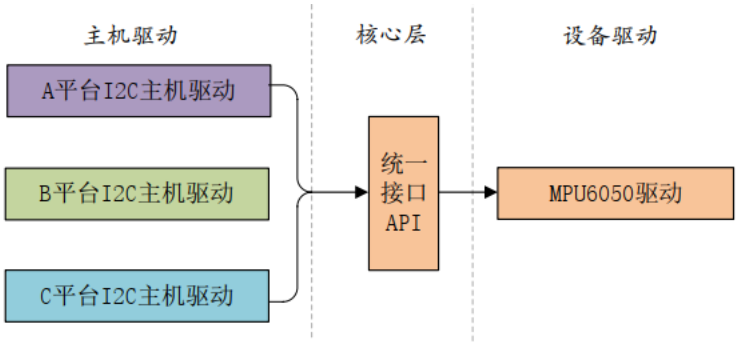
[L23]Platform 设备驱动

对于Linux这么庞大的系统来说,他除了包含系统本身,还包含很多各种驱动,如果说驱动要挨个开发也太麻烦了,开发周期太长,所以都现在Linux都是基于Platform进行分离的,我们之前使用了GPIO子系统,那个严格来说还不算Platform但是却像Platform那样.



所以实际上一套环境里是包含platform_device以及platform_driver的,我们单片机开发TFT显示时候也是显示库和打点的底层函数,是同一个道理,而只需要我们的打点函数和显示库匹配,那么就会完成显示了.特别注意,这个驱动分离,是针对内核层面的,对于用户而言是没什么区别的.Linux内核里可以说一切都是和各种框架打交道(除了少数),只要掌握了,可以说Linux内核开发简直是畅通.

在Linux中由platform_device调用platform_driver来完成工作,而platform_device现在是不推荐的,推荐使用设备树的方法,但是对于驱动编写的人来说,最好是两者都实现,不过由于Linux是罗马一样,一步一步起来的,所以我们也先用普通驱动,再用dts驱动,而实际项目中,最好是都实现.

实际干活的platform_driver格式如下:

```
#incClude <linux/module.h>
#include <linux/of_gpio.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>
#include <linux/platform_device.h>

static int led_probe(struct platform_device *pdev)
{
  /* 当驱动匹配成功,这个函数就会执行,和之前写的驱动的probe没什么区别. */

  return ret;
}

static int led_remove(struct platform_device *dev)
{
  /* 驱动卸载时候执行,和以前的设备驱动没区别. */

  return 0;
}

/* 设备树匹配列表 */
static const struct of_device_id led_of_match[] = {
  { .compatible = "taterli,led" },
  { /* 占位符 */ }
};

static struct platform_driver led_driver = {
  .driver = {
    .name     = "taterli-led", /* 传统方法匹配,他也可以匹配列表. */
    .of_match_table = led_of_match, /* 设备树方法匹配 */
  },
  .probe    = led_probe,
  .remove   = led_remove,
};

static int __init led_driver_init(void)
{
  return platform_driver_register(&led_driver);
}
```

```
static void __exit led_driver_exit(void)
{
  platform_driver_unregister(&led_driver);
}

module_init(led_driver_init);
module_exit(led_driver_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("LED GPIO");
MODULE_LICENSE("GPL");
```

如何通过传统方式绑定呢,那就要匹配driver中的name了,比如接下来我们要用的led驱动相关的device可以这么写.

```
#include <linux/module.h>
#include <linux/platform_device.h>

#define GPIO_DATA_2   0xE000A048
#define GPIO_DIRM_2   0xE000A284
#define GPIO_OUTEN_2  0xE000A288
#define GPIO_INTDIS_2 0xE000A294
#define APER_CLK_CTRL 0xF800012C

static void led_release(struct device *dev)
{

}

static struct resource led_resources[] = {
  [0] = {
    .start  = GPIO_DATA_2,
    .end  = GPIO_DATA_2 + 3,
    .flags  = IORESOURCE_MEM,
  },
  [1] = {
    .start  = GPIO_DIRM_2,
    .end  = GPIO_DIRM_2 + 3,
    .flags  = IORESOURCE_MEM,
  },
  [2] = {
    .start  = GPIO_OUTEN_2,
    .end  = GPIO_OUTEN_2 + 3,
    .flags  = IORESOURCE_MEM,
  },
  [3] = {
    .start  = GPIO_INTDIS_2,
    .end  = GPIO_INTDIS_2 + 3,
    .flags  = IORESOURCE_MEM,
  },
  [4] = {
    .start  = APER_CLK_CTRL,
    .end  = APER_CLK_CTRL + 3,
    .flags  = IORESOURCE_MEM,
  },
};

static struct platform_device led_device = {
  .name = "taterli-led",
  .id = -1,
  .dev = {
    /* 方法都是可选可选实现的. */
    .release = &led_release,
  },
  .num_resources = ARRAY_SIZE(led_resources),
  .resource = led_resources,
};

static int __init led_device_init(void)
{
  return platform_device_register(&led_device);
}

static void __exit led_device_exit(void)
{
  platform_device_unregister(&led_device);
}

module_init(led_device_init);
```

```
module_exit(led_device_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("LED GPIO Device");
MODULE_LICENSE("GPL");
```

