

CHƯƠNG 1: GIỚI THIỆU LẬP TRÌNH HỆ THỐNG NHÚNG KIT ARM MINI2440

1.1 Giới thiệu chung lập trình hệ nhúng:

Hệ thống nhúng vốn rất đa dạng và phong phú, tuy nhiên có rất ít người biết được tầm quan trọng và sự hiện hữu của chúng trong thế giới quanh ta. Từ những hệ thống phức tạp như hàng không vũ trụ, phòng thủ quân sự, máy móc tự động trong công nghiệp, đến những phương tiện di chuyển thông thường như máy bay, xe điện, xe hơi... Những đặc trưng của hệ thống nhúng :

- Hệ thống nhúng (embedded system) được định nghĩa là một hệ thống chuyên dụng, thường có khả năng tự hành và được thiết kế tích hợp vào một hệ thống lớn hơn để thực hiện một chức năng chuyên biệt nào đó.
- Khác với các máy tính đa chức năng (multi-purposes computers), ví dụ như máy vi tính cá nhân (PC), một hệ thống nhúng thường chỉ thực hiện một hoặc một vài chức năng nhất định. Hệ thống nhúng bao gồm cả thiết bị phần cứng và phần mềm, hầu hết đều phải thỏa mãn yêu cầu hoạt động theo thời gian thực (real-time).
- Tùy theo tính chất và yêu cầu, mức độ đáp ứng của hệ thống có thể phải là rất nhanh (ví dụ như hệ thống thăng trong xe hơi hoặc điều khiển thiết bị trong nhà máy), hoặc có thể chấp nhận một mức độ chậm trễ tương đối (ví dụ như điện thoại di động, máy lạnh, ti-vi).

Lập trình ứng dụng trên hệ nhúng phụ thuộc vào nền tảng (platform) phần cứng, phần mềm của hệ nhúng đó.

Hệ nhúng không có hệ điều hành:

- Sử dụng các vi điều khiển có hiệu năng tương đối thấp: 8051, AVR,PIC

-Lập trình bằng C,asm

- Phù hợp với các ứng dụng điều khiển vào/ ra cơ bản, các giao tiếp ngoại vi cơ bản

Hệ nhúng có hệ điều hành:

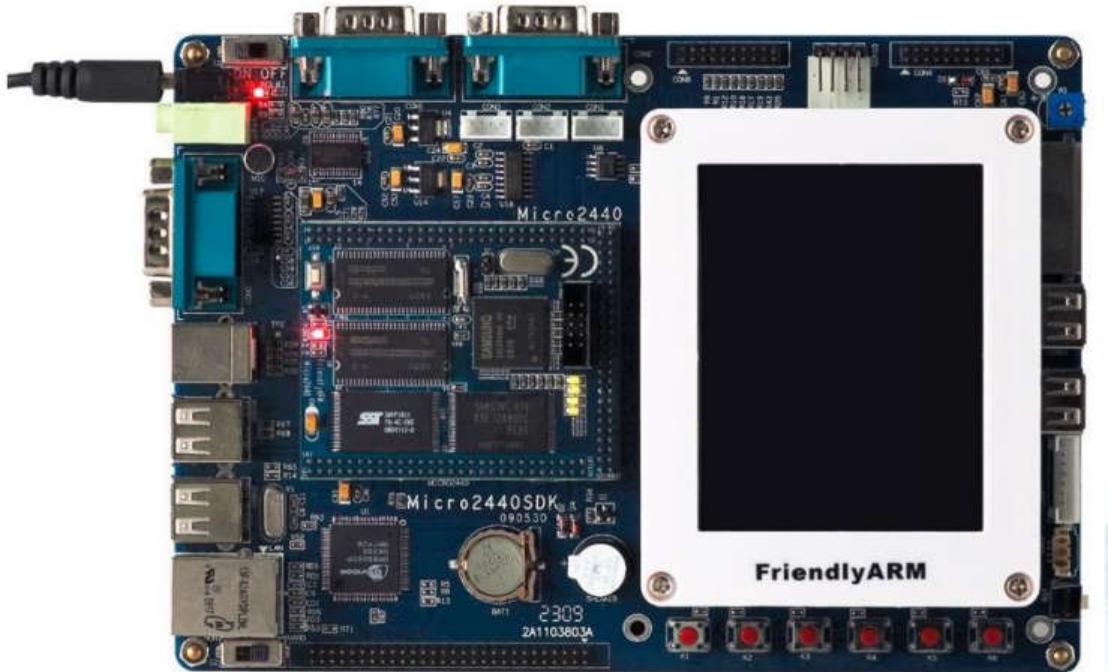
- Sử dụng các vi điều khiển có hiệu năng cao như: AVR 32, ARM 9,ARM 11

-Nền tảng hệ điều hành nhúng: Embedded Linux, Windows CE, Android.

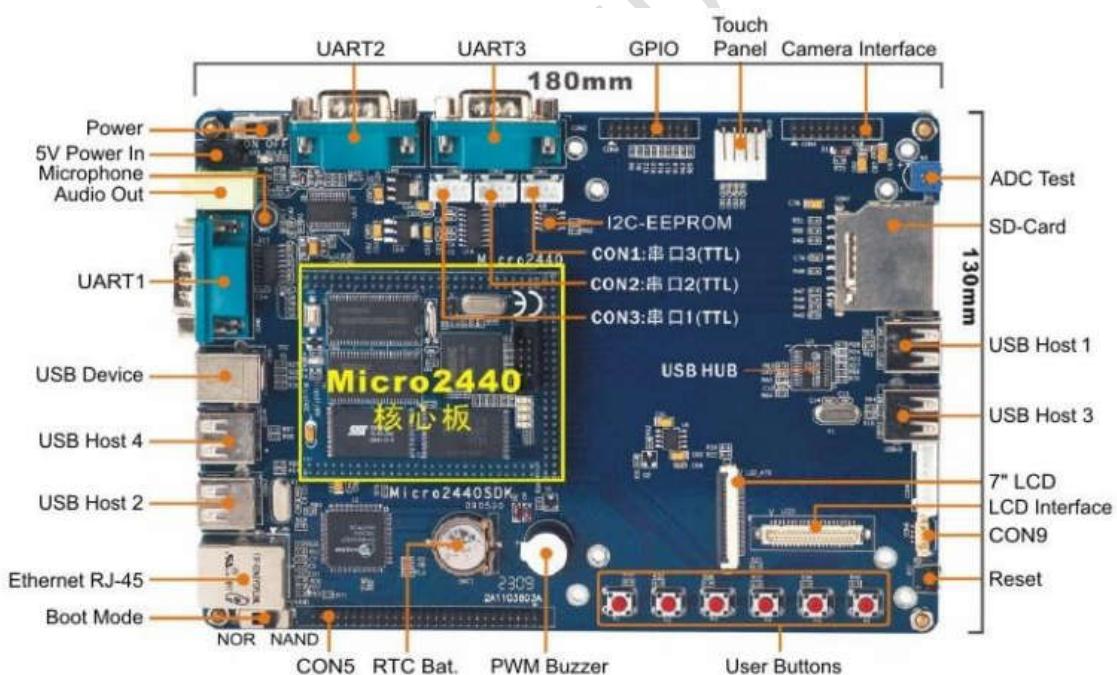
-Ứng dụng nhiều bài toán phức tạp: GPS, xử lý ảnh, ứng dụng client/server

Môn học thực hành này thực hiện trên kit ARM Mini2440 trên nền tảng hệ điều hành Embedded Linux.

1.2 Giới thiệu về kit nhúng mini2440



Hình 1.1



Hình 1.2

Thông số kỹ thuật của kit:

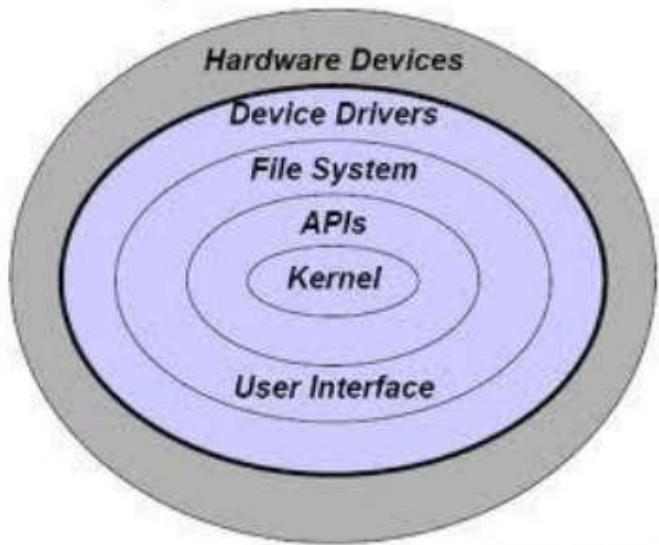
Khối chức năng	Thông số kỹ thuật
CPU	Samsung S3C2440, 400Mhz, max 533Mhz
SDRAM	64M SDRAM

	32 bit Data Bus SDRAM Clock 100Mhz
FLASH	64M hay 128M Nand Flash 2M Nor Flash (được cài đặt sẵn BIOS)
LCD	Màn hình cảm ứng điện trở Tối đa 4096 màu, kích thước 3.5 → 12 inches Độ phân giải: 1024x768
Các ngoại vi kit	1 khối 10/100M Ethernet RJ-45(DM9000) 3 cổng com 1 USB host 1 USB Slave loại B 1 SD card 1 stereo audio out, 1 micro in 1 JTAG 20 pin 4 led đơn 6 nút nhấn (button) 1 buzzer điều khiển sử dụng PWM 1 biến trở chuyển đổi ADC 1 EEPROM giao tiếp chuẩn I2C 1 giao tiếp với cảm biến ảnh(20 chân) 1 pin cho thời gian thực Nguồn 5V
Hệ điều hành	Linux 2.6(đang dùng) Android WinCE 5 hay 6

1.3 Hệ điều hành nhúng Linux:

Tổng quan:

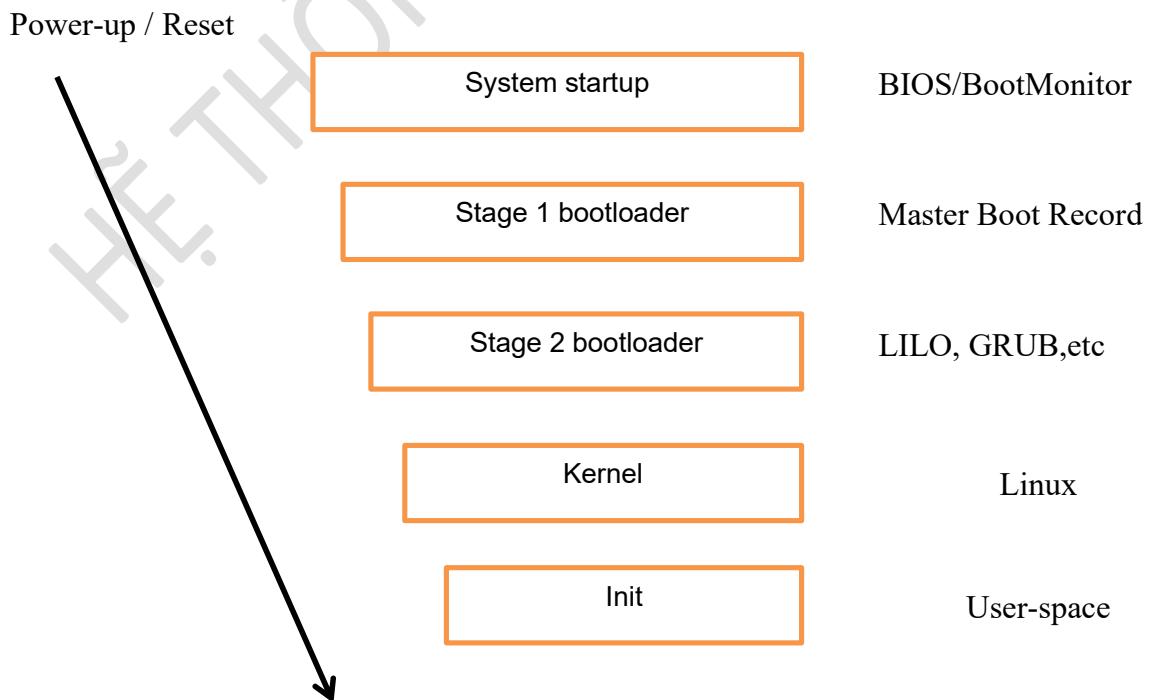
Là hệ điều hành được cài đặt trên hệ thống nhúng của chip ARM của kit. Trong kit ARM 2440 có hỗ trợ Linux, Android, WinCE. Ở phần này ta chỉ khảo sát Linux nhúng. Khác với phần mềm Linux đồ sộ được cài đặt trên PC, phần mềm Linux nhúng chỉ nhỏ gọn được cài đặt trên chip ARM dùng phục vụ việc lập trình hệ thống nhúng. Đặc trưng của hệ điều hành nhúng là có tính tin cậy cao, tính mềm dẻo, dễ dàng nâng cấp để tương thích với hệ thống, đòi hỏi ít bộ nhớ hơn. Có thể hỗ trợ khởi động từ bộ nhớ ROM, Flash và hệ thống không có ổ cứng.

**Hình 1.3**

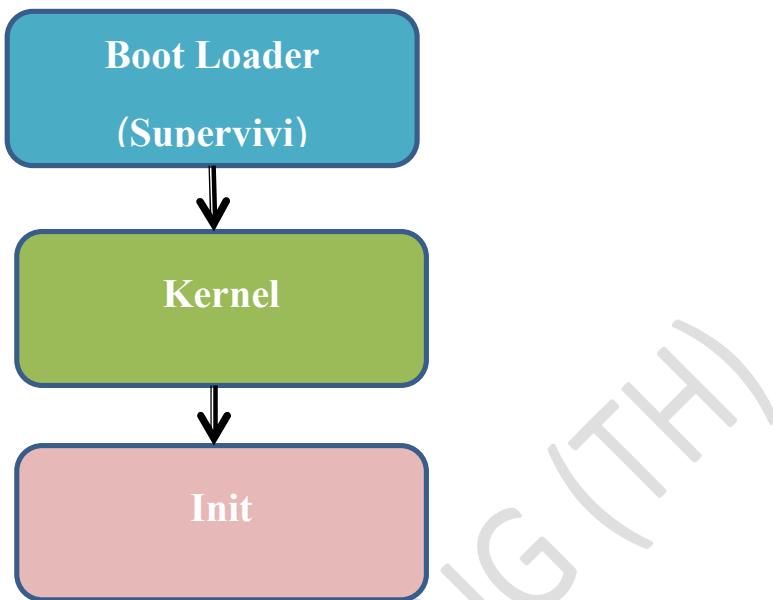
Khi cài đặt linux nhúng thì có hỗ trợ 1 số driver dùng để viết các ứng dụng giao tiếp các ngoại vi như: led, button, adc,pwm.... Người lập trình chỉ cần sử dụng các device file tương tác với các drivers viết các ứng dụng giao tiếp. Ngoài ra nhân hệ điều hành có thể được biên dịch lại để mở rộng các chân port hay để chỉnh sửa hệ thống.

Để thực hành ta cần 1 máy host cài đặt Linux(dùng máy ảo hay thật) và 1 target(kit) cài đặt Linux nhúng, trình biên dịch chéo để biên dịch file hệ thống để nạp xuống kit chạy ứng dụng.

Quá trình boot hệ thống Linux trên PC:



Quá trình boot hệ thống Linux nhúng:



Boot Loader: là chương trình mồi, thực hiện kiểm tra phần cứng hệ thống và nạp nhân (kernel) của hệ điều hành

Kernel: nhân hệ điều hành chứa các thành phần cơ bản nhất của hệ thống file.

Root file system: hệ thống file chứa các modules bổ sung và các phần mềm ứng dụng.

1.4 Cài đặt hệ điều hành linux nhúng và kit ARM mini2440:

Ta có thể cài đặt hệ điều hành linux nhúng từ môi trường windows.

Các phần mềm cần thiết: **Hyper Terminal** dùng để kết nối kit ARM mini2440 qua cổng COM, **DNW** dùng kết nối kit qua cổng USB.

Các file dữ liệu cần thiết: **supervivi-128M**, **zImage_T35**, **rootfs_qtopia_qt4**.

Máy host(PC,laptop) đã cài đặt Linux thật hoặc ảo.

Quá trình cài đặt linux nhúng cho kit ARM trải qua các bước.

Bước 1: Kết nối kit với PC

Kết nối cổng COM0 (UART0) đến cổng COM của máy tính(nếu máy tính không có cổng COM thì dùng sợi dây cáp chuyển đổi USB to COM. Kết nối này cho phép giao tiếp máy tính với phần mềm Bootloader trên kit(chứa sẵn chip NOR cho phép ra lệnh điều khiển)

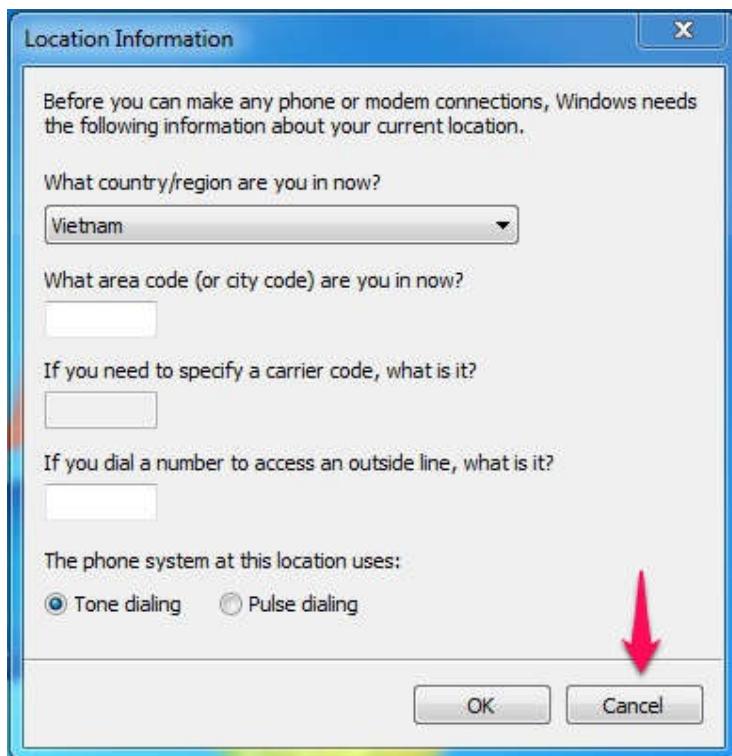
Kết nối kit với PC qua cổng USB(phải cài driver cho máy tính nhận kit Friendly Arm Mini 2440). Kết nối này cho phép truyền file (Bootloader, Image, root file system) từ máy tính lên kit.

Chuyển công tắc gạt S2 (trên kit) sang vị trí NOR Flash để cho phép kit khởi động từ NOR Flash(chế độ khởi động hệ điều hành khi đã cài đặt thành công trên chip NAND Flash khi công tắc chuyển sang vị trí NAND Flash).

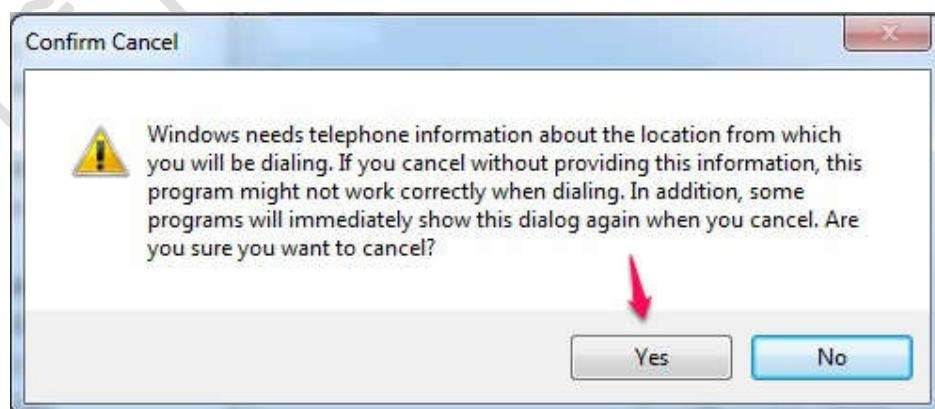
Cấp nguồn điện cho kit từ adapter 5V 2A(chưa bật công tắc nguồn)

Bước 2: Tiến hành cài đặt

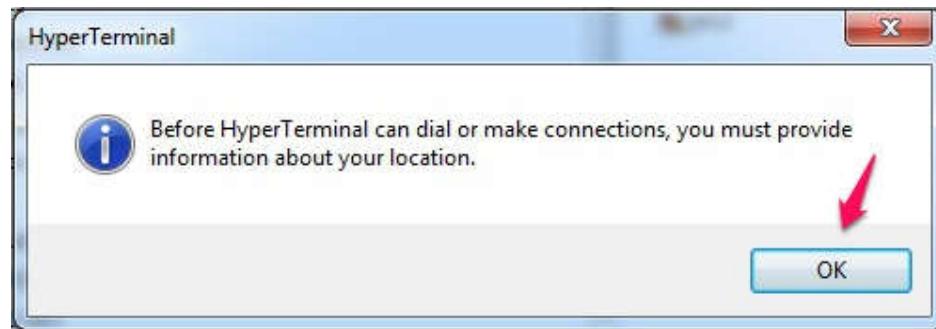
Mở phần mềm Hyper Terminal



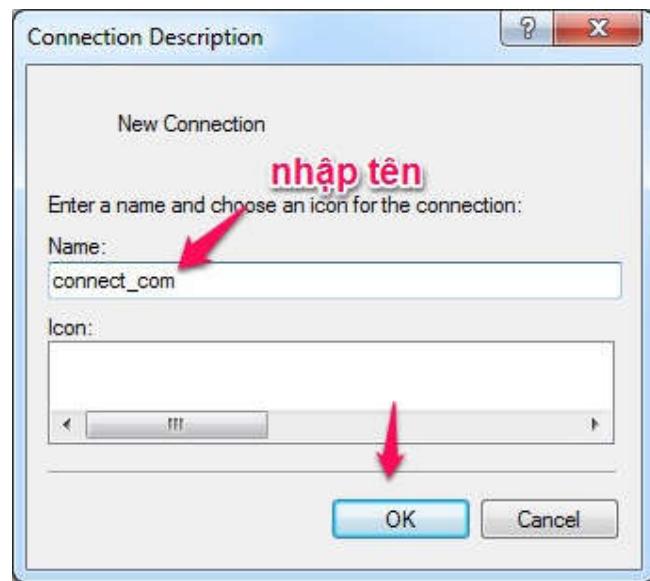
Hình 1. 4



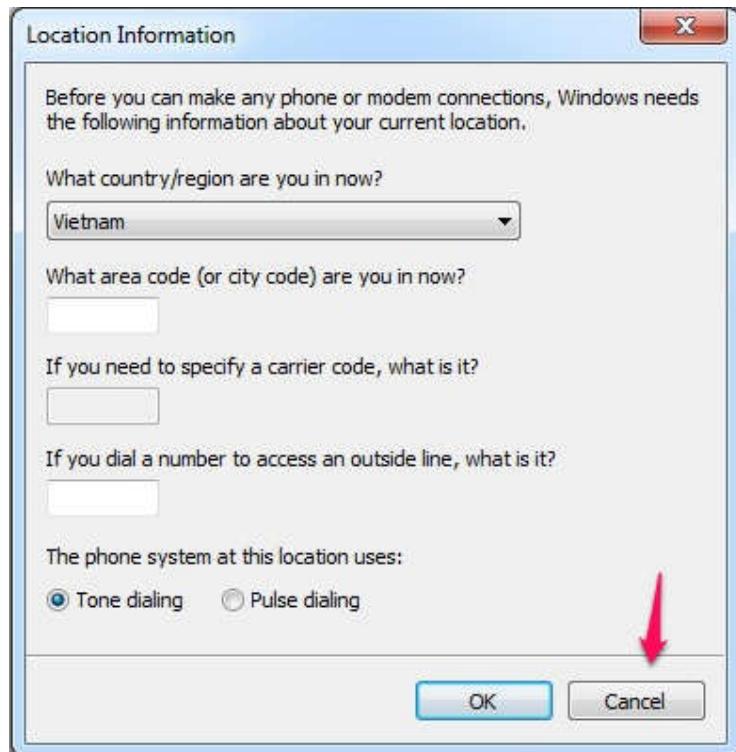
Hình 1. 5



Hình 1. 6



Hình 1. 7



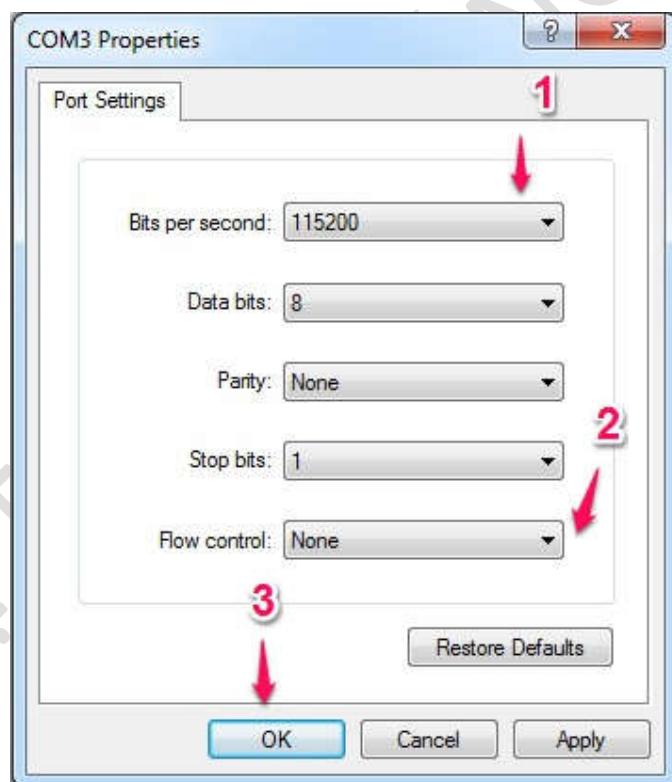
Hình 1.8



Hình 1.9

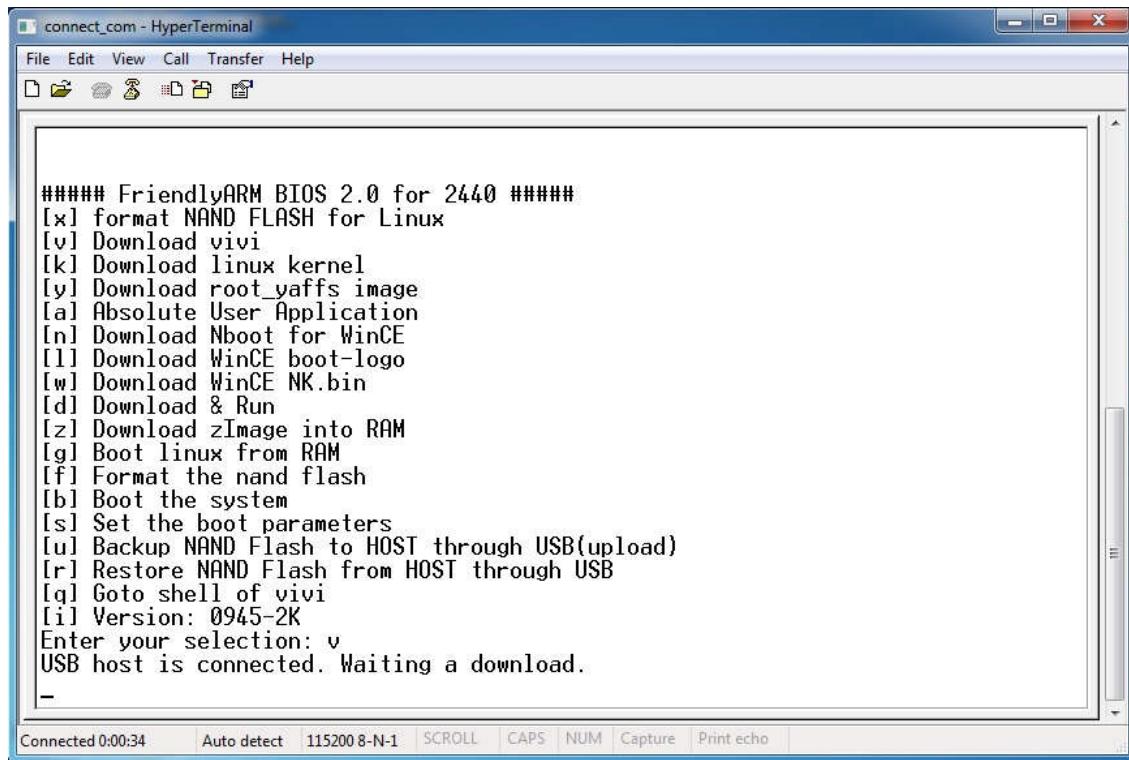


Hình 1. 10



Hình 1. 11

Nhấn phím reset trên kit, màn hình HyperTerminal xuất hiện chọn v để load Supervivi_128:

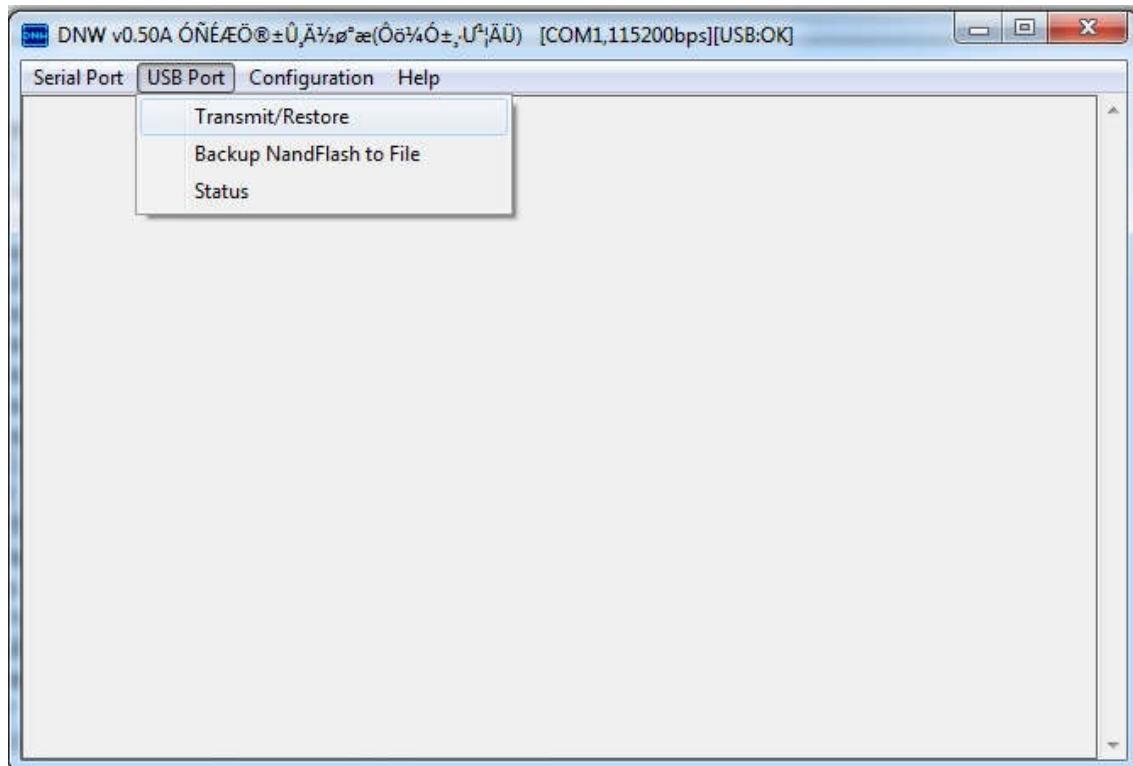


Hình 1.12

Mở phần mềm DNW. Vào:

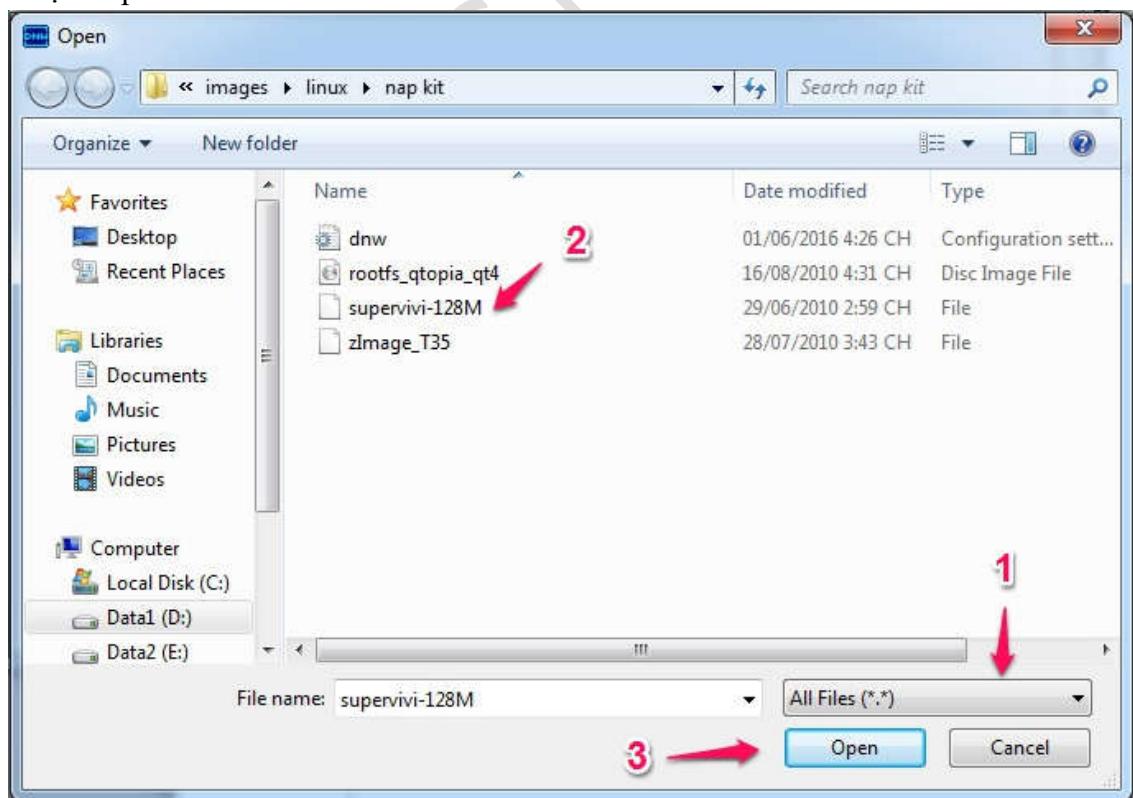
Serial Port-Connect: để kết nối đến COM1 của kit.

USB Port-Transmit/Restore: truyền file qua USB Port



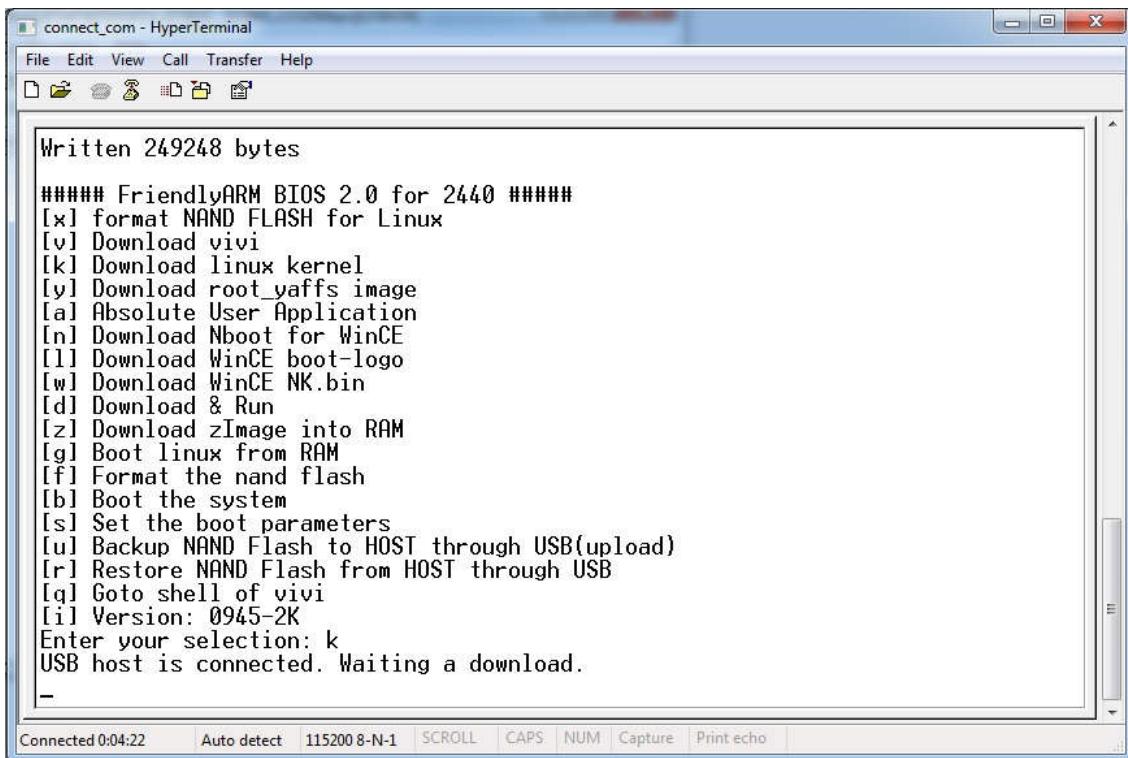
Hình 1.13

Chọn Supervivi-128M:



Hình 1. 14

Mở phần mềm HyperTerminal chọn k để load nhân linux:



The screenshot shows a Windows-style window titled "connect_com - HyperTerminal". The menu bar includes File, Edit, View, Call, Transfer, and Help. Below the menu is a toolbar with icons for copy, paste, cut, find, and others. The main window displays a list of options:

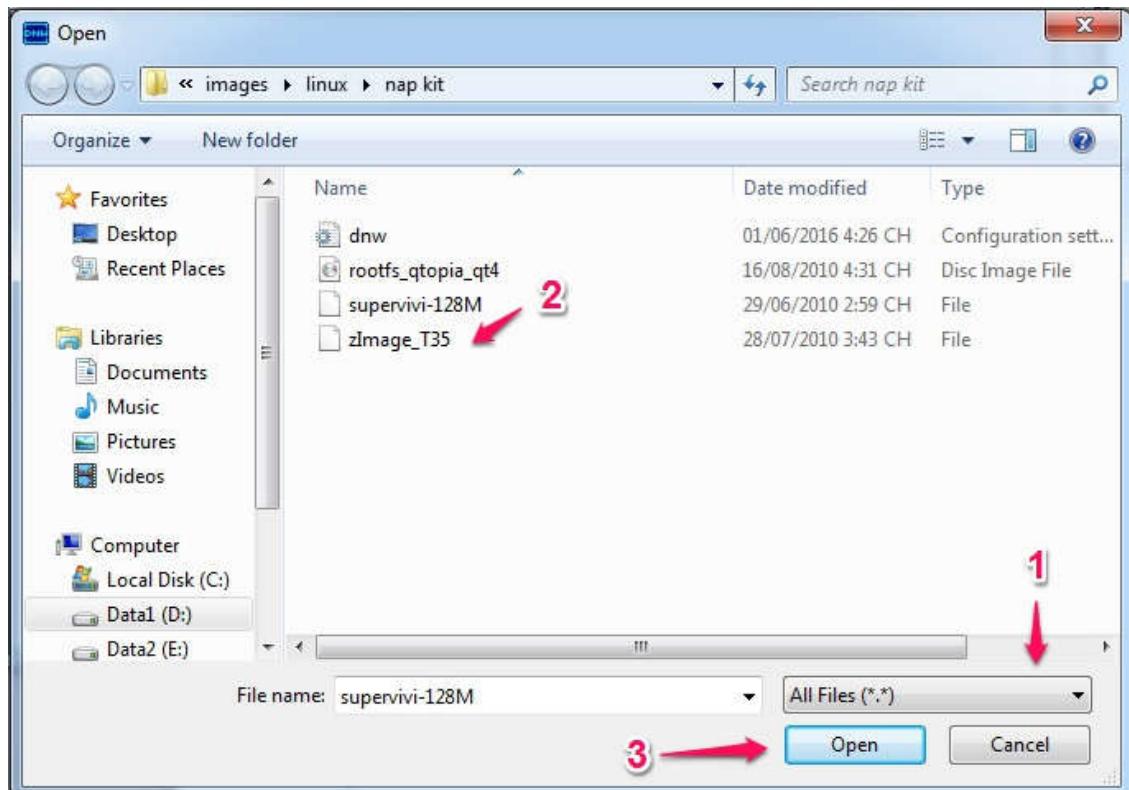
```
Written 249248 bytes
#####
FriendlyARM BIOS 2.0 for 2440 #####
[x] format NAND FLASH for Linux
[v] Download vivi
[k] Download linux kernel
[y] Download root_yaffs image
[a] Absolute User Application
[n] Download Nboot for WinCE
[l] Download WinCE boot-logo
[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 0945-2K
Enter your selection: k
USB host is connected. Waiting a download.
-
```

At the bottom of the window, there is a status bar with the following information: Connected 0:04:22, Auto detect, 115200 8-N-1, SCROLL, CAPS, NUM, Capture, Print echo.

