HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA KỸ THUẬT ĐIỆN TỬ 1

BÁO CÁO

MÔN HỌC: THỰC TẬP CHUYÊN SÂU

Đề tài: Xây dựng kernel module TFT cho Raspberry Pi chạy nền Linux no-graphic

Sinh viên thực hiện : Nguyễn Tiến Duy – B20DCDT037

Lớp học : D20DTMT1

Lớp thực tập : 03

Giảng viên hướng dẫn : Đinh Quang Ngọc

HÀ NỘI – 2023

Nhận xét của giảng viên

Lời nói đầu

Hệ thống nhúng đang ngày càng phát triển, điều kiện về tài nguyên phần cứng ngày càng mở rộng đặc biệt là sự mạnh mẽ của các máy tính nhúng như raspberry pi, beagle-bone,… Máy tính nhúng chạy nền hệ điều hành linux no-graphic tuy nhanh, nhẹ nhưng việc tương tác gặp khó khăn khi không có nhiều phương tiện để thể hiện thông tin. Một module màn hình cho máy tính nhúng sẽ giải quyết vấn đề này, cung cấp một vài giao diện đồ họa giúp cho quá trình thể hiện thông tin trở nên dễ dàng hơn.

Đề tài nghiên cứu của em là ***Xây dựng kernel module TFT cho Raspberry Pi3 chạy nền linux no-graphic.*** Phát triển các module driver cho nhân linux bao gồm nhiều kiến thức liên quan từ quản lý hệ điều hành cho đến lập trình điều khiển phần cứng, vì vậy mức độ khó của đề tài là ở mức cao, sinh viên thực hiện nghiên cứu từ đó có thể nâng cao khả năng tự tìm hiểu giải quyết vấn đề khó.

Mục tiêu của đề tài là phát triển một kernel module màn hình TFT như một phương tiện hỗ trợ tương tác, hiển thị các thông tin, giao diện theo ý người dùng, thể hiện một cách trực quan các ứng dụng khác nhau như debug thông tin, giao diện điều khiển, giải trí…

Sử dụng module màn hình phổ biến là TFT ILI9341 để thực hiện mục tiêu đề tài với những ưu điểm như kích cỡ màn hình phù hợp, chất lượng hiển thị màu tốt, không quá khó để lập trình điều khiển. Lựa chọn Raspberry Pi 3 chạy hệ điều hành Ubuntu làm môi trường để phát triển kernel module bởi sự phổ biến của chúng cũng như cộng đồng phát triển lớn mạnh sẽ đẩy nhanh quá trình nghiên cứu. Nội dung nghiên cứu của đề tài bao gồm 3 chương chính:

Chương I: Giới thiệu về phát triển kernel module trên Linux – nội dung chương tập trung giải thích các khái niệm liên quan và quy trình xây dựng một kernel module.

Chương II: Xây dựng kernel module TFT cho Raspberry Pi 3 – trình bày các nội dung chính của đề tài, đưa ra các mô hình xây dựng và triển khai phát triển kernel module TFT kết hợp phần cứng của Raspberry Pi.

Chương III: Xây dựng ứng dụng đồ họa trên kernel module TFT – xây dựng và chạy chương trình đồ họa hỗ trợ bởi module TFT.

Mục lục

Danh mục hình ảnh

Chương I: Giới thiệu về phát triển kernel module trên Linux

1.1. Các khái niệm xung quanh kernel module

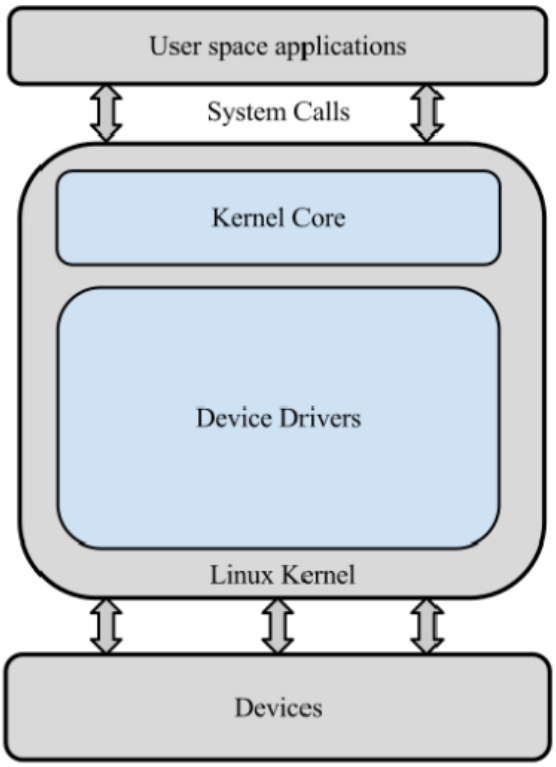
1.1.1. User mode, kernel mode, kernel module

User mode: Đây là chế độ dành cho ứng dụng non-kernel thực thi, tại đây tiến trình chỉ có thể thực hiện các hoạt động không yêu cầu đặc quyền. Để thực hiện được các hoạt động với đặc quyền cao hơn ứng dụng phải chuyển sang Kernel Mode.

Kernel mode: Trong chế độ này ứng dụng sẽ có quyền hạn cao nhất, có thể truy cập trực tiếp vào tài nguyên phần cứng và thực hiện các chức năng cốt lõi của hệ thống.

System call: Là công cụ để giúp một ứng dụng chuyển từ User mode sang Kernel mode, nơi có thể sử dụng các tài nguyên ở mức phần cứng.

→ Để quản lý và vận hành các tiến trình trong hệ thống, hệ điều hành Linux cơ bản sẽ dựa trên 3 thứ kể trên, mục tiêu là có thể đạt hiệu quả vận hành tốt nhất, tổ chức chương trình mạch lạc và bảo về tài nguyên.



Hình 1.1.1. Cấu trúc quản lý hệ thống của Linux

Kernel module: Linux được thiết kế cho phép người sử dụng nạp thêm một đoạn mã chương trình vào bên trong kernel và trở thành một phần của nó. Cơ chế này được gọi là kernel module.

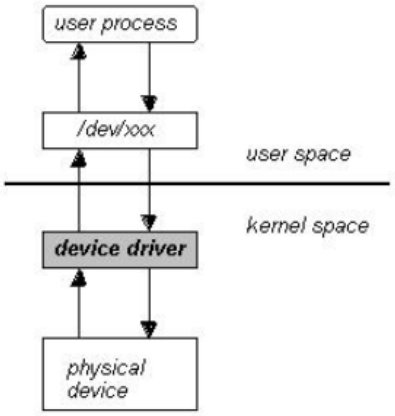
Có 2 phương pháp để thêm đoạn mã code vào Linux kernel:

- Thêm code vào kernel source và thực hiện biên dịch lại kernel
- Thêm code vào kernel source khi kernel đang chạy. Quá trình này được thực hiện bằng cách sử dụng các câu lệnh insmod và rmmod các file kernel module.

1.1.2. Character Device Driver

Device File:

Linux được thiết kế để chạy trong rất nhiều các nền tảng khác nhau và hỗ trợ nhiều loại device: TV, ô tô, smarthome, PC… Để quản lý chúng hệ điều hành cung cấp một loại file đặc biệt là Device File tương ứng với mỗi phần cứng được sử dụng trên hệ thống (/dev).



Hình 1.1.2. Vai trò của device file trong hệ thống

Các chương trình trên tầng user có thể truy cập vào thiết bị tương ứng với các file nằm trong thư mục /dev và thực hiện các system call thích hợp trên thiết bị, các lệnh này sau đó sẽ được chuyển tới driver đã được liên kết với device file của thiết bị đó.

Có 3 loại device file chính để Linux quản lý các thiết bị: Character device, Block device và Network device, trong đó Character device được sử dụng phổ biến nhất.

Character Device:

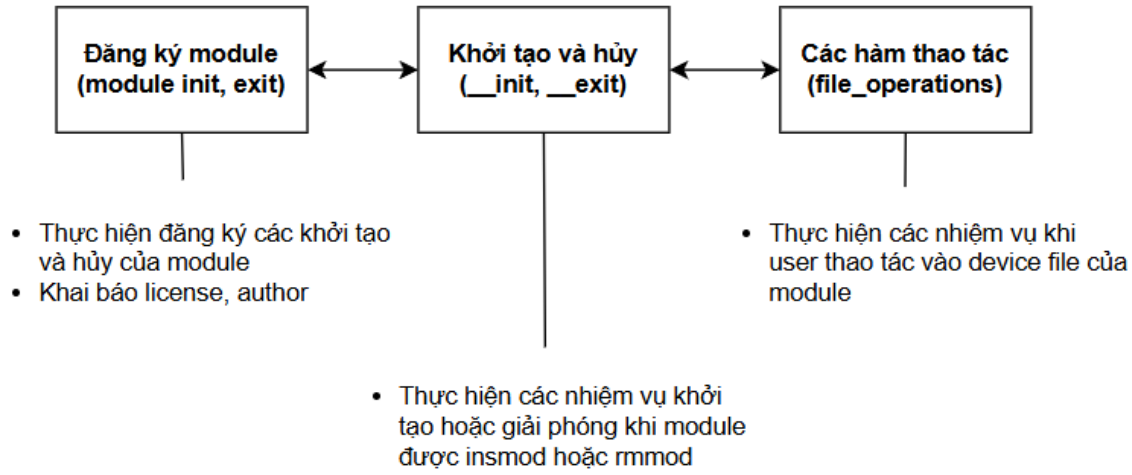
Các character device được mô tả trong kernel thông qua một struct data "cdev", là một phần của API để đăng ký và quản lý thiết bị trong kernel. Struct này được khai báo trong include/linux/cdev.h

```
struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        dev_t dev;
        [...]
};
```

const struct file_operations *ops: Con trỏ trỏ đến một cấu trúc file_operations chứa các con trỏ hàm thực hiện các thao tác trên thiết bị như open, read, write,… Đây là thành phần cấu trúc quan trọng quyết định nhiệm vụ và mục đích của kernel module sở hữu nó.