如果是设备树就简单一些,熟悉吗,就是之前的LED的dts.

```
led {
    compatible = "taterli,led";
    status = "okay";
    default-state = "on";

    led-gpio = <&gpio0 54 GPIO_ACTIVE_HIGH>;
}
```

让这个框架,套到之前的驱动里,传统不用gpio子系统的方法要这么修改.

1. 引入头文件.
2. 修改led_init的ioremap,使得地址从device声明中取出.
3. 修改module_init和module_exit方法至platform相关函数.
4. 修改原来的init和exit方法的定义和返回.
5. 添加platform_driver结构体.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/platform_device.h>

#define KERNEL_LED_DEVIE_CNT 1
#define KERNEL_LED_NAME "kernel_led"

static void __iomem *DATA;
static void __iomem *DIRM;
static void __iomem *OUTEN;
static void __iomem *INTDIS;
static void __iomem *APER;

struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
};

static struct kernel_led_dev dev;

/* 设备打开时候会被调用 */
static int led_open(struct inode *inode, struct file *filp)
{
    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t led_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offset)
{
    int ret;
```

```
    char kbuf[1];

    if (cnt != 1)
    {
        return -EFAULT;
    }

    kbuf[0] = readl(DATA) & EMIO_PIN;
    ret = copy_to_user(buf, kbuf, cnt);
    if (ret)
    {
        /* 复制失败了 */
        return -EFAULT;
    }

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t led_write(struct file *filp, const char __user *buf, size_t cnt, loff_t *offset)
{
    int ret;
    int val;
    char kbuf[1];

    if (cnt != 1)
    {
        return -EFAULT;
    }

    ret = copy_from_user(kbuf, buf, cnt);
    if (ret)
    {
        /* 复制失败了 */
        return -EFAULT;
    }

    val = readl(DATA);
    if (kbuf[0] == 0)
    {
        val &= ~EMIO_PIN;
    }
    else
    {
        val |= EMIO_PIN;
    }

    writel(val, DATA);

    return 0;
}

/* 设备释放时候会被调用 */
static int led_release(struct inode *inode, struct file *filp)
{
    return 0;
}

static struct file_operations fops =
{
        .owner = THIS_MODULE,
        .open = led_open,
        .read = led_read,
        .write = led_write,
        .release = led_release,
};

static int led_init(struct platform_device *mdev)
{
    int i;
    int val;
    int ret;
    struct resource *res[5];

    for (i = 0; i < 5; i++)
    {
        res[i] = platform_get_resource(mdev, IORESOURCE_MEM, i);
        if (!res[i])
        {
            return -ENXIO;
        }
```

```
    }

    /* 寄存器映射 */
    DATA = ioremap(res[0]->start, resource_size(res[0]));
    DIRM = ioremap(res[1]->start, resource_size(res[1]));
    OUTEN = ioremap(res[2]->start, resource_size(res[2]));
    INTDIS = ioremap(res[3]->start, resource_size(res[3]));
    APER = ioremap(res[4]->start, resource_size(res[4]));

