnh cơ bản điều khiển set() và clear() bit cho các chân gpio*/

/*Basic commands*/

/*Lệnh set() và clear() bit cho các chân chọn LEDs*/

#define SET_P00()        gpio_set_value(P00,1)
#define SET_P01()        gpio_set_value(P01,1)
#define SET_P02()        gpio_set_value(P02,1)
#define SET_P03()        gpio_set_value(P03,1)
#define SET_P04()        gpio_set_value(P04,1)
#define SET_P05()        gpio_set_value(P05,1)
#define SET_P06()        gpio_set_value(P06,1)
#define SET_P07()        gpio_set_value(P07,1)

#define CLEAR_P00()      gpio_set_value(P00,0)
#define CLEAR_P01()      gpio_set_value(P01,0)
#define CLEAR_P02()      gpio_set_value(P02,0)
```



```
#define CLEAR_P03()                gpio_set_value(P03,0)
#define CLEAR_P04()                gpio_set_value(P04,0)
#define CLEAR_P05()                gpio_set_value(P05,0)
#define CLEAR_P06()                gpio_set_value(P06,0)
#define CLEAR_P07()                gpio_set_value(P07,0)
/*Các lệnh set() và clear() bit cho các chân dữ liệu LEDs*/

#define SET_A()                    gpio_set_value(A,1)
#define SET_B()                    gpio_set_value(B,1)
#define SET_C()                    gpio_set_value(C,1)
#define SET_D()                    gpio_set_value(D,1)
#define SET_E()                    gpio_set_value(E,1)
#define SET_F()                    gpio_set_value(F,1)
#define SET_G()                    gpio_set_value(G,1)

#define CLEAR_A()                  gpio_set_value(A,0)
#define CLEAR_B()                  gpio_set_value(B,0)
#define CLEAR_C()                  gpio_set_value(C,0)
#define CLEAR_D()                  gpio_set_value(D,0)
#define CLEAR_E()                  gpio_set_value(E,0)
#define CLEAR_F()                  gpio_set_value(F,0)
#define CLEAR_G()                  gpio_set_value(G,0)

/*Định nghĩa các số định danh lệnh cho giao diện hàm ioctl() */
#define SWEEP_LED_DEV_MAGIC   'B'
#define SWEEP_LED_DELAY _IO(SWEEP_LED_DEV_MAGIC, 0)
#define SWEEP_LED_NUMBER_DISPLAY _IO(SWEEP_LED_DEV_MAGIC, 1)
#define SWEEP_LED_TIME_DISPLAY _IO(SWEEP_LED_DEV_MAGIC, 2)

/* Counter is 1, if the device is not opened and zero (or less) if opened. */
static atomic_t sweep_seg_led_open_cnt = ATOMIC_INIT(1);
/*Bộ nhớ đệm hiển thị trong driver, chứa những số nguyên được giải mã sang LEDs 7 đoạn*/
unsigned char DataDisplay[8]={0,1,2,3,4,5,6,7};
/*Bộ giải mã LEDs 7 đoạn, bao gồm các mã 7 đoạn từ 0 đến 9, các ký tự đặc biệt “-“, “_” và ký tự rỗng*/
unsigned char SevSegCode[]= { 0xC0, 0xF9, 0xA4, 0xB0,
0x99, 0x92, 0x82, 0xF8, 0x80, 0x90, 0x3F, 0x77, 0xFF};
```

```
/*Biến lưu trạng thái LEDs tích cực*/
int i;
/*Con trỏ nền của thanh ghi điều khiển timer0 tích hợp trong vi điều khiển*/
void __iomem *at91tc0_base;
/*Con trỏ cấu trúc clock của timer0*/
struct clk *at91tc0_clk;
/*Hàm trì hoãn thời gian dùng jiffies trong kernel, thay thế cho các hàm trì hoãn thời gian trong user; Hàm trì hoãn thời gian có độ phân giải 10ms*/
void sweep_led_delay(unsigned int delay) {
    /*Khai báo biến lưu thời điểm tương lai cần trì hoãn*/
    long int time_delay;
    /*Cập nhật thời điểm tương lai cần trì hoãn*/
    time_delay = jiffies + delay;
    /*So sánh thời điểm hiện tại với thời điểm tương lai*/
    while (time_before(jiffies, time_delay)) {
        /*Thực hiện chia tiến trình trong quá trình trì hoãn thời gian*/
        schedule();
    }
}
/*Hàm ghi dữ liệu 8 bits cho LEDs tích cực hiển thị*/
void sweep_seg_led_write_data_active_led(char data)
{
    (data&(1<<0)) ? SET_P00() : CLEAR_P00();
    (data&(1<<1)) ? SET_P01() : CLEAR_P01();
    (data&(1<<2)) ? SET_P02() : CLEAR_P02();
    (data&(1<<3)) ? SET_P03() : CLEAR_P03();
    (data&(1<<4)) ? SET_P04() : CLEAR_P04();
    (data&(1<<5)) ? SET_P05() : CLEAR_P05();
    (data&(1<<6)) ? SET_P06() : CLEAR_P06();
    (data&(1<<7)) ? SET_P07() : CLEAR_P07();
}
/*Hàm ghi dữ liệu mã 7 đoạn cho LEDS hiển thị*/
```

```
void sweep_seg_led_write_data_led(char data)
{
    (data&(1<<0)) ? SET_A() : CLEAR_A();
    (data&(1<<1)) ? SET_B() : CLEAR_B();
    (data&(1<<2)) ? SET_C() : CLEAR_C();
    (data&(1<<3)) ? SET_D() : CLEAR_D();
    (data&(1<<4)) ? SET_E() : CLEAR_E();
    (data&(1<<5)) ? SET_F() : CLEAR_F();
    (data&(1<<6)) ? SET_G() : CLEAR_G();
}

/*Hàm chọn LEDs tích cực*/
void active_led_choice(char number) {
    sweep_seg_led_write_data_active_led(~(1<<(number)));
}

/*Hàm xuất dữ liệu cho LEDs hiển thị*/
void data_led_stransmitt (char data) {
    sweep_seg_led_write_data_led (SevSegCode[data]);
}

/*Giải mã số 8 chữ số sang từng ký tự số BCD, lưu vào vùng nhớ đệm
hiển thị LEDs*/
void sweep_led_number_display (unsigned long int data) {
    DataDisplay[0] = data % 10;
    DataDisplay[1] = (data % 100)/10;
    DataDisplay[2] = (data % 1000)/100;
    DataDisplay[3] = (data % 10000)/1000;
    DataDisplay[4] = (data % 100000)/10000;
    DataDisplay[5] = (data % 1000000)/100000;
    DataDisplay[6] = (data % 10000000)/1000000;
    DataDisplay[7] = data/10000000;
}

/*Giải mã giờ phút giây thành những ký số BCD, và ký tự đặc biệt lưu
vào vùng nhớ đệm hiển thị LEDs*/
void sweep_led_time_display(int hh, int mm, int ss) {
    DataDisplay[0] = ss%10;
    DataDisplay[1] = ss/10;
    DataDisplay[2] = 10; //Ký tự "--"
```

```
DataDisplay[3] = mm%10;
DataDisplay[4] = mm/10;
DataDisplay[5] = 10; //Ký tự "-"
DataDisplay[6] = hh%10;
DataDisplay[7] = hh/10;
}

/*Hàm phục vụ ngắt timer thực hiện quét LEDs 7 đoạn, tại một thời điểm
chỉ hiển thị 1 LEDs với mã 7 đoạn của LED đó*/
static irqreturn_t at91tc0_isr(int irq, void *dev_id) {
    int status;

    /*Đọc thanh ghi trạng thái của timer0 reset lại timer sau khi xảy ra
ngắt*/
    status = ioread32(at91tc0_base + AT91_TC_SR);
    /*Truyền dữ liệu cho LEDs i*/
    data_led_stransmitt(DataDisplay[i]);
    /*Chọn tích cực cho LEDs i*/
    active_led_choice(i);
    /*Cập nhật biến trạng thái cho LEDs tiếp theo*/
    i++;
    /*Giới hạn số LEDs hiển thị*/
    if (i==8) i = 0;
    /*Kết thúc quá trình ngắt trở lại chương trình driver chính*/
    return IRQ_HANDLED;
}

/*Khai báo và định nghĩa giao diện hàm write(), nhận dữ liệu từ user
application là một mảng 8 bytes cập nhật vào bộ nhớ đệm ghi trong
driver*/
static ssize_t dis_seg_led_write (struct file *filp,
unsigned char __iomem buf[], size_t bufsize, loff_t
*f_pos)
{
    /*Bộ nhớ đệm ghi nhận dữ liệu từ user*/
    unsigned char write_buf[8];
    /*Biến lưu kích thước trả về khi ghi thành công*/
```

```
int write_size = 0;
int i;
/*Thực hiện hàm copy_from_user() nhận dữ liệu từ user đến driver */
if (copy_from_user (write_buf, buf, 8) != 0) {
    return -EFAULT;
} else {
    write_size = bufsize;
}
/*Chuyển dữ liệu từ bộ đệm ghi sang bộ đệm hiển thị LEDs */
for (i=0; i<8; i++) {
    DataDisplay[i] = write_buf [i];
}
return write_size;
}

/*Khai báo và định nghĩa giao diện hàm ioctl()*/
static int
sweep_seg_led_ioctl(struct inode * inode, struct file *
file, unsigned int cmd,unsigned long int arg[])
{
    int retval=0;
    switch (cmd) {
        /*Trong trường hợp lệnh delay trong kernel, gọi hàm
        sweep_led_delay*/
        case SWEEP_LED_DELAY:
            sweep_led_delay(arg[0]);
            break;
        /*Trong trường hợp lệnh xuất số hiển thị, gọi hàm
        sweep_led_number_display() để giải mã và cập nhật thông tin
        cho bộ đệm hiển thị LEDs*/
        case SWEEP_LED_NUMBER_DISPLAY:
            sweep_led_number_display(arg[0]);
            break;
    }
}
```

```
/*Trong trường hợp hiển thị thời gian, gọi hàm
sweep_led_time_display() để giải mã và xuất ra bộ nhớ đệm
hiển thị dạng HH-MM-SS*/
case SWEEP_LED_TIME_DISPLAY:
    sweep_led_time_display(arg[0],      arg[1],
        arg[2]);
break;
/*Trong trường hợp không có lệnh hỗ trợ thì in ra lỗi cho
người lập trình*/
default:
    printk("The function you type does not
    exist\n");
    retval=-1;
break;
}
}

/*Khai báo và định nghĩa giao diện hàm open() */
static int
sweep_seg_led_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&sweep_seg_led_open_cnt)) {
        atomic_inc(&sweep_seg_led_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
        already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

/*Khai báo và định nghĩa giao diện hàm close*/
static int
```

```
sweep_seg_led_close(struct inode * inode, struct file *
file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&sweep_seg_led_open_cnt);

    return 0;
}

/*Định nghĩa và cập nhật cấu trúc file operations */
struct file_operations sweep_seg_led_fops = {
    .write      = dis_seg_led_write,
    .ioctl      = sweep_seg_led_ioctl,
    .open       = sweep_seg_led_open,
    .release    = sweep_seg_led_close,
};

/*Định nghĩa và cập nhật cấu trúc i_node */
static struct miscdevice sweep_seg_led_dev = {
    .minor       = MISC_DYNAMIC_MINOR,
    .name        = "sweep_seg_led",
    .fops        = &sweep_seg_led_fops,
};

/*Hàm thực thi khi driver được cài đặt vào hệ thống*/
static int __init
sweep_seg_led_mod_init(void)
{
    int ret=0;

    /*Khai báo các chân trong gpio đã định nghĩa thành chế độ ngõ ra */
    gpio_request (P00, NULL);
    gpio_request (P01, NULL);
    gpio_request (P02, NULL);
    gpio_request (P03, NULL);
    gpio_request (P04, NULL);
    gpio_request (P05, NULL);
    gpio_request (P06, NULL);
    gpio_request (P07, NULL);
}
```

```
at91_set_GPIO_periph (P00, 1);
at91_set_GPIO_periph (P01, 1);
at91_set_GPIO_periph (P02, 1);
at91_set_GPIO_periph (P03, 1);
at91_set_GPIO_periph (P04, 1);
at91_set_GPIO_periph (P05, 1);
at91_set_GPIO_periph (P06, 1);
at91_set_GPIO_periph (P07, 1);
```

```
gpio_direction_output(P00, 0);
gpio_direction_output(P01, 0);
gpio_direction_output(P02, 0);
gpio_direction_output(P03, 0);
gpio_direction_output(P04, 0);
gpio_direction_output(P05, 0);
gpio_direction_output(P06, 0);
gpio_direction_output(P07, 0);
```

```
gpio_request (A, NULL);
gpio_request (B, NULL);
gpio_request (C, NULL);
gpio_request (D, NULL);
gpio_request (E, NULL);
gpio_request (F, NULL);
gpio_request (G, NULL);
```

```
at91_set_GPIO_periph (A, 1);
at91_set_GPIO_periph (B, 1);
at91_set_GPIO_periph (C, 1);
at91_set_GPIO_periph (D, 1);
at91_set_GPIO_periph (E, 1);
at91_set_GPIO_periph (F, 1);
at91_set_GPIO_periph (G, 1);
```

```
gpio_direction_output(A, 0);
gpio_direction_output(B, 0);
```



```
gpio_direction_output(C, 0);
gpio_direction_output(D, 0);
gpio_direction_output(E, 0);
gpio_direction_output(F, 0);
gpio_direction_output(G, 0);

/*Khai báo timer0 cho hệ thống*/
at91tc0_clk = clk_get(NULL, "tc0_clk");
/*Cho phép clock hoạt động*/
clk_enable(at91tc0_clk);
/*Di chuyển con trỏ timer0 đến địa chỉ thanh ghi nền cho timer
counter 0*/
at91tc0_base = ioremap_nocache(AT91SAM9260_BASE_TC0,
64);
/*Kiểm tra lỗi trong quá trình định vị*/
if (at91tc0_base == NULL)
{
    printk(KERN_INFO "at91adc: TC0 memory mapping
failed\n");
    ret = -EACCES;
    goto exit_5;
}
/*Cập nhật thông số cho timer, khởi tạo định thời ngắt timer0 với chu
kỳ ngắt là 1ms*/
// Configure TC0 in waveform mode, TIMER_CLK1 and to
generate interrupt on RC compare.
// Load 50000 to RC so that with TIMER_CLK1 = MCK/2 =
50MHz, the interrupt will be
// generated every 1/50MHz * 50000 = 20nS * 50000 = 1
milli second.
// NOTE: Even though AT91_TC_RC is a 32-bit register,
only 16-bits are programmable.

iowrite32(50000, (at91tc0_base + AT91_TC_RC));
iowrite32((AT91_TC_WAVE | AT91_TC_WAVESEL_UP_AUTO),
(at91tc0_base + AT91_TC_CMR));
```

```
iowrite32(AT91_TC_CPCS, (at91tc0_base + AT91_TC_IER));
iowrite32((AT91_TC_SWTRG | AT91_TC_CLKEN),
(at91tc0_base + AT91_TC_CCR));

//Khởi tạo ngắt cho timer0

ret = request_irq(AT91SAM9260_ID_TC0, //ID ngắt timer0
at91tc0_isr, //Trở đến hàm phục vụ ngắt
0, // Định nghĩa cờ ngắt có thể chia sẻ
"sweep_seg_led_irq",/*Tên ngắt hiển thị trong /proc/interrupts*/
NULL); //Dữ liệu riêng trong quá trình chia sẻ ngắt
/*In ra thông báo lỗi nếu khởi tạo ngắt không thành công*/
if (ret != 0)
{
printf(KERN_INFO "sweep_seg_led_irq: Timer interrupt
request failed\n");
ret = -EBUSY;
goto exit_6;
}
/*Đăng ký thiết bị vào hệ thống*/
ret = misc_register(&sweep_seg_led_dev);
/*In thông báo cho người dùng khi thiết bị cài đặt thành công*/
printf(KERN_INFO "sweep_seg_led: Loaded module\n");
return ret;
/*Giải phóng bộ nhớ khi có lỗi xảy ra*/
exit_6:
iounmap(at91tc0_base);
exit_5:
clk_disable(at91tc0_clk);
return ret;
}
/*Hàm được thực thi khi tháo gỡ driver ra khỏi hệ thống */
static void __exit
sweep_seg_led_mod_exit(void)
{
/*Giải phóng vùng nhớ dành cho timer0*/
```

```
    iounmap(at91tc0_base);  
    /*Giải phóng clock dành cho timer0*/  
    clk_disable(at91tc0_clk);  
    /*Giải phóng ngắt dành cho timer0*/  
    free_irq( AT91SAM9260_ID_TC0, // Interrupt number  
            NULL); // Private data for shared interrupts  
    /*Tháo driver ra hệ thống*/  
    misc_deregister(&sweep_seg_led_dev);  
    /*In thông báo cho người dùng driver đã được tháo gỡ */  
    printk(KERN_INFO "sweep_seg_led: Unloaded module\n");  
}  
  
/*Cài đặt các hàm vào các macro init và exit*/  
module_init (sweep_seg_led_mod_init);  
module_exit (sweep_seg_led_mod_exit);  
  
/*Các thông tin khái quát cho driver */  
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("coolwarmboy");  
MODULE_DESCRIPTION("Character device for for generic gpio  
api");
```

### **3. Chương trình application:**

```
/*Khai báo thư viện cần thiết cho các lệnh dùng trong chương trình*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <linux/ioctl.h>  
  
/*Khai báo số định danh lệnh dùng cho hàm ioctl*/  
#define SWEEP_LED_DEV_MAGIC  'B'  
#define SWEEP_LED_DELAY _IO(SWEEP_LED_DEV_MAGIC, 0)  
#define SWEEP_LED_NUMBER_DISPLAY _IO(SWEEP_LED_DEV_MAGIC, 1)  
#define SWEEP_LED_TIME_DISPLAY _IO(SWEEP_LED_DEV_MAGIC, 2)  
  
/*Biến lưu số mô tả tập tin thiết bị khi được mở*/  
int sweep_seg_led_fd;  
  
/*Bộ nhớ đệm dữ liệu cho hàm ioctl() chép qua user*/
```

```
unsigned long int ioctl_buf[3]={0,0,0};
/*Bộ nhớ đệm cho hàm write chứa dữ liệu cần hiển thị ra LEDs*/
unsigned char write_buf[8]={0,8,0,1,0,1,7,0};
/*Các biến phục vụ cho chức năng đếm của chương trình*/
unsigned long XX, YY, T, counter;
/*Các biến phục vụ cho chức năng đếm thời gian của chương trình*/
unsigned char HH, MM, SS;
/*Hàm in hướng dẫn cho người dùng trong trường hợp phát sinh lỗi cú pháp*/
int
print_usage (void) {
    printf ("display_number|count_number|count_time    (XX|HH
    YY|MM T|SS)\n");
    return -1;
}
/*Hàm delay trong user, gọi hàm delay trong driver*/
void user_delay(unsigned long int delay) {
    ioctl_buf[0] = delay;
    ioctl (sweep_seg_led_fd, SWEEP_LED_DELAY, ioctl_buf);
}
/*Hàm truyền các thông số giờ phút giây sang bộ nhớ đệm*/
void user_transmit_time(unsigned long int HH, unsigned
long int MM, unsigned long int SS) {
    ioctl_buf[0] = HH;
    ioctl_buf[1] = MM;
    ioctl_buf[2] = SS;
    ioctl(sweep_seg_led_fd,SWEEP_LED_TIME_DISPLAY,ioctl_buf
);
}
/*Hàm main() được khai báo dạng tham số nhập từ người dùng để lựa
chọn chức năng thực thi, và các thông số cần thiết cho từng chức năng */
int
main(int argc, char **argv)
{
    int res;
```

```
/*Trước khi thao tác thì phải mở tập tin thiết bị, đồng thời kiểm
tra lỗi trong quá trình mở*/
if ((sweep_seg_led_fd = open("/dev/sweep_seg_led",
O_RDWR)) < 0)
{
printf("Error whilst opening /dev/sweep_seg_led
device\n");
return -1;
}
/*Phân biệt các trường hợp lệnh khác nhau*/
switch (argc) {
case 2:
/*Thực hiện chức năng hiển thị mã số sinh viên*/
if (!strcmp(argv[1], "display_number")) {
/*Gọi giao diện hàm write() ghi vùng nhớ đệm từ user sang
driver*/
write (sweep_seg_led_fd, write_buf, 8);
} else {
return print_usage();
}
break;
case 5:
/*Trong trường hợp đếm số hiển thị LEDs*/
if (!strcmp(argv[1], "count_number")) {
/*Cập nhật các tham số từ người dùng*/
T = atoi(argv[4]);
XX = atoi(argv[2]);
YY = atoi(argv[3]);
counter = XX;
/*Thực hiện đếm theo quy định của người dùng*/
while (counter != YY) {
ioctl_buf[0] = counter;
ioctl(sweep_seg_led_fd,
SWEEP_LED_NUMBER_DISPLAY, ioctl_buf);
user_delay(T);
}
```

```
        if (counter < YY) {
            counter++;
        } else {
            counter--;
        }
    }

    ioctl_buf[0] = counter;
    ioctl(sweep_seg_led_fd,
        SWEEP_LED_NUMBER_DISPLAY, ioctl_buf);
    /*Chức năng đếm thời gian hiển thị LEDs*/
} else if (!strcmp(argv[1], "count_time")) {
    /*Cập nhật giờ phút giây từ người dùng*/
    HH = atoi(argv[2]);
    MM = atoi(argv[3]);
    SS = atoi(argv[4]);
    /*Chuyển sang vùng nhớ đệm của ioctl()*/
    user_transmit_time(HH,MM,SS);
    user_delay(100);
    /*Thực hiện đếm thời gian giờ phút giây*/
    while (1) {
        if (SS++ == 59) {
            SS = 0;
            if (MM++ == 59) {
                MM = 0;
                if (HH++ == 23) HH = 0;
            }
        }
        user_transmit_time(HH,MM,SS);
        user_delay(100);
    }
} else {
    return print_usage();
}

break;

default:
    return print_usage();
```

```
        break;
    }
    return 0;
}
```

#### **4. Biên dịch và thực thi chương trình:**

Biên dịch *driver* bằng tập tin Makefile có nội dung sau:

```
export ARCH=arm
export CROSS_COMPILE=arm-none-linux-gnueabi-
obj-m += 3_Sweep_Seg_led_dev.o
all:
    make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD)
modules
clean:
    make -C /home/arm/project/kernel/linux-2.6.30 M=$(PWD)
clean
```

Biên dịch chương trình ứng dụng bằng dòng lệnh sau:

```
arm-none-linux-gnueabi-gcc      3_Sweep_Seg_led_app.c      -o
Sweep_Seg_led_app
```

Chép tập tin *driver* và *application* sau khi biên dịch vào kit;

Cài đặt *driver* vào hệ thống bằng lệnh:

```
insmod 3_Sweep_Seg_led_dev.ko
```

Chạy chương trình ứng dụng:

- Chức năng hiển thị số:

```
./Sweep_Seg_led_app display_number
```

- Chức năng đếm số hiển thị:

```
./Sweep_Seg_led_app count_number 1 10 100
```

- Chức năng đếm thời gian:

```
./Sweep_Seg_led_app count_time 12 12 12
```

Người học quan sát và kiểm tra kết quả thực thi dự án.

#### **5. Kết luận và bài tập:**

##### **a. Kết luận:**

Trong bài này chúng ta đã điều khiển thành công 8 LEDs 7 đoạn dùng phương pháp quét. Kiến thức quan trọng nhất trong bài, bên cạnh phương pháp quét led, là cách khởi tạo ngắt từ timer vật lý tích hợp trong vi điều khiển. Người học cần phải

hiểu rõ các bước khởi tạo ngắt, cách cài đặt các thông số cho timer để đạt được các khoảng thời gian ngắt cần thiết

***b. Bài tập:***

1. Cải tiến *driver* điều khiển quét LEDs 7 đoạn 3\_Sweep\_Seg\_led\_dev.c trên sao cho có thể xóa được số 0 vô nghĩa khi hiển thị số.
2. Cải tiến *driver* điều khiển quét LEDs 7 đoạn 3\_Sweep\_Seg\_led\_dev.c trên sao cho có thể hiển thị được số âm trên 8 LEDs 7 đoạn.
3. Viết chương trình ứng dụng *user application* thực hiện các phép toán “+” “-” “x” và “:” kết quả hiển thị trên LEDs 7 đoạn. Các toán hạng được nhập từ người dùng.



**BÀI 4****GIAO TIẾP ĐIỀU KHIỂN  
LCD 16x2****I. Phác thảo dự án:**

Bằng phương pháp truy xuất các chân gpio theo một quy luật nào đó chúng ta có thể điều khiển nhiều thiết bị khác nhau. Trong những bài trước, chúng ta đã điều khiển LEDs đơn, LEDs 7 đoạn, trong bài này chúng ta sẽ thực hành điều khiển một thiết bị hiển thị khác là LCD thuộc loại 16x2. Lý thuyết về nguyên lý hoạt động của LCD đã được nghiên cứu trong các tài liệu chuyên ngành khác, hoặc các bạn có thể tham khảo trong datasheet của thiết bị. Ở đây chúng ta không nhắc lại mà chỉ tập vào viết driver và chương trình điều khiển trong hệ thống nhúng. Các lệnh thao tác sẽ được giải thích kỹ trong quá trình lập trình.

**a. Yêu cầu dự án:**

*Dự án điều khiển LCD bao gồm các chức năng sau:*

- Hiển thị thông tin nhập vào từ người dùng: Người dùng nhập vào một chuỗi ký tự trong màn hình console (*terminal display*), chuỗi ký tự này được xuất hiện trong LCD.
- Các thông số về ngày tháng năm của hệ thống được cập nhật và hiển thị trong LCD.
- Đếm hiển thị trên LCD: Người dùng sẽ nhập các thông số về chu kỳ, giới hạn 1, giới hạn 2. Thực hiện đếm tương tự như thuật toán của bài quét LEDs 7 đoạn. Các thông số này đều được hiển thị trên LCDs.

Sau đây chúng ta sẽ tiến hành phân công nhiệm vụ thực hiện của *driver* và *application*.

**b. Phân công nhiệm vụ:****• Driver:**

Sử dụng giao diện `ioctl()` để thực hiện các thao tác điều khiển LCD cơ bản như:

- Nhận mã lệnh điều khiển từ *user application* sau đó xuất ra các chân gpio ghi vào thanh ghi lệnh của LCDs;

- Nhận dữ liệu ký tự từ *user application*, xuất ra các chân `gpio` ghi vào thanh ghi dữ liệu của LCDs;
- Và một số lệnh khác như: Trì hoãn thời gian dùng `jiffies`, di chuyển con trỏ về đầu dòng hiển thị, bật tắt hiển thị, ...

- *Application*:

Chương trình trong *application* sử dụng những chức năng của *driver* hỗ trợ, lập trình thành những hàm có khả năng lớn hơn để thực hiện những yêu cầu của dự án.

Chương trình dùng phương pháp lập trình có tham số lựa chọn từ người dùng để thực hiện nhiệm vụ theo yêu cầu. Những nhiệm vụ đó là:

- Hiển thị thông tin từ người dùng:

Người dùng chạy chương trình theo cú pháp sau:

```
<tên chương trình> display_string <Chuỗi_ký_tự_cần_hiển_thị>
```

Trong đó:

<Tên chương trình> là tên chương trình ứng dụng sau khi biên dịch;

< Chuỗi\_ký\_tự\_cần\_hiển\_thị> là chuỗi thông tin người dùng muốn hiển thị ra LCD;

`display_string` là tham số chức năng phải nhập để thực hiện nhiệm vụ;

Sau khi nhận được yêu cầu, chương trình sẽ tách từng ký tự trong chuỗi vừa nhập lần lượt hiển thị trên LCDs.

- Hiển thị các thông số ngày tháng năm của hệ thống ra LCD:

Người dùng nhập cú pháp thực thi chương trình như sau:

```
<tên chương trình> display_time
```

Trong đó:

`display_time` là tham số chức năng phải nhập để thực hiện nhiệm vụ;

Ngày tháng năm được hiển thị trên LCD có dạng:

```
The present time is:
```

```
DD/MM/YYYY
```

- Đếm hiển thị trên LCD:

Người dùng nhập lệnh thực thi chương trình theo cú pháp:

```
<tên chương trình> display_counter XX YY T
```

Trong đó:

`display_counter` là tham số chức năng cần phải nhập để thực hiện nhiệm vụ;

`xx` là giới hạn 1;

`yy` là giới hạn 2;

*(\*\*Giới hạn 1 và giới hạn 2 nằm trong khoảng từ 00 đến 99);*

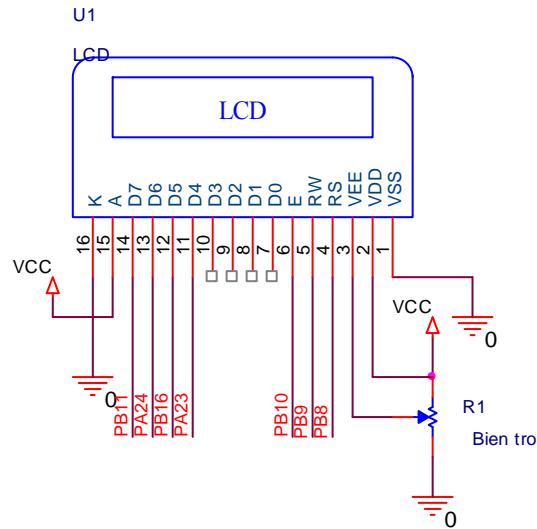
`T` là chu kỳ thay đổi trạng thái đếm, đơn vị là ms;

Chương trình sẽ thực hiện đếm từ `xx` đến `yy` với chu kỳ thay đổi là `T`(ms).

Hiển thị trên LCD theo dạng:

Hàng 1: `xx yy`

Hàng 2: `<counter display>`

**II. Thực hiện:****1. Kết nối phần cứng theo sơ đồ sau:****2. Chương trình driver:**

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <asm/gpio.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/atomic.h>
#include <linux/jiffies.h>
#include <linux/sched.h>

#define DRVNAME      "LCDDriver"
#define DEVNAME      "LCDDevice"

/*-----LCD COMMAND DEFINED-----*/
#define IOC_LCDDEVICE_MAGIC      'B'
#define LCD_CONTROL_WRITE        _IO(IOC_LCDDEVICE_MAGIC, 15)
#define LCD_DATA_WRITE           _IO(IOC_LCDDEVICE_MAGIC, 16)
#define LCD_INIT                 _IO(IOC_LCDDEVICE_MAGIC, 17)
#define LCD_HOME                 _IO(IOC_LCDDEVICE_MAGIC, 18)
#define LCD_CLEAR                _IO(IOC_LCDDEVICE_MAGIC, 19)
```

```
#define LCD_DISP_ON_C          _IO(IOC_LCDDEVICE_MAGIC, 21)
#define LCD_DISP_OFF_C         _IO(IOC_LCDDEVICE_MAGIC, 22)
#define LCD_CUR_MOV_LEFT_C     _IO(IOC_LCDDEVICE_MAGIC, 23)
#define LCD_CUR_MOV_RIGHT_C    _IO(IOC_LCDDEVICE_MAGIC, 24)
#define LCD_DIS_MOV_LEFT_C     _IO(IOC_LCDDEVICE_MAGIC, 25)
#define LCD_DIS_MOV_RIGHT_C    _IO(IOC_LCDDEVICE_MAGIC, 29)
#define LCD_DELAY_MSEC         _IO(IOC_LCDDEVICE_MAGIC, 30)
#define LCD_DELAY_USEC         _IO(IOC_LCDDEVICE_MAGIC, 31)

/*-----LCD PIN CONTROL + DATA-----*/
#define LCD_RW_PIN AT91_PIN_PB9

#define LCD_EN_PIN AT91_PIN_PB10
#define LCD_RS_PIN AT91_PIN_PB8
#define LCD_D4_PIN AT91_PIN_PA23
#define LCD_D5_PIN AT91_PIN_PB16
#define LCD_D6_PIN AT91_PIN_PA24
#define LCD_D7_PIN AT91_PIN_PB11

/*-----LCD command datas-----*/

/* LCD memory map */
/*Vị trí con trỏ bắt đầu dòng đầu tiên*/
#define LCD_LINE0_ADDR 0x00
/*Vị trí con trỏ bắt đầu dòng thứ hai*/
#define LCD_LINE1_ADDR 0x40
/*Vị trí con trỏ bắt đầu dòng thứ ba*/
#define LCD_LINE2_ADDR 0x14
/*Vị trí con trỏ bắt đầu dòng thứ tư*/
#define LCD_LINE3_ADDR 0x54

/* Mã lệnh cơ bản của LCD */
/*Con trỏ địa chỉ RAM hiển thị*/
#define LCD_DD_RAM_PTR 0x80
/*Con trỏ địa chỉ RAM tạo ký tự*/
#define LCD_CG_RAM_PTR 0x40
/*Lệnh xóa dữ liệu hiển thị, con trỏ hiển thị về 0*/
```

```
#define LCD_CLEAR_DISPLAY 0x01
/*Lệnh đưa con trỏ về điểm bắt đầu dòng đầu tiên, dữ liệu bị dịch sẽ trở
về vị trí cũ*/
#define LCD_RETURN_HOME 0x02
/*Lệnh cài đặt chế độ giao tiếp 4 bits và 2 dòng hiển thị*/
#define LCD_DISP_INIT 0x28
/*Con trỏ hiển thị tăng sau khi ghi dữ liệu vào RAM hiển thị*/
#define LCD_INC_MODE 0x06
/*Bật hiển thị và con trỏ nhập nháy*/
#define LCD_DISP_ON 0x0C
/*Tắt hiển thị và con trỏ*/
#define LCD_DISP_OFF 0x08
/*Lệnh bật con trỏ */
#define LCD_CURSOR_ON 0x04
/*Lệnh tắt con trỏ*/
#define LCD_CURSOR_OFF 0x00
/*Di chuyển con trỏ và dữ liệu hiển thị sang bên trái*/
#define LCD_CUR_MOV_LEFT 0x10
/*Di chuyển con trỏ và dữ liệu sang bên phải*/
#define LCD_CUR_MOV_RIGHT 0x14
/*Dữ liệu hiển thị được dịch sang trái*/
#define LCD_DIS_MOV_LEFT 0x18
/*Dữ liệu hiển thị được dịch sang phải*/
#define LCD_DIS_MOV_RIGHT 0x1C
/*Trạng thái LDC đang bận, dùng để kiểm tra trạng thái của LCD*/
#define LCD_BUSY 0x80

/*Các lệnh Set và Clear bit cân bản*/
#define SET_LCD_RS_Line() gpio_set_value(LCD_RS_PIN,1)
#define SET_LCD_BL_Line() gpio_set_value(LCD_BL_PIN,1)
#define SET_LCD_EN_Line() gpio_set_value(LCD_EN_PIN,1)

#define CLR_LCD_RS_Line() gpio_set_value(LCD_RS_PIN,0)
#define CLR_LCD_BL_Line() gpio_set_value(LCD_BL_PIN,0)
```

```
#define CLR_LCD_EN_Line() gpio_set_value(LCD_EN_PIN,0)

#define SET_LCD_D4_Line() gpio_set_value(LCD_D4_PIN,1)
#define SET_LCD_D5_Line() gpio_set_value(LCD_D5_PIN,1)
#define SET_LCD_D6_Line() gpio_set_value(LCD_D6_PIN,1)
#define SET_LCD_D7_Line() gpio_set_value(LCD_D7_PIN,1)

#define CLR_LCD_D4_Line() gpio_set_value(LCD_D4_PIN,0)
#define CLR_LCD_D5_Line() gpio_set_value(LCD_D5_PIN,0)
#define CLR_LCD_D6_Line() gpio_set_value(LCD_D6_PIN,0)
#define CLR_LCD_D7_Line() gpio_set_value(LCD_D7_PIN,0)
/*-----*/

/*Những lệnh trong driver hỗ trợ

void lcd_Control_Write(uint8_t data);
void lcd_Data_Write(uint8_t data);
void lcd_Home(void);
void lcd_Clear(void);
void lcd_Goto_XY(uint8_t xy);*/

/*Hàm trì hoãn thời gian chuyển đơn vị us sang jiffies*/
void lcd_delay_usec(unsigned int u) {
    long int time_delay;
    time_delay = jiffies + usecs_to_jiffies(u);
    while (time_before(jiffies, time_delay)) {
        schedule();
    }
}

/*Hàm trì hoãn thời gian chuyển đơn vị ms sang jiffies*/
void lcd_Delay_mSec (unsigned long m) {
    long int time_delay;
    time_delay = jiffies + msecs_to_jiffies(m);
    while (time_before(jiffies, time_delay)) {
        schedule();
    }
}

/*Hàm chuyển dữ liệu 4 bit thành các trạng thái trong chân gpio*/
void lcd_write_data_port(uint8_t data)
```

```
{
    (data&(1<<0)) ? SET_LCD_D4_Line() : CLR_LCD_D4_Line();
    (data&(1<<1)) ? SET_LCD_D5_Line() : CLR_LCD_D5_Line();
    (data&(1<<2)) ? SET_LCD_D6_Line() : CLR_LCD_D6_Line();
    (data&(1<<3)) ? SET_LCD_D7_Line() : CLR_LCD_D7_Line();
}

/*Hàm ghi dữ liệu 8 bits vào LCD, ghi 2 lần 4 bits vào LCDs để được 8
bits vì đang ở chế độ giao tiếp 4 bits.*/
void lcd_data_line_write(uint8_t data)
{
    /*Đầu tiên ghi 4 bits cao vào LCDs, sau đó đến 4 bit thấp*/
    /*Chuẩn bị tạo xung cạnh xuống cho EN, cho EN lên mức cao*/
    SET_LCD_EN_Line();
    /*Ghi 4 bit thấp vào các chân dữ liệu*/
    lcd_write_data_port((data>>4) & 0x000F);
    /*Tạo xung cạnh xuống ghi 4 bit thấp vào thanh ghi*/
    CLR_LCD_EN_Line();
    /*Trì hoãn một khoảng thời gian chờ thực thi lệnh*/
    lcd_delay_usec(50);
    /*Tiếp theo ghi 4 bits thấp của dữ liệu vào thanh ghi*/
    SET_LCD_EN_Line();
    lcd_write_data_port(data & 0x000F);
    CLR_LCD_EN_Line();
    lcd_delay_usec(50);
}

/*Hàm ghi mã lệnh vào thanh ghi lệnh trong LCD*/
void lcd_Control_Write(uint8_t data)
{
    /*Trì hoãn thời gian chờ thực thi những lệnh trước đó*/
    lcd_delay_usec(50);
    /*Chọn thanh ghi lệnh*/
    CLR_LCD_RS_Line();
    /*Ghi mã lệnh vào thanh ghi lệnh*/
    lcd_data_line_write(data);
}
```



*/\*Hàm ghi dữ liệu vào thanh ghi dữ liệu trong LCDs\*/*

```
void lcd_Data_Write(uint8_t data)
```

```
{
```

*/\*Trì hoãn thời gian chờ thực thi những lệnh trước đó\*/*

```
lcd_delay_usec(50);
```

*/\*Chọn thanh ghi dữ liệu trong LCDs\*/*

```
SET_LCD_RS_Line();
```

*/\*Ghi dữ liệu vào thanh ghi\*/*

```
lcd_data_line_write(data);
```

```
}
```

*/\*-----Init the LCD-----\*/*

*/\*Hàm khởi tạo các thông số cho LCD làm việc theo yêu cầu của người lập trình\*/*

```
void lcd_Init(void)
```

```
{
```

*/\*Cài đặt LCD hoạt động theo chế độ giao tiếp 4 bits\*/*

```
lcd_Control_Write(0x33);
```

*/\*Trì hoãn thời gian thực thi lệnh\*/*

```
lcd_delay_usec(1000);
```

```
lcd_Control_Write(0x32);
```

```
lcd_delay_usec(1000);
```

*/\*Cài đặt trạng thái ban đầu của LCDs\*/*

```
lcd_Control_Write(LCD_DISP_INIT);
```

*/\*Gọi lệnh xóa dữ liệu hiển thị trong LCDs\*/*

```
lcd_Control_Write(LCD_CLEAR_DISPLAY);
```

*/\*Trì hoãn thời gian 60ms\*/*

```
lcd_delay_usec(60000);
```

*/\*Cài đặt chế độ con trỏ tăng dần sau khi ghi dữ liệu\*/*

```
lcd_Control_Write(LCD_INC_MODE);
```

*/\*Bật hiển thị cho LCD\*/*

```
lcd_Control_Write(LCD_DISP_ON);
```

*/\*Chuyển con trỏ hiển thị đến vị trí bắt đầu dòng đầu tiên\*/*

```
lcd_Control_Write(LCD_RETURN_HOME);
```

```
}

/*Hàm chức năng chuyển con trỏ về vị trí bắt đầu dòng đầu tiên*/
void lcd_Home(void)
{
    lcd_Control_Write(LCD_RETURN_HOME);
}

/*Hàm chức năng xóa dữ liệu hiển thị*/
void lcd_Clear(void)
{
    lcd_Control_Write(LCD_CLEAR_DISPLAY);
}

/*Hàm chức năng bật hiển thị cho LCD*/
void lcd_Display_On(void)
{
    lcd_Control_Write(LCD_DISP_ON);
}

/*Hàm tắt hiển thị cho LCDs*/
void lcd_Display_Off(void)
{
    lcd_Control_Write(LCD_DISP_OFF);
}

/*Hàm dịch chuyển dữ liệu hiển thị về bên trái*/
void lcd_Dis_Mov_Left(void)
{
    lcd_Control_Write(LCD_DIS_MOV_LEFT);
}

/*Hàm dịch chuyển dữ liệu hiển thị về bên phải*/
void lcd_Dis_Mov_Right(void)
{
    lcd_Control_Write(LCD_DIS_MOV_RIGHT);
}

/*Hàm di chuyển con trỏ về bên trái*/
void lcd_Cursor_Move_Left(void)
{
    lcd_Control_Write(LCD_CUR_MOV_LEFT);
}
```

*/\*Hàm di chuyển con trỏ về bên phải\*/*

```
void lcd_Cursor_Move_Right(void)
{
    lcd_Control_Write(LCD_CUR_MOV_RIGHT);
}
```

```
static atomic_t lcd_open_cnt = ATOMIC_INIT(1);
```

*/\*Giao diện hàm ioctl() thực hiện chức năng do user application yêu cầu\*/*

```
static int
lcd_ioctl(struct inode * inode, struct file * file, unsigned
int cmd, unsigned long arg)
{
    int retval = 0;
    switch (cmd)
    {
        /*Chức năng nhận mã lệnh thực thi từ user application*/
        case LCD_CONTROL_WRITE:
            lcd_Control_Write(arg);
            break;
        /*Chức năng nhận dữ liệu hiển thị từ user application*/
        case LCD_DATA_WRITE:
            lcd_Data_Write(arg);
            break;
        /*Chức năng trì hoãn thời gian dùng jiffies, chuyển ms thành jiffies*/
        case LCD_DELAY_MSEC:
            lcd_Delay_mSec(arg);
            break;
        /*Chức năng trì hoãn thời gian dùng jiffies, chuyển us thành jiffies*/
        case LCD_DELAY_USEC:
            lcd_delay_usec(arg);
            break;
        /*Chức năng chuyển con trỏ về hàng đầu tiên của hàng 1*/
        case LCD_HOME:
            lcd_Home();
            break;
    }
```

```
/*Chức năng xóa hiển thị*/
case LCD_CLEAR:
    lcd_Clear();
    break;

/*Chức năng bật hiển thị*/
case LCD_DISP_ON_C:
    lcd_Display_On();
    break;

/*Chức năng xóa hiển thị*/
case LCD_DISP_OFF_C:
    lcd_Display_Off();
    break;

/*Di chuyển con trỏ và dữ liệu sang trái*/
case LCD_DIS_MOV_LEFT_C:
    lcd_Dis_Mov_Left();
    break;

/*Di chuyển con trỏ và dữ liệu sang phải*/
case LCD_DIS_MOV_RIGHT_C:
    lcd_Dis_Mov_Right();
    break;

/*Di chuyển con trỏ sang trái*/
case LCD_CUR_MOV_LEFT_C:
    lcd_Cursor_Move_Left();
    break;

/*Di chuyển con trỏ sang phải*/
case LCD_CUR_MOV_RIGHT_C:
    lcd_Cursor_Move_Right();
    break;

/*Chức năng khởi tạo LCD hoạt động theo chế độ 4 bits, 2 dòng hiển thị*/
case LCD_INIT:
    lcd_Init();
    break;

/*Trong trường hợp không có lệnh nào hỗ trợ*/
default:
```

```
        retval = -EINVAL;
        break;
    }
    return retval;
}

/*Khai báo và định nghĩa giao diện hàm open()*/
static int
lcd_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&lcd_open_cnt)) {
        atomic_inc(&lcd_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

/*Khai báo và định nghĩa giao diện hàm close()*/
static int
lcd_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&lcd_open_cnt);

    return 0;
}

/*Khai báo và định nghĩa các lệnh mà LCD hỗ trợ file operations*/
struct file_operations lcd_fops = {
    .ioctl      = lcd_ioctl,
    .open       = lcd_open,
    .release    = lcd_close,
};
```

*/\*Khai báo và định nghĩa thiết bị\*/*

```
static struct miscdevice lcd_dev = {  
    .minor          = MISC_DYNAMIC_MINOR,  
    .name           = "lcd_dev",  
    .fops           = &lcd_fops,  
};
```

*/\*Hàm được thực thi khi driver được cài đặt vào hệ thống\*/*

```
static int __init  
lcd_mod_init(void)  
{
```

```
    int i;
```

*/\*Khởi tạo các chân gpio là ngõ ra\*/*

```
gpio_request (LCD_RW_PIN, NULL);  
gpio_request (LCD_EN_PIN, NULL);  
gpio_request (LCD_RS_PIN, NULL);  
gpio_request (LCD_D4_PIN, NULL);  
gpio_request (LCD_D5_PIN, NULL);  
gpio_request (LCD_D6_PIN, NULL);  
gpio_request (LCD_D7_PIN, NULL);
```

```
at91_set_GPIO_periph (LCD_RW_PIN, 1);  
at91_set_GPIO_periph (LCD_EN_PIN, 1);  
at91_set_GPIO_periph (LCD_RS_PIN, 1);  
at91_set_GPIO_periph (LCD_D4_PIN, 1);  
at91_set_GPIO_periph (LCD_D5_PIN, 1);  
at91_set_GPIO_periph (LCD_D6_PIN, 1);  
at91_set_GPIO_periph (LCD_D7_PIN, 1);
```

```
gpio_direction_output(LCD_RW_PIN,0);  
gpio_direction_output(LCD_EN_PIN,0);  
gpio_direction_output(LCD_RS_PIN,0);  
gpio_direction_output(LCD_D4_PIN,0);  
gpio_direction_output(LCD_D5_PIN,0);  
gpio_direction_output(LCD_D6_PIN,0);  
gpio_direction_output(LCD_D7_PIN,0);
```

*/\*Lần lượt in thông báo cho người dùng, chuẩn bị cài đặt driver\*/*

```
        for (i=5;i>=0;i--) {
            lcd_delay_usec(1000000);
            printk("Please wait ... %ds\n",i);
        }
        /*Gọi hàm khởi tạo LCD*/
        lcd_Init();
        /*Gọi hàm xóa LCD*/
        lcd_Clear();
        /*In thông báo cho người dùng cài đặt thành công*/
        printk(KERN_ALERT "Welcome to our LCD world\n");
        /*Cài đặt cài đặt driver vào hệ thống*/
        return misc_register(&lcd_dev);
    }

    /*Hàm thực thi khi driver bị gỡ bỏ*/
    static void __exit
    lcd_mod_exit(void)
    {
        printk(KERN_ALERT "Goodbye for all best\n");
        misc_deregister(&lcd_dev);
    }

    /*Gán hàm init và exit vào các macro*/
    module_init (lcd_mod_init);
    module_exit (lcd_mod_exit);
    /*Những thông tin tổng quát về driver*/
    MODULE_LICENSE("GPL");
    MODULE_AUTHOR("Coolwarmboy / OpenWrt");
    MODULE_DESCRIPTION("Character device for for generic lcd
    driver");
```

### **3. Chương trình application:**

```
/*Khai báo thư viện cho các lệnh cần dùng trong chương trình*/
#include <stdint.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <getopt.h>
#include <fcntl.h>
#include <time.h> //Thư viện hỗ trợ cho các hàm thao tác thời gian

#include <sys/types.h>
#include <sys/stat.h>
#include <linux/ioctl.h>

/*Định nghĩa các số định danh lệnh cho giao diện hàm ioctl() mỗi số tương đương với một chức năng trong driver hỗ trợ*/

    /*-----LCD COMMAND DEFINED-----*/
#define IOC_LCDDEVICE_MAGIC          'B'
#define LCD_CONTROL_WRITE            _IO(IOC_LCDDEVICE_MAGIC, 15)
#define LCD_DATA_WRITE               _IO(IOC_LCDDEVICE_MAGIC, 16)
#define LCD_INIT                     _IO(IOC_LCDDEVICE_MAGIC, 17)
#define LCD_HOME                     _IO(IOC_LCDDEVICE_MAGIC, 18)
#define LCD_CLEAR                     _IO(IOC_LCDDEVICE_MAGIC, 19)
#define LCD_DISP_ON_C                _IO(IOC_LCDDEVICE_MAGIC, 21)
#define LCD_DISP_OFF_C               _IO(IOC_LCDDEVICE_MAGIC, 22)
#define LCD_CUR_MOV_LEFT_C           _IO(IOC_LCDDEVICE_MAGIC, 23)
#define LCD_CUR_MOV_RIGHT_C          _IO(IOC_LCDDEVICE_MAGIC, 24)
#define LCD_DIS_MOV_LEFT_C           _IO(IOC_LCDDEVICE_MAGIC, 25)
#define LCD_DIS_MOV_RIGHT_C          _IO(IOC_LCDDEVICE_MAGIC, 29)
#define LCD_DELAY_MSEC               _IO(IOC_LCDDEVICE_MAGIC, 30)
#define LCD_DELAY_USEC               _IO(IOC_LCDDEVICE_MAGIC, 31)

/*Định nghĩa địa chỉ đầu tiên của mỗi dòng trong LCD, ở đây điều khiển LCD có hai dòng 16 ký tự*/

#define LCD_LINE0_ADDR 0x00 //Start of line 0 in the DD-Ram
#define LCD_LINE1_ADDR 0x40 //Start of line 1 in the DD-Ram

/*Xác lập bit quy định địa chỉ con trỏ dữ liệu*/

#define LCD_DD_RAM_PTR 0x80 //Address Display Data RAM pointer

/*Định nghĩa những lệnh cơ bản thường sử dụng, làm cho chương trình gọn dễ hiểu trong quá trình lập trình*/
```



```
/*-----Functions of LCD-----*/
#define lcd_Control_Write() ioctl(fd_lcd, LCD_CONTROL_WRITE, data)
#define lcd_Data_Write()    ioctl(fd_lcd, LCD_DATA_WRITE, data)
#define lcd_Init()          ioctl(fd_lcd, LCD_INIT)
#define lcd_Home()          ioctl(fd_lcd, LCD_HOME)
#define lcd_Clear()         ioctl(fd_lcd, LCD_CLEAR)
#define lcd_Display_On()    ioctl(fd_lcd, LCD_DISP_ON_C)
#define lcd_Display_Off()   ioctl(fd_lcd, LCD_DISP_OFF_C)
#define lcd_Cursor_Move_Left() ioctl(fd_lcd, LCD_CUR_MOV_LEFT_C)
#define lcd_Cursor_Move_Right() ioctl(fd_lcd, LCD_CUR_MOV_RIGHT_C)
#define lcd_Display_Move_Left() ioctl(fd_lcd, LCD_DIS_MOV_LEFT_C)
#define lcd_Display_Move_Right() ioctl(fd_lcd, LCD_DIS_MOV_RIGHT_C)
#define lcd_Delay_mSec()    ioctl(fd_lcd, LCD_DELAY_MSEC, data)
#define lcd_Delay_uSec()    ioctl(fd_lcd, LCD_DELAY_USEC, data)
```

*/\*Phần khai báo các biến toàn cục\*/*

```
int fd_lcd; //Biến lưu số mô tả tập tin khi driver LCD được mở
int counter_value; //Biến lưu giá trị đếm hiện tại cho chức năng đếm
/*Delay for unsigned long data mSecond*/
```

*/\*Hàm dịch trái ký tự hiển thị X vị trí, do người lập trình nhập vào\*/*

```
void Display_Shift_Left(int X) {
    int i;
    for (i=0; i<X; i++) {
        /*Gọi hàm dịch trái ký tự hiển thị X lần*/
        lcd_Display_Move_Left();
    }
}
```

*/\*Hàm dịch phải ký tự hiển thị X vị trí, do người lập trình nhập vào\*/*

```
void Display_Shift_Right(int X) {
    int i;
    for (i=0; i<X; i++) {
        /*Gọi hàm dịch phải X lần*/
        lcd_Display_Move_Right();
    }
}
```

```
    }  
}  
  
/*Di chuyển con trỏ hiển thị đến vị trí x, y. Trong đó x là số thứ tự dòng của LCD  
bắt đầu từ 0; y là số thứ tự cột của LCD bắt đầu từ 0*/  
void lcd_Goto_XY(uint8_t x, uint8_t y)  
{  
    /*Định nghĩa thanh ghi dữ liệu con trỏ của Data Display RAM*/  
    register uint8_t DDRAMAddr;  
    /*So sánh số x để chọn số dòng cho phù hợp*/  
    switch(x)  
    {  
        /*Trong trường hợp dòng thứ nhất, dòng 0. Tiến hành cộng giá trị địa chỉ  
        đầu dòng cho số cột y*/  
        case 0: DDRAMAddr = LCD_LINE0_ADDR+y; break;  
        /*Trong trường hợp dòng thứ hai, dòng 1. Tiến hành cộng giá trị địa chỉ  
        đầu dòng cho số cột y*/  
        case 1: DDRAMAddr = LCD_LINE1_ADDR+y; break;  
        /*Trong trường hợp dòng 3, 4. Trường hợp này áp dụng cho LCD 16x4.  
        Thuật toán cũng tương tự như 2 trường hợp trên*/  
        //case 2: DDRAMAddr = LCD_LINE2_ADDR+y; break; for LCD 16x4 or  
        //20x4 only  
        //case 3: DDRAMAddr = LCD_LINE3_ADDR+y; break;  
        /*Trong những trường hợp khác vẫn tính là dòng thứ 1*/  
        default: DDRAMAddr = LCD_LINE0_ADDR+y;  
    }  
    /*Cuối cùng gọi hàm giao diện ioctl() để đặt giá trị địa chỉ đã định nghĩa  
    phía trên*/  
    ioctl(fd_lcd, LCD_CONTROL_WRITE, LCD_DD_RAM_PTR | DDRAMAddr);  
}  
  
/*Hàm xuất chuỗi ký tự tại vị trí bắt đầu là dòng x và cột y*/  
void Display_Print_Data(char *string, uint8_t x, uint8_t y)  
{  
    /*Đầu tiên di chuyển con trỏ đến vị trí mong muốn x và y*/  
    lcd_Goto_XY(x, y);
```

*/\*Xuất lần lượt từng ký tự trong chuỗi string ra LCD hiển thị. Lặp lại cho đến khi kết thúc chuỗi \*/*

```
while (*string) {  
    ioctl(fd_lcd, LCD_DATA_WRITE, *string++);  
}
```

```
}
```

*/\*Hàm xuất một ký tự tại vị trí bắt đầu là dòng x và cột y\*/*

```
void Display_Print_Char(int data, uint8_t x, uint8_t y)
```

```
{
```

*/\*Di chuyển con trỏ đến vị trí cần in ký tự ra LCD\*/*

```
    lcd_Goto_XY(x,y);
```

*/\*Gọi hàm xuất ký tự ra LCD qua giao diện ioctl()\*/*

```
    lcd_Data_Write ();
```

```
}
```

*/\*Chương trình con khởi tạo hiển thị LCD\*/*

```
void Display_Init_Display (void) {
```

*/\*Biến lưu số lần chớp tắt hiển thị, dùng cho vòng lặp for()\*/*

```
    int i;
```

*/\*Gọi hàm xóa hiển thị của LCD\*/*

```
    lcd_Clear();
```

*/\* Xuất chuỗi ký tự thông báo khởi tạo LCD thành công\*/*

```
    Display_Print_Data ("LCD display",0,2);
```

```
    Display_Print_Data ("I am your slave!",1,0);
```

*/\*Chớp tắt hiển thị 3 lần\*/*

```
    for (i = 0; i < 3; i ++ ) {
```

*/\*Gọi hàm tắt hiển thị\*/*

```
        lcd_Display_Off();
```

*/\*Trì hoãn thời gian trong 500ms\*/*

```
        ioctl(fd_lcd, LCD_DELAY_MSEC, 500);
```

*/\*Gọi hàm bật hiển thị\*/*

```
        lcd_Display_On();
```

*/\*Trì hoãn thời gian trong 500ms\*/*

```
        ioctl(fd_lcd, LCD_DELAY_MSEC, 500);
```

```
    }
```

*/\*Xóa hiển thị kết thúc lời chào\*/*

```
    lcd_Clear();
```

```
}
```

*/\*Chương trình con in thông báo lỗi ra LCD\*/*

```
void Display_print_error_0(void) {
```

```
    Display_Print_Data ("LCD information:",1,2);
```

```
    Display_Print_Data ("Error while typing!!!",0,0);
```

```
}
```

*/\*Hàm in ra hướng dẫn cho người dùng trong trường hợp người dùng nhập sai cú pháp lệnh\*/*

```
int print_usage(void) {
```

```
    printf("The command you required is not supported\n");
```

```
    printf("./4_lcd_app
```

```
display_string|display_time|display_counter      <string>|<XX>
```

```
<YY> <Period>\n");
```

```
    Display_print_error_0();
```

```
    return -1;
```

```
}
```

```
////////////////////////////////////
```

*/\*Hàm xuất chuỗi ký tự hỗ trợ cho chế độ xuất ký tự của chương trình\*/*

```
void display_string(char *string) {
```

```
    Display_Print_Data("Your string is:",0,0);
```

```
    Display_Print_Data(string,1,0);
```

```
}
```

*/\*Hàm cập nhật thời gian hiện tại và xuất ra LCD hiển thị. Hỗ trợ cho chế độ hiển thị thời gian của chương trình\*/*

```
void display_time(void) {
```

*/\*Biến lưu cấu trúc thời gian theo dạng năm tháng ngày giờ ... \*/*

```
    struct tm *tm_ptr;
```

*/\*Biến lưu thời gian hiện tại của hệ thống\*/*

```
    time_t the_time;
```

*/\*Hiển thị các thông tin ban đầu trên LCD\*/*

```
    Display_Print_Data ("Date:   /   /   ",0,0);
```

```
    Display_Print_Data ("Time:   :   :   ",1,0);
```

*/\*Vòng lặp cập nhật thời gian hiện tại và xuất dữ liệu ra LCD\*/*

```
while (1) {  
    /*Lấy thời gian hiện tại của hệ thống*/  
    (void) time(&the_time);  
    /*Chuyển đổi thời gian hiện tại của hệ thống sang cấu trúc thời gian dạng struc  
tm*/  
    tm_ptr = gmtime(&the_time);  
    /*Lần lượt chuyển đổi các giá trị thời gian sang ký tự và xuất ra LCD. Thuật toán  
chuyển đổi:  
    - Chuyển sang số BCD từ số nguyên có 2 chữ số;  
    - Lần lượt cộng các số BCD cho 48 để chuyển sang các ký số trong bảng mã  
ascii*/  
    /*Chuyển đổi và hiển thị năm ra LCD*/  
    Display_Print_Char(((tm_ptr->tm_year)/10)+48,0,6);  
    Display_Print_Char(((tm_ptr->tm_year)%10)+48,0,7);  
    /*Chuyển đổi và hiển thị tháng ra LCD*/  
    Display_Print_Char(((tm_ptr->tm_mon+1)/10)+48,0,9);  
    Display_Print_Char(((tm_ptr->tm_mon+1)%10)+48,0,10);  
    /*Chuyển đổi và hiển thị ngày ra LCD*/  
    Display_Print_Char(((tm_ptr->tm_mday)/10)+48,0,12);  
    Display_Print_Char(((tm_ptr->tm_mday)%10)+48,0,13);  
    /*Chuyển đổi và hiển thị giờ ra LCD*/  
    Display_Print_Char(((tm_ptr->tm_hour)/10)+48,1,6);  
    Display_Print_Char(((tm_ptr->tm_hour)%10)+48,1,7);  
    /*Chuyển đổi và hiển thị giờ ra LCD*/  
    Display_Print_Char(((tm_ptr->tm_min)/10)+48,1,9);  
    Display_Print_Char(((tm_ptr->tm_min)%10)+48,1,10);  
    /*Chuyển đổi và hiển thị giờ ra LCD*/  
    Display_Print_Char(((tm_ptr->tm_sec)/10)+48,1,12);  
    Display_Print_Char(((tm_ptr->tm_sec)%10)+48,1,13);  
    /*Trì hoãn cập nhật thời gian*/  
    usleep(200000);  
}  
}
```

////////////////////////////////////

```
/*Các hàm hỗ trợ cho chức năng đếm số của chương trình*/
/*Hàm chuyển đổi số từ 00 đến 99 thành 2 chữ số hiển thị ra LCD tại vị trí tương ứng start_v_pos (cột bắt đầu) và h_pos(dòng bắt đầu)*/
void Display_Number(int number, int start_v_pos, int h_pos) {
    Display_Print_Char((number/10)+48,h_pos,start_v_pos);
    Display_Print_Char((number%10)+48,h_pos,start_v_pos+1);
}
/*Chương trình con đếm chính trong chức năng đếm số*/
void display_counter(int lmt1, int lmt2, int period_ms) {
/*Biến lưu chu kỳ thời gian us */
    long int period_us;
/*Biến lưu số lần chớp tắt hiển thị khi quá trình đếm thành công*/
    int i;
/*In thông tin cố định ban đầu lên LCD */
    Display_Print_Data ("Start:   End:   ",0,0);
    Display_Print_Data ("Count Value:",1,0);
/*Hiển thị số giới hạn Start*/
    Display_Number(lmt1,6,0);
/*Hiển thị số giới hạn End*/
    Display_Number(lmt2,13,0);
/*Cập nhật giá trị ban đầu cho counter_value*/
    counter_value = lmt1;
/*Chuyển thời gian ms sang us dùng cho hàm usleep()*/
    period_us = period_ms*1000;
/*Quá trình đếm bắt đầu, đếm từ giá trị Start cho đến giá trị End*/
    while (counter_value != lmt2) {
/*Hiển thị giá trị đếm hiện tại lên LCD*/
        Display_Number(counter_value,12,1);
/*Trì hoãn thời gian đếm*/
        usleep(period_us);
/*Tăng hoặc giảm giá trị đếm hiện tại cho đến khi bằng với giá trị End*/
        if (counter_value < lmt2) {
            counter_value++;
        } else {

```

```
        counter_value--;
    }
}

/*Khi quá trình đếm kết thúc in ra thông báo cho người dùng biết*/
    Display_Print_Data("Count complete!", 1, 0);
/*Chớp tắt 3 lần trong với tần số 1Hz*/
    for (i = 0; i < 3; i ++ ) {
        lcd_Display_Off();
        ioctl(fd_lcd, LCD_DELAY_MSEC, 500);
        lcd_Display_On();
        ioctl(fd_lcd, LCD_DELAY_MSEC, 500);
    }
}

/*Chương trình chính khai báo dưới dạng nhận tham số từ người dùng*/
int
main(int argc, char **argv)
{
    /*Mở tập tin thiết bị trước thao tác, lấy về số mô tả tập tin lưu trong biến fd_lcd.
    Kiểm tra lỗi trong quá trình thao tác*/
    if ((fd_lcd = open("/dev/lcd_dev", O_RDWR)) < 0)
    {
        printf("Error whilst opening /dev/lcd_dev\n");
        return -1;
    }

    /*Gọi các hàm khởi tạo LCD ban đầu*/
    printf("LCD initializing ...\n");
    lcd_Init();
    lcd_Clear();
    Display_Init_Display();

    /*So sánh các tham số lựa chọn chức năng thực thi chương trình*/
    switch (argc) {
        case 2:
            /*Trong trường hợp hiển thị thời gian*/
            if (!strcmp(argv[1], "display_time")) {
```

```
        display_time();
    } else {
        return print_usage();
    }
    break;
    case 3:
        /*Trong trường hợp hiển thị chuỗi ký tự*/
        if (!strcmp(argv[1], "display_string")) {
            display_string(argv[2]);
        } else {
            return print_usage();
        }
        break;
    case 5:
        /*Trong trường hợp đếm số hiển thị trong khoảng từ 00 đến 99*/
        if (!strcmp(argv[1], "display_counter")) {
            display_counter(atoi(argv[2]), atoi(argv[3]),
                            atoi(argv[4]));
        } else {
            return print_usage();
        }
        break;
    default:
        /*Trong trường hợp không có lệnh nào hỗ trợ in ra thông báo hướng dẫn cho người dùng */
        return print_usage();
        break;
    }
    return 0;
}
```

#### **4. Biên dịch và thực thi chương trình:**

Người học tự biên dịch chương trình *driver* và chương trình *application*. Sau khi biên dịch chép vào kit tiến hành kiểm tra kết quả trong từng trường hợp tương ứng với từng chức năng khác nhau của chương trình. Rút ra nhận xét và kinh nghiệm thực hiện.