1.1.3. Xây dựng kernel module đơn giản

Cấu trúc mã nguồn của một kernel module:



Hình 1.1.3. Cấu trúc chung của một kernel module

Mỗi kernel module đều bao gồm 3 thành phần chính như hình vẽ, các hàm chức năng được triển khai và đăng ký tương ứng sẽ quyết định nhiệm vụ và mục đích của kernel module đấy.

Triển khai kernel module điều khiển LED:

Để hiểu rõ hơn về quy trình triển khai một kernel module cách tốt nhất là thực hiện xây dựng và chạy thử một kernel module đơn giản, ví dụ như điều khiển GPIO bật tắt đèn LED.


```
static const struct proc_ops tie_proc_fops =
{
        .proc_read = tie_read,
        .proc_write = tie_write,
};

static int __init gpio_driver_init(void)
{
        printk("Hi my friend, I'm just GPIO\n");

        gpio_register = (int*)ioremap(HW_GPIO_ADDR, PAGE_SIZE);
        if (gpio_register == NULL)
        {
                printk("Failed to map GPIO memory\n");
                return -1;
        }
        printk("Successfully mapped in GPIO memory\n");

        tie_proc = proc_create("tie-gpio-1", 0666, NULL, &tie_proc_fops);
        return 0;
}

static void __exit gpio_driver_exit(void)
{
        printk("Bye my friend, nomore GPIO\n");
        iounmap(gpio_register);
        proc_remove(tie_proc);
        return;
}

module_init(gpio_driver_init);
module_exit(gpio_driver_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("TIE");
MODULE_DESCRIPTION("HELLO KERNEL MODULE");
MODULE_VERSION("1.0");
```

Hình 1.1.3.a. Kernel module điều khiển LED (1)

Linux header cung cấp các thư viện chuẩn giúp ta triển khai mã nguồn cho driver, theo cấu trúc chung của một kernel module thành phần không thể thiếu đó là đăng ký hàm khởi tạo và giải phóng, thực hiện đăng ký qua module_init() và module_exit().

Các hàm chứa macroc __init, __exit là các hàm ta thực sự triển khai các nhiệm vụ khởi tạo và giải phóng cho driver này. Bao gồm việc đăng ký các vùng nhớ GPIO và tạo process thao tác file. Các biến chức năng như gpio_register, tie_proc được khai báo toàn cục.


```c
static struct proc_dir_entry *tie_proc = NULL;

#define MAX_SIZE 1024
static char data[MAX_SIZE];

#define HW_GPIO_ADDR    0x3f200000
static unsigned int *gpio_register = NULL;

static void tie_gpio_pin_on(unsigned int pin)
{
        unsigned int fsel_index = pin/10;
        unsigned int fsel_bitpos = pin%10;
        unsigned int *gpio_fsel = gpio_register + fsel_index;
        unsigned int *gpio_on_register = (unsigned int*)((char*)(gpio_register) + 0x1c);

        *gpio_fsel &= ~(7 << (fsel_bitpos*3));
        *gpio_fsel |= (1 << (fsel_bitpos*3));
        *gpio_on_register |= (1 << pin);
}
static void tie_gpio_pin_off(unsigned int pin)
{
        unsigned int *gpio_off_register = (unsigned int*)((char*)(gpio_register) + 0x28);
        *gpio_off_register |= (1 << pin);
}

ssize_t tie_read(struct file *file, char __user *user, size_t size, loff_t *off)
{
        return copy_to_user(user, "Hello!\n", 7) ? 0 : 7;
}

ssize_t tie_write(struct file *file, const char __user *user, size_t size, loff_t *off)
{
        unsigned int pin = UINT_MAX;
        unsigned int value = UINT_MAX;

        memset(data, 0x00, sizeof(data));
        if (size > MAX_SIZE) size = MAX_SIZE;

        if (copy_from_user(data, user, size)) return 0;

        printk("Data catch: %s\n", data);

        if (sscanf(data, "%d,%d", &pin, &value) != 2)
        {
                printk("Invalid format data!\n");
                return size;
        }

        if (pin > 22 || pin < 0)
        {
                printk("Invalid pin number!\n");
                return size;
        }
```

Hình 1.1.3.b. Kernel module điều khiển LED (2)

Các hàm *tie_read()* hoặc *tie_write()* sẽ được gọi tương ứng khi file đại diện của driver được user đọc hoặc ghi. Tùy thuộc vào dữ liệu người dùng gửi xuống, các hàm này sẽ thực hiện nhiệm vụ tương ứng đúng như mục đích của driver. Ở đây driver sẽ thực hiện điều khiển LED vì vậy ta cần các hàm thao tác GPIO của phần cứng.

Để thao tác vào thanh ghi của phần cứng hệ thống để điều khiển GPIO, việc cần làm đầu tiên đó là mapping vùng nhớ ảo và vùng nhớ vật lý. Như trong hàm __init module, việc mapping được thực hiện tại đây thông qua hàm *ioremap()*, một hàm do linux header cung cấp. Sau đó các địa chỉ này có thể thao tác tương ứng với địa chỉ vật lý, việc điều khiển các giá trị đầu ra GPIO có thể thực hiện theo như mô tả tài liệu của phần cứng hệ thống.

(Link mô tả phần cứng của Raspberry Pi 3: BCM2835-ARM-Peripherals.pdf (raspberrypi.org))

Trong hàm xử lý dữ liệu người dùng, sau khi copy dữ liệu được user gửi xuống thông qua hàm *copy_from_user()* , lúc này tùy thuộc vào format dữ liệu bao gồm chân điều khiển và giá trị đầu ra (ví dụ "21,0"), hàm thực hiện điều khiển GPIO on-off tương ứng tại chân đấy được thực thi.

```
if (pin > 22 || pin < 0)
{
        printk("Invalid pin number!\n");
        return size;
}

if (value != 0 && value != 1)
{
        printk("Invalid value number!\n");
        return size;
}

printk("So you mean: pin %d with value %d\n", pin, value);
if (value == 1)
{
        tie_gpio_pin_on(pin);
}
else if (value == 0)
{
        tie_gpio_pin_off(pin);
}
```

Hình 1.1.3.c. Kernel module điều khiển LED (3)

Hoàn thành xây dựng mã nguồn, việc tiếp theo đó là build, cài đặt và sử dụng module. Các công cụ hỗ trợ việc build bao gồm:

- *Trình biên dịch (gcc): sudo apt install gcc*
- *Thư viện linux headers: sudo apt install linux-headers-${uname -r}*
- *Makefile: sudo apt install make*

Với công cụ make việc build sẽ được thực hiện nhanh chóng và dễ dàng hơn. Mỗi một hệ thống phần cứng và phiên bản hệ điều hành sẽ được linux cung cấp thư viện headers riêng để GCC có thể build file .ko (kernel module) chính xác cho hệ thống đó.

```
KDIR = /lib/modules/$(shell uname -r)/build

all:
        make -C $(KDIR) M=$(shell pwd) modules

clean:
        make -C $(KDIR) M=$(shell pwd) clean
~
~
```

Hình 1.1.3.d. Cấu trúc Makefile thực hiện build .ko

Thực hiện build tại chính folder chứa mã nguồn kernel module qua câu lệnh make. Kết quả cho ra một tập các file, trong đó file .ko là file kernel module của driver. Insmod module và sẵn sàng sử dụng driver thông qua một chương trình ứng dụng nhấp nháy LED như sau.

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

int main()
{
        int f = open("/proc/tie-gpio-1", O_RDWR);