    /* 初始化 */
    val = readl(APER);
    val |= GPIO_CLK_EN;
    writel(val, APER);

    val = readl(INTDIS);
    val |= EMIO_PIN;
    writel(val, INTDIS);

    val = readl(DIRM);
    val |= EMIO_PIN;
    writel(val, DIRM);

    val = readl(OUTEN);
    val |= EMIO_PIN;
    writel(val, OUTEN);

    val = readl(DATA);
    val |= EMIO_PIN;
    writel(val, DATA);

    printk("APER reg 0x%08x\n", readl(APER));
    printk("INTDIS reg 0x%08x\n", readl(INTDIS));
    printk("DIRM reg 0x%08x\n", readl(DIRM));
    printk("OUTEN reg 0x%08x\n", readl(OUTEN));
    printk("DATA reg 0x%08x\n", readl(DATA));

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid, 0, KERNEL_LED_DEVIE_CNT, KERNEL_LED_NAME);
    if (ret)
    {
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev, &fops);

    ret = cdev_add(&dev.cdev, dev.devid, KERNEL_LED_DEVIE_CNT);
    if (ret)
    {
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE, KERNEL_LED_NAME);
    if (IS_ERR(dev.class))
    {
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }

    dev.device = device_create(dev.class, NULL, dev.devid, NULL, KERNEL_LED_NAME);
    if (IS_ERR(dev.device))
    {
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid, KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 寄存器取消映射 */
```

```
    iounmap(DATA);
    iounmap(DIRM);
    iounmap(OUTEN);
    iounmap(INTDIS);
    iounmap(APER);
    return ret;
}

static int led_exit(struct platform_device *mdev)
{
    device_destroy(dev.class, dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid, KERNEL_LED_DEVIE_CNT);

    iounmap(DATA);
    iounmap(DIRM);
    iounmap(OUTEN);
    iounmap(INTDIS);
    iounmap(APER);

    return 0;
}

static struct platform_driver led_driver = {
    .driver = {
        .name = "taterli-led",
    },
    .probe = led_init,
    .remove = led_exit,
};

static int __init led_driver_init(void)
{
    return platform_driver_register(&led_driver);
}

static void __exit led_driver_exit(void)
{
    platform_driver_unregister(&led_driver);
}

module_init(led_driver_init);
module_exit(led_driver_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

测试就是先加载drvier,再加载device,再运行用户空间测试,相信大家都会了,就不多说了,还不会的就得翻查前面学习,当然实际上模块加载顺序错了也没关系,但是最好还是按照推荐方法.

如果是dts方法,那么怎么修改呢,我们的dts已经用到了gpio子系统,因此我们从那个例子开始改起,与前面例子差不多,唯一区别是换成gpio子系统,多了一个匹配列表,并且不需要再检测status和compatible了,因为匹配代表检测OK,连节点查找都省了,所以他节点也可以随便是其他名字,只要他compatible属性正确就行.

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h>         /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h>    /* gpio子系统相关 */
#include <linux/platform_device.h>
```

```
#define KERNEL_LED_DEVIE_CNT 1
#define KERNEL_LED_NAME "kernel_led"

struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio;               /* gpio编号 */
};

static struct kernel_led_dev dev;

/* 设备打开时候会被调用 */
static int led_open(struct inode *inode, struct file *filp)
{
    filp->private_data = &dev;

    return 0;
}

/* 设备读取时候会被调用 */
static ssize_t led_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offset)
{
    int ret;
    char kbuf[1];

    if (cnt != 1)
    {
        return -EFAULT;
    }

    ret = gpio_get_value(dev.gpio);
    if (ret < 0)
    {
        return ret;
    }

    /* 不是高就是低! */
    kbuf[0] = ret;
    ret = copy_to_user(buf, kbuf, cnt);
    if (ret)
    {
        /* 复制失败了 */
        return -EFAULT;
    }

    return 0;
}

/* 设备写入时候会被调用 */
static ssize_t led_write(struct file *filp, const char __user *buf, size_t cnt, loff_t *offset)
{
    int ret;
    char kbuf[1];

    if (cnt != 1)
    {
        return -EFAULT;
    }

    ret = copy_from_user(kbuf, buf, cnt);
    if (ret)
    {
        /* 复制失败了 */
        return -EFAULT;
    }

    gpio_set_value(dev.gpio, kbuf[0] ? 1 : 0);

    return 0;
}

/* 设备释放时候会被调用 */
static int led_release(struct inode *inode, struct file *filp)
{
    return 0;
```

