天嵌科技

广州天嵌计算机科技有限公司

地址：广东省广州市天河区天河北路 908 号高科大厦 B 座 908　邮编:510000
电话：020-38373101-805 804（销售）　020-38373101-807（技术支持）　传真：020-38373101-816
E-mail: support@embedsky.net 官方网站：http://www.embedsky.net 或 http://bbs.embedsky.net

TQ6410 PDA 开发板 Linux 驱动开发教程

V1.0.1（20110831）

广州天嵌计算机科技有限公司荣誉出品

首发网站：www.embedsky.net

版权声明

前言

这份教程是天嵌科技提供的针对 TQ6410 的 Linux 的系列教程的第一册，以后推出的第二册第三册均在本次的版本号上进行增加。

本教程将针对 TQ6410 开发的 Linux。简要的说明内核源码的获取以及配置编译过程。侧重于讲解驱动的开发过程，如果要详细了解内核移植的步骤、过程，可以阅读 TQ2440 中的《Linux 移植之 Step By Step》。关于 TQ2440 的 Linux 移植教程可以从 BBS 获得，BBS 地址：bbs.embedsky.net。其教程下载地址：http://soft.embedsky.net/files/cd_iso/TQ2440%E5%BC%80%E5%8F%91%E6%9D%BF%E9%85%8D%E5%A5%97%E6%95%99%E6%9D%90%E9%9B%86_201006.rar

在本教程的第一册中，主要讲解 Linux 内核中的输入子系统，同时也配合 HS0038 红外接收头驱动，按键驱动来进一步说明一下 Linux 子系统。另外增加解析 ADC 驱动，蜂鸣器驱动。

本教程的初衷是和读者们分享以下知识点：

a) 解析 Linux 内核中的输入子系统，包括其处理输入设备注册的细节以及处理输入设备事件的过程
b) 通过按键驱动和 HS0038 红外接收头驱动来进一步了解输入设备驱动的注册的过程
c) 举例说明 platform driver 和 platform device 在平台设备驱动注册过程中的关系
d) 阐述 Linux 内核中字符驱动注册的流程
e) 通过 ADC 字符驱动，实践字符驱动编写过程，了解内核中 ADC 驱动的一些接口的利用以及了解 S3C6410 中的 ADC 相关寄存器的设置。
f) 通过蜂鸣器驱动，进一步熟悉字符驱动的编写流程，了解定时器的 PWM 功能设置和利用

天嵌科技——研发部
2011 年 08 月 31 日（V1.0.1 版本）

更新说明

V1.0.1 暂无更新。

目录

第一章 Linux 输入子系统

1.1 input 子系统简介

Linux 内核的 Input 子系统主要有三部组合而成，他们是 Input driver，Input core 以及 Input handler。当事件触发时，也就是有输入设备输入（按键，触摸屏，鼠标等），整个子系统的大概处理过程是：

Input driver→ Input core→Input handler→用户空间。如下图所示：

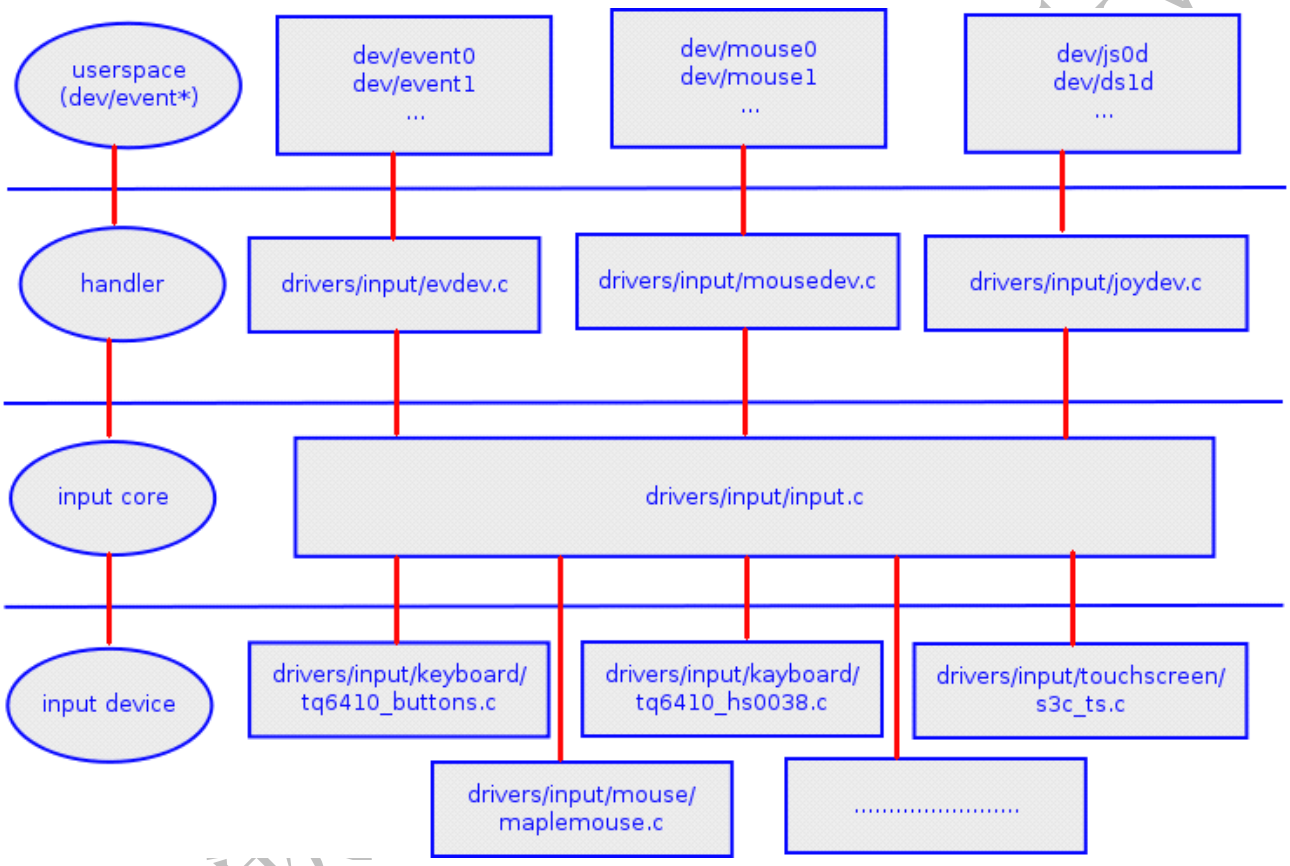


图 1-1

1.2 Input 子系统中一些重要的接口介绍

a） struct input_dev *input_allocate_device(void)

定义于 drivers/input/input.c 中 input_allocate_device 函数为 input_dev（在 include/linux/input.h 中定义）分配内存空间，初始化其内嵌的 device，这里还初始化了 input_dev 结构中的 h_list 和 node 链表头，node 是用于将该 input_dev 挂到 input_dev_list 上，而 h_list 指向的量表用于存放 input_handle(include/linux/input.h 中定义)。

b） void input_set_capability(struct input_dev *dev, unsigned int type, unsigned int code)

该函数用于告知 input 子系统它可以报告的事件，它定义于 drivers/input/input.c 中。这个函数中根据不同的事件响应类型，设置对应的事件码 code.其实它最终也是调用：

static inline void __set_bit(int nr, volatile unsigned long *addr)

该函数位于 include/asm-generic/bitops/non-atomic.h

c） **int input_register_device(struct input_dev *dev)**

该函数用于注册输入子系统的设备。定义于 drivers/input/input.c 中。

d） **void input_unregister_device(struct input_dev *dev)**

该函数用于注销输入子系统中的输入设备。定义于 drivers/input/input.c 中。

e） **void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)**

该函数是在发生输入事件时，向子系统报告事件的。定义于 drivers/input/input.c 中。这个函数都分别被一下的函数调用。

static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)

static inline void input_report_rel(struct input_dev *dev, unsigned int code, int value)

static inline void input_report_abs(struct input_dev *dev, unsigned int code, int value)

它们都是用于报告 EV_KEY、EV_REL、EV_ABS 等事件的函数。

f） **static inline void input_sync(struct input_dev *dev)**

用来告诉上层，本次的事件已经完成了.

1.3 事件响应概念

Linux 内核中，定义了几种不同的事件响应类型，它们定义于 include/linux/input.h 中，如下截图：

```
/*
 * Event types
 */

#define EV_SYN                  0x00
#define EV_KEY                  0x01
#define EV_REL                  0x02
#define EV_ABS                  0x03
#define EV_MSC                  0x04
#define EV_SW                   0x05
#define EV_LED                  0x11
#define EV_SND                  0x12
#define EV_REP                  0x14
#define EV_FF                   0x15
#define EV_PWR                  0x16
#define EV_FF_STATUS            0x17
#define EV_MAX                  0x1f
#define EV_CNT                  (EV_MAX+1)
```

另外在文件 include/linux/input.h 中也定义了许多响应码（事件码）。部分截图如下：

```
#define KEY_RESERVED            0
#define KEY_ESC                 1
#define KEY_1                   2
#define KEY_2                   3
#define KEY_3                   4
#define KEY_4                   5
#define KEY_5                   6
#define KEY_6                   7
#define KEY_7                   8
#define KEY_8                   9
#define KEY_9                   10
#define KEY_0                   11
#define KEY_MINUS               12
#define KEY_EQUAL               13
#define KEY_BACKSPACE           14
#define KEY_TAB                 15
#define KEY_Q                   16
#define KEY_W                   17
#define KEY_E                   18
#define KEY_R                   19
#define KEY_T                   20
#define KEY_Y                   21
#define KEY_U                   22
#define KEY_I                   23
#define KEY_O                   24
#define KEY_P                   25
#define KEY_LEFTBRACE           26
#define KEY_RIGHTBRACE          27
#define KEY_ENTER               28
```

通过前面的重要函数介绍，得知驱动层是靠 **input_event** 函数来向输入子系统传递事件的信息的。在这个函数，它会针对传递的输入设备，判断其可用的事件类型，以及有效的事件码，做出相应的处理。所以要注册输入设备，也必须要有事件类型设置，和事件码的概念。重要的说就是一个设备可以支持一个或多个事件类型；每个事件类型下面还需要设置具体的触发事件码。

1.4 注册一个输入设备的基本要求

在内核源码中的说明文档中 Documentation/input/input-programming.txt，给出了注册一个输入设备的最基本的例子。一下是文档中的例子源码：

```
#include <linux/input.h>
#include <linux/module.h>
#include <linux/init.h>

#include <asm/irq.h>
#include <asm/io.h>

static struct input_dev *button_dev;
```

```
static irqreturn_t button_interrupt(int irq, void *dummy)
{
	input_report_key(button_dev, BTN_0, inb(BUTTON_PORT) & 1);
	input_sync(button_dev);
	return IRQ_HANDLED;
}

static int __init button_init(void)
{
	int error;

	if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
			printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
			return -EBUSY;
		}

	button_dev = input_allocate_device();
	if (!button_dev) {
		printk(KERN_ERR "button.c: Not enough memory\n");
		error = -ENOMEM;
		goto err_free_irq;
	}

	button_dev->evbit[0] = BIT_MASK(EV_KEY);
	button_dev->keybit[BIT_WORD(BTN_0)] = BIT_MASK(BTN_0);

	error = input_register_device(button_dev);
	if (error) {
		printk(KERN_ERR "button.c: Failed to register device\n");
		goto err_free_dev;
	}

	return 0;

 err_free_dev:
	input_free_device(button_dev);
 err_free_irq:
	free_irq(BUTTON_IRQ, button_interrupt);
	return error;
}

static void __exit button_exit(void)
{
		input_unregister_device(button_dev);
```

```
    free_irq(BUTTON_IRQ, button_interrupt);
}

module_init(button_init);
module_exit(button_exit);
```

概括性的看这个例子，里面仅仅提供了一个设备注册的处理函数 button_init，设备注销函数 button_exit 以及输入设备触发响应的中断处理函数 button_interrupt。

先概要的理解 button_init 函数：

```
static int __init button_init(void)
{
    int error;
    //这里将输入的设备相应处理以中断方式来处理，这里先注册中断，这就是 button_interrupt
    //存在的理由
    if (request_irq(BUTTON_IRQ, button_interrupt, 0, "button", NULL)) {
                printk(KERN_ERR "button.c: Can't allocate irq %d\n", button_irq);
                return -EBUSY;
        }
    //这个之前提到过，是为新的输入设备分配空间的，这是输入设备注册开始时必须
    //要完成的任务之一
    button_dev = input_allocate_device();
    if (!button_dev) {
        printk(KERN_ERR "button.c: Not enough memory\n");
        error = -ENOMEM;
        goto err_free_irq;
    }

    //设置输入设备接受的事件类型
    button_dev->evbit[0] = BIT_MASK(EV_KEY);
    //设置有效的事件码
    button_dev->keybit[BIT_WORD(BTN_0)] = BIT_MASK(BTN_0);
    //注册输入设备
    error = input_register_device(button_dev);
    if (error) {
        printk(KERN_ERR "button.c: Failed to register device\n");
        goto err_free_dev;
    }

    return 0;

 err_free_dev:
    input_free_device(button_dev);
 err_free_irq:
    free_irq(BUTTON_IRQ, button_interrupt);
    return error;
```

```
}
```

对于 button_exit(void)函数，就是处理卸载输入设备以及释放中断。下面看看 button_interrupt 处理函数

```
static irqreturn_t    button_interrupt(int irq, void *dummy)
{
    input_report_key(button_dev, BTN_0, inb(BUTTON_PORT) & 1);
    input_sync(button_dev);
    return IRQ_HANDLED;
}
```

当中断发生，也就是输入设备触发了中断时，中断中处理的是通过 input_report_key 向子系统报告事件，input_report_key 是针对事件类型为 EV_KEY 的响应，向子系统报告的接口。它最终也是调用了 input_event 来处理。这个在 1.2 章节中的 e） 项有说明了。

