

**ĐẠI HỌC QUỐC GIA HÀ NỘI**  
**TRƯỜNG ĐẠI HỌC CÔNG NGHỆ**

-----\*\*\*-----



**BÁO CÁO QUÁ TRÌNH TÌM HIỂU**  
**VỀ DEVICE DRIVER VỚI CHIP ZYNQ**

**Giảng viên hướng dẫn:** TS. Nguyễn Kiêm Hùng

**Đơn vị thực tập:** Gnode 5G - VHT

**Sinh viên thực hiện:** Luyện Huy Tín

**Lớp:** K64K2

**Mã sinh viên:** 19020636

**Email:** 19020636@vnu.edu.vn

**Chuyên ngành:** Kỹ thuật máy tính

**Hà Nội – 2022**

## Mục lục

I. Tìm hiểu lý thuyết về Device Driver.....	5
1. Device Driver .....	5
1.1. User space và kernel space.....	5
1.2. Cấu trúc của Driver.....	6
2. Gắn kernel module.....	8
II. Lý thuyết về Character device .....	9
1. Character device .....	9
1.1. Sơ đồ khối.....	9
1.2. Cấu trúc của Character driver.....	11
2. Phân tích cụ thể.....	11
2.1. Device file .....	12
2.2. Device number.....	13
2.3. Cấp phát bộ nhớ và khởi tạo thiết bị.....	15
2.4. Các hàm Entry point .....	16
3. Giao tiếp với phần cứng.....	19
3.1. Truy cập phần cứng trên hệ điều hành Linux .....	19
II. Triển Khai .....	20
1. Trình bày vấn đề cần giải quyết .....	20
2. Ý tưởng thực hiện .....	20
2.1. Xây dựng Character device .....	21
2.2. Truy cập tới phần cứng của KIT ZCU102.....	25
2.3. Gắn devices lên kernel .....	28
3. Kết quả - nhận xét - đánh giá .....	29
3.1. Kết quả .....	29
3.2. Nhận xét - đánh giá.....	31
III. Platform Device Driver và Device Tree .....	31
1. Lý thuyết về Platform device driver .....	32
1.1. Platform driver.....	32

1.2. Platform device .....	33
1.3. Match driver và device.....	34
2. Lý thuyết về Device Tree .....	35
2.1. Cơ chế của Device tree .....	35
2.2. Mối quan hệ giữa Platform device driver và Device tree.....	36
IV. Phát triển driver điều khiển đèn led hỗ trợ Device Tree dựa trên Platform Driver .....	37
1. Trình bày vấn đề cần giải quyết .....	38
2. Ý tưởng thực hiện .....	38
2.1. Tạo và đăng kí driver và device .....	38
2.2. Xây dựng các trường trong driver .....	39
3. Kết quả - Đánh giá – Nhận xét.....	41
3.1. Kết quả .....	41
3.2. Đánh giá – Nhận xét .....	43

## Danh mục hình ảnh

Hình 1. Kernel space tương tác user space .....	5
Hình 2. Các file được tạo khi make .....	8
Hình 3. Sơ đồ hoạt động của Character device .....	9
Hình 4. System call tương tác với entry point .....	11
Hình 5. Tạo device file bằng cách thủ công .....	12
Hình 6. Kiểm tra device number của các devices .....	14
Hình 7. Các file có trong thư mục device tree .....	26
Hình 8. Xem địa chỉ của IP tại file pl.dtsi .....	26
Hình 9. Bảng địa chỉ offset của AXI GPIO .....	27
Hình 10. Các file sau khi thực hiện make .....	29
Hình 11. Điều khiển đèn led sáng .....	30
Hình 12. Điều khiển đèn led tối .....	30

# I. Tìm hiểu lý thuyết về Device Driver

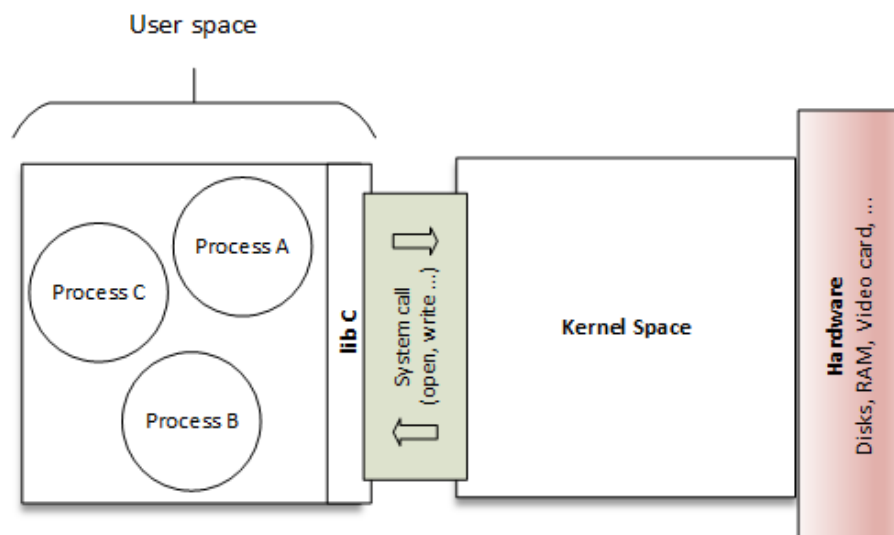
Chúng ta đều biết rằng khi sử dụng bất kì một thiết bị thông minh nào (máy tính, điện thoại, ...), ta đều dễ dàng nhìn thấy được những phần cứng và có thể sử dụng phần mềm trên những thiết bị đó. Tuy nhiên, để có thể kết nối hai thứ đó lại với nhau, ta cần thêm một thứ khác, phần đó được gọi là firmware, nó bao gồm hệ điều hành, các driver hoặc các apps... Cụ thể trong báo cáo này, ta sẽ tìm hiểu và bàn luận về Device driver.

## 1. Device Driver

Driver là một phần mềm, nó được tạo ra để điều khiển và quản lý thiết bị phần cứng, driver có thể nằm ở kernel space (chế độ toàn quyền) hoặc user space (chế độ không toàn quyền). Ở đây chúng ta sẽ quan tâm nhiều hơn driver nằm trên kernel space của hệ điều hành Linux. Vậy kernel space và user space là gì?

### 1.1. User space và kernel space

Đầu tiên, chúng ta cần hiểu rằng user space và kernel space là hai khái niệm hoàn toàn trừu tượng. Nó mô tả toàn bộ không gian bộ nhớ và quyền truy cập vào bộ nhớ. Có thể hiểu rằng kernel sẽ được sử dụng toàn quyền bộ nhớ trong khi user sẽ bị hạn chế truy cập. Đây là một đặc điểm của CPU hiện đại, cho phép nó hoạt động ở chế độ toàn quyền hoặc không toàn quyền.



Hình 1. Kernel space tương tác user space

Sơ đồ trên biểu thị sự phân chia giữa kernel space và user space. System call đại diện cho cầu nối giữa chúng. Chúng ta có thể miêu tả 2 không gian như sau:

- Kernel space: Đây là một tập hợp của các địa chỉ, nơi mà kernel được lưu trữ và thực thi. Kernel space là một phạm vi bộ nhớ, được sở hữu bởi kernel, được bảo vệ khi truy cập bởi flags, việc này để ngăn ngừa bất kỳ một user app độc hại nào vô tình truy cập tới kernel. Mặt khác, kernel có thể truy cập tất cả hệ thống bộ nhớ khi nó chạy với độ ưu tiên cao nhất trong hệ thống. Ở chế độ kernel, CPU có thể truy cập tới tất cả không gian địa chỉ(cả kernel và user)
- User space: Đây là một tập hợp các địa chỉ nơi mà các chương trình bình thường (như getdit) bị hạn chế quyền thực thi. Bạn có thể coi nó như một chiếc hộp kín, vậy nên user program không thể tác động đến bộ nhớ hoặc bất kỳ tài nguyên nào khác mà được sở hữu bởi chương trình khác. Ở chế độ user, CPU chỉ có thể truy cập bộ nhớ được gắn với vùng nhớ mà user space có quyền truy cập. Cách duy nhất để user app chạy trong kernel space là thông qua system call. Ví dụ, có read, write, open, close, mmap, ...Code trong User space chạy với độ ưu tiên thấp. Khi một tiến trình thể hiện một system call, một interrupt được gửi để kernel, khi đó nó sẽ được bật chế độ đặc quyền vậy nên tiến trình có thể chạy trong kernel space. Khi system call được trả về, kernel sẽ tắt chế độ đặc quyền và tiến trình sẽ bị đóng hộp lại

## 1.2. Cấu trúc của Driver

Ở phần trước chúng ta đã hiểu định nghĩa về device driver, trong phần này chúng ta sẽ tìm hiểu về các loại device, về cơ bản Device driver sẽ được chia thành 2 loại chính:

- Character device: là một thiết bị có trình điều khiển giao tiếp bằng cách gửi và nhận các ký tự đơn (byte, octet). Ví dụ - cổng nối tiếp, cổng song song, card âm thanh, bàn phím.
- Block device: là một thiết bị có trình điều khiển giao tiếp bằng cách gửi toàn bộ khối dữ liệu. Ví dụ - đĩa cứng, máy ảnh USB, Disk-On-Key.
- Ngoài ra chúng ta có thêm Network device.

Mọi Driver từ đơn giản đến phức tạp, đều được nâng cấp, phát triển dựa trên một bộ khung cơ bản nhất, ta sẽ cùng xem bộ khung của driver và sẽ đi sâu vào tìm hiểu từng thành phần của nó.