```
}

static struct file_operations fops =
    {
        .owner = THIS_MODULE,
        .open = led_open,
        .read = led_read,
        .write = led_write,
        .release = led_release,
};

static int led_init(struct platform_device *mdev)
{
    const char *str;
    int ret;

    /* 新增的从dts获取数据的过程 */
    dev.nd = mdev->dev.of_node;

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd, "led-gpio", 0);
    if (!gpio_is_valid(dev.gpio))
    {
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio, "taterli-kernel-led");
    if (ret < 0)
    {
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(dev.nd, "default-state", &str);
    if (ret < 0)
    {
        return -EINVAL;
    }

    if (!strcmp(str, "on"))
    {
        /* 设置输出和默认电平 */
        gpio_direction_output(dev.gpio, 1);
    }
    else if (!strcmp(str, "off"))
    {
        gpio_direction_output(dev.gpio, 0);
    }
    else
    {
        return -EINVAL;
    }

    /* 不需要寄存器映射了,因为有子系统! */

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid, 0, KERNEL_LED_DEVIE_CNT, KERNEL_LED_NAME);
    if (ret)
    {
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev, &fops);

    ret = cdev_add(&dev.cdev, dev.devid, KERNEL_LED_DEVIE_CNT);
    if (ret)
    {
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE, KERNEL_LED_NAME);
    if (IS_ERR(dev.class))
    {
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }
```

```
    dev.device = device_create(dev.class, NULL, dev.devid, NULL, KERNEL_LED_NAME);
    if (IS_ERR(dev.device))
    {
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid, KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

static int led_exit(struct platform_device *mdev)
{
    device_destroy(dev.class, dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid, KERNEL_LED_DEVIE_CNT);

    gpio_free(dev.gpio);

    return 0;
}

/* 匹配列表 */
static const struct of_device_id led_of_match[] = {
    {.compatible = "taterli,led"},
    {}
};

static struct platform_driver led_driver = {
    .driver = {
        .name = "taterl-led", /* 即使不用也要保留一个!  */
        .of_match_table = led_of_match,
    },
    .probe = led_init,
    .remove = led_exit,
};

static int __init led_driver_init(void)
{
    return platform_driver_register(&led_driver);
}

static void __exit led_driver_exit(void)
{
    platform_driver_unregister(&led_driver);
}

module_init(led_driver_init);
module_exit(led_driver_exit);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

不过,现在代码已经好几百行了,我们有很多每次都用,而且一眼就知道很通用的代码块,他们其实完全能合并成一块,回顾一下初始化中我们干了什么.

1. 申请一个设备号.
2. 初始化/注册/添加 cdev.

3. 创建类.
4. 创建设备.

但是哪种设备不是大概这些流程啊,其实这些是设备最原始的接口函数,实际上像led,按键这些,甚至屏幕,触摸这些,都有标准,快捷的初始化方法,但是我们要直到,用上面的四步可以创建任何字符设备,但是用简便方法,只能创建不太奇葩的设备.

接着继续看我们的写入方法,还要判断用户输入的是什么,比如1是点亮,0是灭掉,那么这个是约定吗?别人不可以用0是点亮,1是灭掉吗?能的,为了解决这个问题,那么一类设备就应该给一类定义,先试试在用户空间操作LED,通过系统自带的GPIO驱动.

```
xilinx@pynq:/sys/class/gpio/gpiochip906$ cat label
zynq_gpio
```

可以看到ZYNQ_GPIO其实是导出了的,他实际范围是GPIO906~GPIO1023(分别对应54个MIO和64个EMIO),GPIO906是GPIO控制器的名字,为什么范围这么定义,这只能说设计一开始就这样定下来的,比如我们要操作EMIO第二GPIO,即961号IO,操作前记得在PL端预先配置,然后试试这个IO的功能,大家不妨多测试一下.

```
root@pynq:/sys/class/gpio# echo 961 > /sys/class/gpio/export
root@pynq:/sys/class/gpio# ls
export  gpio961  gpiochip906  unexport
root@pynq:/sys/class/gpio# cd gpio961
root@pynq:/sys/class/gpio/gpio961# ls
active_low  device  direction  edge  power  subsystem  uevent  value
root@pynq:/sys/class/gpio/gpio961# cat direction
in
root@pynq:/sys/class/gpio/gpio961# echo out > direction
root@pynq:/sys/class/gpio/gpio961# echo 1 > value
root@pynq:/sys/class/gpio/gpio961# echo 0 > value
```