        while(1)
        {
                write(f, "21,1", 4);
                usleep(1000 * 200);
                write(f, "21,0", 4);
                usleep(1000 * 200);
        }
}
```

Hình 1.1.3.e. Chương trình ứng dụng nhấp nháy LED

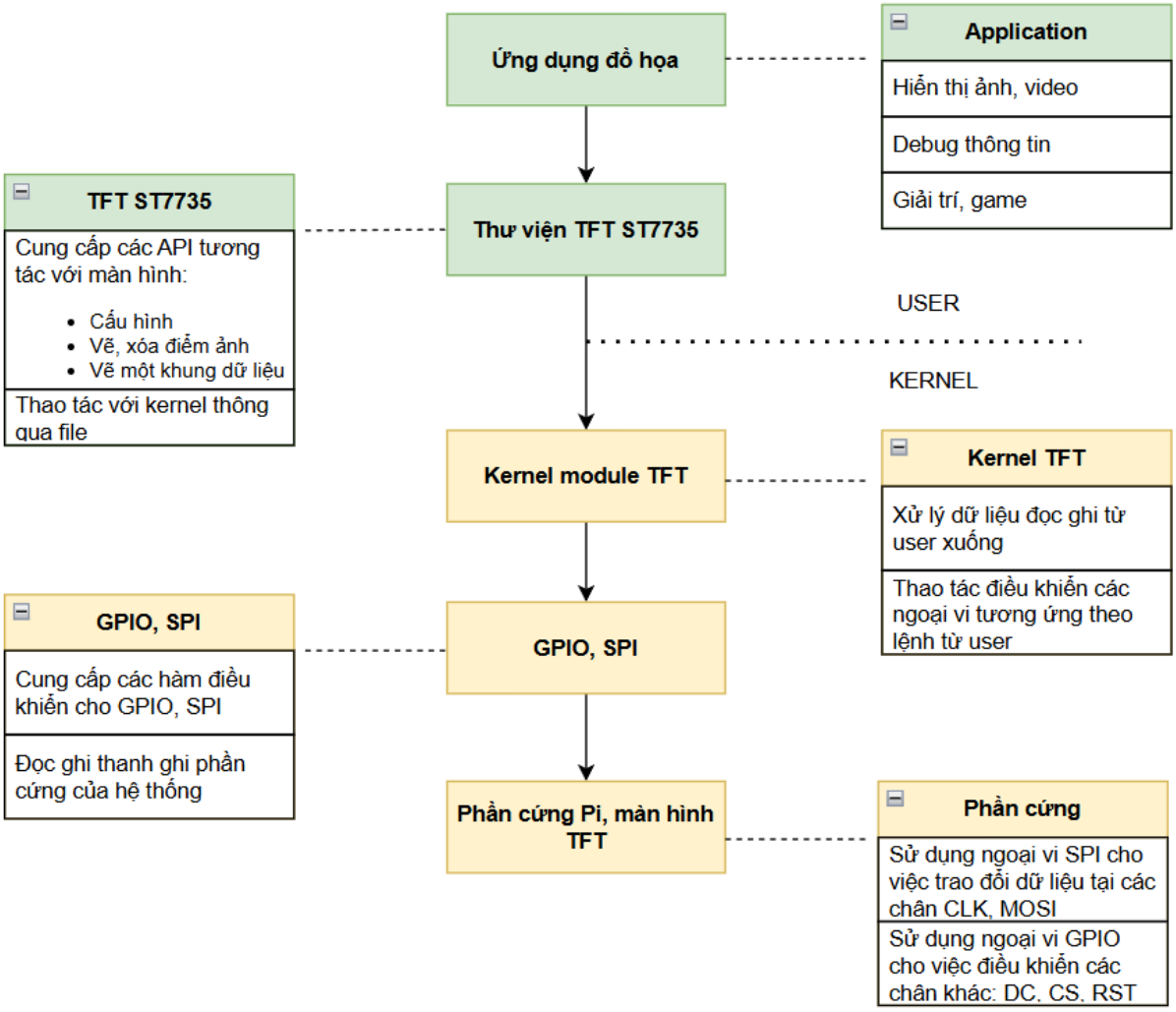
Thực hiện mở file đại diện của driver "/proc/tie-gpio-1", gửi dữ liệu xuống driver theo form để thực hiện tắt bật chân 21 sau mỗi 200ms. Sử dụng dmesg để theo dõi các hoạt động của module.

```
duyrasp@ubuntu:~/module/gpio$ sudo insmod tie_driver_gpio.ko
duyrasp@ubuntu:~/module/gpio$ sudo dmesg
[ 3402.274568] Hi my friend, I'm just GPIO
[ 3402.274605] Successfully mapped in GPIO memory
duyrasp@ubuntu:~/module/gpio$ ./a
^C
duyrasp@ubuntu:~/module/gpio$ sudo dmesg
[ 3402.274568] Hi my friend, I'm just GPIO
[ 3402.274605] Successfully mapped in GPIO memory
[ 3420.185814] Data catch: 21,1
[ 3420.185850] So you mean: pin 21 with value 1
[ 3420.686072] Data catch: 21,0
[ 3420.686108] So you mean: pin 21 with value 0
[ 3421.186351] Data catch: 21,1
[ 3421.186383] So you mean: pin 21 with value 1
duyrasp@ubuntu:~/module/gpio$ sudo rmmod tie_driver_gpio
duyrasp@ubuntu:~/module/gpio$ sudo dmesg
[ 3402.274568] Hi my friend, I'm just GPIO
[ 3402.274605] Successfully mapped in GPIO memory
[ 3420.185814] Data catch: 21,1
[ 3420.185850] So you mean: pin 21 with value 1
[ 3420.686072] Data catch: 21,0
[ 3420.686108] So you mean: pin 21 with value 0
[ 3421.186351] Data catch: 21,1
[ 3421.186383] So you mean: pin 21 with value 1
[ 3453.090832] Bye my friend, nomore GPIO
duyrasp@ubuntu:~/module/gpio$
```

Hình 1.1.3.f. Theo dõi hoạt động của module

Chương II: Xây dựng kernel module TFT cho Raspberry Pi 3

2.1. Mô hình xây dựng



Hình 2.1. Mô hình xây dựng theo nhiệm vụ từng khối

Quá trình xây dựng chia làm 2 phần chính: ***Xây dựng module điều khiển dưới tầng kernel*** và ***Xây dựng ứng dụng trên tầng application.*** Chia nhỏ bài toán theo từng khối đi cùng với nhiệm vụ của nó sẽ giúp giải quyết bài toán dễ dàng hơn, xác định được các công việc cần làm và vấn đề cần giải quyết theo từng bước một.

Phần cốt lõi của hệ thống đó là xây dựng được module điều khiển dưới tầng kernel và cũng là phần cần giải quyết nhiều vấn đề nhất của bài toán. Phần còn lại sẽ quyết định ứng dụng của kernel module vào từng mục đích cụ thể nhưng cơ bản cũng dựa trên các xử lý đồ họa, xử lý ảnh.

2.2. Xây dựng module điều khiển TFT tầng kernel

2.2.1. Kernel module TFT

Mọi kernel module đều có khung cấu trúc chung, bao gồm hàm cấu hình, giải phóng và các hàm điều khiển. Triển khai xây dựng khung cấu trúc mã nguồn tương tự như module điều khiển LED đã xây dựng trong chương I.

Cấu hình module:

```
static int my_driver_init(void){
    //Đăng ký character devive
    if(alloc_chrdev_region(&my_device_nb, 0, 1, DRIVER_NAME)){
        printk("Device could not be allocated \n");
        return -1;
    }
    long major = (my_device_nb >> 20);
    long minor = (my_device_nb & 0xfffff);
    if((my_class = class_create(THIS_MODULE,DRIVER_CLASS)) == NULL){
        printk("device class can not be create\n");
        unregister_chrdev_region(my_device_nb,1);
        return -2;
    }
    if(device_create(my_class,NULL,my_device_nb,NULL,DRIVER_NAME) == NULL){
        printk("can not create device file\n");
        unregister_chrdev_region(my_device_nb,1);
        class_destroy(my_class);
        return -3;
    }

    //Cấu hình character device sử dụng fops của driver
    cdev_init(&my_device,&fops);
    if(cdev_add(&my_device,my_device_nb,1) == 1){
        printk("registering device fail\n");
        device_destroy(my_class,my_device_nb);
    }

    //Mapping vùng nhớ các macro
    //Cấu hình ngoại vi

    return 0;
}
```

Hình 2.2.1a. Hàm cấu hình của module

Đăng ký character device: tạo device file trong hệ thống bao gồm các thông tin về class của module và cặp số major-minor thông qua các hàm tạo và cấp phát như hình trên.

Cấu hình character device: tham chiếu file_operation điều khiển và cặp số cho cdev. Sau bước này trong hệ thống sẽ tạo một device file đại diện cho kernel module TFT.

Mapping vùng nhớ, cấu hình ngoại vi: Khởi tạo các vùng nhớ và cài đặt ngoại vi để tiến hành điều khiển.

Giải phóng module:

```
static void my_driver_exit(void){
    //Giải phóng character device
    cdev_del(&my_device);
    device_destroy(my_class,my_device_nb);
    class_destroy(my_class);
    unregister_chrdev_region(my_device_nb,1);

    //Unmap các vùng nhớ vật lý
    iounmap(gpio_base_register);
    iounmap(spi_base_register);
}

//Đăng ký các hàm cấu hình và thoát cho module kernel
module_init(my_driver_init);
module_exit(my_driver_exit);
```

Hình 2.2.1b. Hàm giải phóng của module

Thực hiện ngược lại các công việc hàm cấu hình nhằm giải phóng các vùng nhớ, các file đã sử dụng cho việc cấu hình trước đó, nhường lại các vùng nhớ cho các module khác.

File operations:

```
static int driver_open(struct inode *device_file, struct file *instance);
static int driver_close(struct inode *device_file, struct file *instance);

static ssize_t driver_read(struct file *File, char *user_buffer, size_t size, loff_t *off_s){
    //Thực hiện gửi dữ liệu lên người dùng
    ...
}

static ssize_t driver_write(struct file *File, const char *user_buffer, size_t size, loff_t *off_s){
    //Thực hiện đọc dữ liệu từ người dùng gửi đến
    //Tùy theo lệnh nhận được sẽ thực hiện các hành động tương ứng
    //như gửi byte qua spi, điều khiển gpio
    //Sử dụng thư viện ST7735 để điều khiển màn hình
    ...
}