Đầu tiên là cấu trúc của một driver:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
static int __init helloworld_init(void) {
pr_info("Hello world!\n");
```

```
return 0;
}
static void __exit helloworld_exit(void) {
pr_info("End of the world\n");
}
module_init(helloworld_init);
module_exit(helloworld_exit);
MODULE_AUTHOR("John Madieu <john.madieu@gmail.com>");
MODULE_LICENSE("GPL");
```

### 1.2.1. Thành phần `__init` và `__exit`

Ta có thể thấy được hai macro `__init` và `__exit` xuất hiện trong cấu trúc của module, trên thực tế đây là hai kernel macros, được định nghĩa trong file `init.h` trong thư mục `include/linux`.

- Đối với `__init`: là hàm được thực thi đầu tiên khi thực hiện cài đặt driver vào hệ thống linux bằng lệnh shell `insmod driver_name.ko`. Hàm `init` có một vai trò quan trọng trong lập trình driver. Ban đầu hàm sẽ gọi thực thi các hàm xác định số định danh thiết bị, cập nhật các thông số của cấu trúc tập tin, cấu trúc lệnh, đăng ký các cấu trúc vào hệ thống linux, khởi tạo các thông số điều khiển ban đầu cho các thiết bị ngoại vi mà driver điều khiển. Hàm `init` có kiểu dữ liệu trả về dạng `int`, nếu trả về static nghĩa là hàm `init` chỉ dùng riêng cho driver sở hữu.
- Đối với `__exit`: là hàm được thực hiện ngay trước khi driver được tháo gỡ khỏi hệ thống bằng lệnh shell `rmmod device.ko`. Những tác vụ bên trong hàm `exit` được lập trình nhằm khôi phục lại trạng thái hệ thống trước khi cài đặt driver. Chẳng hạn như giải phóng vùng nhớ, timer, vô hiệu hóa các nguồn phát sinh ngắt, ...

### 1.2.2. Thành phần Licensing

Trong module, license được xác định bằng macro: `MODULE_LICENSE("GPL")`;

License sẽ xác định source code có nên được chia sẻ với các nhà phát triển. `MODULE_LICENSE()` sẽ nói với kernel rằng ta đang sử dụng giấy phép nào. Nó có một ảnh hưởng đến hành vi mô-đun của chúng ta, vì giấy phép không tương thích với GPL sẽ dẫn đến mô-đun không thể xem, sử dụng các dịch vụ, chức năng được xuất bởi hạt nhân thông qua Macro

`EXPORT_SYMBOL_GPL ()`, hiển thị các ký hiệu cho các mô-đun tương thích với GPL.

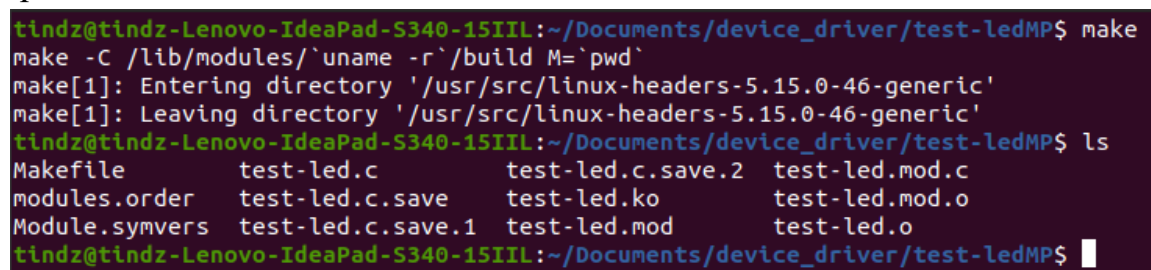
## 2. Gắn kernel module

Sau khi chúng ta hoàn thành được cấu trúc của một driver, làm thế nào để chúng ta có thể gắn lên được kernel?

Cũng giống như các chương trình khác, để kernel có thể hiểu được yêu cầu của chúng ta, ta cần biên dịch những file .c sang file riêng của module, cụ thể là file .ko. Để thực hiện điều đó ta cần có một file đặc biệt, đó là Makefile.

Makefile là một tập script chứa các thông tin : Cấu trúc project (file, dependence), và các lệnh để tạo ra file. Các file được tạo ra gọi là Target, các file phụ thuộc được gọi là Dependence, lệnh được dùng để compile source code được gọi là Action. 1 cú pháp bao gồm Target, dependence, Action được gọi là 1 Rule. Ta sẽ tìm hiểu sâu hơn về Makefile ở phần sau

Sau khi đã có Makefile, ta sẽ thực hiện biên dịch chương trình thông qua câu lệnh “make”. Lúc này trong thư mục sẽ xuất hiện nhiều file khác, một trong số đó là file .ko, ta sẽ sử dụng file này để gắn lên kernel. Để dễ hình dung hơn, ta hãy quan sát hình ảnh sau:



```
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~/Documents/device_driver/test-ledMP$ make
make -C /lib/modules/$(uname -r)/build M=$(pwd)
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-46-generic'
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-46-generic'
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~/Documents/device_driver/test-ledMP$ ls
Makefile      test-led.c      test-led.c.save.2  test-led.mod.c
modules.order test-led.c.save  test-led.ko        test-led.mod.o
Module.symvers test-led.c.save.1 test-led.mod      test-led.o
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~/Documents/device_driver/test-ledMP$
```

Hình 2. Các file được tạo khi make

Sau khi đã có được file .ko, ta sẽ có 2 cách để gắn driver lên kernel:

- Sử dụng modprobe: Khi sử dụng modprobe, nó chỉ tải mô-đun trong / lib / modules / \$ (uname -r), depmod tính toán các dependencies của tất cả các mô-đun có trong thư mục / lib / modules / \$ (uname -r) và đặt thông tin dependencies vào tệp / lib / modules / \$ (uname -r) /modules.dep. Tức là không được khai báo trong modules.dep. Ta sẽ không thể gắn modules lên kernel

`modprobe module-name`

- Sử dụng insmod: Khác với modprobe, insmod sẽ gắn modules lên kernel, từ file .ko, mà không cần biết nó nằm ở vị trí nào. Câu lệnh sử dụng insmod

`insmod filename [module-options]`



Sau khi gắn module thành công, ta sử dụng lệnh “dmesg” để kiểm tra xem module có hoạt động hay không

## II. Lý thuyết về Character device

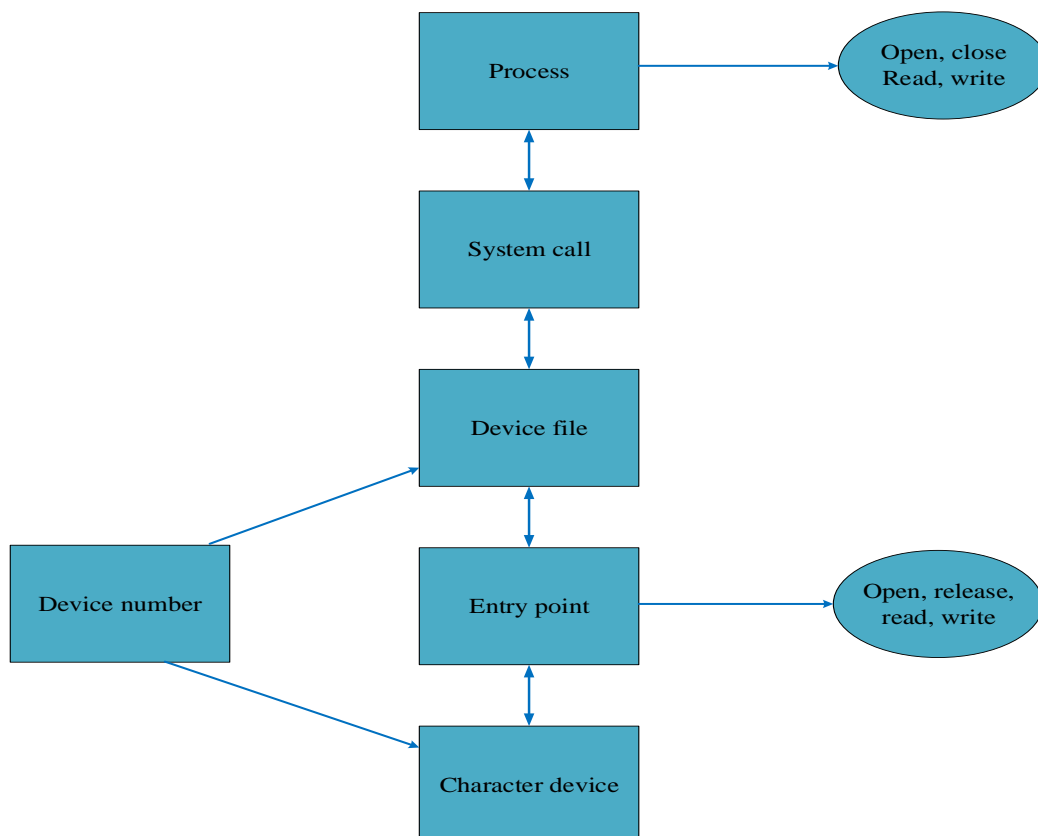
Như đã đề cập từ phần trước, device driver sẽ được chia thành 3 loại chính, nội dung trình bày của phần này sẽ đi sâu vào tìm hiểu về Character device là một trong 3 loại device cơ bản.

### 1. Character device

Character device chuyển đổi dữ liệu từ user app đến kernel. Những đặc điểm và chức năng của device sẽ được lưu ở trong một file đặc biệt có tên là device file được chứa trong thư mục /dev/, chúng ta có thể sử dụng nó để thay đổi dữ liệu giữa thiết bị và user app, và file này cũng cho phép ta điều khiển trạng thái của device.

#### 1.1. Sơ đồ khối

Ta có thể tóm tắt hoạt động của Character Device như sơ đồ dưới đây:



Hình 3. Sơ đồ hoạt động của Character device

Mô tả quá trình như sau:

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.

**Mỗi một system call sẽ tương ứng với một entry point. Vậy mối quan hệ giữa entry point và system call là như thế nào ?**

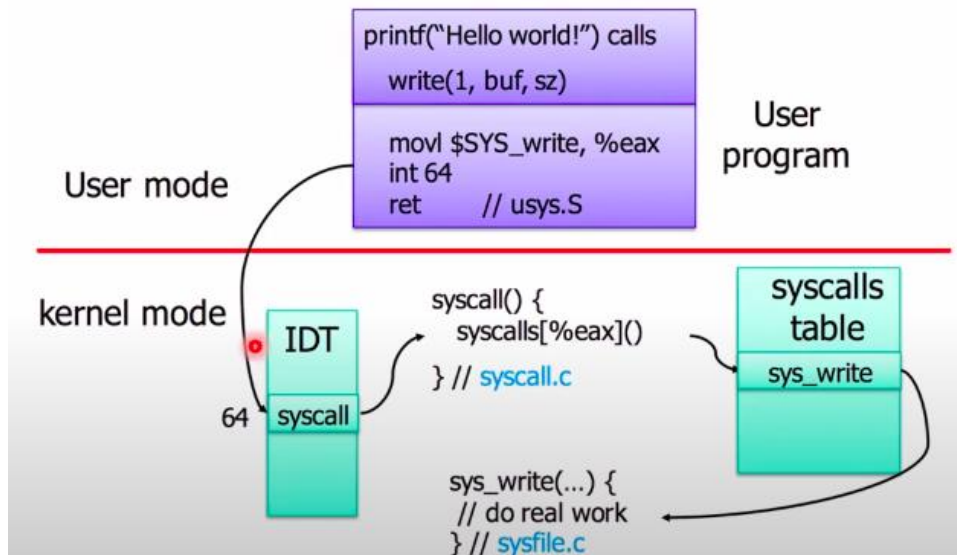
Đầu tiên chúng ta cần biết rằng system call và entry point sẽ tương tác với nhau qua 2 table trung gian:

- Interrupt vector table: Là một table có nhiệm vụ xử lý mọi system call được gọi ra bằng cách tìm giá trị offset phù hợp thông qua hàm `system_call_handler`.
- Sys call table: Là một table có nhiệm vụ sử dụng offset để tìm ra địa chỉ chứa “địa chỉ của entry point” nằm trên Ram.

Bây giờ, chúng ta sẽ lấy một ví dụ về system call write để dễ hình dung hơn về luồng hoạt động của nó.

- Đầu tiên, sau khi ta gọi hàm write trong chương trình, system call write sẽ được gọi ra.
- Kernel sẽ gọi ra Interrupt vector table để xử lý system call write, thông qua `system_call_handler`, nó biết được offset của system call write là 4 (chúng ta đang giả định như vậy).
- Sau khi có được offset, kernel sẽ gọi ra Sys call table, nó sẽ lấy địa chỉ base nhân với giá trị offset (4) để tìm ra nơi chứa địa chỉ của entry point write
- Sau khi tìm ra nơi đó, Kernel tiếp tục theo địa chỉ được chứa trong đó để tìm đến nơi có entry point write, và kết thúc quá trình.

Quá trình sẽ được minh họa như hình dưới đây:



Hình 4. System call tương tác với entry point

Đó là cách mà system call và entry point tìm được nhau khi 1 system call được gọi.

## 1.2. Cấu trúc của Character driver

Cấu trúc của char driver gồm 2 phần:

- Phần 1: OS specific
  - + 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 ngắt.
- Phần 2: Device specific:
  - + Nhóm các hàm khởi tạo/giải phóng 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.

## 2. Phân tích cụ thể

Ta sẽ tiến hành tìm hiểu từng phần của character driver. Cách sử dụng là luồng hoạt động của từng phần đó.

## 2.1. Device file

Là một loại file được tạo ra dùng để 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.

Tuy nhiên, kết quả của các tương tác này thường không giống với kết quả tương tác với file thông thường. 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.

### 2.1.1. Tạo device file thủ công

Chúng ta có thể tạo device file thông qua công cụ mknod. Cách sử dụng công cụ này như sau. Ta gõ lệnh với cú pháp sau vào terminal.