这是个GPIO,他只有1和0两种关系,这是约定的,内核中其实也有一个专门的led-class,他除了1和0,还支持亮度,系统上也有对应的文件夹,就/sys/class/led,不妨可以进去看看,属性有不同.

```
root@pynq:/sys/class/leds/mmc0::# ls
brightness  device  max_brightness  power  subsystem  trigger  uevent
root@pynq:/sys/class/leds/mmc0::# cat brightness
0
root@pynq:/sys/class/leds/mmc0::# echo 255 > brightness
```

能看到,他提供更多的属性,其中有亮度,其中约定中最大亮度是255,最小亮度是0,那么就算是个渐变灯,我们也能控制了,我们实际上也要实现LED相关的函数,说了那么多,那如何把上面的gpio子系统方法改成led-class方法.

引入头文件:

```
#include <linux/leds.h>
```

字符设备相关的全部换成led_classdev:

```
// 修改前

struct kernel_led_dev
{
    dev_t devid;
    struct cdev cdev;
    struct class *class;
    struct device *device;
    int major;
    int minor;
    struct device_node *nd; /* 设备节点 */
    int gpio;               /* gpio编号 */
};

// 修改后

struct kernel_led_dev
{
```

```
    struct led_classdev cdev; /* Led设备 */
    int gpio;               /* gpio编号 */
};
```

修改初始化函数,但是不再直接写cdev,另外绑定亮度设置的函数,一般要提供两个接口,我这里都是同一个了.

```
// 修改前

static int led_init(struct platform_device *mdev)
{
    const char *str;
    int ret;

    /* 新增的从dts获取数据的过程 */
    dev.nd = mdev->dev.of_node;

    /* IO当然也可以是一个数组 */
    dev.gpio = of_get_named_gpio(dev.nd, "led-gpio", 0);
    if (!gpio_is_valid(dev.gpio))
    {
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev.gpio, "taterli-kernel-led");
    if (ret < 0)
    {
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(dev.nd, "default-state", &str);
    if (ret < 0)
    {
        return -EINVAL;
    }

    if (!strcmp(str, "on"))
    {
        /* 设置输出和默认电平 */
        gpio_direction_output(dev.gpio, 1);
    }
    else if (!strcmp(str, "off"))
    {
        gpio_direction_output(dev.gpio, 0);
    }
    else
    {
        return -EINVAL;
    }

    /* 不需要寄存器映射了,因为有子系统! */

    /* 申请一个设备号 */
    ret = alloc_chrdev_region(&dev.devid, 0, KERNEL_LED_DEVIE_CNT, KERNEL_LED_NAME);
    if (ret)
    {
        goto alloc_fail;
    }
    dev.major = MAJOR(dev.devid);
    dev.minor = MINOR(dev.devid);

    dev.cdev.owner = THIS_MODULE;
    cdev_init(&dev.cdev, &fops);

    ret = cdev_add(&dev.cdev, dev.devid, KERNEL_LED_DEVIE_CNT);
    if (ret)
    {
        goto add_fail;
    }

    dev.class = class_create(THIS_MODULE, KERNEL_LED_NAME);
    if (IS_ERR(dev.class))
    {
        ret = PTR_ERR(dev.class);
        goto class_fail;
    }
```

```
    dev.device = device_create(dev.class, NULL, dev.devid, NULL, KERNEL_LED_NAME);
    if (IS_ERR(dev.device))
    {
        ret = PTR_ERR(dev.class);
        goto dev_fail;
    }

    return 0;

dev_fail:
    class_destroy(dev.class);

class_fail:
    cdev_del(&dev.cdev);

add_fail:
    unregister_chrdev_region(dev.devid, KERNEL_LED_DEVIE_CNT);

alloc_fail:
    /* 这里就清爽很多了,释放IO就行. */
    gpio_free(dev.gpio);
    return ret;
}

// 修改后

static int led_init(struct platform_device *mdev)
{
    struct kernel_led_dev *dev;
    struct led_classdev *led_cdev;
    const char *str;
    int ret;

    /* 为dev指针分配内存 */
    dev = devm_kzalloc(&mdev->dev, sizeof(struct kernel_led_dev), GFP_KERNEL);
    if (!dev)
        return -ENOMEM;

    platform_set_drvdata(mdev, dev);

    /* IO当然也可以是一个数组 */
    dev->gpio = of_get_named_gpio(mdev->dev.of_node, "led-gpio", 0);
    if (!gpio_is_valid(dev->gpio))
    {
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev->gpio, "taterli-kernel-led");
    if (ret < 0)
    {
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(mdev->dev.of_node, "default-state", &str);
    if (ret < 0)
    {
        return -EINVAL;
    }

    if (!strcmp(str, "on"))
    {
        /* 设置输出和默认电平 */
        gpio_direction_output(dev->gpio, 1);
    }
    else if (!strcmp(str, "off"))
    {
        gpio_direction_output(dev->gpio, 0);
    }
    else
    {
        return -EINVAL;
    }

    /* 初始化led_cdev变量 */
    led_cdev = &dev->cdev;
    led_cdev->name = "kernel_led";                                  /* 设置设备名字 */
    led_cdev->brightness = LED_OFF;                                 /* 设置LED初始亮度,LED_OFF就是0. */
```

```
    led_cdev->max_brightness = LED_FULL;                            /* 设置LED最大亮度.LED_FULL就是255,如果不是调光灯也可以写1,用户能读取这个知道该怎
    led_cdev->brightness_set = led_brightness_set;                  /* 亮度设置函数(不可打断) */
    led_cdev->brightness_set_blocking = led_brightness_set_blocking; /* 两端设置函数(可打断)*/

    /* 注册LED设备 */
    return led_classdev_register(&mdev->dev, led_cdev);
}
```