进入子系统处理后，再利用 input_sync 函数来告诉告诉上层，子系统处理事件是否完毕。

通过这个简单的例子，可以概括的说明注册一个输入设备驱动的步骤如下：

1、驱动中要有一个由输入设备触发的处理函数

这个处理过程主要是向子系统汇报事件，向上层交代处理结果。

2、一个初始化函数

初始化包括注册触发时的处理接口，创建新设备，设置事件类型，设置事件码。

3、一个注销函数

注销输入设备，如果有中断注册，那也需要注销中断。

1.5 输入设备注册过程简析

输入设备的种类很多，但他们在基于内核子系统的注册过程，步骤差不多。内核中针对不同类型的驱动注册也多，这里针对于 evdev 的注册过程进行简析。

1.5.1 重要结构体的关系

在输入设备注册过程，可以切分为 input device，handler，handle 这三个结构的处理，其实主要目的是了解这三者之间是如何联系在一起，那就了解输入设备在 Linux 子系统中的注册过程。

a) input device 功能说明

input device 的功能，之前的 1.4 章节有介绍。它就是针对实际设备端口的初始化，响应接口的设置，响应类型的设置等，也就是它生成了新的输入设备。

专门用来注册它的函数是 int input_register_device(struct input_dev *dev)，系统中所有的输入设备在注册后都会被增加进入 input_dev_list 链表中去，在 int input_register_device(struct input_dev *dev)中就有这样一句代码 list_add_tail(&dev->node, &input_dev_list);将其加入到 input_dev_list 链表中去

b) handler 的功能说明

handler 的功能，对于每一个新注册的 input device 都有一个和它对应的 handler。handler 的主要功能是提供了输入设备对应的一些操作接口，例如经常看到的 write， read， ioctl 等。它是属于 struct input_handler 结构体。该结构体定义于 include/linux/input.h 中。里面的主要成员有 fops ，专门用来设定对应输入设备的读，写等控制，另外一个成员是 connect， 它就是用来指定将 handler 和 input device 关联的处理接口。专门用于注册 handler 的函数是 int input_register_handler(struct input_handler *handler)，同样的所有注册的 handler 都被加入到 input_handler_list 链表中去，注册函数中利用 list_add_tail(&handler->node,

&input_handler_list);来完成的。

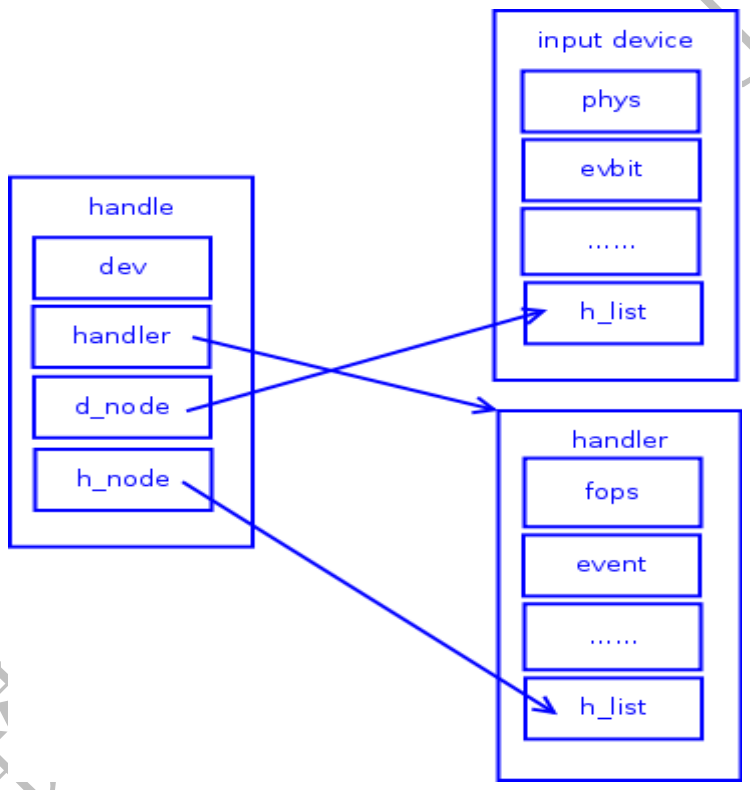
c) handle 的功能说明

handle 的功能，handle 的功能就是将 input device 和其对应的 handler 对应起来，它就像一座桥，将两者准确的连接在一起，这样在事件被处理的时候，调用对应的 handler 中的处理接口接容易了。专门用于注册 handle 的函数是 int input_register_handle(struct input_handle *handle)。在该函数中利用以下代码将其直接和 input device 关联起来

```
if (handler->filter)
    list_add_rcu(&handle->d_node, &dev->h_list); //将 handle 加入 input device 的 h_list 链表，从表头加入
else
    list_add_tail_rcu(&handle->d_node, &dev->h_list); //将 handle 加入 input device 的 h_list 链表，从表尾加入
```

另外也利用 list_add_tail_rcu(&handle->h_node, &handler->h_list); 语句将其加入到对应的 handler 的 h_list 列表中

这三者的关系可以用以下截图表示：



1.5.2 重要注册函数的要点解析

简述完输入设备在子系统中的 input device ， handler， handle 三者关系之后，下面主要看实际的注册函数进行逐步的解析一下，这样可以进一步看清楚其注册的流程。

a）handler 的注册过程

```
int input_register_handler(struct input_handler *handler)
{
    struct input_dev *dev;
    int retval;
```

```
    retval = mutex_lock_interruptible(&input_mutex);
    if (retval)
        return retval;
    //初始化 handler 的 h_list 列表
    INIT_LIST_HEAD(&handler->h_list);

    if (handler->fops != NULL) {
//以 handler->minor 右移五位做为索引值插入到 input_table[ ]中
//input_table 将来可以利用它来检索 handler，这也是根据设备节点
//取得其次设备号之后来实现的。
        if (input_table[handler->minor >> 5]) {
            retval = -EBUSY;
            goto out;
        }
        input_table[handler->minor >> 5] = handler;
    }
    //将其加入到 input_handler_list 链表中去
    list_add_tail(&handler->node, &input_handler_list);
    //检索 handler，让其和在 input_dev_list 链表上的输入设备关联
    //其实这里就算没有设备可以关联，也没有关系，因为在输入设备
    //注册的时候，它会去处理，寻找和它关联的 handler.
    list_for_each_entry(dev, &input_dev_list, node)
        input_attach_handler(dev, handler);

    input_wakeup_procfs_readers();

 out:
    mutex_unlock(&input_mutex);
    return retval;
}
```

概括其注册流程如下：

初始化其 h_list 成员列表→ 将其加入 input_table 中→ 加入到 input_handler_list→ 检索与之关联的输入设备。

b） input device 的注册过程

```
int input_register_device(struct input_dev *dev)
{
    static atomic_t input_no = ATOMIC_INIT(0);
    struct input_handler *handler;
    const char *path;
    int error;

    /* Every input device generates EV_SYN/SYN_REPORT events. */
    __set_bit(EV_SYN, dev->evbit);//增加设置所有输入设备都具备的事件类型
```

```
    /* KEY_RESERVED is not supposed to be transmitted to userspace. */
    __clear_bit(KEY_RESERVED, dev->keybit);

    /* Make sure that bitmasks not mentioned in dev->evbit are clean. */
    input_cleanse_bitmasks(dev);

    if (!dev->hint_events_per_packet)
        dev->hint_events_per_packet =
                    input_estimate_events_per_packet(dev);

    /*
     * If delay and period are pre-set by the driver, then autorepeating
     * is handled by the driver itself and we don't do it in input.c.
     */
    init_timer(&dev->timer);
    if (!dev->rep[REP_DELAY] && !dev->rep[REP_PERIOD]) {
        dev->timer.data = (long) dev;
        dev->timer.function = input_repeat_key;
        dev->rep[REP_DELAY] = 250;
        dev->rep[REP_PERIOD] = 33;
    }

    if (!dev->getkeycode)//如果没有定义 getkeycode 接口，这用默认的
        dev->getkeycode = input_default_getkeycode;

    if (!dev->setkeycode) //如果没有定义 setkeycode 接口，这用默认的
        dev->setkeycode = input_default_setkeycode;

    dev_set_name(&dev->dev, "input%ld",
                 (unsigned long) atomic_inc_return(&input_no) - 1);

    error = device_add(&dev->dev); //将 input_dev 中封装的 device 注册到 sysfs
    if (error)
        return error;

    path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
    pr_info("%s as %s\n",
        dev->name ? dev->name : "Unspecified device",
        path ? path : "N/A");
    kfree(path);

    error = mutex_lock_interruptible(&input_mutex);
    if (error) {
        device_del(&dev->dev);
```

```
            return error;
        }
    //将新注册的 input device 加入到 input_dev_list 链表中
        list_add_tail(&dev->node, &input_dev_list);
    //下面就是将新注册的 input device 和 handler 关联的处理过程
    //以及注册对应的 handle 过程
        list_for_each_entry(handler, &input_handler_list, node)
            input_attach_handler(dev, handler);

        input_wakeup_procfs_readers();

        mutex_unlock(&input_mutex);

        return 0;
    }
```

Input device 的注册过程，最主要的环节就是于 handler 关联的处理过程，这个必须弄清楚。下面看看 **input_attach_handler(dev, handler)** 函数都做了什么事情。

```
static int input_attach_handler(struct input_dev *dev, struct input_handler *handler)
{
    const struct input_device_id *id;
    int error;
    id = input_match_device(handler, dev);//检索与 input device 对应的 handler

    if (!id)
        return -ENODEV;
    //在检索到了与 input device 对应的 handler 之后，
    //调用其对应的 connnect 接口来将 input device 和 handler 关联起来
    //在这一过程，也就注册了 handle 和生成相应的设备节点（/dev/event0 等这样的名称)
    error = handler->connect(handler, dev, id);
    if (error && error != -ENODEV)
        pr_err("failed to attach handler %s to device %s, error: %d\n",
                handler->name, kobject_name(&dev->dev.kobj), error);

    return error;
}
```

所以要了解 input device 如何与 handler 关联上，只能去了解 input_match_device(handler, dev)这个函数如何检索与 input device 匹配的 handler，以及 对应的 handler 提供的 connect 接口是如何关联他们的的。先看看 input_match_device(handler, dev) 函数是如何检索的

```
static const struct input_device_id *input_match_device(struct input_handler *handler,
                                    struct input_dev *dev)
{
    const struct input_device_id *id;
    int i;
    for (id = handler->id_table; id->flags || id->driver_info; id++) {
```

```
        //检索他们的总线类型是否相同，如果 flags 中有设置
        if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
            if (id->bustype != dev->id.bustype)
                continue;
        //检索他们的设备厂商是否相同，如果 flags 中有设置
        if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
            if (id->vendor != dev->id.vendor)
                continue;
        //检索它们的设备号是否相同，如果 flags 中有设置
        if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
            if (id->product != dev->id.product)
                continue;
        //检索它们的版本是否相同，如果 flags 中有设置
        if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
            if (id->version != dev->id.version)
                continue;

        MATCH_BIT(evbit,  EV_MAX);
        MATCH_BIT(keybit, KEY_MAX);
        MATCH_BIT(relbit, REL_MAX);
        MATCH_BIT(absbit, ABS_MAX);
        MATCH_BIT(mscbit, MSC_MAX);
        MATCH_BIT(ledbit, LED_MAX);
        MATCH_BIT(sndbit, SND_MAX);
        MATCH_BIT(ffbit,  FF_MAX);
        MATCH_BIT(swbit,  SW_MAX);

        if (!handler->match || handler->match(handler, dev)){
            return id;
        //从以上的检索过程，只有它们的各个检索项都匹配以后，才返回与之对应的 id
        }
    }
    return NULL;
}
```

在找到匹配的 handler 之后，那么就调用 handler 的 connnect 接口来将 input device 和它关联起来，这个关联的过程，也就涉及了 handle 的注册过程和设备节点的创建。下面以 evdev 设备和它的 handler 连接过程为例子，了解一下这个处理过程：

```
static int evdev_connect(struct input_handler *handler, struct input_dev *dev,
                 const struct input_device_id *id)
{
    struct evdev *evdev;
    int minor;
    int error;
```

```
    for (minor = 0; minor < EVDEV_MINORS; minor++)
        if (!evdev_table[minor]) //寻找 evdev_table 中的第一个空元素，用来存放这个新的 evdev
            break;

    if (minor == EVDEV_MINORS) {
        pr_err("no more free evdev devices\n");
        return -ENFILE;
    }

    evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL);
    if (!evdev)
        return -ENOMEM;

    INIT_LIST_HEAD(&evdev->client_list);
    spin_lock_init(&evdev->client_lock);
    mutex_init(&evdev->mutex);
    init_waitqueue_head(&evdev->wait);
    dev_set_name(&evdev->dev, "event%d", minor);
    evdev->exist = true;
    evdev->minor = minor;
    //初始化 evdev 中的 handle
    evdev->handle.dev = input_get_device(dev);
    evdev->handle.name = dev_name(&evdev->dev);
    evdev->handle.handler = handler;
    evdev->handle.private = evdev;
    //创建对应的设备节点，EVDEV_MINOR_BASE(64)+minor 的结果就是此设备号
    //而 INPUT_MAJOR(13)就是 evdev 设备的主设备号，如下图所示：
    evdev->dev.devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor);
```


```
    evdev->dev.class = &input_class;
    evdev->dev.parent = &dev->dev;
    evdev->dev.release = evdev_free;
```

```
    device_initialize(&evdev->dev);

    //注册 evdev 中的 handle，这个就是下面要了解的 handle 注册
  error = input_register_handle(&evdev->handle);
    if (error)
        goto err_free_evdev;

    error = evdev_install_chrdev(evdev);
    if (error)
        goto err_unregister_handle;

    error = device_add(&evdev->dev);
    if (error)
        goto err_cleanup_evdev;
    return 0;

 err_cleanup_evdev:
        evdev_cleanup(evdev);
 err_unregister_handle:
        input_unregister_handle(&evdev->handle);
 err_free_evdev:
        put_device(&evdev->dev);
    return error;
}
```

c）handle 的注册过程

其实它的注册函数也就是处理将 handle 加入到 input device 的 h_list 以及 handler 的 h_list 链表中去而已。

```
int input_register_handle(struct input_handle *handle)
{
        struct input_handler *handler = handle->handler;
        struct input_dev *dev = handle->dev;
        int error;

    /*
     * We take dev->mutex here to prevent race with
     * input_release_device().
     */
    error = mutex_lock_interruptible(&dev->mutex);
    if (error)
        return error;

    /*
     * Filters go to the head of the list, normal handlers
     * to the tail.
```

```
        */
        //handle 加入到 input device 的 h_list
        if (handler->filter)
                list_add_rcu(&handle->d_node, &dev->h_list);
        else
                list_add_tail_rcu(&handle->d_node, &dev->h_list);

        mutex_unlock(&dev->mutex);

        /*
         * Since we are supposed to be called from ->connect()
         * which is mutually exclusive with ->disconnect()
         * we can't be racing with input_unregister_handle()
         * and so separate lock is not needed here.
         */
        //将其加入到 handler 的 h_list 链表中
list_add_tail_rcu(&handle->h_node, &handler->h_list);

        if (handler->start)
                handler->start(handle);

        return 0;
}
```