//Đăng ký các hàm điều khiển
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = driver_open,
    .release = driver_close,
    .read = driver_read,
    .write = driver_write
};
```

Hình 2.2.1c. File operations và các hàm đọc ghi file

File operations *fops* là struct được cdev tham chiếu đến, bao gồm các biến thành viên như module sở hữu, các con trỏ hàm đóng mở, đọc ghi file.

Các con trỏ hàm khi được gán thì các đối số phải đảm bảo đúng nguyên mẫu hàm của con trỏ đó. Hàm ghi file đóng vai trò chủ chốt nhất khi nó được gọi mỗi khi user ghi dữ liệu vào device file của module, các dữ liệu này được copy vào space của module, đóng vai trò như các tập lệnh điều khiển mà user mong muốn module thực hiện.

2.2.2. Các hàm thao tác GPIO, SPI của BCM2835

Mục đích của các hàm này là thực hiện các thao tác điều khiển ngoại vi GPIO và SPI của phần cứng hệ thống, cụ thể là chip BCM2835 của Raspberry Pi3. Các hàm này chủ yếu sẽ được gọi khi cấu hình module và tại hàm đọc ghi của file operations, nhằm thực hiện các lệnh điều khiển từ user.

Phương pháp chung khi xây dựng các hàm thao tác GPIO và SPI bao gồm:

- *Điều khiển thông qua macro địa chỉ các thanh ghi của chip*
- *Xây dựng các hàm điều khiển cơ bản: lập, xóa, chọn chức năng, enable, transfer data*
- *Mapping giữa địa chỉ ảo của hệ điều hành và địa chỉ vật lý*

GPIO:

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W

Hình 2.2.2a. Địa chỉ các thanh ghi chức năng của GPIO

Trong tài liệu của chip, từ trang 90 mô tả các thông số địa chỉ và cách thao tác các bit để điều khiển các chức năng của GPIO. Các thanh ghi chức năng cần thiết bao gồm:

- Thanh ghi địa chỉ gốc
- Thanh ghi chọn chức năng cho GPIO
- Thanh ghi lập giá trị cho output
- Thanh ghi xóa giá trị cho output

```
void gpio_pin_set_func(uint8_t pin, uint32_t sel_mask, uint32_t gpio_sel_offset)
{
    uint32_t temp = readl(gpio_base_register + gpio_sel_offset);
    temp |= ((uint32_t)(sel_mask << (3*(pin%10))));
    writel(temp,gpio_base_register + gpio_sel_offset);
}

static void gpio_pin_set(uint8_t pin, uint32_t gpio_set_offset){
    uint32_t temp = ((uint32_t)(1UL << (pin%31)));
    writel(temp,gpio_base_register + gpio_set_offset);
}

static void gpio_pin_clear(uint32_t pin,uint32_t gpio_clr_offset)
{
    uint32_t temp = ((uint32_t)(1UL << (pin%31)));
    writel(temp,gpio_base_register + gpio_clr_offset);
}
```

Hình 2.2.2b. Xây dựng các hàm điều khiển cơ bản

- Hàm chọn chức năng: Chọn chế độ cho GPIO theo bảng các chức năng của nó, mô tả tại trang 102
- Hàm lập: Đưa giá trị đầu ra GPIO là mức 1
- Hàm xóa: Đưa giá trị đầu ra GPIO là mức 0
- Hàm ghi: Đưa giá trị đầu ra GPIO là mức 1 hoặc 0 theo mong muốn

Mapping các vùng nhớ thông qua các hàm chức năng của hệ điều hành, hàm *ioremap()*. Mapping cho địa chỉ gốc GPIO và SPI, đồng thời thực hiện giải phóng các vùng nhớ trong hàm giải phóng module thông qua hàm *iounmap()*.

```
gpio_base_register = (int*)ioremap(BCM_2835_GPIO_BASE_ADDRESS,PAGE_SIZE);
if(gpio_base_register == NULL)printk("failed to map gpio\n");

spi_base_register = (int*)ioremap(BCM_2835_SPI_BASE_ADDRESS,PAGE_SIZE);
if(spi_base_register == NULL)printk("failed to map spi\n");
```

```
iounmap(gpio_base_register);
iounmap(spi_base_register);
```

Hình 2.2.2c. Mapping địa chỉ gốc GPIO và SPI

SPI:

10.5 SPI Register Map

The BCM2835 devices has only one SPI interface of this type. It is referred to in all the documentation as SPI0. It has two additional mini SPI interfaces (SPI1 and SPI2). The specifiation of those can be found under *2.3 Universal SPI Master (2x)*.

The base address of this SPI0 interface is 0x7E204000.

SPI Address Map			
Address Offset	**Register Name**	**Description**	**Size**
0x0	CS	SPI Master Control and Status	32
0x4	FIFO	SPI Master TX and RX FIFOs	32
0x8	CLK	SPI Master Clock Divider	32

Hình 2.2.2c. Địa chỉ các thanh ghi chức năng của SPI

Trong tài liệu của chip, trang 152 mô tả các thông số địa chỉ và cách thao tác các bit để điều khiển các chức năng của SPI. Các thanh ghi chức năng bao gồm:

- Thanh ghi địa chỉ gốc
- Thanh ghi điều khiển trạng thái
- Thanh ghi chia tần số clock
- Thanh ghi dữ liệu FIFO

```c
static void spi_init(uint8_t cs_pin){
    // clear old select
    writel(0x00000000,gpio_base_register + GPIO_GPFSEL0_OFFSET); /// pin 7,8,9
    writel(0x00000000,gpio_base_register + GPIO_GPFSEL1_OFFSET); /// pin 10,11
    writel(0x00000000,gpio_base_register + GPIO_GPFSEL2_OFFSET); /// pin 23, 24, 25

    // change gpio select to AF 0
    gpio_pin_set_func(cs_pin,GPIO_FSELX_OUTPUT_MASK,GPIO_GPFSEL0_OFFSET);
    gpio_pin_set_func(SPI_MISO_PIN,GPIO_FSELX_AF0_MASK,GPIO_GPFSEL0_OFFSET);
    gpio_pin_set_func(SPI_MOSI_PIN,GPIO_FSELX_AF0_MASK,GPIO_GPFSEL1_OFFSET);
    gpio_pin_set_func(SPI_SCK_PIN,GPIO_FSELX_AF0_MASK,GPIO_GPFSEL1_OFFSET);

    // change gpio select for dc,rst, light pin to GPIO_OUT_PP
    gpio_pin_set_func(LCD_DC_PIN,GPIO_FSELX_OUTPUT_MASK,GPIO_GPFSEL2_OFFSET);
    gpio_pin_set_func(LCD_LIGHT_PIN,GPIO_FSELX_OUTPUT_MASK,GPIO_GPFSEL2_OFFSET);
    gpio_pin_set_func(LCD_RST_PIN,GPIO_FSELX_OUTPUT_MASK,GPIO_GPFSEL2_OFFSET);

    // config pupd for all pin
    gpio_pin_enable(0,cs_pin,GPIO_GPPUDCLK0_OFFSET);
    gpio_pin_enable(0,SPI_MISO_PIN,GPIO_GPPUDCLK0_OFFSET);
    gpio_pin_enable(0,SPI_MOSI_PIN,GPIO_GPPUDCLK0_OFFSET);
    gpio_pin_enable(0,SPI_SCK_PIN,GPIO_GPPUDCLK0_OFFSET);
    gpio_pin_enable(0,LCD_DC_PIN,GPIO_GPPUDCLK0_OFFSET);
    gpio_pin_enable(0,LCD_LIGHT_PIN,GPIO_GPPUDCLK0_OFFSET);
    gpio_pin_enable(0,LCD_RST_PIN,GPIO_GPPUDCLK0_OFFSET);

    /*-----------------------------------------------*/
    // config frequency div
    writel(SPI_CLK_DIV_4,spi_base_register + SPI_CLK_OFFSET);

    // config cs register
    uint32_t temp_cs = readl(spi_base_register + SPI_CS_OFFSET);

    temp_cs = SPI_CS_CPOL_LOW \
        | SPI_CS_CPHA_LOW \
        | SPI_CS_CLEAR_TX_FIFO \
        | SPI_CS_CLEAR_RX_FIFO;

    writel(temp_cs,spi_base_register + SPI_CS_OFFSET);

    gpio_reg0_write(LCD_RST_PIN,0); /// reset
    gpio_reg0_write(LCD_RST_PIN,1); /// keep rst at high

}
```

```c
static void spi_deinit(uint8_t cs_pin)
{
    writel(0x00000000,spi_base_register + SPI_CLK_OFFSET); /// reset cdiv register
    writel(0x00000000,spi_base_register + SPI_CS_OFFSET); /// reset spi cs register
    writel(0x00000000,gpio_base_register + GPIO_GPFSEL2_OFFSET); /// pin 23, 24, 25

    gpio_reg0_write(SPI_CS_PIN,0); /// reset
    gpio_reg0_write(SPI_MISO_PIN,0); /// reset
    gpio_reg0_write(SPI_MOSI_PIN,0); /// reset
    gpio_reg0_write(SPI_SCK_PIN,0); /// reset

    gpio_reg0_write(LCD_LIGHT_PIN,0);
    gpio_reg0_write(LCD_DC_PIN,0);

    gpio_reg0_write(LCD_RST_PIN,0); /// reset
    gpio_reg0_write(LCD_RST_PIN,1); /// keep rst at high

    writel(0x00000000,gpio_base_register + GPIO_GPFSEL0_OFFSET); /// pin 7,8,9
    writel(0x00000000,gpio_base_register + GPIO_GPFSEL1_OFFSET); /// pin 10,11
    writel(0x00000000,gpio_base_register + GPIO_GPFSEL2_OFFSET); /// pin 23, 24, 25
}
```

Hình 2.2.2d. Các hàm cấu hình và giải phóng ngoại vi SPI

Trong các hàm cấu hình và giải phóng SPI, sử dụng các hàm chức năng đã xây dựng cho GPIO để thực hiện cấu hình các chân chức năng cho SPI, ngoài ra cũng thiết lập các thông số khác cho SPI về mode hoạt động, tần số chia cho clock.