替换注销函数为led-class方式.

```
// 修改前

static int led_exit(struct platform_device *mdev)
{
    device_destroy(dev.class, dev.devid);

    class_destroy(dev.class);

    cdev_del(&dev.cdev);

    unregister_chrdev_region(dev.devid, KERNEL_LED_DEVIE_CNT);

    gpio_free(dev.gpio);

    return 0;
}

// 修改后

static int led_exit(struct platform_device *mdev)
{
    struct kernel_led_dev *led_data = platform_get_drvdata(mdev);
    led_classdev_unregister(&led_data->cdev);

    return 0;
}
```

设置亮度函数中,我们这里用到了一个静态内敛函数,这个在内核中很常见,就是只需要传入结构体的其中一个地址,就知道结构体的首地址.

```
static void led_brightness_set(struct led_classdev *led_cdev,
      enum led_brightness value)
{
  struct kernel_led_dev*led_data = container_of(led_cdev, struct kernel_led_dev, cdev);
  int level;

  if (value == LED_OFF)
    level = 0;
  else
    level = 1;

  gpio_set_value(led_data->gpio, level);
}

static int led_brightness_set_blocking(struct led_classdev *led_cdev,
      enum led_brightness value)
{
  led_brightness_set(led_cdev, value);
  return 0;
}
```

最后再把module_init和module_exit套成module_platform_driver,这样显得更加简洁,毕竟都是那两个函数.

```
// 修改前

static int __init led_driver_init(void)
{
    return platform_driver_register(&led_driver);
}

static void __exit led_driver_exit(void)
{
```

```
    platform_driver_unregister(&led_driver);
}

module_init(led_driver_init);
module_exit(led_driver_exit);

// 修改后(同时删除上面两个函数!)

module_platform_driver(led_driver);
```

把没用的重复包含的头文件去掉.

```
//修改前

#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>
#include <linux/cdev.h>
#include <linux/of.h>         /* dts操作相关 */
#include <linux/of_address.h> /* dts地址相关 */
#include <linux/of_gpio.h>    /* gpio子系统相关 */
#include <linux/platform_device.h>
#include <linux/leds.h>

// 修改后

#include <linux/module.h>
#include <linux/of_gpio.h>
#include <linux/platform_device.h>
#include <linux/leds.h>
```