**III. Kết luận và bài tập:****a. Kết luận:**

Trong bài này chúng ta đã nghiên cứu thêm một ứng dụng nữa trong việc điều khiển các chân gpio bằng các lệnh set và clear từng chân cơ bản. Chúng ta đã thực hành sử dụng các lệnh thao tác với thời gian trong việc cập nhật và hiển thị trên thiết bị phần cứng LCD.

Chương trình *driver* và *application* trong dự án này mặc dù đã điều khiển thành công LCD nhưng chương trình vẫn còn chưa tối ưu như: Những thao tác điều khiển LCD vẫn còn nằm trong *user application* quá nhiều, khó áp dụng *driver* sang những dự án khác. Các bạn sẽ thực hiện tối ưu hóa hoạt động của *driver* LCD thông qua những bài tập yêu cầu trong phần sau.

**b. Bài tập:**

1. Thêm chức năng di chuyển con trỏ đến vị trí x và y bất kỳ trên LCD trong *driver* điều khiển LCD trong ví dụ trên. Trong đó: x là vị trí cột, y là vị trí dòng của LCD 16x2. Gợi ý: Chức năng tương tự như hàm void `lcd_Goto_XY(uint8_t x, uint8_t y)` trong *user application* nhưng chuyển qua *driver* thực hiện thông qua giao diện hàm `ioctl()`.
2. Thêm chức năng hiển thị chuỗi ký tự vào *driver* điều khiển LCD theo yêu cầu sau:
  - *Driver* nhận chuỗi ký tự thông qua giao diện hàm `write()` được gọi từ *user application*;
  - Chương trình *driver* thực hiện ghi những ký tự này vào LCD hiển thị.

*\*\*Chức năng này của driver tương tự như hàm void `Display_Print_Data(char *string, uint8_t x, uint8_t y)` trong user application.*
3. Biên dịch *driver* và lưu vào thư viện để có thể sử dụng trong những dự án khác.

**BÀI 5****GIAO TIẾP ĐIỀU KHIỂN  
GPIO NGÕ VÀO ĐẾM XUNG****I. Phác thảo dự án:**

Trong các bài trước chúng ta chỉ sử dụng các chân của Vi Xử Lý ở chế độ gpio ngõ ra (output). Khi sử dụng các chân của Vi Xử Lý ở chế độ output thì chúng ta có thể điều khiển các thiết bị ngoại vi (led đơn, led 7 đoạn, ic...), nhưng trong thực tế có những trường hợp Vi Xử Lý phải cập nhật các tín hiệu từ bên ngoài (tín hiệu của nút nhấn, ma trận phím, ...). Trong những trường hợp này chúng ta phải dùng chân của Vi Xử Lý ở chế độ gpio ngõ vào (input).

Ngoài ra, trong các dự án trước, với một ứng dụng chúng ta viết một chương trình Driver và một chương trình User Application. Với phương pháp lập trình này, chúng ta đã phần nào đó hiểu được điểm mạnh của việc lập trình ứng dụng bằng phương pháp nhúng trong việc chia nhiệm vụ cho Driver và User Application để điều khiển thiết bị. Nhưng trong dự án này chúng ta sẽ làm quen với phương pháp lập trình nhúng mới là sử dụng một User Application điều khiển nhiều Driver khác nhau (có nghĩa là với một ứng dụng chúng ta có thể viết nhiều Driver khác nhau và dùng một chương trình User Application để điều khiển các Driver này). Có thể nói phương pháp này sẽ làm nổi bật lên điểm mạnh của việc lập trình ứng dụng trên hệ thống nhúng một cách rõ rệt hơn so với phương pháp lập trình thông thường cho Vi Xử Lý. Đối với các thiết bị, modun khác nhau chúng ta có thể viết các Driver để điều khiển các modun này (mỗi modun một Driver) và có thể chạy các Driver này cùng lúc với. Như vậy chúng ta có thể điều khiển nhiều thiết bị trong cùng một lúc một cách độc lập.

Chú ý: để tránh sự xung khắc giữa các Driver với nhau chúng ta không được sử dụng một chân của Vi Xử Lý cho nhiều Driver. Một chân của Vi Xử Lý chỉ được dùng cho một Driver duy nhất.

**a. Yêu cầu dự án:**

Trong dự án này chúng ta sẽ đếm xung từ bên ngoài (nút nhấn) và hiển thị số xung đếm được trên 8 led 7 đoạn theo phương pháp quét.

Cú pháp lệnh của chương trình như sau:

```
./< counter_display_7seg_app > <start>
```

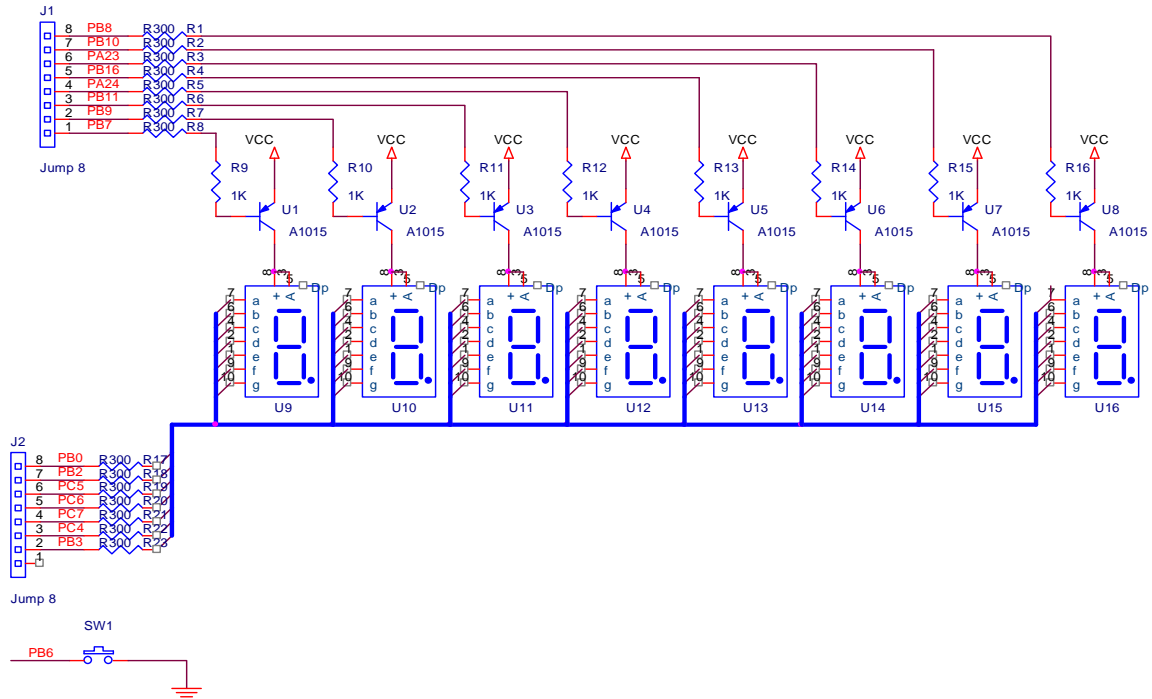
Trong đó:

< counter\_display\_7seg\_app > là tên chương trình User Application sau khi đã biên dịch.

< start > là lệnh bắt đầu chương trình đếm xung.

### b. Phân công nhiệm vụ:

- **Driver:** trong dự án này Driver sẽ có hai nhiệm vụ là:
  - *Nhiệm vụ thứ nhất:* cập nhật xung đếm từ bên ngoài, nếu có xung tác động thì sẽ gửi tín hiệu sang cho User Application.
  - *Nhiệm vụ thứ hai:* nhận dữ liệu từ User Application, điều khiển modul Led 7 đoạn theo phương pháp quét thể hiện thị số xung đếm được.Như đã nói từ trước, trong dự án này chúng ta sẽ viết hai Driver. Driver thứ nhất làm nhiệm vụ thứ nhất. Driver thứ hai làm nhiệm vụ thứ hai.
- **User Application:** có nhiệm vụ cập nhật tín hiệu của Driver thứ nhất, nếu tín hiệu báo có xung thì sẽ tăng biến đếm xung lên một đơn vị, sau đó truyền biến đếm xung này sang cho Driver thứ hai để hiển thị lên modul Led 7 đoạn.

**II. Thực hiện:****1. Kết nối phần cứng:****2. Chương trình driver:**

- **Driver thứ nhất:** Tên update\_input\_dev.c

*/\* Khai báo các thư viện cần thiết cho chương trình \*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <mach/at91_tc.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>
#include <linux/clk.h>
#include <linux/irq.h>
#include <linux/time.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
```

```
#include <linux/delay.h>
#include <linux/timer.h>

/* Khai báo tên Driver và tên thiết bị */
#define DRVNAME      "update_input_dev"
#define DEVNAME      "update_input"

/* Định nghĩa chân của Vi Xử Lý dùng trong chương trình */
#define COUNTER_PIN  AT91_PIN_PB6

/* Khai báo biến kiểm tra Driver đã được mở hay chưa */
static atomic_t update_input_open_cnt = ATOMIC_INIT(1);

/* Chương trình cập nhật trạng thái của chân Vi Xử Lý (COUNTER_PIN) dùng
để đếm xung từ bên ngoài */
static ssize_t update_input_read (struct file *filp, char
__iomem buf[], size_t bufsize, loff_t *f_pos)
{
    /* Khai báo bộ đệm đọc */
    char driver_read_buf[1];

    /* Nếu chân COUNTER_PIN được nối đất (giá trị đọc vào từ chân này là 0) thì
cho driver_read_buf[0] bằng 0, ngược lại thì cho driver_read_buf[0] = 1 */
    if(gpio_get_value(COUNTER_PIN) == 0)
        driver_read_buf[0] = 0;
    else
        driver_read_buf[0] = 1;

    /* Truyền driver_read_buf sang cho User Application, nếu quá trình truyền thất
bại thì báo lỗi cho người dùng biết */
    if(copy_to_user(buf, driver_read_buf, bufsize) != 0)
    {
        printk("Can't read Driver \n");
        return -EFAULT;
    }
}

/* Chương trình mở Driver */
static int
update_input_open(struct inode *inode, struct file *file)
{
    int result = 0;
```

```
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&update_input_open_cnt)) {
        atomic_inc(&update_input_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
        already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

/* Chương trình đóng Driver */
static int
update_input_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&update_input_open_cnt);

    return 0;
}

/* Cấu trúc file_operations của Driver */
struct file_operations update_input_fops = {
    .read = update_input_read,
    .open  = update_input_open,
    .release = update_input_close,
};

static struct miscdevice update_input_dev = {
    .minor          = MISC_DYNAMIC_MINOR,
    .name           = "update_input",
    .fops           = &update_input_fops,
};

/* Chương trình khởi tạo của Driver, được thực hiện khi người dùng gọi lệnh
insmod */
static int __init
update_input_mod_init(void)
{

```

```
int ret=0;

/* Khai báo khởi tạo chân COUNTER_PIN ở chế độ gpio ngõ vào có điện trở
kéo lên bên trong */

gpio_free(COUNTER_PIN);
gpio_request (COUNTER_PIN, NULL);
at91_set_GPIO_periph (COUNTER_PIN, 1);
gpio_direction_input(COUNTER_PIN);
at91_set_deglitch(COUNTER_PIN,1);
misc_register(&update_input_dev);
printk(KERN_INFO "sweep_seg_led: Loaded module\n");
return ret;
}

static void __exit
update_input_mod_exit(void)
{
    misc_deregister(&update_input_dev);
    printk(KERN_INFO "sweep_seg_led: Unloaded module\n");
}

module_init (update_input_mod_init);
module_exit (update_input_mod_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("TranCamNhan");
MODULE_DESCRIPTION("Character device for for generic gpio
api");
```

- **Driver thứ hai:** Tên sweep\_7seg\_led\_dev.c

```
/* Khai báo các thư viện cần thiết cho chương trình */
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <mach/at91_tc.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>
```

```
#include <linux/clock.h>
#include <linux/irq.h>
#include <linux/time.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/delay.h>
#include <linux/timer.h>

/* Khai báo tên Driver và tên thiết bị */

#define DRVNAME      "sweep_7seg_led_dev"
#define DEVNAME      "sweep_7seg_led"

/* Định nghĩa các chân của Vi Xử Lý sử dụng trong chương trình */

#define P00          AT91_PIN_PB8
#define P01          AT91_PIN_PB10
#define P02          AT91_PIN_PA23
#define P03          AT91_PIN_PB16
#define P04          AT91_PIN_PA24
#define P05          AT91_PIN_PB11
#define P06          AT91_PIN_PB9
#define P07          AT91_PIN_PB7
#define A            AT91_PIN_PB0
#define B            AT91_PIN_PB2
#define C            AT91_PIN_PC5
#define D            AT91_PIN_PC6
#define E            AT91_PIN_PC7
#define F            AT91_PIN_PC4
#define G            AT91_PIN_PB3

/* Định nghĩa lệnh Set và Clear các chân của Vi Xử Lý */

#define SET_P00()     gpio_set_value(P00,1)
#define SET_P01()     gpio_set_value(P01,1)
#define SET_P02()     gpio_set_value(P02,1)
#define SET_P03()     gpio_set_value(P03,1)
#define SET_P04()     gpio_set_value(P04,1)
#define SET_P05()     gpio_set_value(P05,1)
#define SET_P06()     gpio_set_value(P06,1)
#define SET_P07()     gpio_set_value(P07,1)
```



```
#define CLEAR_P00() gpio_set_value(P00,0)
#define CLEAR_P01()      gpio_set_value(P01,0)
#define CLEAR_P02()      gpio_set_value(P02,0)
#define CLEAR_P03()      gpio_set_value(P03,0)
#define CLEAR_P04()      gpio_set_value(P04,0)
#define CLEAR_P05()      gpio_set_value(P05,0)
#define CLEAR_P06()      gpio_set_value(P06,0)
#define CLEAR_P07()      gpio_set_value(P07,0)

#define SET_A()          gpio_set_value(A,1)
#define SET_B()          gpio_set_value(B,1)
#define SET_C()          gpio_set_value(C,1)
#define SET_D()          gpio_set_value(D,1)
#define SET_E()          gpio_set_value(E,1)
#define SET_F()          gpio_set_value(F,1)
#define SET_G()          gpio_set_value(G,1)

#define CLEAR_A()        gpio_set_value(A,0)
#define CLEAR_B()        gpio_set_value(B,0)
#define CLEAR_C()        gpio_set_value(C,0)
#define CLEAR_D()        gpio_set_value(D,0)
#define CLEAR_E()        gpio_set_value(E,0)
#define CLEAR_F()        gpio_set_value(F,0)
#define CLEAR_G()        gpio_set_value(G,0)

/*Định nghĩa số định danh lệnh và lệnh sử dụng trong ioctl */
#define SWEEP_LED_DEV_MAGIC  'B'
#define UPDATE_DATA_SWEEP_7SEG _IO(SWEEP_LED_DEV_MAGIC,1)

/* Khai báo các biến cần thiết sử dụng trong chương trình */
static atomic_t sweep_7seg_led_open_cnt = ATOMIC_INIT(1);
char DataDisplay[8]={0,12,12,12,12,12,12,12};
char SevSegCode[]={ 0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92, 0x82,
                    0xF8, 0x80, 0x90, 0x3F, 0x77, 0xFF};
char ChooseLedActive[]={ 0xfe, 0xfd, 0xfb, 0xf7, 0xef, 0xdf,
                          0xbf, 0x7f};

int i=0;

/* Khai báo cấu trúc timer ảo trong kernel với tên my_timer */
```

```
struct timer_list my_timer;

/* Chương trình chọn Led 7 đoạn tích cực */
void choose_led_active(char data) {
    (data&(1<<0)) ? SET_P00() : CLEAR_P00();
    (data&(1<<1)) ? SET_P01() : CLEAR_P01();
    (data&(1<<2)) ? SET_P02() : CLEAR_P02();
    (data&(1<<3)) ? SET_P03() : CLEAR_P03();
    (data&(1<<4)) ? SET_P04() : CLEAR_P04();
    (data&(1<<5)) ? SET_P05() : CLEAR_P05();
    (data&(1<<6)) ? SET_P06() : CLEAR_P06();
    (data&(1<<7)) ? SET_P07() : CLEAR_P07();
}

/* Chương trình viết dữ liệu hiển thị ra các chân Vi Xử Lý */
void write_data_led(char data) {
    (data&(1<<0)) ? SET_A() : CLEAR_A();
    (data&(1<<1)) ? SET_B() : CLEAR_B();
    (data&(1<<2)) ? SET_C() : CLEAR_C();
    (data&(1<<3)) ? SET_D() : CLEAR_D();
    (data&(1<<4)) ? SET_E() : CLEAR_E();
    (data&(1<<5)) ? SET_F() : CLEAR_F();
    (data&(1<<6)) ? SET_G() : CLEAR_G();
}

/* Chương trình chuyển một số có nhiều chữ số ra các số BCD */
void hex_to_bcd (unsigned long int data) {
    DataDisplay[0] = data % 10;
    DataDisplay[1] = (data % 100) / 10;
    DataDisplay[2] = (data % 1000) / 100;
    DataDisplay[3] = (data % 10000) / 1000;
    DataDisplay[4] = (data % 100000) / 10000;
    DataDisplay[5] = (data % 1000000) / 100000;
    DataDisplay[6] = (data % 10000000) / 1000000;
    DataDisplay[7] = data / 10000000;

    /* Chương trình xóa số 0 vô nghĩa */
    if (data / 10000000 == 0) {
        DataDisplay[7] = 12;
        if ((data % 10000000) / 1000000 == 0) {
```

```
        DataDisplay[6] = 12;
    if ( (data % 1000000)/100000 == 0) {
        DataDisplay[5] = 12;
        if ((data % 100000)/10000 == 0) {
            DataDisplay[4] = 12;
            if ( (data % 10000)/1000 == 0) {
                DataDisplay[3] = 12;
                if( (data % 1000)/100 == 0) {
                    DataDisplay[2] = 12;
                    if( (data % 100)/10 == 0)
                        DataDisplay[1] = 12;
                }
            }
        }
    }
}
```

*/\* Chương trình phục vụ ngắt: cứ mỗi 1ms chương trình này sẽ được hiện một lần, chương trình này có nhiệm vụ chọn lần lượt chọn các Led tích cực và xuất dữ liệu hiển thị ứng với từng Led \*/*

```
void
sweep_dis_irq(unsigned long data) {
    choose_led_active(ChooseLedActive[i]);
    write_data_led(SevSegCode[DataDisplay[i]]);
    i++;
    if(i==8)
        i=0;
}
```

*/\* Khởi tạo lại giá trị của timer ảo với chu kỳ là 1ms\*/*

```
mod_timer (&my_timer, jiffies + 1);
}
```

*/\* chương trình nhận dữ liệu từ User Application và chuyển dữ liệu vừa nhận sang các số BCD \*/*

```
static int
sweep_7seg_led_ioctl(struct inode * inode, struct file * file,
unsigned int cmd,unsigned long int arg) {
```

```
int retval=0;
switch (cmd) {
    case UPDATE_DATA_SWEEP_7SEG:
        hex_to_bcd(arg);
        break;
    default:
        printk("The function you type does not exist\n");
        retval=-1;
        break;
}
}

static int
sweep_7seg_led_open(struct inode *inode, struct file *file) {
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&sweep_7seg_led_open_cnt)) {
        atomic_inc(&sweep_7seg_led_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

static int
sweep_7seg_led_close(struct inode * inode, struct file * file){
    smp_mb__before_atomic_inc();
    atomic_inc(&sweep_7seg_led_open_cnt);
    return 0;
}

struct file_operations sweep_7seg_led_fops = {
    .ioctl    = sweep_7seg_led_ioctl,
    .open     = sweep_7seg_led_open,
    .release  = sweep_7seg_led_close,
};

static struct miscdevice sweep_7seg_led_dev = {
```

```
.minor          = MISC_DYNAMIC_MINOR,
.name           = "sweep_7seg_led",
.fops           = &sweep_7seg_led_fops,
};

static int __init
sweep_7seg_led_mod_init(void) {
    int ret=0;

    /* Khởi tạo các chân của Vi Xử Lý ở chế độ ngõ ra, có điện trở kéo lên */

    gpio_request (P00, NULL);
    gpio_request (P01, NULL);
    gpio_request (P02, NULL);
    gpio_request (P03, NULL);
    gpio_request (P04, NULL);
    gpio_request (P05, NULL);
    gpio_request (P06, NULL);
    gpio_request (P07, NULL);
    at91_set_GPIO_periph (P00, 1);
    at91_set_GPIO_periph (P01, 1);
    at91_set_GPIO_periph (P02, 1);
    at91_set_GPIO_periph (P03, 1);
    at91_set_GPIO_periph (P04, 1);
    at91_set_GPIO_periph (P05, 1);
    at91_set_GPIO_periph (P06, 1);
    at91_set_GPIO_periph (P07, 1);
    gpio_direction_output(P00, 0);
    gpio_direction_output(P01, 0);
    gpio_direction_output(P02, 0);
    gpio_direction_output(P03, 0);
    gpio_direction_output(P04, 0);
    gpio_direction_output(P05, 0);
    gpio_direction_output(P06, 0);
    gpio_direction_output(P07, 0);
    gpio_request (A, NULL);
    gpio_request (B, NULL);
    gpio_request (C, NULL);
    gpio_request (D, NULL);
    gpio_request (E, NULL);
```

```
gpio_request (F, NULL);
gpio_request (G, NULL);
at91_set_GPIO_periph (A, 1);
at91_set_GPIO_periph (B, 1);
at91_set_GPIO_periph (C, 1);
at91_set_GPIO_periph (D, 1);
at91_set_GPIO_periph (E, 1);
at91_set_GPIO_periph (F, 1);
at91_set_GPIO_periph (G, 1);
gpio_direction_output(A, 0);
gpio_direction_output(B, 0);
gpio_direction_output(C, 0);
gpio_direction_output(D, 0);
gpio_direction_output(E, 0);
gpio_direction_output(F, 0);
gpio_direction_output(G, 0);
/* Khởi tạo timer ảo */
init_timer (&my_timer);
my_timer.expires = jiffies+1; //chu kỳ đầu tiên là 1ms
my_timer.data = 0;
my_timer.function = sweep_dis_irq;
add_timer (&my_timer);

misc_register(&sweep_7seg_led_dev);
printk(KERN_INFO "sweep_seg_led: Loaded module\n");
return ret;
}

static void __exit
sweep_7seg_led_mod_exit(void){
    del_timer_sync(&my_timer);
    misc_deregister(&sweep_7seg_led_dev);
    printk(KERN_INFO "sweep_seg_led: Unloaded module\n");
}

module_init (sweep_7seg_led_mod_init);
module_exit (sweep_7seg_led_mod_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("TranCamNhan");
```

```
MODULE_DESCRIPTION("Character device for for generic gpio  
api");
```

### **3. Chương trình application:**

- **User Application:** counter\_display\_7seg\_app.c

```
/* Khai báo các thư viện cần thiết dùng trong chương trình */
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <linux/ioctl.h>  
#include <pthread.h>  
#include <unistd.h>
```

```
/* Định nghĩa số định danh lệnh và lệnh dùng trong ioctl */
```

```
#define SWEEP_LED_DEV_MAGIC 'B'  
#define UPDATE_DATA_SWEEP_7SEG _IO(SWEEP_LED_DEV_MAGIC, 1)
```

```
/* Khai báo các biến cần thiết trong chương trình */
```

```
int update_input_fd;  
int sweep_7seg_led_fd;
```

```
/* Chương trình in ra trên màn hình cú pháp lệnh sử dụng chương trình */
```

```
int print_usage(void) {  
    printf("./counter_display_7seg_app <number>\n");  
    return -1;  
}
```

```
/* Chương trình chính */
```

```
int  
main(int argc, char **argv) {  
    char user_read_buf[1];  
    long int counter = 0;
```

```
/* Mở Driver update_input_dev.ko */
```

```
    if ((update_input_fd = open("/dev/update_input", O_RDWR)) <  
0) {  
        printf("Error whilst opening /dev/update_input  
device\n");  
        return -1;
```

```
    }  
  
    /* Mở Driver sweep_7seg_led_dev.ko */  
  
    if ((sweep_7seg_led_fd = open("/dev/sweep_7seg_led", O_RDWR))  
        < 0) {  
        printf("Error whilst opening /dev/sweep_7seg_led  
        device\n");  
        return -1;  
    }  
  
    if (argc != 2)  
        print_usage();  
    else if (!strcmp (argv[1], "start"))  
        while(1) {  
  
            /* Chương trình đọc trạng thái chân đếm xung từ Driver update_input_dev. */  
  
            if (read(update_input_fd, user_read_buf, 1) < 0)  
                printf("User: read from driver fail\n");  
            else {  
  
                /* Nếu chân đếm xung ở mức thấp thì sẽ tăng biến counter lên một đơn vị và  
                truyền biến counter này sang cho Driver sweep_7seg_led_dev.ko */  
  
                if (user_read_buf[0] == 0) {  
                    usleep(50000); //Delay để chống dội  
                    counter++;  
                    ioctl(sweep_7seg_led_fd,  
                        UPDATE_DATA_SWEEP_7SEG, counter);  
                    printf("value counter: %d\n", counter);  
  
                    /* Nếu chân đếm xung vẫn ở mức thấp thì tiếp tục cập nhật trạng thái của chân  
                    này, khi nào trạng thái của chân này lên cao thì mới ngưng cập nhật trạng thái  
                    chân này */  
  
                    while (user_read_buf[0] == 0)  
                    {  
                        read(update_input_fd, user_read_buf, 1);  
                    }  
                    usleep(50000); //Delay để chống dội  
                }  
            }  
        }  
    }  
    else
```