```
static inline uint8_t spi_byte_transfer(uint8_t data)
{
    while(!(readl(spi_base_register + SPI_CS_OFFSET) & SPI_CS_TXD_FLAG_MASK)){};
        writeb(data,spi_base_register + SPI_FIFO_OFFSET);
    while(!(readl(spi_base_register + SPI_CS_OFFSET) & SPI_CS_RXD_FLAG_MASK)){};
        uint8_t dump = readb(spi_base_register + SPI_FIFO_OFFSET);
    return dump;
}
```

Hình 2.2.2e. Hàm truyền một byte dữ liệu qua SPI

Hàm transfer gửi dữ liệu đến chân MOSI theo từng byte, đảm bảo các byte được truyền xong trước khi rời khỏi hàm.

```
static void spi_transfer(uint8_t cs_pin, uint8_t cmd, char *buffer, uint32_t size) {

    gpio_reg0_write(cs_pin,0); // enable slave
    uint8_t nop_data = 0;
    switch(cmd){
        case 0: //CMD LIGHT
        {
            gpio_reg0_write(LCD_DC_PIN,0);
            gpio_reg0_write(LCD_LIGHT_PIN,*(buffer));
            nop_data = 1;
        } break;

        case 1: //CMD RESET
        {
            gpio_reg0_write(LCD_DC_PIN,0);
            gpio_reg0_write(LCD_RST_PIN,0);
            delay_us(50*1000);
            gpio_reg0_write(LCD_RST_PIN,1);
            delay_us(50*1000);
            nop_data = 1;
        } break;

        case 2: //CMD COMMAND
        {
            gpio_reg0_write(LCD_DC_PIN,0); /// switch to control command by reset DC pin
        } break;

        case 3: //CMD DATA
        {
            gpio_reg0_write(LCD_DC_PIN,1); /// switch to data command by set DC pin
        } break;
    }

    if (nop_data) spi_byte_transfer(0x00);
    else
    {
        for (uint32_t i = 0; i < size; ++i )
        {
            spi_byte_transfer(*(buffer+i));
        }
    }
    gpio_reg0_write(cs_pin,1);
```

Hình 2.2.2f. Hàm truyền một dữ liệu theo lệnh

Hàm truyền dữ liệu theo lệnh sẽ phục vụ cho việc thực hiện các lệnh truyền dữ liệu mà user mong muốn thông qua hàm ghi file của driver. Bao gồm các lệnh như điều khiển đèn nền, reset màn hình, lệnh truyền dữ liệu, lệnh truyền lệnh điều khiển. User lúc này có thể dễ dàng giao tiếp với driver thông qua các lệnh dưới dạng chuỗi ký tự.

2.2.3. Phần cứng Pi, màn hình TFT

Tài liệu cung cấp các thông tin về cấu trúc và cách triển khai các địa chỉ ảo và địa chỉ vật lý trên chip BCM2835. Trong đó cần lưu ý về các triển khai địa chỉ vật lý của chip mô tả trong trang 6 của tài liệu, cụ thể:

- Địa chỉ ảo của ngoại vi bắt đầu từ 0x7E000000, đây là địa chỉ được sử dụng để mô tả địa chỉ các thành ghi trong tài liệu.
- Địa chỉ vật lý của ngoại vi bắt đầu từ 0x3F000000, đây là địa chỉ thật của các thành ghi và sử dụng chúng để triển khai các macro.

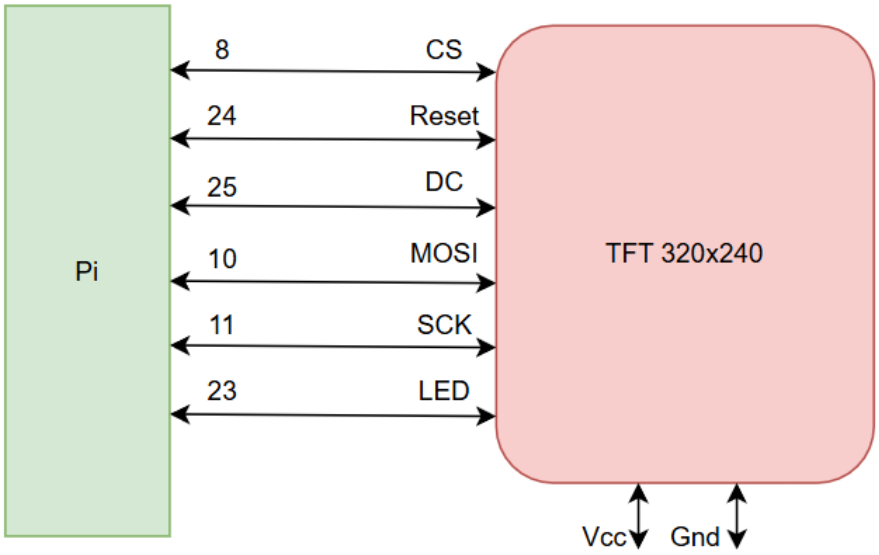
```
#define _BV(pos) (1UL << (pos))
/*GPIO Macro*/
#define BCM_2835_GPIO_BASE_ADDRESS ((uint32_t)0x3f200000)
// select function register
#define GPIO_GPFSEL0_OFFSET ((uint32_t)0x00)
#define GPIO_GPFSEL1_OFFSET ((uint32_t)0x04)
#define GPIO_GPFSEL2_OFFSET ((uint32_t)0x08)
// gpio set register
#define GPIO_GPSET0_OFFSET ((uint32_t)0x1C)
// gpio clear register
#define GPIO_GPCLR0_OFFSET ((uint32_t)0x28)
// gpio pull up/down
#define GPIO_GPPUD_OFFSET ((uint32_t)0x94)
#define GPIO_GPPUDCLK0_OFFSET ((uint32_t)0x98)
#define GPIO_GPPUDCLK1_OFFSET ((uint32_t)0x9C)
// GPIO Select function register
#define GPIO_FSELX_INPUT_MASK ((uint32_t)0x0)
#define GPIO_FSELX_OUTPUT_MASK ((uint32_t)0x1)
#define GPIO_FSELX_AF0_MASK ((uint32_t)0x4)
#define GPIO_FSELX_AF1_MASK ((uint32_t)0x5)
#define GPIO_FSELX_AF2_MASK ((uint32_t)0x6)
#define GPIO_FSELX_AF3_MASK ((uint32_t)0x7)
#define GPIO_FSELX_AF4_MASK ((uint32_t)0x3)
#define GPIO_FSELX_AF5_MASK ((uint32_t)0x2)
```

Hình 2.2.3a. Macro địa chỉ các thanh ghi chức năng GPIO

```
/*SPI Macro*/
#define BCM_2835_SPI_BASE_ADDRESS 0x3f204000

#define SPI_CS_OFFSET ((uint32_t)0x00)
#define SPI_FIFO_OFFSET ((uint32_t)0x04)
#define SPI_CLK_OFFSET ((uint32_t)0x08)
#define SPI_DLEN_OFFSET ((uint32_t)0x0C)
// CS
#define SPI_CS_CSPOL2_MASK _BV(23)
#define SPI_CS_CSPOL1_MASK _BV(22)
#define SPI_CS_CSPOL0_MASK _BV(21)

#define SPI_CS_RXF_FLAG_MASK _BV(20)
#define SPI_CS_RXR_FLAG_MASK _BV(19)
#define SPI_CS_TXD_FLAG_MASK _BV(18)
#define SPI_CS_RXD_FLAG_MASK _BV(17)
#define SPI_CS_DONE_FLAG_MASK _BV(16)

#define SPI_CS_CLEAR_NONE ((uint32_t)0x00000000)
#define SPI_CS_CLEAR_TX_FIFO ((uint32_t)0x00000010)
#define SPI_CS_CLEAR_RX_FIFO ((uint32_t)0x00000020)

#define SPI_CS_CPOL_HIGH _BV(3)
#define SPI_CS_CPOL_LOW  ((uint32_t)0x00000000)
#define SPI_CS_CPHA_HIGH _BV(2)
#define SPI_CS_CPHA_LOW  ((uint32_t)0x00000000)

#define SPI_CS_CS0_ ((uint32_t)0x00000000)
#define SPI_CS_CS1_ ((uint32_t)0x00000001)
#define SPI_CS_CS2_ ((uint32_t)0x00000002)

// clock divide (core freq = 250MHz)
#define SPI_CLK_DIV_1 _BV(0)
#define SPI_CLK_DIV_2 _BV(1)
#define SPI_CLK_DIV_4 _BV(2)
...
#define SPI_CLK_DIV_64 _BV(6)
#define SPI_CLK_DIV_128 _BV(7)
```

Hình 2.2.3a. Macro địa chỉ các thanh ghi chức năng SPI

Về kết nối phần cứng, sử dụng ngoại vi SPI và các GPIO khác theo các chân mô tả trong sơ đồ pinout của Pi 3, kết nối với các chân tương ứng với màn hình TFT, mô tả như sơ đồ bên dưới.



Hình 2.2.3b. Sơ đồ kết nối phần cứng

2.2.4. Build và cài đặt

Như vậy mã nguồn cho module dưới tầng kerel đến đấy đã hoàn thiện, việc còn lại là build mã nguồn và cài đặt vào hệ thống. Sử dụng Makefile link đến compiler và linux-headers đã cài đặt, tương tự như thực hiện với module điều khiển LED.