Hình 1. 15

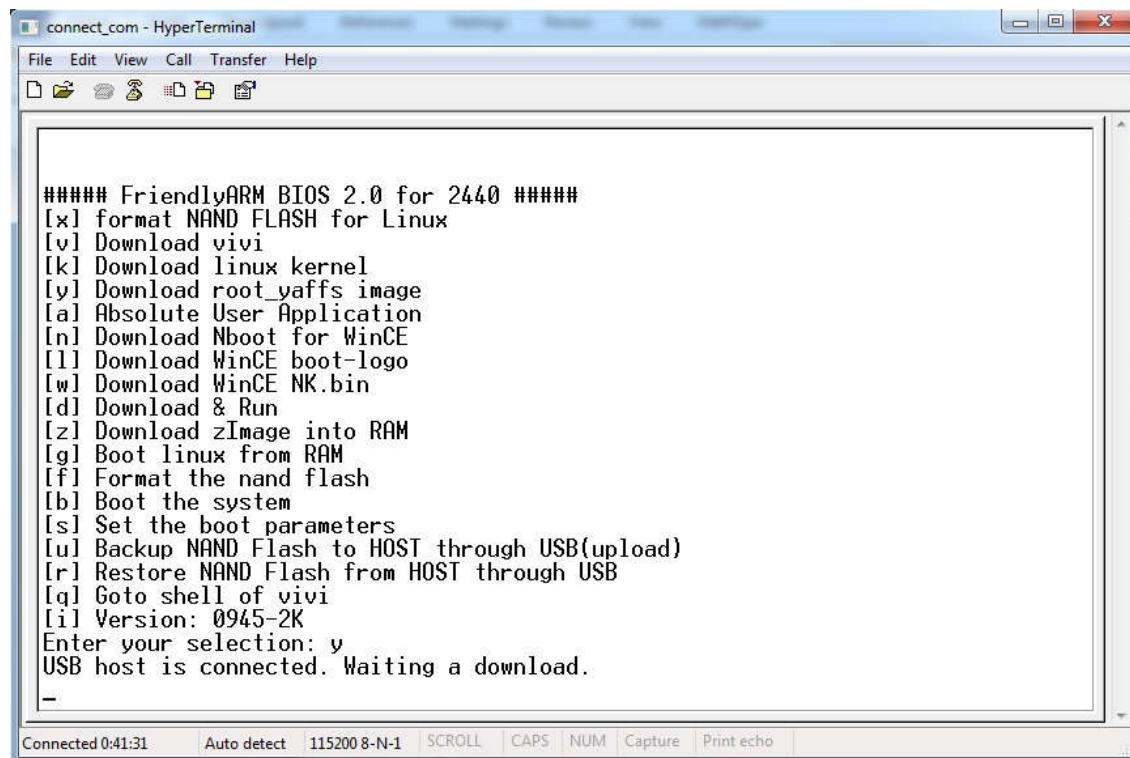
Mở phần mềm DNW. Vào: USB Port-Transmit/Restore:

Chọn zImage_T35



Hình 1. 16

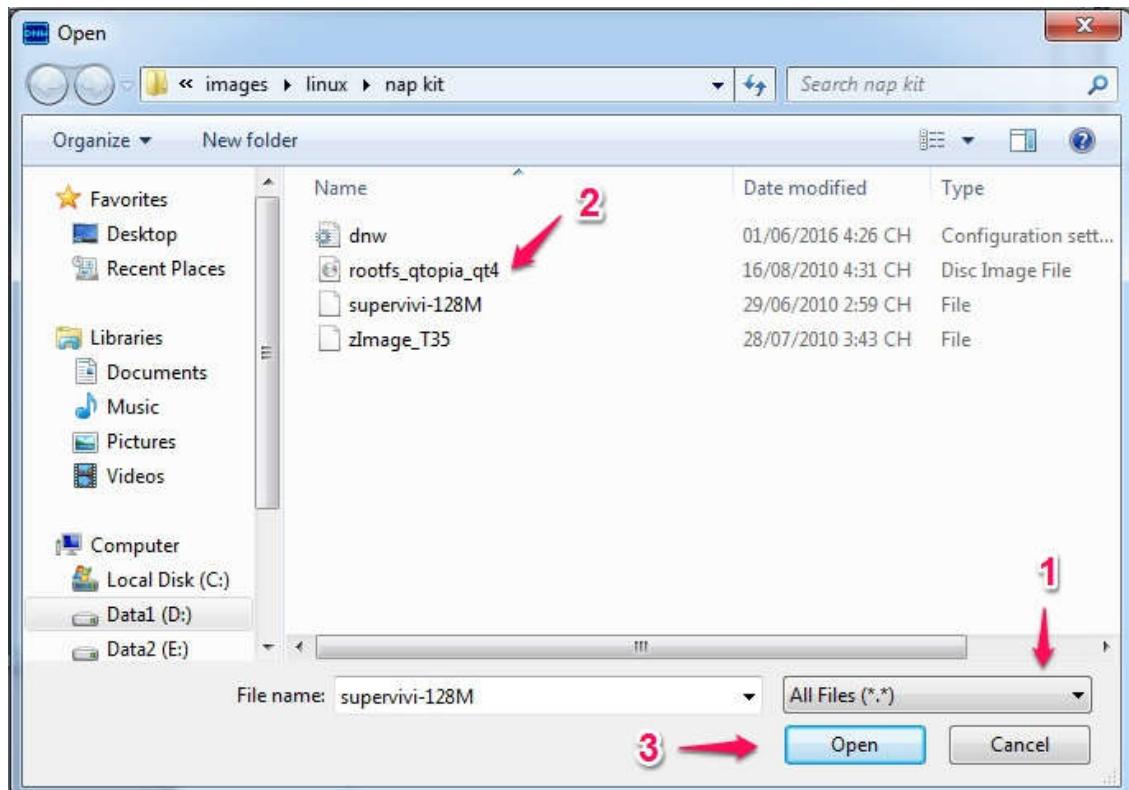
Mở phần mềm HyperTerminal chọn y để load rootfs file:



Hình 1.17

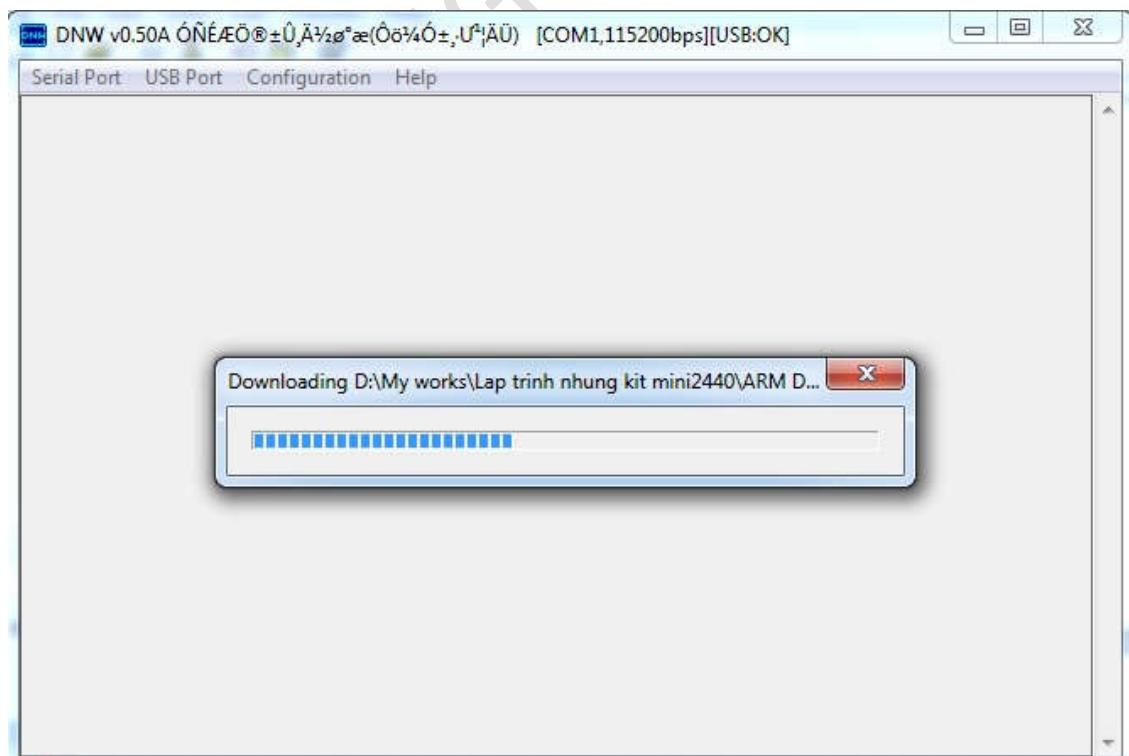
Mở phần mềm DNW. Vào: USB Port-Transmit/Restore:

Chọn rootfs_qtopia_qt4



Hình 1.18

Đợi load khoảng vài phút:



Hình 1. 19

Sau khi cài xong ta gạt công tắc sang NAND, reset lại kit, màn hình hyber Terminal xuất hiện:

```

connect_com - HyperTerminal
File Edit View Call Transfer Help
File Open Save Exit
UDA134X SoC Audio Codec
asoc: UDA134X <-> s3c24xx-i2s mapping ok
ALSA device list:
#0: S3C24XX_UDA134X (UDA134X)
TCP cubic registered
NET: Registered protocol family 17
s3c2410-rtc s3c2410-rtc: hctosys: invalid date/time
yaffs: dev is 32505859 name is "mtblockquote"
yaffs: passed flags ""
yaffs: Attempting MTD mount on 31.3, "mtblockquote"
yaffs: auto selecting yaffs2
block 764 is bad
yaffs_read_super: isCheckpointed 0
VFS: Mounted root (yaffs filesystem) on device 31:3.
Freeing init memory: 156K
[29/Nov/1999:16:00:01 +0000] boa: server version Boa/0.94.13
[29/Nov/1999:16:00:01 +0000] boa: server built Jul 26 2010 at 15:58:29.
[29/Nov/1999:16:00:01 +0000] boa: starting server pid=681, port 80
Try to bring eth0 interface up.....eth0: link down
Done

Please press Enter to activate this console.
[root@FriendlyARM /]#

```

Connected 0:12:17 Auto detect 115200 8-N-1 SCROLL CAPS NUM Capture Print echo

Hình 1. 20

Khi xuất hiện [root@FriendlyARM/]# tức là đang ở Terminal của linux nhúng kit ARM. Mặc định account của kit có user:root, pass: không có

Đổi password của root: #passwd

Ta nhập password mới của root 2 lần để xác định.

Hình 1. 21

1.5 Cài đặt linux máy host và gói phần mềm hỗ trợ việc lập trình kit :

Phần cài đặt máy host Linux ta có thể dùng máy thật hoặc máy ảo. Về cài đặt máy thật chỉ cài trực tiếp vào hệ điều hành chính Linux. Tuy nhiên môi trường làm việc của chúng ta dùng rất nhiều phần mềm trên Windows nên người ta thường dùng máy ảo Linux. Cài đặt hệ điều hành Windows làm hệ điều hành chính và dùng phần mềm tạo máy ảo (Virtual Box hoặc VmWare). Phần cài đặt máy ảo Linux tác giả không trình bày ở đây sinh viên tự tham khảo

1.5.1 Một số gói phần mềm cài đặt trong Ubuntu

➤ Cài đặt trình biên dịch chéo arm-linux-gcc:

Trình biên dịch chéo cho phép biên dịch ứng dụng viết bằng C/C++ chạy trên nền tảng ARM(kit ARM mini2440 với hệ điều hành nhúng linux). Sử dụng chương trình này để biên dịch và nạp xuống kit để thực thi.

Chuẩn bị: file **arm-linux-gcc-4.3.2.tar.gz** có sẵn trên máy linux(host). Các bước thực hiện:

Bước 1:

Chuyển đến thư mục chứa file nén **arm-linux-gcc-4.3.2.tar.gz** và tiến hành giải nén bằng lệnh:

\$ tar -zxf arm-linux-gcc-4.3.2.tar.gz

Kết quả giải nén được thư mục chứa trình biên dịch có đường dẫn như sau:

usr/local/arm/4.3.2/bin

Nên copy thư mục local (chứa trình biên dịch đã giải nén) đặt trong thư mục /usr của File System trên máy tính Linux. Vị trí để trình biên dịch chéo là không bắt buộc, tuy nhiên thường chứa trên thư mục /usr để thuận tiện cho các công việc tiếp theo. Hiện tại các bạn đang sử dụng user bình thường (đang sử dụng user: embedded) thì sẽ không thể paste vào thư mục hệ thống usr, mặc định chỉ được sử dụng với user root. Để có thể copy và paste trực tiếp thư mục **FriendlyARM** trên user đang sử dụng các bạn dùng lệnh sau:

\$sudo chown -R embedded:embedded /usr

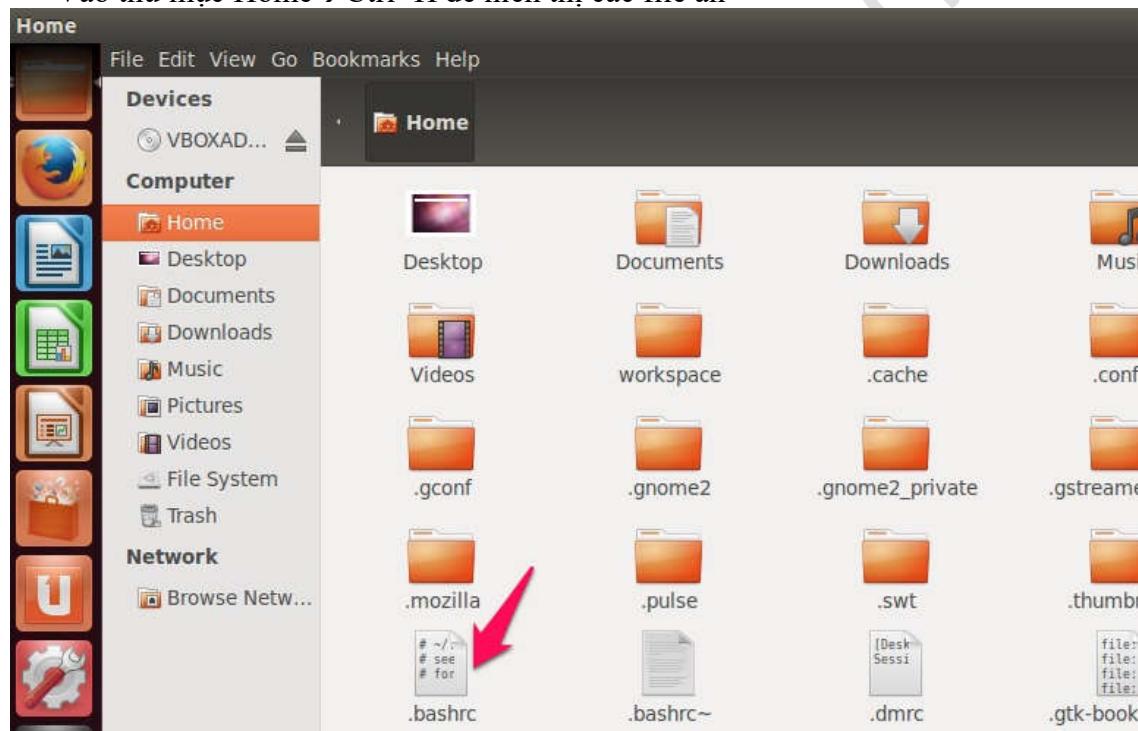
với **embedded** (tên user), **/usr** (đường dẫn tới thư mục cần thay đổi quyền truy cập).

Bước 2:

Cập nhật biến môi trường PATH cho trình biên dịch chéo để trình biên dịch chéo có thể sử dụng mọi lúc mọi nơi tức là gọi lệnh trên các cửa sổ lệnh mà không cần chuyển đến thư mục chứa.

Thực hiện:

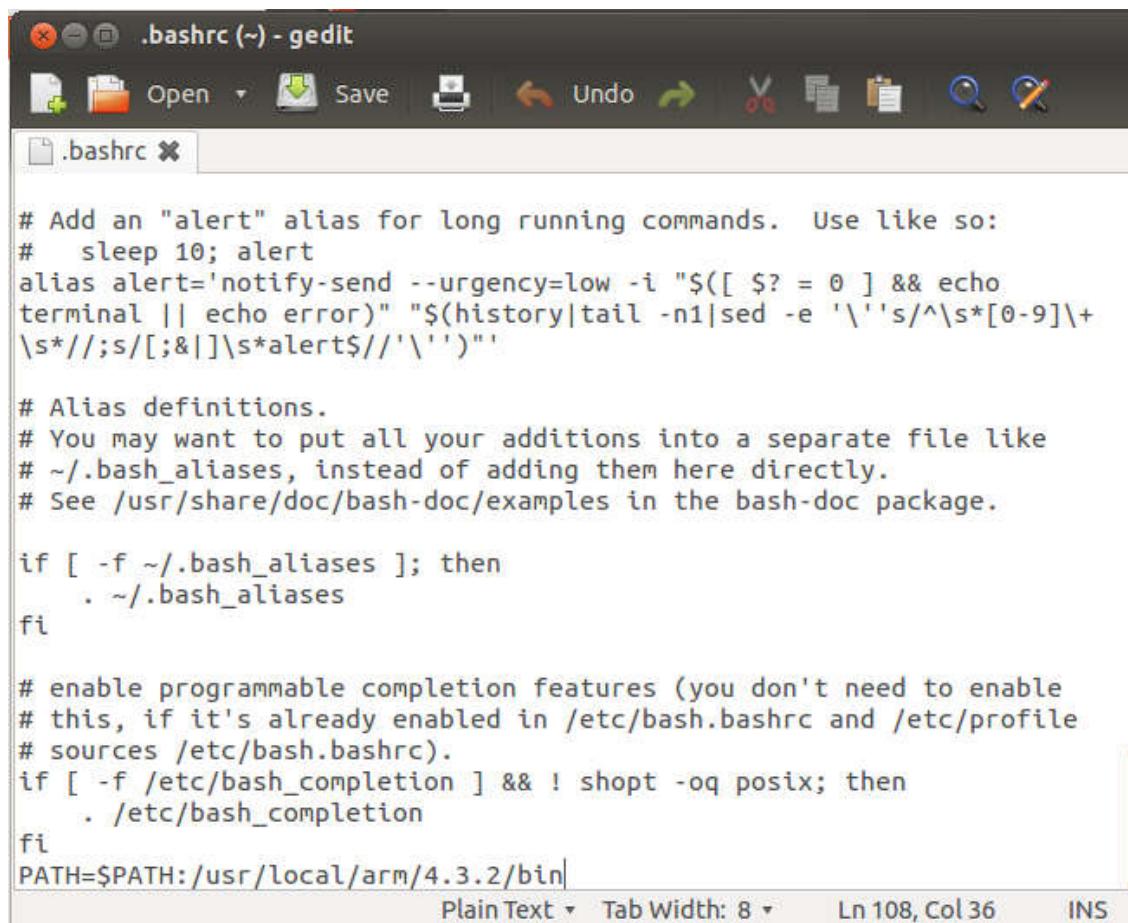
Vào thư mục Home → Ctrl+H để hiển thị các file ẩn



Hình 1. 22

Mở file .bashrc ra chèn đường dẫn vào cuối file .bashrc:

Cấu trúc: **PATH=\$PATH:[Đường dẫn tới thư mục bin của arm-linux-gcc]**
→PATH=\$PATH:/usr/local/arm/4.3.2/bin



```

.bashrc (~) - gedit

.bashrc

# Add an "alert" alias for long running commands. Use like so:
# sleep 10; alert
alias alert='notify-send --urgency=low -i "$( [ $? = 0 ] && echo terminal || echo error)" "$(history|tail -n1|sed -e '\''s/^\s*/\n/g; s/[;&]\s*alert$/'\'')"

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if [ -f /etc/bash_completion ] && ! shopt -oq posix; then
    . /etc/bash_completion
fi
PATH=$PATH:/usr/local/arm/4.3.2/bin

```

Plain Text ▾ Tab Width: 8 ▾ Ln 108, Col 36 INS

Hình 1. 23**Bước 3:**

Cài đặt 1 số thư viện 32 bit:

\$ sudo apt-get install ia32-libs

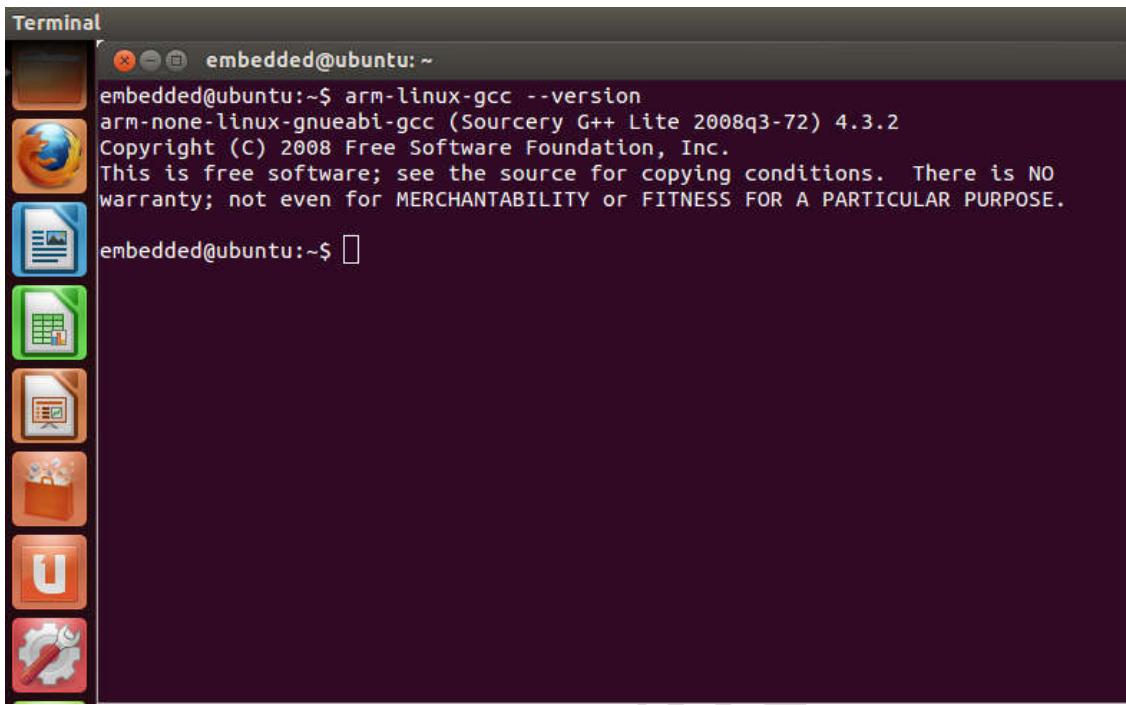
Việc làm này tốn thời gian khoảng mười mấy phút

Bước 4:

Kiểm tra việc cài đặt có thành công không?

\$ arm-linux-gcc --version

Khi xuất hiện dòng chữ như hình dưới là cài đặt thành công



Hình 1. 24

- Cài đặt Eclipse CDT:

Gói phần mềm này giúp ta việc ứng dụng C trên linux, biên dịch chéo chạy debug được dễ dàng hơn.

Thực hiện:

\$sudo apt-get install eclipse eclipse-cdt g++

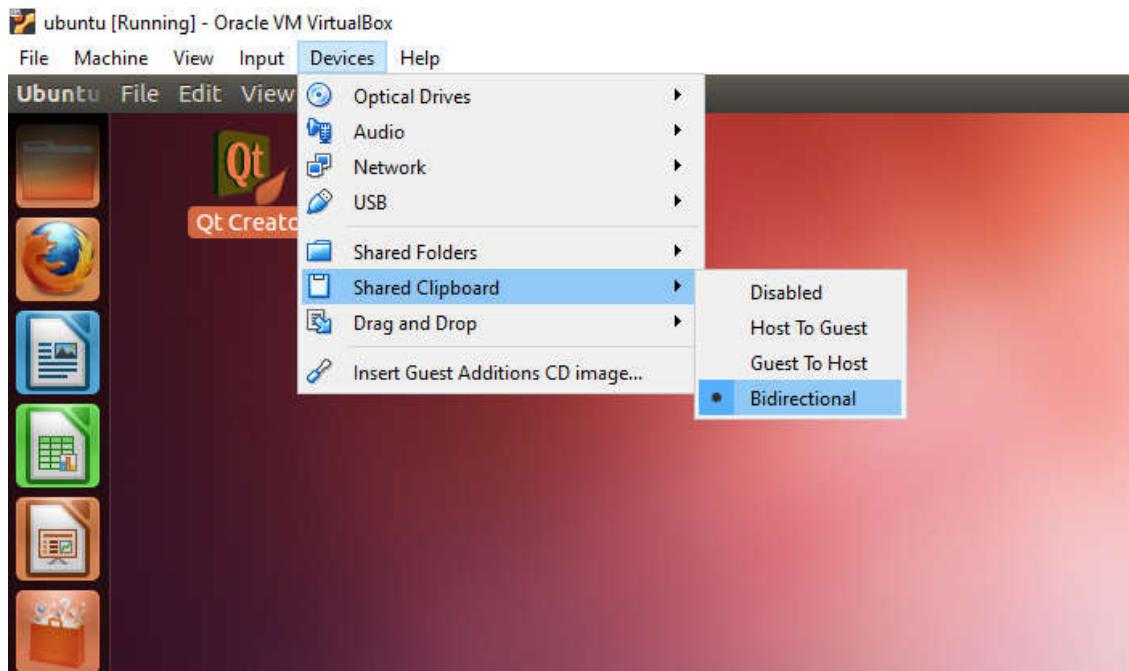
Việc làm này tốn thời gian khoảng vài chục phút

1.5.2 Chia sẻ clipboard, thư mục, drag and drop giữa máy thật và máy ảo trong VirtualBox:

Việc lập trình nhúng ta cần chuyển dữ liệu từ môi trường windows sang linux liên tục. Do đó việc chia sẻ dữ liệu giữa máy thật và máy ảo trong VirtualBox là rất cần thiết.

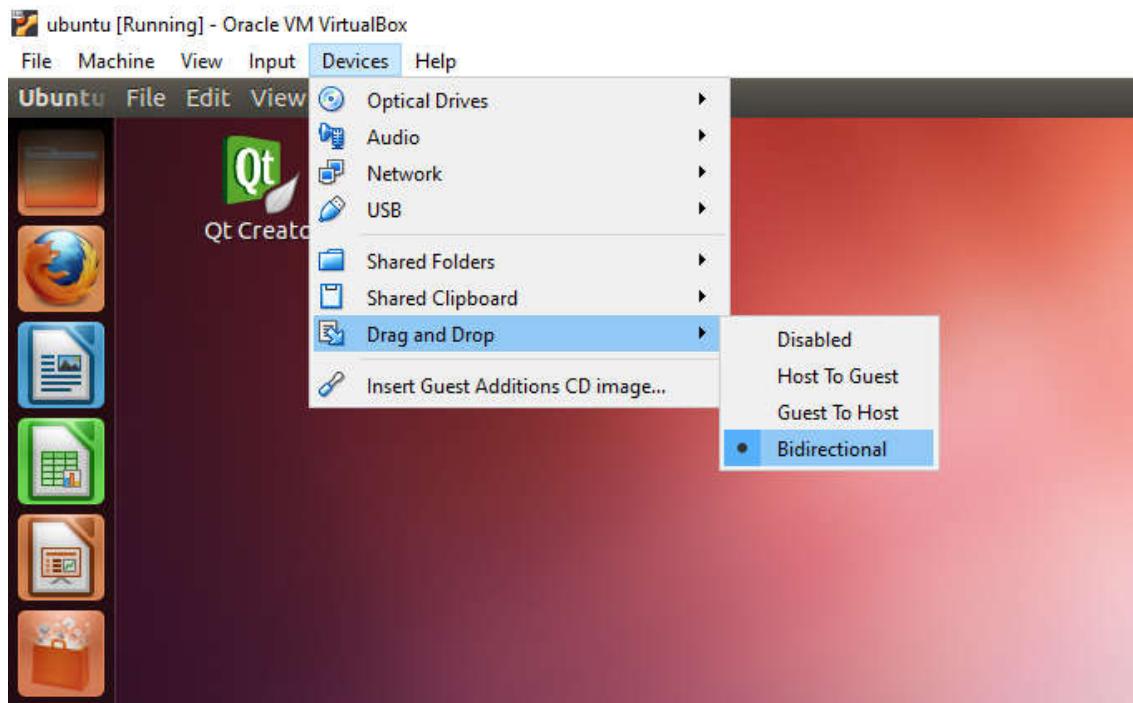
Thực hiện:

- Devices-Shared Clipboard-Bidirectional



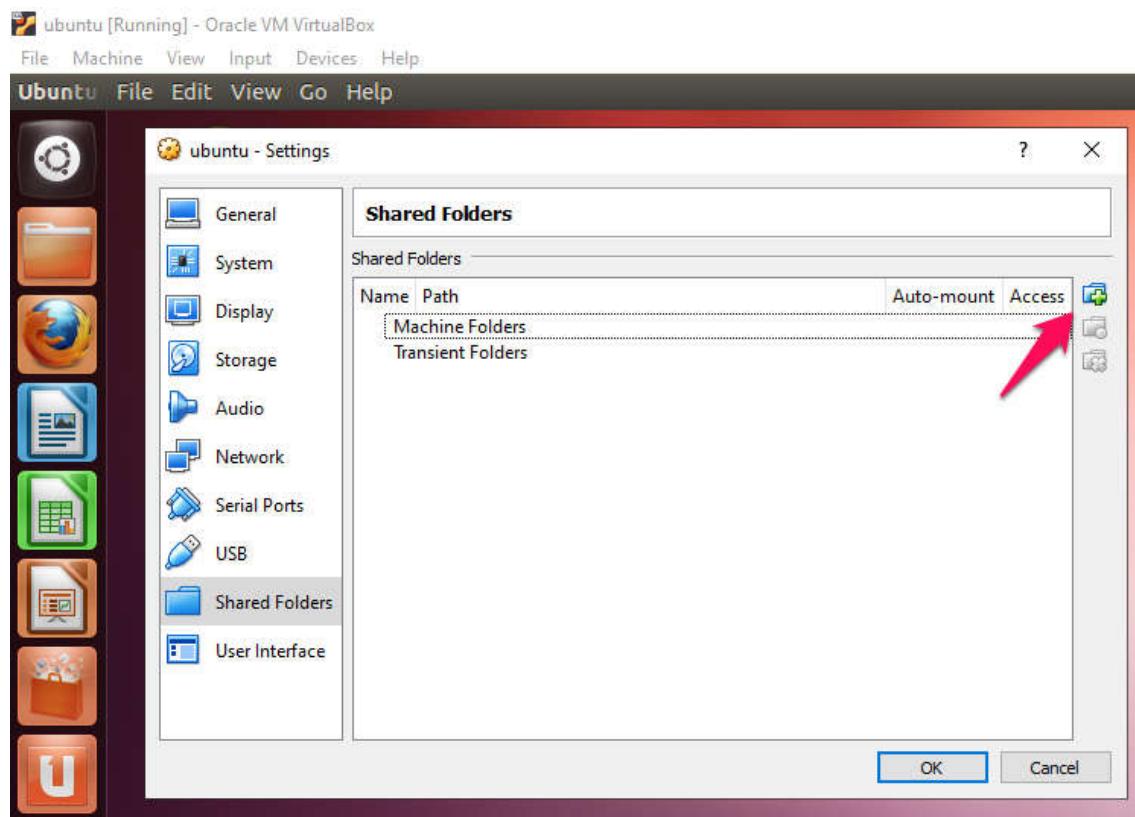
Hình 1. 25

- Devices-Drag'n'Drop-Bidirectional

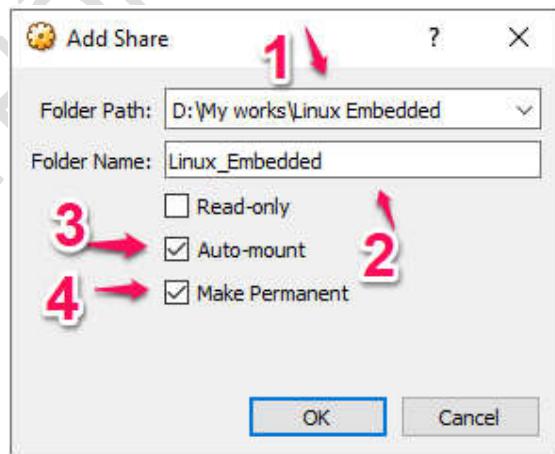


Hình 1. 26

- Devices-Shared Folders Setting...
Add a new shared folder definition



Giả sử ta share thư mục linux embedded



Hình 1. 27

- Khi shared xong restart lại máy ảo tìm đến đường dẫn sẽ thấy thư mục sharelinux và đây là thư mục được shared giữa windows và linux. Nếu mở thư mục này có báo lỗi:

**"The folder contents could not be displayed.
You do not have the permissions necessary to view the contents of "sf_Main".
"**

Mở terminal gõ lệnh sau:

\$sudo adduser user vboxsf

Trong đó: user là tên tài khoản của bạn. Ở đây user là embedded.

Logout và login sau đó vào thư mục shared sẽ thấy được nội dung

- Một số câu lệnh thông thường sử dụng trong linux:

▪ **Lệnh liên quan đến hệ thống :**

- exit: thoát khỏi cửa sổ dòng lệnh.
- logout: tương tự exit.
- reboot: khởi động lại hệ thống.
- halt: tắt máy.
- startx: khởi động chế độ xwindows từ cửa sổ terminal.
- mount: gắn hệ thống tập tin từ một thiết bị lưu trữ vào cây thư mục chính.
- umount: ngược với lệnh mount.

▪ **Lệnh thao tác trên tập tin**

- ls: lấy danh sách tất cả các file và thư mục trong thư mục hiện hành.
- pwd: xuất đường dẫn của thư mục làm việc.
- cd: thay đổi thư mục làm việc đến một thư mục mới.
- mkdir: tạo thư mục mới.
- rmdir: xoá thư mục rỗng.
- cp: copy một hay nhiều tập tin đến thư mục mới.
- mv: đổi tên hay di chuyển tập tin, thư mục.
- rm: xoá tập tin.
- wc: đếm số dòng, số ký tự... trong tập tin.
- touch: tạo một tập tin.
- cat: xem nội dung tập tin.
- vi: khởi động trình soạn thảo văn bản vi.
- df: kiểm tra dung lượng đĩa.
- du: xem dung lượng đĩa đã dùng cho một số tập tin nhất định

▪ **Lệnh khi làm việc trên terminal**

- clear: xoá trống cửa sổ dòng lệnh.
- date: xem ngày, giờ hệ thống.
- cal: xem lịch hệ thống.

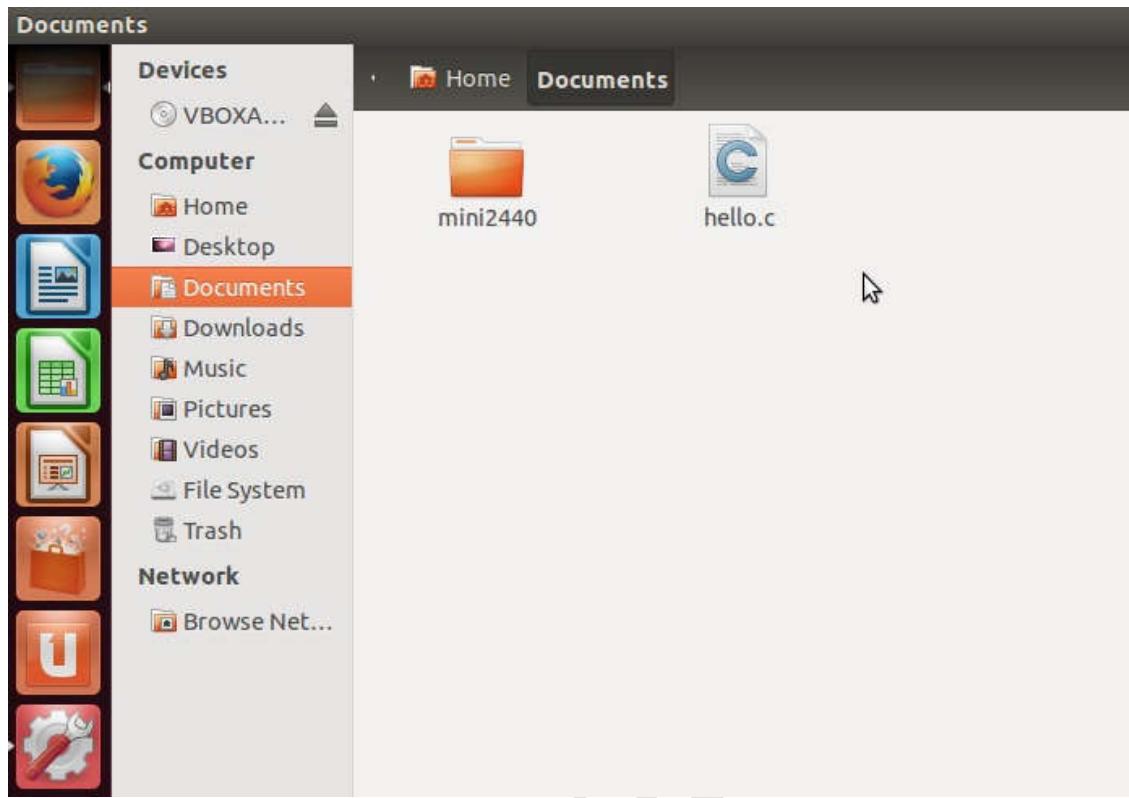
▪ **Lệnh quản lí hệ thống**

- rpm: kiểm tra gói đã cài đặt hay chưa, hoặc cài đặt một gói, hoặc sử dụng để gỡ bỏ một gói.
- ps: kiểm tra hệ thống tiến trình đang chạy.
- kill: dừng tiến trình khi tiến trình bị treo. Chỉ có người dùng super-user mới có thể dừng tất cả các tiến trình còn người dùng bình thường chỉ có thể dừng tiến trình mà mình tạo ra.
- top: hiển thị sự hoạt động của các tiến trình, đặc biệt là thông tin về tài nguyên hệ thống và việc sử dụng các tài nguyên đó của từng tiến trình.
- pstree: hiển thị tất cả các tiến trình dưới dạng cây.
- sleep: cho hệ thống ngừng hoạt động trong một khoảng thời gian.
- useradd: tạo một người dùng mới.
- groupadd: tạo một nhóm người dùng mới.
- passwd: thay đổi password cho người dùng.
- userdel: xoá người dùng đã tạo.
- groupdel: xoá nhóm người dùng đã tạo.
- gpasswd: thay đổi password của một nhóm người dùng.
- su: cho phép đăng nhập với tư cách người dùng khác.
- groups: hiển thị nhóm của user hiện tại.
- who: cho biết ai đang đăng nhập hệ thống.
- w: tương tự như lệnh who.
- man: xem hướng dẫn về dòng lệnh như cú pháp, các tham số...

1.6 Viết chương trình chạy trên máy host linux và trên target (kit):

1.6.1 Viết mã nguồn 1 chương trình tên hello:

Vào thư mục Document, click phải chuột-Create New Document đặt tên file là hello.c

**Hình 1. 28**

Soạn thảo file mã nguồn C như sau:

```
#include "stdio.h"
int main(int argc, char**argv)
{
    int i=0;
    printf("Hello world, lap trinh arm linux\n");
    printf("Ten chuong trinh la: %s\n", argv[0]);
    if(argc<2)
        printf("Chuong trinh khong co tham so dau vao nao\n");
    else
    {
        for(i=1;i<argc;i++)
        {
            printf("Tham so thu %d, noi dung %s\n", i, argv[i]);
        }
    }
    return 0;
}
```

1.6.2 Biên dịch chương trình dùng gcc để chạy trên máy host linux:

Vào terminal chuyển đến thư mục chứa file hello.c

\$ cd Documents/

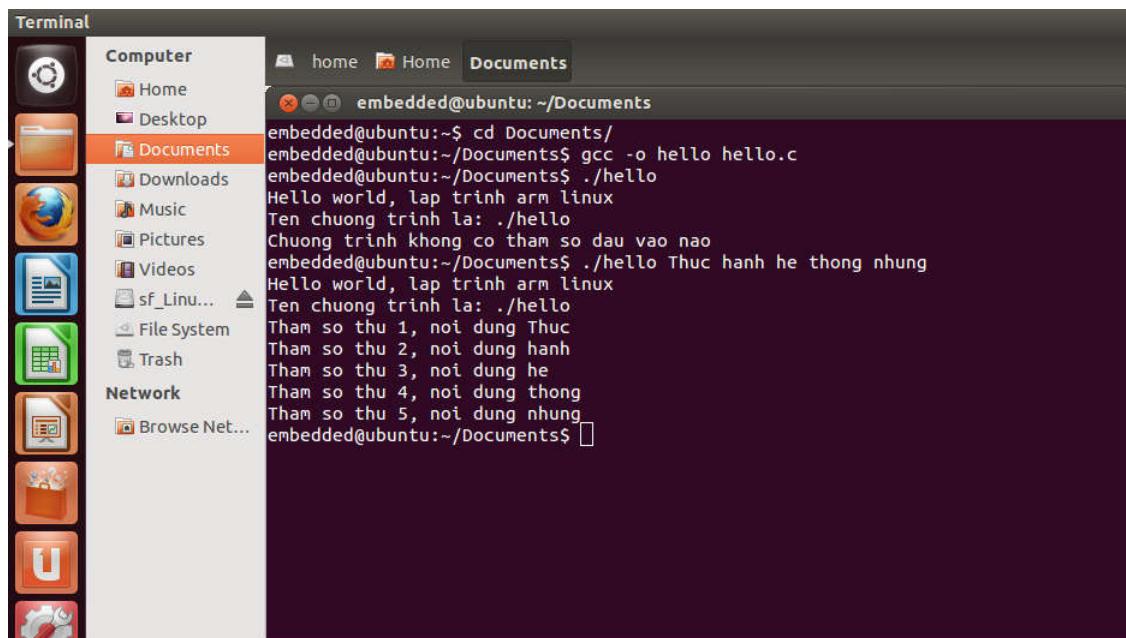
Biên dịch bằng lệnh:

\$ gcc -o hello hello.c

Kết quả biên dịch tạo ra file hello nằm cùng thư mục hello.c

Chạy chương trình dùng lệnh:

\$./hello



Hình 1. 29

1.6.3 Biên dịch chương trình dùng trình biên dịch chéo:

Cách 1: Sử dụng lệnh của trình biên dịch chéo(máy linux host đã cài đặt trình biên dịch chéo arm-linux-gcc)

- ✓ Chuyển đến thư mục chứa file mã nguồn hello.c đã có
- ✓ Biên dịch bằng lệnh: **\$ arm-linux-gcc -o hello hello.c**
- ✓ Kết quả biên dịch là file thực thi trên arm linux(file hello) (thêm tham số -g khi biên dịch để được file kết quả cho phép debug).

Cách 2: Tạo và sử dụng Makefile:

- ✓ Make là 1 công cụ cho phép quản lý quá trình biên dịch, liên kết của 1 dự án với nhiều file mã nguồn.
- ✓ Tạo Makefile để lưu các lệnh biên dịch theo định dạng của Makefile.
CC=arm-linux-gcc
all: hello.c

```
$(CC)-g-o hello hello.c
```

clear:

```
rm hello
```

- ✓ Makefile đặt tại cùng thư mục với các file mã nguồn. Sử dụng lệnh make để chạy Makefile và biên dịch chương trình
 - make all: biên dịch chương trình
 - make clear: xóa file thực thi đã biên dịch

1.6.4 Nạp chương trình đã biên dịch xuống kit và chạy chương trình trên kit:

Vào thư mục chứa file hello.c, dùng trình biên chéo để biên dịch.