```
mknod -m <quyền truy cập> <tên device file> <kiểu device>
<major> <minor>
```

<quyền truy cập> dùng để thiết lập quyền đọc/ghi/thực thi cho file  
<tên device file> là tên sẽ xuất hiện trong thư mục /dev  
<kiểu device> thể hiện device file là character (c) hay block (b)  
<major> số major number  
<minor> số minor number

Sau khi tạo device file thành công, ta thực hiện kiểm tra lại như hình dưới đây

```
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~$ sudo mknod -m 666 /dev/drv_led c 211 0
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~$ ls -la /dev/drv_led
crw-rw-rw- 1 root root 211, 0 Sep 27 21:06 /dev/drv_led
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~$
```

Hình 5. Tạo device file bằng cách thủ công

### 2.1.2. Tạo device file tự động

Phương pháp này 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 ấy được gọi là uevent (user event). Sau khi nhận được uevent, udevd sẽ tạo ra một device file trong thư mục /dev/.

Để tạo device file tự động, ta thực hiện 3 bước sau:

- Tham chiếu đến thư viện <linux/device.h>

- Tạo 1 lớp các thiết bị

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

Hàm `class_create` có chức năng, tạo 1 lớp các thiết bị có trong thư mục `/sys/class`. Lớp này chứa liên kết thông tin của các thiết bị cùng loại.

Cùng với đó, ta có hàm hủy class

```
void class_destroy(struct class *cls)
```

Tạo 1 thiết bị có trong lớp đó

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

Hàm `device_create` có chức năng tạo 1 thiết bị trong lớp vừa tạo. Udev sẽ tạo ra 1 device file tương ứng trong `/dev/`.

Ta cũng có hàm hủy device

```
void device_destroy(struct class *cls, dev_t devt)
```

## 2.2. 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 biết được device driver nào phù hợp với device file nào
- Minor number: giúp device driver biết được cần điều khiển device nào

Linux kernel cũng có các hàm hoặc macro để hỗ trợ chúng ta làm việc với biến kiểu `dev_t`. Chúng bao gồm:

HÀM MACRO	Ý NGHĨA
-----------	---------

int MAJOR(dev_t dev)	Trả về major number từ device number
int MINOR(dev_t dev)	Trả về minor number từ device number
MKDEV(int major, int minor)	Gộp số major và minor để tạo thành e device number
unsigned imajor(struct inode* inode)	Trả về major number từ cấu trúc inode cấu trúc inode dùng để mô tả device file)
unsigned iminor(struct inode* inode)	Trả về minor number từ inode mô tả device file

### 2.2.1. Cấp phát tĩnh

Đầu tiên ta cần chọn được major number, Để biết được nên chọn giá trị nào, ta gõ lệnh “cat /proc/devices” (hình 1). Cột đầu tiên chứa các major number đã được sử dụng bởi các driver khác. Vì thế, ta không chọn những số này. Ngoại trừ những số này, ta có thể chọn bất cứ số nào trong khoảng từ 0 đến ( $2^{12} - 1$ ) làm major number.

```
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~$ 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
```

Hình 6. Kiểm tra device number của các devices

Sau khi đã chọn được major number, ta sẽ 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 phát 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.

Hàm **register\_chrdev\_region** có chức năng đăng ký 1 bộ gồm các count số, bắt đầu từ first cho character device có tên là name

```
int register_chrdev_region (dev_t first, unsigned count, const char *name)
```

Khi không còn dùng đến device number nữa, chúng ta cũng cần giải phóng nó. Hàm giải phóng device number thường được đặt trong hàm kết thúc của char driver. Hàm giải phóng device number có nguyên mẫu như sau:

```
void unregister_chrdev_region(dev_t first, unsigned int cnt)
```

### 2.2.2. Cấp phát động

Nếu lựa chọn phương pháp cấp phát tĩnh device number, thì device number đó có thể đã được sử dụng trên những máy tính khác, dẫn tới char driver không hoạt động được trên các máy tính ấy. Để giải quyết vấn đề này, lập trình viên nên sử dụng phương pháp cấp phát động device number.

Trong phương pháp này, Linux kernel cung cấp một hàm là **alloc\_chrdev\_region** với chức năng là tìm ra một giá trị có thể dùng làm device number. Ta thường gọi hàm này trong hàm khởi tạo của char driver.

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

Tương tự như cấp phát tĩnh, sau khi không còn dùng đến device number nữa, ta cần giải phóng nó.

## 2.3. Cấp phát bộ nhớ và khởi tạo thiết bị

### 2.3.1. Cấp phát bộ nhớ

Để thực hiện cấp phát bộ nhớ dưới kernel space, ta thường dùng hàm **kmalloc**. Đầu tiên ta cần tham chiếu đến thư viện `<linux/slab.h>`

Tiếp đó, ta sử dụng câu lệnh

```
kmalloc(N, GFP_KERNEL);
```

Nếu thành công, hàm này sẽ trả về địa chỉ của vùng nhớ đã được kernel cấp phát. Nếu thất bại, hàm này sẽ trả về NULL. Trong khi lập trình, ta nên kiểm tra giá trị trả về để có những hành động phù hợp.

### 2.3.2. Khởi tạo thiết bị

Dưới góc độ lập trình, các thiết bị vật lý gồm các thanh ghi. Các thanh ghi được phân làm ba loại:

- Thanh ghi điều khiển
- Thanh ghi dữ liệu
- Thanh ghi trạng thái

Công việc khởi tạo thiết bị vật lý thực chất là khởi tạo các thanh ghi nói trên. Để khởi tạo các thanh ghi một cách đúng đắn, ta cần đọc tài liệu datasheet của thiết bị, đặc biệt là mục "Register map".

## 2.4. Các hàm Entry point

Linux kernel cần có một cấu trúc dùng để mô tả các character device. Cấu trúc ấy 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.

Linux kernel sử dụng cấu trúc cdev để mô tả một character device, ta cần tham chiếu đến thư viện <linux/cdev.h> để có thể sử dụng cdev. Cấu trúc này bao gồm một 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 */  
    ...  
}
```

Thêm vào đó, Linux kernel cung cấp các hàm để đọc hoặc ghi giá trị vào các trường của cdev. Ta nên sử dụng các hàm này, thay vì đọc/ghi trực tiếp. Dưới đây là một số hàm ta hay sử dụng:



Tên hàm	Chức năng
struct cdev *cdev_alloc(void)	Cấp phát bộ nhớ cho cấu trúc cdev
void cdev_init(struct cdev *p, struct file_operations *fops)	Khởi tạo các trường của cấu trúc cdev
int cdev_add(struct cdev *p, dev_t first, unsigned count)	Đăng kí cấu trúc cdev với Linux kernel và liên kết cấu trúc này với device file
void cdev_del(struct cdev *p)	Hủy bỏ vùng nhớ của cấu trúc cdev

### 2.4.1. Hàm Open và Release

Open và release là 2 hàm của trường file\_operations, để sử dụng được ta cần tham chiếu đến thư viện <linux/fs.h>

Đầu tiên ta có cấu trúc của hàm open

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

Chức năng của hàm open là 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...

Tiếp theo, ta cùng xem cấu trúc của hàm release

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

Hàm release có chức năng khi có một system call close trên user space, nó sẽ được gọi tới để thực hiện một số việc như tắt thiết bị, hoặc làm thiết bị ngừng hoạt động.