删除所有传统的文件操作代码,以及全局定义的dev,因为在这里不需要我们再记录这些数据,我们在初始化时候已经devm_kzalloc了,最后文件整体就这个样子.

```
#include <linux/module.h>
#include <linux/of_gpio.h>
#include <linux/platform_device.h>
#include <linux/leds.h>

struct kernel_led_dev
{
    struct led_classdev cdev; /* Led设备 */
    int gpio;                 /* gpio编号 */
};

static void led_brightness_set(struct led_classdev *led_cdev,
                               enum led_brightness value)
{
    struct kernel_led_dev *led_data = container_of(led_cdev, struct kernel_led_dev, cdev);
    int level;

    if (value == LED_OFF)
        level = 0;
    else
        level = 1;

    gpio_set_value(led_data->gpio, level);
}

static int led_brightness_set_blocking(struct led_classdev *led_cdev,
                                       enum led_brightness value)
{
    led_brightness_set(led_cdev, value);
    return 0;
}

static int led_init(struct platform_device *mdev)
```

```
{
    struct kernel_led_dev *dev;
    struct led_classdev *led_cdev;
    const char *str;
    int ret;

    /* 为dev指针分配内存 */
    dev = devm_kzalloc(&mdev->dev, sizeof(struct kernel_led_dev), GFP_KERNEL);
    if (!dev)
        return -ENOMEM;

    platform_set_drvdata(mdev, dev);

    /* IO当然也可以是一个数组 */
    dev->gpio = of_get_named_gpio(mdev->dev.of_node, "led-gpio", 0);
    if (!gpio_is_valid(dev->gpio))
    {
        /* IO是独占资源,因此可能申请失败! */
        return -EINVAL;
    }

    /* 申请IO并给一个名字 */
    ret = gpio_request(dev->gpio, "taterli-kernel-led");
    if (ret < 0)
    {
        /* 除了返回EINVAL,也可以返回上一层传递的错误. */
        return ret;
    }

    ret = of_property_read_string(mdev->dev.of_node, "default-state", &str);
    if (ret < 0)
    {
        return -EINVAL;
    }

    if (!strcmp(str, "on"))
    {
        /* 设置输出和默认电平 */
        gpio_direction_output(dev->gpio, 1);
    }
    else if (!strcmp(str, "off"))
    {
        gpio_direction_output(dev->gpio, 0);
    }
    else
    {
        return -EINVAL;
    }

    /* 初始化led_cdev变量 */
    led_cdev = &dev->cdev;
    led_cdev->name = "kernel_led";                                  /* 设置设备名字 */
    led_cdev->brightness = LED_OFF;                                 /* 设置LED初始亮度,LED_OFF就是0. */
    led_cdev->max_brightness = LED_FULL;                            /* 设置LED最大亮度.LED_FULL就是255,如果不是调光灯也可以写1,用户能读取这个知道该怎
    led_cdev->brightness_set = led_brightness_set;                  /* 亮度设置函数(不可打断) */
    led_cdev->brightness_set_blocking = led_brightness_set_blocking; /* 两端设置函数(可打断)*/

    /* 注册LED设备 */
    return led_classdev_register(&mdev->dev, led_cdev);
}

static int led_exit(struct platform_device *mdev)
{
    struct kernel_led_dev *led_data = platform_get_drvdata(mdev);
    led_classdev_unregister(&led_data->cdev);

    return 0;
}

/* 匹配列表 */
static const struct of_device_id led_of_match[] = {
    {.compatible = "taterli,led"},
    {}};

static struct platform_driver led_driver = {
    .driver = {
        .name = "taterl-led",
        .of_match_table = led_of_match,
    },
    .probe = led_init,
    .remove = led_exit,
```

```
};

module_platform_driver(led_driver);

MODULE_AUTHOR("Taterli <admin@taterli.com>");
MODULE_DESCRIPTION("Led GPIO");
MODULE_LICENSE("GPL");
```

验证过程稍微有些改变,因为我们的LED具体操作到/sys/class/leds/kernel_led了,最后还要引用一句,如果为了实现一些常见的LED功能,还可以使用内核提供的gpio-leds驱动,这里给出一个例子,写入此设备树后,可以在系统的/sys/class/leds/hello操作自己的LED,如何操作?看标准文档就行了.

```
leds {
    compatible = "gpio-leds";

    hello {
      label = "hello";
      gpios = <&gpio0 55 0>;
      linux,default-trigger = "heartbeat";
    };
};
```

到现在,Linux的开发已经算入门了,其他开发也是各种platform,各种绑定对着做,但是Linux博大精深,绝对不是一两下能掌握透彻的,这是一个终生学习的过程.