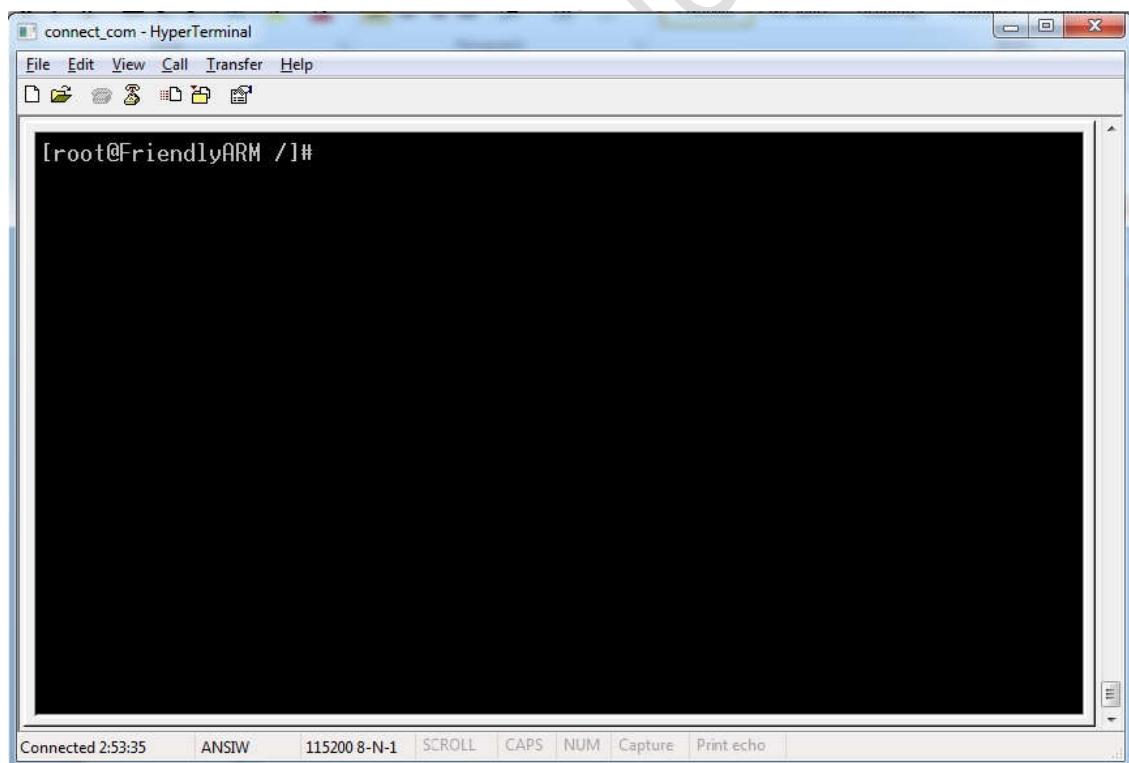
Thực hiện:

```
$ arm-linux-gcc -o hello hello.c
```

Lúc này trong thư mục chứa file hello.c có xuất hiện file đã biên dịch chéo **hello**.

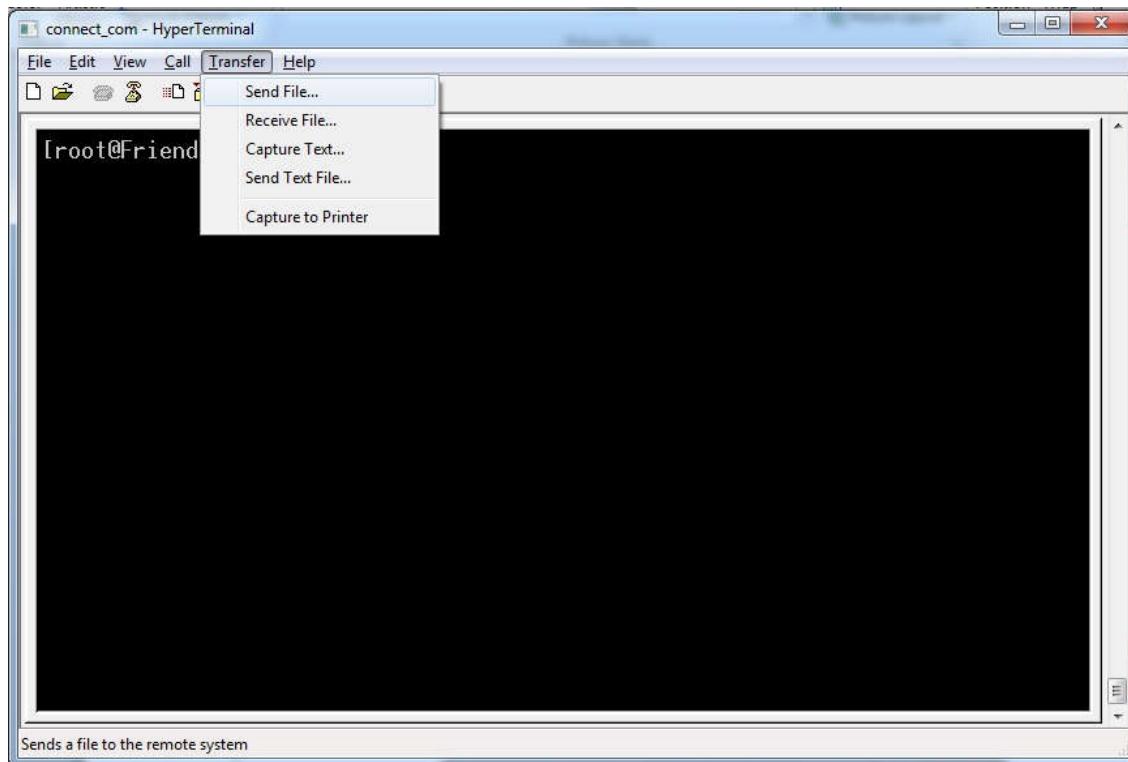
a. Truyền dữ liệu từ PC xuống kit qua cổng COM:

Kết nối kit với PC qua hyper Terminal như trình bày ở trên. Kết nối thành công sẽ xuất hiện dòng chữ **[root@FriendlyARM /]#**.



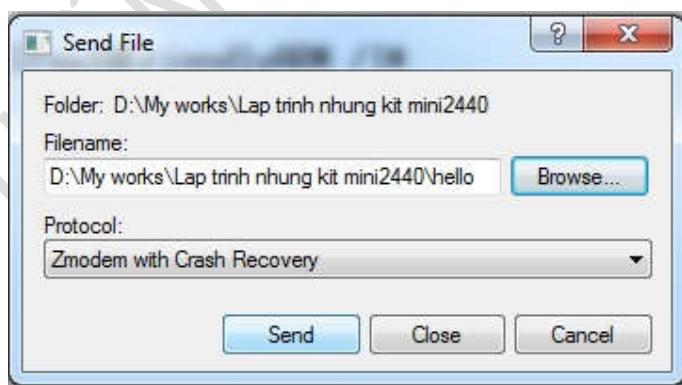
Hình 1. 30

Vào Transfer-Send File...



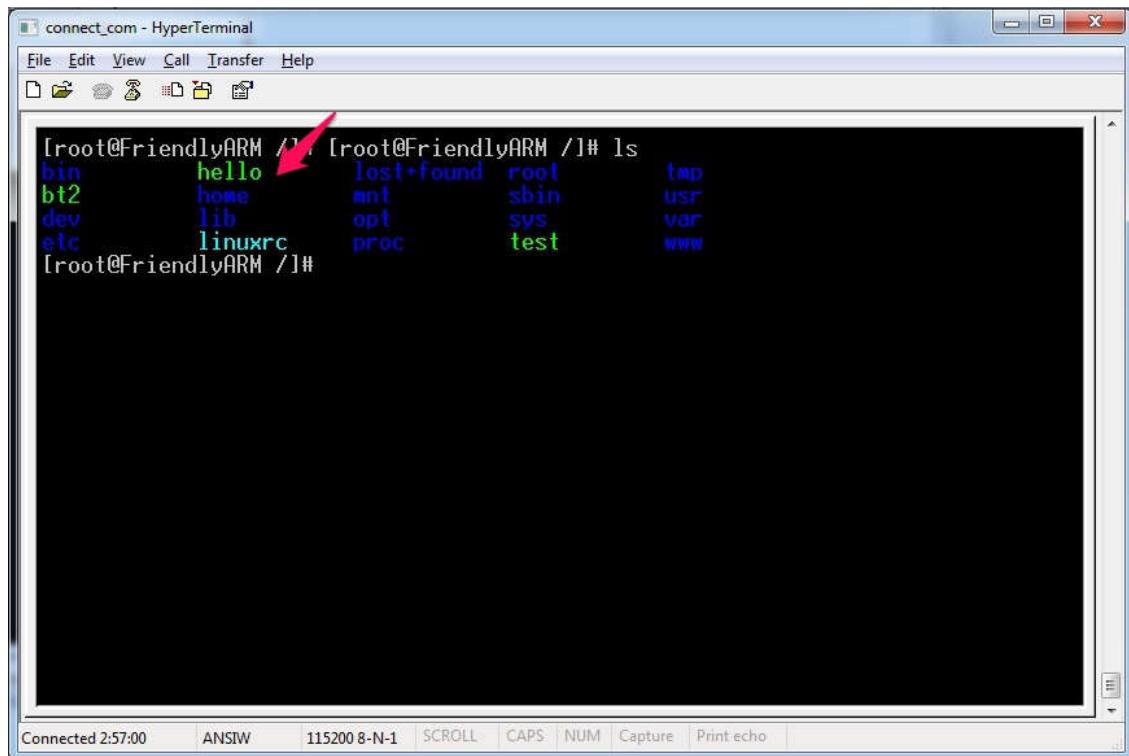
Hình 1. 31

Browse đến thư mục chứa file đã biên dịch chéo **hello**(lưu ý ta phải share thư mục chứa file đã biên dịch chéo **hello** hoặc chép file này vào thư mục đã share) sau đó nhấn **Send**



Hình 1. 32

Kiểm tra file đã tồn tại trong linux nhúng:
ls



```
[root@FriendlyARM /]# ls
bin      hello    lost+found  root      tmp
bt2      home     mnt        sbin     usr
dev      lib       opt        sys      var
etc      linuxrc   proc      test
[root@FriendlyARM /]#
```

Hình 1.33

Chạy chương trình: **./hello**. Lúc này xuất hiện dòng chữ Permisson denied: báo chưa cho phép chạy chương trình này. Để có thể chạy được ta phải cấp quyền để chạy dùng lệnh:

```
chmod 777 hello
./hello
```

```

[root@FriendlyARM /]# ls
bin      hello      lost+found  root      tmp
bt2      home      mnt        sbin      usr
dev      lib       opt        sys       var
etc      linuxrc   proc      test      www

[root@FriendlyARM /]# ./hello
Hello world, lap trinh arm linux
Ten chuong trinh la: ./hello
Chuong trinh khong co tham so dau vao nao
[root@FriendlyARM /]# ./hello thi ngleiem he thong nhung
Hello world, lap trinh arm linux
Ten chuong trinh la: ./hello
Tham so thu 1, noi dung thi
Tham so thu 2, noi dung ngleiem
Tham so thu 3, noi dung he
Tham so thu 4, noi dung thong
Tham so thu 5, noi dung nhung
[root@FriendlyARM /]# _

```

Hình 1. 34

So sánh kết quả chạy trên máy linux(host) và linux(target) hoàn toàn giống nhau.

b. Truyền dữ liệu từ PC xuống kit qua cổng LAN:

Để truyền file từ máy host linux xuống kit qua cổng LAN chúng ta kết nối cáp LAN giữa máy tính và KIT bằng phần mềm gFTP(phần mềm giao tiếp truyền file trên Ubuntu)

Bước 1: Cài đặt gFTP

Thông thường gFTP đã cài đặt sẵn trên Ubuntu. Nếu chưa cài đặt có thể cài đặt từ internet:

\$ sudo apt-get install gftp

Bước 2: Cấu hình cổng LAN trên máy tính Linux

Kit Arm khi cài đặt có địa chỉ IP mặc định là: **192.168.1.230**, netmask 255.255.255.0. Do vậy để kết nối giữa PC và kit qua cổng LAN cần phải cấu hình để máy tính có cùng giải địa chỉ này. Thực hiện ifconfig cấu hình cho ethernet interface(eth0)

\$ sudo ifconfig eth0 192.168.1.30 netmask 255.255.255.0 up

Sau đó dùng lệnh ping để kiểm tra máy host có thấy được kit không

\$ ping 192.168.1.230

Bước 3: Kết nối máy host(PC) và target(kit) dùng gFTP

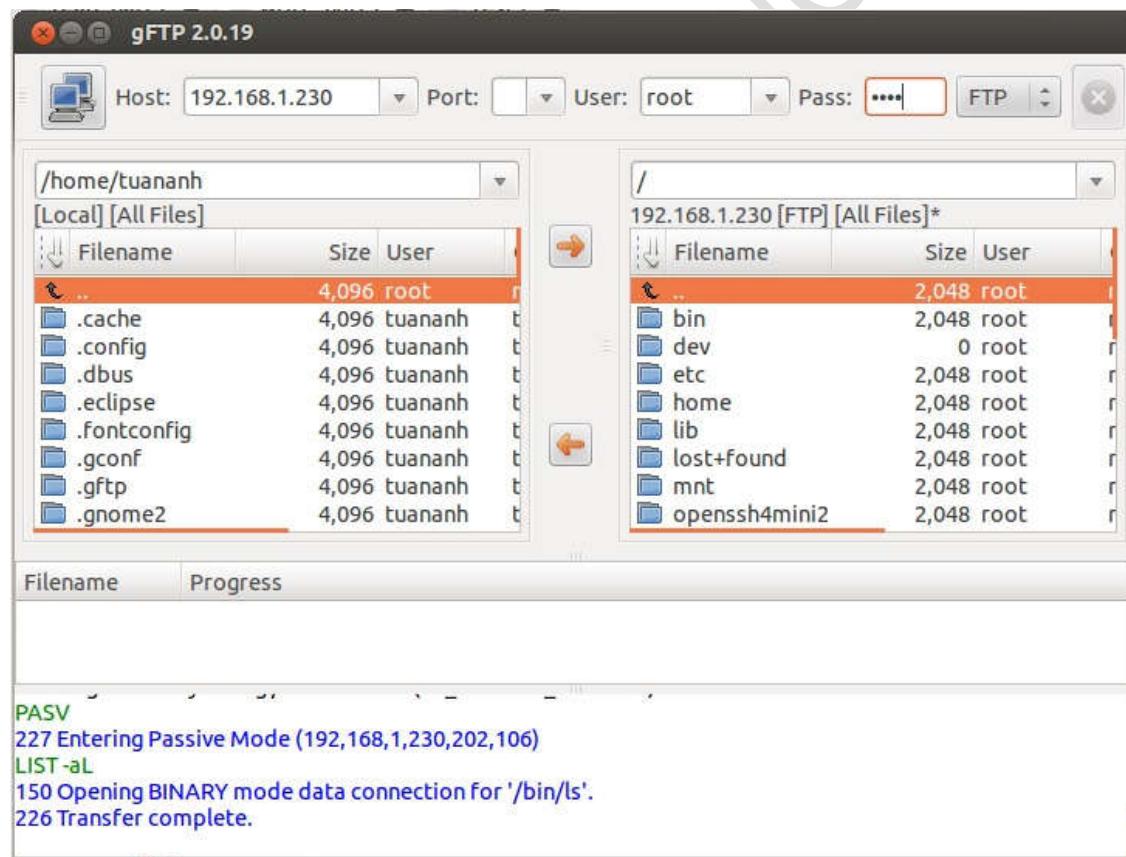
Mở phần mềm gFTP: Applications->Internet->gFTP

Thiết lập các tham số:

- Host: 192.168.1.230(địa chỉ IP kit)
- User: root(user kit)
- Pass: 1234(đã đặt pass kit là root)

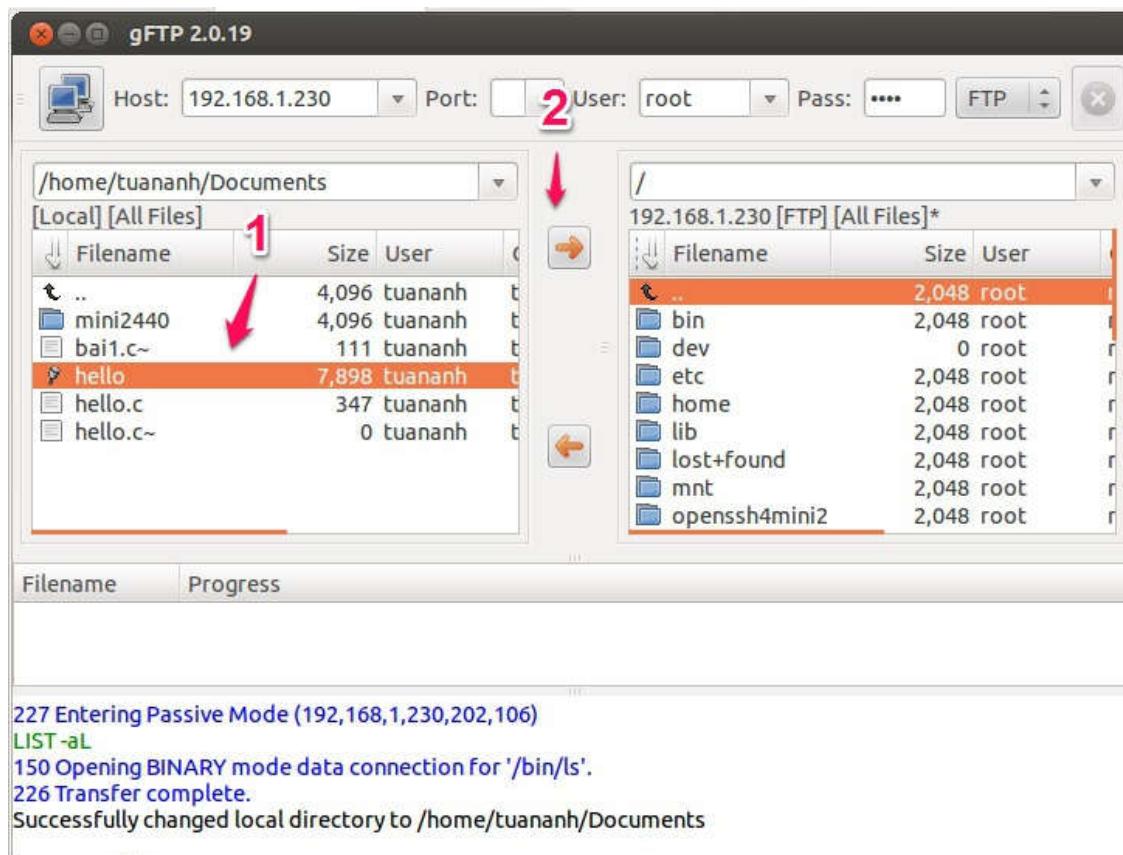
Mở kết nối:

- Phía bên tay trái là thư mục của máy host
- Phía bên tay phải là thư mục của kit ARM chứa các file trong hệ điều hành nhúng linux.



Hình 1. 35

Ta có thể copy file từ máy host sang kit ARM dễ dàng từ gFTP:

**Hình 1.36**

Để chạy chương trình ta thực hiện lệnh telnet đến kit. Dùng lệnh

\$ telnet 192.168.1.230

FriendlyARM login:**root**

Password: **1234**

```

Terminal
embedded@ubuntu: ~
embedded@ubuntu:~$ telnet 192.168.1.230
Trying 192.168.1.230...
Connected to 192.168.1.230.
Escape character is '^]'.

Kernel 2.6.32.2-FriendlyARM on (/dev/pts/0)
FriendlyARM login: root
Password:
[root@FriendlyARM /]# chmod 777 hello
[root@FriendlyARM /]# ./hello
Hello world, lap trinh arm linux
Ten chuong trinh la: ./hello
Chuong trinh khong co tham so dau vao nao
[root@FriendlyARM /]# ./hello Thuc hanh he thong nhung
Hello world, lap trinh arm linux
Ten chuong trinh la: ./hello
Tham so thu 1, noi dung Thuc
Tham so thu 2, noi dung hanh
Tham so thu 3, noi dung he
Tham so thu 4, noi dung thong
Tham so thu 5, noi dung nhung
[root@FriendlyARM /]#

```

Hình 1.37

1.7 Viết chương trình C dùng eclipse trong Linux:

Thực hiện project dùng eclipse trong linux giúp chúng ta quản lý project, debug chương trình dễ dàng hơn.

❖ Cài đặt phần mềm:

Có 2 cách để cài đặt phần mềm:

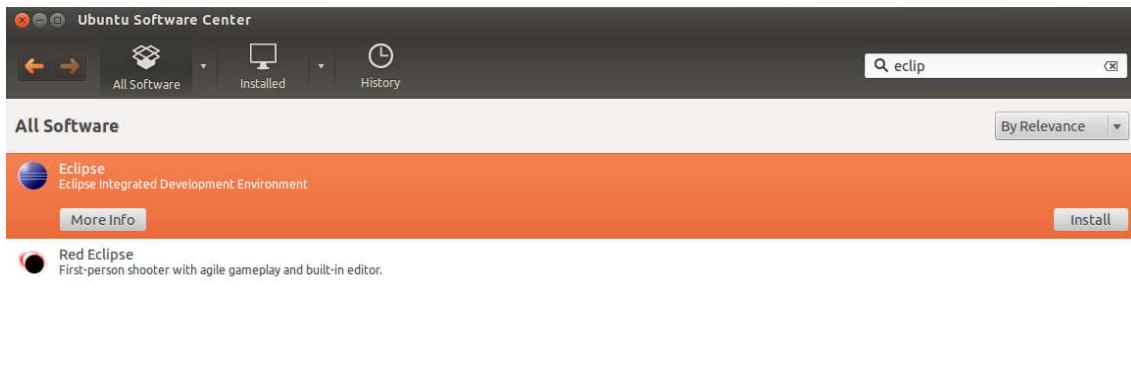
Cách 1:

Vào terminal thực hiện lệnh:

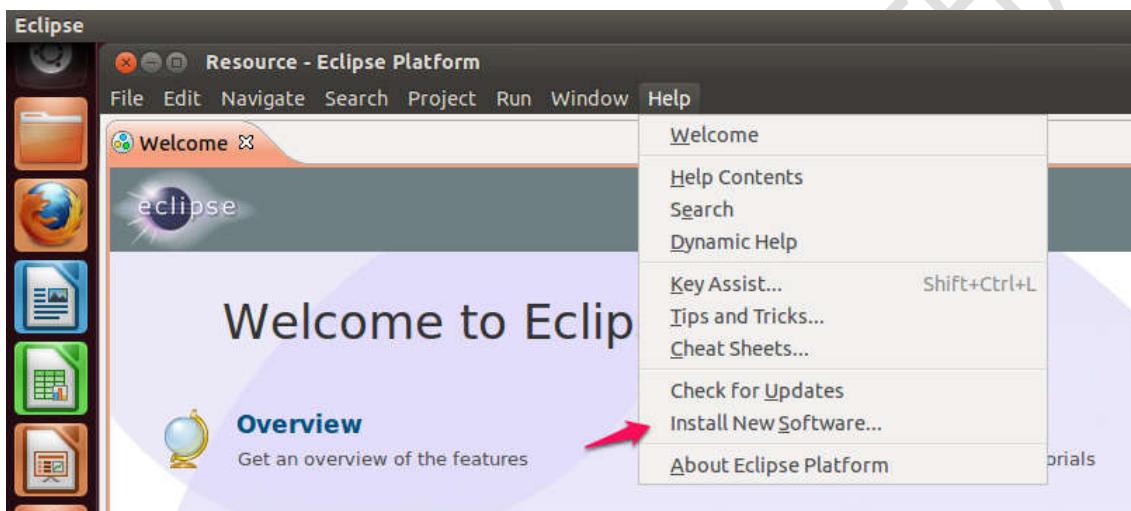
`$sudo apt-get install eclipse eclipse-cdt g++`

Cách 2:

Vào Ubuntu Software Center gõ vào ô tìm kiếm: eclipse, sau đó nhấn vào install.



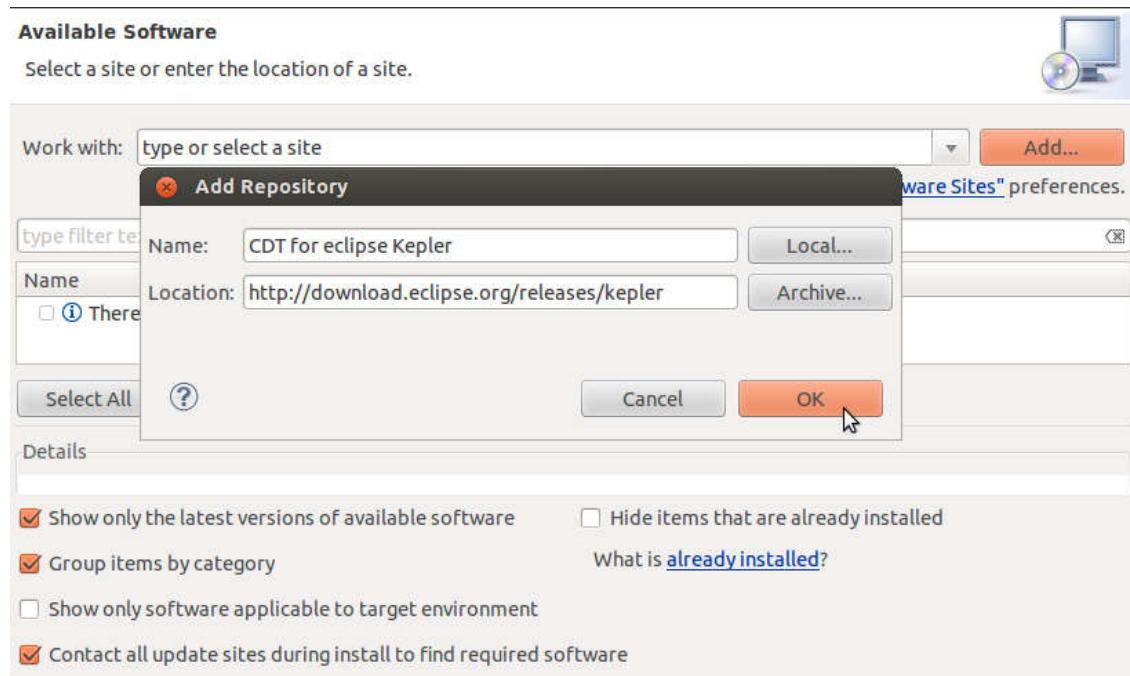
Cần cài đặt thêm Eclipse CDT(C/C++ development tools) để có thể lập trình C/C++ như sau: mở Eclipse vào menu Help->Install New Software.



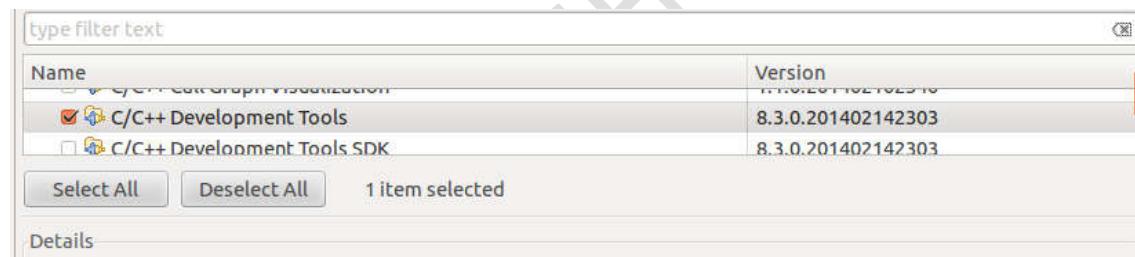
Nhấn vào nút Add...

Name: **CDT for eclipse Kepler**

Location: <http://download.eclipse.org/releases/kepler>



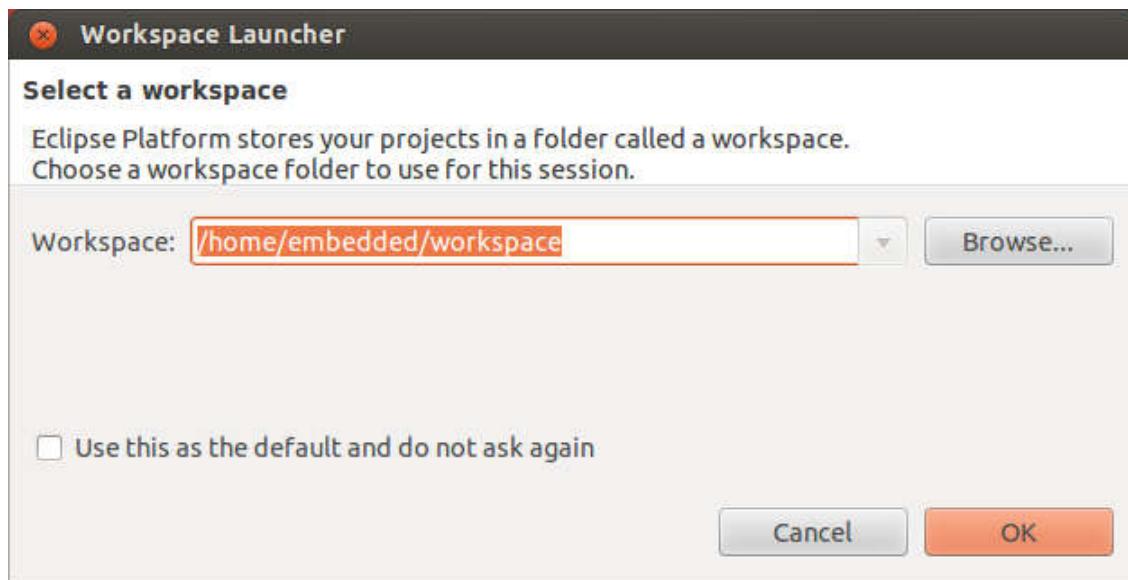
Đợi nó load các plugin, tìm đến Programming Language và chọn C/C++ Development Tools và nhấn Next→ ...->Finish.



❖ Sử dụng eclipse lập trình C:

Bước 1: Tạo project

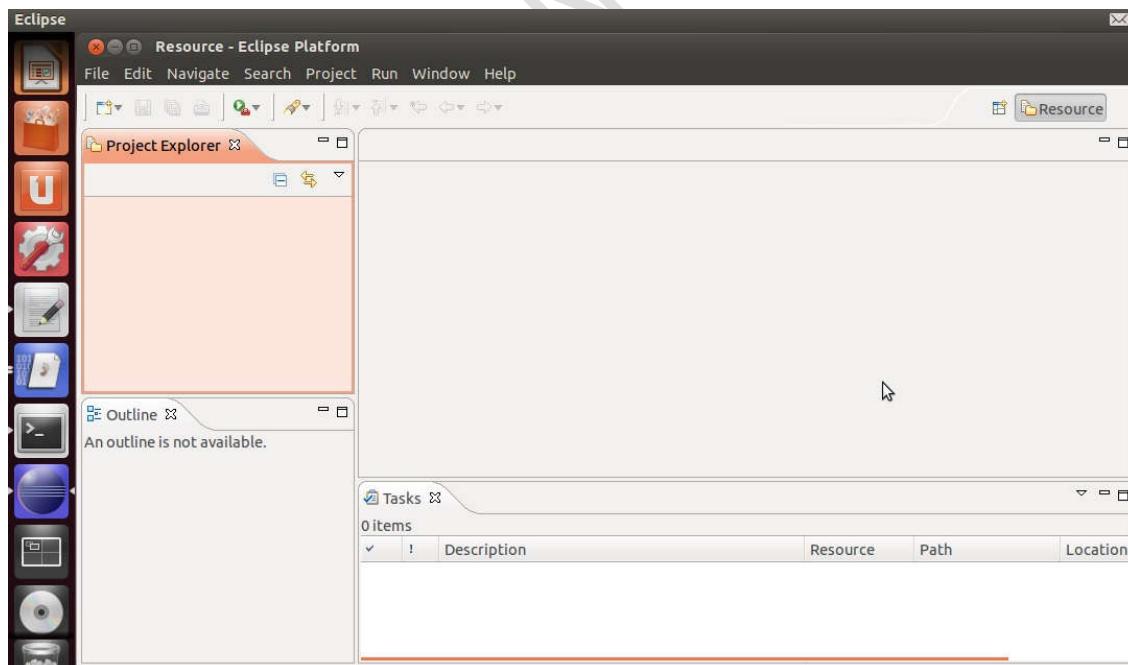
Mở phần mềm eclipse : Applications->Internet->eclipse(đã cài đặt c/c++ package)



Hình 1. 38

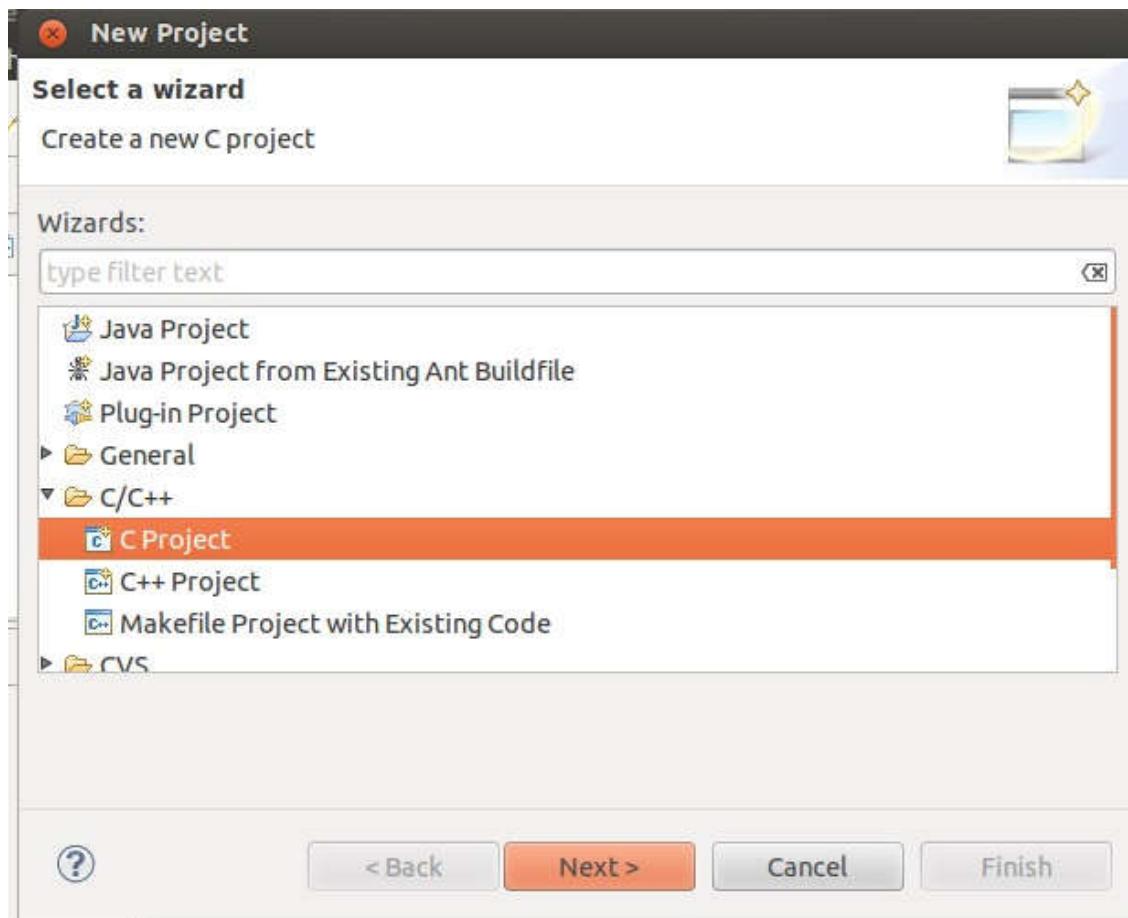
Browse đến thư mục chứa project → OK

File-New Project

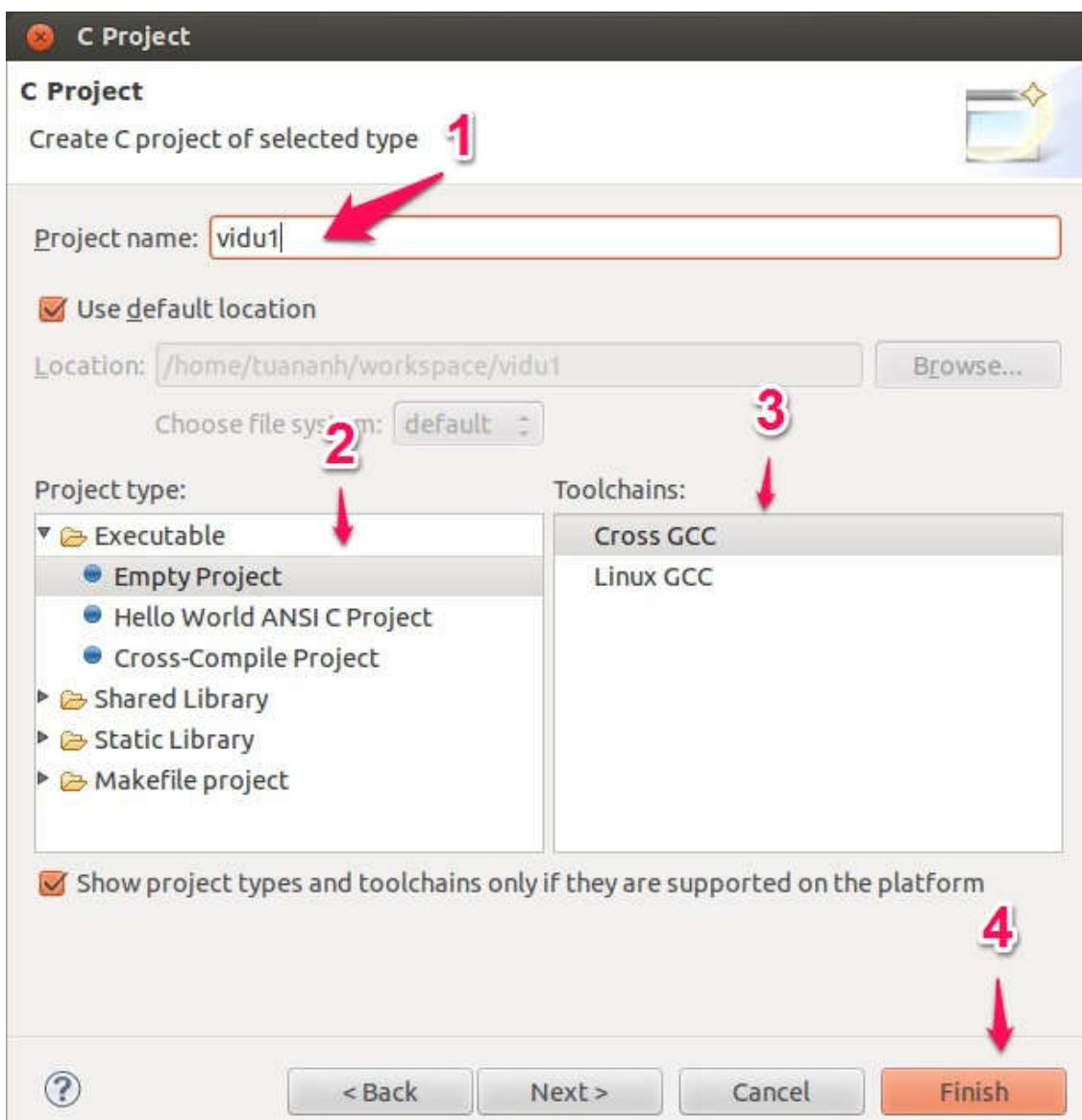
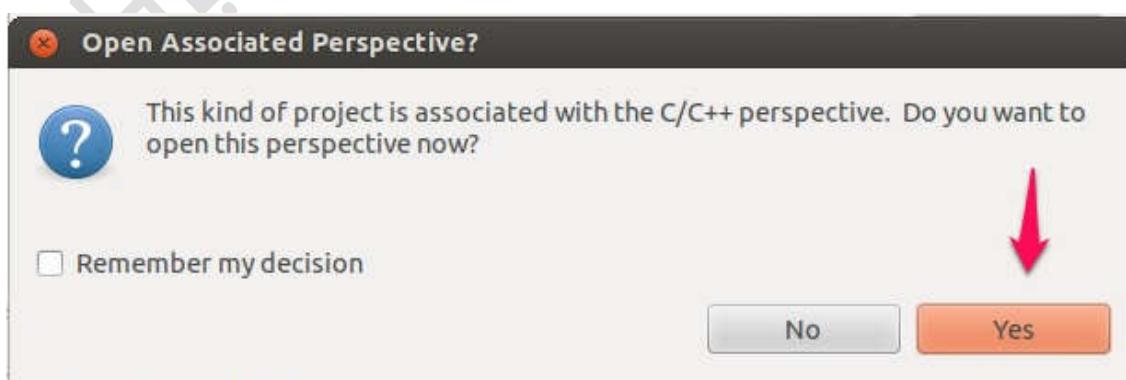


