

# **ZYNQ MPSoC Development Platform**

## **Linux Driver Tutorial**



## Version Record

Version	Date	Release By	Description
Rev1.03	2022-04-30	Rachel Zhou	First Release

The ALINX official Documents are in Chinese, and the English version was translated by Shanghai Tianhui Trading Company. They has not been officially Review by ALINX and are for reference only. If there are any errors, please send email feedback to [support@aithtech.com](mailto:support@aithtech.com) for correction.

Amazon Store: <https://www.amazon.com/alinx>

Aliexpress Store:

<https://alinxfpga.aliexpress.com/store/911112202?spm=a2g0o.detail.1000007.1.704e2bedqLBW90>

Ebay Store: <https://www.ebay.com/str/alinxfpga>

Customer Service Information

Skype: [rachelhust@163.com](mailto:rachelhust@163.com)

Wechat: [rachelhust](#)

Email: [support@aithtech.com](mailto:support@aithtech.com) and [rachehust@163.com](mailto:rachehust@163.com)

# Content

Version Record.....	2
Content.....	3
Part 1: Character Device.....	13
Part 1.1: Linux Driver Introduction.....	13
Part 1.1.1: The Relationship Between Linux Application and Driver.....	13
Part 1.1.2: Linux Driver Classification.....	15
Part 1.2: Development Steps of Character.....	16
Part 1.2.1: Implement System Calls.....	17
Part 1.2.2: Loading and unloading of drive modules....	18
Part 1.2.3: Registration and Cancellation of Character Device.....	18
Part 1.2.4: Linux Device Number.....	19
Part 1.2.5: Implement Device Operation Functions.....	19
Part 1.2.6: Add Driver Description Information.....	20
Part 1.3: Character Device Driver Development.....	21
Part 1.3.1: Hardware Schematic and Data Sheet.....	21
Part 1.3.2: Writing a Character Device Driver.....	23
Part 1.3.3: Add a New Driver in the Petalinux Custom System.....	27
Part 1.3.4: Use QT in Ubuntu Virtual Machine to Write Test APP.....	30
Part 1.3.5: Run test.....	35
Part 2: A New Way of Writing Character Devices.....	38
Part 2.1: Device Number Management.....	38
Part 2.2: New Registration Method.....	40

Part 2.2.1: Character Device Data Structure.....	40
Part 2.2.2: cdev Structure Initialization.....	40
Part 2.2.3: New Registration and Deregistration Functions.....	41
Part 2.3: Automatically Create Device Files.....	42
Part 2.3.1: mdev.....	42
Part 2.3.2: Class Creation and Deletion.....	42
Part 2.3.3: Creation and Deletion of Device Nodes.....	42
Part 2.4: Character Device New Driver Experiment.....	44
Part 2.4.1: Hardware Schematic and Data Sheet.....	44
Part 2.4.2: Writing a Character Device Driver.....	44
Part 2.4.3: Write Test APP.....	48
Part 2.4.4: Run Test.....	48
Part 3: Device Tree and of Function.....	50
Part 3.1: Device Tree.....	50
Part 3.1.1: DTS, DTB and DTSL.....	51
Part 3.1.2: node.....	52
Part 3.1.3: attributes.....	53
Part 3.1.4: view the device tree in the file system.....	56
Part 3.1.5: modify the device tree.....	56
Part 3.2: of Function.....	57
Part 3.2.1: of function to find node.....	57
Part 3.2.2: of function to extract attributes.....	59
Part 3.3: led Drive Experiment under the Device Tree.....	61
Part 3.3.1: Schematic.....	61
Part 3.3.2: Modify the Device Tree.....	61
Part 3.3.3: Driver.....	62
Part 3.3.4: test program.....	66
Part 3.3.5: run test.....	66

---

Part 4: pinctrl and gpio Subsystem.....	68
Part 4.1: pinctrl Subsystem.....	68
Part 4.1.1: Device Tree Nodes and Attributes of pinctrl Subsystem of zynq Platform.....	69
Part 4.1.2: Use of pinctrl Subsystem.....	73
Part 4.2: gpio Subsystem.....	73
Part 4.2.1: gpio in the device tree.....	73
Part 4.2.2: gpio interface function.....	74
Part 4.3: Experiment.....	77
Part 4.3.1: Schematic.....	77
Part 4.3.2: Device tree.....	77
Part 4.3.3: Driver code.....	78
Part 4.3.4: Test code.....	82
Part 4.3.3: Run test.....	82
Part 5: Concurrent Processing.....	84
Part 5.1: Concurrency in Linux.....	84
Part 5.2: Linux concurrent processing.....	84
Part 5.2.1: Atomic operations.....	84
Part 5.2.2: Lock mechanism.....	86
Part 5.2.3: Semaphore.....	88
Part 5.3: Experiment.....	89
Part 5.3.1: Schematic.....	89
Part 5.3.2: Device tree.....	89
Part 5.3.3: Driver Code.....	89
Part 5.3.4: Test code.....	96
Part 5.3.5: Run test.....	97
Part 6: gpio input.....	99
Part 6.1: Schematic.....	99
Part 6.2: Device tree.....	100

---

Part 6.3: Driver code.....	101
Part 6.4: Test code.....	104
Part 6.5: Run test.....	105
Part 7: Timer.....	107
Part 7.1: Timers in the Linux kernel.....	107
Part 7.1.1: Timers in the Linux kernel.....	107
Part 7.1.2: Number of beats.....	109
Part 7.1.3: Kernel timer and related functions.....	110
Part 7.2: Experiment.....	112
Part 7.2.1: Schematic.....	112
Part 7.2.2: Device tree.....	112
Part 7.2.3: Driver code.....	112
Part 7.2.4: Test code.....	116
Part 7.2.5: Run test.....	117
Part 8: Interrupt.....	119
Part 8.1: Linux Interrupt Framework Introduction.....	119
Part 8.1.1: Interface function.....	119
Part 8.1.2: The Bottom Half of Linux.....	122
Part 8.1.3: Interrupts in the device tree.....	126
Part 8.2: Experiment.....	128
Part 8.2.1: Schematic.....	128
Part 8.2.2: Device tree.....	128
Part 8.2.3: Driver code.....	129
Part 8.2.4: Test code.....	133
Part 8.2.5: Run test.....	133
Part 9: Blocking IO.....	135
Part 9.1: Blocking and Non-blocking, Synchronous and Asynchronous, and IO operation.....	135
Part 9.2: Blocking IO.....	136

---

Part 9.2.1: Waiting queue.....	136
Part 9.3: Experiment.....	139
Part 9.3.1: Schematic.....	140
Part 9.3.2: Device tree.....	140
Part 9.3.3: Driver code.....	140
Part 9.3.4: Test code.....	146
Part 9.3.5: Run test.....	146
Part 10: Non-Blocking IO.....	148
Part 10.1: NIO in Linux.....	148
Part 10.1.1: Polling method in the application.....	148
Part 10.1.2: Poll function in the driver.....	152
Part 10.2: Experiment.....	153
Part 10.2.1: Schematic.....	153
Part 10.2.2: Device tree.....	153
Part 10.2.3: Driver code.....	153
Part 10.2.4: Test code.....	159
Part 10.2.5: Run test.....	160
Part 11: Asynchronous IO.....	162
Part 11.1: Asynchronous IO in Linux.....	162
Part 11.1.1: Signal.....	162
Part 11.1.2: Use of signals in applications.....	163
Part 11.1.3: Realization of signal in driver.....	164
Part 11.2: Experiment.....	165
Part 11.2.1: Schematic.....	165
Part 11.2.2: Device tree.....	166
Part 11.2.3: Driver code.....	166
Part 11.2.4: Test code.....	171
Part 11.2.5: Run test.....	173
Part 12: Platform.....	175

---

Part 12.1: Drive Separation.....	175
Part 12.2: Platform Model.....	177
Part 12.2.1: platform_bus.....	178
Part 12.2.2: platform_driver.....	179
Part 12.2.3: platform_device.....	181
Part 12.3: Experiment.....	182
Part 12.3.1: Schematic.....	182
Part 12.3.2: Device tree.....	183
Part 12.3.3: Driver code.....	183
Part 12.3.4: Test code.....	189
Part 12.3.5: Run test.....	189
Part 13: Platform and Device Tree.....	191
Part 13.1: Platform under the device tree.....	191
Part 13.2: Experiment.....	192
Part 13.2.1: Schematic.....	192
Part 13.2.2: Device tree.....	192
Part 13.2.3: Driver code.....	193
Part 13.2.4: Test code.....	197
Part 13.2.5: Run Test.....	198
Part 14: MISC device driver.....	199
Part 14.1: Introduction to MISC device.....	199
Part 14.2: Use of MISC framework.....	200
Part 14.3: Experiment.....	202
Part 14.3.1: Schematic.....	202
Part 14.3.2: Device tree.....	202
Part 14.3.3: Driver code.....	202
Part 14.3.4: Test code.....	207
Part 14.3.5: Run Test.....	207
Part 15: Input Subsystem.....	209

---

Part 15.1: Introduction to the Input Subsystem.....	209
Part 15.1.1: The Use of Input Subsystem to Drive the Framework.....	210
Part 15.1.2: Use of input subsystem application.....	214
Part 15.3: Experiment.....	214
Part 15.3.1: Schematic.....	214
Part 15.3.2: Device tree.....	214
Part 15.3.3: Driver code.....	215
Part 15.3.4: Test code.....	219
Part 15.3.5: Run Test.....	221
Part 16: pwm Drive.....	224
Part 16.1: pwm Implementation on zynq.....	224
Part 16.1.1: Modify the Vivado Project.....	224
Part 16.1.2: Method of Controlling pwm Output.....	226
Part 16.2: Pwm framework in Linux.....	228
Part 16.2.1:pwm_device.....	229
Part 16.2.2.....	231
Part 16.2.3: Example.....	233
Part 16.3: Experiment.....	235
Part 16.3.1: Schematic.....	236
Part 16.3.2: Device tree.....	236
Part 16.3.3: Driver code.....	236
Part 16.3.4: Test code.....	240
Part 16.3.5: Run Test.....	241
Part 17: I2C Driver.....	243
Part 17.1: I2C Driver Framework.....	243
Part 17.1.1: I2C controller driver.....	243
Part 17.1.2: I2C Device Driver.....	245
Part 17.1.3: I2C device driver implementation process	248

---

Part 17.3: Experiment.....	250
Part 17.3.1: Schematic.....	250
Part 17.3.2: Device Tree.....	252
Part 17.3.3: driver.....	252
Part 17.3.4: Test Code.....	257
Part 17.3.5: Run Test.....	258
Part 18: USB Driver.....	260
Part 18.1: USB Recognition Process.....	260
Part 18.2: USB Transmission.....	261
Part 18.2.1: USB Transmission Mode.....	261
Part 18.2.2: Endpoint.....	262
Part 18.2.3: Pipe.....	262
Part 18.3: USB bus driver.....	263
Part 18.4: USB device driver.....	264
Part 18.5: urb request block.....	266
Part 18.6: Experiment.....	269
Part 18.6.1: Schematic.....	269
Part 18.6.2: Device tree.....	269
Part 18.6.3: Driver code.....	269
Part 18.6.4: Run Test.....	272
Part 19: SPI Drive.....	277
Part 19.1: SPI Controller Drive.....	277
Part 19.2: SPI Device Drive.....	278
Part 19.3.....	281
Part 19.4: Experiment.....	283
Part 19.4.1: Schematic.....	283
Part 19.4.2: Device tree.....	284
Part 19.4.3: Driver code.....	285
Part 19.4.4: Test code.....	291

---

Part 19.4.5: Run Test.....	291
Part 20: Uart Driver.....	293
Part 20.1: uart Driver Framework.....	293
Part 20.2: uart Driver in xilinx.....	297
Part 21: Block Device Driver.....	302
Part 21.1: Introduction to Block Devices.....	302
Part 21.2: Block device access in the kernel.....	303
Part 21.3: Block Device Framework.....	305
Part 21.3.1: gendisk structure.....	306
Part 21.3.2: request_queue.....	308
Part 21.4: Experiment.....	308
Part 21.4.1: Driver code.....	308
Part 21.4.2: Run Test.....	313
Part 22:NIC driver.....	315
Part 22.1: Linux Network Card Driver.....	315
Part 22.1.1: Initialization.....	316
Part 22.1.2: Transmit Packet.....	319
Part 22.1.3: Receive Packet.....	321
Part 22.2: Experiment.....	322
Part 22.2.1: Driver code.....	322
Part 22.2.2: Run Test.....	325
Part 23: DMA Driver.....	327
Part 23.1: DMA Introduction.....	327
Part 23.2: DMAC.....	327
Part 23.2.1: DAM Channels.....	327
Part 23.2.2: DAM Request Lines.....	328
Part 23.2.3: Transfer size, Transferwide,Burst size... ..	328
Part 23.2.4: scatter-gather.....	329
Part 23.3: DMA in Linux.....	329

---

Part 23.3.1: DMA Controller Driver Framework.....	330
Part 23.4.2: DMA Client Driver Framework.....	336
Part 24: Multi-touch screen driver.....	343
Part 24.1: Multi-touch Screen in Input Subsystem.....	343
Part 24.2: Multi-touch screen and SoC interface.....	344
Part 24.3: vivado project and petalinux.....	345
Part 24.4: Schematic.....	346
Part 24.5: Device Tree.....	346
Part 24.6: Multi-touch Screen Driver.....	347
Part 25: LCD Drive.....	355
Part 25.1: framebuffer frame.....	355
Part 25.2: VDMA.....	357
Part 25.3: DRM framework.....	359

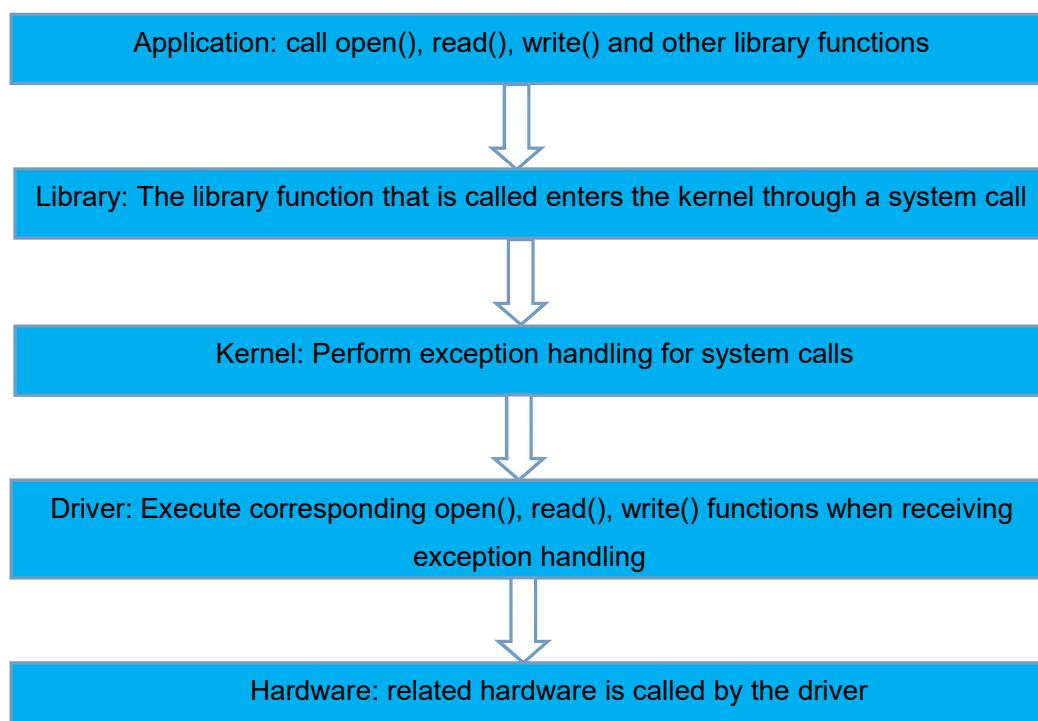
# Part 1: Character Device

## Part 1.1: Linux Driver Introduction

Before writing the program, learn about basic Linux device and driver related knowledge.

### Part 1.1.1: The Relationship Between Linux Application and Driver

The steps for a Linux application to call the driver are roughly as follows:



- 1) The application calls the open() function provided by the library function to open a device file
- 2) The library causes a CPU exception based on the input parameters of the open() function and enters the kernel
- 3) The kernel's exception handling function finds the corresponding driver according to the input parameters, returns the file handle to the library, and the library function returns to the application

- 4) The application program then uses the obtained file handle to call write(), read() and other functions to issue control instructions
- 5) The library causes a CPU exception based on the input parameters of functions such as write() and read(), and enters the kernel
- 6) The kernel's exception handling function calls the corresponding driver according to the input parameters to perform the corresponding operation

The device file opened by “**open()**” in step 1) is the relevant file generated in the directory “**/dev**” after the driver is successfully loaded, and is the entry point for the application to call the corresponding hardware.

In this process, the open(), read(), write() and other interface functions provided by the library involved in the application become system calls. They execute a certain instruction to cause an exception to enter the kernel, which is the communication between the application and the operating hardware. After the application executes the system call, it will eventually use the corresponding function in the driver function, and the functions such as open(), read(), write() in the driver function need to be implemented by the driver developer.

The application program runs in user space, and the driver program runs in kernel space. The Linux system can restrict the application program to run in a certain memory block through the MMU to avoid the application program error causing the entire system to crash. The driver program running in the kernel space is a part of the system, and driver errors may affect the entire system, so be careful when implementing the driver.

### Part 1.1.2: Linux Driver Classification

Drivers can be divided into three categories: character device drivers, block device drivers, and network device drivers

Character devices are devices accessed by bytes. For example, a serial port that receives data with one byte is a typical character device. There is also a simpler LED light driver that we will talk about next, and the relatively complex ones such as audio are all character devices, and character devices account for the largest proportion of Linux peripherals.

The characteristic of block devices is that they access data in a certain format, and the specific format is determined by the file system. They usually focus on storage devices, such as EMMC, FLASH, SD card, EEPROM, etc. The characteristics of storage devices are based on storage blocks, so they are named block devices.

For applications, the access methods of block devices and character devices are basically the same, that is, the driver implements system calls, and the application calls the corresponding library functions through device files (such as /dev/mtd0, /dev/i2c-0, etc.) . Of course, this requires the driver developer to ensure that when the block device driver is implemented, the data is organized into blocks in a certain format before reading and writing. In addition, the block device also needs to provide some interfaces to other parts of the kernel, so that the block device can be stored on the file system to mount the block device. Therefore, the realization of the block device driver is much more complicated than the character device driver.

Network devices are special. The communication between application programs and network device drivers is completely

different from the above two devices. A set of data packet transfer functions provided by the library and kernel replace functions such as open(), read(), and write(). However, network devices are also easy to understand and identify. WIFI or wired interfaces are all network devices.

In the device driver, there will also be a situation where a driver contains multiple device drivers, such as FLASH (block device) connected to SPI (character device), WIFI (network device) with USB interface (character device). Next, we will learn the drivers of various devices step by step, starting with the simplest character devices.

## Part 1.2: Development Steps of Character

The Linux driver has a framework. The existing framework is convenient for us to add new drivers. After familiarizing with the existing framework, we can get the new device driver by slightly modifying the existing driver, and learning the existing framework is one of our goals.

After being familiar with the framework, the steps to implement device drivers are roughly as follows:

- 1) Check the schematic diagram and data sheet to understand the device operation method
- 2) Modify the device tree file
- 3) Apply a framework similar to the device, or find the driver code of a similar device in the kernel, and modify it directly to realize driver initialization and operation functions
- 4) Compile the driver into the kernel or separately compile and load the driver
- 5) Write application test driver

It is not difficult to see that the key and difficult point in the above

steps is step 3), below we describe in detail what needs to be implemented in step 3) in the character device driver.

### Part 1.2.1: Implement System Calls

System calls are device operation functions, which are the core of character device drivers. The data structure “file\_operations” is defined in the kernel file “include/linux/fs.h”, which includes all kernel driver operation functions, and the content is as follows:

```
1. struct file_operations {
2.     struct module *owner;
3.     loff_t (*llseek) (struct file *, loff_t, int);
4.     ssize_t (*read) (struct file *, char_user *, size_t, loff_t *);
5.     ssize_t (*write) (struct file *, const char_user *, size_t, loff_t *);
6.     ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
7.     ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
8.     int (*iterate) (struct file *, struct dir_context *);
9.     int (*iterate_shared) (struct file *, struct dir_context *);
10.    unsigned int (*poll) (struct file *, struct poll_table_struct *);
11.    long (*unlock_ioctl) (struct file *, unsigned int, unsigned long);
12.    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
13.    int (*mmap) (struct file *, struct vm_area_struct *);
14.    int (*open) (struct inode *, struct file *);
15.    int (*flush) (struct file *, fl_owner_t id);
16.    int (*release) (struct inode *, struct file *);
17.    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
18.    int (*fasync) (int, struct file *, int);
19.    int (*lock) (struct file *, int, struct file_lock *);
20.    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
21.    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
22.                                       unsigned long);
23.    int (*check_flags)(int);
24.    int (*flock) (struct file *, int, struct file_lock *);
25.    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsign
26.                           ed int);
26.    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsign
27.                           ed int);
27.    int (*setlease)(struct file *, long, struct file_lock **, void **);
28.    long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
29.    void (*show_fdinfo)(struct seq_file *m, struct file *f);
30.    #ifndef CONFIG_MMU
31.        unsigned (*mmap_capabilities)(struct file *);
32.    #endif
33.    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *, loff_t, size_t, unsign
34.                             ed int);
35.    int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t, u64);
36.    ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *, u64);
37. }
```

Briefly introduce the members and functions that will be used in the next experiment:

The member variable **owner** is generally set to **THIS\_MODULE**.

The **open()** function has been mentioned many times and is used to open device files.

The `write()` function is used to write or send data to the device.

The `read()` function is used to read data from the device.

The `release()` function is used to close the device file, which corresponds to the `close()` function in the application.

### Part 1.2.2: Loading and unloading of drive modules

After the driver is completed, you can choose to compile it into the kernel, or use the command “`insmod`” to load it into the system after being compiled separately. The latter is much more convenient for driver development. After the driver is compiled separately, you will get a driver module file with a suffix of “`.ko`”. For example, to load a driver module named “`led.ko`” , just use the command “`insmod led.ko`” . Of course, when there is loading, there is unloading. The unloading command is “`rmmod`” . Unloading the driver is slightly different from loading the driver, not “`rmmod led.ko`” . You can use the “`lsmod`” command to view the loaded driver first, then use the “**rmmod command to add a space**” and add the `lsmod` command to list the driver name you want to uninstall.

When implementing the driver, you need to register two operation functions, the registration function is:

`module_init(xxx_init); //register module loading function`

`module_exit(xxx_exit); //register module uninstall function`

`xxx_init` is the driver entry function, and `xxx_exit` is the driver exit function. When the `insmod` command is executed, the `xxx_init` function will be called. The `xxx_exit` function is the same. The two function prototypes are also very simple, and we will talk about them when we code later.

### Part 1.2.3: Registration and Cancellation of Character Device

The driver entry function is the first function executed when the

---

driver is loaded. We need to seize this opportunity to initialize the driver in the entry function, that is, register the character device. The functions for registering and deregistering character devices are

```
int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops);  
void unregister_chrdev(unsigned int major, const char *name);
```

The two functions involve a total of three input parameters:

**major:** major device number, each device under the linux system has a device number, and there is a **minor** device number under the major device

**name:** device name;

**fops:** file\_operations type pointer, device operation function collection variables

#### Part 1.2.4: Linux Device Number

Each device file has a **major** and **minor** device number. The major device number is unique. Each **major** device has a minor device number **minor**. The minor device number is also unique under this major device. Enter “ **cat /proc/devices** ” command under Linux system to view the registered major device number.

#### Part 1.2.5: Implement Device Operation Functions

In the next experiment to light up the led, we will use those device operation functions. Let's take a look at the steps to light up the led first.

First initialize the IO connected to the led, enable the clock, set the IO as output, etc. The **open()** function is the beginning of the device file associated with the application, and the initialization of the device can be placed in the open() function. Realizing the open() function must implement the corresponding **release()** function. After the IO initialization is completed, we need to light up the led, which is

actually to write the corresponding value into the register corresponding to the IO, and write the function `write()`. Correspondingly, the `read()` function can read the current state of IO.

### Part 1.2.6: Add Driver Description Information

In the driver code, we need to add driver description information such as `LICENSE` information and author information, where `LICENSE` must be added, the method is as follows:

```
MODULE_LICENSE("GPL");
```

Now that we know the registration driver entry and exit functions, the registration and deregistration of character devices, and the device operation functions that need to be implemented, the following framework can be roughly written:

```
1. /* 驱动名称 */
2. #define DEVICE_NAME      "gpio_leds"
3. /* 驱动主设备号 */
4. #define GPIO_LED_MAJOR    200
5.
6. /* open 函数实现, 对应到 Linux 系统调用函数的 open 函数 */
7. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
8. {
9.     return 0;
10. }
11.
12. /* write 函数实现, 对应到 Linux 系统调用函数的 write 函数 */
13. static ssize_t gpio_leds_write(struct file *file_p, const char_user *buf, size_t len, loff_t *loff_t_p)
14. {
15.     return 0;
16. }
17.
18. /* release 函数实现, 对应到 Linux 系统调用函数的 close 函数 */
19. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
20. {
21.     return 0;
22. }
23.
24. /* file_operations 结构体声明, 是上面 open、write 实现函数与系统调用函数对应的关键 */
25. static struct file_operations gpio_leds_fops = {
26.     .owner    = THIS_MODULE,
27.     .open     = gpio_leds_open,
28.     .write    = gpio_leds_write,
29.     .release  = gpio_leds_release,
30. };
31.
32. /* 模块加载时会调用的函数 */
33. static int __init gpio_led_init(void)
34. {
35.     int ret;
36.
37.     /* 通过模块主设备号、名称、模块带有的功能函数(及 file_operations 结构体)来注册模块 */
38.     ret = register_chrdev(GPIO_LED_MAJOR, DEVICE_NAME, &gpio_leds_fops);
39.     if (ret < 0)
40.     {
41.         return ret;
```

```
42.     }
43.     else
44.     {
45.
46.     }
47.     return 0;
48. }
49.
50. /* 卸载模块 */
51. static void_exit gpio_led_exit(void)
52. {
53.     /* 注销模块，释放模块对这个设备号和名称的占用 */
54.     unregister_chrdev(GPIO_LED_MAJOR, DEVICE_NAME);
55. }
56.
57. /* 注册模块入口和出口函数 */
58. module_init(gpio_led_init);
59. module_exit(gpio_led_exit);
60.
61. /* 添加 LICENSE 信息 */
62. MODULE_LICENSE("GPL");
```

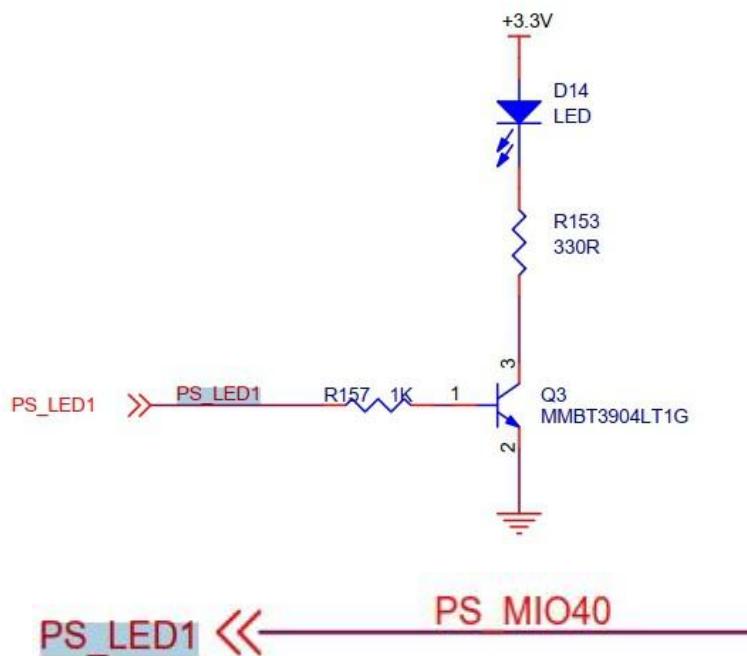
Here, I named the driver "gpio\_leds". The device number is set to 200, my device number 200 is not occupied, the actual experiment should be changed according to the actual situation.

## Part 1.3: Character Device Driver Development

Now that we have a general understanding of the implementation of character devices, we will implement it step by step. Write the device driver for the led "PS LED" on the FPGA development board, and the led can be turned on and off through the driver

### Part 1.3.1: Hardware Schematic and Data Sheet

Open the Schematic of the AU3EG development board and check the connection method of the PS LED.



It can be seen that the target led is connected to the MIO40 pin on the PS side of the development board.

Then open the register manual of Ultra scale+ "ug1087-zynq-ultrascale-registers" (it is an HTML file package, you can find it by searching ug1087 on the xilinx official website). Search for the keyword MIO and find GPIO Module.

Module Name	GPIO Module
Modules of this Type	GPIO
Base Address	0xFF0A0000 (GPIO)
Description	GPIO Controller

The base address of the GPIO register is 0xFF0A0000. To control the GPIO, we need three steps: enable, set the direction, and control the output. MIO40 is in GPIO BANK1. Correspondingly find the enable register OEN\_1: 0xFF0A0248, the direction register DIRM\_1: 0xFF0A0244, and the control register DATA\_1: 0xFF0A0044.

### Part 1.3.2: Writing a Character Device Driver

Above we have got the general framework of the led driver, the next step is to set the relevant registers in the corresponding function. Realize the led initialization in the open() function, realize the control of the led in the write() function, and complete the enabling of the led in the release() function. The final code is as follows:

```

1. /**
2.  *Author : ALINX Electronic Technology (Shanghai) Co., Ltd.
3.  *Website: http://www.alinx.com
4.  *Address: Room 202, building 18,
5.  No.518 xinbrick Road,
6.  Songjiang District, Shanghai
7.  *Created: 2020-3-2
8.  *Version: 1.0
9.  ** ===== */
10.
11. #include <linux/module.h>
12. #include <linux/kernel.h>
13. #include <linux/fs.h>
14. #include <linux/init.h>
15. #include <linux/ide.h>
16. #include <linux/types.h>
17.
18. /* 驱动名称 */
19. #define DEVICE_NAME      "gpio_leds"
20. /* 驱动主设备号 */
21. #define GPIO_LED_MAJOR    200
22.
23. /* gpio 寄存器虚拟地址 */
24. static unsigned long gpio_add_minor;
25. /* gpio 寄存器物理基地址 */
26. #define GPIO_BASE          0xFF0A0000
27. /* gpio 寄存器所占空间大小 */
28. #define GPIO_SIZE          0x1000
29. /* gpio 方向寄存器 */
30. #define GPIO_DIR_M_1        (unsigned int*)(0x000000000000244 + (unsigned long)gpio_add_minor)
31. /* gpio 使能寄存器 */
32. #define GPIO_OEN_1          (unsigned int*)(0x000000000000248 + (unsigned long)gpio_add_minor)
33. /* gpio 控制寄存器 */
34. #define GPIO_DATA_1         (unsigned int*)(0x0000000000000044 + (unsigned long)gpio_add_minor)
35.
36. /* open 函数实现, 对应到 Linux 系统调用函数的 open 函数 */
37. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
38. {
39.     printk("gpio_test module open\n");
40.
41.     return 0;
42. }
43.
44.
45. /* write 函数实现, 对应到 Linux 系统调用函数的 write 函数 */
46. static ssize_t gpio_leds_write(struct file *file_p, const char user *buf, size_t len, loff_t *loff_t_p)
47. {
48.     int rst;
49.     char writeBuf[5] = {0};
50.
51.     printk("gpio_test module write\n");
52.
53.     rst = copy_from_user(writeBuf, buf, len);
54.     if(0 != rst)
55.     {

```

```
56.         return -1;
57.     }
58.
59.     if(1 != len)
60.     {
61.         printk("gpio_test len err\n");
62.         return -2;
63.     }
64.     if(1 == writeBuf[0])
65.     {
66.         *GPIO_DATA_1 |= 0x00004000;
67.         printk("gpio_test ON *GPIO_DATA_1 = 0x%X\r\n", *GPIO_DATA_1);
68.     }
69.     else if(0 == writeBuf[0])
70.     {
71.         *GPIO_DATA_1 &= 0xFFFFBFFF;
72.         printk("gpio_test OFF *GPIO_DATA_1 = 0x%X\r\n", *GPIO_DATA_1);
73.     }
74.     else
75.     {
76.         printk("gpio_test para err\n");
77.         return -3;
78.     }
79.
80.     return 0;
81. }
82.
83. /* release 函数实现, 对应到 Linux 系统调用函数的 close 函数 */
84. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
85. {
86.     printk("gpio_test module release\n");
87.     return 0;
88. }
89.
90. /* file_operations 结构体声明, 是上面 open、write 实现函数与系统调用函数对应的关键 */
91. static struct file_operations gpio_leds_fops =
92. {
93.     .owner    = THIS_MODULE,
94.     .open     = gpio_leds_open,
95.     .write    = gpio_leds_write,
96.     .release  = gpio_leds_release,
97. };
98. /* 模块加载时会调用的函数 */
99. static int init gpio_led_init(void)
100. {
101.     int ret;
102.
103.     /* 通过模块主设备号、名称、模块带有的功能函数(及 file_operations 结构体)来注册模块 */
104.     ret = register_chrdev(GPIO_LED_MAJOR, DEVICE_NAME, &gpio_leds_fops);
105.     if (ret < 0)
106.     {
107.         printk("gpio_led_dev_init_ng\n");
108.         return ret;
109.     }
110.     else
111.     {
112.         /* 注册成功 */
113.         printk("gpio_led_dev_init_ok\n");
114.         /* 把需要修改的物理地址映射到虚拟地址 */
115.         gpio_add_minor = ioremap_wc(GPIO_BASE, GPIO_SIZE);
116.         printk("gpio_add_minor = 0x%lx\n", gpio_add_minor);
117.         printk("GPIO_DIRM_1    = 0x%lx\n", (unsigned long)GPIO_DIRM_1);
118.         printk("GPIO_OEN_1      = 0x%lx\n", (unsigned long)GPIO_OEN_1);
119.
120.         /* MIO_0 设置成输出 */
121.         *GPIO_DIRM_1 |= 0x00004000;
122.         /* MIO_0 使能 */
123.         *GPIO_OEN_1 |= 0x00004000;
124.
125.         printk("*GPIO_DIRM_1    = 0x%lx\n", *GPIO_DIRM_1);
126.         printk("*GPIO_OEN_1      = 0x%lx\n", *GPIO_OEN_1);
127.     }
128.     return 0;
129. }
130.
131. /* 卸载模块 */
132. static void exitgpio_led_exit(void)
133. {
```

```
134.     *GPIO_OEN_1 &= 0xFFFFBFFF;
135.
136.     /* 释放对虚拟地址的占用 */
137.     iounmap(gpio_add_minor);
138.     /* 注销模块，释放模块对这个设备号和名称的占用 */
139.     unregister_chrdev(GPIO_LED_MAJOR, DEVICE_NAME);
140.
141.     printk("gpio_led_dev_exit_ok\n");
142. }
143.
144. /* 标记加载、卸载函数 */
145. module_init(gpio_led_init);
146. module_exit(gpio_led_exit);
147.
148. /* 驱动描述信息 */
149. MODULE_AUTHOR("Alinx");
150. MODULE_ALIAS("gpio_led");
151. MODULE_DESCRIPTION("GPIO LED driver");
152. MODULE_VERSION("v1.0");
153. MODULE_LICENSE("GPL");
```

The `printf()` function that appears on line 39 is a function that outputs a string to the console in the kernel mode. It is used here for debugging, which is equivalent to `printf()` in the application. The `printf()` function has a message level, which is defined in the header file `include/linux/kern_levels.h`, as follows:

```
1. #define KERN_EMERG KERN_SOH "" /* system is unusable */
2. #define KERN_ALERT KERN_SOH "" /* action must be taken immediately */
3. #define KERN_CRIT KERN_SOH "" /* critical conditions */
4. #define KERN_ERR KERN_SOH "" /* error conditions */
5. #define KERN_WARNING KERN_SOH "" /* warning conditions */
6. #define KERN_NOTICE KERN_SOH "" /* normal but significant condition */
7. #define KERN_INFO KERN_SOH "" /* informational */
8. #define KERN_DEBUG KERN_SOH "" /* debug-level messages */
```

Among them, 0 has the highest priority and 7 has the lowest priority. If you want to set the message level, you can set it as follows:

```
printf(KERN_INFO"gpio_test module open\n");
```

If the message level is not set, then `printf()` will use the default level `MESSAGE_LOGLEVEL_DEFAULT`, which is 4. Only the message level is greater than the macro `CONSOLE_LOGLEVEL_DEFAULT` defined in the header file `include/linux/printk.h`, the message will be printed, and the value of `CONSOLE_LOGLEVEL_DEFAULT` is 7.

The function `ioremap_wc()` in line 115 is used to map a physical address to a virtual address. In Linux, due to the MMU memory mapping relationship, we cannot directly manipulate the physical

address, but need to map the physical address to the virtual address and then manipulate the corresponding virtual address. Note that AU3EG is a 64-bit SoC. The linux system we are running here is also 64-bit. `ioremap_wc()` is used for 64-bit systems and `ioremap()` is used for 32-bit systems. The pointer size is also 64 bits, you need to use the `unsigned long` type to get.

`ioremap()` is defined in the header file `arch/arm/include/asm/io.h`, as follows:

```
#define ioremap(cookie,size)_arm_ioremap((cookie),(size),MT_DEVICE)
```

**Cookie** refers to the physical address, and **size** is the length of the address to be mapped. The 115 line of code is the base address of the GPIO register, the range is that the entire GPIO register is mapped to the virtual address and assigned to the global variable `gpio_add_minor`, and then the variable `gpio_add_minor` can be read and written directly.

The `iounmap()` function on line 137 is the virtual address release function opposite to `ioremap()`. The input parameter is the first address of the virtual address returned by the `ioremap()` function, such as `gpio_add_minor` here.

The rest is register operations. For the operation of virtual addresses, we all directly access them through pointers in our example, but Linux has a recommended method of reading and writing instead of using pointers, as follows:

Read Function:

```
u8 readb(const volatile void_iomem *addr);  
u16 readw(const volatile void_iomem *addr);  
u32 readl(const volatile void_iomem *addr);
```

Write Function:

```
void writeb(u8 value, volatile void_iomem *addr);  
void writew(u16 value, volatile void_iomem *addr);
```

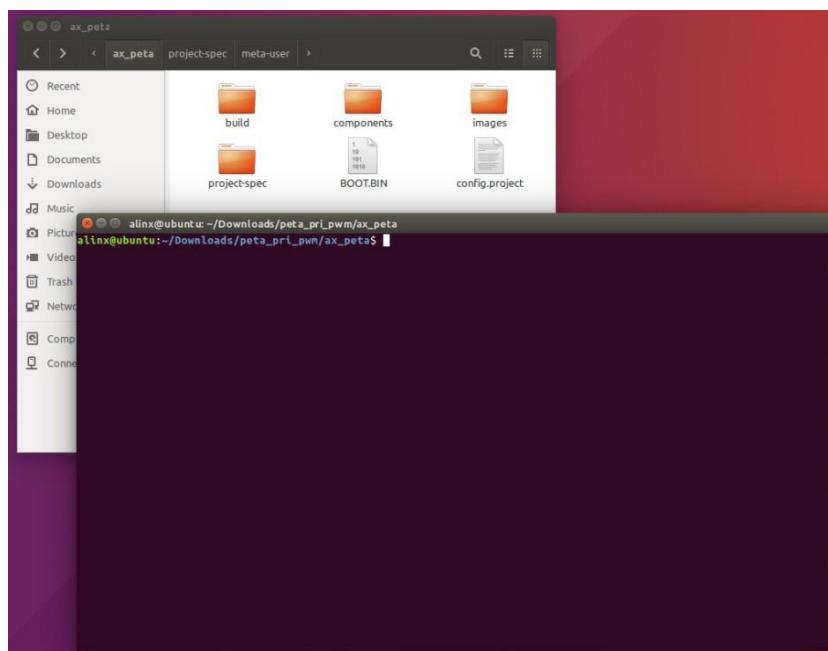
```
void writel(u32 value, volatile void_iomem *addr);
```

**value** is the value to be written, and **addr** is the address to be manipulated.

The **write()** function on line 146 judges whether the value of **\_buf** input by the user is 0 or 1 and executes the light-on and light-off operations accordingly.

### Part 1.3.3: Add a New Driver in the Petalinux Custom System

Refer to the previous tutorial for the method of petalinux to customize the Linux system. After getting the customized system, open the terminal and enter the root directory of the customized system, as shown below:



**ax\_peta** is the system engineering root directory that I customized with petalinux

- 1) First enter the following command in the terminal to set the

**petalinux** environment variable:

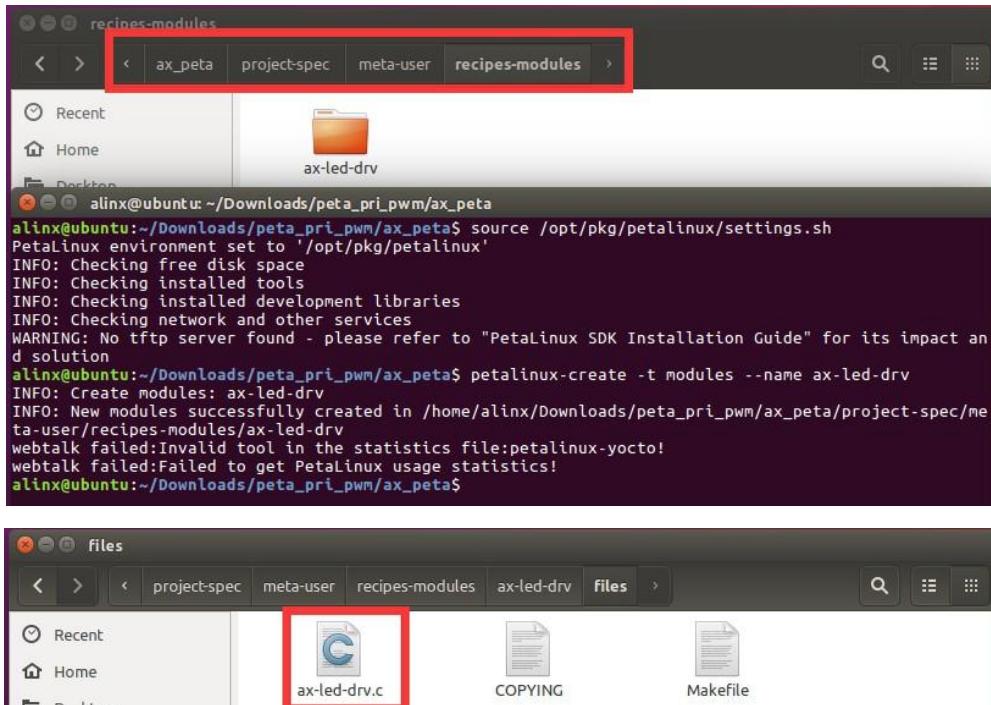
```
source /opt/pkg/petalinux/settings.sh
```

- 2) Then enter the following command to add a new driver:

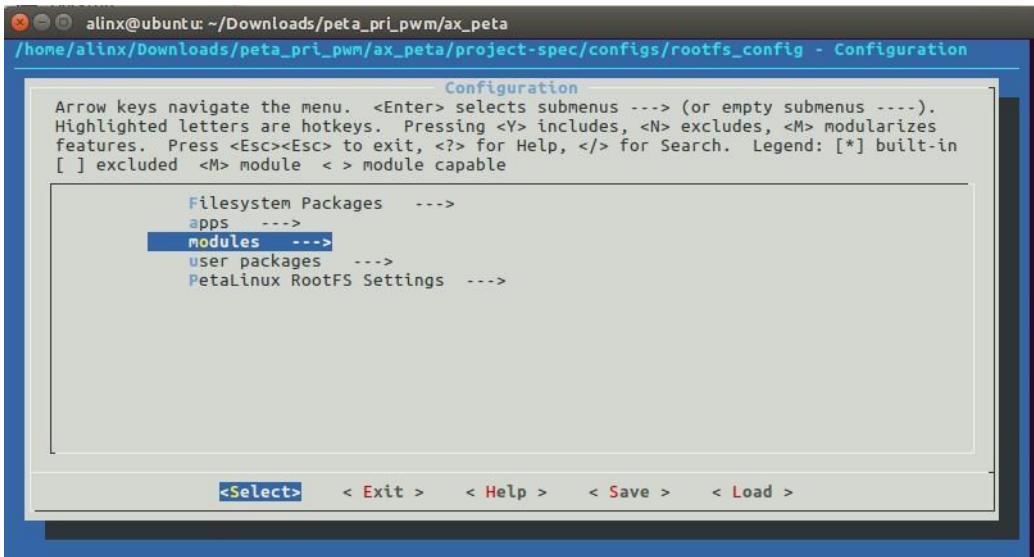
```
petalinux-create -t modules --name ax-led-drv
```

- 3) Check the directory
-

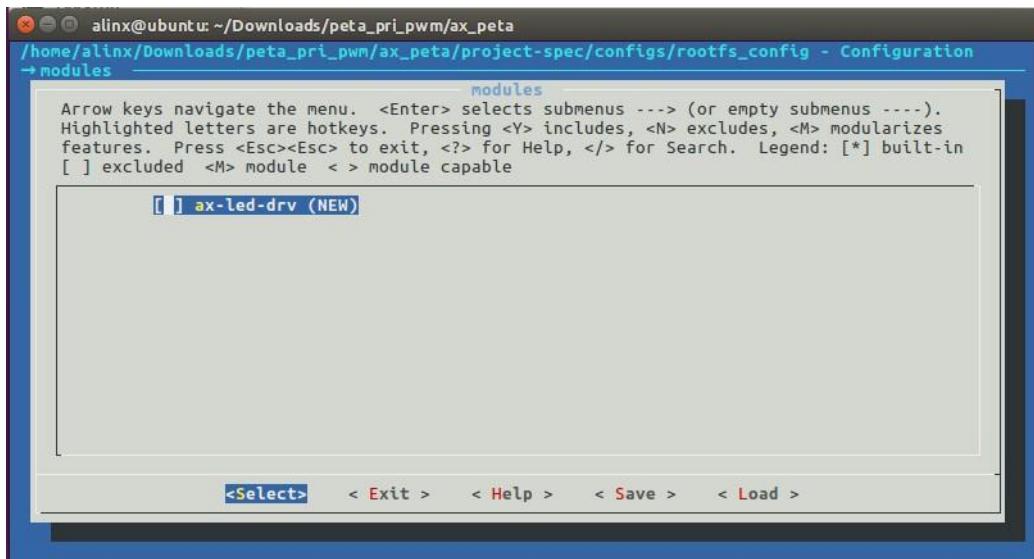
"[.../ax\\_peta/project-spec/meta-user/recipes-modules](#)", there is a folder named "[ax-led-drv](#)" below, and then enter the directory "[.../ax\\_peta/project-spec/meta-user/recipes-modules/ax-led-drv/files](#)", the file named "[ax-led-drv.c](#)" in the directory is the driver file created by petalinux for us.



- 4) Open this c file and paste the led driver code we implemented earlier to save and exit. Open the terminal to the previous “[ax\\_peta](#)” directory, enter the command “[petalinux-config -c rootfs](#)”, and then the interactive interface will pop up as shown below:



- 5) Press the “up” and “down” keys to move to the module option, and press the “space ”to enter the new interface as follows:



The “ ax-led-drv” in the option is our newly added driver. Press the space to select it, and an asterisk [\*] will appear in the square brackets in front of it. Press the left and right direction keys to move the option to <Save>, press Enter to save the settings, and then select <Exit> to exit the interface. After compiling the petalinux project, this driver will be compiled.

- 6) After exiting the interactive interface, enter the command in the terminal:

```
petalinux-build
```

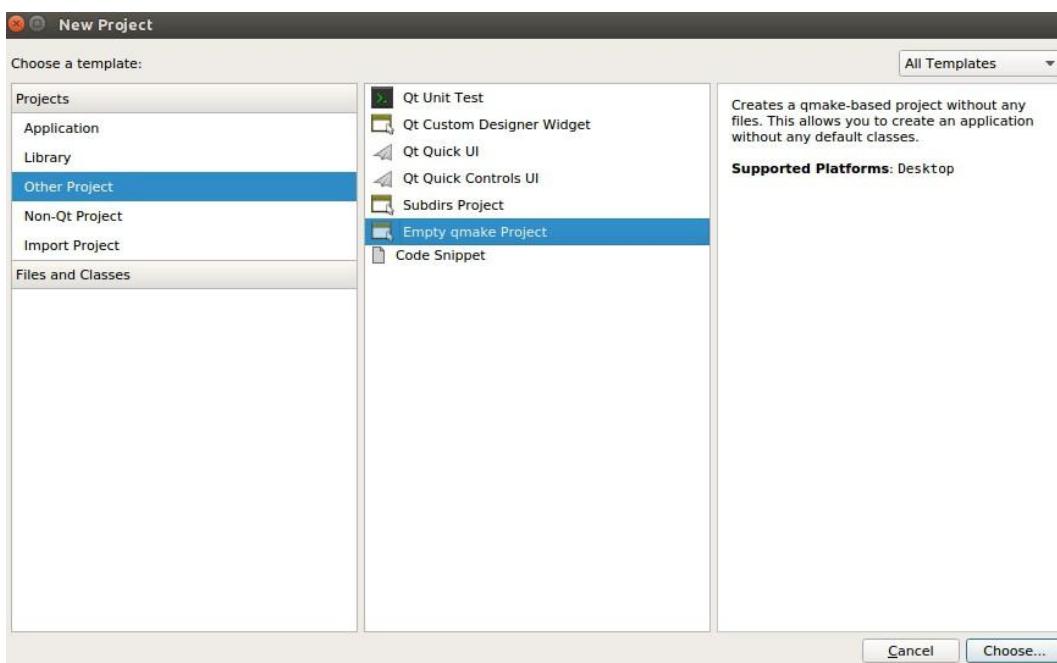
Compile the [petalinux](#) project, search for "ax-led-drv.ko" in the "ax\_peta" directory after the compilation is complete, it is the required driver module file, save it, and we will load the driver module into the system later.



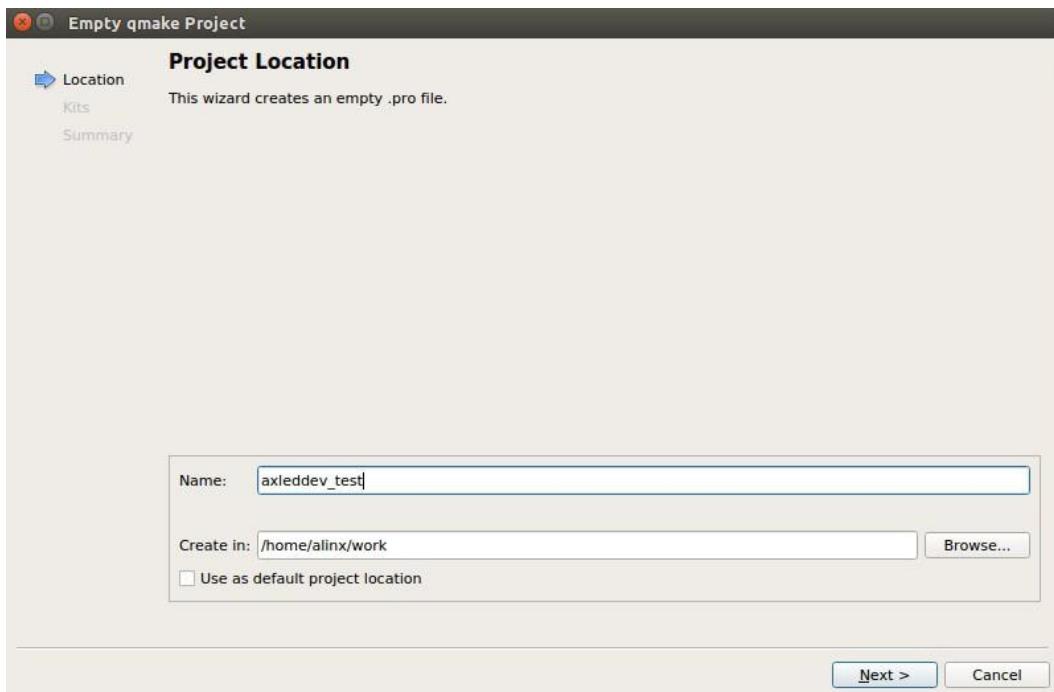
#### Part 1.3.4: Use QT in Ubuntu Virtual Machine to Write Test APP

After the driver code is completed, its function needs to be tested. For the convenience of testing, here we use QT in the Ubuntu virtual machine to write the test program. For the application method of QT in ubuntu virtual machine, please refer to previous tutorial.

- 1) Open QT and click the New Project button to create a new empty project as shown below:



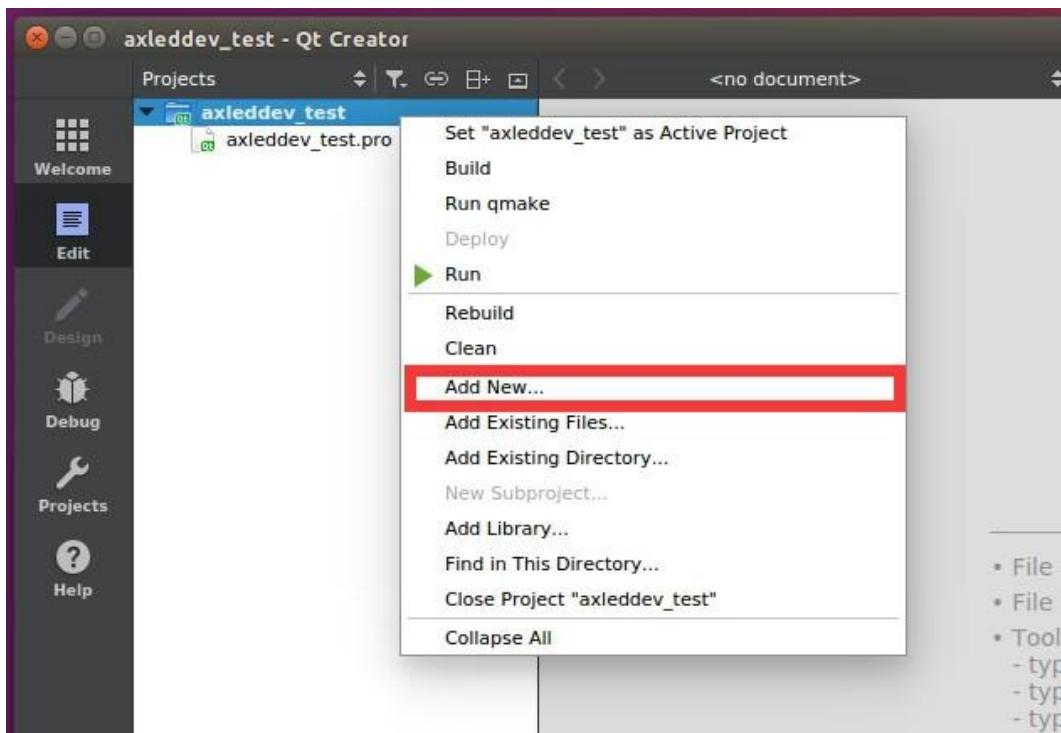
- 2) The test project here is named "axleddev\_test"



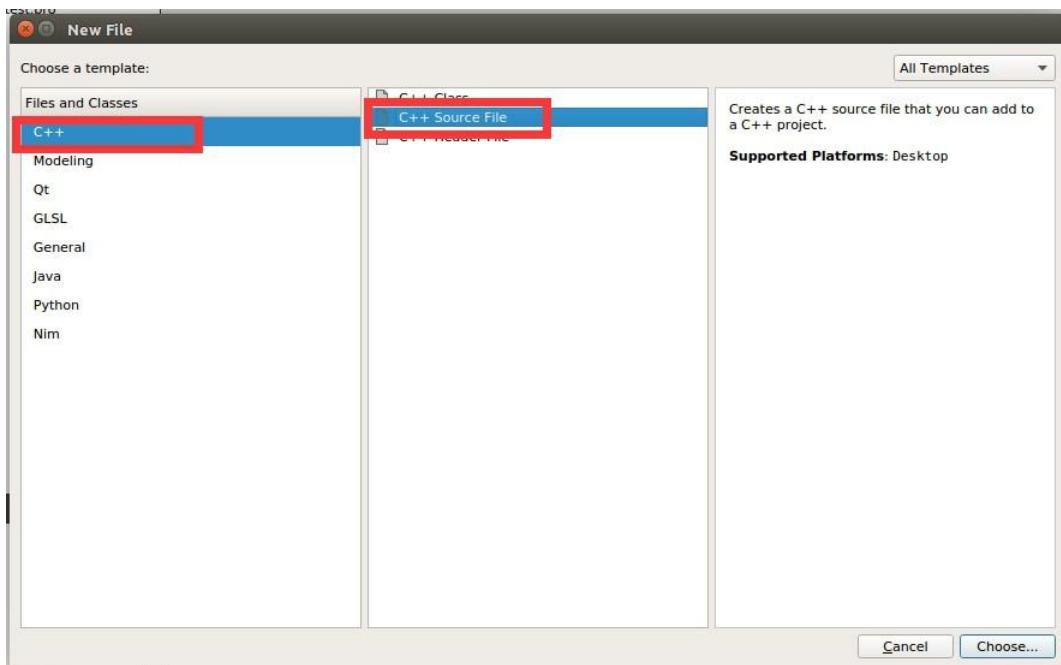
- 3) Choose “**IDE 5.7.1 GCC 4bit**”. If you don't have this option here, please refer to previous tutorial to set up the cross-editing toolchain of QT.



- 4) Open the newly created project, right-click on the project directory, and click the “**Add New**” option



## 5) Add “C++ Source File” and name it “main.c”



## 6) Open “main.c” and enter the following code:

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <unistd.h>
4. #include <fcntl.h>
5.
6. int main(int argc, char **argv)
7. {
8.     int fd;
9.     char buf;
10.
11.    if(3 != argc)
12.    {
```

```
13.     printf("none para\n");
14.     return -1;
15. }
16.
17. fd = open(argv[1], O_RDWR);
18. if(fd < 0)
19. {
20.     printf("Can't open file %s\r\n", argv[1]);
21.     return -1;
22. }
23.
24. if(!strcmp("on",argv[2]))
25. {
26.     printf("ps_led1 on\n");
27.     buf = 1;
28.     write(fd, &buf, 1);
29. }
30. else if(!strcmp("off",argv[2]))
31. {
32.     printf("ps_led1 off\n");
33.     buf = 0;
34.     write(fd, &buf, 1);
35. }
36. else
37. {
38.     printf("wrong para\n");
39.     return -2;
40. }
41.
42. close(fd);
43. return 0;
44. }
```

The system call function in the application program is slightly different from the function format in the kernel driver

The prototype of the **open()** function on line 17 is:

```
int open (const char *_file, int _oflag, ...);
```

The parameter description is as follows:

**\_file:** device file. We enter the device file name through the second parameter when running the program.

**\_oflag:** File opening mode, one of three must be selected:

```
#define O_RDONLY      00 // Read Only
#define O_WRONLY       01 //Write Only
#define O_RDWR         02 // Read and Write
```

There are other check options, so I won't mention them here.

**Return value:** If the file is opened successfully, the file handle is returned.

The **write()** function prototype on line 28 is:

```
ssize_t write (int_fd, const void *_buf, size_t_n);
```

The parameter description is as follows:

**\_fd:** The file handle returned by the open() function.

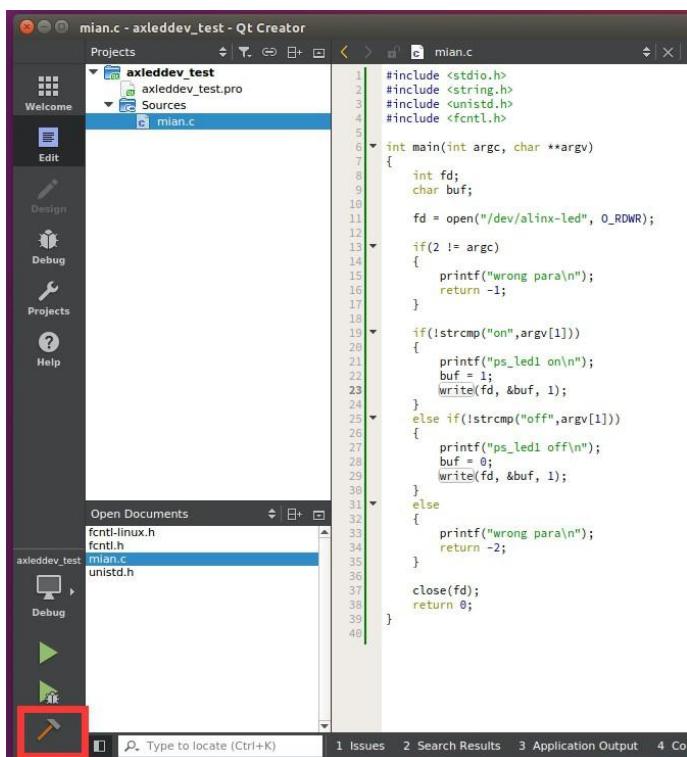
\_buf: The first address of the data to be written.

\_n: The length of data to be written.

Here, the call method of the **write()** function corresponds to the implementation in the previous driver. Only two values of 0 or 1 are input to notify the driver to turn on or off the light.

Line 42 calls the **close()** function to release the file handle after the driver file is used. The input parameter is the file handle. After the close() function is executed, the **release()** function implemented in the final driver will be executed.

- 7) After the test code is completed, click the small hammer in the lower right corner of QT to compile the project, as shown below:



- 8) After the compilation is complete, find the folder generated by the compilation at the same level and directory as the QT project. The "**axleddev\_test**" inside is the executable file that our FPGA development board can run. What this program wants to achieve is to turn on the light when input “**on**”, and turn off when input “**off**”.



### Part 1.3.5: Run test

The driver module and test program are available, and the test can be started.

Note that the FPGA development board should run the linux system compiled from the petalinux project the same as the driver we created.

- 1) Open the serial port tool, power on the FPGA development board, and log in to the linux system

```

COM4 - PuTTY
version 2.88 booting
Starting udev
udevd[761]: starting version 3.2
random: udevd: uninitialized urandom read (16 bytes read)
random: udevd: uninitialized urandom read (16 bytes read)
random: udevd: uninitialized urandom read (16 bytes read)
udevd[762]: starting eudev-3.2
random: udevd: uninitialized urandom read (16 bytes read)
FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
Populating dev cache
random: ddi: uninitialized urandom read (512 bytes read)
hwclock: can't open '/dev/misc/rtc': No such file or directory
Thu Feb 13 06:32:30 UTC 2020
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting internet superserver: inetd.
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
  Removing any system startup links for run-postinsts ...
/etc/rcS.d/S99run-postinsts
INIT: Entering runlevel: 5
Configuring network interfaces... IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
udhcpc (v1.24.1) started
Sending discover...
Sending discover...
Sending discover...
No lease, forking to background
done.
Starting Dropbear SSH server: random: dropbearkey: uninitialized urandom read (32 bytes read)
Generating key, this may take a while...
random: dropbearkey: uninitialized urandom read (32 bytes read)
random: dropbearkey: uninitialized urandom read (32 bytes read)
Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQDzKZfkwmR0utH71FmPgdVho8/XYFPiGlkaHlK/9Z+qF2SzFlU0D19VYfUafDu2s6t6erandom: dropbear: uninitialized urandom read (32 bytes read)
lrm+dMf8ii7pBIA26HMouGrWK1lg7YsEH17JdEdGNExBjUmkI0qG48wSpqY31jcAU1OM+3aXVkhTsxZNu3A3vmaunXRlLUQ+1lDGSVFbQ01wAf1w8jPTx6gA8qoBFdBs+sk5y0xW86iqUwEuypt8Ar0cpu
szpjCVTAKEJ4g0j1iH0LMNhlpSlW+fyIB4E0IKZGRtKKhnuNgvAyvdZ5151CdKa2CtOcfcxpuQfkzPpCBnfZffIRxpgfmAj+d3qJJ1kLi3PnPdi/HOR root@ax_peta
Fingerprint: md5 00:c4:e7:34:de:a6:69:a0:8a:4a:13:5a:9a:e7:37:a5
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done
Starting tcf-agent: random: tcf-agent: uninitialized urandom read (16 bytes read)
OK

Petalinux 2017.4 ax_peta /dev/ttys0

ax_peta login: root
Password:
root@ax_peta:~#

```

- 2) Enter the following command to mount the working path of “ubuntu” to the “/mnt” path of the FPGA development board using the NFS service.

```

mount -t nfs -o noblock 192.168.1.107:/home/ilinx/work /mnt
cd /mnt

```

```
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
```

Among them, **192.168.1.107** is the ip of my ubuntu virtual machine, and “**/home/alinx/work**” is the working path of my “**ubuntu**”. These two need to be modified and adjusted according to the actual situation.

- 3) Put the previously obtained driver module “**ax-led-drv.ko**” and the executable file “**axleddev\_test**” into the work path, which is “**/home/alinx/work**” here.
- 4) Load the driver module “**Insmod ax-led-drv.ko**”. The console prints “**gpio\_led\_dev\_init\_ok**”, and the loading is successful.

```
root@ax_peta:/mnt# insmod ax-led-drv.ko
ax_led_drv: loading out-of-tree module taints kernel.
gpio_led_dev_init_ok
```

- 5) Earlier we mentioned using the command “**cat /proc/devices**” to view the device number that has been used. Now that our device driver is loaded successfully, we can also use this command to view whether the device number has been successfully registered. It does already exist.

```
root@ax_peta:/mnt# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
10 misc
13 input
21 sg
29 fb
81 video4linux
89 i2c
90 mtd
116 alsa
128 ptm
136 pts
180 usb
189 uec_dvci
200 gpio_leds
245 uio
246 xdevcfg
247 watchdog
248 iio
```

- 6) So far we still lack the device file, the command to create the device file is:

```
mknod /dev/ilinx-led c 200 0
```

Then execute the command “ls /dev”, you can find the device file we need.

- 7) The device file is also ready, then you can run the test program, the lighting command is as follows:

```
./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug/axleddev_test /dev/ilinx-led on
```

```
root      ./mnt# ./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug/axleddev_test /dev/ilinx-led on
[25994.442260] gpio_test module open
ps_led1 on
[25994.445734] gpio_test module write
[25994.450159] gpio_test ON *GPIO_DATA_1 = 0x3FFFFFFF
[25994.454954] gpio_test module release
```

Led is lit, and the printing information used for debugging is also output successfully.

- 8) Turn off led:

```
./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug/axleddev_test /dev/ilinx-led off
```

- 9) Uninstall the driver module and test it:

```
rmmmod ax_led_drv
```

```
root@ax_peta:/mnt# rmmmod ax_led_drv
gpio_led_dev_exit_ok
```

# Part 2: A New Way of Writing Character Devices

## Part 2.1: Device Number Management

In the experiment in the previous chapter, the device number was directly written in the driver code. This will cause a lot of trouble:

- 1) Before compiling the driver code, you must first check the occupancy of the device number in the target system;
- 2) After replacing the device, the device number that was previously written in the driver may have been occupied;
- 3) The original driver registration function `register_chrdev()` has only the major device number and no minor device number in the input parameters, which means that one device will occupy all the minor device numbers, which is very wasteful.

In response to these problems, the Linux kernel proposes a new method of character device registration, and the kernel manages the device number. Added two new device number registration functions.

- 1) When the driver needs to give the major device number, use the function to register the device number:

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

### Input parameter description:

**from:** The starting device number that needs to be applied for, the `dev_t` type, which replaces the original major device number and minor device number. In the case of specifying the major and minor device numbers, you can get its value through the method “[from=MKDEV\(major, minor\)](#)”

**count:** The number of device numbers that need to be applied for, generally only one is required.

**name:** The device name

Example:

```
1. int major = 200; //主设备号指定为 200
2. int minor = 0; //次设备号为 0
3. dev_t devid = MKDEV(major, minor); //通过主次设备号获得设备号
4. /*向内核注册设备号*/
5. register_chrdev_region(devid, 1, "xxx-dev");
```

- 2) When the driver does not need to specify the major device number, use the function:

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name)
```

**Input parameter description:**

**dev:** Device number pointer. If the major and minor device numbers are not specified, the device number is allocated by the kernel. Therefore, the pointer is passed in to obtain the device number. After the registration is successful, the method “**major=MAJOR(\*dev)**” and “**minor = MINOR( \*dev)**”, to obtain the major and minor device numbers separately. If you don't need to use the major and minor device numbers, you don't need to get them.

**baseminor:** the starting address of the minor device number.

**count:** The number of device numbers that need to be applied for, generally only one is required.

**name:** The device name

Example:

```
1. dev_t devid; //设备号
2. /* 申请设备号 */
3. alloc_chrdev_region(&devid, 0, 1, "xxx-dev");
```

- 3) Just use the same function to cancel the device number

```
void unregister_chrdev_region(dev_t from, unsigned count)
```

The meaning of the input parameters is consistent with the registration function.

Example:

```
1. dev_t devid;      //设备号
2. /* 申请设备号 */
3. alloc_chrdev_region(&devid, 0, 1, "xxx-dev");
4. /* 注销设备号 */
5. unregister_chrdev_region(devid, 1);
```

## Part 2.2: New Registration Method

As mentioned above, there is a problem with the registration function `register_chrdev()`. Corresponding to the new device number management method, `register_chrdev()` is now abandoned and a new method is used to register character devices.

### Part 2.2.1: Character Device Data Structure

Use the `cdev` structure to define a character device, which is defined in “`include/linux/cdev.h`”, as follows:

```
1. struct cdev {
2.     struct kobject kobj;
3.     struct module *owner;
4.     const struct file_operations *ops;
5.     struct list_head list;
6.     dev_t dev;
7.     unsigned int count;
8. };
```

Important variables:

**owner**: Generally set to `THIS_MODULE`

**ops**: Device operation function pointer

**dev**: Device number

### Part 2.2.2: cdev Structure Initialization

The `cdev` structure variable needs to be initialized with the `cdev_init()` function after the definition, the function prototype is

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
```

**cdev**: character device structure pointer

**fops**: device operation function collection structure pointer

For example:

```
1. /* 字符设备 */
2. struct cdev ax_cdev = {
3.     .owner = THIS_MODULE,
4. };
5.
```

```
6. /* 设备操作函数 */
7. static struct file_operations ax_fops = {
8.     .owner = THIS_MODULE,
9.     .open.....
10. };
11.
12. /* ax_cdev 变量初始化 */
13. cdev_init(&ax_cdev, &ax_fops);
```

### Part 2.2.3: New Registration and Dereistration Functions

After initializing the character structure variable, you can use this variable to register the character device with the **Linux** system. Use the new registration function **cdev\_add**, the prototype is:

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

**p**: The character device structure variable after the above initialization

**dev**: device number;

**count**: The number of devices to be added.

The registration function has changed, and the unregistering and unloading function is the same. The **unregister\_chrdev()** function is no longer used, and the **cdev\_del()** function is used instead. The prototype:

```
void cdev_del(struct cdev *p)
```

The input parameter is the character device structure variable.

Combined with the device number, add the example in Part 2.2.2:

```
1. /* 字符设备 */
2. struct cdev ax_cdev = {
3.     .owner = THIS_MODULE,
4. };
5.
6. /* 设备操作函数 */
7. static struct file_operations ax_fops = {
8.     .owner = THIS_MODULE,
9.     .open.....
10. };
11.
12. /* 设备号 */
13. dev_t devid;
14. /* 申请设备号 */
15. alloc_chrdev_region(&devid, 0, 1, "xxx-dev");
16.
17. /* ax_cdev 变量初始化 */
18. cdev_init(&ax_cdev, &ax_fops);
19. /* 注册字符设备 */
20. cdev_add(&ax_cdev, devid, 1);
21. .....
22. /* 卸载字符设备 */
23. cdev_del(&ax_cdev);
```

## Part 2.3: Automatically Create Device Files

### Part 2.3.1: mdev

**mdev** is a user program, a simplified version of **udev**. It can detect and create or delete device files based on the status of the hardware devices in the system. After the driver module is loaded, the device node file is automatically created in the “**/dev**” directory, and the device node is automatically deleted after the driver module is uninstalled. Let's see how to implement it next.

### Part 2.3.2: Class Creation and Deletion

You need to create a class before creating a device. The device is created under the class. The “**struct class**” structure of the class is defined in “**include/linux/device.h**” and needs to be created using the function “**class\_create()**”. **class\_create()** is a macro definition:

```
1. #define class_create(owner, name) \
2. ({ \
3.     static struct lock_class_key _key; \
4.     __class_create(owner, name, &_key); \
5. }) \
6. \
7. struct class *_class_create(struct module *owner, const char *name, struct lock_class_key *key)
```

After unfolding, you can see:

**Input parameters:**

**owner:** The owner values that have appeared so far are **THIS\_MODULE**

**name:** The name of the class.

**Return value:** structure pointer of struct class type.

When uninstalling the driver, you need to delete the class, use the function **class\_destroy()**, the prototype is as follows:

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

**cls** is the class to be deleted.

### Part 2.3.3: Creation and Deletion of Device Nodes

After creating the class, use the `device_create()` function to create a device under the class. The prototype is:

```

1. struct device *device_create(struct class *class,
2.                               struct device *parent,
3.                               dev_t devt,
4.                               void *drvdata,
5.                               const char *fmt, ...)
```

Parameter Description:

**class**: The class introduced in the previous section, the device will be created under this class

**parent**: parent device, fill in NULL if there is no parent device

**devt**: device number

**drvdata**: data that may be used by the device, if not, fill in NULL;

**fmt**: Device name, for example, when “**fmt=axled**”, the “**/dev/axled**” file will be generated after the device is created. The delete device function is:

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

The meaning of the input parameters is the same as above.

The implementation of automatically creating device nodes is generally placed in the driver entry function, combined with the creation of the class in the previous section and the device number, the implementation example of automatically creating device nodes is as follows

```

1. struct class *class;      /* 类 */
2. struct device *device;    /* 设备 */
3. dev_t devid;             /* 设备号 */
4.
5. /* 驱动入口函数 */
6. static int init_xxx_init(void)
7. {
8.     ....
9.     /* 申请设备号 */
10.    alloc_chrdev_region(&devid, 0, 1, "xxx-dev");
11.    ....
12.    /* 创建类 */
13.    class = class_create(THIS_MODULE, "xxx");
14.    /* 创建设备 */
15.    device = device_create(class, NULL, devid, NULL, "xxx");
16.
17.    return 0;
18. }
19.
20. /* 驱动出口函数 */
21. static void exit_xxx_exit(void)
```

```
22. {
23.     /* 删除设备 */
24.     device_destroy(class, devid);
25.     /* 删除类 */
26.     class_destroy(class);
27.     /* 注销字符设备 */
28.     unregister_chrdev_region(devid, 1);
29. }
30.
31. module_init(led_init);
32. module_exit(led_exit);
```

## Part 2.4: Character Device New Driver Experiment

Now that we understand the key points related to the new writing method of the character device driver, let's try it out. In this chapter, we will write the device driver for the led "PS LED1" on the FPGA development board, which can be turned on and off by the driver.

### Part 2.4.1: Hardware Schematic and Data Sheet

Same as Part 1.3.1.

### Part 2.4.2: Writing a Character Device Driver

Use [petalinux](#) to create a new driver, the method is also the same as the previous chapter, and the repeated steps will be skipped. Here I created a new driver named "[ax-newled-drv](#)". Open the file ax-newled-drv.c and enter the following:

```
1. /**
2.  *Author : ALINX Electronic Technology (Shanghai) Co., Ltd.
3.  *Website: http://www.alinx.com
4.  *Address: Room 202, building 18,
5.  *          No.518 xinbrick Road,
6.  *          Songjiang District, Shanghai
7.  *Created: 2020-3-2
8.  *Version: 1.0
9.  */
10.
11. #include <linux/module.h>
12. #include <linux/kernel.h>
13. #include <linux/fs.h>
14. #include <linux/init.h>
15. #include <linux/ide.h>
16. #include <linux/types.h>
17. #include <linux/errno.h>
18. #include <linux/cdev.h>
19. #include <linux/device.h>
20. #include <asm/uaccess.h>
21.
22. /* 设备节点名称 */
23. #define DEVICE_NAME      "gpio_leds"
24. /* 设备号个数 */
25. #define DEVID_COUNT      1
26. /* 驱动个数 */
27. #define DRIVE_COUNT       1
28. /* 主设备号 */
```

```
29. #define MAJOR
30. /* 次设备号 */
31. #define MINOR          0
32.
33. /* gpio 寄存器虚拟地址 */
34. static unsigned long gpio_add_minor;
35. /* gpio 寄存器物理基地址 */
36. #define GPIO_BASE        0xFF0A0000
37. /* gpio 寄存器所占空间大小 */
38. #define GPIO_SIZE         0x1000
39. /* gpio 方向寄存器 */
40. #define GPIO_DIRM_1       (unsigned int*)(0x000000000000244 + (unsigned long)gpio_add_minor)
41. /* gpio 使能寄存器 */
42. #define GPIO_OEN_1        (unsigned int*)(0x000000000000248 + (unsigned long)gpio_add_minor)
43. /* gpio 控制寄存器 */
44. #define GPIO_DATA_1       (unsigned int*)(0x0000000000000044 + (unsigned long)gpio_add_minor)
45.
46. /* 把驱动代码中会用到的数据打包进设备结构体 */
47. struct alinx_char_dev{
48.     dev_t          devid;      //设备号
49.     struct cdev    cdev;       //字符设备
50.     struct class   *class;     //类
51.     struct device  *device;    //设备节点
52. };
53. /* 声明设备结构体 */
54. static struct alinx_char_dev alinx_char = {
55.     .cdev = {
56.         .owner = THIS_MODULE,
57.     },
58. };
59.
60. /* open 函数实现, 对应到 Linux 系统调用函数的 open 函数 */
61. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
62. {
63.     printk("gpio_test module open\n");
64.
65.     return 0;
66. }
67.
68.
69. /* write 函数实现, 对应到 Linux 系统调用函数的 write 函数 */
70. static ssize_t gpio_leds_write(struct file *file_p, const char user *buf, size_t len, loff_t *loff_t_p)
71. {
72.     int rst;
73.     char writeBuf[5] = {0};
74.
75.     printk("gpio_test module write\n");
76.
77.     rst = copy_from_user(writeBuf, buf, len);
78.     if(0 != rst)
79.     {
80.         return -1;
81.     }
82.
83.     if(1 != len)
84.     {
85.         printk("gpio_test len err\n");
86.         return -2;
87.     }
88.     if(1 == writeBuf[0])
89.     {
90.         *GPIO_DATA_1 |= 0x00004000;
91.         printk("gpio_test ON\n");
92.     }
93.     else if(0 == writeBuf[0])
94.     {
95.         *GPIO_DATA_1 &= 0xFFFFBFFF;
96.         printk("gpio_test OFF\n");
97.     }
98.     else
99.     {
100.        printk("gpio_test para err\n");
101.       return -3;
102. }
```

```
102.    }
103.
104.    return 0;
105. }
106.
107. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
108. static int gpio_leds_release(struct inode *inode_p, struct file*file_p)
109. {
110.     printk("gpio_test module release\n");
111.     return 0;
112. }
113.
114. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
115. static struct file_operations ax_char_fops = {
116.     .owner    = THIS_MODULE,
117.     .open     = gpio_leds_open,
118.     .write    = gpio_leds_write,
119.     .release  = gpio_leds_release,
120. };
121.
122. /* 模块加载时会调用的函数 */
123. static int initgpio_led_init(void)
124. {
125.     /* 注册设备号 */
126.     alloc_chrdev_region(&alinx_char.devid, MINOR, DEVID_COUNT, DEVICE_NAME);
127.
128.     /* 初始化字符设备结构体 */
129.     cdev_init(&alinx_char.cdev, &ax_char_fops);
130.
131.     /* 注册字符设备 */
132.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
133.
134.     /* 创建类 */
135.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
136.     if(IS_ERR(alinx_char.class))
137.     {
138.         return PTR_ERR(alinx_char.class);
139.     }
140.
141.     /* 创建设备节点 */
142.     alinx_char.device = device_create(alinx_char.class, NULL, alinx_char.devid, NULL, DEVICE_NA
E);
143.     if (IS_ERR(alinx_char.device))
144.     {
145.         return PTR_ERR(alinx_char.device);
146.     }
147.
148.     /* 把需要修改的物理地址映射到虚拟地址 */
149.     gpio_add_minor = ioremap_wc(GPIO_BASE, GPIO_SIZE);
150.
151.     /* MIO_0 设置成输出 */
152.     *GPIO_DIRM_1 |= 0x00004000;
153.     /* MIO_0 使能 */
154.     *GPIO_OEN_1 |= 0x00004000;
155.
156.     return 0;
157. }
158.
159. /* 卸载模块 */
160. static void exitgpio_led_exit(void)
161. {
162.     /* 释放对虚拟地址的占用 */
163.     iounmap(gpio_add_minor);
164.
165.     /* 注销字符设备 */
166.     cdev_del(&alinx_char.cdev);
167.
168.     /* 注销设备号 */
169.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
170.
171.     /* 删除设备节点 */
172.     device_destroy(alinx_char.class, alinx_char.devid);
173.
174.     /* 删除类 */
175.     class_destroy(alinx_char.class);
176.
177.     printk("gpio_led_dev_exit_ok\n");
```

```
178. }
179.
180. /* 标记加载、卸载函数 */
181. module_init(gpio_led_init);
182. module_exit(gpio_led_exit);
183.
184. /* 驱动描述信息 */
185. MODULE_AUTHOR("Alinx");
186. MODULE_ALIAS("gpio_led");
187. MODULE_DESCRIPTION("NEW GPIO LED driver");
188. MODULE_VERSION("v1.0");
189. MODULE_LICENSE("GPL");
190. DULE_LICENSE("GPL");
```

Compared with the previous chapter, the changed part is **bolded**. In addition to some new macro definitions and parameter declarations, only the entry function and the exit function are changed.

Lines 23-31, several new macro definitions are added. **DEVICE\_NAME** represents the device name and also the device node name. Finally, we need to find the device file with this name in the “**/dev** ”directory.

**DEVID\_COUNT** and **DEVICE\_COUNT** represent the number of device numbers and device nodes respectively. Generally, a device corresponds to a driver, so there is only one device in a driver code, so here are all 1.

MAJOR main device number, here we get the device number through the function, the main device number does not need to be set by yourself. The MAJOR minor device number generally starts at 0.

In lines 47~58, a new structure type is created, and a variable is declared with this type. Pack the data types such as device numbers, character devices, classes, and device nodes that will be used later into this structure. This is much more convenient when using variables, and it also increases readability. And when we use private data later, this structure will also bring convenience, and we will elaborate on it when we use it.

Lines 123~157 are the key points. In the driver entry function, all the knowledge points mentioned in this chapter (applying for device

numbers, initializing and registering character devices, creating classes and device nodes) are all integrated.

The exit function in lines 160~178 is to cancel and delete corresponding to the registration and creation in the entry function.

Combining the contents of the previous sections, this new driver code is not difficult to understand. After completion, recompile the driver module file "ax-newled-dev.ko" on the ubuntu virtual machine.

#### Part 2.4.3: Write Test APP

The test APP is consistent with the content in **Part 1.3.4** of the previous chapter, and the test program in the previous chapter can be used directly.

#### Part 2.4.4: Run Test

The test method is not the same as before. Power on the FPGA development board and hang it from the working directory of the virtual machine to the "/mnt" path of the FPGA development board.

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
```

- 1) Load the driver first, execute the command: insmod ax-newled-dev.ko
- 2) The driver is loaded successfully, and then check if the device file is created successfully, execute the command: ls/dev

```
root@ax_peta:/mnt# ls /dev
block          loop-control      mtd2ro        ram13
bus            loop0           mtd3          ram14
char           loop1           mtd3ro        ram15
console        loop2           mtdblock0     ram2
cpu_dma_latency loop3           mtdblock1     ram3
disk           loop4           mtdblock2     ram4
fd             loop5           mtdblock3     ram5
full           loop6           network_latency ram6
loop           loop7           network_throughput ram7
mem            mem             null          ram8
gpiochip0      memory_bandwidth port          ram9
gpiochip1      mmcblk0         psaux        random
gpiochip2      mmcblk0p1       ptmx         shm
i2c-0          mmcblk0p1       pts          snd
i2c-1          mtab            ram0         stderr
iio:device0    mtd0            ram1         stdin
initctl        mtd0ro          ram10        stdout
input          mtd1            ram11        tty
kmsg           mtd1ro          ram12        tty0
log            mtd2
root@ax_peta:/mnt#
```

- 3) The device node file already exists, you can use the test APP to try the driver, execute the following command:

```
cd./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug/axleddev_test
./axleddev_test /dev/gpio_leds on
```

```
root@...:/mnt# ./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug/axleddev_test /dev/gpio_leds on
[28230.312738] gpio_test module open
ps_led1 on
[28230.316210] gpio_test module write
[28230.320630] gpio_test ON
[28230.323174] gpio_test module release
```

- 4) The led is lit, and finally in the test uninstall the driver, execute the command:

```
rmmmod ax-newled-dev
```

If you are not sure about the driver name, you can execute the “**lsmod**” command to check it.

```
root@ax_peta:/mnt# rmmmod ax-newled-dev
gpio_led_dev_exit_ok
root@ax_peta:/mnt#
```

- 5) After deleting the device, confirm whether the device node file has been deleted:

```
ls /dev/gpio_led
```

```
root@ax_peta:/mnt# ls /dev/gpio_led
ls: /dev/gpio_led: No such file or directory
root@ax_peta:/mnt#
```

No problem, the test was successful.