```
        print_usage();  
    return 0;  
}
```

### **III. Kết luận và bài tập:**

#### **1. Kết Luận:**

Như vậy trong dự án này chúng ta đã đếm xung cạnh xuống chân PB6 của Vi Xử Lý và hiển thị số xung đếm được ra 8 Led 7 đoạn theo phương pháp quét.

Chúng ta đã viết hai Driver khác nhau và một User Application để thực hiện yêu cầu của dự án này. Thực ra đối với dự án này chúng ta có thể chỉ cần viết một Driver và một User Application là có thể thực hiện được yêu cầu. Nên với cách viết chương trình như bài này là nhằm giới thiệu đến cho người học phương pháp viết nhiều Driver cho một ứng dụng, để chúng ta có thể áp dụng phương pháp này giải quyết các ứng dụng có yêu cầu phức tạp hơn.

Trong bài này chúng ta cũng đã tập làm quen với việc sử dụng ngắt timer ảo của kernel với chu kỳ 1ms.

Chú ý: Trong bài này chúng ta đã thay đổi giá trị của số HZ là 1000. Nghĩa là chúng ta sẽ có 1000 tick trong 1 giây trong kernel ( mỗi tick cách nhau 1ms).

#### **2. Bài tập:**

1. Viết chương trình đếm lên từ 5 đến 55 và về lại 5 hiển thị trên modul 8 Led 7 đoạn, có xóa số 0 vô nghĩa. Viết 2 Driver và một chương trình User Application.
2. Viết chương trình đếm lên xuống nằm trong khoảng 0 đến 255 hiển thị trên modul 8 Led 7 đoạn, xóa số 0 vô nghĩa. Sử dụng hai nút nhấn, một nút nhấn dùng để đếm lên, một nút nhấn dùng để đếm xuống. Viết 2 Driver và một chương trình User Application.

**BÀI 6****GIAO TIẾP ĐIỀU KHIỂN  
LED MA TRẬN 8x8****I. Phác thảo dự án:**

Hiện nay, ma trận Led là linh kiện điện tử được sử dụng phổ biến trong công nghiệp quảng cáo. Không giống như Led 7 đoạn chỉ có thể hiển thị những con số, ma trận Led có thể hiển thị được tất cả các ký tự, hình ảnh mà người dùng mong muốn. Ngoài ra ma trận Led còn hiển thị được nhiều hiệu ứng đa dạng khác nhau như: dịch trái, dịch phải, chuyển động của sự vật....

Ngoài thực tế, ma trận Led có nhiều loại khác nhau với kích thước khác nhau, số lượng Led khác nhau, số lượng màu Led khác nhau. Thông thường một ma trận Led có ít nhất 8x8 Led đơn trên nó, vì số lượng Led trên một ma trận Led rất lớn nên chúng ta không thể dùng phương pháp truy xuất trực tiếp từng Led được. Nên để hiển thị được các ký tự hình ảnh trên ma trận Led chúng ta phải dùng phương pháp quét.

Kỹ thuật quét ma trận Led chúng ta đã được học trong môn Thực Tập Vi Xử Lý nên trong quyển sách này chúng tôi sẽ không trình bày lại. Chúng ta chỉ tập trung vào việc làm thế nào để quét và điều khiển ma trận Led bằng hệ thống nhúng.

**a. Yêu cầu dự án:**

Trong dự án này chúng ta sẽ điều khiển ma trận Led 8X8 hai màu xanh đỏ hiển thị tất cả các ký tự có trên bàn phím máy vi tính. Vì ma trận Led chỉ có 8 hàng và 8 cột nên mỗi lần chúng ta chỉ hiển thị được một ký tự. Cú pháp lệnh khi thực thi như sau:

<tên chương trình> <ký tự muốn hiển thị>

Trong đó:

<tên chương trình> là tên chương trình sau khi đã biên dịch xong.

<ký tự muốn hiển thị> là ký tự muốn hiển thị trên ma trận Led.

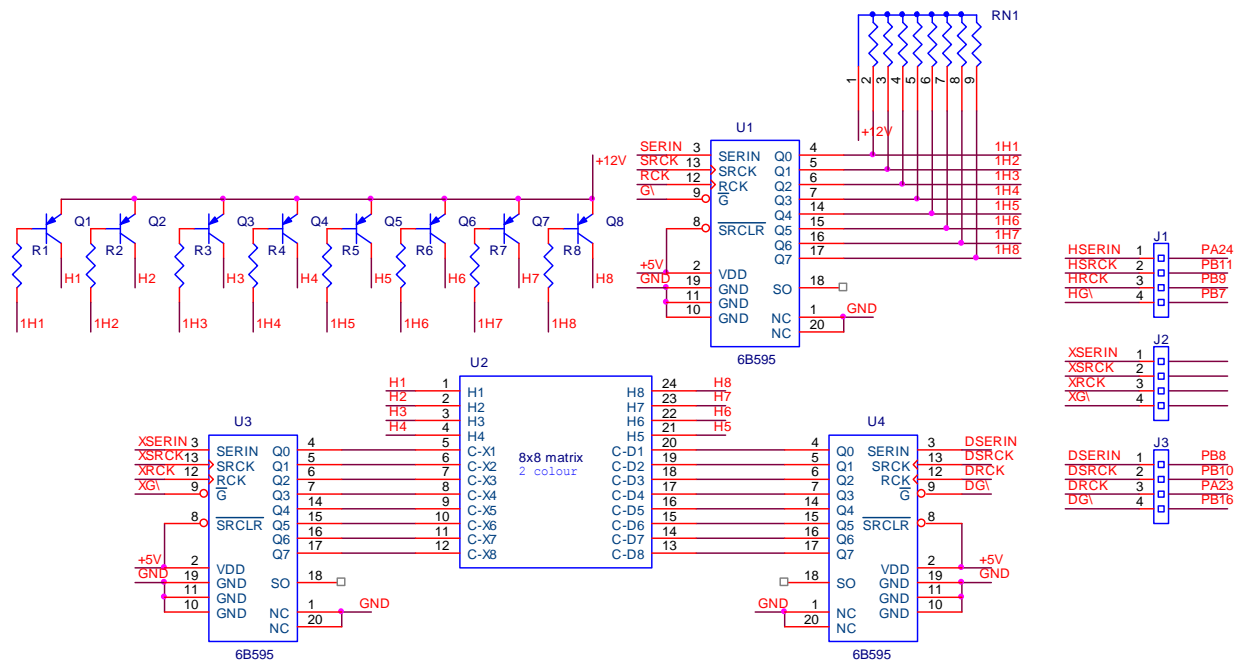
**b. Phân công nhiệm vụ:**

- **Driver:** Nhận dữ liệu là mã Ascii của ký tự muốn hiển thị từ User Application thông qua hàm `write()`, chuyển đổi mã Ascii này sang mã ma trận (một ký tự mã Ascii tương ứng với 5 byte của mã ma trận) và quét modul ma trận Led theo các mã ma trận này.

- **User Application:** Nhận tham số từ người dùng và chuyển tham số này sang cho Driver.

## II. Thực hiện:

### 1. Kết nối phần cứng theo sơ đồ sau:



### 2. Viết chương trình:

- **Driver:** Tên Matrix\_Led\_Display\_dev.c

*/\* khai báo các thư viện cần thiết trong chương trình \*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <mach/at91_tc.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>
#include <linux/clock.h>
#include <linux/irq.h>
#include <linux/time.h>
#include <linux/jiffies.h>
```

```
#include <linux/sched.h>
#include <linux/delay.h>

/* khai báo tên Driver và tên thiết bị */

#define DRVNAME      "matrix_led_dev"
#define DEVNAME      "matrix_led"

/* khai báo các chân dùng để điều khiển modum ma trận Led */

#define DATA_C      AT91_PIN_PB8
#define CLOCK_C      AT91_PIN_PB10
#define LATCH_C      AT91_PIN_PA23
#define OE_C         AT91_PIN_PB16
#define DATA_R      AT91_PIN_PA24
#define CLOCK_R      AT91_PIN_PB11
#define LATCH_R      AT91_PIN_PB9
#define OE_R         AT91_PIN_PB7

/* các lệnh tạo mức High và Low cho các chân điều khiển modun Led */

#define SET_DATA_C()      gpio_set_value(DATA_C,1)
#define SET_CLOCK_C()     gpio_set_value(CLOCK_C,1)
#define SET_LATCH_C()     gpio_set_value(LATCH_C,1)
#define SET_OE_C()        gpio_set_value(OE_C,1)
#define SET_DATA_R()      gpio_set_value(DATA_R,1)
#define SET_CLOCK_R()     gpio_set_value(CLOCK_R,1)
#define SET_LATCH_R()     gpio_set_value(LATCH_R,1)
#define SET_OE_R()        gpio_set_value(OE_R,1)

#define CLEAR_DATA_C()    gpio_set_value(DATA_C,0)
#define CLEAR_CLOCK_C()   gpio_set_value(CLOCK_C,0)
#define CLEAR_LATCH_C()   gpio_set_value(LATCH_C,0)
#define CLEAR_OE_C()      gpio_set_value(OE_C,0)
#define CLEAR_DATA_R()    gpio_set_value(DATA_R,0)
#define CLEAR_CLOCK_R()   gpio_set_value(CLOCK_R,0)
#define CLEAR_LATCH_R()   gpio_set_value(LATCH_R,0)
#define CLEAR_OE_R()      gpio_set_value(OE_R,0)

/* khai báo các biến cần dùng trong chương trình */

static atomic_t matrix_led_open_cnt = ATOMIC_INIT(1);
int i=0;
```

```
/* DataDisplay là mảng chứa dữ liệu hiển thị ra ma trận Led (dữ liệu dịch ra  
IC 6B595 điều khiển hàng), mảng có 8 phần tử ứng với dữ liệu hàng của 8 cột  
ma trận Led */  
unsigned char DataDisplay[8];  
unsigned char MatrixCode[5];  
  
/*mảng ColumnCode tạo dữ liệu một bit dịch chuyển trên 8 cột của ma trận Led  
*/  
unsigned char  
ColumnCode[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};  
  
/* bảng mã font Asscii sang ma trận Led*/  
unsigned char font[]={  
0xFF,0xFF,0xFF,0xFF,0xFF, //SPACE 0  
0xFF,0xFF,0xA0,0xFF,0xFF, //! 1  
0xFF,0xFF,0xF8,0xF4,0xFF, //' 2  
0xEB,0x80,0xEB,0x80,0xEB, //# 3  
0xDB,0xD5,0x80,0xD5,0xED, //$ 4  
0xD8,0xEA,0x94,0xAB,0x8D, //% 5  
0xC9,0xB6,0xA9,0xDF,0xAF, //& 6  
0xFF,0xFF,0xF8,0xF4,0xFF, //' 7  
0xFF,0xE3,0xDD,0xBE,0xFF, //( 8  
0xFF,0xBE,0xDD,0xE3,0xFF, //) 9  
0xD5,0xE3,0x80,0xE3,0xD5, //* 10  
0xF7,0xF7,0xC1,0xF7,0xF7, //+ 11  
0xFF,0xA7,0xC7,0xFF,0xFF, //, 12  
0xF7,0xF7,0xF7,0xF7,0xF7, //- 13  
0xFF,0x9F,0x9F,0xFF,0xFF, //x 14  
0xFF,0xC9,0xC9,0xFF,0xFF, /// 15  
0xC1,0xAE,0xB6,0xBA,0xC1, //0 16  
0xFF,0xBD,0x80,0xBF,0xFF, //1 17  
0x8D,0xB6,0xB6,0xB6,0xB9, //2 18  
0xDD,0xBE,0xB6,0xB6,0xC9, //3 19  
0xE7,0xEB,0xED,0x80,0xEF, //4 20  
0xD8,0xBA,0xBA,0xBA,0xC6, //5 21  
0xC3,0xB5,0xB6,0xB6,0xCF, //6 22  
0xFE,0x8E,0xF6,0xFA,0xFC, //7 23  
0xC9,0xB6,0xB6,0xB6,0xC9, //8 24
```

0xF9, 0xB6, 0xB6, 0xD6, 0xE1,	//9	25
0xFF, 0xC9, 0xC9, 0xFF, 0xFF,	//:	26
0xFF, 0xA4, 0xC4, 0xFF, 0xFF,	////	27
0xF7, 0xEB, 0xDD, 0xBE, 0xFF,	//<	28
0xEB, 0xEB, 0xEB, 0xEB, 0xEB,	//=	29
0xFF, 0xBE, 0xDD, 0xEB, 0xF7,	//>	30
0xFD, 0xFE, 0xAE, 0xF6, 0xF9,	//?	31
0xCD, 0xB6, 0x8E, 0xBE, 0xC1,	//@	32
0x83, 0xF5, 0xF6, 0xF5, 0x83,	//A	33
0xBE, 0x80, 0xB6, 0xB6, 0xC9,	//B	34
0xC1, 0xBE, 0xBE, 0xBE, 0xDD,	//C	35
0xBE, 0x80, 0xBE, 0xBE, 0xC1,	//D	36
0x80, 0xB6, 0xB6, 0xB6, 0xBE,	//E	37
0x80, 0xF6, 0xF6, 0xFE, 0xFE,	//F	38
0xC1, 0xBE, 0xB6, 0xB6, 0xC5,	//G	39
0x80, 0xF7, 0xF7, 0xF7, 0x80,	//H	40
0xFF, 0xBE, 0x80, 0xBE, 0xFF,	//I	41
0xDF, 0xBF, 0xBE, 0xC0, 0xFE,	//J	42
0x80, 0xF7, 0xEB, 0xDD, 0xBE,	//K	43
0x80, 0xBF, 0xBF, 0xBF, 0xFF,	//L	44
0x80, 0xFD, 0xF3, 0xFD, 0x80,	//M	45
0x80, 0xFD, 0xFB, 0xF7, 0x80,	//N	46
0xC1, 0xBE, 0xBE, 0xBE, 0xC1,	//O	47
0x80, 0xF6, 0xF6, 0xF6, 0xF9,	//P	48
0xC1, 0xBE, 0xAE, 0xDE, 0xA1,	//Q	49
0x80, 0xF6, 0xE6, 0xD6, 0xB9,	//R	50
0xD9, 0xB6, 0xB6, 0xB6, 0xCD,	//S	51
0xFE, 0xFE, 0x80, 0xFE, 0xFE,	//T	52
0xC0, 0xBF, 0xBF, 0xBF, 0xC0,	//U	53
0xE0, 0xDF, 0xBF, 0xDF, 0xE0,	//V	54
0xC0, 0xBF, 0xCF, 0xBF, 0xC0,	//W	55
0x9C, 0xEB, 0xF7, 0xEB, 0x9C,	//X	56
0xFC, 0xFB, 0x87, 0xFB, 0xFC,	//Y	57
0x9E, 0xAE, 0xB6, 0xBA, 0xBC,	//Z	58
0xFF, 0x80, 0xBE, 0xBE, 0xFF,	//[	59
0xFD, 0xFB, 0xF7, 0xEF, 0xDF,	//\	60
0xFF, 0xBE, 0xBE, 0x80, 0xFF,	//]	61

```
0xFB, 0xFD, 0xFE, 0xFD, 0xFB,      //^      62
0x7F, 0x7F, 0x7F, 0x7F, 0x7F,      //_      63
0xFF, 0xFF, 0xF8, 0xF4, 0xFF,      //'      64
0xDF, 0xAB, 0xAB, 0xAB, 0xC7,      //a      65
0x80, 0xC7, 0xBB, 0xBB, 0xC7,      //b      66
0xFF, 0xC7, 0xBB, 0xBB, 0xBB,      //c      67
0xC7, 0xBB, 0xBB, 0xC7, 0x80,      //d      68
0xC7, 0xAB, 0xAB, 0xAB, 0xF7,      //e      69
0xF7, 0x81, 0xF6, 0xF6, 0xFD,      //f      70
0xF7, 0xAB, 0xAB, 0xAB, 0xC3,      //g      71
0x80, 0xF7, 0xFB, 0xFB, 0x87,      //h      72
0xFF, 0xBB, 0x82, 0xBF, 0xFF,      //I      73
0xDF, 0xBF, 0xBB, 0xC2, 0xFF,      //j      74
0xFF, 0x80, 0xEF, 0xD7, 0xBB,      //k      75
0xFF, 0xBE, 0x80, 0xBF, 0xFF,      //l      76
0x83, 0xFB, 0x87, 0xFB, 0x87,      //m      77
0x83, 0xF7, 0xFB, 0xFB, 0x87,      //n      78
0xC7, 0xBB, 0xBB, 0xBB, 0xC7,      //o      79
0x83, 0xEB, 0xEB, 0xEB, 0xF7,      //p      80
0xF7, 0xEB, 0xEB, 0xEB, 0x83,      //q      81
0x83, 0xF7, 0xFB, 0xFB, 0xF7,      //r      82
0xB7, 0xAB, 0xAB, 0xAB, 0xDB,      //s      83
0xFF, 0xFB, 0xC0, 0xBB, 0xBB,      //t      84
0xC3, 0xBF, 0xBF, 0xDF, 0x83,      //u      85
0xE3, 0xDF, 0xBF, 0xDF, 0xE3,      //v      86
0xC3, 0xBF, 0xCF, 0xBF, 0xC3,      //w      87
0xBB, 0xD7, 0xEF, 0xD7, 0xBB,      //x      88
0xF3, 0xAF, 0xAF, 0xAF, 0xC3,      //y      89
0xBB, 0x9B, 0xAB, 0xB3, 0xBB,      //z      90
0xFB, 0xE1, 0xE0, 0xE1, 0xFB,      //^      91
0xE3, 0xE3, 0xC1, 0xE3, 0xF7,      //->     92
0xF7, 0xE3, 0xC1, 0xE3, 0xE3,      //<-    93
0xEF, 0xC3, 0x83, 0xC3, 0xEF,      //      94
0xFF, 0xFF, 0xFF, 0xFF, 0xFF //BLANK CHAR 95
};

void __iomem *at91tc0_base;
```

```
struct clk *at91tc0_clk;

/*chương trình viết dữ liệu hàng ra các port của IC 6B595 điều khiển hàng */
void row_write_data(unsigned char data)
{
    int j;
    for (j=0;j<8;j++)
    {
        /*set hoặc clear chân DATA_R tùy thuộc vào giá trị của data */
        (data&(1<<j)) ? SET_DATA_R() : CLEAR_DATA_R();

        /* tạo xung ở chân CLOCK_R để dịch dữ liệu ra IC 6B595 */
        SET_CLOCK_R();
        CLEAR_CLOCK_R();
    }

    /* tạo xung ở chân LATCH_R để chốt dữ liệu trong IC 6B595 */
    SET_LATCH_R();
    CLEAR_LATCH_R();
}

/*chương trình viết dữ liệu cột ra các port của IC 6B595 điều khiển cột */
void column_write_data(unsigned char data)
{
    int j;
    for (j=0;j<8;j++)
    {
        (data&(1<<j)) ? SET_DATA_C() : CLEAR_DATA_C();
        SET_CLOCK_C();
        CLEAR_CLOCK_C();
    }

    SET_LATCH_C();
    CLEAR_LATCH_C();
}

/* chương trình phục vụ ngắt, cứ 1ms chương trình phục vụ ngắt này sẽ được
thực hiện một lần */
static irqreturn_t at91tc0_isr(int irq, void *dev_id) {
    int status;

    // Read TC0 status register to reset RC compare status.
```



```
status = ioread32(at91tc0_base + AT91_TC_SR);  
/*set chân OE_C và OE_R để cách ly dữ liệu bên trong 6B595 và dữ liệu đã  
xuất ra các chân của 6B595 */  
SET_OE_C();  
SET_OE_R();  
/* viết dữ liệu hàng và cột tương ứng ra modul ma trận Led */  
row_write_data(DataDisplay[i]);  
column_write_data(ColumnCode[i]);  
/*clear chân OE_C và OE_R để xuất dữ liệu bên trong 6B595 ra các chân dữ  
liệu của 6B595 */  
CLEAR_OE_C();  
CLEAR_OE_R();  
/* biến i = 0 ứng với vị trí cột đầu tiên của ma trận Led, i = 1 ứng với vị trí cột  
thứ 2 của ma trận Led, tương tự i = 7 ứng với vị trí cột cuối cùng của ma trận  
Led */  
i++;  
/* khi vị trí quét đến cột thứ 8 của ma trận Led thì ta quay lại vị trí thứ nhất */  
if (i==8) {  
    i = 0;  
}  
return IRQ_HANDLED;  
}  
/* chương trình chuyển mã Asscii sang mã ma trận Led */  
void ascii_to_matrix_led (unsigned char data)  
{  
    MatrixCode[0] = font[data * 5 +0];  
    MatrixCode[1] = font[data * 5 +1];  
    MatrixCode[2] = font[data * 5 +2];  
    MatrixCode[3] = font[data * 5 +3];  
    MatrixCode[4] = font[data * 5 +4];  
}  
/* chương trình nhận dữ liệu từ User Application */  
static ssize_t matrix_led_write (struct file *filp, unsigned  
char __iomem buf[], size_t bufsize, loff_t *f_pos)  
{
```

```
    unsigned char write_buf[1];
    int write_size = 0;
    /*nhận dữ liệu từ user và chép vào write_buf*/
    if (copy_from_user (write_buf, buf, 1) != 0)
    {
        return -EFAULT;
    }
    else
    {
        write_size = bufsize;
        /*trong bảng mã Asscii ký tự khoảng trắng có giá trị là 32, nhưng trong bảng
        font ma trận Led ký tự khoảng trắng lại là 5 byte đầu tiên trong mảng nên khi
        chuyển từ mã Asscii sang mã ma trận Led chúng ta phải dùng (mã Asscii - 32)
        */

        ascii_to_matrix_led(write_buf[0]-32);

        printk("ascii to matrix led %d complete\n",
write_buf[0]);
        /* chép mã ma trận của ký tự muốn hiển thị vào mảng hiển thị*/
        DataDisplay[0] = ~0xFF;
        DataDisplay[1] = ~MatrixCode[0];
        DataDisplay[2] = ~MatrixCode[1];
        DataDisplay[3] = ~MatrixCode[2];
        DataDisplay[4] = ~MatrixCode[3];
        DataDisplay[5] = ~MatrixCode[4];
        DataDisplay[6] = ~0xFF;
        DataDisplay[7] = ~0xFF;

    }
    return write_size;
}

static int
matrix_led_open(struct inode *inode, struct file *file)
{
    int result = 0;
```

```
        unsigned int dev_minor = MINOR(inode->i_rdev);
        if (!atomic_dec_and_test(&matrix_led_open_cnt)) {
            atomic_inc(&matrix_led_open_cnt);
            printk(KERN_ERR DRVNAME ": Device with minor ID %d
            already in use\n", dev_minor);
            result = -EBUSY;
            goto out;
        }
    out:
        return result;
    }

    static int
    matrix_led_close(struct inode * inode, struct file * file)
    {
        smp_mb__before_atomic_inc();
        atomic_inc(&matrix_led_open_cnt);

        return 0;
    }

    struct file_operations matrix_led_fops = {
        .write      = matrix_led_write,
        .open       = matrix_led_open,
        .release    = matrix_led_close,
    };

    static struct miscdevice matrix_led_dev = {
        .minor      = MISC_DYNAMIC_MINOR,
        .name       = "matrix_led",
        .fops       = &matrix_led_fops,
    };

    static int __init
    matrix_led_mod_init(void)
    {
        int ret=0;
```

```
gpio_request (DATA_C, NULL);
gpio_request (CLOCK_C, NULL);
gpio_request (LATCH_C, NULL);
gpio_request (OE_C, NULL);
gpio_request (DATA_R, NULL);
gpio_request (CLOCK_R, NULL);
gpio_request (LATCH_R, NULL);
gpio_request (OE_R, NULL);

at91_set_GPIO_periph (DATA_C, 1);
at91_set_GPIO_periph (CLOCK_C, 1);
at91_set_GPIO_periph (LATCH_C, 1);
at91_set_GPIO_periph (OE_C, 1);
at91_set_GPIO_periph (DATA_R, 1);
at91_set_GPIO_periph (CLOCK_R, 1);
at91_set_GPIO_periph (LATCH_R, 1);
at91_set_GPIO_periph (OE_R, 1);

gpio_direction_output(DATA_C, 0);
gpio_direction_output(CLOCK_C, 0);
gpio_direction_output(LATCH_C, 0);
gpio_direction_output(OE_C, 0);
gpio_direction_output(DATA_R, 0);
gpio_direction_output(CLOCK_R, 0);
gpio_direction_output(LATCH_R, 0);
gpio_direction_output(OE_R, 0);

at91tc0_clk = clk_get(NULL, // Device pointer - not required.
                    "tc0_clk"); // Clock name.
clk_enable(at91tc0_clk);
at91tc0_base = ioremap_nocache(AT91SAM9260_BASE_TC0,
64);
if (at91tc0_base == NULL)
{
    printk(KERN_INFO "at91adc: TC0 memory mapping
failed\n");
}
```

```
        ret = -EACCES;
        goto exit_5;
    }

    /* Configure TC0 in waveform mode, TIMER_CLK1 and to generate
    interrupt on RC compare.

    Load 50000 to RC so that with TIMER_CLK1 = MCK/2 = 50MHz, the
    interrupt will be

    generated every 1/50MHz * 50000 = 20nS * 50000 = 1 milli second.

    NOTE: Even though AT91_TC_RC is a 32-bit register, only 16-bits are
    programmable.

    */

    iowrite32(50000, (at91tc0_base + AT91_TC_RC));
    iowrite32((AT91_TC_WAVE | AT91_TC_WAVESSEL_UP_AUTO),
    (at91tc0_base + AT91_TC_CMR));
    iowrite32(AT91_TC_CPCS, (at91tc0_base + AT91_TC_IER));
    iowrite32((AT91_TC_SWTRG | AT91_TC_CLKEN), (at91tc0_base +
    AT91_TC_CCR));

    /*Install interrupt for TC0*/

    ret = request_irq(AT91SAM9260_ID_TC0, // Interrupt number
    at91tc0_isr, // Pointer to the interrupt sub-routine
    0, // Flags - fast, shared or contributing to entropy pool
    "matrix_led_irq", // Device name to show as owner in /proc/interrupts
    NULL); // Private data for shared interrupts

    if (ret != 0)
    {
        printk(KERN_INFO "matrix_led_irq: Timer interrupt
        request failed\n");
        ret = -EBUSY;
        goto exit_6;
    }

    misc_register(&matrix_led_dev);
    printk(KERN_INFO "matrix_led: Loaded module\n");
    return ret;
```

```
exit_6:
iounmap(at91tc0_base);
exit_5:
clk_disable(at91tc0_clk);
return ret;
}

static void __exit
matrix_led_mod_exit(void)
{
    iounmap(at91tc0_base);
    clk_disable(at91tc0_clk);
    // Free TC0 IRQ.

    free_irq(AT91SAM9260_ID_TC0, //Interrupt number
    NULL); // Private data for shared interrupts
    misc_deregister(&matrix_led_dev);
    printk(KERN_INFO "matrix_led: Unloaded module\n");
}

module_init (matrix_led_mod_init);
module_exit (matrix_led_mod_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("TranCamNhan");
MODULE_DESCRIPTION("Character device for for generic gpio
api");
```

- **User Application:** Tên Matrix\_Led\_Display\_app.c

*/\* khai báo các thư viện cần dùng trong chương trình \*/*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>
```

```
#include <pthread.h>

/* khai báo các biến cần dùng trong chương trình */
int matrix_led_fd;
unsigned long int ioctl_buf[3]={0,0,0};
unsigned char write_buf[1];

/* chương trình in ra màn hình cú pháp lệnh để hiển thị chữ ra ma trận Led */
void print_usage() {
    printf("matrixled_app <letter>\n");
    exit(0);
}

/* chương trình chính */
int main(int argc, unsigned char **argv)
{
    int res;

    /* mở tập tin thiết bị trước khi thao tác */
    if ((matrix_led_fd = open("/dev/matrix_led", O_RDWR)) < 0)
    {
        /* nếu quá trình mở tập tin thất bại thì in ra màn hình báo lỗi cho người
        dùng biết và trả về mã lỗi */
        printf("Error whilst opening /dev/matrix_led
        device\n");
        return -1;
    }

    if (argc != 2)
    {
        /* khi người dùng sai cú pháp lệnh thì in ra hướng dẫn cú pháp lệnh */
        print_usage();
    }
    else
    /* nếu người dùng đúng cú pháp lệnh thì tiến hành chép dữ liệu vào
    write_buf[0] và truyền qua cho driver bằng hàm write() */
    write_buf[0] = argv[1][0];
    write(matrix_led_fd, write_buf, 1);
    printf ("User: sent %s to driver\n",argv[1][0]);
}
```

```
return 0;  
}
```

### **3. Biên dịch và thực thi chương trình:**

Biên dịch chương trình, nạp vào Kit KM9260 và chạy chương trình.

### **III. Mở rộng dự án:**

Vì modun ma trận Led chỉ có 8 hàng và 8 cột nên dự án này chỉ có thể xuất mỗi lần 1 ký tự. Để xuất được nhiều ký tự hơn, cũng như có thể hiển thị được một chuỗi ký tự, một câu thì chúng ta phải dùng thêm hiệu ứng dịch chữ. Chúng ta có thể dịch chữ từ trái sang phải hoặc từ phải sang trái.

Chương trình sau đây có thể hiển thị một chuỗi ký tự bất kỳ nhập từ bàn phím với ký tự khoảng trắng là dấu “\_” (vì cấu trúc lệnh của chương trình main trong User Application, nếu dùng khoảng trắng thì chương trình main sẽ hiểu là các tham số argv khác nhau, khuyết điểm này sẽ được khắc phục ở cuối dự án này), có thể sử dụng hiệu ứng dịch trái, dịch phải, đứng yên và có thể thay đổi tốc độ dịch chữ.

Cú pháp lệnh khi sử dụng như sau:

<tên chương trình> <tên lệnh> <tham số>

Trong đó:

<tên chương trình> là tên chương trình sau khi biên dịch

<tên lệnh> là các lệnh dùng để điều khiển ma trận Led. Có các lệnh như:

`update_content`: thay đổi nội dung muốn hiển thị, lệnh này có tham số là chuỗi ký tự muốn hiển thị.

`update_speed`: thay đổi tốc độ dịch chữ, lệnh này có tham số là tốc độ dịch chữ thay đổi từ 1 đến 100. Ứng với giá trị 1 thì tốc độ dịch chữ là chậm nhất, 100 là tốc độ dịch chữ là nhanh nhất

`shift_left`: hiệu ứng dịch trái, lệnh này không có tham số.

`shift_right`: hiệu ứng dịch phải, lệnh này không có tham số.

`pause`: hiệu ứng đứng yên hình ảnh đang được hiển thị, lệnh này không có tham số.

<tham số> là tham số của lệnh.

#### **❖ Chương trình:**



- **Driver:** Tên Matrix\_Led\_Shift\_Display\_dev.c

```
/* khai báo các thư viện cần thiết cho chương trình */
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <mach/at91_tc.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>
#include <linux/clock.h>
#include <linux/irq.h>
#include <linux/time.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/delay.h>

/* khai báo tên thiết bị và tên driver */
#define DRVNAME      "matrix_led_dev"
#define DEVNAME      "matrix_led"

/* khai báo các chân của vi xử lý dùng trong dự án này */
#define DATA_C      AT91_PIN_PB8
#define CLOCK_C      AT91_PIN_PB10
#define LATCH_C      AT91_PIN_PA23
#define OE_C         AT91_PIN_PB16
#define DATA_R      AT91_PIN_PA24
#define CLOCK_R      AT91_PIN_PB11
#define LATCH_R      AT91_PIN_PB9
#define OE_R         AT91_PIN_PB7

/* lệnh set và clear các chân đã khai báo phần trên */
#define SET_DATA_C()  gpio_set_value(DATA_C,1)
#define SET_CLOCK_C() gpio_set_value(CLOCK_C,1)
#define SET_LATCH_C() gpio_set_value(LATCH_C,1)
#define SET_OE_C()    gpio_set_value(OE_C,1)
#define SET_DATA_R()  gpio_set_value(DATA_R,1)
```

```
#define SET_CLOCK_R()      gpio_set_value(CLOCK_R,1)
#define SET_LATCH_R()      gpio_set_value(LATCH_R,1)
#define SET_OE_R()         gpio_set_value(OE_R,1)
#define CLEAR_DATA_C()     gpio_set_value(DATA_C,0)
#define CLEAR_CLOCK_C()    gpio_set_value(CLOCK_C,0)
#define CLEAR_LATCH_C()    gpio_set_value(LATCH_C,0)
#define CLEAR_OE_C()       gpio_set_value(OE_C,0)
#define CLEAR_DATA_R()     gpio_set_value(DATA_R,0)
#define CLEAR_CLOCK_R()    gpio_set_value(CLOCK_R,0)
#define CLEAR_LATCH_R()    gpio_set_value(LATCH_R,0)
#define CLEAR_OE_R()       gpio_set_value(OE_R,0)

/*định nghĩa các lệnh ioctl dùng trong chương trình */
#define MATRIX_LED_DEV_MAGIC  'B'
#define SHIFT_LEFT  _IOWR(MATRIX_LED_DEV_MAGIC, 1,unsigned long)
#define SHIFT_RIGHT _IOWR(MATRIX_LED_DEV_MAGIC, 2,unsigned long)
#define UPDATE_SPEED _IOWR(MATRIX_LED_DEV_MAGIC, 3,unsigned long)
#define PAUSE       _IOWR(MATRIX_LED_DEV_MAGIC, 4,unsigned long)