Hình 1. 39

C/C++ → C Project → Next

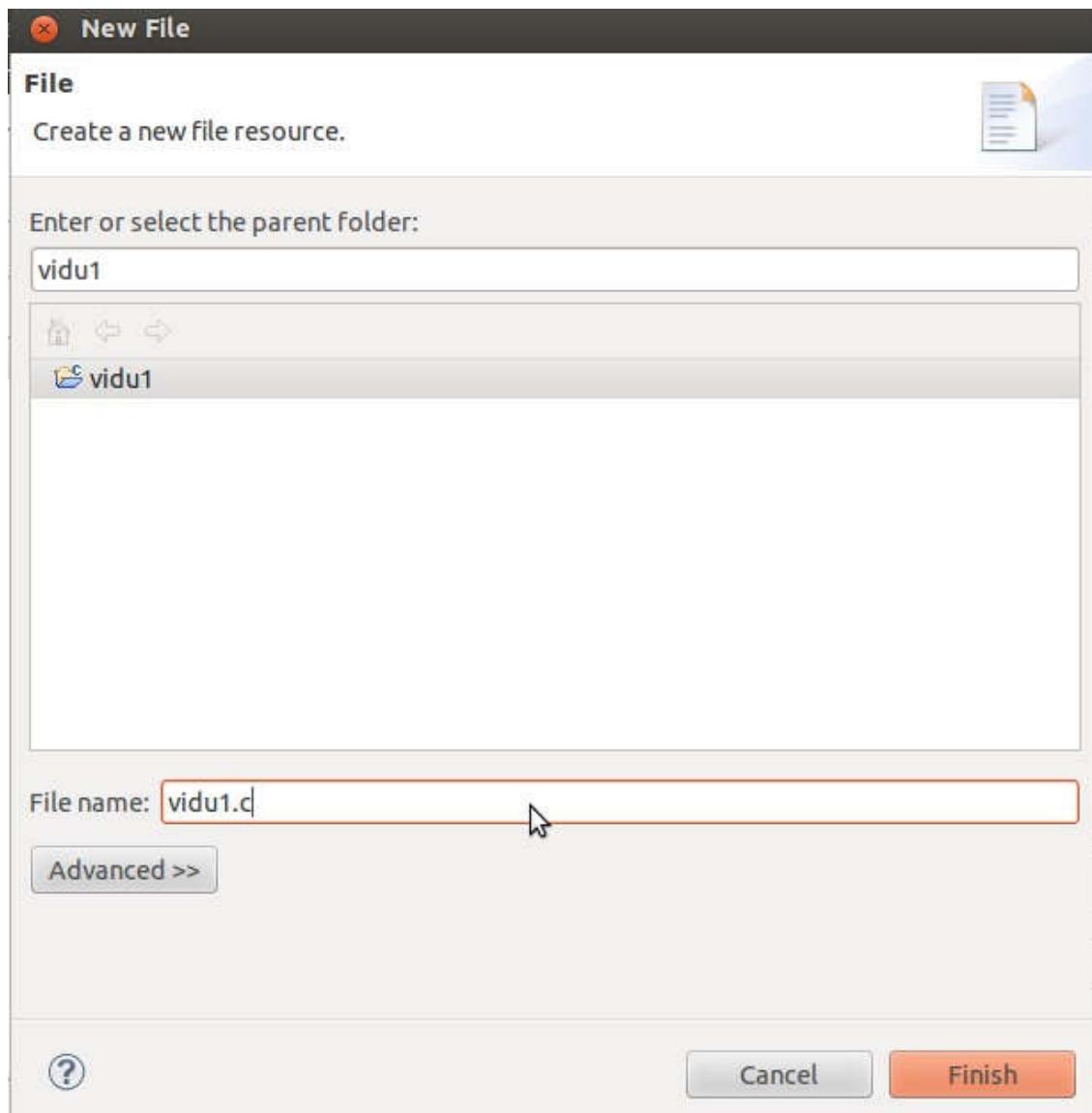


Hình 1. 40

**Hình 1. 41****Hình 1. 42**

Click phải chuột vidu1-New-File

Vào File Name: **vidu1.c**



Hình 1. 43

Bước 2: Viết code và thiết lập

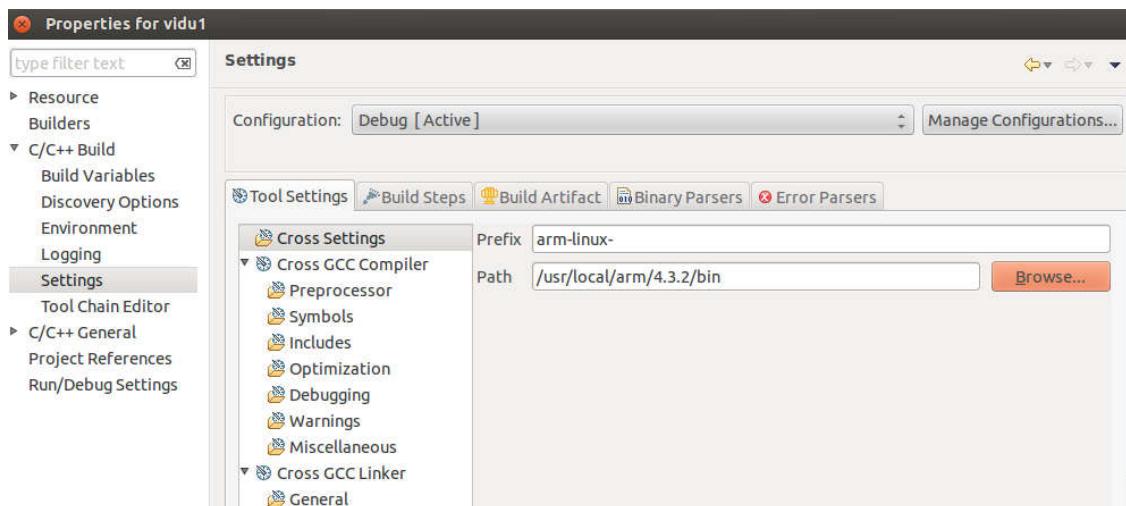
Vào chương trình soạn thảo viết code

Project-Properties

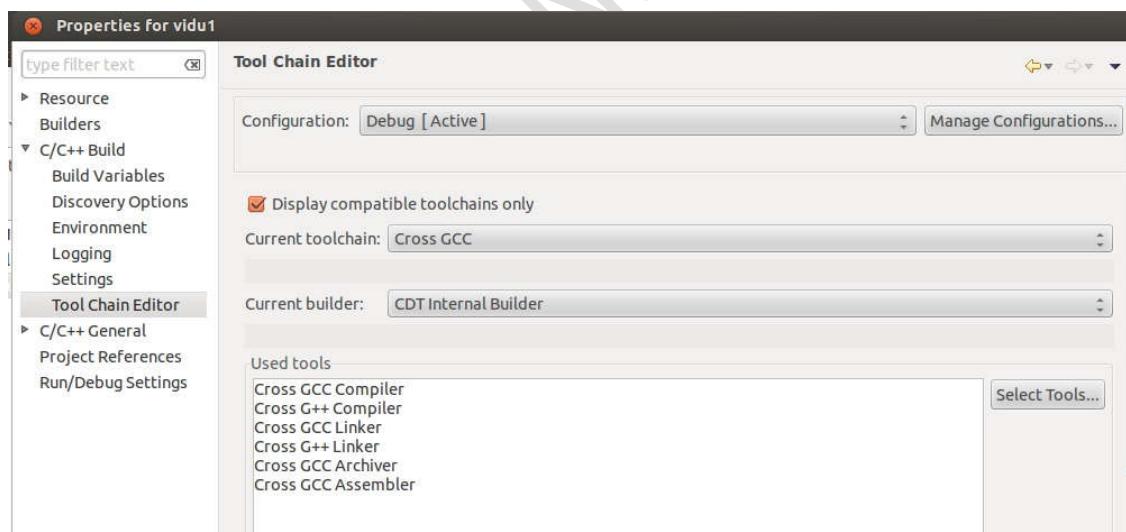
C/C++ Build → Setting

Prefix: **arm-linux-**

Path: Browse đến đường dẫn **usr/local/arm /4.3.2/bin**

**Hình 1. 44**

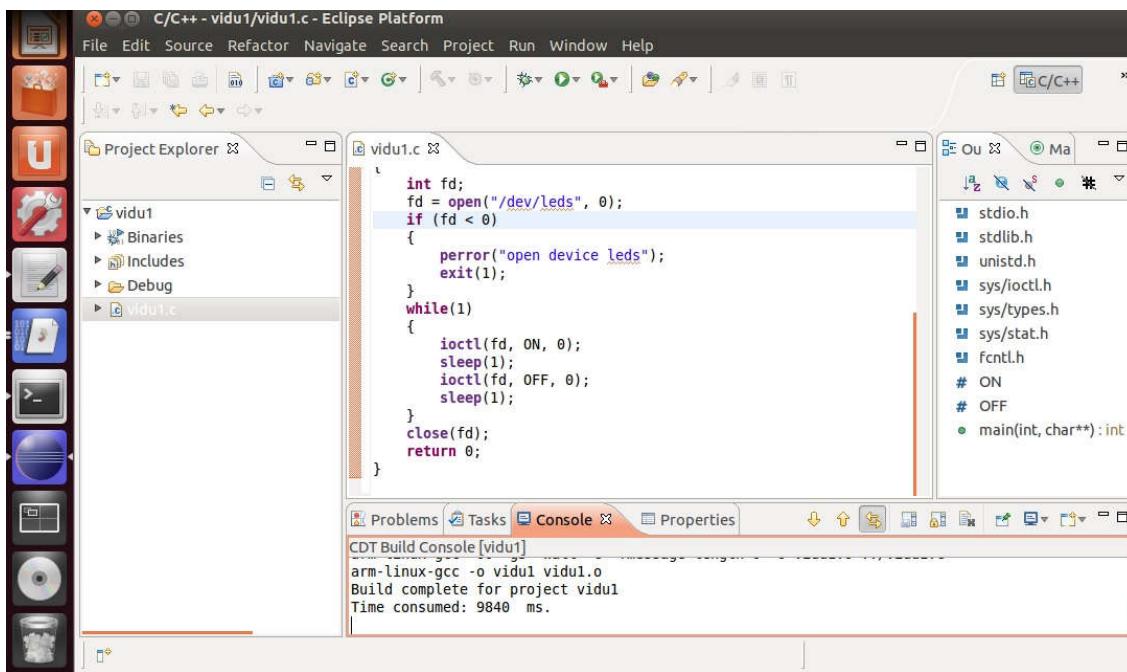
C/C++ Build → Tool Chain Editor

Current builder: **CDT Internal Builder** → OK**Hình 1. 45****Bước 3:** Biên dịch project

Vào Project → Build All(hoặc Ctrl B). Nếu project không có lỗi thì trên tab Console xuất hiện dòng chữ:

Build complete for project ...

Time consumed:....



Hình 1. 46

Lúc này trong thư mục Debug của project vừa tạo đã có file **vidu1** đã được biên dịch xong.

1.8 Biên dịch lại nhân của hệ điều hành linux nhúng:

Mục đích của việc biên dịch lại nhân(kernel) của hệ điều hành là ta muốn tùy chỉnh lại nhân của hệ điều hành theo nhu cầu sử dụng mà ta mong muốn. Sau đó ta sẽ nạp lại nhân của hệ điều hành.

Yêu cầu: Máy host đã cài đặt trình biên dịch chéo, file linux-2.6.32.2-mini2440-20110413.tar.gz.

Các bước thực hiện:

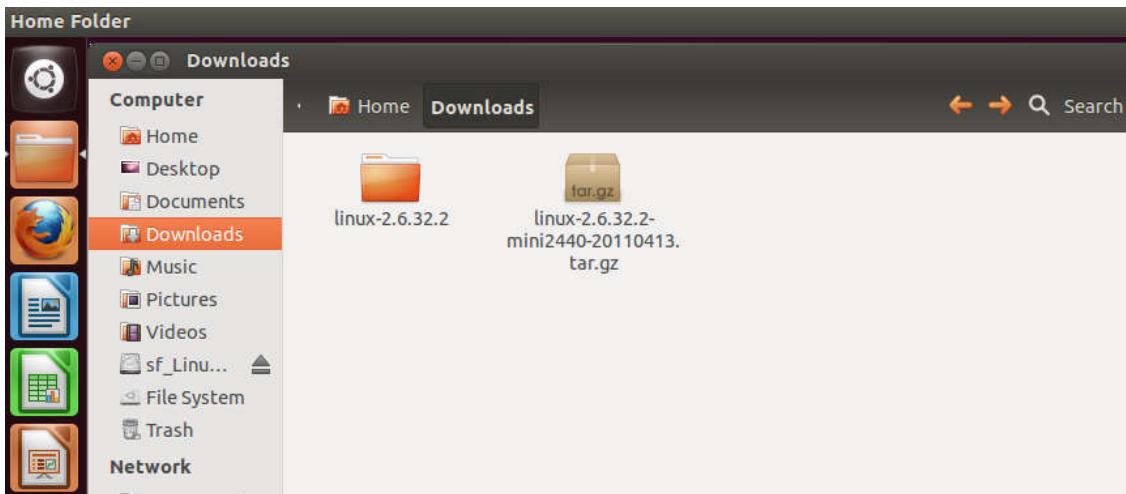
Bước 1:

Chuyển vào thư mục có file chứa mã nguồn và giải nén:

cd Download (thư mục Download file mã nguồn)

tar -xvzf linux-2.6.32.2-mini2440-20110413.tar.gz

Kết quả giải nén được thư mục **linux-2.6.32.2**



Bước 2:

Chuyển vào thư mục có file chứa mã nguồn và giải nén:

cd linux-2.6.32.2/

Copy đè file cấu hình có sẵn tương ứng lên file .config(ẩn):

sudo cp config_mini2440_t35 .config (t35 tương ứng với màn hình kit mini2440 đang sử dụng).

Bước 3:

Dùng lệnh để mở menu cấu hình:

sudo make menuconfig

Có thẻ máy host thiếu 1 số thư viện cần thiết ta cần cài đặt thêm vào để vào menu cấu hình.

```
embedded@ubuntu:~/Downloads/linux-2.6.32.2$ cp config_mini2440_t35 .config
[sudo] password for embedded:
HOSTCC scripts/basic/fixedep
HOSTCC scripts/basic/docproc
HOSTCC scripts/basic/hash
HOSTCC scripts/kconfig/conf.o
scripts/kconfig/conf.c: In function 'conf_sym':
scripts/kconfig/conf.c:159:6: warning: variable 'type' set but not used [-Wunused-but-set-variable]
scripts/kconfig/conf.c: In function 'conf_choice':
scripts/kconfig/conf.c:231:6: warning: variable 'type' set but not used [-Wunused-but-set-variable]
scripts/kconfig/conf.c:307:9: warning: ignoring return value of 'fgets', declared with attribute warn_unused_result [-Wunused-result]
scripts/kconfig/conf.c: In function 'conf_askvalue':
scripts/kconfig/conf.c:105:8: warning: ignoring return value of 'fgets', declared with attribute warn_unused_result [-Wunused-result]
HOSTCC scripts/kconfig/kgettext.o
*** Unable to find the ncurses libraries or the
*** required header files.
*** 'make menuconfig' requires the ncurses libraries.
***
*** Install ncurses (ncurses-devel) and try again.
***
make[1]: *** [scripts/kconfig/dochecklxdialog] Error 1
make: *** [menuconfig] Error 2
embedded@ubuntu:~/Downloads/linux-2.6.32.2$ sudo apt-get install ncurses-dev
Reading package lists... Done
Building dependency tree
Reading state information... Done
Note, selecting 'libncurses5-dev' instead of 'ncurses-dev'
The following extra packages will be installed:
  libtinfo-dev
```

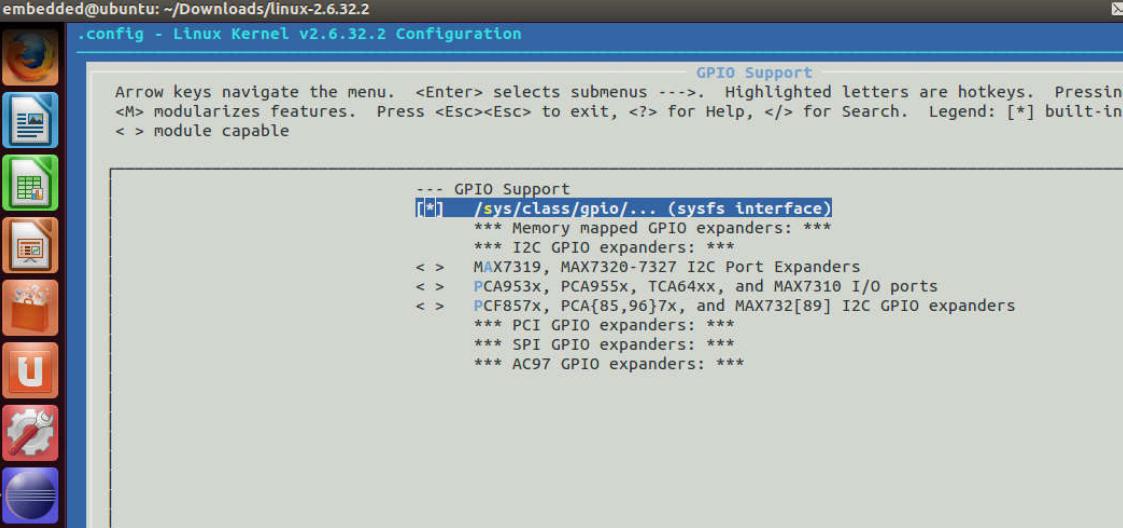
Menu cấu hình cho phép chọn lựa các thành phần module, driver được sử dụng trong nhân linux.

Để sử dụng các gpio mở rộng ta chọn lựa:

Device driver →

[*] GPIO Support

[*] /sys/class/gpio/...(sysfs interface)



```

embedded@ubuntu: ~/Downloads/linux-2.6.32.2
.config - Linux Kernel v2.6.32.2 Configuration

GPIO Support
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
< > module capable

--- GPIO Support
[*] /sys/class/gpio/... (sysfs interface)
    *** Memory mapped GPIO expanders: ***
    *** I2C GPIO expanders: ***
    < > MAX7319, MAX7320-7327 I2C Port Expanders
    < > PCA953x, PCA955x, TCA64xx, and MAX7310 I/O ports
    < > PCF857x, PCA[85,96]7x, and MAX732[89] I2C GPIO expanders
    *** PCI GPIO expanders: ***
    *** SPI GPIO expanders: ***
    *** AC97 GPIO expanders: ***

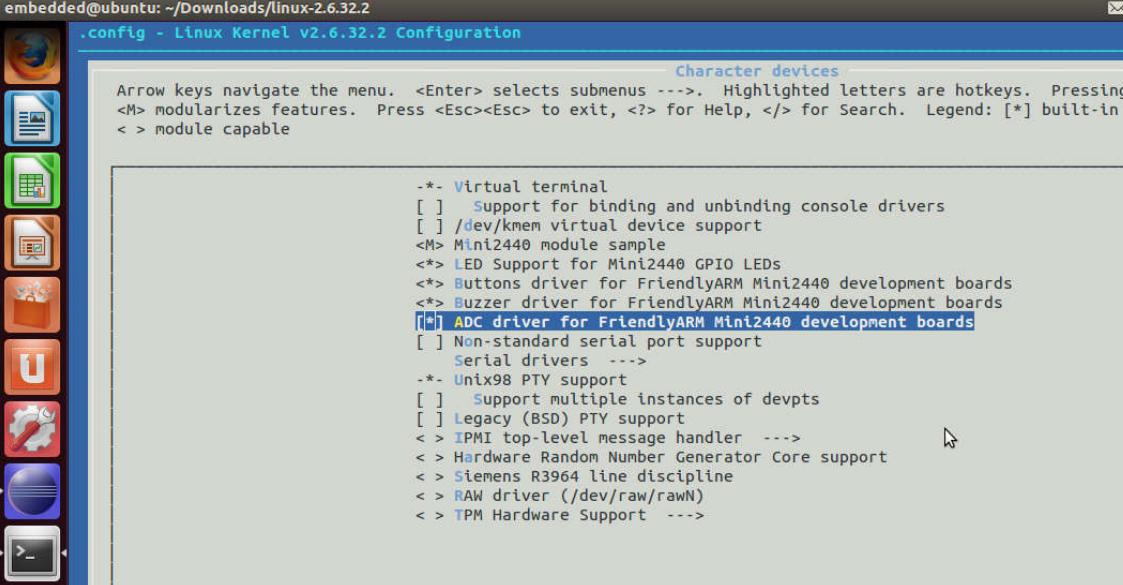
```

Để sử dụng driver giao tiếp với các ngoại vi led, button, buzzer, adc ta chọn các mục tương ứng.

Device driver

[*]Character device

- <*> LED Support for Mini2440 GPIO LEDs
- <*> Buttons driver for FriendlyARM Mini2440 development boards
- <*> Buzzer driver for FriendlyARM Mini2440 development boards
- [*] ADC driver for FriendlyARM Mini2440 development boards



```

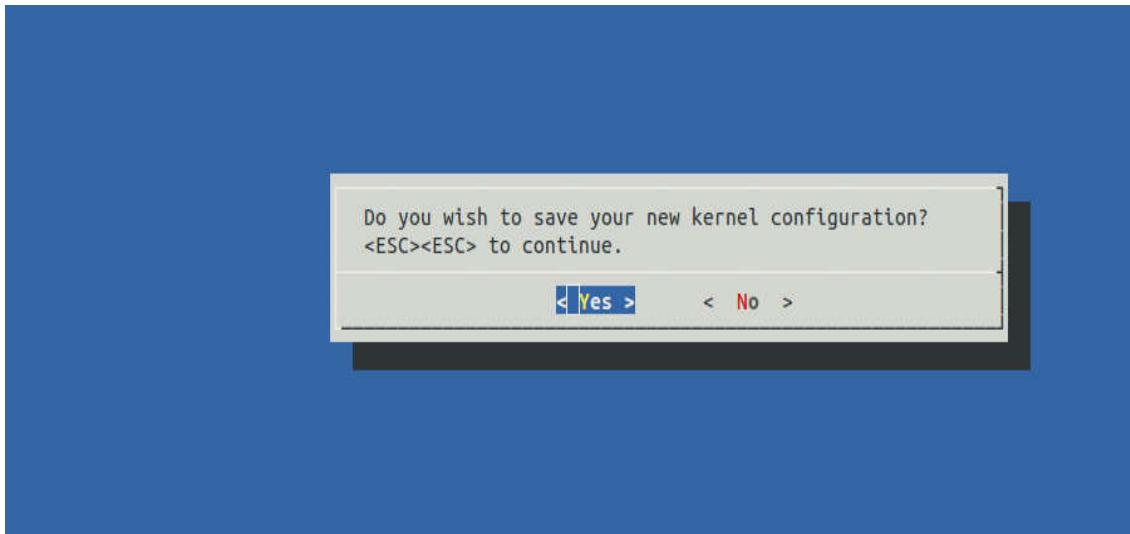
embedded@ubuntu: ~/Downloads/linux-2.6.32.2
.config - Linux Kernel v2.6.32.2 Configuration

Character devices
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in
< > module capable

-- Virtual terminal
[ ] Support for binding and unbinding console drivers
[ ] /dev/kmem virtual device support
<M> Mini2440 module sample
<*> LED Support for Mini2440 GPIO LEDs
<*> Buttons driver for FriendlyARM Mini2440 development boards
<*> Buzzer driver for FriendlyARM Mini2440 development boards
[*] ADC driver for FriendlyARM Mini2440 development boards
[ ] Non-standard serial port support
    Serial drivers --->
-- Unix98 PTY support
[ ] Support multiple instances of devpts
[ ] Legacy (BSD) PTY support
< > IPMI top-level message handler --->
< > Hardware Random Number Generator Core support
< > Siemens R3964 line discipline
< > RAW driver (/dev/raw/rawN)
< > TPM Hardware Support --->

```

Sau khi chọn xong ta lưu lại cấu hình đã chọn.



Bước 4:

Thực hiện biên dịch nhân hệ điều hành:

make

```
ls -l arch/arm/boot
SYSPMAP System.map
SYSPMAP .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
GZIP arch/arm/boot/compressed/piggy.gz
AS arch/arm/boot/compressed/piggy.o
LD arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
Building modules, stage 2.
MODPOST 16 modules
embedded@ubuntu:~/Downloads/linux-2.6.32.2$
```

A screenshot of a terminal window with a dark background. On the left side, there is a vertical toolbar with several icons: a blue sphere, a white square with a black border, a white square with a black diagonal line, a white square with a black grid, and a trash can icon. The main area of the terminal shows the output of a "make" command. It includes file listing commands like "ls -l", compilation steps like "OBJCOPY", assembly steps like "AS", linking steps like "LD", and final file creation steps like "OBJCOPY" and "Kernel: arch/arm/boot/zImage is ready". It also shows the progress of building modules and the number of modules MODPOSTed. The prompt at the bottom is "embedded@ubuntu:~/Downloads/linux-2.6.32.2\$".

Khi biên dịch thành công thì file ảnh (zImage) của hệ điều hành nằm trong thư mục linux-2.6.32.2/arch/arm/boot/zImage. File này được sử dụng để nạp lên kit.

***Biên dịch nhân dùng để biên dịch driver:**

Để lập trình driver ta phải biên dịch nhân tạo ra kernel build system (gồm các header file). Thực hiện sau bước 2:

make modules

sudo make modules_install

```
embedded@ubuntu:~/Downloads/linux-2.6.32.2$ make modules
    CHK    include/linux/version.h
make[1]: `include/asm-arm/mach-types.h' is up to date.
    CHK    include/linux/utsrelease.h
    SYMLINK include/asm -> include/asm-arm
    CALL   scripts/checksyscalls.sh
Building modules, stage 2.
MODPOST 16 modules
embedded@ubuntu:~/Downloads/linux-2.6.32.2$ sudo make modules_install
make: arm-linux-gcc: Command not found
    INSTALL drivers/char/mini2440_hello_module.ko
    INSTALL drivers/misc/eeprom/eeprom_93cx6.ko
    INSTALL drivers/net/wireless/ath/ar9170/ar9170usb.ko
    INSTALL drivers/net/wireless/ath/ath.ko
    INSTALL drivers/net/wireless/rt2x00/rt2500usb.ko
    INSTALL drivers/net/wireless/rt2x00/rt2800usb.ko
    INSTALL drivers/net/wireless/rt2x00/rt2x00lib.ko
    INSTALL drivers/net/wireless/rt2x00/rt2x00usb.ko
    INSTALL drivers/net/wireless/rt2x00/rt73usb.ko
    INSTALL drivers/net/wireless/rtl818x/rtl8187.ko
    INSTALL drivers/net/wireless/zd1201.ko
    INSTALL drivers/net/wireless/zd1211rw/zd1211rw.ko
    INSTALL drivers/scsi/scsi_wait_scan.ko
    INSTALL net/mac80211/mac80211.ko
    INSTALL net/wireless/cfg80211.ko
    INSTALL net/wireless/lib80211.ko
    DEPMOD 2.6.32.2-FriendlyARM
embedded@ubuntu:~/Downloads/linux-2.6.32.2$ 
```

Sau khi thực hiện sau tạo ra kernel build nằm trong thư mục /lib/modules.

1.9 Câu hỏi ôn tập chương 1:

Bài 1: Trình bày sự giống nhau và khác nhau giữa hệ điều hành linux của máy host và kit nhúng.

Bài 2: Nêu ý nghĩa của trình biên dịch chéo arm-linux-gcc.

Bài 3: Vì sao phải biên dịch lại nhân của hệ điều hành.

HỆ THỐNG NHỦNG (TH)

CHƯƠNG 2: LẬP TRÌNH ĐIỀU KHIỂN LED

2.1 Mục đích:

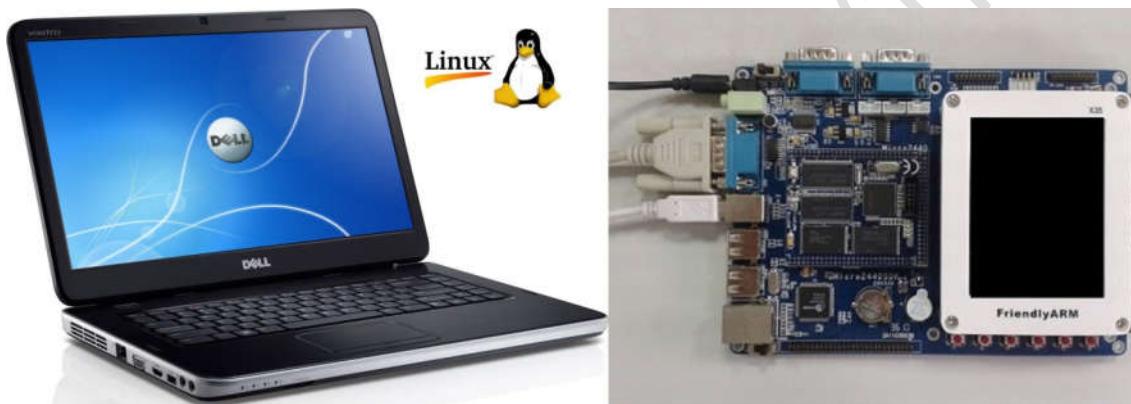
- Hiểu được mô hình giao tiếp từ app-device led
- Viết được các ứng dụng giao tiếp gpio với led sử dụng gpio driver led có sẵn
- Biên dịch, nạp và chạy chương trình trên kit

2.2 Yêu cầu:

- Hiểu được các điều khiển led sáng tắt
- Có kiến thức về lập trình C
- Có kiến thức về linux.

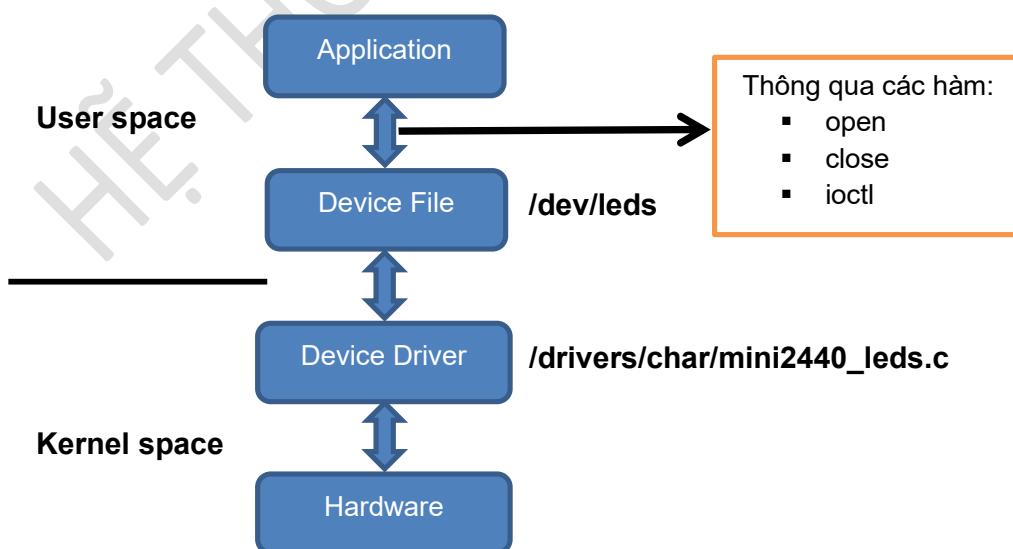
2.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN



Hình 2. 1

2.4 Mô hình giao tiếp App-Device leds:



Hình 2. 2

❖ Device file:

Không phải là một file thông thường, không phải là một vùng dữ liệu trên hệ thống file.

- Lệnh kiểm tra: ls -l /dev
- Quy trình đọc ghi device file
 - Giao tiếp với device driver
 - Đọc ghi phần cứng của thiết bị
- Các loại device files:
 - Character device: đọc ghi một chuỗi các byte dữ liệu

Ví dụ:

```
-crw-rw-rw  1 root      root  5, 0 Jan 27 2015 tty  
-crw-rw---- 1 root      tty    4, 0 Jan 27 2015 tty0  
-crw-rw---- 1 root      tty    4, 1 Jan 27 2015 tty1
```

- Block device: đọc và ghi một khối dữ liệu

Ví dụ:

```
brw-rw---- 1 root      root  31, 0 Jan 27 2015 mtdblock0  
brw-rw---- 1 root      root  31, 1 Jan 27 2015 mtdblock1  
brw-rw---- 1 root      root  31, 2 Jan 27 2015 mtdblock2
```

❖ Device number:

Mỗi device được xác định bởi hai số định danh là major number, minor number

- Major number: xác định thiết bị này sử dụng driver nào.
- Minor number: phân biệt giữa các thiết bị khác nhau cùng sử dụng chung một device driver.

Ví dụ:

```
brw-rw---- 1 root      root  31, 0 Jan 27 2015 mtdblock0
```

Loại thiết bị: block device: brw-rw----

Tài khoản người dùng: root

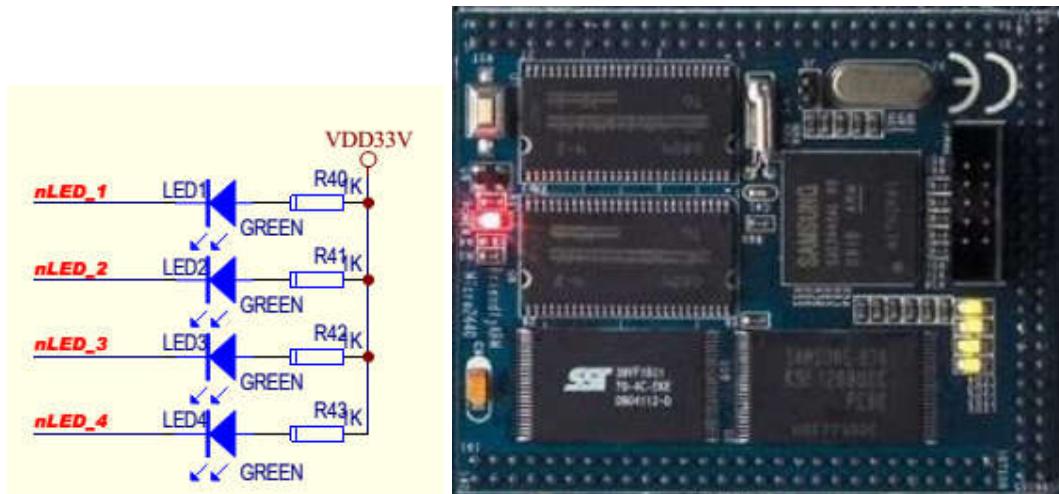
Major và minor number: 31,0

❖ Kiểm tra nhóm thiết bị:

```
#cat /proc/devices
```

2.5 Lập trình giao tiếp với led đơn:

4 led đơn được nối vào vi điều khiển qua 4 chân GPIO5,6,7,8 như sơ đồ hình dưới
Việc lập trình led đơn dựa trên driver sẵn có trên embedded linux khi cài đặt.



Hình 2.3

2.5.1 Viết chương trình điều khiển cho led 0 sáng tắt theo chu kỳ trong thời gian 1 giây:

Code:

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define ON 1
#define OFF 0

int main(int argc, char **argv)
{
```

```

int fd;

fd = open("/dev/leds", 0);

if (fd < 0)

{

    perror("open device leds");

    exit(1);

}

while(1)

{

    ioctl(fd, ON, 0);

    sleep(1);

    ioctl(fd, OFF, 0);

    sleep(1);

}

close(fd);

return 0;
}

```

- **Giải thích:**

Đoạn code trên sử dụng các hàm open, close, ioctl, sleep để thực hiện chương trình.

-Hàm **open**:

Header file	#include<fcntl.h>
Cấu trúc	int open(const char *path, int oflag,...);
Giá trị trả về	-1: nếu device mở không thành công ≥0: nếu device mở thành công
Mô tả	Mở device
Oflag	O_RDONLY(0): mở chỉ đọc O_WRONLY(1): mở chỉ ghi O_RDWR(2): mở để đọc và ghi

--	--

-Hàm **close**:

Header file	#include<unistd.h>
Cáu trúc	int close(int fd);
Giá trị trả về	-1: nếu device đóng không thành công ≥0: nếu device đóng thành công
Mô tả	Đóng device

-Hàm **ioctl**:

Cáu trúc: int ioctl(struct file * filp, unsigned int cmd, unsigned long arg)

filp: tên file mô tả device

cmd: command

arg: argument

Ví dụ: hàm ioctl(fd,on_off,led_no);

- o on_off=1 → led sáng, on_off=0 → led tắt
- o led_no: vị trí led có số từ 0 → 3

-Hàm **sleep(s)**, **usleep(us)**:

Header file: #include<unistd.h>

sleep(s): tạo trễ với thời gian s giây

usleep(us): tạo trễ với thời gian us micro giây

2.5.2 Biên dịch và nạp xuống kit

Dùng trình biên dịch chéo **arm-linux-gcc** để biên dịch và nạp xuống kit theo hướng dẫn ở chương 1. Mặc định thì chương trình chạy led-player do đó để tránh xung đột ta phải tắt chương trình led-player này. Có hai cách để tắt:

Cách 1:

Dùng lệnh tắt tạm thời. Khi reset kit thì chương trình led-player vẫn chạy.

#/etc/rc.d/init.d/leds stop

Cách 2:

Chỉnh sửa chương trình khi reset không load leds start.

#vi /etc/init.d/rcS

- Nhấn a để chỉnh sửa.
- Tìm đến dòng chứa /etc/rc/d/init.d/leds start chèn ký tự '#' vào trước dòng lệnh này.
- Nhấn thoát và lưu chỉnh sửa này(ESC :wq).
- Reset lại kit.

2.6 Bài tập thực hành:

Bài 1: Viết chương trình điều khiển 4 led đơn chớp tắt với chu kỳ 500ms

Bài 2: Viết chương trình cho 4 led đếm lên theo giá trị nhị phân từ 0000→1111 rồi tắt với thời gian 300ms

Bài 3: Viết chương trình điều khiển 4 led đơn chạy đuôi với chu kỳ 400ms

Bài 4: Viết chương trình dùng bàn phím để xem trạng thái và điều khiển 4 led đơn

- ✓ Nhấn phím ‘s’ sẽ hiển thị trạng thái 4 led trên terminal.

Ví dụ: led 1: on, led 2: off, led 3: on, led 4: off

- ✓ Nhấn phím 1: đảo trạng thái hiện tại của led 1(led 1: on khi nhấn phím ‘1’ sẽ off và ngược lại)
- ✓ Nhấn phím 2: đảo trạng thái hiện tại của led 2(led 2: on khi nhấn phím ‘2’ sẽ off và ngược lại)
- ✓ Nhấn phím 3: đảo trạng thái hiện tại của led 3(led 3: on khi nhấn phím ‘3’ sẽ off và ngược lại)
- ✓ Nhấn phím 4: đảo trạng thái hiện tại của led 4(led 4: on khi nhấn phím ‘4’ sẽ off và ngược lại)

***Ghi chú:** Mặc định 4 led đều tắt

Bài 5: Viết chương trình dùng bàn phím để điều khiển 4 led đơn.

- ✓ Nhấn phím 1: 4 led chạy đuôi theo chu kỳ với thời gian 300ms
- ✓ Nhấn phím 2: 4 led sáng tắt theo chu kỳ với thời gian 300ms
- ✓ Nhấn phím 3: Dừng trạng thái đang chạy

***Ghi chú:** Mặc định 4 led đều tắt

CHƯƠNG 3: LẬP TRÌNH ĐIỀU KHIỂN NÚT NHẤN

3.1 Mục đích:

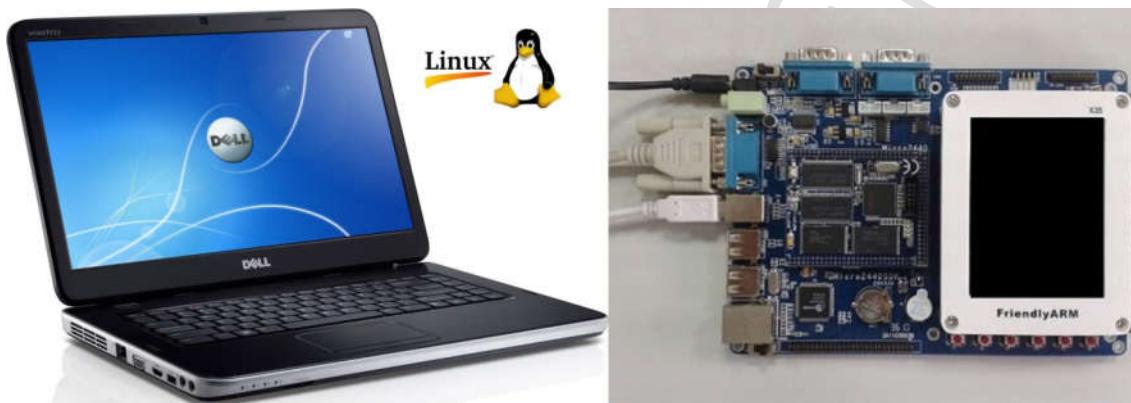
- Hiểu được mô hình giao tiếp từ app-device nút nhấn.
- Viết được các ứng dụng giao tiếp gpio với nút nhấn sử dụng gpio driver nút nhấn có sẵn.
- Biên dịch, nạp và chạy chương trình trên kit.

3.2 Yêu cầu:

- Phải hiểu rõ phần cứng và cách lập trình điều khiển nút nhấn để biết trạng thái nút nhấn khi nhấn và khi nhả
- Có kiến thức về lập trình C
- Có kiến thức về linux

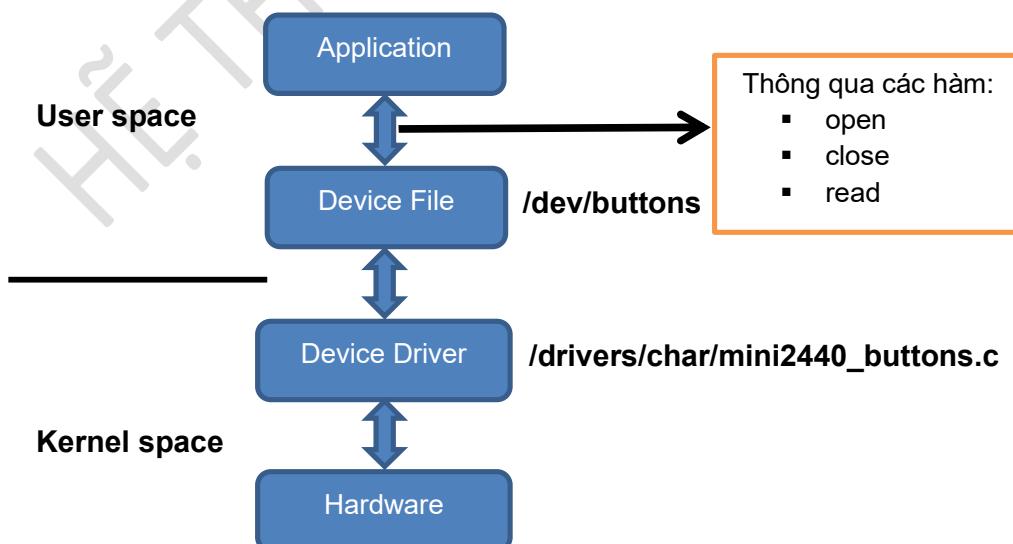
3.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN.



Hình 3. 1

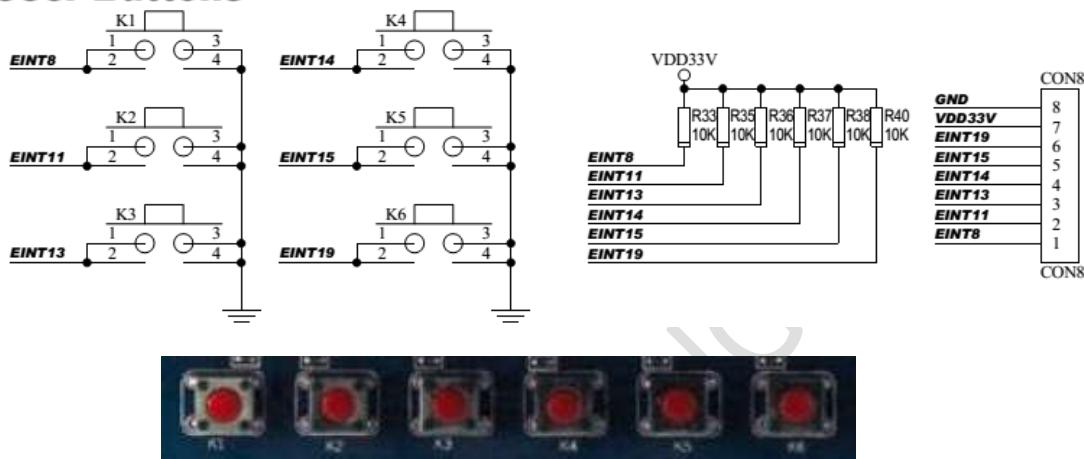
3.4 Mô hình giao tiếp App-Device button:



Hình 3. 2

3.5 Lập trình giao tiếp với nút nhấn:

6 nút nhấn được nối vào vi điều khiển qua 6 chân EINT8, EINT11, EINT13, EINT14, EINT15, EINT19 như sơ đồ hình dưới. Việc lập trình nút nhấn dựa trên driver sẵn có trên embedded linux khi cài đặt.

User Buttons**Hình 3. 3**

3.5.1 Viết chương trình kiểm tra trạng thái các nút nhấn và hiển thị lên màn hình:

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>
```

```

int main(int argc, char** argv)
{
    int buttons_fd;
    char buttons[6] = {'0', '0', '0', '0', '0', '0'}; //Mảng lưu trạng thái của 6 nút bấm
    buttons_fd = open("/dev/buttons", 0); //Mở file
    if (buttons_fd < 0) //Kiểm tra quá trình mở file
    {
        perror("open device buttons");
        exit(1);
    }
    while(1) //Hỏi vòng kiểm tra trạng thái các nút bấm
    {
        char current_buttons[6];
        int count_of_changed_key;
        int i;
        //Đọc trạng thái các nút bấm
        if (read(buttons_fd, current_buttons, sizeof current_buttons) != sizeof
current_buttons)
        {
            perror("read buttons:");
            exit(1);
        }
        //Kiểm tra trạng thái các nút bấm và in ra trạng thái phù hợp (Key up
hay Key down)
        for (i = 0, count_of_changed_key = 0; i < sizeof buttons / sizeof
buttons[0]; i++)

```

```

    {
        if(buttons[i] != current_buttons[i])
        {
            buttons[i] = current_buttons[i];
            printf("key %d is %s", i+1, buttons[i] == '0' ? "up" :
"down");
            count_of_changed_key++;
        }
        if(count_of_changed_key) {
            printf("\n");
        }
    }
    close(buttons_fd); //Đóng file thiết bị
    return 0;
}

```

- **Giải thích:**

Đoạn code trên sử dụng các hàm open, close, ioctl, sleep để thực hiện chương trình

-Hàm **open**:

Header file	#include<fcntl.h>
Cấu trúc	int open(const char *path, int oflag,...);
Giá trị trả về	-1: nếu device mở không thành công ≥0: nếu device mở thành công
Mô tả	Mở device
Oflag	O_RDONLY(0): mở chỉ đọc O_WRONLY(1): mở chỉ ghi O_RDWR(2): mở để đọc và ghi

-Hàm **close**:

Header file	#include<unistd.h>
Câu trúc	int close(int fd);
Giá trị trả về	-1: nếu device đóng không thành công ≥0: nếu device đóng thành công
Mô tả	Đóng device

-Hàm **read**:

Câu trúc: read(buttons_fd, current_buttons, sizeof current_buttons)

- buttons_fd: tên file mô tả device
- current_buttons: trạng thái của các nút tương ứng
 - ✓ '1': key down
 - ✓ '0': key up
- current_buttons: số byte cần đọc

3.5.2 Biên dịch và nạp xuống kit:

Dùng trình biên dịch chéo **arm-linux-gcc** để biên dịch và nạp xuống kit theo hướng dẫn ở chương 1.

```

embedded@ubuntu: ~
Escape character is '^].
Kernel 2.6.32.2-FriendlyARM on (/dev/pts/1)
FriendlyARM login: root
Password:
[root@FriendlyARM /]# chmod 777 bt1
[root@FriendlyARM /]# ./bt1
key 1 is down
key 1 is up
key 2 is down
key 2 is up
key 2 is down
key 2 is up
key 3 is down
key 3 is up
key 4 is down
key 4 is up
key 4 is down

```

Hình 3. 4

3.6 Bài tập thực hành:

Bài 1: Viết chương trình điều khiển 4 nút nhấn K1,K2,K3,K4 với 4 led đơn tương ứng. Biết rằng khi nhấn nút K1,K2,K3,K4 thì 4 led tương ứng sáng khi nhả phím 4 led tắt.