## Part 3: Device Tree and of Function

### Part 3.1: Device Tree

ARM Linux 3.0 did not have an application device tree before. At that time, ARM Linux users used C language data structures to describe the devices on the board, and each device corresponds to a description file. There are countless types of ARM boards, and in order to ensure the versatility of the kernel source code, the Linux community stuffed all these description files into it, which caused the Linux kernel to be full of junk code and angered Linus, the father of Linux. After that, the ARM community learned that PowerPC introduced a device tree to replace the previous method.

The device tree has other advantages besides using no code. When the hardware corresponding to the driver changes, there is no need to recompile the kernel or driver. You only need to compile the device tree file separately, and pass the device tree compilation result to the kernel when the board is started. (Unfortunately, this advantage is not reflected in petalinux. In the result of petalinux compilation, u-boot, kernel, and device tree are packaged together in [image.ub](#) instead of being provided separately. So change the device tree, it is often used petalinux-build command to recompile the kernel as a whole. In fact, it can be found from the previous two chapters that when changing or adding drivers, the kernel is also compiled as a whole. In fact, it is possible to compile the drivers separately. However, in order to unify the operation steps, the [petalinux-build](#) command is used.)

The device tree is a tree-like device structure. The "tree" is a vivid metaphor, with the system bus as the trunk, and other devices such

as I2C, SPI, and GPIO controllers hanging on the system bus as branches. And these branches mounted on the trunk have their own mounted devices, such as EEPROM, RTC and other devices mounted on the I2C, the branched structure of the trunk is like a tree.

### Part 3.1.1: DTS, DTB and DTSI

Usually the file extension of the device tree that we edit is **.dts** (device tree source), and the device tree file read by the kernel is the binary file **.dtb** (device tree binary large object) obtained after compilation. We can also modify the DTB file directly, but the binary file modification is very inconvenient, and it is not the original intention of the device tree design. Here I will focus on the DTS file.

The content of the DTS file looks a bit like a JSON data structure at first glance, with curly braces inside the braces. Of course there is a difference. DTS has its own set of grammatical rules. For details, please refer to the official document ["devicetree-specification-v0.2.pdf"](#).

First look at the content of a section of DTS file, an excerpt of the file "[arch\arm\boot\dts\zynq-zc702.dts](#)":

```
1. #include "zynq-7000.dtsi"
2.
3. / {
4.     model = "Zynq ZC702 Development Board";
5.     compatible = "xlnx,zynq-zc702", "xlnx,zynq-7000";
6.
7.     ....
8.
9.     memory@0 {
10.         device_type = "memory";
11.         reg = <0x0 0x40000000>;
12.     };
13.
14.     ....
15.
16.     gpio-keys {
17.         compatible = "gpio-keys";
18.         #address-cells = <1>;
19.         #size-cells = <0>;
20.         autorepeat;
21.         sw14 {
22.             label = "sw14";
23.             gpios = <&gpio0 12 0>;
24.             linux,code = <108>; /* down */
25.             wakeup-source;
26.             autorepeat;
27.         };
28.     };
29.
```

```
28.  
29.      ....  
30.    };  
31.};
```

Combine this code to get a general understanding of the syntax format of DTS.

Line 1 is the statement containing the header file. Like the C language, DTS files use “#include” to refer to header files, and can refer to “.h”, “.dts” and “.dtsi” files. The “.dtsi” file is a file specially used for writing DTS header files. The syntax is the same as that of DTS files, and it is the first choice for writing DTS header files.

### Part 3.1.2: node

#### ➤ Common node format

In the DTS file, devices are represented by nodes in the format:

```
1. [label:] node-name[@unit-address] {  
2.     [properties definitions]  
3.     [child nodes]  
4. };
```

The part in [] is optional.

**node-name**: device node name, the root node is represented by “/”. The “/” in the third line is the root node of the device “[Zynq ZC702 Development Board](#)”. The **memory** in line 9, **gpio-keys** in line 11, and **sw14** in line 21 are all device node names.

**unit-address**: generally refers to the device address or the first address of the register, such as **memory@0** in line 9 of the code. If the device does not have a device address or register, there is no need to write it.

**label**: another name for the device, used to easily access the node. For example, a node named **slcr@f8000000**, if you want to access it normally, you need to use the name **slcr@f8000000** to access. If you label this node with **slcr1: slcr@f8000000**, you only need to use **&slcr1** to access the node.

The content in `{}` is the content of the node. There are two types of node content. **[properties definitions]** are the node properties, which will be explained later. **[child nodes]** is the child node hanging on this device node. The format of the child node is the same as described above.

### ➤ Special node aliases

The **aliases** node is used to define aliases, similar to label, and the format is as follows:

```
1.   aliases {
2.     ethernet0 = "&gem0";
3.     serial0 = "&uart1";
4.   };
```

Then you can use `&gem` to access the node **ethernet**.

#### Part 3.1.3: attributes

There are 4 forms of node properties **[properties definitions]**:

##### 1) [label:]property-name;

The attribute is empty. Such as the “**autorepeat**” at lines 20 and 26 in the sample code.

##### 2) [label:]property-name = <arrays of cells>;

The value content enclosed in `<>` is a collection of 32-bit data. For example, `reg = <0x0 0x40000000>` in line 11 of the sample code.

##### 3) [label:]property-name = "string";

Use `""` to include character strings, such as `compatible = "xlnx,zynq-zc702", "xlnx,zynq-7000"` in the line 5.

##### 4) [label:]property-name = [bytestring];

The character sequence enclosed in `[]` is relatively rare. Suppose there is an attribute represented as `memory-addr = [0011223344]`; this is equivalent to `memory-addr = [00 11 22 33 44]`; Its value is a sequence of 5 byte numbers, and this byte number is composed of two hexadecimal numbers.

Attributes can be customized by users, and there are many standard attributes. Here are some common standard attributes:

➤ **compatible attribute**

The compatible attribute value is a list of strings, which is the key to the device and driver association. Define an **OF** matching table in the driver code to correspond to the **compatible** attribute. For example, in line 17 of the sample code, the **compatible = "gpio-keys"** of the node **gpio-keys**, then the corresponding driver code will have the following definitions to correspond to it:

```
1. static const struct of_device_id gpiokey_ids[] = {  
2.     { .compatible = "gpio-keys", },  
3.     { /* sentinel */ }  
4. };
```

**struct of\_device\_id** is the data type of **OF** matching table. When the **compatible** value in the driver **OF** matching table does not correspond to the **compatible** value in the device tree node, the node will use this driver. But at present we only use the character device framework, this framework does not need the OF matching table. When we use the bus device model later, we will talk about the OF matching table.

The **compatible** attribute in the root node indicates which platforms the board is compatible with. There are generally two values, the former represents the model of the board, and the latter represents the chip used.

➤ **model attribute**

The value of the **model** attribute is also a string, generally used to represent the name of the board, similar to the **compatible** attribute in the root node.

➤ **#address-cells, #size-cells, and reg attributes**

The **#address-cells** attribute indicates how many u32 integers

are used to describe the address in the **reg** attribute of the child node of the current node.

The **#size-cells** attribute indicates how many u32 integers are used in the reg attribute of the current node's child node to describe the size length.

The **reg** attribute is generally used to describe the register address range information of a certain peripheral, the format is **reg = <address1 length1 address2 length2 address3 length3.....>**. The **#address-cells** attribute in the parent node refers to the size of the address in the reg attribute, and **#size-cells** refers to the size of the length in the reg attribute.

This is also not reflected in the sample code. In order to facilitate understanding, here is an extreme example:

```
1. ax-parent {  
2.     #address-cells = <2>;  
3.     #size-cells = <1>;  
4.     ....  
5.  
6.     ax-son {  
7.         reg = <0x00000001 0x00000002 0x00000003  
8.                     0x00000004 0x00000005 0x00000006>;  
9.         ....  
10.    }  
11. }
```

The **reg** attribute of the child node **ax-son** has 6 **u32** data, and **#address-cells** in the parent node **ax-parent** is equal to 2. Therefore, the value of the **reg** attribute of the child node **ax-son** indicates that there are two **u32** numbers for address, that is, **0x00000001** and **0x00000002** are both **address** values. Similarly, **#size-cells** is equal to 1, so the value of **length1** has only one **u32** data equal to **0x00000003**. The following three **u32** data are the values of **address2** and **length2**.

#### ➤ device\_type attribute

This attribute can only be used for **cpu** nodes or **memory** nodes.

In the **cpu** node, **device\_type = "cpu"**, in the **memory** node,

---

`device_type = "memory".`

➤ **phandle attribute**

The value of the phandle attribute must be unique, and its function is similar to the label, which is used to reference nodes.

```
1. ax-node-1 {  
2.     phandle = <1>;  
3.     interrupt-controller;  
4. }  
5.  
6. ax-node-2 {  
7.     interrupt-parent = <1>;  
8. }
```

In the node **ax-node-1**, the **phandle** attribute is **<1>**, and the **interrupt-parent** attribute in **ax-node-2** needs to specify the parent node, and assign the value to **<1>**.

The standard attributes are introduced first, and there are some special ones that will be used later.

#### Part 3.1.4: view the device tree in the file system

When the kernel is started, it will parse the node information in the **DTB** file and create a folder corresponding to the node in the “**/proc/devicetree**” directory of the root file system.

```
root@ax_peta:~# cd /proc/device-tree  
root@ax_peta:/proc/device-tree# ls  
#address-cells  alinxled      chosen      fpga-full      pmu@f8891000  
#size-cells     alinxpwm     compatible    memory      usb_phy@0  
aliases         amba        cpus        model  
alinkxkey       amba_pl     fixedregulator  name  
root@ax_peta:/proc/device-tree#
```

Enter the corresponding node, you can view the various attributes of the node.

```
root@ax_peta:/proc/device-tree# cd memory/  
root@ax_peta:/proc/device-tree/memory# ls  
device_type  name      reg  
root@ax_peta:/proc/device-tree/memory# cat device_type  
memory  
root@ax_peta:/proc/device-tree/memory#
```

#### Part 3.1.5: modify the device tree

The device tree itself has certain standards. Different chip manufacturers have some different custom standards for the device

---

tree. When we modify the device tree, some need to follow these standards, but when we don't know the standard, we can go to the kernel source directory. You can view instructions and guidance in the kernel source directory [/Documentation/devicetree/bindings](#). If you can't find it, you can only consult the chip manufacturer.

## Part 3.2: of Function

The Linux kernel provides the “**of**” function to let us get the information in the device tree. The name “**of**” comes from the prefix “**of\_**” of these functions. The prototype of the “**of**” function is defined in the kernel directory “[include/linux/of.h](#)”. In this section, we introduce some commonly used “**of**” functions, and it's not too late to learn about the ones that are not introduced when they are used.

### Part 3.2.1: of function to find node

#### ➤ **device\_node** structure

The **device\_node** structure is also defined in “[include/linux/of.h](#)”, as the return value of the “**of**” function to find the node, and is used to describe the device node to the kernel.

#### ➤ **of\_find\_node\_by\_name** function

**of\_find\_node\_by\_name ()** :Find a node by node name, the function prototype is:

```
struct device_node *of_find_node_by_name(struct device_node *from, const char *name);
```

Parameter Description:

**from:** Search from this node, and search from the root node when **NULL** is entered.

**name:** target node name

**Return value:** Find the target node and return the **device\_node** structure. **NULL** is returned when not found.

#### ➤ **of\_find\_node\_by\_type** function

---

**of\_find\_node\_by\_type ()**: Find the node through the **device\_type** property, the function prototype is:

```
struct device_node *of_find_node_by_type(struct device_node *from, const char *type);
```

Parameter Description:

**from**: Search from this node, and search from the root node when **NULL** is entered.

**type**: **device\_type** attribute value

**Return value**: Find the target node and return the **device\_node** structure. **NULL** is returned when not found.

➤ **of\_find\_compatible\_node function**

**of\_find\_compatible\_node ()**: Find the node through the **device\_type** and **compatible** properties, the function prototype is:

```
struct device_node *of_find_compatible_node(struct device_node *from, const char *type,  
const char *compatible);
```

Parameter Description:

**from**: Search from this node, and search from the root node when **NULL** is entered.

**type**: **device\_type** attribute value, ignored when entering **NULL**

**compatible**: compatible attribute value.

**Return value**: Find the target node and return the **device\_node** structure. **NULL** is returned when not found.

➤ **of\_find\_node\_by\_path function**

**of\_find\_node\_by\_path ()**: Find the node through the node path, the function prototype is

```
struct device_node *of_find_node_by_path(const char *path);
```

Parameter Description:

**path**: the full path of the node. Take the sample code in Part 3.1.1 as an example, the full path of the **sw14** device on line 21 is **"/gpio-keys/sw14"**.

**Return value**: Find the target node and return the **device\_node**

---

structure. **NULL** is returned when not found.

### Part 3.2.2: of function to extract attributes

#### ➤ property structure

The **property** structure is also defined in “[include/linux/of.h](#)”, as the return value of the “**of**” function that extracts attributes, and is used to describe node attributes to the kernel.

#### ➤ of\_find\_property function

**of\_find\_property function:** find properties by device node, property name, and property value. Function prototype:

```
property *of_find_property(const struct device_node *np, const char *name, int *len);
```

Parameter Description:

**np:** device node.

**name:** The target attribute name.

**len:** The length of the target attribute value.

**Return value:** target attribute.

#### ➤ of\_property\_read\_u32\_array function

**of\_property\_read\_u32\_array():** Used to obtain multiple data of properties with multiple values. The function name and **u32** in the input function represent the size of a single value of the target property, which can be replaced with **u8**, **u16**, **u32**. Function prototype:

```
int of_property_read_u32_array(const struct device_node *np,const char *propname, u32  
*out_values, size_t size);
```

Parameter Description:

**np:** device node.

**propname:** The target attribute name.

**out\_values:** The received data pointer. The received data will be saved to this address.

**size:** Number of data to be read

**Return value:**

0: Read successfully

-EINVAL: The attribute does not exist

-ENODATA: The attribute has no data

-EOVERFLOW: The number of attribute value data is less than

`size`

➤ **of\_property\_read\_u32 function**

**of\_property\_read\_u32()**: used to obtain property data with only a single value. The function name and u32 in the input function represent the size of a single value of the target property, which can be replaced with **u8**, **u16**, **u32**. Function prototype:

```
int of_property_read_u32(const struct device_node *np,const char *propname, u32  
out_value);
```

Parameter Description:

**np**: device node.

**proname**: The target attribute name.

**out\_values**: Target data pointer, the received data will be saved to this address

**Return value:**

0: Read successfully

-EINVAL: The attribute does not exist

-ENODATA: The attribute has no data

➤ **of\_property\_read\_string function**

**of\_property\_read\_string function**: used to collect the string value in the property, the function prototype is as follows:

```
int of_property_read_string(struct device_node *np, const char *propname, const char  
**out_string);
```

Parameter Description:

**np**: device node.

**proname**: The target attribute name.

---

**out\_string:** The target string pointer, the read string will be saved to this address.

**Return value:** Return 0, read successfully

### Part 3.3: led Drive Experiment under the Device Tree

After getting to know the device tree and the “**of**” function in the above two sections, it is difficult to understand what is written in writing, but it is still necessary to understand deeply through experiments. This section will still use a simple led to test, and will not cover all the above, but in the future experiments, the device tree will be used all the time, so don't worry, just slowly master it in the subsequent experiments.

#### Part 3.3.1: Schematic

Same as part1.3.1

#### Part 3.3.2: Modify the Device Tree

The **petalinux** project file provides the device tree file for us to modify In the project directory

"**ax\_peta/project-spec/meta-user/recipes-bsp/device-tree/file.s**", **ax\_peta** is the name of my petalinux project, which needs to be modified according to its actual situation.

Open the file "**system-conf.dtsi**" and add the following node content under the root node:

```
1. alinxled {
2.     compatible = "alinkled";
3.     reg = <
4.         0xFF0A0244 0x04      /* gpio 方向寄存器 */
5.         0xFF0A0248 0x04      /* gpio 使能寄存器 */
6.         0xFF0A0044 0x04      /* gpio 控制寄存器 */
7.     >;
};
```

The node name is **alinkled**, the compatibility attribute value is

"**alinxled**", and the value in **reg** is the led-related register used in the previous two experiments.

If your "**system-conf.dtsi**" file is empty, then write a root directory yourself, and then put the contents above into the root directory.

### Part 3.3.3: Driver

Use **petalinux** to create a new driver named "**ax-dtled-drv**". Don't forget to use "**petalinux-config -c rootfs**" command to select the new driver.

Enter the following code in the "**ax-dtled-drv.c**" file:

```
1. /** ===== */
2. *Author : ALINX Electronic Technology (Shanghai) Co., Ltd.
3. *Website: http://www.alinx.com
4. *Address: Room 202, building 18,
5.           No.518 xinbrick Road,
6.           Songjiang District, Shanghai
7. *Created: 2020-3-2
8. *Version: 1.0
9. ** ===== */
10.
11. #include <linux/module.h>
12. #include <linux/kernel.h>
13. #include <linux/fs.h>
14. #include <linux/init.h>
15. #include <linux/ide.h>
16. #include <linux/types.h>
17. #include <linux/errno.h>
18. #include <linux/cdev.h>
19. #include <linux/of.h>
20. #include <linux/device.h>
21. #include <asm/uaccess.h>
22.
23. /* 设备节点名称 */
24. #define DEVICE_NAME      "gpio_leds"
25. /* 设备号个数 */
26. #define DEVID_COUNT      1
27. /* 驱动个数 */
28. #define DRIVE_COUNT       1
29. /* 主设备号 */
30. #define MAJOR
31. /* 次设备号 */
32. #define MINOR            0
33.
34. /* gpio 寄存器虚拟地址 */
35. static u32 *GPIO_DIRM_1;
36. /* gpio 使能寄存器 */
37. static u32 *GPIO_OEN_1;
38. /* gpio 控制寄存器 */
39. static u32 *GPIO_DATA_1;
```

```
40.
41. /* 把驱动代码中会用到的数据打包进设备结构体 */
42. struct alinx_char_dev{
43.     dev_t          devid;      //设备号
44.     struct cdev    cdev;       //字符设备
45.     struct class   *class;     //类
46.     struct device  *device;    //设备
47.     struct device_node *nd;    //设备树的设备节点
48. };
49. /* 声明设备结构体 */
50. static struct alinx_char_dev alinx_char = {
51.     .cdev = {
52.         .owner = THIS_MODULE,
53.     },
54. };
55.
56. /* open 函数实现，对应到Linux 系统调用函数的 open 函数 */
57. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
58. {
59.     printk("gpio_test module open\n");
60.
61.     return 0;
62. }
63.
64.
65. /* write 函数实现，对应到 Linux 系统调用函数的write 函数 */
66. static ssize_t gpio_leds_write(struct file *file_p, const char user *buf, size_t len, loff_t *loff_t_p)
67. {
68.     int rst;
69.     char writeBuf[5] = {0};
70.
71.     printk("gpio_test module write\n");
72.
73.     rst = copy_from_user(writeBuf, buf, len);
74.     if(0 != rst)
75.     {
76.         return -1;
77.     }
78.
79.     if(1 != len)
80.     {
81.         printk("gpio_test len err\n");
82.         return -2;
83.     }
84.     if(1 == writeBuf[0])
85.     {
86.         *GPIO_DATA_1 |= 0x00004000;
87.         printk("gpio_test ON\n");
88.     }
89.     else if(0 == writeBuf[0])
90.     {
91.         *GPIO_DATA_1 &= 0xFFFFBFFF;
92.         printk("gpio_test OFF\n");
93.     }
94.     else
95.     {
96.         printk("gpio_test para err\n");
97.         return -3;
98.     }
99.
100.    return 0;
101. }
102.
103. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
```

```
104. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
105. {
106.     printk("gpio_test module release\n");
107.     return 0;
108. }
109.
110. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
111. static struct file_operations ax_char_fops = {
112.     .owner    = THIS_MODULE,
113.     .open     = gpio_leds_open,
114.     .write    = gpio_leds_write,
115.     .release  = gpio_leds_release,
116. };
117.
118. /* 模块加载时会调用的函数 */
119. static int init gpio_led_init(void)
120. {
121.     /* 用于接受返回值 */
122.     u32 ret = 0;
123.     /* 存放 reg 数据的数组 */
124.     u32 reg_data[10];
125.
126.     /* 通过节点名称获取节点 */
127.     alinx_char.nd = of_find_node_by_name(NULL, "alinxled");
128.     /* 4、获取 reg 属性内容 */
129.     ret = of_property_read_u32_array(alinx_char.nd, "reg", reg_data, 8);
130.     if(ret < 0)
131.     {
132.         printk("get reg failed!\r\n");
133.         return -1;
134.     }
135.     else
136.     {
137.         /* do nothing */
138.     }
139.
140.     /* 把需要修改的物理地址映射到虚拟地址 */
141.     GPIO_DIRM_1    = ioremap_wc(reg_data[0], reg_data[1]);
142.     GPIO_OEN_1    = ioremap_wc(reg_data[2], reg_data[3]);
143.     GPIO_DATA_1   = ioremap_wc(reg_data[4], reg_data[5]);
144.
145.     /* 注册设备号 */
146.     alloc_chrdev_region(&alink_char.devid, MINOR, DEVID_COUNT, DEVICE_NAME);
147.
148.     /* 初始化字符设备结构体 */
149.     cdev_init(&alink_char.cdev, &ax_char_fops);
150.
151.     /* 注册字符设备 */
152.     cdev_add(&alink_char.cdev, alink_char.devid, DRIVE_COUNT);
153.
154.     /* 创建类 */
155.     alink_char.class = class_create(THIS_MODULE, DEVICE_NAME);
156.     if(IS_ERR(alink_char.class))
157.     {
158.         return PTR_ERR(alink_char.class);
159.     }
160.
161.     /* 创建设备节点 */
162.     alink_char.device = device_create(alink_char.class, NULL,
163.                                         alink_char.devid, NULL,
164.                                         DEVICE_NAME);
165.     if (IS_ERR(alink_char.device))
166.     {
```

```
167.         return PTR_ERR(alinx_char.device);
168.     }
169.
170.     /* MIO_0 设置成输出 */
171.     *GPIO_DIRM_1 |= 0x00004000;
172.     /* MIO_0 使能 */
173.     *GPIO_OEN_1 |= 0x00004000;
174.
175.     return 0;
176. }
177.
178. /* 卸载模块 */
179. static void exit gpio_led_exit(void)
180. {
181.     /* 注销字符设备 */
182.     cdev_del(&alinx_char.cdev);
183.
184.     /* 注销设备号 */
185.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
186.
187.     /* 删除设备节点 */
188.     device_destroy(alinx_char.class, alinx_char.devid);
189.
190.     /* 删除类 */
191.     class_destroy(alinx_char.class);
192.
193.     /* 释放对虚拟地址的占用 */
194.     iounmap(GPIO_DIRM_1);
195.     iounmap(GPIO_OEN_1);
196.     iounmap(GPIO_DATA_1);
197.
198.     printk("gpio_led_dev_exit_ok\n");
199. }
200.
201. /* 标记加载、卸载函数 */
202. module_init(gpio_led_init);
203. module_exit(gpio_led_exit);
204.
205. /* 驱动描述信息 */
206. MODULE_AUTHOR("Alinx");
207. MODULE_ALIAS("gpio_led");
208. MODULE_DESCRIPTION("DEVICE TREE GPIO LED driver");
209. MODULE_VERSION("v1.0");
210. MODULE_LICENSE("GPL");
```

The main changes are concentrated in the line 127~143 of the entry function.

Line 127 uses the `of_find_node_by_name` function to get the node by the node name, because the `alinkled` node is hung in the root directory, so the first parameter is `NULL`.

Line 129, after getting the node, get the data of the `reg` attribute in the node, because the `reg` attribute stores the register address and size we need. There are 3 addresses and 3 sizes in total, so the total

number of data is 6.

The other operations are basically the same as in the previous chapter.

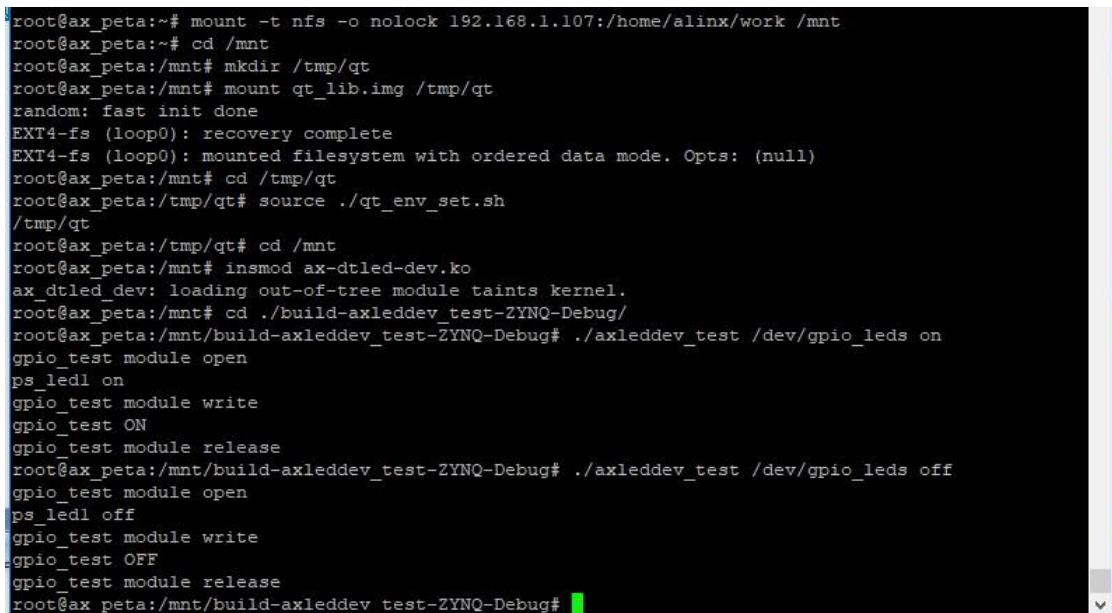
Because the device tree is modified, in Part 3.1, **petalinux** will compile new **BOOT.bin** and **image.ub** after modifying the device tree. So don't forget to copy the new **BOOT.bin** and **image.ub** to SD after compiling, and restart the FPGA development board.

#### Part 3.3.4: test program

Same as Part1.3.4 and can be used directly.

#### Part 3.3.5: run test

Because the APP is the same, the test method is still the same, as long as the led can be successfully lit, it will be successful.



```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/ilinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
random: fast init done
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ax-dtled-dev.ko
ax_dtled_dev: loading out-of-tree module taints kernel.
root@ax_peta:/mnt# cd ./build-axleddev_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
gpio_test module open
ps_led1 on
gpio_test module write
gpio_test ON
gpio_test module release
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds off
gpio_test module open
ps_led1 off
gpio_test module write
gpio_test OFF
gpio_test module release
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug#
```

In addition, after the system is running, check whether there is any **alinxled** node we added in the following “**/proc/device-tree**” path, and check the content.

```
root@ax_peta:~# cd /proc/device-tree
root@ax_peta:/proc/device-tree# ls
#address-cells    amba          cpus          model
#size-cells       amba_pl       fixedregulator name
aliases           chosen         fpga-full    pmu@f8891000
alinxled          compatible     memory        usb_phy@0
root@ax_peta:/proc/device-tree# cd ./alinxled/
root@ax_peta:/proc/device-tree/alinxled# ls
compatible      name          reg
root@ax_peta:/proc/device-tree/alinxled# cat ./compatible
alinxled
root@ax_peta:/proc/device-tree/alinxled# cat ./name
alinxled
root@ax_peta:/proc/device-tree/alinxled#
```

## Part 4: pinctrl and gpio Subsystem

In the previous experiment, we operated the equipment directly to operate the register. Although the device tree was used in the previous chapter, it only puts the register information in the device tree, which is essentially the same. Direct manipulation of registers to control hardware resources is very convenient for coders, but it must be very crashing for people who read codes. Moreover, writing a large number of register addresses in the device tree is not only difficult to read, but also easy to make mistakes in use. For example, in the previous chapter, 4 registers were used to control the leds only, and the order of the register addresses in the device tree was not ruled. They were all self-determined, and the addresses received in the driver could not be matched and confirmed. Finally, we It is easy to misalign when setting the register.

The **pinctrl** subsystem can cope with this problem, and at the same time, the gpio subsystem needs to be combined with the leds used in our experiments, that is, the gpio operation.

### Part 4.1: pinctrl Subsystem

The Linux kernel provides a pinctrl subsystem to unify the pin management method of chip manufacturers. It will help us to complete the following tasks:

- 1) Get pin information from the device tree
- 2) Set the multiplexing function of the pin
- 3) Configure the electrical characteristics of the pin

To use the pinctrl subsystem, you only need to configure the relevant node information in the device tree, and the rest of the work is done by the pinctrl subsystem.

---

### Part 4.1.1: Device Tree Nodes and Attributes of pinctrl Subsystem of zynq Platform

Different chip pin configuration methods are different, and the pinctrl subsystem implementation of corresponding chip manufacturers will also be different, but we still need to care about the writing of the device tree. Just look at an example:

```
1. firmware {
2.     zynqmp_firmware: zynqmp-firmware {
3.         compatible = "xlnx,zynqmp-firmware";
4.         method = "smc";
5.         pinctrl0: pinctrl {
6.             compatible = "xlnx,zynqmp-pinctrl";
7.             pinctrl_uart1_default: uart1-default {
8.                 mux {
9.                     groups = "uart0_4_grp";
10.                    function = "uart0";
11.                };
12.                conf {
13.                    groups = "uart0_4_grp";
14.                    slew-rate = <SLEW_RATE_SLOW>;
15.                    io-standard = <IO_STANDARD_LVCMOS18>;
16.                };
17.                conf-rx {
18.                    pins = "MIO18";
19.                    bias-high-impedance;
20.                };
21.                conf-tx {
22.                    pins = "MIO19";
23.                    bias-disable;
24.                    schmitt-cmos = <PIN_INPUT_TYPE_CMOS>;
25.                };
26.            };
27.        };
28.    };
29.};
```

The firmware node is under the root node. The node we need to care about is **pinctrl0**, which is a child node under **firmware\zynqmp\_firmware**. As for the two nodes of **firmware** and **zynqmp\_firmware**, we don't have to worry about it. We can guess that these two nodes are a firmware driver implemented by **xilinx** for **zynqmp**. The **pinctrl** subsystem is also implemented on this basis. If you are interested can go to the kernel according to the **compatible** attribute. Find the corresponding source code implementation.

The part related to **pinctrl** is the **pinctrl0** node at the beginning of the fifth line. The source code of the **pinctrl** driver can also be found through the **compatible** attribute. We only care about how to modify the device tree. **pinctrl0** is the node of the **pinctrl** subsystem. All devices that use the **pinctrl** subsystem add sub-nodes under this node. The format of the child node is the **pinctrl\_uart1\_default** node in the example. There are no attributes under this node, but 4 child nodes: **mux**, **conf**, **conf-rx**, and **conf-tx**. Among them, **mux** is the multiplexing function of the configuration pin, the other three are the electrical characteristics of the configuration pin, the general characteristics are set in **conf**, and the input pins, related characteristics are set in **conf-rx**, and set output pins and related characteristics in **conf-tx**.

The format of the sub-nodes is like **pinctrl\_uart1\_default** in the example. node. There are no attributes under this node, but 4 child nodes: mux, conf, conf-rx, and conf-tx. Among them, mux is the multiplexing function of the configuration pin, the other three are the electrical characteristics of the configuration pin, general characteristics are set in conf, input pins and related characteristics are set in conf-rx, and output pins and output pins are set in conf-tx.

Related features.

In the **mux** node, there are two necessary attributes:

**groups:** Select the pin group list of this multiplexing function. The Optional values refer to the “[Documentation\devicetree\bindings\pinctrl\xlnx,zynqmp-pinctrl.txt](#)” file in the kernel source code, and search for the keyword “**Valid values for groups are:**”. As for which group should be selected, it needs to correspond to the pins used (that is, the pins attribute in the conf node). Take **uart0\_4\_grp**, **MIO18**, and **MIO19** in the example as

---

examples. We can find their correspondence in the register manual.

#### MIO\_PIN\_18 (IOU\_SLCR) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:8	rw	0x0	reserved
L3_SEL	7:5	rw	0x0	Level 3 Mux Select: 0: GPIO [18] input/output bank 0. 1: CAN0 RX input. 2: I2C0 SCL input/output clock. 3: LPD SWDT clock output. 4: SPI1 MISO input/output. 5: TTC2 clock input. 6: UART0 RxD input. <small>7.. reserved</small>

In the option of **MIO18**, there is UART0, so the group prefix selected by groups here is **uart0**. So where does the label 4 in **uart0\_4\_grp** come from? In fact, it is just a sequential label. From MIO\_PIN\_0 to MIO\_PIN\_18, you will find that MIO18 is (starting from 0) number 4 uart0.

**function:** Select the multiplexing function of pin grouping. The optional values refer to the kernel source code “[\Documentation\devicetree\bindings\pinctrl\xlnx,zynqmp-pinctrl.txt](#)”, search for the keyword “Valid values for function are:”. The choice of function is very clear, here MIO is used as uart0, then function is selected as uart0.

The **conf** node can be divided into multiple groups, because the electrical characteristics of each **pin** in the **pin** group with the same multiplexing function may be different. For example, **uart0** in the example, one **pin** is used as an output without setting the bias, and the other is used as an output. The input needs to be set to high impedance. The name of the **conf** node must contain the keyword “**conf**”. Other keywords are only used to distinguish the meaning of the group. For example, the **conf-rx** in the example refers to the **conf** node that is suitable for configuring the input pin. The naming **conf-in** has the same effect. Even the exchange of **rx** and **tx** in the example

has no effect on the use, and is only used for the distinction between the above.

The necessary attributes of the **conf** node are either **pin** or **group**. Use the attribute **group** when setting the pin group, such as the **conf** node in line 10 in the example, and use the attribute **pin** when setting a single **pin** or multiple pins that are not in a group, such as **conf-rx** on line 15 and **conf-tx** on line 19.

The **group** attribute is the same as in the **mux** node above.

**pin:** Use the pin list of this “**conf**”. The optional values are: MIO0 ~ MIO77.

The conf node also has some optional attributes:

**io-standard:** Set the **io** level standard, optional values are: IO\_STANDARD\_LVCMOS33、IO\_STANDARD\_LVCMOS18

**bias-disable, bias-high-impedance, bias-pull-up:** Set the pin bias, no assignment is required, and the required bias attribute needs to be included in the corresponding conf node.

**slew-rate:** Set the conversion rate. When it is equal to 0, it is slow, and when it is 1, it is fast.

**low-power-disable, low-power-enable:** To enable low-power mode, just include the required bias attributes in the corresponding **conf** node.

**schmitt-cmos:** Choose to use **schmitt** or **cmos**, optional values are **PIN\_INPUT\_TYPE\_SCHMITT**, **PIN\_INPUT\_TYPE\_CMOS**.

There are other optional attributes not introduced one by one, they can be found in the kernel source directory

In the Documentation

“**Documentation/devicetree/bindings/pinctrl/pinctrl-binding.txt**” file, search for “**== Generic pin configuration node content ==**” entry to view.

### Part 4.1.2: Use of pinctrl Subsystem

After adding the device child node to the **pinctrl** node, the system does not go back to initialize the device, we need to call it in the “**client device**” node. for example:

```
1. device {  
2.     pinctrl-names = "active", "idle";  
3.     pinctrl-0 = <&pinctrl_uart1_default>;  
4.     pinctrl-1 = <&state_1_node_a &state_1_node_b>;  
5. };
```

**device**: client device node, which is the device that calls **pinctrl**.

**pinctrl-0** is a required attribute, which is the **pinctrl** node application list corresponding to the **pin** of this device. If this device corresponds to multiple **pins**, add the attribute **pinctrl-0~n**. If a pin corresponds to multiple **pinctrls**, just like the **pinctrl-1** attribute in line 4, multiple **pinctrl** nodes are generally attached.

**pinctrl-names** is the name list of **pinctrl0~n** below. The naming is customized but it should be as easy to read as possible and fit the actual situation.

## Part 4.2: gpio Subsystem

In the **pinctrl** subsystem, if the function is configured as **gpio**, the gpio subsystem is needed. The gpio subsystem helps us realize gpio initialization and provides some interface functions for us to operate gpio. What we have to do is to configure the information in the device tree, and then understand the usage of the interface function.

### Part 4.2.1: gpio in the device tree

In the “[Documentation/devicetree/bindings/gpio/gpio-zynq.tx](#)” file, the description of the zynq series chip **gpio** in the device tree is introduced. We can also find the default configuration of **gpio** in the existing zynq device tree, as follows :

In the “[Documentation/devicetree/bindings/gpio/gpio-zynq.txt](#)” file, the description of the zynq series chip **gpio** in the number of devices is introduced, and **zynqmp** is similar. In the existing **zynqmp** device tree “[\arch\arm64\boot\dts\xilinx\zynqmp.dtsi](#)”, the default configuration of **gpio** can also be found, as follows:

```
1. gpio: gpio@ff0a0000 {
2.     compatible = "xlnx,zynqmp-gpio-1.0";
3.     status = "disabled";
4.     #gpio-cells = <0x2>;
5.     interrupt-parent = <&gic>;
6.     interrupts = <0 16 4>;
7.     interrupt-controller;
8.     #interrupt-cells = <2>;
9.     reg = <0x0 0xff0a0000 0x0 0x1000>;
10.    gpio-controller;
11.    power-domains = <&zynqmp_firmware PD_GPIO>;
12.};
```

These attributes are all necessary attributes.

**interrupt-parent**, **interrupts**, **interrupt-controller**, **#interrupt-cells**:

These four attributes are interrupt-related, and we will talk about interrupts later.

**#gpio-cells**: Two cells are required to refer to this **gpio** node, and its value must be 2, that is, two cells are required when referencing the node, the first cell is the **gpio** number, and the second cell is used to refer optional parameter, the second cell is not used in zynq, fill in 0. For example: **alinxled-gpios = <&gpio0 0 0>**;, which is the **io** of **gpio0\_mio0** used by the client device.

**compatible**: Compatibility, must be "[xlnx, zynq-gpio-1.0](#)" or "[xlnx, zynqmp-gpio-1.0](#)".

**gpio-controller**: Specify the node as the **gpio** controller.

**reg**: The first address and range of this group of **gpio** registers.

Here we don't need to modify anything, just keep the existing configuration.

#### Part 4.2.2: gpio interface function

The **gpio** subsystem provides interface functions to replace our direct register operations to separate the driver program into layers. In this section, we introduce some commonly used **gpio** interface functions.

#### ➤ **of function extension**

Before introducing the **gpio** interface function, first extend a gpio-related “**of**” function “**of\_get\_named\_gpio**”, which is used to obtain the **gpio number** through the node attributes in the device tree such as "**gpios = <&gpio0 0 0 &gpio0 1 0>**". The function prototype is as follows:

```
int of_get_named_gpio(struct device_node *np, const char *propname, int index)
```

#### **Parameter Description:**

**np**: device node.

**propname**: The name of the property from which the **gpio** number is obtained.

**index**: The gpio subscript to be obtained.

**Return value**: gpio number; if a negative value is returned, the acquisition fails.

#### ➤ **gpio\_request()**

Before using **gpio** for other operations, you need to apply for an **io** using the **gpio\_request()** function. The interfaces provided by the Linux kernel that I have come into contact with so far are all based on this object-oriented idea. First declare an object and then manipulate the object.

#### Function prototype:

```
int gpio_request(unsigned int gpio, const char *label)
```

#### **Parameter Description:**

**gpio**: The gpio label that needs to be applied for is obtained through the “**of\_get\_named\_gpio**” function described above.

**label:** The label of the applied gpio.

**Return value:** 0 means success, others means failure.

➤ **gpio\_free()**

Release **gpio**, as opposed to **gpio\_request()**, the prototype:

```
void gpio_free(unsigned int gpio)
```

**gpio:** The gpio label to be released.

➤ **gpio\_direction\_input()**

Set **gpio** as input, function prototype:

```
int gpio_direction_input(unsigned gpio)
```

**gpio:** The gpio label to be set.

**Return value:** 0 means success, negative value means failure.

➤ **gpio\_direction\_output()**

Set **gpio** as output, function prototype:

```
int gpio_direction_output(unsigned gpio, int value)
```

**gpio:** The gpio label to be set.

**value:** The default level of the output.

**Return value:** 0 means success, negative value means failure.

➤ **gpio\_get\_value()**

Read the level of **gpio**, prototype:

```
int gpio_get_value(unsigned gpio)
```

**gpio:** The gpio label to be collected.

**Return value:** The received level, a negative value indicates failure.

➤ **gpio\_set\_value()**

Set the level of **gpio**, prototype:

```
void gpio_set_value(unsigned gpio, int value)
```

**gpio:** The **gpio** label to be operated.

**value:** The level to be written.

## Part 4.3: Experiment

The necessary knowledge is understood, and then verified through experiments. Still use led to test.

### Part 4.3.1: Schematic

Same as Part 1.3.1

### Part 4.3.2: Device tree

Open the file "**system-conf.dtsi**" and add the following node content under the root node:

```
1. #include <dt-bindings/pinctrl/pinctrl-zynqmp.h>
2.
3. /include/ "system-conf.dtsi"
4.
5. &gpio {
6.     status = "okay";
7. };
8.
9. &pinctrl0 {
10.    status = "okay";
11.
12.    pinctrl_led_default: led-default {
13.        mux {
14.            groups = "gpio0_40_grp";
15.            function = "gpio0";
16.        };
17.
18.        conf {
19.            pins = "MIO40";
20.            io-standard = <IO_STANDARD_LVCMOS33>;
21.            bias-disable;
22.            slew-rate = <SLEW_RATE_SLOW>;
23.        };
24.    };
25. };
26.
27. / {
28.     alinxled {
29.         compatible = "alinx-led";
30.         pinctrl-names = "default";
31.         pinctrl-0 = <&pinctrl_led_default>;
32.         alinxled-gpios = <&gpio 40 0>;
33.     };
34. };
```

Line 1 contains a header file. The definitions of "**IO\_STANDARD\_LVCMOS33**" and "**SLEW\_RATE\_SLOW**" are all in this header file. If they are not included, compilation errors will occur.

**Lines 5~7** are references to the **gpio** node. As we mentioned earlier, you need to modify the status attribute to "okey". If it remains disabled, the of function related to gpio will be unavailable.

**Lines 9~25** are **pinctrl** nodes, and you also need to modify the **status** attribute. Then add the child node **pinctrl\_led\_default: led-default**, and configure the pin used for the experimental **led** under this node. **MIO40** is in the group **gpio0\_40\_grp**, and the function is **gpio0**. Because there is only one pin, only one **conf** node is enough, the conversion rate is set to **SLEW\_RATE\_SLOW**, the level standard is **IO\_STANDARD\_LVCMOS33**, the bias is disabled, and the configuration target is "**MIO40**"

**Line 28** is our led device node. For the **gpio** subsystem and **pinctrl** subsystem, it is the client device. The **pin** node is called through the attribute **pinctrl-0 = <&pinctrl\_led\_default>**, and the attribute **alinxled-gpios = <&gpio0 0 0>** to bind **gpio**. The name of this attribute **alinxled-gpios** needs label, and we will get the **gpio** label with the same name later.

#### Part 4.3.3: Driver code

Use **petalinux** to create a new driver named "**ax-pinoled-drv**", and execute "**petalinux-config -c rootfs**" command to select the newly added driver.

Enter the following code in the "**ax-pinoled -drv.c**" file:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/fs.h>
4. #include <linux/init.h>
5. #include <linux/ide.h>
6. #include <linux/types.h>
7. #include <linux/errno.h>
8. #include <linux/cdev.h>
9. #include <linux/of.h>
10. #include <linux/of_address.h>
11. #include <linux/of_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
```

```
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <asm/uaccess.h>
17. #include <asm/io.h>
18.
19. /* 设备节点名称 */
20. #define DEVICE_NAME      "gpio_leds"
21. /* 设备号个数 */
22. #define DEVID_COUNT      1
23. /* 驱动个数 */
24. #define DRIVE_COUNT       1
25. /* 主设备号 */
26. #define MAJOR1
27. /* 次设备号 */
28. #define MINOR1            0
29. /* LED 点亮时输入的值 */
30. #define ALINX_LED_ON      1
31. /* LED 熄灭时输入的值 */
32. #define ALINX_LED_OFF     0
33.
34.
35. /* 把驱动代码中会用到的数据打包进设备结构体 */
36. struct alinx_char_dev{
37.     dev_t          devid;           //设备号
38.     struct cdev    cdev;            //字符设备
39.     struct class   *class;          //类
40.     struct device  *device;         //设备
41.     struct device_node *nd;        //设备树的设备节点
42.     int            alinx_led_gpio; //gpio 号
43. };
44. /* 声明设备结构体 */
45. static struct alinx_char_dev alinx_char = {
46.     .cdev = {
47.         .owner = THIS_MODULE,
48.     },
49. };
50.
51. /* open 函数实现，对应到Linux 系统调用函数的 open 函数 */
52. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
53. {
54.     /* 设置私有数据 */
55.     file_p->private_data = &alinkx_char;
56.     printk("gpio_test module open\n");
57.     return 0;
58. }
59.
60.
61. /* write 函数实现，对应到 Linux 系统调用函数的write 函数 */
62. static ssize_t gpio_leds_write(struct file *file_p, const char user *buf, size_t len, loff_t *loff_t_p)
63. {
64.     int retval;
65.     unsigned char databuf[1];
66.     /* 获取私有数据 */
67.     struct alinx_char_dev *dev = file_p->private_data;
68.
69.     retval = copy_from_user(databuf, buf, len);
70.     if(retval < 0)
71.     {
72.         printk("alink led write failed\r\n");
73.         return -EFAULT;
74.     }
75.
76.     if(databuf[0] == ALINX_LED_ON)
```

```
77.     {
78.         gpio_set_value(dev->alinx_led_gpio, !!1);
79.     }
80.     else if(databuf[0] == ALINX_LED_OFF)
81.     {
82.         gpio_set_value(dev->alinx_led_gpio, !!0);
83.     }
84.     else
85.     {
86.         printk("gpio_test para err\n");
87.     }
88.
89.     return 0;
90. }
91.
92. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
93. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
94. {
95.     printk("gpio_test module release\n");
96.     return 0;
97. }
98.
99. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关
键 */
100. static struct file_operations ax_char_fops = {
101.     .owner    = THIS_MODULE,
102.     .open     = gpio_leds_open,
103.     .write    = gpio_leds_write,
104.     .release  = gpio_leds_release,
105. };
106.
107. /* 模块加载时会调用的函数 */
108. static int init gpio_led_init(void)
109. {
110.     /* 用于接受返回值 */
111.     u32 ret = 0;
112.
113.     /* 获得设备节点 */
114.     alinx_char.nd = of_find_node_by_path("/alinkled");
115.     if(alink_char.nd == NULL)
116.     {
117.         printk("alink_char node not find\r\n");
118.         return -EINVAL;
119.     }
120.     else
121.     {
122.         printk("alink_char node find\r\n");
123.     }
124.
125.     /* 获得节点中 gpio 标号 */
126.     alinx_char.alinx_led_gpio = of_get_named_gpio(alinx_char.nd, "alinkled-gpi
os", 0);
127.     if(alink_char.alinx_led_gpio < 0)
128.     {
129.         printk("can not get alinkled-gpios");
130.         return -EINVAL;
131.     }
132.     printk("alinkled-gpio num = %d\r\n", alinx_char.alinx_led_gpio);
133.
134.     /* 申请 gpio 标号对应的引脚 */
135.     ret = gpio_request(alinx_char.alinx_led_gpio, "alinkled");
136.     if(ret != 0)
137.     {
138.         printk("can not request gpio\r\n");
139.     }
```

```
140.
141.     /* 把这个 io 设置为输出 */
142.     ret = gpio_direction_output(alinx_char.alinx_led_gpio, 1);
143.     if(ret < 0)
144.     {
145.         printk("can not set gpio\r\n");
146.     }
147.
148.     /* 注册设备号 */
149.     alloc_chrdev_region(&alinx_char.devid, MINOR1, DEVID_COUNT, DEVICE_NAME);
150.
151.     /* 初始化字符设备结构体 */
152.     cdev_init(&alinx_char.cdev, &ax_char_fops);
153.
154.     /* 注册字符设备 */
155.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
156.
157.     /* 创建类 */
158.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
159.     if(IS_ERR(alinx_char.class))
160.     {
161.         return PTR_ERR(alinx_char.class);
162.     }
163.
164.     /* 创建设备节点 */
165.     alinx_char.device = device_create(alinx_char.class, NULL,
166.                                         alinx_char.devid, NULL,
167.                                         DEVICE_NAME);
168.     if (IS_ERR(alinx_char.device))
169.     {
170.         return PTR_ERR(alinx_char.device);
171.     }
172.
173.     return 0;
174. }
175.
176. /* 卸载模块 */
177. static void exit gpio_led_exit(void)
178. {
179.     /* 释放 gpio */
180.     gpio_free(alinx_char.alinx_led_gpio);
181.
182.     /* 注销字符设备 */
183.     cdev_del(&alinx_char.cdev);
184.
185.     /* 注销设备号 */
186.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
187.
188.     /* 删除设备节点 */
189.     device_destroy(alinx_char.class, alinx_char.devid);
190.
191.     /* 删除类 */
192.     class_destroy(alinx_char.class);
193.
194.     printk("gpio_led_dev_exit_ok\r\n");
195. }
196.
197. /* 标记加载、卸载函数 */
198. module_init(gpio_led_init);
199. module_exit(gpio_led_exit);
200.
201. /* 驱动描述信息 */
202. MODULE_AUTHOR("Alinx");
```

```
203. MODULE_ALIAS("gpio_led");
204. MODULE_DESCRIPTION("PINCTRL AND GPIO LED driver");
205. MODULE_VERSION("v1.0");
206. MODULE_LICENSE("GPL");
```

Compared with the driver code in the previous chapter, the differences are **bolded**. The main reason is that in the entry function, the **gpio** subsystem is used to obtain **gpio** and initialize it instead of the original operation of obtaining registers and then initializing.

**Line 43** defines the **gpio** label in the device structure to facilitate subsequent acquisition operations

**Line 55** was deleted by initializing **io** by register.

**Line 56** sets the private data in the **open** function, and then can be used in the **write** function.

**Line 68** gets private data.

**Lines 79 and 83** use the **gpio\_set\_value** function to set the **gpio** level, use the acquired private data to apply the **gpio** label, and use **!!** to binarize the second input parameter.

**Line 115** gets the **/alinxled** node.

**Line 127** uses the **of\_get\_named\_gpio** function to get the **gpio** label through the attribute **alinxled-gpios**.

**Line 136** uses the **gpio\_request** function to apply for the pin corresponding to the label.

**Line 143** uses the **gpio\_direction\_output** function to set **gpio** as output.

**Line 181** adds **gpio** release operation when uninstalling the module.

#### Part 4.3.4: Test code

Same as **Part 1.3.4** and can be used directly.

#### Part 4.3.3: Run test

Because the APP is the same, the test method is still the same, as long as the led can be successfully lit, it will be successful.

```
root@ax_peta:/mnt# insmod ./ax-pinioled-dev.ko
ax_pinioled_dev: loading out-of-tree module taints kernel.
alinx_char node find
alinxled-gpio num = 899
root@ax_peta:/mnt# 

root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
gpio_test module open
ps_led1 on
alink led on
gpio_test module release
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# 
```

## Part 5: Concurrent Processing

Multitasking operating systems such as Linux cannot avoid concurrency problems. In this chapter, we have a brief understanding of concurrency and concurrency processing.

### Part 5.1: Concurrency in Linux

The so-called concurrency means that multiple tasks access the same resource at the same time, and memory is the resource. Memory errors are likely to lead to system crashes, so we must try our best to avoid concurrent problems.

There are many reasons for concurrent access, such as multi-threaded access, preemption, interruption, etc. These are system mechanisms that are difficult to avoid. To avoid the problems caused by concurrent access, it is better to identify and protect the shared resources that will be accessed at the same time and the programs that will access the shared resources, that is, the critical section. Of course, in many cases, it is not easy to identify the resources that need to be protected, which we can still continue to accumulate.

### Part 5.2: Linux concurrent processing

After identifying the shared resources, we can use the mechanism of protecting shared resources provided by the Linux kernel to avoid concurrent access. In this section, we introduce several concurrent processing methods provided by the kernel.

#### Part 5.2.1: Atomic operations

Atomic operations refer to operations that cannot be divided or

interrupted. The Linux kernel provides atomic variables and a series of functions to achieve atomic operations. In the file “[include/linux/types.h](#)”, atomic variables are defined as follows:

```
1. typedef struct {
2.     int counter;
3. } atomic_t;
```

The method of defining atomic variables is:

```
1. atomic_t a = ATOMIC_INIT(x);
```

**ATOMIC\_INIT(x)** is a macro definition used to initialize atomic variables, **x** is the initial value we assign, and the initial value that is not needed can be omitted.

The read and write operations of atomic variables also need the interface functions provided by the kernel. In addition to the **ATOMIC\_INIT(x)** mentioned above, some commonly used interface functions are as follows:

```
int atomic_read(atomic_t *v): Read v and return  
void atomic_set(atomic_t *v, int i): Write i to v  
void atomic_add(int i, atomic_t *v): the value of v minus i  
void atomic_sub(int i, atomic_t *v): the value of v plus i  
void atomic_inc(atomic_t *v): v increases  
void atomic_dec(atomic_t *v): v decrement  
int atomic_inc_return(atomic_t *v): v increments and returns  
int atomic_dec_return(atomic_t *v): v decrement and return
```

Of course, there is more than that. In addition to operations on integers, the kernel also provides atomic bit operations, so let's learn more when you need it.

The specific usage will also be analyzed in subsequent experiments.

### Part 5.2.2: Lock mechanism

Atomic operations can only protect integer variables and bits, and are not applicable in many scenarios. For example, when you need to protect structure variables with variable structures, you need to use other mechanisms, such as the lock mechanism described in this section. There are also many types of locks in the kernel. First, a commonly used spin lock is introduced.

The mechanism of spin lock is that when a thread wants to access a shared resource, it needs to acquire the corresponding lock and then unlock it. As long as the lock is not released to unlock, other threads cannot acquire the lock and cannot access the shared resource. At this time, threads that need but have not acquired the spin lock will always be in a waiting state (blocked). The waiting state of the thread wastes a lot of processor resources, so the critical area of the spin lock should be as light as possible, and there should be no calling or blocking functions or logic.

The lock is an image metaphor, in the final analysis, it is a structure, defined as follows:

```
1. typedef struct spinlock {
2.     union {
3.         struct raw_spinlock rlock;
4. #ifdef CONFIG_DEBUG_LOCK_ALLOC
5. # define LOCK_PADSIZE (offsetof(struct raw_spinlock, dep_map))
6.         struct {
7.             u8 padding[LOCK_PADSIZE];
8.             struct lockdep_map dep_map;
9.         };
10.    #endif
11. };
12. } spinlock_t;
```

Similarly, with the data structure, there must be corresponding operation functions. The commonly used spinlock interface functions are:

`int spin_lock_init(spinlock_t *lock)`: Spinlock initialization

**void spin\_lock(spinlock\_t \*lock):** Acquire spin lock

**void spin\_unlock(spinlock\_t \*lock):** Release spin lock

**int spin\_trylock(spinlock\_t \*lock):** Acquire the spin lock, return 0 if it is not acquired

**int spin\_is\_locked(spinlock\_t \*lock):** Check whether the spin lock has been acquired, return 0 if it has been acquired, and other values have not been acquired.

**void spin\_lock\_irq(spinlock\_t \*lock):** Disable local interrupts and acquire spin lock

**void spin\_unlock\_irq(spinlock\_t \*lock):** Resume the local interrupt and release the spin lock

**void spin\_lock\_irqsave(spinlock\_t \*lock,unsigned long flags):** Save the interrupt status, disable the local interrupt, and acquire the spin lock, flags are the interrupt status

**void spin\_unlock\_irqrestore(spinlock\_t\*lock, unsigned long flags):** The interruption is restored to the saved state, the local interruption is restored, and the spinlock is released.

### **Use spin lock, need to pay attention:**

#### **1) The critical section using the spin lock must not go to sleep.**

After the spin lock is successfully acquired, the kernel will disable the preemption mechanism. Suppose there is now a critical section A that acquires a spin lock, and then enters sleep to actively give up the right to use the CPU, and thread B starts running. The critical section of thread B also wants to access shared data, but the spin lock has been occupied, and the critical section of thread B has been waiting for the spin lock. At this time, kernel preemption is prohibited and critical section A cannot obtain the initiative. The lock is released, and a deadlock occurs.

#### **2) Avoid interrupt preemption in the critical area.** Assuming that a thread has acquired the spin lock and was preempted by the interrupt before it is released, if the interrupt also needs to obtain shared data and also apply for the spin lock, the interrupt will enter the waiting state, which is not good. Deadlock directly. The way to

avoid terminal preemption here is to use the `spin_lock_irq` function and `spin_lock_irqsave` function described above to apply for a spin lock and disable local interrupts.

- 3) **The critical section should be as short as possible.** The reason is explained above.

Example

```
1. spinlock_t lock;
2. spin_lock (&gpioled.lock); //上锁
3. /* 临界区 */
4. spin_unlock (&gpioled.lock); //解锁
```

The critical area is between locking and unlocking.

### Part 5.2.3: Semaphore

Compared with spin locks, semaphores have two advantages:

- 1) The semaphore can let the threads waiting for the semaphore go to sleep, reducing the CPU usage;
- 2) Semaphores support multiple threads to simultaneously access shared resources.

The semaphore structure is defined as follows:

```
1. struct semaphore {
2.     raw_spinlock_t lock;
3.     unsigned int count;
4.     struct list_head wait_list;
5. };
```

The element `count` refers to the number of threads that the semaphore supports simultaneously accessing shared resources.

Commonly used semaphore interface functions are:

`void sema_init(struct semaphore *sem, int val)` : Initialize the semaphore and set the signal value (the number of simultaneous accesses) to val.

`void down(struct semaphore *sem)` : Acquire the semaphore. When it fails, it will go to sleep and cannot be interrupted by the signal. It cannot be used for interruption.

`int down_trylock(struct semaphore *sem)` : Get the semaphore, will not enter sleep, return 0

successfully

**int down\_interruptible(struct semaphore \*sem)**: Acquire the semaphore. When it fails, it will go to sleep but it can be interrupted by the signal. It cannot be used for interruption. When the sleep is interrupted, it returns to a non-zero value.

**void up(struct semaphore \*sem)**: Release semaphore

Points to note when using semaphores:

- 1) After the thread sleeps, it will switch threads. If the critical area of the semaphore is very short, it will cause frequent thread switching, which will also bring a lot of overhead. Therefore, in contrast to spin locks, semaphores are not suitable for situations where the use of shared resources is short.
- 2) Interrupts cannot sleep, so you cannot apply for semaphores by going to sleep in interrupts;
- 3) When a semaphore allows multiple threads to access shared resources at the same time (count> 1), it cannot be used for exclusive access (mutual exclusive access means that only one thread can access shared resources at a time).

Example

```
1. struct semaphore sem;
2. sema_init(&sem, 1);
3. down(&sem);
4. /* 临界区 */
5. up(&sem);
```

## Part 5.3: Experiment

### Part 5.3.1: Schematic

Samp as Part 1.3.1

### Part 5.3.2: Device tree

Samp as Part 4.3.2

### Part 5.3.3: Driver Code

Take the driver code in the previous chapter as an example, first identify shared resources. For the driver of the led device, the shared resource is the led device. If multiple applications call this driver, the led device will be operated by multiple applications. Furthermore, it is the led device node, that is, the device file `/dev/gpio_leds`. When operating this device, no other applications can operate it. And the critical section starts from occupying this node (that is, the open function opens the device file `/dev/gpio_leds`) to releasing the node (close device file).

There are three mechanisms to experiment in this chapter, but there are a lot of repeated codes. For simplicity, I wrote all three mechanisms in one code. Use the macro definition switch to separate. Use petalinux to create a new driver named "`ax-concled-drv`", and enter the following code in `ax-concled-drv.c`:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/fs.h>
4. #include <linux/init.h>
5. #include <linux/ide.h>
6. #include <linux/types.h>
7. #include <linux/errno.h>
8. #include <linux/cdev.h>
9. #include <linux/of.h>
10. #include <linux/of_address.h>
11. #include <linux/of_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <asm/uaccess.h>
17. #include <asm/mach/map.h>
18. #include <asm/io.h>
19.
20. /* 设备节点名称 */
21. #define DEVICE_NAME      "gpio_leds"
22. /* 设备号个数 */
23. #define DEVID_COUNT      1
24. /* 驱动个数 */
25. #define DRIVE_COUNT       1
26. /* 主设备号 */
27. #define MAJOR1            1
28. /* 次设备号 */
29. #define MINOR1            0
30. /* LED 点亮时输入的值 */
31. #define ALINX_LED_ON      1
32. /* LED 熄灭时输入的值 */
33. #define ALINX_LED_OFF     0
34.
35. /* 原子变量开关 */
36. #define ATOMIC_T_ON
37. /* 自旋锁开关 */
38. // #define SPINKLOCK_T_ON
```

```
39. /* 信号量开关 */
40. #define SEMAPHORE_ON
41.
42. /* 把驱动代码中会用到的数据打包进设备结构体 */
43. struct alinx_char_dev{
44.     dev_t             devid;          //设备号
45.     struct cdev       cdev;           //字符设备
46.     struct class      *class;         //类
47.     struct device     *device;        //设备
48.     struct device_node *nd;           //设备树的设备节点
49.     int               alinx_led_gpio; //gpio 号
50.
51. #ifdef ATOMIC_T_ON
52.     atomic_t          lock;           //原子变量
53. #endif
54.
55. #ifdef SPINLOCK_T_ON
56.     spinlock_t        lock;           //自旋锁变量
57.     int               source_status; //资源占用状态
58. #endif
59.
60. #ifdef SEMAPHORE_ON
61.     struct semaphore   lock;
62. #endif
63. };
64. /* 声明设备结构体 */
65. static struct alinx_char_dev alinx_char = {
66.     .cdev = {
67.         .owner = THIS_MODULE,
68.     },
69. };
70.
71. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
72. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
73. {
74.     /* 应用程序调用了 open 函数表示需要调用共享资源 */
75. #ifdef ATOMIC_T_ON
76.     /* 通过判断原子变量的值来判断资源的占用状态 */
77.     if (!atomic_read(&alinkx_char.lock))
78.     {
79.         /* 若原子变量值为 0，则资源没有被占用，
80.            此时把原子变量加 1，表示之后资源就被占用了 */
81.         atomic_inc(&alinkx_char.lock);
82.     }
83.     else
84.     {
85.         /* 否则资源被占用，返回忙碌 */
86.         return -EBUSY;
87.     }
88. #endif
89.
90. #ifdef SPINLOCK_T_ON
91.     /* 获取自旋锁 */
92.     spin_lock(&alinkx_char.lock);
93.     /* 判断资源占用状态 */
94.     if(!alinkx_char.source_status)
95.     {
96.         /* 为 0 则未被占用，
97.            此时把状态值加 1，表示之后资源就被占用了 */
98.         alinkx_char.source_status++;
99.         /* 释放锁 */
100.        spin_unlock(&alinkx_char.lock);
101.    }
102.    else
103.    {
104.        /* 释放锁 */
105.        spin_unlock(&alinkx_char.lock);
106.        /* 否则资源被占用，返回忙碌 */
107.        return -EBUSY;
108.    }
109. #endif
110.
111. #ifdef SEMAPHORE_ON
112.     /* 获取信号量 */
113.     down(&alinkx_char.lock);
114. #endif
```

```
115.
116.     /* 设置私有数据 */
117.     file_p->private_data = &alinx_char;
118.     printk("gpio_test module open\n");
119.     return 0;
120. }
121.
122.
123. /* write 函数实现, 对应到 Linux 系统调用函数的 write 函数 */
124. static ssize_t gpio_leds_write(struct file *file_p, const char user *buf, size_t len, loff_t *loff_t_p)
125. {
126.     int retval;
127.     unsigned char databuf[1];
128.     /* 获取私有数据 */
129.     struct alinx_char_dev *dev = file_p->private_data;
130.
131.     retval = copy_from_user(databuf, buf, len);
132.     if(retval < 0)
133.     {
134.         printk("alink led write failed\r\n");
135.         return -EFAULT;
136.     }
137.
138.     if(databuf[0] == ALINX_LED_ON)
139.     {
140.         gpio_set_value(dev->alink_led_gpio, !!1);
141.     }
142.     else if(databuf[0] == ALINX_LED_OFF)
143.     {
144.         gpio_set_value(dev->alink_led_gpio, !!0);
145.     }
146.     else
147.     {
148.         printk("gpio_test para err\r\n");
149.     }
150.
151.     return 0;
152. }
153.
154. /* release 函数实现, 对应到 Linux 系统调用函数的 close 函数 */
155. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
156. {
157.     /* 应用程序调用 close 函数, 宣布资源已使用完毕 */
158. #ifdef ATOMIC_T_ON
159.     /* 原子变量恢复为 0, 表示资源已使用完毕 */
160.     atomic_set(&alink_char.lock, 0);
161. #endif
162.
163. #ifdef SPINKLOCK_T_ON
164.     /* 获取自旋锁 */
165.     spin_lock(&alink_char.lock);
166.     /* 资源占用状态恢复为 0, 表示资源已使用完毕 */
167.     alinx_char.source_status = 0;
168.     /* 释放锁 */
169.     spin_unlock(&alink_char.lock);
170. #endif
171.
172. #ifdef SEMAPHORE_ON
173.     /* 释放信号量 */
174.     up(&alink_char.lock);
175. #endif
176.
177.     printk("gpio_test module release\n");
178.     return 0;
179. }
180.
181. /* file_operations 结构体声明, 是上面 open、write 实现函数与系统调用函数对应的关键 */
182. static struct file_operations ax_char_fops = {
183.     .owner    = THIS_MODULE,
184.     .open     = gpio_leds_open,
185.     .write    = gpio_leds_write,
186.     .release  = gpio_leds_release,
187. };
188.
189. /* 模块加载时会调用的函数 */
190. static int init_gpio_led_init(void)
191. {
```

```
192.     /* 用于接受返回值 */
193.     u32 ret = 0;
194.
195. #ifdef ATOMIC_T_ON
196.     /* 设置原子变量为 0, 即资源为未被占用的状态 */
197.     atomic_set(&alinx_char.lock, 0);
198.#endif
199.
200. #ifdef SPINLOCK_T_ON
201.     /* 初始化自旋锁 */
202.     spin_lock_init(&alinx_char.lock);
203.     /* 初始化资源占用状态为 0, 意为资源没有被占用 */
204.     alinx_char.source_status = 0;
205.#endif
206.
207. #ifdef SEMAPHORE_ON
208.     /* 初始化信号量 */
209.     sema_init(alinx_char.lock, 1);
210.#endif
211.
212.     /* 获取设备节点 */
213.     alinx_char.nd = of_find_node_by_path("/alinkled");
214.     if(alinx_char.nd == NULL)
215.     {
216.         printk("alink_char node not find\r\n");
217.         return -EINVAL;
218.     }
219.     else
220.     {
221.         printk("alink_char node find\r\n");
222.     }
223.
224.     /* 获取节点中 gpio 标号 */
225.     alinx_char.alinx_led_gpio = of_get_named_gpio(alinx_char.nd, "alinkled-gpios", 0);
226.     if(alinx_char.alinx_led_gpio < 0)
227.     {
228.         printk("can not get alinkled-gpios");
229.         return -EINVAL;
230.     }
231.     printk("alinkled-gpio num = %d\r\n", alinx_char.alinx_led_gpio);
232.
233.     /* 申请 gpio 标号对应的引脚 */
234.     ret = gpio_request(alinx_char.alinx_led_gpio, "alinkled");
235.     if(ret != 0)
236.     {
237.         printk("can not request gpio\r\n");
238.     }
239.
240.     /* 把这个 io 设置为输出 */
241.     ret = gpio_direction_output(alinx_char.alinx_led_gpio, 1);
242.     if(ret < 0)
243.     {
244.         printk("can not set gpio\r\n");
245.     }
246.
247.     /* 注册设备号 */
248.     alloc_chrdev_region(&alink_char.devid, MINOR1, DEVID_COUNT, DEVICE_NAME);
249.
250.     /* 初始化字符设备结构体 */
251.     cdev_init(&alink_char.cdev, &ax_char_fops);
252.
253.     /* 注册字符设备 */
254.     cdev_add(&alink_char.cdev, alinx_char.devid, DRIVE_COUNT);
255.
256.     /* 创建类 */
257.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
258.     if(IS_ERR(alinx_char.class))
259.     {
260.         return PTR_ERR(alinx_char.class);
261.     }
262.
263.     /* 创建设备节点 */
264.     alinx_char.device = device_create(alinx_char.class, NULL,
265.                                         alinx_char.devid, NULL,
266.                                         DEVICE_NAME);
267.     if (IS_ERR(alinx_char.device))
268.     {
```

```

269.         return PTR_ERR(alinx_char.device);
270.     }
271.
272.     return 0;
273. }
274.
275. /* 卸载模块 */
276. static void exit gpio_led_exit(void)
277. {
278.     /* 释放 gpio */
279.     gpio_free(alinx_char.alinx_led_gpio);
280.
281.     /* 注销字符设备 */
282.     cdev_del(&alinx_char.cdev);
283.
284.     /* 注销设备号 */
285.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
286.
287.     /* 删除设备节点 */
288.     device_destroy(alinx_char.class, alinx_char.devid);
289.
290.     /* 删除类 */
291.     class_destroy(alinx_char.class);
292.
293.     printk("gpio_led_dev_exit_ok\n");
294. }
295.
296. /* 标记加载、卸载函数 */
297. module_init(gpio_led_init);
298. module_exit(gpio_led_exit);
299.
300. /* 驱动描述信息 */
301. MODULE_AUTHOR("Alinx");
302. MODULE_ALIAS("gpio_led");
303. MODULE_DESCRIPTION("CONCURRENT driver");
304. MODULE_VERSION("v1.0");
305. MODULE_LICENSE("GPL");

```

The parts that are different from the previous chapter are bolded.

The changes mainly focus on:

- 1) Add members to the structure definition
- 2) In the driver entry function, add initialization operation
- 3) Apply in the **open** function
- 4) Release in the **release** function

The macro-defined switch is divided into three parts, corresponding to atomic operations, spin locks, and semaphores.

Their usage is similar. Analyze in order

### Atomic operation:

Looking at the related part of the macro definition

**ATOMIC\_T\_ON**, the idea is:

- 1) **Line 52** defines an atomic variable and uses it as a sign of the shared resource usage status. 0 means not occupied, and other

values are already occupied.

- 2) **Line 197** initializes the atomic variable to 0 in the driver entry function, that is, the shared resource is not occupied.
- 3) **Line 77**, in the open function, first judge the resource occupancy status. If the atomic variable is not 0, it is occupied and returns to busy. Otherwise, the resource is free, make the atomic variable not 0, and then the open function returns 0, which means that the shared resource device node is successfully called.
- 4) **Line 160**, in the release function, the application releases the shared resource. At this time, the atomic variable is set to 0, and the shared resource is restored to an idle state.

In fact, under this kind of process, the shared resources have not been completely safe, because the if statement in line 77 is not an atomic operation. During the execution of this statement, it may be preempted or interrupted, causing the judgment result to be different from the actual have difference. The spin locks and semaphores that follow have no such problems.

### Spin lock:

Look at the related part of the macro definition **SPINKLOCK\_T\_ON**. If you want to compile the spinlock version, don't forget to uncomment the **Line 38** and comment the **Line 36** and **Line 40**.

As mentioned earlier, the critical section of the spin lock should be as short as possible. Normally, the critical section is from when the application calls **open** to when the application calls **close**. The time in between is not determined by the driver, it may be very long, all locked in the open function, unlocking in the release function will risk a very long critical section. This can be achieved with the help of the idea of flag used in the above atomic variable experiment:

- 1) **Line 56** defines a spin lock, and **line 57** defines a flag **source\_status** to record the usage status of the resource, 0 means not occupied, and other values are already occupied.
- 2) **Line 202** initializes the spin lock in the driver entry function, and initializes the resource state variable to the idle state 0.
- 3) **Line 92** is locked. The critical area is only a judgment resource status value. If it is not occupied, it is marked as occupied and then unlocked. If it is occupied, it is directly unlocked and returns to **busy**. In this way, even if the resource is occupied, the thread will not wait forever.
- 4) **Line 165**, in the release function, after acquiring the lock, set the resource occupation state to 0, and then unlock it.

#### **signal:**

The usage of the semaphore is very simple, because the critical section of the semaphore is long, so we can safely lock in open and unlock in release. There is no need to use flags. Of course, how long the critical region is depends on the application.

#### **Part 5.3.4: Test code**

The test code is slightly modified based on the previous chapter. Create a new QT project as "**axledlong\_test**", create a new **main.c**, and enter the following code:

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <unistd.h>
4. #include <fcntl.h>
5.
6. int main(int argc, char **argv)
7. {
8.     int fd;
9.     char buf;
10.    int count;
11.
12.    if(3 != argc)
13.    {
14.        printf("none para\n");
15.        return -1;
16.    }
17.
18.    fd = open(argv[1], O_RDWR);
```

```
19.     if(fd < 0)
20.     {
21.         printf("Can't open file %s\r\n", argv[1]);
22.         return -1;
23.     }
24.
25.     if(!strcmp("on",argv[2]))
26.     {
27.         printf("ps_led1 on\r\n");
28.         buf = 1;
29.         write(fd, &buf, 1);
30.     }
31.     else if(!strcmp("off",argv[2]))
32.     {
33.         printf("ps_led1 off\r\n");
34.         buf = 0;
35.         write(fd, &buf, 1);
36.     }
37.     else
38.     {
39.         printf("wrong para\r\n");
40.         return -2;
41.     }
42.
43.     count = 20;
44.     while(count --)
45.     {
46.         sleep(1);
47.     }
48.
49.     close(fd);
50.     return 0;
51. }
```

Add code from **line 43** to **line 47**, and use **sleep** for 20 seconds to simulate the following 20 seconds of resource occupation. Others are the same as before.

### Part 5.3.5: Run test

The step of loading the driver is skipped. As shown below:

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-concled-dev.ko
ax_concled_dev: loading out-of-tree module taints kernel.
alinx_char node find
alinkled-gpio num = 899
root@ax_peta:/mnt#
```

Use the following command to open the test program:

```
./axledlong_test /dev/gpio_leds on&
./axledlong_test /dev/gpio_leds off
```

"&" means running the app in the background, you can use the top command to view the programs running in the background.

```
root@ax_peta:/mnt/build-axedlong_test-ZYNQ-Debug# ./axedlong_test /dev/gpio_leds on&
[1] 1328
gpio_test module open
ps_ledl on
root@ax_peta:/mnt/build-axedlong_test-ZYNQ-Debug# ./axedlong_test /dev/gpio_leds off
Can't open file /dev/gpio_leds
root@ax_peta:/mnt/build-axedlong_test-ZYNQ-Debug# gpio_test module release

[1]+  Done                  ./axedlong_test /dev/gpio_leds on
root@ax_peta:/mnt/build-axedlong_test-ZYNQ-Debug# ./axedlong_test /dev/gpio_leds off
gpio_test module open
ps_ledl off
```

It can be seen that if you execute **app** to turn on the led first, this thread occupies the device node. Execute **off** again, and the app returns "**Can't open file /dev/gpio\_leds**" instead of turning off the led directly. When "**gpio\_test module release**" is output, it means that the app thread at the beginning has released the resources, and then execute **off**, the led is closed.