/* Khai báo các biến cần thiết trong chương trình */
static atomic_t matrix_led_open_cnt = ATOMIC_INIT(1);
int letter,i=0,cycle=0,slip=0,shift=0,shift_speed = 50,pause=0;
unsigned char DataDisplay[100];
unsigned char MatrixCode[5];
unsigned char ColumnCode[] ={0x80, 0x40, 0x20, 0x10, 0x08,
0x04, 0x02, 0x01};
unsigned char font[] = {
0xFF,0xFF,0xFF,0xFF,0xFF,//SPACE      0
0xFF,0xFF,0xA0,0xFF,0xFF,//!          1
0xFF,0xFF,0xF8,0xF4,0xFF,//'          2
0xEB,0x80,0xEB,0x80,0xEB,//#          3
0xDB,0xD5,0x80,0xD5,0xED,//$          4
0xD8,0xEA,0x94,0xAB,0x8D,//%          5
0xC9,0xB6,0xA9,0xDF,0xAF,//&          6
0xFF,0xFF,0xF8,0xF4,0xFF,//'          7
0xFF,0xE3,0xDD,0xBE,0xFF,//(          8
0xFF,0xBE,0xDD,0xE3,0xFF,//)          9
0xD5,0xE3,0x80,0xE3,0xD5,//*         10
0xF7,0xF7,0xC1,0xF7,0xF7,//+         11
```

0xFF, 0xA7, 0xC7, 0xFF, 0xFF, //	12
0xF7, 0xF7, 0xF7, 0xF7, 0xF7, //-	13
0xFF, 0x9F, 0x9F, 0xFF, 0xFF, //x	14
0xFF, 0xC9, 0xC9, 0xFF, 0xFF, ///	15
0xC1, 0xAE, 0xB6, 0xBA, 0xC1, //0	16
0xFF, 0xBD, 0x80, 0xBF, 0xFF, //1	17
0x8D, 0xB6, 0xB6, 0xB6, 0xB9, //2	18
0xDD, 0xBE, 0xB6, 0xB6, 0xC9, //3	19
0xE7, 0xEB, 0xED, 0x80, 0xEF, //4	20
0xD8, 0xBA, 0xBA, 0xBA, 0xC6, //5	21
0xC3, 0xB5, 0xB6, 0xB6, 0xCF, //6	22
0xFE, 0x8E, 0xF6, 0xFA, 0xFC, //7	23
0xC9, 0xB6, 0xB6, 0xB6, 0xC9, //8	24
0xF9, 0xB6, 0xB6, 0xD6, 0xE1, //9	25
0xFF, 0xC9, 0xC9, 0xFF, 0xFF, //:	26
0xFF, 0xA4, 0xC4, 0xFF, 0xFF, ///	27
0xF7, 0xEB, 0xDD, 0xBE, 0xFF, //<	28
0xEB, 0xEB, 0xEB, 0xEB, 0xEB, // =	29
0xFF, 0xBE, 0xDD, 0xEB, 0xF7, //>	30
0xFD, 0xFE, 0xAE, 0xF6, 0xF9, //?	31
0xCD, 0xB6, 0x8E, 0xBE, 0xC1, //@	32
0x83, 0xF5, 0xF6, 0xF5, 0x83, //A	33
0xBE, 0x80, 0xB6, 0xB6, 0xC9, //B	34
0xC1, 0xBE, 0xBE, 0xBE, 0xDD, //C	35
0xBE, 0x80, 0xBE, 0xBE, 0xC1, //D	36
0x80, 0xB6, 0xB6, 0xB6, 0xBE, //E	37
0x80, 0xF6, 0xF6, 0xFE, 0xFE, //F	38
0xC1, 0xBE, 0xB6, 0xB6, 0xC5, //G	39
0x80, 0xF7, 0xF7, 0xF7, 0x80, //H	40
0xFF, 0xBE, 0x80, 0xBE, 0xFF, //I	41
0xDF, 0xBF, 0xBE, 0xC0, 0xFE, //J	42
0x80, 0xF7, 0xEB, 0xDD, 0xBE, //K	43
0x80, 0xBF, 0xBF, 0xBF, 0xFF, //L	44
0x80, 0xFD, 0xF3, 0xFD, 0x80, //M	45
0x80, 0xFD, 0xFB, 0xF7, 0x80, //N	46
0xC1, 0xBE, 0xBE, 0xBE, 0xC1, //O	47
0x80, 0xF6, 0xF6, 0xF6, 0xF9, //P	48

0xC1, 0xBE, 0xAE, 0xDE, 0xA1, //Q	49
0x80, 0xF6, 0xE6, 0xD6, 0xB9, //R	50
0xD9, 0xB6, 0xB6, 0xB6, 0xCD, //S	51
0xFE, 0xFE, 0x80, 0xFE, 0xFE, //T	52
0xC0, 0xBF, 0xBF, 0xBF, 0xC0, //U	53
0xE0, 0xDF, 0xBF, 0xDF, 0xE0, //V	54
0xC0, 0xBF, 0xCF, 0xBF, 0xC0, //W	55
0x9C, 0xEB, 0xF7, 0xEB, 0x9C, //X	56
0xFC, 0xFB, 0x87, 0xFB, 0xFC, //Y	57
0x9E, 0xAE, 0xB6, 0xBA, 0xBC, //Z	58
0xFF, 0x80, 0xBE, 0xBE, 0xFF, // [	59
0xFD, 0xFB, 0xF7, 0xEF, 0xDF, // \	60
0xFF, 0xBE, 0xBE, 0x80, 0xFF, // ]	61
0xFB, 0xFD, 0xFE, 0xFD, 0xFB, // ^	62
0x7F, 0x7F, 0x7F, 0x7F, 0x7F, // _	63
0xFF, 0xFF, 0xF8, 0xF4, 0xFF, // '	64
0xDF, 0xAB, 0xAB, 0xAB, 0xC7, //a	65
0x80, 0xC7, 0xBB, 0xBB, 0xC7, //b	66
0xFF, 0xC7, 0xBB, 0xBB, 0xBB, //c	67
0xC7, 0xBB, 0xBB, 0xC7, 0x80, //d	68
0xC7, 0xAB, 0xAB, 0xAB, 0xF7, //e	69
0xF7, 0x81, 0xF6, 0xF6, 0xFD, //f	70
0xF7, 0xAB, 0xAB, 0xAB, 0xC3, //g	71
0x80, 0xF7, 0xFB, 0xFB, 0x87, //h	72
0xFF, 0xBB, 0x82, 0xBF, 0xFF, //i	73
0xDF, 0xBF, 0xBB, 0xC2, 0xFF, //j	74
0xFF, 0x80, 0xEF, 0xD7, 0xBB, //k	75
0xFF, 0xBE, 0x80, 0xBF, 0xFF, //l	76
0x83, 0xFB, 0x87, 0xFB, 0x87, //m	77
0x83, 0xF7, 0xFB, 0xFB, 0x87, //n	78
0xC7, 0xBB, 0xBB, 0xBB, 0xC7, //o	79
0x83, 0xEB, 0xEB, 0xEB, 0xF7, //p	80
0xF7, 0xEB, 0xEB, 0xEB, 0x83, //q	81
0x83, 0xF7, 0xFB, 0xFB, 0xF7, //r	82
0xB7, 0xAB, 0xAB, 0xAB, 0xDB, //s	83
0xFF, 0xFB, 0xC0, 0xBB, 0xBB, //t	84
0xC3, 0xBF, 0xBF, 0xDF, 0x83, //u	85

```
0xE3,0xDF,0xBF,0xDF,0xE3,>//v      86
0xC3,0xBF,0xCF,0xBF,0xC3,>//w      87
0xBB,0xD7,0xEF,0xD7,0xBB,>//x      88
0xF3,0xAF,0xAF,0xAF,0xC3,>//y      89
0xBB,0x9B,0xAB,0xB3,0xBB,>//z      90
0xFB,0xE1,0xE0,0xE1,0xFB,>//^      91
0xE3,0xE3,0xC1,0xE3,0xF7,>//->    93
0xF7,0xE3,0xC1,0xE3,0xE3,>//<-    93
0xEF,0xC3,0x83,0xC3,0xEF,>//      94
0xFF,0xFF,0xFF,0xFF,0xFF,>//BLANK CHAR 95
};

void __iomem *at91tc0_base;
struct clk *at91tc0_clk;

void row_write_data(unsigned char data) {
    int j;
    for (j=0;j<8;j++){
        (data&(128>>j))? SET_DATA_R():CLEAR_DATA_R();
        SET_CLOCK_R();
        CLEAR_CLOCK_R();
    }
    SET_LATCH_R();
    CLEAR_LATCH_R();
}

void column_write_data(unsigned char data) {
    int j;
    for (j=0;j<8;j++){
        (data&(1<<j))? SET_DATA_C():CLEAR_DATA_C();
        SET_CLOCK_C();
        CLEAR_CLOCK_C();
    }
    SET_LATCH_C();
    CLEAR_LATCH_C();
}

static irqreturn_t at91tc0_isr(int irq, void *dev_id) {
    int status;

    // Read TC0 status register to reset RC compare status.
```

```
status = ioread32(at91tc0_base + AT91_TC_SR);
if (shift!=0) {
if (cycle < shift_speed) {
    SET_OE_C();
    SET_OE_R();
    row_write_data(DataDisplay[slip+i]);
    column_write_data(ColumnCode[i]);
    CLEAR_OE_C();
    CLEAR_OE_R();
    i++;
    if (i==8) {
        i = 0;
        cycle++;
    }
} else {
    cycle = 0;
    if (pause==0) {
        if (shift==1) {
            slip++;
            if (slip == letter*5+8) {
                slip = 0;
            }
        } else {
            slip--;
            if (slip == 0) {
                slip = letter*5+8;
            }
        }
    }
}
return IRQ_HANDLED;
}

void ascii_to_matrix_led (unsigned char data) {
    MatrixCode[0] = font[data * 5 +0];
```

```
    MatrixCode[1] = font[data * 5 +1];
    MatrixCode[2] = font[data * 5 +2];
    MatrixCode[3] = font[data * 5 +3];
    MatrixCode[4] = font[data * 5 +4];
}

static int matrix_led_ioctl (struct inode *inode, struct file
*file, unsigned int cmd, unsigned long arg[]){
    int retval;
    switch (cmd) {
        case SHIFT_LEFT:
            shift = 1;
            pause = 0;
            break;
        case SHIFT_RIGHT:
            shift = 2;
            pause = 0;
            break;
        case UPDATE_SPEED: shift_speed = 101 - arg[0];
            break;
        case PAUSE: pause =1;
            break;
        default:
            printk ("Driver: Don't have this operation \n");
            retval = -EINVAL;
            break;
    }
    return retval;
}

static ssize_t matrix_led_write (struct file *filp, unsigned
char __iomem buf[], size_t bufsize, loff_t *f_pos) {
    unsigned char write_buf[100] ;
    int write_size = 0, letter_byte;
    if (copy_from_user (write_buf, buf, bufsize) != 0) {
        return -EFAULT;
    } else {
```

```
        write_size = bufsize;
        printk("write size: %d\n",write_size);
        DataDisplay[0] = 0x0;
        DataDisplay[1] = 0x0;
        DataDisplay[2] = 0x0;
        DataDisplay[3] = 0x0;
        DataDisplay[4] = 0x0;
        DataDisplay[5] = 0x0;
        DataDisplay[6] = 0x0;
        DataDisplay[7] = 0x0;

        for (letter=0;letter < write_size;letter++){
            if (write_buf[letter] == 95) {
                write_buf[letter] = 32;
            }
            ascii_to_matrix_led(write_buf[letter]-32);
            for(letter_byte=0;letter_byte < 5; letter_byte) {
                DataDisplay[letter*5+8+letter_byte] =
                    ~MatrixCode[letter_byte];
            }
        }
        DataDisplay[letter*5+8+0] = 0x0;
        DataDisplay[letter*5+8+1] = 0x0;
        DataDisplay[letter*5+8+2] = 0x0;
        DataDisplay[letter*5+8+3] = 0x0;
        DataDisplay[letter*5+8+4] = 0x0;
        DataDisplay[letter*5+8+5] = 0x0;
        DataDisplay[letter*5+8+6] = 0x0;
        DataDisplay[letter*5+8+7] = 0x0;
        slip = 0;
        cycle = 0;
        i = 0;
    }
    return write_size;
}

static int
```



```
matrix_led_open(struct inode *inode, struct file *file) {
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&matrix_led_open_cnt)) {
        atomic_inc(&matrix_led_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d
        already in use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

static int
matrix_led_close(struct inode * inode, struct file * file) {
    smp_mb__before_atomic_inc();
    atomic_inc(&matrix_led_open_cnt);
    return 0;
}

struct file_operations matrix_led_fops = {
    .write      = matrix_led_write,
    .ioctl      = matrix_led_ioctl,
    .open       = matrix_led_open,
    .release    = matrix_led_close,
};

static struct miscdevice matrix_led_dev = {
    .minor       = MISC_DYNAMIC_MINOR,
    .name        = "matrix_led",
    .fops        = &matrix_led_fops,
};

static int __init
matrix_led_mod_init(void) {
    int ret=0;
```

```
gpio_request (DATA_C, NULL);
gpio_request (CLOCK_C, NULL);
gpio_request (LATCH_C, NULL);
gpio_request (OE_C, NULL);
gpio_request (DATA_R, NULL);
gpio_request (CLOCK_R, NULL);
gpio_request (LATCH_R, NULL);
gpio_request (OE_R, NULL);

at91_set_GPIO_periph (DATA_C, 1);
at91_set_GPIO_periph (CLOCK_C, 1);
at91_set_GPIO_periph (LATCH_C, 1);
at91_set_GPIO_periph (OE_C, 1);
at91_set_GPIO_periph (DATA_R, 1);
at91_set_GPIO_periph (CLOCK_R, 1);
at91_set_GPIO_periph (LATCH_R, 1);
at91_set_GPIO_periph (OE_R, 1);

gpio_direction_output (DATA_C, 0);
gpio_direction_output (CLOCK_C, 0);
gpio_direction_output (LATCH_C, 0);
gpio_direction_output (OE_C, 0);
gpio_direction_output (DATA_R, 0);
gpio_direction_output (CLOCK_R, 0);
gpio_direction_output (LATCH_R, 0);
gpio_direction_output (OE_R, 0);

at91tc0_clk = clk_get (NULL, //Device pointer - not required.
                        "tc0_clk"); // Clock name.
clk_enable (at91tc0_clk);
at91tc0_base = ioremap_nocache (AT91SAM9260_BASE_TC0,
64);
if (at91tc0_base == NULL) {
    printk (KERN_INFO "TC0 memory mapping failed\n");
    ret = -EACCES;
    goto exit_5;
```

```
    }

    iowrite32(50000, (at91tc0_base + AT91_TC_RC));
    iowrite32((AT91_TC_WAVE | AT91_TC_WAVESEL_UP_AUTO),
    (at91tc0_base + AT91_TC_CMR));
    iowrite32(AT91_TC_CPCS, (at91tc0_base + AT91_TC_IER));
    iowrite32((AT91_TC_SWTRG | AT91_TC_CLKEN), (at91tc0_base +
    AT91_TC_CCR));

    //Install interrupt for TC0.
    ret = request_irq(    AT91SAM9260_ID_TC0,
                        at91tc0_isr, 0,
                        "matrix_led_irq",NULL);

    if (ret != 0) {
        printk(KERN_INFO "matrix_led_irq: Timer interrupt
        request failed\n");
        ret = -EBUSY;
        goto exit_6;
    }

    misc_register(&matrix_led_dev);
    printk(KERN_INFO "matrix_led: Loaded module\n");
    return ret;
exit_6:
    iounmap(at91tc0_base);
exit_5:
    clk_disable(at91tc0_clk);
    return ret;
}

static void __exit
matrix_led_mod_exit(void){
    iounmap(at91tc0_base);
    clk_disable(at91tc0_clk);
    // Free TC0 IRQ.
    free_irq(    AT91SAM9260_ID_TC0, // Interrupt number
            NULL); // Private data for shared interrupts
    misc_deregister(&matrix_led_dev);
}
```

```
        printk(KERN_INFO "matrix_led: Unloaded module\n");
    }

    module_init (matrix_led_mod_init);
    module_exit (matrix_led_mod_exit);

    MODULE_LICENSE("GPL");
    MODULE_AUTHOR("TranCamNhan");
    MODULE_DESCRIPTION("Character device for for generic gpio
    api");
```

- **User Application:** Tên Matrix\_Led\_Shift\_Display\_app.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/ioctl.h>
#include <pthread.h>

#define MATRIX_LED_DEV_MAGIC  'B'
#define SHIFT_LEFT      _IOWR(MATRIX_LED_DEV_MAGIC, 1,unsigned long)
#define SHIFT_RIGHT     _IOWR(MATRIX_LED_DEV_MAGIC, 2,unsigned long)
#define UPDATE_SPEED    _IOWR(MATRIX_LED_DEV_MAGIC, 3,unsigned long)
#define PAUSE           _IOWR(MATRIX_LED_DEV_MAGIC, 4,unsigned long)

int matrix_led_fd;
unsigned long ioctl_buf[3]={0,0,0};
unsigned char write_buf[100];
void
print_usage(){
    printf("matrixled_app update_content <sentence which you
    want to display on MatrixLed, Space = '_'>\n");
    printf("matrixled_app  update_speed  <speed from 1 to
    100>\n");
    printf("matrixled_app  shift_left\n");
    printf("matrixled_app  shift_right\n");
    printf("matrixled_app  pause\n");
```

```
    exit(0);
}

int
main(int argc, unsigned char **argv) {
    int res,i;
    if ((matrix_led_fd = open("/dev/matrix_led", O_RDWR)) < 0) {
        printf("Error whilst opening /dev/matrix_led
        device\n");
        return -1;
    }

    if (argc == 2) {
        if (!strcmp (argv[1], "pause")){
            ioctl(matrix_led_fd, PAUSE, ioctl_buf);
        } else if (!strcmp(argv[1], "shift_left")){
            ioctl(matrix_led_fd, SHIFT_LEFT, ioctl_buf);
        } else if (!strcmp(argv[1], "shift_right")){
            ioctl(matrix_led_fd, SHIFT_RIGHT, ioctl_buf);
        } else print_usage();
    } else if (argc == 3) {
        if (!strcmp(argv[1], "update_speed")){
            ioctl_buf[0] = atoi(argv[2]);
            ioctl(matrix_led_fd, UPDATE_SPEED, ioctl_buf);
        } else if (!strcmp(argv[1], "update_content")){
            for (i=0; i<strlen(argv[2]); i++){
                write_buf[i] = argv[2][i];
            }
            write(matrix_led_fd, write_buf, strlen(argv[2]));
            printf("Update Content Complete: <%s> \n", argv[2]);
        } else print_usage();
    } else print_usage();
    return 0;
}
```

#### **IV. Kết luận và bài tập:**

##### **1. Kết luận:**

Trong dự án này chúng ta đã sử dụng phương pháp quét để hiển thị một ký tự bất kỳ, một chuỗi ký tự bất kỳ trên ma trận Led 8X8. Chương trình của chúng ta có 2 phần là Driver và User Application.

Như đã nói trong phần trên, trong chương trình này khi chúng ta muốn hiển thị khoảng trắng trên ma trận Led thì chúng ta phải nhập dữ liệu là dấu « \_ » nên sẽ gây ra sự khó chịu khi sử dụng đối với người dùng chương trình và chúng ta không thể hiển thị dấu « \_ » trên ma trận Led được. Để khắc phục vấn đề này, thì trong phần lệnh `update_content` thay vì chúng ta nhập dữ liệu vào tham số `argv[2]` của lệnh `main` thì chúng ta nạp dữ liệu này vào trong một biến kiểu chuỗi bằng lệnh `gets()`. Sau đó truyền biến này sang cho Driver, lúc đó chúng ta có thể khắc phục được khuyết điểm của chương trình trên.

Ngoài ra chương trình trên chỉ hiển thị ma trận Led một màu. Vì vậy chúng ta có thể mở rộng ứng dụng bằng cách viết thêm Driver hiển thị hai màu dựa vào Driver trên.

## **2. Bài tập :**

1. Viết chương trình (Driver và User Application) hiển thị dữ liệu trên ma trận Led một màu có các hiệu ứng dịch trái, dịch phải, đứng yên, thay đổi tốc độ dịch và nhập dữ liệu muốn hiển thị bằng lệnh `gets()`.
2. Viết chương trình (Driver và User Application) hiển thị dữ liệu trên ma trận Led hai màu có các hiệu ứng dịch trái, dịch phải, đứng yên, thay đổi tốc độ dịch, chọn được màu chữ hiển thị và nhập dữ liệu muốn hiển thị bằng lệnh `gets()`.

**BÀI 6**

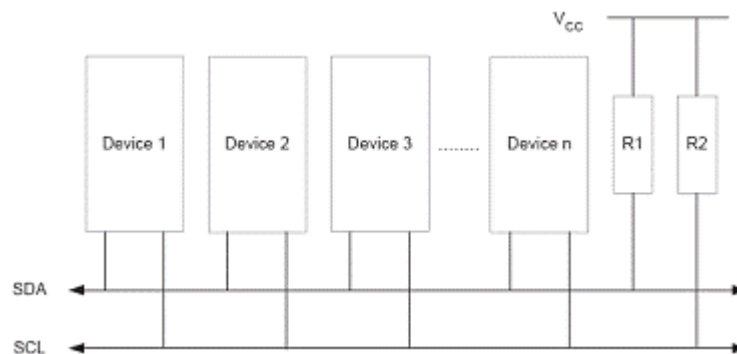
**GIAO TIẾP ĐIỀU KHIỂN  
ADC0809**

**BÀI 7****GIAO TIẾP ĐIỀU KHIỂN  
MODULE I2C****7-1- TỔNG QUAN VỀ I2C:****I. Giới thiệu I<sup>2</sup>C:**

I2C là chuẩn truyền thông nối tiếp do hãng điện tử Philips xây dựng vào năm 1990. I2C có khả năng truyền nối tiếp đa chủ, nghĩa là trong một bus có thể có nhiều hơn một thiết bị làm Master. Chuẩn I2C thường được tìm thấy trong những thiết bị điện tử như EEPROM 24CXX, realtime DS1307,... và một số thiết bị khác.

*(Các kiến thức về I<sup>2</sup>C được trình bày sau đây được sưu tầm từ trang web [www.hocavr.com](http://www.hocavr.com)).*

Chuẩn I2C được kết nối theo dạng sau:



Trong chuẩn I2C có hai đường dây dùng để truyền và nhận dữ liệu: SDA, và SCL. Trong đó SDA là đường dữ liệu truyền nhận. Dữ liệu truyền nhận là các bit 0 và 1 tương ứng với mức thấp và mức cao. Vị trí bit trong dữ liệu được đồng bộ hóa bởi xung clock thông qua đường SCL. Như vậy đường SCL là đường cung cấp xung đồng bộ dữ liệu. Cả hai đường SCL và SDA đều có cấu hình cực góp hở (Open-collector) vì thế khi được lắp đặt trong hệ thống cần phải có điện trở kéo lên Vcc để tạo mức thấp và mức cao trong quá trình truyền dữ liệu.

Các thiết bị trong truyền theo chuẩn I2C trong cùng một hệ thống được kết nối song song, mỗi thiết bị đều có địa chỉ riêng dùng để phân biệt với nhau trong quá trình truyền và nhận dữ liệu. Tại một thời điểm chỉ có hai thiết bị, một thiết bị đóng vai trò là slave và một thiết bị đóng vai trò là master, truyền và nhận dữ liệu. Các thiết bị khác trong cùng một bus bị vô hiệu hóa.



**II. Các thuật ngữ và giao thức truyền trong I<sup>2</sup>C:**

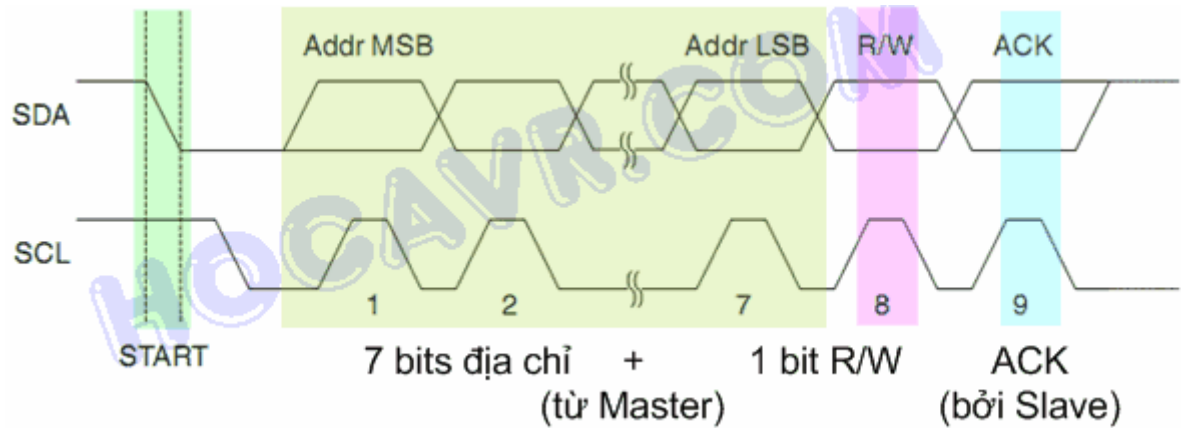
Sau đây là các thuật ngữ thường được sử dụng trong chế độ truyền nối tiếp I<sup>2</sup>C, trong mỗi thuật ngữ chúng ta sẽ tìm hiểu về định nghĩa và vai trò trong giao thức. Trên cơ sở đó sẽ hiểu rõ cách thức hoạt động của chuẩn I<sup>2</sup>C:

- **Master:** Là thiết bị khởi động quá trình truyền nhận. Thiết bị này sẽ phát ra các bit dữ liệu trên đường SDA và xung đồng bộ trên đường SCL. (Trong giáo trình này Master là chip ARM9260).
- **Slave:** Là thiết bị phục vụ cho Master khi được yêu cầu. Mỗi một thiết bị có một địa chỉ riêng. Địa chỉ thiết bị được quy định như sau: Mỗi loại thiết bị khác nhau (chẳng hạn EEPROM, Realtime, hay ADC, ...) nhà sản xuất sẽ quy định một địa chỉ khác nhau, được gọi là địa chỉ thiết bị. Bên cạnh đó còn có địa chỉ phân biệt giữa các thiết bị cùng loại (Chẳng hạn nhiều EEPROM 24CXX được kết nối chung với nhau), địa chỉ này được người sử dụng thiết bị quy định.
- **SDA-Serial Data:** Là đường truyền dữ liệu nối tiếp. Đường này sẽ đảm nhận nhiệm vụ truyền thông tin về địa chỉ và dữ liệu theo thứ tự từng bit. Lưu ý trong chuẩn I<sup>2</sup>C thì bit MSB được truyền trước nhất, cuối cùng là LSB, cách truyền này ngược với chuẩn truyền nối tiếp UART.
- **SCL-Serial clock:** Là đường giữ nhịp cho truyền và nhận dữ liệu. Nhịp đồng bộ dữ liệu do thiết bị đóng vai trò là Master phát ra. Quy định của quá trình truyền dữ liệu này là: Tại thời điểm mức cao của SCL dữ liệu trên chân SDA không được thay đổi trạng thái. Dữ liệu trên chân SDA chỉ được quyền thay đổi trạng thái khi xung SCL ở mức thấp. Khi chân SCL mức cao, nếu SDA thay đổi trạng thái thì thiết bị I<sup>2</sup>C nhận ra là các bit Start hoặc Stop;
- **START transmittion:** Là trạng thái bắt đầu quá trình truyền và nhận dữ liệu được quy định: trạng thái chân SCL mức cao và chân SDA chuyển trạng thái từ mức cao xuống mức thấp, cạnh xuống.
- **STOP transmittion:** Là trạng thái kết thúc quá trình truyền và nhận dữ liệu được quy định: trạng thái chân SCL mức cao và chân SDA chuyển trạng thái từ mức thấp lên mức cao;
- **Address:** Là địa chỉ thiết bị, thông thường được quy định có 7 bit;

- **Data:** Là dữ liệu truyền nhận, thông thường được quy định có 8 bit;
- **Repeat Start-Bắt đầu lặp lại:** Khoảng giữa hai trạng thái Start và Stop là khoảng bận của đường truyền, các Master khác không được tác động vào đường truyền trong khoảng này. Trường hợp sau khi kết thúc truyền nhận mà Master không gửi trạng thái Stop mà gửi thêm trạng thái Start để tiếp tục truyền nhận dữ liệu, quá trình này gọi là Repeat Start. Trường hợp này xuất hiện khi Master muốn lấy hoặc truyền dữ liệu liên tiếp từ các Slave.

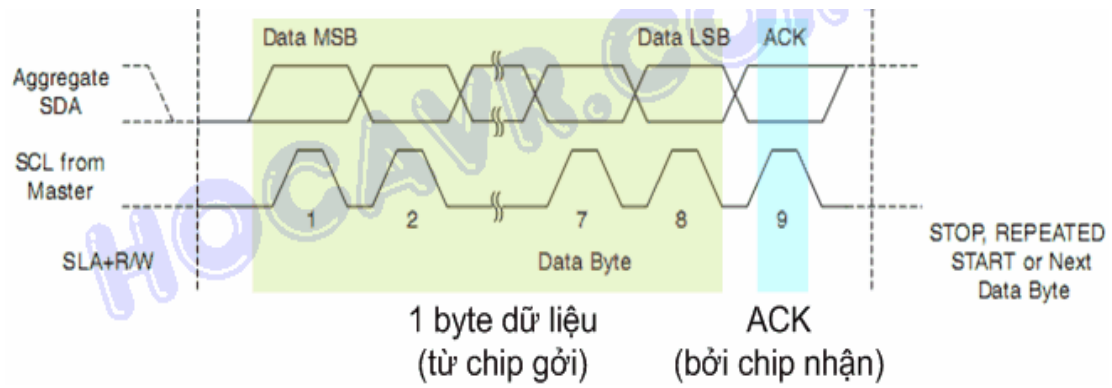
- **Address Packet Format-Định dạng gói dữ liệu:** Trong mạng I2C tất cả các thiết bị đều có thể là Master hay Slave. Mỗi thiết bị có một địa chỉ cố định gọi là Device address. Khi một Master muốn giao tiếp với một Slave, trước hết nó sẽ tạo ra một trạng thái START tiếp theo sẽ gửi địa chỉ thiết bị của Slave cần giao tiếp trên đường truyền, thì sẽ xuất hiện thuật ngữ gói địa chỉ (Address packet). Gói địa chỉ trong I2C có định dạng 9 bits trong đó 7 bits đầu (gọi là SLA, được gửi liền sau START) chứa địa chỉ Slave (Một số trường hợp địa chỉ có 10 bits), một bit READ/WRITE và một bit ACK (Acknowledge) xác nhận thông tin địa chỉ. Địa chỉ có độ dài 7 bits nên số thiết bị tối đa có thể giao tiếp là 128 thiết bị. Nhưng có một số địa chỉ không được hiểu là địa chỉ thiết bị. Các địa chỉ đó là 1111xxx (tức là các địa chỉ lớn hơn hoặc bằng 120 không được dùng). Riêng địa chỉ 0 được dùng cho cuộc gọi chung. Bit READ/WRITE được truyền theo sau 7 bits địa chỉ là biết báo cho Slave biết là Master muốn đọc hay ghi dữ liệu vào Slave. Nếu bit này bằng 0 (mức thấp) thì quá trình ghi dữ liệu từ Master đến Slave được yêu cầu, nếu bit này bằng 1 thì Master muốn đọc dữ liệu từ Slave về. 8 bits trên (SLA+RD/WR) được Master phát ra sau khi phát START, nếu một Slave trên mạng nhận ra rằng địa chỉ mà Master yêu cầu trùng khớp với địa chỉ của mình, nó sẽ đáp trả lại Master bằng cách đáp trả lại bằng tín hiệu xác nhận ACK bằng cách kéo chân SDA xuống thấp trong xung giữ nhịp thứ 9. Ngược lại nếu không có Slave đáp ứng lại, thì chân SDA vẫn giữ ở mức cao trong xung giữ nhịp thứ 9 thì gọi là tín hiệu không xác nhận NOT ACK, lúc này Master cần phải có những ứng xử phù hợp tùy theo mỗi trường hợp cụ thể, ví dụ Master có thể gửi trạng thái STOP và sau đó phát lại địa chỉ Slave khác... Như vậy, trong 9 bit của gói địa chỉ thì chỉ có 8 bits được gửi từ Master, bit còn lại là do Slave. Ví dụ Master muốn yêu cầu “đọc” dữ liệu từ Slave có địa chỉ

43, nó cần phát ra cho Slave một byte như sau:  $(43 \ll 1) + 1$ , trong đó  $(43 \ll 1)$  là dịch số 43 về bên trái 1 vị trí vì 7 bits địa chỉ nằm ở các vị trí cao trong gói địa chỉ, sau đó cộng giá trị này với 1 tức là quá trình đọc dữ liệu được yêu cầu. Hình bên dưới mô tả quá trình phát gói địa chỉ thiết bị từ Master đến Slave;



- **General call- Cuộc gọi chung:** Khi Master phát đi gói địa chỉ có dạng 0 (thực chất là 0+W) tức nó muốn thực hiện một cuộc gọi chung đến tất cả các Slave. Tất nhiên cho phép hay không cho phép là do cuộc gọi chung quyết định. Nếu Slave được cài đặt cho phép cuộc gọi chung, chúng sẽ đáp lại Master bằng bit ACK. Cuộc gọi chung thường xảy ra khi Master muốn gửi dữ liệu chung đến tất cả các Slave. Chú ý là cuộc gọi chung có dạng 0+R là vô nghĩa vì không bao giờ có trường hợp là Master nhận dữ liệu từ tất cả các Slave vào cùng một thời điểm.

- **Data Packet Format-Định dạng gói dữ liệu:** Sau khi địa chỉ đã được phát đi, Slave sẽ đáp ứng lại Master bằng ACK thì quá trình truyền nhận dữ liệu bắt đầu giữa cặp Master và Slave này. Tùy vào bit R/W trong gói địa chỉ, dữ liệu có thể được truyền theo hướng từ Master đến Slave hay từ Slave đến Master. Dù đi chuyển theo hướng nào thì gói dữ liệu luôn bao gồm 9 bits trong đó 8 bits đầu là dữ liệu, 1 bit cuối là ACK. 8 bits dữ liệu do thiết bị phát gửi và bit ACK do thiết bị nhận tạo ra. Ví dụ Master tiến hành gửi dữ liệu đến Slave, nó sẽ phát ra 8 bits dữ liệu, Slave nhận và phát ra ACK (kéo SDA xuống 0 ở chân thứ 9) sau đó Master sẽ quyết định gửi tiếp byte dữ liệu khác hay không. Nếu Slave phát tín hiệu NOT ACK (không tác động chân SDA xuống mức thấp ở xung giữ nhịp thứ 9) sau khi nhận dữ liệu thì Master sẽ kết thúc quá trình gửi bằng cách phát đi trạng thái STOP. Hình bên dưới mô tả định dạng gói dữ liệu trong I2C.