***Ghi chú:** Mặc định 4 led đều tắt

Bài 2: Viết chương trình điều khiển 4 nút nhấn K1,K2,K3,K4 với 4 led đơn tương ứng. Biết rằng khi nhấn nút K1,K2,K3,K4 thì 4 led tương ứng sẽ đảo trạng thái (sáng thành tắt và tắt thành sáng)

***Ghi chú:** Mặc định 4 led đều tắt

Bài 3: Viết chương trình điều khiển 4 nút nhấn K1,K2,K3, K4 với 4 led đơn tương ứng.

-Nhấn nút K1 4 led sáng/tắt theo chu kỳ với $t=500\text{ms}$ lặp lại trong 2 lần

-Nhấn nút K2 4 led chạy đuôi theo chu kỳ với $t=500\text{ms}$ lặp lại trong 3 lần

-Nhấn nút K3 4 led đếm lên nhị phân với $t=500\text{ms}$ lặp lại 1 lần

-Nhấn nút K4 4 led sáng dần từ trên xuống dưới với $t=500\text{ms}$ lặp lại 1 lần

***Ghi chú:** Mặc định 4 led đều tắt.

CHƯƠNG 4: LẬP TRÌNH ĐIỀU CHẾ ĐỘ RỘNG XUNG PWM

4.1 Mục đích:

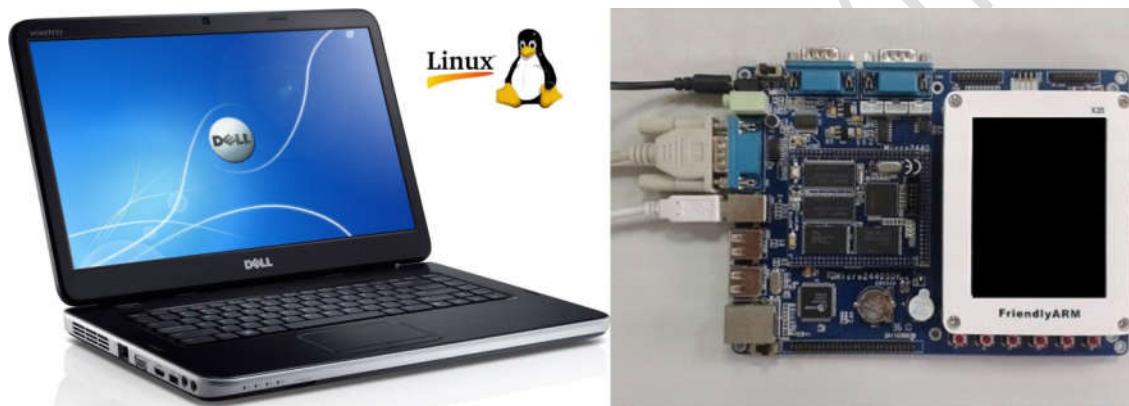
- Hiểu được mô hình giao tiếp từ app-device PWM.
- Viết được các ứng dụng giao tiếp PWM sử dụng driver PWM có sẵn.
- Biên dịch, nạp và chạy chương trình trên kit.

4.2 Yêu cầu:

- Hiểu về điều chế độ rộng xung PWM cũng như sơ đồ phần cứng PWM kit
- Có kiến thức về lập trình C
- Có kiến thức về linux

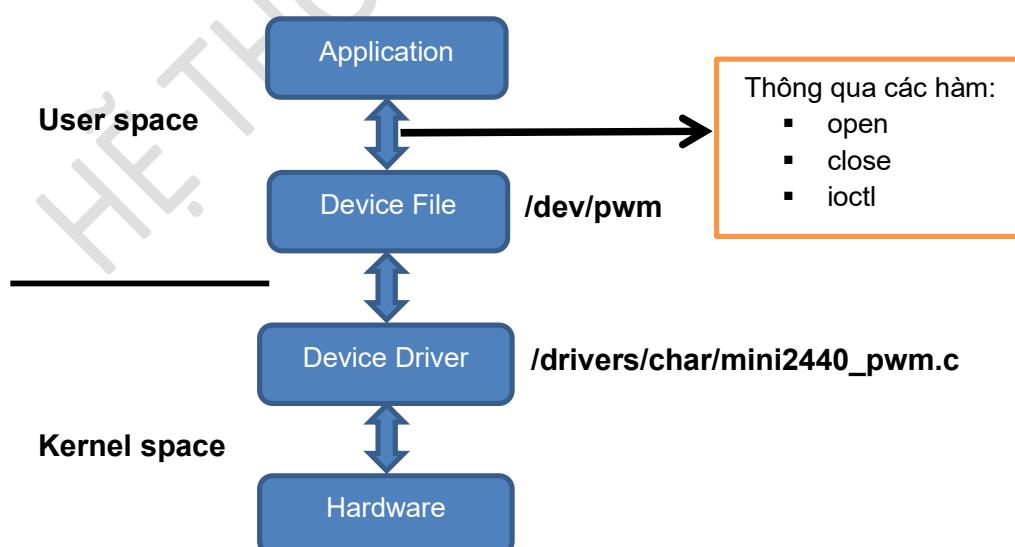
4.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN



Hình 4. 1

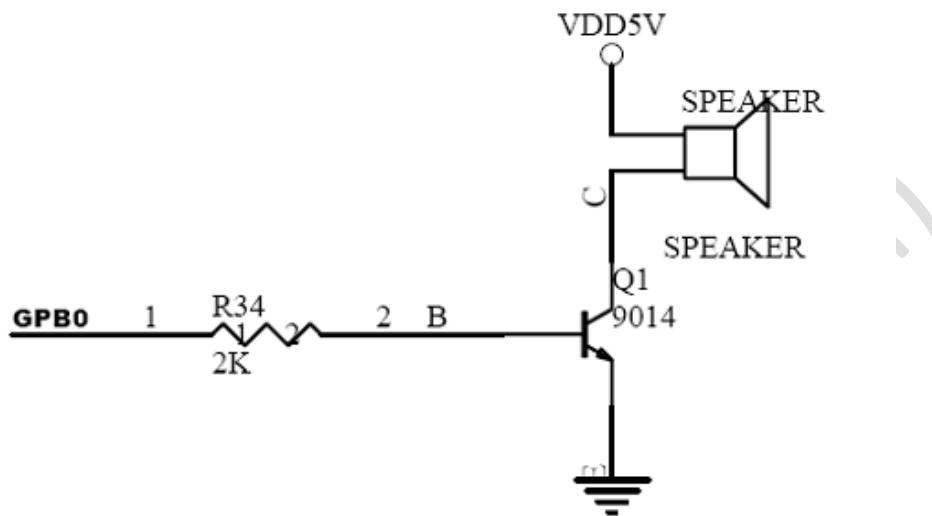
4.4 Mô hình giao tiếp App-Device PWM:



Hình 4. 2

4.5 Lập trình giao tiếp với PWM buzzer:

Buzzer được điều khiển bởi tín hiệu PWM. PWM được nối với chân GPB0. Ta lập trình điều khiển xuất tín hiệu ra ở chân GPB0 để tạo ra các tín hiệu âm thanh khác nhau.



Hình 4.3

4.5.1 Viết chương trình điều khiển tần số PWM qua 2 phím ‘+’ và ‘-’ trên bàn phím. Khi nhấn phím ‘+’ thì tần số tăng 100 và phím ‘-’ tần số giảm 100. Nhấn nhấn phím ESC dừng chế độ PWM. Biết tần số mặc định ở chế độ PWM có tần số bằng 1000.

Code:

```
#include <stdio.h>
#include <termios.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define PWM_IOCTL_SET_FREQ 1
```

```

#define PWM_IOCTL_STOP      0
#define ESC_KEY    0x1b

void set_buzzer_freq(int fd, int freq)
{
    // this IOCTL command is the key to set frequency
    int ret = ioctl(fd, PWM_IOCTL_SET_FREQ, freq);
    if(ret < 0)
    {
        perror("set the frequency of the buzzer");
        exit(1);
    }
}

void stop_buzzer(int fd)
{
    int ret = ioctl(fd, PWM_IOCTL_STOP);
    if(ret < 0)
    {
        perror("stop the buzzer");
        exit(1);
    }
    close(fd);
}

int main(int argc, char **argv)
{

```

```
int fd, freq = 1000 ;  
  
char key = 0xff;  
  
  
fd = open("/dev/pwm", 0);  
  
if (fd < 0)  
{  
    perror("open pwm_buzzer device");  
    exit(1);  
  
}  
  
  
printf( "\nBUZZER TEST ( PWM Control )\n" );  
printf( "Press 'ESC+ENTER' key to Exit this program\n\n" );  
set_buzzer_freq(fd, freq);  
printf( "\tFreq = %d\n", freq );  
  
  
while( 1 )  
{  
    key = getchar();  
    printf("key:%d\n",key);  
    sleep(1);  
    switch(key)  
    {  
  
        case ESC_KEY:  
            stop_buzzer(fd);  
            exit(0);  
    }  
}
```

```

case '+':
    freq += 100;
    printf( "\tFreq = %d\n", freq );
    set_buzzer_freq(fd, freq);
    break;

case '-':
    freq -= 100;
    if(freq<0) freq = 0;
    printf( "\tFreq = %d\n", freq );
    set_buzzer_freq(fd, freq);
    break;

default:
    printf("key:%d invalid\n",key);
    break;
}
}
}
}

```

- **Giải thích:**

Đoạn code trên sử dụng các hàm open, close, ioctl, sleep để thực hiện chương trình

-Hàm **open**:

Header file	#include<fcntl.h>
Cấu trúc	int open(const char *path, int oflag,...);
Giá trị trả về	-1: nếu device mở không thành công ≥0: nếu device mở thành công
Mô tả	Mở device
Oflag	O_RDONLY(0): mở chỉ đọc O_WRONLY(1): mở chỉ ghi O_RDWR(2): mở để đọc và ghi

--	--

-Hàm **close**:

Header file	#include<unistd.h>
Cáu trúc	int close(int fd);
Giá trị trả về	-1: nếu device đóng không thành công ≥0: nếu device đóng thành công
Mô tả	Đóng device

-Hàm **ioctl**:

Cáu trúc: ioctl(fd,1,f) hoặc ioctl(fd,0)

- ioctl(fd,1,f): thiết lập tần số f cho PWM. Giá trị tần số nghe được trong khoảng 20-20000Hz.
- ioctl(fd,0): tắt chế độ PWM.

4.5.2 Biên dịch và nạp xuống kit:

Dùng trình biên dịch chéo **arm-linux-gcc** để biên dịch và nạp xuống kit theo hướng dẫn ở chương 1.

```

Terminal
[embedded@ubuntu: ~/Documents]
[root@FriendlyARM /]# ./pwm
BUZZER TEST ( PWM Control )
Press 'ESC+ENTER' key to Exit this program

        Freq = 1000
+
key:43      Freq = 1100
key:10 invalid
-
key:45      Freq = 1000
key:10 invalid
^[
key:27
[root@FriendlyARM /]#

```

4.6 Bài tập thực hành:

Bài 1: Viết ứng dụng về xung PWM. Cứ 2s thì tần số xung tăng lên 100Hz biết tần số xung ban đầu là 500Hz và tăng đến khi 2000Hz thì dừng chế độ PWM.

Bài 2: Viết ứng dụng nút nhấn kết hợp với xung PWM.

- ✓ Nhấn K1(K1: down) tạo tần số xung 500Hz
- ✓ Nhấn K2(K2: down) tạo tần số xung 1000Hz
- ✓ Nhấn K3(K3: down) tạo tần số xung 1500Hz
- ✓ Trạng thái reset hay nút nhấn nhả(Ki: up i:0→2) thì stop chế độ PWM

Bài 3: Viết ứng dụng nút nhấn kết hợp với xung PWM. Mặc định tần số xung PWM=1000.

- ✓ Nhấn nhả K1 tần số xung tăng 100Hz
- ✓ Nhấn nhả K2 tần số xung tăng 50Hz
- ✓ Nhấn nhả K3 tần số xung giảm 100Hz
- ✓ Nhấn nhả K4 tần số xung giảm 50Hz
- ✓ Nhấn nhả K5 stop chế độ PWM

Biết rằng tần số xung tối đa là 2000Hz và tối thiểu là 1Hz.

CHƯƠNG 5: LẬP TRÌNH CHUYỂN ĐỔI ADC

5.1 Mục đích:

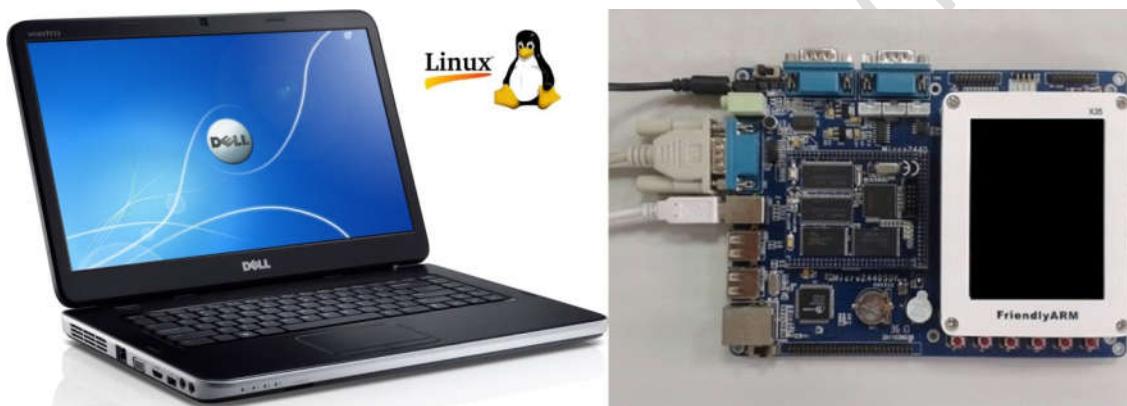
- Hiểu được mô hình giao tiếp từ app-device ADC.
- Viết được các ứng dụng chuyển đổi sử dụng driver ADC có sẵn.
- Biên dịch, nạp và chạy chương trình trên kit.

5.2 Yêu cầu:

- Hiểu được các chuyển đổi tương tự sang số ADC cũng như sơ đồ phần cứng ADC kit.
- Có kiến thức về lập trình C
- Có kiến thức về linux

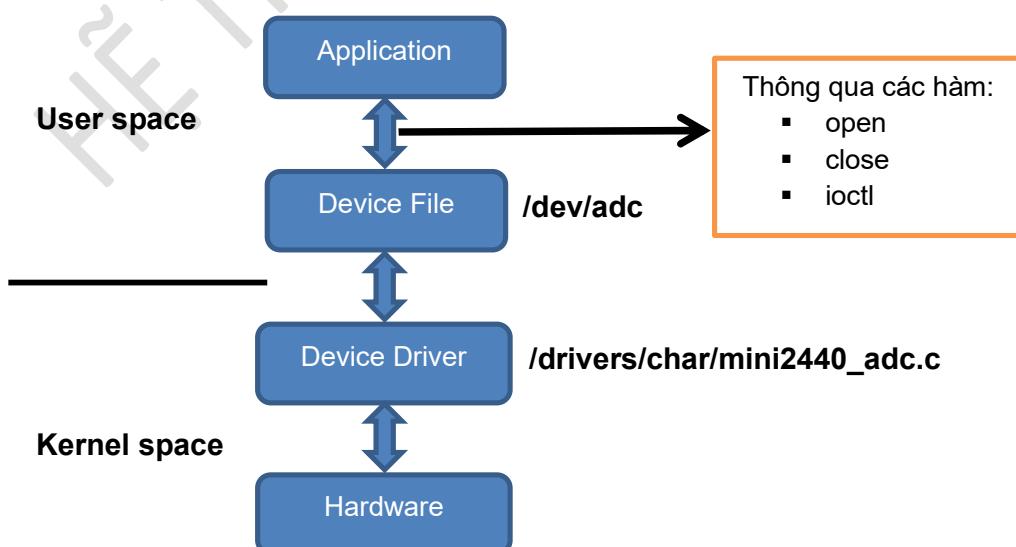
5.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN



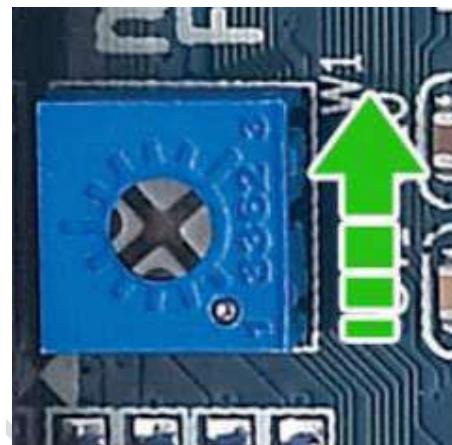
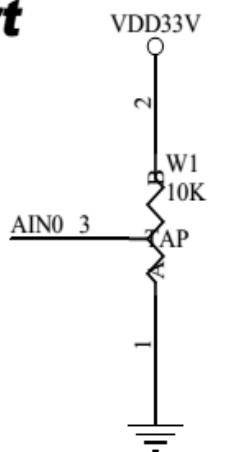
Hình 5.1

5.4 Mô hình giao tiếp App-Device ADC:



Hình 5. 2**5.5 Lập trình chuyển đổi ADC:**

Kit Arm mini2440 có 4 kênh chuyển đổi AD (CON4-GPIO) cho những mục đích khác nhau. Tuy nhiên chỉ có kênh AIN0 được kết nối qua biến trở để chuyển đổi AD.

AD Convert**Hình 5. 3**

5.5.1 Viết chương trình liên tục đọc giá trị ADC từ biến trở W1 qua chân AIN0. Kết quả đọc được (0-1023) hiển thị lên màn hình. Chỉnh biến trở để quan sát kết quả.

Code:

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <linux/fs.h>
#include <errno.h>
#include <string.h>

```

```
int main(void)
{
    printf("Press Ctrl-C to stop\n");
    int fd = open("/dev/adc", 0);
    if (fd < 0)
    {
        perror("open ADC device:");
        return 1;
    }
    while(1)
    {
        char buffer[30];
        int len = read(fd, buffer, sizeof buffer -1);
        if (len > 0)
        {
            buffer[len] = '\0';
            int value = -1;
            sscanf(buffer, "%d", &value); //doc du lieu theo dinh dang
            printf("ADC Value: %d\n", value);
        }
        else
        {
            perror("read ADC device:");
            return 1;
        }
    }
}
```

```

    usleep(500*1000);

}

close(fd);

}

```

- **Giải thích:**

Đoạn code trên sử dụng các hàm open, close, ioctl, sleep để thực hiện chương trình

-Hàm **open**:

Header file	#include<fcntl.h>
Cấu trúc	int open(const char *path, int oflag,...);
Giá trị trả về	-1: nếu device mở không thành công ≥0: nếu device mở thành công
Mô tả	Mở device
Oflag	O_RDONLY(0): mở chỉ đọc O_WRONLY(1): mở chỉ ghi O_RDWR(2): mở để đọc và ghi

-Hàm **close**:

Header file	#include<unistd.h>
Cấu trúc	int close(int fd);
Giá trị trả về	-1: nếu device đóng không thành công ≥0: nếu device đóng thành công
Mô tả	Đóng device

-Hàm **read**:

Cấu trúc: num=read(fd,buffer,sizeof buffer-1)

- fd: tên file mô tả device.
- buffer: là một mảng ký tự(chuỗi)
- num: chiều dài chuỗi đọc được

-Hàm **sscanf**:

Là hàm đọc dữ liệu theo định dạng từ string đầu vào và lưu vào các biến.

Cấu trúc:

```
int sscanf(const char *buffer, const char *format, argument)
```

Tham số:

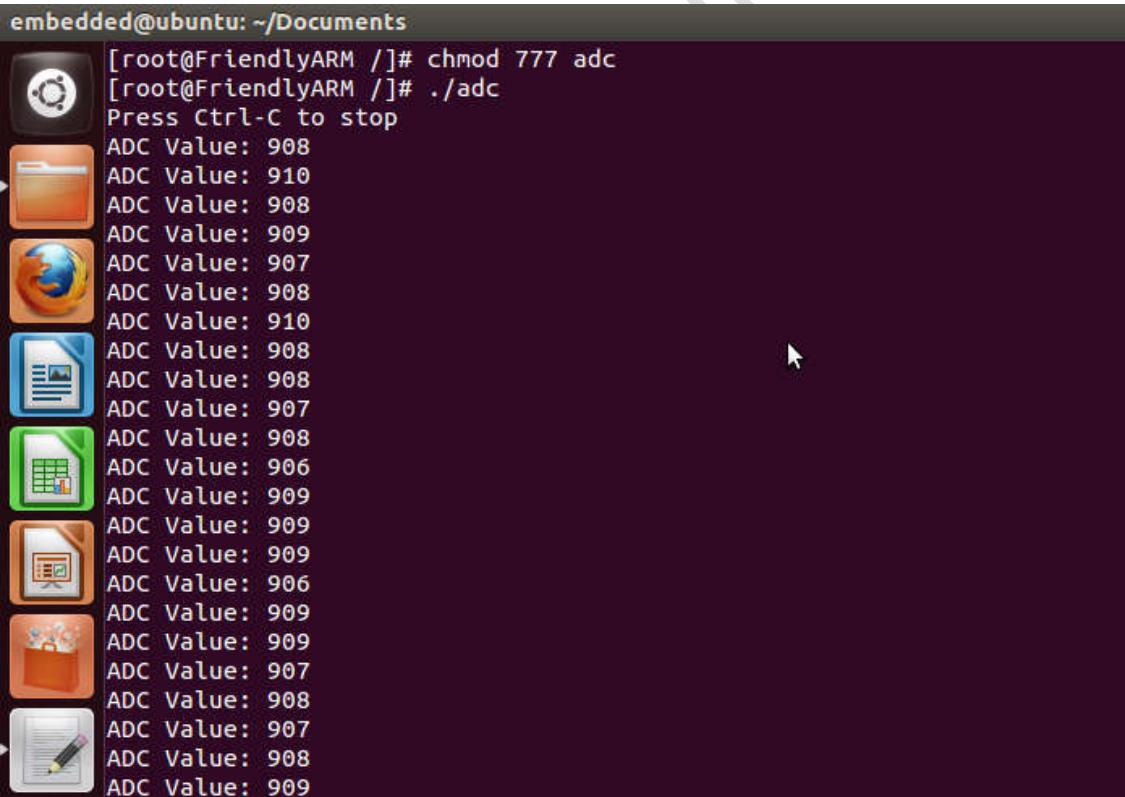
- buffer: dữ liệu đầu vào
- format: string định dạng dữ liệu
- argument: biến lưu trữ

Ý nghĩa:

- Hàm trả về trường dữ liệu buffer nếu đọc thành công
- Hàm trả về EOF nếu xảy ra lỗi hoặc kết thúc buffer đầu vào
- Hàm trả về -1 nếu buffer đầu vào NULL

5.5.2 Biên dịch và nạp xuống kit:

Dùng trình biên dịch chéo **arm-linux-gcc** để biên dịch và nạp xuống kit theo hướng dẫn ở chương 1.



```
embedded@ubuntu: ~/Documents
[root@FriendlyARM /]# chmod 777 adc
[root@FriendlyARM /]# ./adc
Press Ctrl-C to stop
ADC Value: 908
ADC Value: 910
ADC Value: 908
ADC Value: 909
ADC Value: 907
ADC Value: 908
ADC Value: 910
ADC Value: 908
ADC Value: 908
ADC Value: 907
ADC Value: 908
ADC Value: 906
ADC Value: 909
ADC Value: 909
ADC Value: 909
ADC Value: 906
ADC Value: 909
ADC Value: 909
ADC Value: 907
ADC Value: 908
ADC Value: 907
ADC Value: 908
ADC Value: 909
```

5.6 Bài tập thực hành:

Bài 1: Viết ứng dụng chuyển đổi adc kết hợp với led. Chương trình liên tục đọc kết quả chuyển đổi **adc**(**adc_value**) hiển thị led.

- $0 \leq \text{adc_value} < 128$: led 1 sáng, các led còn lại tắt
- $128 \leq \text{adc_value} < 256$: led 2 sáng, các led còn lại tắt
- $256 \leq \text{adc_value} < 512$: led 3 sáng, các led còn lại tắt
- $512 \leq \text{adc_value} \leq 1023$: led 4 sáng, các led còn lại tắt

Bài 2: Viết ứng dụng chuyển đổi adc kết hợp với điều chế xung PWM. Chương trình liên tục đọc kết quả chuyển đổi **adc**(**adc_value**) và xuất ra xung PWM.

- $0 \leq \text{adc_value} < 400$: xuất ra xung có tần số 400Hz
- $400 \leq \text{adc_value} < 800$: xuất ra xung có tần số 800Hz
- $800 \leq \text{adc_value} \leq 1023$: xuất ra xung có tần số 1200Hz

Bài 3: Viết ứng dụng chuyển đổi adc kết hợp với điều chế xung PWM. Chương trình liên tục đọc kết quả chuyển đổi **adc**(**adc_value**) và xuất ra xung PWM với tần số xung PWM bằng với **adc_value**.

CHƯƠNG 6: HỆ THỐNG FILE, THƯ MỤC TRONG LINUX

6.1 Mục đích:

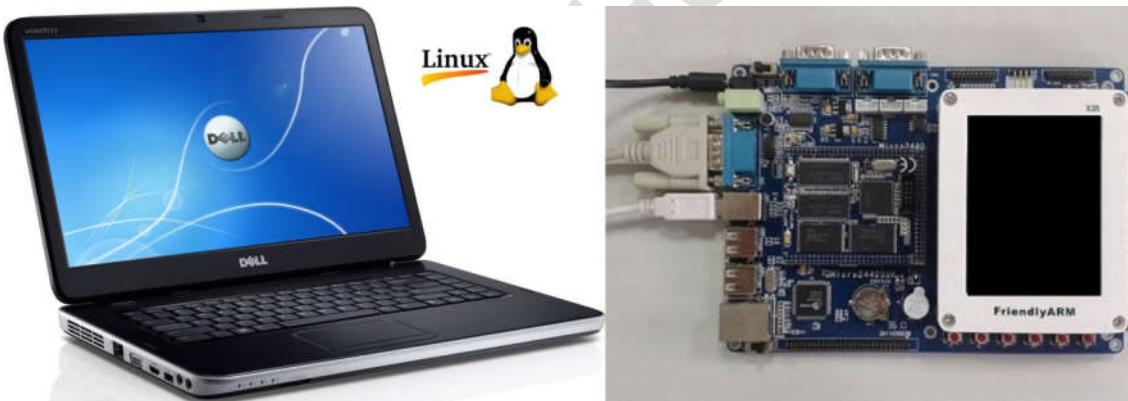
- Hiểu được cấu trúc hệ thống file, thư mục trong Linux
- Hiểu được ý nghĩa các hàm dùng để xử lý file như fputc, fgetc, fputs, fgets, fread, fwrite, fseek
- Viết được các ứng dụng về xử lý file trên nền embedded linux.
- Biên dịch, nạp và chạy chương trình trên kit.

6.2 Yêu cầu:

- Có kiến thức về file, thư mục
- Lập trình giao tiếp với nút nhấn(học ở chương 3)
- Có kiến thức về lập trình C
- Có kiến thức về linux.

6.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN



Hình 6.1

6.4 Khái quát về cấu trúc hệ thống file trong linux:

Trong hệ thống Linux tất cả file cộng với thư mục được lưu theo một cấu trúc hình cây duy nhất. Gốc của cây thư mục này được gọi là root và ký hiệu bằng ký tự số trái /. Khi bạn tạo mới hay chép 1 file nào đó vào trong thư mục bạn phải có quyền w(write) trên đó. Khi bạn muốn mở hay muốn đọc nội dung thư mục(bằng lệnh ls) bạn phải có quyền r(read). Nếu bạn muốn mở thư mục(dùng lệnh cd để chuyển) bạn phải có quyền x.

Các file và thư mục bình thường như: bai1.docx, test.c, content.txt(giống win), /home/user/document...

Các kiểu file đặc biệt:

- Biểu tượng liên kết (Symbolic link) file: file này chứa các thông tin liên kết đến 1 file thực thụ khác(giống khái niệm shortcut trong windows)
- File thiết bị dạng ký tự (character) hay khối (block). Các file này thường là file thiết bị(device file) chủ yếu được lưu trong thư mục /dev.
- Các file được tạo ra nhằm hỗ trợ giao tiếp liên tiến trình như pipe, socket...

Cách truy xuất file trong linux:

Thông qua thư viện chuẩn: #include<stdio.h>

Thông qua các hàm truy xuất cấp thấp

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

6.4.1 Thư viện xuất nhập chuẩn(standard I/O Library):

Thư viện các hàm xuất nhập chuẩn của C được khai báo trong file include<stdio.h>. Thư viện theo chuẩn ANSI C nên có thể sử dụng nó trong mọi hệ điều hành. Các hàm xử lý file trong thư viện bao gồm:

- fopen, fclose
- fread, fwrite
- fflush
- fseek
- fgetc, getc, getchar
- fputc, putc, putchar
- fgets, gets
- printf, fprintf, sprintf
- scanf, fscanf, sscanf

6.4.2 Đóng mở file:

Mở file:

fopen(const char *filename, const char *mode);

*filename: đường dẫn đến file

*mode: chế độ mở file “r”, “w”, “rw”, “a”.

Hàm trả về NULL nếu mở không thành công.

Ví dụ:

FILE *f_read,*f_write,*f_readwrite,*f_append;

```

/* Mở file chỉ để đọc */
f_read=fopen("Document/data_read.txt","r");
/* Mở file để ghi nếu file không tồn tại thì tạo file mới. Nếu file tồn tại thì nội dung của nó sẽ bị xóa*/
f_write=fopen("Document/data_write.txt","w");
/* Mở file để đọc/ghi nếu file tồn tại thì nội dung của nó sẽ bị ghi đè lên*/
f_readwrite=fopen("Document/data_readwrite.txt","rw");
/*Mở file để ghi nội dung vào cuối file*/
f_append=fopen("Document/data_a.txt","a");

```

Đóng file:

```
int fclose(FILE *stream);
```

Hàm trả về -1 nếu đóng file không thành công và trả về giá trị ≥ 0 nếu đóng file thành công.

6.4.3 Đọc dữ liệu từ file:**Đọc 1 ký tự**

```
int fgetc(FILE *stream);
```

Hàm trả về ký tự đọc được nếu thành công và EOF(end of file) nếu không thành công.

Ví dụ 1: Đếm xem file myfile.txt có bao nhiêu ký tự \$

```

/*fgetc example: money counter */
#include <stdio.h>
int main ()
{
    FILE * pFile;
    int c;
    int n = 0;
    pFile=fopen ("myfile.txt","r");
    if(pFile==NULL) perror ("Error opening file");
    else
    {
        do
        {
            c = fgetc (pFile);
            if(c == '$') n++;
        } while (c != EOF);
        fclose (pFile);
        printf ("The file contains %d dollar sign characters ($).\n",n);
    }
    return 0;
}

```

Đọc 1 khối dữ liệu

```
fread(void *ptr,size_t size,size_t count,FILE *stream);
```

Tham số:

- ptr: buffer chứa data đọc được
- size: kích thước mỗi phần tử
- count: số phần tử cần đọc
- stream: con trỏ file

Đọc 1 chuỗi kết thúc bằng ký tự xuống dòng + tự động thêm ký tự NULL vào cuối buffer đọc ra

```
char* fgets(char *str,int num,FILE *stream);
```

Tham số:

- str: biến con trỏ lưu mảng ký tự(chuỗi) đọc vào
- num: số ký tự lớn nhất được copy vào str
- stream: con trỏ file

Giá trị trả về:

- EOF nếu không thành công
- str chứa dữ liệu đọc được nếu thành công.

Ví dụ 2: Viết chương trình mở file myfile.txt để đọc 100 ký tự dòng đầu tiên và xuất ra màn hình

```
/*fgets example */
#include <stdio.h>

int main()
{
    FILE *pFile;
    char mystring [100];

    pFile = fopen ("myfile.txt" , "r");
    if (pFile == NULL) perror ("Error opening file");
    else
    {
        if (fgets (mystring , 100 , pFile) != NULL )
            puts (mystring);
        fclose (pFile);
    }
    return 0;
}
```

6.4.4 Ghi dữ liệu vào file:

Ghi 1 ký tự

int fputc(int character, FILE* stream)

Tham số:

- character: ký tự muốn ghi
- stream: con trỏ file

Giá trị trả về:

- EOF nếu không thành công
- Số ký tự được ghi nếu thành công

Ví dụ 3: Viết chương trình ghi các ký tự từ A→Z vào file alphabet.txt

```
/*fputc example: alphabet writer */
#include <stdio.h>
```

```
int main ()
{
    FILE * pFile;
    char c;

    pFile = fopen ("alphabet.txt", "w");
    if (pFile!=NULL)
    {
        for (c = 'A' ; c <= 'Z' ; c++)
            fputc ( c , pFile );
        fclose (pFile);
    }
    return 0;
}
```

Ghi 1 khối dữ liệu

fwrite (const void * ptr, size_t size, size_t count, FILE * stream);

Tham số:

- ptr: buffer chứa data ghi được
- size: kích thước kiểu dữ liệu mỗi phần tử được ghi
- count: số phần tử
- stream: con trỏ file

Ví dụ 4: Viết chương trình mở file file.txt để ghi nội dung vào file. Sau đó đọc file đã ghi. Kết quả đọc file sẽ xuất ra màn hình.

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

```
#include <string.h>
```

```

int main()
{
    FILE *fp;
    char c[] = "Example for function fread and fwrite";
    char buffer[100];

    /* Open file for both reading and writing */
    fp = fopen("file.txt", "rw");

    /* Write data to the file */
    fwrite(c, strlen(c) + 1, 1, fp);

    /* Seek to the beginning of the file */
    fseek(fp, SEEK_SET, 0);

    /* Read and display data */
    fread(buffer, strlen(c)+1, 1, fp);
    printf("%s\n", buffer);
    fclose(fp);

    return(0);
}

```

Ghi từ đầu chuỗi đến ký tự NULL(kết thúc chuỗi)

int fputs (const char * str, FILE * stream);

Tham số:

- str: chuỗi cần ghi
- stream: con trỏ file

Giá trị trả về:

- EOF nếu ghi không thành công
- Chuỗi được ghi vào file nếu thành công

Ví dụ 5: Viết chương trình đọc chuỗi ký tự nhập từ bàn phím sau đó mở file mylog.txt để tiếp tục ghi chuỗi ký tự đọc này vào cuối file.