This is the test state for atomic variables. The steps of the other two experiments are also the same. The phenomenon of spin lock is consistent with the experimental phenomenon of atomic variables. The semaphore is slightly different, because in the implementation of the semaphore, it will sleep to wait for resources, so when it is **off**, it will not directly return an error, but will wait for the output of "**gpio\_test module release**", and then automatically turn off the led.

## Part 6: gpio input

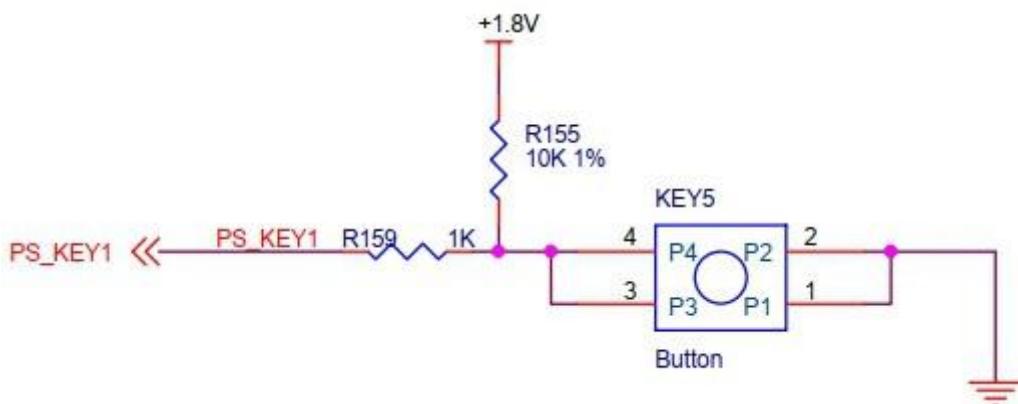
In this chapter, we will do a **gpio** output experiment to consolidate the **pinctrl** and **gpio** subsystems, and also pave the way for future learning.

The most classic example of **gpio** output is the key. Let's use the key to do a simple **gpio** input experiment. The goal of the experiment is that the application program reads the key state through the read function, and if the key is pressed, the LED level is reversed once. Use the keys to control the LED on and off.

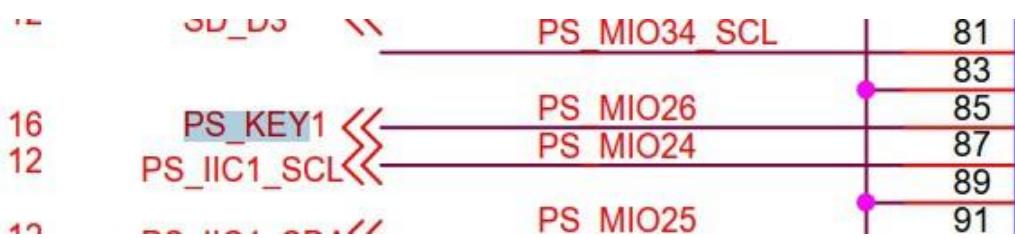
### Part 6.1: Schematic

The led part is the same as **Part 1.3.1**.

For the key part, use **ps\_key1** on the FPGA development board, which corresponds to **KEY2** on the schematic, and the other end of the key is grounded. Press the key, **MIOy\_KEY1** will be pulled down.



Then find the IO connected to PS\_KEY1, which is MIO26.



## Part 6.2: Device tree

Open the **system-user.dtsi** file and add the following nodes to the root node:

```
1. #include <dt-bindings/pinctrl/pinctrl-zynqmp.h>
2.
3. /include/ "system-conf.dtsi"
4.
5. &gpio {
6.     status = "okay";
7. };
8.
9. &pinctrl0 {
10.    status = "okay";
11.
12.    pinctrl_led_default: led-default {
13.        mux {
14.            groups = "gpio0_40_grp";
15.            function = "gpio0";
16.        };
17.
18.        conf {
19.            pins = "MIO40";
20.            io-standard = <IO_STANDARD_LVCMOS33>;
21.            bias-disable;
22.            slew-rate = <SLEW_RATE_SLOW>;
23.        };
24.    };
25.
26.    pinctrl_key_default: key-default {
27.        mux {
28.            groups = "gpio0_26_grp";
29.            function = "gpio0";
30.        };
31.
32.        conf {
33.            pins = "MIO26";
34.            io-standard = <IO_STANDARD_LVCMOS33>;
35.            bias-high-impedance;
36.            slew-rate = <SLEW_RATE_SLOW>;
37.        };
38.    };
39. };
40.
41. / {
42.     alinxled {
43.         compatible = "alinx-led";
44.         pinctrl-names = "default";
45.         pinctrl-0 = <&pinctrl_led_default>;
46.         alinxled-gpios = <&gpio 40 0>;
47.     };
48.
49.     alinxkey {
50.         compatible = "alinx-key";
51.         pinctrl-names = "default";
52.         pinctrl-0 = <&pinctrl_key_default>;
53.         alinxkey-gpios = <&gpio 26 0>;
54.     };
55. };
```

The led-related parts are the same as before.

The IO used by the key is **MIO26**, so **groups = "gpio0\_26\_grp"**, **pins = "MIO26"**, **alinxkey-gpio = <&gpio0 260>**, the electrical characteristics of the input pins are configured as **bias-high-impedance**.

### Part 6.3: Driver code

Use petalinux to create a new driver named "**ax-key-dev**", and enter the following code in **ax-key-dev**:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/fs.h>
4. #include <linux/init.h>
5. #include <linux/ide.h>
6. #include <linux/types.h>
7. #include <linux/errno.h>
8. #include <linux/cdev.h>
9. #include <linux/of.h>
10. #include <linux/of_address.h>
11. #include <linux/of_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <linux/delay.h>
17. #include <asm/uaccess.h>
18. #include <asm/mach/map.h>
19. #include <asm/io.h>
20.
21. /* 设备节点名称 */
22. #define DEVICE_NAME      "gpio_key"
23. /* 设备号个数 */
24. #define DEVID_COUNT      1
25. /* 驱动个数 */
26. #define DRIVE_COUNT       1
27. /* 主设备号 */
28. #define MAJOR1
29. /* 次设备号 */
30. #define MINOR1           0
31.
32. /* 把驱动代码中会用到的数据打包进设备结构体 */
33. struct alinx_char_dev{
34.     dev_t          devid;          //设备号
35.     struct cdev    cdev;          //字符设备
36.     struct class   *class;        //类
37.     struct device   *device;       //设备
38.     struct device_node *nd;        //设备树的设备节点
39.     int            alinx_key_gpio; //gpio 号
40. };
41. /* 声明设备结构体 */
42. static struct alinx_char_dev alinx_char = {
43.     .cdev = {
44.         .owner = THIS_MODULE,
45.     },
46. };
47.
48. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
49. static int gpio_key_open(struct inode *inode_p, struct file *file_p)
50. {
51.     /* 设置私有数据 */
52.     file_p->private_data = &alink_char;
```

```
53.     printk("gpio_test module open\n");
54.     return 0;
55. }
56.
57.
58. /* write 函数实现，对应到 Linux 系统调用函数的 write 函数 */
59. static ssize_t gpio_key_read(struct file *file_p, char user *buf, size_t len, loff_t *loff_t_p )
60. {
61.     int ret = 0;
62.     /* 返回按键的值 */
63.     unsigned int key_value = 0;
64.     /* 获取私有数据 */
65.     struct alinx_char_dev *dev = file_p->private_data;
66.
67.     /* 检查按键是否被按下 */
68.     if(0 == gpio_get_value(dev->alinkx_key_gpio))
69.     {
70.         /* 按键被按下 */
71.         /* 防抖 */
72.         mdelay(50);
73.         /* 等待按键抬起 */
74.         while(!gpio_get_value(dev->alinkx_key_gpio));
75.         key_value = 1;
76.     }
77.     else
78.     {
79.         /* 按键未被按下 */
80.     }
81.     /* 返回按键状态 */
82.     ret = copy_to_user(buf, &key_value, sizeof(key_value));
83.
84.     return ret;
85. }
86.
87. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
88. static int gpio_key_release(struct inode *inode_p, struct file *file_p)
89. {
90.     printk("gpio_test module release\n");
91.     return 0;
92. }
93.
94. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
95. static struct file_operations ax_char_fops = {
96.     .owner    = THIS_MODULE,
97.     .open     = gpio_key_open,
98.     .read     = gpio_key_read,
99.     .release  = gpio_key_release,
100. };
101.
102. /* 模块加载时会调用的函数 */
103. static int initgpio_key_init(void)
104. {
105.     /* 用于接受返回值 */
106.     u32 ret = 0;
107.
108.     /* 获取设备节点 */
109.     alinx_char.nd = of_find_node_by_path("/alinkxkey");
110.     if(alinx_char.nd == NULL)
111.     {
112.         printk("alinkx_char node not find\r\n");
113.         return -EINVAL;
114.     }
115.     else
116.     {
117.         printk("alinkx_char node find\r\n");
118.     }
119.
120.     /* 获取节点中 gpio 标号 */
121.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinkxkey-gpios", 0);
122.     if(alinx_char.alinx_key_gpio < 0)
123.     {
124.         printk("can not get alinkxkey-gpios");
125.         return -EINVAL;
126.     }
127.     printk("alinkxkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
128.
```

```
129. /* 申请 gpio 标号对应的引脚 */
130. ret = gpio_request(alinx_char.alinx_key_gpio, "alinxkey");
131. if(ret != 0)
132. {
133.     printk("can not request gpio\r\n");
134.     return -EINVAL;
135. }
136.
137. /* 把这个 io 设置为输入 */
138. ret = gpio_direction_input(alinx_char.alinx_key_gpio);
139. if(ret < 0)
140. {
141.     printk("can not set gpio\r\n");
142.     return -EINVAL;
143. }
144.
145. /* 注册设备号 */
146. alloc_chrdev_region(&alinx_char.devid, MINOR1, DEVID_COUNT, DEVICE_NAME);
147.
148. /* 初始化字符设备结构体 */
149. cdev_init(&alinx_char.cdev, &ax_char_fops);
150.
151. /* 注册字符设备 */
152. cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
153.
154. /* 创建类 */
155. alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
156. if(IS_ERR(alinx_char.class))
157. {
158.     return PTR_ERR(alinx_char.class);
159. }
160.
161. /* 创建设备节点 */
162. alinx_char.device = device_create(alinx_char.class, NULL,
163.                                     alinx_char.devid, NULL,
164.                                     DEVICE_NAME);
165. if (IS_ERR(alinx_char.device))
166. {
167.     return PTR_ERR(alinx_char.device);
168. }
169.
170. return 0;
171. }
172.
173. /* 卸载模块 */
174. static void exitgpio_key_exit(void)
175. {
176.     /* 释放 gpio */
177.     gpio_free(alinx_char.alinx_key_gpio);
178.
179.     /* 注销字符设备 */
180.     cdev_del(&alinx_char.cdev);
181.
182.     /* 注销设备号 */
183.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
184.
185.     /* 删除设备节点 */
186.     device_destroy(alinx_char.class, alinx_char.devid);
187.
188.     /* 删除类 */
189.     class_destroy(alinx_char.class);
190.
191.     printk("gpio_key_dev_exit_ok\r\n");
192. }
193.
194. /* 标记加载、卸载函数 */
195. module_init(gpio_key_init);
196. module_exit(gpio_key_exit);
197.
198. /* 驱动描述信息 */
199. MODULE_AUTHOR("Alinx");
200. MODULE_ALIAS("gpio_key");
201. MODULE_DESCRIPTION("GPIO OUT driver");
202. MODULE_VERSION("v1.0");
203. MODULE_LICENSE("GPL");
```

It is very similar to the **Part 4**, the changed part has been bolded.

**Line 138** was originally set as output and changed to input.

**Line 58~82** was original **write function** and changed to **read function**. Use the **gpio\_get\_value** function to read the IO level every time. If a low level is detected, the button is pressed, and after the button is pressed, the normal delay debounces, waiting to be lifted.

The read function finally returns the read level to the user **buf**.

## Part 6.4: Test code

Create a new QT project named "**ax-key-test**", create a new **main.c**, and enter the following code:

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <unistd.h>
4. #include <fcntl.h>
5.
6. int main(int argc, char *argv[])
7. {
8.     int fd, fd_1 ,ret;
9.     char *filename, led_value = 0;
10.    unsigned int key_value;
11.
12.    if(argc != 2)
13.    {
14.        printf("Error Usage\r\n");
15.        return -1;
16.    }
17.
18.    filename = argv[1];
19.    fd = open(filename, O_RDWR);
20.    if(fd < 0)
21.    {
22.        printf("file %s open failed\r\n", argv[1]);
23.        return -1;
24.    }
25.
26.    while(1)
27.    {
28.        ret = read(fd, &key_value, sizeof(key_value));
29.        if(ret < 0)
30.        {
31.            printf("read failed\r\n");
32.            break;
33.        }
34.        if(1 == key_value)
35.        {
36.            printf("ps_key1 press\r\n");
37.            led_value = !led_value;
38.
39.            fd_1 = open("/dev/gpio_leds", O_RDWR);
40.            if(fd_1 < 0)
41.            {
42.                printf("file /dev/gpio_leds open failed\r\n");
43.                break;
44.            }
45.
46.            ret = write(fd_1, &led_value, sizeof(led_value));
47.            if(ret < 0)
48.            {
```

```
49.         printf("write failed\r\n");
50.         break;
51.     }
52.
53.     ret = close(fd_1);
54.     if(ret < 0)
55.     {
56.         printf("file /dev/gpio_leds close failed\r\n");
57.         break;
58.     }
59. }
60.
61.
62. ret = close(fd);
63. if(ret < 0)
64. {
65.     printf("file %s close failed\r\n", argv[1]);
66.     return -1;
67. }
68.
69. return 0;
70. }
```

Because to light up the led, you need to use the device node of the previous **led**. Here, use the device node “ `/dev/gpio_leds` ” directly. Remember to load the led driver when testing.

In the while loop on **line 26**, the read function is continuously called to **read** the key status. Once it is read that the key is pressed, write to the **io** of the led, and write the opposite value each time, to achieve that each time the key is pressed, the led state is reversed.

## Part 6.5: Run test

Because the device tree is modified, the image.ub file in the SD card must be updated. Because the led and key are used at the same time, two drivers must be loaded. The led driver can use any of the above chapters. Proceed as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/ilinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-concled-drv.ko
```

```
insmod ./ax-key-drv.ko  
cd ./build-ax-key-test-IDE_5_7_1_GCC_64bit-Debug/  
../ax-key-test /dev/gpio_key
```

The IP and path are adjusted according to the actual situation.

The operation result is as follows:

```
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt  
root@ax_peta:~# cd /mnt  
root@ax_peta:/mnt# mkdir /tmp/qt  
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt  
random: fast init done  
EXT4-fs (loop0): recovery complete  
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)  
root@ax_peta:/mnt# cd /tmp/qt  
root@ax_peta:/tmp/qt# source ./qt_env_set.sh  
/tmp/qt  
root@ax_peta:/tmp/qt# cd /mnt  
root@ax_peta:/mnt# insmod ./ax-concled-dev.ko  
ax_concled_dev: loading out-of-tree module taints kernel.  
alinx_char node find  
alinkled-gpio num = 899  
root@ax_peta:/mnt# insmod ./ax-key-  
ax-key-dev.ko ax-key-test/  
root@ax_peta:/mnt# insmod ./ax-key-dev.ko  
alink_char node find  
alinkxkey-gpio num = 949
```

After executing the app, the first sentence "[gpio\\_test module open](#)" printed out is printed when the [open](#) function opens [/dev/gpio\\_key](#), and then press the key, the LED status is reversed once and printed [ps\\_key1 press, gpio\\_test module open , Gpio\\_test module release](#) three lines of information

## Part 7: Timer

This chapter briefly introduces the timers provided by the Linux kernel and their usage.

### Part 7.1: Timers in the Linux kernel

The realization of the timer in the kernel depends on the hardware timer. The hardware timer provides a fixed frequency clock source and generates interrupts. The system uses this interrupt for timing. The timer function in the kernel is implemented on the basis of this timing.

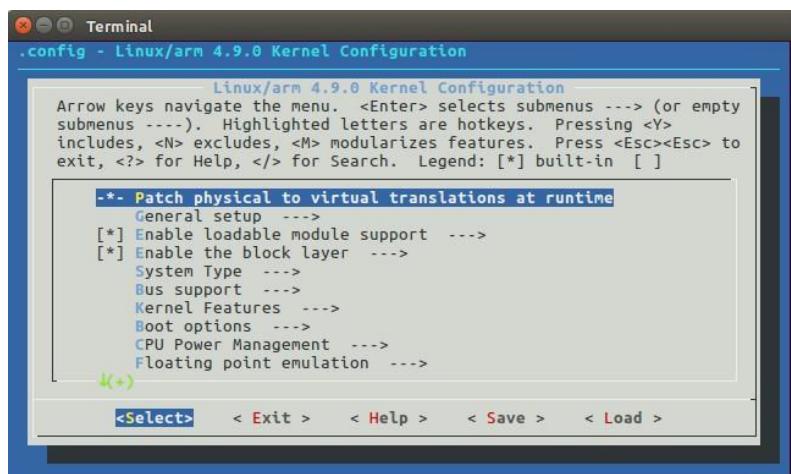
#### Part 7.1.1: Timers in the Linux kernel

The frequency at which system interrupts are generated is the system frequency, also called system beat, which is the number of beats per second. The system beat can be set. The setting method in petalinux is:

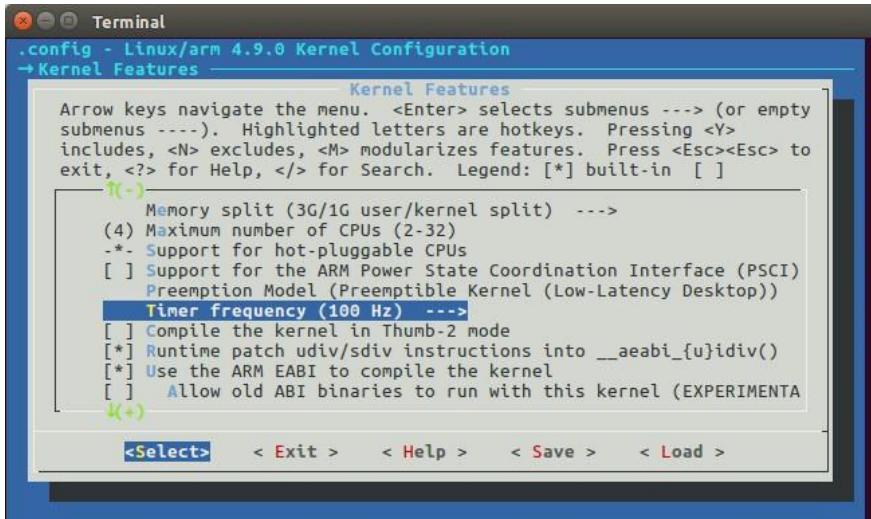
Open the terminal and enter the command in the petalinux project directory:

```
petalinux-config -c kernel
```

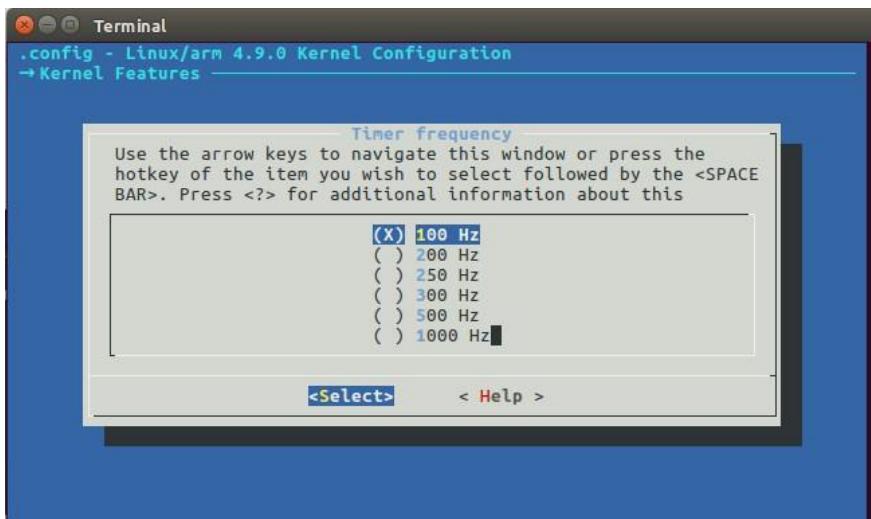
After waiting for a period of time, the interactive interface will pop up as shown below:



Use the up and down direction keys to move the option to "Kernel Features" and press Enter, as shown in the figure below:



Then press the arrow keys to select the option "Timer frequency (100 Hz)" and press Enter (this option is hidden below, not in the initial interface), you can see the system frequency options:



The default selection here is 100Hz, 100 beats per second.

The higher the system frequency, the higher the clock accuracy, 100Hz time accuracy can reach 10ms, and 1000Hz can reach 1ms. At the same time, the frequency of interrupt generation increases, the greater the burden on the **cpu**. This needs to be selected according to the actual situation, our experiment using 100Hz is enough.

The system clock is defined in "[include/asm-generic/param.h](#)",

which defines **HZ** for the macro, which will eventually correspond to the system clock value we set.

### Part 7.1.2: Number of beats

The method of system timing is to record the number of beats. System running time = number of beats/system frequency, in seconds.

The kernel uses the global variable **jiffies** to record the number of beats, which is defined in the file “[include/linux/jiffies.h](#)”:

```
extern u64 jiffy_data jiffies_64;  
extern unsigned long volatile jiffy_data jiffies;
```

**jiffies\_64** is used for 64-bit systems and **jiffies** is used for 32-bit systems. Since it is a variable used for counting, there is a risk of overflow. Not to mention 64-bit, 32-bit jiffies will overflow and wrap around 500 days at a system frequency of 100Hz, and wrap around 50 days at a frequency of 1000Hz. What problems will wrap bring, for example:

```
1. unsigned long time_mark = jiffies + HZ * 5;  
2.  
3. /* 业务处理 */  
4.  
5. if(time_mark > jiffies)  
6. {  
7.     /* 未超时 */  
8. }  
9. else  
10. {  
11.    /* 超时 */  
12. }
```

In this example, first set a time **time\_mark** to the current number of beats plus 5 seconds of beats, which is the value that the system beat jiffies will reach after 5 seconds. The purpose of this sample code is to perform timeout processing if the business processing time exceeds 5 seconds. In extreme cases, for example, **jiffies** is 0xFFFFFDA7 at this time, and **time\_mark** is equal to 0xFFFFFFF9B. If the business processing time takes 7 seconds, **jiffies** will be reversed

because it exceeds the maximum value of 0xFFFFFFFF, and start counting again from 0 to 0x64, which is far less than time\_mark, clearly timed out, but entered the condition of not timed out.

In this regard, the kernel cleverly avoids the problem of wraparound by forcing the data to convert unsigned numbers into signed numbers. Instead of using traditional comparison operators to compare, the following macros are used instead:

```
#define time_after(unknown,known) ((long)(known) - (long)(unknown)<0)
#define time_before(unkonwn,known) ((long)(unknown) - (long)(known)<0)
#define time_after_eq(unknown,known) ((long)(unknown) - (long)(known)>=0)
#define time_before_eq(unknown,known) ((long)(known) -(long)(unknown)>=0)
```

Take **time\_after\_eq** as an example, the parameter **unknow** generally refers to **jiffies**, and **know** is usually the time we set, such as **time\_mark** in the above example. **after\_eq** is greater than or equal to, if **unknow** is greater than or equal to **know**, that is, jiffies is greater than **time\_mark**, it returns **true**, which is time out for the above routine.

### Part 7.1.3: Kernel timer and related functions

#### ➤ Timer\_list

The timer in the Linux kernel is represented by the data structure “**timer\_list**”. Defined in “[include/linux/timer.h](#)”:

```
1. struct timer_list {
2.     /*
3.      * All fields that change during normal runtime grouped to the
4.      * same cacheline
5.      */
6.     struct hlist_node    entry;
7.     unsigned long        expires;
8.     void                (*function)(unsigned long);
9.     unsigned long        data;
10.    u32                 flags;
11.
12. #ifdef CONFIG_TIMER_STATS
13.    int                 start_pid;
14.    void                *start_site;
15.    char                start_comm[16];
16. #endif
17. #ifdef CONFIG_LOCKDEP
18.    struct lockdep_map lockdep_map;
19. #endif
20. }
```

Important member introduction:

**Expires:** Set the number of beats at the time when the timer expires. If we need to define a 10s timer, then expires should be equal to jiffies+10\*HZ. For convenience, the kernel also provides a series of functions that interchange milliseconds, subtle, nanoseconds, and jiffies:

```
/*Millisecond swap */
int jiffies_to_msecs(const unsigned long j)
long msecs_to_jiffies(const unsigned int m)

/*Microsecond sw*/
int jiffies_to_usecs(const unsigned long j)
long usecs_to_jiffies(const unsigned int u)

/*Nanosecond swap */
u64 jiffies_to_nsecs(const unsigned long j)
unsigned long nsecs_to_jiffies(u64 n)
```

**function:** The function pointer is returned when the timer timeout. The operations that need to be done after the timeout are implemented in this function.

**data:** used to set the input parameters of the function.

PS: The members in rows 9 to 15 have been deleted in the new kernel.

➤ **Timer related functions are:**

**1) void init\_timer(struct timer\_list \*timer)**

Initialize the timer, the input parameter **timer** is the timer that needs to be initialized

**2) void add\_timer(struct timer\_list \*timer)**

Register the timer with the kernel. After registration, the timer starts to run. The input parameter **timer** is the timer that needs to be registered.

**3) int del\_timer(struct timer\_list \* timer)**

---

Delete the timer, no matter whether the **timer** is running or not, the running timer will stop timing directly. The input parameter timer is the timer to be deleted. If the deleted timer is running, it returns 1, otherwise it returns 0.

#### 4) **int del\_timer\_sync(struct timer\_list \*timer)**

Delete the timer, the running timer will wait until the use is completed before deleting. The input parameter **timer** is the timer to be deleted. If the deleted timer is running, it returns 1, otherwise it returns 0.

#### 5) **int mod\_timer(struct timer\_list \*timer, unsigned long expires)**

Modify the expires of the timer, that is, the timeout period. If the timer is not running, activate the timer. The input parameter **timer** is the timer that needs to be modified. **expires** is the modified timeout period.

## Part 7.2: Experiment

The experiment is to use a timer to make the led flashing, and the application can set the led flashing cycle by inputting parameters .

### Part 7.2.1: Schematic

Same as **Part 1.3.1**

### Part 7.2.2: Device tree

Same as **Part 4.3.2**

### Part 7.2.3: Driver code

Use petalinux to create a new driver named "**ax-timer-dev**" and enter the following code in **ax-timer -dev**:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/fs.h>
4. #include <linux/init.h>
```

```
5. #include <linux/ide.h>
6. #include <linux/types.h>
7. #include <linux/errno.h>
8. #include <linux/cdev.h>
9. #include <linux/of.h>
10. #include <linux/of_address.h>
11. #include <linux/of_gpio.h>
12. #include <linux/device.h>
13. #include <linux/delay.h>
14. #include <linux/init.h>
15. #include <linux/gpio.h>
16. #include <linux/semaphore.h>
17. #include <linux/timer.h>
18. #include <asm/uaccess.h>
19. #include <asm/io.h>
20.
21. /* 设备节点名称 */
22. #define DEVICE_NAME      "timer_led"
23. /* 设备号个数 */
24. #define DEVID_COUNT      1
25. /* 驱动个数 */
26. #define DRIVE_COUNT       1
27. /* 主设备号 */
28. #define MAJOR_U
29. /* 次设备号 */
30. #define MINOR_U          0
31.
32.
33. /* 把驱动代码中会用到的数据打包进设备结构体 */
34. struct alinx_char_dev{
35.     dev_t             devid;           //设备号
36.     struct cdev       cdev;            //字符设备
37.     struct class     *class;          //类
38.     struct device    *device;         //设备
39.     struct device_node *nd;           //设备树的设备节点
40.     int               alinx_led_gpio; //gpio 号
41.     char              led_status;     //gpio 状态
42.     unsigned int      time_count;    //定时器时间
43.     struct timer_list timer;         //定时器
44. };
45. /* 声明设备结构体 */
46. static struct alinx_char_dev alinx_char = {
47.     .cdev = {
48.         .owner = THIS_MODULE,
49.     },
50. };
51.
52. void timer_function(struct timer_list *timer)
53. {
54.     /* 反转 led 状态 */
55.     alinx_char.led_status = !alinkx_char.led_status;
56.     /* 设置 led */
57.     gpio_set_value(alinx_char.alinx_led_gpio, alinx_char.led_status);
58.     /* 重新开始计时 */
59.     mod_timer(timer, jiffies + msecs_to_jiffies(alinx_char.time_count));
60. }
61.
62. /* open 函数实现, 对应到 Linux 系统调用函数的open 函数 */
63. static int timer_led_open(struct inode *inode_p, struct file *file_p)
64. {
65.     printk("gpio_test module open\n");
66.     return 0;
67. }
68.
69.
70. /* write 函数实现, 对应到Linux 系统调用函数的 write 函数 */
```

```
71. static ssize_t timer_led_write(struct file *file_p, const char user *buf, size_t l
    en, loff_t *loff_t_p)
72. {
73.     int retval;
74.     /* 获取用户数据 */
75.     retval = copy_from_user(&alinx_char.time_count, buf, len);
76.     /* 设置好timer 后先点亮 led */
77.     alinx_char.led_status = 1;
78.     gpio_set_value(alinx_char.alinx_led_gpio, alinx_char.led_status);
79.     /* 开启 timer */
80.     mod_timer(&alinx_char.timer, jiffies + msecs_to_jiffies(alinx_char.time_count));
81.
82.     return 0;
83. }
84.
85. /* release 函数实现，对应到Linux 系统调用函数的close 函数 */
86. static int timer_led_release(struct inode *inode_p, struct file *file_p)
87. {
88.     printk("gpio_test module release\n");
89.     /* 删除定时器 */
90.     del_timer_sync(&alinx_char.timer);
91.     return 0;
92. }
93.
94. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
95. static struct file_operations ax_char_fops = {
96.     .owner    = THIS_MODULE,
97.     .open     = timer_led_open,
98.     .write    = timer_led_write,
99.     .release  = timer_led_release,
100. };
101.
102. /* 模块加载时会调用的函数 */
103. static int init timer_led_init(void)
104. {
105.     /* 用于接受返回值 */
106.     u32 ret = 0;
107.
108.     /* 获取 led 设备节点 */
109.     alinx_char.nd = of_find_node_by_path("/alinkled");
110.     if(alinx_char.nd == NULL)
111.     {
112.         printk("alink_char node not find\r\n");
113.         return -EINVAL;
114.     }
115.     else
116.     {
117.         printk("alink_char node find\r\n");
118.     }
119.
120.     /* 获取节点中 gpio 标号 */
121.     alinx_char.alinx_led_gpio = of_get_named_gpio(alinx_char.nd, "alinkled-gpios",
122.     0);
123.     if(alinx_char.alinx_led_gpio < 0)
124.     {
125.         printk("can not get alinxled-gpios");
126.         return -EINVAL;
127.     }
128.     printk("alinkled-gpio num = %d\r\n", alinx_char.alinx_led_gpio);
129.
130.     /* 申请 gpio 标号对应的引脚 */
131.     ret = gpio_request(alinx_char.alinx_led_gpio, "alinkled");
132.     if(ret != 0)
133.     {
134.         printk("can not request gpio\r\n");
135.     }
```

```
136.     /* 把这个io 设置为输出 */
137.     ret = gpio_direction_output(alinx_char.alinx_led_gpio, 1);
138.     if(ret < 0)
139.     {
140.         printk("can not set gpio\r\n");
141.     }
142.
143.     /* 注册设备号 */
144.     alloc_chrdev_region(&alinx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
145.
146.     /* 初始化字符设备结构体 */
147.     cdev_init(&alinx_char.cdev, &ax_char_fops);
148.
149.     /* 注册字符设备 */
150.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
151.
152.     /* 创建类 */
153.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
154.     if(IS_ERR(alinx_char.class))
155.     {
156.         return PTR_ERR(alinx_char.class);
157.     }
158.
159.     /* 创建设备节点 */
160.     alinx_char.device = device_create(alinx_char.class, NULL,
161.                                         alinx_char.devid, NULL,
162.                                         DEVICE_NAME);
163.     if (IS_ERR(alinx_char.device))
164.     {
165.         return PTR_ERR(alinx_char.device);
166.     }
167.
168.     /* 设置定时器回掉函数&初始化定时器 */
169.     timer_setup(&alinx_char.timer, timer_function, NULL);
170.
171.     return 0;
172. }
173.
174. /* 卸载模块 */
175. static void_exit timer_led_exit(void)
176. {
177.     /* 释放 gpio */
178.     gpio_free(alinx_char.alinx_led_gpio);
179.
180.     /* 注销字符设备 */
181.     cdev_del(&alinx_char.cdev);
182.
183.     /* 注销设备号 */
184.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
185.
186.     /* 删除设备节点 */
187.     device_destroy(alinx_char.class, alinx_char.devid);
188.
189.     /* 删除类 */
190.     class_destroy(alinx_char.class);
191.
192.     printk("timer_led_dev_exit_ok\n");
193. }
194.
195. /* 标记加载、卸载函数 */
196. module_init(timer_led_init);
197. module_exit(timer_led_exit);
198.
199. /* 驱动描述信息 */
200. MODULE_AUTHOR("Alinx");
201. MODULE_ALIAS("gpio_led");
202. MODULE_DESCRIPTION("TIMER LED driver");
203. MODULE_VERSION("v1.0");
```

```
|204. MODULE_LICENSE("GPL");
```

Also pay attention to the bold part, the others are the regular character device registration framework.

**Line 42** defines a char number to record the state of led.

**Line 43** defines an unsigned int to record the time of the timer.

**Line 44** defines a timer.

**Lines 53-61** are the return function of the timer, which will be executed after the timer expires. In this function, perform a state inversion of the led and restart the timing.

**Line 72** open function, set the **timer** time according to the value input by the application, and turn on the **timer**, using the **mod\_timer** function.

**Line 91** release function to delete the timer

**Line 169**, in the entry function, set the timer's return function to **timer\_function()** and initialize the timer.

#### Part 7.2.4: Test code

Create a new **QT** project named "**timer-test**", create a new **main.c**, and enter the following code:

```
1. #include "stdio.h"
2. #include "unistd.h"
3. #include "sys/types.h"
4. #include "sys/stat.h"
5. #include "fcntl.h"
6. #include "stdlib.h"
7. #include "string.h"
8. #include "linux/ioctl.h"
9.
10. int main(int argc, char *argv[])
11. {
12.     int fd, ret;
13.     char *filename;
14.     unsigned int interval_new, interval_old = 0;
15.
16.     if(argc != 2)
17.     {
18.         printf("Error Usage!\r\n");
19.         return -1;
20.     }
21.
22.     filename = argv[1];
23.
24.     fd = open(filename, O_RDWR);
25.     if(fd < 0)
26.     {
27.         printf("can not open file %s\r\n", filename);
```

```
28.         return -1;
29.     }
30.
31.     while(1)
32.     {
33.         printf("Input interval:");
34.         scanf("%d", &interval_new);
35.
36.         if(interval_new != interval_old)
37.         {
38.             interval_old = interval_new;
39.             ret = write(fd, &interval_new, sizeof(interval_new));
40.             if(ret < 0)
41.             {
42.                 printf("write failed\r\n");
43.             }
44.             else
45.             {
46.                 printf("interval refreshed!\r\n");
47.             }
48.         }
49.         else
50.         {
51.             printf("same interval!");
52.         }
53.     }
54.     close(fd);
55. }
```

The content is very simple.

**Line 34** waits for the user to enter the time interval.

**Line 36** judgment time interval, if not changed, ignore it. If it changes, call the write function to restart the timer with a new interval.

### Part 7.2.5: Run test

The test steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/ilinx/work /mnt //IP The path is adjusted  
according to the actual situationcd /mnt  
mkdir /tmp/qt  
mount qt_lib.img /tmp/qt  
cd /tmp/qt  
source ./qt_env_set.sh  
cd /mnt  
. /ax-timer-drv.ko  
cd ./build-timer_test-IDE_5_7_1_GCC_64bit-Debug/ //app The directory may be different,  
adjust according to the actual situation  
. /timer_test /dev/timer_led
```

The result is as follows:

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
random: fast init done
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mntrandom: crng init done
root@ax_peta:/mnt# insmod ./ax-timer-dev.ko
ax_timer_dev: loading out-of-tree module taints kernel.
alinx_char node find
alinxled-gpio num = 899
root@ax_peta:/mnt/build-timer_test-ZYNQ-Debug# ./timer_test /dev/timer_led
gpio_test module open
Input interval:500
interval refreshed!
Input interval:100
interval refreshed!
Input interval:1000
interval refreshed!
Input interval:
```

After running the app, enter different time intervals and observe the changes of the led.

## Part 8: Interrupt

Interrupt is a very commonly used function, and a complete interrupt framework is also implemented in the Linux kernel. In this chapter, we will learn the simple usage of interrupts in the Linux kernel.

### Part 8.1: Linux Interrupt Framework Introduction

#### Part 8.1.1: Interface function

The interrupt framework in the Linux kernel is quite easy to use. Generally, three things need to be done: apply for an interrupt, implement an interrupt service function, and enable an interrupt. The corresponding interface function is as follows

##### ➤ **Interrupt request and release function**

- 1) Interrupt request function `request_irq`, this function may cause sleep, the interrupt will be enabled after the request is successful.

The function prototype:

```
Int request_irq(unsignedintirq,irq_handler_t handler, unsigned long flags, const char*name,void*dev);
```

#### Parameter Description:

**irq:** The requested interrupt number. The interrupt number can also be called the interrupt line. It is the only identifier of the interrupt and the basis for the kernel to find the corresponding interrupt service function.

**handler:** Interrupt service function, a function that will be executed after the interrupt is triggered.

**flags:** Interrupt flags, used to set interrupt triggering methods and other characteristics. Commonly used flags are:

```
/* No trigger */  
IRQF_TRIGGER_NONE  
/* Rising edge trigger */  
IRQF_TRIGGER_RISING  
/* Falling edge trigger */  
IRQF_TRIGGER_FALLING  
/* High level trigger */  
IRQF_TRIGGER_HIGH  
/* Falling edge trigger */  
IRQF_TRIGGER_LOW  
/* Single interrupt */  
IRQF_ONESHOT  
/* As a timer interrupt */  
IRQF_TIMER  
/* Shared interrupt, this flag is needed when multiple devices share an interrupt number */  
IRQF_SHARED
```

All flags and definitions can be viewed in the ["/include/linux/interrupt.h"](#) file. The interrupt flag can be combined with the “|” sign, such as “[IRQF\\_TRIGGER\\_RISING|IRQF\\_ONESHOT](#)” means a single interrupt triggered by a rising edge.

**name:** Interrupt name. After the interrupt application is successful, you can find this name in the ["/proc/interrupts"](#) file.

**dev:** When the flag is set to [IRQF\\_SHARED](#), use **dev** to distinguish different devices, and the value of **dev** will be passed to the second parameter of the interrupt service function.

**Return value:** 0-application is successful, -EBUSY-interrupt has been occupied, other values indicate application failure.

- 2) The interrupt release function “[free\\_irq](#)”, which is opposite to the interrupt request, if the target interrupt is not a shared interrupt, then the [free\\_irq](#) function will disable the interrupt and delete the interrupt service function after releasing the interrupt. The

prototype is as follows

```
void free_irq(unsigned int irq, void *dev);
```

### Parameter Description:

**irq**: Interrupts that need to be released

**dev**: If the released interrupt is a shared interrupt, use this parameter to distinguish the specific interrupt. Only when all devs under the shared interrupt are released, the free\_irq function will disable the interrupt and remove the interrupt service function.

### ➤ Implement service request function

The format of the interrupt service function is:

```
irqreturn_t (*irq_handler_t) (int, void *)
```

The first parameter is the interrupt number corresponding to the **int** type interrupt service function.

The second parameter is a **general pointer**, which needs to be consistent with the parameter **dev** of the **request\_irq** function.

The return value **irqreturn\_t** is an enumerated type, and generally returns a value in the following way in the service function:

```
return IRQ_RETVAL(IRQ_HANDLED);
```

### ➤ Interrupt enable and disable functions

#### 1) **enable\_irq(unsigned int irq)**、**disable\_irq(unsigned int irq)** and **disable\_irq\_nosync(unsigned int irq)**

**enable\_irq** and **disable\_irq** are interrupt enable function and disable function respectively, **irq** is the target interrupt number. **disable\_irq** will wait for the interrupt service function of the target interrupt to complete before disabling the interrupt. If you want to disable the interrupt immediately, you can use the

`disable_irq_nosync()` function.

## 2) `local_irq_enable()` and `local_irq_disable()`

The `local_irq_enable()` function is used to enable the interrupt system of the current processor.

The `local_irq_disable()` function is used to disable the interrupt system of the current processor.

## 3) `local_irq_save(flags)` and `local_irq_restore(flags)`

The `local_irq_save(flags)` function is also used to disable the current processor interrupt, but it will save the previous interrupt state in the input parameter `flags`. The `local_irq_restore(flags)` function restores the interrupt to the state recorded in `flags`.

### Part 8.1.2: The Bottom Half of Linux

The upper half and the lower half are mechanisms introduced to minimize the processing of interrupt service functions. The upper half is the interrupt service function executed immediately after the interrupt is triggered, and the lower half is the delay processing of the interrupt service function. Because the priority of the interrupt is high, if the processing content is too much, the processor will be occupied for a long time, which will affect the operation of other codes, so the upper half should be as short as possible. In a bare-metal program, tree flags in the interrupt handling function, and then go to the main program loop to inquire and judge this flag in turn, and then do the corresponding operation, which is a kind of thinking of the upper half and the lower half. The upper part in Linux refers to the interrupt service function `irq_handler_t`. As for which tasks are placed in the upper half and which are placed in the lower half, there is no clear boundary. Generally, we put the tasks that are sensitive to time and cannot be interrupted in the upper half, and other tasks can be

considered in the lower half.

Linux also provides some perfect mechanisms for the lower half:

### 1) Soft interrupt

The soft interrupt structure is defined in the include/linux/interrupt.h file, as follows:

```
1. struct softirq_action
2. {
3.     void (*action)(struct softirq_action *);
4. };
```

The kernel defines the global soft interrupt vector table in the “kernel/softirq.c” file:

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

**NR\_SOFTIRQS** is the maximum value of the enumeration type, which is defined in [include/linux/interrupt.h](#):

```
1. enum
2. {
3.     HI_SOFTIRQ=0,
4.     TIMER_SOFTIRQ,
5.     NET_TX_SOFTIRQ,
6.     NET_RX_SOFTIRQ,
7.     BLOCK_SOFTIRQ,
8.     IRQ_POLL_SOFTIRQ,
9.     TASKLET_SOFTIRQ,
10.    SCHED_SOFTIRQ,
11.    HRTIMER_SOFTIRQ,
12.    RCU_SOFTIRQ,
13.
14.    NR_SOFTIRQS
15.};
```

It represents ten soft interrupts. To use soft interrupts, you can only register with the soft interrupt vector table defined by the kernel. Soft interrupts need to use functions:

```
void (int nr, void (*action)(struct softirq_action *));
```

#### Parameter Description:

**nr:** Smaller than enumeration value of NR\_SOFTIRQS.

**action:** the corresponding soft interrupt service function.

Soft interrupts must be registered statically at compile time. After

the registration is completed, `raise_softirq (unsigned int nr)` is used to trigger, and `nr` is the soft interrupt that needs to be triggered.

That is, the soft interrupt that needs to be triggered.

But the lower half of the mechanism usually does not use soft interrupts, but uses the `tasklets` mechanism described below.

## 2) tasklets mechanism

The Linux kernel initializes the soft interrupt in the `softirq_int` function, where `HI_SOFTIRQ` and `TASKLET_SOFTIRQ` are enabled by default. The `tasklets` mechanism is implemented on the basis of these two soft interrupts.

The structure definition of `tasklets` is also in the header file “`include/linux/interrupt.h`”, which is defined as follows:

```
1. struct tasklet_struct
2. {
3.     struct tasklet_struct *next;
4.     unsigned long state;
5.     atomic_t count;
6.     void (*func)(unsigned long);
7.     unsigned long data;
8. };
```

Among them, `func` is equivalent to the interrupt service function of `tasklet`. The definition and initialization of `tasklet` can be directly completed with the following macro definition:

```
DECLARE_TASKLET(name, func, data)
```

**name:** The name of the tasklet.

**func:** The processing function when tasklet is triggered.

**data:** The input parameters passed to func.

After the initialization is complete, call the following function to activate `tasklet`

```
tasklet_schedule(struct tasklet_struct *t)
```

After activation, `tasklet's` service function will run at the right time.

When used as the lower half of the interrupt, call this function in the upper half. If you want to use a **tasklet** with a higher priority, use the **tasklet\_hi\_schedule(struct tasklet\_struct \*t)** function to activate.

The use of the lower half of the tasklet mechanism, example:

```
1. /* 定义 tasklet */
2. struct tasklet_struct example;
3. /* tasklet 处理函数 */
4. void testtasklet_func(unsigned long data)
5. {
6.     /* tasklet 具体处理内容 */
7. }
8. /* 中断处理函数 */
9. irqreturn_t test_handler(int irq, void *dev_id)
10. {
11.     /* 调度 tasklet */
12.     tasklet_schedule(&example);
13. }
14. /* 驱动入口函数 */
15. static int xxxx_init(void)
16. {
17.     /* 初始化 tasklet */
18.     tasklet_init(&example, testtasklet_func, data);
19.     /* 注册中断处理函数 */
20.     request_irq(irq, test_handler, 0, "name", &dev);
21. }
```

### 3) Work queue

The work queue is also the realization of the lower half. In contrast to **tasklets**, work queues are blockable, so they cannot run in interrupt context. We don't need to worry about the queue implementation of the work queue. To use the work queue, we only need to set a job.

The work structure is **work\_struct**, which is defined in the **/include/linux/workqueue.h** file:

```
1. struct work_struct {
2.     atomic_long_t data;
3.     struct list_head entry;
4.     work_func_t func;
5. #ifdef CONFIG_LOCKDEP
6.     struct lockdep_map lockdep_map;
7. #endif
8. };
```

**func** is the function to be processed. The following macro definitions can be used to create and initialize jobs:

```
DECLARE_WORK(n, f)
```

**n:** The name of the work structure **work\_struct** that needs to be created and initialized.

**f:** The function to be processed by the work queue.

After the initialization is complete, use the following function to call the work queue:

```
bool schedule_work(struct work_struct *work)
```

**work:** The work to be called.

**Return value:** 0 succeeds, 1 fails.

Use of the lower half of the **workqueue** mechanism, example:

```
1. /* 定义工作(work) */
2. struct work_struct example;
3. /* work 处理函数 */
4. void work_func_t(struct work_struct *work);
5. {
6.     /* work 具体处理内容 */
7. }
8. /* 中断处理函数 */
9. irqreturn_t test_handler(int irq, void *dev_id)
10. {
11.     /* 调度 work */
12.     schedule_work(&example);
13. }
14. /* 驱动入口函数 */
15. static int xxxx_init(void)
16. {
17.     .....
18.     /* 初始化 work */
19.     INIT_WORK(&example, work_func_t);
20.     /* 注册中断处理函数 */
21.     request_irq(irq, test_handler, 0, "name", &dev);
22. }
```

### Part 8.1.3: Interrupts in the device tree

In the device tree, the general interrupt setting method can refer to the document "[Documentation/devicetree/bindings/arm/arm,gic.txt](#)". The settings of the Xilinx device tree interrupt controller are slightly different from the general settings of the Linux kernel. You can check the document "[Documentation/devicetree/bindings/arm/xilinx,intc.txt](#)" for details. Look at the last example of this file:

```
1. axi_intc_0: interrupt-controller@41800000 {  
2.     #interrupt-cells = <2>;  
3.     compatible = "xlnx,xps-intc-1.00.a";  
4.     interrupt-controller;  
5.     interrupt-parent = <&ps7_scugic_0>;  
6.     interrupts = <0 29 4>;  
7.     reg = <0x41800000 0x10000>;  
8.     xlnx,kind-of-intr = <0x1>;  
9.     xlnx,num-intr-inputs = <0x1>;  
10. };
```

Looking back at the chapter of the gpio subsystem, when talking about the device tree of gpio at that time, these interrupt-related attributes already appeared.

**Line 2**, "#interrupt-cells" is the attribute of the interrupt controller node, used to describe the number of "interrupts" attribute values in the child node. Generally, the value of "#interrupt-cells" of the parent node is 3, and the value of the three 32-bit integers of a cell of "interrupts" of the child node is <interrupt domain interrupt number triggering method>, if the attribute of the parent node is 2 , It is <interrupt number trigger method>.

**Line 4**, the attribute "interrupt-controller" represents that this node is an interrupt controller.

**Line 5**, the "interrupt-parent" attribute indicates which interrupt controller this device belongs to. If it does not have this attribute, it will automatically be attached to the "interrupt-parent" of the parent node.

**Line 6**, "interrupts", the first value is 0 for SPI interrupt, 1 for PPI interrupt. In zynq, if the first value is 0, the interrupt number is equal to the second value plus 32.

**Line 8**, "xlnx, kind-of-intr" indicates that the interrupt type is specified for each possible interrupt. 1 indicates edge, and 0 indicates level.

**Line 9**, the "xlnx,num-intr-inputs" attribute specifies the number of interrupts supported by the specific implementation of the controller,

ranging from 1 to 32.

```
unsigned int irq_of_parse_and_map(struct device_node *dev, int index);
```

**dev** is the device node

**index** is the index of the attribute "interrupts" element, because the position of the interrupt number may be different.

**Return Value:** is the interrupt number.

To use this function, we need to set the "interrupts" attribute in the corresponding device.

For **gpio**, the kernel provides a more convenient function to get the interrupt number:

```
int gpio_to_irq(unsigned int gpio);
```

**gpio** is the gpio number that needs to apply for an interrupt number.

**Return Value:** is the interrupt number.

**gpio** under **zynq** is a shared interrupt. It is troublesome to set the "interrupts" attribute for a single **io**. The **gpio\_to\_irq** function does a lot for us. For the **gpio** interrupt experiment later, we will use this function directly.

## Part 8.2: Experiment

This chapter writes a key-press interrupt driver, press the key to trigger the interrupt, and after the interrupt is triggered, a 50ms timer is started in the interrupt service function to achieve key debounce.

### Part 8.2.1: Schematic

The led part is the same as **Part 1.3.1**.

The key part is the same as in **Part 6.1**.

### Part 8.2.2: Device tree

Same as **Part 6.2**

### Part 8.2.3: Driver code

Use petalinux to create a new driver named "**ax-irq-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-irq-drv.c** file:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/init.h>
4. #include <linux/ide.h>
5. #include <linux/types.h>
6. #include <linux/errno.h>
7. #include <linux/cdev.h>
8. #include <linux/of.h>
9. #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <asm/uaccess.h>
20. #include <asm/io.h>
21.
22. /* 设备节点名称 */
23. #define DEVICE_NAME      "interrupt_led"
24. /* 设备号个数 */
25. #define DEVID_COUNT      1
26. /* 驱动个数 */
27. #define DRIVE_COUNT       1
28. /* 主设备号 */
29. #define MAJOR_U           0
30. /* 次设备号 */
31. #define MINOR_U          0
32.
33. /* 把驱动代码中会用到的数据打包进设备结构体 */
34. struct alinx_char_dev {
35.     /** 字符设备框架 ***/
36.     dev_t             devid;           //设备号
37.     struct cdev        cdev;            //字符设备
38.     struct class       *class;          //类
39.     struct device      *device;         //设备
40.     struct device_node *nd;            //设备树的设备节点
41.     /** 并发处理 ***/
42.     spinlock_t        lock;             //自旋锁变量
43.     /** gpio ***/
44.     int               alinx_key_gpio; //gpio 号
45.     int               key_sts;          //记录按键状态，为 1 时被按下
46.     /** 中断 ***/
47.     unsigned int       irq;              //中断号
48.     /** 定时器 ***/
49.     struct timer_list  timer;           //定时器
50. };
51. /* 声明设备结构体 */
52. static struct alinx_char_dev alinx_char = {
53.     .cdev = {
```

```
54.         .owner = THIS_MODULE,
55.     },
56. };
57.
58. /** 回掉 ***/
59. /* 中断服务函数 */
60. static irqreturn_t key_handler(int irq, void *dev)
61. {
62.     /* 按键按下或抬起时会进入中断 */
63.     /* 开启 50 毫秒的定时器用作防抖动 */
64.     mod_timer(&alinx_char.timer, jiffies + msecs_to_jiffies(50));
65.     return IRQ_RETVAL(IRQ_HANDLED);
66. }
67.
68. /* 定时器服务函数 */
69. void timer_function(struct timer_list *timer)
70. {
71.     unsigned long flags;
72.     /* 获取锁 */
73.     spin_lock_irqsave(&alinx_char.lock, flags);
74.
75.     /* value 用于获取按键值 */
76.     unsigned char value;
77.     /* 获取按键值 */
78.     value = gpio_get_value(alinx_char.alinx_key_gpio);
79.     if(value == 0)
80.     {
81.         /* 按键按下，状态置 1 */
82.         alinx_char.key_sts = 1;
83.     }
84.     else
85.     {
86.         /* 按键抬起 */
87.     }
88.
89.     /* 释放锁 */
90.     spin_unlock_irqrestore(&alinx_char.lock, flags);
91. }
92.
93. /** 系统调用实现 ***/
94. /* open 函数实现，对应到 Linux 系统调用函数的open 函数 */
95. static int char_drv_open(struct inode *inode_p, struct file *file_p)
96. {
97.     printk("gpio_test module open\n");
98.     return 0;
99. }
100.
101.
102. /* read 函数实现，对应到 Linux 系统调用函数的write 函数 */
103. static ssize_t char_drv_read(struct file *file_p, char user *buf, size_t len, lo ff_t *loff_t_p)
104. {
105.     unsigned long flags;
106.     int ret;
107.     /* 获取锁 */
108.     spin_lock_irqsave(&alinx_char.lock, flags);
109.
110.    /* keysts 用于读取按键状态 */
111.    /* 返回按键状态值 */
112.    ret = copy_to_user(buf, &alinx_char.key_sts, sizeof(alinx_char.key_sts));
113.    /* 清除按键状态 */
114.    alinx_char.key_sts = 0;
115.
116.    /* 释放锁 */
117.    spin_unlock_irqrestore(&alinx_char.lock, flags);
118.    return 0;
119. }
120.
```

```
121. /* release 函数实现，对应到Linux 系统调用函数的close 函数 */
122. static int char_drv_release(struct inode *inode_p, struct file *file_p)
123. {
124.     printk("gpio_test module release\n");
125.     return 0;
126. }
127.
128. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
129. static struct file_operations ax_char_fops = {
130.     .owner    = THIS_MODULE,
131.     .open     = char_drv_open,
132.     .read     = char_drv_read,
133.     .release  = char_drv_release,
134. };
135.
136. /* 模块加载时会调用的函数 */
137. static int init char_drv_init(void)
138. {
139.     /* 用于接受返回值 */
140.     u32 ret = 0;
141.
142.     /** 并发处理 */
143.     /* 初始化自旋锁 */
144.     spin_lock_init(&alinx_char.lock);
145.
146.     /** gpio 框架 */
147.     /* 获取设备节点 */
148.     alinx_char.nd = of_find_node_by_path("/alinkkey");
149.     if(alinx_char.nd == NULL)
150.     {
151.         printk("alink_char node not find\r\n");
152.         return -EINVAL;
153.     }
154.     else
155.     {
156.         printk("alink_char node find\r\n");
157.     }
158.
159.     /* 获取节点中 gpio 标号 */
160.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinkkey-gpios",
161.     0);
162.     if(alinx_char.alinx_key_gpio < 0)
163.     {
164.         printk("can not get alinkkey-gpios");
165.         return -EINVAL;
166.     }
167.     printk("alinkkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
168.
169.     /* 申请 gpio 标号对应的引脚 */
170.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinkkey");
171.     if(ret != 0)
172.     {
173.         printk("can not request gpio\r\n");
174.         return -EINVAL;
175.     }
176.     /* 把这个io 设置为输入 */
177.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
178.     if(ret < 0)
179.     {
180.         printk("can not set gpio\r\n");
181.         return -EINVAL;
182.     }
183.
184.     /** 中断 */
185.     /* 获取中断号 */
186.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
187.     /* 申请中断 */
```

```
188.     ret = request_irq(alinx_char.irq,
189.                           key_handler,
190.                           IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
191.                           "alinxkey",
192.                           NULL);
193.     if(ret < 0)
194.     {
195.         printk("irq %d request failed\r\n", alinx_char.irq);
196.         return -EFAULT;
197.     }
198.
199. /** 定时器 **/
200.     timer_setup(&alink_char.timer, timer_function, NULL);
201.
202. /** 字符设备框架 **/
203. /* 注册设备号 */
204. alloc_chrdev_region(&alink_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
205.
206. /* 初始化字符设备结构体 */
207. cdev_init(&alink_char.cdev, &ax_char_fops);
208.
209. /* 注册字符设备 */
210. cdev_add(&alink_char.cdev, alinx_char.devid, DRIVE_COUNT);
211.
212. /* 创建类 */
213. alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
214. if(IS_ERR(alinx_char.class))
215. {
216.     return PTR_ERR(alinx_char.class);
217. }
218.
219. /* 创建设备节点 */
220. alinx_char.device = device_create(alinx_char.class, NULL,
221.                                     alinx_char.devid, NULL,
222.                                     DEVICE_NAME);
223. if (IS_ERR(alinx_char.device))
224. {
225.     return PTR_ERR(alinx_char.device);
226. }
227.
228. return 0;
229. }
230.
231. /* 卸载模块 */
232. static void exit char_drv_exit(void)
233. {
234. /** gpio **/
235. /* 释放 gpio */
236. gpio_free(alinx_char.alinx_key_gpio);
237.
238. /** 中断 **/
239. /* 释放中断 */
240. free_irq(alinx_char.irq, NULL);
241.
242. /** 定时器 **/
243. /* 删除定时器 */
244. del_timer_sync(&alink_char.timer);
245.
246. /** 字符设备框架 **/
247. /* 注销字符设备 */
248. cdev_del(&alink_char.cdev);
249.
250. /* 注销设备号 */
251. unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
252.
253. /* 删除设备节点 */
254. device_destroy(alinx_char.class, alinx_char.devid);
255.
```

```
256.     /* 删除类 */
257.     class_destroy(alinx_char.class);
258.
259.     printk("timer_led_dev_exit_ok\n");
260. }
261.
262. /* 标记加载、卸载函数 */
263. module_init(char_drv_init);
264. module_exit(char_drv_exit);
265.
266. /* 驱动描述信息 */
267. MODULE_AUTHOR("Alinx");
268. MODULE_ALIAS("alinx char");
269. MODULE_DESCRIPTION("INTERRUPT LED driver");
270. MODULE_VERSION("v1.0");
271. MODULE_LICENSE("GPL");
```

**Line 197**, in the driver entry function, after initializing **gpio**, use the **gpio\_to\_irq** function to obtain the interrupt number through the **gpio** port number.

**Line 199**, apply for an interrupt to the kernel through the interrupt number. The rising or falling edge triggers, named "**alinkkey**", and the interrupt service function is **key\_handler**.

Contrasting with the interruption steps mentioned earlier, now we only need to implement the **key\_handler** function.

**Line 71** implements the **key\_handler**. The content is very simple. First, start a 50ms timer, and then return **IRQ\_RETVAL (IRQ\_HANDLED)**.

**Line 252**, the registered interrupt number is released in the driver exit function.

Regarding the object protected by the spin lock, it is actually the value of **alink\_char.key\_sts**, because this value is operated in the read function, and it is also operated in the interrupt-on timer return function. These two operations may occur at the same time., So need to protect

#### Part 8.2.4: Test code

Same as part 6

#### Part 8.2.5: Run test

---

The experiment is to use **ps key1** on the FPGA development board to control **ps led1**. The test steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt  
cd /mnt  
mkdir /tmp/qt  
mount qt_lib.img /tmp/qt  
cd /tmp/qt  
source ./qt_env_set.sh  
cd /mnt  
insmod ./ax-concled-drv.ko  
insmod ./ax-irq-drv.ko  
cd ./build-ax-key-test-IDE_5_7_1_GCC_64bit-Debug/  
./ax-key-test /dev/interrupt_led
```

The IP and path are adjusted according to the actual situation.  
The test result is also the same as in **Part 6**.

## Part 9: Blocking IO

**IO** is the abbreviation of “**Input stream/Output stream**”, that is, input and output. For the driver, IO is the access and operation of the device resource by the user program. Next, briefly talk about several IO modules and Linux support for them.

### Part 9.1: Blocking and Non-blocking, Synchronous and Asynchronous, and IO operation

Blocking and non-blocking, synchronous and asynchronous are several unavoidable states in IO operations. Let's look at it separately:

- 1) **Blocking IO** means that if an operation does not meet the execution conditions, it will remain in a waiting state until the conditions are met.
- 2) **Non-blocking IO** means that if an operation does not meet the execution conditions, it will not wait and return an unexecuted result.
- 3) **Synchronous IO** means that when multiple operations occur at the same time, these operations need to be queued for execution one by one.
- 4) **Asynchronous IO** means that when multiple operations occur at the same time, these operations can be executed together.

For the driver, IO operations can generally be understood as reading and writing to peripherals. The complete IO operation has two stages:

**The first stage:** Check whether the peripheral data is ready;

**The second stage:** Data is ready, read and write peripheral data.

Combine it to see:

**Blocking IO, non-blocking IO:** When the application receives an

IO request. If the target peripheral or data is not ready, for blocking IO, it will wait in the read method until the data is ready. Non-blocking IO will directly return that the data is not ready, and the application will deal with the NG situation, re-read or other. It can be seen that blocking IO and non-blocking IO are reflected in the first stage of IO operation

**Synchronous IO, asynchronous IO:** Synchronous IO and asynchronous IO are actually for the interaction between the application and the kernel. After the application receives the IO request, if the data is not ready, the application needs to continuously poll until it is ready Perform the second stage. For asynchronous IO, after the application receives the IO request, the first and second stages are all completed by the kernel. Of course, the driver is also part of the kernel.

## Part 9.2: Blocking IO

The **blocking IO** mentioned here is actually synchronous blocking IO. In the blocking access of Linux, when the application calls the **read()** function to collect data from the device, if the device or the data is not ready, it will go to sleep and give up CPU resources. When ready, it will wake up and return the data to the application. The kernel provides a waiting queue mechanism to realize the sleep and wake-up work here.

### Part 9.2.1: Waiting queue

The waiting queue is a queue composed of processes. Linux will divide the process into different queues according to different states in the system execution, and the waiting queue is one of them.

The steps to use the waiting queue in the driver are as follows:

## 1) Create and initialize the waiting queue

The way to create a waiting queue is to create a waiting queue head, and adding items under the queue head is the queue. The queue header is defined in “[include/linux/wait.h](#)”, the details are as follows:

```
1. struct wait_queue_head {  
2.     spinlock_t      lock;  
3.     struct list_head  task_list;  
4. };  
5. typedef struct wait_queue_head wait_queue_head_t;
```

After defining the queue head, use the following function to initialize the queue head:

```
void init_waitqueue_head(wait_queue_head_t *q)
```

You can also use macro definitions

```
DECLARE_WAIT_QUEUE_HEAD_ONSTACK(name)
```

Complete the creation and initialization of the queue head at one time, and name is the name of the queue head.

## 2) Create a waiting queue item representing the process

The waiting queue items are also defined in the [include/linux/wait.h](#) header file, which can be defined by macros:

```
DECLARE_WAITQUEUE(name, tsk)
```

Complete the definition and initialization of the queue item at one time, **name** is the name of the queue item, **tsk** is the process referred to by the queue item, and is generally set to **current**. **current** is a global variable in the kernel, representing the current process.

## 3) Add or remove waiting queue items to the waiting queue and go to sleep

When the device or data is inaccessible, add the process to the queue and use the interface function:

```
void add_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
```

“**q**” is the head of the queue that needs to be added, and “**wait**” is the queue item that needs to be added. Use function after adding

```
__set_current_state(state_value);
```

Set the process state, **state\_value** can be:

**TASK\_UNINTERRUPTIBLE** sleep cannot be interrupted by signal

**TASK\_INTERRUPTIBLE** sleep can be interrupted by a signal.

Then call the task switching function

```
schedule();
```

Put the current process to sleep. If it is awakened, it will run down from the position of this function.

Then, if the process is set to **TASK\_INTERRUPTIBLE** state, if necessary, it is necessary to determine whether the process is awakened by a signal. If it is, it is awakened by mistake, and the process needs to be put to sleep again.

Use function

```
signal_pending(current)
```

Judge whether the current process is awakened by a signal, “**current**” is the current process, and returns true if it is.

After the process is awakened, use

```
set_current_state(TASK_RUNNING)
```

Set the current process to the running state.

If the device is accessible, the queue item is still removed from the queue header, use the function:

```
void remove_wait_queue(wait_queue_head_t *q, wait_queue_t *wait)
```

#### 4) Actively wake up or wait for an event

After the process sleeps, use the following two functions to actively wake up the entire queue:

```
void wake_up(wait_queue_head_t *q)
void wake_up_interruptible(wait_queue_head_t *q)
```

The `wake_up` function can wake up processes in “**TASK\_INTERRUPTIBLE**” and “**TASK\_UNINTERRUPTIBLE**” states.

The “`wake_up_interruptible`” function can only wake up processes in the “**ASK\_INTERRUPTIBLE**” state.

In addition to active wake-up, it can also be set to automatically wake up after a certain condition is met. Linux provides these macros:

```
/*This function will set the process to “TASK_UNINTERRUPTIBLE”, the queue “wq” will be woken up when the condition is true (condition), and will be blocked waiting for the condition to be true */
wait_event(wq, condition)
/* Similar to “wait_event”, but with a timeout mechanism added, “timeout” is the timeout unit of jiffies, after the time is up, the queue “wq” will be woken up even if the conditions are not met */
wait_event_timeout(wq, condition, timeout)
/*Similar to “wait_event”, but will set the process to “TASK_INTERRUPTIBLE” */
wait_event_interruptible(wq, condition)
/*Similar to “wait_event_timeout”, but will set the process to “TASK_INTERRUPTIBLE”*/
wait_event_interruptible_timeout(wq, condition, timeout)
```

## Part 9.3: Experiment

In the previous key press experiment, the way the test program reads the key state is to continuously call the read method in the while loop. And the read method we implemented in the driver simply returns the current value of the key. As a result, the test program and the driver are always active, resulting in a high cpu occupancy rate. Take the example in the previous chapter as an example, use the “`./ax-key-test /dev/interrupt_led&`” command to make the “**ax-key-test**” program run in the background. Then use the top

command to view the occupancy of the cpu, as shown below:

PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
1274	1254	root	R	13164	1.2	0	49.9	./ax-key-test /dev/interrupt_led
1275	1254	root	R	3024	0.2	1	0.1	top
269	2	root	SW	0	0.0	1	0.1	[kworker/1:1]
1247	1	root	S	14352	1.3	1	0.0	/usr/sbin/tcf-agent -d -L - -10
761	1	root	S	3052	0.3	1	0.0	/sbin/udevd -d

The dual-cpu soc one key-press program occupies 49.9% of the resources of almost one cpu, which is obviously undesirable.

Analyze, the application polls the read function to read the button state, most of the time it reads the state that is not pressed, and what we need to capture is only the state of the button being pressed, can it be understood as , If the button is not pressed, it means that the data we need is not ready yet? On this basis, we can use the waiting queue to make the read process in the driver go to sleep when the button is not pressed, the read function of the application will not get the return value, and it will not poll all the time, thereby reducing the cpu Occupancy rate. Then when the button is pressed, the process is awakened, and the driver can meet the requirements of capturing the action of the button being pressed.

### Part 9.3.1: Schematic

The led part is the same as **Part 1.3.1**.

The key part is the same as in **Part 6.1**.

### Part 9.3.2: Device tree

Same as **Part 6.2**

### Part 9.3.3: Driver code

Use petalinux to create a new driver named "**ax-bio-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-bio-drv** file:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/init.h>
4. #include <linux/ide.h>
5. #include <linux/types.h>
6. #include <linux/errno.h>
7. #include <linux/cdev.h>
8. #include <linux/of.h>
9. #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <asm/uaccess.h>
20. #include <asm/io.h>
21.
22. /* 设备节点名称 */
23. #define DEVICE_NAME      "bio_led"
24. /* 设备号个数 */
25. #define DEVID_COUNT      1
26. /* 驱动个数 */
27. #define DRIVE_COUNT       1
28. /* 主设备号 */
29. #define MAJOR_U
30. /* 次设备号 */
31. #define MINOR_U          0
32.
33. /* 把驱动代码中会用到的数据打包进设备结构体 */
34. struct alinx_char_dev {
35.     /** 字符设备框架 ***/
36.     dev_t             devid;           //设备号
37.     struct cdev        cdev;            //字符设备
38.     struct class       *class;          //类
39.     struct device      *device;         //设备
40.     struct device_node *nd;            //设备树的设备节点
41.     /** gpio **/
42.     int               alinx_key_gpio; //gpio 号
43.     /** 并发处理 **/
44.     atomic_t          key_sts;         //记录按键状态，为 1 时被按下
45.     /** 中断 **/
46.     unsigned int       irq;             //中断号
47.     /** 定时器 **/
48.     struct timer_list  timer;           //定时器
49.     /** 等待队列 **/
50.     wait_queue_head_t  wait_q_h;       //等待队列头
51. };
52. /* 声明设备结构体 */
53. static struct alinx_char_dev alinx_char = {
54.     .cdev = {
55.         .owner = THIS_MODULE,
56.     },
57. };
58.
59. /** 回掉 **/
60. /* 中断服务函数 */
61. static irqreturn_t key_handler(int irq, void *dev)
62. {
63.     /* 按键按下或抬起时会进入中断 */
64.     /* 开启 50 毫秒的定时器用作防抖动 */
65.     mod_timer(&alink_char.timer, jiffies + msecs_to_jiffies(50));
66.     return IRQ_RETVAL(IRQ_HANDLED);
67. }
```

```
68.
69. /* 定时器服务函数 */
70. void timer_function(struct timer_list *timer)
71. {
72.     /* value 用于获取按键值 */
73.     unsigned char value;
74.     /* 获取按键值 */
75.     value = gpio_get_value(alinx_char.alinx_key_gpio);
76.     if(value == 0)
77.     {
78.         /* 按键按下，状态置 1 */
79.         atomic_set(&alinx_char.key_sts, 1);
80.     /** 等待队列 */
81.         /* 唤醒进程 */
82.         wake_up_interruptible(&alinx_char.wait_q_h);
83.     }
84.     else
85.     {
86.         /* 按键抬起 */
87.     }
88. }
89.
90. /** 系统调用实现 */
91. /* open 函数实现，对应到 Linux 系统调用函数的open 函数 */
92. static int char_drv_open(struct inode *inode_p, struct file *file_p)
93. {
94.     printk("gpio_test module open\n");
95.     return 0;
96. }
97.
98.
99. /* read 函数实现，对应到 Linux 系统调用函数的write 函数 */
100. static ssize_t char_drv_read(struct file *file_p, char user *buf, size_t len, lo
101. {
102.     unsigned int keysts = 0;
103.     int ret;
104.
105.     /* 读取 key 的状态 */
106.     keysts = atomic_read(&alinx_char.key_sts);
107.     /* 判断当前按键状态 */
108.     if(!keysts)
109.     {
110.         /* 按键未被按下(数据未准备好) */
111.         /* 以当前进程创建并初始化为队列项 */
112.         DECLARE_WAITQUEUE(queue_mem, current);
113.         /* 把当前进程的队列项添加到队列头 */
114.         add_wait_queue(&alinx_char.wait_q_h, &queue_mem);
115.         /* 设置当前进程成为可被信号打断的状态 */
116.         _set_current_state(TASK_INTERRUPTIBLE);
117.         /* 切换进程，是当前进程休眠 */
118.         schedule();
119.
120.         /* 被唤醒，修改当前进程状态为RUNNING */
121.         set_current_state(TASK_RUNNING);
122.         /* 把当前进程的队列项从队列头中删除 */
123.         remove_wait_queue(&alinx_char.wait_q_h, &queue_mem);
124.
125.         /* 判断是否是被信号唤醒 */
126.         if(signal_pending(current))
127.         {
128.             /* 如果是直接返回错误 */
129.             return -ERESTARTSYS;
130.         }
131.     else
132.     {
133.         /* 被按键唤醒 */
```

```
134.         }
135.     }
136. else
137. {
138.     /* 按键被按下(数据准备好了) */
139. }
140.
141.     /* 读取 key 的状态 */
142.     keysts = atomic_read(&alinx_char.key_sts);
143.     /* 返回按键状态值 */
144.     ret = copy_to_user(buf, &keysts, sizeof(keysts));
145.     /* 清除按键状态 */
146.     atomic_set(&alinx_char.key_sts, 0);
147.     return 0;
148. }
149.
150. /* release 函数实现, 对应到Linux 系统调用函数的close 函数 */
151. static int char_drv_release(struct inode *inode_p, struct file *file_p)
152. {
153.     printk("gpio_test module release\n");
154.     return 0;
155. }
156.
157. /* file_operations 结构体声明, 是上面 open、write 实现函数与系统调用函数对应的关键 */
158. static struct file_operations ax_char_fops = {
159.     .owner    = THIS_MODULE,
160.     .open     = char_drv_open,
161.     .read     = char_drv_read,
162.     .release  = char_drv_release,
163. };
164.
165. /* 模块加载时会调用的函数 */
166. static int init char_drv_init(void)
167. {
168.     /* 用于接受返回值 */
169.     u32 ret = 0;
170.
171.     /** 并发处理 */
172.     /* 初始化原子变量 */
173.     atomic_set(&alinx_char.key_sts, 0);
174.
175.     /** gpio 框架 */
176.     /* 获取设备节点 */
177.     alinx_char.nd = of_find_node_by_path("/alinkkey");
178.     if(alinx_char.nd == NULL)
179.     {
180.         printk("alink_char node not find\r\n");
181.         return -EINVAL;
182.     }
183.     else
184.     {
185.         printk("alink_char node find\r\n");
186.     }
187.
188.     /* 获取节点中 gpio 标号 */
189.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinkkey-gpios",
190.     0);
191.     if(alinx_char.alinx_key_gpio < 0)
192.     {
193.         printk("can not get alinkkey-gpios");
194.         return -EINVAL;
195.     }
196.     printk("alinkkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
197.     /* 申请 gpio 标号对应的引脚 */
198.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinkkey");
199.     if(ret != 0)
200.     {
```

```
201.         printk("can not request gpio\r\n");
202.         return -EINVAL;
203.     }
204.
205.     /* 把这个io 设置为输入 */
206.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
207.     if(ret < 0)
208.     {
209.         printk("can not set gpio\r\n");
210.         return -EINVAL;
211.     }
212.
213. /** 中断 */
214. /* 获取中断号 */
215. alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
216. /* 申请中断 */
217. ret = request_irq(alinx_char.irq,
218.                     key_handler,
219.                     IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
220.                     "alinxkey",
221.                     NULL);
222. if(ret < 0)
223. {
224.     printk("irq %d request failed\r\n", alinx_char.irq);
225.     return -EFAULT;
226. }
227.
228. /** 定时器 */
229. timer_setup(&alinx_char.timer, timer_function, NULL);
230.
231. /** 等待队列 */
232. init_waitqueue_head(&alinx_char.wait_q_h);
233.
234. /** 字符设备框架 */
235. /* 注册设备号 */
236. alloc_chrdev_region(&alinx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
237.
238. /* 初始化字符设备结构体 */
239. cdev_init(&alinx_char.cdev, &ax_char_fops);
240.
241. /* 注册字符设备 */
242. cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
243.
244. /* 创建类 */
245. alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
246. if(IS_ERR(alinx_char.class))
247. {
248.     return PTR_ERR(alinx_char.class);
249. }
250.
251. /* 创建设备节点 */
252. alinx_char.device = device_create(alinx_char.class, NULL,
253.                                     alinx_char.devid, NULL,
254.                                     DEVICE_NAME);
255. if (IS_ERR(alinx_char.device))
256. {
257.     return PTR_ERR(alinx_char.device);
258. }
259.
260. return 0;
261. }
262.
263. /* 卸载模块 */
264. static void exit char_drv_exit(void)
265. {
266. /** gpio */
267. /* 释放 gpio */
268. gpio_free(alinx_char.alinx_key_gpio);
```

```
269.  
270. /** 中断 **/  
271. /* 释放中断 */  
272. free_irq(alinx_char.irq, NULL);  
273.  
274. /** 定时器 **/  
275. /* 删除定时器 */  
276. del_timer_sync(&alinx_char.timer);  
277.  
278. /** 字符设备框架 **/  
279. /* 注销字符设备 */  
280. cdev_del(&alinx_char.cdev);  
281.  
282. /* 注销设备号 */  
283. unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);  
284.  
285. /* 删除设备节点 */  
286. device_destroy(alinx_char.class, alinx_char.devid);  
287.  
288. /* 删除类 */  
289. class_destroy(alinx_char.class);  
290.  
291. printk("timer_led_dev_exit_ok\n");  
292. }  
293.  
294. /* 标记加载、卸载函数 */  
295. module_init(char_drv_init);  
296. module_exit(char_drv_exit);  
297.  
298. /* 驱动描述信息 */  
299. MODULE_AUTHOR("Alinx");  
300. MODULE_ALIAS("alinx_char");  
301. MODULE_DESCRIPTION("BIO LED driver");  
302. MODULE_VERSION("v1.0");  
303. MODULE_LICENSE("GPL");
```

The program is modified on the basis of the interrupt driver in the previous chapter, and the main modification is concentrated in the **read** function.

We replaced the spin lock with an atomic variable and only protected the read and write of the **key\_sts** state value.

**Line 50** first defines a waiting queue head.

**Line 234** of the entry function initializes the head of the queue.

The application receives the state of the key through the read method, so first go to the read function to make some changes. As mentioned earlier, the data is considered ready when the key is pressed. After entering the read function, at lines **107~109**, we first judge the state of the key. If the key is not pressed, use the waiting queue and wait for the key to be pressed.

**Line 113**, a queue item named “`queue_mem`” is created and initialized with the current process.

**Line 115**, add the queue item to the head of the queue.

**Line 117**, the process is set to a state that can be interrupted by a signal, and then **Line 119** calls `schedule` to switch the process and put the current process to sleep.

Sleeping requires a corresponding wake-up opportunity. We are waiting for the key to be pressed. Therefore, the wake-up can be executed in the interrupt of the key. In the callback of the 93-line timer, it is finally determined that the key is pressed at the same time. Call `wake_up_interruptible(&alinx_char.wait_q_h);` to wake up the waiting queue.

After waking up, we return to **Line 122**, and then continue to run at the position where we were sleeping, first call `set_current_state(TASK_RUNNING);` set the current process state to `RUNNING`.

**Line 124** then deletes the queue item from the head of the queue. Since the process can be awakened by a signal, it is also necessary to determine whether the process is awakened by a signal, and if it is, an error is directly returned. If not, the value of the key is returned to the user.

#### Part 9.3.4: Test code

Same as part 6

#### Part 9.3.5: Run test

The test steps are as follows

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt  
cd /mnt  
mkdir /tmp/qt
```

```
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-concled-drv.ko
insmod ./ax-bio-drv.ko
cd ./build-ax-key-test-IDE_5_7_1_GCC_64bit-Debug/
./ax-key-test /dev/bio_led&
top
```

The IP and path are adjusted according to the actual situation.

The test result is also the same as in **Part 6**.

In addition, let's look at the cpu usage of the test program.

PID	PPID	USER	STAT	VSZ	%VSZ	CPU	%CPU	COMMAND
1329	1254	root	R	3024	0.2	1	0.1	top
1247	1	root	S	14352	1.3	1	0.0	/usr/sbin/tcf-agent -d -L- -I
1328	1254	root	S	13164	1.2	0	0.0	./ax-key-test /dev/bio_led
761	1	root	S	3052	0.3	0	0.0	/sbin/udevd -d
1254	1252	root	S	2988	0.2	1	0.0	-sh

The cpu occupancy can almost be ignored, because the test program has not been changed, so it seems that the waiting queue is working.