```
KDIR = /lib/modules/$(shell uname -r)/build
DRIVER_NAME = my_driver
DEVICE_FILE = /dev/my-spi-driver
###################################################################

all: clean build install chmod

###################################################################

build:
	make -C $(KDIR) M=$(shell pwd) modules CC=gcc CFLAGS="-std=gnu99"
clean:
	make -C $(KDIR) M=$(shell pwd) clean
```

Hình 2.2.4a. Cấu trúc Makefile thực hiện build kernel module (1)

Sau khi thực hiện build thành công, tiến hành insmod và cấp quyền thao tác device file.

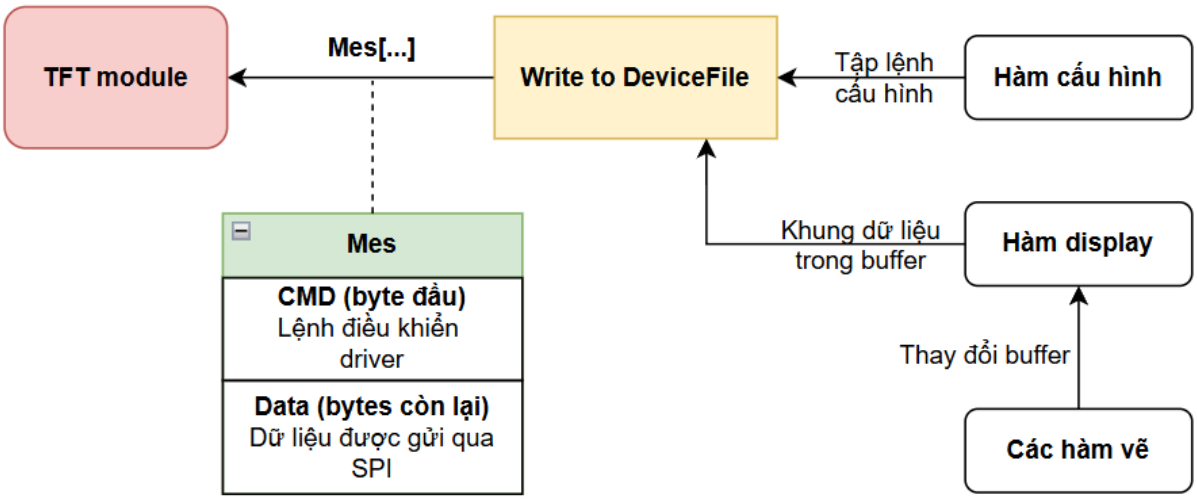
```
install: $(DRIVER_NAME).ko
    sudo insmod $(DRIVER_NAME).ko

remove: $(DRIVER_NAME).ko
    sudo rmmod $(DRIVER_NAME).ko

chmod:
    if lsmod | grep -q "$(DRIVER_NAME)"; then \
       echo "Driver is installed"; \
       sudo chmod 777 $(DEVICE_FILE); \
    else \
       echo "Driver is not installed"; \
    fi
```

Hình 2.2.4b. Cấu trúc Makefile thực hiện build kernel module (2)

Lúc này kernel module TFT sẵn sàng nhận các dữ liệu, lệnh điều khiển từ trên tầng user xuống mỗi khi có thao tác đóng mở, đọc ghi device file của module. Lưu ý tùy theo hệ điều hành chạy trên Pi đôi khi sẽ phải yêu cầu enable ngoại vi, cụ thể như Raspian OS cần enable SPI trong chế độ config nhưng đối với Ubuntu thì module có thể chạy ổn định ngay sau khi được insmod.

2.3. Xây dựng thư viện điều khiển TFT tầng application

Xây dựng thư viện cung cấp các hàm thực hiện các nhiệm vụ mô tả như trong mô hình xây dựng như cấu hình, vẽ điểm ảnh, vẽ khung dữ liệu. Để thực hiện được các nhiệm vụ ấy, thư viện cần có một hàm đóng vai trò cầu nối, hàm này sẽ thao tác với device file và ghi dữ liệu user mong muốn vào đó.



Hình 2.3. Mô hình điều khiển TFT trên tầng application

2.3.1. Hàm cầu nối với tầng kernel

Tùy theo CMD (byte đầu tiên trong mảng Mes) driver sẽ thực hiện các nhiệm vụ điều khiển tương ứng như đã mô tả trong phần xử lý dữ liệu dưới tầng kernel. Chuỗi data kèm theo (bytes còn lại) là chuỗi dữ liệu sẽ được gửi qua SPI đến màn hình.

```
static void write_to_device_file(uint8_t command, uint8_t data)
{
    int f= open(device_file_path, O_RDWR);
    char charint[2];
    charint[0] = command;
    charint[1] = data;
    write(f, charint, sizeof(charint));
    close(f);
}
static uint8_t arr[1+2*320*240];
static void write_arr_to_devfile(uint8_t* data, uint32_t size)
{
    int f= open(device_file_path, O_RDWR);
    arr[0] = CMD_DATA;
    memmove(arr+1, data, size);
    write(f, arr, size+1);
    close(f);
}
```

Hình 2.3.1. Hàm cầu nối - hàm thao tác với device file

write_to_device_file: Đầu vào gồm lệnh điều khiển (command) và 1 byte dữ liệu đi kèm. Hàm này được sử dụng khi user muốn điều khiển đèn nền, reset màn hình, gửi lệnh đến màn hình…đặc điểm chung là chỉ gửi ít hơn 1 byte qua SPI mỗi lần mở và ghi vào device file.

write_arr_to_devfile: Đầu vào gồm một mảng byte dữ liệu và kích thước của mảng đấy. Hàm này được sử dụng khi user muốn gửi một mảng dữ liệu đến màn hình chỉ trong một lần mở và đóng device file.

2.3.2. Các hàm chức năng của thư viện

```
//Hàm gửi các tập lệnh cấu hình
void tft_init();

//Hàm gửi mảng dữ liệu trong bảng vẽ (buffer)
void tft_display();
//Hàm reset dữ liệu trong bảng vẽ
void tft_clear(uint16_t color);

//Các hàm thao tác với bảng vẽ
uint16_t tft_get_point( int16_t x, int16_t y );
void tft_draw_point( int16_t x, int16_t y, uint16_t color );
void tft_draw_point_scale( int16_t x, int16_t y, uint16_t color,
                           uint8_t scale );

void tft_draw_number( int16_t x, int16_t y, int16_t num,
                      uint16_t color );
void tft_draw_string( int16_t x, int16_t y, char *str,
                      uint16_t color, const tFont *font );
```

Hình 2.3.2. Các hàm chức năng chính của thư viện điều khiển

Hàm gửi các tập lệnh cấu hình thiết lập các thông số cho màn hình như chọn mode hoạt động, tốc độ quét tối đa, chế độ màu,…Các lệnh cấu hình mô tả trong tài liệu phần cứng của màn hình TFT ST7735, sử dụng hàm cầu nối thực hiện gửi từng byte lệnh đó đến SPI.

```
void tft_write_reg(uint8_t reg) {
      write_to_device_file(CMD_CO
MMAND, reg);
}
void tft_write_data(uint8_t
data){
      write_to_device_file(CMD_DA
TA, data);
}
```

```
void tft_init() {
  write_to_device_file(CMD_RST,1);
  tft_write_reg(0x11);//Sleep out
    ...
  tft_write_reg(0xB1);//Framerate
  tft_write_data(0x05);
    ...
  tft_write_data(0x05);
  tft_write_reg(0x29);//DisplayOn
}
```

Hàm gửi mảng dữ liệu trong bảng vẽ (buffer) đóng vai trò như hàm hiển thị các điểm ảnh trên màn hình. Bảng vẽ ở đây là một mảng 1 chiều mô tả cho mọi điểm ảnh có trong khung hình, mỗi điểm ảnh là 2 bytes dữ liệu (16bits màu). Sử dụng hàm cầu nối gửi một mảng dữ liệu với kích thước là diện tích khung hình.

```
static uint8_t buff[2*320*240];

void tft_display()
{
    tft_set_cursor(0, 0);
    tft_write_reg(TFT_CMD_RAM_PREPARE);
    write_arr_to_devfile(buff, 2*320*240);
}
```

Các hàm chức năng còn lại cơ bản sẽ thay đổi dữ liệu trong bảng vẽ từ đó thay đổi dữ liệu sẽ hiển thị lên màn hình, các hàm này sẽ hỗ trợ cho các thư viện graphics có sẵn thực hiện vẽ các giao diện đồ họa, hình ảnh một cách dễ dàng.

```
void tft_clear(uint16_t color) {
    uint32_t i = 320 * 240;
    while (i--)
    {
        buff[2*i]   = (uint8_t)(color>>8);
        buff[2*i+1] = (uint8_t)(color);
    }
}
void tft_draw_point_raw(uint16_t x, uint16_t y, uint16_t color){
    tft_set_cursor(x, y);
    tft_write_reg(TFT_CMD_RAM_PREPARE);
    tft_write_data((uint8_t)(color >> 8));
    tft_write_data((uint8_t)color);
}
void tft_draw_point(int16_t x, int16_t y, uint16_t color){
    if ( x < 0 || y < 0 || x >= ST7735_TFTWIDTH
      || y >= ST7735_TFTHEIGHT ) return;