```
/* fputs example */
#include <stdio.h>
int main ()
{
    FILE * pFile;
    char sentence [256];

    printf ("Enter sentence to append: ");
    fgets (sentence, 256, stdin);
    pFile = fopen ("mylog.txt", "a");
    fputs (sentence, pFile);
    fclose (pFile);
    return 0;
}
```

6.4.5 Di chuyển con trỏ file:

int fseek (FILE * stream, long int offset, int origin);

Tham số:

- stream: con trỏ file
- offset: vị trí tương đối di chuyển đến tính từ origin
- origin: vị trí tham chiếu lấy các giá trị sau
 - ✓ SEEK_SET : vị trí tính từ đầu file
 - ✓ SEEK_CUR: vị trí tính từ vị trí hiện tại
 - ✓ SEEK_END: vị trí tính từ cuối file.

Giá trị trả về:

- -1 không thành công
- 0 nếu thành công

Ví dụ 6: Xem nội dung file example.txt sau khi chạy đoạn code sau:

```
/* fseek example */
#include <stdio.h>

int main ()
{
    FILE * pFile;
    pFile = fopen ( "example.txt" , "wb" );
    fputs ( "This is an apple." , pFile );
    fseek ( pFile , 9 , SEEK_SET );
```

```
fputs( "sam", pFile );
fclose( pFile );
return 0;
}
```

HỆ THỐNG NHỦNG (TH)

6.5 Bài tập thực hành

Bài 1: Viết ứng dụng kết hợp giữa file và nút nhấn:

- Đếm số lần nút nhấn/nhả K1 và lưu vào file **text.txt**
- Nhấn/ nhả K2 để xem số lần nhấn nút K1
- Nhấn K3 để xóa số lần nhấn nút về 0

Bài 2: Viết ứng dụng kết hợp giữa file, nút nhấn, và led. Mở file **input.txt** để đọc, ghi.

- Nhấn K1 để xem trong file **input.txt** có bao nhiêu ký tự ‘a’ hiển thị lên màn hình đồng thời hiển thị số dư trong chia 16 của số ký tự ‘a’ lên 4 led đơn.
- Nhấn K2 để đổi tất cả ký tự ‘a’ thành ‘A’ trong file **input.txt**
- Nhấn nút K3 ghi thêm vào cuối file **input.txt** số ký tự a

CHƯƠNG 7: LẬP TRÌNH ĐA TUYẾN TRONG LINUX

7.1 Mục đích:

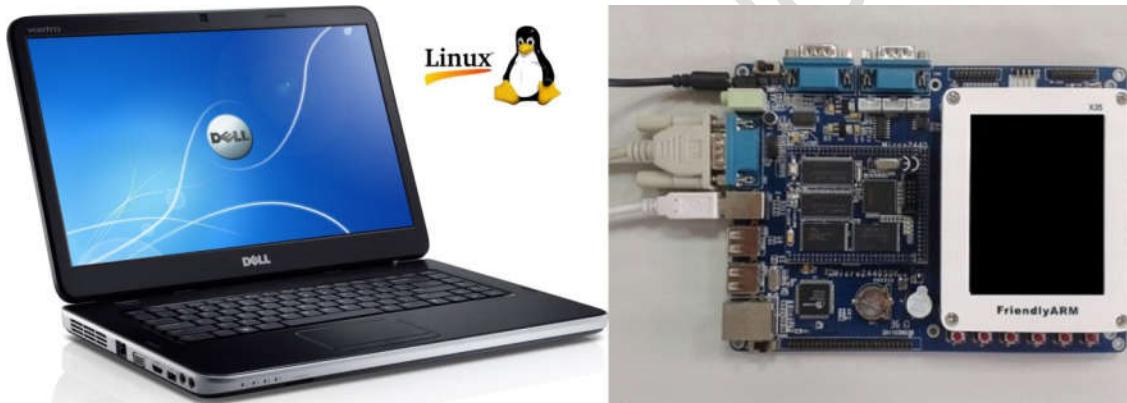
- Hiểu được ý nghĩa lập trình đa tuyến(multi-thread)
- Hiểu được cách tạo, hủy, thực thi, chờ, đồng bộ hóa tuyến với mutex, thay đổi thuộc tính, hủy bỏ, chấm dứt tuyến
- Áp dụng được lập trình đa tuyến với ứng dụng leds - buttons
- Biên dịch, nạp và chạy chương trình trên kit.

7.2 Yêu cầu:

- Lập trình giao tiếp với các ngoại vi: led, button, PWM, ADC
- Có kiến thức về lập trình C
- Có kiến thức về linux

7.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN



Hình 7.1

7.4 Khái quát về lập trình đa tuyến:

Nếu hệ điều hành có nhiều tiến trình thì bên trong mỗi tiến trình ta có thể tạo nhiều tuyến hoạt động song song với nhau tương tự như cách tiến trình hoạt động song song bên trong hệ điều hành. Ưu điểm của tuyến là chúng hoạt động trong cùng không gian địa chỉ của tiến trình. Tập hợp một nhóm các tuyến có thể cùng chia sẻ chung vùng nhớ của một tiến trình và do đó có thể sử dụng chung một biến toàn cục, vùng nhớ heap, bảng mô tả file.. của tiến trình.

Ưu điểm của sử dụng tuyến trong tiến trình hơn hẳn cách lập trình tuần tự. Nhiều thao tác xuất nhập hoặc hiển thị dữ liệu có thể được tách rời và phân cho các tuyến chạy độc lập thực thi.

7.4.1 Kiểm tra thư viện hỗ trợ tuyến:

Kiểm tra giá trị của hằng số `_POSIX_VERSION`.

- Nếu hằng số này không được định nghĩa hoặc có giá trị <199506L → hệ điều hành không hỗ trợ tuyén
- Nếu hằng số $\geq 199506L \rightarrow$ hệ điều hành có hỗ trợ tuyén.

Ví dụ 1: testthread.c

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
    printf("POSIX version is set to %ld\n", _POSIX_VERSION);
    if(_POSIX_VERSION<199506L)
        printf("This system does not support thread!\n");
    else
        printf("Thread is available in this system!\n");
}
```

Biên dịch và chạy chương trình:

```
$ gcc -o testthread testthread.c
$ ./testthread
```

```

embedded@ubuntu: ~/Documents/thread
embedded@ubuntu: ~/Documents/thread$ cd thread/
embedded@ubuntu: ~/Documents/thread$ gcc -o testthread testthread.c
embedded@ubuntu: ~/Documents/thread$ ./testthread
POSIX version is set to 200809
Thread is available in this system!
embedded@ubuntu: ~/Documents/thread$ 

```

Hình 7.2

7.4.2 Tạo lập tuyến và hủy tuyến:

Khi chương chính bắt đầu hoạt động nó chính là một tuyến. Nếu tiến trình không sử dụng thêm tuyến nào khác thì tiến trình và tuyến là một, tuyến điều khiển hàm ***main()*** là tuyến chính. Các tuyến khác do tiến trình tạo ra là tuyến phụ. Để tạo ra tuyến mới ngoài tuyến chính ta gọi hàm ***pthread_create()***:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread,pthread_attr_t *attr, void
*(start_routine)(void *), void *arg);
```

Tham số:

**thread*: lưu thông tin về tuyến khi tạo ra

**attr*: đặt thuộc tính cho tuyến. Nếu giá trị NULL yêu cầu tuyến tạo ra với thuộc tính mặc định

*(*start_routine*)(*void* *): địa chỉ của hàm thực thi tuyến

**arg*: địa chỉ đến vùng dữ liệu sẽ truyền cho hàm thực thi tuyến

Ví dụ 2: *create_thread.c*

```
#include <stdio.h>
#include <pthread.h> //khai bao cac ham xu ly tuyen
/*Ham thuc thi tuyen*/
void* do_loop(void* data)
{
    int i;
```

```
for(i=0;i<10;i++)
{
    sleep(1); //Dung 1 s
    printf("Thread '%d'-Got '%d'\n",data,i);
}

pthread_exit(NULL); //Cham dut tuyen

}

int main(void)
{
    pthread_t p_thread;
    int thr_id;
    int a=1;
    int b=2;
    /*Tao tuyen*/
    thr_id=pthread_create(&p_thread, NULL, do_loop,(void*)a);
    do_loop((void*)b);
    return 0;
}
```

Biên dịch thread và chạy chương trình:

```
$ gcc -o create_thread create_thread.c -lpthread
./create_thread
```

```

embedded@ubuntu:~/Documents/thread
embedded@ubuntu:~/Documents/thread$ gcc -o create_thread create_thread.c -lpthread
create_thread.c: In function 'do_loop':
create_thread.c:10:4: warning: format '%d' expects argument of type 'int', but argument 2 has type 'void *' [-Wformat]
create_thread.c: In function 'main':
create_thread.c:24:50: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
create_thread.c:24:71: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
embedded@ubuntu:~/Documents/thread$ ./create_thread
Thread '1'-Got '0'
Thread '2'-Got '0'
Thread '2'-Got '1'
Thread '1'-Got '1'
Thread '2'-Got '2'
Thread '2'-Got '3'
Thread '1'-Got '3'
Thread '1'-Got '4'
Thread '2'-Got '4'
Thread '2'-Got '5'
Thread '1'-Got '5'
Thread '1'-Got '6'
Thread '2'-Got '6'
Thread '2'-Got '7'
Thread '1'-Got '7'
Thread '2'-Got '8'
Thread '1'-Got '8'
Thread '1'-Got '9'
Thread '2'-Got '9'
embedded@ubuntu:~/Documents/thread$ 

```

Hình 7.3

*Giải thích:

Hàm ***do_loop()*** để in ra các số nguyên và nó được thực thi hai nơi: một là trong tuyến chính hàm ***int main(void)*** và một là tuyến phụ được tạo ra bởi hàm ***pthread_create()***. Như ta thấy kết quả hiển thị gần như song song và đang xen lẫn nhau giữa hai tuyến.

Hàm hủy tuyến:

pthread_exit(): được dùng để chấm dứt một tuyến hiện hành và chỉ dùng khi muốn thoát ra khỏi tuyến bất ngờ giữa quá trình thực thi. Ngoài ra còn có cấu trúc:

void pthread_exit(void *retval)

Tham số ****retval***: là giá trị trả về từ tuyến

7.4.3 Chờ tuyến kết thúc

Để đợi tuyến kết thúc và nhận kết quả trả về ta dùng hàm:

```
#include<pthread.h>
```

```
int pthread_join(pthread_t th,void *thread_return);
```

Tham số:

th: tuyến muốn chờ

****thread_return***: con trỏ đến vùng chứa giá trị trả về của tuyến

Ví dụ 3: *thread_wait.c*

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

char message[]="Hello World";

void *do_thread(void *data)
{
    printf("Thread function is executing....\n");
    printf("Thread data is %s ",(char*)message);
    sleep(3);
    strcpy(message, "Bye!");
    pthread_exit("Thank you for using my thread\n");
}

int main() //Chuong trinh chinh
{
    int res;
    pthread_t a_thread;
    void *thread_result;
    /*Tao va thuc thi tuyen*/
    res(pthread_create(&a_thread,NULL,do_thread,(void*)message));
    if(res!=0)
    {
        perror("Thread created error");
    }
}
```

```

    exit(EXIT_FAILURE);

}

/*Doi tuyen ket thuc*/

printf("Waiting for thread to finish...\n");

res=pthread_join(a_thread,&thread_result);

if(res!=0)

{

    perror("Thread wait error");

    exit(EXIT_FAILURE);

}

/*In ket qua tra ve cua tuyen*/

printf("Thread completed, it returned %s\n",(char*)thread_result);

printf("Message is now %s\n",message);

return 0;

}

```

Biên dịch và chạy chương trình:

```
$ gcc -o thread_wait thread_wait.c -lpthread
```

```
$ ./thread_wait
```

```

embedded@ubuntu:~/Documents/thread$ gcc -o thread_wait thread_wait.c -lpthread
thread_wait.c: In function 'do_thread':
thread_wait.c:12:3: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
embedded@ubuntu:~/Documents/thread$ ./thread_wait
Waiting for thread to finish...
Thread function is executing....
Thread data is Hello World Thread completed, it returned Thank you for using my thread
Message is now Bye!
embedded@ubuntu:~/Documents/thread$ 
```

Hình 7.4

***Giải thích:**

Hàm do_thread() được dùng để thực thi tuyến. Ta truyền dữ liệu vào hàm thông qua đối số thứ tư của hàm tạo tuyến pthread_create(). Bên trong do_thread() sau khi in ra thông báo cho biết tuyến đang thực thi và dữ liệu truyền vào tuyến ta tạm dừng tuyến trong 3 giây. Bên trong chương trình chính pthread_join() được gọi để chờ tuyến a_thread kết thúc. Khi hết 3 giây, tuyến thực thi hàm do_thread() thực hiện hai thao tác là thay đổi dữ liệu của biến toàn cục message. Trả về dữ liệu thông qua hàm pthread_exit(). Khi hàm pthread_join() trả về, ta in dữ liệu và kết quả của tuyến ra màn hình.

7.5 Dùng lập trình đa tuyến viết ứng dụng giao tiếp buttons và leds chạy trên kit:

Viết một ứng dụng thực hiện cho leds chạy sáng đuôi, sử dụng nút nhấn K1,K2 dùng để thay đổi (tăng/ giảm) tốc độ chạy.

Code: led_btn_thread.c

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/select.h>
#include <sys/time.h>
#include <errno.h>

#define ON 1
#define OFF 0
```

```

int led_fd, button_fd; // device file

/* Bien luu thoi gian delay */

static int t = 1000; //don vi la milisecond, ban dau mac dinh la 1000 ms

/* Ham sleep ms su dung usleep cua linux */

void sleepms(int ms)

{
    //usleep in us, sleep in second

    usleep(1000*ms); //convert to microseconds

    return;
};

void* btn_polling(void* param);

int main(int argc, char** argv)

{
    /*Bien cau truc chua tham so se truyen cho ham xu ly cua thread */
    const char *thread_parms = "button thread";

    pthread_t thread_id; // thread

    int led_no; //So hieu led 0-4

    //Mang chua gia tri trang thai 6 button se doc

    char buttons[6] = {'0', '0', '0', '0', '0', '0'};

    int i;

    //Mo thiet bi (led port), can kiem tra chinh xac ten trong /dev

    led_fd = open("/dev/leds", 0);

    if (led_fd < 0)

```

```

    {
        perror("open device leds");
        exit(1);
    }

    else printf("open device led ok\n");

    //Mo thiet bi (button port)
    button_fd = open("/dev/buttons", 0); //mo button port
    if(button_fd < 0)

    {
        perror("open device buttons");
        exit(1);
    }

    else printf("open device button ok\n");

    thread_id = pthread_create(&thread_id, NULL, &btn_polling, (void
*)thread_parms);

    //Ban dau tat ca cac led deu off
    for(i=0;i<4;i++) ioctl(led_fd, OFF, i);

    led_no=0;
    while(1)

    {
        //Bat led led_no
        ioctl(led_fd, ON, led_no);

        //Sleep in t ms
        sleepms(t);
    }
}

```

```

//Tat led led_no va chuan bi bat luon led ke tiep

ioctl(led_fd, OFF, led_no);

led_no++;

if(led_no == 4) led_no = 0;

}

close(button_fd);

close(led_fd);

return 0;
}

void* btn_polling(void* param)
{
/*const char thread_parms* = "button thread";*/
char cur_btn[2], old_btn[2] = {'0', '0'};

int i;

/*Lien tuc tham do trang thai nut bam (K1, K2 co duoc an)*/
for(;;)
{
    int num=read(button_fd, cur_btn, sizeof(cur_btn));
    if(num != sizeof(cur_btn))
    {
        perror("read buttons:");
        exit(1);
    }
    //Chi can doc K1, K2 tuong ung voi tang/giam led speed
    //Doc K1
}

```

```

if(old_btn[0] != cur_btn[0])
{
    sleepms(300); //delay de phim nay len
    old_btn[0] = cur_btn[0];
    t = t+50; //tang thoi gian delay
    printf("K1 is pressed/released t = %dms\n",t);
}

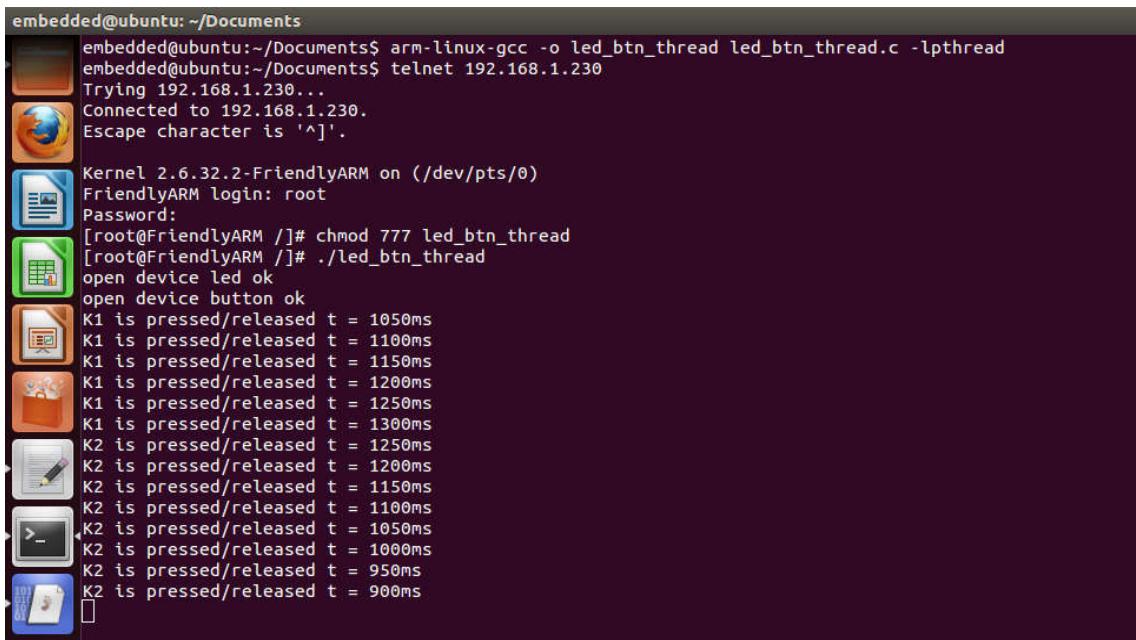
//Doc K2

if(old_btn[1] != cur_btn[1])
{
    sleepms(300); //delay de phim nay len
    old_btn[1] = cur_btn[1];
    t = t-50; //giam thoi gian delay
    if(t < 100) t = 100;
    printf("K2 is pressed/released t = %dms\n",t);
}
}
}

```

Biên dịch và chạy chương trình:

```
$ arm-linux-gcc led_btn_thread led_btn_thread.c -lpthread
```



```

embedded@ubuntu: ~/Documents
embedded@ubuntu:~/Documents$ arm-linux-gcc -o led_btn_thread led_btn_thread.c -lpthread
embedded@ubuntu:~/Documents$ telnet 192.168.1.230
Trying 192.168.1.230...
Connected to 192.168.1.230.
Escape character is '^]'.

Kernel 2.6.32.2-FriendlyARM on (/dev/pts/0)
FriendlyARM login: root
Password:
[root@FriendlyARM /]# chmod 777 led_btn_thread
[root@FriendlyARM /]# ./led_btn_thread
open device led ok
open device button ok
K1 is pressed/released t = 1050ms
K1 is pressed/released t = 1100ms
K1 is pressed/released t = 1150ms
K1 is pressed/released t = 1200ms
K1 is pressed/released t = 1250ms
K1 is pressed/released t = 1300ms
K2 is pressed/released t = 1250ms
K2 is pressed/released t = 1200ms
K2 is pressed/released t = 1150ms
K2 is pressed/released t = 1100ms
K2 is pressed/released t = 1050ms
K2 is pressed/released t = 1000ms
K2 is pressed/released t = 950ms
K2 is pressed/released t = 900ms

```

Hình 7.5***Giải thích:**

Chương trình gồm 1 tuyến chính thực hiện hiệu ứng led đuôi trong một vòng lặp vô hạn với thời gian bật tắt led mặc định là 1 giây. Thời gian trễ có thể điều chỉnh (tăng/giảm) khi nhấn nút K1, K2. Button driver thực hiện giao tiếp kiểu thăm dò (polling) cần sử dụng 1 tuyến riêng dùng để thay đổi giá trị t khi nhấn nút K1,K2 song song với công việc chính là điều khiển hiệu ứng led chạy đuôi.

7.6 Bài tập thực hành:

Bài 1: Viết ứng dụng kết hợp giữa nút nhấn và PWM sử dụng đa tuyến. Ở chế mặc định hay reset PWM xuất ra xung có tần số 1000Hz

- Khi nhấn/ nhả K1 tần số tăng thêm 50Hz cho đến khi đạt tần số 3000Hz thì không tăng nữa.
- Khi nhấn/nhả K2 tần số giảm đi 50Hz biết giảm đến 50Hz thì không giảm nữa.

Bài 2: Viết ứng dụng kết hợp giữa nút nhấn và PWM sử dụng đa tuyến. PWM xuất ra tần số bắt đầu từ 500Hz kéo dài với thời gian 1 giây thì tần số sẽ tăng thêm 100Hz cho đến khi đạt 3000Hz kết thúc 1 chu kỳ và quá trình trên lặp lại liên tục.

- Khi nhấn/ nhả K1 thời gian sẽ tăng thêm 100ms
- Khi nhấn/nhả K2 thời gian sẽ giảm đi 100ms và khi đạt 100ms thì không giảm nữa.

Bài 3: Viết ứng dụng kết hợp adc và led sử dụng đa tuyến. Chương trình thực hiện việc 4 led hiển thị sáng dần từ trên xuống dưới theo chu kỳ với thời gian chuyển trạng thái phụ thuộc vào giá trị đọc được từ chuyển đổi adc.

Giá trị đọc được từ chuyển đổi adc	Thời gian chuyển trạng thái(ms)
0 → 100	100
101 → 200	200
201 → 300	300
301 → 400	400
401 → 500	500
501 → 600	600
601 → 700	700
701 → 800	800
801 → 900	900
901 → 1000	1000
1001 → 1023	1100

CHƯƠNG 8: LẬP TRÌNH DRIVER TRONG LINUX

8.1 Mục đích:

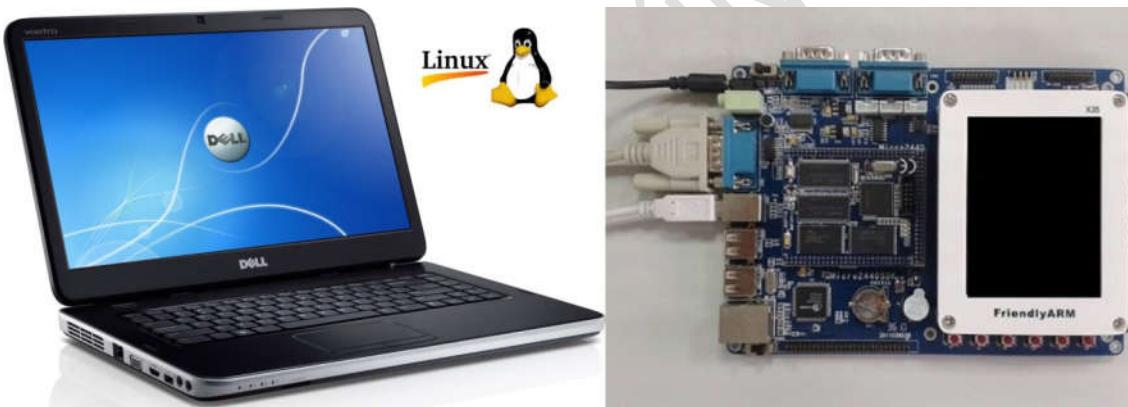
- Hiểu được sơ đồ cấu trúc từ tầng ứng dụng đến phần cứng.
- Hiểu được các bước về lập trình driver
- Lập trình được các driver đơn giản đã học led, button,pwm và sử dụng các driver đã tạo viết ứng dụng.

8.2 Yêu cầu:

- Có hiểu biết về phần cứng các device.
- Có kiến thức về lập trình C
- Có kiến thức về linux

8.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc.
- Bộ kit thực hành Micro2440 , dây cáp COM, USB, LAN
- Máy host đã biên dịch nhân hệ điều hành dùng để biên dịch driver nằm trong thư mục /lib/modules(xem phần 1.8)



8.4 Lý thuyết về Linux device driver:

8.4.1 Giới thiệu về Linux driver:

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

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

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

- Phân loại theo chức năng: Hard disk controller, Graphic controller, Keyboard controller.
- Phân loại theo các chuẩn giao tiếp với các thiết bị: PCI controller, USB controller, I2C controller.

Driver hướng dẫn CPU làm việc với bộ điều khiển được gọi là bus driver. Còn driver hướng dẫn CPU làm việc với thiết bị thì được gọi là device driver.

Trong phần này chúng ta khảo sát device driver để lập trình.

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

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

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

8.4.2 Linux kernel module:

Linux được thiết kế để làm việc với hàng tỉ thiết bị. Nhưng ta không thể đưa tất cả các driver vào trong kernel được, vì sẽ làm cho kích thước kernel rất lớn. Giải pháp cho vấn đề này đó là: thiết kế các driver dưới dạng module tách rời với kernel. Trong quá trình hoạt động, driver nào cần thiết sẽ được lắp vào kernel, còn driver nào không cần thiết sẽ bị tháo ra khỏi kernel (dynamic loading).

Linux kernel module là một file với tên mở rộng là (.ko). Nó sẽ được lắp vào hoặc tháo ra khỏi kernel khi cần thiết. Chính vì vậy, nó còn có một tên gọi khác là loadable kernel module. Một trong những kiểu loadable kernel module phổ biến đó là driver. Việc thiết kế driver theo kiểu loadable module mang lại 3 lợi ích:

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

Phần lớn các driver đều là các loadable kernel module, nhưng không phải là tất cả và vẫn có một số driver được tích hợp luôn vào trong kernel, đặc biệt là các bus driver. Chúng được gọi là built-in driver. Các device driver thường sẽ là các loadable kernel module. Trong chương này các thuật ngữ module, device driver, loadable kernel module, Linux kernel module, loadable module, kernel module đều được hiểu là một. Khi cần sử dụng module nào mà kernel space chưa có thì ta phải “gắn” module vào kernel space. Quá trình này có thể diễn ra một cách tự động, với trình tự sau:

- Bước 1: Kernel kích hoạt tiến trình **modprobe** cùng với tham số truyền vào là tên của module (ví dụ xxx.ko).
- Bước 2: Tiến trình modprobe kiểm tra file */lib/modules/<kernel-version>/modules.dep* xem xxx.ko có phụ thuộc vào module nào khác không. Giả sử xxx.ko phụ thuộc vào module yyy.ko.
- Bước 3: Tiến trình modprobe sẽ kích hoạt tiến trình **insmod** để đưa các module phụ thuộc vào trước (yyy.ko), rồi mới tới module cần thiết (xxx.ko).

Như vậy, các module được đưa vào kernel space dưới sự giúp đỡ của tiến trình modprobe.

Ví dụ 1: Viết driver đơn giản đầu tiên.

Driver được viết bằng C, nhưng không có hàm main(). Hơn nữa, bởi vì driver được nạp và liên kết với hệ điều hành, nên nó cần được biên dịch giống cách biên dịch nhân hệ điều hành, và các header files được sử dụng trong mã nguồn driver chỉ là những cái mà nhân hệ điều hành cung cấp, không bao giờ có các hàm của thư viện lập trình C (mà thường để trong thư mục /usr/include).

Một điểm thú vị khác là mã nguồn nhân được lập trình theo kiểu hướng đối tượng trong C, mã nguồn driver cũng tương tự như vậy. Bất kỳ một driver nào trên Linux đều có một hàm tạo (constructor) và một hàm hủy (destructor). Hàm tạo của driver được gọi khi driver được nạp vào nhân hệ thống, và hàm hủy được gọi khi gỡ driver khỏi hệ thống (dùng lệnh rmmod). Hai hàm này cũng giống như các hàm thông thường, ngoại trừ việc chúng cần có các chỉ thị __init và __exit tương ứng và phải được đăng ký sử dụng các macros module_init() và module_exit(), (được định nghĩa trong file tiêu đề module.h).

Bước 1: Tạo một file first.c đặt trong 1 thư mục nào đó.

Bước 2: Vào đoạn code sau đặt trong file first.c

```
/* first.c - first device driver code */

#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>

static int __init first_init(void) /* Constructor */
{
    printk("Welcome the first device driver\n");
    return 0;
}

static void __exit first_exit(void) /* Destructor */
{
    printk("Goodbye the first device driver");
}

module_init(first_init);
```

```

module_exit(first_exit);

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Phan Tuan Anh");

MODULE_DESCRIPTION("The simplest character device driver");

```

Trong ví dụ trên, ta chỉ cần tham chiếu tới file của Linux kernel là <linux/module.h>. File này chứa 2 macro quan trọng, là: **module_init()** và **module_exit()**. Do đó, dù viết bất cứ kernel module nào, ta cũng cần tham chiếu tới <linux/module.h>.

- **module_init** giúp xác định hàm nào sẽ được thực thi ngay sau khi lắp module vào kernel.
- **module_exit** giúp xác định hàm nào được thực thi ngay trước khi tháo module ra khỏi kernel.

Trong ví dụ trên, **init_hello()** là hàm được gọi ngay sau khi module hello được lắp vào, và **exit_hello()** là hàm được gọi ngay trước khi module hello bị tháo ra khỏi kernel.

Macro **_init** thường đi kèm với hàm khởi tạo. Trong ví dụ trên, macro **_init** xuất hiện trước tên hàm **first_init**. Macro này giúp kernel biết rằng, hàm **first_init()** chỉ phải thực thi lúc khởi tạo, nên vùng nhớ chứa hàm này có thể được giải phóng sau khi nó thực thi xong mà không ảnh hưởng gì.

Tương tự, macro **_exit** thường đi kèm với hàm kết thúc. Trong ví dụ trên, **_exit** xuất hiện trước tên hàm **first_exit**. Macro này cho kernel biết, khi lắp module vào kernel thì chưa cần đưa hàm **first_exit** vào trong bộ nhớ RAM. Chỉ khi chuẩn bị tháo module ra khỏi kernel, hàm **first_exit** mới cần được đưa vào RAM và thực thi.

Hàm **printf** để ghi lại quá trình hoạt động của module. Việc này được gọi là logging. Mục đích của việc logging là để phục vụ quá trình gỡ lỗi sau này (debug). Ta có thể sử dụng lệnh **dmesg** để xem quá trình hoạt động của kernel kể từ lúc nó khởi động.

Các macro nằm ở cuối ví dụ trên cung cấp các thông tin về module. Ta có thể sử dụng lệnh **modinfo** để xem các thông tin của một module:

- Macro **MODULE_AUTHOR** cho biết ai là người tạo ra module.
- Macro **MODULE_DESCRIPTION()** cho biết module làm được những gì.
- **MODULE_LICENSE** cho biết người dùng có cần phải trả phí nếu sử dụng module hay không. Trong ví dụ trên, giấy phép sử dụng module thuộc loại GPL. Với giấy phép sử dụng GPL, người dùng có thể sử dụng module miễn phí.

Bước 3: Tạo Makefile nằm chung thư mục với file first.c để biên dịch driver. Code của Make file:

```
obj-m += first.o
```

```
all:
```

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

clean:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Bước 4: Biên dịch driver, nạp vào và tháo ra khỏi kernel:

Di chuyển vào thư mục chứa first.c. Tốt nhất ta nên thực hiện với quyền root để dễ dàng thực hiện.

Vào lệnh :

make

Nếu biên dịch thành công sẽ tạo ra file *.ko nằm chung thư mục với file first.c

Ta nạp vào kernel dùng lệnh sau:

insmod first.ko

Xem các module đã được nạp dùng lệnh:

lsmod | head -10

Tháo module ra khỏi kernel dùng lệnh:

rmmmod first.ko

Để xem quá trình hoạt động của module dùng lệnh:

dmesg

```
root@ubuntu:/home/embedded/Documents/driver/bai1
embedded@ubuntu:~/Documents/driver/bai1$ su
Password:
root@ubuntu:/home/embedded/Documents/driver/bai1# make
make -C /lib/modules/3.2.0-23-generic/build M=/home/embedded/Documents/driver/bai1 modules
make[1]: Entering directory `/usr/src/linux-headers-3.2.0-23-generic'
  Building modules, stage 2.
    MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-3.2.0-23-generic'
root@ubuntu:/home/embedded/Documents/driver/bai1# insmod first.ko
root@ubuntu:/home/embedded/Documents/driver/bai1# lsmod | head -10
Module           Size  Used by
first            12425  0
second           12456  0
vesafb            13844  1
vboxsf            42737  1
snd_intel8x0      38570  2
snd_ac97_codec    134826  1 snd_intel8x0
ac97_bus          12730  1 snd_ac97_codec
snd_pcm            97188  2 snd_intel8x0,snd_ac97_codec
joydev             17693  0
root@ubuntu:/home/embedded/Documents/driver/bai1# rmmmod first.ko
root@ubuntu:/home/embedded/Documents/driver/bai1#
```

8.4.3 Giới thiệu character driver:

8.4.3.1 Device file:

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

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

Từ hình 1, chúng ta nhận thấy rằng:

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

Nguyên tắc hoạt động của device file:

Vì là một loại file, nên hoạt động tương tác với device file cũng tương tự như với các file thông thường. Các tương tác này bao gồm mở file (open), đóng file (close), ghi dữ liệu vào file (write), đọc dữ liệu từ file (read). Tuy nhiên, kết quả của các tương tác này thường không giống với kết quả tương tác với file thông thường. Đối với file thông thường, nếu bạn ghi dữ liệu vào là A, ngay sau đó bạn đọc dữ liệu từ file ra, thì vẫn là A. Nhưng điều này có thể không đúng với device file. Xét audio device file. Khi bạn ghi dữ liệu vào file này thì dữ liệu được đưa tới loa, còn khi bạn đọc dữ liệu từ file này, thì dữ liệu đó đến từ microphone.

Việc kết nối từ ứng dụng đến thiết bị được thực hiện hoàn chỉnh thông qua 4 thực thể chính liên quan gồm:

1. Application (ứng dụng)
2. Character device file (File thiết bị)
3. Character device driver (Driver thiết bị)
4. Character device (Thiết bị)

Một điểm thú vị là các thực thể trên có thể tồn tại một cách độc lập trên 1 hệ thống mà không cần sự tham gia của các thực thể khác. Việc tồn tại của mỗi thực thể không có nghĩa là chúng đã được liên kết đến các thực thể khác để tạo ra kết nối hoàn chỉnh. Các kết nối giữa chúng cần được thực hiện một cách tường minh. Một ứng dụng kết nối đến file thiết bị bằng cách gọi một hàm mở file thiết bị đó. Các file thiết bị liên kết đến driver của nó bằng thao tác đăng ký được thực hiện trong driver. Một driver được kết nối đến một thiết bị thông qua các thao tác mức thấp với thiết bị phần cứng đặc trưng.

Như vậy, chúng ta đã tạo ra một kết nối hoàn chỉnh từ ứng dụng đến thiết bị phần cứng. Lưu ý rằng, file thiết bị không phải là thiết bị thực sự, nó chỉ là một thực thể nằm giữ thiết bị thực sự.

8.4.3.2 Device number:

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

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

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

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

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

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

8.4.3.3 Cấu trúc của character driver:

Cấu trúc của character driver gồm 2 phần:

- Phần OS specific gồm các nhóm hàm sau:
 - ❖ Hàm khởi tạo. Hàm này chịu trách nhiệm:
 - Yêu cầu kernel cấp phát device number.
 - Yêu cầu kernel tạo device file.
 - Yêu cầu kernel cấp phát bộ nhớ cho các cấu trúc dữ liệu của driver và khởi tạo chúng.
 - Yêu cầu khởi tạo thiết bị vật lý.
 - Đăng ký các hàm entry point với kernel.
 - Đăng ký hàm xử lý ngắn.
 - ❖ Hàm kết thúc. Hàm này làm ngược lại những gì hàm khởi tạo đã làm.
 - ❖ Các hàm entry point. Ví dụ open(), release(), read(), write(), ioctl(), ...
- Phần device specific gồm các nhóm hàm sau:
 - ❖ Nhóm các hàm khởi tạo/giải phóng thiết bị.
 - ❖ Nhóm các hàm đọc/ghi vào các thanh ghi của thiết bị.
 - Đọc/ghi các thanh ghi dữ liệu.

- Lấy thông tin từ các thanh ghi trạng thái.
- Thiết lập lệnh cho các thanh ghi điều khiển.
 - ❖ Nhóm các hàm xử lý ngắn.

8.4.4 Cấp phát tĩnh device number:

Có 2 phương pháp cấp phát device number: cấp phát động và cấp phát tĩnh. Trước khi tìm hiểu phương pháp cấp phát tĩnh, ta cần tìm hiểu xem Linux kernel biểu diễn device number như thế nào?

Biểu diễn device number.

Linux kernel sử dụng cấu trúc **dev_t** để biểu diễn device number. Cấu trúc này có kích thước 32 bit, trong đó major number chiếm 12 bits, minor number chiếm 20 bits. Linux kernel cũng có các hàm hoặc macro để hỗ trợ chúng ta làm việc với biến kiểu **dev_t**. (định nghĩa trong linux/types.h) chứa cả số liệu major và minor.

Sử dụng các Macros (định nghĩa trong linux/kdev_t.h):

- **MAJOR(dev_t dev)** lấy số hiệu major từ tham số dev
- **MINOR(dev_t dev)** lấy số hiệu minor từ tham số dev
- **MKDEV(int major, int minor)** tạo ra dữ liệu dev từ cặp số hiệu major và minor.

Việc kết nối giữa file thiết bị và driver được thực hiện thông qua 2 bước sau:

1. Đăng ký số hiệu <major, minor> cho file thiết bị
2. Kết nối các thao tác file thiết bị với các hàm tương ứng trong driver.

Để đăng ký số hiệu <major, minor> dùng phương pháp cấp phát tĩnh:

- Dùng hàm **MKDEV** để tạo số major number.
- Gọi hàm **register_chrdev_region** để đăng ký device number với kernel. Nếu giá trị này vẫn chưa được sử dụng bởi driver nào, thì kernel sẽ cấp giá trị đó cho driver và quá trình đăng ký sẽ thành công. Ngược lại, nếu đã có driver sử dụng giá trị đó rồi, thì kernel sẽ từ chối cấp phát và quá trình đăng ký thất bại.
- Gọi hàm giải phóng **unregister_chrdev_region** device number khi không còn dùng đến device number nữa thường được đặt trong hàm kết thúc của char driver.

Cú pháp của từng hàm:

```
int register_chrdev_region(dev_t first, unsigned int cnt, char *name);
```

❖ Chức năng:

Đăng ký một dải gồm **cnt** device number bắt đầu từ **first** cho character device có tên là **name**.

❖ Tham số:

- **first**: device number đầu tiên muốn <major, first_minor>.
- **cnt**: số lượng device number mà ta muốn đăng ký. Driver sẽ đăng ký với kernel các device number từ <major, first_minor> đến <major, first_minor+cnt-1>.
- ***name**: tên của character device. Tên này sẽ xuất hiện trong thư mục /proc/devices và khác với tên device file trong thư mục /dev.

❖ Giá trị trả về:

- Nếu đăng ký thành công sẽ trả về 0
- Nếu đăng ký thất bại sẽ trả về giá trị <0.

void int register_chrdev_region(dev_t first, unsigned int cnt)

Ví dụ 2: Cấp phát tịnh số device number:

Code:

```
/* second.c – second device driver code */

#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/fs.h>

struct char_driver{
    dev_t major_num;
}c_drv;

static int __init second_init(void) /* Constructor */
{
    int ret;
    c_drv.major_num=MKDEV(123,0);
    ret=register_chrdev_region(c_drv.major_num,1,"CharSample");
}
```

```

if(ret<0){

    printk("Failed to register device number\n");

    return -1;

}

	printk("Initialize char driver successfully\n");

return 0;

}

static void __exit second_exit(void) /* Destructor */

{

unregister_chrdev_region(c_drv.major_num, 1);

}

module_init(second_init);

module_exit(second_exit);

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Phan Tuan Anh");/* tac gia cua module */

MODULE_DESCRIPTION("Allocate device number statically");/* mo ta chuc nang cua module */

```

Dùng Makefile giống ví dụ 1 ta biên dịch ra module tạo file **second.ko**.

Nạp module vào kernel dùng lệnh **insmod**

Dùng lệnh **cat /proc/devices** sẽ thấy xuất hiện dòng lệnh **123 CharSample** vì mới xin Linux kernel cấp phát đúng số device number được yêu cầu.

```

embedded@ubuntu:~/Documents/driver/bai2$ su
Password:
root@ubuntu:/home/embedded/Documents/driver/bai2# make
make -C /lib/modules/3.2.0-23-generic/build M=/home/embedded/Documents/driver/bai2 modules
make[1]: Entering directory `/usr/src/linux-headers-3.2.0-23-generic'
      Building modules, stage 2.
      MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-3.2.0-23-generic'
root@ubuntu:/home/embedded/Documents/driver/bai2# insmod second.ko
root@ubuntu:/home/embedded/Documents/driver/bai2# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttys
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 29 fb
 99 ppdev
108 ppp
116 alsa
123 CharSample

```

8.4.5 Cấp phát động device number:

Nếu lựa chọn phương pháp cấp phát tĩnh device number, thì device number đó có thể đã được sử dụng trên driver khác và dẫn đến không hoạt động được. Như vậy mỗi khi sử dụng bộ số device number buộc lập trình viên phải chọn bộ number chưa sử dụng để dùng. Để giải quyết vấn đề không phải tìm bộ device number chưa sử dụng, lập trình viên nên sử dụng phương pháp cấp phát động device number. Linux kernel cung cấp một hàm là **alloc_chrdev_region** nhiệm vụ hàm này là tìm ra một giá trị dùng làm device number.

Cú pháp:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int cnt, char *name)
```

❖ Chức năng:

Yêu cầu kernel cấp phát một dải gồm cnt device number cho char device có tên là *name.

❖ Tham số:

- *dev : con trỏ này chứa giá trị trả về của hàm. Device number đầu tiên của dải sẽ được trả về thông qua biến này.
- firstminor: giá trị minor của số device number đầu tiên trong dải.
- cnt : là số lượng device number mà hàm này yêu cầu cấp phát.
- *name : tên của character device. Tên này sẽ xuất hiện trong thư mục /proc/devices.

❖ Giá trị trả về:

- Nếu tìm được một device number, hàm này sẽ trả về 0. Device number đầu tiên trong dải sẽ được trả qua tham số *dev.
- Nếu không thể tìm được một device number nào, hàm sẽ trả về số nguyên âm.

Ví dụ 3: Cấp phát động số device number:

Code:

```
/* third.c – third device driver code */

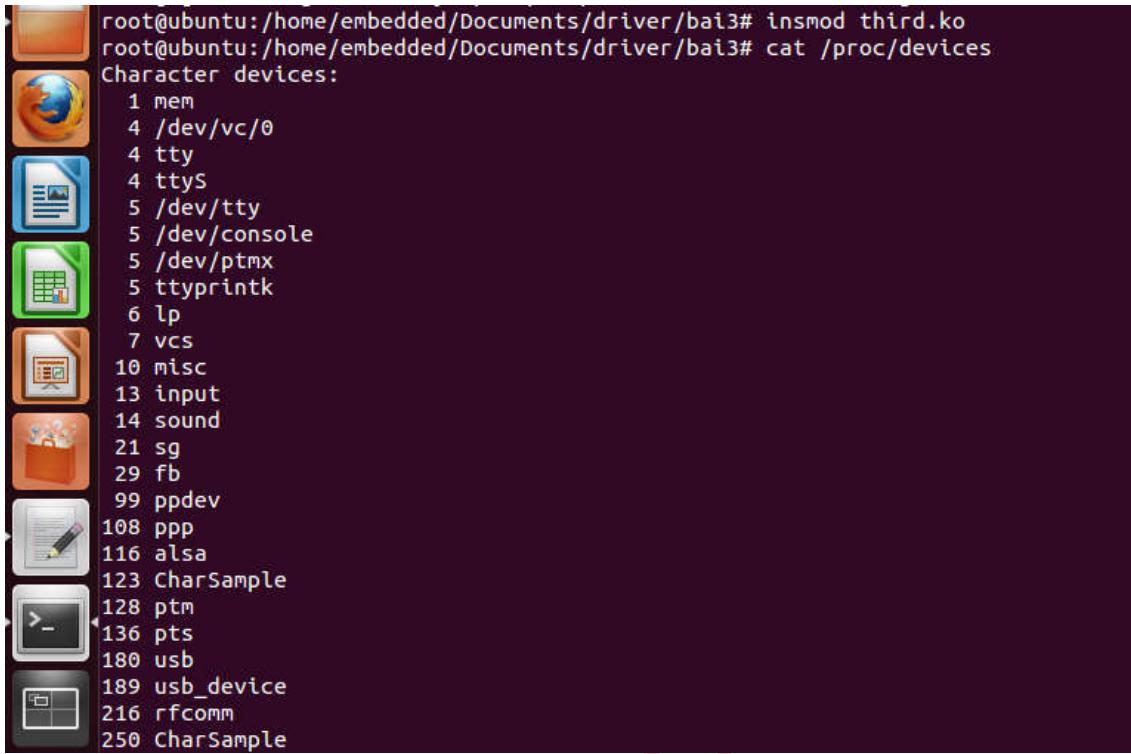
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/fs.h>

struct char_driver{
    dev_t major_num;
}c_drv;

static int __init third_init(void) /* Constructor */
{
    int ret;
    if((ret = alloc_chrdev_region(&c_drv.major_num, 0, 1, "CharSample")) < 0)
    {
        printk("Failed to register device number dynamically \n");
        return -1;
    }
    printk("Allocated           device           number
(%d,%d)\n",MAJOR(c_drv.major_num),MINOR(c_drv.major_num));
}
```

```
return 0;  
}  
  
static void __exit third_exit(void) /* Destructor */  
{  
    unregister_chrdev_region(c_drv.major_num, 1);  
}  
  
module_init(third_init);  
module_exit(third_exit);  
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Phan Tuan Anh");/* tac gia cua module */  
MODULE_DESCRIPTION("Allocate device number dynamically");/* mo ta chuc nang  
cua module */
```

Biên dịch driver và nạp vào kernel module như ví dụ 2, sau đó dùng lệnh **cat /proc/devices** ta được:



```

root@ubuntu:/home/embedded/Documents/driver/bai3# insmod third.ko
root@ubuntu:/home/embedded/Documents/driver/bai3# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttys
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttyprintk
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 29 fb
 99 ppdev
108 ppp
116 alsa
123 CharSample
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
250 CharSample

```

Lần này ta không tự chọn số device number mà linux kerner chọn **250 CharSample**.

8.4.6 Tạo device file:

Ở phần trước chúng ta yêu cầu Linux cấp phát device number, chứ chưa tạo ra device file. Trong phần này sẽ giúp chúng ta tìm hiểu các cách thức tạo ra một device file bên trong thư mục `/dev`. Có hai cách thức đó là:

- Tạo device file một cách thủ công.
- Tạo device file một cách tự động.