If you want to close the program running in the background, you can use the **kill** command and add the corresponding **PID** in the top command. For example, if we want to close the ax-key-test program here, just use the command “**kill 1328**”.

## Part 10: Non-Blocking IO

This chapter introduces another IO model, non-blocking IO (NIO), which is synchronous non-blocking IO. As mentioned in the previous chapter, the two phases of IO operation are first query and then read and write, while the processing of non-blocking IO in the query phase is different from blocking IO. Before the application needs to perform an IO operation, a query is initiated. The driver returns the query result according to the data situation. If the query result is NG, the application does not perform read or write operations. If the application program has to read and write, it will continue to query until the driver returns the data preparation is complete, then the next read and write operation will be done.

### Part 10.1: NIO in Linux

Non-blocking IO is handled by polling. Linux provides a polling mechanism for applications and calls the corresponding driver system

#### Part 10.1.1: Polling method in the application

Three polling methods are provided in the application: “**select**”, “**poll**”, and “**epoll**”. In fact, they are also a solution for multiplexing IO, so I won't go into it here.

##### 1) Select

select has good cross-platform support, but there is a maximum limit on the number of file descriptors that a single process can monitor (generally 1024 in Linux).

Function prototype:

```
int select(int maxfdp, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval  
*timeout);
```

### Parameter Description:

**maxfdp:** is the range of all file descriptors in the set, which is equal to the maximum value of all file descriptors plus 1.

**readfds:** The “**struct fd\_set**” structure can be understood as a collection of file descriptors, that is, a file handle. Each bit of it represents a descriptor. “**readfds**” is used to monitor the read changes of the specified descriptor set. As long as one of them is readable, “**select**” will return a value greater than 0. You can use these macros to manipulate the “**fd\_set**” variable:

```
/* Set all positions of fdset to 0, clear the relationship between fdset and all file handles */
FD_ZERO(fd_set *fdset)

/* set a position of fdset 1, associate fdset with file handle fd */
FD_SET(int fd, fd_set *fdset)

/* Set a position of fdset to 0, cancel the association between fdset and file handle */
FD_CLR(int fd, fd_set *fdset)

/* Determine whether a file handle is 1, that is, whether it is operable */
FD_ISSET(int fd, fd_set *fdset)
```

**writelfds:** used to see whether the file is writable.

**errorfds:** used to monitor file exceptions.

**timeout:** “**struct timeval**” is used to represent the time value, there are two members, one is **seconds**, the other is **milliseconds**. It is defined as follows:

```
1. struct timeval{
2.     long tv_sec;    /* seconds */
3.     long tv_usec;   /* microseconds */
4. }
```

There are three situations for this parameter value

If **NULL** is passed in, that is, no time structure is passed in, it is a blocking function and will not return until a file descriptor is changed.

If you set both seconds and microseconds to 0, it is a non-blocking function and will return immediately. If the file is not

changed, it will be 0, and if it is changed, it will be a positive value.

If the value is greater than 0, it means the timeout period. If **select** does not detect the file descriptor change within the **timeout** time, it will directly return to 0, and if there is change, it will return a positive value.

Example:

```
1. void main(void)
2. {
3.     /* ret 获取返回值, fd 获取文件句柄 */
4.     int ret, fd;
5.     /* 定义一个监视文件读变化的描述符集合 */
6.     fd_set readfds;
7.     /* 定义一个超时时间结构体 */
8.     struct timeval timeout;
9.
10.    /* 获取文件句柄, O_NONBLOCK 表示非阻塞访问 */
11.    fd = open("dev_xxx", O_RDWR | O_NONBLOCK);
12.
13.    /* 初始化描述符集合 */
14.    FD_ZERO(&readfds);
15.    /* 把文件句柄 fd 指向的文件添加到描述符集合 */
16.    FD_SET(fd, &readfds);
17.
18.    /* 超时时间初始化为 1.5 秒 */
19.    timeout.tv_sec = 1;
20.    timeout.tv_usec = 500000;
21.
22.    /* 调用 select, 注意第一个参数为 fd+1 */
23.    ret = select(fd + 1, &readfds, NULL, NULL, &timeout);
24.
25.    switch (ret)
26.    {
27.        case 0:
28.        {
29.            /* 超时 */
30.            break;
31.        }
32.        case -1:
33.        {
34.            /* 出错 */
35.            break;
36.        }
37.        default:
38.        {
39.            /* 监视的文件可操作 */
40.            /* 判断可操作的文件是不是文件句柄 fd 指向的文件 */
41.            if(FD_ISSET(fd, &readfds))
42.            {
43.                /* 操作文件 */
44.            }
45.            break;
46.        }
47.    }
48. }
```

**Line 23**, before calling the **select** function, a lot of preparatory work is done, mainly the initialization of the input parameters of the “**select**” function.

**Note Line 11:** The “**O\_NONBLOCK**” attribute in the input

parameter of the “**open**” function. If you need non-blocking access to the file, you need to add this attribute.

**Line 41**, when **ret** returns greater than 0, use the macro definition **FD\_ISSET** to determine whether the operable handle is the handle we need. In the case of only waiting for a file, this judgment is not necessary.

## 2) Poll

**Poll** is essentially the same as **select**, but its maximum number of connections is unlimited.

Function prototype

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

**Parameter Description:**

**fds**: “**struct pollfd**” structure is a combination of file handle and event, defined as follows:

```
1. struct pollfd {
2.     int fd;
3.     short events;
4.     short revents;
5. };
```

**fd** is the file handle, **events** is the type of events that need to be monitored for this file, and **revents** is the type of events returned by the kernel. The event types are:

```
POLLIN    //There is data to read
POLLPRI   //There is urgent data to read
POLLOUT   //data can be written
POLLERR   //An error occurred in the specified file descriptor
POLLHUP   //The specified file descriptor is suspended
POLLNVAL  //invalid request
POLLRDNORM //There is data to read
```

**nfds**: The number of file handles monitored by **poll**, that is, the length of the **fds** array.

**timeout**: Timeout period, in milliseconds.

## Example:

```
1. void main(void)
2. {
3.     /* ret 获取返回值, fd 获取文件句柄 */
4.     int ret, fd;
5.     /* 定义 struct pollfd 结构体变量 */
6.     struct pollfd fds[1];
7.
8.     /* 非阻塞访问文件 */
9.     fd = open(filename, O_RDWR | O_NONBLOCK);
10.
11.    /* 初始化 struct pollfd 结构体变量 */
12.    fds[0].fd = fd;
13.    fds[0].events = POLLIN;
14.
15.    /* 调用 poll */
16.    ret = poll(fds, sizeof(fds), 1500);
17.    if(ret == 0)
18.    {
19.        /* 超时 */
20.    }
21.    else if (ret < 0)
22.    {
23.        /* 错误 */
24.    }
25.    else
26.    {
27.        /* 操作数据 */
28.    }
29. }
```

### 3) Epoll

It can be understood as event poll, which is designed for IO query, and is often used for network programming.

#### Part 10.1.2: Poll function in the driver

When calling “**select**”, “**poll**”, “**epoll**” in the application, the system call will execute the “**poll**” function of the “**file\_operations**” in the driver. That is the function we need to implement. The prototype is as follows:

```
unsigned int (*poll) (struct file *filp, struct poll_table_struct *wait)
```

#### Parameter Description:

**filp**: The value passed by the application, the target file handle obtained after the application is opened.

**wait**: The value passed by the application, representing the application thread. We need to call “**poll\_wait**” in the “**poll**” function. Add the application thread wait to the “**poll\_table**” waiting queue. The

“**poll\_wait**” function prototype is as follows:

```
void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_table *p)
```

**wait** is passed to the “**poll\_wait**” function as the parameter “**p**”.

**Return value:** The return value is the same as the event type in the “**struct pollfd**” structure.

## Part 10.2: Experiment

The experimental of this chapter is the same as that of the previous chapter. Use **ps\_key1** to control **ps\_led1** and reduce the CPU usage compared to Chapter 8.

### Part 10.2.1: Schematic

The led part is the same as **Part 1.3.1**.

The key part is the same as in **Part 6.1**.

### Part 10.2.2: Device tree

Same as **Part 6.2**

### Part 10.2.3: Driver code

Use petalinux to create a new driver named "**ax-bio-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver. Enter the following code in the **ax-bio-drv** file:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/init.h>
4. #include <linux/ide.h>
5. #include <linux/types.h>
6. #include <linux/errno.h>
7. #include <linux/cdev.h>
8. #include <linux/of.h>
9. #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <linux/wait.h>
```

```
20. #include <linux/poll.h>
21. #include <asm/uaccess.h>
22. #include <asm/io.h>
23.
24. /* 设备节点名称 */
25. #define DEVICE_NAME      "nio_led"
26. /* 设备号个数 */
27. #define DEVID_COUNT      1
28. /* 驱动个数 */
29. #define DRIVE_COUNT       1
30. /* 主设备号 */
31. #define MAJOR_U            0
32. /* 次设备号 */
33. #define MINOR_U           0
34.
35. /* 把驱动代码中会用到的数据打包进设备结构体 */
36. struct alinx_char_dev {
37.     /** 字符设备框架 ***/
38.     dev_t          devid;          //设备号
39.     struct cdev    cdev;          //字符设备
40.     struct class   *class;        //类
41.     struct device  *device;       //设备
42.     struct device_node *nd;      //设备树的设备节点
43.     /** gpio **/
44.     int            alinx_key_gpio; //gpio 号
45.     /** 并发处理 **/
46.     atomic_t       key_sts;        //记录按键状态，为 1 时被按下
47.     /** 中断 **/
48.     unsigned int   irq;           //中断号
49.     /** 定时器 **/
50.     struct timer_list timer;      //定时器
51.     /** 等待队列 **/
52.     wait_queue_head_t wait_q_h;   //等待队列头
53. };
54. /* 声明设备结构体 */
55. static struct alinx_char_dev alinx_char = {
56.     .cdev = {
57.         .owner = THIS_MODULE,
58.     },
59. };
60.
61. /** 回掉 **/
62. /* 中断服务函数 */
63. static irqreturn_t key_handler(int irq, void *dev)
64. {
65.     /* 按键按下或抬起时会进入中断 */
66.     /* 开启 50 毫秒的定时器用作防抖动 */
67.     mod_timer(&alinkx_char.timer, jiffies + msecs_to_jiffies(50));
68.     return IRQ_RETVAL(IRQ_HANDLED);
69. }
70.
71. /* 定时器服务函数 */
72. void timer_function(struct timer_list *timer)
73. {
74.     /* value 用于获取按键值 */
75.     unsigned char value;
76.     /* 获取按键值 */
77.     value = gpio_get_value(alinx_char.alinx_key_gpio);
78.     if(value == 0)
79.     {
80.         /* 按键按下，状态置 1 */
81.         atomic_set(&alinkx_char.key_sts, 1);
82.     /** 等待队列 **/
83.         /* 唤醒进程 */
84.         wake_up_interruptible(&alinkx_char.wait_q_h);
85.     }
}
```

```
86.     else
87.     {
88.         /* 按键抬起 */
89.     }
90. }
91.
92. /** 系统调用实现 */
93. /* open 函数实现, 对应到 Linux 系统调用函数的open 函数 */
94. static int char_drv_open(struct inode *inode_p, struct file *file_p)
95. {
96.     printk("gpio_test module open\n");
97.     return 0;
98. }
99.
100. /* read 函数实现, 对应到 Linux 系统调用函数的write 函数 */
101. static ssize_t char_drv_read(struct file *file_p, char user *buf, size_t len, lo f
f_t *loff_t_p)
102. {
103.     unsigned int keysts = 0;
104.     int ret;
105.
106.     /* 读取 key 的状态 */
107.     keysts = atomic_read(&alinx_char.key_sts);
108.     /* 判断文件打开方式 */
109.     if(file_p->f_flags & O_NONBLOCK)
110.     {
111.         /* 如果是非阻塞访问, 说明以满足读取条件 */
112.     }
113.     /* 判断当前按键状态 */
114.     else if(!keysts)
115.     {
116.         /* 按键未被按下(数据未准备好) */
117.         /* 以当前进程创建并初始化为队列项 */
118.         DECLARE_WAITQUEUE(queue_mem, current);
119.         /* 把当前进程的队列项添加到队列头 */
120.         add_wait_queue(&alinx_char.wait_q_h, &queue_mem);
121.         /* 设置当前进成为可被信号打断的状态 */
122.         _set_current_state(TASK_INTERRUPTIBLE);
123.         /* 切换进程, 是当前进程休眠 */
124.         schedule();
125.
126.         /* 被唤醒, 修改当前进程状态为RUNNING */
127.         set_current_state(TASK_RUNNING);
128.         /* 把当前进程的队列项从队列头中删除 */
129.         remove_wait_queue(&alinx_char.wait_q_h, &queue_mem);
130.
131.         /* 判断是否是被信号唤醒 */
132.         if(signal_pending(current))
133.         {
134.             /* 如果是直接返回错误 */
135.             return -ERESTARTSYS;
136.         }
137.         else
138.         {
139.             /* 被按键唤醒 */
140.         }
141.     }
142.     else
143.     {
144.         /* 按键被按下(数据准备好了) */
145.     }
146.
147.     /* 读取 key 的状态 */
148.     keysts = atomic_read(&alinx_char.key_sts);
149.     /* 返回按键状态值 */
150.     ret = copy_to_user(buf, &keysts, sizeof(keysts));
151.     /* 清除按键状态 */
```

```
152.     atomic_set(&alinx_char.key_sts, 0);
153.     return 0;
154. }
155.
156. /* poll 函数实现 */
157. unsigned int char_drv_poll(struct file *filp, struct poll_table_struct *wait)
158. {
159.     unsigned int ret = 0;
160.
161.     /* 将应用程序添加到等待队列中 */
162.     poll_wait(filp, &alinx_char.wait_q_h, wait);
163.
164.     /* 判断 key 的状态 */
165.     if(atomic_read(&alinx_char.key_sts))
166.     {
167.         /* key 准备好了，返回数据可读 */
168.         ret = POLLIN;
169.     }
170.     else
171.     {
172.     }
173. }
174.
175.     return ret;
176. }
177.
178. /* release 函数实现，对应到Linux 系统调用函数的close 函数 */
179. static int char_drv_release(struct inode *inode_p, struct file *file_p)
180. {
181.     printk("gpio_test module release\n");
182.     return 0;
183. }
184.
185. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
186. static struct file_operations ax_char_fops = {
187.     .owner    = THIS_MODULE,
188.     .open     = char_drv_open,
189.     .read     = char_drv_read,
190.     .poll     = char_drv_poll,
191.     .release  = char_drv_release,
192. };
193.
194. /* 模块加载时会调用的函数 */
195. static int init char_drv_init(void)
196. {
197.     /* 用于接受返回值 */
198.     u32 ret = 0;
199.
200.     /** 并发处理 */
201.     /* 初始化原子变量 */
202.     atomic_set(&alinx_char.key_sts, 0);
203.
204.     /** gpio 框架 */
205.     /* 获取设备节点 */
206.     alinx_char.nd = of_find_node_by_path("/alinkkey");
207.     if(alinx_char.nd == NULL)
208.     {
209.         printk("alink_char node not find\r\n");
210.         return -EINVAL;
211.     }
212.     else
213.     {
214.         printk("alink_char node find\r\n");
215.     }
216.
217.     /* 获取节点中 gpio 标号 */
218.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinkkey-gpios",
0);
```

```
219.     if(alinx_char.alinx_key_gpio < 0)
220.     {
221.         printk("can not get alinxkey-gpios");
222.         return -EINVAL;
223.     }
224.     printk("alinkxkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
225.
226.     /* 申请 gpio 标号对应的引脚 */
227.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinkxkey");
228.     if(ret != 0)
229.     {
230.         printk("can not request gpio\r\n");
231.         return -EINVAL;
232.     }
233.
234.     /* 把这个io 设置为输入 */
235.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
236.     if(ret < 0)
237.     {
238.         printk("can not set gpio\r\n");
239.         return -EINVAL;
240.     }
241.
242.     /** 中断 */
243.     /* 获取中断号 */
244.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
245.     /* 申请中断 */
246.     ret = request_irq(alinx_char.irq,
247.                         key_handler,
248.                         IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
249.                         "alinkxkey",
250.                         NULL);
251.     if(ret < 0)
252.     {
253.         printk("irq %d request failed\r\n", alinx_char.irq);
254.         return -EFAULT;
255.     }
256.
257.     /** 定时器 */
258.     timer_setup(&alinkx_char.timer, timer_function, NULL);
259.
260.     /** 等待队列 */
261.     init_waitqueue_head(&alinkx_char.wait_q_h);
262.
263.     /** 字符设备框架 */
264.     /* 注册设备号 */
265.     alloc_chrdev_region(&alinkx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
266.
267.     /* 初始化字符设备结构体 */
268.     cdev_init(&alinkx_char.cdev, &ax_char_fops);
269.
270.     /* 注册字符设备 */
271.     cdev_add(&alinkx_char.cdev, alinx_char.devid, DRIVE_COUNT);
272.
273.     /* 创建类 */
274.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
275.     if(IS_ERR(alinx_char.class))
276.     {
277.         return PTR_ERR(alinx_char.class);
278.     }
279.
280.     /* 创建设备节点 */
281.     alinx_char.device = device_create(alinx_char.class, NULL,
282.                                         alinx_char.devid, NULL,
283.                                         DEVICE_NAME);
284.     if (IS_ERR(alinx_char.device))
285.     {
286.         return PTR_ERR(alinx_char.device);
```

```
287.     }
288.
289.     return 0;
290. }
291.
292. /* 卸载模块 */
293. static void exit char_drv_exit(void)
294. {
295.     /** gpio **/
296.     /* 释放 gpio */
297.     gpio_free(alinx_char.alinx_key_gpio);
298.
299.     /** 中断 **/
300.     /* 释放中断 */
301.     free_irq(alinx_char.irq, NULL);
302.
303.     /** 定时器 **/
304.     /* 删除定时器 */
305.     del_timer_sync(&alinx_char.timer);
306.
307.     /** 字符设备框架 **/
308.     /* 注销字符设备 */
309.     cdev_del(&alinx_char.cdev);
310.
311.     /* 注销设备号 */
312.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
313.
314.     /* 删除设备节点 */
315.     device_destroy(alinx_char.class, alinx_char.devid);
316.
317.     /* 删除类 */
318.     class_destroy(alinx_char.class);
319.
320.     printk("timer_led_dev_exit_ok\n");
321. }
322.
323. /* 标记加载、卸载函数 */
324. module_init(char_drv_init);
325. module_exit(char_drv_exit);
326.
327. /* 驱动描述信息 */
328. MODULE_AUTHOR("Alinx");
329. MODULE_ALIAS("alinx_char");
330. MODULE_DESCRIPTION("NIO LED driver");
331. MODULE_VERSION("v1.0");
332. MODULE_LICENSE("GPL");
```

On the basis of the code in the previous chapter, the driver code adds a **poll** function and slightly modifies the **read** function.

**Line 191**, add the **poll** function to the **file\_operations** structure.

**Line 158**, implement the poll function, call the **poll\_wait** function, and then judge the data status, if the data is ready, return the **POLLIN** status flag.

**Line 110** is slightly modified in the read function. First, determine the file opening mode. If it is accessed in a non-blocking way, it will not do queue-related operations and return the data directly to the user,

otherwise it will be processed as blocked access.

#### Part 10.2.4: Test code

The test code is modified on the basis of **Part 6.4**, the new QT project is named "**ax\_nioled\_test**", the new **main.c** is created, and the following code is entered

```
1. #include "stdio.h"
2. #include "unistd.h"
3. #include "sys/types.h"
4. #include "sys/stat.h"
5. #include "fcntl.h"
6. #include "stdlib.h"
7. #include "string.h"
8. #include "poll.h"
9. #include "sys/select.h"
10. #include "sys/time.h"
11. #include "linux/ioctl.h"
12.
13. int main(int argc, char *argv[])
14. {
15.
16.     /* ret 获取返回值, fd 获取文件句柄 */
17.     int ret, fd, fd_1;
18.     /* 定义一个监视文件读变化的描述符合集 */
19.     fd_set readfds;
20.     /* 定义一个超时时间结构体 */
21.     struct timeval timeout;
22.
23.     char *filename, led_value = 0;
24.     unsigned int key_value;
25.
26.     if(argc != 2)
27.     {
28.         printf("Error Usage\r\n");
29.         return -1;
30.     }
31.
32.     filename = argv[1];
33.     /* 获取文件句柄, O_NONBLOCK 表示非阻塞访问 */
34.     fd = open(filename, O_RDWR | O_NONBLOCK);
35.     if(fd < 0)
36.     {
37.         printf("can not open file %s\r\n", filename);
38.         return -1;
39.     }
40.
41.     while(1)
42.     {
43.         /* 初始化描述符合集 */
44.         FD_ZERO(&readfds);
45.         /* 把文件句柄 fd 指向的文件添加到描述符 */
46.         FD_SET(fd, &readfds);
47.
48.         /* 超时时间初始化为 1.5 秒 */
49.         timeout.tv_sec = 1;
50.         timeout.tv_usec = 500000;
51.
52.         /* 调用 select, 注意第一个参数为 fd+1 */
53.         ret = select(fd + 1, &readfds, NULL, NULL, &timeout);
54.         switch (ret)
55.         {
56.             case 0:
57.             {
58.                 /* 超时 */
59.                 break;
60.             }
61.             case -1:
```

```
62.          {
63.              /* 出错 */
64.              break;
65.          }
66.      default:
67.      {
68.          /* 监视的文件可操作 */
69.          /* 判断可操作的文件是不是文件句柄 fd 指向的文件 */
70.          if(FD_ISSET(fd, &readfds))
71.          {
72.              /* 操作文件 */
73.              ret = read(fd, &key_value, sizeof(key_value));
74.              if(ret < 0)
75.              {
76.                  printf("read failed\r\n");
77.                  break;
78.              }
79.              printf("key_value = %d\r\n", key_value);
80.              if(1 == key_value)
81.              {
82.                  printf("ps_key1 press\r\n");
83.                  led_value = !led_value;
84.
85.                  fd_l = open("/dev/gpio_leds", O_RDWR);
86.                  if(fd_l < 0)
87.                  {
88.                      printf("file /dev/gpio_leds open failed\r\n");
89.                      break;
90.                  }
91.
92.                  ret = write(fd_l, &led_value, sizeof(led_value));
93.                  if(ret < 0)
94.                  {
95.                      printf("write failed\r\n");
96.                      break;
97.                  }
98.
99.                  ret = close(fd_l);
100.                 if(ret < 0)
101.                 {
102.                     printf("file /dev/gpio_leds close failed\r\n");
103.                     break;
104.                 }
105.             }
106.         }
107.     }
108. }
109. }
110. }
111. close(fd);
112. return ret;
113. }
```

Starting from the read function in **Line 73**, the following code is the same as that in **Part 6.4**. By judging the state of the key, the state of the led is changed.

Before calling **read** function, call the **select** function to check the data status. The usage is the same as the usage in **Part 10.1.1**, so the explanation will not be repeated.

### Part 10.2.5: Run test

The test method and experimental results are the same as the

previous chapter, and the steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/ilinx/work /mnt  
cd /mnt  
mkdir /tmp/qt  
mount qt_lib.img /tmp/qt  
cd /tmp/qt  
source ./qt_env_set.sh  
cd /mnt  
insmod ./ax-concled-drv.ko  
insmod ./ax-nio-drv.ko  
cd ./ /mnt/build-ax_nioled_test-IDE_5_7_1_GCC_64bit-Debug  
. /ax-key-test /dev/nio_led&  
top
```

In addition, you can try it and change the **timeout** period in the test to **0** or **NULL** to observe the phenomenon.

## Part 11: Asynchronous IO

The asynchronous IO mentioned here should be called "signal-driven asynchronous I/O" accurately, and it can also be an asynchronous notification. The blocking and non-blocking IO mentioned in the previous two chapters are all synchronous IO and require the application to continuously poll whether the device can be accessed. Under the asynchronous IO model, when the device can be accessed, the driver can actively notify the application to access it. Its form is similar to the interrupt at the hardware level, It can be understood as an analog interrupt mechanism implemented by software.

### Part 11.1: Asynchronous IO in Linux

#### Part 11.1.1: Signal

The Linux system uses signals to realize the asynchronous IO mechanism. Lines 34~72 of the header file “[arch/xtensa/include/uapi/asm/signal.h](#)” define all the signals supported by the Linux system, which is equivalent to the interrupt number in the interrupt system. The use of signals and interrupts are also very similar. A signal corresponds to a Call function, and when the signal is received, the corresponding Call function is executed.

For the application, the signal is an analog interrupt method, which can be used by calling the interface, and the driver needs to implement the corresponding method to provide the underlying support. Let's look at the usage and implementation methods of signals in the application and driver respectively.

### Part 11.1.2: Use of signals in applications

The steps for using signals in an application are as follows:

#### 1) Specify the signal and specify the corresponding signal processing function

Use the following functions to select a signal and specify the corresponding signal processing function:

```
sighandler_t signal(int signum, sighandler_t handler)
```

##### Parameter Description:

**signum:** the signal to be selected, choose one of the macro definitions from **lines 34~72** in the file “arch/xtensa/include/uapi/asm/signal.h”.

**handler:** The corresponding signal processing function.

**Return value:** Return the previous processing function of the signal on success, and return **SGI\_ERR** on failure.

The prototype of the signal processing function **sighandler\_t** is as follows:

```
typedef void (*sighandler_t)(int)
```

#### 2) Set the process ID that will receive the signal

```
fcntl(fd, F_SETOWN, getpid());
```

**fd** is the device file handle, the **F\_SETOWN** command means to set the thread ID that will receive the **SIGIO** or **SIGURG** signal, **getpid()** is to get the current thread ID.

#### 3) Get the current thread state and make it enter the FASYNC state

Use the following method to get the current process status:

```
flag = fcntl(fd, F_GETFL);
```

**flag** is an integer, **fd** is the device file handle, and the **F\_GETFL** command means to get the current thread status. After obtaining the

current thread state, based on the current state, set the process to FASYNC state, that is, turn on the asynchronous notification function of the current thread, use the following command:

```
fcntl(fd, F_SETFL, flags | FASYNC);
```

**fd** is the device file handle, **F\_SETFL** command means to set the current process state. In **flag | FASYNC**, the **flag** is the current line state obtained, or **FASYNC** is added to the current state. When the **FASYNC** state of the thread is set, **fasync** in the **file\_operations** operation function of the corresponding driver will be called.

### Part 11.1.3: Realization of signal in driver

To support signals in the driver, the following steps are required:

- 1) Declare the **fasync\_struct** structure pointer in the device structure:

```
struct xxx_dev {
    .....
    struct fasync_struct *fasync;
}
```

- 2) Implement **fasync** function

The realization of the **fasync()** function generally only needs to pass the 3 parameters of the function and the **fasync\_struct** structure pointer into the **fasync\_helper()** function, as follows:

```
static int xxx_fasync(int fd, struct file *filp, int mode)
{
    struct xxx_dev *dev = filp->private_data;
    return fasync_helper(fd, filp, mode, &dev->fasync);
}

static struct file_operations xxx_ops = {
    .....
    .fasync = xxx_fasync,
};
```

Then call **fasync** in the release function to release the **fasync\_struct** structure, as follows:

```
static int xxx_release(struct inode *inode, struct file *filp)
{
    .....
    return xxx_fasync(-1, filp, 0);
}
```

### 3) Signal when the device is accessible

After the application starts the asynchronous notification, it is waiting for the signal that the device is operable. The driver needs to receive the signal when the device is operable, using the function:

```
void kill_fasync(struct fasync_struct **fp, int sig, int band)
```

#### Parameter Description:

**fp**: The address of the target **fasync\_struct** structure variable pointer.

**sig**: The type of signal to be sent, ranging from 34 to 72 lines of macro definition in the file “[arch/xtensa/include/uapi/asm/signal.h](#),” which needs to be consistent with the requirements of the application.

**band**: Set to “**POLL\_IN**” when the device is writable, and “**POLL\_OUT**” when it is readable.

## Part 11.2: Experiment

The driver adds the implementation of asynchronous IO based on the previous chapter. Then complete the corresponding test procedure.

### Part 11.2.1: Schematic

The led part is the same as **Part 1.3.1**.

The key part is the same as in **Part 6.1**.

## Part 11.2.2: Device tree

Same as Part 6.2

## Part 11.2.3: Driver code

Use petalinux to create a new driver named "**ax-fasync-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-fasync-drv** file:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/init.h>
4. #include <linux/ide.h>
5. #include <linux/types.h>
6. #include <linux/errno.h>
7. #include <linux/cdev.h>
8. #include <linux/of.h>
9. #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <linux/wait.h>
20. #include <linux/poll.h>
21. #include <linux/fcntl.h>
22. #include <asm/uaccess.h>
23. #include <asm/io.h>
24.
25. /* 设备节点名称 */
26. #define DEVICE_NAME      "fasync_led"
27. /* 设备号个数 */
28. #define DEVID_COUNT       1
29. /* 驱动个数 */
30. #define DRIVE_COUNT        1
31. /* 主设备号 */
32. #define MAJOR_U            0
33. /* 次设备号 */
34. #define MINOR_U           0
35.
36. /* 把驱动代码中会用到的数据打包进设备结构体 */
37. struct alinx_char_dev {
38.     dev_t          devid;           //设备号
39.     struct cdev    cdev;            //字符设备
40.     struct class   *class;          //类
41.     struct device  *device;         //设备
42.     struct device_node *nd;         //设备树的设备节点
43.     int            alinx_key_gpio; //gpio 号
44.     atomic_t       key_sts;         //记录按键状态，为1时被按下
45.     unsigned int   irq;             //中断号
46.     struct timer_list timer;        //定时器
47.     wait_queue_head_t wait_q_h;    //等待队列头
48.     struct fasync_struct *fasync;  //异步信号
```

```
49. };
50. /* 声明设备结构体 */
51. static struct alinx_char_dev alinx_char = {
52.     .cdev = {
53.         .owner = THIS_MODULE,
54.     },
55. };
56.
57. /* 中断服务函数 */
58. static irqreturn_t key_handler(int irq, void *dev_in)
59. {
60.     /* 按键按下或抬起时会进入中断 */
61.     struct alinx_char_dev *dev = (struct alinx_char_dev *)dev_in;
62.     /* 开启 50 毫秒的定时器用作防抖动 */
63. //     dev->timer.data = (volatile long)dev_in;
64.     mod_timer(&dev->timer, jiffies + msecs_to_jiffies(50));
65.     return IRQ_RETVAL(IRQ_HANDLED);
66. }
67.
68. /* 定时器服务函数 */
69. void timer_function(struct timer_list *timer)
70. {
71.     struct alinx_char_dev *dev = &alinkx_char;
72.     /* value 用于获取按键值 */
73.     unsigned char value;
74.     /* 获取按键值 */
75.     value = gpio_get_value(dev->alinkx_key_gpio);
76.     if(value == 0)
77.     {
78.         /* 按键按下，状态置 1 */
79.         atomic_set(&dev->key_sts, 1);
80.         /* fasync 有没有初始化过 */
81.         if(dev->fasync)
82.         {
83.             /* 初始化过说明应用程序调用过 */
84.             kill_fasync(&dev->fasync, SIGIO, POLL_OUT);
85.         }
86.         else if((current->state & TASK_INTERRUPTIBLE) != 0)
87.         {
88.             /* 是等待队列，需要唤醒进程 */
89.             wake_up_interruptible(&dev->wait_q_h);
90.         }
91.         else
92.         {
93.             /* do nothing */
94.         }
95.     }
96.     else
97.     {
98.         /* 按键抬起 */
99.     }
100. }
101.
102. /* open 函数实现，对应到 Linux 系统调用函数的open 函数 */
103. static int char_drv_open(struct inode *inode_p, struct file *file_p)
104. {
105.     printk("gpio_test module open\n");
106.     file_p->private_data = &alinkx_char;
107.     return 0;
108. }
109.
110. /* read 函数实现，对应到 Linux 系统调用函数的write 函数 */
111. static ssize_t char_drv_read(struct file *file_p, char user *buf, size_t len, lo_f
f_t *loff_t_p)
112. {
113.     unsigned int keysts = 0;
114.     int ret;
```

```
116.     struct alinx_char_dev *dev = (struct alinx_char_dev *)file_p->private_data;
117.
118.     /* 读取 key 的状态 */
119.     keysts = atomic_read(&dev->key_sts);
120.     /* 判断文件打开方式 */
121.     if(file_p->f_flags & O_NONBLOCK)
122.     {
123.         /* 如果是非阻塞访问，说明已满足读取条件 */
124.     }
125.     /* 判断当前按键状态 */
126.     else if(!keysts)
127.     {
128.         /* 按键未被按下(数据未准备好) */
129.         /* 以当前进程创建并初始化为队列项 */
130.         DECLARE_WAITQUEUE(queue_mem, current);
131.         /* 把当前进程的队列项添加到队列头 */
132.         add_wait_queue(&dev->wait_q_h, &queue_mem);
133.         /* 设置当前进成为可被信号打断的状态 */
134.         _set_current_state(TASK_INTERRUPTIBLE);
135.         /* 切换进程，是当前进程休眠 */
136.         schedule();
137.
138.         /* 被唤醒，修改当前进程状态为RUNNING */
139.         set_current_state(TASK_RUNNING);
140.         /* 把当前进程的队列项从队列头中删除 */
141.         remove_wait_queue(&dev->wait_q_h, &queue_mem);
142.
143.         /* 判断是否是被信号唤醒 */
144.         if(signal_pending(current))
145.         {
146.             /* 如果是直接返回错误 */
147.             return -ERESTARTSYS;
148.         }
149.     else
150.     {
151.         /* 被按键唤醒 */
152.     }
153. }
154. else
155. {
156.     /* 按键被按下(数据准备好了) */
157. }
158.
159. /* 读取 key 的状态 */
160. keysts = atomic_read(&dev->key_sts);
161. /* 返回按键状态值 */
162. ret = copy_to_user(buf, &keysts, sizeof(keysts));
163. /* 清除按键状态 */
164. atomic_set(&dev->key_sts, 0);
165. return 0;
166. }
167.
168. /* poll 函数实现 */
169. unsigned int char_drv_poll(struct file *file_p, struct poll_table_struct *wait)
170. {
171.     unsigned int ret = 0;
172.
173.     struct alinx_char_dev *dev = (struct alinx_char_dev *)file_p->private_data;
174.
175.     /* 将应用程序添加到等待队列中 */
176.     poll_wait(file_p, &dev->wait_q_h, wait);
177.
178.     /* 判断 key 的状态 */
179.     if(atomic_read(&dev->key_sts))
180.     {
181.         /* key 准备好了，返回数据可读 */
182.         ret = POLLIN;
```

```
183.     }
184.     else
185.     {
186.     }
187.     }
188.
189.     return ret;
190. }
191.
192. /* fasync 函数实现 */
193. static int char_drv_fasync(int fd, struct file *file_p, int mode)
194. {
195.     struct alinx_char_dev *dev = (struct alinx_char_dev *)file_p->private_data;
196.     return fasync_helper(fd, file_p, mode, &dev->fasync);
197. }
198.
199. /* release 函数实现, 对应到Linux 系统调用函数的close 函数 */
200. static int char_drv_release(struct inode *inode_p, struct file *file_p)
201. {
202.     printk("gpio_test module release\n");
203.     return char_drv_fasync(-1, file_p, 0);
204. }
205.
206. /* file_operations 结构体声明, 是上面 open、write 实现函数与系统调用函数对应的关键 */
207. static struct file_operations ax_char_fops = {
208.     .owner      = THIS_MODULE,
209.     .open       = char_drv_open,
210.     .read       = char_drv_read,
211.     .poll       = char_drv_poll,
212.     .fasync    = char_drv_fasync,
213.     .release   = char_drv_release,
214. };
215.
216. /* 模块加载时会调用的函数 */
217. static int init char_drv_init(void)
218. {
219.     /* 用于接受返回值 */
220.     u32 ret = 0;
221.
222.     /* 初始化原子变量 */
223.     atomic_set(&alinkx_char.key_sts, 0);
224.
225.     /* 获取设备节点 */
226.     alinx_char.nd = of_find_node_by_path("/alinkxkey");
227.     if(alinkx_char.nd == NULL)
228.     {
229.         printk("alinkx_char node not find\r\n");
230.         return -EINVAL;
231.     }
232.     else
233.     {
234.         printk("alinkx_char node find\r\n");
235.     }
236.
237.     /* 获取节点中 gpio 标号 */
238.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinkxkey-gpios",
239.     0);
240.     if(alinkx_char.alinx_key_gpio < 0)
241.     {
242.         printk("can not get alinxkey-gpios");
243.         return -EINVAL;
244.     }
245.     printk("alinkxkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
246.
247.     /* 申请 gpio 标号对应的引脚 */
248.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinkxkey");
249.     if(ret != 0)
250.     {
251.         printk("can not request gpio\r\n");
```

```
251.         return -EINVAL;
252.     }
253.
254.     /* 把这个io 设置为输入 */
255.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
256.     if(ret < 0)
257.     {
258.         printk("can not set gpio\r\n");
259.         return -EINVAL;
260.     }
261.
262.     /* 获取中断号 */
263.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
264.     /* 申请中断 */
265.     ret = request_irq(alinx_char.irq,
266.                         key_handler,
267.                         IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
268.                         "alinxkey",
269.                         &alinkx_char);
270.     if(ret < 0)
271.     {
272.         printk("irq %d request failed\r\n", alinx_char.irq);
273.         return -EFAULT;
274.     }
275.
276.     timer_setup(&alinkx_char.timer, timer_function, 0);
277.
278.     init_waitqueue_head(&alinkx_char.wait_q_h);
279.
280.     /* 注册设备号 */
281.     alloc_chrdev_region(&alinkx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
282.
283.     /* 初始化字符设备结构体 */
284.     cdev_init(&alinkx_char.cdev, &ax_char_fops);
285.
286.     /* 注册字符设备 */
287.     cdev_add(&alinkx_char.cdev, alinx_char.devid, DRIVE_COUNT);
288.
289.     /* 创建类 */
290.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
291.     if(IS_ERR(alinx_char.class))
292.     {
293.         return PTR_ERR(alinx_char.class);
294.     }
295.
296.     /* 创建设备节点 */
297.     alinx_char.device = device_create(alinx_char.class, NULL,
298.                                         alinx_char.devid, NULL,
299.                                         DEVICE_NAME);
300.     if (IS_ERR(alinx_char.device))
301.     {
302.         return PTR_ERR(alinx_char.device);
303.     }
304.
305.     return 0;
306. }
307.
308. /* 卸载模块 */
309. static void_exit char_drv_exit(void)
310. {
311.     /* 释放 gpio */
312.     gpio_free(alinx_char.alinx_key_gpio);
313.
314.     /* 释放中断 */
315.     free_irq(alinx_char.irq, NULL);
316.
317.     /* 删除定时器 */
318.     del_timer_sync(&alinkx_char.timer);
```

```
319.  
320.     /* 注销字符设备 */  
321.     cdev_del(&alinx_char.cdev);  
322.  
323.     /* 注销设备号 */  
324.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);  
325.  
326.     /* 删除设备节点 */  
327.     device_destroy(alinx_char.class, alinx_char.devid);  
328.  
329.     /* 删除类 */  
330.     class_destroy(alinx_char.class);  
331.  
332.     printk("timer_led_dev_exit_ok\n");  
333. }  
334.  
335. /* 标记加载、卸载函数 */  
336. module_init(char_drv_init);  
337. module_exit(char_drv_exit);  
338.  
339. /* 驱动描述信息 */  
340. MODULE_AUTHOR("Alinx");  
341. MODULE_ALIAS("alinx_char");  
342. MODULE_DESCRIPTION("FASYNC LED driver");  
343. MODULE_VERSION("v1.0");  
344. MODULE_LICENSE("GPL");
```

**Line 21**, add the header file [linux/fcntl.h](#).

**Line 48**, add the [fasync\\_struct](#) structure to the device structure.

**Line 81**, in the timer processing function, after confirming that the key is pressed, first determine whether the [fasync\\_struct](#) structure has been initialized. If it is initialized, it means that the [fasync](#) function has been called. That is, the application can receive the asynchronous notification, so it sends the corresponding signal.

**Lines 193~197** implement the [fasync](#) function, which simply calls [fasync\\_helper](#) to initialize the [fasync\\_struct](#) structure.

**Line 200**, the [fasync](#) function is called in the release function to release the [fasync\\_struct](#) structure.

**Line 212**, add the [fasync](#) function we implemented to the [file\\_operations](#) structure.

#### Part 11.2.4: Test code

Create a new QT project named "[ax\\_fasync\\_test](#)", create a new [main.c](#), and enter the following code:

```
1. #include "stdio.h"
2. #include "unistd.h"
3. #include "sys/types.h"
4. #include "sys/stat.h"
5. #include "fcntl.h"
6. #include "stdlib.h"
7. #include "string.h"
8. #include "poll.h"
9. #include "sys/select.h"
10. #include "sys/time.h"
11. #include "linux/ioctl.h"
12. #include "signal.h"
13.
14. static int fd = 0, fd_l = 0;
15.
16. static void sigio_signal_func()
17. {
18.     int ret = 0;
19.     static char led_value = 0;
20.     unsigned int key_value;
21.
22.     /* 获取按键状态 */
23.     ret = read(fd, &key_value, sizeof(key_value));
24.     if(ret < 0)
25.     {
26.         printf("read failed\r\n");
27.     }
28.
29.     /* 判断按键状态 */
30.     if(1 == key_value)
31.     {
32.         /* 按键被按下，改变吗 led 状态 */
33.         printf("ps_key1 press\r\n");
34.         led_value = !led_value;
35.
36.         fd_l = open("/dev/gpio_leds", O_RDWR);
37.         if(fd_l < 0)
38.         {
39.             printf("file /dev/gpio_leds open failed\r\n");
40.         }
41.
42.         ret = write(fd_l, &led_value, sizeof(led_value));
43.         if(ret < 0)
44.         {
45.             printf("write failed\r\n");
46.         }
47.
48.         ret = close(fd_l);
49.         if(ret < 0)
50.         {
51.             printf("file /dev/gpio_leds close failed\r\n");
52.         }
53.     }
54. }
55.
56. int main(int argc, char *argv[])
57. {
58.     int flags = 0;
59.     char *filename;
60.
61.     if(argc != 2)
62.     {
63.         printf("wrong para\n");
64.         return -1;
65.     }
66.
67.     filename = argv[1];
68.     fd = open(filename, O_RDWR);
69.     if(fd < 0)
70.     {
71.         printf("can not open file %s\r\n", filename);
72.         return -1;
73.     }
74.
75.     /* 指定信号 SIGIO，并绑定处理函数 */
76.     signal(SIGIO, sigio_signal_func);
77.     /* 把当前线程指定为将接收信号的进程 */
78.     fcntl(fd, F_SETOWN, getpid());
```

```
79.     /* 获取当前线程状态 */
80.     flags = fcntl(fd, F_GETFD);
81.     /* 设置当前线程为 FASYNC 状态 */
82.     fcntl(fd, F_SETFL, flags | FASYNC);
83.
84.     while(1)
85.     {
86.         sleep(2);
87.     }
88.
89.     close(fd);
90.     return 0;
91. }
```

### Note:

Line 19 adds **static** to led\_value.

Lines 75~82 just follow the steps mentioned in **Part 11.1.2**, and will not repeat the explanation.

### Part 11.2.5: Run test

The test method and experimental results are the same as the previous chapter, and the steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-concler-drv.ko
insmod ./ax-fasync-drv.ko
cd ./mnt/build-ax_fasync_test-IDE_5_7_1_GCC_64bit-Debug
./ax_fasync_test /dev/fasync_led&
top
```

The results are as follows:

```
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.107:/home/alink/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
random: fast init done
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-concled-drv.ko
ax_concled dev: loading out-of-tree module taints kernel.
alink_char node find
alinkled-gpio num = 899
root@ax_peta:/mnt# insmod ./ax-fasync-drv.ko
alink_char node find
alinkxkey-gpio num = 949
root@ax_peta:/mnt# cd ./build-ax_fasync_test-ZYNQ-Debug
root@ax_peta:/mnt/build-ax_fasync_test-ZYNQ-Debug# ./ax_fasync_test /dev/fasync_led&
[1] 1266
root@ax_peta:/mnt/build-ax_fasync_test-ZYNQ-Debug# gpio_test module open
ps_keyl press
gpio_test module open
gpio_test module release
ps_keyl press
gpio_test module open
gpio_test module release
```

```
Mem: 57072K used, 973180K free, 13408K shrd, 1196K buff, 43064K cached
CPU: 0.0% usr 0.0% sys 0.0% nic 100% idle 0.0% io 0.0% irq 0.0% sirq
Load average: 0.00 0.00 0.00 1/87 1267
PID  PPID USER      STAT  VSZ %VSZ CPU %CPU COMMAND
1242    1 root      S    14352  1.3   1  0.0 /usr/sbin/tcf-agent -d -L- -10
1266  1249 root      S    13164  1.2   1  0.0 ./ax_fasync_test /dev/fasync_led
1249  1247 root      S     2988  0.2   1  0.0 -sh
  756    1 root      S    2972  0.2   1  0.0 /sbin/udevd -d
1169    1 root      S    2916  0.2   0  0.0 /usr/sbin/inetd
1228    1 root      S    2788  0.2   0  0.0 /sbin/syslogd -n -O /var/log/messages
1231    1 root      S    2788  0.2   1  0.0 /sbin/klogd -n
1267  1249 root      R    2788  0.2   1  0.0 top
1213    1 root      S    2788  0.2   0  0.0 udhcpc -R -b -p /var/run/udhcpc.eth0.pid -i eth0
1248    1 root      S    2788  0.2   1  0.0 /sbin/getty 38400 ttym
1247    1 root      S    2756  0.2   0  0.0 {start_getty} /bin/sh /bin/start_getty 115200 ttym
1220    1 root      S    2364  0.2   0  0.0 /usr/sbin/dropbear -r /etc/dropbear/dropbear_rsa_h
  1    0 root      S    1700  0.1   1  0.0 init
1254    2 root      SW<    0  0.0   0  0.0 [kworker/0:1H]
```

## Part 12: Platform

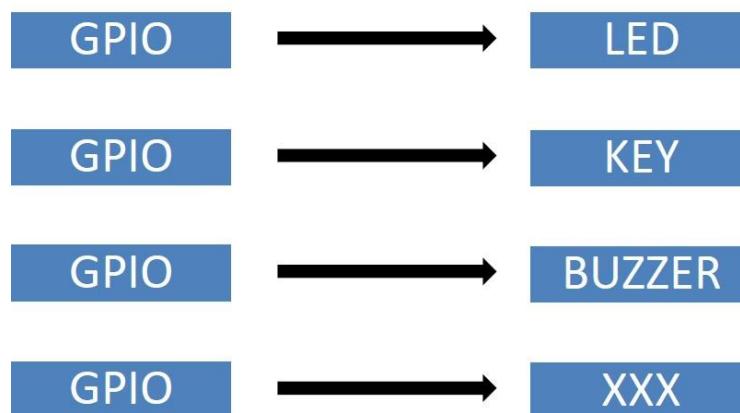
The character device framework has always been used to write simple IO read and write drivers, but when encountering complex buses and devices, only the character device framework cannot handle it.

For example, when multiple devices are mounted on the SPI bus, how should the drivers of these devices be implemented? How to implement SPI driver? How are they related? Although the function can be realized, the final code is complicated and does not conform to the Linux design philosophy. In this case, the idea of driver separation is needed. The Linux kernel provides a platform device framework.

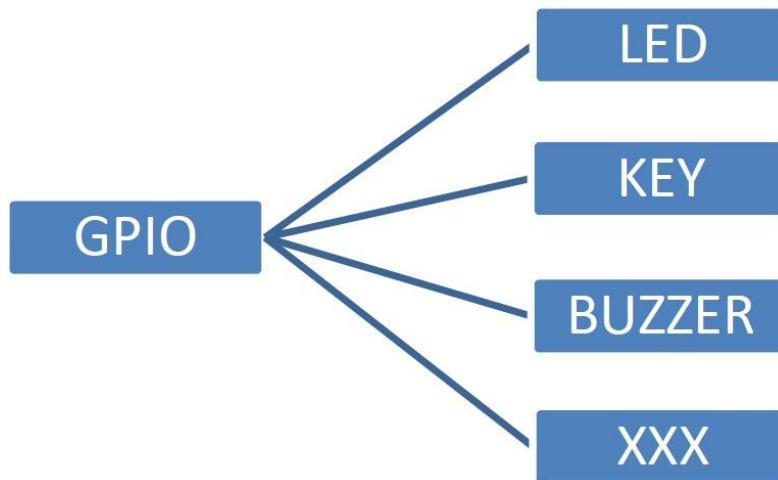
### Part 12.1: Drive Separation

Before introducing platform, let's briefly introduce the idea of driver separation.

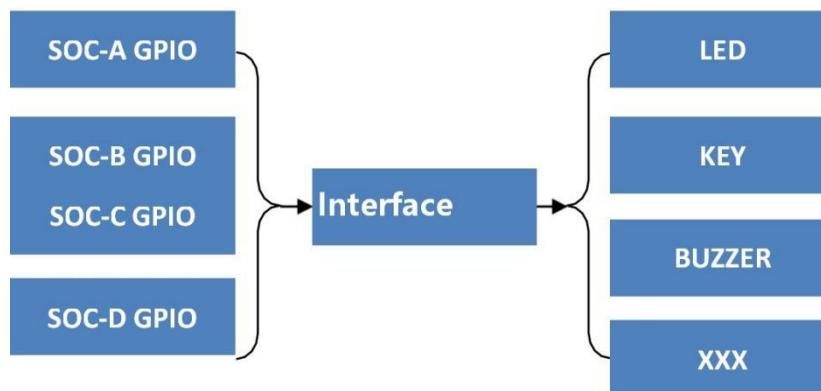
Have you ever thought about this question? In the LED lighting experiment introduced earlier, is it IO driver or LED driver? For the time being combined as an **IO\_LED** driver, then we used the buttons again and had the **IO\_KEY** driver. If we want to use the buzzer now, the **IO\_BUZZER** driver will appear. For each additional device, there will be an additional **IO\_XXX** driver.



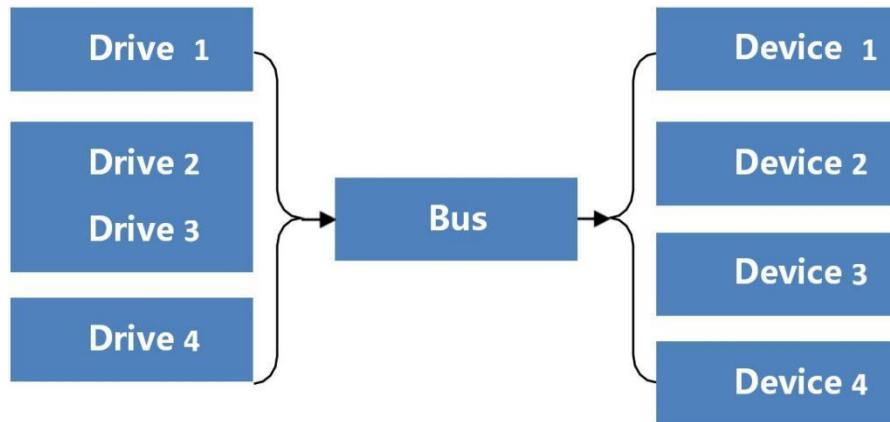
However, in these IO\_XXX drivers, the IO part is almost the same. In order to ensure the simplicity of the code, the IO part is extracted separately and written as an IO driver, and then the device driver is left with the contents of the device itself. Just call the interface in the IO driver.



Here comes the question. A major advantage of Linux is that it is easy to transplant. Now what will happen if these device drivers need to be run on other hardware platforms? The implementation of the GPIO bottom layer on SOCs of different manufacturers is different, so every time a new SOC appears, a GPIO driver needs to be rewritten. Of course, this is inevitable, but these GPIO drivers still have a common part, that is, the interface between the IO and the device. The interface part of these GPIOs is extracted and written as a driver separately.



This forms the separation of drivers. One side is the hardware resource of the SOC, and the other side is the user equipment, and they are connected through a unified interface. The idea of driver separation has brought many benefits. In the long-term development of Linux, a lot of redundant code has been omitted. SOC manufacturers provide drivers for SOC on-chip peripherals based on a unified interface, and equipment manufacturers also provide device drivers based on a unified interface. The user also needs to select the SOC and peripherals so that they can be easily linked together. This idea has also continued to a single SOC, and the on-chip driver (referred to as the driver later) and the device driver (referred to as the device later) are related through the bus protocol.



When adding a driver to the kernel, the bus will find the corresponding device, and when adding a device, it will find the corresponding driver. This is the bus, driver, and device models in Linux, and platform is the realization of this mode

## Part 12.2: Platform Model

Bus devices such as SPI can easily correspond to bus, driver, and device models, but not all resources on SOC have the concept of a bus, such as GPIO. In order to correspond to the bus, driver, and device models, the platform model defines platform\_bus virtual bus,

platform\_driver and platform\_device to correspond to the model bus, driver, and device, respectively. One thing to note is that although the platform device model has "platform device" in its name, but it is not a new device type independent of the three Linux devices. The emphasized model is just a framework. In use, it cannot be separated from character devices, block devices, and network devices.

### Part 12.2.1: platform\_bus

The **bus\_type** structure is used in the kernel to represent the bus, which is defined in the file “[include/linux/device.h](#)”:

```
1. struct bus_type {
2.     .....
3.     int     )(struct device *dev, struct device_driver *drv);
4.     .....
5. };
```

The **match** function in the member is the key to the match between the driver and the device. One of its two input parameters is **drv** and the other is **dev**, that is, the driver and the device. Each bus type must implement this function.

The **platform** virtual bus is also a variable of type **bus\_type**, defined as follows in the file “[drivers/base/platform.c](#)”

```
1. struct bus_type platform_bus_type = {
2.     .name      = "platform",
3.     .dev_groups = platform_dev_groups,
4.     .match     = platform_match,
5.     .uevent    = platform_uevent,
6.     .pm= &platform_dev_pm_ops,
7. };
```

The implementation of the match function, **platform\_match** function, is also in this file:

```
1. static int platform_match(struct device *dev, struct device_driver *drv)
2. {
3.     struct platform_device *pdev = to_platform_device(dev);
4.     struct platform_driver *pdrv = to_platform_driver(drv);
5.
6.     /* When driver_override is set, only bind to the matching driver */
7.     if (pdev->driver_override)
8.         return !strcmp(pdev->driver_override, drv->name);
9.
10.    /* Attempt an OF style match first */
11.    if (of_driver_match_device(dev, drv))
12.        return 1;
```

```
13.     /* Then try ACPI style match */
14.     if (acpi_driver_match_device(dev, drv))
15.         return 1;
16.
17.     /* Then try to match against the id table */
18.     if (pdrv->id_table)
19.         return platform_match_id(pdrv->id_table, pdev) != NULL;
20.
21.     /* fall-back to driver name match */
22.     return (strcmp(pdev->name, drv->name) == 0);
23.
24. }
```

The **platform\_match** function provides 4 matching methods.

Lines 11~12, the **OF** matching table matching method will be used under the device tree. One of the input parameters of **match**. There is a member variable **of\_match\_table** in the data type of **drv**. the **probe** in the driver code will be called.

If the device tree is not used, the fourth matching method in line 23 is generally used to directly compare the name members in the driver and the device.

The platform bus does not need to be managed at the driver level

### Part 12.2.2: platform\_driver

#### 1) Define and initialize platform\_driver

The **platform** driver is represented by the “**platform\_driver**” structure, in the header file “**include/linux/platform\_device.h**”

```
1. struct platform_driver {
2.     int (*probe)(struct platform_device *); 
3.     int (*remove)(struct platform_device *); 
4.     void (*shutdown)(struct platform_device *); 
5.     int (*suspend)(struct platform_device *, pm_message_t state); 
6.     int (*resume)(struct platform_device *); 
7.     struct device_driver driver; 
8.     const struct platform_device_id *id_table; 
9.     bool prevent_deferred_probe; 
10.};
```

##### a. Member probe

As mentioned earlier, the **probe** function will execute this function when the device and the driver match successfully. In this function, call the content originally called in the driver entry function, such as the “**cdev\_init**” function called in the character device.

##### b. Member remove

The **remove** function is executed when the driver or the corresponding device is unregistered. As opposed to **probe**, this function calls the content originally called in the driver exit function, such as the **cdev\_del** function in the character device and other initialization content.

### c. Member driver

The driver is of the **struct device\_driver** type, **device\_driver** is the basic device driver type, and **platform\_driver** is extended on the basis of **device\_driver**, so you need to include elements of this type to use its members.

There is a member variable in the **device\_driver** structure

```
const struct of_device_id*of_match_table;
```

**of\_match\_table** is the member used in the **OF** matching table matching method mentioned above. There is a member named **compatible** in the **of\_device\_id** structure. “**Compatible**” in the device tree is compared with this member.

There is also a member named “**name**” in the **of\_device\_id** structure, which needs to be the same as the “**name**” field in the platform device.

## 2) platform\_driver registration and deregistration

After defining **struct platform\_driver**, you need to call the following function in the driver entry function to register the **platform** driver, replacing the original initialization content:

```
int platform_driver_register(struct platform_driver*driver);
```

It returns 0 if the registration is successful and negative if it fails.

Do the corresponding unregister operation in the exit function to replace the original unregister content:

```
void platform_driver_unregister(struct platform_driver *drv);
```

### Part 12.2.3: platform\_device

The **platform** device is represented by the structure “**platform\_device**”. In the kernel that supports the device tree, the device tree can be used instead of **platform\_device**, but **platform\_device** is still reserved. Let's first understand the complete process of **platform**, and then combine the device tree.

#### 1) platform\_device structure

The “**platform\_device**” structure is defined in “**include/linux/platform\_device.h**”, and the content is as follows:

```
1. struct platform_device {  
2.     const char *name;  
3.     int id;  
4.     bool id_auto;  
5.     struct device dev;  
6.     u32 num_resources;  
7.     struct resource *resource;  
8.  
9.     const struct platform_device_id *id_entry;  
10.    char *driver_override; /* Driver name to force a match */  
11.  
12.    /* MFD cell pointer */  
13.    struct mfd_cell *mfd_cell;  
14.  
15.    /* arch specific additions */  
16.    struct pdev_archdata archdata;  
17.};
```

The member **name** is used to match the driver and needs to be the same as the **name** in **platform\_driver**.

The member **id** represents the number of the current device in this type of device, and another device of this type is assigned the **id** value of **-1**.

The member **num\_resources** represents the number of resources.

The member **resource** is an array of resources, and **struct resource** is defined as follows:

```
1. struct resource {  
2.     resource_size_t start;  
3.     resource_size_t end;  
4.     const char *name;  
5.     unsigned long flags;  
6.     struct resource *parent, *sibling, *child;  
7.};
```

“**start**” represents the start address of the resource, “**end**” represents the end address of the resource. “**name**” is the name of the resource, “**flags**” indicate the resource type. The macro definition of the resource type is in **Line 29~105** of “[include/linux/iport.h](#)”.

## 2) platform\_device registration and deregistration

After declaring the initialization of the “**platform\_device**” structure, use the following method to register “**platform\_device**”:

```
int platform_device_register(struct platform_device *pdev);
```

When logging off the device, use the following method to log off:

```
void platform_device_unregister(struct platform_device *pdev);
```

## 3) Methods of obtaining resources

After the platform device has set the resources, the platform driver can obtain the resource information through the following functions:

```
struct resource *platform_get_resource(struct platform_device *dev,unsigned int type,  
unsigned int num)
```

### Parameter Description:

**dev**: target platform device.

**type**: the flags member of the resource structure of the “**platform\_device**” structure member mentioned above.

**num**: Specify the subscript of the type resource.

**Return value**: resource information, the “**resource**” structure type pointer is returned on success, and “**NULL**” is returned on failure.

## Part 12.3: Experiment

In this chapter, we use the platform architecture to implement a simple lighting led experiment.

### Part 12.3.1: Schematic

The led part is the same as **Part 1.3.1**.

### Part 12.3.2: Device tree

The experiments in this chapter use **platform\_device** to represent the device instead of the device tree.

### Part 12.3.3: Driver code

The driver is divided into two parts: the driver and the device.

- 1) Complete the driver code first. Use **petalinux** to create a new driver named "**ax-platform-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-fasync-drv.c** file:

```
1. #include <linux/types.h>
2. #include <linux/kernel.h>
3. #include <linux/delay.h>
4. #include <linux/ide.h>
5. #include <linux/init.h>
6. #include <linux/module.h>
7. #include <linux/errno.h>
8. #include <linux/gpio.h>
9. #include <linux/cdev.h>
10. #include <linux/device.h>
11. #include <linux/of_gpio.h>
12. #include <linux/semaphore.h>
13. #include <linux/timer.h>
14. #include <linux/irq.h>
15. #include <linux/wait.h>
16. #include <linux/poll.h>
17. #include <linux/fs.h>
18. #include <linux/fcntl.h>
19. #include <linux/platform_device.h>
20. #include <asm/uaccess.h>
21. #include <asm/io.h>
22.
23. /* 设备节点名称 */
24. #define DEVICE_NAME      "gpio_leds"
25. /* 设备号个数 */
26. #define DEVID_COUNT      1
27. /* 驱动个数 */
28. #define DRIVE_COUNT       1
29. /* 主设备号 */
30. #define MAJOR
31. /* 次设备号 */
32. #define MINOR            0
33.
34. /* gpio 寄存器虚拟地址 */
35. static u32 *GPIO_DIRM_1;
36. /* gpio 使能寄存器 */
37. static u32 *GPIO_OEN_1;
38. /* gpio 控制寄存器 */
39. static u32 *GPIO_DATA_1;
40. /* AMBA 外设时钟使能寄存器 */
41. static u32 *APER_CLK_CTRL;
42.
43. /* 把驱动代码中会用到的数据打包进设备结构体 */
44. struct alinx_char_dev{
45.     dev_t          devid;      //设备号
46.     struct cdev    cdev;       //字符设备
```

```
47.     struct class      *class;      //类
48.     struct device     *device;     //设备
49. };
50. /* 声明设备结构体 */
51. static struct alinx_char_dev alinx_char = {
52.     .cdev = {
53.         .owner = THIS_MODULE,
54.     },
55. };
56.
57. /* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
58. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
59. {
60.     /* 设置私有数据 */
61.     file_p->private_data = &alinkx_char;
62.
63.     return 0;
64. }
65.
66. /* write 函数实现，对应到 Linux 系统调用函数的 write 函数 */
67. static ssize_t gpio_leds_write(struct file *file_p, const char user *buf, size_t len, loff_t *loff_t_p)
68. {
69.     int rst;
70.     char writeBuf[5] = {0};
71.
72.     rst = copy_from_user(writeBuf, buf, len);
73.     if(0 != rst)
74.     {
75.         return -1;
76.     }
77.
78.     if(1 != len)
79.     {
80.         printk("gpio_test len err\n");
81.         return -2;
82.     }
83.     if(1 == writeBuf[0])
84.     {
85.         *GPIO_DATA_1 |= 0x00004000;
86.     }
87.     else if(0 == writeBuf[0])
88.     {
89.         *GPIO_DATA_1 &= 0xFFFFBFFF;
90.     }
91.     else
92.     {
93.         printk("gpio_test para err\n");
94.         return -3;
95.     }
96.
97.     return 0;
98. }
99.
100. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
101. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
102. {
103.     return 0;
104. }
105.
106. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
107. static struct file_operations ax_char_fops = {
108.     .owner    = THIS_MODULE,
109.     .open     = gpio_leds_open,
110.     .write    = gpio_leds_write,
111.     .release  = gpio_leds_release,
112. };
113.
114. /* probe 函数实现，驱动和设备匹配时会被调用 */
115. static int gpio_leds_probe(struct platform_device *dev)
116. {
117.     /* 资源大小 */
118.     int regsize[3];
119.     /* 资源信息 */
120.     struct resource *led_source[3];
121.
122.     int i;
123.     for(i = 0; i < 3; i ++)
```

```
124.    {
125.        /* 获取 dev 中的 IORESOURCE_MEM 资源 */
126.        led_source[i] = platform_get_resource(dev, IORESOURCE_MEM, i);
127.        /* 返回 NULL 获取资源失败 */
128.        if(!led_source[i])
129.        {
130.            dev_err(&dev->dev, "get resource %d failed\r\n", i);
131.            return -ENXIO;
132.        }
133.        /* 获取当前资源大小 */
134.        regsize[i] = resource_size(led_source[i]);
135.    }
136.
137.    /* 把需要修改的物理地址映射到虚拟地址 */
138.    GPIO_DIRM_1    = ioremap(led_source[0]->start, regsize[0]);
139.    GPIO_OEN_1     = ioremap(led_source[1]->start, regsize[1]);
140.    GPIO_DATA_1   = ioremap(led_source[2]->start, regsize[2]);
141.
142.    /* MIO_0 设置成输出 */
143.    *GPIO_DIRM_1 |= 0x00004000;
144.    /* MIO_0 使能 */
145.    *GPIO_OEN_1 |= 0x00004000;
146.
147.    /* 注册设备号 */
148.    alloc_chrdev_region(&alinx_char.devid, MINOR, DEVID_COUNT, DEVICE_NAME);
149.
150.    /* 初始化字符设备结构体 */
151.    cdev_init(&alinx_char.cdev, &ax_char_fops);
152.
153.    /* 注册字符设备 */
154.    cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
155.
156.    /* 创建类 */
157.    alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
158.    if(IS_ERR(alinx_char.class))
159.    {
160.        return PTR_ERR(alinx_char.class);
161.    }
162.
163.    /* 创建设备节点 */
164.    alinx_char.device = device_create(alinx_char.class, NULL,
165.                                       alinx_char.devid, NULL,
166.                                       DEVICE_NAME);
167.    if (IS_ERR(alinx_char.device))
168.    {
169.        return PTR_ERR(alinx_char.device);
170.    }
171.
172.    return 0;
173. }
174.
175. static int gpio_leds_remove(struct platform_device *dev)
176. {
177.    /* 注销字符设备 */
178.    cdev_del(&alinx_char.cdev);
179.
180.    /* 注销设备号 */
181.    unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
182.
183.    /* 删除设备节点 */
184.    device_destroy(alinx_char.class, alinx_char.devid);
185.
186.    /* 删除类 */
187.    class_destroy(alinx_char.class);
188.
189.    /* 释放对虚拟地址的占用 */
190.    *GPIO_OEN_1 &= 0xFFFFBFFF;
191.    iounmap(GPIO_DIRM_1);
192.    iounmap(GPIO_OEN_1);
193.    iounmap(GPIO_DATA_1);
194.    return 0;
195. }
196.
197. /* 声明并初始化 platform 驱动 */
198. static struct platform_driver led_driver = {
199.     .driver = {
200.         /* 将会用 name 字段和设备匹配, 这里 name 命名为 alinx-led */
```

```
201.         .name = "alinx-led",
202.     },
203.     .probe = gpio_leds_probe,
204.     .remove = gpio_leds_remove,
205. };
206.
207. /* 驱动入口函数 */
208. static int initgpio_led_drv_init(void)
209. {
210.     /* 在入口函数中调用 platform_driver_register, 注册 platform 驱动 */
211.     return platform_driver_register(&led_driver);
212. }
213.
214. /* 驱动出口函数 */
215. static void exitgpio_led_dev_exit(void)
216. {
217.     /* 在出口函数中调用 platform_driver_unregister, 卸载 platform 驱动 */
218.     platform_driver_unregister(&led_driver);
219. }
220.
221. /* 标记加载、卸载函数 */
222. module_init(gpio_led_drv_init);
223. module_exit(gpio_led_dev_exit);
224.
225. /* 驱动描述信息 */
226. MODULE_AUTHOR("Alinx");
227. MODULE_ALIAS("gpio_led");
228. MODULE_DESCRIPTION("PLATFORM LED driver");
229. MODULE_VERSION("v1.0");
230. MODULE_LICENSE("GPL");
```

Compared with the driver code under the device tree in Part 2, the character device part is almost the same. The **open** function, **write** function, and **release** function are all familiar character device driver writing methods.

**Line 115** implements the probe function, copies the content in the driver entry function of the driver code in [Part2](#), and modifies the resource acquisition method. The second one is to obtain resource information from the device tree, modified to obtain it from the platform device. Using the “[platform\\_get\\_resource](#)” function, the resource size can be obtained through the “[resource\\_size](#)” function.

**Line 175** implements the remove function, just copy the content in the driver export function of the driver code in Part 2, even without modification.

**Line 198** defines “[platform\\_driver](#)” and initializes it.

The name on **Line 201** is named “[alink-led](#)”, and then when implementing “[platform\\_device](#)”, it must be consistent.

**Line 211** registers [platform\\_driver](#) in the driver entry function.

The 218 line unregisters platform\_driver in the export function.

- 2) Then complete the device part, use **petalinux** to create a new driver named "**ax-platform-dev**", and execute "**petalinux-config -c rootfs**" command to select the new program.

Enter the following code in the "**ax-platform-dev.c**" file:

```
1. #include <linux/init.h>
2. #include <linux/module.h>
3. #include <linux/errno.h>
4. #include <linux/gpio.h>
5. #include <linux/cdev.h>
6. #include <linux/device.h>
7. #include <linux/of_gpio.h>
8. #include <linux/semaphore.h>
9. #include <linux/timer.h>
10. #include <linux/irq.h>
11. #include <linux/wait.h>
12. #include <linux/poll.h>
13. #include <linux/fs.h>
14. #include <linux/fcntl.h>
15. #include <linux/platform_device.h>
16. #include <asm/uaccess.h>
17. #include <asm/io.h>
18.
19. /* 寄存器首地址 */
20. /* gpio 方向寄存器 */
21. #define GPIO_DIRM_1          0xFF0A0244
22. /* gpio 使能寄存器 */
23. #define GPIO_OEN_1            0xFF0A0248
24. /* gpio 控制寄存器 */
25. #define GPIO_DATA_1           0xFF0A0044
26. /* 寄存器大小 */
27. #define REGISTER_LENGTH      4
28.
29. /* 删除设备时会执行此函数 */
30. static void led_release(struct device *dev)
31. {
32.     printk("led device released\n");
33. }
34.
35. /* 初始化 LED 的设备信息，即寄存器信息 */
36. static struct resource led_resources[] =
37. {
38.     {
39.         .start = GPIO_DIRM_1,
40.         .end   = GPIO_DIRM_1 + REGISTER_LENGTH - 1,
41.         /* 寄存器当作内存处理 */
42.         .flags = IORESOURCE_MEM,
43.     },
44.     {
45.         .start = GPIO_OEN_1,
46.         .end   = GPIO_OEN_1 + REGISTER_LENGTH - 1,
47.         .flags = IORESOURCE_MEM,
48.     },
49.     {
50.         .start = GPIO_DATA_1,
51.         .end   = GPIO_DATA_1 + REGISTER_LENGTH - 1,
52.         .flags = IORESOURCE_MEM,
53.     },
54. };
55.
56. /* 声明并初始化 platform_device */
57. static struct platform_device led_device =
58. {
59.     /* 名字和 driver 中的 name 一致 */
60.     .name = "alinx-led",
61.     /* 只有一个设备 */
62.     .id = -1,
```

```
63.     .dev = {  
64.         /* 设置 release 函数 */  
65.         .release = &led_release,  
66.     },  
67.     /* 设置资源个数 */  
68.     .num_resources = ARRAY_SIZE(led_resources),  
69.     /* 设置资源信息 */  
70.     .resource = led_resources,  
71. };  
72.  
73. /* 入口函数 */  
74. static int init_led_device_init(void)  
75. {  
76.     /* 在入口函数中调用 platform_driver_register, 注册 platform 驱动 */  
77.     return platform_device_register(&led_device);  
78. }  
79.  
80. /* 出口函数 */  
81. static void exit_led_device_exit(void)  
82. {  
83.     /* 在出口函数中调用 platform_driver_unregister, 卸载 platform 驱动 */  
84.     platform_device_unregister(&led_device);  
85. }  
86.  
87. /* 标记加载、卸载函数 */  
88. module_init(led_device_init);  
89. module_exit(led_device_exit);  
90.  
91. /* 驱动描述信息 */  
92. MODULE_AUTHOR("Alinx");  
93. MODULE_ALIAS("gpio_led");  
94. MODULE_DESCRIPTION("PLATFORM LED device");  
95. MODULE_VERSION("v1.0");  
96. MODULE_LICENSE("GPL");
```

**platform\_device** also uses the driver entry and exit method, and the method of loading the device is the same as the driver using the **insmod** command.

### **platform\_device**, two key points

One is the **struct resource** array starting at line 39. This is the device information we need. The three member variables that each element needs to initialize are “**start**”, “**end**”, and “**flags**”. When we call the “**resource\_size**” function in “**drv**”, it will be based on start and end return the resource size, flags is one of the basis for “**platform\_get\_resource**” function to obtain resource information.

The second is the implementation of the “**platform\_device**” structure in line 65. The key is that the “**name**” field must be consistent with the “**drv**”. The value of the “**num\_resource**” member can be obtained through the macro “**ARRAY\_SIZE**” to obtain the number of resources.

**Line 33** implements the release function, which will be executed when the device is deleted.

**Line 85** registers the platform\_device device in the entry function.

**Line 92** unregisters the platform\_device device in the exit function.

#### Part 12.3.4: Test code

The test APP is consistent with **Part 1.3.4**, and you can use the test program in **Part 1**.

#### Part 12.3.5: Run test

The test steps are as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/ilinx/work /mnt  
cd /mnt  
mkdir /tmp/qt  
mount qt_lib.img /tmp/qt  
cd /tmp/qt  
source ./qt_env_set.sh  
cd /mnt  
insmod ax-platform-dev.ko  
insmod ax-platform-drv.ko  
cd ./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug/  
../axleddev_test /dev/gpio_leds on
```

The IP and path are adjusted according to the actual situation.

The result of the experiment is that **ps\_led1** on the FPGA development board will be lit or extinguished:

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ax-platform-dev.ko
root@ax_peta:/mnt# insmod ax-platform-drv.ko
[drm] load() is defered & will be called again
root@ax_peta:/mnt# cd ./build-axeddev_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug# ./axeddev_test /dev/gpio_leds on
ps_ledl on
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug#
```

If we delete the device, the device file is gone:

```
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug# rmmod ax_platform_dev
led device released
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug# ./axeddev_test /dev/gpio_leds on
Can't open file /dev/gpio_leds
```

Reload it again, it exists again, indicating that the probe function is executed when dev and drv match

```
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug# insmod ../ax-platform-dev.ko
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug# [drm] load() is defered & will be called again
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug# ./axeddev_test /dev/gpio_leds on
ps_ledl on
root@ax_peta:/mnt/build-axeddev_test-ZYNQ-Debug#
```

# Part 13: Platform and Device Tree

## Part 13.1: Platform under the device tree

After introducing the device tree, the **platform\_device** in the platform structure can be replaced by the device tree. Compared with the original platform driver, the platform driver under the device tree only needs to put the device information described in **platform\_device** into the device tree and modify the resource reading method in **paltform\_drvier**.

### 1) Describe device information in the device tree

When using **platform\_device**, we can match the device and driver through the “**name**” field or “**id\_table**”. When **platform\_device** becomes the device tree, according to the introduction in **platform\_bus** in the previous chapter, we use the **of\_match\_table** method to match. “**of\_match\_table**”, for the device tree, the thing to do is to ensure that the “**compatible**” attribute of the device node is consistent with the “**compatible**” in **platform\_driver**.

```
1. alinxled {  
2.     compatible = "alinx-led";  
3.     pinctrl-names = "default";  
4.     pinctrl-0 = <&pinctrl_led_default>;  
5.     alinxled-gpios = <&gpio0 0 0>;  
6. };
```

In addition to this point, there is nothing special about the writing of the device tree.

### 2) of\_match\_table

The “**compatible**” attribute setting in “**paltform\_drvier**” has been introduced in the previous chapter. “**compatible**” is located in “**paltform\_drvier->device\_driver-> of\_device\_id->compatible**”, and it must be consistent with the “**compatible**” field in the device tree.

The member of “`of_device_id`” structure in “`paltform_drvier`” structure is named “`of_match_table`”, “`of`” match table. The initialization example is as follows:

```
1. static const struct of_device_id led_of_match[] = {  
2.     /* compatible 字段和设备树中保持一致 */  
3.     { .compatible = "alinx-led" },  
4.     {/* Sentinel */}  
5. };
```

Pay Attention:

The last member of “`of_device_id`” must be empty.

Even if the “`name`” field is not used for matching, the “`name`” field in the “`device_driver`” structure still needs to be reserved, otherwise an error may occur when loading the driver.

### 3) Modify the method of `paltform_drvier` to read device information

Since “`platform_device`” has been replaced by a device tree, the way “`paltform_drvier`” reads device information must be replaced by the “`of`” function. Reviewing the “`of`” function in Part 4, you can start to implement the platform driver code under the device tree.

## Part 13.2: Experiment

The experiment in this chapter is based on the previous chapter, using the device tree instead of “`platform_device`” to implement a simple light-up led experiment using the platform framework.