    uint32_t idx = (2*x) + (y*2*ST7735_TFTWIDTH);
    buff[idx] = (uint8_t)(color >> 8);
    buff[idx+1] = (uint8_t)(color);
}
```

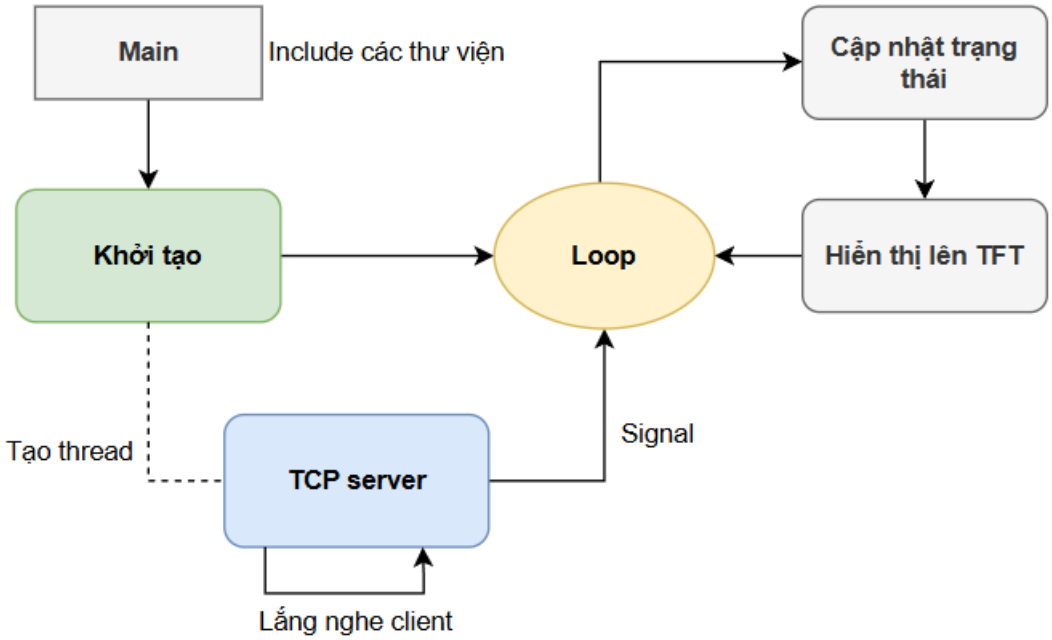
Từ đây chương trình ứng dụng có thể include thư viện và sử dụng các hàm chức năng đã được khai báo, ví dụ chương trình test như sau:

```
#include "tie_st7735.h"
int main() {
    tft_init();//cấu hình
    tft_clear(0xffff);//white
    tft_display();//hiển thị
    return 0;
}
```

Chương III: Xây dựng ứng dụng đồ họa trên kernel module TFT

3.1. Mô hình ứng dụng đơn giản sử dụng kernel module TFT

Mô hình chung của một ứng dụng là cập nhật các trạng thái, xử lý logic và hiển thị trạng thái hiện tại lên màn hình, thực hiện chúng trong một vòng lặp chính. Ngoài ra để điều khiển thay đổi trạng thái ứng dụng, sử dụng mô hình TCP server-client để trao đổi tín hiệu điều khiển từ xa, TCP server chạy trên một thread phụ trong chương trình chính.



Hình 3.1. Mô hình chung một ứng dụng tương tác từ xa

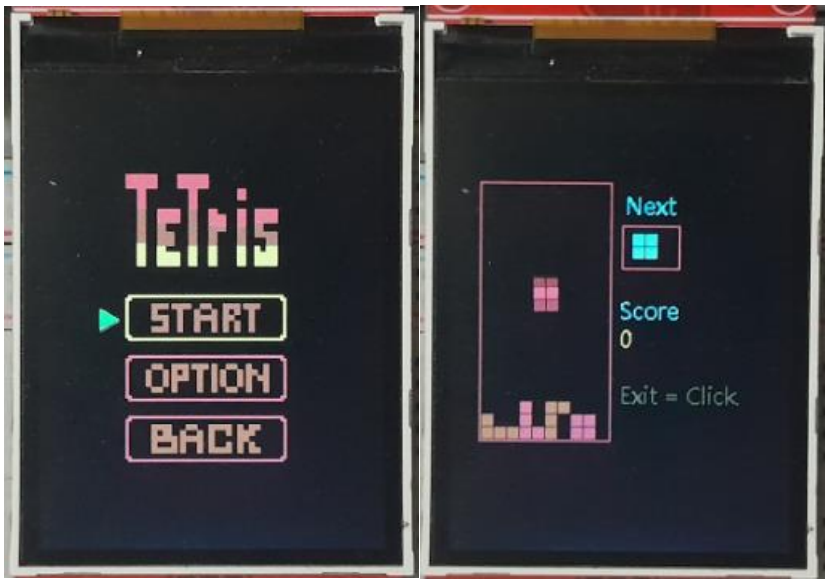
Chương trình chính (main) khai báo các thư viện cần thiết hỗ trợ cho việc triển khai các chức năng như multi-thread, timer, đồ họa, điều khiển. Khởi tạo các đối tượng liên quan như cấu hình module TFT, cấu hình TCP server, tạo thread, khởi tạo dữ liệu ứng dụng.

Thread chính (Loop) cung cấp các số liệu cập nhật về timer, cập nhật tín hiệu điều khiển cho trạng thái ứng dụng và xử lý đồ họa. Thread phụ lắng nghe các message từ client, từ đó xử lý đưa ra các signal điều khiển ứng dụng theo mong muốn của client. Bởi ứng dụng có xử lý nhiều luồng nên cần sử dụng các kỹ thuật xử lý đồng bộ và bất đồng bộ như mutex để đảm bảo các hoạt động của ứng dụng không bị xung đột với nhau.

3.2. Phát triển trò chơi đơn giản sử dụng kernel module TFT

Mô phỏng lại trò chơi xếp gạch – Tetris chạy trên board Pi 3 hiển thị lên màn hình TFT thông qua kernel module TFT, điều khiển trò chơi trên một thiết bị khác trong cùng một mạng local với Pi 3.

Trò chơi bao gồm một bảng các ô trống và các khối gạch, nhiệm vụ của người chơi đó là xếp các khối gạch rơi từ phía trên cùng của bảng sao cho lấp đầy mỗi hàng của bảng thì điểm số sẽ được cộng và hàng được lấp đầy đó cũng biến mất sẵn sàng cho các lượt xếp tiếp theo. Trò chơi kết thúc khi người chơi không thể tiếp tục xếp thêm bất kỳ hàng nào cho trong bảng.



Hình 3.2. Giao diện chính của tetris

Xây dựng trò chơi bao gồm 2 giao diện chính là giao diện menu và giao diện chơi. Trong 2 giao diện trên các đối tượng đồ họa bao gồm các ảnh (tiêu đề, nút start, option, back, mũi tên), các khối hình chữ nhật và chuỗi ký tự (score, next). Từ đây xây dựng thư viện cung cấp các hàm chức năng để thể hiện các đối tượng kể trên.

```
typedef struct
{
    uint16_t w, h;
    uint64_t len_of_frame;//amount element each of bmp
    uint16_t frame;//amount frame of animation
    t_bmp_data *anim;//duration and counting active of
each frame
    id_anim id;//enum id of animation
} t_anim;
```

Một đối tượng hoạt ảnh được thể hiện qua struct t_anim bao gồm các thông số như kích thước hoạt ảnh, số khung hình, dữ liệu, id. Đi kèm với đó là các hàm chức năng để thể hiện nó trong bảng vẽ màn hình.

```
typedef struct
{
    int16_t x;
    int16_t y;
} vec;
vec vec_inst( int16_t x, int16_t y );
void vec_copy( vec* own, vec other );
void vec_add( vec* own, vec* other );

typedef struct
{
    vec topleft;
    vec dimens;
    uint8_t scale;
    const t_anim* sAnim;
    uint32_t anim_cnt;
    uint32_t anim_dura;
    uint8_t anim_idx;
} entity;

void ent_init( entity* ent, const t_anim* sAnim );
void ent_scale( entity* ent, uint8_t scale );
void ent_update( entity* ent, uint32_t dtime );
void ent_draw( entity* ent );
void ent_draw_at( vec topleft, entity* ent );
void ent_set_idx( entity* ent, uint8_t idx );
```

Các đối tượng khác như vẽ hình chữ nhật, chuỗi ký tự, chuỗi số có thể sử dụng trực tiếp các hàm của thư viện TFT ST7735 để thể hiện. Ngoài ra trong thư viện utils trên cũng cung cấp các hàm khác như lấy số ngẫu nhiên, cập nhật thời gian, convert,…và các hàm xử lý signal điều khiển.

```
//Lấy số ngẫu nhiên dựa trên timer
uint16_t util_random( uint16_t begin, uint16_t end );
//Convert string về số nguyên lớn
bool str_to_u64(const char* str, uint64_t* num);
//Cập nhật thời gian
uint32_t tick_dtime();

typedef enum
{
    S_CTRL_UP, S_CTRL_DN, S_CTRL_RT, S_CTRL_LT,
    S_CTRL_ENT,
    S_CTRL_COUNT
} t_ctrl;
void signal_ctrl_set( t_ctrl ctrl );
bool signal_ctrl_get( t_ctrl ctrl );
void signal_ctrl_clear();
void signal_mtx_lock();
void signal_mtx_unlock();
```

Logic của trò chơi Tetris cơ bản đều xung quanh một bảng các ô màu, các ô màu mới sẽ xuất hiện từ trên xuống, các hàng ô nào được lấp đầy sẽ biến mất và ghi điểm. Đối tượng trò chơi ta sẽ quản lý bằng một ***board*** các ***cell***.

