【代码解析】W2C关于GPIO灯与按钮

2017年6月1日
17:16

GPIO

W2C
定义：
./arch/mips/ath79/mach-ap147.c

```
#define AP147_GPIO_LED_WPS              13
#define AP147_GPIO_LED_WLAN             12
#define AP147_GPIO_LED_LAN4             11
#define AP147_GPIO_BTN_WPS              17
static struct gpio_led ap147_leds_gpio[] __initdata = {
        {
                .name           = "ap147:red",
                .gpio           = AP147_GPIO_LED_WPS,
                .active_low     = 1,
        },
        {
                .name           = "ap147:blue",
                .gpio           = AP147_GPIO_LED_WLAN,
                .active_low     = 1,
        },
        {
                .name           = "ap147:green",
                .gpio           = AP147_GPIO_LED_LAN4,
                .active_low     = 1,
        }
};

static struct gpio_keys_button ap147_gpio_keys[] __initdata = {
        {
                .desc           = "WPS button",
                .type           = EV_KEY,
                .code           = KEY_WPS_BUTTON,
                .debounce_interval = AP147_KEYS_DEBOUNCE_INTERVAL,
                .gpio           = AP147_GPIO_BTN_WPS,
                .active_low     = 1,
        },
};
```

注册：
./arch/mips/ath79/mach-ap147.c

```
ath79_register_leds_gpio(-1, ARRAY_SIZE(ap147_leds_gpio),
                  ap147_leds_gpio);
```

```
ath79_register_gpio_keys_polled(-1, AP147_KEYS_POLL_INTERVAL,
                  ARRAY_SIZE(ap147_gpio_keys),
                  ap147_gpio_keys)
```

./arch/mips/ath79/dev-leds-gpio.c

【gpio灯需要挂在总线上，首先注册一个总线设备leds-gpio（该设备统管所有gpio灯），gpio灯放入leds-gpio】

```
void __init ath79_register_leds_gpio(int id,
                                      unsigned num_leds,
                                      struct gpio_led *leds)
{
        struct platform_device *pdev;
        struct gpio_led_platform_data pdata;
        struct gpio_led *p;
        int err;

        p = kmalloc(num_leds * sizeof(*p), GFP_KERNEL);
        if (!p)
                return;

        memcpy(p, leds, num_leds * sizeof(*p));

        pdev = platform_device_alloc("leds-gpio", id);
        if (!pdev)
                goto err_free_leds;

        memset(&pdata, 0, sizeof(pdata));
        pdata.num_leds = num_leds;
        pdata.leds = p;

        err = platform_device_add_data(pdev, &pdata, sizeof(pdata));
        if (err)
                goto err_put_pdev;

        err = platform_device_add(pdev);
        if (err)
                goto err_put_pdev;

        return;

err_put_pdev:
        platform_device_put(pdev);

err_free_leds:
        kfree(p);
}
```

./arch/mips/ath79/dev-gpio-buttons.c

【gpio按钮需要挂在总线上，首先注册一个总线设备gpio-keys-polled（该设备统管所有gpio按钮），gpio按钮放入gpio-keys-polled】

```
void __init ath79_register_gpio_keys_polled(int id,
                                            unsigned poll_interval,
                                            unsigned nbuttons,
                                            struct gpio_keys_button
*buttons)
{
        struct platform_device *pdev;
        struct gpio_keys_platform_data pdata;
        struct gpio_keys_button *p;
        int err;

        p = kmalloc(nbuttons * sizeof(*p), GFP_KERNEL);
        if (!p)
                return;

        memcpy(p, buttons, nbuttons * sizeof(*p));

        pdev = platform_device_alloc("gpio-keys-polled", id);
        if (!pdev)
                goto err_free_buttons;

        memset(&pdata, 0, sizeof(pdata));
        pdata.poll_interval = poll_interval;
        pdata.nbuttons = nbuttons;
        pdata.buttons = p;

        err = platform_device_add_data(pdev, &pdata, sizeof(pdata));
        if (err)
                goto err_put_pdev;

        err = platform_device_add(pdev);
        if (err)
                goto err_put_pdev;

        return;

err_put_pdev:
        platform_device_put(pdev);

err_free_buttons:
        kfree(p);
}
```