### Part 13.2.1: Schematic

Same as Part 1.3.1

### Part 13.2.2: Device tree

Here we use the device tree under the `pinctrl` subsystem and `gpio` subsystem in Part 4. The writing is the same, just use it directly. Just pay attention to the “`compatible`” attribute in the device node,

which should be consistent with the “**compatible**” in “**platform\_driver**”.

```
1. ....
2. amba {
3.     ....
4.
5.     slcr@f8000000 {
6.         pinctrl@700 {
7.             pinctrl_led_default: led-default {
8.                 mux {
9.                     groups = "gpio0_0_grp";
10.                    function = "gpio0";
11.                };
12.
13.                conf {
14.                    pins = "MIO0";
15.                    io-standard = <1>;
16.                    bias-disable;
17.                    slew-rate = <0>;
18.                };
19.            };
20.        };
21.    };
22. };
23.
24. alinxled {
25.     compatible = "alinx-led";
26.     pinctrl-names = "default";
27.     pinctrl-0 = <&pinctrl_led_default>;
28.     alinxled-gpios = <&gpio0 0 0>;
29. };
30. ....
```

### Part 13.2.3: Driver code

Use **petalinux** to create a new driver named “**ax-platformdt-drv**”, and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-platformdt-drv.c** file:

```
1. #include <linux/types.h>
2. #include <linux/kernel.h>
3. #include <linux/delay.h>
4. #include <linux/ide.h>
5. #include <linux/init.h>
6. #include <linux/module.h>
7. #include <linux/errno.h>
8. #include <linux/gpio.h>
9. #include <linux/cdev.h>
10. #include <linux/device.h>
11. #include <linux/of_gpio.h>
12. #include <linux/semaphore.h>
13. #include <linux/timer.h>
14. #include <linux/irq.h>
15. #include <linux/wait.h>
```

```
16. #include <linux/poll.h>
17. #include <linux/fs.h>
18. #include <linux/fcntl.h>
19. #include <linux/platform_device.h>
20.
21. #include <asm/uaccess.h>
22. #include <asm/io.h>
23.
24. /* 设备节点名称 */
25. #define DEVICE_NAME      "gpio_leds"
26. /* 设备号个数 */
27. #define DEVID_COUNT      1
28. /* 驱动个数 */
29. #define DRIVE_COUNT       1
30. /* 主设备号 */
31. #define MAJOR_AX
32. /* 次设备号 */
33. #define MINOR_AX          0
34. /* LED 点亮时输入的值 */
35. #define ALINX_LED_ON       1
36. /* LED 熄灭时输入的值 */
37. #define ALINX_LED_OFF      0
38.
39. /* 把驱动代码中会用到的数据打包进设备结构体 */
40. struct alinx_char_dev{
41.     dev_t           devid;        //设备号
42.     struct cdev     cdev;         //字符设备
43.     struct class    *class;       //类
44.     struct device   *device;      //设备
45.     struct device_node *nd;       //设备树的设备节点
46.     int             ax_led_gpio; //gpio 号
47. };
48. /* 声明设备结构体 */
49. static struct alinx_char_dev alinx_char = {
50.     .cdev = {
51.         .owner = THIS_MODULE,
52.     },
53. };
54.
55. /* open 函数实现，对应到Linux 系统调用函数的 open 函数 */
56. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
57. {
58.     /* 设置私有数据 */
59.     file_p->private_data = &alinkx_char;
60.
61.     return 0;
62. }
63.
64. /* write 函数实现，对应到 Linux 系统调用函数的write 函数 */
65. static ssize_t gpio_leds_write(struct file *file_p, const char_user *buf, size_t len, loff_t *loff_t_p)
66. {
67.     int retvalue;
68.     unsigned char databuf[1];
69.     /* 获取私有数据 */
70.     struct alinx_char_dev *dev = file_p->private_data;
71.
72.     /* 获得用户数据 */
73.     retvalue = copy_from_user(databuf, buf, len);
74.     if(retvalue < 0)
75.     {
76.         printk("alink led write failed\r\n");
77.         return -EFAULT;
78.     }
```

```
79.
80.     if(datbuf[0] == ALINX_LED_ON)
81.     {
82.         /* gpio_set_value 方法设置GPIO 的值，使用!!对 0 或者 1 二值化 */
83.         gpio_set_value(dev->ax_led_gpio, !!1);
84.     }
85.     else if(datbuf[0] == ALINX_LED_OFF)
86.     {
87.         gpio_set_value(dev->ax_led_gpio, !!0);
88.     }
89.     else
90.     {
91.         printk("gpio_test para err\n");
92.     }
93.
94.     return 0;
95. }
96.
97. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
98. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
99. {
100.    return 0;
101. }
102.
103. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关
键 */
104. static struct file_operations ax_char_fops = {
105.     .owner    = THIS_MODULE,
106.     .open     = gpio_leds_open,
107.     .write    = gpio_leds_write,
108.     .release  = gpio_leds_release,
109. };
110.
111. /* probe 函数实现，驱动和设备匹配时会被调用 */
112. static int gpio_leds_probe(struct platform_device *dev)
113. {
114.     /* 用于接受返回值 */
115.     u32 ret = 0;
116.
117.     /* 获取设备节点 */
118.     alinx_char.nd = of_find_node_by_path("/alinxled");
119.     if(alinx_char.nd == NULL)
120.     {
121.         printk("gpioled node not find\r\n");
122.         return -EINVAL;
123.     }
124.
125.     /* 获取节点中 gpio 标号 */
126.     alinx_char.ax_led_gpio = of_get_named_gpio(alinx_char.nd, "alinkled-gpios",
127.     0);
128.     if(alinx_char.ax_led_gpio < 0)
129.     {
130.         printk("can not get alinkled-gpios\r\n");
131.         return -EINVAL;
132.     }
133.     /* 申请 gpio 标号对应的引脚 */
134.     ret = gpio_request(alinx_char.ax_led_gpio, "alinkled");
135.     if(ret != 0)
136.     {
137.         printk("can not request gpio\r\n");
138.     }
139.     /* 把这个 io 设置为输出 */
140.     ret = gpio_direction_output(alinx_char.ax_led_gpio, 1);
141.     if(ret < 0)
```

```
142.     {
143.         printk("can not set gpio\r\n");
144.     }
145.
146.     /* 注册设备号 */
147.     alloc_chrdev_region(&alinx_char.devid, MINOR_AX, DEVID_COUNT, DEVICE_NAME);
148.
149.     /* 初始化字符设备结构体 */
150.     cdev_init(&alinx_char.cdev, &ax_char_fops);
151.
152.     /* 注册字符设备 */
153.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
154.
155.     /* 创建类 */
156.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
157.     if (IS_ERR(alinx_char.class))
158.     {
159.         return PTR_ERR(alinx_char.class);
160.     }
161.
162.     /* 创建设备节点 */
163.     alinx_char.device = device_create(alinx_char.class, NULL,
164.                                         alinx_char.devid, NULL,
165.                                         DEVICE_NAME);
166.     if (IS_ERR(alinx_char.device))
167.     {
168.         return PTR_ERR(alinx_char.device);
169.     }
170.
171.     return 0;
172. }
173.
174. static int gpio_leds_remove(struct platform_device *dev)
175. {
176.     /* 注销字符设备 */
177.     cdev_del(&alinx_char.cdev);
178.
179.     /* 注销设备号 */
180.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
181.
182.     /* 删除设备节点 */
183.     device_destroy(alinx_char.class, alinx_char.devid);
184.
185.     /* 删除类 */
186.     class_destroy(alinx_char.class);
187.     return 0;
188. }
189.
190. /* 初始化 of_match_table */
191. static const struct of_device_id led_of_match[] = {
192.     /* compatible 字段和设备树中保持一致 */
193.     { .compatible = "alinx-led" },
194.     {/* Sentinel */}
195. };
196.
197.
198. /* 声明并初始化 platform 驱动 */
199. static struct platform_driver led_driver = {
200.     .driver = {
201.         /* name 字段需要保留 */
202.         .name = "alinx-led",
203.         /* 用 of_match_table 替代 name 匹配 */
204.         .of_match_table = led_of_match,
```

```
205.     },
206.     .probe  = gpio_leds_probe,
207.     .remove = gpio_leds_remove,
208. };
209.
210. /* 驱动入口函数 */
211. static int init gpio_led_drv_init(void)
212. {
213.     /* 在入口函数中调用 platform_driver_register, 注册 platform 驱动 */
214.     return platform_driver_register(&led_driver);
215. }
216.
217. /* 驱动出口函数 */
218. static void exit gpio_led_dev_exit(void)
219. {
220.     /* 在出口函数中调用 platform_driver_unregister, 卸载 platform 驱动 */
221.     platform_driver_unregister(&led_driver);
222. }
223.
224. /* 标记加载、卸载函数 */
225. module_init(gpio_led_drv_init);
226. module_exit(gpio_led_dev_exit);
227.
228. /* 驱动描述信息 */
229. MODULE_AUTHOR("Alinx");
230. MODULE_ALIAS("gpio_led");
231. MODULE_DESCRIPTION("PLATFORM DT LED driver");
232. MODULE_VERSION("v1.0");
233. MODULE_LICENSE("GPL");
```

Combining the driver code in Part 4 with the code in Part 12, the code in this chapter is actually a combination of the driver code in these two chapters.

The platform structure follows the structure of Part 12, and the method of obtaining device tree resources follows the method of Part 4. The “`read`”, “`write`”, and “`release`” methods are also the same as in Part 4.

The only thing to pay attention to is the driver field in Line 200, and the “`name`” should be kept. The “`compatible`” field in “`of_match_table`” must be consistent with that in the device tree.

#### Part 13.2.4: Test code

The test APP is consistent with **Part 1.3.4**, and you can use the test program in **Part 1**.

### Part 13.2.5: Run Test

The test method is basically the same as the previous chapter, and the command to load the device is omitted. The steps are as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ax-platformdt-drv.ko
cd ./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug
./axleddev_test /dev/gpio_leds on
```

The IP and path are adjusted according to the actual situation.

In addition to the led experiment results, the debugging results in the serial port tool are as follows:

```
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
random: fast init done
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ax-platformdt-drv.ko
ax_platformdt_drv: loading out-of-tree module taints kernel.
[drm] load() is deferred & will be called again
root@ax_peta:/mnt# cd ./build-axleddev_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
ps_led1 on
```

## Part 14: MISC device driver

Do so many experiments using character devices. Comparing the driver codes of these chapters, it is not difficult to find that a large part of the code in the character device framework is the same. Therefore, the Linux kernel proposes a MISC driver model to encapsulate these common codes.

### Part 14.1: Introduction to MISC device

MISC, such as buzzer, adc, led and other difficult-to-classify devices, can be driven by the MISC framework.

The MISC driver model is actually an encapsulation of the character device driver model, which is essentially a character device. The source directory of MISC is [/drivers/char/misc.c](#). Compared with the general character device driver framework, the benefits of this layer of MISC encapsulation are as follows:

#### 1) Save the major device number

In Line 214 of the “[file /drivers/char/misc.c](#)”, in the “[misc\\_register](#)” function, use the macro “[MISC\\_MAJOR](#)” to indicate the major device number to register the minor device number. The “[MISC\\_MAJOR](#)” macro is defined in line 25 of the file “[include/uapi/linux/major.h](#)”, The value is [10](#), which means that the major device number of the “[MISC](#)” device is fixed at [10](#). Character devices registered with the “[MISC](#)” framework will be assigned a minor device number under the major device number [10](#) (some minor device numbers are already occupied by the kernel). However, a major device number must be assigned when the common character device frame is registered. Therefore, compared with the general character device frame, the use of the “[MISC](#)” frame can save the

major device number.

## 2) Concise and easy to use

The general character device driver framework needs to use a series of functions of `alloc_chrdev_region()`, `cdev_init()`, `cdev_add()`, `class_create()`, `device_create()` to complete the creation of the device, but the MISC framework uses the `misc_register()` function to do the device creation process with good optimization and packaging, we need to call this function to complete the creation of the device, which is much more convenient.

## 3) Drive layering

MISC is also a manifestation of the hierarchical thinking of Linux drivers, which provides attribution types for devices that are difficult to classify and facilitates their use.

### Part 14.2: Use of MISC framework

The use of the MISC framework is just a replacement for the general character device framework.

Methods as below.

#### 1) Create and initialize the miscdevice structure

“`misdveice`” means “`misc`” device, defined in the file “`include/linux/miscdevice.h`”, as follows:

```
1. struct miscdevice {
2.     int minor;
3.     const char *name;
4.     const struct file_operations *fops;
5.     struct list_head list;
6.     struct device *parent;
7.     struct device *this_device;
8.     const struct attribute_group **groups;
9.     const char *nodename;
10.    umode_t mode;
11.};
```

There is no major device number in the structure member, because the major device number of “`misc`” devices is fixed to 10.

After declaring the “`misdveice`” structure variable, three member

variables need to be initialized:

The minor on the **Line 2** is the sub-device number. Devices registered with the “**misc**” framework are distinguished by sub-device numbers. Some “**misc**” sub-device numbers are already occupied in the kernel. You can view the occupied sub-device numbers in **lines 14~54** of the file “[include/linux/miscdevice.h](#)” (the line numbers of different kernel source codes may be different). If you need to check the occupied sub-device number in the system, you can “**ls**” the path “[/sys/dev/char](#)” as shown in the figure below. The first item in the “**ls**” result is **10:1**, the **10** before the colon is the main device, and the **1** after the colon is the minor device number, which means that the minor device number 1 under the major device number 10 is already occupied.

```
root@ax_peta:/sys/dev/char# ls
10:1  10:61  1:1  1:8  254:0  4:11  4:17  4:22  4:28  4:33  4:39  4:44  4:5  4:55  4:60  4:9  7:128
90:2
10:130 10:62  1:11 1:9  254:1  4:12  4:18  4:23  4:29  4:34  4:4  4:45  4:50  4:56  4:61  5:0  7:129
90:3
10:20  10:63  1:3  246:0  254:2  4:13  4:19  4:24  4:3  4:35  4:40  4:46  4:51  4:57  4:62  5:1  89:0
90:4
10:237 116:33 1:4  247:0  4:0  4:14  4:2  4:25  4:30  4:36  4:41  4:47  4:52  4:58  4:63  5:2  89:1
90:5
10:59  13:63  1:5  248:0  4:1  4:15  4:20  4:26  4:31  4:37  4:42  4:48  4:53  4:59  4:7  7:0  90:0
90:6
10:60  189:0  1:7  253:0  4:10  4:16  4:21  4:27  4:32  4:38  4:43  4:49  4:54  4:6  4:8  7:1  90:1
90:7
```

**Line 3**, “**name**” device name. The node name that will be generated in the “[/dev](#)” directory.

**Line 4**, “**fops**”, device operation function set. Same as “[fops](#)” implemented in character devices

## 2) Use **misc\_register** function to register the device instead of the original registration process

After the “[miscdevice](#)” is declared and initialized, use the “[misc\\_register](#)” function to register the “**misc**” device. The function is defined in the file “[/drivers/char/misc.c](#)”. The prototype is as follows:

```
int misc_register(struct miscdevice * misc)
```

**misc** is the [miscdevice](#) structure variable.

It returns **0** if the registration is successful, and returns a negative

number if it fails.

### 3) Use `misc_deregister` function to deregister the device instead of the original `deregister` process

The opposite of registration is the unregister function, which replaces a series of unregister operations in the driver exit function for general character devices. The function prototype:

```
void misc_deregister(struct miscdevice *misc)
```

**misc** is the `miscdevice` structure variable.

## Part 14.3: Experiment

The experiment in this chapter is modified on the basis of the previous chapter **Part 13**. There is almost no difference between combining the “**misc**” framework under the platform framework and modifying the general character device framework to the “**misc**” framework.

### Part 14.3.1: Schematic

Same as Part 1.3.1

### Part 14.3.2: Device tree

Same as **Part 13.2.2**

### Part 14.3.3: Driver code

Use **petalinux** to create a new driver named "**ax-misc-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-misc-drv.c** file:

```
1. #include <linux/types.h>
2. #include <linux/kernel.h>
3. #include <linux/delay.h>
4. #include <linux/ide.h>
5. #include <linux/init.h>
6. #include <linux/module.h>
7. #include <linux/errno.h>
```

```

8. #include <linux/gpio.h>
9. #include <linux/cdev.h>
10. #include <linux/device.h>
11. #include <linux/of_gpio.h>
12. #include <linux/semaphore.h>
13. #include <linux/timer.h>
14. #include <linux/irq.h>
15. #include <linux/wait.h>
16. #include <linux/poll.h>
17. #include <linux/fs.h>
18. #include <linux/fcntl.h>
19. #include <linux/platform_device.h>
20. #include <linux/miscdevice.h>
21.
22. #include <asm/uaccess.h>
23. #include <asm/io.h>
24.
25. /* 设备节点名称 */
26. #define DEVICE_NAME      "gpio_leds"
27. /* 设备号个数 */
28. #define DEVID_COUNT      1
29. /* 驱动个数 */
30. #define DRIVE_COUNT       1
31. /* 主设备号 */
32. #define MAJOR_AX
33. /* 次设备号 */
34. #define MINOR_AX          20
35. /* LED 点亮时输入的值 */
36. #define ALINX_LED_ON       1
37. /* LED 熄灭时输入的值 */
38. #define ALINX_LED_OFF      0
39.
40. /* 把驱动代码中会用到的数据打包进设备结构体 */
41. struct alinx_char_dev{
42.     dev_t           devid;        //设备号
43.     struct cdev     cdev;         //字符设备
44.     struct class    *class;       //类
45.     struct device   *device;      //设备
46.     struct device_node *nd;       //设备树的设备节点
47.     int             ax_led_gpio; //gpio 号
48. };
49. /* 声明设备结构体 */
50. static struct alinx_char_dev alinx_char = {
51.     .cdev = {
52.         .owner = THIS_MODULE,
53.     },
54. };
55.
56. /* open 函数实现，对应到Linux 系统调用函数的 open 函数 */
57. static int gpio_leds_open(struct inode *inode_p, struct file *file_p)
58. {
59.     /* 设置私有数据 */
60.     file_p->private_data = &alink_char;
61.
62.     return 0;
63. }
64.
65. /* write 函数实现，对应到 Linux 系统调用函数的write 函数 */
66. static ssize_t gpio_leds_write(struct file *file_p, const char_user *buf, size_t len, loff_t *loff_t_p)
67. {
68.     int retvalue;
69.     unsigned char databuf[1];
70.     /* 获取私有数据 */

```

```
71.     struct alinx_char_dev *dev = file_p->private_data;
72.
73.     /* 获取用户数据 */
74.     retval = copy_from_user(databuf, buf, len);
75.     if(retval < 0)
76.     {
77.         printk("alink led write failed\r\n");
78.         return -EFAULT;
79.     }
80.
81.     if(databuf[0] == ALINX_LED_ON)
82.     {
83.         /* gpio_set_value 方法设置GPIO 的值，使用!!对 0 或者 1 二值化 */
84.         gpio_set_value(dev->ax_led_gpio, !!1);
85.     }
86.     else if(databuf[0] == ALINX_LED_OFF)
87.     {
88.         gpio_set_value(dev->ax_led_gpio, !!0);
89.     }
90.     else
91.     {
92.         printk("gpio_test para err\n");
93.     }
94.
95.     return 0;
96. }
97.
98. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
99. static int gpio_leds_release(struct inode *inode_p, struct file *file_p)
100. {
101.     return 0;
102. }
103.
104. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
105. static struct file_operations ax_char_fops = {
106.     .owner    = THIS_MODULE,
107.     .open     = gpio_leds_open,
108.     .write    = gpio_leds_write,
109.     .release  = gpio_leds_release,
110. };
111.
112. /* MISC 设备结构体 */
113. static struct miscdevice led_miscdev = {
114.     .minor = MINOR_AX,
115.     .name  = DEVICE_NAME,
116.     /* file_operations 结构体 */
117.     .fops   = &ax_char_fops,
118. };
119.
120. /* probe 函数实现，驱动和设备匹配时会被调用 */
121. static int gpio_leds_probe(struct platform_device *dev)
122. {
123.     /* 用于接受返回值 */
124.     u32 ret = 0;
125.
126.     /* 获得设备节点 */
127.     alinx_char.nd = of_find_node_by_path("/alinkled");
128.     if(alinx_char.nd == NULL)
129.     {
130.         printk("gpioled node host find\r\n");
131.         return -EINVAL;
132.     }
133.
134.     /* 获得节点中 gpio 标号 */
```

```
135.     alinx_char.ax_led_gpio = of_get_named_gpio(alinx_char.nd, "alinxled-gpios", 0);
136.     if(alinx_char.ax_led_gpio < 0)
137.     {
138.         printk("can not get alinxled-gpios\r\n");
139.         return -EINVAL;
140.     }
141.
142.     /* 申请 gpio 标号对应的引脚 */
143.     ret = gpio_request(alinx_char.ax_led_gpio, "alinxled");
144.     if(ret != 0)
145.     {
146.         printk("can not request gpio\r\n");
147.     }
148.     /* 把这个 io 设置为输出 */
149.     ret = gpio_direction_output(alinx_char.ax_led_gpio, 1);
150.     if(ret < 0)
151.     {
152.         printk("can not set gpio\r\n");
153.     }
154.
155.     /*
156.     alloc_chrdev_region(&alinx_char.devid, MINOR_AX, DEVID_COUNT, DEVICE_NAME) ;
157.
158.     cdev_init(&alinx_char.cdev, &ax_char_fops);
159.
160.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
161.
162.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
163.     if(IS_ERR(alinx_char.class))
164.     {
165.         return PTR_ERR(alinx_char.class);
166.     }
167.
168.     alinx_char.device = device_create(alinx_char.class, NULL,
169.                                         alinx_char.devid, NULL,
170.                                         DEVICE_NAME);
171.     if (IS_ERR(alinx_char.device))
172.     {
173.         return PTR_ERR(alinx_char.device);
174.     }
175.     */
176.
177.     /* 注册 misc 设备 */
178.     ret = misc_register(&led_miscdev);
179.     if(ret < 0)
180.     {
181.         printk("misc device register failed\r\n");
182.         return -EFAULT;
183.     }
184.
185.     return 0;
186.
187. static int gpio_leds_remove(struct platform_device *dev)
188. {
189.     /*
190.     cdev_del(&alinx_char.cdev);
191.
192.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
193.
194.     device_destroy(alinx_char.class, alinx_char.devid);
195.
196.     class_destroy(alinx_char.class);
197.     */
198.
```

```
198.
199.     /* 注销 misc 设备 */
200.     misc_deregister(&led_miscdev);
201.     return 0;
202. }
203.
204. /* 初始化 of_match_table */
205. static const struct of_device_id led_of_match[] = {
206.     /* compatible 字段和设备树中保持一致 */
207.     { .compatible = "alinx-led" },
208.     {/* Sentinel */}
209. };
210.
211.
212. /* 声明并初始化 platform 驱动 */
213. static struct platform_driver led_driver = {
214.     .driver = {
215.         /* name 字段需要保留 */
216.         .name = "alinx-led",
217.         /* 用 of_match_table 代替 name 匹配 */
218.         .of_match_table = led_of_match,
219.     },
220.     .probe = gpio_leds_probe,
221.     .remove = gpio_leds_remove,
222. };
223.
224. /* 驱动入口函数 */
225. static int init gpio_led_drv_init(void)
226. {
227.     /* 在入口函数中调用 platform_driver_register, 注册 platform 驱动 */
228.     return platform_driver_register(&led_driver);
229. }
230.
231. /* 驱动出口函数 */
232. static void exit gpio_led_dev_exit(void)
233. {
234.     /* 在出口函数中调用 platform_driver_unregister, 卸载 platform 驱动 */
235.     platform_driver_unregister(&led_driver);
236. }
237.
238. /* 标记加载、卸载函数 */
239. module_init(gpio_led_drv_init);
240. module_exit(gpio_led_dev_exit);
241.
242. /* 驱动描述信息 */
243. MODULE_AUTHOR("Alinx");
244. MODULE_ALIAS("gpio_led");
245. MODULE_DESCRIPTION("MISC LED driver");
246. MODULE_VERSION("v1.0");
247. MODULE_LICENSE("GPL");
```

Compared with the previous chapter **Part 13**, there are few changes, see the **bold** part.

**Line 34**, Set the minor device number to 20. First check the minor device numbers that have been occupied by the kernel. Here, the 20 we used is not occupied, and other minor device numbers that are not occupied can be used.

**Lines 113~118**, define and initialize the miscdevice structure.

**Lines 155~175**, the commented out part of the driver entry function is replaced with the misc\_register function of **lines 117~182** to achieve the purpose of registering the device. Obviously, the code has been simplified a lot.

**Lines 189~197**, the cancellation function in the driver exit function is replaced with “**misc\_deregister**” in **Line 200**. It also simplifies a lot.

#### Part 14.3.4: Test code

The test APP is consistent with **Part 1.3.4**, and you can use the test program in **Part 1**.

#### Part 14.3.5: Run Test

The test method is basically the same as the previous chapter **Part 13**, and the command to load the device is omitted. The steps are as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt  
cd /mnt  
mkdir /tmp/qt  
mount qt_lib.img /tmp/qt  
cd /tmp/qt  
source ./qt_env_set.sh  
cd /mnt  
insmod ax-misc-drv.ko  
cd ./build-axleddev_test-IDE_5_7_1_GCC_64bit-Debug  
../axleddev_test /dev/gpio_leds on
```

The IP and path are adjusted according to the actual situation.

The debugging results in the serial port tool are as follows:

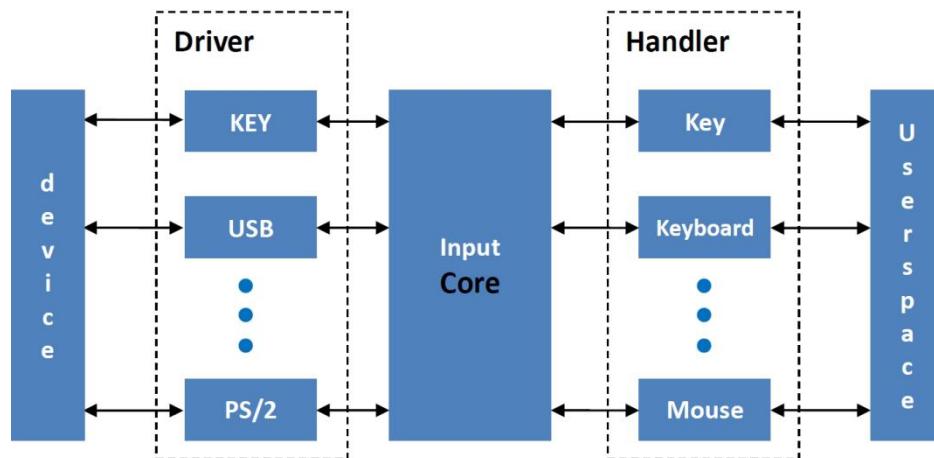
```
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-misc-drv.ko
ax misc_drv: loading out-of-tree module taints kernel.
[drm] load() is defered & will be called again
root@ax_peta:/mnt# cd ./build-axleddev_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-axleddev_test-ZYNQ-Debug# ./axleddev_test /dev/gpio_leds on
ps_led1 on
```

## Part 15: Input Subsystem

Linux supports many input devices, such as keyboards, mice, touch screens, or simple keystrokes. The principles of these input devices are different. In order to manage these diverse input devices, the Linux kernel provides an input subsystem framework.

### Part 15.1: Introduction to the Input Subsystem

Compared with the previously learned **platform** framework **misc** framework, the complexity of the input subsystem framework is slightly higher. It is divided into **the event layer**, **the core layer**, **the device driver layer**, **the upper, middle and lower layers**, as follows:



It can be seen that multiple different **input** devices can be registered in the input subsystem.

- 1) Driver corresponds to the lower-level driver and interfaces with various input devices.
- 2) Handler corresponds to the upper-level event and is connected to the user space through the device node file. Different handler device node file names are also different.
- 3) The Core core layer is responsible for the coordination of the upper and lower layers, and realizes the data interaction between the

upper and lower layers. Events in the lower layer will activate the entire system, and the core will upload the events to the upper layer and finally pass it to the user space.

The input subsystem solves the problem of data interaction between different input devices and application layer, and provides a lot of convenience for input device driver development.

#### Part 15.1.1: The Use of Input Subsystem to Drive the Framework

The core of the input subsystem is implemented in the file “[drivers/input/input.c](#)”. We find the “[input\\_init\(\)](#)” function of **Line 2416**.

```
2416. ....
2417. static int __init input_init(void)
2418. {
2419.     int err;
2420.
2421.     err = class_register(&input_class);
2422.     if (err) {
2423.         pr_err("unable to register input_dev class\n");
2424.         return err;
2425.     }
2426.
2427.     err = input_proc_init();
2428.     if (err)
2429.         goto fail1;
2430.
2431.     err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
2432.                                 INPUT_MAX_CHAR_DEVICES, "input");
2433.     if (err) {
2434.         pr_err("unable to register char major %d", INPUT_MAJOR);
2435.         goto fail2;
2436.     }
2437.
2438.     return 0;
2439.
2440. fail2: input_proc_exit();
2441. fail1: class_unregister(&input_class);
2442.     return err;
2443. }
2444. ....
2445. ....
```

**Line 2452** uses [subsys\\_initcall\(input\\_init\)](#) to mark the [input\\_init](#) function. [Subsys\\_initcall](#) acts on [module\\_init](#) similarly to

mark the driver entry function. The difference is that `subsys_initcall` marks a statically loaded driver. In other words, the `input_init` function is a function that will be executed after the system is started, so let's see what work is done in the `input_init` function.

**Line 2420** first registered `input_class`.

**Line 2430** uses the major device number `INPUT_MAJOR` to register the `input` device. `INPUT_MAJOR` is defined in the file “`include/uapi/linux/major.h`” with a value of 13.

In other words, we use the `input` subsystem to load the input device, no need to register the character device, and we need to register `input_dev` with the `input` subsystem.

Next, we will introduce what `input_dev` is and how to use the `input` subsystem.

## 1) Apply for `input_dev` structure variable

To use the `input` subsystem, you need to register an input device. The `input_dev` structure is used to represent the input device and is defined in the file “`include/linux/input.h`” as follows:

```
121. struct input_dev {  
122.     const char *name;  
123.     const char *phys;  
124.     const char *uniq;  
125.     struct input_id id;  
126.  
127.     unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];  
128.  
129.     unsigned long evbit[BITS_TO_LONGS(EV_CNT)];  
130.     unsigned long keybit[BITS_TO_LONGS(KEY_CNT)];  
131.     unsigned long relbit[BITS_TO_LONGS(REL_CNT)];  
132.     unsigned long absbit[BITS_TO_LONGS(ABS_CNT)];  
133.     unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)];  
134.     unsigned long ledbit[BITS_TO_LONGS(LED_CNT)];  
135.     unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];  
136.     unsigned long ffbit[BITS_TO_LONGS(FF_CNT)];  
137.     unsigned long swbit[BITS_TO_LONGS(SW_CNT)];  
138.  
139. ....  
140.  
141.     bool devres_managed;  
142.};
```

Lines 129~137 are bitmaps of some events, which are used to transfer events. The optional event types are defined in the file “[include/uapi/linux/input-event-codes.h](#)”. The first bitmap “**evbit**” is used to set which of the following events are enabled. For example, if we need to use “**keybit**”, we need to define the corresponding macro in the evbit as “**EV\_KEY**”.

To apply for the **input\_dev** structure variable, use the following function:

```
struct input_dev *input_allocate_device(void)
```

To release the **input\_dev** structure variable use the following function

```
void input_free_device(struct input_dev *dev)
```

## 2) Initialize the **input\_dev** structure variable

After successfully applying for **input\_dev**, first initialize some member variables, such as the **name** field. The main thing that needs to be initialized is the event bitmap. The methods of setting the event bitmap are as follows for our experiment:

- a. Assign directly, such as **input\_dev->evbit[0] = BIT\_MASK(EV\_KEY);**
- b. Use the **set\_bit()** method, such as **set\_bit (EV\_KEY, input\_dev->evbit);**

To clear the event use **\_clear\_bit (EV\_KEY, input\_dev->evbit);**

- c. Use the **input\_set\_capability** method, such as **input\_set\_capability (input\_dev, EV\_KEY, KEY\_Q).** This function is actually the **\_set\_bit** method. This sentence is actually equivalent to **set\_bit(KEY\_Q, input\_dev->keybit);** **input\_set\_capability** is equivalent to a further setting of **evbit**, so this function cannot set **evbit**.

### 3) Register `input_dev` structure variable

After the initialization is complete, you can register the `input_dev` structure variable with the system, and use the function:

```
int input_register_device(struct input_dev *dev)
```

Return 0 to register successfully, and return a negative value to register failed.

The relative unregister uses the following function:

```
void input_unregister_device(struct input_dev *dev)
```

Note that after the `input_register_device` function is successfully registered, if you need to release the `input_dev` structure, you need to use `input_unregister_device` before calling `input_free_device`.

### 4) Reporting of input events

After setting the `input_dev` structure variable, you can use this variable to report the captured input event to the kernel, using the function:

```
void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int  
value)
```

#### Parameter Description:

**dev:** `input_dev` structure pointer;

**type:** Event type to be reported, optional value is [Event types](#), defined in line 37~48 in file [include/uapi/linux/input-event-codes.h](#).

**code:** Event code, defined in the file [include/uapi/linux/input-event-codes.h](#), and select according to the type. For example, when the type is `EV_KEY`, the code can be `KEY_0`, `KEY_1`, etc.

**value:** The value corresponding to the event. For example, in the key event, a value can be specified to indicate that the case is pressed or released.

The `input_event()` function can report all event types. The Linux

kernel also provides some specific event report functions, which are actually encapsulation of the `input_event()` function.

```
void input_sync(struct input_dev *dev);
```

`dev` is the set `input_dev` structure variable

### Part 15.1.2: Use of input subsystem application

After the driver reports the input event, the application can get the input event. Use the `read()` method to get the input time. However, the input parameters are special, and the structure variable `input_event` needs to be used. `input_event` is defined in the file `include/uapi/linux/input.h`, as follows:

```
struct input_event
{
    struct timeval
    time;
    __u16 type;
    __u16 code;
    __s32 value;
```

`time` represents the time, and the `timeval` structure has two members, representing seconds and subtle.

The `type`, `code`, and `value` correspond to the input parameters of the `input_event()` function mentioned earlier.

### Part 15.3: Experiment

The experiment in this chapter is modified on the basis of Part 8 interrupt button experiment driver code.

#### Part 15.3.1: Schematic

Same as Part 6.1

#### Part 15.3.2: Device tree

Same as Part 6.2

### Part 15.3.3: Driver code

Use **petalinux** to create a new driver named "**ax-input-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-input-drv.c** file:

```
1. #include <linux/kernel.h>
2. #include <linux/module.h>
3. #include <linux/init.h>
4. #include <linux/ide.h>
5. #include <linux/types.h>
6. #include <linux/errno.h>
7. #include <linux/cdev.h>
8. #include <linux/of.h>
9. #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <linux/input.h>
20. #include <asm/uaccess.h>
21.
22. #include <asm/io.h>
23.
24. /* 设备节点名称 */
25. #define INPUT_DEV_NAME "input_key"
26.
27. /* 把驱动代码中会用到的数据打包进设备结构体 */
28. struct alinx_char_dev {
29.     dev_t          devid;           //设备号
30.     struct cdev    cdev;            //字符设备
31.     struct class   *class;          //类
32.     struct device   *device;         //设备
33.     struct device_node *nd;        //设备树的设备节点
34.     spinlock_t      lock;            //自旋锁变量
35.     int             alinx_key_gpio; //gpio 号
36.     unsigned int    irq;              //中断号
37.     struct timer_list timer;        //定时器
38.     struct input_dev *inputdev;     //input_dev 结构体
39.     unsigned char   code;            //input 事件码
40. };
41. /* 声明设备结构体 */
42. static struct alinx_char_dev alinx_char = {
43.     .cdev = {
44.         .owner = THIS_MODULE,
45.     },
46. };
47.
48. /* 中断服务函数 */
49. static irqreturn_t key_handler(int irq, void *dev)
50. {
51.     /* 按键按下或抬起时会进入中断 */
```

```

52.     struct alinx_char_dev *cdev = (struct alinx_char_dev *)dev;
53.
54.     /* 开启 50 毫秒的定时器用作防抖动 */
55.     mod_timer(&cdev->timer, jiffies + msecs_to_jiffies(50));
56.     return IRQ_RETVAL(IRQ_HANDLED);
57. }
58.
59. /* 定时器服务函数 */
60. void timer_function(struct timer_list *timer)
61. {
62.     unsigned long flags;
63.     struct alinx_char_dev *dev = &alinkx_char;
64.     /* value 用于获取按键值 */
65.     unsigned char value;
66.
67.     /* 获取锁 */
68.     spin_lock_irqsave(&dev->lock, flags);
69.
70.     /* 获取按键值 */
71.     value = gpio_get_value(dev->alinkx_key_gpio);
72.
73.     if(value == 0)
74.     {
75.         /* 按键按下，状态置 1 */
76.         input_report_key(dev->inputdev, dev->code, 0);
77.         input_sync(dev->inputdev);
78.     }
79.     else
80.     {
81.         /* 按键抬起 */
82.         input_report_key(dev->inputdev, dev->code, 1);
83.         input_sync(dev->inputdev);
84.     }
85.
86.     /* 释放锁 */
87.     spin_unlock_irqrestore(&dev->lock, flags);
88. }
89.
90. /* 模块加载时会调用的函数 */
91. static int init char_drv_init(void)
92. {
93.     /* 用于接受返回值 */
94.     u32 ret = 0;
95.
96.     /* 初始化自旋锁 */
97.     spin_lock_init(&alinkx_char.lock);
98.
99.     /* 获取设备节点 */
100.    alinx_char.nd = of_find_node_by_path("/alinkxkey");
101.    if(alinx_char.nd == NULL)
102.    {
103.        printk("alinkx_char node not find\r\n");
104.        return -EINVAL;
105.    }
106.    else
107.    {
108.        printk("alinkx_char node find\r\n");
109.    }
110.
111.    /* 获取节点中 gpio 标号 */
112.    alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinkxkey-gpi
os", 0);
113.    if(alinx_char.alinx_key_gpio < 0)
114.    {

```

```
115.         printk("can not get alinxkey-gpios");
116.         return -EINVAL;
117.     }
118.     printk("alinkkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
119.
120.     /* 申请 gpio 标号对应的引脚 */
121.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinkkey");
122.     if(ret != 0)
123.     {
124.         printk("can not request gpio\r\n");
125.         return -EINVAL;
126.     }
127.
128.     /* 把这个 io 设置为输入 */
129.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
130.     if(ret < 0)
131.     {
132.         printk("can not set gpio\r\n");
133.         return -EINVAL;
134.     }
135.
136.     /* 获取中断号 */
137.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
138.     /* 申请中断 */
139.     ret = request_irq(alinx_char.irq,
140.                         key_handler,
141.                         IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
142.                         "alinkkey",
143.                         &alinkx_char);
144.     if(ret < 0)
145.     {
146.         printk("irq %d request failed\r\n", alinx_char.irq);
147.         return -EFAULT;
148.     }
149.
150.     timer_setup(&alinkx_char.timer, timer_function, 0);
151.
152.
153.     /* 设置事件码为 KEY_0 */
154.     alinx_char.code = KEY_0;
155.
156.     /* 申请 input_dev 结构体变量 */
157.     alinx_char.inputdev = input_allocate_device();
158.
159.     alinx_char.inputdev->name = INPUT_DEV_NAME;
160.     /* 设置按键事件 */
161.     _set_bit(EV_KEY, alinx_char.inputdev->evbit);
162.     /* 设置按键重复事件 */
163.     _set_bit(EV_REP, alinx_char.inputdev->evbit);
164.     /* 设置按键事件码 */
165.     _set_bit(KEY_0, alinx_char.inputdev->keybit);
166.
167.     /* 注册 input_dev 结构体变量 */
168.     ret = input_register_device(alinx_char.inputdev);
169.     if(ret) {
170.         printk("register input device failed\r\n");
171.         return ret;
172.     }
173.
174.     return 0;
175. }
176.
177. /* 卸载模块 */
178. static void exit char_drv_exit(void)
179. {
```

```
180. /* 删除定时器 */
181. del_timer_sync(&alinx_char.timer);
182. /* 释放中断号 */
183. free_irq(alinx_char.irq, &alinx_char);
184. /* 注销 input_dev 结构体变量 */
185. input_unregister_device(alinx_char.inputdev);
186. /* 释放 input_dev 结构体变量 */
187. input_free_device(alinx_char.inputdev);
188. }
189.
190. /* 标记加载、卸载函数 */
191. module_init(char_drv_init);
192. module_exit(char_drv_exit);
193.
194. /* 驱动描述信息 */
195. MODULE_AUTHOR("Alinx");
196. MODULE_ALIAS("alinx char");
197. MODULE_DESCRIPTION("INPUT LED driver");
198. MODULE_VERSION("v1.0");
199. MODULE_LICENSE("GPL");
```

Focus on the **bold** part.

**Line 19** contains the `input.h` header file.

**Line 38** adds `struct input_dev` type pointer member variable to the device structure.

**Line 39**, add an event code

**Line 76**, in the return function of timer debounce, if the key is pressed, use the `input_report_key()` function to report the event. The `input_report_key()` function is defined in the file `include/linux/input.h`. The content is very simple, that is, `input_event(dev, EV_KEY, code, !value)` is called; the function is specifically used to report the time of the key press. The first parameter of `input_report_key()` is a pointer to the `input_dev` structure type, which is the input device we applied for in the driver. The second parameter is the event code, and the third parameter is the value corresponding to the event.

**Line 77**, call `input_sync(dev->inputdev)` to submit the report.

**Lines 82~83**, similarly report the key release event, here we use the third parameter value to distinguish between press and release, **1** means press, **0** means release.

In the driver entry function, before **Line 153**, the information in the

device tree is obtained as in **Part 8**, and the interrupt and timer are set.

In **Line 154**, the event code is assigned to **KEY\_0**, and then when we write the application, we need to correspond to the value of **KEY\_0**.

In **Line 157**, use the **input\_allocate\_device()** function to apply for structure variables.

**Lines 159~165**, initialize the **input\_dev** structure variable. Assign a value to name first.

**Line 161**, call **\_set\_bit** to set the key event in **evbit** to **1**.

**Line 163**, set repeated keys in the same way.

**Line 165**, set the event code of the key event to **KEY\_0**.

In the driver exit function, **Line 185** calls **input\_unregister\_device** to unregister **input\_dev**, and **line 187** calls **input\_free\_device** to release the space of the **input\_dev** structure variable.

In the whole process, we did not implement the **file\_operations** device operation function set, but used the operation functions in the **input** subsystem. There is no further registration of character devices, so using the **input** subsystem, it is equivalent to registering the device with the **input** subsystem instead of registering with the kernel. The object of our direct communication is the **input** subsystem instead of the kernel. The interaction with the kernel is given to the input subsystem. This is different from the **misc** device framework mentioned earlier.

#### Part 15.3.4: Test code

Create a new QT project named “**ax\_inputkey\_test**”, create a new **main.c**, and enter the following code:

```
1. #include "stdio.h"
2. #include "unistd.h"
3. #include <fcntl.h>
4. #include <linux/input.h>
5. /*定义一个input_event 结构体变量获取事件*/
6. static struct input_event inputhead;
7.
8. /* 点亮火熄灭led */
9. int led_change_sts()
10. {
11.     int fd, ret;
12.     static char led_value = 0;
13.
14.     fd = open("/dev/gpio_leds", O_RDWR);
15.     if(fd < 0)
16.     {
17.         printf("file /dev/gpio_leds open failed\r\n");
18.     }
19.
20.     led_value = !led_value;
21.     ret = write(fd, &led_value, sizeof(led_value));
22.
23.     if(ret < 0)
24.     {
25.         printf("write failed\r\n");
26.     }
27.
28.     ret = close(fd);
29.     if(ret < 0)
30.     {
31.         printf("file /dev/gpio_leds close failed\r\n");
32.     }
33.
34.     return ret;
35. }
36.
37. int main(int argc, char *argv[])
38. {
39.     int fd;
40.     int err = 0;
41.     char *filename;
42.
43.     filename = argv[1];
44.
45.     /* 验证输入参数个数 */
46.     if(argc != 2) {
47.         printf("Error Usage\r\n");
48.         return -1;
49.     }
50.
51.     /* 打开输入的设备文件，获取文件句柄 */
52.     fd = open(filename, O_RDWR);
53.     if(fd < 0) {
54.         /* 打开文件失败 */
55.         printf("can not open file %s\r\n", filename);
56.         return -1;
57.     }
58.
59.     while(1)
60.     {
61.         err = read(fd, &inputhead, sizeof(inputhead));
62.         if(err > 0)
63.         {
64.             switch(inputhead.type)
65.             {
66.                 case EV_KEY:
67.                     if(KEY_0 == inputhead.code)
68.                     {
69.                         if(0 == inputhead.value)
70.                         {
71.                             /* 按键抬起 */
72.                             err = led_change_sts();
```

```
73.         }
74.     }
75.     {
76.         /* 按键按下 */
77.     }
78. }
79. else
80. {
81.     /* ignore */
82. }
83. break;
84.
85. default :
86.     /* ignore */
87.     break;
88. }
89.
90. if(err < 0)
91. {
92.     printf("led open failed");
93. }
94. else
95. {
96.     printf("get data failed\r\n");
97. }
98. }
99. }
100.
101. return 0;
102.}
```

**Line 4** contains the header file “[input.h](#)”.

**Line 6**, define the [input\\_event](#) structure variable to get the event.

**Line 8~35**, take out the code to turn on or off the led, so that you can see the main function more clearly.

**Line 61** of the main function, the read function is called to receive the event. The second parameter here is the [input\\_event](#) structure variable.

**Line 64**, first determine the event type. If it is of [EV\\_KEY](#) type, enter the [case](#) on Line 66, and ignore other types.

**Line 67**, then judge whether the event code is [KEY\\_0](#), if not, ignore it, if yes, continue to judge.

**Line 69**, judge the value corresponding to the event. In the driver code, 0 means the key is released, and 1 means it is pressed. When the key is released, the end of a key, so after the key is released, the state of the led is switched once.

#### Part 15.3.5: Run Test

The device node file generated under the input subsystem framework is under the path “[/dev/input](#)”. When using `insmod` to load the input device, the system will prompt "input: input\_key as [/devices/virtual/input/input0](#)". Generally, the device file will correspond to `input0` here, which is “[/dev/input/event0](#)”.

The steps are as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/ilinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ax-input-drv.ko
insmod ax-concled-drv.ko
cd ./build-ax_inputkey_test-IDE_5_7_1_GCC_64bit-Debug/
./ax_inputkey_test/dev/input/event0
```

The IP and path are adjusted according to the actual situation.

The debugging results in the serial port tool are as follows:

```
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.107:/home/ilinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
random: fast init done
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ax-input-drv.ko
ax_input_drv: loading out-of-tree module taints kernel.
alinx_char node find
alinxkey-gpio num = 949
input: input_key as /devices/virtual/input/input0
root@ax_peta:/mnt# insmod ax-concled-drv.ko
alinx_char node find
alinxled-gpio num = 899
root@ax_peta:/mnt# cd ./build-ax_inputkey_test-ZYNQ-Debug
root@ax_peta:/mnt/build-ax_inputkey_test-ZYNQ-Debug# ./ax_inputkey_test /dev/input/event0
```

You can look at the `input` subsystem, the CPU usage of the keys is almost zero.

```
root@ax_peta:/mnt/build-ax_inputkey_test-ZYNQ-Debug# ./ax_inputkey_test /dev/input/event0&
[1] 1273
root@ax_peta:/mnt/build-ax_inputkey_test-ZYNQ-Debug# top
Mem: 56700K used, 973552K free, 13496K shrd, 1196K buff, 42908K cached
CPU: 0.0% usr 0.0% sys 0.0% nic 100% idle 0.0% io 0.0% irq 0.0% sirq
Load average: 0.05 0.02 0.00 1/88 1274
 PID  PPID USER      STAT  VSZ %VSZ CPU %CPU COMMAND
1247    1 root      S   14352  1.3   1  0.0 /usr/sbin/tcf-agent -d -L -l0
1273  1254 root      S   13164  1.2   0  0.0 ./ax_inputkey_test /dev/input/event0
1254  1252 root      S   2988  0.2   1  0.0 -sh
1174    1 root      S   2916  0.2   1  0.0 /usr/sbin/inetd
  761    1 root      S   2868  0.2   0  0.0 /sbin/udevd -d
1233    1 root      S   2788  0.2   1  0.0 /sbin/syslogd -n -O /var/log/messages
1226    1 root      S   2720  0.2   0  0.0 /sbin/klogd -n
```

## Part 16: pwm Drive

pwm is used a lot in embedded devices, often used to control motors, control vibration devices, adjust backlights, breathing lights, and so on. This chapter learns how to implement pwm driver in Linux system on zynq platform. The Linux system we used before was built completely according to the steps here. In the process of configuring hardware information in vivado, the pwm output was not set, so it needs to be modified on this basis.

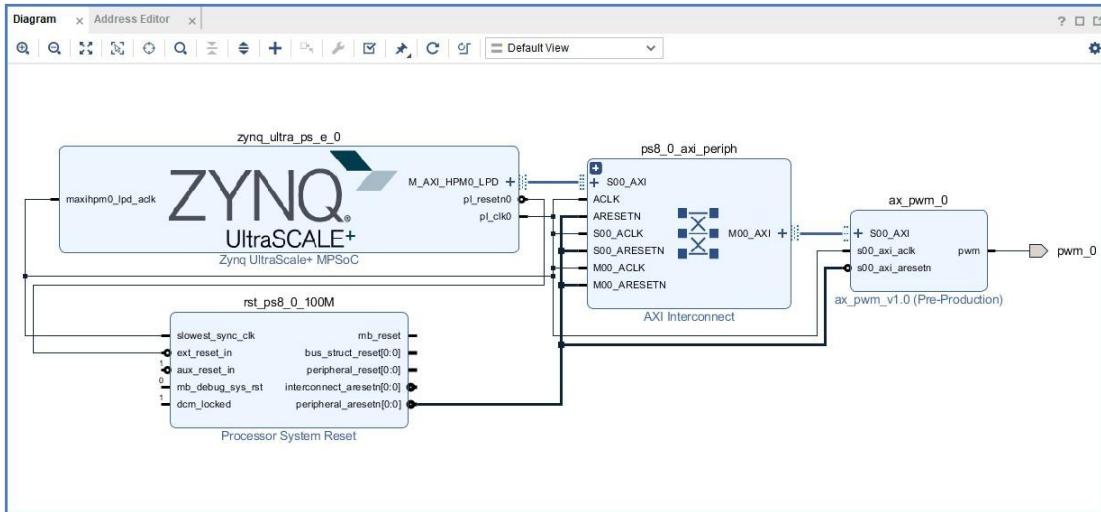
### Part 16.1: pwm Implementation on zynq

#### Part 16.1.1: Modify the Vivado Project

The output of pwm on the zynqmp platform requires the resources of the pl side (fpga). In the hardware description file used earlier, pwm output is not set, so it needs to be modified on the basis of this vivado project. The vivado project refers to 01\_ps\_base.zip in the source code package.

The modification method refers to the vivado custom IP method learned earlier. Here we take pwm as an example to introduce how to make a custom IP. What we need to do here is to add the pwm custom ip made previous on the basis of the vivado project.

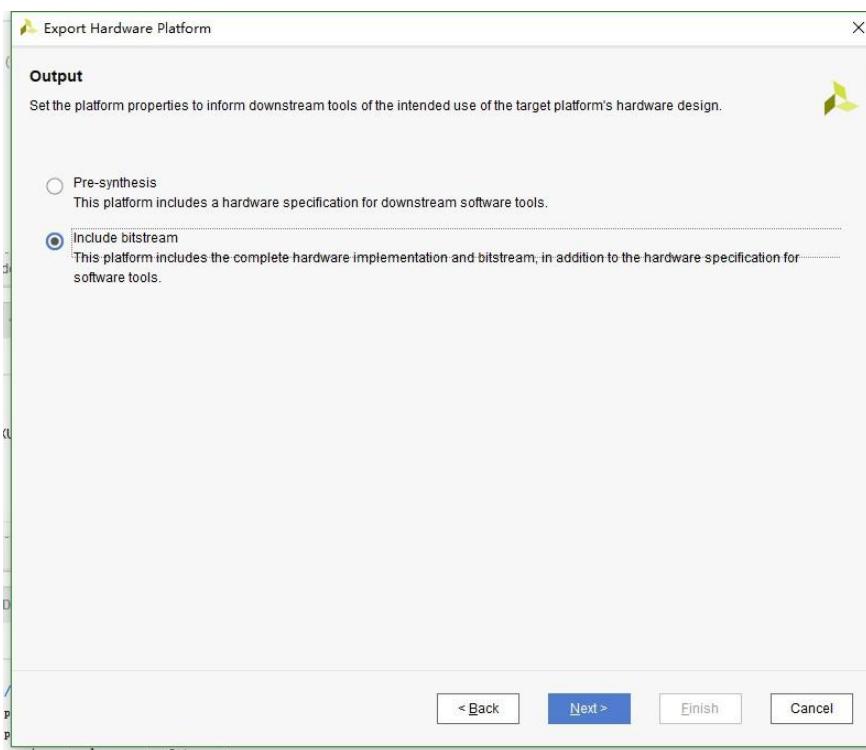
The whole process will not be explained in detail here. After adding the **ip** of **pwm**, the result is as follows:



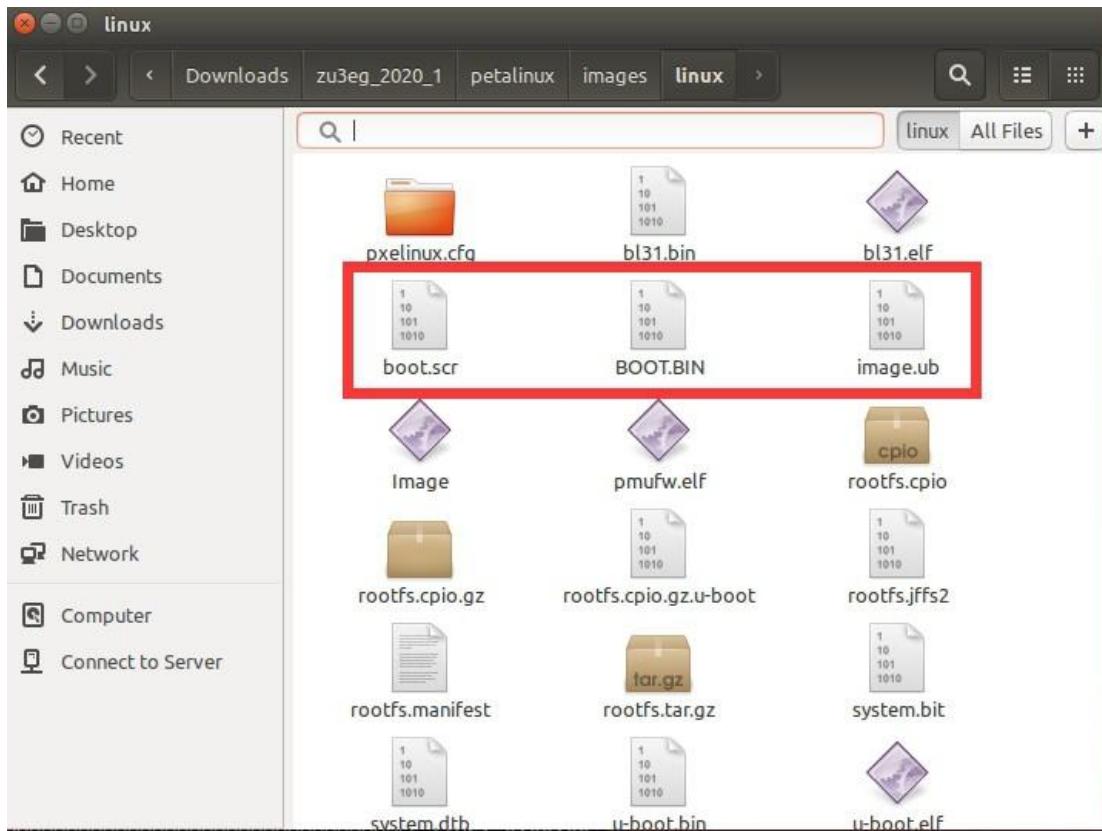
After adding **ip**, you need to modify **xdc** to change the pin constraints. Here we are going to use **pwm** to control the **PL\_LED** on the FPGA development board. Create a new **pwm.xdc** file and add the following content to constrain the pins.

```
set_property IOSTANDARD LVCMOS33 [get_ports pwm_0]
set_property PACKAGE_PIN AE12 [get_ports pwm_0]
```

After the modification is completed, compile and generate bitstream, and export the hardware description file. Note that when exporting, the bitstream must be included.

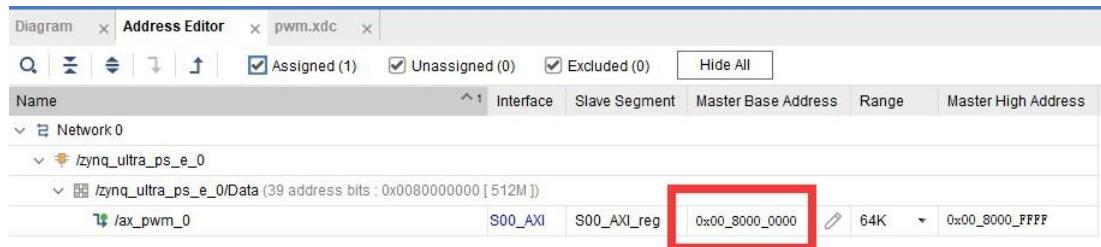
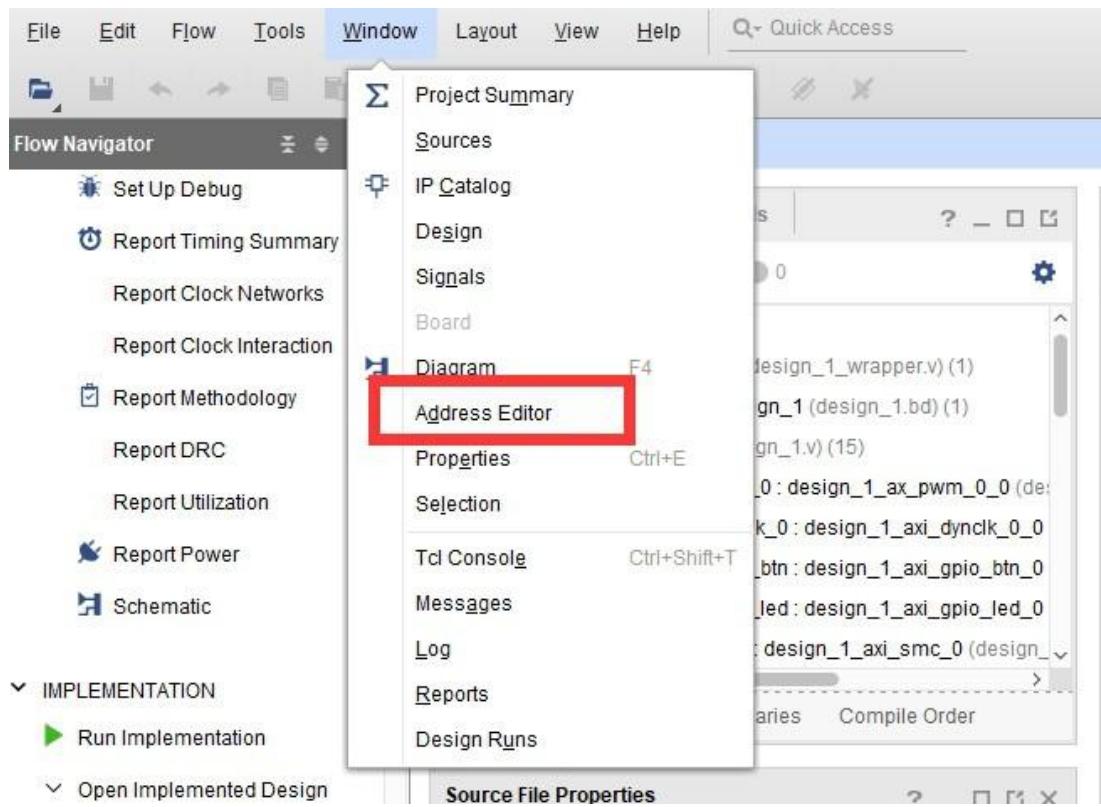


Then use the newly obtained .xsa file to reconfigure the kernel, repackage BOOT.BIN after compilation, and replace the original boot.scr, BOOT.BIN, and image.nb files in the SD card



### Part 16.1.2: Method of Controlling pwm Output

There are two key parameters to control the **pwm** output, one is the frequency, and the other is the duty cycle. After we configure the hardware information above, how to operate these two parameters of **pwm**. Custom **IP** can be understood as configuring the register resource of fpga to become the device we want. After adding the IP, we can operate the **fpga** resource through the operator register just like operating the **ARM** resource. First of all, we must find the first address of the register of the **pwm** device. Open the **Address Editor** in the **Window** option of the menu bar. As shown below:



You can find the register range corresponding to [ip](#). The first address of this register is automatically allocated by the [vivado](#) project, so it may be different each time. Of course, you can also modify it manually.

Only found the first address of the register, but still don't know the specific setting method of the register. here we refer to the following code, as shown in the figure below:

```
unsigned int duty;
int main()
{
    init_platform();
    print("Hello World\n\r");
    //pwm out period = frequency(pwm_out) * (2 ** N) / frequency(clk);
    AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR, AX_PWM_S00_AXI_SLV_REG0_OFFSET, 17179); //200hz
    while (1)
    {
        for (duty = 0x00000000; duty < 0xffffffff; duty = duty + 100000) {
            AX_PWM_mWriteReg(XPAR_AX_PWM_0_S00_AXI_BASEADDR, AX_PWM_S00_AXI_SLV_REG1_OFFSET, duty);
            usleep(100);
        }
    }
    cleanup_platform();
    return 0;
}
```

The **AX\_PWM\_mWriteReg** function is used to set the value of the register. The first parameter is the first address, the second parameter is the offset, and the third parameter is the target value.

The first call of the **AX\_PWM\_mWriteReg** function is to set the frequency. The value of **XPAR\_AX\_PWM\_0\_S00\_AXI\_BASEADDR** is the first address we found in the **Address Editor** interface above. Look at the offset value of **AX\_PWM\_S00\_AXI\_SLV\_REG0\_OFFSET** as **0**. In other words, the register for setting the frequency is the first address.

The second call is to set the duty cycle, the first address has not changed, and the offset value of **AX\_PWM\_S00\_AXI\_SLV\_REG1\_OFFSET** is 4, that is, the first address plus 4.

## Part 16.2: Pwm framework in Linux

The complete **pwm** framework is provided in the Linux kernel. However, you need to pay attention to using this framework on zynq. The pwm on zynq is realized with the help of FPGA resources. The register settings of custom ip may vary from person to person. Therefore, to use this framework, **you need to ensure that the custom ip is consistent**. Furthermore, if it is another simple **pwm** application, such as the led breathing light in the experiment in this

chapter, using this frame is a bit overkill. Although this framework will not be used in the experiments in this chapter, we still briefly introduce it to understand the structure and design ideas.

The `pwm` framework is still integrated into the driver's layered thinking. The `pwm` driver is divided into two parts, `pwm_device (device)` and `pwm_chip (chip)`. `pwm_chip` means `pwm chip`, which can output single or multiple `pwm` waveforms, and `pwm_device` is In order to use one or more of the devices of `pwm_chip`, take the led breathing light in this chapter as an example, led is `pwm_device`, and our customized `pwmip` is `pwm_chip`. The `pwm` framework is generally used in conjunction with the `platform` framework. Next we introduce their implementation methods respectively

### Part 16.2.1:pwm\_device

The device using `pwm` needs to pay attention to four parameters: enable, polarity, frequency, and duty cycle. The `pwm` framework provides the following methods to correspond to it:

- 1) `int pwm_enable(struct pwm_device *pwm)`, use `pwm_enable` function to enable `pwm` output
- 2) `void pwm_disable(struct pwm_device * pwm);` `pwm_disable` is used to disable `pwm` output.
- 3) `int pwm_set_polarity(struct pwm_device *pwm, enum pwm_polarity polarity);` `pwm_set_polarity` is used to set the polarity. The optional parameters are `PWM_POLARITY_NORMAL` (normal) and `PWM_POLARITY_INVERSED` (inverted).
- 4) `int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns);` The `pwm_config` function is used to set the duty cycle and frequency. The parameter `duty_ns` is the duty cycle and `period_ns` is the frequency.

These interface functions are defined in the file `include/linux/pwm.h`, and they all need a `struct pwm_device` pointer as an operation handle. The `pwm_device` structure refers to the device that uses `pwm`. If we want to use these interfaces, we need to obtain a `pwm_device` structure first.

The method of obtaining the `pwm_device` structure is also defined in the file `include/linux/pwm.h`. as follows:

- 1) `struct pwm_device *pwm_get(struct device *dev, const char *con_id);` and `struct pwm_device *of_pwm_get(struct device_node *np, const char *con_id);`

These two methods will obtain `pwm_device` from the device tree node of the specified device (parameter `dev`) or directly from the device tree node (parameter `np`). The format requirements for the device tree are as follows:

```
bl: backlight {  
    pwms = <&pwm 0 500000 PWM_POLARITY_INVERTED>;  
    pwm-names = "backlight";  
}
```

**bl: backlight** is the device node name. Both attributes in the node are required

**pwms** refers to the list of `pwm` used by this device. The first parameter `&pwm` actually refers to the `pwm_chip` node, which will be discussed when introducing `pwm_chip`. The second parameter refers to the device number of `platform_device`, which is determined according to the actual situation and may not be used. The third parameter 500000 is the default period of `pwm`, and the unit is nanoseconds. The fourth parameter is an optional field, indicating polarity.

**pwm-names** is the `pwm` device name corresponding to `pwms`.

"**backlight**" corresponds to `&pwm 0 500 0000`

---

## PWM\_POLARITY\_INVERTED.

The function parameter `con_id` is used to specify the matching `pwm` device. If `NULL`, it will return the first device in `pwms`. If you specify a name such as "backlight", it will return the corresponding device.

- 2) `void pwm_put(struct pwm_device *pwm);`

Put opposite to get function is used to release `pwm_device`.

These are the things `pwm_device` need to pay attention to. After the `pwm` device calls the interface function, it will eventually call the operation function in `pwm_chip`. Next, let's look at `pwm_chip`.

### Part 16.2.2:

The key function of `pwm_chip` is that it provides the operation function set corresponding to the `pwm_device` interface function. The `pwm_chip` structure is defined in `include/linux/pwm.h`, as follows:

```
struct pwm_chip {
    struct device          *dev;
    struct list_head        list;
    const struct pwm_ops    *ops;
    int                     base;
    unsigned int            npwm;
    struct pwm_device      *pwms;
    struct pwm_device *    (*of_xlate)(struct pwm_chip *pc, const struct
of_phandle_args *args);
    unsigned int            of_pwm_n_cells;
    bool                   can_sleep;
};
```

`dev` is the device corresponding to `pwm chip`, which is generally specified by the platform driver corresponding to `pwm driver` and must be provided.

`ops` is the set of operation functions corresponding to the interface functions in `pwm_device` and must be provided.

**npwm** is the number of **pwm\_device** that pwm chip can support and must be provided.

**pwms** is the array of **pwm device** in the pwm chip, the kernel will allocate it by itself, and no manual setting is required.

Focus on the **struct pwm\_ops** structure to correct **ops**.

**struct pwm\_ops** is defined as follows:

```
struct pwm_ops {
    int (*request)(struct pwm_chip *chip, struct pwm_device *pwm);
    void (*free)(struct pwm_chip *chip, struct pwm_device *pwm);
    int (*config)(struct pwm_chip *chip, struct pwm_device *pwm, int duty_ns, int
    period_ns);
    int (*set_polarity)(struct pwm_chip *chip, struct pwm_device *pwm, enum
    pwm_polarity polarity);
    int (*enable)(struct pwm_chip *chip, struct pwm_device *pwm);
    void (*disable)(struct pwm_chip *chip, struct pwm_device *pwm);
#ifndef CONFIG_DEBUG_FS
    void (*dbg_show)(struct pwm_chip *chip, struct seq_file *s);
#endif
    struct module *owner;
};
```

The **request** and **free** functions have been deprecated.

The function corresponds to the **pwm\_config** function in the **pwm\_device** interface function, which configures the frequency and duty cycle of **pwm\_device**. Must provide.

The **enable** and **disable** functions correspond to the **pwm\_enable** and **pwm\_disable** functions in the **pwm\_device** interface function, and they are used to enable/disable **pwm** signal output. Must provide.

The **set\_polarity** function corresponds to the **pwm\_set\_polarity** function in the **pwm\_device** interface function to set the polarity of the **pwm** signal. Optional.

After defining a **pwm\_chip**, you need to implement at least three functions in **ops: config, enable, and disable**. After the initialization is complete, use the following function to register **pwm\_chip** with the kernel:

```
int pwmchip_add(struct pwm_chip *chip);
```

Relative unregister use function

```
int pwmchip_remove(struct pwm_chip *chip);
```

When implementing functions such as **config**, device information needs to be obtained through the device tree. The node format of **pwm\_chip** in the device tree is not fixed, just add relevant information. as follows:

```
pwm: pwm@43C20000
{
    compatible = "test-pwm";
    reg = <0x43C20000 0x100>;
};
```

In the node of **pwm\_device** above, there is a reference to **pwm\_chip "pwms = <&pwm 0 5000000 PWM\_PO LARITY\_INVERTED>"**. The **&pwm** here actually refers to the **pwm\_chip0** node such as "**pwm: pwm@43C20000**". The purpose of quoting is to associate **device** with **chip**.

The dedicated **pwm** chip manufacturer will provide the **pwm\_chip** driver, but xilinx bin did not provide it, maybe considering the diversity of custom ip. Therefore, to use the **pwm** framework on the zynq platform, it is also need to drive developers to implement **pwm\_chip**, quite difficult.

### Part 16.2.3: Example

**Device tree:**

```
1.  pwm: pwm@43C20000 {
2.      compatible = "alinx-pwm";
3.      reg = <0x43C20000 1>;
4.      #pwm-cells = <2>;
5.  };
```

```
6.
7.     pwm-led {
8.         compatible = "pwm-led";
9.         pwms = <&pwm 0 5000000>;
10.    };
```

### pwm\_chip:

Combining with the **platform** framework, get information from the device tree and register **pwm\_chip**.

```
1.  struct ax_pwm_chip {
2.      struct pwm_chip chip;
3.  };
4.
5.  struct ax_pwm_chip ax_pwm;
6.
7. static int ax_pwm_config(struct pwm_chip *chip, struct pwm_device *pwm, int duty_ns, int period_ns)
8. {
9.     return 0;
10. }
11.
12. static int ax_pwm_enable(struct pwm_chip *chip, struct pwm_device *pwm)
13. {
14.     return 0;
15. }
16.
17. static void ax_pwm_disable(struct pwm_chip *chip, struct pwm_device *pwm)
18. {
19.
20. }
21.
22. static const struct pwm_ops ax_pwm_ops = {
23.     .owner = THIS_MODULE,
24.     .config = ax_pwm_config,
25.     .enable = ax_pwm_enable,
26.     .disable = ax_pwm_disable,
27. };
28.
29. static int ax_pwm_probe(struct platform_device *pdev)
30. {
31.     int err;
32.
33.     ax_pwm.chip.dev = &pdev->dev;
34.     ax_pwm.chip.ops = &ax_pwm_ops;
35.     ax_pwm.chip.npwm = 1;
36.
37.     err = pwmchip_add(&ax_pwm.chip);
38.     if (err < 0)
39.     {
40.         return err;
41.     }
42.
43.     return 0;
44. }
45.
46. static int ax_pwm_remove(struct platform_device *pdev)
47. {
48.     int err;
49.
50.     err = pwmchip_remove(&ax_pwm.chip);
51.     if (err < 0)
52.     {
53.         return err;
54.     }
55.
56.     return 0;
57. }
58.
59. static const struct of_device_id of_ax_pwm_match[] = {
60.     { .compatible = "alinx-pwm", },
61.     { /* Sentinel */ },
62. };
63.
64. static struct platform_driver ax_pwm_driver = {
65.     .driver = {
66.         .name = "alink-pwm",
67.         .of_match_table = of_ax_pwm_match,
```

```
68.     },
69.     .probe = ax_pwm_probe,
70.     .remove = ax_pwm_remove,
71. };
72.
73. module_platform_driver(ax_pwm_driver);
```

### pwm\_device:

Combined with the **misc** device framework, register **pwm\_device**.

```
1. #define PWM_ON 0x100001
2. #define PWM_OFF 0x100002
3.
4. struct pwm_device *ax_pwm_dev;
5.
6. static long ax_pwm_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
7. {
8.     int ret;
9.     switch(cmd) {
10.         case PWM_ON:
11.             ret = pwm_config(ax_pwm_dev, 200000, 500000);
12.             pwm_enable(ax_pwm_dev);
13.             break;
14.
15.         case PWM_OFF:
16.             ret = pwm_config(ax_pwm_dev, 0, 500000);
17.             pwm_disable(ax_pwm_dev);
18.             break;
19.     }
20.     return 0;
21. }
22.
23. static struct file_operations ax_pwm_fops = {
24.     .owner = THIS_MODULE,
25.     .unlocked_ioctl = ax_pwm_ioctl,
26. };
27.
28. static struct miscdevice pwm_misc = {
29.     .minor = MISC_DYNAMIC_MINOR,
30.     .name = "ax-pwm",
31.     .fops = &ax_pwm_fops
32. };
33. static int ax_pwm_init(void)
34. {
35.     int ret;
36.     struct device_node *nd;
37.
38.     nd = of_find_node_by_path("/pwm-led");
39.
40.     ax_pwm_dev = of_pwm_get(nd, "pwm-led");
41.
42.     misc_register(&pwm_misc);
43.     return 0;
44. }
45.
46. static void ax_pwm_exit(void)
47. {
48.     pwm_put(ax_pwm_dev);
49. }
50. module_init(ax_pwm_init);
51. module_exit(ax_pwm_exit);
```

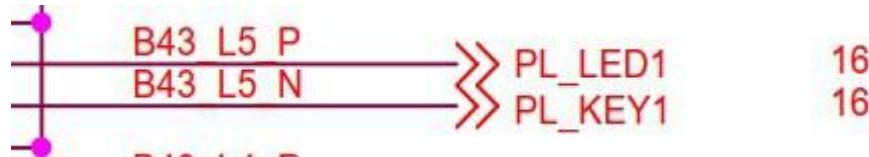
---

## Part 16.3: Experiment

The experiment in this chapter combines platform and misc framework to realize a simple led breathing light.

### Part 16.3.1: Schematic

In the carrier board schematic, find the connector pin that connects to PL\_LED.



In the core board schematic diagram, the pin corresponding to pl is found through the connector pin, which is AE12, which is the pin bound in the xdc file when we modify the vivado project above.



### Part 16.3.2: Device tree

Open the **system-user.dtsi** file and add the following nodes to the root node:

```

1. alinxpwm {
2.     compatible = "alinx-pwm";
3.     reg-freq = <0x8000000001>;
4.     reg-duty = <0x8000000041>;
5. };

```

The compatible node “compatibility” is "**alink-pwm**", and the "of" matching table of platform in the subsequent driver code needs to be consistent with this attribute.

The physical addresses of the frequency and duty cycle of “**reg-freq**” and “**reg-duty**” are still obtained from “vivado”.

### Part 16.3.3: Driver code

Use **petalinux** to create a new driver named "**ax-pwm**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-pwm.c** file:

```
1. #include <linux/types.h>
2. #include <linux/kernel.h>
3. #include <linux/delay.h>
4. #include <linux/ide.h>
5. #include <linux/init.h>
6. #include <linux/module.h>
7. #include <linux/errno.h>
8. #include <linux/of.h>
9. #include <linux/cdev.h>
10. #include <linux/device.h>
11. #include <linux/semaphore.h>
12. #include <linux/timer.h>
13. #include <linux/irq.h>
14. #include <linux/wait.h>
15. #include <linux/poll.h>
16. #include <linux/fs.h>
17. #include <linux/fcntl.h>
18. #include <linux/platform_device.h>
19. #include <linux/miscdevice.h>
20. #include <asm/uaccess.h>
21.
22. /* 设备节点名称 */
23. #define DEVICE_NAME      "ax_pwm"
24. /* 设备号个数 */
25. #define DEVID_COUNT      1
26. /* 驱动个数 */
27. #define DRIVE_COUNT       1
28. /* 主设备号 */
29. #define MAJOR_AX          0
29. /* 次设备号 */
30. #define MINOR_AX          1
31. #define PWM_FREQ          0x100001
32. /* 设置占空比 */
33. #define PWM_DUTY          0x100002
34.
35. /* 把驱动代码中会用到的数据打包进设备结构体 */
36. struct alinx_char_dev{
37.     dev_t           devid;        //设备号
38.     struct cdev      cdev;        //字符设备
39.     struct device_node *nd;        //设备树的设备节点
40.     unsigned int    *freq;        //频率的寄存器虚拟地址
41.     unsigned int    *duty;        //占空比的寄存器虚拟地址
42. };
43.
44. /* 声明设备结构体 */
45. static struct alinx_char_dev alinx_char = {
46.     .cdev = {
47.         .owner = THIS_MODULE, 49.
48.     },
49. };
50. };
51.
52. /* open 函数实现，对应到 Linux 系统调用函数的open 函数 */
53. static int ax_pwm_open(struct inode *inode_p, struct file *file_p)
54. {
55.     /* 设置私有数据 */
56.     file_p->private_data = &alinkx_char;
57.
58.     return 0;
59. }
60.
61. /* ioctl 函数实现，对应到Linux 系统调用函数的 ioctl 函数 */
62. static long ax_pwm_ioctl(struct file *file_p, unsigned int cmd, unsigned long arg)
63. {
64.     /* 获取私有数据 */
65.     struct alinx_char_dev *dev = file_p->private_data;
66.
67.     switch(cmd)
68.     {
```

```
69.     case PWM_FREQ:
70.     {
71.         *(dev->freq) = (unsigned int)arg;
72.     }
73.     break;
74. }
75.
76.     case PWM_DUTY:
77.     {
78.         *(dev->duty) = (unsigned int)arg;
79.     }
80.     break;
81.
82.     default :
83.     {
84.         break;
85.     }
86. }
87.
88.     return 0;
89. }
90.
91. /* release 函数实现，对应到Linux 系统调用函数的close 函数 */
92. static int ax_pwm_release(struct inode *inode_p, struct file *file_p)
93. {
94.     return 0;
95. }
96.
97. /* file_operations 结构体声明 */
98. static struct file_operations ax_char_fops = {
99.     .owner          = THIS_MODULE,
100.    .open           = ax_pwm_open,
101.    .unlocked_ioctl = ax_pwm_ioctl,
102.    .release        = ax_pwm_release,
103. };
104.
105. /* MISC 设备结构体 */
106. static struct miscdevice led_misctdev = {
107.     /* 自动分配次设备号 */
108.     .minor = MISC_DYNAMIC_MINOR,
109.     .name = DEVICE_NAME,
110.     /* file_operations 结构体 */
111.     .fops = &ax_char_fops,
112. };
113.
114. /* probe 函数实现，驱动和设备匹配时会被调用 */
115. static int ax_pwm_probe(struct platform_device *dev)
116. {
117.     /* 用于接受返回值 */
118.     u32 ret = 0;
119.     /* 频率的寄存器物理地址 */
120.     unsigned int freq_addr;
121.     /* 占空比的寄存器物理地址 */
122.     unsigned int duty_addr;
123.
124.     /* 获取设备节点 */
125.     alinx_char.nd = of_find_node_by_path("/alinxpwm");
126.     if(alinx_char.nd == NULL)
127.     {
128.         printk("gpioled node not find\r\n");
129.         return -EINVAL;
130.     }
131.
132.     /* 获取寄存器中 freq 的地址 */
133.     of_property_read_u32(alinx_char.nd, "reg-freq", &freq_addr);
134.     if(!freq_addr)
135.     {
136.         printk("can not get reg-freq\r\n");
137.         return -EINVAL;
```

```
137.     }
138.     else
139.     {
140.         /* 映射地址 */
141.         alinx_char.freq = ioremap_wc(freq_addr, 4);
142.     }
143.
144.     /* 获取寄存器中 duty 的地址 */
145.     of_property_read_u32(alinx_char.nd, "reg-duty", &duty_addr);
146.     if(!duty_addr)
147.     {
148.         printk("can not get reg-duty\r\n");
149.         iounmap((unsigned int *)alink_char.freq);
150.         return -EINVAL;
151.     }
152.     else
153.     {
154.         /* 映射地址 */
155.         alinx_char.duty = ioremap_wc(duty_addr, 4);
156.     }
157.     /* 注册 misc 设备 */
158.     ret = misc_register(&led_misctdev);
159.     if(ret < 0)
160.     {
161.         printk("misc device register failed\r\n");
162.         return -EFAULT;
163.     }
164.     return 0;
165. }
166.
167. static int ax_pwm_remove(struct platform_device *dev)
168. {
169.     /* 释放虚拟地址 */
170.     iounmap((unsigned int *)alink_char.freq);
171.     iounmap((unsigned int *)alink_char.duty);
172.     /* 注销 misc 设备 */
173.     misc_deregister(&led_misctdev);
174.     return 0;
175. }
176.
177. /* 初始化of_match_table */
178. static const struct of_device_id pwm_of_match[] =
179. {
180.     /* compatible 字段和设备树中保持一致 */
181.     { .compatible = "alink-pwm" },
182.     /* Sentinel */
183. };
184.
185. /* 声明并初始化platform 驱动 */
186. static struct platform_driver pwm_driver =
187. {
188.     .driver = {
189.         /* name 字段需要保留 */
190.         .name = "alink-pwm",
191.         /* 用 of_match_table 代替 name 匹配 */
192.         .of_match_table = pwm_of_match,
193.     },
194.     .probe = ax_pwm_probe,
195.     .remove = ax_pwm_remove,
196. };
197. /* 驱动入口函数 */
198. static int_init pwm_drv_init(void)
199. {
200.     /* 在入口函数中调用 platform_driver_register, 注册 platform 驱动 */
201.     return platform_driver_register(&pwm_driver);
202. }
203.
204. /* 驱动出口函数 */
```

```
205. static void exit_pwm_drv_exit(void)
206. {
207.     /* 在出口函数中调用 platform_driver_register, 卸载 platform 驱动 */
208.     platform_driver_unregister(&pwm_driver);
209. }
210.
211. /* 标记加载、卸载函数 */
212. module_init(pwm_drv_init);
213. module_exit(pwm_drv_exit);
214.
215. /* 驱动描述信息 */
216. MODULE_AUTHOR("Alinx");
217. MODULE_ALIAS("pwm_led");
218. MODULE_DESCRIPTION("PWM LED driver");
219. MODULE_VERSION("v1.0");
220. MODULE_LICENSE("GPL");
```

The part of the “**paltform**” and “**misc**” framework can refer to **Part14**, almost unchanged.

The **pwm** processing is also very simple, get the address from the device tree, and then manipulate the address in the operation function.

**Lines 33~36** define two macros, which are used for the “**cmd**” of “**ioctl**”, which respectively represent the setting frequency and the setting duty cycle.

**Line 43**, “**freq**” is used to get the address of the set frequency register from the device tree.

**Line 44**, “**freq**” is used to get the address of the set duty cycle register from the device tree.

#### Part 16.3.4: Test code

Create a new QT project named “**ax\_pwm\_test**”, create a new **main.c**, and enter the following code:

```
1. #include "stdio.h"
2. #include "unistd.h"
3. #include "sys/types.h"
4. #include "sys/stat.h"
5. #include "fcntl.h"
6. #include "stdlib.h"
7. #include "string.h"
8. #include "sys/ioctl.h"
9.
10. #define FREQ_DEFAULT    1717900
11. /* 设置频率 */
12. #define PWM_FREQ        0x100001
13. /* 设置占空比 */
14. #define PWM_DUTY        0x100002
15.
16. int main(int argc, char *argv[])
```

```
17. {
18.     int fd, retval, flag = 0;
19.     char *filename;
20.     unsigned int duty = 0xffffffff;
21.
22.     if(argc != 2)
23.     {
24.         printf("Error Usage\r\n");
25.         return -1;
26.     }
27.
28.     filename = argv[1];
29.     fd = open(filename, O_RDWR);
30.     if(fd < 0)
31.     {
32.         printf("file %s open failed\r\n", argv[1]);
33.         return -1;
34.     }
35.
36.     /* 设置频率 */
37.     retval = ioctl(fd, PWM_FREQ, FREQ_DEFAULT);
38.     if(retval < 0)
39.     {
40.         printf("pwm Failed\r\n");
41.         close(fd);
42.         return -1;
43.     }
44.
45.     while(1)
46.     {
47.         if(duty <= 0xffffffff && 0 == flag)
48.         {
49.             duty += 2000000;
50.         }
51.         else if(duty >= 0x0fffffff)
52.         {
53.             duty -= 500000;
54.             flag = 1;
55.         }
56.         else
57.         {
58.             flag = 0;
59.         }
60.
61.         /* 设置占空比 */
62.         retval = ioctl(fd, PWM_DUTY, duty);
63.         if(retval < 0)
64.         {
65.             printf("pwm Failed\r\n");
66.             close(fd);
67.             return -1;
68.         }
69.         usleep(5);
70.     }
71.
72.
73.     return 0;
74. }
```

### Part 16.3.5: Run Test

The steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
```

```
insmod ax-pwm-drv.ko  
cd ./build-ax-pwm-test-IDE_5_7_1_GCC_64bit-Debug  
../ax-pwm-test /dev/ax_pwm
```

The IP and path are adjusted according to the actual situation.