- **Phối hợp gói địa chỉ và gói dữ liệu:**

Một quá trình truyền nhận thường được bắt đầu từ Master. Master phát đi một bit trạng thái START sau đó gửi gói địa chỉ SLA+R/W trên đường truyền. Tiếp theo nếu có một Slave đáp ứng lại, dữ liệu có thể truyền nhận liên tiếp trên đường truyền (1 hoặc nhiều byte liên tiếp). Khung truyền thông thường được mô tả như hình bên dưới.



Multi-Master Bus-Đường truyền đa thiết bị chủ: Như đã trình bày ở trên, I2C là chuẩn truyền thông đa thiết bị chủ, nghĩa là tại một thời điểm có thể có nhiều hơn 1 thiết bị làm Master nếu các thiết bị này phát ra bit trạng thái START cùng một lúc. Nếu các Master có cùng yêu cầu và thao tác đối với Slave thì chúng có thể cùng tồn tại và quá trình truyền nhận có thể thành công. Tuy nhiên trong đa số trường hợp sẽ có một số Master bị thất lạc. Một Master bị thất lạc khi nó truyền/nhận một mức cao trong khi một Master khác truyền/nhận mức thấp.

### III. Kết luận:

Chúng ta đã tìm hiểu những kiến thức căn bản nhất về chuẩn giao tiếp I2C, trong đó bao gồm định nghĩa về các thuật ngữ, cấu trúc gói địa chỉ và gói dữ liệu. Thế nhưng đây chỉ mới là các kiến thức căn bản chung nhất giúp cho chúng ta hiểu nguyên lý hoạt động của hệ thống giao tiếp I2C. Trong những bài sau, chúng ta sẽ tìm hiểu cách mà hệ thống linux dùng để quản lý đường truyền I2C giao tiếp với các thiết bị Slave khác. Đồng thời sẽ đưa ra những bài tập ví dụ viết chương trình user

giao tiếp với eeprom 24c08. Trên cơ sở những phương pháp lập trình này người học có thể áp dụng điều khiển các thiết bị hoạt động theo chuẩn I2C khác.

**7-2- I2C TRONG LINUX:****I. Giới thiệu:**

Giao tiếp theo chuẩn I2C được chia làm 2 thành phần, khởi tạo giao thức truyền từ *Master* và quy định giao thức nhận của *Slave*. Ở đây chip AT91SAM9260 sẽ đóng vai trò là *Master* được điều khiển bởi hệ điều hành nhúng *Linux*. Và *Slave* là các thiết bị ngoại vi do hệ thống *Linux* điều khiển. Giao thức truyền của *Master* phải phù hợp với giao thức nhận của *Slave* thì chúng mới có thể thực hiện thành công việc truyền và nhận dữ liệu với nhau.

Bài này sẽ nghiên cứu các giao thức mà hệ điều hành *Linux* hỗ trợ giao tiếp với nhiều thiết bị ngoại vi khác nhau. Để sau khi hiểu nguyên lý hoạt động của từng thiết bị *Slave* chúng ta có thể áp dụng vào điều khiển dễ dàng.

Các kiến thức có liên quan đến chuẩn I2C của hệ điều hành *Linux* được viết rất đầy đủ trong thư mục Documentation/I2C của mã nguồn *kernel*. Những thông tin sau đây được biên soạn từ tài liệu này, nếu có những vấn đề không được trình bày rõ các bạn có thể tham khảo thêm.

*Linux* quy định những thuật ngữ sau để thuận tiện cho việc mô tả các giao thức I2C:

S	(1 bit)	:	Start bit
P	(1 bit)	:	Stop bit
Rd/Wr	(1 bit)	:	Read/Write bit. Rd = 1, Wr = 0.
A, NA	(1 bit)	:	Bit ACK hoặc bit NOT ACK.
Addr	(7 bits)	:	Là 7 bits địa chỉ thiết bị (device address). Địa chỉ này có thể mở rộng lên thành 10 bits.
Comm	(8 bits)	:	Là 8 bits địa chỉ của thanh ghi trong thiết bị. Mỗi thiết bị bao gồm có nhiều ô nhớ khác nhau, mỗi ô nhớ chứa một thông tin riêng và có duy nhất một địa chỉ.
Data	(8 bits)	:	Là 8 bits dữ liệu truyền/nhận, trong chế độ truyền theo byte. Trong chế độ truyền theo word, được hiểu là 8 bits cao DataHigh hoặc 8 bits thấp DataLow.

Count (8 bits): Là 8 bits quy định số bytes của khối dữ liệu trong chế độ truyền/nhận dữ liệu theo khối.

Lưu ý: Những tham số chứa trong dấu ngoặc vuông [...] được hiểu là được truyền từ thiết bị *Slave* đến *Master* ngược lại được truyền từ *Master* đến *Slave*. Ví dụ:

Data : Là dữ liệu truyền từ *Master* đến *Slave*;

[Data] : Là dữ liệu truyền từ *Slave* đến *Master*;

*Linux* xây dựng một giao thức để giao tiếp với chuẩn I2C mang tên SMBus (System Management Bus) để giao tiếp với các thiết bị được quy định theo chuẩn I2C. Có nhiều thiết bị ngoại vi khác nhau thì cũng sẽ có nhiều cấu trúc giao tiếp khác nhau tương ứng với từng chức năng truy xuất mà thiết bị hỗ trợ. Nhiệm vụ của người lập trình là phải lựa chọn giao thức SMBus phù hợp để giao tiếp với thiết bị cần điều khiển. Phần tiếp theo sẽ trình bày các giao thức SMBus thường được sử dụng mà *Linux* hỗ trợ.

## **II. Các giao diện hàm trong driver I2C:**

### **a. Giới thiệu về driver I2C trong Linux:**

Thông thường các thiết bị I2C được điều khiển bởi lớp *kernel*, thông qua các lệnh được định nghĩa trong mã nguồn *Linux*. Thế nhưng chúng ta cũng có thể sử dụng những giao diện được định nghĩa sẵn trong *driver* I2C do *Linux* hỗ trợ. Nghĩa là thay vì điều khiển trực tiếp thông qua các hàm trong *kernel*, chúng ta sẽ điều khiển gián tiếp thông qua các giao diện hàm trong *driver*. Các giao diện hàm này sẽ gọi các hàm trong *kernel* hỗ trợ để điều khiển *Slave* theo yêu cầu từ *user*.

*Driver* I2C được chứa trong thư mục *drivers/I2C* của mã nguồn *Linux*. Thông thường chúng ta sử dụng các hàm trong tập tin *driver* I2C-device.c. Trong tập tin I2C-dev.c định nghĩa các giao diện hàm như: `read()`, `write()` và `ioctl()` và một số những giao diện khác phục vụ cho đóng và mở tập tin thiết bị. Trong đó `read()` và `write()` dùng để đọc và ghi vào thiết bị *Slave* theo dạng từng khối. Ngoài ra bằng cách dùng giao diện hàm `ioctl()` chúng ta có thể thực hiện tất cả các thao tác đọc/ghi trên thiết bị *Slave* và một số những thao tác khác như định địa chỉ thiết bị, kiểm tra các hàm chức năng, chọn chế độ giao tiếp 8 bits hay 10 bits địa chỉ.

Tiếp theo chúng ta sẽ tìm hiểu các giao diện `read()`, `write()` và `ioctl()`.

**b. Giao tiếp với thiết bị I2C Slave thông qua driver I2C:**

Để thuận tiện cho việc tìm hiểu và ứng dụng các hàm giao diện sao cho thuận lợi nhất, chúng ta sẽ tìm hiểu theo hướng nghiên cứu các bước thực hiện và giải thích từng đoạn chương trình ví dụ. Các đoạn chương trình ví dụ này nằm trong *user application*, sử dụng các hàm hỗ trợ trong *driver I2C*.

**Bước 1:** Đầu tiên muốn giao tiếp với Bus I2C chúng ta phải xác định sự tồn tại của giao diện tập tin thiết bị trong thư mục `/dev/`. Thường thì tập tin thiết bị này có tên là `/dev/i2c-0`. Nếu không có tập tin thiết bị này trong thư mục `/dev/` chúng ta phải biên dịch lại mã nguồn *kernel* và check vào mục biên dịch *driver I2C*. Sau khi biên dịch xong, cài đặt vào kit và khởi động lại hệ thống.

**Bước 2:** Trong *user application*, mở tập tin thiết bị I2C-0 chuẩn bị thao tác;

```
/*Biến lưu số mô tả tập tin khi thiết bị được mở*/
int fd_I2C;
/*Gọi giao diện hàm mở tập tin thiết bị*/
fd_I2C = open("/dev/I2C-0, O_RDWR);
/*Kiểm tra lỗi trong quá trình mở tập tin thiết bị*/
if (fd_I2C < 0) {
    /* ERROR HANDLING; you can check errno to see what went wrong */
}
/*Nếu có lỗi xảy ra thì thoát chương trình đang gọi*/
exit(1);
}
```

**Bước 3:** Xác định địa chỉ thiết bị muốn giao tiếp;

Bằng cách gọi giao diện hàm `ioctl()` với tham số `I2C_SLAVE` như sau:

```
/*Biến lưu địa chỉ thiết bị cần giao tiếp*/
int addr = 0x40;
/*Gọi hàm ioctl() để xác định địa chỉ thiết bị với số định danh lệnh là I2C_SLAVE*/
if (ioctl(fd_I2C, I2C_SLAVE, addr) < 0) {
    /*ERROR HANDLING; you can check errno to see what went wrong*/
}
```



```
/*Thoát khỏi chương trình khi có lỗi xảy ra*/
```

```
    exit(1);  
}
```

**Bước 4:** Sử dụng các hàm giao diện `read()`, `write()` và `ioctl()` để truyền/nhận dữ liệu với *Slave*.

- Dùng giao diện hàm `read` để đọc dữ liệu từ *Slave*:

```
/*Khai báo biến đệm lưu giá trị trả về khi gọi hàm read()*/
```

```
char buf[10];
```

```
/*Gọi hàm read() đọc dữ liệu hiện tại của thiết bị Slave, kích thước đọc về là 1 byte*/
```

```
if (read(fd_I2C, buf, 1) != 1) {
```

```
/* ERROR HANDLING: I2C transaction failed */
```

```
/*Trong trường hợp có lỗi xảy ra thoát khỏi chương trình thực thi*/
```

```
    exit (1);
```

```
}
```

```
/*Lúc này dữ liệu đã chứa trong bộ nhớ đệm buf[0]*/
```

Cấu trúc giao thức này như sau:

**S Addr Rd [A] [Data] NA P**

Trong trường hợp đọc về nhiều byte liên tiếp, chúng ta cũng dùng giao diện hàm `read()` nhưng với độ dài `n`: `read(fd_I2C, buf, n)`; Khi đó cấu trúc của giao thức ngày như sau:

**S Addr Rd [A] [Data] [A] [Data] ... [A] [Data] NA P**

*\*\*Trong một số thiết bị Slave dùng I2C, có hỗ trợ một thanh ghi dùng để lưu địa chỉ hiện tại truy xuất dữ liệu, do đó khi gọi hàm `read()` dữ liệu đọc về sẽ là nội dung của ô nhớ có địa chỉ lưu trong thanh ghi này. Khi đọc 1 byte thì nội dung của thanh ghi địa chỉ này sẽ tăng lên 1 đơn vị.*

- Dùng giao diện hàm `write` để ghi dữ liệu vào *Slave*:

Chúng ta cũng có thể dùng giao diện hàm `write()` để ghi dữ liệu vào *Slave*. Khi đó cấu trúc của giao thức này như sau:

- Trong trường hợp ghi 1 byte vào thiết bị *Slave*:

**S Addr Wr [A] Data [A] P**

- Trong trường hợp ghi nhiều byte vào thiết bị *Slave*:

S Addr Wr [A] Comm [A] Data [A] ... Data [A] P

Ví dụ sau sẽ minh họa cách ghi bộ nhớ đệm có độ dài 3 bytes vào thiết bị *Slave*:

```
buf[0] = Data0;
buf[1] = Data1;
buf[2] = Data2;
if (write(fd_I2C, buf, 3) != 3) {
/* ERROR HANDLING: I2C transaction failed */
}
```

- Dùng giao diện hàm `ioctl()` truyền/nhận dữ liệu và các chức năng điều khiển khác trong *driver* I2C, mỗi chức năng sẽ tương ứng với các tham số lệnh khác nhau.
  - `ioctl(file, I2C_SLAVE, long addr)` : Dùng để thay đổi địa chỉ thiết bị muốn giao tiếp. Địa chỉ này là giá trị địa chỉ khi chưa thêm bit R/W vào vị trí bit thứ 8 (LSB) của gói địa chỉ.
  - `ioctl(file, I2C_TENBIT, long select)`: Dùng để quy định chế độ giao tiếp 8 bits hay 10 bits địa chỉ. Chế độ giao tiếp địa chỉ 10 bits nếu `select` khác 0, chế độ giao tiếp địa chỉ 7 bit nếu `select` bằng 0. Và mặc định là chế độ giao tiếp 7 bits địa chỉ. Hàm này chỉ hợp lệ khi *driver* I2C có hỗ trợ `I2C_FUNC_10BIT_ADDR`.
  - `ioctl(file, I2C_PEC, long select)`: Dùng để khởi động hay tắt chế độ kiểm tra lỗi trong quá trình truyền. Bật chế độ kiểm tra lỗi khi `select` khác 0, tắt chế độ kiểm tra lỗi khi `select` bằng 0. Mặc định là tắt kiểm tra lỗi. Hàm này chỉ hợp lệ khi *driver* có hỗ trợ `I2C_FUNC_SMBUS_PEC`;
  - `ioctl(file, I2C_FUNCS, unsigned long *funcs)`: Dùng để kiểm tra chức năng `*funcs` của *driver* I2C có hỗ trợ hay không.
  - `ioctl(file, I2C_RDWR, struct I2C_rdwr_ioctl_data *msgset)`: Dùng để kết hợp hay quá trình đọc và ghi trong có ngăn cách bởi trạng thái STOP. Hàm chỉ hợp lệ khi *driver* có hỗ trợ chức năng `I2C_FUNC_I2C`. Hàm này có tham số là `struct`

I2C\_rdwr\_ioctl\_data \*msgset. Với cấu trúc I2C\_rdwr\_ioctl\_data được định nghĩa như sau:

```
struct I2C_rdwr_ioctl_data {  
    struct I2C_msg *msgs;    /* Con trỏ đến mảng dữ liệu của I2C  
    */  
    int nmsgs;                /* Số lượng I2C_msg muốn đọc và  
    ghi */  
}
```

với cấu trúc struct I2C\_msg lại được định nghĩa như sau:

```
struct I2C_msg {  
    __u16 addr;  
    __u16 flags;  
#define I2C_M_TEN                0x0010  
#define I2C_M_RD                0x0001  
#define I2C_M_NOSTART            0x4000  
#define I2C_M_REV_DIR_ADDR      0x2000  
#define I2C_M_IGNORE_NAK        0x1000  
#define I2C_M_NO_RD_ACK         0x0800  
#define I2C_M_RECV_LEN          0x0400  
    __u16 len;  
    __u8 * buf;  
}
```

Như vậy mỗi một phần tử msg[] sẽ chứa một con trỏ khác, con trỏ này chính là bộ đệm đọc hay ghi tùy thuộc vào giá trị của cờ I2C\_M\_RD được bật tương ứng cho từng msg[]. Bên cạnh đó msg[] còn có thông số địa chỉ thiết bị, chiều dài bộ nhớ đệm, ...

- ioctl(file, I2C\_SMBUS, struct I2C\_smbus\_ioctl\_data \*args):  
Bản thân hàm này không gọi thực thi một chức năng cụ thể. Mà nó sẽ gọi thực thi một trong những hàm sau đây tùy vào các tham số chứa trong \*arg. Các chức năng được sử dụng tùy theo quy định của tham số \*args.

### **III. Các giao thức SMBus:**

#### **1. Giao thức SMBus Quick Command:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_write_quick(int file, __u8 value);
```

Giao thức này chỉ truyền một bit cho thiết bị *Slave* nằm tại vị trí của bit R/W có cấu trúc như sau:

**A Addr Rd/Wr [A] P**

### **2. Giao thức SMBus Receive Byte:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_read_byte(int file);
```

Giao thức này ra lệnh đọc một byte tại địa chỉ hiện tại từ thiết bị *Slave*. Thông thường lệnh này được dùng sau các lệnh khác dùng để quy định địa chỉ muốn đọc. Giao thức có cấu trúc như sau:

**S Addr Rd [A] [Data] NA P**

### **3. Giao thức SMBus Send Byte:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_write_byte(int file, __u8 value);
```

Giao thức này dùng để truyền một byte xuống thiết bị *Slave* có cấu trúc:

**S Addr Wr [A] Data [A] P**

### **4. Giao thức SMBus Read Byte:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_read_byte_data(int file, __u8 command);
```

Giao thức này dùng để nhận một byte có địa chỉ thanh ghi là command trong thiết bị *Slave*. Giao thức này tương đương với hai giao thức send byte và receive byte hoạt động liên tiếp nhưng không có bit STOP ngăn cách giữa hai giao thức này. Cấu trúc của giao thức SMBus như sau:

**S Addr Wr [A] Comm [A] S Addr Rd [A] [Data] NA P**

### **5. Giao thức SMBus Read Word:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_read_word_data(int file, __u8 command);
```

Giao thức này dùng để đọc một word data, bao gồm byte thấp và byte cao bắt đầu tại địa chỉ command. Giao thức có cấu trúc:

**S Addr Wr [A] Comm [A] S Addr Rd [A] [DataLow] A [DataHigh] NA P**

### **6. Giao thức SMBus Write Byte:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_write_byte_data(int file, __u8 command, __u8 value);
```

Giao thức này dùng để ghi một byte dữ liệu đến thanh ghi có địa chỉ command có cấu trúc như sau:

**S Addr Wr [A] Comm [A] Data [A] P**

### **7. Giao thức SMBus Write Word:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_write_word_data(int file, __u8 command, __u16 value);
```

Giao thức này dùng để ghi một word dữ liệu có 2 byte, HighByte và LowByte, đến thanh ghi có địa chỉ bắt đầu là command. Giao thức có cấu trúc:

**S Addr Wr [A] Comm [A] DataLow [A] DataHigh [A] P**

### **8. Giao thức SMBus Block Read:**

Giao thức này được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_read_block_data(int file, __u8 command, __u8 *values);
```

Dùng để đọc từ thiết bị *Slave* một khối dữ liệu có chiều dài lên tới 32 bytes (Tùy theo khả năng của thiết bị *Slave*) có địa chỉ bắt đầu từ command với số byte đọc được quy định trong tham số count. Giao thức này có cấu trúc sau:

**S Addr Wr [A] Comm [A]**

**S Addr Rd [A] [Count] A [Data] A [Data] A ... A [Data] NA P**

### **9. Giao thức SMBus Block Write:**

Được định nghĩa theo dạng hàm sau:

```
__s32 I2C_smbus_write_block_data(int file, __u8 command, __u8 length, __u8 *values);
```

Giao thức này dùng để ghi một khối dữ liệu có chiều dài quy định bởi length (tối đa là 32 bits) tại địa chỉ bắt đầu là command. Giao thức có cấu trúc như sau:

**S Addr Wr [A] Comm [A] Count [A] Data [A] Data [A] ... [A] Data [A] P**

*\*\*Trên đây là 9 giao thức căn bản và thường sử dụng nhất trong giao tiếp theo chuẩn I2C. Đây chỉ mới là những giao thức do Linux quy định sẵn cho Master khi*

*muốn thao tác với thiết bị Slave. Để giao tiếp thành công với mỗi thiết bị, trước tiên chúng ta phải tìm hiểu quy định về cách thức giao tiếp của thiết bị đó, sau đó sẽ áp dụng một hay nhiều giao thức SMBus trên để thực hiện một tác vụ cụ thể do thiết bị Slave hỗ trợ.*

#### **IV. Kết luận:**

Trong bài này chúng ta đã nghiên cứu những nguyên lý và các bước căn bản về thao tác truyền dữ liệu theo chuẩn I2C trong *driver* I2C\_dev. *Driver* I2C do *Linux* hỗ trợ bao gồm tất cả những giao thức phù hợp với các thiết bị *Slave* khác nhau. Trong các bài sau, chúng ta sẽ đi tìm hiểu giao thức truyền nhận của một số thiết bị *Slave* đồng thời sẽ áp dụng các hàm trong *driver* I2C điều khiển truy xuất dữ liệu từ các thiết bị này.

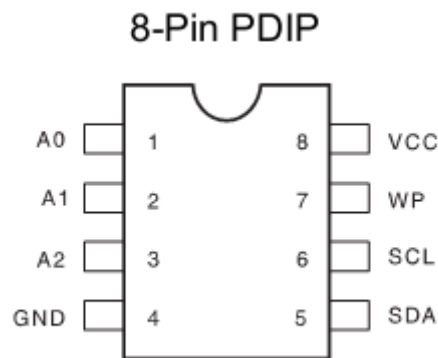
**7-3- THỰC HÀNH GIAO TIẾP EEPROM I2C 24C08:****I. Giới thiệu chung về EEPROM:**

**\*\***Để quyển sách này có sự tập trung và đúng hướng, trong phần này chúng tôi không trình bày một cách chi tiết cấu tạo và nguyên lý hoạt động mà chỉ trình bày giao thức truyền dữ liệu theo chuẩn I2C của EEPROM 24C08. Để từ đó áp dụng những lệnh đã học trong hai bài trước vào điều khiển thành công thiết bị Slave này.

**1. Mô tả:**

EEPROM 24C08 có những chức năng sau:

- Dung lượng 8Kb (1KB);
- Là ROM có thể lập trình và xóa bằng xung điện;
- Chế độ truyền theo chuẩn I2C;

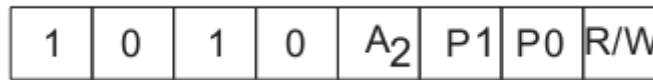
**2. Sơ đồ chân:**

*Trong đó:*

- A0, A1, A2: Là 3 chân địa chỉ ngõ vào dùng để chọn địa chỉ phân biệt nhiều eeprom ghép song song với nhau. Đối với eeprom 24C08 chỉ có 1 chân A2 được phép chọn lựa địa chỉ. Như vậy chỉ có 2 eeprom 24C08 được nối chung với nhau trên 1 bus I2C.
- SDA và SCL: Là chân dữ liệu và chân clock trong chuẩn I2C;
- WP: Là chân bảo vệ chống ghi vào eeprom;
- VCC và GND là hai chân cấp nguồn cho eeprom;

**3. Địa chỉ thiết bị:**

Địa chỉ thiết bị eeprom 24C08 được quy định như trong hình vẽ sau:

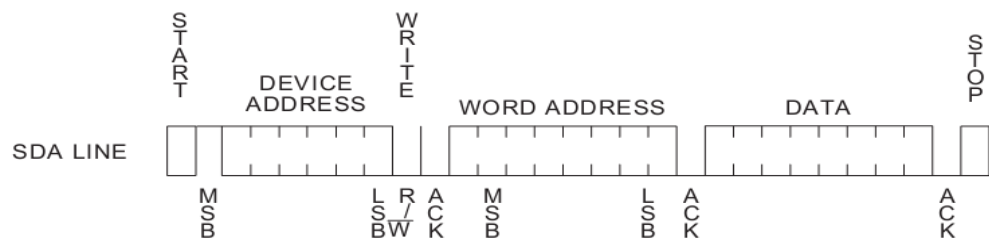


#### 4. *Giao thức ghi dữ liệu:*

EEPROM 24Cxx có hai chế độ ghi dữ liệu, ghi theo từng byte và ghi theo từng block.

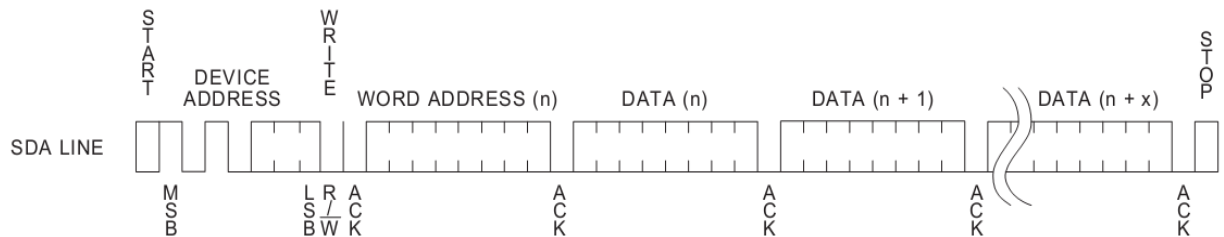
##### a. *Ghi theo từng byte:*

Chế độ này được minh họa bằng giao thức sau:



##### b. *Ghi theo từng block:*

Chế độ này được minh họa bằng giao thức sau:



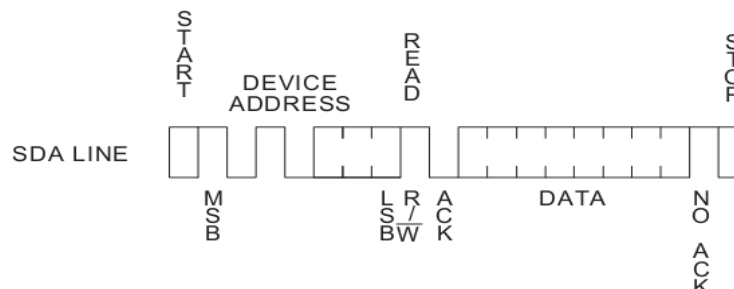
**\*\*EEPROM 24C08 có thể hỗ trợ chế độ ghi dữ liệu theo trang có độ dài lên tới 16 bytes.**

#### 5. *Giao thức đọc dữ liệu:*

EEPROM 24Cxx có 3 chế độ đọc dữ liệu, đọc tại địa chỉ hiện tại, đọc ngẫu nhiên và đọc theo tuần tự;

##### a. *Đọc tại địa chỉ hiện tại:*

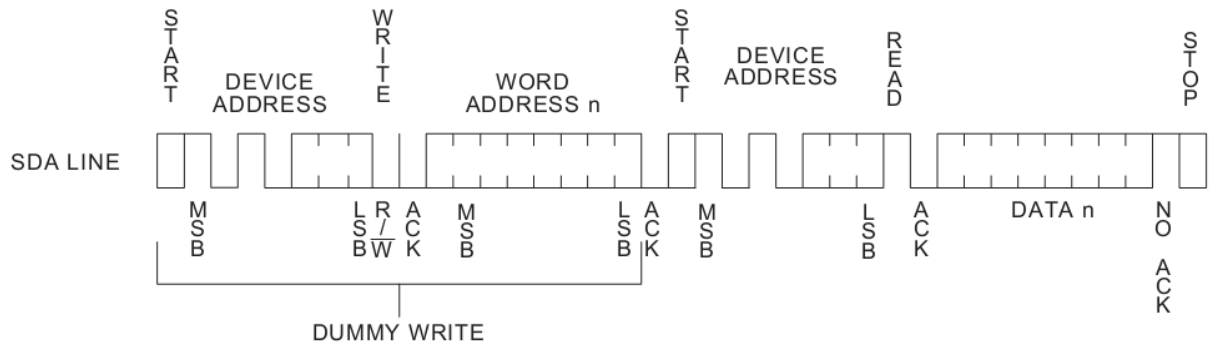
Chế độ này được minh họa bằng giao thức sau:



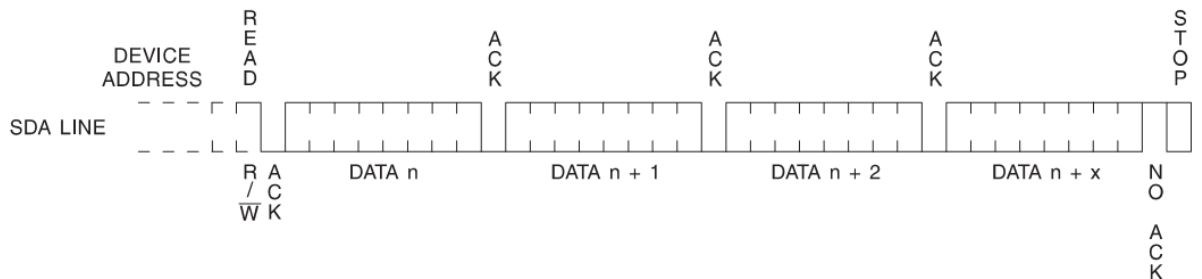


**b. Đọc ngẫu nhiên:**

Chế độ này được minh họa bằng giao thức sau:

**c. Đọc theo tuần tự:**

Chế độ này được minh họa bằng giao thức sau:

**II. Dự án điều khiển EEPROM 24C08:****1. Phác thảo dự án:**

Mục đích của dự án là làm cho người học có khả năng sử dụng những hàm trong driver I2C hỗ trợ để điều khiển EEPROM 24C08. Bên cạnh đó còn giúp người học rèn luyện cách viết một chương trình user application sử dụng thư viện liên kết động.

**a. Yêu cầu dự án:**

Chương trình được xây dựng thao tác với eeprom 24Cxx với các chức năng sau:

- Ghi một chuỗi thông tin bao gồm 256 bytes vào các ô nhớ trong eeprom 24Cxx bắt đầu từ 0 kết thúc 255. Để thực hiện chức năng này, người sử dụng chương trình nhập câu lệnh thực thi theo cú pháp sau:

```
./<tên chương trình> write_numbers
```

- Đọc lần lượt thông tin của các ô nhớ có địa chỉ từ 0 đến 255 trong eeprom 24Cxx xuất ra màn hình hiển thị. Để thực hiện chức năng này người sử dụng chương trình phải nhập câu lệnh thực thi theo cú pháp sau:

```
./<tên chương trình> read_numbers
```

- Ghi chuỗi ký tự được nhập từ người dùng vào eeprom bắt đầu từ ô nhớ có địa chỉ 00h, số ký tự được ghi phụ thuộc vào chiều dài của chuỗi ký tự. Để thực hiện chức năng này, người sử dụng chương trình phải nhập câu lệnh thực thi theo cú pháp sau:

```
./<tên chương trình> write_string
```

- Đọc chuỗi ký tự từ eeprom bắt đầu từ ô nhớ có địa chỉ 00h, số ký tự muốn đọc do người sử dụng chương trình quy định. Để thực hiện chức năng này người dùng phải nhập câu lệnh thực thi theo cú pháp sau:

```
./<tên chương trình> read_string <số ký tự muốn đọc>
```

### ***b. Phân công nhiệm vụ:***

- *Driver:*

Sử dụng driver I2C đã được hỗ trợ sẵn trong kernel. (Các hàm chức năng hỗ trợ trong driver I2C đã được chúng tôi trình bày kỹ trong bài trước).

- *Application:*

Chương trình trong user được xây dựng thành nhiều “lớp” con khác nhau, được chia thành các tập tin như: 24cXX.h, 24cXX.c và eeprom.c;

- Tập tin chương trình chính mang tên eeprom.c chứa hàm main() được khai báo dưới dạng cấu trúc tham số để thu thập thông tin từ người dùng. eeprom.c gọi các hàm được định nghĩa trong các tập tin khác (Chủ yếu là tập tin 24cXX.h). eeprom.c thực hiện 4 nhiệm vụ:

- Mở tập tin thiết bị driver I2C mang tên I2C-0 trong thư mục /dev/ sau đó quy định các thông số như: Địa chỉ Slave thiết bị, số bits địa chỉ thanh ghi, ... lưu vào cấu trúc eeprom để sử dụng trong những lần tiếp theo.