发现：

./drivers/leds/leds-gpio.c

【加载leds-gpio驱动，gpio_led_probe通过pdata->num_leds不等于0来发现gpio-leds设备】

```
static int __devinit gpio_led_probe(struct platform_device *pdev)
{
        struct gpio_led_platform_data *pdata = pdev->dev.platform_data;
        struct gpio_leds_priv *priv;
        int i, ret = 0;

        if (pdata && pdata->num_leds) {
                priv = kzalloc(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]);
                                kfree(priv);
                                return ret;
                        }
                }
        } else {
                priv = gpio_leds_create_of(pdev);
                if (!priv)
                        return -ENODEV;
        }

        platform_set_drvdata(pdev, priv);

        return 0;
}
```

【发现到一个设备后，create_gpio_led配置gpio灯的属性，通过led_classdev_register向/sys/class/leds进行注册，用户通过/sys/class/leds属性进行操作】

```
static int __devinit create_gpio_led(const struct gpio_led *template,
        struct gpio_led_data *led_dat, struct device *parent,
        int (*blink_set)(unsigned, int, unsigned long *, unsigned long *))
{
        int ret, state;
```

```
        led_dat->gpio = -1;

        /* skip leds that aren't available */
        if (!gpio_is_valid(template->gpio)) {
                printk(KERN_INFO "Skipping unavailable LED gpio %d (%s)\n",
                                template->gpio, template->name);
                return 0;
        }

        ret = gpio_request(template->gpio, template->name);
        if (ret < 0)
                return ret;

        led_dat->cdev.name = template->name;
        led_dat->cdev.default_trigger = template->default_trigger;
        led_dat->gpio = template->gpio;
        led_dat->can_sleep = gpio_cansleep(template->gpio);
        led_dat->active_low = template->active_low;
        led_dat->blinking = 0;
        if (blink_set) {
                led_dat->platform_gpio_blink_set = blink_set;
                led_dat->cdev.blink_set = gpio_blink_set;
        }
        led_dat->cdev.brightness_set = gpio_led_set;
        if (template->default_state == LEDS_GPIO_DEFSTATE_KEEP)
                state = !!gpio_get_value_cansleep(led_dat->gpio) ^
led_dat->active_low;
        else
                state = (template->default_state == LEDS_GPIO_DEFSTATE_ON);
        led_dat->cdev.brightness = state ? LED_FULL : LED_OFF;
        if (!template->retain_state_suspended)
                led_dat->cdev.flags |= LED_CORE_SUSPENDRESUME;

        ret = gpio_direction_output(led_dat->gpio, led_dat->active_low ^
state);
        if (ret < 0)
                goto err;

        INIT_WORK(&led_dat->work, gpio_led_work);

        ret = led_classdev_register(parent, &led_dat->cdev);
        if (ret < 0)
                goto err;

        return 0;
err:
        gpio_free(led_dat->gpio);
        return ret;
}
```

../gpio-button-hotplug/gpio-button-hotplug.c

【加载gpio-keys-polled驱动，gpio_keys_polled_probe通过pdata->nbuttons不等于0来发现gpio-keys设备，并且启动gpio_keys_polled_poll用来轮询gpio按钮状态】

```
static int __devinit gpio_keys_polled_probe(struct platform_device *pdev)
{
	struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
	struct device *dev = &pdev->dev;
	struct gpio_keys_polled_dev *bdev;
	int error;
	int i;

	if (!pdata || !pdata->poll_interval)
		return -EINVAL;

	bdev = kzalloc(sizeof(struct gpio_keys_polled_dev) +
			pdata->nbuttons * sizeof(struct gpio_keys_button_data),
			GFP_KERNEL);
	if (!bdev) {
		dev_err(dev, "no memory for private data\n");
		return -ENOMEM;
	}

	for (i = 0; i < pdata->nbuttons; i++) {
		struct gpio_keys_button *button = &pdata->buttons[i];
		struct gpio_keys_button_data *bdata = &bdev->data[i];
		unsigned int gpio = button->gpio;

		if (button->wakeup) {
			dev_err(dev, DRV_NAME " does not support wakeup\n");
			error = -EINVAL;
			goto err_free_gpio;
		}

		error = gpio_request(gpio,
				     button->desc ? button->desc : DRV_NAME);
		if (error) {
			dev_err(dev, "unable to claim gpio %u, err=%d\n",
				gpio, error);
			goto err_free_gpio;
		}

		error = gpio_direction_input(gpio);
		if (error) {
			dev_err(dev,
				"unable to set direction on gpio %u, err=%d\n",
				gpio, error);
			goto err_free_gpio;
```

```
        }

        bdata->can_sleep = gpio_cansleep(gpio);
        bdata->last_state = 0;
        bdata->threshold = DIV_ROUND_UP(button->debounce_interval,
                                        pdata->poll_interval);
    }

    bdev->dev = &pdev->dev;
    bdev->pdata = pdata;
    platform_set_drvdata(pdev, bdev);

    INIT_DELAYED_WORK(&bdev->work, gpio_keys_polled_poll);

    gpio_keys_polled_open(bdev);

    return 0;

err_free_gpio:
    while (--i >= 0)
        gpio_free(pdata->buttons[i].gpio);

    kfree(bdev);
    platform_set_drvdata(pdev, NULL);

    return error;
}
```