The debugging results in the serial port tool are as follows:

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt  
root@ax_peta:~# cd /mnt  
root@ax_peta:/mnt# mkdir /tmp/qt  
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt  
random: fast init done  
EXT4-fs (loop0): recovery complete  
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)  
root@ax_peta:/mnt# cd /tmp/qt  
root@ax_peta:/tmp/qt# source ./qt_env_set.sh  
/tmp/qt  
root@ax_peta:/tmp/qt# cd /mnt  
root@ax_peta:/mnt# insmod ax-pwm.ko  
ax_pwm: loading out-of-tree module taints kernel.  
[drm] load() is deferred & will be called again  
root@ax_peta:/mnt# cd ./build-ax-pwm-test-ZYNQ-Debug  
root@ax_peta:/mnt/build-ax-pwm-test-ZYNQ-Debug# ./ax-pwm-test /dev/ax_pwm
```

**PL\_LED4** on the FPGA development board starts to blink like a breathing light.

## Part 17: I2C Driver

I2C is a commonly used serial interface bus, and a complete I2C driver framework is also provided in the Linux kernel. This chapter will take a look at the I2C driver framework provided in the Linux kernel.

### Part 17.1: I2C Driver Framework

In both the **platform** driver framework and the **pwm** driver framework, the separation of drivers has been mentioned, that is, the separation of controllers or buses and devices. **I2C** is also a similar structure, divided into **I2C bus** and **I2C devices**. The bus is the I2C resource of the chip itself, and the device is the user device external to the I2C, such as RTC and EEPROM. Let's look at their implementation methods separately.

The structure **i2c\_adapter** is used in the kernel to represent the I2C controller, and the **i2c\_adapter** structure is defined in the file **include/linux/i2c.h** as follows:

#### Part 17.1.1: I2C controller driver

The structure **i2c\_adapter** is used in the kernel to represent the **I2C** controller, and the **i2c\_adapter** structure is defined in the file **include/linux/i2c.h** as follows:

```
546. struct i2c_adapter {  
547.     struct module *owner;  
548.     unsigned int class; /* classes to allow probing for */  
549.     const struct i2c_algorithm *algo; /* the algorithm to access the bus */  
550.     void *algo_data;  
551.  
552.     /* data fields that are valid for all devices */  
553.     const struct i2c_lock_operations *lock_ops;  
554.     struct rt_mutex bus_lock;  
555.     struct rt_mutex mux_lock;  
556.  
557.     int timeout; /* in jiffies */  
558.     int retries;  
559.     struct device dev; /* the adapter device */  
560.  
561.     int nr;  
562.     char name[48];
```

```
563.     struct completion dev_released;
564.
565.     struct mutex userspace_clients_lock;
566.     struct list_head userspace_clients;
567.
568.     struct i2c_bus_recovery_info *bus_recovery_info;
569.     const struct i2c_adapter_quirks *quirks;
570.};
```

The **const struct i2c\_algorithm** pointer member variable **algo** in line 549 is a collection of interface functions for **I2C** devices to access the bus, and is a method of communication between **I2C devices** and **I2C controllers**.

The **i2c\_algorithm** structure is defined as follows:

```
407. struct i2c_algorithm {
408.     /* If an adapter algorithm can't do I2C-level access, set master_xfer
409.      to NULL. If an adapter algorithm can do SMBus access, set
410.      smbus_xfer. If set to NULL, the SMBus protocol is simulated
411.      using common I2C messages */
412.     /* master_xfer should return the number of messages successfully
413.      processed, or a negative value on error */
414.     int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
415.                        int num);
416.     int (*smbus_xfer)(struct i2c_adapter *adap, u16 addr,
417.                        unsigned short flags, char read_write,
418.                        u8 command, int size, union i2c_smbus_data *data);
419.
420.     /* To determine what the adapter supports */
421.     u32 (*functionality)(struct i2c_adapter *);
422.
423.#if IS_ENABLED(CONFIG_I2C_SLAVE)
424.     int (*reg_slave)(struct i2c_client *client);
425.     int (*unreg_slave)(struct i2c_client *client);
426.#endif
```

The **master\_xfer** function on **Line 414** is the function used to communicate with the I2C device.

The **smbus\_xfer** function in **Line 416** is the transfer function of smbus.

The realization of the I2C bus driver must first define and initialize a variable of the **i2c\_adapter** structure. The initialization includes the realization of the function in **i2c\_algorithm**, and at least the **master\_xfer** function.

After initialization, use the following function to register **i2c\_algorithm** with the kernel:

```
int i2c_add_adapter(struct i2c_adapter *adapter)
```

```
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
```

The difference between these two functions is that the former uses a dynamic bus number, while the latter uses a static bus number. The input parameter is the initialized **i2c\_add\_adapter** structure variable. This time 0 registration is successful, this time negative value registration fails.

The relative unregister function is:

```
void i2c_del_adapter(struct i2c_adapter * adap)
```

The input parameter **adap** is the I2C controller to be deleted.

The driver of the **I2C** controller is generally provided by the chip manufacturer. Find the node of the **I2C** controller from the device tree. The **compatible** attribute in the node can be found as **cdns**, **i2c-r1p10**. Through this **compatible**, you can track down the **I2C** controller driver used. It is **drivers\i2c\busses\i2c-cadence.c**. I2C controller driver introduces these first, focusing on I2C device driver.

### Part 17.1.2: I2C Device Driver

The device driver is divided into two parts, the device **i2c\_client** and the driver **i2c\_driver**.

#### 1) i2c\_client

**i2c\_client** is used to describe device information and is defined as follows:

```
228. struct i2c_client {  
229.     unsigned short flags;      /* div., see below */  
230.     unsigned short addr;       /* chip address - NOTE: 7bit */  
231.             /* addresses are stored in the */  
232.             /* _LOWER_ 7 bits */  
233.     char name[I2C_NAME_SIZE];  
234.     struct i2c_adapter *adapter; /* the adapter we sit on */  
235.     struct device dev;        /* the device structure */  
236.     int irq;                /* irq issued by device */  
237.     struct list_head detected;  
238. #if IS_ENABLED(CONFIG_I2C_SLAVE)  
239.     i2c_slave_cb_t slave_cb;   /* callback for slave mode */  
240.#endif  
241.};
```

The **addr** in **Line 230** represents the chip address, which is

stored in the lower 7 bits.

The **name** in **Line 233** represents the device name.

The **adapter** bit device in **Line 234** corresponds to the **I2C** controller.

Each **I2C** device corresponds to an **i2c\_client**.

## 2) i2c\_driver

**i2c\_driver** is the focus of I2C processing in the Linux framework, defined as follows

```
169. struct i2c_driver {  
170.     unsigned int class;  
171.  
172.     /* Notifies the driver that a new bus has appeared. You should avoid  
173.      * using this, it will be removed in a near future.  
174.      */  
175.     int (*attach_adapter)(struct i2c_adapter *) deprecate;  
176.  
177.     /* Standard driver model interfaces */  
178.     int (*probe)(struct i2c_client *, const struct i2c_device_id *);  
179.     int (*remove)(struct i2c_client *);  
180.  
181.     /* driver model interfaces that don't relate to enumeration */  
182.     void (*shutdown)(struct i2c_client *);  
183.  
184.     /* Alert callback, for example for the SMBus alert protocol.  
185.      * The format and meaning of the data value depends on the protocol.  
186.      * For the SMBus alert protocol, there is a single bit of data passed  
187.      * as the alert response's low bit ("event flag").  
188.      * For the SMBus Host Notify protocol, the data corresponds to the  
189.      * 16-bit payload data reported by the slave device acting as master.  
190.      */  
191.     void (*alert)(struct i2c_client *, enum i2c_alert_protocol protocol,  
192.                    unsigned int data);  
193.  
194.     /* a ioctl like command that can be used to perform specific functions  
195.      * with the device.  
196.      */  
197.     int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);  
198.  
199.     struct device_driver driver;  
200.     const struct i2c_device_id *id_table;  
201.  
202.     /* Device detection callback for automatic device creation */  
203.     int (*detect)(struct i2c_client *, struct i2c_board_info *);  
204.     const unsigned short *address_list;  
205.     struct list_head clients;  
206.};
```

The **probe** function in **Line 178** is similar to that in the platform framework. It will be executed after the **I2C** device and the driver are successfully matched.

The **device\_driver** structure variable **driver** in **Line 199** is used to match the device. Similar to the **platform** framework, if you use the

device tree, you need to set the **compatible** property in **driver.of\_match\_table**.

The **id\_table** in the **Line 200** is similar to the **platform** framework. This matching table is used when the device tree is not used. After defining and initializing **i2c\_driver**, use the following function to register with the kernel:

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
```

The **owner** is generally **THIS\_MODULE**.

The **driver** is the **i2c\_driver** that needs to be registered. Return **0** for success, negative value for failure.

The relative unregister function is:

```
void i2c_del_driver(struct i2c_driver *driver)
```

I2C device driver writing example:

```
1. static int ax_probe(struct i2c_client *client, const struct i2c_device_id *id)
2. {
3.     return 0;
4. }
5.
6. static int ax_remove(struct i2c_client *client)
7. {
8.     return 0;
9. }
10.
11. static const struct of_device_id ax_of_match[] =
12. {
13.     { .compatible = "alinx-xxx",
14.       /* sentinel */}
15. };
16.
17. static struct i2c_driver ax_driver = {
18.     .driver = {
19.         .owner = THIS_MODULE,
20.         .name   = "alinx-xxx",
21.         .of_match_table = ax_of_match,
22.     },
23.     .probe = ax_probe,
24.     .remove = ax_remove,
25. };
26.
27.
28. static int ax_init(void)
29. {
30.     i2c_add_driver(&ax_driver);
31.     return 0;
32. }
33.
34. static void ax_exit(void)
35. {
36.     i2c_del_driver(&ax_driver);
37. }
38.
39. module_init(ax_init);
```

```
|40. module_exit(ax_exit);
```

### Part 17.1.3: I2C device driver implementation process

#### 1) Device tree

First, add the device node to the corresponding I2C node, as follows:



The **i2c0** in the first line is the node of the controller. **Line 11** refers to the controller node **&i2c**, and adds the device node **axrtc** in it. The **@** following the device node name is the address **68** of the device. There are two key attributes of device nodes. **Compatibility** is used to match the device driver. **reg** is the same as the value after the node name **@**, and both are device addresses.

After the device tree is configured, the device address can be obtained in the **probe** function.

#### 2) Data sending and receiving

The **I2C** method is implemented by the **i2c\_transfer** function in the kernel, which will eventually call the **master\_xfer** function in the **i2c** controller driver.

The **i2c\_transfer** function is defined in **include/linux/i2c.h**, and the prototype is as follows:

```
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
```

The parameter **adap** can be obtained in the **probe** function. When the probe function is called, the first input parameter is the **struct i2c\_client \*client** of the corresponding node in the device tree. The **adap** in the **client** is the corresponding controller.

The parameter **msgs** is the data to be sent.

The parameter **num** is the number of **msgs** to be sent.

If a negative value is returned, it fails, and the returned non-negative value is the number sent by **msgs**.

**msgs** is a pointer of type **struct i2c\_msg**, **struct i2c\_msg** is defined in **include/uapi/linux/i2c.h** as follows:

```

1. struct i2c_msg {
2.     __u16 addr; /* slave address */
3.     __u16 flags;
4. #define I2C_M_RD      0x0001 /* read data, from slave to master */
5.                      /* I2C_M_RD is guaranteed to be 0x0001! */
6. #define I2C_M_TEN     0x0010 /* this is a ten bit chip address */
7. #define I2C_M_RECV_LEN 0x0400 /* length will be first received byte */
8. #define I2C_M_NO_RD_ACK 0x0800 /* if I2C_FUNC_PROTOCOL_MANGLING */
9. #define I2C_M_IGNORE_NAK 0x1000 /* if I2C_FUNC_PROTOCOL_MANGLING */
10. #define I2C_M_REV_DIR_ADDR 0x2000 /* if I2C_FUNC_PROTOCOL_MANGLING */
11. #define I2C_M_NOSTART   0x4000 /* if I2C_FUNC_NOSTART */
12. #define I2C_M_STOP      0x8000 /* if I2C_FUNC_PROTOCOL_MANGLING */
13.     __u16 len; /* msg length */
14.     __u8 *buf; /* pointer to msg data */
15. };

```

According to the value of **flags**, the **i2c\_transfer** function performs different tasks, including reading and writing.

Before calling the **i2c\_transfer** function, you need to create the **struct i2c\_msg** variable first, as follows:

```

1. static int ax_read_regs(struct axrtc_dev *dev, u8 reg, void *val, int len)
2. {
3.     int ret;
4.     struct i2c_msg msg[2];
5.     struct i2c_client *client = (struct i2c_client *)dev->private_data;
6.
7.     msg[0].addr = client->addr;
8.     msg[0].flags = 0;
9.     msg[0].buf = &reg;
10.    msg[0].len = 1;
11.
12.    msg[1].addr = client->addr;
13.    msg[1].flags = I2C_M_RD;
14.    msg[1].buf = val;
15.    msg[1].len = len;
16.

```

```
17.     ret = i2c_transfer(client->adapter, msg, 2);
18.     if(2 == ret)
19.     {
20.         ret = 0;
21.     }
22.     else
23.     {
24.         printk("i2c read failed %d", ret);
25.         ret = -EREMOTEIO;
26.     }
27.
28.     return ret;
29. }
```

**Line 4** defines the array **msg** of type **struct i2c\_msg**.

The client here gets the value in **dev->private\_data**, and **dev->private\_data** is the private variable set in the probe. It will be analyzed in the following experiments. First look at the structure of **msg**.

**Line 7~10** construct **msg[0]**, **addr** is the device address value, just use **addr** in **client**. When **flags** is assigned 0, it is write. When **flags** is writing, the value of **buf** is the first address of the written data. **len** is the length of the written data.

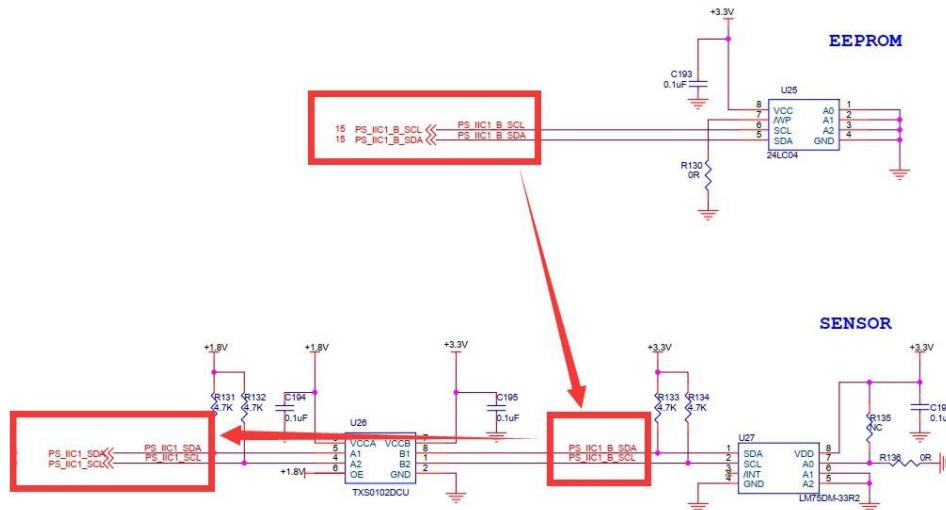
**Line 12~15** construct **msg[1]**, **flag** equal to **I2C\_M\_RD** is read data, at this time **buf** is the first address of the **buffer** for storing read data. **len** is the length of the read data.

### Part 17.3: Experiment

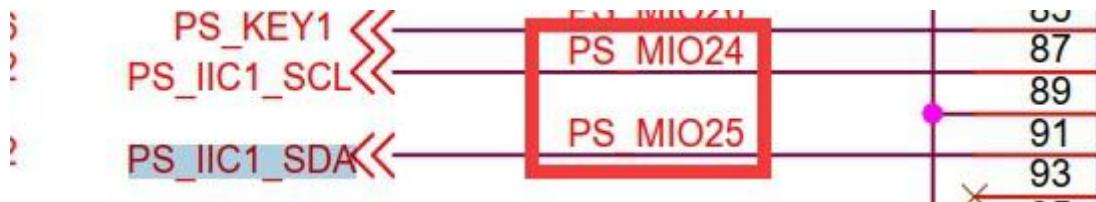
The experiment in this chapter uses **I2C** to do an **eeprom** data reading and writing experiment on the FPGA development board. The main goal is to verify whether the **i2c** framework is used correctly, first write data to the **eeprom** and then read it out. If the two data are consistent, it will succeed.

#### Part 17.3.1: Schematic

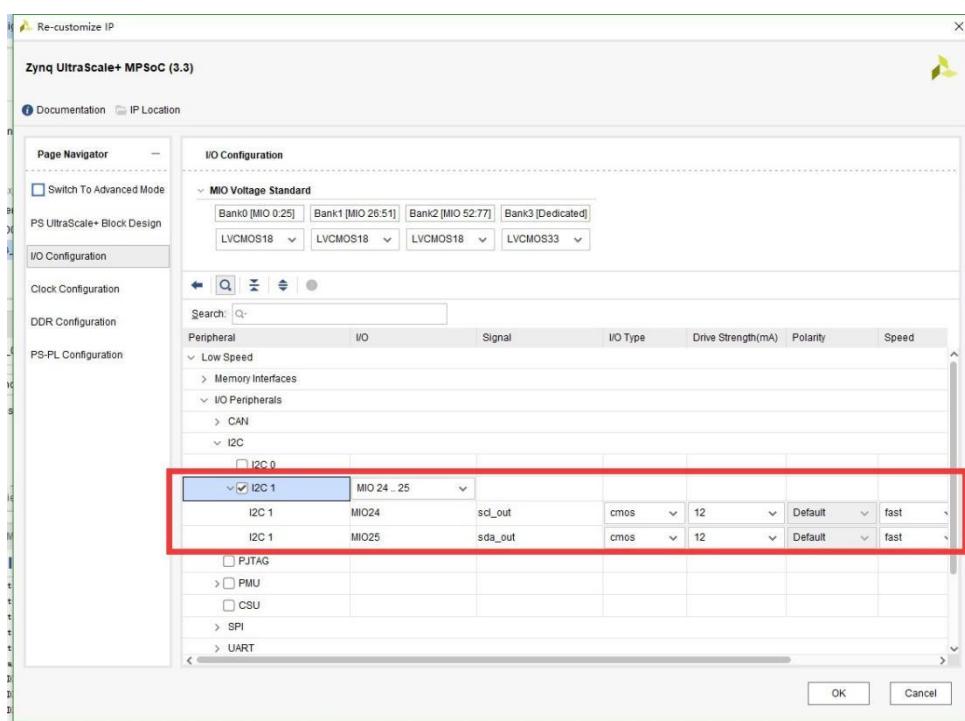
Find the **iic** connected to the **eeprom** and share it with the sensor. Through a conversion chip, finally to **PS\_IIC1\_SDA**.



Find the pins connected to the chip through **PS\_IIC1\_SDA**, **PS\_MIO24** and **PS\_MIO25** on the ps side.



Check the configuration of ultra scale+ in vivado. It has been configured here. Configure the I2C1 channel to the two pins **PS\_MIO24** and **PS\_MIO25**.



### Part 17.3.2: Device Tree

Open the **system-user.dtsi** file and add the following node outside the root directory:

```
1. &i2c1 {  
2.     clock-frequency = <100000>;  
3.  
4.     ax-e2p1@50 {  
5.         compatible = "ax-e2p1";  
6.         reg = <0x50>;  
7.     };  
8. };
```

The first line uses the **i2c1** node, because when we configured the pins in **vivado** above, the pins of the **eeprom** were constrained to **i2c1**, so the corresponding device tree needs to be added to the **i2c\_1** node.

**Line 2** sets the clock of **i2c\_1** to **100hz**.

Add the **eeprom** node **ax-e2p1@50** in **Line 4**, and the device address is **50**. Add the **compatible** attribute equal to "**x-e2p1**" for matching with the driver. Adding the **reg** attribute is equal to the device address **0x50**.

### Part 17.3.3: driver

Use **petalinux** to create a new driver named "**ax-i2c**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-i2c.c** file:

```
1. #include <linux/types.h>  
2. #include <linux/kernel.h>  
3. #include <linux/delay.h>  
4. #include <linux/ide.h>  
5. #include <linux/init.h>  
6. #include <linux/module.h>  
7. #include <linux/errno.h>  
8. #include <linux/cdev.h>  
9. #include <linux/device.h>  
10. #include <linux/string.h>  
11. #include <linux/timer.h>  
12. #include <linux/i2c.h>  
13. #include <linux/irq.h>  
14. #include <linux/wait.h>  
15. #include <linux/poll.h>
```

```
16. #include <linux/fs.h>
17. #include <linux/fcntl.h>
18. #include <linux/platform_device.h>
19.
20. #include <asm/uaccess.h>
21. #include <asm/io.h>
22.
23. /* 驱动个数 */
24. #define AX_I2C_CNT 1
25. /* 设备节点名称 */
26. #define AX_I2C_NAME "ax_i2c_e2p"
27.
28. struct ax_i2c_dev {
29.     dev_t devid;           //设备号
30.     struct cdev cdev;     //字符设备
31.     struct class *class;  //类
32.     struct device *device; //设备
33.     int major;            //主设备号
34.     void *private_data;   //用于在probe函数中获取client
35. };
36. /* 声明设备结构体变量 */
37. struct ax_i2c_dev axi2cdev;
38.
39. /* i2c 数据读取
40.  * struct ax_i2c_dev *dev : 设备结构体
41.  * u8 reg : 数据在目标设备中的地址
42.  * void *val : 数据 buffer 首地址
43.  * int len : 数据长度
44. */
45. static int ax_i2c_read_regs(struct ax_i2c_dev *dev, u8 reg, void *val, int len)
46. {
47.     int ret;
48.     /* 构建 msg, 读取时一般使用一个至少两个元素的msg 数组
49.      第一个元素用于发送目标数据地址(写), 第二个元素发送buffer 地址(读) */
50.     struct i2c_msg msg[2];
51.     /* 从设备结构体变量中获取 client 数据 */
52.     struct i2c_client *client = (struct i2c_client *)dev->private_data;
53.
54.     /* 构造 msg */
55.     msg[0].addr = client->addr;    //设置设备地址
56.     msg[0].flags = 0;              //标记为写, 先给eprom 发送读取数据的所在地址
57.     msg[0].buf = &reg;            //读取数据的所在地址
58.     msg[0].len = 1;               //地址数据长度, 只发送首地址的话长度就为 1
59.
60.     msg[1].addr = client->addr;    //设置设备地址
61.     msg[1].flags = I2C_M_RD;       //标记为读
62.     msg[1].buf = val;             //数据读出的buffer 地址
63.     msg[1].len = len;             //读取数据长度
64.
65.     /* 调用 i2c_transfer 发送 msg */
66.     ret = i2c_transfer(client->adapter, msg, 2);
67.     if(2 == ret)
68.     {
69.         ret = 0;
70.     }
71.     else
72.     {
73.         printk("i2c read failed %d\r\n", ret);
74.         ret = -EREMOTEIO;
75.     }
76.
77.     return ret;
78. }
79.
80. /* i2c 数据写入
81.  * struct ax_i2c_dev *dev : 设备结构体
82.  * u8 reg : 数据在目标设备中的地址
83.  * void *val : 数据 buffer 首地址
84.  * int len : 数据长度
85. */
```

```
86. static s32 ax_i2c_write_regs(struct ax_i2c_dev *dev, u8 reg, u8 *buf, int len)
87. {
88.     int ret;
89.     /* 数据 buffer */
90.     u8 b[100] = {0};
91.     /* 构建 msg */
92.     struct i2c_msg msg;
93.     /* 从设备结构体变量中获取 client 数据 */
94.     struct i2c_client *client = (struct i2c_client *)dev->private_data;
95.
96.     /* 把写入目标地址放在 buffer 的第一个元素中首先发送 */
97.     b[0] = reg;
98.     /* 把需要发送的数据拷贝到随后的地址中 */
99.     memcpy(&b[1], buf, 100 > len ? len : 100);
100.
101.    /* 构建 msg */
102.    msg.addr = client->addr;      //设置设备地址
103.    msg.flags = 0;                //标记为写
104.    msg.buf = b;                  //数据写入的buffer 地址
105.    msg.len = len + 1;            //写入的数据长度，因为除了用户数据外，//还需要发送数据地址所以要+1
106.
107.
108.    /* 调用 i2c_transfer 发送 msg */
109.    ret = i2c_transfer(client->adapter, &msg, 1);
110.
111.    if(1 == ret)
112.    {
113.        ret = 0;
114.    }
115.    else
116.    {
117.        printk("i2c write failed %d\r\n", ret);
118.        ret = -EREMOTEIO;
119.    }
120.    return ret;
121.}
122.
123./* open 函数实现，对应到 Linux 系统调用函数的open 函数 */
124.static int ax_i2c_open(struct inode *inode, struct file *filp)
125.{ 
126.    /* 设置私有数据 */
127.    filp->private_data = &ax_i2cdev;
128.    return 0;
129.}
130.
131./* read 函数实现，对应到 Linux 系统调用函数的read 函数 */
132.static ssize_t ax_i2c_read(struct file *file, char user *buf, size_t size, loff_t *offset)
133.{ 
134.    /* 获取私有数据 */
135.    struct ax_i2c_dev *dev = (struct ax_i2c_dev *)file->private_data;
136.    /* 读取数据 buffer */
137.    char b[100] = {0};
138.    int ret = 0;
139.    /* 从 0 地址开始读，这里只是为了实验方便使用了read 并且把地址写死了，实际的应用中不应该在驱动中把地址写死，可以尝试使用ioctl 去实现灵活的方法 */
140.    ax_i2c_read_regs(dev, 0x00, b, 100 > size ? size : 100);
141.
142.    /* 把读取到的数据拷贝到用户读取的地址 */
143.    ret = copy_to_user(buf, b, 100 > size ? size : 100);
144.    return 0;
145.}
146.
147.
148./* write 函数实现，对应到Linux 系统调用函数的 write 函数 */
149.static ssize_t ax_i2c_write(struct file *file, const char user *buf, size_t size, loff_t *offset)
150.{ 
151.    /* 获取私有数据 */
152.    struct ax_i2c_dev *dev = (struct ax_i2c_dev *)file->private_data;
153.    /* 写入数据的buffer */
154.    static char user_data[100] = {0};
```

```
155. int ret = 0;
156. /* 获取用户需要发送的数据 */
157. ret = copy_from_user(user_data, buf, 100 > size ? size : 100);
158. if(ret < 0)
159. {
160.     printk("copy user data failed\r\n");
161.     return ret;
162. }
163. /* 和读对应的从 0 开始写 */
164. ax_i2c_write_regs(dev, 0x00, user_data, size);
165.
166. return 0;
167.}
168.
169./* release 函数实现，对应到 Linux 系统调用函数的close 函数 */
170.static int ax_i2c_release(struct inode *inode, struct file *filp)
171.{ 
172.    return 0;
173.}
174.
175./* file_operations 结构体声明 */
176.static const struct file_operations ax_i2c_ops = {
177.    .owner = THIS_MODULE,
178.    .open = ax_i2c_open,
179.    .read = ax_i2c_read,
180.    .write = ax_i2c_write,
181.    .release = ax_i2c_release,
182.};
183.
184./* probe 函数实现，驱动和设备匹配时会被调用 */
185.static int axi2c_probe(struct i2c_client *client, const struct i2c_device_id *id)
186.{ 
187.    printk("eeprom probe\r\n");
188.    /* 构建设备号 */
189.    alloc_chrdev_region(&axi2cdev.devid, 0, AX_I2C_CNT, AX_I2C_NAME);
190.
191.    /* 注册设备 */
192.    cdev_init(&axi2cdev.cdev, &ax_i2c_ops);
193.    cdev_add(&axi2cdev.cdev, axi2cdev.devid, AX_I2C_CNT);
194.
195.    /* 创建类 */
196.    axi2cdev.class = class_create(THIS_MODULE, AX_I2C_NAME);
197.    if(IS_ERR(axi2cdev.class))
198.    {
199.        return PTR_ERR(axi2cdev.class);
200.    }
201.
202.    /* 创建设备 */
203.    axi2cdev.device = device_create(axi2cdev.class, NULL, axi2cdev.devid, NULL, AX_I2C_NAME);
204.    if(IS_ERR(axi2cdev.device))
205.    {
206.        return PTR_ERR(axi2cdev.device);
207.    }
208.
209.    axi2cdev.private_data = client;
210.
211.    return 0;
212.}
213.
214./* remove 函数实现，驱动卸载时会被调用 */
215.static int axi2c_remove(struct i2c_client *client)
216.{ 
217.    /* 删除设备 */
218.    cdev_del(&axi2cdev.cdev);
219.    unregister_chrdev_region(axi2cdev.major, AX_I2C_CNT);
220.    /* 注销类 */
221.    device_destroy(axi2cdev.class, axi2cdev.devid);
222.    class_destroy(axi2cdev.class);
223.
224.}
225.
```

```

226./* 匹配表，设备树下的匹配方式 */
227.static const struct of_device_id axi2c_of_match[] =
228.{
229.    { .compatible = "ax-e2p1",
230.      /* sentinel */}
231.};
232.
233./* 传统的id_table 匹配方式 */
234.static const struct i2c_device_id axi2c_id[] = {
235.    {"ax-e2p1"},
236.    {}
237.};
238.
239./* 声明并初始化i2c 驱动 */
240.static struct i2c_driver axi2c_driver = {
241.    .driver = {
242.        .owner = THIS_MODULE,
243.        .name = "ax-e2p1",
244.        /* 用 of_match_table 匹配 */
245.        .of_match_table = axi2c_of_match,
246.    },
247.    /* 使用传统的方式匹配 */
248.    .id_table = axi2c_id,
249.    .probe = axi2c_probe,
250.    .remove = axi2c_remove,
251.};
252.
253./* 驱动入口函数 */
254.static int init ax_i2c_init(void)
255.{
256.    /* 在入口函数中调用 i2c_add_driver，注册驱动 */
257.    return i2c_add_driver(&axi2c_driver);
258.}
259.
260./* 驱动出口函数 */
261.static void exit ax_i2c_exit(void)
262.{
263.    /* 在出口函数中调用 i2c_del_driver，卸载驱动 */
264.    i2c_del_driver(&axi2c_driver);
265.}
266.
267./* 标记加载、卸载函数 */
268.module_init(ax_i2c_init);
269.module_exit(ax_i2c_exit);
270.
271./* 驱动描述信息 */
272.MODULE_AUTHOR("Alinx");
273.MODULE_ALIAS("pwm_led");
274.MODULE_DESCRIPTION("I2C EEPROM driver");
275.MODULE_VERSION("v1.0");
276.MODULE_LICENSE("GPL");

```

**Line 34**, add a **void** pointer to get the **client**.

**Lines 45 to 78** implement a general **i2c** reading function. The **msg** is constructed in the **i2c** receiving process. Generally, a two-element array is constructed. The first **msg** is marked as the address for sending and receiving data, and the second **msg** is marked as being read.

**Line 86 to Line 121**, realize the general **i2c** writing function.

**Line 127**, set the private data in the **open** function.

**Line 132** implements the **read** function and calls the general read function implemented earlier.

**Line 149** implements the **write** function, which calls the general write function implemented earlier.

**Line 185**, implement the **probe** function, the content is the registration of the character device.

**Line 209**, the **client** of the **probe** input parameter is assigned to the private data in the device structure variable. **ax\_i2c\_dev** type handle that will be used in **i2c** read and write functions.

**Line 227** defines the of matching table, and the **compatible** field is consistent with that in the device tree.

**Line 234** defines an **id\_table** for traditional matching.

**Line 240**, declare and initialize **i2c\_driver**.

**Line 257**, use the **i2c\_add\_driver** function to register the i2c driver in the driver entry function. The **i2c\_add\_driver** function is an encapsulation of **i2c\_register\_driver**, eliminating the need to enter **THIS\_MODULE**.

**Line 264**, use **i2c\_del\_driver** in the driver export function to delete the driver.

#### Part 17.3.4: Test Code

Create a new QT project named "**ax-i2c-test**", create a new **main.c**, and enter the following code:

```
1. #include "stdio.h"
2. #include "unistd.h"
3. #include "sys/types.h"
4. #include "sys/stat.h"
5. #include "fcntl.h"
6. #include "stdlib.h"
7. #include "string.h"
8. #include "assert.h"
9.
10. int main(int argc, char *argv[])
11. {
12.     int fd, ret;
13.     char *filename;
14.     char buffer[3] = {0};
```

```
15.
16.     if(argc != 2)
17.     {
18.         printf("Error Usage\r\n");
19.         return -1;
20.     }
21.
22.     filename = argv[1];
23.     fd = open(filename, O_RDWR);
24.     if(fd < 0)
25.     {
26.         printf("file %s open failed\r\n", argv[1]);
27.         return -1;
28.     }
29.
30.    /* 随便写入一些数据 */
31.    buffer[0] = 0x5A;
32.    buffer[1] = 0x55;
33.    buffer[2] = 0xAA;
34.    ret = write(fd, buffer, sizeof(buffer));
35.    if(ret < 0)
36.    {
37.        printf("write failed\r\n");
38.    }
39.    /* 在控制台打印写入的数据 */
40.    printf("write data %X, %X, %X\r\n", buffer[0], buffer[1], buffer[2]);
41.
42.    /* 初始化 buffer, 再用来读取数据 */
43.    memset(buffer, 0, sizeof(buffer));
44.    usleep(4000);
45.    /* 读出数据 */
46.    ret = read(fd, buffer, sizeof(buffer));
47.    if(ret < 0)
48.    {
49.        printf("read failed\r\n");
50.    }
51.    /* 在控制台打印读出的数据 */
52.    printf("read data %X, %X, %X\r\n", buffer[0], buffer[1], buffer[2]);
53.
54.    close(fd);
55.
56.    return 0;
57. }
```

The test program is simply to read and write and print out separately, compare the read and write results.

Add a delay to line 44, otherwise it may fail to write. The delay time is adjusted according to the situation.

### Part 17.3.5: Run Test

The steps are as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
```

```
source ./qt_env_set.sh  
cd /mnt  
insmod ./ax-i2c.ko  
cd ./build-ax_i2c_test-ZYNQ-Debug/  
../ax_i2c_test /dev/ax_i2c_e2p
```

The IP and path are adjusted according to the actual situation.

The debugging results in the serial port tool are as follows:

```
root@ax_peta:~# cd /mnt  
root@ax_peta:/mnt# mkdir /tmp/qt  
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt  
EXT4-fs (loop0): recovery complete  
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)  
root@ax_peta:/mnt# cd /tmp/qt  
root@ax_peta:/tmp/qt# source ./qt_env_set.sh  
/tmp/qt  
root@ax_peta:/tmp/qt# cd /mnt  
root@ax_peta:/mnt# insmod ./ax-i2c.ko  
ax_i2c: loading out-of-tree module taints kernel.  
eeprom probe  
[drm] load() is defered & will be called again  
root@ax_peta:/mnt# cd ./build-ax_i2c_test-ZYNQ-Debug/  
root@ax_peta:/mnt/build-ax_i2c_test-ZYNQ-Debug# ./.ax_i2c_test /dev/ax_i2c_e2p  
write data 5A, 55, AA  
read data 5A, 55, AA  
root@ax_peta:/mnt/build-ax_i2c_test-ZYNQ-Debug# █
```

The reading and writing results are consistent and the test is successful.

## Part 18: USB Driver

usb is the general term for Universal Serial Bus. Like windows, Linux also supports almost all usb devices such as mice, keyboards, printers, etc. In this chapter, let's learn about usb related content in Linux.

### Part 18.1: USB Recognition Process

After the usb device is connected to the host, the matching process is as follows:

#### 1) Hardware detection

The **usb** interface has four wires: **5V, GND, D-, D+**. The **D-** or **D+** in the USB interface of the host has a pull-down resistor, and it is low when there is no load. The internal **D-** or **D+** of the usb interface on the device side has a pull-up resistor. After the usb device is connected to the host, the **D-** or **D+** of the host's USB interface will be pulled high, so that the host detects the access of the USB device from a hardware perspective.

#### 2) Handshake match

After the host detects the USB device access, it will interact with the device. The host actively initiates a descriptor for obtaining device information, and the device needs to return the descriptor in a fixed format. If it is under the windows desktop system, we can now see the pop-up window of **xxx** device access.

#### 3) Assign address

Multiple usb devices can be connected to a host. In order to distinguish these devices, the host will assign device addresses to the devices after the handshake is successful, and the commands sent by the host will include address information.

Before the handshake, the host will use the number 0 to interact with the device.

## **Part 18.2: USB Transmission**

usb transmission is a master-slave structure, the master is the host, and the slave is the device. All transmissions are initiated by the host, and the device has no ability to actively notify the host. The input and output we often talk about are all described from the perspective of the host, such as input device keyboard, mouse, output device display and so on.

### **Part 18.2.1: USB Transmission Mode**

Based on the master-slave structure, usb can be divided into four transmission modes:

#### **1) Control transmission**

Control transmission is a special transmission method, which is relatively complicated. When the usb device is connected to the host, the host uses the control transmission method to send control commands to configure the device. At the same time, it is necessary to obtain the descriptor of the usb device through the control transmission to identify the device, and the control transmission is used for data exchange during the enumeration of the device. Reliable and timeliness can be guaranteed.

#### **2) Bulk transfer**

Batch transmission is generally used in occasions where the amount of data is large but the time requirement is not high, such as USB flash drives, printers, etc. Reliable and does not guarantee timeliness.

#### **3) Interrupt transmission**

Interrupted transmission is generally used in occasions where the amount of data is small, discontinuous, and real-time requirements are high, such as such as mouse and keyboard input devices.

#### 4) Isochronous transmission

Isochronous transmission is generally used in occasions with large data volume, continuous and high real-time requirements, but reliability is difficult to guarantee, generally used in cameras, microphones and other equipment

#### Part 18.2.2: Endpoint

The endpoint is the **usb** transmission object. The endpoint is an abstract concept in the **usb** protocol stack level, and it can be said that it is a concept that exists for software workers. Starting from the **host** host in the **usb** protocol, a host can be connected to one or more **devices**. This level is a physical **usb** connection. A device can have one or more **interfaces**. This layer is a logical concept from the beginning. A two-in-one usb device such as a headset has two interfaces for microphone and earphone. An **interface** can have one or more **endpoints**. This is also a logical concept. For example, a usb flash drive needs to have an input endpoint and an output endpoint.

In addition to functional endpoints, each **usb** device must have endpoint **0**. Endpoint **0** is generally used for device initialization and configuration, supports control transmission, and is always configured when the device is connected to the host.

#### Part 18.2.3: Pipe

The **pipe** is the association between the endpoint on the device and the software on the host. Pipelines represent the ability to move data between software on the host through memory buffers and

---

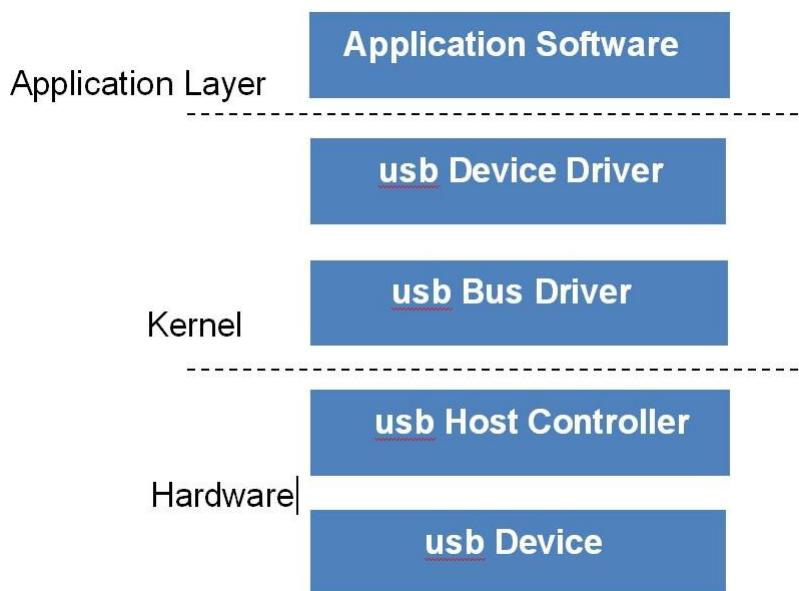
endpoints on the device. There are two mutually exclusive pipeline communication modes:

**Stream:** The data transmitted through the pipe has no structure defined by **USB**.

**Message:** The data moved through the pipe has some USB-defined structure.

### Part 18.3: USB bus driver

The bus driver is no longer unfamiliar. The previous palform driver and i2c driver are both of this idea. Similarly, usb is also divided into two drivers: bus and device. The level of usb driver framework is as follows:



The usb bus driver and the hardware are connected, and play a role of undertaking. The initialization process of usb mentioned above requires a bus driver to complete. The bus driver needs to do these:

- 1) Identify usb device
- 2) Find and install the corresponding device driver
- 3) Provides harmonic functions for device drivers

The bus driver is generally provided by the chip manufacturer.

Open the FPGA development board, connect the mouse and pull it out, you can see the following information in the console:

```
root@ax_peta:~# usb 1-1: new low-speed USB device number 2 using ci_hdrc
input: Logitech USB Optical Mouse as /devices/soc0/amba/e0002000.usb1/ci_hdrc.0/usb1/1-1:1.0/0003:046D:C077.0001/input/input0
hid-generic 0003:046D:C077.0001: input: USB HID v1.11 Mouse [Logitech USB Optical Mouse] on usb-ci_hdrc.0-1/input0
[drm] load() is defered & will be called again
usb 1-1: USB disconnect, device number 2
```

## Part 18.4: USB device driver

The usb device driver is also divided into two parts: device and driver. The difference from i2c is that usb only needs to implement the driver part, and the device part is obtained by the bus driver.

When a **USB** device is connected, an interrupt **hub\_irq()** will be generated, and a numbered address **choose\_address(udev)** will be assigned in the interrupt. The bus tells the device **hub\_set\_address()** of this **address**. Then issue a command to get the device descriptor **usb\_get\_device\_descriptor()**. After obtaining the device information, a device **device\_add()** will be registered. This **device** will be placed in the device linked list of the **usb** bus **usb\_bus\_type**. After the **usb** driver is registered, it will take out the device driver structure **usb\_driver** from the driver list of the **usb** bus, compare **usb\_interface** with the **id\_table** of **usb\_driver**, and call the **probe** function in **usb\_driver** if the match succeeds.

What we have to do is to define, initialize and register **usb\_driver**, and implement the **probe** function.

Let's take a look at the **usb\_driver** structure first, which is defined in the file **include/linux/usb.h** as follows:

```
1. struct usb_driver {
2.     const char *name;
3.
4.     int (*probe) (struct usb_interface *intf,
5.                   const struct usb_device_id *id);
6.
7.     void (*disconnect) (struct usb_interface *intf);
8.
9.     int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
10.                           void *buf);
11.
12.    int (*suspend) (struct usb_interface *intf, pm_message_t message);
```

```
13.     int (*resume) (struct usb_interface *intf);
14.     int (*reset_resume)(struct usb_interface *intf);
15.
16.     int (*pre_reset)(struct usb_interface *intf);
17.     int (*post_reset)(struct usb_interface *intf);
18.
19.     const struct usb_device_id *id_table;
20.
21.     struct usb_dynids dynids;
22.     struct usbdrv_wrap drvwrap;
23.     unsigned int no_dynamic_id:1;
24.     unsigned int supports_autosuspend:1;
25.     unsigned int disable_hub_initiated_lpm:1;
26.     unsigned int soft_unbind:1;
27. };
```

**name** is the device name.

The **probe** function is a function that will be executed after the device is successfully matched and must be implemented.

The **disconnect** function is executed when the device is unavailable, such as when the device is unplugged.

**id\_table** is used to match devices.

**usb\_device\_id** is defined in [include/linux/mod\\_devicetable.h](#) and can be initialized with the following macro:

```
USB_INTERFACE_INFO(cl,sc,pr)
```

**cl** is the class, **sc** is the sub class, and **pr** is the protocol. These device descriptors are defined in the file [include\linux\usb\Ch9.h](#). The bus driver will match the device and driver based on these descriptors.

After the **usb\_driver** is defined and initialized, use the following macro to register with the kernel:

```
usb_register(driver)
```

Instead, use the following function to log out:

```
void usb_deregister(struct usb_driver *);
```

USB device driver example:

```
1. static struct usb_device_id usb_id_table [] =
2. {
3.     { USB_INTERFACE_INFO(XXX, XXX, XXX) },
4.     { }
5. };
6.
7.
8. static int usb_probe(struct usb_interface *intf, const struct usb_device_id *id)
```

```

9. {
10.     return 0;
11. }
12.
13. static void usb_disconnect(struct usb_interface *intf)
14. {
15.
16. }
17.
18. static struct usb_driver usb_driver = {
19.     .name      = "xxx",
20.     .probe     = usbmouse_as_key_probe,
21.     .disconnect = usbmouse_as_key_disconnect,
22.     .id_table   = usbmouse_as_key_id_table,
23. };
24.
25.
26. static int usb_init(void)
27. {
28.     usb_register(&usb_driver);
29.     return 0;
30. }
31.
32. static void usb_exit(void)
33. {
34.     usb_deregister(&usb_driver);
35. }
36.
37. module_init(usb_init);
38. module_exit(usb_exit);
39.
40. MODULE_LICENSE("GPL");

```

The input parameter `struct usb_interface *intf` of the `probe` function and `disconnect` function can use the function `interface_to_usbdev()` to obtain `usb_device`, such as

```
struct usb_device *dev = interface_to_usbdev(intf);
```

## Part 18.5: urb request block

`urb` is the basic data structure used to describe the communication of `urb` device in the device driver, and it is the object of endpoint processing. The operation of `usb` device needs to use `urb`. `urb` is defined in the file `include/linux/usb.h`, its definition and comments are as follows:

```

1. struct urb {
2.     /* 私有的：只能由 USB 核心和主机控制器访问的字段 */
3.     struct kref kref; /*urb 引用计数 */
4.     void *hcpriv; /* 主机控制器私有数据 */
5.     atomic_t use_count; /* 并发传输计数 */
6.     u8 reject; /* 传输将失败 */
7.     int unlink; /* unlink 错误码 */
8.     /* 公共的： 可以被驱动使用的字段 */
9.     struct list_head urb_list; /* 链表头 */
10.    struct usb_anchor *anchor;

```

```

11. struct usb_device *dev; /* 关联的 USB 设备 */
12. struct usb_host_endpoint *ep;
13. unsigned int pipe; /* 管道信息 */
14. int status; /* URB 的当前状态 */
15. unsigned int transfer_flags; /* URB_SHORT_NOT_OK | ... */
16. void *transfer_buffer; /* 发送数据到设备或从设备接收数据的缓冲区 */
17. dma_addr_t transfer_dma; /* 用来以 DMA 方式向设备传输数据的缓冲区 */
18. int transfer_buffer_length; /* transfer_buffer 或 transfer_dma 指向缓冲区的大小 */
19.
20. int actual_length; /* URB 结束后, 发送或接收数据的实际长度 */
21. unsigned char *setup_packet; /* 指向控制URB 的设置数据包的指针 */
22. dma_addr_t setup_dma; /* 控制URB 的设置数据包的DMA 缓冲区 */
23. int start_frame; /* 等时传输中用于设置或返回初始帧 */
24. int number_of_packets; /* 等时传输中等时缓冲区数量 */
25. int interval; /* URB 被轮询到的时间间隔 (对中断和等时 urb 有效) */
26. int error_count; /* 等时传输错误数量 */
27. void *context; /* completion 函数上下文 */
28. usb_complete_t complete; /* 当 URB 被完全传输或发生错误时, 被调用 */
29. /* 单个URB 一次可定义多个等时传输时, 描述各个等时传输 */
30. struct usb_iso_packet_descriptor iso_frame_desc[0];
31. };

```

The use process of urb is as follows:

## 1) Create urb

Use the following function to create an **urb**

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

Use the following function to release an **urb**

```
void usb_free_urb(struct urb *urb);
```

## 2) Fill urb

### 3) For interrupt **urb**, use the following function to initialize:

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe,
void *transfer_buffer, int buffer_length, usb_complete_t complete, void *context,
int interval);
```

#### Parameter Description:

**urb**: the pointer of the **urb** to be initialized

**dev**: the usb device to be sent to the urb

**pipe**: The specific endpoint of the **usb** device that the urb is to be sent to, created using the **usb\_sndctrlpipe()** or **usb\_rcvctrlpipe()** function;

**transfer\_buffer**: A pointer to the buffer for sending or receiving data. It cannot be a static buffer, it must be allocated using **kmalloc()**;

**buffer\_length**: the size of the buffer pointed to by the

**transfer\_buffer** pointer;

**complete**: the callback function when the **urb** is completed;

**context**: the context to complete the processing function;

**interval**: **urb** scheduling interval.

- 4) For batch **urb**, use the following function to initialize

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,unsigned int pipe,  
void *transfer_buffer,int buffer_length, usb_complete_t complete,void *context);
```

The pipe here needs to be created using the **usb\_sndbulkpipe()**

or **usb\_rcvbulkpipe()** function. Other parameters are the same as **usb\_fill\_int\_urb()**.

- 5) For the control urb, use the following function to initialize

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,unsigned int pipe,  
unsigned char *setup_packet,void *transfer_buffer, int  
buffer_length,usb_complete_t complete, void *context);
```

**setup\_packet**: The setup packet to be sent to the endpoint.

The pipe here needs to be created using the **usb\_sndctrlpipe()** or **usb\_rcvctrlpipe()** function. Other parameters are the same as **usb\_fill\_int\_urb()**.

- 6) Submit **urb**

After filling, use the following function to submit the **urb**:

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

**mem\_flags** has the following definitions:

**GFP\_ATOMIC**: In the interrupt service function, bottom half, tasklet, timer processing function, and urb completion return function, it is used if the caller holds a spin lock or read-write lock and when the driver modifies the current process to **non-TASK\_RUNNING**.

**GFP\_NOIO**: Used in block I/O and error handling paths of storage devices

**GFP\_KERNEL**: Use this flag in other situations.

After the **urb** is submitted, if the **urb** is successfully sent to the

device, there is an error in the data transmission, or the driver uses `usb_unlink_urb()` or `usb_kill_urb()` to actively cancel the `urb`, the `urb` will end. When the `urb` ends, you can use the member variable status to view the reason for the end.

## Part 18.6: Experiment

This chapter writes a USB mouse click action capture experiment. When the left button is clicked, 1 will be output on the console, and the right button will output 0 on the console.

### Part 18.6.1: Schematic

As mentioned earlier, the hardware is related to the `usb` bus driver. The `usb` bus driver is provided by the chip manufacturer. The `usb` connection on the FPGA development board is the same as the xilinx model, so the `usb` bus driver does not need to be modified.

### Part 18.6.2: Device tree

Just keep the current `usb` node

### Part 18.6.3: Driver code

Use `petalinux` to create a new driver named "`ax-usb-drv`", and execute the `petalinux-config -c rootfs` command to select the new driver.

Enter the following code in the `ax-usb-drv.c` file:

```
1. #include <linux/kernel.h>
2. #include <linux/slab.h>
3. #include <linux/module.h>
4. #include <linux/init.h>
5. #include <linux/usb/input.h>
6. #include <linux/hid.h>
7.
8. /* 定义一个输入事件，表示鼠标的点击事件 */
9. static struct input_dev *mouse_dev;
10. /* 定义缓冲区首地址 */
11. static char *usb_buf;
12. /* dma 缓冲区 */
13. static dma_addr_t usb_buf_dma;
14. /* 缓冲区长度 */
15. static int usb_buf_len;
16. /* 定义一个urb */
```

```
17. static struct urb      *mouse_urb;
18.
19. static void ax_usb_irq(struct urb *urb)
20. {
21.     static unsigned char pre_sts;
22.     int i;
23.
24.     /* 左键发生了变化 */
25.     if ((pre_sts & 0x01) != (usb_buf[0] & 0x01))
26.     {
27.         printk("lf click\n");
28.         input_event(mouse_dev, EV_KEY, KEY_L, (usb_buf[0] & 0x01) ? 1 : 0);
29.         input_sync(mouse_dev);
30.     }
31.
32.     /* 右键发生了变化 */
33.     if ((pre_sts & 0x02) != (usb_buf[0] & 0x02))
34.     {
35.         printk("rt click\n");
36.         input_event(mouse_dev, EV_KEY, KEY_S, (usb_buf[0] & 0x02) ? 1 : 0);
37.         input_sync(mouse_dev);
38.     }
39.
40.     /* 记录当前状态 */
41.     pre_sts = usb_buf[0];
42.
43.     /* 重新提交 urb */
44.     usb_submit_urb(mouse_urb, GFP_KERNEL);
45. }
46.
47. static int ax_usb_probe(struct usb_interface *intf, const struct usb_device_id *id)
48. {
49.     /* 获取usb_device */
50.     struct usb_device *dev = interface_to_usbdev(intf);
51.     struct usb_host_interface *interface;
52.     struct usb_endpoint_descriptor *endpoint;
53.     int pipe;
54.
55.     /* 获取端点 */
56.     interface = intf->cur_altsetting;
57.     endpoint = &interface->endpoint[0].desc;
58.
59.     /* 分配input_dev */
60.     mouse_dev = input_allocate_device();
61.     /* 设置input_dev */
62.     set_bit(EV_KEY, mouse_dev->evbit);
63.     set_bit(EV_REP, mouse_dev->evbit);
64.     set_bit(KEY_L, mouse_dev->keybit);
65.     set_bit(KEY_S, mouse_dev->keybit);
66.     /* 注册input_dev */
67.     input_register_device(mouse_dev);
68.
69.     /* 获取USB设备端点对应的管道 */
70.     pipe = usb_rcvintpipe(dev, endpoint->bEndpointAddress);
71.
72.     /* 获取端点最大长度作为缓冲区长度 */
73.     usb_buf_len = endpoint->wMaxPacketSize;
74.
75.     /* 分配缓冲区 */
76.     usb_buf = usb_alloc_coherent(dev, usb_buf_len, GFP_ATOMIC, &usb_buf_dma);
77.
78.     /* 创建urb */
79.     mouse_urb = usb_alloc_urb(0, GFP_KERNEL);
80.
81.     /* 分配urb */
82.     usb_fill_int_urb(mouse_urb, dev, pipe, usb_buf, usb_buf_len, ax_usb_irq, NULL, endpoint->bInterval);
83.     mouse_urb->transfer_dma = usb_buf_dma;
84.     mouse_urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
85.
86.     /* 提交urb */
87.     usb_submit_urb(mouse_urb, GFP_KERNEL);
```

```
88.     return 0;
89. }
90. }
91.
92. static void ax_usb_disconnect(struct usb_interface *intf)
93. {
94.     struct usb_device *dev = interface_to_usbdev(intf);
95.
96.     /* 主动结束 urb */
97.     usb_kill_urb(mouse_urb);
98.     /* 释放urb */
99.     usb_free_urb(mouse_urb);
100.    /* 释放缓冲区 */
101.    usb_free_coherent(dev, usb_buf_len, usb_buf, &usb_buf_dma);
102.    /* 注销输入事件 */
103.    input_unregister_device(mouse_dev);
104.    /* 释放输入事件 */
105.    input_free_device(mouse_dev);
106.}
107.
108./* 定义初始化id_table */
109.static struct usb_device_id ax_usb_id_table [] = {
110.    /* 鼠标mouse 接口描述符里类是HID类，子类boot，协议 mouse */
111.    {
112.        USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HID,
113.                            USB_INTERFACE_SUBCLASS_BOOT,
114.                            USB_INTERFACE_PROTOCOL_MOUSE)
115.    }, { }
116.};
117.
118./* 定义并初始化usb_driver */
119.static struct usb_driver ax_usb_driver = {
120.    .name      = "ax_usb_test",
121.    .probe     = ax_usb_probe,
122.    .disconnect = ax_usb_disconnect,
123.    .id_table   = ax_usb_id_table,
124.};
125.
126./* 驱动入口函数 */
127.static int ax_usb_init(void)
128.{{
129.    /* 注册usb_driver */
130.    return usb_register(&ax_usb_driver);
131.}
132.
133./* 驱动出口函数 */
134.static void ax_usb_exit(void)
135.{{
136.    /* 注销usb_driver */
137.    usb_deregister(&ax_usb_driver);
138.}
139.
140./* 标记加载、卸载函数 */
141.module_init(ax_usb_init);
142.module_exit(ax_usb_exit);
143.
144./* 驱动描述信息 */
145.MODULE_AUTHOR("Alinx");
146.MODULE_ALIAS("pwm_led");
147.MODULE_DESCRIPTION("USB TEST driver");
148.MODULE_VERSION("v1.0");
149.MODULE_LICENSE("GPL");
```

The framework of **usb\_driver** is very simple. The key part is the processing in the **probe** function and **urb** callback function.

Combined with the **input** subsystem, the usb mouse is simulated

as a key.

**Line 56**, the **probe** function gets the interface from **intf**, and then gets the endpoint from the interface.

**Line 70** gets the pipeline.

**Line 73~79** allocate buffer.

These are all preparing for the allocation of urb.

**Lines 79~87**, Create **urb** and use the parameters obtained above to allocate **urb**, and submit it after completion.

**usb\_fill\_int\_urb** registers the terminal to input endpoint data. When the input device **usb** mouse moves, the terminal function **ax\_usb\_irq** will be triggered.

**Lines 25~38** are used in the terminal function to determine the **input** type through the input subsystem and print the corresponding information.

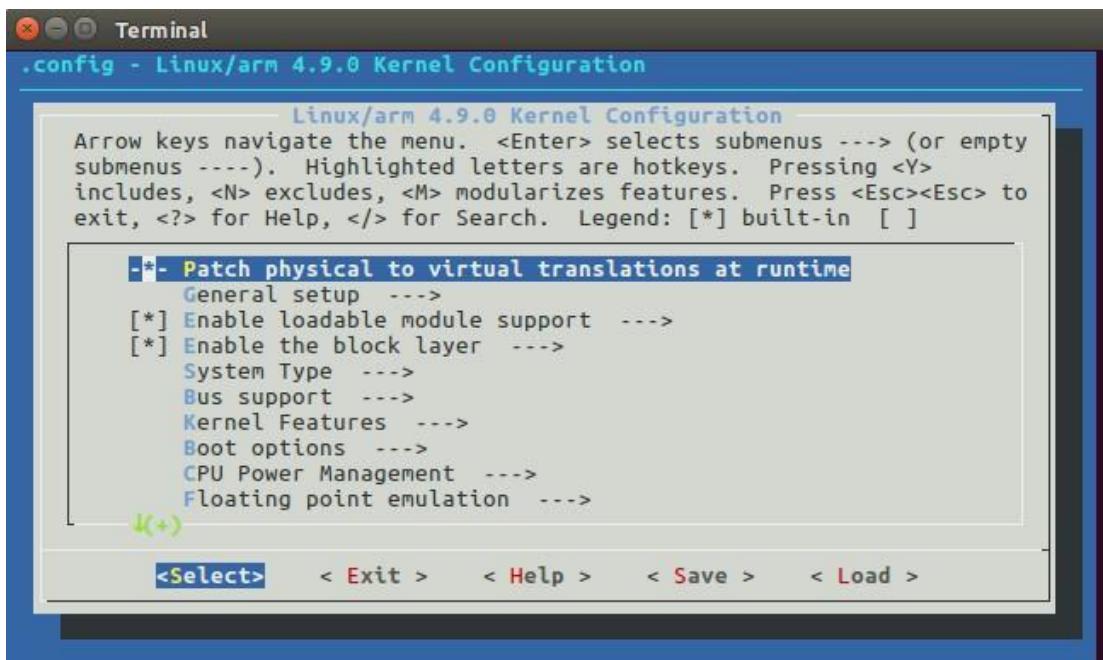
#### Part 18.6.4: Run Test

The program of this experiment is loaded with a mouse, which will conflict with the original one in the system, so you need to remove the system's own mouse driver first. Methods as below:

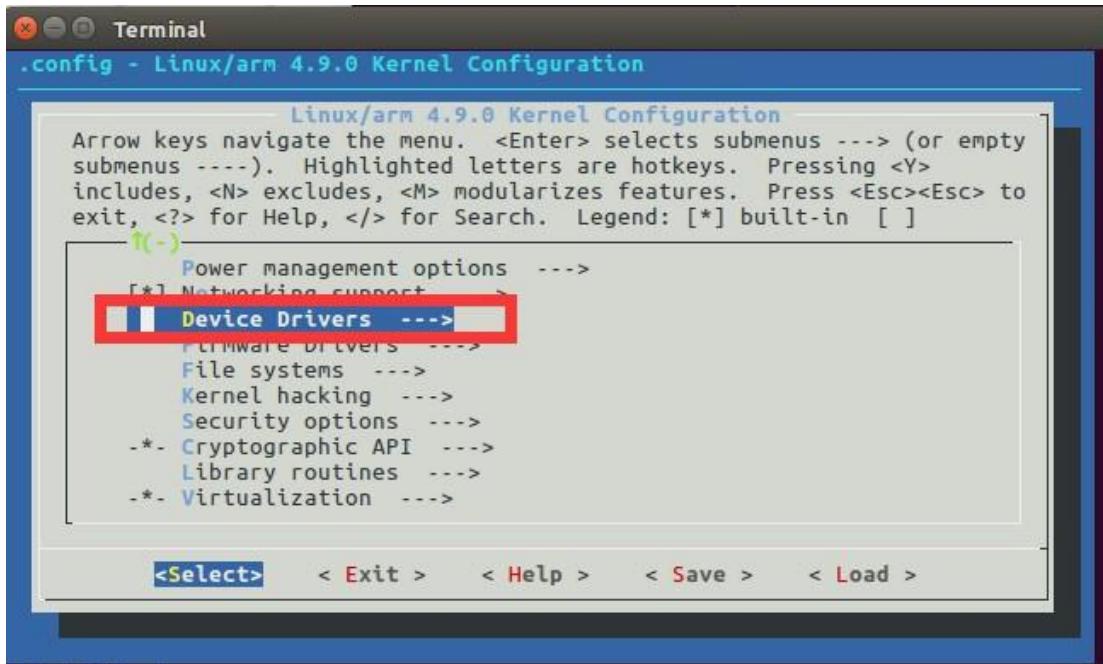
- 1) Enter the command to configure the kernel **petalinux-config -c kernel** in the terminal, and the configuration interface will pop up as follows:

```
alinx@ubuntu:~/Downloads/peta_pri_pwm/ax_peta$ petalinux-config -c kernel
[INFO] generating Kconfig for project

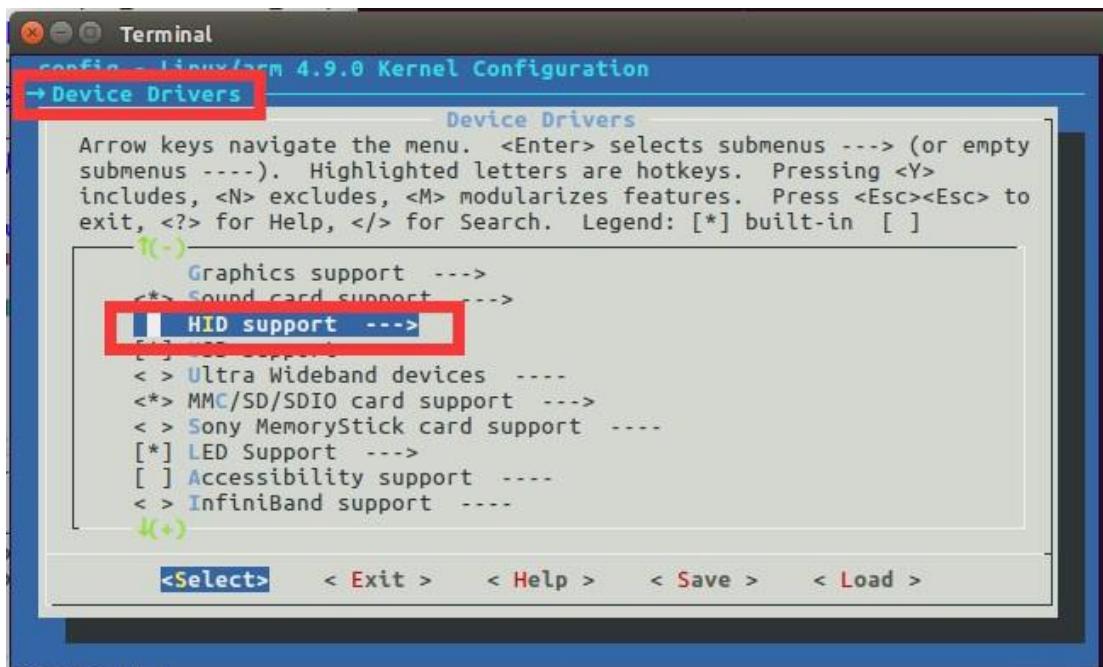
[INFO] sourcing bitbake
[INFO] generating plnxtool conf
[INFO] generating meta-plnx-generated layer
~/Downloads/peta_pri_pwm/ax_peta/build/misc/plnx-generated ~/Downloads/peta_pri_
pwm/ax_peta
~/Downloads/peta_pri_pwm/ax_peta
[INFO] generating machine configuration
[INFO] configuring: kernel
[INFO] generating kernel configuration files
[INFO] bitbake virtual/kernel -c menuconfig
Loading cache: 100% |#####| Time: 0:00:00
Loaded 3278 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:01
Parsing of 2487 .bb files complete (2455 cached, 32 parsed). 3280 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#####| Time: 0:00:04
NOTE: Executing RunQueue Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and al
l succeeded.
Loading cache: 100% |#####| Time: 0:00:00
Loaded 3278 entries from dependency cache.
Parsing recipes: 100% |#####| Time: 0:00:01
Parsing of 2487 .bb files complete (2455 cached, 32 parsed). 3280 targets, 226 s
kipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#####| Time: 0:00:04
NOTE: Executing SetScene Tasks
NOTE: Executing RunQueue Tasks
Currently 1 running tasks (318 of 318) 99% |#####
0: linux-xlnx-4.9-xilinx-v2017.4+gitAUTOINC+b450e900fd-r0 do_menuconfig - 24s (p
id 122206)
```



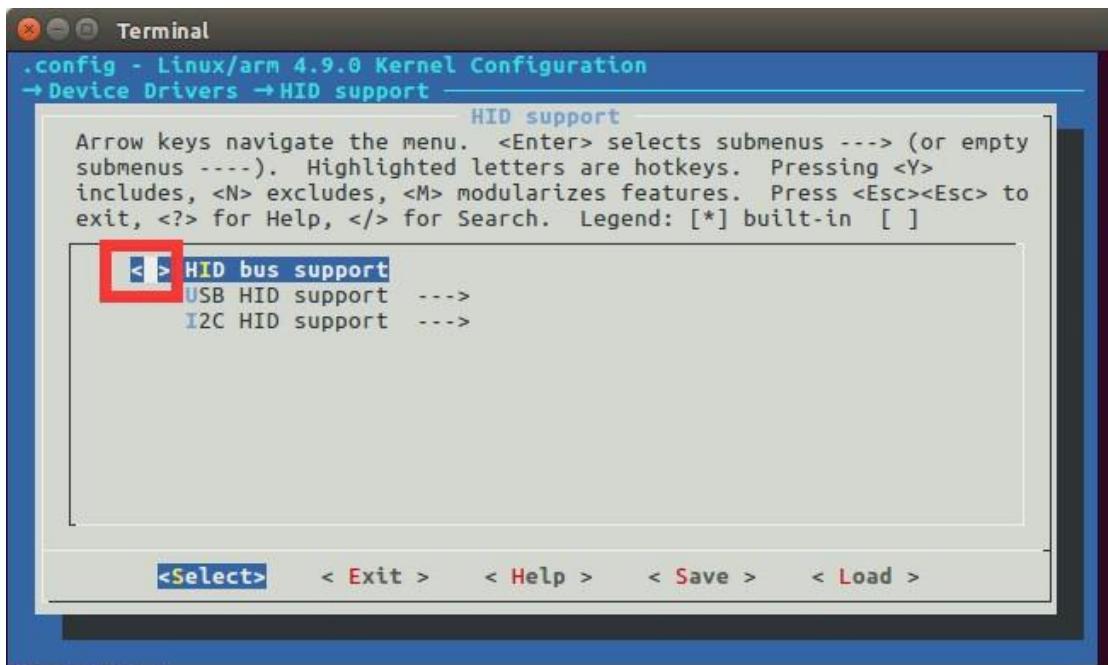
- 2) Press Enter to enter the **Device Drivers** sub-option in the configuration interface



### 3) Enter the **HID support** sub-option



### 4) After entering, press the space to turn the **HID bus support** option to an empty state, and then enter the **USB HID support** sub-option, Also adjust the <> to empty selection state.



After that, the steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-usb-drv.ko
```

The IP and path are adjusted according to the actual situation.

Then connect the usb mouse. Click the left button and the right button to view the console output. as follows:

```
insmod ./ax-usb-drv.ko
usbcore: registered new interface driver ax_usb_test
root@ax_peta:/mnt# usb 1-1: new low-speed USB device number 29 using ci_hdrc
input: Unspecified device as /devices/virtual/input/input27
[drm] load() is defered & will be called again
lf click
lf click
lf click
lf click
lf click
lf click
rt click
```

After unplugging the mouse, the disconnect message will be printed.

