【代码解析】W3关于GPIO灯与按钮

2017年6月13日
19:50

定义与注册：
./qcom-ipq40xx-ap.dk01.1.dtsi

【在linux内核中，平台代码是相当繁琐，不同平台支持的硬件规格一样，导致需要注册的设备也不相同，产生许多重复的代码。平台硬件规格不一样，这完全可以相当于一种配置，并不需要写成c语言代码，因此便诞生了dts设备树文件】

```
gpio_keys {
    pinctrl-0 = <&wps_pins>;
    pinctrl-names = "default";
    compatible = "gpio-keys";

    button@1 {
        label = "wps";
        linux,code = <KEY_WPS_BUTTON>;
        gpios = <&tlmm 63 GPIO_ACTIVE_LOW>;
        linux,input-type = <1>;
    };
};

leds {
    compatible = "gpio-leds";

    led@0 {
        label = "led_green";
        gpios = <&tlmm 0 GPIO_ACTIVE_HIGH>;
        default-state = "off";
    };

    led@3 {
        label = "led_red";
        gpios = <&tlmm 3 GPIO_ACTIVE_HIGH>;
        default-state = "off";
    };

    led@58 {
        label = "led_blue";
        gpios = <&tlmm 58 GPIO_ACTIVE_HIGH>;
        default-state = "off";
    };
};
```

发现：
./drivers/leds/leds-gpio.c
【一旦进入gpio_led_probe函数，在platform代码中已经解析过dts设备树，通过

compatible = "gpio-leds"找匹配节点。dev_get_platdata从dts中解析出leds数据结构gpio_led_platform_data】

```
static const struct of_device_id of_gpio_leds_match[] = {
    { .compatible = "gpio-leds", },
    {},
};

static int gpio_led_probe(struct platform_device *pdev)
{
    struct gpio_led_platform_data *pdata = dev_get_platdata(&pdev->dev);
    struct gpio_leds_priv *priv;
    int i, ret = 0;


    if (pdata && pdata->num_leds) {
        priv = devm_kzalloc(&pdev->dev,
                sizeof_gpio_leds_priv(pdata->num_leds),
                    GFP_KERNEL);
        if (!priv)
            return -ENOMEM;

        priv->num_leds = pdata->num_leds;
        for (i = 0; i < priv->num_leds; i++) {
            ret = create_gpio_led(&pdata->leds[i],
                         &priv->leds[i],
                         &pdev->dev, pdata->gpio_blink_set);
            if (ret < 0) {
                /* On failure: unwind the led creations */
                for (i = i - 1; i >= 0; i--)
                    delete_gpio_led(&priv->leds[i]);
                return ret;
            }
        }
    } else {
        priv = gpio_leds_create_of(pdev);
        if (IS_ERR(priv))
            return PTR_ERR(priv);
    }

    platform_set_drvdata(pdev, priv);

    return 0;
}
```

./gpio-button-hotplug.c

【

1、驱动gpio-button-hotplug通过.compatible = "gpio-keys"来匹配dts节点，找到

节点后进入gpio_keys_probe函数，这里可看出使用的不是poll方法驱动，而是gpio-keys驱动。

2、gpio_keys_get_devtree_pdata通过of系统函数获取定义的设备数据。

3、devm_request_irq注册gpio中断，中断处理函数为button_handle_irq。

】

```
static struct of_device_id gpio_keys_of_match[] = {
    { .compatible = "gpio-keys", },
    { },
};

static struct of_device_id gpio_keys_polled_of_match[] = {
    { .compatible = "gpio-keys-polled", },
    { },
};

static int gpio_keys_probe(struct platform_device *pdev)
{
    struct gpio_keys_platform_data *pdata;
    struct gpio_keys_button_dev *bdev;
    int ret, i;


    ret = gpio_keys_button_probe(pdev, &bdev, 0);

    if (ret)
        return ret;

    pdata = pdev->dev.platform_data;
    for (i = 0; i < pdata->nbuttons; i++) {
        struct gpio_keys_button *button = &pdata->buttons[i];
        struct gpio_keys_button_data *bdata = &bdev->data[i];

        if (bdata->can_sleep) {
            dev_err(&pdev->dev, "skipping gpio:%d, it can sleep\n",
            button->gpio);
            continue;
        }
        if (!button->irq)
            button->irq = gpio_to_irq(button->gpio);
        if (button->irq < 0) {
            dev_err(&pdev->dev, "failed to get irq for gpio:%d\n",
            button->gpio);
            continue;
        }
        ret = devm_request_irq(&pdev->dev, button->irq, button_handle_irq,
                    IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING,
                    dev_name(&pdev->dev), bdata);
        if (ret)
            dev_err(&pdev->dev, "failed to request irq:%d for gpio:%d\n",
            button->irq, button->gpio);
        else
            dev_dbg(&pdev->dev, "gpio:%d has irq:%d\n", button->gpio,
            button->irq);

        if (bdata->b->type == EV_SW)
            button_hotplug_event(bdata, EV_SW,
            gpio_button_get_value(bdata));
    }

    return 0;
}
```