### 2.4.2. Hàm Read và Write

Đối với hàm Read, ta cần hiểu được các bước khi đọc dữ liệu từ một char device.

Đầu tiên, tiến trình gọi system call open để mở device file tương ứng của char device này. Sau đó, tiến trình gọi system call read để đọc dữ liệu

từ device file này. Khi đó, Linux kernel sẽ kích hoạt entry point read của char driver tương ứng. Cuối cùng, entry point read đọc dữ liệu từ char device ra rồi trả về cho tiến trình trên user space.

Ta có cấu trúc của hàm Read

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

Chức năng của hàm này là đọc dữ liệu từ char device buffer vào kernel buffer, sau đó sao chép dữ liệu từ kernel buffer vào trong user buffer. Để thực hiện được chức năng trên ta cần làm những bước sau:

- Đọc size byte dữ liệu tính từ vị trí \*off trên buffer của char device vào kernel buffer.
- Sử dụng hàm copy\_to\_user để sao chép dữ liệu từ kernel buffer vào user buffer.
- Cập nhật lại giá trị \*off để lần đọc dữ liệu tiếp theo sẽ bắt đầu từ vị trí mới.
- Trả về số lượng byte đã đọc được từ char device.

Đối với hàm Write, ta cũng cần hiểu các bước khi ghi dữ liệu vào một char device như sau. Đầu tiên, tiến trình gọi system call open để mở device file tương ứng của char device này. Sau đó, tiến trình gọi system call write để ghi dữ liệu vào device file này. Khi đó, Linux kernel sẽ kích hoạt entry point write của char driver tương ứng. Cuối cùng, entry point write sẽ đọc dữ liệu từ tiến trình trên user space rồi ghi dữ liệu này vào char device.

Ta có cấu trúc của hàm Write

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

Hàm write có chức năng sao chép dữ liệu từ user buffer sang kernel buffer, sau đó ghi dữ liệu từ kernel buffer vào char device.

Tương tự như hàm `read`, để có thể thực hiện được chức năng này, ta cần triển khai entry point `write` như sau

- Sử dụng hàm `copy_from_user` để sao chép `size` byte dữ liệu từ user buffer kernel buffer.
- Ghi dữ liệu từ kernel buffer vào buffer của char device, kể từ vị trí `*off` trên buffer của char device.
- Cập nhật lại giá trị `*off` để lần ghi dữ liệu tiếp theo sẽ bắt đầu từ vị trí mới.
- Trả về số lượng byte đã ghi vào char device.

### 3. Giao tiếp với phần cứng

#### 3.1. Truy cập phần cứng trên hệ điều hành Linux

Để có thể giao tiếp với phần cứng, ta cần tham chiếu đến thư viện `<linux/ioport.h>`

Có 2 cách để trao đổi dữ liệu giữa bộ xử lý và ngoại vi:

- Port I/O: Thường được sử dụng trong kiến trúc x86
- Memory mapped I/O: Thường được sử dụng trong kiến trúc arm

##### 3.1.1. Port I/O

##### 3.1.2. Memory mapped I/O

Ở đây chúng ta sẽ bàn luận về memory mapped I/O do ta sử dụng kiến trúc arm.

Để có thể truy cập vào phần cứng trên hệ điều hành Linux thông qua MMIO, ta cần có 3 bước:

- B1: Yêu cầu truy cập tới MMIO từ kernel

Khi ta muốn sử dụng 1 khu vực nào đó có trong sự quản lý của kernel, ta cần phải xin kernel cấp phép sử dụng khu vực đó. Để làm được điều này ta sử dụng hàm

```
struct resource *request_mem_region(unsigned long start, unsigned long len, char name)
```

Start: Địa chỉ bắt đầu của khu vực cần cấp phát

Len: kích thước của khu vực

Name: tên khu vực

- B2: Map địa chỉ của thanh ghi vật lý tới địa chỉ ảo  
Do cơ chế hoạt động của kernel không thể sử dụng được các địa chỉ vật lý thông thường, nên ta cần phải tạo 1 địa chỉ ảo và map nó vào với địa chỉ vật lý. Ta sử dụng hàm dưới đây

```
void *ioremap(unsigned long phy_addr, unsigned long size)
Phy_addr: địa chỉ vật lý
size: kích thước của khu vực
```

- B3: Sử dụng API để đọc và ghi tới thanh ghi.  
Khi đã map thành công địa chỉ ảo và địa chỉ vật lý, ta cần sử dụng kernel API để đọc ghi dữ liệu tới thanh ghi

Đọc	Ghi
readl(address)	writel(unsigned value, address)

## II. Triển Khai

Sau khi đã tìm hiểu về lý thuyết của Char device, ta cần 1 ví dụ minh họa cụ thể về cách sử dụng cách hàm entry point, tương tác với device file. Phần này, chúng ta sẽ đi xây dựng 1 char device đơn giản dùng để điều khiển đèn led thông qua câu lệnh echo

### 1. Trình bày vấn đề cần giải quyết

Chúng ta cần xây dựng một char device theo kiến trúc arm để có thể gắn lên kernel của KIT zynq hoặc zynqMP. Thao tác điều khiển dãy đèn led có trên những bộ KIT thông qua các lệnh gọi echo.

Cụ thể, khi truyền tham số 1 vào thì đèn sẽ phát sáng, 0 thì đèn sẽ tắt. Thông qua đó sẽ giúp ta hiểu được sâu sắc về các hoạt động của character device, luồng hoạt động của tầng user space với kernel space như thế nào. Nhằm củng cố và nắm chắc kiến thức hơn.

### 2. Ý tưởng thực hiện

Để xây dựng 1 char device điều khiển đèn led, ta sẽ chia thành 2 phần. Cụ thể

- Phần 1: Xây dựng bộ khung của 1 char device (bao gồm các entry point open, release, read, write, các hàm init, exit)
- Phần 2: Truy cập phần cứng tới KIT (khai báo thanh ghi vật lý, map tới địa chỉ ảo, xây dựng các hàm đọc ghi vào thanh ghi)

## 2.1. Xây dựng Character device

Đầu tiên, như mọi bài toán khác, ta đều cần phải khai báo những thư viện cần thiết sẽ được sử dụng trong các hàm của mình. Ở đây, để tạo char device, ta sẽ có những thư viện cần thiết như sau

```
#include <linux/device.h>
#include <linux/slab.h>
#include <linux/cdev.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/uaccess.h>
#include <linux/ioport.h>
#include <linux/io.h>
```

Tiếp đó, ta cần khởi tạo một struct drvled, bên trong chứa các trường cần thiết mà ta sẽ sử dụng để cấu thành lên một char device

```
static struct {
    dev_t num; // device number của device
    struct class *vclass; // tạo class cho thiết bị
    struct device *vdevice; //tạo device file trong /dev/
    struct cdev *vcdev; //cấu trúc cdev để mô tả device
    unsigned int led_status; //trạng thái của đèn
    void __iomem *regbase; //địa chỉ ảo của struct
}drvled;
```

Sau khi tạo một struct có đầy đủ những trường cần sử dụng, ta tiến hành tạo 2 hàm quan trọng là init và exit

Bên trong hàm init, ta thực hiện những bước sau:

- Cấp quyền truy cập từ hệ điều hành Linux
- Map virtual address vào physical address
- Cấp phát device number
- Tạo device file

- Đăng kí các entry point với kernel( phần sau sẽ nói rõ hơn về các entry point)

Cụ thể, ta sẽ cùng xem phần code bên dưới đây:

```
static int __init drvled_init(void)
{
    int ret = 0;
    /*cap phat quyen truy cap*/
    if(!request_mem_region(GPIO_BASE, GPIO_SIZE, "drvled"))
    {
        printk("KHONG TRUY CAP DUOC PHAN CUNG");
        ret = -EBUSY;
    }
    /*map dia chi vat ly va dia chi ao*/
    drvled.regbase = ioremap(GPIO_BASE, GPIO_SIZE);
    if(drvled.regbase == NULL)
    {
        printk("failed");
        ret = -ENOMEM;
    }
    /*cap phat device number*/
    ret = alloc_chrdev_region(&drvled.num, 0, 1, "test-led");
    if(ret < 0)
    {
        printk("failed");
    }
    /* tao device file */
    drvled.vclass = class_create(THIS_MODULE, "class_drvled");
    if(drvled.vclass == NULL)
    {
        printk("failed");
        class_destroy(drvled.vclass);
    }
    drvled.vdevice = device_create(drvled.vclass, NULL, drvled.num, NULL,
"drv_led");
    if(IS_ERR(drvled.vdevice))
    {
        printk("failed");
        device_destroy(drvled.vclass, drvled.num);
    }
    /* dang ki entry point voi kernel*/
    drvled.vcdev = cdev_alloc();
```

```

if(drvled.vcdev == NULL)
{
    printk("failed");
}

cdev_init(&drvled.vcdev, &drvled.fops);
ret = cdev_add(&drvled.vcdev, drvled.num, 1);
if(ret < 0)
{
    printk("failed");
}

drvled.directions();
drvled.select(1);
printk("success");
return 0;
}

```

Tiếp theo đến hàm `exit`, nhiệm vụ quan trọng nhất của hàm này là giải phóng những gì đã được cấp phát trên hàm `init`, cụ thể:

- Giải phóng vùng nhớ được kernel cấp phát
- Giải phóng địa chỉ ảo
- Giải phóng `cdev`
- Giải phóng device number
- Xóa bộ nhớ device file

Cụ thể như code dưới đây:

```

static void __exit drvled_exit(void)
{
    /*giai phong vung nho*/
    release_mem_region(GPIO_BASE, GPIO_SIZE);
    /*giai phong dia chi ao*/
    iounmap(drvled.regbase);
    /*giai phong cdev*/
    cdev_del(&drvled.vcdev);
    /*xoa bo nho device file*/
    class_destroy(drvled.vclass);
    device_destroy(drvled.vclass, drvled.num);
    /* giai phong device number */
}

```

```
unregister_chrdev_region(drvled.num, 1);  
printk("exit");  
}
```

Sau khi đã tạo thành công 2 hàm quan trọng nhất của một device, ta cần tiến hành tạo các entry point cho trường file\_operations của cdev. Trong bài toán này, ta chỉ cần tạo hàm read và write.