```
rt click
rt click
usb 1-1: USB disconnect, device number 29
```

## Part 19: SPI Drive

Like usb and i2c, the SPI driver framework is also divided into bus (controller) driver and device driver. The controller driver is provided by the kernel and we don't need to worry about it. The focus is on the implementation of the device driver.

### Part 19.1: SPI Controller Drive

Get an overview of the **SPI** controller bus. The **spi\_master** structure is used in the kernel to represent an SPI host controller, which is defined in **Line 402** of the file **include/linux/spi/spi.h**. Compare with the i2c bus frame, there are also data transfer functions **transfer**, **transfer\_one\_message**, etc. in **spi\_master**.

The difference of **transfer\_one\_message** is that the data is packed into **spi\_message** and transmitted in the form of a queue. These functions are provided by the chip manufacturer, we just need to understand a little.

#### 1) **spi\_master** application and release

After defining the **spi\_master** structure variable, use the following method to apply to the kernel:

```
struct spi_master *spi_alloc_master(struct device *dev, unsigned size)
```

**dev** is generally the **dev** member variable in **platform\_device**.

**size** is the size of private data. Private data can be obtained through function **spi\_master\_get\_devdata**. The application is successfully returned to **spi\_master**.

To release **spi\_master** use the following function:

```
void spi_master_put(struct spi_master *master)
```

#### 2) **spi\_master** registration and cancellation

After **spi\_master** is initialized, use the following function to

register:

```
int spi_register_master(struct spi_master *master)
```

Use the following function to log out:

```
void spi_unregister_master(struct spi_master *master)
```

## Part 19.2: SPI Device Drive

### 1) spi\_driver

The **spi\_driver** structure is used in the Linux kernel to represent the **spi** device driver. The **spi\_driver** structure is defined in the file **include/linux/spi/spi.h** as follows:

```
1. struct spi_driver {
2.     const struct spi_device_id *id_table;
3.     int (*probe)(struct spi_device *spi);
4.     int (*remove)(struct spi_device *spi);
5.     void (*shutdown)(struct spi_device *spi);
6.     struct device_driver driver;
7. };
```

**id\_table** is used to match **spi** devices. If the match succeeds, the **probe** function will be executed. When the device is removed, the **remove** function is executed.

After **spi\_driver** is initialized, use the following function to register with the kernel:

```
int spi_register_driver(struct spi_driver *sdrv)
```

Use the following function to log off **spi\_driver**:

```
void spi_unregister_driver(struct spi_driver *sdrv)
```

The **spi\_driver** framework example is as follows:

```
1. static int ax_probe(struct spi_device *spi)
2. {
3.     return 0;
4. }
5.
6. static int ax_remove(struct spi_device *spi)
7. {
8.     return 0;
9. }
10.
11. static const struct spi_device_id ax_id[] = {
12.     {"xxx", 0},
13.     { }
14. };
15.
```

```
16. static const struct of_device_id ax_of_match[] = {
17.     { .compatible = "xxx" },
18.     { }
19. };
20.
21. static struct spi_driver ax_driver = {
22.     .probe = ax_probe,
23.     .remove = ax_remove,
24.     .driver = {
25.         .owner = THIS_MODULE,
26.         .name = "xxx",
27.         .of_match_table = ax_of_match,
28.     },
29.     .id_table = ax_id,
30. };
31.
32. static int init ax_init(void)
33. {
34.     return spi_register_driver(&ax_driver);
35. }
36.
37. static void exit ax_exit(void)
38. {
39.     spi_unregister_driver(&ax_driver);
40. }
41.
42. module_init(ax_init);
43. module_exit(ax_exit);
```

To write a spi device driver, to implement two functions **probe** and **remove**.

## 2) spi device

The **spi\_device** structure is used in the kernel to represent **spi** devices. After the introduction of the device tree, **spi\_device** is rarely used. Focus on the description of **spi** device methods in the device tree. For the device tree node format of **spi** devices, please refer to the description in the file **Documentation\devicetree\bindings\mtd**.

Examples are as follows:

```
1. qspi: spi@e000d000 {
2.     clock-names = "ref_clk", "pclk";
3.     clocks = <&cclk 10>, <&cclk 43>;
4.     compatible = "xlnx,zynq-qspi-1.0";
5.     status = "disabled";
6.     interrupt-parent = <&intc>;
7.     interrupts = <0 19 4>;
8.     reg = <0xe000d000 0x1000>;
9.     #address-cells = <1>;
10.    #size-cells = <0>;
11.
12.    flash: w25q256@0
13.    {
14.        #address-cells = <1>;
15.        #size-cells = <1>;
16.        compatible = "w25q256";
17.        reg = <0>;
18.        spi-max-frequency = <40000000>;
19.        m25p,fast-read;
```

```
20.      };
21.  };
```

**Line 1**, `qspi` is one of the `spi` buses on `zynq`, that is, the bus node.

**Lines 2~10** are the attributes of the `spi` bus node. This part is provided by xilinx. The hardware-related information can correspond to the register manual

**Line 12**, the device node, it is a device connected to the spi bus of qspi. Flash is another name, and this node is a flash chip. w25q256 is the node name, and the `@0` behind it means that the device is connected to channel 0 of the spi bus.

**Line 16**, `compatible` is an important attribute used when the driver and the device are matched, and its value needs to be consistent with the `compatible` field in `spi_driver`. This is the same as encountered in the previous bus frame.

**Line 17**, the `reg` attribute is the same as after `@`, which means the `spi` channel.

**Line 18**, the `spi-max-frequency` attribute setting is the highest frequency of `spi`, where the frequency is `20Mhz`.

**Line 19**, `fast-read` means that this device supports fast collection. According to the actual situation, remove it if it does not support it.

### 3) Matching of drive and device

This part is also very similar to `i2c`. The matching of the driver and the device is completed in the bus (controller) driver.

The `spi` bus is defined as the structure `spi_bus_type`, in the file `drivers/spi/spi.c`, as follows:

```
1.  ruct bus_type spi_bus_type = {
2.      .name = "spi",
3.      .dev_groups = spi_dev_groups,
4.      .match = spi_match_device,
5.      .uevent = spi_uevent,
6.  };
```

The **match** function in the kernel is implemented as the function **spi\_match\_device()**, as follows:

```
1. static int spi_match_device(struct device *dev, struct device_driver *drv)
2. {
3.     const struct spi_device *spi = to_spi_device(dev);
4.     const struct spi_driver *sdrv = to_spi_driver(drv);
5.
6.     /* Attempt an OF style match */
7.     if (of_driver_match_device(dev, drv))
8.         return 1;
9.
10.    /* Then try ACPI */
11.    if (acpi_driver_match_device(dev, drv))
12.        return 1;
13.
14.    if (sdrv->id_table)
15.
16.        return !spi_match_id(sdrv->id_table, spi);
17.
18.    return strcmp(spi->modalias, drv->name) == 0;
19. }
```

There are four matching methods: **device tree**, **ACPI**, **id\_table** and **name**, similar to the previous bus driver framework.

### Part 19.3:

Some structures and functions are needed to transmit and receive data in **spi** device driver

#### 1) spi\_transfer

**spi\_transfer** is used to describe the transfer information of **spi**, which is defined as follows:

```
1. struct spi_transfer {
2.     /* it's ok if tx_buf == rx_buf (right?)
3.      * for MicroWire, one buffer must be null
4.      * buffers must work with dma_*map_single() calls, unless
5.      *   spi_message.is_dma_mapped reports a pre-existing mapping
6.      */
7.     const void *tx_buf;
8.     void *rx_buf;
9.     unsigned len;
10.
11.    dma_addr_t tx_dma;
12.    dma_addr_t rx_dma;
13.    struct sg_table tx_sg;
14.    struct sg_table rx_sg;
15.
16.    unsigned cs_change:1;
17.    unsigned tx_nb_bits:3;
18.    unsigned rx_nb_bits:3;
19. #define SPI_NBITS_SINGLE 0x01 /* 1bit transfer */
20. #define SPI_NBITS_DUAL 0x02 /* 2bits transfer */
21. #define SPI_NBITS_QUAD 0x04 /* 4bits transfer */
22.    u8 bits_per_word;
```

```
23.     u16      delay_usecs;
24.     u32      speed_hz;
25.     u32      dummy;
26.     struct list_head transfer_list;
27. };
```

**tx\_buf** and **rx\_buf** are to save the transmitting and receiving data respectively. **len** is the data length, **spi** is full-duplex communication, the length of data transmitted and received in word communication is the same, so only one **len** is enough.

## 2) spi\_message

**spi\_message** is equivalent to the transmitting queue of **spi\_transfer**, **spi\_transfer** needs to be added to **spi\_message** to transmit.

### 3) spi\_message\_init()

**spi\_message** needs to be initialized with the function **spi\_message\_init()**.

### 4) spi\_message\_add\_tail()

After **spi\_message** is initialized, use **spi\_message\_add\_tail()** to add **spi\_transfer** to **spi\_message**. The function prototype is:

```
void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m)
```

### 5) spi\_sync()

The **spi\_sync()** function uses synchronous blocking to transmit **spi\_message**. The prototype is as follows:

```
int spi_sync(struct spi_device *spi, struct spi_message *message)
```

### 6) spi\_async()

The **spi\_async()** function uses asynchronous non-blocking mode to transmit **spi\_message**, the prototype is as follows:

```
int spi_async(struct spi_device *spi, struct spi_message *message)
```

The overall steps are as follows, the specific writing refers to the experiment:

```
1. static int ax_spi_send(struct spi_device *spi, u8 *buf, int len)
2. {
3.     int ret;
```

```

4.     struct spi_message msg;
5.     struct spi_transfer trans =
6.     {
7.         .tx_buf = buf,
8.         .len = len,
9.     };
10.    spi_message_init(&msg);
11.    spi_message_add_tail(trans, &msg);
12.    ret = spi_sync(spi, &msg);
13.    return ret;
14. }

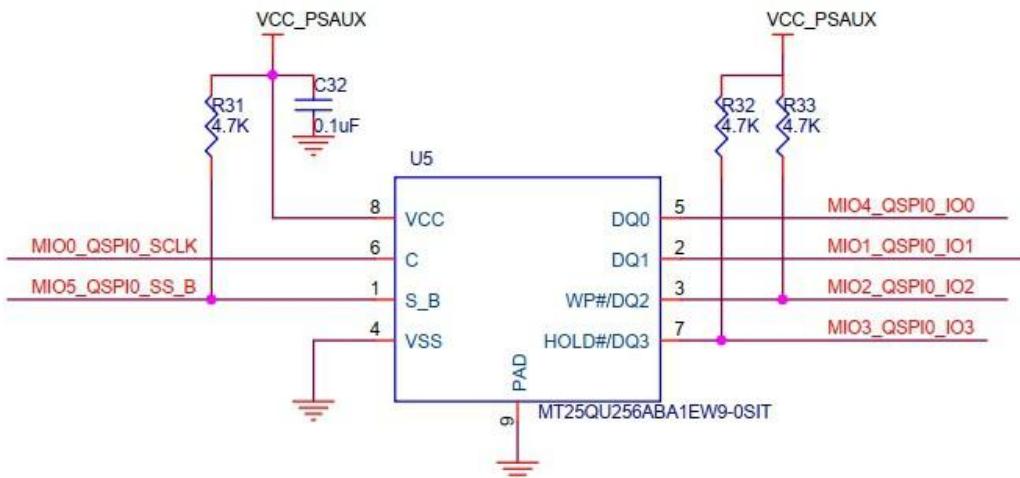
```

## Part 19.4: Experiment

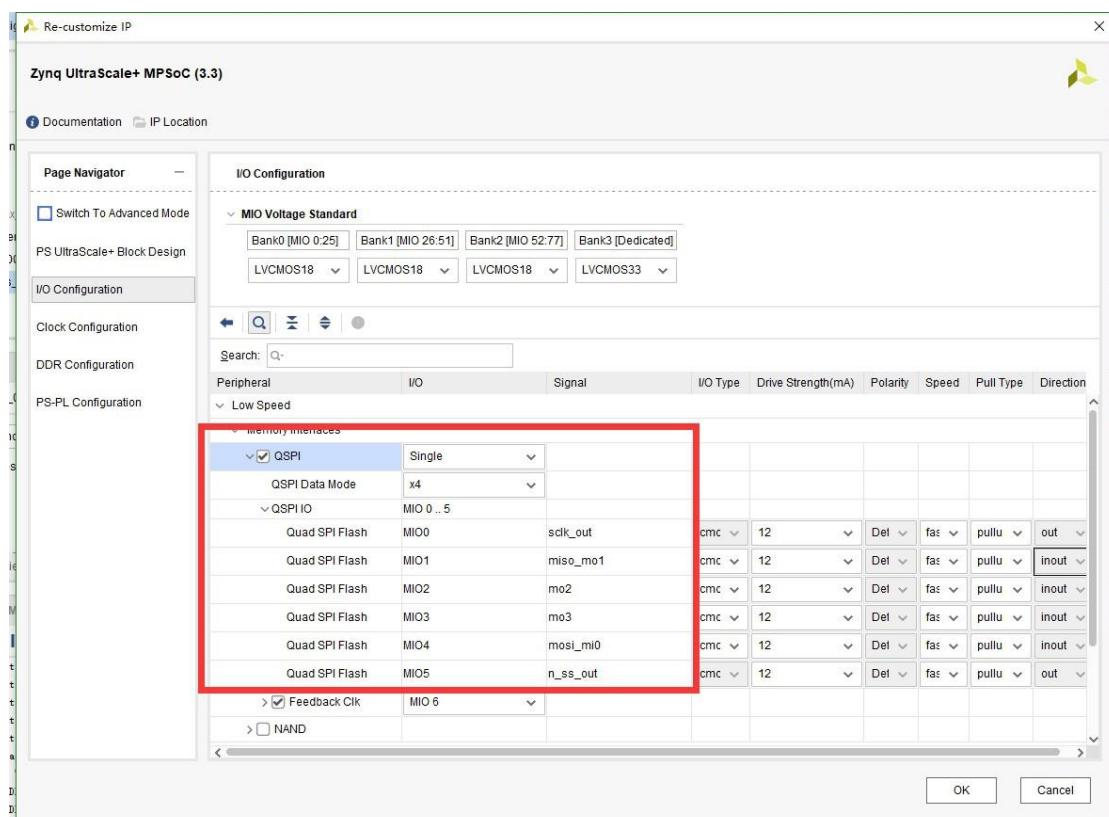
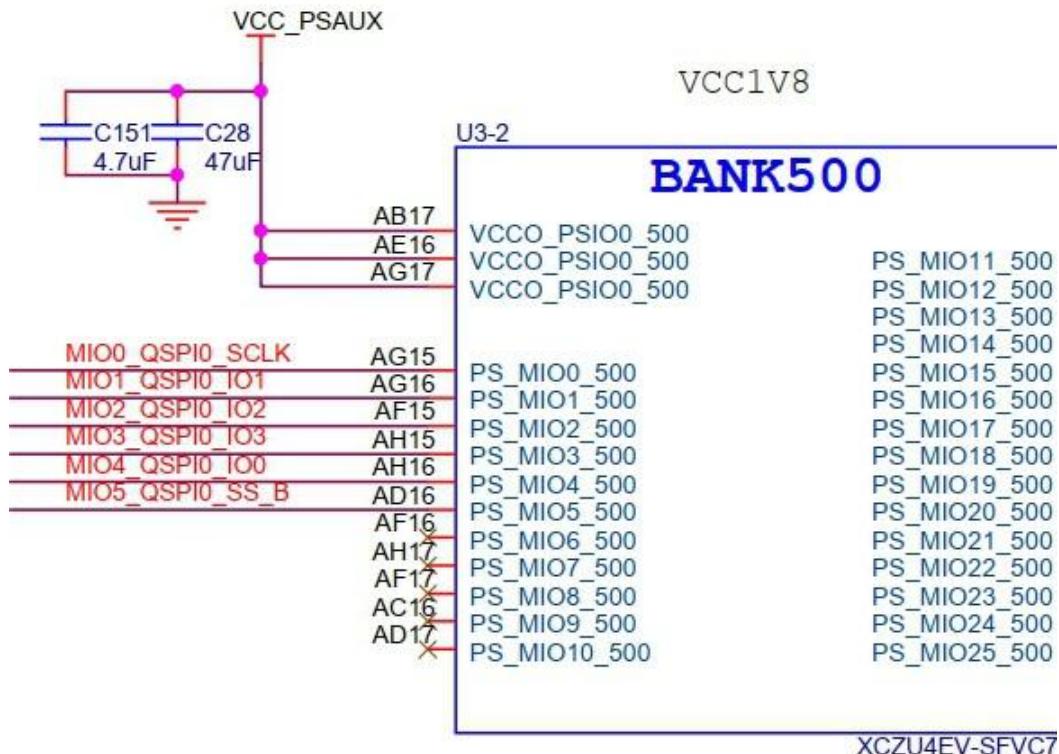
This chapter uses QSPI on zynqmp to read and write qflash. The idea is similar to the **Part 17: I2C Driver**

### Part 19.4.1: Schematic

The flash chip found in the schematic is as follows. The data line and clock line are all driven by the bus, so we don't need to worry about it. Of course, if you need to implement the bus driver yourself, you need to know which pin of the chip these lines are connected to, and what configuration needs to be made for this pin



I won't look at the specific description of the pins here. In vivado, you can also find the corresponding configuration.



### Part 19.4.2: Device tree

Open the **system-user.dtsi** file and add the following node

outside the root directory:

```
1. &qspi {
2.     status = "okay";
3.
4.     qflash: mt25qu256@0 {
5.         #address-cells = <1>;
6.         #size-cells = <1>;
7.         compatible = "mt25qu256";
8.         reg = <0>;
9.         spi-max-frequency = <40000000>;
10.        m25p,fast-read;
11.
12.    };
13.};
```

The **qflash** node already exists in the **zynq** device tree, we just need to change the **compatible** attribute.

#### Part 19.4.3: Driver code

Use **petalinux** to create a new driver named "**ax-spi-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver. Enter the following code in the **ax-spi-drv.c** file:

```
1. #include <linux/err.h>
2. #include <linux/errno.h>
3. #include <linux/device.h>
4. #include <linux/mtd/mtd.h>
5. #include <linux/mtd/partitions.h>
6. #include <linux/spi/spi.h>
7. #include <linux/spi/flash.h>
8. #include <linux/types.h>
9. #include <linux/kernel.h>
10. #include <linux/ide.h>
11. #include <linux/init.h>
12. #include <linux/module.h>
13. #include <linux/cdev.h>
14. #include <linux/fs.h>
15. #include <linux/fcntl.h>
16. #include <linux/platform_device.h>
17. #include <asm/uaccess.h>
18. #include <asm/io.h>
19. #include <linux/delay.h>
20.
21. /* 驱动个数 */
22. #define AX_FLASH_CNT 1
23. /* 设备节点名称 */
24. #define AX_FLASH_NAME "ax_flash"
25.
26. /* Flash 操作命令 */
27. #define CMD_WRITE_ENABLE 0x06
28. #define CMD_BULK_ERASE 0xc7
29. #define CMD_SECTOR_ERASE 0xD8
30. #define CMD_READ_BYTES 0x03
31. #define CMD_PAGE_PROGRAM 0x02
```

```
32. #define CMD_MAX          4
33.
34. struct ax_qflash_dev {
35.     dev_t    devid;           //设备号
36.     struct cdev cdev;        //字符设备
37.     struct class *class;     //类
38.     struct device *device;   //设备
39.     int major;              //主设备号
40.     void *private_data;      //私有数据, 获取 spi_device
41.     char cmd[CMD_MAX];       //SPI 命令和地址
42. };
43.
44. struct ax_qflash_dev ax_qflash;
45.
46. static int read_sr(struct ax_qflash_dev *flash)
47. {
48.     ssize_t retval;
49.     u8 code = 0x05;
50.     u8 val;
51.     struct spi_device *spi = (struct spi_device *)flash->private_data;
52.
53.     retval = spi_write_then_read(spi, &code, 1, &val, 1);
54.
55.     if (retval < 0)
56.     {
57.         return retval;
58.     }
59.
60.     return val;
61. }
62.
63. static int wait_till_ready(struct ax_qflash_dev *flash)
64. {
65.     int count;
66.     int sr;
67.
68.     for (count = 0; count < 100000; count++)
69.     {
70.         sr = read_sr(flash);
71.         if (!(sr & 0x00000001))
72.         {
73.             break;
74.         }
75.     }
76.
77.     return 1;
78. }
79.
80. static int ax_spi_erase(struct ax_qflash_dev *dev, loff_t addr)
81. {
82.     int ret;
83.     char cmd_buf[1] = {0};
84.     struct spi_device *spi = (struct spi_device *)dev->private_data;
85.     struct spi_transfer trans[2] = {0};
86.     struct spi_message msg;
87.     spi_message_init(&msg);
88.
89.     wait_till_ready(dev);
90.
91.     /* 写使能 */
92.     cmd_buf[0] = CMD_WRITE_ENABLE;
93.     spi_write(spi, cmd_buf, 1);
94.
95.     dev->cmd[0] = CMD_SECTOR_ERASE;
96.     dev->cmd[1] = addr >> 16;
```

```
97.     dev->cmd[2] = addr >> 8;
98.     dev->cmd[3] = addr;
99.
100.    trans[0].tx_buf = dev->cmd;
101.    trans[0].len = CMD_MAX;
102.    spi_message_add_tail(&trans[0], &msg);
103.
104.    ret = spi_sync(spi, &msg);
105.
106.    return ret;
107. }
108.
109. static int ax_spi_write(struct ax_qflash_dev *dev, loff_t addr, const char *buf, size_t len)
110. {
111.     int ret;
112.     char cmd_buf[1] = {0};
113.     struct spi_device *spi = (struct spi_device *)dev->private_data;
114.     struct spi_transfer trans[2] = {0};
115.     struct spi_message msg;
116.     spi_message_init(&msg);
117.
118.     wait_till_ready(dev);
119.
120.     /* 写使能 */
121.     cmd_buf[0] = CMD_WRITE_ENABLE;
122.     spi_write(spi, cmd_buf, 1);
123.
124.     dev->cmd[0] = CMD_PAGE_PROGRAM;
125.     dev->cmd[1] = addr >> 16;
126.     dev->cmd[2] = addr >> 8;
127.     dev->cmd[3] = addr;
128.
129.     trans[0].tx_buf = dev->cmd;
130.     trans[0].len = CMD_MAX;
131.     spi_message_add_tail(&trans[0], &msg);
132.
133.     trans[1].tx_buf = buf;
134.     trans[1].len = len;
135.     spi_message_add_tail(&trans[1], &msg);
136.
137.     ret = spi_sync(spi, &msg);
138.
139.     return ret;
140. }
141.
142. static int ax_spi_read(struct ax_qflash_dev *dev, loff_t addr, const char *buf, size_t len)
143. {
144.     int ret;
145.     struct spi_device *spi = (struct spi_device *)dev->private_data;
146.     struct spi_transfer trans[2] = {0};
147.     struct spi_message msg;
148.     spi_message_init(&msg);
149.
150.     wait_till_ready(dev);
151.
152.     dev->cmd[0] = CMD_READ_BYTES;
153.     dev->cmd[1] = addr >> 16;
154.     dev->cmd[2] = addr >> 8;
155.     dev->cmd[3] = addr;
156.
157.     trans[0].tx_buf = dev->cmd;
158.     trans[0].len = CMD_MAX;
159.     spi_message_add_tail(&trans[0], &msg);
160.
```

```
161.     trans[1].rx_buf = buf;
162.     trans[1].len = len;
163.     spi_message_add_tail(&trans[1], &msg);
164.
165.     ret = spi_sync(spi, &msg);
166.
167.     return ret;
168. }
169.
170. /* open 函数实现，对应到Linux 系统调用函数的 open 函数 */
171. static int ax_flash_open(struct inode *inode, struct file *filp)
172. {
173.     /* 设置私有数据 */
174.     filp->private_data = &ax_qflash;
175.
176.     return 0;
177. }
178.
179. /* read 函数实现，对应到Linux 系统调用函数的 read 函数 */
180. static ssize_t ax_flash_read(struct file *file, char user *buf, size_t size ,
181.                             loff_t *offset)
182. {
183.     /* 获取私有数据 */
184.     struct ax_qflash_dev *dev = (struct ax_qflash_dev *)file->private_data;
185.     /* 读取数据 buffer */
186.     char b[200] = {0};
187.     int ret = 0;
188.
189.     /* 读取数据 */
190.     ax_spi_read(dev, 0x00, b, 200 > size ? size : 200);
191.
192.     /* 把读取到的数据拷贝到用户读取的地址 */
193.     ret = copy_to_user(buf, b, 200 > size ? size : 200);
194.     return 0;
195.
196. /* write 函数实现，对应到 Linux 系统调用函数的write 函数 */
197. static ssize_t ax_flash_write(struct file *file, const char user *buf, size_t
198.                             size, loff_t *offset)
199. {
200.     /* 获取私有数据 */
201.     struct ax_qflash_dev *dev = (struct ax_qflash_dev *)file->private_data;
202.     /* 写入数据的 buffer */
203.     static char user_data[200] = {0};
204.     int ret = 0;
205.     /* 获取用户需要发送的数据 */
206.     ret = copy_from_user(user_data, buf, 200 > size ? size : 200);
207.     if(ret < 0)
208.     {
209.         printk("copy user data failed\r\n");
210.         return ret;
211.
212.     /* 写入数据 */
213.     ax_spi_erase(dev, 0x00);
214.     mdelay(1000);
215.     ax_spi_write(dev, 0x00, user_data, 200 > size ? size : 200);
216.
217.     return 0;
218. }
219.
220. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
221. static int ax_flash_release(struct inode *inode, struct file *filp)
222. {
```

```
223.     return 0;
224. }
225.
226. /* file_operations 结构体声明 */
227. static const struct file_operations ax_flash_ops = {
228.     .owner = THIS_MODULE,
229.     .open = ax_flash_open,
230.     .read = ax_flash_read,
231.     .write = ax_flash_write,
232.     .release = ax_flash_release,
233. };
234.
235. static int ax_spi_probe(struct spi_device *spi)
236. {
237.     char cmd_buf[1] = {0};
238.     printk("flash probe\r\n");
239.
240.     /* 构建设备号 */
241.     alloc_chrdev_region(&ax_qflash.devid, 0, AX_FLASH_CNT, AX_FLASH_NAME);
242.
243.     /* 注册设备 */
244.     cdev_init(&ax_qflash.cdev, &ax_flash_ops);
245.     cdev_add(&ax_qflash.cdev, ax_qflash.devid, AX_FLASH_CNT);
246.
247.     /* 创建类 */
248.     ax_qflash.class = class_create(THIS_MODULE, AX_FLASH_NAME);
249.     if(IS_ERR(ax_qflash.class))
250.     {
251.         return PTR_ERR(ax_qflash.class);
252.     }
253.
254.     /* 创建设备 */
255.     ax_qflash.device = device_create(ax_qflash.class, NULL, ax_qflash.devid, N
ULL, AX_FLASH_NAME);
256.     if(IS_ERR(ax_qflash.device))
257.     {
258.         return PTR_ERR(ax_qflash.device);
259.     }
260.
261.     ax_qflash.private_data = spi;
262.
263.     return 0;
264. }
265.
266. static int ax_spi_remove(struct spi_device *spi)
267. {
268.     /* 删除设备 */
269.     cdev_del(&ax_qflash.cdev);
270.     unregister_chrdev_region(ax_qflash.major, AX_FLASH_CNT);
271.     /* 注销类 */
272.     device_destroy(ax_qflash.class, ax_qflash.devid);
273.     class_destroy(ax_qflash.class);
274.     return 0;
275. }
276.
277. static const struct spi_device_id ax_id_table[] = {
278.     {"mt25qu256", 0},
279.     {}
280. };
281.
282. static const struct of_device_id ax_of_match[] = {
283.     { .compatible = "mt25qu256" },
284.     {}
285. };
286.
```

```
287. static struct spi_driver ax_spi_driver = {  
288.     .probe = ax_spi_probe,  
289.     .remove = ax_spi_remove,  
290.     .driver = {  
291.         .owner = THIS_MODULE,  
292.         .name = "mt25qu256",  
293.         .of_match_table = ax_of_match,  
294.     },  
295.     .id_table = ax_id_table,  
296. };  
297.  
298. static int init ax_init(void)  
299. {  
300.     return spi_register_driver(&ax_spi_driver);  
301. }  
302.  
303. static void exit ax_exit(void)  
304. {  
305.     spi_unregister_driver(&ax_spi_driver);  
306. }  
307.  
308. module_init(ax_init);  
309. module_exit(ax_exit);  
310.  
311. /* 驱动描述信息 */  
312. MODULE_AUTHOR("Alinx");  
313. MODULE_ALIAS("qspi_flash");  
314. MODULE_DESCRIPTION("I2C FLASH driver");  
315. MODULE_VERSION("v1.0");  
316. MODULE_LICENSE("GPL");
```

**Lines 27~31** are the macro definitions of flash operation commands, and the operation commands can be viewed in the flash chip manual.

**Lines 46~78** are to implement a function that waits for the flash preparation to complete. `read_sr()` is to read the status register, check the status of the flash, and then continuously read the status in the `wait_till_ready()` function, and return until the ready status is read or timeout. The register related information can be viewed in the flash chip manual.

**Line 80~168**, read and write implementation of `spi-flash`. Read and write operations need to be divided into two steps. The first step is to transmit the command and the target address of the slave, and the second is to transmit the address of the data to be sent or the buffer to read the data. Note that the transmitting buffer in the first step is `trans.tx_buf`, and in the second step, `tx_buf` is transmitted and read

as `re_buf`. The transmitting step is as mentioned before, first package `spi_transfer`, then add it to `spi_message`, and use `spi_sync` function to transmit. The erasing here is the page selection erasing, the whole chip erasing does not need two steps, just send the instruction.

**Lines 113~149** are the implementation of the read and write functions of the character device, which are basically the same as the implementation in **Part 17: I2C Driver**. So the following test can also directly use the test program in **Part 17: I2C Driver**.

**Lines 180~218** are `probe` implementations, which will be executed after successful matching. The **Lines 241~259** in the `probe` first register the character device. The **Line 261** assigns `spi` to the private variable of the device structure.

After that is the realization of the matching method. Compared with the bus framework so far, except for the different types, everything else is the same.

#### Part 19.4.4: Test code

You can use the test code in Part 17 directly.

#### Part 19.4.5: Run Test

The test method steps are as follows:

```
mount -t nfs -o nolock 192.168.1.107:/home/ilinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-spi-drv.ko
cd ./build-ax_i2c_test-ZYNQ-Debug/
./ax_i2c_test /dev/ax_flash
```

The IP and path are adjusted according to the actual situation.

The debugging results in the serial port tool are as follows:

```
root@ax_peta:~# mount -t nfs -o noblock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-spi-drv.ko
ax_spi_drv: loading out-of-tree module taints kernel.
flash probe
ax_qflash node find
Manufacture ID: 0xef
Device ID: 0x4019
[drm] load() is deferred & will be called again
root@ax_peta:/mnt# cd ./build-ax_i2c_test-ZYNQ-Debug/
root@ax_peta:/mnt/build-ax_i2c_test-ZYNQ-Debug# ./ax_i2c_test /dev/ax_flash
write data 5A, 55, AA
read data 5A, 55, AA
root@ax_peta:/mnt/build-ax_i2c_test-ZYNQ-Debug# 
```

The reading and writing results are consistent and the test is successful.

## Part 20: Uart Driver

The serial port is a commonly used peripheral. It is often used for communication with other MCUs, 2G sensors, GPS and other modules. The Linux system is no exception. We use the serial port tool to interact with the FPGA development board, which undoubtedly uses the serial port. There are many level standards for serial ports, such as TTL, 232, 485, etc., but their drivers are all the same. In the embedded Linux system, the serial port is regarded as a terminal device ([tty](#)), including three structures: [uart\\_driver](#), [uart\\_port](#), [uart\\_ops](#) are defined in the file [include/serial\\_core.h](#). To implement a uart driver, you only need to implement these three structures.

The driver of uart is generally provided by the chip manufacturer. The uart driver provided by xilinx is in [drivers/tty/serial/xilinx\\_uartps.c](#). As long as we add the corresponding serial port node in the device tree, the serial port peripherals can be used. In this chapter, we will learn about the serial port driver framework in Linux and understand the implementation of xilinx serial port driver.

### Part 20.1: uart Driver Framework

#### 1) [usb\\_driver](#)

Use [usb\\_driver](#) in the Linux kernel to represent, [uart\\_driver](#) contains information such as serial device name, serial driver name, major and minor device numbers, and includes [tty\\_driver](#), so that the underlying serial driver does not need to care about [tty\\_driver](#).

[uart\\_driver](#) is defined in the file [include/linux/serial\\_core.h](#), as follows:

```
1. struct uart_driver {  
2.     struct module    *owner;  
3.     const char      *driver_name;
```

```

4.   const char      *dev_name;
5.   int             major;
6.   int             minor;
7.   int             nr;
8.   struct console  *cons;
9.
10.  /*
11.   * these are private; the low level driver should not
12.   * touch these; they should be initialised to NULL
13.   */
14.  struct uart_state *state;
15.  struct tty_driver *tty_driver;
16. };

```

**owner** is generally **THIS\_MODULE**.

**driver\_name** is the driver name.

**dev\_name** is the device name.

**cons** is the console.

After defining **uart\_driver**, use your function below to register with the kernel:

```
int uart_register_driver(struct uart_driver *drv)
```

**drv**: **uart\_driver** to be registered.

Return value: 0 success, negative value failure.

Relative unregister **uart\_driver** uses the following function:

```
void uart_unregister_driver(struct uart_driver *drv)
```

**drv** is the **uart\_driver** to be unregistered.

## 2) uart\_prot

**uart\_port** represents a specific uart port, which is used to describe the I/O port or I/O memory address, FIFO size, port type and other information of a uart port. Defined in the file **include/linux/serial\_core.h**, as follows:

```

117. struct uart_port {
118.   spinlock_t      lock;          /* port lock */
119.   unsigned long    iobase;        /* in/out[bwl] */
120.   unsigned char_  iomem *membase; /* read/write[bwl] */
121.   unsigned int     (*serial_in)(struct uart_port *, int);
122.   void            (*serial_out)(struct uart_port *, int, int);
123.   void            (*set_termios)(struct uart_port *,
124.                                 struct ktermios *new,
125.                                 struct ktermios *old);
126.   unsigned int     (*get_mctrl)(struct uart_port *);
127.   void            (*set_mctrl)(struct uart_port *, unsigned int);
128.   int             (*startup)(struct uart_port *port);
129.   void            (*shutdown)(struct uart_port *port);
130.   .....

```

```

237. const struct uart_ops    *ops;
238. unsigned int           custom_divisor;
239. unsigned int           line;          /* port index */
240. unsigned int           minor;
241. resource_size_t        mapbase;       /* for ioremap */
242. resource_size_t        mapsize;
243. struct device          *dev;          /* parent device */
244. unsigned char          hub6;          /* this should be in the 8250 driver */
245. unsigned char          suspended;
246. unsigned char          irq_wake;
247. unsigned char          unused[2];
248. struct attribute_group *attr_group;   /* port specific attributes */
249. const struct attribute_group **tty_groups; /* all attributes (serial core use only) */
250. struct serial_rs485    rs485;
251. void                  *private_data;  /* generic platform data pointer */
252.};

```

The most important one is the **ops** serial port driver operation function set. After **uart\_port** is defined, it needs to be associated with **uart\_driver**. Use the following function to add **uart\_port** to **uart\_driver**:

```
int uart_add_one_port(struct uart_driver *drv, struct uart_port *uport)
```

**drv:** Register the **uart\_driver** corresponding to the target **uart\_port**.

**uport:** **uart\_port** to be added to **uart\_driver**.

**Return value:** 0 success; negative value failure.

Relatively use the following function to delete **uart\_port** from **uart\_driver**:

```
int uart_remove_one_port(struct uart_driver *drv, struct uart_port *uport)
```

**drv:** Uninstall the **uart\_driver** corresponding to the target **uart\_port**.

**uport:** **uart\_port** to be uninstalled.

**Return value:** 0 success; negative value failure.

### 3) **uart\_ops**

**uart\_ops** is an important member of **uart\_port**, and is a collection of **uart** specific driver functions. The kernel uses the serial port to transmit and receive data and ultimately calls the function in **ops**. **uart\_ops** is defined in the file **include/linux/serial\_core.h**, as follows:

```

1. struct uart_ops {
2.     unsigned int (*tx_empty)(struct uart_port *);
3.     void (*set_mctrl)(struct uart_port *, unsigned int mctrl);
4.     unsigned int (*get_mctrl)(struct uart_port *);
5.     void (*stop_tx)(struct uart_port *);
6.     void (*start_tx)(struct uart_port *);
7.     void (*throttle)(struct uart_port *);
8.     void (*unthrottle)(struct uart_port *);
9.     void (*send_xchar)(struct uart_port *, char ch);
10.    void (*stop_rx)(struct uart_port *);
11.    void (*enable_ms)(struct uart_port *);
12.    void (*break_ctl)(struct uart_port *, int ctl);
13.    int (*startup)(struct uart_port *);
14.    void (*shutdown)(struct uart_port *);
15.    void (*flush_buffer)(struct uart_port *);
16.    void (*set_termios)(struct uart_port *, struct ktermios *new,
17.                        struct ktermios *old);
18.    void (*set_ldisc)(struct uart_port *, struct ktermios *);
19.    void (*pm)(struct uart_port *, unsigned int state,
20.                unsigned int oldstate);
21.
22. /*
23.  * Return a string describing the type of the port
24.  */
25. const char *(*type)(struct uart_port *);
26.
27. /*
28.  * Release IO and memory resources used by the port.
29.  * This includes iounmap if necessary.
30.  */
31. void (*release_port)(struct uart_port *);
32.
33. /*
34.  * Request IO and memory resources used by the port.
35.  * This includes iomapping the port if necessary.
36.  */
37. int (*request_port)(struct uart_port *);
38. void (*config_port)(struct uart_port *, int);
39. int (*verify_port)(struct uart_port *, struct serial_struct *);
40. int (*ioctl)(struct uart_port *, unsigned int, unsigned long);
41. #ifdef CONFIG_CONSOLE_POLL
42. int (*poll_init)(struct uart_port *);
43. void (*poll_put_char)(struct uart_port *, unsigned char);
44. int (*poll_get_char)(struct uart_port *);
45. #endif
46. };

```

The functions in `uart_ops` need to be implemented by the underlying developers and are the part that directly deals with registers. For the specific meaning of these functions in the `uart_ops` structure, please refer to the [documentation/serial/driver](#).

In summary, the main tasks to be completed by a serial port driver are:

- 1) Define and initialize structure variables such as `uart_driver`, `uart_ops`, `uart_port`, etc.
- 2) When the driver module is loaded, use `uart_register_driver()` and `uart_add_one_port()` to register `uart_driver` and add ports. When

the driver module is unloaded, use the functions `uart_unregister_driver()` and `uart_remove_one_port()` to unregister `uart_driver` and remove the port.

- 3) Implement the member functions in `uart_ops` according to the datasheet of the specific hardware. Then correspond to the specific driver code. Take a look at the `uart` implementation of `xilinx`.

## Part 20.2: uart Driver in xilinx

First look at the node description of `uart` in the device tree and open the file `zynqmp.dtsi`. Find the `uart` related nodes as follows:

```

1. uart0: serial@ff000000 {
2.     u-boot,dm-pre-reloc;
3.     compatible = "cdns,uart-r1p12", "xlnx,xuartps";
4.     status = "disabled";
5.     interrupt-parent = <&gic>;
6.     interrupts = <0 21 4>;
7.     reg = <0x0 0xff000000 0x0 0x1000>;
8.     clock-names = "uart_clk", "pclk";
9.     power-domains = <&zynqmp_firmware PD_UART_0>;
10. };
11.
12. uart1: serial@ff010000 {
13.     u-boot,dm-pre-reloc;
14.     compatible = "cdns,uart-r1p12", "xlnx,xuartps";
15.     status = "disabled";
16.     interrupt-parent = <&gic>;
17.     interrupts = <0 22 4>;
18.     reg = <0x0 0xff010000 0x0 0x1000>;
19.     clock-names = "uart_clk", "pclk";
20.     power-domains = <&zynqmp_firmware PD_UART_1>;
21. };

```

The two `uart` nodes in the node are both `ps` serial ports, and the `ps` side of `zynqmp` has only two serial ports. If you need more, you need to use `pl` side resources.

First find the corresponding driver code according to the `compatible` property, which is the file `drivers/tty/serial/xilinx_uartps.c`. Where `of_device_id` is as follows:

```

1. /* Match table for of_platform binding */
2. static const struct of_device_id cdns_uart_of_match[] = {
3.     { .compatible = "xlnx,xuartps", },
4.     { .compatible = "cdns,uart-r1p8", },
5.     { .compatible = "cdns,uart-r1p12", .data = &zynqmp_uart_def },

```

```

6.     { .compatible = "xlnx,zynqmp-uart", .data = &zynqmp_uart_def },
7.     {}
8. };

```

In Line 1614 of the driver code [xilinx\\_uartps.c](#), you will find the following code:

```

1. static struct platform_driver cdns_uart_platform_driver = {
2.     .probe    = cdns_uart_probe,
3.     .remove   = cdns_uart_remove,
4.     .driver   = {
5.         .name = CDNS_UART_NAME,
6.         .of_match_table = cdns_uart_of_match,
7.         .pm = &cdns_uart_dev_pm_ops,
8.     },
9. };

```

It can be seen that **uart** is essentially a **platform** driver.

Then correspond to this driver code according to the **uart** framework mentioned earlier.

### 1) uart\_driver

```

1. static struct uart_driver cdns_uart_uart_driver = {
2.     .owner      = THIS_MODULE,
3.     .driver_name = CDNS_UART_NAME,
4.     .dev_name   = CDNS_UART_TTY_NAME,
5.     .major      = CDNS_UART_MAJOR,
6.     .minor      = CDNS_UART_MINOR,
7.     .nr         = CDNS_UART_NR_PORTS,
8. #ifdef CONFIG_SERIAL_XILINX_PS_UART_CONSOLE
9.     .cons       = &cdns_uart_console,
10. #endif
11. };
12. ....
13. static int __init cdns_uart_init(void)
14. {
15.     int retval = 0;
16.
17.     /* Register the cdns_uart driver with the serial core */
18.     retval = uart_register_driver(&cdns_uart_uart_driver);
19.     if (retval)
20.         return retval;
21.     ....
22.     return retval;
23. }
24.
25. static void __exit cdns_uart_exit(void)
26. {
27.     /* Unregister the platform driver */
28.     platform_driver_unregister(&cdns_uart_platform_driver);
29.     ....
30. }

```

The variable **cdns\_uart\_uart\_driver** of **uart\_driver** type can be found and initialized.

After that, there are corresponding registration and unregister

methods in the driver entry function and exit function.

## 2) uart port

```

1. static struct uart_port cdns_uart_port[CDNS_UART_NR_PORTS];
2.
3. /**
4. * cdns_uart_get_port - Configure the port from platform device resource info
5. * @id: Port id
6. *
7. * Return: a pointer to a uart_port or NULL for failure
8. */
9. static struct uart_port *cdns_uart_get_port(int id)
10. {
11.     struct uart_port *port;
12.
13.     /* Try the given port id if failed use default method */
14.     if (cdns_uart_port[id].mapbase != 0) {
15.         /* Find the next unused port */
16.         for (id = 0; id < CDNS_UART_NR_PORTS; id++)
17.             if (cdns_uart_port[id].mapbase == 0)
18.                 break;
19.     }
20.
21.     if (id >= CDNS_UART_NR_PORTS)
22.         return NULL;
23.
24.     port = &cdns_uart_port[id];
25.
26.     /* At this point, we've got an empty uart_port struct, initialize it */
27.     spin_lock_init(&port->lock);
28.     port->membase = NULL;
29.     port->irq = 0;
30.     port->type = PORT_UNKNOWN;
31.     port->iotype = UPIO_MEM32;
32.     port->flags = UPF_BOOT_AUTOCONF;
33.     port->ops = &cdns_uart_ops;
34.     port->fifosize = CDNS_UART_FIFO_SIZE;
35.     port->line = id;
36.     port->dev = NULL;
37.     return port;
38. }
39.
40. static int cdns_uart_probe(struct platform_device *pdev)
41. {
42.     int rc, id, irq;
43.     struct uart_port *port;
44.     struct resource *res;
45.     struct cdns_uart *cdns_uart_data;
46.     const struct of_device_id *match;
47.     .....
48.     /* Look for a serialN alias */
49.     id = of_alias_get_id(pdev->dev.of_node, "serial");
50.     if (id < 0)
51.         id = 0;
52.
53.     /* Initialize the port structure */
54.     port = cdns_uart_get_port(id);
55.
56.     if (!port) {
57.         dev_err(&pdev->dev, "Cannot get uart_port structure\n");
58.         rc = -ENODEV;
59.         goto err_out_notif_unreg;
60.     }

```

```
61.      /*
62.       * Register the port.
63.       * This function also registers this device with the tty layer
64.       * and triggers invocation of the config_port() entry point.
65.       */
66.
67.     port->mapbase = res->start;
68.     port->irq = irq;
69.     port->dev = &pdev->dev;
70.     port->uartclk = clk_get_rate(cdns_uart_data->uartclk);
71.     port->private_data = cdns_uart_data;
72.     cdns_uart_data->port = port;
73.     platform_set_drvdata(pdev, port);
74.
75.     ....
76.     rc = uart_add_one_port(&cdns_uart_uart_driver, port);
77.     if (rc) {
78.         dev_err(&pdev->dev,
79.                 "uart_add_one_port() failed; err=%i\n", rc);
80.         goto err_out_pm_disable;
81.     }
82.
83.     ....
84. }
85.
```

In the program, you can find the above code snippet. First, the **uart\_port** type array **cdns\_uart\_port[CDNS\_UART\_NR\_PORTS]** is defined.

The function **cdns\_uart\_get\_port** is implemented to initialize **uart\_port**.

In Line 73 of the **probe** function (Line 1554 in the actual source code), add **uart\_port** to the private data of the device driver structure so that it can be called when the **ops** function is implemented later.

Also on Line 76 of the **probe** function (Line 1561 of the actual source code), call **uart\_add\_one\_port** to add **uart\_port** to **uart\_driver**.

### 3) uart\_ops

The **uart\_ops** variable is defined on Line 1081, named **cdns\_uart\_ops**, as follows:

```
1. static const struct uart_ops cdns_uart_ops = {
2.     .set_mctrl = cdns_uart_set_mctrl,
3.     .get_mctrl = cdns_uart_get_mctrl,
4.     .start_tx = cdns_uart_start_tx,
5.     .stop_tx = cdns_uart_stop_tx,
6.     .stop_rx = cdns_uart_stop_rx,
```

```
7.     .tx_empty    = cdns_uart_tx_empty,
8.     .break_ctl   = cdns_uart_break_ctl,
9.     .set_termios  = cdns_uart_set_termios,
10.    .startup     = cdns_uart_startup,
11.    .shutdown    = cdns_uart_shutdown,
12.    .pm          = cdns_uart_pm,
13.    .type         = cdns_uart_type,
14.    .verify_port  = cdns_uart_verify_port,
15.    .request_port = cdns_uart_request_port,
16.    .release_port = cdns_uart_release_port,
17.    .config_port  = cdns_uart_config_port,
18. #ifdef CONFIG_CONSOLE_POLL
19.    .poll_get_char = cdns_uart_poll_get_char,
20.    .poll_put_char = cdns_uart_poll_put_char,
21. #endif
22. };
```

In the `uart_port` initialization function on **Line 1112**, the `ops` is assigned to `cdns_uart_ops`.

The functions in `cdns_uart_ops` are the specific driver functions of `uart`.

## Part 21: Block Device Driver

Linux devices are divided into character devices, block devices, and network devices. The frameworks introduced above are all combined with character devices for experiments, now let's talk about block devices.

### Part 21.1: Introduction to Block Devices

The block device is distinguished from the character device must have his particularity.

Let's go back to the character device first. In the previous character device experiment, the character device framework can almost guarantee universality, and the few differences are the implementation of the operation functions in **file\_operations**. Since versatility can be guaranteed, why introduce the concept of block devices?

The most prominent feature of a block device compared to a character device is that it needs to support batch data read and write operations. Of course, multiple calls to read and write character devices can also achieve the ultimate goal, but specific hardware operations will encounter several problems:

- 1) Some block devices such as flash need to erase the current sector before writing
- 2) The mechanical structure of some block devices needs to be read and written in address order to achieve the highest efficiency

Therefore, the block device driver should have these functions:

- 1) There is a buffer for reading sector data
- 2) There is a read and write operation queue, and the read and write sequence can be optimized by address, and the introduction of the

block device framework is to deal with these problems.

## Part 21.2: Block device access in the kernel

From the application layer to the hardware, the access process of the block device is as follows:

- 1) The **app** of the application layer calls **read** and other methods to manipulate files such as **txt**, etc.
- 2) Through the file system **ext2**, **ext3**, etc., the operation of the file is converted into the read and write to the sector
- 3) Call the corresponding operation function in the block device driver
- 4) Operating hardware

The difference with character devices is that there is an extra file system in the middle. In this step, there is a key function **ll\_rw\_block()**. From this function, it enters the device level.

The **ll\_rw\_block()** function is defined in the file **fs/buffer.c**. as follows:

```
1. void ll_rw_block(int op, int op_flags, int nr, struct buffer_head *bhs[])
2. {
3.     int i;
4.
5.     for (i = 0; i < nr; i++) {
6.         struct buffer_head *bh = bhs[i];
7.
8.         if (!trylock_buffer(bh))
9.             continue;
10.        if (op == WRITE) {
11.            if (test_clear_buffer_dirty(bh)) {
12.                bh->b_end_io = end_buffer_write_sync;
13.                get_bh(bh);
14.                submit_bh(op, op_flags, bh);
15.                continue;
16.            }
17.        } else {
18.            if (!buffer_uptodate(bh)) {
19.                bh->b_end_io = end_buffer_read_sync;
20.                get_bh(bh);
21.                submit_bh(op, op_flags, bh);
22.                continue;
23.            }
24.        }
}
```

```

25.         unlock_buffer(bh);
26.     }
27. }
```

The **buffer\_head** structure in the input parameter is the descriptor of the buffer, storing buffer information, defined as follows

```

1. struct buffer_head {
2.     unsigned long b_state;           //缓冲区状态
3.     struct buffer_head *b_this_page; //页缓冲区头
4.     struct page *b_page;           //存储缓冲区所在页
5.     sector_t b_blocknr;            //块号
6.     size_t b_size;                //块大小
7.     char *b_data;                 //页数据缓冲区
8.
9.     struct block_device *b_bdev;    //块设备
10.
11.    bh_end_io_t *b_end_io;         //I/O 完成后执行函数指针
12.
13.    void *b_private;              //b_end_io 的输入参数
14.
15.    struct list_head b_assoc_buffers; //映射链表
16.
17. /* mapping this buffer is associated with */
18.    struct address_space *b_assoc_map;
19.    atomic_t b_count;              //缓冲区计数
20.};
```

Look at the final operation on **lines 14 and 21** in the **ll\_rw\_block()** function, the **submit\_bh()** function. **submit\_bh()** is a simplified encapsulation of the function **submit\_bh\_wbc()**, **submit\_bh\_wbc()** is defined as follows (original notes are omitted):

```

1. int submit_bh(int rw, struct buffer_head * bh)
2. {
3.     struct bio *bio;                  //定义一个
4.     bio(block input output),也就是块设备i/o
5.     BUG_ON(!buffer_locked(bh));
6.     BUG_ON(!buffer_mapped(bh));
7.     BUG_ON(!bh->b_end_io);
8.     BUG_ON(buffer_delay(bh));
9.     BUG_ON(buffer_unwritten(bh));
10.
11.    if (test_set_buffer_req(bh) && (op == REQ_OP_WRITE))
12.        clear_buffer_write_io_error(bh);
13.    bio = bio_alloc(GFP_NOIO, 1);      //分配 bio
14.
15.    if (wbc) {
16.        wbc_init_bio(wbc, bio);
```

```
17.         wbc_account_io(wbc, bh->b_page, bh->b_size);
18.     }
19.
20.     /* 构造 bio */
21.     bio->bi_sector = bh->b_blocknr * (bh->b_size >> 9); //存放逻辑块号
22.     bio->bi_bdev = bh->b_bdev;                                //存放对应的块设备
23.
24.     bio_add_page(bio, bh->b_page, bh->b_size, bh_offset(bh));
25.     BUG_ON(bio->bi_iter.bi_size != bh->b_size);
26.
27.     bio->bi_end_io = end_bio_bh_io_sync;                         //设置 i/o 回调函数
28.     bio->bi_private = bh;                                       //回调函数指针，指向哪个缓冲区
29.     bio->bi_flags |= bio_flags;
30.
31.     guard_bio_eod(op, bio);
32.
33.     if (buffer_meta(bh))
34.         op_flags |= REQ_META;
35.     if (buffer_prio(bh))
36.         op_flags |= REQ_PRIO;
37.     bio_set_op_attrs(bio, op, op_flags);
38.
39.     submit_bio(bio);                                            //提交 bio
40.     return 0;
41. }
```

The work here is mainly to use `buffer_head` to create a `bio`, `bio` is block `input/output`, block device input and output. Finally, the `submit_bio()` function is called to submit the `bio` to the kernel queue.

Looking further down, the `submit_bio()` function finally calls `generic_make_request()`, then calls `_generic_make_request()`, then to `_make_request()`, the `elv_merge()` function in the `_make_request()` function is the elevator merge in the kernel. Elevator merge realizes the optimization of the read and write order according to the address order we mentioned above.

### Part 21.3: Block Device Framework

The character device framework can refer to the character device driver code used in the kernel `drivers\block\z2ram.c` or others.

The general steps are as follows:

- 1) Allocate a **gendisk** structure
- 2) Allocate a queue **request\_queue\_t**
- 3) Set the members of the **gendisk** structure
- 4) Register **gendisk** structure

### Part 21.3.1: gendisk structure

The **gendisk** structure is defined in **genhd.h**, as follows

```
1. struct gendisk {  
2.     /* major, first_minor and minors are input parameters only,  
3.      * don't use directly. Use disk_devt() and disk_max_parts().  
4.      */  
5.     int major;           /* major number of driver */  
6.     int first_minor;  
7.     int minors;          /* maximum number of minors, =1 for  
8.                           * disks that can't be partitioned */  
9.  
10.    char disk_name[DISK_NAME_LEN]; /* name of major driver */  
11.    char *(*devnode)(struct gendisk *gd, umode_t *mode);  
12.  
13.    unsigned int events;        /* supported events */  
14.    unsigned int async_events; /* async events, subset of all */  
15.  
16.    /* Array of pointers to partitions indexed by partno.  
17.       * Protected with matching bdev lock but stat and other  
18.       * non-critical accesses use RCU. Always access through  
19.       * helpers.  
20.      */  
21.    struct disk_part_tbl_rCU *part_tbl;  
22.    struct hd_struct part0;  
23.  
24.    const struct block_device_operations *fops;  
25.    struct request_queue *queue;  
26.    void *private_data;  
27.  
28.    int flags;  
29.    struct kobject *slave_dir;  
30.  
31.    struct timer_rand_state *random;  
32.    atomic_t sync_io;          /* RAID */  
33.    struct disk_events *ev;  
34. #ifdef CONFIG_BLK_DEV_INTEGRITY  
35.    struct kobject integrity_kobj;  
36. #endif /* CONFIG_BLK_DEV_INTEGRITY */  
37.    int node_id;  
38.    struct badblocks *bb;  
39.};
```

Introduction of some members:

**major:** The major device number of the device.

---

**first\_minor**: the starting minor device number.

**minors**: The number of minor device numbers, also known as the number of partitions. The value of 1 means that partitions are not possible.

**disk\_name**: device name.

**part**: Partition table information.

**fops**: The collection of block device operations. The operation function of a block device is quite different from that of a character device. If you encounter it, you will study it.

**queue**: Application queue, used to manage the pointer of the device IO application queue.

**private\_data**: Private data.

**capacity**: the number of sectors.

The operation of **gendisk** is roughly as follows

- 1) In the block device driver, first define a **gendisk** structure variable, use the function **struct gendisk \*alloc\_disk (int minors)** in the entry function to allocate space, and **minors** is the number of partitions.
- 2) Assign queue.
- 3) Use the **register\_blkdev** function to apply to the kernel for the major device number of the block device.
- 4) Use the function **set\_capacity()** to set the number of sectors.
- 5) Set other parameters.
- 6) Use the function **add\_disk()** to register **gendisk** with the kernel.
- 7) The opposite operation is implemented in the exit function:

Use **put\_disk()** and **del\_gendisk()** to unregister and release the **gendisk** structure.

Use **kfree()** to release the disk sector cache.

Use **blk\_cleanup\_queue()** to clear the application queue in the

memory.

Use `unregister_blkdev()` to unload the block device.

### Part 21.3.2: request\_queue

First define a `request_queue` structure variable. Use the following function to allocate:

```
struct request_queue *blk_init_queue(request_fn_proc *rfn, spinlock_t *lock)
```

`rfn` is the queue processing function.

`lock` is a spin lock.

Use the `blk_init_queue` function to get the `request_queue` and copy it to `ramblock_disk.queue`. The implementation of the `rfn` function here is the focus of the block device.

The use of the block device framework is explained in the experimental routine.

## Part 21.4: Experiment

The experiment in this chapter simulates the memory as a block device to operate, focusing on familiarizing with the block device framework and the process of writing block device drivers.

### Part 21.4.1: Driver code

Use `petalinux` to create a new driver named "`ax-block-drv`", and execute the `petalinux-config -c rootfs` command to select the new driver.

Enter the following code in the `ax-block-drv.c` file:

```
1. #include <linux/module.h>
2. #include <linux/errno.h>
3. #include <linux/interrupt.h>
4. #include <linux/mm.h>
5. #include <linux/fs.h>
6. #include <linux/kernel.h>
7. #include <linux/timer.h>
8. #include <linux/genhd.h>
9. #include <linux/hdreg.h>
```

```
10. #include <linux/ioport.h>
11. #include <linux/init.h>
12. #include <linux/wait.h>
13. #include <linux/blkdev.h>
14. #include <linux/blkpg.h>
15. #include <linux/delay.h>
16. #include <linux/io.h>
17. #include <asm/mach/map.h>
18. #include <asm/uaccess.h>
19. #include <asm/dma.h>
20.
21. #define AX_BLOCK_SIZE      (1024*64)           //块设备大小
22. #define SECTOR_SIZE        512                 //扇区大小
23. #define SECTORS_NUM         AX_BLOCK_SIZE / SECTOR_SIZE //扇区数
24.
25. #define AX_BLOCK_NAME       "ax_block"          //设备节点名称
26.
27. #define AX_BLOCK_MAJOR      40
28. struct ax_block{
29.     struct gendisk          *block_disk;    //磁盘结构体
30.     struct request_queue     *block_request; //申请队列
31.     unsigned char            *block_buf;     //磁盘地址
32. };
33. /* 声明设备结构体变量 */
34. struct ax_block ax_block_drv;
35. /* 定义一个自旋锁 */
36. static DEFINE_SPINLOCK(ax_block_lock);
37.
38. /* 块设备操作函数集，即使没有操作函数，也需要赋值 */
39. static struct block_device_operations ax_block_fops = {
40.     .owner   = THIS_MODULE,
41. };
42.
43. /* 队列处理函数 */
44. static void ax_block_request(struct request_queue * q)
45. {
46.     struct request *req;
47.
48.     /* 轮询队列中的每个申请 */
49.     req = blk_fetch_request(q);
50.     while(req)
51.     {
52.         unsigned long start;
53.         unsigned long len;
54.         void *buffer = bio_data(req->bio);
55.         /* 获取首地址 */
56.         start = (int)blk_rq_pos(req) * SECTOR_SIZE + ax_block_drv.block_
buf;
57.         /* 获取长度 */
58.         len = blk_rq_cur_bytes(req);
59.
60.         if(rq_data_dir(req) == READ)
61.         {
62.             printk("ax_request read\n");
63.             memcpy(buffer, (char *)start, len);
64.         }
65.         else
66.         {
```

```
67.         printk("ax_request write\n");
68.     }
69.     memcpy((char *)start, buffer, len);
70.     /* 处理完的申请出列 */
71.     if (!blk_end_request_cur(req, 0))
72.         req = blk_fetch_request(q);
73. }
74. }
75.
76. static int init ax_block_init(void)
77. {
78.     /* 分配块设备 */
79.     ax_block_drv.block_disk = alloc_disk(1);
80.     /* 分配申请队列，提供队列处理函数 */
81.     ax_block_drv.block_request = blk_init_queue(ax_block_request, &ax_block_lock);
82.     /* 设置申请队列 */
83.     ax_block_drv.block_disk->queue = ax_block_drv.block_request;
84.     /* 向内核注册块设备 */
85.     register_blkdev(AX_BLOCK_MAJOR, AX_BLOCK_NAME);
86.     /* 主设备号赋给块设备得主设备号字段 */
87.     ax_block_drv.block_disk->major = AX_BLOCK_MAJOR;
88.     /* 设置其他参数 */
89.     ax_block_drv.block_disk->first_minor = 0;
90.     ax_block_drv.block_disk->fops = &ax_block_fops;
91.     sprintf(ax_block_drv.block_disk->disk_name, AX_BLOCK_NAME);
92.     /* 设置扇区数 */
93.     set_capacity(ax_block_drv.block_disk, SECTORS_NUM);
94.     /* 获取缓存，把内存模拟成块设备 */
95.     ax_block_drv.block_buf = kzalloc(AX_BLOCK_SIZE, GFP_KERNEL);
96.     /* 向内核注册 gendisk 结构体 */
97.     add_disk(ax_block_drv.block_disk);
98.     return 0;
99. }
100.
101. static void exit ax_block_exit(void)
102. {
103.     /* 注销 gendisk 结构体 */
104.     put_disk(ax_block_drv.block_disk);
105.     /* 释放 gendisk 结构体 */
106.     del_gendisk(ax_block_drv.block_disk);
107.     /* 释放缓存 */
108.     kfree(ax_block_drv.block_buf);
109.     /* 清空内存中的队列 */
110.     blk_cleanup_queue(ax_block_drv.block_request);
111.     /* 卸载快设备 */
112.     unregister_blkdev(AX_BLOCK_MAJOR, AX_BLOCK_NAME);
113. }
114.
115. module_init(ax_block_init);
116. module_exit(ax_block_exit);
117.
118. /* 驱动描述信息 */
119. MODULE_AUTHOR("Alinx");
120. MODULE_ALIAS("block test");
121. MODULE_DESCRIPTION("BLOCK driver");
122. MODULE_VERSION("v1.0");
```

```
|123 MODULE_LICENSE("GPL");
```

Looking at the block device framework first, the structure of the three parts of the entry function, exit function and operation function is the same as the character device framework. One more is in the driver entry function starting from **Line 76**:

**Line 79**, first use the function `alloc_disk` to allocate block devices.

**Line 81**, use the `blk_init_queue` function to allocate the application queue and provide the queue processing function `ax_block_request`. The implementation of `ax_block_request` function will look at it later.

**Line 83**, set the allocated queue to the member of the block device structure.

**Line 85**, to register the block device with the kernel, you need to attach the device number and device name.

**Line 87** sets the device number

**Line 89** sets the starting minor device number as the starting sector number

**Line 90** sets the operation function set

**Line 91** sets the device name.

**Line 93** sets the number of sectors

**Line 95** uses the `kalloc` function to get the cache to simulate block devices.

**Line 97** uses the `add_disk` function to register the gendisk structure with the kernel.

The reverse unregister operation is done in the driver exit function starting at **Line 101**.

Then look back at the content of the queue processing function on **Line 44**.

The content is not much, you can see that the final processing is Line 63 and Line 68 **memcpy**, copy the read or write content to the **buffer** or memory address. Of course, the **memcpy** method can be used directly because of the use of memory. In actual use of peripheral block devices, the method of reading and writing will be more complicated.

The previous step of **memcpy** is to judge whether this operation is read or write, call the function **rq\_data\_dir** function, if the application is read, it will return **READ**, and the function parameter is a pointer of **struct request** type. Note that in our queue processing function, the input parameter is a pointer of type **struct request\_queue**, which is the application queue. Use the function **blk\_fetch\_request** function to get the first application in the queue from the application queue, such as the **Line 49** operation. The obtained application is the object we will deal with later

There are three key parameters of **memcpy**, data source, data target, and data length. These must be obtained by applying for structure variable pointer **struct request \*req**.

**Line 54** gets the data destination address through **bio\_data(req->bio)**.

**Line 56** first uses **blk\_rq\_pos(req)** to get the current sector, then multiplies it by the sector size, plus the first address of the cache, and it is the first address where the current application data is located.

**Line 58**, use the method **blk\_rq\_cur\_bytes(req)** to get the data length of the current application.

In the queue processing function, each request in the queue needs to be polled. The method is to use the function **blk\_fetch\_request(q)**. If there is a request, it returns a positive value. Finally, after processing, use the **blk\_end\_request\_cur(req, 0)**

---

function to move the processed application out of the queue.

In addition, add `printf` before reading and writing to see the effect.

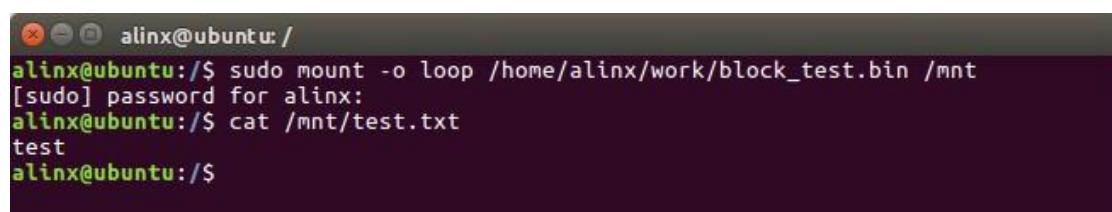
#### Part 21.4.2: Run Test

The steps are as follows:

```
mount -t nfs -o noblock 192.168.1.107:/home/ilinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-block-drv.ko
mkdosfs /dev/ax_block
cd /
mkdir test
mount /dev/ax_block /test/
cd /test/
vi test.txt
(Edit any content, here I entered test)
cd /
umount /test/
cat /dev/ax_block > /mnt/block_test.bin
```

The IP and path are adjusted according to the actual situation.

Then go to the **ubuntu** virtual machine and verify **block\_test.bin**, the steps are as follows:



A screenshot of a terminal window titled "alinx@ubuntu: /". The terminal shows the following command sequence:  
alinx@ubuntu:/\$ sudo mount -o loop /home/ilinx/work/block\_test.bin /mnt  
[sudo] password for alinx:  
alinx@ubuntu:/\$ cat /mnt/test.txt  
test  
alinx@ubuntu:/\$

Mount **block\_test.bin** to the **/mnt** directory and find that there is

the **test.txt** file we created before in the **mnt** directory, and the content is also the content entered before.

The debugging results in the serial port tool are as follows:

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-block-drv.ko
ax_block_drv: loading out-of-tree module taints kernel.
root@ax_peta:/mnt# mkdosfs /dev/ax_block
ax_request read
ax_request read
ax_request write
ax_request write
root@ax_peta:/mnt# cd /
root@ax_peta:# mkdir test
root@ax_peta:# mount /dev/ax_block /test/
ax_request read
root@ax_peta:# cd /test/
root@ax_peta:/test# vi test.txt
root@ax_peta:/test# ax_request write
ax_request write
ax_request write
ax_request write
root@ax_peta:/test# ax_request write
ax_request write

root@ax_peta:/test# cd /
root@ax_peta:# umount /test/
ax_request write
root@ax_peta:# cat /dev/ax_block > /mnt/block_test.bin
ax_request read
root@ax_peta:/#
```

You can also see the result of the elevator algorithm in the block device through the **read** and **write** information of the serial port. The reading and writing of each application will not be interleaved, which improves the efficiency of reading and writing.

In addition, the read and write of the block device is an asynchronous operation. To ensure that each requested operation is completed, you can use the sync command to complete the currently unfinished asynchronous operation.

## Part 22: NIC driver

### Part 22.1: Linux Network Card Driver

Note that the NIC driver mentioned here is not a network driver. The network driver is quite complicated. We know that the OSI seven-layer network model from top to bottom is: application layer, presentation layer, session layer, transport layer, network layer, data link layer, and physical layer, while the TCP/IP four-layer model used in Linux is: application layer , Transport layer, Internet layer, network interface layer.

OSI	Linux TCP/IP
Application Layer	Application Layer
Presentation Layer	
Session Layer	
Transport Layer	Transport Layer
Network Layer	Network Layer
Data Link Layer	Network Interface Layer
Physical Layer	

The network card driver is included in the network driver and belongs to the network interface layer, including the data link layer and the physical layer. It is also the only hardware-related part of the network driver. The others are pure software concepts. For the development of the underlying driver, you only need to care about the hardware-related content, so here we will learn about the network card driver.

The network card driver needs to implement two parts, one is to interact with the upper layer to transmit and receive data, and the other is to process data in combination with specific hardware. The

network interface layer can be subdivided into four layers: network protocol interface layer, network device interface layer, device driver function layer, network device and media layer. The network protocol interface layer is responsible for transmitting or receiving data to the upper layer, the device driver layer is responsible for sending or receiving data to the hardware, the network device interface layer plays the role of data acceptance in the middle, and the network device layer is responsible for the actual data receiving and sending operations.

The key here is the network device interface layer. Linux uses the `net_device` structure to describe a network device information, and summarizes and unifies the different hardware.

From a functional point of view, there are three functions that the network card device needs to achieve, initialization, sending and receiving. We take the `net_device` structure as the entry point and take the three functions that need to be implemented as a guide to look at the implementation of the Linux network card driver.

#### Part 22.1.1: Initialization

The `net_device` structure represents a network card. The initialization of the network card actually means to define a `net_device` structure variable and initialize this variable. First look at the specific definition of the `net_device` structure, in the file `include/linux/netdevice.h`, as follows (with omissions and changes):

```
1. struct net_device {  
2.     char          name[IFNAMSIZ]; //网卡设备名称  
3.     struct hlist_node  name_hlist;  
4.     char          *ifalias;  
5.  
6.     unsigned long    mem_end;   //该设备的内存结束地址  
7.     unsigned long    mem_start; //该设备的内存起始地址  
8.     unsigned long    base_addr; //该设备的内存I/O 基地址  
9.     int             irq;      //该设备的中断号  
10.
```

```

11.     atomic_t      carrier_changes;
12.
13.     unsigned long    state;      //网络设备和网络适配器的状态信息
14.
15.     .....
16.
17.     struct net_device_stats stats; //用来保存统计信息的net_device_stats 结构体
18.
19.     .....
20.
21.     const struct net_device_ops *netdev_ops;
22.                               //网络设备操作函数
23.
24.     .....
25.     unsigned int      flags;      //flags 指网络接口标志,常见取值如下
26.                               //IFF_UP:设备被激活并可以开始发送数据包时设置
27.                               //IFF_AUTOMEDIA:设置设备可在多种媒介间切换
28.                               //IFF_BROADCAST:允许广播
29.                               //IFF_DEBUG:调试模式
30.                               //IFF_LOOPBACK:回环
31.                               //IFF_NOARP:无可执行 ARP,点对点接口就不需要运行
32.                               ARP
33.                               //IFF_POINTOPOINT:点到点链路
34.     .....
35.     unsigned int      mtu;       //最大数据包
36.     unsigned short    type;      //接口的硬件类型
37.     unsigned short    hard_header_len;
38.                               //硬件帧头长度,一般被赋为 ETH_HLEN 值为 14
39.
40.     .....
41.     unsigned long    last_rx;    //接收数据包的时间戳,调用 netif_rx() 后赋上
42.                               jiffies 即可
43.     unsigned char    *dev_addr;  //MAC 地址
44.     .....
45. };

```

In addition to some basic information names and addresses, the most critical member is the pointer variable `netdev_ops` of type `struct net_device_ops`, which contains the set of network device operation functions, and the transmit package function we need to implement are also included. Take a look at his specific definition , Also in this file, as follows (with omissions):

```

1. struct net_device_ops {
2.
3.     .....
4.     int      (*ndo_open)(struct net_device *dev);
5.     int      (*ndo_stop)(struct net_device *dev);
6.     netdev_tx_t   (*ndo_start_xmit)(struct sk_buff *skb,
7.                                     struct net_device *dev);
8. };

```

`ndo_open` is used to open the network device, `ndo_stop` is used to close the network device.

The `ndo_start_xmit` function is the key transmit package function.

The `struct net_device_stats` member in the `net_device` structure, the `net_device_stats` structure is defined as follows:

```
1. struct net_device_stats {
2.     unsigned long rx_packets;
3.     unsigned long tx_packets;
4.     unsigned long rx_bytes;
5.     unsigned long tx_bytes;
6.     unsigned long rx_errors;
7.     unsigned long tx_errors;
8.     unsigned long rx_dropped;
9.     unsigned long tx_dropped;
10.    unsigned long multicast;
11.    unsigned long collisions;
12.    unsigned long rx_length_errors;
13.    unsigned long rx_over_errors;
14.    unsigned long rx_crc_errors;
15.    unsigned long rx_frame_errors;
16.    unsigned long rx_fifo_errors;
17.    unsigned long rx_missed_errors;
18.    unsigned long tx_aborted_errors;
19.    unsigned long tx_carrier_errors;
20.    unsigned long tx_fifo_errors;
21.    unsigned long tx_heartbeat_errors;
22.    unsigned long tx_window_errors;
23.    unsigned long rx_compressed;
24.    unsigned long tx_compressed;
25.};
```

In fact, it is a collection of statistical information of the network status. We will operate this `stats` in the transceiver processing function to record the status.

It can be seen that the `net_device` structure variable is the key to the network card device. The initialization of the network card device is the initialization of the `net_device` structure variable. The steps are as follows:

- 1) Define a `net_device` structure variable
  - 2) Use the function `alloc_netdev()` to allocate a `net_device` structure variable
  - 3) Initialize hardware registers
-

- 4) Set **net\_device** structure variable member variable
- 5) Use **register\_netdev()** to register **net\_device** structure variables

**alloc\_netdev()** is a macro definition, as follows:

```
alloc_netdev(sizeof_priv, name, name_assign_type, setup)
```

The first parameter **sizeof\_priv** is the size of the private variable.

The second parameter **name** is the device name.

The third parameter **name\_assign\_type** is the source name of the device.

The fourth parameter, **setup**, is the pointer to the **setup()** function of **net\_device**, and the parameter received by the **setup()** function is **struct**. The **net\_device** pointer is used to preset the value of the **net\_device** member.

The only input parameter of **register\_netdev()** is the pointer of the **net\_device** structure variable that needs to be registered.

### Part 22.1.2: Transmit Packet

Transmit Packet function **ndo\_start\_xmit** needs to be built by ourselves, the prototype is as follows:

```
netdev_tx_t (*ndo_start_xmit)(struct sk_buff *skb, struct net_device *dev);
```

The first input parameter **struct sk\_buff** is only a **socket** buffer, used for data transfer between layers in the network model. **struct sk\_buff** is defined in the file **include/linux/skbuff.h** and is a doubly linked list, as follows (with omissions):

```
1. struct sk_buff {  
2.     union {  
3.         struct {  
4.             /* These two members must be first. */  
5.             struct sk_buff    *next; //指向下一个sk_buff 结构体  
6.             struct sk_buff    *prev; //指向前一个sk_buff 结构体  
7.             ....  
8.         };  
9.         ....  
10.    };  
11.    ....  
12.    ....  
13.};
```

```

14.     unsigned int      len,      //数据包的总长度,包括线性数据和非线性数据
15.             data_len;    //非线性的数据长度
16.             __u16        mac_len,   //mac 包头长度
17.             __u16        hdr_len;
18.
19.     ....
20.
21.     __u32          priority;  //当前 sk_buff 结构体的优先级
22.
23.     ....
24.
25.     __be16         protocol;  //存放上层的协议类型,可以通过 eth_type_trans()来获
取
26.     __u16          transport_header; //传输层头部的偏移值
27.     __u16          network_header; //网络层头部的偏移值
28.     __u16          mac_header;   //MAC 数据链路层头部的偏移值
29.
30.     ....
31.
32. /* public: */
33.
34. /* These elements must be at the end, see alloc_skb() for details. */
35.     sk_buff_data_t tail;      //指向缓冲区的数据包末尾
36.     sk_buff_data_t end;       //指向缓冲区的末尾
37.     unsigned char   *head,    //指向缓冲区的协议头开始位置
38.                 *data;      //指向缓冲区的数据包开始位置
39.     ....
40. };

```

The data segment arrangement space in **sk\_buff** is as follows:



The data segment is divided into several segments

MAC header	IP head	Data Type	Data
------------	---------	-----------	------

The MAC header is represented by the structure **ethhdr**.

The IP header is represented by the structure **iphdr**. The data type is one byte.

The specific tasks to be done in the transmit package function **ndo\_start\_xmit** are as follows:

- 1) Call the function **netif\_stop\_queue()** to stop uploading data from the upper layer.
- 2) Transmit and receive data through hardware.
- 3) Call the function **dev\_kfree\_skb()** to release **sk\_buff**.

- 4) When the transmission succeeds and enters the interrupt, the statistics information is updated and the `netif_wake_queue()` function is called to make the upper layer continue to transmit data.
- 5) When sending overtime, call the `netif_wake_queue()` function in the `ndo_tx_timeout` function in `net_device_ops` to make the upper layer continue to send data.

The prototype of `dev_kfree_skb()` is as follows:

```
#define dev_kfree_skb(a)    consume_skb(a)  
void consume_skb(struct sk_buff *skb)
```

The prototype of `netif_wake_queue()` is as follows:

```
static inline void netif_wake_queue(struct net_device *dev)
```

The prototype of `netif_stop_queue()` is as follows:

```
static inline void netif_stop_queue(struct net_device *dev)
```

### Part 22.1.3: Receive Packet

The Receive packets are generally handled in interrupts. The processing process is:

- 1) Use the `alloc_skb()` function to construct a `sk_buff`.
- 2) Use `skb_reserve(sk_buff,2)` to shift the data packet in `sk_buff` by 2 bytes to free up the head space in `sk_buff`.
- 3) Read the data received on the hardware of the network device and copy the data to the data pointer of the `sk_buff` member.
- 4) Use the `eth_type_trans()` function to get the upper layer protocol, and assign the return value to the `sk_buff` member protocol.
- 5) Update the statistics, and finally use `netif_rx()` to upload `sk_buff` to the upper layer protocol.

## Part 22.2: Experiment

### Part 22.2.1: Driver code

Use **petalinux** to create a new driver named "**ax-netcard-drv**", and execute the **petalinux-config -c rootfs** command to select the new driver.

Enter the following code in the **ax-netcard-drv.c** file:

```
1. #include <linux/module.h>
2. #include <linux/kernel.h>
3. #include <linux/types.h>
4. #include <linux/fcntl.h>
5. #include <linux/ioport.h>
6. #include <linux/in.h>
7. #include <linux/skbuff.h>
8. #include <linux/string.h>
9. #include <linux/init.h>
10. #include <linux/bitops.h>
11. #include <linux/ip.h>
12. #include <linux/netdevice.h>
13. #include <linux/etherdevice.h>
14. #include <asm/io.h>
15. #include <asm/irq.h>
16.
17. /* 定义一个 net_device 结构体变量 */
18. static struct net_device *ax_net_dev;
19.
20. /* 模拟接收，构造一个虚拟的 sk_buff 上报，并更新统计信息 */
21. static void ax_net_rx(struct sk_buff *skb, struct net_device *dev)
22. {
23.     unsigned char *type;
24.     struct iphdr *ih;
25.     _be32 *saddr, *daddr, tmp;
26.     unsigned char tmp_dev_addr[ETH_ALEN];
27.     struct ethhdr *ethhdr;
28.
29.     struct sk_buff *rx_skb;
30.
31.     /* 交换接受和发送方的 mac 地址 */
32.     ethhdr = (struct ethhdr *)skb->data;
33.     memcpy(tmp_dev_addr, ethhdr->h_dest, ETH_ALEN);
34.     memcpy(ethhdr->h_dest, ethhdr->h_source, ETH_ALEN);
35.     memcpy(ethhdr->h_source, tmp_dev_addr, ETH_ALEN);
36.
37.     /* 交换接受和发送方的 ip 地址 */
38.     ih = (struct iphdr *)(skb->data + sizeof(struct ethhdr));
39.     saddr = &ih->saddr;
40.     daddr = &ih->daddr;
41.
42.     tmp = *saddr;
43.     *saddr = *daddr;
44.     *daddr = tmp;
45.
46.     type = skb->data + sizeof(struct ethhdr) + sizeof(struct iphdr);
47.     /* 修改类型，0 表示 reply */
48.     *type = 0;
49.
```

```
50.     ih->check = 0;
51.     ih->check = ip_fast_csum((unsigned char *)ih,ih->ihl);
52.
53.     /* 构造 sk_buff */
54.     rx_skb = dev_alloc_skb(skb->len + 2);
55.     skb_reserve(rx_skb, 2);
56.     memcpy(skb_put(rx_skb, skb->len), skb->data, skb->len);
57.
58.     rx_skb->dev = dev;
59.     rx_skb->protocol = eth_type_trans(rx_skb, dev);
60.     rx_skb->ip_summed = CHECKSUM_UNNECESSARY;
61.     dev->stats.rx_packets++;
62.     dev->stats.rx_bytes += skb->len;
63.
64.     /* 提交 sk_buff */
65.     netif_rx(rx_skb);
66. }
67.
68. static netdev_tx_t ax_net_tx(struct sk_buff *skb, struct net_device *dev)
69. {
70.     static int cnt = 0;
71.
72.     /* 停止上层数据下传队列 */
73.     netif_stop_queue(dev);
74.     /* 模拟接收，以达到一个完成的发送接收过程 */
75.     ax_net_rx(skb, dev);
76.     /* 释放 skb */
77.     dev_kfree_skb (skb);
78.     /* 发送完成，恢复上层数据下传队列 */
79.     netif_wake_queue(dev);
80.     /* 更新统计信息 */
81.     dev->stats.tx_packets++;
82.     dev->stats.tx_bytes += skb->len;
83.
84.     return NETDEV_TX_OK;
85. }
86.
87. /* 网卡设备操作函数集 */
88. static const struct net_device_ops ax_netdev_ops =
89. {
90.     .ndo_start_xmit = ax_net_tx,
91. };
92.
93. /* 驱动入口函数 */
94. static int init ax_net_init(void)
95. {
96.     /* 分配 net_device 结构体 */
97.     ax_net_dev = alloc_netdev(0, "ax_net%d", NET_NAME_UNKNOWN, ether_setup);
98.     /* 设置操作函数集 */
99.     ax_net_dev->netdev_ops = &ax_netdev_ops;
100.
101.    /* 设置 MAC 地址 */
102.    ax_net_dev->dev_addr[0] = 0x0A;
103.    ax_net_dev->dev_addr[1] = 0x0B;
104.    ax_net_dev->dev_addr[2] = 0x0C;
105.    ax_net_dev->dev_addr[3] = 0x0D;
106.    ax_net_dev->dev_addr[4] = 0x0E;
107.    ax_net_dev->dev_addr[5] = 0x0F;
108.    /* 设置 ping 功能 */
109.    ax_net_dev->flags |= IFF_NOARP;
110.    ax_net_dev->features |= NETIF_F_CSUM_MASK;
111.    /* 注册网卡驱动 */
112.    register_netdev(ax_net_dev);
113.
```

```
114.     return 0;
115. }
116.
117. /* 驱动出口函数 */
118. static void exit ax_net_exit(void)
119. {
120.     unregister_netdev(ax_net_dev);
121.     free_netdev(ax_net_dev);
122. }
123.
124. module_init(ax_net_init);
125. module_exit(ax_net_exit);
126.
127. /* 驱动描述信息 */
128. MODULE_AUTHOR("Alinx");
129. MODULE_ALIAS("net card test");
130. MODULE_DESCRIPTION("NET CARD driver");
131. MODULE_VERSION("v1.0");
132. MODULE_LICENSE("GPL");
```

**Line 94**, in the driver entry function, first use `alloc_netdev()` to allocate a `net_device` structure, set the members of the `net_device` structure variable, and use `register_netdev()` to register the `net_device` structure with the kernel after setting. Setting the variable members of the `net_device` structure is mainly the operation function and lines 109 and 110 to set the `ping` function.

**Line 68**, in the transmit packet function, first call the `netif_stop_queue(dev)` function to stop the upper layer from transmitting data packets, and then if it is a real network card, you should call the hardware interface to transmit data. Because this is a simulated network card, directly call the function accepted by the simulation to indicate that the transmission is complete. Look at it after accepting the function `ax_net_rx()`. After transmitting, release `skb_buff` and call the `netif_wake_queue()` function restores the upper layer to send data packets. Finally update the statistics.

The simulated receive function starting at **Line 31** should be handled in the receive interrupt. In the receiving function, modify the data in `sk_buff` to send it as a receiving `sk_buff`. The accrual method is to swap the sending and receiving mac addresses in the `erhhdr` and `iphdr` structures. use the `ip_fast_csum` function to retrieve the

`iphdr` checksum. Then set the `flag`, `0x08` means sending and receiving `ping` packet, and it needs to be changed to `0` if it is accepted. Reconstruct `sk_buff` and re-assign value. Use the `eth_type_trans` function to get the upper layer protocol. Finally call `netif_rx` to submit `sk_buff`

### Part 21.1.2: Run Test

The steps are as follows

```
mount -t nfs -o noblock 192.168.1.107:/home/alink/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-netcard-drv.ko
```

The IP and path are adjusted according to the actual situation.

After loading the driver, use the `ls /sys/class/net/` command to check whether the network card driver exists. Then use the command `ifconfig ax_net0 3.3.3.3` to set `ip`, and then you can `ping` test.

Then use the `ifconfig` command to view the network card information, check the `mac` address information and the statistics of the number of packets sent and received.

```
root@ax_peta:~# mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
root@ax_peta:~# cd /mnt
root@ax_peta:/mnt# mkdir /tmp/qt
root@ax_peta:/mnt# mount qt_lib.img /tmp/qt
EXT4-fs (loop0): recovery complete
EXT4-fs (loop0): mounted filesystem with ordered data mode. Opts: (null)
root@ax_peta:/mnt# cd /tmp/qt
root@ax_peta:/tmp/qt# source ./qt_env_set.sh
/tmp/qt
root@ax_peta:/tmp/qt# cd /mnt
root@ax_peta:/mnt# insmod ./ax-netcard-drv.ko
ax_netcard_drv: loading out-of-tree module taints kernel.
ax net0: mixed HW and IP checksum settings.
root@ax_peta:/mnt# ls /sys/class/net/
mx net0 eth0 lo sit0
root@ax_peta:/mnt# ifconfig ax_net0 3.3.3.3
root@ax_peta:/mnt# ping 3.3.3.4
PING 3.3.3.4 (3.3.3.4): 56 data bytes
64 bytes from 3.3.3.4: seq=0 ttl=64 time=0.186 ms
64 bytes from 3.3.3.4: seq=1 ttl=64 time=0.080 ms
64 bytes from 3.3.3.4: seq=2 ttl=64 time=0.067 ms
64 bytes from 3.3.3.4: seq=3 ttl=64 time=0.063 ms
64 bytes from 3.3.3.4: seq=4 ttl=64 time=0.063 ms
64 bytes from 3.3.3.4: seq=5 ttl=64 time=0.060 ms
...
-- 3.3.3.4 ping statistics --
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.060/0.086/0.186 ms
root@ax_peta:/mnt# irconfig
ax_net0 Link encap:Ethernet HWaddr 0A:0B:0C:0D:0E:0F
      inet addr:3.3.3.3 Bcast:3.255.255.255 Mask:255.0.0.0
      inet6 addr: fe80::0a0b:0cff:fe0d:0ortlo/64 Scope:Link
        UP BROADCAST RUNNING NOARP MULTICAST MTU:1500 Metric:1
        RX packets:17 errors:0 dropped:0 overruns:0 frame:0
        TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:2343 (2.2 KiB) TX bytes:2343 (2.2 KiB)

eth0 Link encap:Ethernet HWaddr 00:0A:85:00:1B:53
      inet addr:192.168.1.56 Bcast:192.168.1.255 Mask:255.255.255.0
      inet6 addr: ::1a1/128 Scope:Host
        UP BROADCAST RUNNING MTU:1500 Metric:1
        RX packets:5093 errors:0 dropped:1 overruns:0 frame:0
        TX packets:2082 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:6069490 (5.7 MiB) TX bytes:208918 (204.0 KiB)
        Interrupt:29 Base address:0xb000

lo Link encap:Local Loopback
      inet addr:127.0.0.1 Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING MTU:65536 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

root@ax_peta:/mnt#
```

## Part 23: DMA Driver

### Part 23.1: DMA Introduction

CPU and memory are both indispensable parts of a computer. The speed of the CPU is limited by the memory, and at the same time it has to wait for the memory to process and cannot process other events. Therefore, using the CPU to move memory data is a waste of resources. Therefore, **DMA**, a device dedicated to handling memory data, came into being.

DMA is the abbreviation of Direct Memory Access, that is, direct memory read and write, the so-called direct, that is, memory to memory, not through the CPU. DMA can support data interaction from memory to peripheral, peripheral to memory, and memory to memory, saving a lot of CPU resources when necessary. Of course, although DMA hardly occupies the CPU, it still occupies the system bus.

### Part 23.2: DMAC

The **DMAC** here refers to the DMA controller. A peripheral supporting DMA does not mean that it can actively initiate DMA operations. It can only be said that at least it supports passive DMA data reading and writing. Earlier we said that DMA hardly occupies CPU resources, because the DMA transfer needs to be initiated by the CPU, and then all the data is transferred to the DMAC. During the DMAC handling process, the following key concepts are involved.

#### Part 23.2.1: DAM Channels

Most DMACs support multiple DMAs, that is, DMA channels. DMA channel is equivalent to the data transmission channel provided by DMAC, that is, the method of data transmission, which solves the

problem of how to transmit. In fact, DMA channel is an abstract concept, not a physical channel. As we said earlier, DMA transfers still occupy bus resources, and bus resources are very limited, so simultaneous multiple DMA transfers are not realistic.