工作：
./drivers/leds/leds-gpio.c
【gpio led的处理方式与之前W2C的方式一致（可参见《【代码解析】W2C关于GPIO灯与按钮》），用户均是通过/sys/class/leds下的属性进行操作】

./gpio-button-hotplug.c
【采用中断机制处理gpio key，在gpio_keys_button_probe函数中定义一个定时器bdata->timer和执行程序bdata->work_irq】

```
INIT_WORK(&bdata->work_irq, gpio_keys_gpio_work_func);
setup_timer(&bdata->timer, gpio_keys_gpio_timer,
            (unsigned long)bdata);
```

【中断处理函数设置定时器延迟触发，是为了减少中断函数占用时间，异步执行处理函数】

```
static irqreturn_t button_handle_irq(int irq, void *_bdata)
{
    struct gpio_keys_button_data *bdata = (struct gpio_keys_button_data *)
    _bdata;

    mod_timer(&bdata->timer,
            jiffies + msecs_to_jiffies(bdata->b->debounce_interval));

    return IRQ_HANDLED;
}
```

【定时器被触发，调用bdata->work_irq进行处理gpio key】

```
static void gpio_keys_gpio_timer(unsigned long _data)
{
    struct gpio_keys_button_data *bdata = (struct gpio_keys_button_data
    *)_data;

    schedule_work(&bdata->work_irq);
}
```

【gpio_keys_gpio_work_func通过button_hotplug_event发送uevent事件。bdata->last_state表示是按下还是弹出。priv->seen表示持续了多长时间，一般来说，按下

到弹出的持续时间对我们才有意义，弹出到按下的持续时间无意义，但代码为了流程一致，还是会进行统计。】

```
static void gpio_keys_gpio_work_func(struct work_struct *work_irq)
{
    struct gpio_keys_button_data *bdata =
        container_of(work_irq, struct gpio_keys_button_data, work_irq);
    struct bh_priv *priv = &bdata->bh;

    if (bdata->last_state == -1) {
        priv->seen = jiffies;
        BH_DBG("first key event detected - seen '%lu'\n", priv->seen);
    }

    bdata->last_state = gpio_button_get_value(bdata);
    button_hotplug_event(bdata, bdata->b->type ?: EV_KEY, bdata->
    last_state);
}
```

发送uevent：

【button_hotplug_fill_event填充事件字符串】

```
static int button_hotplug_fill_event(struct bh_event *event)
{
    int ret;

    ret = bh_event_add_var(event, 0, "HOME=%s", "/");
    if (ret)
        return ret;

    ret = bh_event_add_var(event, 0, "PATH=%s",
                    "/sbin:/bin:/usr/sbin:/usr/bin");
    if (ret)
        return ret;

    ret = bh_event_add_var(event, 0, "SUBSYSTEM=%s", "button");
    if (ret)
        return ret;

    ret = bh_event_add_var(event, 0, "ACTION=%s", event->action);
    if (ret)
        return ret;

    ret = bh_event_add_var(event, 0, "BUTTON=%s", event->name);
    if (ret)
        return ret;

    if (event->type == EV_SW) {
        ret = bh_event_add_var(event, 0, "TYPE=%s", "switch");
        if (ret)
            return ret;
    }

    ret = bh_event_add_var(event, 0, "SEEN=%ld", event->seen);
    if (ret)
        return ret;

    ret = bh_event_add_var(event, 0, "SEQNUM=%llu", uevent_next_seqnum());

    return ret;
}
```

【button_hotplug_work通过broadcast_uevent广播事件】

```
static void button_hotplug_work(struct work_struct *work)
{
    struct bh_event *event = container_of(work, struct bh_event, work);
    int ret = 0;

    event->skb = alloc_skb(BH_SKB_SIZE, GFP_KERNEL);
    if (!event->skb)
        goto out_free_event;

    ret = bh_event_add_var(event, 0, "%s@", event->action);
    if (ret)
        goto out_free_skb;

    ret = button_hotplug_fill_event(event);
    if (ret)
        goto out_free_skb;

    NETLINK_CB(event->skb).dst_group = 1;
    broadcast_uevent(event->skb, 0, 1, GFP_KERNEL);

 out_free_skb:
    if (ret) {
        BH_ERR("work error %d\n", ret);
        kfree_skb(event->skb);
    }
 out_free_event:
    kfree(event);
}
```