Đối với hàm read, ta sẽ sử dụng hàm copy\_to\_user bên trong để có thể đọc được dữ liệu từ kernel buffer, ta sẽ mô tả qua quá trình hàm như sau. Tạo một mảng có 2 phần tử mang giá trị là OFF và ON tương ứng với 0 và 1 của mảng, mảng này sẽ hiển thị trạng thái của device file. Ta sẽ dùng hàm strlen để tính ra kích thước của mảng, sau đó gán vào biến ssize, biến ssize sẽ tương ứng với kích thước byte của 2 phần tử 0 và 1. Sử dụng hàm copy\_to\_user để đưa dữ liệu từ kernel lên user, cuối cùng sẽ cộng vào off giá trị ssize để lưu vị trí tiếp theo sẽ được đọc. Cụ thể:

```
static ssize_t read_drvled(struct file *file, char __user *buff, size_t size, loff_t  
*off)  
{  
    char *msg[] = {"OFF\n", "ON\n"};  
    int ssize;  
    ssize = strlen(msg[drvled.led_status]);  
    if(copy_to_user(buff, msg[drvled.led_status], ssize))  
    {  
        return -EFAULT;  
    }  
    off += ssize;  
    return ssize;  
}
```

Sau khi tạo được hàm read, ta sẽ tạo hàm write, với mục đích ghi dữ liệu từ user xuống kernel. Mô tả về hàm write như sau, ta sẽ tạo 1 biến kernel\_buf để lưu dữ liệu được ghi từ user thông qua hàm copy\_from\_user. Nếu kernel\_buf = 0, ta sẽ set up cho đèn sáng và ngược lại, thông qua hàm drvled\_select, ta sẽ bàn luận về hàm này sau. Cụ thể:



```
static ssize_t write_drvled(struct file *file, const char __user *buff, size_t size,
loff_t *off)
{
    char kernel_buf = 0;
    if(copy_from_user(&kernel_buf, buff, 1))
        return -EFAULT;
    if(kernel_buf == '1'){
        drvled_select(1);
        printk("led_on");
    }
    else if(kernel_buf == '0'){
        drvled_select(0);
        printk("led_off");
    }
    return size;
}
```

Trong hàm write, ta có nhắc đến hàm driverled\_select. Đây là hàm dùng để cài đặt trạng thái của đèn bật hoặc tắt, bằng cách gán giá trị status truyền từ user vào trường led\_status của device. Cụ thể như sau:

```
static void drvled_select(unsigned int status)
{
    drvled.led_status = status;
}
```

Sau khi hoàn thiện hàm drvled\_select, ta bản bộ khung của char điều khiển đèn led đã được hoàn thành.

## 2.2. Truy cập tới phần cứng của KIT ZCU102

Để có thể điều khiển phần cứng của mạch, điều bắt buộc là phải truy cập tới địa chỉ thanh ghi của nó, ở trường hợp led cũng vậy, ta cần tương tác với các thanh ghi dùng để điều khiển led sáng và tối. Cụ thể ở đây là 2 thanh ghi:

- DATA: Thanh ghi dữ liệu, khi ta truyền giá trị 1 vào trong kernel, DATA sẽ có trách nhiệm điều khiển cho led sáng và ngược lại
- DIRECT: Thanh ghi điều khiển, sẽ được dùng khi ta muốn các I/O port là input hay output.

Tiếp đó, ta cần biết về cơ chế truy xuất thanh ghi của kernel, ở mọi kiến trúc, địa chỉ vật lý hoặc ảo của thanh ghi luôn được chia thành 2 phần:

- Register Base: Hiểu đơn giản đây là loại thanh ghi mang địa chỉ bắt đầu của tất cả các thanh ghi cần sử dụng
- Register offset: Đây là loại thanh ghi mà ta sẽ dùng để truy xuất vào cụ thể từng thanh ghi một.

Ta sẽ cần tìm được địa chỉ base của AXI GPIO bus và offset của thanh ghi DATA, DIRECT.

Ta sẽ có 2 cách để tìm thấy địa chỉ base của AXI GPIO:

- Tìm trong phần address design của Vivado khi ta xây dựng hardware file
- Tìm trong device - tree

Ở đây, ta sẽ lựa chọn cách số 2. Để vào được thư mục device-tree, ta cần truy cập vào project của KIT, vào file ở trong đường dẫn như sau: /components/plnx\_workspace/device-tree/device-tree/. Sau khi vào thành công các file device tree sẽ hiển thị như dưới:

design_1.bda	include	ps7_init_gpl.h	system-conf.dtsi
design_1_wrapper.bit	pcw.dtsi	ps7_init.h	system-top.dts
device-tree.mss	pl.dtsi	ps7_init.html	zedboard.dtsi
drivers	ps7_init.c	ps7_init.tcl	zynq-7000.dtsi
hardware_description.xsa	ps7_init_gpl.c	skeleton.dtsi	

*Hình 7. Các file có trong thư mục device tree*

Tiếp tục mở file pl.dtsi để kiểm tra địa chỉ base của AXI GPIO

```
{
    amba_pl: amba_pl {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "simple-bus";
        ranges ;
        led_ip_0: led_ip@43c00000 {
            clock-names = "s_axi_aclk";
            clocks = <&clkc 15>;
            compatible = "xlnx,led-ip-1.0";
            reg = <0x43c00000 0x10000>;
        };
    };
};
```

*Hình 8. Xem địa chỉ của IP tại file pl.dtsi*

Dễ dàng quan sát được, thanh ghi Reg mang địa chỉ 0x43c00000 và có kích thước là 0x10000.

Sau khi đã có địa chỉ base, ta tiếp tục có địa chỉ offset của thanh ghi DATA và DIRECT trong bảng được trích dẫn trong tài liệu pg144(AXI GPIO) như sau sau:

Table 2-4: Registers

Address Space Offset <sup>(3)</sup>	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER <sup>(1)</sup>	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER <sup>(1)</sup>	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR <sup>(1)</sup>	R/TOW <sup>(2)</sup>	0x0	IP Interrupt Status Register.

Hình 9. Bảng địa chỉ offset của AXI GPIO

Sau khi đã hoàn thành, ta sẽ tiến hành define các thanh ghi như sau:

```
#define GPIO_BASE    0x43c00000
#define GPIO_SIZE    0x10000

#define GPIO_DATA_0   0x0
#define GPIO_DIR_0    0x4
#define GPIO_BIT      (1 << 7) | (1 << 6) | (1 << 5) //dùng để dịch bit
```

Ta cần chỉnh sửa lại hàm drvled\_select để hàm đọc dữ liệu từ thanh ghi DATA, ở đây, ta sẽ sử dụng 2 API kernel là readl( dùng để đọc dữ liệu từ thanh ghi), và writel( dùng để lưu dữ liệu vào thanh ghi). Cụ thể như sau

```
static void drvled_select(unsigned int status)
{
    u32 val = 0;
    val = readl(drvled.regbase + GPIO_DATA_0);
    if(status == 1)
    {
        val |= GPIO_BIT;
    }
    else
        val &= ~GPIO_BIT;
    writel(val, drvled.regbase + GPIO_DATA_0);
    drvled.led_status = status;
}
```

Sau khi chỉnh sửa xong, ta đã hoàn thành char device điều khiển đèn led.

## 2.3. Gắn devices lên kernel

Sau khi đã hoàn thành được character device, ta sẽ thực hiện gắn module lên kernel. Như đã đề cập ở phần lý thuyết, để kernel có thể hiểu được chương trình, ta cần biên dịch chương trình sang file .ko bằng makefile.

### 2.3.1. Biên dịch chương trình

Ta sẽ cùng xem và phân tích source code của file Makefile

```
obj-m      = test.o

all:
    make ARCH=arm64 CROSS_COMPILE=$(CROSS) -C
$(KERNEL) M=$(PWD) modules

clean:
    make -C $(KERNEL) M=$(PWD) clean
```

Chúng ta sẽ phân tích cấu trúc này từ trên xuống:

- obj-m: Lệnh này sẽ khai báo file.o được chuyển đổi từ file.c của ta
- all: Là target mặc định, khi bạn thực hiện lệnh make thì chương trình make sẽ tiến hành compile để tạo ra chương trình cuối cùng tương ứng với target này
- clean: Tiến hành clean chương trình khi đã hoàn thành

Để có thể make được module sang kiến trúc mà ta mong muốn, ở đây là kiến trúc arm64, trong trường all ta sẽ cần trỏ vào 2 thành phần:

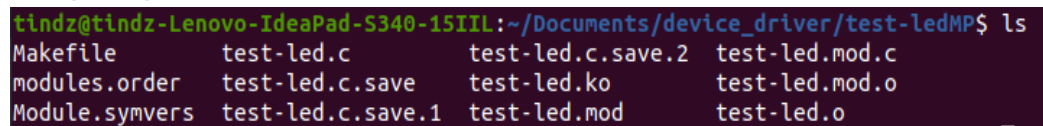
- Compiler: Đầu tiên ta cần chỉ định kiểu kiến trúc sẽ được biên dịch ra thông qua lệnh ARCH, tiếp theo ta sẽ gán biến CROSS vào CROSS\_COMPILE để có thể link tới trình biên dịch khi make
- Kernel của KIT ZCU102: ở đây ta sẽ tạo một biến có tên là KERNEL, khi make ta sẽ gán đường dẫn của kernel tới biến này

Để dễ hình dung hơn, ta sẽ cùng xem cách thức hoạt động của Makefile. Đầu tiên, Kbuild trong kernel sẽ xây dựng file .o từ file.c, sẽ được dùng để liên kết sau khi thực hiện xong các bước khác để tạo ra file module là file .ko. Sau đó, kernel sẽ xem target trong phần all để thực hiện, nó sẽ tiến hành tìm đến compiler thông qua đường dẫn được gán vào biến, tiếp đến nó thay đổi thư mục hiện tại thành thư mục kernel với tùy chọn -C. Ở đó nó tìm thấy makefile cấp có trong kernel. Tùy chọn M = khiến makefile đó di chuyển trở lại thư mục nguồn mô-đun của ta trước khi cố gắng xây dựng mục tiêu mô-đun.