Tùy theo tham số lựa chọn của người dùng mà thực hiện một trong 4 chức năng sau:

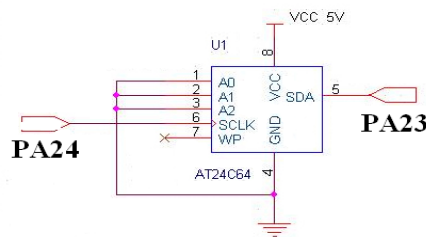
- Ghi lần lượt các số từ 00h đến FFh đến các ô nhớ có địa chỉ từ 00h đến FFh trong eeprom 24Cxx;
- Đọc nội dung của các ô nhớ từ 00h đến FFh lần lượt ghi ra màn hình hiển thị;

- Ghi chuỗi ký tự được nhập từ người dùng vào eeprom 24Cxx, với giới hạn số ký tự do eeprom quy định. Vị trí ghi bắt đầu từ địa chỉ 00h.
- Đọc chuỗi ký tự được lưu trong eeprom ghi ra màn hình hiển thị, với kích thước đọc do người lập trình quy định (nhập từ tham số người dùng);

## **2. Thực hiện:**

### **a. Kết nối phần cứng:**

Các bạn kết nối phần cứng theo sơ đồ sau đây:



### **b. Chương trình driver:**

Driver được sử dụng trong dự án này mang tên I2C-0 trong thư mục /dev/. Những chức năng lệnh trong driver được trình bày kỹ trong bài trước.

### **c. Chương trình application:**

- Tập tin chương trình chính: eeprom.c

*/\*Khai báo thư viện cho các hàm cần dùng trong chương trình\*/*

```
#include <stdio.h>
```

```
#include <fcntl.h>
```

```
#include <getopt.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include "24cXX.h" /*Gán tập tin định nghĩa 24cXX.h vào chương trình chính*/
```

*/\*Hàm in hướng dẫn cho người dùng trong trường hợp nhập sai cú pháp\*/*

```
void use(void)
```

```
{
```

```
printf("./i2c
read_numbers|read_string|write_numbers|write_string (<number
for read_string>|<string_to_write_string>");
exit(1);
}

/*Hàm thực hiện chức năng đọc nội dung của eeprom struct eeprom *e, từ ô nhớ
có địa chỉ int addr, kích thước muốn đọc là int size. Mỗi lần đọc, thông tin được
xuất ra màn hình dưới dạng số hex.*/

static int read_from_eeprom(struct eeprom *e, int addr, int size)
{
    /*Khai báo biến lưu ký tự đọc về và biến đếm để đọc về một khối ký tự*/
    int ch, i;
    /*Vòng lặp đọc thông tin từ eeprom tại địa chỉ addr đến size, sau khi đọc
    tăng giá trị addr lên 1 đơn vị*/
    for(i = 0; i < size; ++i, ++addr)
    {
        /*Đọc thông tin từ eeprom tại địa chỉ addr đồng thời kiểm tra lỗi
        trong quá trình đọc*/
        if((ch = eeprom_read_byte(e, addr)) < 0)
            /*In ra thông báo trong trường hợp có lỗi xảy ra*/
            printf("read error\n");
        /*Cứ mỗi 16 lần in thông tin thì xuống hàng một lần*/
        if( (i % 16) == 0 )
            printf("\n %.4x|  ", addr);
        /*Cứ mỗi 8 lần in thông tin thì thêm một khoảng trắng*/
        else if( (i % 8) == 0 )
            printf("  ");
        printf("%.2x ", ch);
    }
    /*In hai ký tự xuống hàng khi kết thúc quá trình đọc thông tin*/
    printf("\n\n");
    return 0;
}

/*Hàm thực hiện chức năng ghi số từ 00 đến ff bắt đầu từ địa chỉ addr*/
```

```
static int write_to_eeprom(struct eeprom *e, int addr)
{
    /*Biến đếm điều khiển vòng lặp ghi dữ liệu vào eeprom*/
    int i;
    /*Vòng lặp từ 0 đến 256, mỗi lần đọc tăng giá trị addr lên 1 đơn vị chuẩn bị cho lần đọc tiếp theo*/
    for(i=0; i<256; i++, addr++)
    {
        /*Ghi thông tin ra màn hình trước khi ghi vào eeprom*/
        if( (i % 16) == 0 )
            printf("\n %.4x|  ", addr);
        else if( (i % 8) == 0 )
            printf("  ");
        printf("%.2x ", i);
        /*Ghi dữ liệu vào eeprom tại địa chỉ addr, kiểm tra lỗi trong quá trình ghi dữ liệu*/
        if(eeprom_write_byte(e, addr, i)<0)
            printf("write error\n");
    }
    /*In ký tự xuống dòng khi quá trình đọc kết thúc*/
    printf("\n");
    return 0;
}

/*Hàm thực hiện chức năng ghi chuỗi thông tin từ người dùng vào eeprom*/
int test_write_ee(struct eeprom *e, char *data, int size)
{
    int i;
    /*Vòng lặp ghi từng ký tự trong chuỗi char *data vào eeprom tại địa chỉ i*/
    for(i=0; i<size; i++){
        /*Ghi ký tự data[i] vào địa chỉ i trong eeprom đồng thời kiểm tra lỗi trong quá trình ghi dữ liệu*/
        if(eeprom_write_byte(e, i, data[i])<0){
            printf("write error\n");
            return -1;
        }
    }
}
```

```
    }

    /*In thông báo khi quá trình ghi kết thúc*/
    printf("Writing finishes!");
    printf("\n");
    return 0;
}

/*Hàm thực hiện chức năng đọc ký tự từ eeprom bắt đầu từ địa chỉ 00, kích thước là size*/
int test_read_ee(struct eeprom *e, int size)
{
    int i;
    char ch;

    /*Đọc ký tự từ 0 đến size ghi ra màn hình hiển thị*/
    for(i=0;i<size;i++)
    {
        if((ch = eeprom_read_byte(e, i)) < 0) {
            printf("read error\n");
            return -1;
        }

        /*Dùng hàm putchar() để ghi ký tự ra màn hình hiển thị*/
        putchar(ch);
    }
    printf("\n");
    return 0;
}

int main(int argc, char** argv)
{
    /*Khai báo cấu trúc eeprom lưu những thông tin loại eeprom, số bit địa chỉ thành ghi, .. cấu trúc được định nghĩa trong tập tin 24cXX.h*/
    struct eeprom e;

    /*Số ký tự nhập vào*/
    int number_of_char;

    /*Thông báo mở tập tin thiết bị*/
    printf("Open /dev/i2c-0 with 8 bit mode\n");
```

```
/*Mở tập tin thiết bị i2c, cập nhật địa chỉ eeprom, hàm eeprom được định nghĩa trong tập tin 24cXX.c và 24cXX.h*/
if (eeprom_open("/dev/i2c-0", 0x50, EEPROM_TYPE_8BIT_ADDR, &e) <
0)

    printf("unable to open eeprom device file \n");
/*Trong trường hợp thực hiện chức năng đọc giá trị ô nhớ từ 00h đến ffh ghi ra màn hình hiển thị*/
if (!strcmp(argv[1], "read_numbers")) {
    fprintf(stderr, "Reading 256 bytes from 0x0\n");
    read_from_eeprom(&e, 0, 256);
/*Trong trường hợp thực hiện chức năng 1, ghi số từ 00 đến ff lên các ô nhớ trong eeprom bắt đầu từ địa chỉ 00h*/
} else if (!strcmp(argv[1], "write_numbers")) {
    fprintf(stderr, "Writing 0x00-0xff into 24C08 \n");
    write_to_eeprom(&e, 0);
    printf("Writing finishes!");
/*Trong trường hợp thực hiện chức năng đọc chuỗi thông tin từ eeprom hiển thị ra màn hình*/
} else if (!strcmp(argv[1], "read_string")) {
    number_of_char = atoi(argv[2]);
    printf("Reading eeprom from address 0x00 with size %d\n", number_of_char);
    printf ("The read string is: ");
    test_read_ee(&e, number_of_char);
/*Trong trường hợp thực hiện chức năng ghi chuỗi thông tin từ người dùng vào eeprom*/
} else if (!strcmp(argv[1], "write_string")) {
    number_of_char = strlen(argv[2]);
    fprintf(stderr, "Writing eeprom from address 0x00 with size %d \n", number_of_char);
    test_write_ee(&e, argv[2], number_of_char);
/*Trong trường hợp có lỗi xảy ra do cú pháp*/
} else {
    use();
    exit(1);
}
```

```
    }  
    eeprom_close(&e);  
    return 0;  
}
```

Hai tập tin sau được dùng để định nghĩa những hàm được sử dụng trong eeprom.c, truy xuất trực tiếp đến các hàm trong driver I2C-0 để điều khiển eeprom. Nếu có nhu cầu các bạn có thể đọc thêm để nghiên cứu ý nghĩa của từng hàm trong tập tin 24cXX.c. Nếu không chúng ta sẽ sử dụng những hàm chức năng được định nghĩa trong tập tin 24cXX.h như là những hàm hỗ trợ sẵn trong thư viện liên kết tĩnh.

- Tập tin thư viện: 24Cxx.h

```
/*Khai báo tập tin định nghĩa 24cXX.h*/  
#ifndef _24CXX_H_  
#define _24CXX_H_  
  
/*Gán thư viện của driver i2c-dev cho chuẩn i2c*/  
#include <linux/i2c-dev.h>  
#include <linux/i2c.h>  
  
/*Định nghĩa hằng số quy định chế độ truy xuất địa chỉ*/  
#define EEPROM_TYPE_UNKNOWN 0  
#define EEPROM_TYPE_8BIT_ADDR    1/*Chế độ truy xuất địa chỉ 8 bits*/  
#define EEPROM_TYPE_16BIT_ADDR   2/*Chế độ truy xuất địa chỉ 16 bits*/  
  
/*Định nghĩa cấu trúc eeprom  
Cấu trúc này bao gồm những thông tin sau:  
char *dev: con trỏ char lưu tên đường dẫn thiết bị  
int addr: Biến lưu địa chỉ thiết bị Slave  
int fd: Biến lưu số mô tả tập tin thiết bị khi được mở  
int type: Biến lưu kiểu eeprom truy xuất*/  
struct eeprom  
{  
    char *dev;        // device file i.e. /dev/i2c-N  
    int addr;         // i2c address  
    int fd;           // file descriptor  
    int type;         // eeprom type  
};
```



/\*

*Hàm eeprom\_open() dùng để mở tập tin thiết bị Slave eeprom;*

*Bao gồm các thông số sau:*

*char \*dev\_fqn: Đường dẫn đến tập tin thiết bị muốn mở;*

*int addr: Địa chỉ của thiết bị Slave muốn mở;*

*int type: Loại eeprom cần truy xuất;*

*struct eeprom: Sau khi mở thành công tập tin thiết bị với địa chỉ addr, loại*

*eeprom type, ... Các thông tin này sẽ được cập nhật trong các field tương ứng của cấu trúc struct eeprom;*

\*/

```
int eeprom_open(char *dev_fqn, int addr, int type, struct  
eeprom*);
```

/\*

*Hàm eeprom\_close() dùng để đóng thiết bị eeprom sau khi truy xuất xong;*

*Hàm bao gồm các tham số sau:*

*struct eeprom \*e: Là con trỏ thiết bị eeprom muốn đóng, sau khi đóng những thông tin trong các field của thiết bị \*e được khôi phục lại trạng thái ban đầu;*

\*/

```
int eeprom_close(struct eeprom *e);
```

/\*

*Hàm eeprom\_read\_byte() dùng để đọc một byte dữ liệu có địa chỉ \_\_u16*

*mem\_addr của thiết bị struct eeprom\* e; Giá trị đọc được sẽ trả về làm giá trị của hàm;*

*Lưu ý: Trước khi sử dụng hàm này, phải đảm bảo là thiết bị struct eeprom\* e đã được cập nhật địa chỉ trước đó bằng cách gọi hàm eeprom\_oepn() và hàm này sẽ gọi thực thi hàm ioctl(fd, I2C\_SLAVE, address); Được hỗ trợ trong driver*

*/dev/i2c-N*

\*/

```
int eeprom_read_byte(struct eeprom* e, __u16 mem_addr);
```

/\*

*Hàm `eeprom_read_current_byte()` đọc giá trị ô nhớ có địa chỉ hiện tại chứa trong thanh ghi địa chỉ của `eeprom struct eeprom *e`;*

*\*/*

```
int eeprom_read_current_byte(struct eeprom *e);
```

*/\**

*Hàm `eeprom_write_byte()` dùng để ghi giá trị `__u8 data` vào ô nhớ có địa chỉ `__u16 mem_addr` của thiết bị `struct eeprom *c`;*

*\*/*

```
int eeprom_write_byte(struct eeprom *e, __u16 mem_addr, __u8 data);
```

```
#endif
```

- Tập tin chương trình con: 24cXX.c

*/\*Tập tin 24cXX có nhiệm vụ định nghĩa những hàm đã khai báo trong tập tin 24cXX.h\*/*

```
#include <stdio.h>
```

```
#include <fcntl.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <linux/fs.h>
```

```
#include <sys/types.h>
```

```
#include <sys/ioctl.h>
```

```
#include <errno.h>
```

```
#include <assert.h>
```

```
#include <string.h>
```

```
#include "24cXX.h"
```

*/\*Hàm `i2c_smbus_access()` định nghĩa một giao thức tổng quát trong truyền|nhận dữ liệu thông qua chuẩn i2c. Nó bao hàm tất cả những giao thức khác, việc lựa chọn giao thức thực hiện do các tham số trong hàm quy định;*

*Hàm bao gồm những tham số căn bản như sau:*

*int file: Số mô tả tập tin thiết bị được mở trong hệ thống;*

*char read\_write: Cờ cho biết là lệnh đọc hay ghi vào Slave;*

*\_\_u8 command: Byte dữ liệu muốn ghi vào Slave; Thường thì command có vai trò là địa chỉ thanh ghi trong thiết bị muốn truy xuất;*

*int size: Là kích thước tính theo bytes của khối dữ liệu muốn truy xuất;*

*union i2c\_smbus\_data \*data: Là cấu trúc dữ liệu truyền nhận trong smBus i2c;\*/*

```
static inline __s32
i2c_smbus_access(int file, char read_write, __u8 command,int size,
union i2c_smbus_data *data){
    /*Khai báo biến lưu cấu trúc dữ liệu truyền nhận trong ioctl*/
    struct i2c_smbus_ioctl_data args;
    /*Cập nhật thông tin đọc hay ghi*/
    args.read_write = read_write;//read/write
    /*Cập nhật thông tin command cho câu lệnh*/
    args.command = command;
    /*Cập nhật kích thước dữ liệu muốn đọc hay ghi*/
    args.size = size;//size of data
    /*Lưu dữ liệu trả về trong trường hợp đọc, Dữ liệu muốn ghi trong trường
    hợp ghi */
    args.data = data;
    return ioctl(file,I2C_SMBUS,&args);
}
/*
```

*Hàm i2c\_smbus\_read\_byte() đọc về giá trị của ô nhớ có địa chỉ lưu trong thanh ghi địa chỉ của eeprom; Hàm có tham số là int file: Là số mô tả tập tin thiết bị được mở.*

```
*/
static inline __s32
i2c_smbus_read_byte(int file){
    /*Biến cấu trúc dữ liệu truy xuất từ thiết bị I2C*/
    union i2c_smbus_data data;
    if(i2c_smbus_access(file,I2C_SMBUS_READ,0,I2C_SMBUS_BYTE,&data))
        return -1;
    else
        return 0xFF & data.byte;
}
```

*/\*Hàm i2c\_smbus\_write\_byte() dùng để ghi dữ liệu có chiều dài 1 byte vào eeprom nhằm mục đích là cập nhật giá trị cho thanh ghi địa chỉ trong eeprom;*

*Hàm có các tham số như sau:*

*int file: Số mô tả tập tin thiết bị được mở;*

*\_\_u8 value: Giá trị muốn ghi vào eeprom, nói đúng hơn là địa chỉ muốn cập nhật\*/*

```
static inline __s32
i2c_smbus_write_byte(int file, __u8 value)
{
    return i2c_smbus_access(file, I2C_SMBUS_WRITE, value,
                            I2C_SMBUS_BYTE, NULL);
}
```

*/\*Hàm i2c\_smbus\_read\_byte\_data() có nhiệm vụ đọc giá trị của ô nhớ có địa chỉ cụ thể trong eeprom;*

*Hàm có các tham số sau:*

*int file: Số mô tả tập tin thiết bị được mở;*

*\_\_u8 command: Địa chỉ ô nhớ muốn đọc;\*/*

```
static inline __s32
i2c_smbus_read_byte_data(int file, __u8 command) {
    union i2c_smbus_data data;
    if (i2c_smbus_access(file, I2C_SMBUS_READ, command, I2C_SMBUS_BYTE_DATA, &data))
        return -1;
    else
        return 0x0FF & data.byte;
}
```

*/\* i2c\_smbus\_write\_byte\_data() dùng để ghi 1 byte vào ô nhớ có địa chỉ cụ thể trong eeprom;*

*Các tham số cụ thể hàm như sau:*

*int file: Số mô tả tập tin thiết bị đang được mở để thao tác;*

*\_\_u8 command: Địa chỉ thanh ghi muốn ghi dữ liệu;*

*\_\_u8 value: Giá trị dữ liệu muốn ghi;*

*\*/*

```
static inline __s32
```

```
i2c_smbus_write_byte_data(int file, __u8 command, __u8 value){
    union i2c_smbus_data data;
    data.byte = value;
    return i2c_smbus_access(file, I2C_SMBUS_WRITE, command,
                            I2C_SMBUS_BYTE_DATA, &data);
}

/*Hàm i2c_smbus_read_word_data() cũng tương tự như hàm
i2c_smbus_read_byte_data() nhưng dữ liệu trả về là một word có 32 bits*/
static inline __s32 i2c_smbus_read_word_data(int file, __u8
command)
{
    union i2c_smbus_data data;
    if(i2c_smbus_access(file, I2C_SMBUS_READ, command, I2C_SMBUS_WOR
D_DATA, &data))
        return -1;
    else
        return 0xFFFF & data.word;
}

/*Hàm i2c_smbus_write_word_data cũng tương tự như hàm
i2c_smbus_write_byte_data() nhưng dữ liệu trả về là một word có 32 bits */
static inline __s32 i2c_smbus_write_word_data(int file, __u8
command, __u16 value)
{
    union i2c_smbus_data data;
    data.word = value;
    return i2c_smbus_access(file, I2C_SMBUS_WRITE, command,
                            I2C_SMBUS_WORD_DATA, &data);
}

/*Định nghĩa hàm ghi 1 byte vào thiết bị struct eeprom *e*/
static int i2c_write_1b(struct eeprom *e, __u8 buf)
{
    int r;

    /*we must simulate a plain I2C byte write with SMBus functions*/
    r = i2c_smbus_write_byte(e->fd, buf);
    if(r < 0)
        fprintf(stderr, "Error i2c_write_1b: %s\n", strerror(errno));
}
```

```
        usleep(10);
        return r;
    }

    /*Định nghĩa hàm ghi 2 bytes vào thiết bị struct eeprom *e*/
    static int i2c_write_2b(struct eeprom *e, __u8 buf[2])
    {
        int r;

        //we must simulate a plain I2C byte write with SMBus functions

        r = i2c_smbus_write_byte_data(e->fd, buf[0], buf[1]);
        if(r < 0)
            fprintf(stderr, "Error i2c_write_2b: %s\n", strerror(errno));
        usleep(10);
        return r;
    }

    /*Định nghĩa hàm ghi 2 bytes vào thiết bị struct eeprom *e*/
    static int i2c_write_3b(struct eeprom *e, __u8 buf[3])
    {
        int r;

        // we must simulate a plain I2C byte write with SMBus functions
        //the __u16 data field will be byte swapped by the SMBus protocol
        r = i2c_smbus_write_word_data(e->fd, buf[0], buf[2] << 8 |
            buf[1]);
        if(r < 0)
            fprintf(stderr, "Error i2c_write_3b: %s\n", strerror(errno));
        usleep(10);
        return r;
    }

    /*Định nghĩa hàm mở thiết bị struct eeprom *e*/
    int
    eeprom_open(char *dev_fqn, int addr, int type, struct eeprom* e)
    {
        /*Định nghĩa các biến lưu thông tin về hàm hỗ trợ của i2c: funcs;
        Số mô tả tập tin thiết bị fd;
        Biến lưu mã lỗi trả về khi truy xuất: r*/
        int funcs, fd, r;
```

```
/*Xóa các thông tin trong cấu trúc struct eeprom *e*/
e->fd = e->addr = 0;
e->dev = 0;

/*Mở tập tin thiết bị theo đường dẫn dev_fqn*/
fd = open(dev_fqn, O_RDWR);

/*Kiểm tra lỗi trong quá trình mở thiết bị*/
if(fd <= 0)
{
    fprintf(stderr, "Error eeprom_open: %s\n",
    strerror(errno));
    return -1;
}

/*Gọi hàm ioctl trong driver kiểm tra những hàm trong driver hỗ trợ*/
if((r = ioctl(fd, I2C_FUNCS, &funcs) < 0))
{
    fprintf(stderr, "Error eeprom_open: %s\n", strerror(errno));
    return -1;
}

/*Lần lượt kiểm tra những hàm trong driver hỗ trợ*/
CHECK_I2C_FUNC( funcs, I2C_FUNC_SMBUS_READ_BYTE );
CHECK_I2C_FUNC( funcs, I2C_FUNC_SMBUS_WRITE_BYTE );
CHECK_I2C_FUNC( funcs, I2C_FUNC_SMBUS_READ_BYTE_DATA );
CHECK_I2C_FUNC( funcs, I2C_FUNC_SMBUS_WRITE_BYTE_DATA );
CHECK_I2C_FUNC( funcs, I2C_FUNC_SMBUS_READ_WORD_DATA );
CHECK_I2C_FUNC( funcs, I2C_FUNC_SMBUS_WRITE_WORD_DATA );

/*Gọi hàm ioctl() quy định địa chỉ của thiết bị muốn truy xuất*/
if( ( r = ioctl(fd, I2C_SLAVE, addr)) < 0)
{
    fprintf(stderr, "Error eeprom_open: %s\n", strerror(errno));
    return -1;
}

/*Cập nhật các thông tin sau khi mở thiết bị thành công vào cấu trúc struct eeprom *e để phục vụ cho những lần truy xuất thiết bị trong những lần tiếp theo*/
e->fd = fd;
```

```
e->addr = addr;
e->dev = dev_fqn;
e->type = type;
return 0;
}

/*Định nghĩa hàm đóng đóng thiết bị struct eeprom *e*/
int eeprom_close(struct eeprom *e)
{
    /*Gọi giao diện hàm close đóng tập tin thiết bị*/
    close(e->fd);
    /*Khôi phục lại các thông tin ban đầu trong cấu trúc struct eeprom *e*/
    e->fd = -1;
    e->dev = 0;
    e->type = EEPROM_TYPE_UNKNOWN;
    return 0;
}

/*Các hàm hỗ trợ đọc dữ liệu từ eeprom*/
/*Hàm đọc giá trị trong ô nhớ có địa chỉ mem_addr*/
int eeprom_read_byte(struct eeprom* e, __u16 mem_addr)
{
    /*Biến lưu mã lỗi trả về cho hàm trong trường hợp có lỗi xảy ra nếu không
    có lỗi thì trả về giá trị của ô nhớ đọc được*/
    int r;
    /*Gọi hàm ioctl() có chức năng xóa bộ nhớ đệm đọc trong kernel*/
    ioctl(e->fd, BLKFLSBUF);
    /*Kiểm tra từng loại eeprom là loại địa chỉ 8 bits hay 16 bits*/
    /*Trong trường hợp là loại eeprom có địa chỉ 8 bits*/
    if(e->type == EEPROM_TYPE_8BIT_ADDR)
    {
        /*Chỉ lấy 8 bits thấp của mem_addr lưu vào bộ đệm địa chỉ*/
        __u8 buf = mem_addr & 0xff;
        /*Gọi hàm ghi 1 byte địa chỉ vào eeprom*/
        r = i2c_write_1b(e, buf);
        /*Trong trường hợp eeprom loại 16 bits địa chỉ*/
    }
}
```



```
    } else if(e->type == EEPROM_TYPE_16BIT_ADDR) {  
        /*Định nghĩa mảng 2 biến 8 bits lưu byte địa chỉ thấp và byte địa chỉ cao*/  
        __u8 buf[2] = { (mem_addr >> 8) & 0x0ff, mem_addr & 0x0ff };  
        /*Ghi mảng 2 bytes địa chỉ vào eeprom bằng cách gọi hàm i2c_write_2b()*/  
        r = i2c_write_2b(e, buf);  
        /*Các trường hợp còn lại in ra lỗi vì không có loại eeprom được hỗ trợ*/  
    } else {  
        fprintf(stderr, "ERR: unknown eeprom type\n");  
        return -1;  
    }  
    /*Nếu ghi địa chỉ bị lỗi thì thoát chương trình và in ra mã lỗi*/  
    if (r < 0)  
        return r;  
    /*Nếu không có lỗi xảy ra tiếp tục đọc giá trị tại ô nhớ vừa cập nhật địa chỉ*/  
    r = i2c_smbus_read_byte(e->fd);  
    return r;  
}  
  
/*Hàm hỗ trợ eeprom ghi một byte vào địa chỉ cụ thể trong eeprom*/  
int  
eeprom_write_byte(struct eeprom *e, __u16 mem_addr, __u8 data)  
{  
    /*Ban đầu xác định địa chỉ cần ghi dữ liệu*/  
    /*Trong trường hợp eeprom có địa chỉ 8 bits*/  
    if(e->type == EEPROM_TYPE_8BIT_ADDR) {  
        /*Ghi 2 bytes, byte đầu tiên là địa chỉ của ô nhớ; byte tiếp theo là dữ liệu cần ghi*/  
        /*Khai báo mảng lưu 2 bytes thông tin*/  
        __u8 buf[2] = { mem_addr & 0x00ff, data };  
        /*Gọi hàm ghi 2 bytes vào eeprom*/  
        return i2c_write_2b(e, buf);  
    }  
    /*Trong trường hợp eeprom có địa chỉ 16 bits*/  
    /*Đầu tiên ghi 2 bytes (16 bits) địa chỉ vào eeprom cuối cùng ghi 1 bytes dữ liệu*/  
    } else if(e->type == EEPROM_TYPE_16BIT_ADDR) {  
        /*Khai báo mảng chứa 3 bytes thông tin*/
```

```
    __u8 buf[3] =  
    { (mem_addr >> 8) & 0x00ff, mem_addr & 0x00ff, data };  
    /*Gọi hàm ghi 3 bytes vào eeprom*/  
    return i2c_write_3b(e, buf);  
}  
  
/*Các trường hợp khác không thuộc loại eeprom được hỗ trợ*/  
    fprintf(stderr, "ERR: unknown eeprom type\n");  
    return -1;  
}
```

### **3. Biên dịch và thực thi chương trình:**

Biên dịch chương trình bằng câu lệnh sau:

```
arm-none-linux-gnueabi-gcc eeprom.c -o eeprom
```

Chép chương trình đã biên dịch vào kit và tiến hành kiểm tra các chức năng mà chương trình hỗ trợ.

### **III. Kết luận:**

Đến đây chúng ta đã hoàn thành việc ứng dụng những giao diện hàm trong driver I2C do linux hỗ trợ để điều khiển thành công một thiết bị Slave là eeprom 24c08. Bằng cách áp dụng kỹ thuật tương tự trong chương trình, chúng ta có thể tự mình xây dựng các chương trình khác điều khiển tất cả những thiết bị hoạt động theo chuẩn I2C. Do thời gian có hạn nên chúng tôi chỉ đưa ra một chương trình ví dụ về I2C.

Trong bài tiếp theo chúng ta sẽ nghiên cứu một driver khác đó là driver truyền nối tiếp theo chuẩn UART.

**BÀI 8****GIAO TIẾP ĐIỀU KHIỂN  
ADC ON CHIP****8-1 - TỔNG QUAN VỀ ADC ON CHIP:****I. Mô tả chung:**

a. **Đặc tính tổng quát:** ADC tích hợp trong vi điều khiển AT91SAM9260 có những đặc tính nổi bật sau:

- 2 kênh chuyển đổi tương tự số;
- Độ phân giải có thể lựa chọn 8 bits hoặc 10 bits;
- Tốc độ chuyển đổi ở chế độ 10 bits là 312K sample/sec; Chuyển đổi theo phương pháp xấp xỉ liên tiếp;
- Có thể cho phép không cho phép đối với từng kênh chuyển đổi;
- Nguồn xung kích có thể lựa chọn: Hardware-Software trigger; External trigger pin; Timer/Counter 0 to 2 output;

b. **Các chân được sử dụng trong module ADC:**

<i>Ký hiệu</i>	<i>Mô tả</i>
VDDANA	Nguồn cung cấp cho module analog;
ADVREF	Điện áp tham chiếu;
AD0-AD1	Kênh ngõ vào tương tự 0 và 1;
ADTRG	Nguồn xung trigger bên ngoài;

**II. Đặc tính hoạt động:**

ADC sử dụng xung ADC để thực hiện quá trình chuyển đổi của mình. Tốc độ xung chuyển đổi có thể được thay đổi tùy theo mục đích bằng các bits lựa chọn tần số PRESCAL trong thanh ghi Mode Register (ADC\_MR). Tốc độ chuyển đổi có thể nằm trong khoảng từ  $MCK/2$  khi  $PRESCAL=0$  đến  $MCK/128$  khi  $PRESCAL=63$ .

Giá trị điện áp chuyển đổi nằm trong khoảng từ 0V đến giá trị điện áp trên chân ADVREF và sử dụng phương pháp chuyển đổi ADC xấp xỉ liên tiếp.

Độ phân giải của ADC có thể lựa chọn giữa hai giá trị là 8 bits và 10 bits. Chế độ chuyển đổi ADC 8 bits được cài đặt bằng cách set bit LOWRES trong thanh ghi ADC Mode (ADC\_MR). Giá trị chuyển đổi ADC có thể được đọc trong 8 bits thấp của thanh ghi Channel Data Register x (ADC\_CDRx). Chế độ chuyển đổi ADC 10 bits được cài đặt bằng cách clear bit LOWRES trong thanh ghi ADC\_MR. Lúc này giá trị chuyển đổi ADC được đọc trong 2 bytes thấp và cao của thanh ghi ADC\_CDRx. (Trong user interface của module).

*Quá trình chuyển đổi ADC được thực hiện theo các bước sau:*

- Khởi tạo chân ngõ vào tương tự theo chế độ kéo xuống mức 0;
- Chọn mode hoạt động cho ADC;
- Reset lại ADC bằng cách set bit SWRST trong thanh ghi ADC\_CR;
- Cho phép hay không cho phép các kênh chuyển đổi ADC hoạt động;
- Tạo một xung trigger cho bit START trong thanh ghi ADC\_CR;
- Chờ cho đến khi xuất hiện mức cao của bit DRDY trong thanh ghi ADC\_SR;
- Đọc giá trị chuyển đổi trong các thanh ghi tương ứng với từng kênh chuyển đổi;

*Trong đó 3 bước cuối cùng được thực hiện liên tục để cập nhật dữ liệu chuyển đổi.*

### **III. Một số công thức quan trọng:**

#### **a. Công thức tính tốc độ xung ADC:**

$$\text{ADCClock} = \text{MCK} / ((\text{PRESCAL} + 1) * 2);$$

#### **b. Công thức tính thời gian khởi động:**

$$\text{Start Up Time} = (\text{STARTUP} + 1) * 8 / \text{ADCClock};$$

#### **c. Công thức tính thời gian lấy mẫu và giữ:**

$$\text{Sample \& Hold Time} = (\text{SHTIM} + 1) / \text{ADCClock};$$

**8-2 - ĐIỀU KHIỂN NHIỆT ĐỘ DÙNG ADC ON CHIP:****I. Phác thảo dự án:**

Với những kiến thức về ADC On Chip trong bài trước chúng ta sẽ kết hợp với những lệnh truy xuất thanh ghi trong linux để thiết kế một ứng dụng đơn giản đo nhiệt độ dùng LM35 hiển thị trên LED 7 đoạn.