8.4.6.1 Tạo device file một cách thủ công:

Dùng hàm `mknod`. Cú pháp:

`int mknod(const char *pathname, mode_t mode, dev_t dev)`

❖ Chức năng:

Tạo device file dùng thủ công.

❖ Tham số:

- `*pathname`: tên đường dẫn xuất hiện trong thư mục `/dev`
- `mode`: thể hiện device file là character (c) hay block (b)
- `dev`: là số device number (major,minor)

❖ Giá trị trả về:

- Hàm trả về 0 nếu tạo thành công.
- Hàm trả về -1 nếu tạo không thành công.

Theo phương pháp thủ công này thì device file có thể tạo ra trước khi lắp module vào kernel rất hữu ích khi ta tạo device number cấp phát tĩnh.

8.4.6.2 Tạo device file tự động:

Dựa vào tiến trình **udevd** để tạo/hủy các device file trong thư mục `/dev`. Khi viết char driver, lập trình viên sẽ sử dụng một số hàm của Linux kernel để gửi sự kiện lên cho **udevd**. Các sự kiện áy được gọi là **uevent** (user event). Sau khi nhận được **uevent**, **udevd** sẽ tạo ra một device file trong thư mục `/dev`.

Để triển khai phương pháp này, ta thực hiện 3 bước sau:

- Tham chiếu tới thư viện `<linux/device.h>`.
- Tạo một lớp các thiết bị.
- Tạo thiết bị trong lớp.

Tạo lớp thiết bị ta dùng hàm **create_class** , hàm hủy tương ứng là **class_destroy**.

Cú pháp:

struct class* class_create(struct module *owner, const char *name)

❖ Chức năng:

Tạo ra một lớp các thiết bị có tên là [name] trong thư mục `/sys/class`. Lớp này chứa liên kết tới thông tin của các thiết bị cùng loại.

❖ Tham số:

- owner: con trỏ trả về module sở hữu lớp thiết bị này
- *name: tên của lớp các thiết bị

❖ Giá trị trả về:

- Nếu thành công, thư mục có tên [name] được tạo ra trong `/sys/class`. Hàm trả về một con trỏ trả về biến cấu trúc class.
- Nếu thất bại, trả về NULL

void class_destroy(struct class *)

Để tạo thiết bị trong lớp trên ta dùng hàm **device_create** của linux và hàm hủy tương ứng là **device_destroy**.

Cú pháp:

struct device* device_create(struct class* cls, struct device *parent, dev_t devt, void *drvdata, const char *name)

❖ Chức năng:

Tạo ra các thông tin của một thiết bị cụ thể. Khi có thông tin này, udev sẽ tạo ra một device file * tương ứng trong `/dev`.

❖ Tham số:

- *cls: con trỏ trả về lớp các thiết bị. Con trỏ này là kết quả của việc gọi hàm **class_create**
- *parent: con trỏ trả về thiết bị cha của thiết bị này. Nếu thiết bị không có cha, ta truyền vào là NULL
- devt: device number của thiết bị.
- *drvdata: dữ liệu bổ sung. Nếu không có, ta truyền vào là NULL.
- *name: tên của thiết bị, udev sẽ tạo ra device file với tên này trong thư mục `/dev`

void device_destroy(struct class * cls, dev_t devt)

Ví dụ 4: Tạo device file tự động

Code:

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>

struct char_driver{
    dev_t major_num;
    struct class *class_num;
    struct device *dev_num;
}c_drv;

static int __init fourth_init(void) /* Constructor */
{
    int ret;
    if ((ret = alloc_chrdev_region(&c_drv.major_num, 0, 1, "CharSample")) < 0)
    {
        printk("Failed to register device number dynamically \n");
        return -1;
    }
}
```

```

    printk("Allocated device number (%d,%d)\n ", MAJOR(c_drv.major_num),
MINOR(c_drv.major_num));

if (IS_ERR(c_drv.class_num = class_create(THIS_MODULE, "CharClass")))
{
    printk("Failed to create device class\n");
    unregister_chrdev_region(c_drv.major_num, 1);
    return -1;
}

if (IS_ERR(c_drv.dev_num = device_create(c_drv.class_num, NULL,
c_drv.major_num, NULL, "CharDevice")))
{
    printk("Failed to create a device \n");
    class_destroy(c_drv.class_num);
    unregister_chrdev_region(c_drv.major_num, 1);
    return -1;
}

printk("Initialize char driver successfully\n");
return 0;
}

static void __exit fourth_exit(void) /* Destructor */
{
    device_destroy(c_drv.class_num, c_drv.major_num);
}

```

```

class_destroy(c_drv.class_num);

unregister_chrdev_region(c_drv.major_num, 1);

}

```

```

module_init(fourth_init);

module_exit(fourth_exit);

```

MODULE_LICENSE("GPL");

MODULE_AUTHOR("Phan Tuan Anh");

MODULE_DESCRIPTION("Create a device file");

Dùng Makefile giống ví dụ 1 ta biên dịch ra module tạo file **fourth.ko**.

Nạp module vào linux kernel.

Dùng lệnh cat /proc/devices để xem device number “**CharSample**”

Dùng lệnh ls /sys/class để xem tạo lớp “**CharClass**” được tạo.

Dùng lệnh ls /dev để xem device file “**CharDevice**” được tạo.

```

root@ubuntu:/home/embedded/Documents/driver/ba14
root@ubuntu:/home/embedded/Documents/driver/ba14# make
make[1]: Entering directory '/usr/src/linux-headers-3.2.0-23-generic'
Building modules, stage 2.
MODPOST 1 modules
make[1]: Leaving directory '/usr/src/linux-headers-3.2.0-23-generic'
root@ubuntu:/home/embedded/Documents/driver/ba14# insmod fourth.ko
root@ubuntu:/home/embedded/Documents/driver/ba14# cat /proc/devices | head -30
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttys
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  5 ttysize
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
 29 fb
 99 ppdev
108 ppp
116alsa
128 ptm
136 pts
180 usb
189 usb_device
216 rfcomm
250 charSample
251 hidraw
252 usbmon

```

```

root@ubuntu:/home/embedded/Documents/driver/ba1# ls /sys/class
ata_device bdi charClass firmware nmon mfdio_bus net ppp rtc sscsi_host spi_transpor
ata_link block devfreq gpio i2c-adapter mem pci_bus printer sscsi_device sound thermal
ata_port bluetooth dma graphics input misc power_supply regulator sscsi_disk sscsi_generic sscsi_master tty
backlight bsg dmi hidraw leds mmc_host ppdev rfkill sscsi_genic sscsi_master usbmon
root@ubuntu:/home/embedded/Documents/driver/ba1# ls /dev
autofs fbt loop0 pts ram8 sr0 tty18 tty31 tty45 tty59 ttyS13 ttyS27 usbmon1
block fd loop1 ram0 ram9 stdio tty19 tty32 tty46 tty6 ttyS14 ttyS28 vboxguest
bsg full loop-control ram1 random stdln tty2 tty33 tty47 tty60 ttyS15 ttyS29 vboxuser
btrfs-control fuse mapper ram10 rfkill stdout tty20 tty34 tty48 tty61 ttyS16 ttyS30 vcs1
bus hpet mem ram11 rtc tty tty21 tty35 tty49 tty62 ttyS17 ttyS31 vcs2
cdrom hidraw0 nclog ram12 rtc0 tty0 tty22 tty36 tty5 tty63 ttyS18 ttyS32 vcs3
char input net ram13 sda tty1 tty23 tty37 tty50 tty7 ttyS19 ttyS4 vcs4
CharDevice kmsg network_latency ram14 sda1 tty10 tty24 tty38 tty51 tty8 ttyS2 ttyS5 vcs5
console log network_throughput ram15 sda2 tty11 tty25 tty39 tty52 tty9 ttyS20 ttyS6 vcs6
core loop0 null ram2 sda5 tty12 tty26 tty4 tty53 ttyprintk ttyS21 ttyS7 vcs6
cpu loop1 oldmem ram3 sg0 tty13 tty27 tty40 tty54 tty50 ttyS22 ttyS8 vcsa
cpu_dma_latency loop2 port ram4 sg1 tty14 tty28 tty41 tty55 tty51 ttyS23 ttyS9 vcsa1
disk loop3 ppp ram5 shm tty15 tty29 tty42 tty56 tty510 ttyS24 ulnput vcsa2
dvd loop4 psaux ram6 snapshot tty16 tty3 tty43 tty57 tty511 ttyS25 urandom vcsa3
loop5 ptmx ram7 snd tty17 tty30 tty44 tty58 tty512 ttyS26 usbmon0 vcsa4
root@ubuntu:/home/embedded/Documents/driver/ba1# 

```

8.4.7 Đăng ký các entry point:

Linux kernel có một cấu trúc dùng để mô tả các character device. Cấu trúc này là **cdev**. Khi char driver muốn điều khiển một character device, thì char driver phải “nộp” cấu trúc **cdev** cho Linux kernel. Cấu trúc **cdev** có một trường thuộc kiểu cấu trúc **file_operations**. Cấu trúc **file_operations** dùng để mô tả hàm nào làm gì, tương ứng với system call nào. Mỗi hàm đó được gọi là một entry point.

8.4.7.1 Cấu trúc của cdev:

Linux kernel sử dụng cấu trúc **cdev** để mô tả một character device. Nó gồm 1 số trường quan trọng sau:

```

struct cdev {
    ...
    struct file_operations *ops; /* các entry points */
    dev_t dev; /* device number của character device */
    unsigned int count; /* số lượng các thiết bị có cùng major number */ ...
}

```

Các hàm **cdev_alloc**, **cdev_init** và **cdev_add** được dùng để khởi tạo và đăng ký các entry point và linux kernel, **cdev_del** dùng để hủy bỏ vùng nhớ cấu trúc cdev thường được gọi trong hàm kết thúc của char driver.

Cú pháp:

struct cdev *cdev_alloc(void)

- ❖ Chức năng:
Cấp phát bộ nhớ cho cấu trúc cdev.
- ❖ Tham số:
không có
- ❖ Giá trị trả về:
 - Nếu thành công, hàm này trả về địa chỉ của vùng nhớ chứa cấu trúc cdev
 - Nếu thất bại, hàm này trả về NULL.

Cú pháp:

void cdev_init(struct cdev *p, struct file_operations *fops)

- ❖ Chức năng:
Khởi tạo các trường của cấu trúc cdev.
- ❖ Tham số:
 - *p: là địa chỉ của vùng nhớ chứa cấu trúc cdev.
 - *fops: là địa chỉ của vùng nhớ chứa cấu trúc file_operations.

Hàm này không có giá trị trả về.

Cú pháp:

int cdev_add(struct cdev *p, dev_t first, unsigned count)

❖ Chức năng:

Đăng ký cấu trúc cdev với Linux kernel và liên kết cấu trúc này với device file.

❖ Tham số:

- *p : là địa chỉ của vùng nhớ chứa cấu trúc cdev.
- first: là device number của device file đầu tiên trong chuỗi các device file liên kết với cấu trúc cdev. Các device number này có cùng major number.
- count: là số lượng các device file sẽ liên kết với cấu trúc cdev.

❖ Giá trị trả về:

- Nếu thành công, hàm này trả về 0.
- Nếu thất bại, hàm này trả về một số âm.

Chú ý: hàm này được gọi sau hàm cdev_init.

void cdev_del(struct cdev *p)**8.4.7.2 Cấu trúc của file_operations:**

Cấu trúc **file_operations** gồm các con trỏ hàm. Nhiệm vụ của char driver là gán các hàm của char driver cho các con trỏ ấy. Sau khi gán xong, các hàm này trở thành entry point của char driver.

Khi char driver gọi hàm **cdev_add**, Linux kernel sẽ thiết lập mối liên hệ 1-1 giữa system call và entry point. Ví dụ, system call **open** tương ứng với entry point **open**, system call **close** tương ứng với entry point **release**, system call **read** tương ứng với entry point **read**, system call **write** tương ứng với entry point **write**... Nhờ vậy, các tiến trình có thể phát đi các system call **open**, **read**, **write**, **close** trên device file để tương tác với thiết bị vật lý.

8.4.7.3 Các trường của cấu trúc file_operations:

Cấu trúc **file_operations** này chứa nhiều trường. Nhưng ta sẽ tập trung vào các trường cơ bản.

owner:

Nó là con trỏ đến module chủ của cấu trúc, nó đơn giản được khởi tạo **THIS_MODULE**, một macro được định nghĩa trong **<linux/module.h>**.

open:

int (*open)(struct inode *inode, struct file *filp)

❖ Chức năng:

Khi một tiến trình trên user space gọi system call **open** để mở device file tương ứng với char driver này, thì hàm này sẽ được gọi để thực hiện một số việc như kiểm tra thiết bị đã sẵn sàng chưa, khởi tạo thiết bị nếu cần, lưu lại minor number...

❖ Tham số:

- *inode: địa chỉ của cấu trúc **inode**. Cấu trúc này dùng để mô tả các file trong hệ thống.
- *filp: địa chỉ của cấu trúc **file**. Cấu trúc **file** dùng để mô tả một file đang mở.

❖ Giá trị trả về:

- Hàm này trả về 0 để thông báo mở device file thành công.
- Ngược lại, trả về một số khác 0 để thông báo mở device file thất bại.

release(close):

int (*release)(struct inode *inode, struct file *filp)

❖ Chức năng:

Khi một tiến trình trên user space gọi system call close để đóng device file tương ứng với char driver này, thì hàm này sẽ được gọi để thực hiện một số việc như tắt thiết bị, hoặc làm cho thiết bị ngừng hoạt động

❖ Tham số:

Giống như hàm open

❖ Giá trị trả về:

- Hàm này trả về 0 để thông báo đóng device file thành công.
- Ngược lại, trả về một số khác 0 để thông báo đóng device file thất bại.

read:

ssize_t (*read) (struct file *filp, char __user *buff, size_t size, loff_t *off)

❖ Chức năng:

Đọc dữ liệu từ buffer của char device vào kernel buffer sau đó sao chép dữ liệu từ kernel buffer vào trong user buffer của tiến trình.

❖ Tham số:

- *filp: địa chỉ của cấu trúc file. Cấu trúc này mô tả một device file đang mở.
- *buff : địa chỉ của user buffer.
- size: số lượng byte dữ liệu mà tiến trình cần đọc.
- *off : địa chỉ của cấu trúc loff_t. Cấu trúc này cho biết vị trí trên buffer của char device mà dữ liệu bắt đầu được đọc ra.

❖ Giá trị trả về:

- Trả về một số dương thể hiện số byte đã đọc được từ thiết bị. Trong nhiều trường hợp, nếu số này nhỏ hơn tham số [size], thì tiến trình sẽ tiếp tục gọi system call read cho tới khi nào tổng lượng dữ liệu đọc được bằng tham số [size], hoặc cho tới khi toàn bộ dữ liệu trong buffer của char device được đọc hết.
- Trả về 0 thể hiện rằng toàn bộ dữ liệu trong buffer của char device đã được đọc hết.
- Trả về một số âm nếu có lỗi.

write:

ssize_t (*write) (struct file *filp, char __user *buff, size_t size, loff_t *off)

❖ Chức năng:

Sao chép dữ liệu từ user buffer vào trong kernel buffer sau đó ghi dữ liệu từ kernel buffer vào trong buffer của char device.

❖ Tham số:

- *filp : địa chỉ của cấu trúc file. Cấu trúc này mô tả một device file đang mở.
- *buff : địa chỉ của user buffer.
- size: số lượng byte dữ liệu mà tiến trình cần đọc.
- *off: địa chỉ của cấu trúc loff_t. Cấu trúc này cho biết vị trí trong buffer của char device mà dữ liệu bắt đầu được ghi vào.

❖ Giá trị trả về:

- Trả về một số dương thể hiện số byte đã được ghi vào thiết bị. Trong nhiều trường hợp, nếu số này nhỏ hơn tham số [size], thì tiến trình sẽ tiếp tục gọi system call write cho tới khi nào tổng lượng dữ liệu ghi vào bằng tham số [size], hoặc cho tới khi buffer của char device đầy, không thể ghi được nữa.

- Trả về 0 thể hiện rằng buffer của char device đã đầy, không thể ghi vào được nữa.
- Trả về một số âm nếu có lỗi.

unlock_ioctl:

long (*unlocked_ioctl) (struct file *filp, unsigned int cmd, unsigned long arg);

❖ Chức năng:

Điều chỉnh cấu hình của char device hoặc lấy thông tin của char device. Hàm này thường đọc/ghi các thanh ghi điều khiển hoặc thanh ghi trạng thái của char device.

❖ Tham số:

- *filp: địa chỉ của cấu trúc file. Cấu trúc này mô tả một device file đang mở.
- cmd: mã lệnh yêu cầu. Việc tạo ra mã lệnh này giống như tạo ra mã lệnh cho system call ioctl.
- arg: địa chỉ của tham số chứa dữ liệu cần trao đổi giữa tiến trình và char driver.
- ❖ Giá trị trả về:
 - Trả về 0 thể hiện rằng quá trình điều chỉnh cấu hình hoặc lấy thông tin đã thành công.
 - Trả về một số âm nếu có lỗi.

Ví dụ 5: Tạo ra các entry point(open, close, read, write)**Code:**

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>
```

```
struct char_driver{
    dev_t major_num;
    struct class *class_num;
    struct device *dev_num;
```

```
struct cdev cdev_num;  
}  
  
static int my_open(struct inode *i, struct file *f)  
{  
    printk("Driver: open()\n");  
    return 0;  
}  
  
static int my_close(struct inode *i, struct file *f)  
{  
    printk("Driver: close()\n");  
    return 0;  
}  
  
static ssize_t my_read(struct file *f, char __user *buf, size_t len, loff_t *off)  
{  
    printk("Driver: read()\n");  
    return 0;  
}  
  
static ssize_t my_write(struct file *f, const char __user *buf, size_t len, loff_t *off)  
{  
    printk("Driver: write()\n");  
    return len;  
}
```

```

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_close,
    .read = my_read,
    .write = my_write
};

```

```

static int __init fifth_init(void) /* Constructor */
{
    int ret;

    if ((ret = alloc_chrdev_region(&c_drv.major_num, 0, 1, "CharSample")) < 0)
    {
        printk("Failed to register device number dynamically \n");
        return -1;
    }

    printk("Allocated device number (%d,%d) \n"
    ,MAJOR(c_drv.major_num),MINOR(c_drv.major_num));

    if (IS_ERR(c_drv.class_num = class_create(THIS_MODULE, "CharClass")))
    {
        printk("Failed to create device class\n");

        unregister_chrdev_region(c_drv.major_num, 1);

        return -1;
    }
}

```

```

    }

    if (IS_ERR(c_drv.dev_num) == device_create(c_drv.class_num, NULL,
c_drv.major_num, NULL, "CharDevice")))
    {

        printk("Failed to create a device \n");

        class_destroy(c_drv.class_num);

        unregister_chrdev_region(c_drv.major_num, 1);

        return -1;
    }

    cdev_init(&c_drv.cdev_num, &fops);

    if ((ret = cdev_add(&c_drv.cdev_num, c_drv.major_num, 1)) < 0)
    {

        printk("Failed to create entry points \n");

        device_destroy(c_drv.class_num, c_drv.major_num);

        class_destroy(c_drv.class_num);

        unregister_chrdev_region(c_drv.major_num, 1);

        return -1;
    }

    printk("Initialize char driver successfully\n");

    return 0;
}

static void __exit fifth_exit(void) /* Destructor */
{

```

```
cdev_del(&c_drv.cdev_num);  
device_destroy(c_drv.class_num, c_drv.major_num);  
class_destroy(c_drv.class_num);  
unregister_chrdev_region(c_drv.major_num, 1);  
}
```

```
module_init(fifth_init);  
module_exit(fifth_exit);
```

```
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Phan Tuan Anh");  
MODULE_DESCRIPTION("Create entry points");
```

Dùng Makefile giống ví dụ 1 ta biên dịch ra module tạo file **fifth.ko**.

Nạp module vào linux kernel.

Cách xem các device number, class, device file giống như ví dụ 4. Chúng ta xem các lệnh sau để các entry point: open, close, read, write hoạt động như thế nào nhé.

echo “hello” > /dev/CharDevice : ghi chuỗi “hello” vào device file

cat /dev/CharDevice: đọc data từ device file.

dmesg | tail -7: xem lại quá trình hoạt động của kernel.

```

root@ubuntu:/home/embedded/Documents/driver/bai5
root@ubuntu:/home/embedded/Documents/driver/bai5# make
make -C /lib/modules/3.2.0-23-generic/build M=/home/embedded/Documents/driver/bai5 modules
make[1]: Entering directory `/usr/src/linux-headers-3.2.0-23-generic'
  Building modules, stage 2.
    MODPOST 1 modules
make[1]: Leaving directory `/usr/src/linux-headers-3.2.0-23-generic'
root@ubuntu:/home/embedded/Documents/driver/bai5# insmod fifth.ko
root@ubuntu:/home/embedded/Documents/driver/bai5# echo "hello" > /dev/CharDevice
root@ubuntu:/home/embedded/Documents/driver/bai5# cat /dev/CharDevice
root@ubuntu:/home/embedded/Documents/driver/bai5# rmmod fifth.ko
root@ubuntu:/home/embedded/Documents/driver/bai5# dmesg | tail -7
[ 359.317223] Initialize char driver successfully
[ 377.736895] Driver: open()
[ 377.736914] Driver: write()
[ 377.736918] Driver: close()
[ 384.314293] Driver: open()
[ 384.314338] Driver: read()
[ 384.314342] Driver: close()
root@ubuntu:/home/embedded/Documents/driver/bai5#

```

Khi thực hiện lệnh echo để ghi vào device file thì quá trình sẽ diễn ra: mở device file, ghi data vào file và thực hiện lệnh cat để đọc device file quá trình sẽ đọc từ device file. Lúc này thì dưới kernel thực hiện các entry point tương ứng như tầng ứng dụng. Các entry point trong ví dụ trên chỉ đơn giản là dùng lệnh **prink** để in text tương ứng với chức năng của nó.

8.4.8 Misc driver:

Misc driver hay còn gọi driver hỗn hợp. Misc driver là một hình thức character device driver đơn giản nhất. Nó được nhúng trên nền tảng của bus driver. Trong mục này chúng ta tìm hiểu để cách viết một misc driver.

8.4.8.1 Giới thiệu về misc driver:

Tất cả các misc driver có số major của device number là 10 và modules có thể đăng ký số minor riêng với misc driver. Misc driver tự động tạo cdev và không cần phải tạo bằng tay khi chúng ta cần. Misc driver có thể làm đơn giản cho cách viết character device driver. Chúng ta cần được đăng ký 1 misc driver với Linux nó có cấu trúc như sau được định nghĩa trong file include/linux/miscdevice.h.

```

struct miscdevice
{
    int minor; / Subdevice number /
    const char name; / Device name /
    const struct file_operations fops;/ Device operation function set /
    struct list_head list;
    struct device parent;
    struct device this_device;
    const struct attribute_group groups;
    const char nodename;
    umode_t mode;
};

```

Sau khi xác định loại misc driver chúng ta cần thiết lập 3 tham số sau minor, name và fops. Minor là số device con. Số major của misc driver là 10 đã có định. Chúng ta cần phải chỉ rõ số device con, name là tên của misc device. Khi device đăng ký thành công thì tên device file tên name được tạo ra trong thư mục /dev. fops là kiểu cấu trúc

file_operations, và misc driver cần sử dụng những cấu trúc file_operations được thiết lập bởi người dùng. Sau khi thiết lập misc device chúng ta đăng ký nó với hệ thống bằng cách sử dụng hàm **misc_register**. Nguyên mẫu của hàm này:

int misc_register(struct miscdevice *misc)

Trong đó:

misc: tên cấu trúc miscdevice

Hàm trả về 0 nếu đăng ký thành công, trả về giá trị âm nếu thất bại.

Trước đây khi viết character device driver chúng ta cần tạo rất nhiều hàm theo quy trình:

```
alloc_chrdev_region(); //Application equipment No
cdev_init(); // Initialize cdev
cdev_add(); // Add cdev
class_create(); // Create class
device_create(); //Create device
```

Bây giờ chúng ta sử dụng hàm **misc_register** thay thế các hàm trên. Khi cần gỡ device driver module chúng ta sử dụng hàm **misc_deregister**. Nguyên mẫu của hàm này:

int misc_deregister(struct miscdevice *misc)

Hàm trả về 0 nếu đăng ký thành công, trả về giá trị âm nếu thất bại.

Trước đây khi muốn gỡ module đã cài đặt chúng ta cần gọi nhiều hàm theo quy trình:

```
cdev_del(); // Delete cdev
unregister_chrdev_region(); // Cancel device number
device_destroy(); // Delete device
class_destroy(); // Delete class
```

Giờ chúng ta chỉ cần gọi hàm **misc_deregister** để làm điều này.

Tóm lại khi sử dụng cấu trúc **miscdevice** chúng ta đỡ tốn công viết lại nhiều hàm như trước đây điều này làm cho code driver trở nên đơn giản và dễ đọc hơn.

8.4.8.2 Code lại ví dụ 5 theo misc driver:

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/kdev_t.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/cdev.h>
#include <linux/miscdevice.h>

#define MISC_DEVICE_NAME "misc_test"
#define MISC_DEVICE_MINOR 3

static int misc_open(struct inode *inode, struct file *filep)
{
    printk("Misc Driver:open()\n");
    return 0;
```

```

}

static ssize_t misc_read(struct file *filp, char __user *buff, size_t cnt, loff_t *offt)
{
    printk("Misc Driver:read()\n");
    return 0;
}

static ssize_t misc_write(struct file *filp, char __user *buff, size_t cnt, loff_t *offt)
{
    printk("Misc Driver:write()\n");
    return 0;
}

static int misc_release(struct inode *inode, struct file *filep)
{
    printk("Misc Driver:close()\n");
    return 0;
}

/* Operation function structure */
static struct file_operations misc_fops = {
    .owner = THIS_MODULE,
    .open = misc_open,
    .read = misc_read,
    .write = misc_write,
    .release = misc_release,
};

/* misc Equipment structure */
static struct miscdevice miscdevice_test = {
    .minor = MISC_DEVICE_MINOR,
    .name = MISC_DEVICE_NAME,
    .fops = &misc_fops,
};

static __init int misc_init(void)
{
    int ret = misc_register(&miscdevice_test);
    if (ret < 0)
        return -EFAULT;
    return 0;
}

static __exit void misc_exit(void)
{
}

```

```

        misc_deregister(&miscdevice_test);
    }

module_init(misc_init);
module_exit(misc_exit);

MODULE_AUTHOR("Phan Tuan Anh");
MODULE_DESCRIPTION("Create a simple Misc Driver");
MODULE_LICENSE("GPL");

```

8.4.9 Viết driver cho led đơn kit mini 2440 dùng cấu trúc misc_driver:

Để viết 1 driver thực tế, ta phải hiểu về phần cứng của device, các thanh ghi được sử dụng, địa chỉ vật lý, ngắt. Trong datasheet của kit mini2440, phần cứng của led được kết nối.

Thứ tự LED	GPIO tương ứng	Pin CPU tương ứng
LED1	GPB5	K2
LED2	GPB6	L5
LED3	GPB7	K7
LED4	GPB8	K5

GPIO (general purpose input and output port) là chân ngõ vào, ngõ ra cho mục đích chung. Trong chip S3C2440 các chân GPIO có thể sử dụng nhiều mục đích vì vậy khi sử dụng chúng ta phải khai báo kiểu chân: ngõ vào, ngõ ra..

Theo datasheet của PORTB(trang 284 file S3C2440.pdf)

POR T B CONTROL REGISTERS (GPBCON, GPBDAT, GPBUP)

Register	Address	R/W	Description	Reset Value
GPBCON	0x56000010	R/W	Configures the pins of port B	0x0
GPBDAT	0x56000014	R/W	The data register for port B	Undef.
GPBUP	0x56000018	R/W	Pull-up disable register for port B	0x0
Reserved	0x5600001c			

PBCON	Bit	Description
GPB10	[21:20]	00 = Input 01 = Output 10 = nXDREQ0 11 = reserved
GPB9	[19:18]	00 = Input 01 = Output 10 = nXDACK0 11 = reserved
GPB8	[17:16]	00 = Input 01 = Output 10 = nXDREQ1 11 = Reserved
GPB7	[15:14]	00 = Input 01 = Output 10 = nXDACK1 11 = Reserved
GPB6	[13:12]	00 = Input 01 = Output 10 = nXBREQ 11 = reserved
GPB5	[11:10]	00 = Input 01 = Output 10 = nXBACK 11 = reserved

GPB4	[9:8]	00 = Input 01 = Output 10 = TCLK [0] 11 = reserved
GPB3	[7:6]	00 = Input 01 = Output 10 = TOUT3 11 = reserved
GPB2	[5:4]	00 = Input 01 = Output 10 = TOUT2 11 = reserved
GPB1	[3:2]	00 = Input 01 = Output 10 = TOUT1 11 = reserved
GPB0	[1:0]	00 = Input 01 = Output 10 = TOUT0 11 = reserved

GPBDAT	Bit	Description
GPB[10:0]	[10:0]	When the port is configured as input port, the corresponding bit is the pin state. When the port is configured as output port, the pin state is the same as the corresponding bit. When the port is configured as functional pin, the undefined value will be read.

GPBU	Bit	Description
GPB[10:0]	[10:0]	0: The pull up function attached to the corresponding port pin is enabled. 1: The pull up function is disabled.

Trước tiên ta phải cấu hình các pin led kiểu output. GPB5 bit [11:10]=01, GPB6 bit [13:12]=01, GPB7 bit [15:14]=01, GPB8 bit [17:16]=01 và thanh ghi GPBDAT được dùng để thiết lập kiểu của GPIO.

Theo sơ đồ phần cứng chân GPB5,6,7,8 ở mức thấp thì led sáng và mức cao thì led tắt. Ở kernel có 1 số hàm hay macro đã hỗ trợ sẵn để chúng ta có thể dễ dàng sử dụng. Ví dụ: `s3c2410_gpio_cfgpin`, `s3c2410_gpio_setpin`, `s3c2410_gpio_getpin`. Tất cả đều nằm trong file `linux/arch/arm/plat-s3c24xx/gpio.c` Chúng ta tìm hiểu các hàm này:

- `void s3c2410_gpio_cfgpin(unsigned int pin, unsigned int function):`
Dùng để thiết lập chân pin với chức năng function(input, output,...)
- `void s3c2410_gpio_setpin(unsigned int pin, unsigned int dat)`
Dùng để gán chân pin ở mức dat(0,1)
- `unsigned int s3c2410_gpio_getpin(unsigned int pin)`
Dùng để lấy đọc trạng thái của chân pin(mức 0,1)

Trên thực tế chúng ta không cần quan tâm đến cách viết của các macro này miễn là ta biết cách sử dụng chúng để viết driver. Dưới đây chúng ta xem code driver của led đơn trong kit mini 2440.

```
/ *linux/drivers/char/mini2440_leds.c
```

Copyright (c) 2013 Feng Guoqing

mini2440 LEDs Driver

*This file is subject to the terms and conditions of the GNU General Public License. See the file COPYING in the main directory of this archive for more details. */*

```

#include <linux/miscdevice.h>
#include <linux/delay.h>
#include <asm/irq.h>
#include <machregs-gpio.h>
#include <mach/hardware.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/mm.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/delay.h>
#include <linux/moduleparam.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/ioctl.h>
#include <linux/cdev.h>
#include <linux/string.h>
#include <linux/list.h>
#include <linux/pci.h>
#include <linux/gpio.h>
#include <asm/uaccess.h>
#include <asm/atomic.h>
#include <asm/unistd.h>

#define DEVICE_NAME "myleds"

static unsigned long led_table [] = {
S3C2410_GPB(5),
S3C2410_GPB(6),
S3C2410_GPB(7),
S3C2410_GPB(8),
};

static unsigned int led_cfg_table [] = {
S3C2410_GPIO_OUTPUT,
S3C2410_GPIO_OUTPUT,
S3C2410_GPIO_OUTPUT,
S3C2410_GPIO_OUTPUT,
};

static int sbc2440_leds_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    switch(cmd) {
        case 0:

```

```

case 1:
    if(arg > 4) {
        return -EINVAL;
    }
    s3c2410_gpio_setpin(led_table[arg],!cmd);
    return 0;
default:
    return -EINVAL;
}
}

static struct file_operations dev_fops = {
.owner = THIS_MODULE,
.unlocked_ioctl = sbc2440_leds_ioctl,
};

static struct miscdevice misc = {
.minor = MISC_DYNAMIC_MINOR,
.name = DEVICE_NAME,
.fops = &dev_fops,
};

static int __init dev_init(void)
{
int ret;
int i;

for (i = 0; i < 4; i++) {
s3c2410_gpio_cfgpin(led_table[i], led_cfg_table[i]);
s3c2410_gpio_setpin(led_table[i], 1);
}
ret = misc_register(&misc);
printk (DEVICE_NAME"\tinitialized\n");

return ret;
}
static void __exit dev_exit(void)
{
misc_deregister(&misc);
}

module init(dev_init);
module_exit(dev_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("FriendlyARM Inc.");

```

Trong code trên lưu ý có hàm **static int sbc2440_leds_ioctl(struct file f,unsigned int cmd,unsigned long arg)**. Hàm này có 3 tham số f: device file, cmd: là tham số (cmd=0: led tắt, 1: led sáng, arg: là vị trí led(0:3))

Viết Makefile để biên dịch driver:

Driver này chạy trên kit mini2440 nên Makefile có khác so với bài trước.

```
obj-m += mini2440_leds.o
all:
    make -C /lib/modules/2.6.32.2-FriendlyARM/build M=$(PWD) modules
clean:
    make -C /lib/modules/2.6.32.2-FriendlyARM/build M=$(PWD) clean
```

Sau khi biên dịch driver được file

Viết ứng dụng để test driver đã tạo:(tất cả đều đặt trong cùng 1 thư mục với Makefile, và mini2440_leds.c)

```
/*leds.c: ứng dụng dùng để test driver */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/select.h>
#include <sys/time.h>
#include <string.h>
int main(int argc, char **argv)
{
    int status;
    int led_no;
    int fd;
    if(argc != 3 || sscanf(argv[1], "%d", &status) != 1 || sscanf(argv[2], "%d",
&led_no) != 1 ||
        status < 0 || status > 1 || led_no < 0 || led_no > 3)
    {
        fprintf(stderr, "Usage: leds led_no 0|1\n");
        exit(1);
    }
    //Open leds device file
    fd = open("/dev/myleds", 0);
    if(fd < 0) {
        perror("error open device leds\n");
        exit(1);
    }
    ioctl(fd, status, led_no);
    close(fd);
```

```

        return 0;
}

```

Biên dịch chéo ứng dụng đã viết dùng lệnh:

arm-linux-gcc -o leds leds.c

Chuyển 2 file leds và mini2440_leds.ko xuống kit mini2440 qua **gftp**.

Lắp driver cho embedded linux kernel dùng lệnh:

insmod mini2440_leds.ko

Cấp quyền cho file leds

chmod 777 leds

Chạy ứng dụng để xem kết quả test driver:

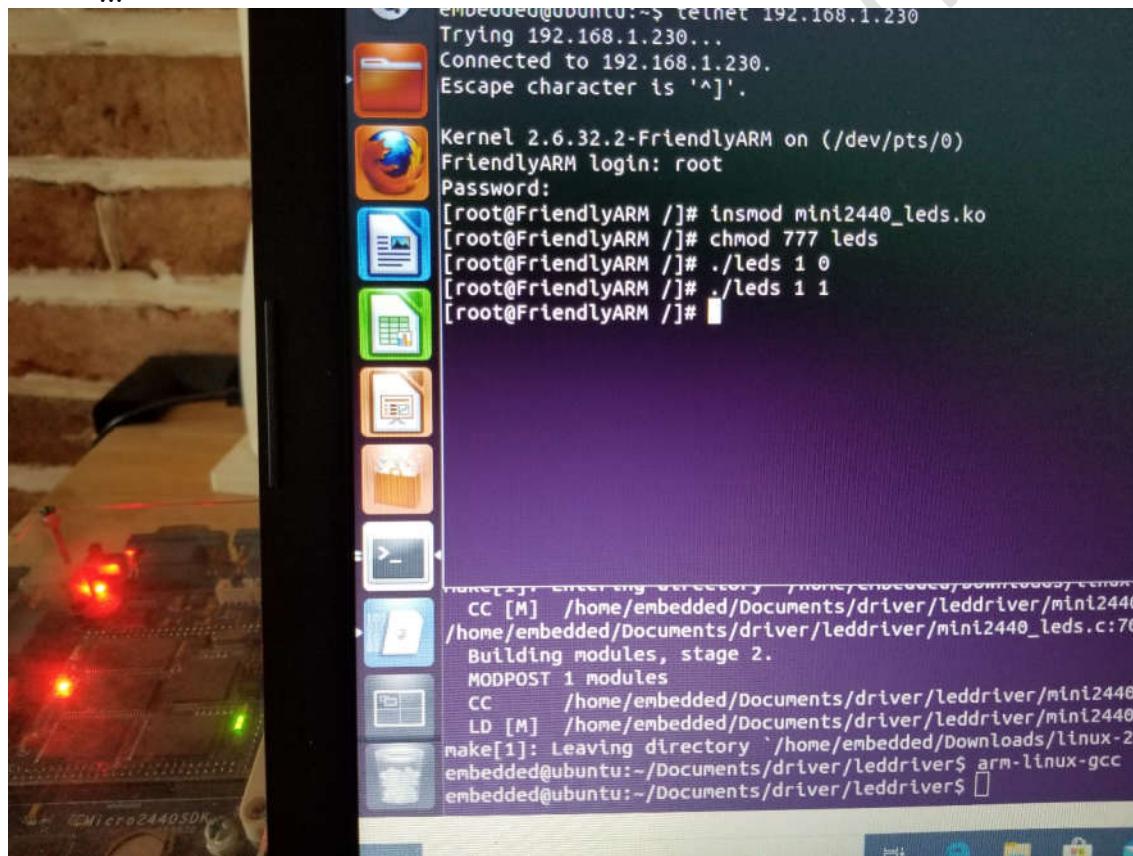
./leds 0 0: led 0 tắt

./leds 1 0: led 0 sáng

./leds 1 1: led 1 sáng

...

...



8.4.10 Viết driver cho nút nhấn kit mini 2440 dùng cấu trúc misc_driver:

/linux/drivers/char/mini2440_buttons.c

Copyright (c) 2013 Feng Guoqing

mini2440 buttons Driver

This file is subject to the terms and conditions of the GNU General Public License. See the file COPYING in the main directory of this archive for more details.

/

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <linux/poll.h>
#include <linux/irq.h>
#include <asm/irq.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <machregs-gpio.h>
#include <mach/hardware.h>
#include <linux/platform_device.h>
#include <linux/cdev.h>
#include <linux/miscdevice.h>
#include <linux/sched.h>
#include <linux/gpio.h>

#define DEVICE_NAME "buttons"

struct button_irq_desc {
    int irq;
    int pin;
    int pin_setting;
    int number;
    char name;
};

static struct button_irq_desc button_irqs [] = {
{IRQ_EINT8, S3C2410_GPG(0), S3C2410_GPG0_EINT8, 0, "KEY0"}, 
{IRQ_EINT11, S3C2410_GPG(3), S3C2410_GPG3_EINT11, 1, "KEY1"}, 
{IRQ_EINT13, S3C2410_GPG(5), S3C2410_GPG5_EINT13, 2, "KEY2"}, 
{IRQ_EINT14, S3C2410_GPG(6), S3C2410_GPG6_EINT14, 3, "KEY3"}, 
{IRQ_EINT15, S3C2410_GPG(7), S3C2410_GPG7_EINT15, 4, "KEY4"}, 
{IRQ_EINT19, S3C2410_GPG(11), S3C2410_GPG11_EINT19, 5, "KEY5"}, 
};

static volatile char key_values [] = {'0', '0', '0', '0', '0', '0'};

static DECLARE_WAIT_QUEUE_HEAD(button_waitq);
```

```

static volatile int ev_press = 0;

static irqreturn_t buttons_interrupt(int irq, void dev_id)
{
    struct button_irq_desc button_irqs = (struct button_irq_desc )dev_id;
    int down;

    down = !gpio_get_value(button_irqs->pin);

    if (down != (key_values[button_irqs->number] & 1)) {
        key_values[button_irqs->number] = '0' + down;
        ev_press = 1;
        wake_up_interruptible(&button_waitq);
    }
    return IRQ_RETVAL(IRQ_HANDLED);
}

static int s3c24xx_buttons_open(struct inode inode, struct file file)
{
    int i;
    int err = 0;
    for (i = 0; i < sizeof(button_irqs)/sizeof(button_irqs[0]); i++) {
        if (button_irqs[i].irq < 0) {
            continue;
        }
        err = request_irq(button_irqs[i].irq, buttons_interrupt, IRQ_TYPE_EDGE_BOTH,
                          button_irqs[i].name, (void )&button_irqs[i]);
        if (err) break;
    }

    if (err) {
        i--;
        for (; i >= 0; i--) {
            if (button_irqs[i].irq < 0) {
                continue;
            }
            disable_irq(button_irqs[i].irq); free_irq(button_irqs[i].irq, (void )&button_irqs[i]);
        }
        return -EBUSY;
    }

    ev_press = 1;

    return 0;
}

```

```

}

static int s3c24xx_buttons_close(struct inode inode, struct file file)
{
int i;
for (i = 0; i < sizeof(button_irqs)/sizeof(button_irqs[0]); i++) {
if (button_irqs[i].irq < 0) {
continue;
}
free_irq(button_irqs[i].irq, (void *)&button_irqs[i]);
}
return 0;
}

static int s3c24xx_buttons_read(struct file *filp, char __user *buff, size_t count, loff_t offp)
{
unsigned long err;
if (!ev_press) {
if (filp->f_flags & O_NONBLOCK) {
return -EAGAIN;
} else {
wait_event_interruptible(button_waitq, ev_press);
}
}
ev_press = 0;

err = copy_to_user(buff, (const void *)key_values, min(sizeof(key_values), count));
return err ? -EFAULT : min(sizeof(key_values), count);
}

static unsigned int s3c24xx_buttons_poll(struct file *file, struct poll_table_struct *wait)
{
unsigned int mask = 0;
poll_wait(file, &button_waitq, wait);
if (ev_press)
mask |= POLLIN | POLLRDNORM;
return mask;
}

static struct file_operations dev_fops = {
.owner = THIS_MODULE,
.open = s3c24xx_buttons_open,
.release = s3c24xx_buttons_close,
}

```

```
.read = s3c24xx_buttons_read,  
.poll = s3c24xx_buttons_poll,  
};  
  
static struct miscdevice misc = {  
.minor = MISC_DYNAMIC_MINOR,  
.name = DEVICE_NAME,  
.fops = &dev_fops,  
};  
  
static int __init dev_init(void)  
{  
int ret;  
  
ret = misc_register(&misc);  
printk(DEVICE_NAME"\tinitialized\n");  

```

Với driver này sinh viên tự tìm hiểu code, viết ứng dụng để test driver này.