简析了整个注册过程，下面以一张图片来说明一下整个注册的流程，以便直观的理解一些。

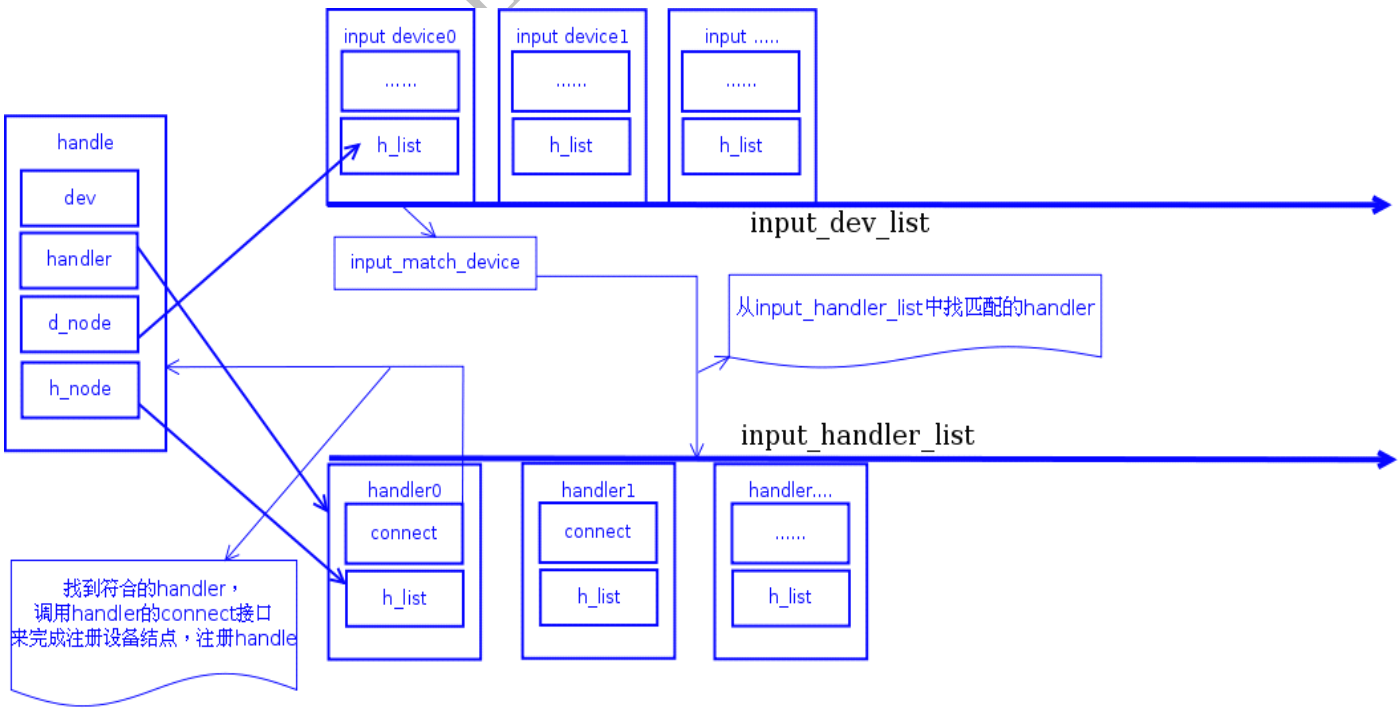


图 1-6

1.6 子系统的处理事件过程简析

操作输入设备的设备节点时，第一时间无非就是打开该设备。在 drivers/input/input.c 中就定义了设备节点的 open 函数入口

```
static int input_open_file(struct inode *inode, struct file *file)
{
	struct input_handler *handler;
	const struct file_operations *old_fops, *new_fops = NULL;
	int err;

	err = mutex_lock_interruptible(&input_mutex);
	if (err)
		return err;

	/* No load-on-demand here? */
//从全局数组 input_table 中检索 handler，
//这个在前面的 handler 注册过程中有将 handler 加入到 input_table 了。
	handler = input_table[iminor(inode) >> 5];
	if (handler)
		new_fops = fops_get(handler->fops);
//取出 handler 之后，将 handler 中提供的各种处理接口
//提取，用于更新当前的 fops 接口的，以便在对应的 input device 调用其对应的 open 等函数
	mutex_unlock(&input_mutex);

	/*
	 * That's _really_ odd. Usually NULL ->open means "nothing special",
	 * not "no device". Oh, well...
	 */
	if (!new_fops || !new_fops->open) {
		fops_put(new_fops);
		err = -ENODEV;
		goto out;
	}
 //如果没有找到对应的处理接口，则说明该设备不存在，返回失败。
	old_fops = file->f_op;
	file->f_op = new_fops;

	err = new_fops->open(inode, file);
	if (err) {
		fops_put(file->f_op);
		file->f_op = fops_get(old_fops);
	}
	fops_put(old_fops);
```

```
out:
    return err;
}
```

设备打开之后，各个接口都准备好了，那么在操作的时候，是靠 input_event 来和子系统报告事件的，这个在之前的简析中有提过。

```
void input_event(struct input_dev *dev,
            unsigned int type, unsigned int code, int value)
{
    unsigned long flags;
    //先检索事件被支持的类型是否在输入设备可以接受的，
    //设备注册的时候有指定设备所接受的事件类型的。
    if (is_event_supported(type, dev->evbit, EV_MAX)) {

        spin_lock_irqsave(&dev->event_lock, flags);
        add_input_randomness(type, code, value);
        //在事件类型合法后，就调用以下的函数来处理
        input_handle_event(dev, type, code, value);
        spin_unlock_irqrestore(&dev->event_lock, flags);
    }
}
```

在事件类检测通过以后，接着就调用 input_handle_event 这个处理继续事件

```
static void input_handle_event(struct input_dev *dev,
                        unsigned int type, unsigned int code, int value)
{
    int disposition = INPUT_IGNORE_EVENT;

    switch (type) {
    //针对不同事件类型的分类处理
    //而且每种类型都必须检索对应的事件码 code 是否被支持
    //只有被支持才会进一步处理，这个事件吗在设备注册的时候也是设置好了的
    case EV_SYN:
        switch (code) {
        case SYN_CONFIG:
            disposition = INPUT_PASS_TO_ALL;
            break;

        case SYN_REPORT:
            if (!dev->sync) {
                dev->sync = true;
                disposition = INPUT_PASS_TO_HANDLERS;
            }
            break;
        case SYN_MT_REPORT:
            dev->sync = false;
```

```
                disposition = INPUT_PASS_TO_HANDLERS;
                break;
        }
        break;

    case EV_KEY:
        if (is_event_supported(code, dev->keybit, KEY_MAX) &&
            !!test_bit(code, dev->key) != value) {

            if (value != 2) {
                __change_bit(code, dev->key);
                if (value)
                    input_start_autorepeat(dev, code);
                else
                    input_stop_autorepeat(dev);
            }

            disposition = INPUT_PASS_TO_HANDLERS;
        }
        break;

    case EV_SW:
        if (is_event_supported(code, dev->swbit, SW_MAX) &&
            !!test_bit(code, dev->sw) != value) {

            __change_bit(code, dev->sw);
            disposition = INPUT_PASS_TO_HANDLERS;
        }
        break;

    case EV_ABS:
        if (is_event_supported(code, dev->absbit, ABS_MAX))
            disposition = input_handle_abs_event(dev, code, &value);

        break;

    case EV_REL:
        if (is_event_supported(code, dev->relbit, REL_MAX) && value)
            disposition = INPUT_PASS_TO_HANDLERS;

        break;

    case EV_MSC:
        if (is_event_supported(code, dev->mscbit, MSC_MAX))
```

```
                disposition = INPUT_PASS_TO_ALL;

            break;

        case EV_LED:
            if (is_event_supported(code, dev->ledbit, LED_MAX) &&
                !!test_bit(code, dev->led) != value) {

                __change_bit(code, dev->led);
                disposition = INPUT_PASS_TO_ALL;
            }
            break;

        case EV_SND:
            if (is_event_supported(code, dev->sndbit, SND_MAX)) {

                if (!!test_bit(code, dev->snd) != !!value)
                    __change_bit(code, dev->snd);
                disposition = INPUT_PASS_TO_ALL;
            }
            break;

        case EV_REP:
            if (code <= REP_MAX && value >= 0 && dev->rep[code] != value) {
                dev->rep[code] = value;
                disposition = INPUT_PASS_TO_ALL;
            }
            break;

        case EV_FF:
            if (value >= 0)
                disposition = INPUT_PASS_TO_ALL;
            break;

        case EV_PWR:
            disposition = INPUT_PASS_TO_ALL;
            break;
    }

    if (disposition != INPUT_IGNORE_EVENT && type != EV_SYN)
        dev->sync = false;
    //在分类检索设置之后，需要输入设备的参与，直接回调设备的 event 函数.
    if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
        dev->event(dev, type, code, value);
```

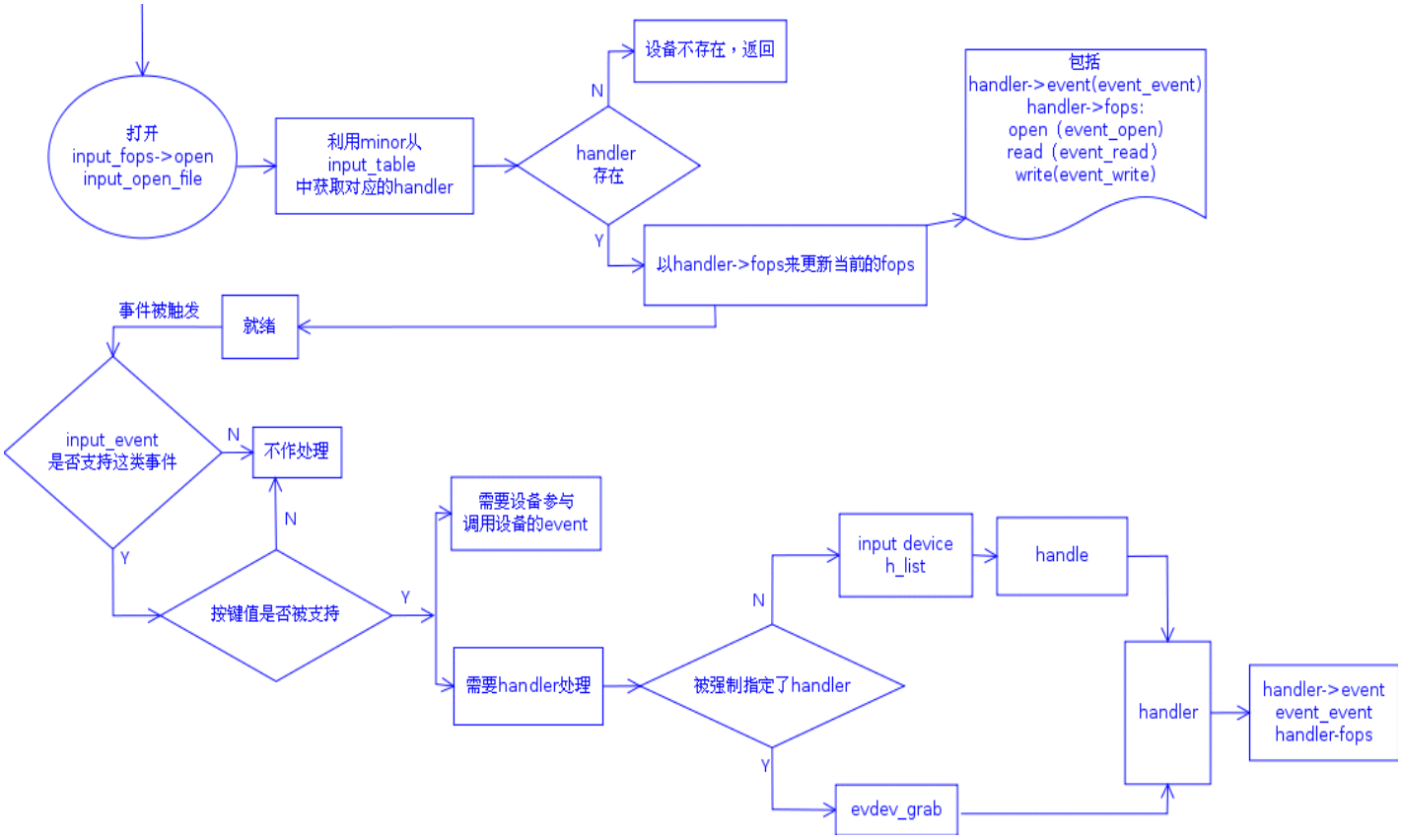
```
    //直接需要对应的 handler 参与. 调用 input_pass_event
    //对于 TQ6410 的按键设备
    if (disposition & INPUT_PASS_TO_HANDLERS)
        input_pass_event(dev, type, code, value);
}
```

解析字符设备处理的过程，到目前，就发现 handler，input_table，事件类型，事件码等都相继的其了它们各自的作用了。下面继续看 input_pass_event(dev, type, code, value); 的处理

```
static void input_pass_event(struct input_dev *dev,
                             unsigned int type, unsigned int code, int value)
{
    struct input_handler *handler;
    struct input_handle *handle;

    rcu_read_lock();

//获取 input device 的被指定的 handle
    handle = rcu_dereference(dev->grab);
    if (handle)//如果存在，这直接利用该 handle 所指向的 handler 中的各种接口来处理
        handle->handler->event(handle, type, code, value);
    else {
        bool filtered = false;
        //否则逐个检索 input device 中的 h_list 链表来获得与之对应的 handle
        //然后在通过 handle 来获得 handler，进一步处理
        list_for_each_entry_rcu(handle, &dev->h_list, d_node) {
            if (!handle->open)
                continue;

            handler = handle->handler;
            if (!handler->filter) {
                if (filtered)
                    break;

                handler->event(handle, type, code, value);

            } else if (handler->filter(handle, type, code, value))
                filtered = true;
        }
    }
    rcu_read_unlock();
}
```