Sau khi có được 2 file là Makefile và file.c, ta sẽ tiến hành make thông qua câu lệnh:

```
make                                KERNEL=<LINUX_SOURCE_DIR>  
CROSS=<TOOLCHAIN_DIR>/bin/aarch64-linux-gnueabihf-
```

Khi đã compile thành công, thư mục sẽ xuất hiện file .ko (là file module sẽ dùng để gắn lên kernel)



```
tindz@tindz-Lenovo-IdeaPad-S340-15IIL:~/Documents/device_driver/test-ledMP$ ls  
Makefile      test-led.c      test-led.c.save.2  test-led.mod.c  
modules.order test-led.c.save  test-led.ko        test-led.mod.o  
Module.symvers test-led.c.save.1 test-led.mod       test-led.o
```

*Hình 10. Các file sau khi thực hiện make*

### 2.3.2. Gắn module lên kernel

Khi đã có file .ko, ta sẽ thực hiện gắn module lên kernel thông qua lệnh insmod. Ta sẽ thực hiện như sau

```
sudo insmod test-led.ko
```

Lúc này module đã được gắn lên kernel và ta có thể thao tác với nó thông qua câu lệnh

## 3. Kết quả - nhận xét - đánh giá

Sau khi hoàn thành tất cả các bước, điều cuối cùng chúng ta cần phải làm là kiểm tra xem liệu driver có hoạt động như chúng ta mong muốn hay không

### 3.1. Kết quả

Ta sẽ thực hiện lệnh echo để truyền giá trị 1 vào device file rồi kiểm tra xem đèn có được bật hay không

```
echo 1 > /dev/drv_led
```

Kiểm tra đèn trên KIT ZCU102, có thể thấy đèn DS44 đã sáng

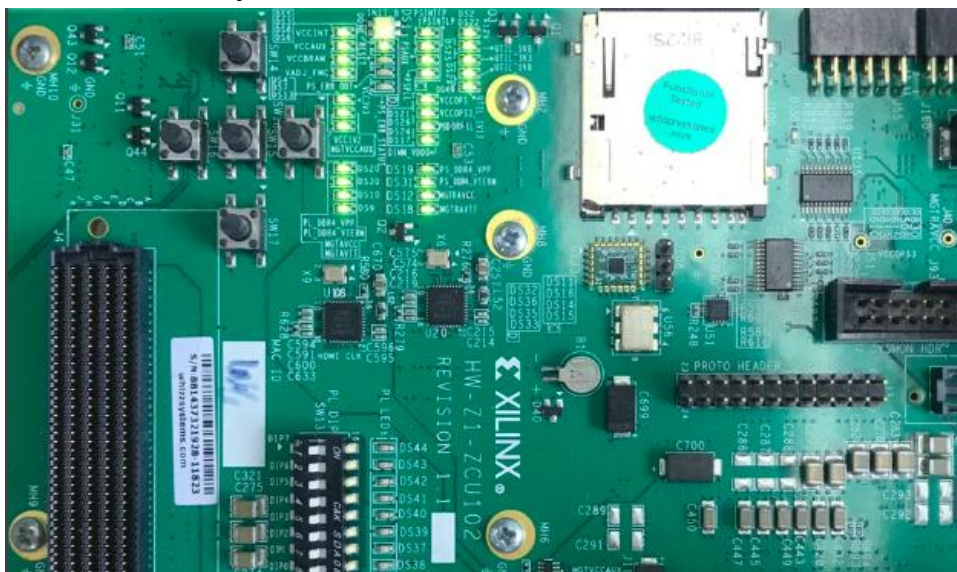


Hình 11. Điều khiển đèn led sáng

Tiếp theo, ta thực hiện tắt đèn

```
echo > 0 /dev/drv_led
```

Kiểm tra có thấy, đèn trên bo mạch đã được tắt



Hình 12. Điều khiển đèn led tối

Như vậy, kết quả đã như mong muốn của ta



### 3.2. Nhận xét - đánh giá

Thông qua mini project này, chúng ta hãy cùng thuật lại những kiến thức đã học được trong suốt quá trình tìm hiểu và xây dựng hệ thống điều khiển đèn led bằng câu lệnh:

- Đầu tiên, ta được biết tới một phần mới trong cấu trúc của Firmware đó là device driver với những khái niệm hoàn toàn mới như: Device file, Makefile, Character device,...
- Tiếp theo, ta được củng cố về những kiến thức lý thuyết đã được tìm hiểu từ trước đó, hình dung một cách thực tế cách mà user space và kernel space tương tác với nhau qua device file như thế nào. Luồng hoạt động hoàn chỉnh của một device driver
- Cách để biên dịch chương trình sang một kiến trúc khác dựa vào Makefile và Compiler.
- Cách xử lý vấn đề khi gặp lỗi, cần tìm hiểu vấn đề từ tổng quát đến chi tiết để tìm được mấu chốt của bài toán, từ đó dựa vào keyword để xử lý được vấn đề gặp phải

Chốt lại, sau khi hoàn thành mini project, ta đã có thêm được nhiều kiến thức mới, tuy nhiên những kiến thức này vẫn đang ở mức độ cơ bản, trong phần sau, chúng ta sẽ cùng tìm hiểu sâu hơn về Platform Driver và Device Tree, cùng như phát triển hệ thống điều khiển đèn led

### III. Platform Device Driver và Device Tree

Chúng ta đều biết rằng, các cảm biến hay module giao tiếp với kernel thông qua các giao thức như I2C, SPI, ... Và những thiết bị sử dụng kiểu giao tiếp nào sẽ được gọi với cái tên của kiểu giao tiếp ấy, chẳng hạn như I2C device hay SPI device. Tuy nhiên, ít ai biết rằng tất cả những kiểu device đó đều được gọi chung với cái tên là Platform device.

Ngoài ra, đối với những bo mạch ở thời nay, hầu hết chúng đều không sử dụng phương pháp quét phần cứng để biết phần cứng có những thứ gì, mà nó sử dụng một cơ chế khác, có tên là Device Tree, để hình dung ra những thứ có trên KIT.

Vậy Platform Device Driver và Device Tree là gì? Mối quan hệ của chúng ra sao? Ta sẽ cùng tìm hiểu về những thứ đó ở trong phần này.

## 1. Lý thuyết về Platform device driver

Chúng ta đều biết về Plug and play nghĩa là thiết bị cắm là chạy. Nó có thể được xử lý bởi kernel ngay lập tức khi chúng ta cắm vào. Đó có thể là USB, PCI Express, hoặc bất kì thiết bị tự động phát hiện khác. Tuy nhiên, một số thiết bị khác cũng tồn tại, nhưng nó không được phát hiện, và kernel cần biết về sự tồn tại của nó để quản lý. Đó là I2C, UART, SPI, và các thiết bị khác có các bus không có khả năng liệt kê.

Chúng ta có những BUS vật lý real như: USB, I2C, SPI, UART, PCI,... Những BUS này được gọi là bus controller, chúng là một phần của SoC, chúng không thể bị xóa, và không được không tự động phát hiện được, vậy nên nó được gọi là Platform device.

Mọi người thường nói platform device là những thiết bị trên chip. Thực tế, đây là một luận điểm đúng nhưng chưa đủ. Những thiết bị được kết nối thông qua I2C hoặc SPI không trên chip nhưng vẫn được tính là platform devices, bởi vì platform devices là những thiết bị không được tự phát hiện. Như vậy, những thiết bị USB, PCIe sẽ được tự phát hiện, vậy nên nó không phải platform device.

Để làm việc với platform devices sẽ cần 2 yêu cầu:

- Đăng kí một platform driver (với tên duy nhất) để quản lý thiết bị của chúng ta
- Đăng kí platform device với tên giống như driver và resource của nó, để cho kernel biết rằng thiết bị của ta ở đó

### 1.1. Platform driver

Trước hết cần chú ý rằng, platform driver dùng để điều khiển platform device, nhưng không phải tất cả các platform device đều được điều khiển bằng platform driver.

Platform driver đơn giản là những driver điều khiển các thiết bị nằm trên pseudo platform bus (nghĩa là các đường bus không chính thống). Ví dụ I2C device và SPI device, chúng là những platform device, và nằm trên I2C hoặc SPI bus (các đường bus chính thống), chứ không phải platform bus.

Ta có cấu trúc của platform driver như sau:



```
static struct platform_driver mypdrv = {
    .probe      = my_pdrv_probe,
    .remove     = my_pdrv_remove,
    .driver      = {
        .name = "my_platform_driver",
        .owner = THIS_MODULE,
    },
};
```

Chúng ta sẽ cùng đề cập tới các trường có trong struct platform driver:

- Hàm probe():
- Hàm remove():
- Struct device\_driver:

**Như đã nói ở phần trước, để làm việc với platform driver chúng ta sẽ cần đăng kí với kernel, để kernel biết về sự tồn tại của nó. Vậy đăng kí như thế nào?**

Ta sẽ có 2 cách để đăng kí:

- Sử dụng hàm platform\_driver\_register(): Khi sử dụng hàm này, nó sẽ đăng kí một danh sách các driver duy trì bởi kernel. Khi đó hàm probe có thể được gọi mỗi khi việc match device và driver xảy ra.
- Sử dụng platform\_driver\_probe(): kernel lập tức chạy một vòng lặp để thử match device và driver bằng tên mà chúng đăng kí, nếu match thì gọi hàm probe, đồng nghĩa với việc device đã xuất hiện. Nếu không match, thì driver bị bỏ qua. Hàm này ngăn việc hàm probe bị gọi về sau vì nó không đăng kí driver với hệ thống. Khi dùng hàm này, hàm probe được đặt trong vùng \_\_init, nghĩa là hàm probe sẽ được giải phóng khi quá trình boot kernel hoàn tất, qua đó hàm probe về sau sẽ không được gọi nữa đồng thời giảm bộ nhớ chiếm dụng của driver. Và luôn nhớ dùng phương pháp sau để đảm bảo rằng device đã xuất hiện trong hệ thống:

```
ret = platform_driver_probe(&mypdrv, my_pdrv_probe);
```

## 1.2. Platform device

Platform device là các device nằm trên platform bus. Cũng giống như những gì ta làm với driver, ta phải báo cho kernel biết rằng device đang cần một driver. Một platform device được biểu diễn bằng struct platform\_device:

```
struct platform_device {
    const char *name;
    u32 id;
    struct device dev;
```

```
u32 num_resources;  
struct resource *resource;  
};
```

### 1.3. Match driver và device

Để có thể match được driver và device với nhau, chúng ta sẽ có 2 cách để thực hiện điều đó:

- Sử dụng Resource và Data: Với cách sử dụng này, sẽ phù hợp với những kernel không hỗ trợ device tree. Chúng ta sẽ đăng kí một source file code.c. Ta sẽ không bàn luận sâu về cách sử dụng này do kernel của chúng ta cho phép hỗ trợ device tree.
- Sử dụng Device Tree: Với cách sử dụng Resource và Data, ta sẽ phải build lại kernel mỗi khi thay đổi cấu hình device. Để đơn giản hơn, người ta tách việc khai báo device ra khỏi kernel source, và sử dụng device tree (DTS). Với device tree thì platform data và resource là như nhau. Device tree là một file mô tả phần cứng và có cấu trúc tương tự như một cây, mỗi device được đại diện bởi một node, và dữ liệu cũng như tài nguyên của thiết bị chính là thuộc tính của các node. Với phương pháp này, ta chỉ cần biên dịch lại mỗi device tree khi có thay đổi.

