[L25]Linux 杂项设备驱动

之前驱动里面,LED有LED的Class,蜂鸣器有蜂鸣器的Class,那么对于无法归类的设备,该怎么办?先考虑下misc,除非misc不能实现,再考虑直接写cdev驱动,整个注册都围绕一个结构体来做.

```
// 文件:include/linux/miscdevice.h
struct miscdevice  {
  int minor;
  const char *name;
  const struct file_operations *fops;
  struct list_head list;
  struct device *parent;
  struct device *this_device;
  const struct attribute_group **groups;
  const char *nodename;
  umode_t mode;
};

extern int misc_register(struct miscdevice *misc);
extern void misc_deregister(struct miscdevice *misc);
```

在这个头文件预定了一些MINOR,我们不应该使用,主设备号是已经规定是10的,所以剩下的设备号数量剩下200个出头,misc设备介于直接写cdev和标准类设备之间,需要自己实现fops,但是又节约很多API,我们还是从platform代码开始改起.

修改设备的结构体:

```
// 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 led_dev
{
```

```
    struct miscdevice mdev;
    int gpio; /* gpio编号 */
};
```

修改初始化过程:

```
//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)
{
```

```
    const char *str;
    int ret;

    /* 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)
    {
        gpio_free(dev.gpio);
        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
    {
        gpio_free(dev.gpio);
        return -EINVAL;
    }

    dev.mdev.name = "taterli-led";
    dev.mdev.minor = MISC_DYNAMIC_MINOR; /* 偷懒可以这么写,让系统分配没用的设备号. */
    dev.mdev.fops = &fops;

    misc_register(&dev.mdev);

    return ret;
}
```

修改移除过程:

```
// 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)
{
    misc_deregister(&dev.mdev);

    gpio_free(dev.gpio);

    return 0;
}
```

精简没用的头文件,注意要包含misc相关头文件miscdevice.h和用户空间访问uacess.h:

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

整体驱动文件:

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

struct led_dev
{
    struct miscdevice mdev;
    int gpio; /* gpio编号 */
```

```
};

static struct 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;

    /* 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)
    {
        gpio_free(dev.gpio);
        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
    {
        gpio_free(dev.gpio);
        return -EINVAL;
    }

    dev.mdev.name = "taterli-led";
    dev.mdev.minor = MISC_DYNAMIC_MINOR; /* 偷懒可以这么写,让系统分配没用的设备号. */
    dev.mdev.fops = &fops;

    misc_register(&dev.mdev);

    return ret;
}

static int led_exit(struct platform_device *mdev)
{
    misc_deregister(&dev.mdev);

    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");
```

现在使用misc既可以实现传统cdev方法一样的效果,也节约很多代码,除非真的有必要,否则misc能满足大多数情况,用户空间的测试程序和dts文件应该不用多说了吧,毕竟我们的read/write是一样的实现,值得注意的是字符设备名换成了dev.mdev.name中指示的.

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

int main(int argc,char **argv){
    int fd,ret;
    char buf[1];

    fd = open("/dev/taterli-led",O_RDWR);
    if(fd < 0){
        return -1;
    }

    for(;;){
        buf[0] = 0;
        ret = write(fd,buf,1);
        if(ret < 0){
            return -2;
        }

        ret = read(fd,buf,1);
        if(ret < 0){
            return -3;
        }
        printf("Current PL Led = %d \n\r",buf[0]);

        usleep(1000 * 1000);

        buf[0] = 1;
        ret = write(fd,buf,1);
        if(ret < 0){
            return -2;
        }

        ret = read(fd,buf,1);
        if(ret < 0){
            return -3;
```

```
        }
        printf("Current PL Led = %d \n\r",buf[0]);

        usleep(1000 * 1000);
    }
}
```

常用的类还有很多,一般情况下先检查有没有对应的标准类,没有再考虑通过这样的方法来实现.