分析至此，已经看到了在注册过程中，所涉及的 input device， handler， handle 它们在事件处理过程的作用，以及注册它们的时候需要设置的一些属性的作用都在事件处理中逐步涉及到，例如事件类型，事件码，input_table，input_device_list，input_handler_list 等。 为了直观，下面以一张图简要的说明一下事

件的 evdev 设备的处理过程。



1.7 小结

至此，对于 Linux 内核的输入子系统的简析就告一段落。在整个子系统中主要围绕着 input device，handle 以及 handler 进行。它们之间的关系，handle 可以说是其它两者之间的桥梁，而 handler 是含有对输入设备 input device 的操作的一些接口。至于输入设备的事件类型以及事件码，在其注册过程是要进行指定的。

第二章　TQ6410 部分驱动解析

2.1 如何将驱动加入到内核

内核的文件很多，为了配合下面的驱动编写说明，这里交代一下如何在内核中增加自己的驱动文件，将自己的驱动编译到内核中去。以较简单的字符设备为例（其他的驱动类似的操作步骤）。

首先要在内核源码的 drivers/char 目录下新建了自己的驱动文件，比如叫做 embedsky.c。

接着修改相同目录下的 Kconfig 文件（drivers/char/Kconfig）。增加一个配置项，这样才能在内核中出现新增加的驱动的选项。例如在 Kconfig 中针对 embedsky.c 增加如下选项：

```
config    EmbedSky_DRIVER    //配置项的名称
      tristate "EMBEDSKY DRIVER" //配置项的说明
      depends on (CPU_S3C6400 || CPU_S3C6410) && MACH_TQ6410 //配置项的依赖关系
      default    y //设置默认情况下被选中
      ---help--- //设置帮助信息
            show help message
```

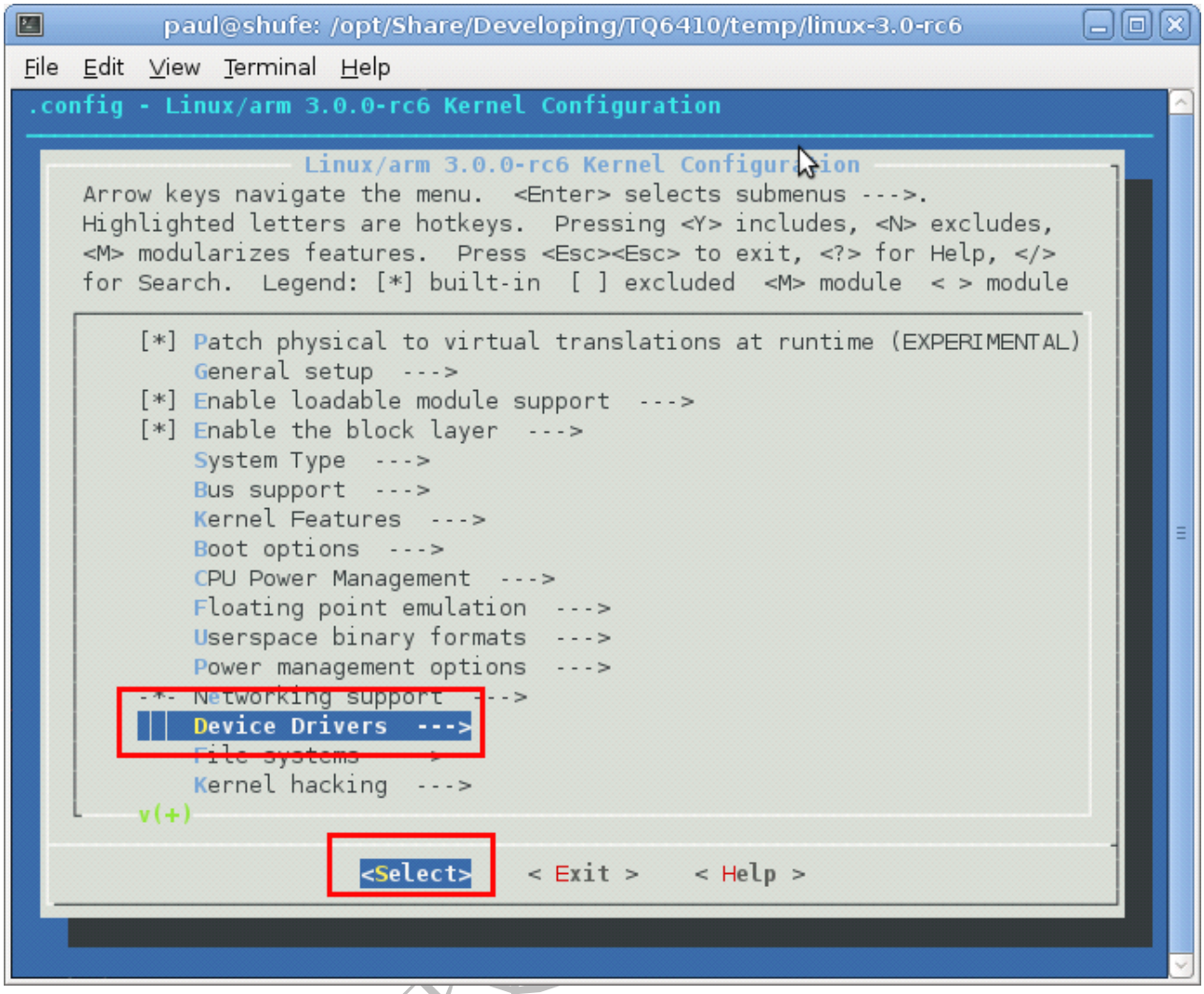
接着在相同文件夹下修改 Makefile（drivers/char/Makefile），增加对应的编译项如下：

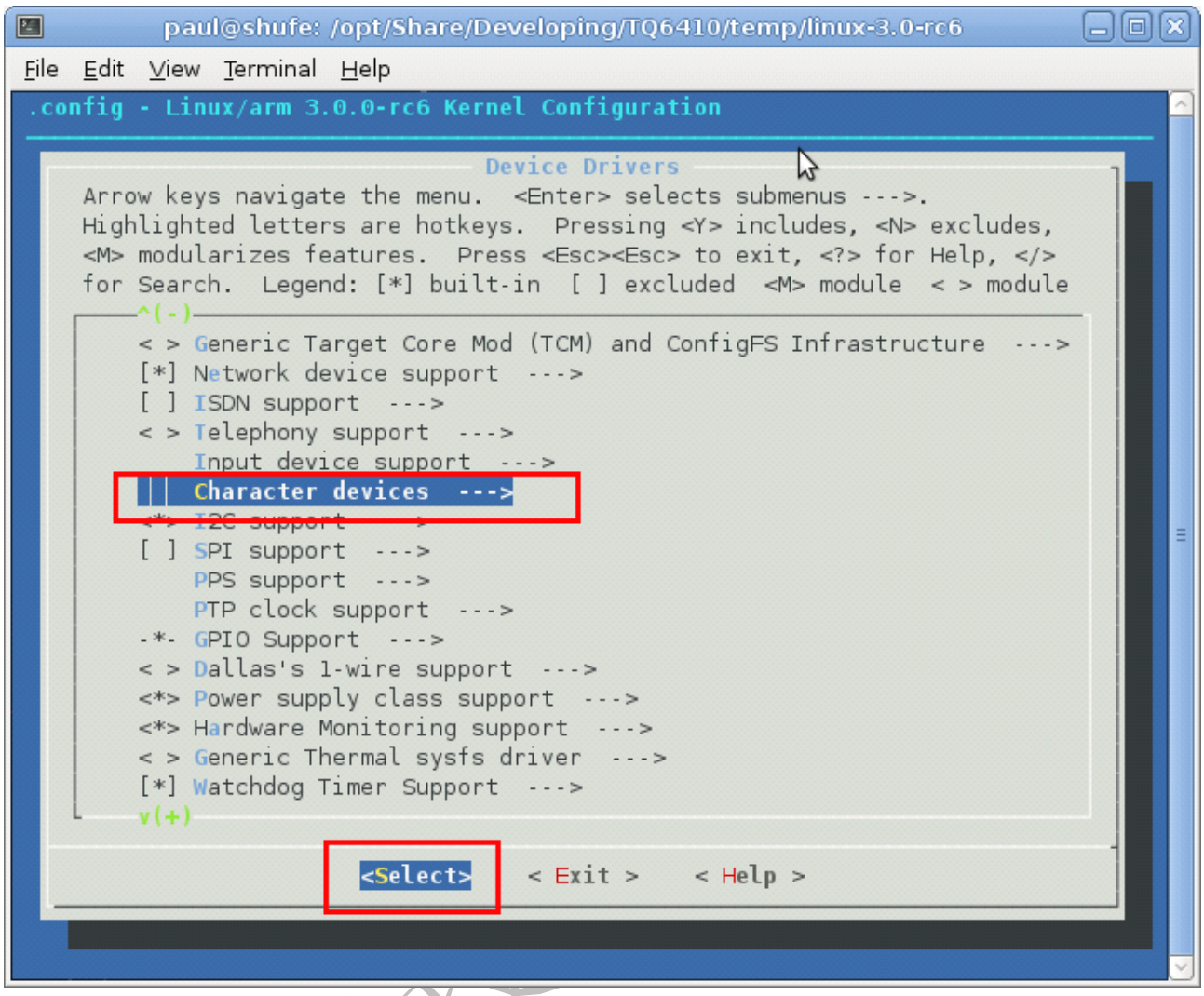
```
obj-$(CONFIG_EmbedSky_DRIVER)        += embedsky.o
//表示如果选上 Kconfig 中的 EmbedSky_DRIVER 选项，那么编译的 embedsky.c 文件
```

设置好以后执行 make menuconfig ，如下图所示：

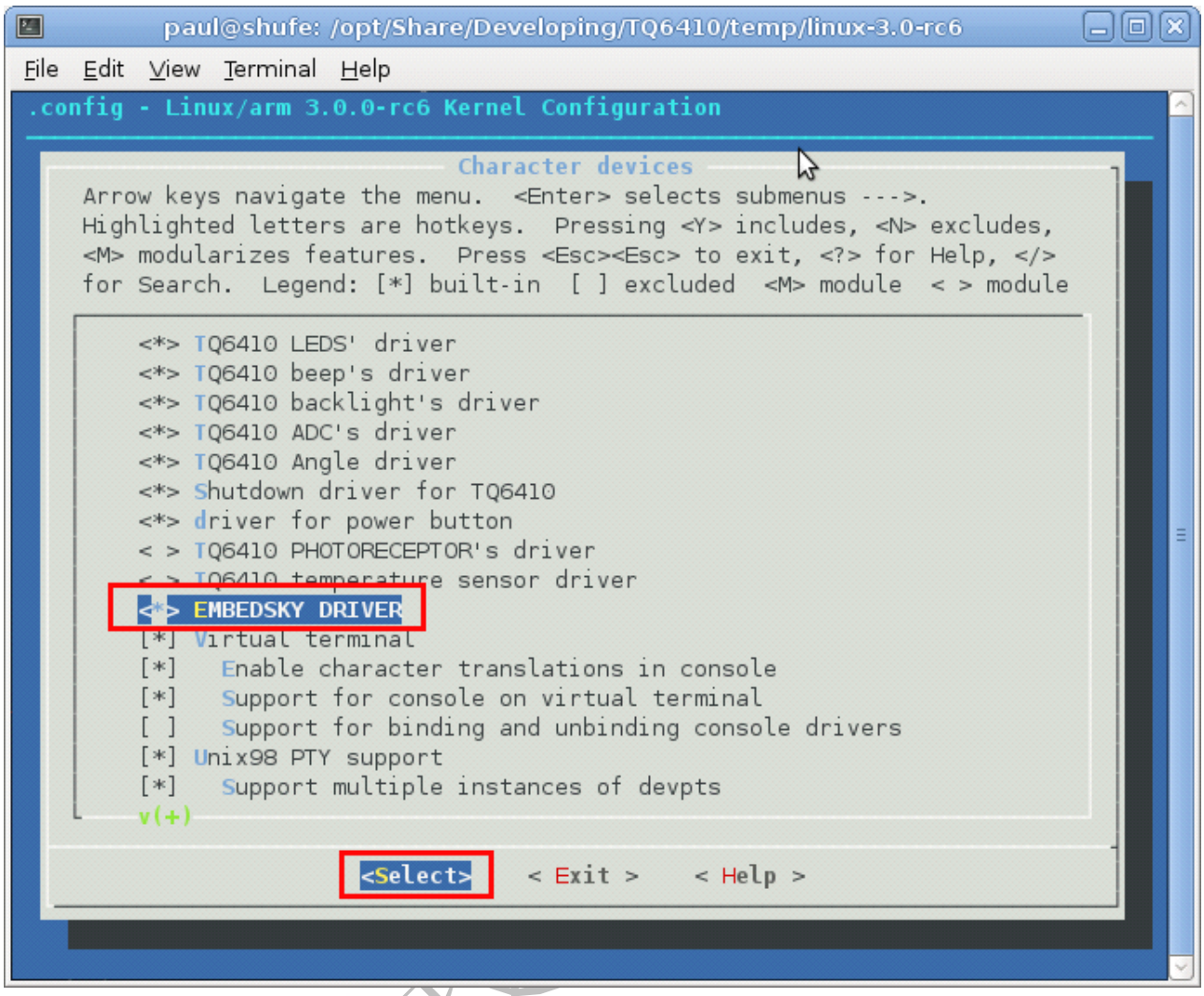
```
AS      .tmp_kallsyms2.o
LD      vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
OBJCOPY arch/arm/boot/Image
Kernel: arch/arm/boot/Image is ready
GZIP    arch/arm/boot/compressed/piggy.gzip
AS      arch/arm/boot/compressed/piggy.gzip.o
SHIPPED arch/arm/boot/compressed/lib1funcs.S
AS      arch/arm/boot/compressed/lib1funcs.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
opy zImage.bin to /opt/tftpboot done.....................
opy zImage.bin to /opt/Share/temp/ done..................
aul@shufe:/opt/Share/Developing/TQ6410/temp/linux-3.0-rc6$ make menuconfig
cripts/kconfig/mconf Kconfig
```