**Vậy làm thế nào để platform device và platform driver có thể match được với nhau?**

Ta đã biết rằng cả device lẫn driver đều phải được đăng kí với kernel, vậy làm sao để kernel biết device nào được điều khiển bởi driver nào? Câu trả lời là `MODULE_DEVICE_TABLE`. Đây là một macro, khi gọi macro này, driver sẽ tạo ra một bảng ID, bảng này sẽ mô tả những device mà driver hỗ trợ. Cùng lúc đó, nếu driver được biên dịch thành module, ta nên đặt tên module giống trường `driver.name`. Nếu không đặt cùng tên, module sẽ không được tự động load khi driver match với device, trừ khi ta sử dụng macro `MODULE_ALIAS` để thêm các tên khác cho module. Ở thời điểm biên dịch, những thông tin này sẽ được lấy từ driver để tạo thành một bảng các device id. Khi kernel cần tìm driver cho một device, nó chạy vòng lặp hàm match trên các driver có trên bus, hàm sẽ rà soát trong các bảng device của driver. Nếu tìm thấy driver có tên trùng với trường `compatible` (với device tree), `device/vendor id` hoặc `name` (với bảng device ID), thì module của driver tương ứng sẽ được load lên. Macro `MODULE_DEVICE_TABLE` được định nghĩa trong `linux/module.h`:

```
#define MODULE_DEVICE_TABLE(type, name)
```

Với type có thể là i2c, spi, acpi, of, platform, usb, pci hoặc một loại bus khác có trong include/linux/mod\_devicetable.h. Điều này phụ thuộc vào device nằm trên loại đường bus nào, hoặc cơ chế match mà ta muốn dùng. Tham số name là con trỏ đến một mảng XXX\_device\_id. Nếu ta cần match I2C device thì nó sẽ là i2c\_device\_id, hoặc với SPI device thì là spi\_device\_id. Cả i2c\_device\_id và spi\_device\_id đều là match device khi ta define device với platform data trên source code (cách cũ). Nếu muốn match device được định nghĩa trong device tree, ta phải sử dụng of\_device\_id (Open Firmware).

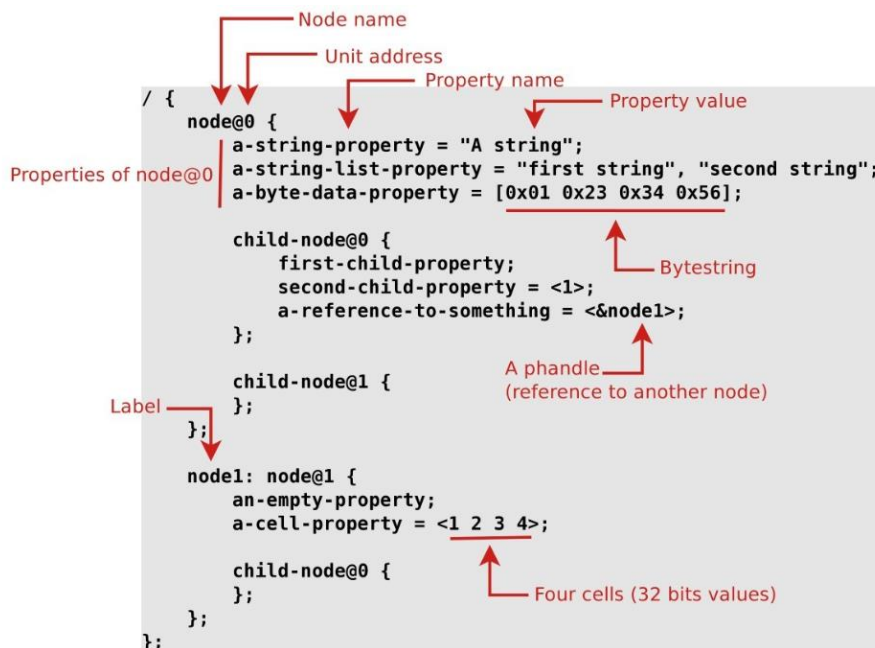
## 2. Lý thuyết về Device Tree

Device Tree là một file dùng để mô tả phần cứng, nó cực kì dễ đọc, chúng ta có thể hình dung device tree giống như một kiểu JSON, nó là một cấu trúc cây nơi mà các thiết bị được đại diện bởi các node mang đặc tính của chính nó. Các đặc tính đó có thể rỗng (chỉ có key, để miêu tả giá trị Boolean) hoặc các cặp key-value nơi mà giá trị có thể chứa một stream byte bất kì.

### 2.1. Cơ chế của Device tree

Ta enable device tree bằng cách đặt option CONFIG\_OF thành Y. Các header cho phép sử dụng các API của DT trong driver là <linux/of.h> và <linux/of\_device.h>.

Ta sẽ có một hình ảnh để mô tả cấu trúc một node trong device tree:



Các kiểu dữ liệu được sử dụng trong device tree bao gồm:

- Chuỗi, nằm trong dấu ngoặc kép. Ta có thể sử dụng dấu phẩy để tạo một danh sách các chuỗi.
- Số nguyên dương 32-bit, nằm trong dấu ngoặc nhọn.
- Dữ liệu boolean, là một thuộc tính trống. Nếu nó được khai báo trong device tree tức giá trị là true, nếu không được khai báo thì tức là false.

**Ta đã hiểu sơ qua về cấu trúc cũng như chức năng của device tree, vậy device tree được sinh ra và hoạt động của nó như thế nào?**

Trước khi chúng ta build một images cho một hệ thống embedded, các thành phần có trên bo mạch sẽ được nhà sản xuất tạo thành các file device tree. Cụ thể là các file DTSI và DTS và được lưu trữ tại 1 vùng nào đó trong project. Ở đây, file DTS dùng để mô tả những phần nằm trên board (ví dụ chân nào được nối với cổng ethernet,...). Trong khi đó DTSI sẽ mô tả các thành phần có trên CPU ( I2C, SPI, ...). Và file DTS sẽ include file DTSI.

Những file device tree này sẽ được gán vào những biến đại diện cho những bo mạch chúng ta có thể sử dụng được. Ta sẽ cần thao tác để kernel hiểu rằng chúng ta đang sử dụng bo mạch nào để nó xuất ra file device tree tương ứng với bo mạch đó.

Sau khi đã build xong images, các file DTS và DTSI sẽ được compile thành các file DTB (là file device tree dành cho kernel), file DTB sẽ được u-boot load vào trong kernel, sau đó kernel sẽ sử dụng DTB để xem trên bo mạch có những thành phần gì.

Đó chính là quá trình device tree được gọi và cách kernel sử dụng device tree trong hệ embedded.

## **2.2. Mối quan hệ giữa Platform device driver và Device tree**

Như đã trình bày ở trên, kể từ phiên bản kernel 4.10, tất cả các thiết bị đều sử dụng device tree để có thể xác định phần cứng. Như vậy có thể khẳng định device tree sẽ gần như luôn luôn xuất hiện.

**Ta đều biết để có thể sử dụng platform device driver, chúng ta sẽ cần đăng kí chúng với các bus, vậy các device hay bus được mô tả ở đâu?**

Chúng sẽ được biểu diễn bên trong device tree dưới dạng các node, ví dụ ta có một thiết bị platform (EEPROM) nằm trên i2c bus nó sẽ được biểu diễn trong device tree như sau:

```
&i2c2 { /* Phandle of the bus node */
pcf8523: rtc@68 {
    compatible = "nxp,pcf8523";
    reg = <0x68>;
};
    eeprom: ee24lc512@55 { /* eeprom device */
        compatible = "packt,ee24lc512";
        reg = <0x55>;
    };
};
```

Như vậy với việc các thông số của device như tên, địa chỉ được mô tả ở device tree, chúng ta sẽ dễ dàng match được driver và device với nhau bằng trường .compatible và biết được device đó nằm trên bus nào để có thể sử dụng type\_bus bên trong macro MODULE\_DEVICE\_TABLE khi cần thông báo sự xuất hiện của device với kernel.

Ta có thể kết luận lại như sau, device tree sẽ là đại diện cho các thông số của các bus và device, và driver sẽ thông qua device tree để match tới các device mà nó cần, cũng như biết được device đó đang nằm trên bus nào.

#### **IV. Phát triển driver điều khiển đèn led hỗ trợ Device Tree dựa trên Platform Driver**

Chúng ta đã cùng xây dựng một hệ thống điều khiển đèn led dựa trên cấu trúc character device và cơ chế hard code (tức là define địa chỉ thanh ghi vào trong source code), điều này sẽ rất phù hợp để cho những người mới tiếp cận, tuy nhiên nó thực sự không tốt mỗi khi ta cần thay đổi địa chỉ, hoặc ví dụ như khi địa chỉ của thanh ghi bị thay đổi, chúng ta sẽ phải ngồi dò lại và chỉnh sửa từng macro, điều này là vô cùng mất thời gian

Vậy nên trong phần này, chúng ta sẽ nâng cấp driver của ta lên để có thể tự động truy xuất được tới địa chỉ thanh ghi và tự động load lên mỗi khi boot kernel.

## 1. Trình bày vấn đề cần giải quyết

Mục tiêu của việc nâng cấp này là sẽ thay đổi phương thức coding cho hệ thống driver của ta, chuyển từ hard code sang hỗ trợ device tree theo cơ chế matching. Từ đó driver có thể tự động load lên kernel mỗi khi boot cho ZCU102 từ đó nó sẽ tự động cập nhật địa chỉ mỗi khi ta thay đổi cấu hình.