工作：

【led-gpio：用户进程通过操作/sys/class/leds下的属性控制灯光的状态】

```
#define GREEN_LED_BRIGHTNESS "/sys/class/leds/ap147:green/brightness"
#define GREEN_LED_TRIGGER "/sys/class/leds/ap147:green/trigger"
#define GREEN_LED_DELAY_ON "/sys/devices/platform/leds-
gpio/leds/ap147:green/delay_on"
#define GREEN_LED_DELAY_OFF "/sys/devices/platform/leds-
gpio/leds/ap147:green/delay_off"

#define BLUE_LED_BRIGHTNESS "/sys/class/leds/ap147:blue/brightness"
#define BLUE_LED_TRIGGER "/sys/class/leds/ap147:blue/trigger"
#define BLUE_LED_DELAY_ON "/sys/class/leds/ap147:blue/delay_on"
#define BLUE_LED_DELAY_OFF "/sys/class/leds/ap147:blue/delay_off"

#define RED_LED_BRIGHTNESS "/sys/class/leds/ap147:red/brightness"
#define RED_LED_TRIGGER "/sys/class/leds/ap147:red/trigger"
#define RED_LED_DELAY_ON "/sys/class/leds/ap147:red/delay_on"
#define RED_LED_DELAY_OFF "/sys/class/leds/ap147:red/delay_off"

//关闭LED_GP正常流程,快闪两秒后,绿色常亮,等待重启
g_led_reset_flag = 1;
```

```
sprintf(cmd, "echo 0 > %s; echo 1 > %s; echo timer > %s; echo 100 > %s;
echo 100 > %s",
    RED_LED_BRIGHTNESS, GREEN_LED_BRIGHTNESS, GREEN_LED_TRIGGER,
    GREEN_LED_DELAY_ON, GREEN_LED_DELAY_OFF);
system(cmd);
sleep(2);
sprintf(cmd, "echo 0 > %s; echo 0 > %s", RED_LED_BRIGHTNESS,
GREEN_LED_BRIGHTNESS);
system(cmd);
```

【gpio-keys：gpio_keys_polled_poll通过pdata->poll_interval指定的间隔来轮询gpio输入状态】

```
static void gpio_keys_polled_poll(struct work_struct *work)
{
    struct gpio_keys_polled_dev *bdev =
        container_of(work, struct gpio_keys_polled_dev, work.work);
    struct gpio_keys_platform_data *pdata = bdev->pdata;
    int i;

    for (i = 0; i < bdev->pdata->nbuttons; i++) {
        struct gpio_keys_button_data *bdata = &bdev->data[i];

        if (bdata->count < bdata->threshold)
            bdata->count++;
        else
            gpio_keys_polled_check_state(&pdata->buttons[i], bdata);
    }
    gpio_keys_polled_queue_work(bdev);
}

static void gpio_keys_polled_queue_work(struct gpio_keys_polled_dev *bdev)
{
    struct gpio_keys_platform_data *pdata = bdev->pdata;
    unsigned long delay = msecs_to_jiffies(pdata->poll_interval);

    if (delay >= HZ)
        delay = round_jiffies_relative(delay);
    schedule_delayed_work(&bdev->work, delay);
}
```

【gpio_keys_polled_check_state检测是否达到预定的触发条件，到达之后通过button_hotplug_event广播netlink事件】

```
static void gpio_keys_polled_check_state(struct gpio_keys_button *button,
                        struct gpio_keys_button_data *bdata)
{
    int state;

    if (bdata->can_sleep)
        state = !!gpio_get_value_cansleep(button->gpio);
```

```
        else
            state = !!gpio_get_value(button->gpio);

        state = !!(state ^ button->active_low);
        if (state != bdata->last_state) {
            unsigned int type = button->type ?: EV_KEY;

            button_hotplug_event(bdata, type, button->code, state);
            bdata->count = 0;
            bdata->last_state = state;
        }
    }
```