进入字符驱动配置项


paul@shufe: /opt/Share/Developing/TQ6410/temp/linux-3.0-rc6
File Edit View Terminal Help
.config - Linux/arm 3.0.0-rc6 Kernel Configuration
Linux/arm 3.0.0-rc6 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module
[*] Patch physical to virtual translations at runtime (EXPERIMENTAL)
General setup --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
System Type --->
Bus support --->
Kernel Features --->
Boot options --->
CPU Power Management --->
Floating point emulation --->
Userspace binary formats --->
Power management options --->
-*- Networking support --->
Device Drivers --->
File systems --->
Kernel hacking --->
v(+)
<Select> < Exit > < Help >



可以看到刚刚加入的选项：



选择了**<*>**就表示编译进入了内核，如果选择为**<M>** <M> EMBEDSKY DRIVER，表示以模块形式编译。这里选择为<*>,选择好后退出保存，执行 make zImage 就可以将自己的驱动编译进入了内核。

2.2 平台设备注册的必备条件

从 Linux2.6 版本的内核开始，注册设备驱动引入了设备驱动管理机制，也就是 platform device 和 platform drivers。在设备的驱动（platform driver）注册之前，它关联的设备（platform device）必先完成注册。因为这个机制的使得设备注册的顺序是：

创建新的设备以及资源→注册新的设备→创建新的设备驱动→注册新驱动

这样分离的优势是便于内核资源的管理，在设备的驱动进行注册之前，先完成了设备的注册，申请了对应的资源（例如设备中断源，地址源等），在设备的驱动进行注册是，在通过内核提供的设备接口处理函数（例如 platform_get_resource 获得 IO 资源地址）来获取对应的设备资源，完成了驱动的注册。

这里暂时不去深入讨论设备和其驱动之间的注册过程的细节。仅仅必要了解，一个设备的驱动需要在内核中注册需要以下基本的条件：

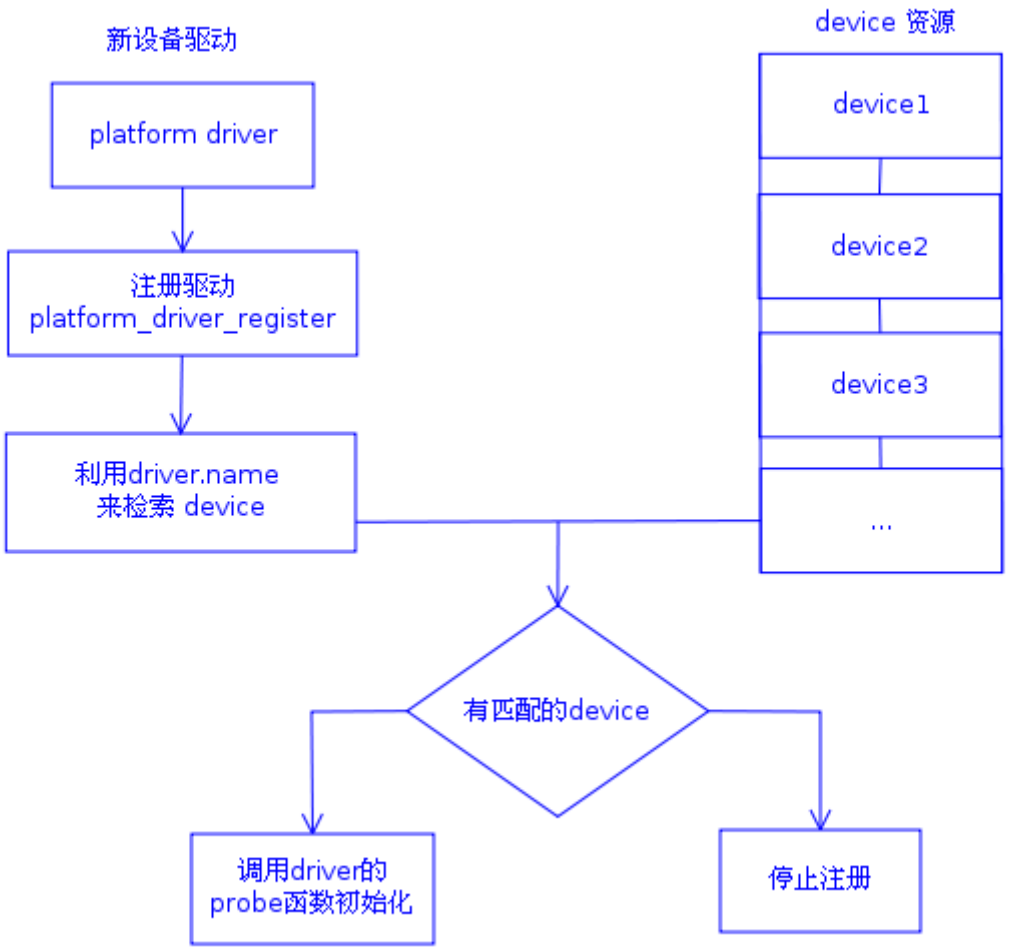
a）有 platform device，其中 device 必须必备驱动注册需要的信息，数据。不同的驱动要求具体也有所不同；

b）platform device 的名称必须和 platform driver 的名字一致，因为在驱动注册时，是靠这个名字来检索对应的 device 获得对应的注册资源信息的；

c）platform device 必先完成注册；

d）创建新的 platform driver 便完成其注册的函数。

platform driver 和 platform device 在驱动注册过程中的关系下图所示：



2.3 HS0038 红外接收头驱动

针对于 TQ6410 开发板上的 hs0038 红外接收头的驱动位于内核源码的 drivers/input/keyboard/tq6410_hs0038.c ，这个仅仅是 platform driver。从 **2.1 章节**中得知，还需要 platform device 。所以它的 platform device 在 arch/arm/mach-s3c64xx/dev-keypad. 文件中的 **tq6410_hs0038_device.**

2.3.1 设备（platform device）注册

a）设备创建

结合红外接收头的 device，也了解一下 platform device 这个结构体，给结构体定义在

include/linux/platform_device.h 中。

```
struct platform_device {
    const char      * name; //设备的名称，必须和 driver 的名称一致
    int             id;   //id 用来指定 bus_id 号，应为同一个驱动可能有多个设备，例如串口
    struct device   dev; //设备的其他详细信息，例如数据，继承，链表关系等
    u32             num_resources; //设备资源数组成员个数
    struct resource * resource; //设备的资源
    const struct platform_device_id     *id_entry; //用于一个驱动多种设备的 id 指定，不常用
    /* MFD cell pointer */
    struct mfd_cell *mfd_cell;
    /* arch specific additions */
    struct pdev_archdata        archdata;
};
```

在这一个结构体中，此时需要关心的是 name，id，dev 中的 platform_data 就可以，所以 hs0038 设备的定义如下：

```
struct platform_device tq6410_hs0038_device = {
    .name           = "tq6410_hs0038",//指定的设备名称
    .id             = -1,//因为 hs0038 仅仅只有一个设备，所以用 id =-1
    .dev            = {
            .platform_data = &tq6410_hs0038_data, //驱动注册时需要的资源
    }
};
```

在 hs0038 的 device 中，看到了一个 tq6410_hs0038_data 这个数据结构体实例，它是 gpio_keys_platform_data 按键数据的结构体实例，gpio_keys_platform_data 结构体定义在 linux/gpio_keys.h 文件中。

```
struct gpio_keys_platform_data {
    struct gpio_keys_button *buttons; //按键的资源数组
    int nbuttons; //按键个数
    unsigned int poll_interval;     /* polling interval in msecs -
                                       for polling driver only */
    unsigned int rep:1;             /* enable input subsystem auto repeat */
    int (*enable)(struct device *dev); //使能设备的接口处理函数
    void (*disable)(struct device *dev);//禁用设备的接口处理函数
    const char *name;               //输入按键的名称
};
```

同理这个结构体，这时候需要注意的是 buttons 以及 nbuttons 成员就可以。所以 tq6410_hs0038_data 定义如下：

```
static struct gpio_keys_platform_data tq6410_hs0038_data = {
            .buttons   = &tq6410_hs0038,//按键信息数组
            .nbuttons = 1,//按键个数，因为就设置一个红外接收头
};
```

接下来要组合一个按键信息数组 tq6410_hs0038 ，以便上边的 tq6410_hs0038_data 调用。tq6410_hs0038 就是 gpio_keys_button 结构体的实力，它同样定义在 linux/gpio_keys.h 中

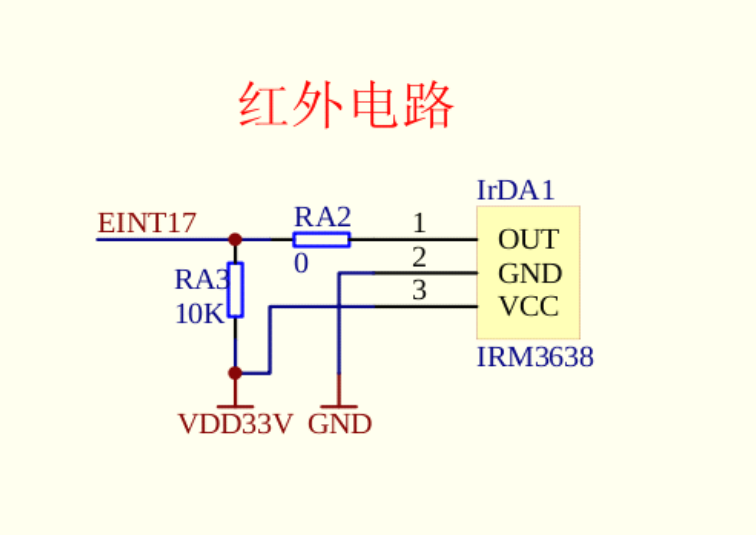
```
struct gpio_keys_button {
```

```
        /* Configuration parameters */
        unsigned int code;  //事件码的指定，在第一章有提过
        int gpio;  //关联的 GPIO 口，就是处于中断脚的 GPIO
        int active_low; //是否要方向处理，高变成低，低变成高
        const char *desc; //对该按键的一点描述
        unsigned int type;  //按键事件类型，在第一章提过
        int wakeup;         //是否作为唤醒源
        int debounce_interval;
        bool can_disable;
        int value;      //
};
```

对于 hs0038 的按键信息， 只要关心的是它的 GPIO 口以及描述 desc 信息就足够了， 因为事件码，事件类型直接在驱动中去设定就可以了，所以 hs0038 的 device 按键信息如下：

```
static struct gpio_keys_button tq6410_hs0038 = {
    .gpio           = S3C64XX_GPL(9),/* 4(- + backspace enter) */
    .desc           = "hs0038",
};
```

从 TQ6410 的底板原理图中，红外电路的一部分，可以知道它是利用外部中断 17 ，对应着 GPL9 GPIO 引脚



b）设备注册（增加）

这样针对 hs0038 的设备结构体定义完成，接下来要将其加入到平台中，也就是注册。打开文件 arch/arm/mach-s3c64xx/mach-tq6410.c ， 里面有一个 tq6410_devices 数组，这个设备数组在平台初始化的时候通过 platform_add_devices(tq6410_devices, ARRAY_SIZE(tq6410_devices)); 来将里面的设备完成了注册。所以要将刚刚完成的 hs0038 设备加入到该数组中（tq6410_devices）；

2.3.2 hs0038 驱动注册

HS0038 红外接收头驱动是 drivers/input/keyboard/tq6410_hs0038.c 文件，对于平台设备驱动的注册，当其通过了和预先注册的设备，通过名字查找匹配后，就会调用 probe 接口函数来完成注册过程。先了解一下 platform driver 的结构，platform_driver 定义于 include/linux/platform_device.h 中。

```
struct platform_driver {
    int (*probe)(struct platform_device *);//驱动初始化时调用
    int (*remove)(struct platform_device *);//驱动卸载时调用
    void (*shutdown)(struct platform_device *);//关机的时候调用
    int (*suspend)(struct platform_device *, pm_message_t state);//进入休眠时调用
    int (*resume)(struct platform_device *);//唤醒时调用
    //驱动的其他属性设置，例如名称，依赖关系
    //其名称可以用来检索匹配的 platform device。（如果 id_table 是空的时候)
    struct device_driver driver;
    //可用的设备 id,针对一种驱动配合多种设备的情况,这个也是出于代码利用来
    //考虑的结果,驱动注册时，先通过 id_table 来找匹配设备（如果 id_table 不空)
    const struct platform_device_id *id_table;
};
```

驱动要注册，必须要有生成一个驱动结构的实例，这个类似于设备的注册一样，需要一个设备结构的实例。所以针对这一点，将关于 hs0038 驱动的结构实例定义如下：

```
static struct platform_driver s3c_hs0038_driver = {
    //指定驱动初始化时调用的接口函数
    .probe          = tq6410_hs0038_probe,
    //指定驱动卸载是调用的接口
    .remove         = __devexit_p(tq6410_hs0038_remove),
    //设置驱动的属性
    .driver         = {
        .name   = "tq6410_hs0038",//设置其名称，必须和 platform device 的一致
        .owner  = THIS_MODULE,//所属关系
#ifdef CONFIG_PM //电源管理接口函数
        .pm = &tq6410_hs0038_pm_ops,
#endif
    },
};
```

以上是设备驱动注册的必备条件之一，另外，设备驱动注册的主要框架还的提供 module_init 和 module_exit 接口。所以以下的函数框架是注册设备驱动必须的。这些要点在 2.1 章节有提过。

```
//驱动初始化时第一时间调用的，在调用之后才去找和 driver 匹配
//的 device，如果找到匹配的才调用 probe 指定的接口函数进行初始化。
static int __init tq6410_hs0038_init(void)
{   //注册 platform driver
    return platform_driver_register(&s3c_hs0038_driver);
}

//驱动卸载的最后调用函数
static void __exit tq6410_hs0038_exit(void)
{//卸载驱动，在 remove 接口被处理完毕之后
    platform_driver_unregister(&s3c_hs0038_driver);
}
//通过以下两个函数来告知内核该驱动的初始化和卸载的接口
```

```
module_init(tq6410_hs0038_init);//注册驱动接口设置
module_exit(tq6410_hs0038_exit);//卸载驱动接口设置

MODULE_AUTHOR("www.embedsky.net");//作者
MODULE_DESCRIPTION("tq6410 GPIO Keyboard Driver");//驱动描述
MODULE_LICENSE("GPL");//许可为 GPL
```