**a. Yêu cầu dự án:**

Yêu cầu của dự án này như sau:

- Người sử dụng nhập vào hai thông số nhiệt độ, nhiệt độ giới hạn trên và nhiệt độ giới hạn dưới;
- Định thời mỗi 1s cập nhật nhiệt độ hiện tại một lần;
- Điều khiển một LED sáng tắt theo quy luật:
  - + LED sáng khi nhiệt độ hiện tại lớn hơn hoặc bằng nhiệt độ giới hạn trên;
  - + LED tắt khi nhiệt độ hiện tại nhỏ hơn hoặc bằng nhiệt độ giới hạn dưới;

**b. Phân công nhiệm vụ:**

- *Chương trình driver:*

Chương trình sử dụng 2 driver tương ứng với hai module khác nhau: Driver điều khiển đọc giá trị chuyển đổi ADC trong CHIP và một driver điều khiển quét LED 7 đoạn. Nhiệm vụ cụ thể của từng driver như sau:

➤ *Driver ADC:* Có tên `at91adc_dev.ko`

- Khởi tạo ADC trong CHIP;
- Sử dụng giao diện hàm `read()` để truyền nhiệt độ chuyển đã được chuyển đổi sang user. Khi gọi hàm `read()` Driver sẽ thực hiện những công việc sau:
  - Kích hoạt bộ chuyển đổi ADC hoạt động;
  - Chờ cho đến khi ADC chuyển đổi xong;
  - Đọc giá trị chuyển đổi;
  - Truyền sang user giá trị đọc được;
- Sử dụng giao diện `ioctl()` để SET và CLEAR một chân gpio để điều khiển động cơ. Giao diện hàm `ioctl()` có hai số định danh lệnh: `ADC_SET_MOTOR` và `ADC_CLEAR_MOTOR` tương ứng với set và clear chân GPIO.

➤ *Driver LED 7 Đoạn:* Có tên là `led_seg_dev.ko`

Thực hiện quét 8 led 7 đoạn hiển thị những giá trị nhiệt độ: Nhiệt độ giới hạn trên, nhiệt độ giới hạn dưới và nhiệt độ hiện tại;

- Sử dụng giao diện hàm `write()` để nhận các thông tin từ nhiệt độ từ user hiển thị ra led;
- Sử dụng phương pháp ngắt timer mềm để cập nhật quét LED hiển thị; Phương pháp điều khiển này cũng tương tự như trong bài thực hành led 7 đoạn.

Nhiệt độ hiển thị trên LEDs theo dạng sau:

<AA> < BB> <CC>

*Trong đó:*

- <AA> là giá trị nhiệt độ giới hạn dưới; (Giới hạn từ 00 đến 99);
- <BB> là giá trị nhiệt độ hiện tại, khả năng hiển thị từ 00 đến 99;
- <CC> là giá trị nhiệt độ giới hạn trên; (Giới hạn hiển thị từ 00 đến 99);

- *Chương trình application:* mang tên at91adc\_app.c

Xây dựng chương trình application theo hướng có tham số nhập từ người dùng. Để chạy chương trình, chúng ta nhập câu lệnh thực thi theo cú pháp sau:

```
./at91adc_app <AA> <CC>
```

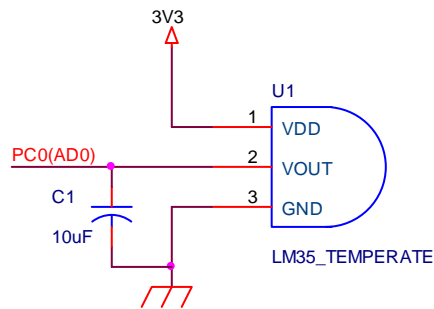
Chương trình application thực hiện những tác vụ:

- Khởi động 2 driver: at91adc\_dev và led\_seg\_dev;
- Định thời gian 1s cập nhật và so sánh nhiệt độ hiện tại với các nhiệt độ giới hạn trên dưới để điều khiển động cơ cho phù hợp.

## **II. Thực hiện:**

### **1. Kết nối phần cứng:**

Các bạn thực hiện kết nối phần cứng theo sơ đồ sau:



### **2. Chương trình driver:**

- **Driver ADC:** Có tên at91adc\_dev.ko

*/\*Khai báo các thư viện cần dùng cho các hàm trong chương trình\*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <asm/uaccess.h>
#include <mach/gpio.h>
#include <asm/gpio.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/atomic.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/fs.h>
#include <linux/clock.h>
#include <mach/at91_adc.h>
```

```
#define DRVNAME "at91adcDriver"
```

```
#define DEVNAME      "at91adcDevice"
/*Định nghĩa số định danh lệnh cho giao diện hàm ioctl của driver ADC*/
#define AT91ADC_DEV_MAGIC  'B'
/*Số định danh lệnh bật motor*/
#define AT91ADC_SET_MOTOR      _IO(AT91ADC_DEV_MAGIC, 10)
/*Số định danh lệnh tắt motor*/
#define AT91ADC_CLEAR_MOTOR    _IO(AT91ADC_DEV_MAGIC, 11)
/*Định nghĩa chân gpio cho chân điều khiển motor*/
#define GPIO_MOTOR      AT91_PIN_PC15
/*Định nghĩa dạng rút gọn cho lệnh set clear gpio*/
#define at91adc_set_motor()      gpio_set_value(GPIO_MOTOR,1)
#define at91adc_clear_motor()    gpio_set_value(GPIO_MOTOR,0)
/*Khai báo biến con trỏ chứa địa chỉ nền của các thanh ghi thao tác với ADC on chip*/
void __iomem *at91adc_base;
/*Khai báo biến con trỏ cấu trúc clock cấp xung cho ADC hoạt động*/
struct clk *at91adc_clk;
static atomic_t at91adc_open_cnt = ATOMIC_INIT(1);
/*Hàm hỗ trợ đọc dữ liệu từ ADC */
unsigned int
at91adc_read_current_value(void) {
/*Khai báo mảng chứa 100 giá trị đọc được từ ADC*/
    int current_value[100];
/*Khai báo biến lưu giá trị trung bình của các giá trị trong mảng current_value[100]*/
    int average=0;
/*Biến điều khiển hỗ trợ cho vòng lặp*/
    int i;
/*Vòng lặp hỗ trợ cho việc đọc các dữ liệu của ADC*/
    for (i = 0; i<100; i++) {

/*Tạo xung trigger cho bit START trong thanh ghi ADC_CR*/
        iowrite32(AT91_ADC_START, (at91adc_base + AT91_ADC_CR));
/*Chờ cho đến khi quá trình chuyển đổi hoàn thành, nghĩa là khi bit ADC_DRDY được set lên mức cao*/
while((ioread32(at91adc_base+AT91_ADC_SR) & AT91_ADC_DRDY) == 0)
        schedule();
/*Chép giá trị đã chuyển đổi chứa trong thanh ghi ADC_CHR(0) vào một phần tử trong mảng bộ nhớ đệm trong kernel*/
        current_value[i]=ioread32(at91adc_base + AT91_ADC_CHR(0));
    }
/*Tính tổng các giá trị chứa trong mảng*/
    for (i=0; i<100; i++) {
        average += current_value[i];
    }
}
```

```
}
/*Trả về giá trị trung bình của các phần tử trong mảng*/
    return average/100;
}

/*Khai báo và định nghĩa giao diện hàm read để cung cấp thông tin giá trị chuyển
đổi cho user application*/
static ssize_t at91adc_read (struct file *filp, unsigned char
__iomem buf[], size_t bufsize, loff_t *f_pos)
{
/*Khai báo bộ đệm cho hàm read*/
    unsigned int buf_read[1];
/*Gọi hàm hỗ trợ đọc dữ liệu đã được lập trình ở trên*/
    buf_read [0]=at91adc_read_current_value();
/*Gọi hàm truyền dữ liệu sang user application khi có yêu cầu có kiểm tra lỗi trong
quá trình truyền*/
    if (copy_to_user(buf, buf_read, bufsize) != 0) {
        printk ("Error whilst copying to user\n");
        return -1;
    }
    return 2;
}

/*Giao diện hàm ioctl() thực hiện set và clear các chân điều khiển động cơ theo
yêu cầu từ user application*/
static int
at91adc_ioctl(struct inode * inode, struct file * file, unsigned
int cmd, unsigned long *arg)
{
    int retval;
    switch (cmd)
    {
/*Trong trường hợp lệnh set chân điều khiển động cơ lên mức 1*/
        case AT91ADC_SET_MOTOR:
            at91adc_set_motor();
            break;
/*Trong trường hợp lệnh set chân điều khiển động cơ xuống mức 0*/
        case AT91ADC_CLEAR_MOTOR:
            at91adc_clear_motor();
            break;
/*Trong trường hợp không có lệnh hỗ trợ trả về mã lỗi*/
        default:
            retval = -EINVAL;
            break;
    }
    return retval;
}
```



```
}
/*Khai báo và định nghĩa giao diện hàm open*/
static int
at91adc_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    /*FIXME: We should really allow multiple applications to open the device
    *at the same time, as long as the apps access different IO pins.
    *The generic gpio-registration functions can be used for that.
    *Two new IOCTLs have to be introduced for that. Need to check userspace
    *compatibility first. --mb */
    if (!atomic_dec_and_test(&at91adc_open_cnt)) {
        atomic_dec(&at91adc_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d already in
        use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}
/*Khai báo và định nghĩa giao diện hàm close*/
static int
at91adc_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&at91adc_open_cnt);
    return 0;
}
struct file_operations at91adc_fops = {
    .read      = at91adc_read,
    .ioctl     = at91adc_ioctl,
    .open      = at91adc_open,
    .release   = at91adc_close,
};
static struct miscdevice at91adc_dev = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = "at91adc_dev",
    .fops       = &at91adc_fops,
};

static int __init
at91adc_mod_init(void)
{
    int ret;
    /*Khởi tạo cấu hình làm việc cho ADC on chip*/
}
```

```
/*Yêu cầu tạo xung cho ADC hoạt động */
at91adc_clk = clk_get(NULL, /*Device pointer - not required.*/
                        "adc_clk"); /*Clock name*/

/*Cho phép nguồn xung hoạt động*/
clk_enable(at91adc_clk);
/*Định vị con trỏ nền adc vào địa chỉ nền vật lý của các thanh ghi điều khiển adc*/
at91adc_base=ioremap_nocache(AT91SAM9260_BASE_ADC, /*Physical address*/
64); /*Number of bytes to be mapped*/
/*Kiểm tra lỗi trong quá trình định vị*/
if (at91adc_base == NULL)
{
    printk(KERN_INFO "at91adc: ADC memory mapping
failed\n");
    ret = -EACCES;
    goto exit_3;
}

/*Khởi tạo chân gpio ngõ vào analog có điện trở kéo xuống*/
at91_set_A_periph(AT91_PIN_PC0, 0);
/*Reset ADC bằng cách set bit ADC_SWRST*/
iowrite32(AT91_ADC_SWRST, (at91adc_base + AT91_ADC_CR));
/*Cho phép kênh 0 của bộ chuyển đổi ADC hoạt động*/
iowrite32(AT91_ADC_CH(0), (at91adc_base + AT91_ADC_CHER));
/*Cài đặt các chế độ hoạt động cho ADC*/
/*Tần số cực đại = 5MHz = MCK / ((PRESCAL+1) * 2)
/*PRESCAL = ((MCK / 5MHz) / 2) - 1 = ((100MHz / 5MHz)/2)-1 = 9
/*Thời gian start up cực đại = 15uS = (STARTUP+1)*8/ADC_CLOCK
/*STARTUP = ((15uS*ADC_CLOCK)/8)-1 = ((15uS*5MHz)/8)-1 = 9
/*Minimum hold time = 1.2uS = (SHTIM+1)/ADC_CLOCK
/*SHTIM = (1.2uS*ADC_CLOCK)-1 = (1.2uS*5MHz)-1 = 5, Use 9 to
/*ensure 2uS hold time.
/*Enable sleep mode and hardware trigger from TIOA output from TC0.*/
iowrite32((AT91_ADC_SHTIM_(9)|AT91_ADC_STARTUP_(9)|
AT91_ADC_PRESCAL_(9)|AT91_ADC_SLEEP|AT91_ADC_TRGEN),
(at91adc_base + AT91_ADC_MR));
/*Khởi tạo chân gpio theo chế độ ngõ ra*/
gpio_request(GPIO_MOTOR, NULL);
at91_set_GPIO_periph (GPIO_MOTOR, 1);
gpio_direction_output(GPIO_MOTOR, 0);
printk(KERN_INFO "at91adc: Loaded module\n");
printk(KERN_ALERT "Welcome to our at91adc world\n");
return misc_register(&at91adc_dev);
exit_3:
```

```
        clk_disable(at91adc_clk);
        return ret;
}
/*Khai báo và định nghĩa hàm exit thực hiện khi tháo driver ra khỏi hệ thống*/
static void __exit
at91adc_mod_exit(void)
{
    /*Vô hiệu hóa hoạt động của xung clock*/
    clk_disable(at91adc_clk);
    /*Giải phóng con trỏ thanh ghi nền điều khiển ADC*/
    iounmap(at91adc_base);
    /*In ra thông báo cho người dùng*/
    printk(KERN_ALERT "Goodbye for all best\n");
    printk(KERN_INFO "at91adc: Unloaded module\n");
    misc_deregister(&at91adc_dev);
}
module_init (at91adc_mod_init);
module_exit (at91adc_mod_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Coolwarmboy / OpenWrt");
MODULE_DESCRIPTION("Character device for for generic at91adc driver");
```

➤ **Driver LED 7 Đoạn:** Có tên là led\_seg\_dev.ko

*/\*Driver điều khiển led 7 đoạn đã được chúng tôi giải thích kỹ trong bài thực hành điều khiển LED 7 đoạn, ở đây chỉ thay đổi các chân gpio và cấu trúc chương trình cho phù hợp với dự án\*/*

```
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/init.h>
#include <mach/at91_tc.h>
#include <asm/gpio.h>
#include <asm/atomic.h>
#include <linux/genhd.h>
#include <linux/miscdevice.h>
#include <asm/uaccess.h>
#include <linux/interrupt.h>
#include <linux/clock.h>
#include <linux/irq.h>
#include <linux/time.h>
#include <linux/jiffies.h>
#include <linux/sched.h>
#include <linux/delay.h>

#define DRVNAME      "led_seg_dev"
#define DEVNAME      "led_seg"

/*-----Port Control-----*/
```

```
#define P00          AT91_PIN_PB0
#define P01          AT91_PIN_PB2
#define P02          AT91_PIN_PC5
#define P03          AT91_PIN_PC6
#define P04          AT91_PIN_PC7
#define P05          AT91_PIN_PC4
#define P06          AT91_PIN_PB3
#define P07          AT91_PIN_PB1

#define A            AT91_PIN_PB7
#define B            AT91_PIN_PB9
#define C            AT91_PIN_PB11
#define D            AT91_PIN_PA24
#define E            AT91_PIN_PB16
#define F            AT91_PIN_PA23
#define G            AT91_PIN_PB10

/*Basic commands*/
#define SET_P00()      gpio_set_value(P00,1)
#define SET_P01()      gpio_set_value(P01,1)
#define SET_P02()      gpio_set_value(P02,1)
#define SET_P03()      gpio_set_value(P03,1)
#define SET_P04()      gpio_set_value(P04,1)
#define SET_P05()      gpio_set_value(P05,1)
#define SET_P06()      gpio_set_value(P06,1)
#define SET_P07()      gpio_set_value(P07,1)

#define CLEAR_P00()    gpio_set_value(P00,0)
#define CLEAR_P01()    gpio_set_value(P01,0)
#define CLEAR_P02()    gpio_set_value(P02,0)
#define CLEAR_P03()    gpio_set_value(P03,0)
#define CLEAR_P04()    gpio_set_value(P04,0)
#define CLEAR_P05()    gpio_set_value(P05,0)
#define CLEAR_P06()    gpio_set_value(P06,0)
#define CLEAR_P07()    gpio_set_value(P07,0)

#define SET_A()         gpio_set_value(A,1)
#define SET_B()         gpio_set_value(B,1)
#define SET_C()         gpio_set_value(C,1)
#define SET_D()         gpio_set_value(D,1)
#define SET_E()         gpio_set_value(E,1)
#define SET_F()         gpio_set_value(F,1)
#define SET_G()         gpio_set_value(G,1)

#define CLEAR_A()       gpio_set_value(A,0)
#define CLEAR_B()       gpio_set_value(B,0)
#define CLEAR_C()       gpio_set_value(C,0)
#define CLEAR_D()       gpio_set_value(D,0)
```

```
#define CLEAR_E()                gpio_set_value(E,0)
#define CLEAR_F()                gpio_set_value(F,0)
#define CLEAR_G()                gpio_set_value(G,0)

#define CYCLE    1

#define LED_SEG_DEV_MAGIC  'B'
#define LED_SEG_UPDATE    _IO(LED_SEG_DEV_MAGIC, 12)

/*Counter is 1, if the device is not opened and zero (or less) if opened.*/
static atomic_t led_seg_open_cnt = ATOMIC_INIT(1);
unsigned char DataDisplay[8]={0,1,2,3,4,5,6,7};
unsigned char SevSegCode[] = {0xC0, 0xF9, 0xA4, 0xB0, 0x99, 0x92,
0x82,0xF8,0x80,0x90,0x3F,0x77,0xFF};
int i;
/*Khai báo cấu trúc timer “mềm”*/
struct timer_list my_timer;
void led_seg_write_data_active_led(char data)
{
    (data&(1<<0)) ? SET_P00():CLEAR_P00();
    (data&(1<<1)) ? SET_P01():CLEAR_P01();
    (data&(1<<2)) ? SET_P02():CLEAR_P02();
    (data&(1<<3)) ? SET_P03():CLEAR_P03();
    (data&(1<<4)) ? SET_P04():CLEAR_P04();
    (data&(1<<5)) ? SET_P05():CLEAR_P05();
    (data&(1<<6)) ? SET_P06():CLEAR_P06();
    (data&(1<<7)) ? SET_P07():CLEAR_P07();
}
void led_seg_write_data_led(char data)
{
    (data&(1<<0)) ? SET_A():CLEAR_A();
    (data&(1<<1)) ? SET_B():CLEAR_B();
    (data&(1<<2)) ? SET_C():CLEAR_C();
    (data&(1<<3)) ? SET_D():CLEAR_D();
    (data&(1<<4)) ? SET_E():CLEAR_E();
    (data&(1<<5)) ? SET_F():CLEAR_F();
    (data&(1<<6)) ? SET_G():CLEAR_G();
}
void active_led_choice(char number) {
    led_seg_write_data_active_led(~(1<<(number)));
}
void data_led_stransmitt (char data) {
    led_seg_write_data_led (SevSegCode[data]);
}
void sweep_led_time_display(int hh, int mm, int ss) {
    DataDisplay[0] = ss%10;
```

```
DataDisplay[1] = ss/10;
DataDisplay[2] = 10;
DataDisplay[3] = mm%10;
DataDisplay[4] = mm/10;
DataDisplay[5] = 10;
DataDisplay[6] = hh%10;
DataDisplay[7] = hh/10;
}
/*Hàm thực thi quét LED khi có ngắt xảy ra*/
void my_timer_function (unsigned long data) {
    /*Xuất mã 7 đoạn thứ của dữ liệu thứ i ra LED 7 đoạn thứ i*/
    data_led_stransmitt(DataDisplay[i]);
    /*Cho phép LED 7 đoạn thứ i tích cực*/
    active_led_choice(i);
    /*Tăng biến i lên 1 đơn vị*/
    i++;
    /*Giới hạn số LED hiển thị là 8*/
    if (i==8) i = 0;
    /*Cài đặt lại thời gian ngắt cho timer là CYCLE=1(ms)*/
    mod_timer (&my_timer, jiffies + CYCLE);
}
/*buf[0] nhiệt độ giới hạn trên; buf[1] nhiệt độ giới hạn dưới*/
static ssize_t led_seg_write (struct file *filp, unsigned char
__iomem buf[], size_t bufsize, loff_t *f_pos)
{
    unsigned char write_buf[2];
    int write_size = 0;
    int i;
    if (copy_from_user (write_buf, buf, bufsize) != 0) {
        return -EFAULT;
    } else {
        write_size = bufsize;
    }
    /*Cập nhật hiển thị LED*/
    DataDisplay[0] = write_buf[0] % 10;
    DataDisplay[1] = write_buf[0] / 10;
    DataDisplay[2] = 12; /*Ký tự khoảng trắng trong LED 7 đoạn*/
    DataDisplay[5] = 12; /*Ký tự khoảng trắng trong LED 7 đoạn*/
    DataDisplay[6] = write_buf[1] % 10;
    DataDisplay[7] = write_buf[1] / 10;

    return write_size;
}
/*Khai báo và định nghĩa giao diện hàm ioctl() phục vụ cho hàm cập nhật dữ liệu
hiện tại hiển thị trên led*/
static int
```

```
led_seg_ioctl(struct inode * inode, struct file * file, unsigned
int cmd, unsigned int arg)
{
    int retval;
    switch (cmd)
    {
        /*Trong trường hợp lệnh update dữ liệu hiển thị trên led 7 đoạn*/
        case LED_SEG_UPDATE:
            /*Giải mã số nguyên từ 00 đến 99 sang 2 số BCD chục và đơn vị, cập nhật thông
tin cho mảng dữ liệu hiển thị*/
                DataDisplay[3] = arg % 10;
                DataDisplay[4] = arg / 10;
            break;
        default:
            retval = -EINVAL;
            break;
    }
    return retval;
}

static int
led_seg_open(struct inode *inode, struct file *file)
{
    int result = 0;
    unsigned int dev_minor = MINOR(inode->i_rdev);
    if (!atomic_dec_and_test(&led_seg_open_cnt)) {
        atomic_inc(&led_seg_open_cnt);
        printk(KERN_ERR DRVNAME ": Device with minor ID %d already in
use\n", dev_minor);
        result = -EBUSY;
        goto out;
    }
out:
    return result;
}

static int
led_seg_close(struct inode * inode, struct file * file)
{
    smp_mb__before_atomic_inc();
    atomic_inc(&led_seg_open_cnt);

    return 0;
}

struct file_operations led_seg_fops = {
    .write = led_seg_write,
    .ioctl = led_seg_ioctl,
```

```
.open      = led_seg_open,
.release   = led_seg_close,
};

static struct miscdevice led_seg_dev = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = "led_seg_dev",
    .fops       = &led_seg_fops,
};

static int __init
led_seg_mod_init(void)
{
    int ret=0;
    gpio_request (P00, NULL);
    gpio_request (P01, NULL);
    gpio_request (P02, NULL);
    gpio_request (P03, NULL);
    gpio_request (P04, NULL);
    gpio_request (P05, NULL);
    gpio_request (P06, NULL);
    gpio_request (P07, NULL);

    at91_set_GPIO_periph (P00, 1);
    at91_set_GPIO_periph (P01, 1);
    at91_set_GPIO_periph (P02, 1);
    at91_set_GPIO_periph (P03, 1);
    at91_set_GPIO_periph (P04, 1);
    at91_set_GPIO_periph (P05, 1);
    at91_set_GPIO_periph (P06, 1);
    at91_set_GPIO_periph (P07, 1);

    gpio_direction_output(P00, 0);
    gpio_direction_output(P01, 0);
    gpio_direction_output(P02, 0);
    gpio_direction_output(P03, 0);
    gpio_direction_output(P04, 0);
    gpio_direction_output(P05, 0);
    gpio_direction_output(P06, 0);
    gpio_direction_output(P07, 0);

    gpio_request (A, NULL);
    gpio_request (B, NULL);
    gpio_request (C, NULL);
    gpio_request (D, NULL);
    gpio_request (E, NULL);
    gpio_request (F, NULL);
    gpio_request (G, NULL);
}
```



```
at91_set_GPIO_periph (A, 1);
at91_set_GPIO_periph (B, 1);
at91_set_GPIO_periph (C, 1);
at91_set_GPIO_periph (D, 1);
at91_set_GPIO_periph (E, 1);
at91_set_GPIO_periph (F, 1);
at91_set_GPIO_periph (G, 1);

gpio_direction_output(A, 0);
gpio_direction_output(B, 0);
gpio_direction_output(C, 0);
gpio_direction_output(D, 0);
gpio_direction_output(E, 0);
gpio_direction_output(F, 0);
gpio_direction_output(G, 0);
/*Khởi tạo timer "mềm" ngắt với chu kỳ thời gian là 1ms*/
/*Khởi tạo timer từ cấu trúc timer đã định nghĩa*/
init_timer (&my_timer);
/*Cài đặt thời gian sinh ra ngắt là CYCLE=1 (ms)*/
my_timer.expires = jiffies + CYCLE;
/*Cung cấp dữ liệu cho hàm phục vụ ngắt*/
my_timer.data = 0;
/*Gán con trỏ hàm phục vụ ngắt cho timer khi có ngắt xảy ra*/
my_timer.function = my_timer_function;
/*Thực hiện cài đặt timer vào hệ thống*/
add_timer (&my_timer);

misc_register(&led_seg_dev);
printk(KERN_INFO "led_seg: Loaded module\n");
return ret;
}

static void __exit
led_seg_mod_exit(void)
{
/*Xóa đồng bộ timer ra khỏi hệ thống để không làm ảnh hưởng đến các lần khởi
tạo timer tiếp theo*/
del_timer_sync(&my_timer);
misc_deregister(&led_seg_dev);
printk(KERN_INFO "led_seg: Unloaded module\n");
}

module_init (led_seg_mod_init);
module_exit (led_seg_mod_exit);

MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("coolwarmboy");  
MODULE_DESCRIPTION("Character device for for generic gpio api");
```

### **3. Chương trình application:**

```
/*Khai báo thư viện dùng cho các hàm trong chương trình*/  
#include <stdint.h>  
#include <unistd.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include <getopt.h>  
#include <fcntl.h>  
#include <linux/ioctl.h>  
  
#include <sys/types.h>  
#include <sys/stat.h>  
/*Khai báo các số định danh lệnh cần dùng cho driver ADC và LED 7 đoạn*/  
/*Số định danh lệnh dùng cho giao diện hàm ioctl() của driver at91adc_dev*/  
#define AT91ADC_DEV_MAGIC 'B'  
#define AT91ADC_SET_MOTOR _IO(AT91ADC_DEV_MAGIC, 10)  
#define AT91ADC_CLEAR_MOTOR _IO(AT91ADC_DEV_MAGIC, 11)  
/*Số định danh lệnh dùng cho giao diện hàm ioctl() của driver led_seg_dev*/  
#define LED_SEG_DEV_MAGIC 'B'  
#define LED_SEG_UPDATE _IO(LED_SEG_DEV_MAGIC, 12)  
  
/*Khai báo các biến để lưu các nhiệt độ giới hạn nhập từ người dùng*/  
unsigned int HighestTemp, LowestTemp;  
/*Các biến lưu số mô tả tập tin thiết bị khi được mở*/  
int fd_at91adc_dev, fd_led_seg_dev;  
/*Chương trình con cập nhật nhiệt độ hiện tại từ driver adc và truyền sang driver LED 7 đoạn*/  
void check_update_cur_temp (void) {  
/*Bộ nhớ đệm lưu giá trị trả về từ driver adc*/  
    unsigned char current_value[2];  
/*Biến lưu nhiệt độ hiện tại sau khi chuyển đổi*/  
    int cur_temp;  
/*Biến lưu giá trị trả về từ driver dưới dạng int*/  
    int cur_value;  
/*Gọi giao diện hàm read đọc thông tin chuyển đổi từ driver adc*/  
    read(fd_at91adc_dev, current_value, 2);  
/*Chuyển byte thấp và byte cao của bộ đệm thành số int*/  
    cur_value = current_value[1]*256 + current_value[0];  
/*Chuyển đổi thông tin chuyển đổi từ driver adc sang nhiệt độ  
Số 0.31867 được tính bằng công thức sau:  $V_{ref}/(2^{10}-1)$   
trong đó  $V_{ref}$  là giá trị đo được từ chân VREFP; 10 là độ phân giải 10 bits của  
ADC on chip*/
```

```
        cur_temp = cur_value * 0.31867;
/*Gọi giao diện hàm ioctl() của driver LED 7 đoạn cập nhật hiển thị nhiệt độ hiện
tại lên LEDs*/
        ioctl(fd_led_seg_dev, LED_SEG_UPDATE, cur_temp);
/*So sánh nhiệt độ hiện tại với các nhiệt độ giới hạn để tắt|mở động cơ cho thích
hợp*/
        if (cur_temp >= HighestTemp) {
/*Trong trường hợp nhiệt độ hiện tại vượt quá nhiệt độ giới hạn trên*/
            /*In ra thông báo cho người dùng*/
            printf("Motor is turned on\n");
            /*Gọi hàm ioctl() của driver adc set pin điều khiển
motor*/
            ioctl(fd_at91adc_dev, AT91ADC_SET_MOTOR);
        } else if (cur_temp <= LowestTemp) {
            /*Trong trường hợp nhiệt độ hiện tại xuống thấp dưới nhiệt độ giới
hạn dưới*/
            printf("Motor is turned off\n");
            /*Gọi giao diện hàm ioctl() clear pin điều khiển motor*/
            ioctl(fd_at91adc_dev, AT91ADC_CLEAR_MOTOR);
        }
    }
}
/*Chương trình chính khai báo dưới dạng có tham số nhập vào từ người dùng */
int
main(int argc, char **argv)
{
    /*Bộ đệm lưu giá trị cần truyền sang driver LED 7 đoạn hiển thị hai nhiệt
độ giới hạn trên và dưới */
    /*buf[0] highest temp; buf[1] lowest temp*/
    unsigned char buf_write[2] = {34, 35};
    /*Mở hai tập tin thiết bị trước khi thao tác*/
    fd_at91adc_dev = open("/dev/at91adc_dev", O_RDWR);
    fd_led_seg_dev = open("/dev/led_seg_dev", O_RDWR);
    /*Cập nhật các thông tin về nhiệt độ giới hạn*/
    HighestTemp = atoi(argv[2]);          /*Highest          Temperature
value*/
    LowestTemp = atoi(argv[1]);          /*Lowest Temperature value*/
    /*Lưu các giá trị giới hạn vào bộ đệm ghi sang driver hiển thị LED 7
đoạn*/
    buf_write[0] = HighestTemp;
    buf_write[1] = LowestTemp;
    /*Gọi giao diện hàm write() ghi dữ liệu trong bộ đệm sang driver LED 7
đoạn*/
    write(fd_led_seg_dev, buf_write, 2);
    /*Dùng vòng lặp liên tục cập nhật điều khiển động cơ, hiển thị nhiệt độ
ra LED với chu kỳ là 1s*/
}
```

```
while (1) {  
    check_update_cur_temp();  
    sleep(1);  
}  
exit (0);  
}
```

### **III. Kết luận và bài tập:**

#### ***a. Kết luận:***

Bên cạnh dùng IC 0809 làm thiết bị chuyển đổi tương tự-số chúng ta cũng có thể dùng module ADC tích hợp sẵn trong CHIP vi điều khiển. Như vậy sẽ tiết kiệm chi phí và rút gọn được số chân IO điều khiển để dùng vào công việc khác. Bên cạnh đó dùng ADC on chip chúng ta có thể thực hiện được nhiều chức năng tiện lợi hơn, độ chính xác có thể linh động thay đổi tùy theo yêu cầu chương trình ứng dụng.

Trong bài này chúng ta đã sửa lại driver hiển thị LED 7 đoạn ứng dụng timer mềm để quét LED đồng thời đã ứng dụng phương pháp lập trình character device driver vào viết driver cho ADC on chip. Kết hợp 2 driver để điều khiển và hiển thị thông tin trên LED 7 đoạn. Người học có thể thay đổi dự án hiển thị trên những thiết bị khác hay ứng dụng driver vào việc lấy mẫu tín hiệu, ... lúc đó ứng dụng sẽ phức tạp hơn. Do thời gian thực hiện cuốn giáo trình này có hạn nên chúng tôi không đi sâu vào nghiên cứu những ứng dụng này.

#### ***b. Bài tập:***

1. Làm lại yêu cầu của dự án trên, nhưng hiển thị các thông tin các nhiệt độ lên LCD. (Áp dụng LCD driver đã viết trong bài thực hành LCD);
2. Tối ưu hóa dự án trên theo yêu cầu sau:
  - Thông tin về các nhiệt độ giới hạn được lưu trong EEPROM 24C08;
  - Khi chương trình được mở, nó sẽ cập nhật hai giá trị nhiệt độ giới hạn từ EEPROM;
  - Thông tin về các giá trị nhiệt độ nếu thay đổi sẽ được cập nhật vào EEPROM;