## 2. Ý tưởng thực hiện

Ta sẽ giải quyết việc này bằng cách xây dựng driver của ta theo cấu trúc của một Platform driver cộng thêm việc sử dụng kiểu OF match để matching driver với device tree. Như vậy ta sẽ có các bước thực hiện như sau:

- Tạo và đăng kí driver và device với kernel
- Xây dựng các trường của driver (probe, exit,...)
- Matching driver với device node trong device tree.

### 2.1. Tạo và đăng kí driver và device

Như đã đề cập từ trước đó, khi ta sử dụng cấu trúc platform driver, ta cần đăng kí driver và device đó với kernel. Trong project này, ta sẽ sử dụng platform\_driver\_register trong hàm init để đăng kí driver. Cụ thể như sau

```
static int __init drvled_init(void){
    //register platform driver
    platform_driver_register(&test_led_drv);
}
```

Sau khi đã đăng kí driver với kernel, ta sẽ cần thông báo cho kernel biết về các device sẽ được matching với driver. Để làm được việc này, chúng ta sẽ cần tạo một entry device tree ( hàm này sẽ có nhiệm vụ sử dụng trường compatible để matching giữa driver với device node có trong device tree). Cấu trúc của hàm như sau:

```
static struct of_device_id gpio_led[] = {
    { .compatible = "xlnx, led-ip-1.0", },
};
```

Ta có thể thấy trường compatible được gán bằng một chuỗi string, chuỗi string này chính là chuỗi string có trong device tree. Chúng ta sẽ hình dung trường compatible giống như một bên trung gian, nó sẽ đứng ra để kết nối giữa thông tin trong device tree với driver.

Sau khi đã có được entry device tree, tức là chúng ta đã có các trường thông tin về device, tuy nhiên đối với kernel, nó vẫn chưa biết về sự xuất hiện của các device này, vậy nên chúng ta cần thông báo tới kernel về sự có mặt của device. Ta sẽ sử dụng macro

```
MODULE_DEVICE_TABLE(of, gpio_led);
```

Macro này sẽ có nhiệm vụ thông báo về sự xuất hiện của device tới kernel, lúc này kernel sẽ tạo một ID table (device table) để quản lý các device được driver sử hữu. Table này sẽ được giải nén để tạo thành một file trong quá trình biên dịch module. Chúng ta có thể xem các thông tin này ở file module.alias được lưu trong /lib/module/kernel-version/

Cuối cùng, ta sẽ tạo một struct platform\_driver bao gồm các trường thông tin cơ bản như chúng ta đã tìm hiểu ở phần lý thuyết. Cụ thể như sau:

```
static struct platform_driver test_led_drv = {
    .probe = test_led_probe,
    .remove = test_led_remove,
    .driver = {
        .name = "TEST_LED",
        .of_match_table = gpio_led,
    },
};
```

Như vậy, cơ bản chúng ta đã hoàn thành việc tạo và đăng ký driver và device với kernel. Trong phần tiếp theo ta sẽ đi sâu hơn vào các trường thông tin này.

## 2.2. Xây dựng các trường trong driver

Sau khi khai báo và đăng ký thành công driver, ta cần xây dựng hàm probe và remove, cũng như các entry point read write cho driver.

Hãy bắt đầu với hàm probe, đây là hàm được gọi ra khi việc matching driver và device đã hoàn thành, nhiệm vụ của nó là khởi tạo các thông số như device number, tạo device file và cấu trúc cdev cho device. Ngoài ra, ta sẽ sử dụng macro of\_property\_read\_u32\_array để lấy giá trị địa chỉ của device trong device tree đưa chúng vào một mảng, từ đó ta sẽ thao tác với các giá trị bên trong mảng đó. Cụ thể như sau:

```
static int test_led_probe(struct platform_device *pdev){
    int ret = 0;
    struct device_node *np;
    np = pdev->dev.of_node;
    unsigned int doc[4];
    of_property_read_u32_array(pdev->dev.of_node, "reg", doc, 4);
    /* cấp phát quyền truy cập*/
```

```

if(request_mem_region(doc[1], doc[3], "drvled") == NULL);
{
    printk("khong truy cap dcphan cung request_mem_region");
    ret =-EBUSY;
}
/* map phy_add vao virtual_add */
drvled.virtual_add = ioremap(doc[1], doc[3]);
if(drvled.virtual_add == NULL)
{
    printk("khong mp dc dia chi ao vao vat ly");
    ret = -ENOMEM;
}

/*cap phat device number*/
ret = alloc_chrdev_region(&drvled.num, 0, 1, "test-led");
if(ret < 0)
{
    printk("khong tao duoc device number");
}
/* tao device file */
drvled.vclass = class_create(THIS_MODULE, "class_drvled");
if(drvled.vclass == NULL)
{
    printk("tao class tao device file loi");
    class_destroy(drvled.vclass);
}
drvled.vdevice = device_create(drvled.vclass, NULL, drvled.num,
NULL, "drv_led");
if(IS_ERR(drvled.vdevice))
{
    printk("failed device_create");
    device_destroy(drvled.vclass, drvled.num);
}
/* dang ki entry point voi kernel*/
drvled.vcdev = cdev_alloc();
if(drvled.vcdev == NULL)
{
    printk("failed cdev_alloc");
}
cdev_init(drvled.vcdev, &drv_fops);
ret = cdev_add(drvled.vcdev, drvled.num, 1);
if(ret < 0)
{

```



```
        printk("failed cdev_add");
    }

    printk("hello");
    //    drvled_directions();
    drvled_select(1);

    return 0;
}
```

Hàm tiếp theo chúng ta cần xây dựng chính là remove, nhiệm vụ của nó chính là giải phóng những thông số đã được khởi tạo bên trong hàm probe khi device được xóa khỏi driver. Cụ thể như sau:

```
static int test_led_remove(struct platform_device *pdev){
    /*giai phong cdev*/
    cdev_del(drvled.vcdev);
    /*xoa bo nho device file*/
    class_destroy(drvled.vclass);
    device_destroy(drvled.vclass, drvled.num);
    /* giai phong device number */
    unregister_chrdev_region(drvled.num, 1);
    printk("goodbye");
    return 0;
}
```

### 3. Kết quả - Đánh giá – Nhận xét

Ta đã cùng hoàn thành việc nâng cấp driver điều khiển đèn led, công việc tiếp theo sẽ là kiểm tra kết quả xem có như mục tiêu ban đầu hướng tới hay không.

#### 3.1. Kết quả

Ta sẽ thực hiện lệnh echo để truyền giá trị 1 vào device file rồi kiểm tra xem đèn có được bật hay không

```
echo 1 > /dev/drv_led
```

Kiểm tra đèn trên KIT ZCU102, có thể thấy đèn DS44 đã sáng

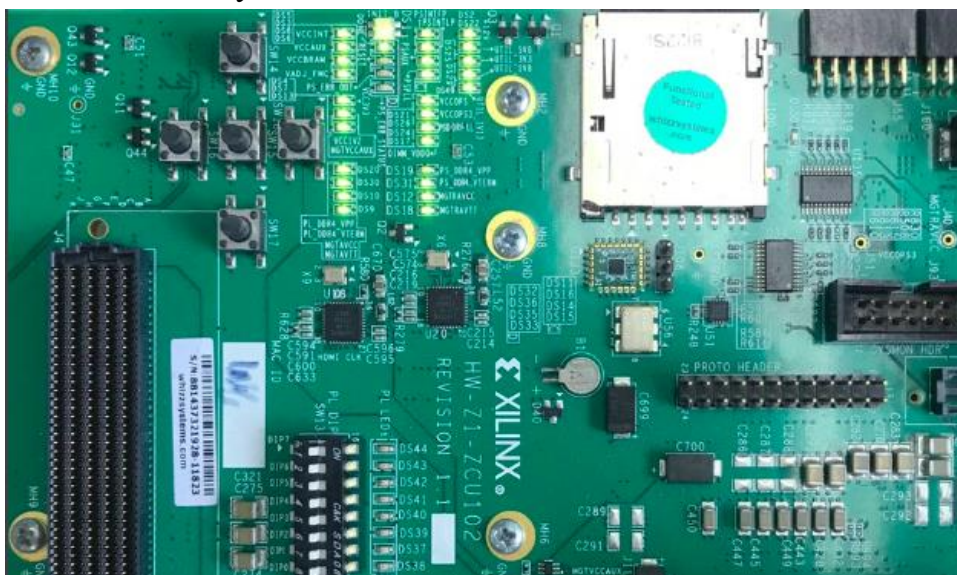


Hình 13. Điều khiển đèn led sáng

Tiếp theo, ta thực hiện tắt đèn

```
echo > 0 /dev/drv_led
```

Kiểm tra có thấy, đèn trên bo mạch đã được tắt



Hình 14. Điều khiển đèn led tối

Như vậy, kết quả đã như mong muốn của ta

### **3.2. Đánh giá – Nhận xét**

Sau khi nâng cấp hệ thống điều khiển đèn led lên hỗ trợ device tree, có thể thấy rằng khi ta boot images cho mạch, driver module sẽ được load lên theo mà không cần phải thực hiện insmod. Như vậy mục đích và ý tưởng ban đầu đề ra đã được hoàn thành. Ta sẽ tổng kết lại về những thứ mới đã được học và thực hành trong phần này:

- Hiểu về 2 khái niệm mới là Platform device driver và Device tree
- Biết được cách match giữa driver tới các device node có trong device tree sử dụng kiểu OF\_match thông qua trường compatible
- Hiểu về luồng hoạt động khi sử dụng driver và device thông qua device tree

Như vậy, ta có thể kết luận lại, thông qua việc nâng cấp hệ thống điều khiển đèn led, chúng ta đã có thêm nhiều kiến thức mới về phần driver. Trong phần tiếp theo, ta sẽ đi sâu hơn vào cụ thể 1 kiểu Platform device và cách viết 1 device node sẽ là như thế nào.

## **V. I2C Client Driver**

### **1. Tìm hiểu lý thuyết**

#### **1.1. Tổng qua về giao thức I2C**

#### **1.2. Kiến trúc I2C driver**

#### **1.3 I2C driver**

#### **1.4. I2C client**

#### **1.5. Giao tiếp giữa driver với client**

#### **1.5. I2C và Device Tree**