到此，整个驱动的框架已经搭建完成，接下来需要完成的是 platform driver 实例中指定，也就是 driver 实例中的 probe,remove 的指定的接口函数。概要它们的接口如下（上面的 s3c_hs0038_driver 中设置）：

```
.probe          = tq6410_hs0038_probe,
.remove         = __devexit_p(tq6410_hs0038_remove),
.pm  = &tq6410_hs0038_pm_ops,
```

所以的主要任务是完成 tq6410_hs0038_probe，tq6410_hs0038_remove 以及 tq6410_hs0038_pm_ops 的定义。首先要定义的是 tq6410_hs0038_probe 的定义，只有完成了该定义了才能知道 tq6410_hs0038_remove 的定义。tq6410_hs0038_probe 定义如下：

对于 probe 函数的参数，在上面的 struct platform_driver 介绍时了解它需要一个 platform_device 指针。现在针对该函数的定义进行解析如下：

```
static int __devinit tq6410_hs0038_probe(struct platform_device *pdev)
{
    int ret = 0;
    //下面是获取之前提到的 platform device 时注册的 device 数据
    tq6410_hs0038_data
    struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
```

```
#ifdef CONFIG_KEYBOARD_HS0038
static struct gpio_keys_button tq6410_hs0038[] = {
        {
                .gpio           = S3C64XX_GPL(9),/* 4(- + backspace enter) */
                .desc           = "hs0038",
        },
};
static struct gpio_keys_platform_data tq6410_hs0038_data = {
        .buttons        = tq6410_hs0038,
        .nbuttons       = ARRAY_SIZE(tq6410_hs0038),
};

struct platform_device tq6410_hs0038_device = {
        .name           = "tq6410_hs0038",
        .id             = -1,
        .dev            = {
                .platform_data  = &tq6410_hs0038_data,
        }
};
#endif
```



```
    struct device *dev = &pdev->dev;
    struct input_dev *input;
    //该结构体是为了在整个驱动中方便设备数据之间的传递而设置的
    //至于它的作用（为什么要有它的存在），后面就会介绍到
    struct tq6410_hs0038 *hs0038;
```

```
struct tq6410_hs0038
{
        unsigned int keycode[ARRAY_SIZE(tq6410_hs0038_keycode)];
        struct input_dev *input;
        spinlock_t lock;
        struct gpio_keys_button *button;
};
```

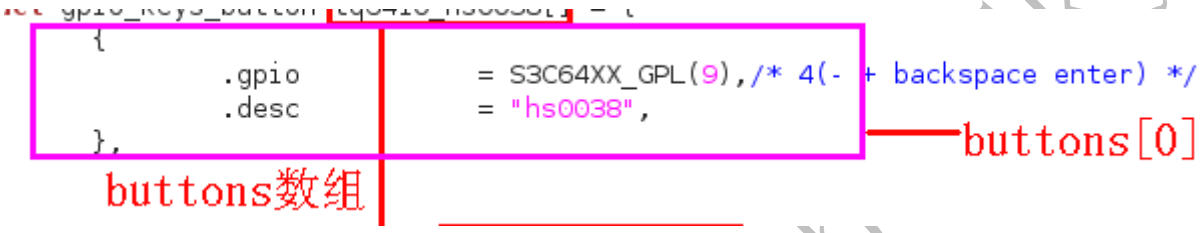
事件码

新注册的输入设备

关联的button

```
int i=0;
printk("%s\n\n",__func__);
//申请内存空间，并将地址赋给 hs0038,因为需要它来存放数据
hs0038= kzalloc(sizeof(struct tq6410_hs0038),GFP_KERNEL);
//这里应该有印象，在 1.4 章节中，提到必须利用该函数为新的
//设备驱动分配空间
input = input_allocate_device();
//判断以上两种内存分配是否成功
if (!input || !hs0038)
{
        dev_err(dev, "failed to allocate state\n");
        ret= -ENOMEM;
        goto err_free_mem;
}
//初始化自旋锁
spin_lock_init(&hs0038->lock);
//给刚刚生成的 hs0038 对象初始化，因为利用它来传递一些数据
hs0038->input = input;
//设置平台驱动的数据，这里也比较重要，只有设置了平台数据，
//在后面的其他函数中才能方便的共享平台数据（有利于数据资源之间的共享）
platform_set_drvdata(pdev, hs0038);
//初始化新的输入设备驱动信息，包括名称，总线类型，开发商，版本信息等
//它们将来有帮助与该设备驱动的 handler 的匹配，这个也是在 1.5.2 章节中介绍了
input->name = "tq6410_hs0038";
input->phys = "tq6410_hs0038/input1";

input->id.bustype = BUS_HOST;
input->id.vendor = 0xabce;
input->id.product = 0xecba;
input->id.version = 0x0100;
//指定事件类型，在 1.5.2 章节中有介绍
input->evbit[0] = BIT(EV_KEY);
//指定事件码，也可以用 set_bit(code, input->keybit);来实现
input->keycode = hs0038->keycode;
input->keycodesize = sizeof(unsigned int);//一个事件码的大小
input->keycodemax = ARRAY_SIZE(tq6410_hs0038_keycode);//总体大小
```

```
    set_bit(EV_KEY, input->evbit);//设置类型


    for (i = 0; i < ARRAY_SIZE(tq6410_hs0038_keycode); i++){
        //初始化可用的事件码，自己定义数组 tq6410_hs0038_keycode 来
        //指定事件码，这样方便日后维护修改
        hs0038->keycode[i]= tq6410_hs0038_keycode[i];
        set_bit(hs0038->keycode[i], input->keybit);//设置键值
    }
    //继续初始化 hs0038，pdata 就是之前定义的 device 中的
    //设备数据,从图 2-5 中我么可以知道 button[0]是如下图的结构
```




```
    hs0038->button = &pdata->buttons[0];
    //到此，看到了设备驱动初始化设置了事件类型，事件码
    //但是还没有看到说的要制定一个触发时的入口函数
    //下面这个函数就是自己定义来处理这个事情的。接下来在具体看看
    ret=hs0038_setup_key(pdev);
    if(ret)
        goto err_gpio_free; //初始化出错处理
    //在初始化相关结构体，变量等之后，新的设备驱动进入真正的注册函数
    ret = input_register_device(input);
    if (ret < 0){
        printk(KERN_ERR "tq6410_hs0038.c: input_register_device() return %d!\n", ret);
        goto   err_input_free;
    }
    printk("Input: S3C GPIO   hs0038 Registered\n");
    return 0;
//出错处理
err_input_free: //注销新对象 input
    input_free_device(input);
err_gpio_free: //释放关联的 GPIO 口
    gpio_free(pdata->buttons[0].gpio);
err_free_mem: //释放 input 占用的内存
    kfree(input);
    kfree(hs0038);
    return ret;
}
```

通过 probe 接口函数，已经完成了注册输入设备驱动的基本要求 1、2 条，如下：

1、驱动中要有一个由输入设备触发的处理函数

这个处理过程主要是向子系统汇报事件，向上层交代处理结果。

2、一个初始化函数

初始化包括注册触发时的处理接口，创建新设备，设置事件类型，设置事件码。

而当中的函数是完成了事件触发时第一响应的处理函数 hs0038_setup_key(pdev,hs0038);设置，看看其定义：

```
static int __devinit hs0038_setup_key(struct platform_device *pdev)
{
    //获取驱动中的设备平台数据，到这里可以看到
    //定义 ta6410_hs0038 结构体（截图如下），以及在 probe 函数中。
```

```
struct tq6410_hs0038
{
        unsigned int keycode[ARRAY_SIZE(tq6410_hs0038_keycode)];
        struct input_dev *input;
        spinlock_t lock;
        struct gpio_keys_button *button;
};
```

事件码

新注册的输入设备

关联的button

```
    //为其申请空间，初始化其成员的目的了，而且利用 platform_set_drvdata(pdev, hs0038);
    //这句代码将其设置为驱动数据
    struct tq6410_hs0038 *hs0038 = platform_get_drvdata(pdev);//获取平台数据
    //从 hs0038 结构体中读取它的 button 成员
    struct gpio_keys_button *button=hs0038->button;
    //再从 button 成员中获取 desc 属性，这个在 platfomr device 中定义了
    const char *desc = button->desc ? button->desc : "hs0038";
```

```
    {
            .gpio           = S3C64XX_GPL(9),/* 4(- + backspace enter) */
            .desc           = "hs0038",
    },
```

buttons[0]

buttons数组

```
    unsigned long irqflags;
    int irq, error;
//申请 GPIO 的使用权，S3C64XX_GPL(9)
    error = gpio_request(button->gpio, NULL);
    if (error < 0) {
        printk("failed to request GPIO %d, error %d\n",button->gpio, error);
        goto fail2;
    }
//设置对应的 GPIO [S3C64XX_GPL(9)]为输入
    error = gpio_direction_input(button->gpio);
    if (error < 0) {
        printk("failed to configure direction for GPIO %d , error %d\n",button->gpio,error);
        goto fail3;
    }
```

```
//这对相应的 GPIO(S3C64XX_GPL(9))获取其中断号
    irq = gpio_to_irq(button->gpio);
    if (irq < 0) {
        error = irq;
        printk("Unable to get irq number for GPIO %d,error %d\n",button->gpio,error);
        goto fail3;
    }
//设置中断类型为下降沿触发
    irqflags = IRQF_TRIGGER_FALLING;
//注册中断，其中中断处理函数是 tq6410_hs0038_interrupt。
//这就是说的事件触发的第一响应处理函数，利用了中断。
//这里稍微说一下 request_irq 的参数。
//第一个 irq 是中断号，第二个是和该中断对应的处理函数，
//第三个是中断类型设置，第四个是对中断的一些描述，
//最后一个是 dev_id 的对象，它在中断处理函数中可以直接访问，所以传递了 hs0038 对象，因为需要利用
//到它
    error = request_irq(irq, tq6410_hs0038_interrupt, irqflags, desc, hs0038);
    if (error) {
        printk("Unable to claim irq %d; error %d\n",irq, error);
        goto fail3;
    }
    return 0;

fail3:
    gpio_free(button->gpio);
fail2:
    return error;
}
```

至此，在 1.4 章节中说明的注册输入设备驱动的几个基本条件的 1，2，也还没有真正得到完成，因为上面的函数 hs0038_setup_key 中，在注册中断时又有了一个中断处理函数 **tq6410_hs0038_interrupt**，一环套一环，还是得进一步完成中断处理函数。

tq6410_hs0038_interrupt 如下：

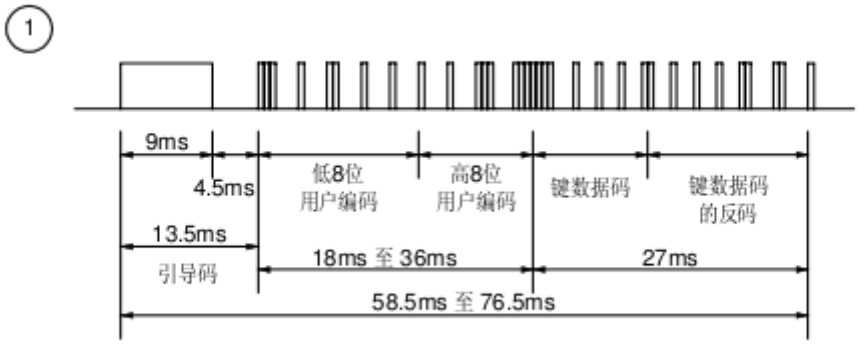
```
static irqreturn_t tq6410_hs0038_interrupt(int irq, void *dev_id)
{
    //dev_id 是在执行
    //request_irq(irq, tq6410_hs0038_interrupt, irqflags, desc, hs0038);
    //时传递的第 5 个参量,传递了 hs0038
    struct tq6410_hs0038 *hs0038 = dev_id;
    unsigned long irqflags = IRQF_TRIGGER_FALLING;
    //进入了中断处理时，暂时关闭中断
    disable_irq_nosync(irq);
    //进入主要的处理过程，也是整个驱动的要点之一
    tq6410_hs0038_read(hs0038,hs0038->button);
    //重新设置中断类型
```

```
    irq_set_irq_type(irq, irqflags);
    //中断处理已经完毕，再一次使能中断
    enable_irq(irq);
    return IRQ_HANDLED;
}
```

分析到此为止，有设备驱动的 probe 函数引出的 hs0038_setup_key 函数，接着由 hs0038_setup_key 调用的 tq6410_hs0038_interrupt 处理函数，和中断中真正处理的函数 tq6410_hs0038_read(hs0038,hs0038->button);使得分析 probe 函数似乎没完没了，其实这些代码的逻辑规划，对于一个驱动开发者来说，是在驱动编写过程一步一步分化出来而已，并不是复杂化，而是将代码优化，便于管理。那么接下来就看看重要的一个处理函数 tq6410_hs0038_read(hs0038,hs0038->button);