The purpose of abstracting the concept of DMA channel is actually to facilitate the management of DMA clients, so that each client has an exclusive channel, but in fact, DMAC will perform sequential arbitration and serial transmission during transmission.

### Part 23.2.2: DAM Request Lines

The **request line** refers to the physical connection between the **DMA** device and the **DMAC**. This line is used by the **DMA** device to notify the DMAC whether it can start transmission, that is, to solve the problem of when to transmit. Usually each **DMA** data receiving node (**endpoint**) has a DMA request line connected to the **DMAC**.

From the perspective of master and slave, **DMA channel** is provided by **DMAC**, which is equivalent to the master controller, and the binding of DMA request line and DMA device is equivalent to slave client. The number of DMA request lines in the system is usually more than that of DMA channel, because the DMA request line is not active every moment.

### Part 23.2.3: Transfer size, Transfer wide,Burst size

**Transfer wide** can be understood as the size of a single transfer of data. Compared with the serial port, the serial port can only transfer one byte at a time, while DMA can select the data size that can be transferred at a time. The **transfer size** on this basis is the number of transfers, not the total size, which means that the total length of DMA transfer is actually **transfer size** multiplied by **transfer wide**.

The **burst size** refers to the internal buffer size of the **DMAC**.

---

When the source or destination of DMA transfer is memory, DMAC will first read the data to the buffer, and then transfer it in or out.

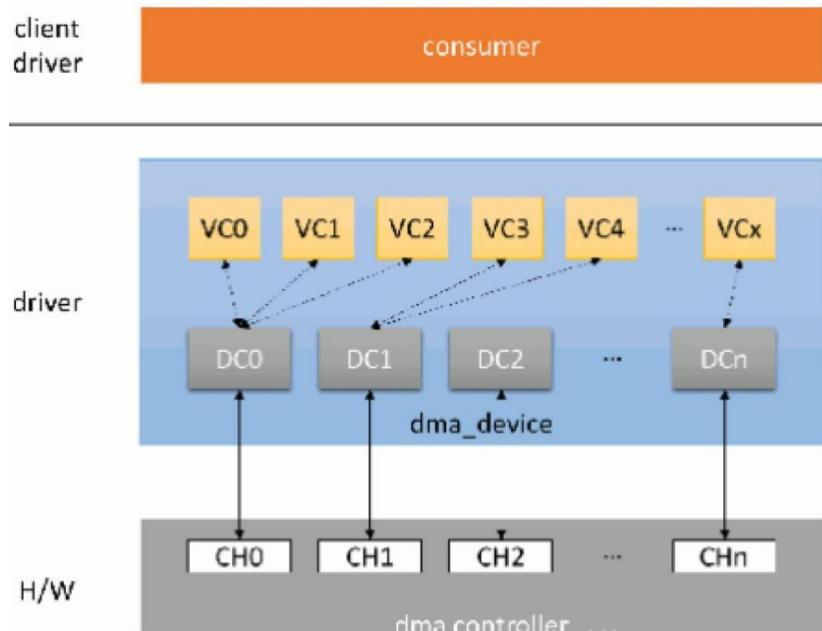
#### Part 23.2.4: scatter-gather

DMA operations must be continuous physical memory. In practical applications, it is inevitable to encounter discontinuous data processing in physical memory. **Scatter-gather** refers to the operation of copying discontinuous data to continuous buffers. This operation process can be realized by software, and there is also direct hardware support. The specific implementation does not need to be considered. The main point here is to emphasize that DMA operations must be continuous physical memory.

### Part 23.3: DMA in Linux

The **Linux DMA engine** framework provides two frameworks, **DMA controller** and **DMA client**. Corresponding to the two perspectives of DMA provider and DMA user. **pl330** is a DMA controller, in fact it is from the perspective of the DMA provider. In our example above, the objects that use **DMA** can actually be specific from memory to memory, and memory is the user of DMA. But there is no distinction between such concepts in the above routine, which brings the limitations of the routine. In fact, the operations of pl330 in the previous routine can be extracted, and they are common to other users. This is also the consistent design idea of linux system. Because of this, the originally uncomplicated DMA is somewhat complicated under this design.

DMA engine framework diagram:



### Part 23.3.1: DMA Controller Driver Framework

The driver of the controller is also available, which is similar to the previous **I2C** and **SPI**. We mainly want to understand the client driver, and the controller driver for a little understanding.

The **DMA controller** framework abstracts the **channel** corresponding to the physical channel of the **DMAC**, and defines a virtual channel. The software can realize multiple virtual channels corresponding to one physical **channel**.

Introduce the data structure mainly involved in the **DMA controller** framework:

#### 1) struct dma\_device

**struct dma\_device** is defined in [include/linux/dmaengine.h](#), as follows

```

1. struct dma_device {
2.
3.     unsigned int chan	cnt;
4.     unsigned int privatecnt;
5.     struct list_head channels;
6.     struct list_head global_node;
7.     struct dma_filter filter;
8.     dma_cap_mask_t cap_mask;
9.     unsigned short max_xor;

```

```
10.     unsigned short max_pq;
11.     enum dmaengine_alignment copy_align;
12.     enum dmaengine_alignment xor_align;
13.     enum dmaengine_alignment pq_align;
14.     enum dmaengine_alignment fill_align;
15. #define DMA_HAS_PQ_CONTINUE (1 << 15)
16.
17.     int dev_id;
18.     struct device *dev;
19.
20.     u32 src_addr_widths;
21.     u32 dst_addr_widths;
22.     u32 directions;
23.     u32 max_burst;
24.     bool descriptor_reuse;
25.     enum dma_residue_granularity residue_granularity;
26.
27.     int (*device_alloc_chan_resources)(struct dma_chan *chan);
28.     void (*device_free_chan_resources)(struct dma_chan *chan);
29.
30.     struct dma_async_tx_descriptor *(*device_prep_dma_memcpy)(
31.         struct dma_chan *chan, dma_addr_t dst, dma_addr_t src,
32.         size_t len, unsigned long flags);
33.     struct dma_async_tx_descriptor *(*device_prep_dma_xor)(
34.         struct dma_chan *chan, dma_addr_t dst, dma_addr_t *src,
35.         unsigned int src_cnt, size_t len, unsigned long flags);
36.     struct dma_async_tx_descriptor *(*device_prep_dma_xor_val)(
37.         struct dma_chan *chan, dma_addr_t *src, unsigned int src_cnt,
38.         size_t len, enum sum_check_flags *result, unsigned long flags);
39.
40.     struct dma_async_tx_descriptor *(*device_prep_dma_pq)(
41.         struct dma_chan *chan, dma_addr_t *dst, dma_addr_t *src,
42.         unsigned int src_cnt, const unsigned char *scf,
43.         size_t len, unsigned long flags);
44.     struct dma_async_tx_descriptor *(*device_prep_dma_pq_val)(
45.         struct dma_chan *chan, dma_addr_t *pq, dma_addr_t *src,
46.         unsigned int src_cnt, const unsigned char *scf, size_t len,
47.         enum sum_check_flags *pqres, unsigned long flags);
48.     struct dma_async_tx_descriptor *(*device_prep_dma_memset)(
49.         struct dma_chan *chan, dma_addr_t dest, int value, size_t len,
50.         unsigned long flags);
51.     struct dma_async_tx_descriptor *(*device_prep_dma_memset_sg)(
52.         struct dma_chan *chan, struct scatterlist *sg,
53.         unsigned int nents, int value, unsigned long flags);
54.     struct dma_async_tx_descriptor *(*device_prep_dma_interrupt)(
55.         struct dma_chan *chan, unsigned long flags);
56.     struct dma_async_tx_descriptor *(*device_prep_dma_sg)(
57.         struct dma_chan *chan,
58.         struct scatterlist *dst_sg, unsigned int dst_nents,
59.         struct scatterlist *src_sg, unsigned int src_nents,
60.         unsigned long flags);
61.     struct dma_async_tx_descriptor *(*device_prep_slave_sg)(
62.         struct dma_chan *chan, struct scatterlist *sgl,
63.         unsigned int sg_len, enum dma_transfer_direction direction,
64.         unsigned long flags, void *context);
65.     struct dma_async_tx_descriptor *(*device_prep_dma_cyclic)(
66.         struct dma_chan *chan, dma_addr_t buf_addr, size_t buf_len,
67.         size_t period_len, enum dma_transfer_direction direction,
```

```
68.         unsigned long flags);
69.         struct dma_async_tx_descriptor *(*device_prep_interleaved_dma)(
70.             struct dma_chan *chan, struct dma_interleaved_template *xt,
71.             unsigned long flags);
72.         struct dma_async_tx_descriptor *(*device_prep_dma_imm_data)(
73.             struct dma_chan *chan, dma_addr_t dst, u64 data,
74.             unsigned long flags);
75.
76.         int (*device_config)(struct dma_chan *chan,
77.             struct dma_slave_config *config);
78.         int (*device_pause)(struct dma_chan *chan);
79.         int (*device_resume)(struct dma_chan *chan);
80.         int (*device_terminate_all)(struct dma_chan *chan);
81.         void (*device_synchronize)(struct dma_chan *chan);
82.
83.         enum dma_status (*device_tx_status)(struct dma_chan *chan,
84.             dma_cookie_t cookie,
85.             struct dma_tx_state *txstate);
86.         void (*device_issue_pending)(struct dma_chan *chan);
87.     };
```

**channels:** the head of the linked list.

**cap\_mask:** indicates the transmission capability of the controller, which needs to correspond to the callback function in the form of **device\_prep\_dma\_xxx**. Common values are as follows:

**DMA\_MEMCPY:** **memory copy** can be performed.

**DMA\_MEMSET:** **memory set** can be performed.

**DMA\_SG:** **scatter list** can be transferred.

**DMA\_CYCLIC:** **cyclic** transmission can be performed.

**DMA\_INTERLEAVE:** Cross transfer is possible.

**src\_addr\_widths:** Indicates which **src** types of widths the

**dst\_addr\_widths:** **dst** type indicating which widths the controller supports.

**directions:** Refer to the enumeration **dma\_transfer\_direction** for the transfer direction values supported by the **controller**.

**max\_burst:** The maximum burst transfer size. **descriptor\_reuse:** Indicates whether the transmission description of the **controller** can be reused.

**device\_alloc\_chan\_resources:** Called when the client applies

for a **channel**.

**device\_free\_chan\_resources**: Called when the client releases the **channel**.

**device\_prep\_dma\_xxx**: Called when the client obtains the transfer descriptor through **dmaengine\_prep\_xxx**.

**device\_config**: It is called when the client calls **dmaengine\_slave\_configchannel**.

**device\_pause**: Called when the client calls **dmaengine\_pause**.

**device\_resume**: Called when the client calls **dmaengine\_resume**.

**device\_terminate\_all**: Called when the client calls **dmaengine\_terminate\_xxx**.

**device\_issue\_pending**: Called when the client calls **dma\_async\_issue\_pending** to start the transfer.

The **DMA controller** driver needs to implement the specific processing content of these functions, which is equivalent to the **ops** operation function in the character device framework.controller supports

## 2) struct dma\_chan

Defined as follows

```
1. struct dma_chan {  
2.     struct dma_device *device;  
3.     dma_cookie_t cookie;  
4.     dma_cookie_t completed_cookie;  
5.  
6.     /* sysfs */  
7.     int chan_id;  
8.     struct dma_chan_dev *dev;  
9.  
10.    struct list_head device_node;  
11.    struct dma_chan_percpu_percpu *local;  
12.    int client_count;  
13.    int table_count;  
14.  
15.    /* DMA router */  
16.    struct dma_router *router;  
17.    void *route_data;
```

```
18.  
19.      void *private;  
20.};
```

**device:** Point to the **dma controller** where the **channel** is located.

**cookie:** The last **cookie** returned by the **dma controller** to the **client** when the client uses the channel as the operating object to obtain the transmission descriptor.

**completed\_cookie:** The **cookie** of the last completed transmission on the current **channel**.

**device\_node:** Add the **channel** to the **channel** list of **dma\_device**.

### 3) struct virt\_dma\_cha

It is defined in the file **drivers/dma/virt-dma.h** as follows:

```
1. struct virt_dma_desc {  
2.     struct dma_async_tx_descriptor tx;  
3.     /* protected by vc.lock */  
4.     struct list_head node;  
5. };  
6.  
7. struct virt_dma_chan {  
8.     struct dma_chan chan;  
9.     struct tasklet_struct task;  
10.    void (*desc_free)(struct virt_dma_desc *);  
11.  
12.    spinlock_t lock;  
13.  
14.    /* protected by vc.lock */  
15.    struct list_head desc_allocated;  
16.    struct list_head desc_submitted;  
17.    struct list_head desc_issued;  
18.    struct list_head desc_completed;  
19.  
20.    struct virt_dma_desc *cyclic;  
21.  
22.};
```

**chan:** A variable of **struct dma\_chan** type used to interact with the **client**.

**task:** A **tasklet**, waiting for the completion of the transmission of the virtual **channel**.

**desc\_allocated**,      **desc\_submitted**,      **desc\_issued**,

**desc\_completed**: Four linked list headers, used to store virtual **channel** descriptors in different states.

The related APIs of the **DMA controller** framework are as follows:

- 1) **struct dma\_device** registration and unregister

```
int dma_async_device_register(struct dma_device *device);
void dma_async_device_unregister(struct dma_device *device);
```

After the **struct dma\_device** variable is initialized, call **dma\_async\_device\_register** to register with the kernel. After successful registration, **dma\_device** will be placed on a global linked list named **dma\_device\_list** for later use.

**dma\_async\_device\_unregister** is a relative unregister function.

- 2) **cookie** related interface

**DMA cookie** represents a continuous memory used by the **DMA engine** in data transfer.

```
static inline void dma_cookie_init(struct dma_chan *chan)
static inline dma_cookie_t dma_cookie_assign(struct dma_async_tx_descriptor *tx)
static inline void dma_cookie_complete(struct dma_async_tx_descriptor *tx)
static inline enum dma_status dma_cookie_status(struct dma_chan *chan, dma_cookie_t
cookie, struct dma_tx_state *state)
```

**dma\_cookie\_init**: Initialize **cookies** and **completed\_cookie** in the channel.

**dma\_cookie\_assign**: Assign a **cookie** to the transfer description of the pointer.

**dma\_cookie\_complete**: When a transmission is completed, this interface can be called to update the **completed\_cookie** field of the **channel** corresponding to the transmission.

**dma\_cookie\_status**: Get the transmission status of the **cookie** specified by the specified **channel**.

DMA controller driver examples can refer to pl330.c (zynqmp reference\drivers\dma\xilinx\zynqmp\_dma.c). The general process is as follows:

- 1) Define the `struct dma_device` variable and initialize it, and implement the necessary return function according to the hardware.
- 2) According to the number of `channels` supported by the `controller`, a `struct dma_chan` variable is defined for each `channel` and initialized, and then each `channel` is added to the `channels` list of `dma_device`.
- 3) Call `dma_async_device_register` to register `dma_device` with the kernel.

#### Part 23.4.2: DMA Client Driver Framework

From the difference of source and target, DMA can be divided into four categories: memory to memory, memory to peripheral, peripheral to memory, and peripheral to peripheral. Because the memory can use `memcpy`, `memset` and other operations, the `linux engine` separates the memory to memory part and provides a set of `API-Async TX API`. The remaining three types share the same structure `Slave-DMA API`. `Slave` here refers to the `client` perspective, which is the user of `DMA`.

The main data structures involved in the `DMA client` framework are as follows:

- 1) `struct dma_slave_config`

Defined in `include/linux/dmaengine.h` as follows:

```
1. struct dma_slave_config {  
2.     enum dma_transfer_direction direction;  
3.     phys_addr_t src_addr;  
4.     phys_addr_t dst_addr;  
5.     enum dma_slave_buswidth src_addr_width;
```

```
6.     enum dma_slave_buswidth dst_addr_width;
7.     u32 src_maxburst;
8.     u32 dst_maxburst;
9.     bool device_fc;
10.    unsigned int slave_id;
11.};
```

**direction:** transfer direction, refer to [enum dma\\_transfer\\_direction](#) for value.

**src\_addr:** The transfer direction is the location where the data is read from the peripheral to the memory or peripheral to the peripheral. Memory to device type [channel](#) does not need to configure this parameter.

**dst\_addr:** The transfer direction is the location where data is written when the memory is to the peripheral or the peripheral is to the peripheral. The [channel](#) from peripheral to memory does not need to configure this parameter.

**src\_addr\_width:** the width of the [src](#) address.

**dst\_addr\_width:** the width of [dst](#) address.

**src\_maxburst:** The maximum transmittable burst size of [src](#), the unit is [src\\_addr\\_width](#).

**dst\_maxburst:** The maximum transmittable burst size of [dst](#), the unit is [dst\\_addr\\_width](#).

**slave\_id:** The identifier of the peripheral for the controller.

## 2) struct dma\_async\_tx\_descriptor

It is defined as follows:

```
12. struct dma_slave_config {
13.     enum dma_transfer_direction direction;
14.     phys_addr_t src_addr;
15.     phys_addr_t dst_addr;
16.     enum dma_slave_buswidth src_addr_width;
17.     enum dma_slave_buswidth dst_addr_width;
18.     u32 src_maxburst;
19.     u32 dst_maxburst;
20.     bool device_fc;
21.     unsigned int slave_id;
22.};
```

**cookie**: an integer number used to track this transmission. Generally, the **controller** maintains an incremental **number** internally. When the **client** obtains the transmission description, the **number** is assigned to the **cookie** and then incremented.

**flags**: enum **dma\_ctrl\_flags** enumeration.

**chan**: The corresponding **channel**.

**tx\_submit**: The callback function provided by the **controller** is used to submit the change descriptor to the list to be transferred.

**desc\_free**: The callback function provided by the **controller** to release the descriptor.

**callback**: **callback** function for the completion of transmission

**callback\_param**: **callback** parameter.

The implementation steps of **DMA client** using **Slave-DMA API** are roughly as follows:

### 1) Apply for DMA channel

```
struct dma_chan *dma_request_chan(struct device *dev, const char *name);  
void dma_release_channel(struct dma_chan *chan);
```

**dma\_request\_chan** apply to the kernel for **dma\_chan**

**dma\_release\_channel** is a relative release **dma\_chan**

### 2) Set DMA channel parameters

Set the parameters using the following function

```
int dmaengine_slave_config(struct dma_chan *chan, struct dma_slave_config *config)
```

First, the configuration content is represented by **struct dma\_slave\_config**, and the configuration information is submitted using this function.

### 3) Get descriptor

Before the **DMA** starts the transfer, the **client** needs to inform the controller of the transferred information (**src**, **dst**, etc.). After the controller confirms, it will return a descriptor to the **client**. The **client**

uses this descriptor to control and track the transfer.

According to different transmission types, use the following three interfaces to get the descriptor

```
struct dma_async_tx_descriptor
    *dmaengine_prep_slave_sg( struct dma_chan *chan, struct
        scatterlist *sgl,
        unsigned int sg_len, enum dma_data_direction direction,
        unsigned long flags);
struct dma_async_tx_descriptor
    *dmaengine_prep_dma_cyclic( struct dma_chan *chan,
        dma_addr_t buf_addr, size_t buf_len, size_t period_len, enum
        dma_data_direction direction);
struct dma_async_tx_descriptor
```

#### 4) Submit to start transfer

Use the following function to submit the descriptor

```
dma_cookie_t dmaengine_submit(struct dma_async_tx_descriptor *desc)
```

Use the following function to start the transfer

```
void dma_async_issue_pending(struct dma_chan *chan);
```

#### 5) Wait for the transfer to end

Waiting for the transmission can be done through the return function, or through functions such as **dma\_async\_is\_tx\_complete** to check whether the transmission is complete. In addition, you can also use the **dmaengine\_pause** and **dmaengine\_resume** functions to pause and terminate the transmission.

The simple **demo** of the **client** is as follows:

```
1. #include <linux/dmaengine.h>
2. #include <linux/dma-mapping.h>
3. #include <linux/types.h>
4. #include <linux/slab.h>
5. #include <linux/module.h>
6. #include <linux/init.h>
7. #include <linux/fs.h>
8. #include <linux/sched.h>
9. #include <linux/miscdevice.h>
10. #include <linux/device.h>
```

```

11. #include <linux/string.h>
12. #include <linux/errno.h>
13. #include <linux/types.h>
14. #include <linux/slab.h>
15. #include <linux/of_device.h>
16. #include <linux/async_tx.h>
17. #include <asm/uaccess.h>
18. #include <asm/delay.h>
19.
20. #define DEVICE_NAME "ax_dma"
21.
22. #define MAX_SIZE (512*64)
23.
24. static char *src;
25. static char *dst;
26. dma_addr_t dma_src;
27. dma_addr_t dma_dst;
28.
29. struct ax_dma_drv {
30.     struct dma_chan *chan;
31.     struct dma_device *dev;
32.     struct dma_async_tx_descriptor *tx;
33.     enum dma_ctrl_flags flags;
34.     dma_cookie_t cookie;
35. };
36. struct ax_dma_drv ax_dma;
37.
38. void dma_cb(void *dma_async_param)
39. {
40.     if(!memcmp(src, dst, MAX_SIZE))
41.     {
42.         printk("dma irq test ok\r\n");
43.     }
44. }
45.
46. static int dma_open(struct inode *inode, struct file *file)
47. {
48.     printk("dma_open\r\n");
49.     return 0;
50. }
51.
52. static int dma_release(struct inode *indoe, struct file *file)
53. {
54.     printk("dma_release\r\n");
55.     return 0;
56. }
57.
58. static ssize_t dma_read(struct file *filp, char user *buf, size_t size, loff_t
59. *ppos)
60. {
61.     int ret = 0;
62.     printk("dma_read\r\n");
63.
64.     ax_dma.tx = ax_dma.dev->device_prep_dma_memcpy(ax_dma.chan, dma_dst, dma_src,
65. MAX_SIZE, ax_dma.flags);
66.     if (!ax_dma.tx){
67.         printk(KERN_INFO "Failed to prepare DMA memcpy");
68.     }
69.
70.     ax_dma.tx->callback = dma_cb;
71.     ax_dma.tx->callback_param = NULL;
72.     ax_dma.cookie = ax_dma.tx->tx_submit(ax_dma.tx);
73.     if (dma_submit_error(ax_dma.cookie)){
74.         printk("DMA tx submit failed");
    }
    dma_async_issue_pending(ax_dma.chan);

```

```
75.
76.     return ret;
77. }
78.
79. static struct file_operations ax_fops =
80. {
81.     .owner    = THIS_MODULE,
82.     .open     = dma_open,
83.     .read     = dma_read,
84.     .release  = dma_release,
85. };
86.
87. static struct miscdevice dma_misc =
88. {
89.     .minor = MISC_DYNAMIC_MINOR,
90.     .name  = DEVICE_NAME,
91.     .fops   = &ax_fops,
92. };
93.
94. static int init dma_init(void)
95. {
96.     int ret=0;
97.     dma_cap_mask_t mask;
98.
99.     ret = misc_register(&dma_misc);
100.    if(ret)
101.    {
102.        printk("misc_register failed!\n");
103.        return 0;
104.    }
105.    printk("drv register ok\n");
106.    of_dma_configure(dma_misc.this_device, dma_misc.this_device->of_node, true)
107. ;
108.    dma_misc.this_device->coherent_dma_mask = 0xffffffff;
109.
110.    //源
111.    src = dma_alloc_coherent(dma_misc.this_device, MAX_SIZE, &dma_src, GFP_KERNEL);
112.    if (NULL == src)
113.    {
114.        printk("can't alloc buffer for src\n");
115.        return -ENOMEM;
116.
117.    //目标
118.    dst = dma_alloc_coherent(dma_misc.this_device, MAX_SIZE, &dma_dst, GFP_KERNEL);
119.    if (NULL == dst)
120.    {
121.        dma_free_coherent(NULL, MAX_SIZE, src, dma_src);
122.        printk("can't alloc buffer for dst\n");
123.        return -ENOMEM;
124.    }
125.    printk("buffer alloc ok\n");
126.
127.    //初始化 mask
128.    dma_cap_zero(mask);
129.    dma_cap_set(DMA_MEMCPY, mask);
130.    ax_dma.chan = dma_request_channel(mask, NULL, NULL);
131.    ax_dma.flags = DMA_CTRL_ACK | DMA_PREP_INTERRUPT;
132.    ax_dma.dev = ax_dma.chan->device;
133.    printk("chan request ok\n");
134.
135.    //给源地址一个初值
136.    memset(src, 0x5A, MAX_SIZE);
137.    //给目标地址一个不一样的初值
```

```
137.     memset(dst, 0xA5, MAX_SIZE);
138.
139.     return 0;
140. }
141.
142. static void exit dma_exit( void )
143. {
144.     dma_release_channel(ax_dma.chan);
145.     dma_free_coherent(dma_misc.this_device, MAX_SIZE, src, dma_src);
146.     dma_free_coherent(dma_misc.this_device, MAX_SIZE, dst, dma_dst);
147.     misc_deregister(&dma_misc);
148. }
149.
150. //驱动入口函数标记
151. module_init(dma_init);
152. //驱动出口函数标记
153. module_exit(dma_exit);
154.
155. /* 驱动描述信息 */
156. MODULE_AUTHOR("Alinx");
157. MODULE_ALIAS("dma");
158. MODULE_DESCRIPTION("DMA driver");
159. MODULE_VERSION("v1.0");
160. MODULE_LICENSE("GPL");
```

The `dma_alloc_coherent()` function in line 110 can no longer pass `NULL` to the first parameter `dev` in the new version of the kernel. And it needs **106~107** lines of code to initialize the relevant parameters of the `dev` passed in.

This code can be compiled into a module for testing. The testing method is to use `dma` to copy the data on `src` to `dst`. The application layer only needs to call the `open` and `read` functions.

## Part 24: Multi-touch screen driver

We are not unfamiliar with multi-touch screens, that is, capacitive screens. Mobile phones and pads are all multi-touch screens. In many application scenarios, single-point touch screens can no longer be adapted. For example, mobile games need to touch multiple points at the same time. Of course, these are not what we need to care about, we just need to figure out how to drive it in linux.

### Part 24.1: Multi-touch Screen in Input Subsystem

Multi-touch screen is also an input device. In the **input** subsystem of **linux** we mentioned earlier, there is also support for multi-touch screen. In the **input** subsystem, **input\_handler** helped us implement the methods in **fops**. What we need to do is to combine specific hardware and send data through the interface when there is data. To implement an input device driver, the steps are roughly as follows:

- 1) Construct **input\_device**
- 2) Set **input\_device** parameter
- 3) Register **input\_device** with the kernel
- 4) When the hardware receives data, report data through **input\_event**

Combined with the multi-touch screen of our hardware device here, the general steps are as follows:

- 1) Allocate **input\_dev** structure

```
struct input_dev *ts_dev;  
ts_dev = input_allocate_device();
```

- 2) Set **input\_dev** structure

Set the types of events that can be generated

```
set_bit(EV_SYN, ts_dev->evbit); // Set Key Event
```

```
set_bit(EV_ABS, ts_dev->evbit);// Set Sliding Event
```

Set the specific events that can be generated in this type of event

```
set_bit(ABS_MT_TRACKING_ID, ts_dev->absbit);  
set_bit(ABS_MT_POSITION_X, ts_dev->absbit);  
set_bit(ABS_MT_POSITION_Y, ts_dev->absbit);
```

Set event scope

```
input_set_abs_params(ts_dev, ABS_MT_TRACKING_ID, 0, MTP_MAX_ID, 0, 0);  
input_set_abs_params(ts_dev, ABS_MT_POSITION_X, 0, MTP_MAX_X, 0, 0);  
input_set_abs_params(ts_dev, ABS_MT_POSITION_Y, 0, MTP_MAX_Y, 0, 0);
```

### 3) Register **input\_dev** structure

```
input_register_device(ts_dev);
```

### 4) Hardware operation

Registration interrupted:

```
request_irq();
```

Report event:

```
input_report_abs(ts_dev, ABS_MT_POSITION_X, mtp_events[i].x);  
input_report_abs(ts_dev, ABS_MT_POSITION_Y, mtp_events[i].y);  
input_report_abs(ts_dev, ABS_MT_TRACKING_ID, mtp_events[i].id);  
input_mt_sync(ts_dev);
```

## Part 24.2: Multi-touch screen and SoC interface

The touch screen is generally connected to the **SoC** through **IIC** or **SPI**, and the software regards the screen as a plane coordinate system. When we touch a certain position on the screen, the touch screen will transmit the coordinates of this point to the **SoC** via **IIC** or **SPI**. For a multi-point touch screen, it only sends multiple points at once.

When the concept of "multiple" exists, there must be one more attribute that needs to be concerned, that is, the relationship between multiple. Here we also have two reporting modes. One is to directly report the position without caring about the relationship between the

contacts, and the other is to add the sequence of the electric shock relationship to the upper position.

Also report the contact location:

```
ABS_MT_POSITION_X x0  
ABS_MT_POSITION_Y y0  
SYN_MT_REPORT  
ABS_MT_POSITION_X x1  
ABS_MT_POSITION_Y y1  
SYN_MT_REPORT
```

Report the contact location and relationship:

```
ABS_MT_TRACKING_ID 0  
ABS_MT_POSITION_X x0  
ABS_MT_POSITION_Y y0  
ABS_MT_TRACKING_ID 1  
ABS_MT_POSITION_X x1  
ABS_MT_POSITION_Y y1  
SYN_MT_REPORT
```

Our FPGA development board platform is interconnected by IIC and multi-touch screen, which means that our multi-touch screen driver will be a hybrid of IIC and input subsystem.

The IIC framework is relatively simple, so let's look at it directly through the code.

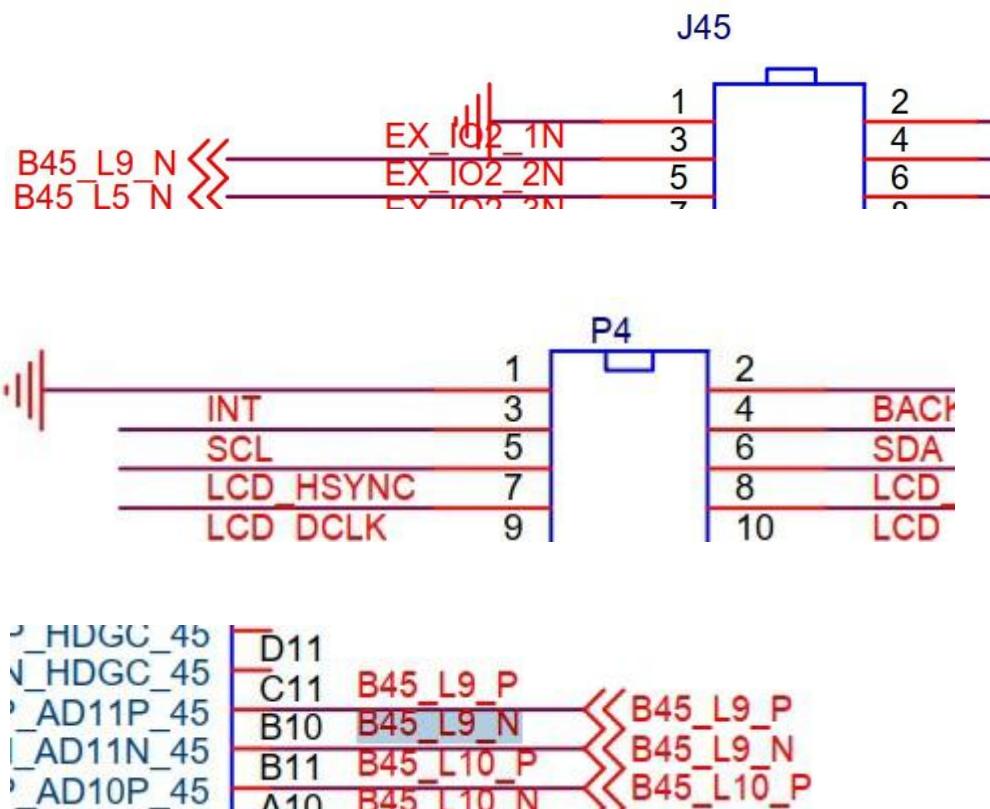
### Part 24.3: vivado project and petalinux

The ALINX platform is connected to the touch screen through the expansion port on the PL end, so it is necessary to use the vivado software to configure the project, and then recreate the petalinux project. We have already experienced the reconstruction of the vivado project and the petalinux project once in the pwm chapter.

The vivado project can directly use the touch screen project in vitis.

## Part 24.4: Schematic

We need to refer to the schematic when setting the xdc file in vivado. The pin of the expansion port corresponds to which pin of the touch screen and which pin of the chip. For example, the expansion port J45:



The B5\_L9\_N pin corresponds to the INT pin of the touch screen and is also connected to the B10 pin of the chip. Finally, the INT pin of the touch screen corresponds to B10, so INT is constrained to B10 in the xdc file.

```
set_property PACKAGE_PIN B10 [get_ports {lcd_intr[0]}]
```

## Part 24.5: Device Tree

The touch screen is regarded as an **IIC** client, and the device tree node is placed under **&i2c0**. For the modification method of the device tree, please refer to the previous chapter. Don't forget to operate in the

new **petalinux** project.

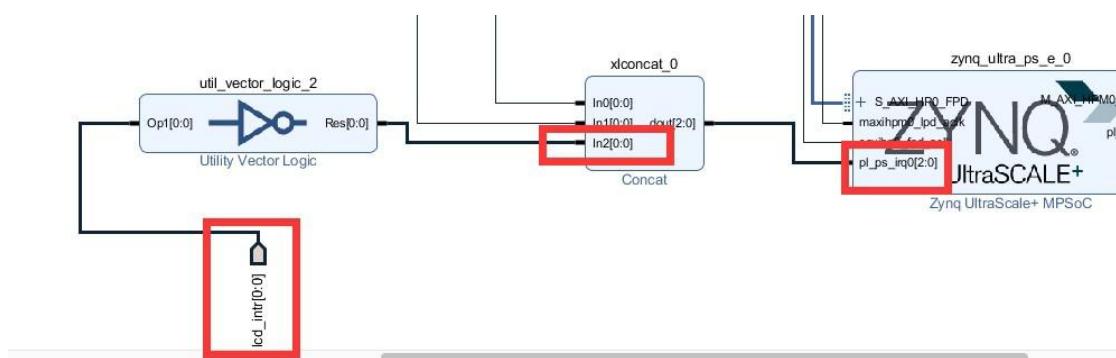
```

1. &i2c0 {
2.     clock-frequency = <100000>;
3.     alinx_an071@38 {
4.         compatible = "alink,an071";
5.         reg = <0x38>;
6.         interrupt-parent = <&gic>;
7.         interrupts = <0 91 4>;
8.     };
9. };

```

Let's talk about on how the interrupt number 91 comes from?

Take a look at this link in vivado.



`lcd_intr` is the pin INT of the touch screen interrupt signal output. As we said above, it is bound to the B10 of Ultra scale+. B10 is the pin of the PL end. The connection on Ultra scale+ is that `pl_ps_irq[2:0]` has three interrupt sources.

INT is connected to in2 of the connector through the ige NAND gate, interrupting `pl_ps_irq[2]`. Open the manual ug1085 and find the table corresponding to the interrupt number.

PL_PS_Group0	121:128	GICP2 [25:31] GICP3[0]	PL to PS interrupt signals 0 to 7. <sup>(3)</sup>
--------------	---------	---------------------------	---

This is the terminal number 123. In the section on interrupts, we said that the actual interrupt number is 32 smaller than the manual, and eventually it is 91.

## Part 24.6: Multi-touch Screen Driver

Create a new driver file `ax-an071-drv` in the new **petalinux**

project:

```
petalinux-create -t modules --name ax-an071-drv
```

Enter the following code in the file **ax-an071-drv.c**

```

1. /**
2.  *Author : ALINX Electronic Technology (Shanghai) Co., Ltd.
3.  *Website: http://www.alinx.com
4.  *Address: Room 202, building 18,
5.  No.518 xinbrick Road,
6.  Songjiang District, Shanghai
7.  *Created: 2020-3-2
8.  *Version: 1.0
9.  */
10.
11. #include <linux/module.h>
12. #include <linux/ratelimit.h>
13. #include <linux/irq.h>
14. #include <linux/interrupt.h>
15. #include <linux/input.h>
16. #include <linux/i2c.h>
17. #include <linux/uaccess.h>
18. #include <linux/delay.h>
19. #include <linux/debugfs.h>
20. #include <linux/slab.h>
21. #include <linux/gpio/consumer.h>
22. #include <linux/input/mt.h>
23. #include <linux/input/touchscreen.h>
24. #include <linux/of_device.h>
25.
26. #define NO_REGISTER          0xff
27.
28. #define TOUCH_EVENT_DOWN      0x00
29. #define TOUCH_EVENT_UP        0x01
30. #define TOUCH_EVENT_ON         0x02
31. #define TOUCH_EVENT_RESERVED   0x03
32.
33. #define EDT_NAME_LEN           23
34. #define EDT_SWITCH_MODE_RETRIES 10
35. #define EDT_SWITCH_MODE_DELAY    5
36. #define EDT_RAW_DATA_RETRIES     100
37. #define EDT_RAW_DATA_DELAY       1
38.
39. #define MAX_SUPPORT_POINTS      1
40.
41. #define SCREEN_MAX_X            800
42. #define SCREEN_MAX_Y            480
43.
44. #define AUO_PIXCIR_MAX_AREA      0xff
45.
46. #define ALINX_READ_COOR_ADDR      0x814E
47.
48. struct alinx_ts_data
49. {
50.     struct i2c_client *client;
51.     struct input_dev *input;
52.     int num_x;
53.     int num_y;
54.
55.     struct gpio_desc *reset_gpio;
56.     struct mutex mutex;
57.     int threshold;
58.     int gain;
59.     int offset;
60.     int report_rate;
61.     int max_support_points;

```

```
62.
63.     char name[EDT_NAME_LEN];
64. };
65.
66. struct edt_i2c_chip_data
67. {
68.     int max_support_points;
69. };
70.
71. static int alinx_ts_readwrite(struct i2c_client *client, u16 wr_len, u8 *wr_buf, u16 rd_len, u8 *rd_buf)
72. {
73.     struct i2c_msg wrmsg[2];
74.     int i = 0;
75.     int ret;
76.
77.     if (wr_len) {
78.         wrmsg[i].addr = client->addr;
79.         wrmsg[i].flags = 0;
80.         wrmsg[i].len = wr_len;
81.         wrmsg[i].buf = wr_buf;
82.         i++;
83.     }
84.     if (rd_len) {
85.         wrmsg[i].addr = client->addr;
86.         wrmsg[i].flags = I2C_M_RD;
87.         wrmsg[i].len = rd_len;
88.         wrmsg[i].buf = rd_buf;
89.         i++;
90.     }
91.
92.
93.     ret = i2c_transfer(client->adapter, wrmsg, i);
94.     if (ret < 0)
95.         return ret;
96.     if (ret != i)
97.         return -EIO;
98.
99.     int nret = ret < 0 ? ret : (ret != ARRAY_SIZE(wrmsg) ? -EIO : 0);
100.
101.    return 0;
102. }
103.
104. static irqreturn_t alinx_ts_isr(int irq, void *dev_id)
105. {
106.
107.     struct alinx_ts_data *ts = dev_id;
108.
109.     disable_irq_nosync(irq);
110.
111.     struct device *dev = &ts->client->dev;
112.     u8 cmd;
113.     u8 rdbuf[63];
114.     int i, offset, tplen, datalen, crclen;
115.     int error;
116.     u32 id, x, y, status, num_touches;
117.     bool update_input = false;
118.
119.     u32 gestid;
120.
121.     cmd = 0x0;
122.     offset = 3;
123.     tplen = 6;
124.     crclen = 0;
125.
126.     if(!ts)
127.         goto out;
128.
129.     memset(rdbuf, 0, sizeof(rdbuf));
130.     datalen = tplen + offset + crclen;
```

```
131.
132.
133.     error = alinx_ts_readwrite(ts->client,
134.                               sizeof(cmd), &cmd,
135.                               datalen, rdbuf);
136.     if (error) {
137.         dev_err_ratelimited(dev, "Unable to fetch data, error: %d\n",
138.                             error);
139.         goto out;
140.     }
141.
142.     u8 *gbuf = &rdbuf[1];
143.     gestid = gbuf[0] & 0xff;
144.
145.
146.     u8 *xbuf = &rdbuf[2];
147.     num_touches = xbuf[0] & 0x0f;
148.
149.     u8 *buf = &rdbuf[offset];
150.     status = buf[0] >> 6;
151.     if (status == TOUCH_EVENT_RESERVED)
152.         goto out;
153.
154.     x = ((buf[0] << 8) | buf[1]) & 0xffff;
155.     y = ((buf[2] << 8) | buf[3]) & 0xffff;
156.     id = (buf[2] >> 4) & 0x0f;
157.
158.     printk("x = %d, y = %d, id = %d\n\n", x, y, id);
159.
160.     if (status != TOUCH_EVENT_UP)
161.     {
162.         input_report_abs(ts->input, ABS_X, x);
163.         input_report_abs(ts->input, ABS_Y, y);
164.         input_event(ts->input, EV_KEY, BTN_TOUCH, 1);
165.         input_report_abs(ts->input, ABS_PRESSURE, 1);
166.     }
167.     else
168.     {
169.         input_report_key(ts->input, BTN_TOUCH, 0);
170.         input_report_abs(ts->input, ABS_PRESSURE, 0);
171.     }
172.
173.     input_sync(ts->input);
174.
175. out:
176.     enable_irq(irq);
177.
178.     return IRQ_HANDLED;
179. }
180.
181. static int alinx_ts_identify(struct i2c_client *client, struct alinx_ts_data *tsdata,
182. a, char *fw_version)
183. {
184.     u8 rdbuf[EDT_NAME_LEN];
185.     char *p;
186.     int error;
187.     char *model_name = tsdata->name;
188.
189.     memset(rdbuf, 0, sizeof(rdbuf));
190.     error = alinx_ts_readwrite(client, 1, "\xbb", EDT_NAME_LEN - 1, rdbuf);
191.     if (error) return error;
192.
193.     error = alinx_ts_readwrite(client, 1, "\xA6", 2, rdbuf);
194.     if (error) return error;
195.
196.     strlcpy(fw_version, rdbuf, 2);
197.
198.     error = alinx_ts_readwrite(client, 1, "\xA8", 1, rdbuf);
199.     if (error) return error;
```

```
200.     snprintf(model_name, EDT_NAME_LEN, "EP0%i%10M09", rdbuf[0] >> 4, rdbuf[0] & 0x)
201.     F);
202.     return 0;
203. }
204.
205. static int alinx_ts_probe(struct i2c_client *client, const struct i2c_device_id *id)
206. {
207.     struct alinx_ts_data *tsdata;
208.     struct input_dev *input;
209.     unsigned long irq_flags;
210.     int error;
211.     char fw_version[EDT_NAME_LEN];
212.
213.     dev_dbg(&client->dev, "probing for Alinx I2C\n");
214.
215.     tsdata = devm_kzalloc(&client->dev, sizeof(*tsdata), GFP_KERNEL);
216.     if (!tsdata)
217.     {
218.         dev_err(&client->dev, "failed to allocate driver data.\n");
219.         return -ENOMEM;
220.     }
221.
222.     tsdata->max_support_points = MAX_SUPPORT_POINTS;
223.
224.     input = devm_input_allocate_device(&client->dev);
225.     if (!input)
226.     {
227.         dev_err(&client->dev, "failed to allocate input device.\n");
228.         error -ENOMEM;
229.         goto err_free_mem;
230.     }
231.
232.     tsdata->client = client;
233.     tsdata->input = input;
234.
235.     error = alinx_ts_identify(client, tsdata, fw_version);
236.     if (error)
237.     {
238.         dev_err(&client->dev, "touchscreen probe failed\n");
239.         goto err_free_mem;
240.     }
241.
242.     input->name = tsdata->name;
243.     input->id.bustype = BUS_I2C;
244.     input->phys = "I2C";
245.     input->dev.parent = &client->dev;
246.
247.     _set_bit(EV_ABS, input->evbit);
248.     _set_bit(EV_KEY, input->evbit);
249.     _set_bit(EV_SYN, input->evbit);
250.     _set_bit(BTN_TOUCH, input->keybit);
251.     _set_bit(ABS_X, input->absbit);
252.     _set_bit(ABS_Y, input->absbit);
253.     _set_bit(ABS_PRESSURE, input->absbit);
254.     input_set_abs_params(input, ABS_X, 0, SCREEN_MAX_X, 0, 0);
255.     input_set_abs_params(input, ABS_Y, 0, SCREEN_MAX_Y, 0, 0);
256.
257.     input_set_drvdata(input, tsdata);
258.     i2c_set_clientdata(client, tsdata);
259.
260.     error = input_register_device(input);
261.     if (error) goto err_free_mem;
262.
263.     irq_flags = irq_get_trigger_type(client->irq);
264.     if (irq_flags == IRQF_TRIGGER_NONE) irq_flags = IRQF_TRIGGER_HIGH;
265.     irq_flags |= IRQF_ONESHOT;
266.
```

```
267.     error = devm_request_threaded_irq(&client->dev, client->irq, NULL, alinx_ts_isr,
268.                                         irq_flags, client->name, tsdata);
269.     if (error)
270.     {
271.         dev_err(&client->dev, "Unable to request touchscreen IRQ.\n");
272.         goto err_free_mem;
273.     }
274.
275.     return 0;
276.
277.
278. err_free_mem:
279.     input_free_device(input);
280.     kfree(tsdata);
281.
282.
283.     return error;
284. }
285.
286. static int alinx_ts_remove(struct i2c_client *client)
287. {
288.     const struct alinx_platform_data *pdata = dev_get_platdata(&client->dev);
289.
290.     struct alinx_ts_data *tsdata = i2c_get_clientdata(client);
291.
292. //    free_irq(client->irq, tsdata);
293.     input_unregister_device(tsdata->input);
294.
295.     kfree(tsdata);
296.
297.     return 0;
298. }
299.
300. static int maybe_unused alinx_ts_suspend(struct device *dev)
301. {
302.     struct i2c_client *client = to_i2c_client(dev);
303.
304.     if (device_may_wakeup(dev)) enable_irq_wake(client->irq);
305.
306.     return 0;
307. }
308.
309. static int maybe_unused alinx_ts_resume(struct device *dev)
310. {
311.     struct i2c_client *client = to_i2c_client(dev);
312.
313.     if (device_may_wakeup(dev)) disable_irq_wake(client->irq);
314.
315.     return 0;
316. }
317.
318. static SIMPLE_DEV_PM_OPS(alinx_ts_pm_ops, alinx_ts_suspend, alinx_ts_resume);
319.
320.
321. static const struct edt_i2c_chip_data alinx_data =
322. {
323.     .max_support_points = 5,
324. };
325.
326.
327. static const struct i2c_device_id alinx_ts_id[] =
328. {
329.     { .name = "alink", .driver_data = (long)&alink_data },
330.     {}
331. };
332. MODULE_DEVICE_TABLE(i2c, alinx_ts_id);
333.
334.
335. static const struct of_device_id alinx_of_match[] =
```

```
336. {
337.     { .compatible = "alinx,an071", .data = &alinx_data },
338.     {}
339.
340. };
341. MODULE_DEVICE_TABLE(of, alinx_of_match);
342.
343. static struct i2c_driver alinx_ts_driver =
344. {
345.     .driver =
346.     {
347.         .owner = THIS_MODULE,
348.         .name = "alink",
349.         .of_match_table = of_match_ptr(alinx_of_match),
350.         .pm = &alink_ts_pm_ops,
351.     },
352.     .id_table = alinx_ts_id,
353.     .probe = alinx_ts_probe,
354.     .remove = alinx_ts_remove,
355. };
356.
357. module_i2c_driver(alinx_ts_driver);
358.
359. MODULE_AUTHOR("Alinx");
360. MODULE_DESCRIPTION("Alinx I2C Touchscreen Driver");
361. MODULE_LICENSE("GPL");
```

The driver program starts from the entry function

**Lines 625**, the work queue is initialized in the [goodix\\_ts\\_init\(\)](#) function to add interrupt processing content.

**Lines 638**, the i2c driver is added by [i2c\\_add\\_driver](#), and then the content of the i2c framework. First see the registered content [goodix\\_ts\\_driver](#).

**Line 613**, define [goodix\\_ts\\_driver](#), mainly look at the [probe](#) function, jump to line 474 to implement the [goodix\\_ts\\_probe\(\)](#) function.

**Line 499**, add a queue item to the queue, the queue item [goodix\\_ts\\_work\\_func\(\)](#) is actually the content to be processed by the interrupt.

**Line 198**, in the function implementation, it mainly analyzes and processes the data returned by i2c. For the specific definition, please refer to the manual of the touch screen. After parsing, we do touch-corresponding operations in the [gtp\\_touch\\_down](#) and [gtp\\_touch\\_up](#) functions.

Line 527, the called `gtp_request_input_dev()` function, the content is to register an input subsystem, and set the click event sliding event and so on.

There are some parameter settings in the code and I won't go into details. To sum up, it is to set and obtain the information of the touch screen through i2c, and then feedback the touch behavior to the system through the input subsystem to respond to the needs of other programs.

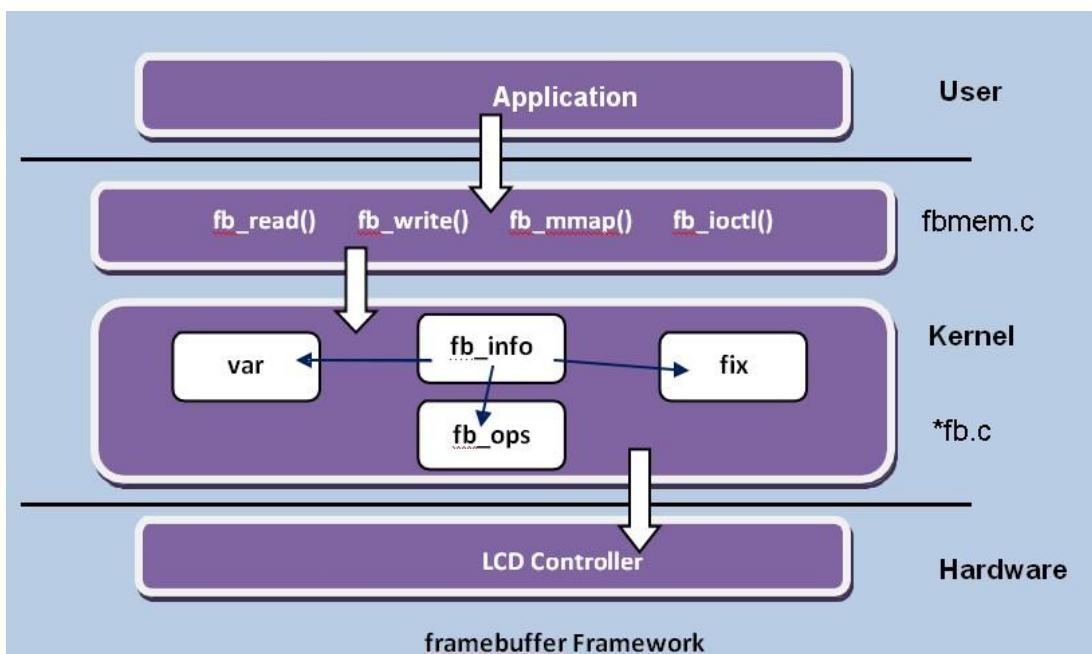
We define `GTP_DEBUG_ON` on line 21 as 1, after compiling, load the driver on the board and click on the screen to see that the serial port interrupt returns to the touch coordinates. But this still does not reflect the role of the input subsystem. In the next chapter, we will combine the LCD display to confirm the final test result.

But this still does not reflect the role of the `input` subsystem. In the next chapter, we will combine the `Lcd` display to confirm the final test result.

## Part 25: LCD Drive

### Part 25.1: framebuffer frame

Generally speaking, the principle of **Lcd** display is as follows: **Lcd** is connected to the main control chip through several data lines (in addition to the data line, there are clock lines, synchronization signal lines, etc.), and the main control chip allocates a memory space for storing the display picture data, transmit picture data to LCD screen display through data line. This memory space is the **framebuffer**. In Linux, the **framebuffer** is abstracted into a device, and the operation of **Lcd** is finally simplified to the operation of **framebuffer**, so the realization of **Lcd** driver in the Linux kernel is the **framebuffer** framework



The **framebuffer** frame is essentially a character device, which is exactly the same as the **misc** device mentioned earlier. It re-encapsulates the character device and creates a class **/sys/class/graphics**. For the application layer, the method of using **framebuffer** is roughly like this: report header file **linux/fb.h**, open

device file `/dev/fbN`, `mmap` mapping, and fill `framebuffer`.

The code related to the `framebuffer` frame is in `/driver/video/fbdev/core/fbmem.c`, and the code related to specific hardware operations is these files in `*fb.c` in the `/driver/video/fbdev` directory. `fbmem.c` provides the operation function interface that connects with the user layer and is associated with `*fb.c`, while `*fb.c` is responsible for the specific implementation of the interface function. What driver developers need to complete is the `*fb.c` part. Xilinx also provides `xilinxfb.c`. With reference to these already implemented `*fb.c`, you can guess that the steps to implement an `Lcd` driver are roughly as follows:

- 1) Use the `framebuffer_alloc` function to register a `fb_info` structure variable

The `fb_info` structure is defined in `/include/linux/fb.h` and is used to describe `fb` devices. The main things we need to care about are: variable parameter `struct fb_var_screeninfo var`, immutable parameter `struct fb_fix_screeninfo fix`, operation function set `struct fb_ops *fbops`. It is also the focus of setting the `fb_info` structure variable.

- 2) Set the `fb_info` structure variable, including the realization of the `fb_ops` operation function set

Set the fixed parameter video memory byte size, type, Lcd line length, etc. Set variable parameters horizontal resolution, vertical resolution, virtual horizontal resolution, virtual vertical resolution, pixel depth, RGB, etc. Other parameters are set according to requirements.

- 3) Complete hardware related operations

To map related registers and complete hardware initialization, specific hardware needs to be combined.

- 4) Use the **register\_framebuffer** function to register the **fb\_info** structure variable with the kernel

This step can also be understood as associating the operation function interfaces in **\*fb.c** and **fbmem.c**.

## Part 25.2: VDMA

VMDA is an IP core provided by xilinx, which is the key to the work of the display device on the ZYNQ device.

What are the benefits of **VDMA**? **VDMA** can easily implement double buffering or multiple buffering mechanisms. The double buffering mechanism solves at least the undesirable effects of screen flickering and tearing. Earlier we talked about the frame buffer. The content in the frame buffer will be directly displayed on the screen. When the data in the frame buffer is not ready, the image on the screen will be damaged. The double buffering adds a **back buffer** to the frame buffer. The data is prepared in the **back buffer**, and then the display of the front and back buffers is exchanged. That is, the original front buffer becomes the back buffer to prepare the next display data, and the back buffer becomes the front buffer to display the image

On the hardware, **VDMA** accesses **DDR** through the **AXI\_HP** interface. Essentially, **VDMA** is an **IP** that transports data, which provides convenience for data removal from **DDR**. The **VDMA** data interface is divided into read and write channels. We can write **AXI-Stream** data streams to **DDR** through the channels. The data in **DDR** can also be read out in **AXI-Stream** data stream format. When data enters **DDR**, we need to store the data in the frame buffer. **VDMA** can control 32 frame buffers, so it can be sent to realize multiple buffers.

**VDMA** is mainly composed of control and status registers, data handling modules and line buffers. Data in and out of DDR must be buffered by the row, and then the data is read and written by the data handling module. The working status of VDMA can be read through the status register.

**VDMA** has the following interfaces:

**AXI-lite:** PS configures **VDMA** through this interface;

**AXI Memory Map write:** Map to memory write;

**AXI Memory Map read:** Map to memory read;

**AXI Stream Write (S2MM):** AXI Stream video stream writes images;

**AXI Stream Read (MM2S):** AXI Stream video stream reads images.

**VDMA** also provides register operations. For details, please refer to the instructions provided by **xilinx**. The use of VDMA in vivado can refer to the touch screen experiment in **course 4**.

Using **VDMA** in Linux actually **xilinx** has already been implemented. In the file **/drivers/dma/xilinx/xilinx\_dma.c**, **axidma**, **cdma** and **vdma** supported by **zynq** are implemented. We can use this driver directly, let's take a look at this section in the device tree:

```
1. &amba_pl {  
2.  
3.     ax_encoder_0:ax_encoder {  
4.         compatible = "ax_lcd,drm-encoder";  
5.     };  
6.  
7.     xilinx_drm {  
8.         compatible = "xlnx,drm";  
9.         xlnx,vtc = <&v_tc_0>;  
10.        xlnx,connector-type = "HDMI-A";  
11.        xlnx,encoder-slave = <&ax_encoder_0>;  
12.        clocks = <&axi_dynclk_0>;  
13.        planes {  
14.            xlnx,pixel-format = "argb8888";  
15.            plane0 {  
16.                dmas = <&axi_vdma_0 0>;  
17.                dma-names = "dma";
```

```
18.          };
19.      };
20.  };
21.};
```

This section of the device tree is captured in the touch screen routine of course4. **Line 16**, `dmas = <&axi_vdma_0 0>`, this is obviously a reference to the node `axi_vdma_0`, where is this node? Search for `axi_vdma_0` in the `petalinux` project, you can find a device tree header file of `pl.dtsi`, in which we found the following nodes:

```
axi_vdma_0: dma@43000000 {
    #dma-cells = <1>;
    clock-names = "s_axi_lite_aclk", "m_axi_mm2s_aclk", "m_axi_mm2s_aclk";
    clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>;
    compatible = "xlnx,axi-vdma-1.00.a";
    interrupt-parent = <&intc>;
    interrupts = <0 29 4>;
    reg = <0x43000000 0x10000>;
    xlnx,addrwidth = <0x20>;
    xlnx,flush-fsync = <0x1>;
    xlnx,num-fstores = <0x1>;
    dma-channel@43000000 {
        compatible = "xlnx,axi-vdma-mm2s-channel";
        interrupts = <0 29 4>;
        xlnx,datawidth = <0x18>;
        xlnx,device-id = <0x0>;
        xlnx,genlock-mode ;
        xlnx,include-dre ;
    };
};
```

However, to find this node, apart from being able to find the source code of the `vdma` driver ([/drivers/dma/xilinx/xilinx\\_dma.c](#)) through the `compatible` attribute, it is meaningless for us to use, unless we are interested in analyzing the source code of the `vdma` driver.

Looking back at the device tree of the reference node, we said that `vdma` is the key to realize `zynqmp` display. Then the driver corresponding to this device tree should be the actual `lcd` driver that replaces the `fb` framework on `zynqmp`.

### Part 25.3: DRM framework

Through the device tree `compatible = "xlnx,drm"` intercepted above, it is not difficult to find that the `drm` framework is used on

---

**ZYNQ** to realize **Lcd** display. The **framebuffer** framework cannot handle **GPU**-based **3D** acceleration requirements, nor can it handle the situation where multiple programs access the video card cooperatively. **DRM (Direct Rendering Manager)** came into being. **DRM** has exclusive access to the video card, and it is responsible for initializing and maintaining the command queue, video card and other hardware resources. To use **GPU** programs to transmit requests to **DRM**, **DRM** arbitrates to avoid conflicts. At the same time, **DRM** is also responsible for the issue of **GPU** switching.

**DRM** is used by the **Linux** kernel to manage display output and buffer allocation. **DRM** consists of two parts: One is the kernel state subsystem, which encapsulates the hardware operation in a layer. The second is to provide a user-mode **libdrm** library, and application program can directly operate the **API** in the library, such as **ioctl** or use the interface provided by the **framebuffer** to perform display-related operations. The specific use of **libdrm** library will not be discussed.

**DRM** covers a lot of problems that needed to be handled in user space before, such as graphical execution manager **GEM**, kernel mode setting **KMS**, these are all **DRM** subsystems.

**GEM** (Graphic Execution Manager) is mainly responsible for the allocation of video memory.

**KMS** (Kernel Mode Setting) is mainly responsible for screen update and setting display parameters (including resolution, refresh rate, etc.).

**KMS** features include these blocks: **Framebuffer**, **CRTC**, **Planes**, **Encoder**, **Connector**.

**Framebuffer** and the **fb** frame mentioned above actually have the same meaning. Both the driver and the application can access it. You

---

need to format it before using it and set the resolution, color and other options. It is only used to describe video memory information, not responsible for the allocation of video memory

**CRTC** is the source of **RGB** signal. The cathode radio tube is connected to the **frame buffer** internally and the **encoder** externally. He will scan the content in **fb** and superimpose the content in **planes** and send it to the **encoder**. Each **CRTC** must have at least one **plane**.

**Planes** is also a memory address. The difference from **framebuffer** is that it provides a high-speed channel for video, which can be superimposed on **framebuffer**, and there can be multiple **planes**.

The **encoder** is to convert the pixel data into the signals required by the display. Different display devices require different electrical signals, such as **DVID**, **VGA**, **MIPI**, etc.

The **connector** usually corresponds to the connectors on the hardware such as **VGA**, **HDMI**, etc. At the same time, it also saves the related information of the display device connected to the connector, such as connection status, **DPMS** status, etc.

The **DRM** driver framework, that is, the encapsulation of the above-mentioned features, is just a brief overview here. The specific implementation of **DRM** can be said to be quite complicated, which is not clear in a few words. Fortunately, **xilinx** also provides us with a complete set of **drm** drivers. In the **/driver/gpu/drm/xilinx** folder, we can use it directly. The method used is to add the corresponding node in the device tree, the specific method can refer to the documentation in the kernel

**“\Documentation\devicetree\bindings\drm\xilinx\xilinx\_drm.txt”**.

Let's look back at the device tree code selected in the previous section. The node **axi\_vdma\_0** that we paid attention to is the

attribute under the node **plane0**. **plane0** is the video channel we mentioned above, that is, the cache channel 0 corresponding to VDMA Corresponds to the **plane0** video channel.

The DRM driver is quite complex, and there are many modules designed. It is impossible to make it clear in a few words here. To achieve LCD display, it is the most efficient to refer to the driver code implementation of other display devices in xilinx. Here we refer to the sdi display modification of xilinx to get an LCD display driver as follows:

```
1. #include <drm/drm_atomic_helper.h>
2. #include <drm/drm_crtc_helper.h>
3. #include <drm/drmP.h>
4. #include <drm/drm_probe_helper.h>
5. #include <linux/clk.h>
6. #include <linux/component.h>
7. #include <linux/device.h>
8. #include <linux/of_device.h>
9. #include <linux/of_graph.h>
10. #include <linux/phy/phy.h>
11. #include <video/videomode.h>
12.
13. #define PIXELS_PER_CLK 2
14.
15. struct alinx_lcd_t {
16.     struct drm_encoder encoder;
17.     struct drm_connector connector;
18.     struct device *dev;
19.
20.     u32 mode_flags;
21.     struct drm_display_mode video_mode;
22.
23.     struct drm_property *sdi_mode;
24.     u32 sdi_mod_prop_val;
25.     struct drm_property *height_out;
26.     u32 height_out_prop_val;
27.     struct drm_property *width_out;
28.     u32 width_out_prop_val;
29.     struct drm_property *in_fmt;
30.     u32 in_fmt_prop_val;
31.     struct drm_property *out_fmt;
32.     u32 out_fmt_prop_val;
33.     struct drm_property *is_frac_prop;
34.     bool is_frac_prop_val;
35. };
36.
37. static const struct drm_display_mode alinx_lcd_001_mode = {
38.     .clock = 33260,
39.     .hdisplay = 800,
40.     .hsync_start = 800 + 40,
41.     .hsync_end = 800 + 40 + 128,
42.     .htotal = 800 + 40 + 128 + 88,
43.     .vdisplay = 480,
44.     .vsync_start = 480 + 10,
45.     .vsync_end = 480 + 10 + 2,
46.     .vtotal = 480 + 10 + 2 + 33,
47.     .vrefresh = 60,
48.     .flags = DRM_MODE_FLAG_NHSYNC | DRM_MODE_FLAG_NVSYNC,
49.     .type = 0,
50.     .name = "800x480",
```

```
51. };
52.
53. static int alinx_lcd_atomic_set_property(struct drm_connector *connector
54. ,
55.                                     struct drm_connector_state *state,
56.                                     struct drm_property *property, uint64_t val
57. )
58. {
59.     struct alinx_lcd_t *sdi = connector_to_sdi(connector);
60.     if (property == sdi->sdi_mode)
61.         sdi->sdi_mod_prop_val = (unsigned int)val;
62.     else if (property == sdi->is_frac_prop)
63.         sdi->is_frac_prop_val = !!val;
64.     else if (property == sdi->height_out)
65.         sdi->height_out_prop_val = (unsigned int)val;
66.     else if (property == sdi->width_out)
67.         sdi->width_out_prop_val = (unsigned int)val;
68.     else if (property == sdi->in_fmt)
69.         sdi->in_fmt_prop_val = (unsigned int)val;
70.     else if (property == sdi->out_fmt)
71.         sdi->out_fmt_prop_val = (unsigned int)val;
72.     else
73.         return -EINVAL;
74.     return 0;
75. }
76. static int alinx_lcd_atomic_get_property(struct drm_connector *connector
77. ,
78.                                     const struct drm_connector_state *state,
79.                                     struct drm_property *property, uint64_t *va
80. )
81. {
82.     struct alinx_lcd_t *sdi = connector_to_sdi(connector);
83.     if (property == sdi->sdi_mode)
84.         sdi->sdi_mod_prop_val = (unsigned int)val;
85.     else if (property == sdi->is_frac_prop)
86.         sdi->is_frac_prop_val = !!val;
87.     else if (property == sdi->height_out)
88.         sdi->height_out_prop_val = (unsigned int)val;
89.     else if (property == sdi->width_out)
90.         sdi->width_out_prop_val = (unsigned int)val;
91.     else if (property == sdi->in_fmt)
92.         sdi->in_fmt_prop_val = (unsigned int)val;
93.     else if (property == sdi->out_fmt)
94.         sdi->out_fmt_prop_val = (unsigned int)val;
95.     else
96.         return -EINVAL;
97.     return 0;
98. }
99. static int alinx_lcd_drm_add_modes(struct drm_connector *connector)
100. {
101.     int num_modes = 0;
102.     struct drm_display_mode *mode;
103.     struct drm_device *dev = connector->dev;
104.     mode = drm_mode_duplicate(dev, &alinx_lcd_001_mode);
105.     drm_mode_probed_add(connector, mode);
106.     num_modes++;
```

```
106.  
107.         return num_modes;  
108. }  
109.  
110. static enum drm_connector_status  
111. alinx_lcd_detect(struct drm_connector *connector, bool force)  
112. {  
113.         return connector_status_connected;  
114. }  
115.  
116. static void alinx_lcd_connector_destroy(struct drm_connector *connecto  
     r)  
117. {  
118.         drm_connector_unregister(connector);  
119.         drm_connector_cleanup(connector);  
120.         connector->dev = NULL;  
121. }  
122.  
123. static const struct drm_connector_funcs alinx_lcd_connector_funcs = {  
  
124.         .detect = alinx_lcd_detect,  
125.         .fill_modes = drm_helper_probe_single_connector_modes,  
126.         .destroy = alinx_lcd_connector_destroy,  
127.         .atomic_duplicate_state = drm_atomic_helper_connector_duplicat  
     e_state,  
128.         .atomic_destroy_state = drm_atomic_helper_connector_destroy_st  
     ate,  
129.         .reset = drm_atomic_helper_connector_reset,  
130.         .atomic_set_property = alinx_lcd_atomic_set_property,  
131.         .atomic_get_property = alinx_lcd_atomic_get_property,  
132. };  
133.  
134. static struct drm_encoder *  
135. alinx_lcd_best_encoder(struct drm_connector *connector)  
136. {  
137.         return &(connector_to_sdi(connector)->encoder);  
138. }  
139.  
140. static int alinx_lcd_get_modes(struct drm_connector *connector)  
141. {  
142.         return alinx_lcd_drm_add_modes(connector);  
143. }  
144.  
145. static struct drm_connector_helper_funcs alinx_lcd_connector_helper_fu  
     ncs = {  
146.         .get_modes = alinx_lcd_get_modes,  
147.         .best_encoder = alinx_lcd_best_encoder,  
148. };  
149.  
150. static void alinx_lcd_drm_connector_create_property(struct drm_connect  
     or *base_connector)  
151. {  
152.         struct drm_device *dev = base_connector->dev;  
153.         struct alinx_lcd_t *sdi = connector_to_sdi(base_connector);  
154.  
155.         sdi->is_frac_prop = drm_property_create_bool(dev, 0, "is_frac");  
156.         sdi->sdi_mode = drm_property_create_range(dev, 0, "sdi_mode", 0, 5  
     );
```

```
157.     sdi->height_out = drm_property_create_range(dev, 0, "height_out",
158.         2, 4096);
159.     sdi->width_out = drm_property_create_range(dev, 0, "width_out", 2,
160.         4096);
161.     sdi->in_fmt = drm_property_create_range(dev, 0, "in_fmt", 0, 16384
162.         );
163.     sdi->out_fmt = drm_property_create_range(dev, 0, "out_fmt", 0, 163
164.         84);
165. }
166. static void alinx_lcd_drm_connector_attach_property(struct drm_connect
167. or *base_connector)
168. {
169.     struct alinx_lcd_t *sdi = connector_to_sdi(base_connector);
170.     struct drm_mode_object *obj = &base_connector->base;
171.     if (sdi->sdi_mode)
172.         drm_object_attach_property(obj, sdi->sdi_mode, 0);
173.     if (sdi->is_frac_prop)
174.         drm_object_attach_property(obj, sdi->is_frac_prop, 0);
175.     if (sdi->height_out)
176.         drm_object_attach_property(obj, sdi->height_out, 0);
177.     if (sdi->width_out)
178.         drm_object_attach_property(obj, sdi->width_out, 0);
179.     if (sdi->in_fmt)
180.         drm_object_attach_property(obj, sdi->in_fmt, 0);
181.     if (sdi->out_fmt)
182.         drm_object_attach_property(obj, sdi->out_fmt, 0);
183. }
184. static int alinx_lcd_create_connector(struct drm_encoder *encoder)
185. {
186.     struct alinx_lcd_t *sdi = encoder_to_sdi(encoder);
187.     struct drm_connector *connector = &sdi->connector;
188.     int ret;
189.
190.     connector->interlace_allowed = true;
191.     connector->doublescan_allowed = true;
192.
193.     connector->interlace_allowed = true;
194.     connector->doublescan_allowed = true;
195.
196.     ret = drm_connector_init(encoder->dev, connector,
197.                             &alinx_lcd_connector_funcs,
198.                             DRM_MODE_CONNECTOR_Unknown);
199.     if (ret) {
200.         dev_err(sdi->dev, "Failed to initialize connector with
201.                 drm\n");
202.         return ret;
203.     }
204.     drm_connector_helper_add(connector, &alinx_lcd_connector_helpe
205. r_funcs);
206.     drm_connector_register(connector);
207.     drm_connector_attach_encoder(connector, encoder);
208.     alinx_lcd_drm_connector_create_property(connector);
209.     alinx_lcd_drm_connector_attach_property(connector);
```

```
209.  
210.         return 0;  
211. }  
212.  
213. static void alinx_lcd_encoder_atomic_mode_set(struct drm_encoder *encoder,  
214.                                                 struct drm_crtc_state *c  
215.                                                 struct drm_connector_state *connector_state)  
216. {  
217.     struct alinx_lcd_t *sdi = encoder_to_sdi(encoder);  
218.     struct drm_display_mode *adjusted_mode = &crtc_state->adjusted  
219.     _mode;  
220.     struct videomode vm;  
221.     u32 sdidx_blank, vtc_blank;  
222.     vm.hactive = adjusted_mode->hdisplay / PIXELS_PER_CLK;  
223.     vm.hfront_porch = (adjusted_mode->hsync_start - adjusted_mode->hdisplay) / PIXELS_PER_CLK;  
224.     vm.hback_porch = (adjusted_mode->htotal - adjusted_mode->hsync_end) / PIXELS_PER_CLK;  
225.     vm.hsync_len = (adjusted_mode->hsync_end - adjusted_mode->hsync_start) / PIXELS_PER_CLK;  
226.  
227.     vm.vactive = adjusted_mode->vdisplay;  
228.     vm.vfront_porch = adjusted_mode->vsync_start - adjusted_mode->vdisplay;  
229.     vm.vback_porch = adjusted_mode->vtotal - adjusted_mode->vsync_end;  
230.     vm.vsync_len = adjusted_mode->vsync_end - adjusted_mode->vsync_start;  
231.     vm.flags = 0;  
232.     if (adjusted_mode->flags & DRM_MODE_FLAG_INTERLACE)  
233.         vm.flags |= DISPLAY_FLAGS_INTERLACED;  
234.     if (adjusted_mode->flags & DRM_MODE_FLAG_PHSYNC)  
235.         vm.flags |= DISPLAY_FLAGS_HSYNC_LOW;  
236.     if (adjusted_mode->flags & DRM_MODE_FLAG_PVSYNC)  
237.         vm.flags |= DISPLAY_FLAGS_VSYNC_LOW;  
238.  
239.     do {  
240.         sdidx_blank = (adjusted_mode->hsync_start -  
241.                         adjusted_mode->hdisplay) +  
242.                         (adjusted_mode->hsync_end -  
243.                         adjusted_mode->hsync_start) +  
244.                         (adjusted_mode->htotal -  
245.                         adjusted_mode->hsync_end);  
246.  
247.         vtc_blank = (vm.hfront_porch + vm.hback_porch +  
248.                         vm.hsync_len) * PIXELS_PER_CLK;  
249.  
250.         if (vtc_blank != sdidx_blank)  
251.             vm.hfront_porch++;  
252.     } while (vtc_blank < sdidx_blank);  
253.  
254.     vm.pixelclock = adjusted_mode->clock * 1000;  
255.  
256.     sdi->video_mode.vdisplay = adjusted_mode->vdisplay;  
257.     sdi->video_mode.hdisplay = adjusted_mode->hdisplay;
```

```
258.         sdi->video_mode.vrefresh = adjusted_mode->vrefresh;
259.         sdi->video_mode.flags = adjusted_mode->flags;
260.     }
261.
262. static void alinx_lcd_commit(struct drm_encoder *encoder)
263. {
264.     struct alinx_lcd_t *sdi = encoder_to_sdi(encoder);
265. }
266.
267. static void alinx_lcd_disable(struct drm_encoder *encoder)
268. {
269.     struct alinx_lcd_t *sdi = encoder_to_sdi(encoder);
270. }
271.
272. static const struct drm_encoder_helper_funcs alinx_lcd_encoder_helper_
273.     funcs = {
274.         .atomic_mode_set    = alinx_lcd_encoder_atomic_mode_set,
275.         .enable            = alinx_lcd_commit,
276.         .disable           = alinx_lcd_disable,
277.     };
278. static const struct drm_encoder_funcs alinx_lcd_encoder_funcs = {
279.     .destroy = drm_encoder_cleanup,
280. };
281.
282. static int alinx_lcd_bind(struct device *dev, struct device *master,
283.                           void *data)
284. {
285.     struct alinx_lcd_t *sdi = dev_get_drvdata(dev);
286.     struct drm_encoder *encoder = &sdi->encoder;
287.     struct drm_device *drm_dev = data;
288.     int ret;
289.
290.     encoder->possible_crtcs = 1;
291.
292.     drm_encoder_init(drm_dev, encoder, &alinx_lcd_encoder_funcs,
293.                      DRM_MODE_ENCODER_TMDS, NULL);
294.
295.     drm_encoder_helper_add(encoder, &alinx_lcd_encoder_helper_func_
296.     s);
297.     ret = alinx_lcd_create_connector(encoder);
298.     if (ret) {
299.         dev_err(sdi->dev, "fail creating connector, ret = %d\n",
300.                 ret);
301.         drm_encoder_cleanup(encoder);
302.     }
303.     return ret;
304.
305. static void alinx_lcd_unbind(struct device *dev, struct device *master
306.                           ,
307.                           void *data)
308. {
309.     struct alinx_lcd_t *sdi = dev_get_drvdata(dev);
310.     drm_encoder_cleanup(&sdi->encoder);
311.     drm_connector_cleanup(&sdi->connector);
312. }
```

```
313.  
314. static const struct component_ops alinx_lcd_component_ops = {  
315.     .bind    = alinx_lcd_bind,  
316.     .unbind   = alinx_lcd_unbind,  
317. };  
318.  
319. static int alinx_lcd_probe(struct platform_device *pdev)  
320. {  
321.     struct device *dev = &pdev->dev;  
322.     struct alinx_lcd_t *sdi;  
323.     int ret;  
324.     struct device_node *ports, *port;  
325.     u32 nports = 0, portmask = 0;  
326.  
327.     sdi = devm_kzalloc(dev, sizeof(*sdi), GFP_KERNEL);  
328.     if (!sdi)  
329.         return -ENOMEM;  
330.  
331.     sdi->dev = dev;  
332.  
333.     platform_set_drvdata(pdev, sdi);  
334.  
335.     ports = of_get_child_by_name(sdi->dev->of_node, "ports");  
336.     if (!ports) {  
337.         dev_dbg(dev, "Searching for port nodes in device node.  
\\n");  
338.         ports = sdi->dev->of_node;  
339.     }  
340.  
341.     for_each_child_of_node(ports, port) {  
342.         struct device_node *endpoint;  
343.         u32 index;  
344.  
345.         if (!port->name || of_node_cmp(port->name, "port")) {  
346.             dev_dbg(dev, "port name is null or node name i  
s not port!\\n");  
347.             continue;  
348.         }  
349.  
350.         endpoint = of_get_next_child(port, NULL);  
351.         if (!endpoint) {  
352.             dev_err(dev, "No remote port at %s\\n", port->n  
ame);  
353.             of_node_put(endpoint);  
354.             ret = -EINVAL;  
355.             return ret;  
356.         }  
357.  
358.         of_node_put(endpoint);  
359.  
360.         ret = of_property_read_u32(port, "reg", &index);  
361.         if (ret) {  
362.             dev_err(dev, "reg property not present - %d\\n"  
, ret);  
363.             return ret;  
364.         }  
365.  
366.         portmask |= (1 << index);
```

```
367.                 nports++;
368.             }
369.         }
370.
371.         if (nports == 1 && portmask & 0x1) {
372.             dev_dbg(dev, "no ancillary port\n");
373.         } else {
374.             dev_err(dev, "Incorrect dt node!\n");
375.             ret = -EINVAL;
376.             return ret;
377.         }
378.
379.         pdev->dev.platform_data = &sdi->video_mode;
380.
381.         ret = component_add(dev, &alinx_lcd_component_ops);
382.
383.         return ret;
384.     }
385.
386. static int alinx_lcd_remove(struct platform_device *pdev)
387. {
388.     component_del(&pdev->dev, &alinx_lcd_component_ops);
389.
390.     return 0;
391. }
392.
393. static const struct of_device_id alinx_lcd_of_match[] = {
394.     { .compatible = "ax-drm-encoder" },
395.     { }
396. };
397. MODULE_DEVICE_TABLE(of, alinx_lcd_of_match);
398.
399. static struct platform_driver sdi_tx_driver = {
400.     .probe = alinx_lcd_probe,
401.     .remove = alinx_lcd_remove,
402.     .driver = {
403.         .name = "alinx-lcd-001",
404.         .of_match_table = alinx_lcd_of_match,
405.     },
406. };
407.
408. module_platform_driver(sdi_tx_driver);
409.
410. MODULE_AUTHOR("guzy<gzy@alinx.com>");
411. MODULE_DESCRIPTION("alinx lcd driver");
412. MODULE_LICENSE("GPL v2");
```

From the probe function, let's forget about the previous settings. Line 381 calls the component\_add function to add an operation function, alinx\_lcd\_bind and alinx\_lcd\_unbind.

In the **alink\_lcd\_bind** function, it can be clearly seen that this driver actually adds an **encoder** and **connector**, which means that the LCD display acts as a **connector** and **encoder** in the entire **drm** system. The vdma mentioned in the previous section corresponds to crtc and fb.

The corresponding device tree is as follows:

```
&amba {  
    ax_lcd_encoder: ax_lcd_encoder {  
  
        compatible = "ax-drm-encoder";  
        ports {  
            #address-cells = <1>;  
            #size-cells = <0>;  
            encoder_lcd_port: port@0 {  
                reg = <0>;  
                lcd_encoder: endpoint {  
                    remote-endpoint = <&dmaengine_crtc>;  
                };  
            };  
        };  
    };  
};
```