```
typedef struct
{
    uint8_t is_fill;
    uint16_t color;
} cell;

#define board_w     10
#define board_h     20
#define cell_d      8
#define board_pad   1
typedef struct
{
    vec pos;
    cell cells[board_h][board_w];
} board;
void board_init( vec pos );
void board_draw();
void board_clear();

void board_set_cell( vec loca, uint16_t color );
void board_clear_cell( vec loca );
bool board_cell_at( vec loca );
bool board_is_cover( vec loca );
uint16_t board_color_at( vec loca );
```

```
void board_draw()
{
    gfx__rect_thin(
        bd.pos, vec_inst(
            board_w*(cell_d+board_pad)+2*board_pad,
            board_h*(cell_d+board_pad)+2*board_pad
        ), COLOR_RED
    );
    vec cell_dimen = vec_inst(cell_d, cell_d);
    for ( int y = 0; y < board_h; ++y )
    {
        for ( int x = 0; x < board_w; ++x )
        {
            if ( bd.cells[y][x].is_fill )
            {
              vec loca = vec_inst(
                bd.pos.x + 2*board_pad + x*(cell_d + board_pad) ,
                bd.pos.y + 2*board_pad + y*(cell_d + board_pad)
              );
              gfx__rect_fill(
                loca, cell_dimen,
                bd.cells[y][x].color
              );
            }
        }
    }
}
```

```
void tet_init();
void tet_refresh();
void tet_spawn_block();
void tet_roll();
void tet_moving( vec dir );
void tet_update( uint16_t dtime );
void tet_draw();
bool tet_is_landing();
bool tet_is_topping();
bool tet_is_quit();
```

Logic trò chơi đã được triển khai trong một file tetris.c, trong đó có một hàm đóng vai trò chính trong việc cập nhật trạng thái trò chơi cũng như xử lý các signal điều khiển, đó là ***tet_update()***.

```
void tet_update( uint16_t dtime ) {
    if( tet.is_quit ) return;
    if( ProcessMenu( dtime ) != TET_MODE_GAME ) return;
    if ( signal_ctrl_get(S_CTRL_ENT) ) {
        tet.mode = TET_MODE_MENU;
        return;
    }

    vec dir = vec_inst(0, 0);
    tet.fall_count += dtime;
    tet.move_count += dtime;
    //Falling
    if( tet.fall_count >= tet.dura_rest ) {
        tet.fall_count = 0;
        dir.y = 1;
    }
    //Event moving
    if( tet.move_count >= tet.dura_rest ) {
        uint16_t last_count = tet.move_count;
        tet.move_count = 0;
        if      ( signal_ctrl_get(S_CTRL_LT) )  dir.x = -1;
        else if ( signal_ctrl_get(S_CTRL_RT) )  dir.x = 1;
        else    tet.move_count = last_count % 60000;
    }
    if ( signal_ctrl_get(S_CTRL_DN) )
        tet.fall_count += dtime * 10;
    if ( signal_ctrl_get(S_CTRL_LT) || signal_ctrl_get(S_CTRL_RT) )
        tet.move_count += dtime * 2;

    if( tet.is_topping ) Eff_NewGame( dtime );
    else {
        if( tet.is_landing ) {
            ProcessGoal();
            tet_spawn_block();
        }
        else if( signal_ctrl_get(S_CTRL_UP) ) {
            if ( tet.is_avail_roll ) {
                tet.is_avail_roll = 0;
                tet_roll();
            }
        }
        else tet.is_avail_roll = 1;

        tet_moving( dir );
    }
}
```

Trong chương trình chính tiến hành khai báo các thư viện, khởi tạo và cập nhật trạng thái các đối tượng mô tả như trong mô hình.

```
#include "tie_define.h"
#include "tie_st7735.h"
#include "utils.h"
#include "tetris.h"
#include "tcp_ser.h"

int main()
{
    //Khởi tạo màn hình
    tft_init();
    tft_clear(0xf1f1);

    //Tạo thread cho TCP server
    pthread_t t_tcp_ser;
    pthread_create(&t_tcp_ser, NULL, thread_tcp_ser, NULL);
    pthread_detach(t_tcp_ser);

    assets_load();  //Convert các file hình ảnh của trò chơi
    tet_init();     //Khởi tạo trò chơi

    uint32_t dtime = 0;
    while (1)
    {
        dtime = tick_dtime();   //Cập nhật thời gian
        tft_clear(COLOR_BLACK);

        //Cập nhật trạng thái trò chơi
        signal_mtx_lock();
        tet_update(dtime);
        signal_ctrl_clear();
        signal_mtx_unlock();

        //Hiển thị trạng thái lên TFT
        tet_draw();
        tft_display();

        //Tốc độ trò chơi
        usleep(1000*100);
    }
    return 0;
}
```

Chương trình đến đây hoàn thiện, sử dụng trình biên dịch gcc và makefile link các thư viện cần thiết để tiến hành build chương trình. Về phía client điều khiển, sử dụng python xây dựng chương trình kết hợp với các thư viện socket, thread, pygame để trao đổi message với Pi.

3.3. Kết quả

Về tổng quan, hệ thống hoạt động đúng như mục tiêu đặt ra, các chức năng hoạt động tốt và ổn định trong thời gian dài. Hệ thống tuy nhiều module kết hợp với nhau nhưng vẫn đảm bảo hoạt động đồng bộ và đúng chức năng nhiệm vụ.

Về kernel module TFT đảm bảo hiển thị các giao diện trực quan và ít bị nhiễu. Màn hình hiển thị được các màu sắc ở mức tốt, tốc độ quét màn hình không quá thấp với tốc độ tối đa đạt được là ~80ms mỗi frame kích thước 320x240 pixels.

Tuy nhiên nhược điểm của module TFT đó là khi hoạt động hết công suất (tốc độ quét cao nhất) chiếm khá nhiều thời gian sử dụng CPU. Điều này có thể giải thích được khi kernel module phải xử lý rất nhiều dữ liệu dưới tầng kernel, với mỗi điểm ảnh cần gửi 2 bytes dữ liệu trong khi phải gửi liên tục 320x240 điểm.

Một vài hình ảnh kết quả chạy chương trình trên Pi 3

```
[  199.101303] DEVICE DRIVER EXIT DONE
[  217.938710] Hello device driver
[  217.938744] my-driver, Major = 238, Minor = 0
[  217.939130] DEVICE DRIVER INIT DONE
duyrasp@server:~/module/spi/demo_tft$ sudo dmesg -c
duyrasp@server:~/module/spi/demo_tft$ ./app
[TCP_SET_TASK_CB] --- Register for [task-ctrl] task info
Socket created
Socket setup
Socket binding
[TCP] - 0.0.0.0:6543: Socket listening
```

Hình 3.3. Một vài kết quả chạy chương trình

Kết luận

Thông qua đề tài nghiên cứu này, ta đã hiểu quy trình và cách thức xây dựng một kernel module hỗ trợ màn hình TFT trên board Raspberry Pi chạy hệ điều hành Ubuntu cũng như là các hệ điều hành no-graphic khác. Màn hình TFT cung cấp khả năng hiển thị đồ họa và thông tin trong các ứng dụng nhúng, và việc tạo một kernel module cho phép chúng ta tương tác với màn hình này một cách trực tiếp từ hệ điều hành.

Trong quá trình nghiên cứu, ta đã tìm hiểu về cấu trúc và cách thức hoạt động của kernel module trên Linux và tìm hiểu về việc tạo một module mới, nắm được các khái niệm căn bản liên quan đến kernel module và giao tiếp với các thiết bị phần cứng thông qua các thanh ghi đặc biệt.

Sau đó, nội dung các chương đã tập trung vào việc xây dựng kernel module cho màn hình TFT trên board Raspberry Pi. Việc này đòi hỏi việc phải tìm hiểu cấu trúc và giao thức giao tiếp của màn hình TFT cụ thể được sử dụng. Chúng ta đã tìm hiểu cách gửi và nhận các tín hiệu điều khiển và dữ liệu để hiển thị thông tin lên màn hình.

Kết quả, đề tài đã thành công trong việc xây dựng một kernel module cho màn hình TFT và thực hiện chạy một chương trình trò chơi đơn giản trên board Raspberry Pi chạy hệ điều hành Ubuntu. Đề tài này mang lại những kiến thức quan trọng về việc phát triển kernel module trên Linux và cung cấp cách tiếp cận để tích hợp màn hình TFT vào các ứng dụng nhúng trên board Raspberry Pi. Đồng thời, nó mở ra cơ hội cho các nghiên cứu và ứng dụng tiếp theo trong lĩnh vực này.

Lời cảm ơn

Trong quá trình thực hiện đề tài nghiên cứu, em đã tích lũy được nhiều kiến thức quý báu và nâng cao kỹ năng chuyên môn của mình. Nhận thức rõ ràng về tầm quan trọng của việc đề cao tinh thần tự giác, tìm tòi và nghiên cứu độc lập, em tin rằng điều này không chỉ giúp em áp dụng lý thuyết vào thực tiễn mà còn là cơ hội để phát triển các kỹ năng mềm quan trọng và giải quyết vấn đề trong công việc.

Em muốn bày tỏ sự cảm kích tới các thầy cô trong lab nghiên cứu của khoa, những người đã hỗ trợ chúng em suốt quá trình thực tập một cách tận tình và chu đáo, sự hướng dẫn và góp ý của thầy cô đã giúp em hoàn thành đề tài nghiên cứu này một cách hiệu quả hơn.

Tuy nhiên, em nhận thức rằng bản báo cáo của mình vẫn còn nhiều hạn chế và thiếu sót. Do đó, em rất mong nhận được những ý kiến đóng góp xây dựng từ thầy cô để có thể hoàn thiện và nâng cao kiến thức của mình trong lĩnh vực này.

Em xin chân thành cảm ơn!

Tài liệu tham khảo

[1] Chip on board Pi3: BCM2835-ARM-Peripherals.pdf (raspberrypi.org)

[2] Low Level Learning YT channel: Raspberry Pi Kernel Development | Writing a Raspberry Pi ARM GPIO Driver in C | Embedded Concepts - YouTube

[3] Vinalinux Blog: Character device driver – Vinalinux

[4] ST7735 congif source code: pimoroni/st7735-python: Python library to control an ST7735 TFT LCD display. Allows simple drawing on the display without installing a kernel module. (github.com)