红外发射器为 SC6121。代码如下：

```
static int tq6410_hs0038_read(struct tq6410_hs0038 *hs0038_data,struct gpio_keys_button *button)
{

        u32 values[4];
        int i=0 , ret = 0;
        ret = gpio_direction_input(button->gpio);

        if (ret < 0) {
            printk("failed to request GPIO %d, error %d\n",button->gpio, ret);
            return ret;
        }
```




```
        //从 SC6121 的发送时需图看，在按键被按下以后先等待 9ms 的上升沿
        //的到来
        while(((__raw_readl(S3C64XX_GPLDAT)>>9) & 1)==0);//在 9ms 内判断 IO 口的值
        //当 9ms 的上升沿到来后，又要等待它进入下降沿，才能进入数据的读取
        while(((__raw_readl(S3C64XX_GPLDAT)>>9) & 1)==1);//等待低电平
        {
            //这里的 i 循环次数设置，是避免意外，在其一直处于高电平时导致程序死循环
            //所以在等待了 4 个 800ns 后，会以失败方式结束读工作
            udelay(800);
            i++;
            if(i>4)
```

```
            return -1;
        }
        //开始读取数据，不去读取第 4 个字节（按键的反码）
        //下面的截图就是 serial_data_read_byte() 函数的过程
        values[0] = serial_data_read_byte(); //读取第一个用户编码
        values[1] = serial_data_read_byte(); //读取第二个用户编码
        values[2] = serial_data_read_byte(); //读取按键数据，这才是真正需要的数据
```

```
static u32 __inline__ serial_data_read_byte(void)
{

        int j;
        u32 CodeTemp=0;
        for(j=1;j<=8;j++) //每个字节8个bit的判断
        {
                //等待上升沿，也就数据位的到来
                while(((__raw_readl(S3C64XX_GPLDAT)>>9) & 1)==0);
                udelay(900);//这些延迟都是调试获得

                //判断该位是否高(1),因为我们用GPL9，所以对应数据寄存器中的位9
                if(((__raw_readl(S3C64XX_GPLDAT)>>9) & 1)==1)
                {
                        udelay(900);//这些延迟都是调试获得
                        CodeTemp=CodeTemp|0x80;//如果是1，高位置1

                        //是从低位开始接收的，所以接满8位前，每接一次，将其右移1位
                        if(j<8) CodeTemp=CodeTemp>>1;
                }
                else //传来的数据位是0
                if(j<8)CodeTemp=CodeTemp>>1;//接满8位前，每接一次，将其右移1位，自动补"0"，
        }

        return CodeTemp;//返回接收的字节数据

}
```

```
        if((values[0]==15)&&(values[1]==255))//判断用户编码，这些有硬件特性决定
        {
            i=values[2]; //获得按键码，i 的取值在[0,15]
            //向输入子系统报告事件，让其处理，1.6 章节中讲过子系统的处理过程
            input_report_key(hs0038_data->input,hs0038_data->keycode[i], 1);
            input_report_key(hs0038_data->input,hs0038_data->keycode[i], 0);
            //hs0038_data->keycode 数组就有 16 个 Linux 内核定义好的键值
```

```
static const unsigned int tq6410_hs0038_keycode[16]=
{
        KEY_7,KEY_4,KEY_1,KEY_0,
        KEY_8,KEY_5,KEY_2,KEY_KPASTERISK,
        KEY_9,KEY_6,KEY_3,KEY_BACKSLASH,
        KEY_KPMINUS,KEY_KPPLUS,KEY_BACKSPACE,KEY_ENTER
};

struct tq6410_hs0038
{
        unsigned int keycode[ARRAY_SIZE(tq6410_hs0038_keycode)];
        struct input_dev *input;
        spinlock_t lock;
        struct gpio_keys_button *button;
};
```

16个按键的键值，便于更改

```
                //等待处理的完成
                input_sync(hs0038_data->input);
                return 0;
        }
        return 0;

    }
```

逐步解析到此，针对 hs0038 红外接收器的 platform driver 编写已经完成了驱动基本条件的前两则：

1、驱动中要有一个由输入设备触发的处理函数

这个处理过程主要是向子系统汇报事件，向上层交代处理结果。

2、一个初始化函数

初始化包括注册触发时的处理接口，创建新设备，设置事件类型，设置事件码。

现在再需要完成第 3 点。那就是一个注销的函数处理，就是驱动中的.remove 对应的接口 tq6410_hs0038_remove,它的主要代码如下：

```
static int __devexit tq6410_hs0038_remove(struct platform_device *pdev)
{
    //获取平台数据，这个在 probe 函数中设置了的
    //这里看到，驱动中的一些数据传递和共享的方式，是比较方便了.
    //在 probe 函数中利用 platform_set_drvdata(pdev, hs0038);进行了设置
    struct tq6410_hs0038 *s3c_hs0038 = platform_get_drvdata(pdev);
    //获取驱动数据
    struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;

    int i=0;

    for (i = 0; i < pdata->nbuttons; i++) {
//释放中断，这个在 probe 函数中注册了中断，所以这里必须释放
        int irq = gpio_to_irq(pdata->buttons[i].gpio);
        free_irq(irq, &pdata->buttons[i]);
```

```
//释放中断对应的 GPIO，在前面有通过 gpio_request 来获得它的使用权
            gpio_free(pdata->buttons[i].gpio);
        }
        //注销新注册的 input 设备
        input_unregister_device(s3c_hs0038->input);
        return 0;
}
```

这样关于 hs0038 的接收器，可以说是完成。不过 TQ6410 要处理电源管理。所以也要定义一下关于电源管理的接口处理函数。.pm = &tq6410_hs0038_pm_ops 这就是指定电源管理的接口结构，其定义如下:

```
static const struct dev_pm_ops tq6410_hs0038_pm_ops = {
        .suspend  = tq6410_hs0038_suspend, //休眠时调用的函数
        .resume         = tq6410_hs0038_resume,//唤醒时调用的函数
};
```

所以针对这两个函数进行定义。

```
static int tq6410_hs0038_suspend(struct device *dev)
{
          //获取 platform device
        struct platform_device *pdev = to_platform_device(dev);
        //再从 platform device 中获取平台数据。
        //在 probe 函数中利用 platform_set_drvdata(pdev, hs0038);进行了设置
        struct tq6410_hs0038 *s3c_hs0038 = platform_get_drvdata(pdev);
        //将对应的 GPIO 口设置为输入
        s3c_gpio_cfgpin(s3c_hs0038->button->gpio, S3C_GPIO_SFN(0));
                                    return 0;
}

static int tq6410_hs0038_resume(struct device *dev)
{
        //获取 platform device
        struct platform_device *pdev = to_platform_device(dev);
        //再从 platform device 中获取平台数据。
        //在 probe 函数中利用 platform_set_drvdata(pdev, hs0038);进行了设置
        struct tq6410_hs0038 *s3c_hs0038 = platform_get_drvdata(pdev);
        //将对应的 GPIO 口设置为中断功能
        s3c_gpio_cfgpin(s3c_hs0038->button->gpio, S3C_GPIO_SFN(3));
              return 0;
}
```

至此 hs0038 红外接收头的驱动已经完成了分析。在这一节的分析过程，一直围绕着 platform device 的结构体中需要的接口函数逐个分析，如下图显示。

```
static struct platform_driver s3c_hs0038_driver = {
        .probe          = tq6410_hs0038_probe,
        .remove         = __devexit_p(tq6410_hs0038_remove),
        .driver         = {
                .name   = "tq6410_hs0038",
                .owner  = THIS_MODULE,
#ifdef CONFIG_PM
                .pm     = &tq6410_hs0038_pm_ops,
#endif
        },
};
```

初始化

卸载处理

电源管理

在分析过程中，不断重复 platform device 的注册的一些要点，以及针对 hs0038 的时序，原理，对代码进行了一些解析。这一过程主要目的是通过驱动去温习 Linux 系统中的输入子系统设备驱动的注册的步骤以及具体的实现，以便进一步理解在第一章中所描述的知识点。

2.4 按键驱动

这一节中，继续结合 Linux 输入子系统，分析 TQ6410 的按键驱动，加深对输入设备驱动的注册与 Linux 内核输入子系统的理解。如图 2-7 所示按键部分的电路图，按键驱动有 6 个按键，从 TQ6410 的底板原理图中，可以知道它利用外部中断 0-5 ，对应 GPIO 口的 GPN0-GPN5，所以 GPIO 设置的时 GPN0-GPN5。用开发板上 LED 灯亮灭来标示键按下的状态。当 KEY1、KEY3、KEY5 按下时 LED1 亮，当 KEY2、KEY4、KEY6 按下时 LED2 亮。

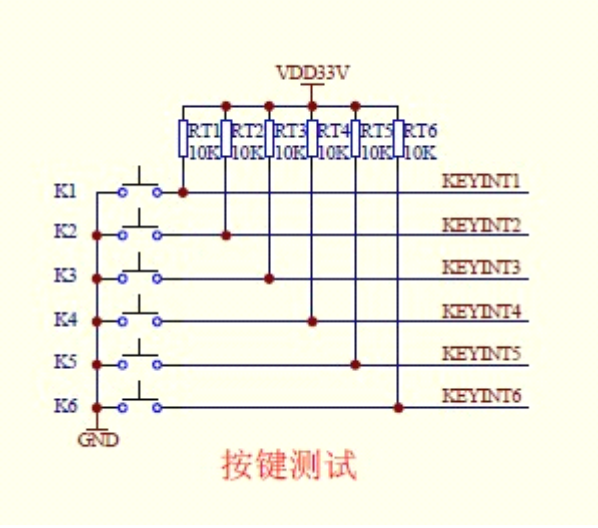


图 2-19

针对于 TQ6410 开发板上的按键驱动位于内核源码的 drivers/input/keyboard/tq6410_buttons.c ，这个仅仅是 platform driver。从 2.1 章节中得知，还需要 platform device 。 所以它的 platform device 在 arch/arm/mach-s3c64xx/dev-keypad.c 文件中的 s3c_device_gpio_button.

2.4.1 设备（platform device）注册

a）设备创建

在上一节中具体介绍的 platform device 结构体中，此时需要关心的是 name，id，dev 中的 platform_data 就可以，buttons 设备的定义如下：

```
struct platform_device s3c_device_gpio_button = {
    .name           = "tq6410-keys",//指定按键的设备名称
    .id             = -1, //因为按键仅仅只有一个设备，所以用 id =-1
    .num_resources      = 0, //按键设备数量
    .dev            = {
    .platform_data = &gpio_button_data, //驱动注册时需要的资源
    }
};
```

在 buttons 的 device 中，看到了一个 gpio_button_data 这个数据结构体实例，它是 gpio_keys_platform_data 按键数据的结构体实例，gpio_keys_platform_data 结构体已在 1.2.1 介绍，在此不在重复介绍。这个结构体中，这时候重点看看 gpio_button_data 中的 buttons 和 nbuttons 成员，gpio_button_data 定义如下：

```
static struct gpio_keys_platform_data gpio_button_data = {
    .buttons  = gpio_buttons,//按键资源数组信息
    .nbuttons = ARRAY_SIZE(gpio_buttons),//按键个数
};
```

接下来要组合一个按键资源数组信息 gpio_buttons ，以便供上边的 gpio_button_data 调用。gpio_buttons 就是 gpio_keys_button 结构体的实例数组。关于 gpio_keys_button 结构体成员可参照在 1.2.1 章节介绍的。

对于 buttons 的按键信息，根据开发板上使用的 GPIO 端口，定义了 6 个按键的信息，分别设置了 GPIO 端口、事件码、desc 信息、触发电平反向标志以及 wakeup 唤醒源标志。最后 buttons 的 device 按键信息如下：

```
static struct gpio_keys_button gpio_buttons[] = {
    {//UP 1
        .gpio           = S3C64XX_GPN(0),//对应的 GPIO 口
        .code           = KEY_UP,//键值设置
        .desc           = "KEY_UP",//描述
        .active_low     = 1,//设置为反向读取电平
        .wakeup         = 0,//不作为唤醒源
    },
    {//DOWN 2
        .gpio           = S3C64XX_GPN(1),
        .code           = KEY_DOWN,//#define KEY_DOWN      108
        .desc           = "KEY_DOWN",
        .active_low     = 1,
        .wakeup         = 0,
    },
    {//LEFT 3
        .gpio           = S3C64XX_GPN(2),
```

```
        .code           = KEY_LEFT,//#define KEY_LEFT 105
        .desc           = "KEY_LEFT",
        .active_low     = 1,
        .wakeup         = 0,
    },
    {//RIGHT 4
        .gpio           = S3C64XX_GPN(3),
        .code           = KEY_RIGHT,
        .desc           = "KEY_RIGHT",
        .active_low     = 1,
        .wakeup         = 0,
    },
    {//ENTERN5
        .gpio           = S3C64XX_GPN(4),
        .code           = KEY_KPENTER,
        .desc           = "KEY_ENTER",
        .active_low     = 1,
        .wakeup         = 0,
    },
    {//EXIT 6
        .gpio           = S3C64XX_GPN(5),
        .code           = KEY_ESC,//#define KEY_ESC 1
        .desc           = "KEY_ESC",
        .active_low     = 1,
        .wakeup         = 0,
    },
};
```

b）设备注册（增加）

这样针对 buttons 的设备结构体定义完成，接下来要将其添加到平台中，即注册。打开文件 arch/arm/mach-s3c64xx/mach-tq6410.c ， 里面有一个 tq6410_devices 数组，这个设备数组在平台初始化的时候通过 platform_add_devices(tq6410_devices, ARRAY_SIZE(tq6410_devices)); 来将里面的设备完成了注册。所以要将刚刚完成的 buttons 设备加入到该数组中（tq6410_devices），如下是 tq6410_devices 数组定义蓝色字体所示。

```
//通过设置 CONFIG_KEYBOARD_TQ6410 开关打开按键设备
#ifdef   CONFIG_KEYBOARD_TQ6410
    &s3c_device_gpio_button,//for 6 buttons
#endif;
```

2.4.2 按键驱动（platform driver）注册

buttons 驱动是 drivers/input/keyboard/tq6410_buttons.c 文件，对于平台设备驱动的注册，当其通过和预先注册的设备，通过名字查找匹配后，调用 probe 接口函数来完成注册过程。驱动要注册，buttons 生成一个 platform driver 驱动结构的实例 gpio_keys_device_driver，将关于 buttons 驱动的结构实例定义如下：

```
static struct platform_driver gpio_keys_device_driver = {
    //指定驱动初始化时调用的接口函数
    .probe          = gpio_keys_probe,
    //指定驱动卸载是调用的接口
    .remove         = __devexit_p(gpio_keys_remove),
    //设置驱动的属性
    .driver         = {
        .name   = "tq6410-keys",//设置其名称，必须和 platform device 的一致
        .owner  = THIS_MODULE, //所属关系
        #ifdef CONFIG_PM//设置电源管理时调用的电源管理接口函数
            .pm = &gpio_keys_pm_ops,
        #endif
    }
};
```

设备驱动注册的主要框架还要提供 module_init 和 module_exit 接口。所以以下的函数框架是注册设备驱动必须的。这些要点在 2.1 章节有提过。

```
//驱动初始化时第一时间调用的，在调用之后才去找和 driver 匹配
//的 device，如果找到匹配的才调用 probe 指定的接口函数进行初始化。
static int __init gpio_keys_init(void)
{
    _init_leds();//驱动程序中使用到的 LED 对应 GPIO 端口的配置
    return platform_driver_register(&gpio_keys_device_driver);  //注册 platform driver
}

static void __exit gpio_keys_exit(void)
{
    platform_driver_unregister(&gpio_keys_device_driver);//卸载驱动，在 remove 接口被处理完毕
}
//通过以下两个函数来告知内核该驱动的初始化和卸载的接口
module_init(gpio_keys_init); //注册驱动接口设置
module_exit(gpio_keys_exit);  //卸载驱动接口设置
```