8.5 Bài tập thực hành:

Bài 1:

-Viết 1 device driver của led (dựa vào ví dụ trên) có tính năng sau:
cmd=0: led tắt, cmd=1: led sáng, cmd=3: led đảo trạng thái(trạng thái hiện tại: tắt, thì điều khiển sáng và ngược lại).

-Viết 1 app để test driver này.

Bài 2:

-Viết 1 device driver của led (dựa vào ví dụ trên) có tính năng sau:
cmd=0: led tắt, cmd=1: led sáng, cmd=3: led chớp tắt 5 lần và thời gian chớp tắt là 300ms

-Viết 1 app để test driver này.

Bài 3:

-Viết 1 device driver kết hợp giữa 4 led và 4 nút nhấn có tính năng: khi người dùng tác động và nút nhấn nào thì led tương ứng đó sáng.

-Viết 1 app để test driver này.

CHƯƠNG 9: LẬP TRÌNH GIAO DIỆN QT TRONG LINUX

9.1 Mục đích:

- Hiểu được sơ đồ cấu trúc từ tầng ứng dụng đến phần cứng.
- Lập trình QT trong linux máy host
- Viết được các ứng dụng dùng QT chạy trên kit dùng các device đã học.

9.2 Yêu cầu:

- Có hiểu biết về phần cứng các device.
- Có kiến thức về lập trình C, lập trình giao diện
- Có kiến thức về linux

9.3 Chuẩn bị:

- PC có hệ điều hành linux đã cài đặt trình biên chéo arm-linux-gcc.
- PC có cài QT everywhere, QT creator.

9.4 Hướng dẫn cách cài đặt QT:

Trong phần này ta sẽ cài đặt QT SDK trên máy host linux để phát triển ứng dụng chạy trên PC và QT everywhere(QT4.6.2) để cho phép ứng dụng chạy trên kit.

9.4.1 Cài đặt QT SDK:

Bước 1: Di chuyển file cài đặt **qt-sdk-linux-x86-opensource-2010.05.1.bin** vào thư mục nào đó để cài đặt và cấp quyền thực thi cho nó:

```
$ chmod u+x qt-sdk-linux-x86-opensource-2010.05.1.bin
```

Bước 2: Thực thi cài đặt từ dòng lệnh:

```
$ ./qt-sdk-linux-x86-opensource-2010.05.1.bin
```

Đợi quá trình cài đặt diễn ra thành công, mặc định thư mục cài đặt chứa tại \$HOME/qtsdk-2010.01/qt/bin

Sau khi cài đặt xong Qt SDK, công cụ Qt Creator cho phép phát triển ứng dụng với lựa chọn mặc định biên dịch trên máy tính Linux. Để biên dịch chéo ứng dụng

thực thi trên KIT FriendlyArm (Embedded Linux) cần cài đặt Qt Everywhere

9.4.2 Cài đặt QT Everywhere trên host Linux:

Mục đích của phần này là tạo ra 1 ứng dụng QT tương thích trên kit mini2440 mà được viết trên máy host linux. Chúng ta cần biên dịch thư viện tslib hỗ trợ màn hình cảm ứng và QT everywhere. Sau đó copy thư viện này lên kit để chạy. Trình tự các bước như sau:

- Biên dịch thư viện tslib
- Cài đặt QT everywhere Qt4.6.2
- Copy thư viện tslib và Qt4.6.2 vào kit mini2440
- Cấu hình môi trường của kit mini2440 để chạy ứng dụng.
- Chạy 1 ứng dụng trên kit.

Trước khi cài đặt chúng ta cần cài đặt 1 số gói phần mềm cũng như thư viện để hỗ trợ:

```
sudo apt-get install g++
```

```

sudo apt-get install libx11-dev
sudo apt-get install libxext-dev
sudo apt-get install xorg-dev
sudo apt-get install autoconf
sudo apt-get install automake
sudo apt-get install libtool
sudo apt-get install zlib-bin
sudo apt-get install zlib1g-dev
sudo apt-get install zlib1g
sudo apt-get install git-core

```

Biên dịch thư viện tslib:

TSLIB là thư viện dùng để sử dụng cho màn hình cảm biến. Nó được sử dụng trên thiết bị nhúng để cung cấp giao diện cho không gian người dùng.

```

$cd /usr/local
$git clone http://github.com/kergoth/tslib.git( quyền root)
$export PATH=/usr/local/arm/4.3.2/bin:$PATH
$export CROSS_COMPILE=arm-none-linux-gnueabi-
$export CC=${CROSS_COMPILE}gcc
$export CFLAGS=-march=armv4t
$export CXX=${CROSS_COMPILE}"g++"
$export AR=${CROSS_COMPILE}"ar"
$export AS=${CROSS_COMPILE}"as"
$export RANLIB=${CROSS_COMPILE}"ranlib"
$export LD=${CROSS_COMPILE}"ld"
$export STRIP=${CROSS_COMPILE}"strip"
$export ac_cv_func_malloc_0_nonnull=yes
$cd /usr/local/tslib
$./autogen-clean.sh
$./autogen.sh

$./configure --host=arm-linux --prefix=/home/tslib --enable-shared=yes --enable-static=yes
$make
$make install

```

Sau khi biên dịch tslib thành công, ta sẽ thấy folder tslib nằm trong thư mục /home.

Biên dịch QT everywhere 4.6.2:

Copy source **qt-everywhere-opensource-src-4.6.2.tar.gz** vào thư mục /usr/local

```
$cd /usr/local
$tar -zvxf qt-everywhere-opensource-src-4.6.2.tar.gz
$cd qt-everywhere-opensource-src-4.6.2
$cd mkspecs/common/
$gedit g++.conf
```

Chỉnh sửa trong file g++.conf:

QMAKE_CFLAGS_RELEASE += -O2 thành

QMAKE_CFLAGS_RELEASE += -O0

Lưu lại file

```
$cd /usr/local/qt-everywhere-opensource-src-4.6.2/mkspecs/qws/linux-arm-g++
$gedit qmake.conf
```

Chỉnh sửa lại file qmake.conf như sau:

```
#  
# qmake configuration for building with arm-linux-g++  
#  
  
include(../common/g++.conf)  
include(../common/linux.conf)  
include(../common/qws.conf)  
# modifications to g++.conf  
  
QMAKE_CC = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-gcc -  
msoft-float -D__GCC_FLOAT_NOT_NEEDED -march=armv4t -mtune=arm920t -  
O0 -lts  
QMAKE_CXX = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-g++ -  
msoft-float -D__GCC_FLOAT_NOT_NEEDED -march=armv4t -mtune=arm920t -  
O0 -lts  
QMAKE_LINK = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-g++ -  
msoft-float -D__GCC_FLOAT_NOT_NEEDED -march=armv4t -mtune=arm920t -  
O0 -lts  
QMAKE_LINK_SHLIB = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-  
g++ -msoft-float -D__GCC_FLOAT_NOT_NEEDED -march=armv4t -  
mtune=arm920t -O0 -lts  
  
# modifications to linux.conf
```

```

QMAKE_AR          = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-ar cqs
QMAKE_OBJCOPY     = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-
objcopy
QMAKE_STRIP       = /usr/local/arm/4.3.2/bin/arm-none-linux-gnueabi-strip

QMAKE_INCDIR    += /usr/local/comp-tslib/include/
QMAKE_LIBDIR    += /usr/local/comp-tslib/lib/

QMAKE_CFLAGS_RELEASE  += -march=armv4 -mtune=arm920t
QMAKE_CFLAGS_DEBUG   += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_MT      += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_MT_DBG  += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_MT_DLL  += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_MT_DLLDBG += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_SHLIB   += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_THREAD  += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_WARN_OFF  += -march=armv4t -mtune=arm920t
QMAKE_CFLAGS_WARN_ON   += -march=armv4t -mtune=arm920t

QMAKE_CXXFLAGS_DEBUG  += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_MT     += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_MT_DBG  += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_MT_DLL  += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_MT_DLLDBG += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_RELEASE  += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_SHLIB   += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_THREAD  += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_WARN_OFF  += -march=armv4t -mtune=arm920t
QMAKE_CXXFLAGS_WARN_ON   += -march=armv4t -mtune=arm920t

```

load(qt_config)

Lưu lại file

```

$mkdir /usr/local/Qt
$cd /usr/local/qt-everywhere-opensource-src-4.6.2
$./configure -embedded arm -xplatform qws/linux-arm-g++ -prefix /usr/local/Qt -qt-
mouse-tslib -little-endian -no-webkit -no-qt3support -no-cups -no-largefile -
optimized-qmake -no-openssl -nomake tools -qt-sql-sqlite -no-3dnow -system-zlib -
qt-gif -qt-libtiff -qt-libpng -qt-libmng -qt-libjpeg -no-opengl -gtkstyle -no-openvg -

```

```
no-xshape -no-xsync -no-xrandr -qt-freetype -qt-gfx-linuxfb -qt-kbd-tty -qt-kbd-
linuxinput -qt-mouse-tslib -qt-mouse-linuxinput
```

Chọn ‘o’ nếu có hỏi *Open Source Edition*

Chọn ‘yes’ để *accept license offer*

\$make

\$make install

Khi biên dịch xong kết quả sẽ nằm trong /usr/local/Qt

Copy thư viện tslib và QT đã biên dịch xong vào kit mini2440:

Tốt nhất ta chuẩn bị 1 USB đã được format FAT32. Nó sẽ copy thư viện tslib, QT và ví dụ QT từ linux host sang kit để chạy chương trình.

Copy thư viện và ví dụ QT vào USB. Gắn USB vào và chuyển sang chế độ dùng trong máy ảo linux. Giả sử USB trong máy ảo linux là Pendrive.

Copy từ linux host sang USB:

```
$cd /usr/local/Qt/lib
$cp *4.6.2 /media/Pendrive/
Đổi tên các file vừa copy có đuôi *.4.6.2 sang đuôi*.4
$cp -r fonts/ /media/Pendrive/
$cd /usr/local/Qt
$cp -r demos/ /media/Pendrive/

$mkdir /media/Pendrive/tslib/lib/
$cd /home/tslib/lib/
$cp -r */media/Pendrive/tslib/lib/
$cp */media/Pendrive/tslib/lib/
```

Gắn USB vào cổng USB của kit và copy từ USB sang kit:

```
$mkdir /usr/local/Qt/lib/
$cp /sdcard/*.4 /usr/local/Qt/lib/
$cp -r /udisk/fonts/ /usr/local/Qt/lib/
$cp -r /udisk/demos/ /mnt/
$cp -r /udisk/tslib/ /usr/local/
```

Trên kit ô USB là udisk

Cấu hình môi trường của kit mini2440 để chạy ứng dụng:

Những ứng dụng viết QT trên máy host khi được biên dịch chéo để chạy trên kit do màn hình, cảm biến trên kit khác trên máy tính nên phải cấu hình lại trên kit để chạy được ứng dụng.

```
$cd /etc/init.d/
```

```
$vi rcS
```

Vô hiệu hóa 3 dòng trên để khi khởi động kit không vào Qtopia.

```
#bin/qtopia&
#echo"                "> /dev/tty1
#ehco"Starting Qtopia, please waiting..." > /dev/tty1
```

```
$cd /etc/
```

```
$vi profile
```

Thêm đoạn lệnh sau vào cuối file:

```
export LD_LIBRARY_PATH=/usr/local/tslib/lib
export QTDIR=/usr/local/Qt
export QWS_MOUSE_PROTO=tslib:/dev/input/event0
export TSLIB_CALIBFILE=/etc/pointercal
export TSLIB_CONFFILE=/usr/local/etc/ts.conf
export TSLIB_CONSOLEDEVICE=none
export TSLIB_FBDEVICE=/dev/fb0
export TSLIB_PLUGINDIR=/usr/local/tslib/lib/ts
export TSLIB_TSDEVICE=/usr/local/tslib/lib/ts
export TSLIB_TSEVENTTYPE=INPUT
export QWS_DISPLAY=LinuxFB:mmWidth=105:mmHeight=140
```

Lưu file lại.

Chạy 1 chương trình mẫu QT:

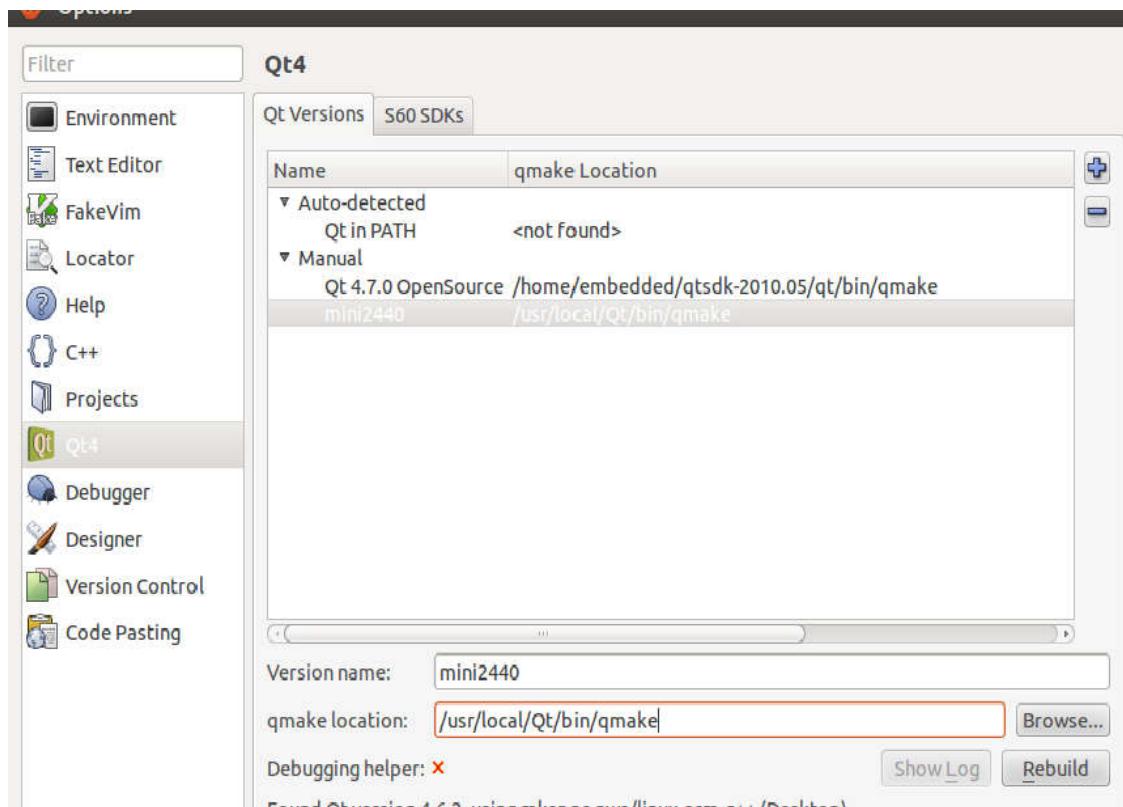
Vào thư mục chứa code của ví dụ:

```
$cd /mnt/demos/embedded/fluidlauncher/
$./fluidlauncher -qws
```

9.4.3 Viết 1 project QT đơn giản để chạy trên kit:

Trước tiên ta tạo 1 biên dịch chéo để viết ứng dụng QT creator trên desktop và biên dịch ứng dụng này chạy được trên kít.

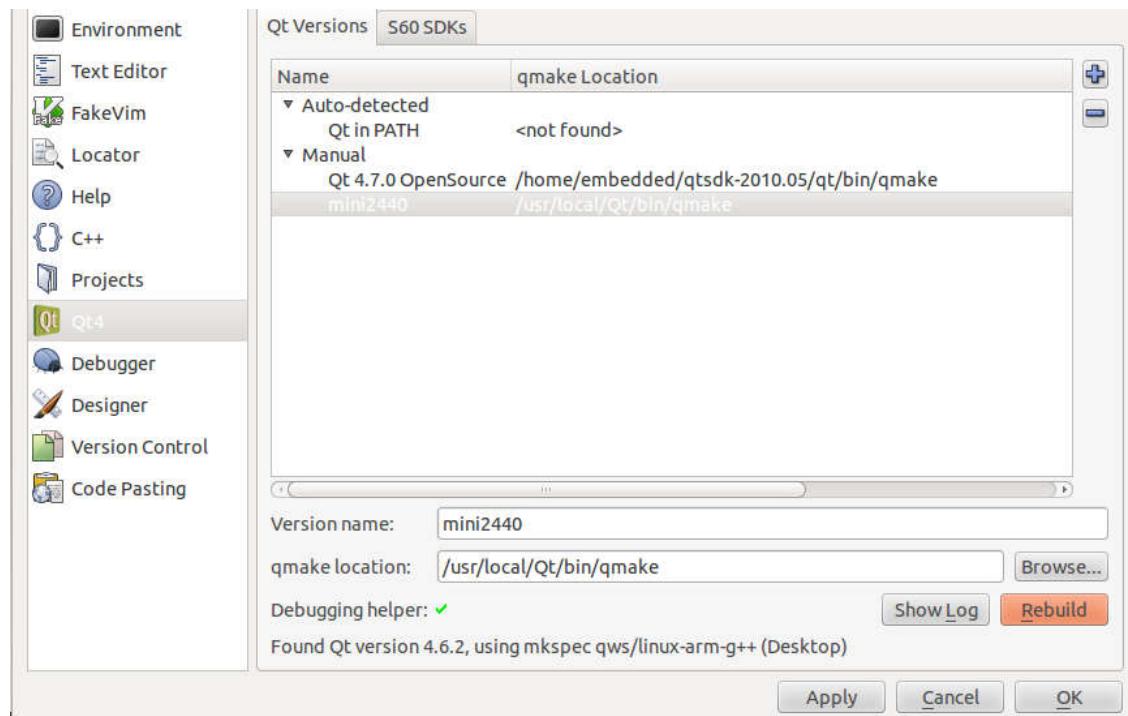
Mở QT creator. Vào Tools-Options-QT4. Click vào dấu + bên phải.



Mục Version name: vào tên mini2440.

qmake location: /usr/local/Qt/bin/qmake.

Sau đó click vào Rebuild. Kết quả

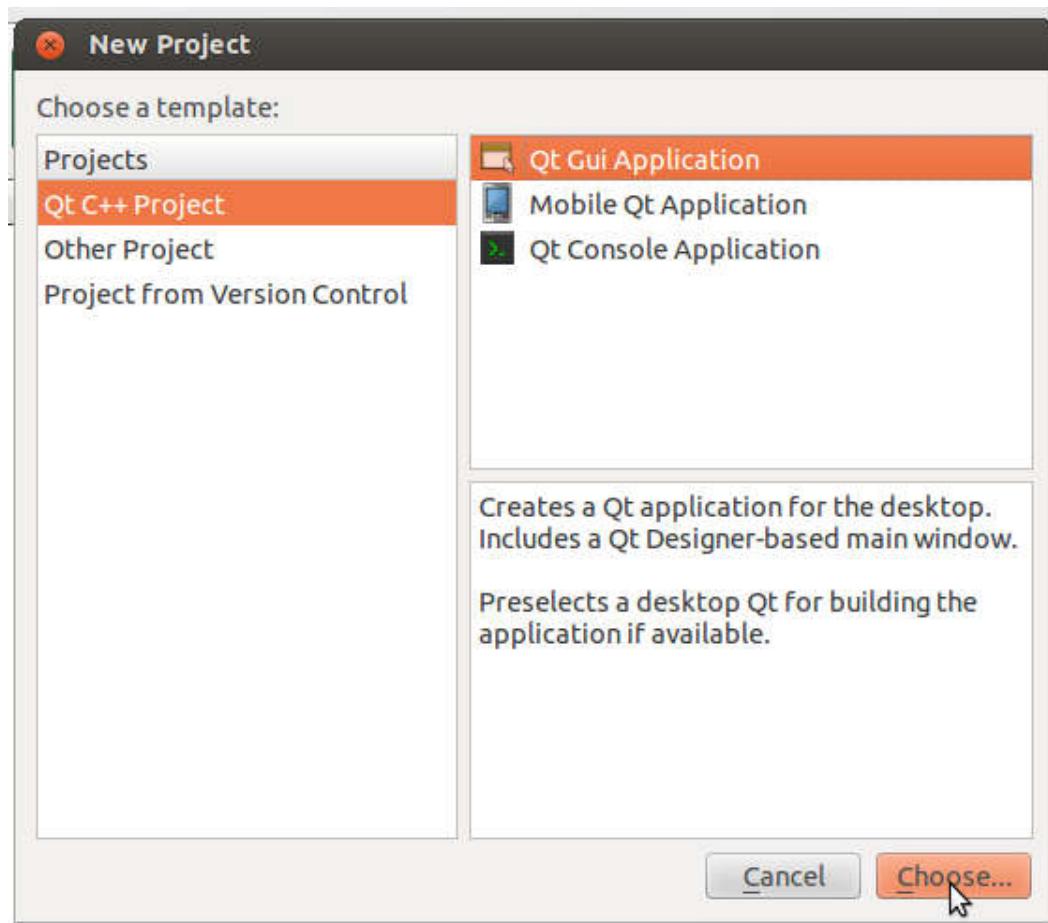


Dòng chữ Found QT version 4.6.2, using mkspec qws/linux-arm-g++(Destop) là tạo thành công. Click nút OK.

Tạo project:

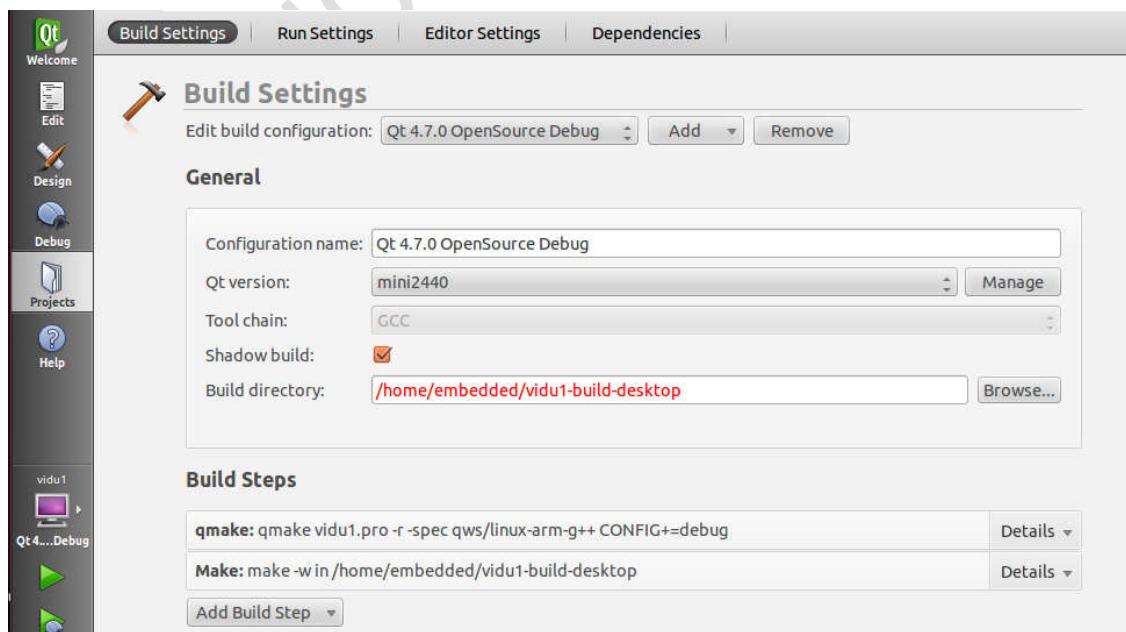
Bước 1:

File → New File or Project → QT Gui Application

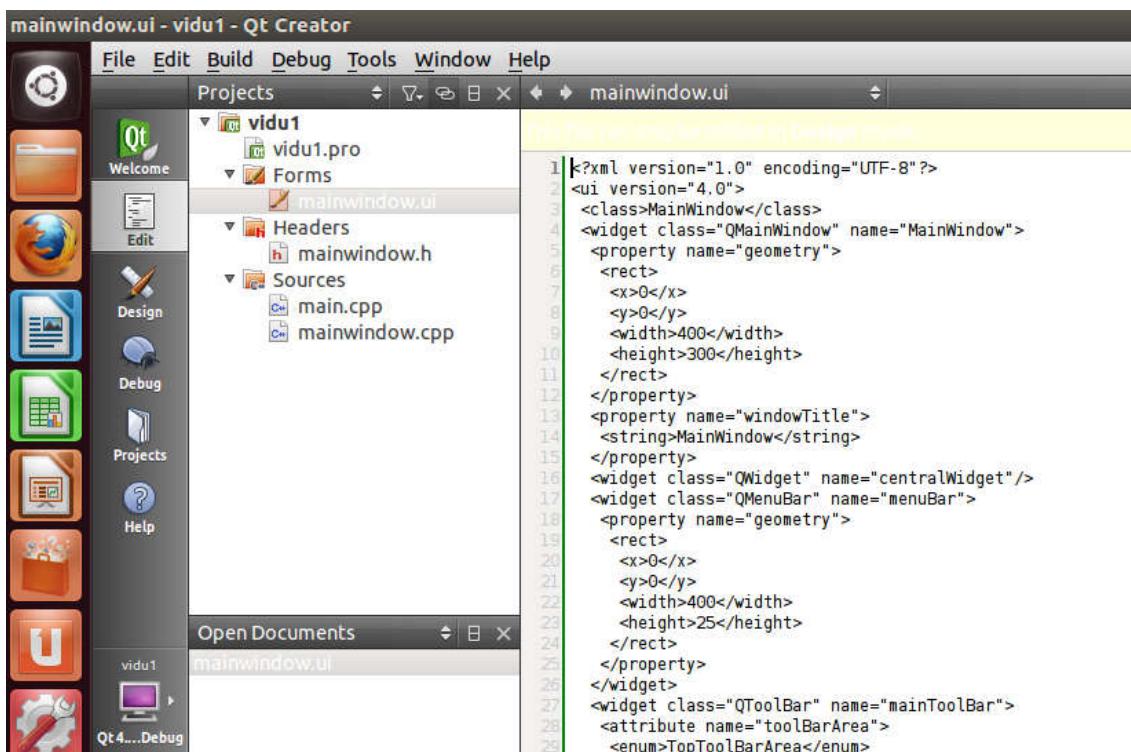


Vào tên vidu1, chọn thư mục chứa project → Next ..next cho đến Finish.

Vào mục Project-Build Settings. Phần QT version chọn mini2440.



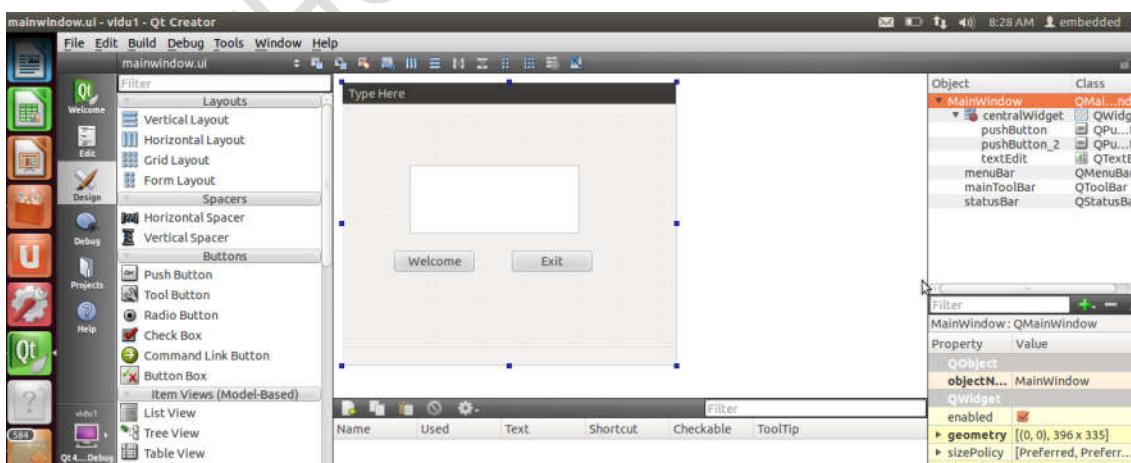
Vào mục Edit xem project đã tạo ra.



Trong project vidu1 có vidu1.pro là tên dự án, phần mainwindow.ui để tạo giao diện, mainwindow.h để khai các biến hay hàm, mainwindow.cpp, main.cpp: dùng để viết code.

Bước 2:

Thiết kế 1 form đơn giản gồm 2 pushButton và 1 textEdit.



Phần thiết kế giao diện có thể dùng phần mềm QT4 designer được cài đặt từ Ubuntu Software Center.

Click phải chuột vào Welcome → Go to slot... → click() → OK

Thêm code cho pushButton này để hiện thị lên textEdit:

```
ui->textEdit->setText("Welcome a QT program");
```

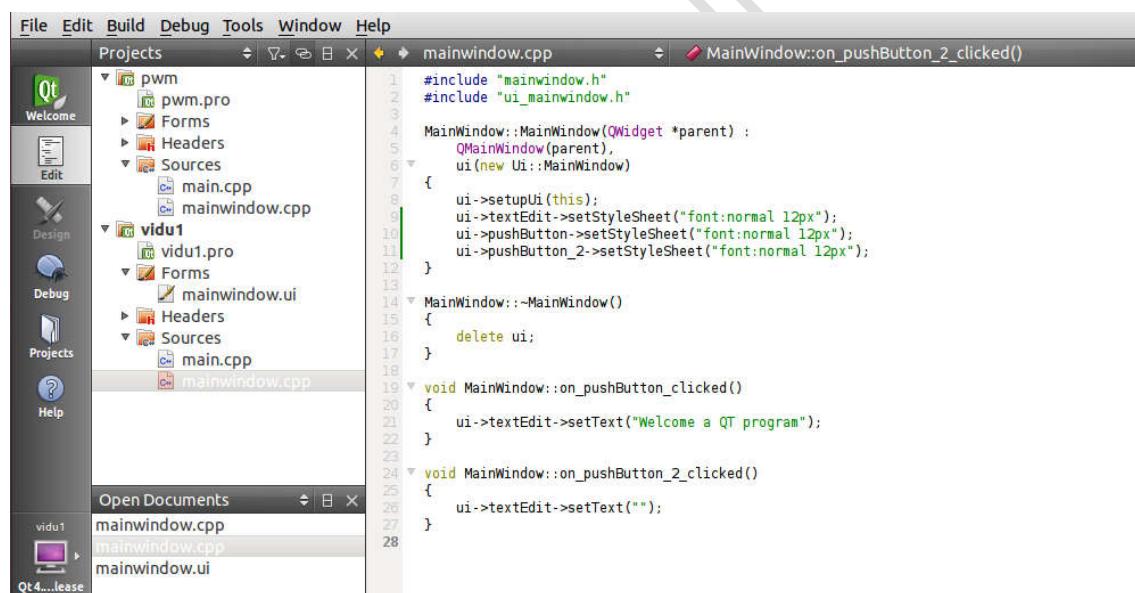
Click phải chuột vào Exit → Go to slot... → click() → OK

Thêm code cho pushButton này để xóa hiển thị trên textEdit:

```
ui->textEdit->setText("");
```

Thêm code trong phần MainWindow::MainWindow(QWidget *parent) để hiển thị kích thước các object

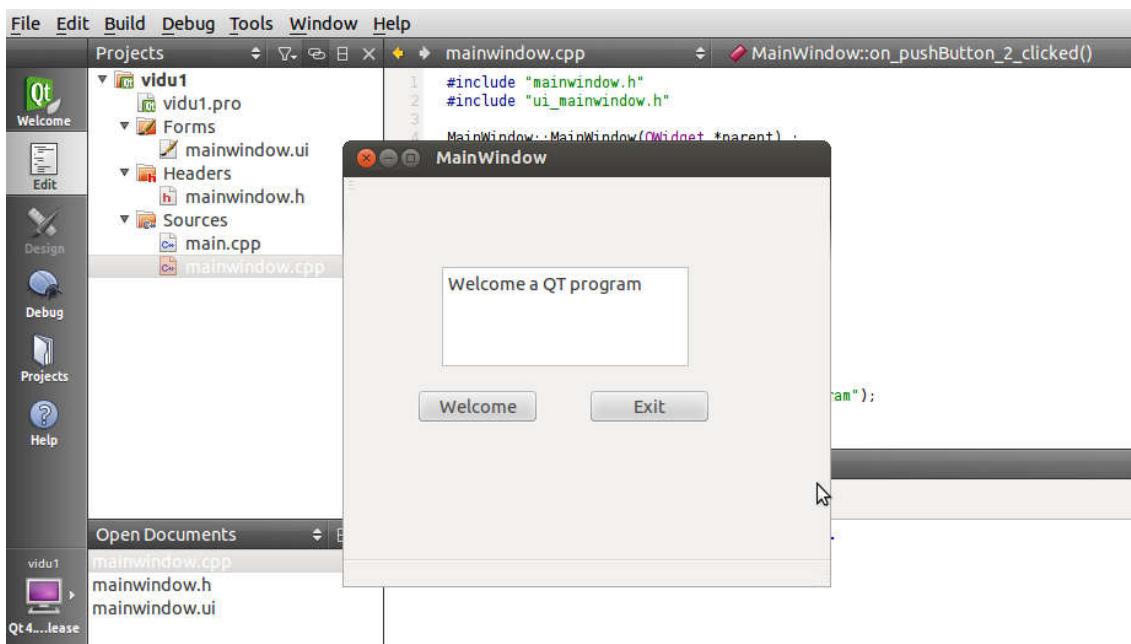
```
ui->textEdit->setStyleSheet("font:normal 12px");  
ui->pushButton->setStyleSheet("font:normal 12px");  
ui->pushButton_2->setStyleSheet("font:normal 12px");
```



Vào phần cấu hình góc trái ở dưới có icon màn hình màu tím vào mục **build** chọn QT4.7.0 OpenSource Realease để chạy trên máy host.

Vào Build-Build Project “vidu1”.

Sau đó click Run để chạy chương trình trên máy host:



Bây giờ chúng ta thiết lập lại chương trình này chạy trên kit.

Dừng chương trình đang chạy, vào lại icon màu tím ở dưới vào mục build chọn **mini2440 Release**.

Vào Build-Rebuild Project “vidu1”.

Vào lúc này chương trình sẽ biên dịch ra file vidu1 nằm trong thư mục vidu1-build-desktop nằm gần với thư mục vidu1. Để chạy chương trình trên linux nhúng ta chuyển file này xuống dưới kit giả sử nằm trong /home.

```
[root@FriendlyARM /home]# chmod 777 vidu1
```

```
[root@FriendlyARM /home]# ./vidu1 -qws
```

Ta có thể tạo một script file **myapp.sh**(giả sử đặt trong /home) để chạy app trên kit với nội dung sau:

```
TH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin
```

```
PSI='[u@|h |W]|$ '
```

```
export PATH
```

```
alias ll='ls -l'
```

```
alias la='ll -a'
```

```
export PSI='|u@|h |w$ '
```

```

export PS2='> '
export PS3='? '
export PS4='[$LINENO]+'
export GST_PLUGIN_PATH=/usr/lib/fsl_mm_linux/lib/gstreamer-0.10
export TSLIB_ROOT=/usr/local
export TSLIB_CONSOLEDEVICE=none
export TSLIB_FBDEVICE=/dev/fb0
export TSLIB_TSDEVICE=/dev/input/event0
export TSLIB_PLUGINDIR=$TSLIB_ROOT/lib/ts
export TSLIB_CONFFILE=$TSLIB_ROOT/etc/ts.conf
export TSLIB_CALIBFILE=/etc/pointercal
export QWS_MOUSE_PROTO=Tslib:/dev/input/event0
export QWS_DISPLAY=LinuxFb:/dev/fb0
export set QT_QWS_FONTDIR=/usr/local/Qt/lib/fonts
export QT_PLUGIN_PATH=/usr/local/Qt/plugins:/usr/local/Qt/plugins/imageformats
export
LD_LIBRARY_PATH=$TSLIB_ROOT/lib:/usr/lib:/usr/local/Qt/lib:$LD_LIBRARY_PATH
export HOME=/root
cd /home
./vidul -qws
hotplug

```

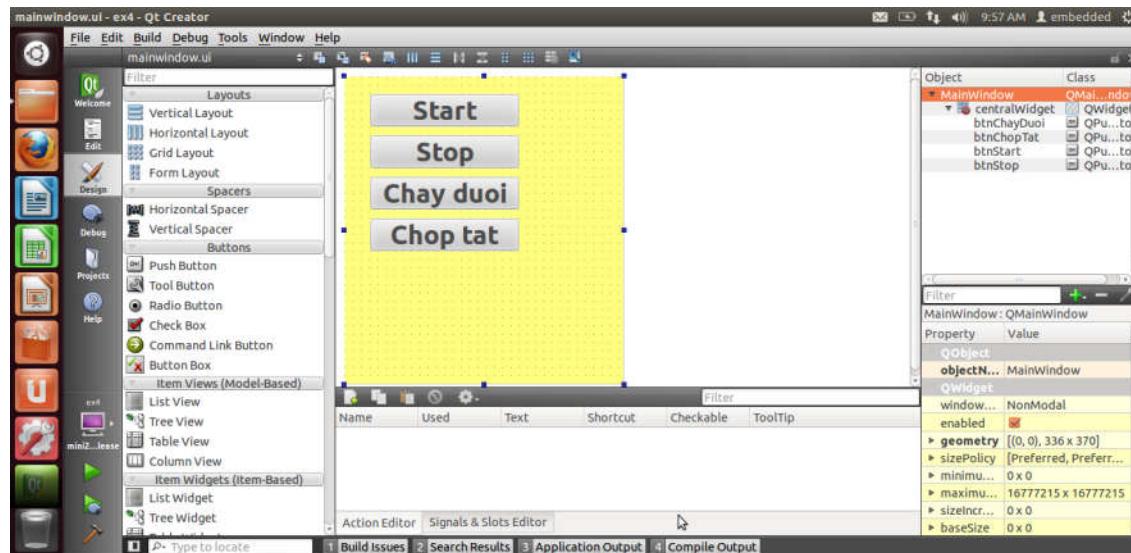
Mỗi khi chạy ứng dụng ta chỉ cần sửa lại tên file và đường dẫn. Kết quả chạy trên kit mini2440:



9.5 Viết 1 ứng dụng QT điều khiển led đơn chạy trên kit:

Thiết kế giao diện:

Giao diện gồm 4 nút nhấn có các chức năng: start,stop,chayduoi,choptat.



Code:

```
/*
```

Decrision: Create a QT interface to control leds

Board: Mini2440

Author: Phan Tuan Anh

```
*/
```

```
#include "mainwindow.h"

#include "ui_mainwindow.h"

#include <fcntl.h>

#include <stdio.h>

#include <stdlib.h>

#include <sys/ioctl.h>

#include <QTimer>
```

```
QTimer *timer1=new QTimer();
```

```
MainWindow::MainWindow(QWidget *parent) :
```

```
    QMainWindow(parent),
```

```
    ui(new Ui::MainWindow)
```

```
{
```

```
    ui->setupUi(this);
```

```
    initial_setup();
```

```
    connect(timer1,SIGNAL(timeout()),this,SLOT(xulyled()));
```

```
}
```

```
MainWindow::~MainWindow()
```

```
{
```

```
    delete ui;
```

```
}
```

```
void MainWindow::initial_setup()
```

```
{
```

```
    MainWindow::ui->btnChayDuo->setStyleSheet("font:bold 26px");
```

```
    MainWindow::ui->btnChopTat->setStyleSheet("font:bold 26px");
```

```
    MainWindow::ui->btnStart->setStyleSheet("font:bold 26px");
```

```
    MainWindow::ui->btnStop->setStyleSheet("font:bold 26px");
```

```
}
```

```
void MainWindow::on_btnStart_clicked()
```

```
{  
    timer1->stop();  
    system("/etc/rc.d/init.d/leds start");  
}  
  
void MainWindow::on_btnStop_clicked()  
{  
    timer1->stop();  
    system("/etc/rc.d/init.d/leds stop");  
}  
  
void MainWindow::display_led(int a)  
{  
    int led3,led2,led1,led0;  
    led3=a/8;  
    led2=(a/4)%2;  
    led1=(a/2)%2;  
    led0=a%2;  
    ioctl(fd,led0,0);  
    ioctl(fd,led1,1);  
    ioctl(fd,led2,2);  
    ioctl(fd,led3,3);  
}  
  
void MainWindow::xulyled()
```

```
{  
    if(status==0)  
    {  
        if(num==0)  
            num=15;  
        else  
            num=0;  
    }  
    else  
    {  
        if(num==0 || num==0b1000)  
            num=1;  
        else  
            num=num<<1;  
    }  
    display_led(num);  
}  
  
void MainWindow::on_btnChayDuoI_clicked()  
{  
    system("/etc/rc.d/init.d/leds stop");  
    if((fd=open("/dev/leds", 0))<0){  
        ::close(fd);  
        return;  
    }
```

```
num=0;  
status=1;  
timer1->start(300);  
}  
  
void MainWindow::on_btnChopTat_clicked()  
{  
    system("/etc/rc.d/init.d/leds stop");  
    if(fd=open("/dev/leds", 0))<0){  
        ::close(fd);  
        return;  
    }  
    num=0;  
    status=0;  
    timer1->start(300);  
}
```

Kết quả chạy trên kit.



9.6 Bài tập thực hành:

Bài 1: Viết 1 giao diện điều khiển **led** giống như ứng dụng led trong hệ điều hành linux nhúng.

Bài 2: Viết 1 giao diện điều khiển **button** giống như ứng dụng button trong hệ điều hành linux nhúng.

Bài 3: Viết 1 giao diện điều khiển **pwm** giống như ứng dụng button trong hệ điều hành linux nhúng.

Bài 4: Viết 1 giao diện điều khiển **adc** giống như ứng dụng button trong hệ điều hành linux nhúng.

Tài liệu tham khảo:

- [1]. Lập trình Linux-Tập 1. Tác giả: Nguyễn Phương Lan- Hoàng Đức Hải
- [2]. Lập trình hệ nhúng. Tác giả: Phạm Ngọc Hưng- ĐHBK Hà Nội
- [3]. Friendly ARM English User Manual.
- [4]. First Steps with Embedded Systems. Tác giả: Byte Craft Limited
- [5]. Linux Device Driver. Tác giả: Jonathan Corbet, Alessandro Rubini, Greg Kroal Hartman.

<https://sites.google.com/site/embedded247/>

<http://hethongnhung.com/>