到此，整个驱动的框架已经搭建完成，接下来需要完成的是 platform driver 实例中指定，也就是 driver 实例中的 probe，remove，pm 的指定的接口函数。概要它们的接口如下：

```
.probe          = gpio_keys_probe,
.remove         = __devexit_p(gpio_keys_remove),
.pm = &gpio_keys_pm_ops,
```

所以主要任务是完成 gpio_keys_probe，gpio_keys_remove 以及 gpio_keys_pm_ops 的实现。首先要实现的是 gpio_keys_probe 的定义如下：

对于 probe 函数的参数，在上面的 struct platform_driver 介绍时了解它需要一个 platform_device 指针。现在针对该函数的定义进行解析如下：

```
static int __devinit gpio_keys_probe(struct platform_device *pdev)
{
    //下面是获取之前提到的 platform device 时注册的 device 数据 gpio_button_data
    //下面截图显示之前注册平台设备的时候定义的按键数据结构
```

struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;

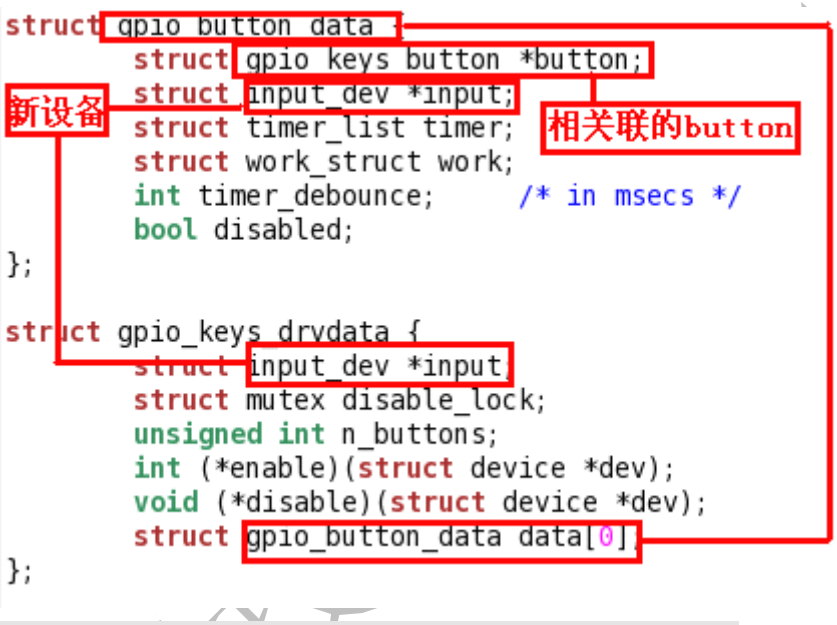
```
static struct gpio_keys_button gpio_buttons[] = {
        {//UP 1
                .gpio           = S3C64XX_GPN(0),
                .code           = KEY_UP,//#define KEY_UP  103
                .desc           = "KEY_UP",
                .active_low     = 1,
                .wakeup         = 0,
        },
        {//DOWN 2
                .gpio           = S3C64XX_GPN(1),
                .code           = KEY_DOWN,//#define KEY_DOWN
                .desc           = "KEY_DOWN",
                .active_low     = 1,
                .wakeup         = 0,
        },
        {//LEFT 3
                .gpio           = S3C64XX_GPN(2),
                .code           = KEY_LEFT,//#define KEY_LEFT 10
                .desc           = "KEY_LEFT",
                .active_low     = 1,
                .wakeup         = 0,
        },
        {//RIGHT 4
                .gpio           = S3C64XX_GPN(3),
                .code           = KEY_RIGHT,//#define KEY_RIGHT
                .desc           = "KEY_RIGHT",
                .active_low     = 1,
                .wakeup         = 0,
        },
        {//OPEN 5
                .gpio           = S3C64XX_GPN(4),
                .code           = KEY_ENTER,// #define KEY_ENTER
                .desc           = "KEY_ENTER",
                .active_low     = 1,
                .wakeup         = 0,
        },
        {//EXIT 6
                .gpio           = S3C64XX_GPN(5),
                .code           = KEY_ESC,//#define KEY_ESC 1
                .desc           = "KEY_ESC",
                .active_low     = 1,
                .wakeup         = 0,
        },
};
static struct gpio_keys_platform_data gpio_button_data = {
        .buttons        = gpio_buttons,
        .nbuttons       = ARRAY_SIZE(gpio_buttons),
};

struct platform_device s3c_device_gpio_button = {
        .name           = "tq6410-keys",
        .id             = -1,
        .num_resources  = 0,
        .dev            = {
                .platform_data  = &gpio_button_data,
        }
};
```

KEY1 KEY2 KEY3 KEY4 KEY5 KEY6

与driver中的name参数一致

struct gpio_keys_drvdata *ddata;

```
//下面是获取之前提到的 platform device 时注册的 device 数据 dev
struct device *dev = &pdev->dev;
//用于定义一个输入设备类型的结构体,该结构体在 include/linux/input.h 文件中定义
struct input_dev *input;
int i, error;
//是否作为唤醒源的标志
int wakeup = 0;
//申请内存空间，并将地址赋给 ddata,需要它来存放数据,关于结构体
//gpio_keys_drvdata 和 gpio_button_data 在本文件中定义，下面会介绍到
ddata =kzalloc(sizeof(struct gpio_keys_drvdata)+
    pdata->nbuttons *sizeof(struct gpio_button_data),GFP_KERNEL);//如下图显示它们的结构体
```




```
//这个之前提到过，是为新的输入设备分配空间的，这是输入设备注册开
//始时必须要完成的任务之一
input = input_allocate_device();
//判断以上两种内存分配是否成功,失败则返回错误码
if (!ddata || !input)
{
    dev_err(dev, "failed to allocate state\n");
    error = -ENOMEM;
    goto fail1;
}
//给刚刚生成的 ddata 对象初始化，利用它来传递一些数据 ddata->input = input;
//得到分配的输入设备
ddata->n_buttons = pdata->nbuttons;//
ddata->enable = pdata->enable;
ddata->disable = pdata->disable;
//初始化互斥锁
mutex_init(&ddata->disable_lock);
//设置平台驱动的数据，这里也比较重要，只有设置了平台数据，
```

```
    //在后面的其他函数中才能方便的共享平台数据(有利于数据资源之间的共享)
    //该函数的实现非常简单，实际的操作为：pdev->dev.driver_data = ddata，
    //device 结构的 driver_data 域指向驱动程序的私有数据空间。
    platform_set_drvdata(pdev, ddata);
    //设置 input 的 private 为 ddata 物理设备
    input_set_drvdata(input, ddata);
    //初始化新的输入设备驱动信息，包括名称，总线类型，开发商，版本信等
    //它们将来有与该设备驱动的 handler 的匹配，这个可参考 1.5.2 章节
    input->name = pdev->name;
    input->phys = "tq6410-keys/input0";
    input->dev.parent = &pdev->dev;
    //输入设备的打开关闭映射
    input->open = gpio_keys_open;
    input->close = gpio_keys_close;

    input->id.bustype = BUS_HOST;
    input->id.vendor = 0x0001;
    input->id.product = 0x0001;
    input->id.version = 0x0100;

    /* Enable auto repeat feature of Linux input subsystem */
    //指定事件类型，在 1.5.2 章节中有介绍
    if (pdata->rep)
        __set_bit(EV_REP, input->evbit);

//继续初始化 ddata，pdata 就是之前定义的 device 中的
//设备数据 其中主要调用 gpio_keys_setup_key 函数来实现，下面会介绍到
for (i = 0; i < pdata->nbuttons; i++) {
    //获得设备的 gpio_keys_button 数据
    struct gpio_keys_button *button = &pdata->buttons[i];
    struct gpio_button_data *bdata = &ddata->data[i];
    unsigned int type = button->type ?: EV_KEY;

    bdata->input = input;
    bdata->button = button;
    error = gpio_keys_setup_key(pdev, bdata, button);
    if (error)
        goto fail2;

    if (button->wakeup)
        wakeup = 1;
        //设置此输入设备可告知的事情
        input_set_capability(input, type, button->code);
    }
```

```
        //
    error = sysfs_create_group(&pdev->dev.kobj, &gpio_keys_attr_group);
    if (error)
    {
        dev_err(dev, "Unable to export keys/switches, error: %d\n",error);
        goto fail2;
    }
    //在初始化相关结构体，变量等之后，新的设备驱动进入真正的注册函数
    error = input_register_device(input);
    //如果出错，进入出错处理
    if (error)
    {
        dev_err(dev, "Unable to register input device, error: %d\n",error);
        goto fail3;
    }
    //设置了 GPIO 按键中断发生时的事件报告
    /* get current state of buttons */
    for (i = 0; i < pdata->nbuttons; i++)
        gpio_keys_report_event(&ddata->data[i]);
    //下面是 1 个同步事件，暗示和前面报告的消息属于 1 个消息组
    input_sync(input);
    //置中断源的唤醒状态
    ice_init_wakeup(&pdev->dev, wakeup);//
    return 0;
    //出错处理，释放内存、取消定时器异步工作和中断资源等
     fail3:
        sysfs_remove_group(&pdev->dev.kobj, &gpio_keys_attr_group);
     fail2:
        while (--i >= 0)
         {
            free_irq(gpio_to_irq(pdata->buttons[i].gpio), &ddata->data[i]);
            if (ddata->data[i].timer_debounce)
            del_timer_sync(&ddata->data[i].timer);
            cancel_work_sync(&ddata->data[i].work);
            gpio_free(pdata->buttons[i].gpio);
        }

    platform_set_drvdata(pdev, NULL);
     fail1:
        input_free_device(input);  // 释放分配的设备
        kfree(ddata);// 释放分配的内存

        return error;
}
```

通过 probe 接口函数，已经完成了注册输入设备驱动的基本要求 1,2 条，如下图

1、驱动中要有一个由输入设备触发的处理函数

这个处理过程主要是向子系统汇报事件，向上层交代处理结果。

2、一个初始化函数

初始化包括注册触发时的处理接口，创建新设备，设置事件类型，设置事件码。

在 probe 函数中，初始化 input 新设备时，有“input->open = gpio_keys_open;input->close = gpio_keys_close;”函数映射。gpio_keys_open 和 gpio_keys_close 在本文件中定义，如下代码所示：

```
static int gpio_keys_open(struct input_dev *input)
{
    //得到输入设备的信息
    struct gpio_keys_drvdata *ddata = input_get_drvdata(input);
    //打开设备使能
    return ddata->enable ? ddata->enable(input->dev.parent) : 0;
}

static void gpio_keys_close(struct input_dev *input)
{
    //得到输入设备的信息
    struct gpio_keys_drvdata *ddata = input_get_drvdata(input);
    //关闭设备使能
    if (ddata->disable)
        ddata->disable(input->dev.parent);
}
```

在上文中提到关于 gpio_keys_drvdata 和 gpio_button_data 的结构体定义如下所示：

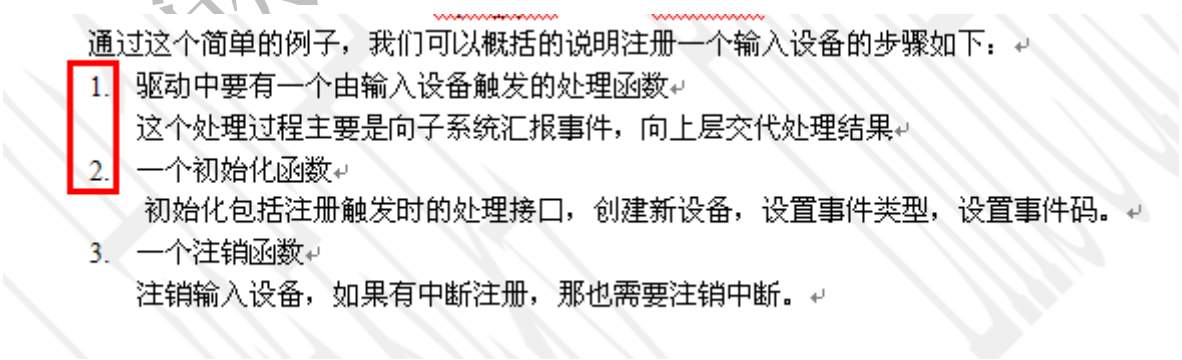
```
struct gpio_button_data
{
    struct gpio_keys_button *button;// gpio_keys_button 结构体用于存储按
    //相关信息
    struct input_dev *input;//输入设备
    struct timer_list timer;
    struct work_struct work;
    int timer_debounce;      /* in msecs */
        bool disabled;
};

struct gpio_keys_drvdata
{
    struct input_dev *input;
    struct mutex disable_lock;
    unsigned int n_buttons;
    int (*enable)(struct device *dev);
    void (*disable)(struct device *dev);
    struct gpio_button_data data[0];
};
```

在初始化 device 设备是用到的函数 gpio_keys_setup_key(pdev, bdata, button)是完成了申请了此 GPIO 按键设备需要的中断号，初始化 timer 及事件触发设置，下面对其源码进行分析：

```
static int __devinit gpio_keys_setup_key(struct platform_device *pdev,
    struct gpio_button_data *bdata,struct gpio_keys_button *button)
{
    //再从 button 成员中获取 desc 属性，这个在 platfomr device 中定义了
    const char *desc = button->desc ? button->desc : "gpio_keys";
    struct device *dev = &pdev->dev;
    unsigned long irqflags;
    int irq, error;

    //初始化 timer ，gpio_keys_timer 在本文件中定义主要是将工作添
    //keventd_wq 队列中
    setup_timer(&bdata->timer, gpio_keys_timer, (unsigned long)bdata);
    //动态创建一个按键报告工作
    INIT_WORK(&bdata->work, gpio_keys_work_func);
    //获取 GPIO 的控制权，如果出错，打印出错信息，并进行出错处理
    error = gpio_request(button->gpio, desc);
    if (error < 0)
    {
        dev_err(dev, "failed to request GPIO %d, error %d\n",
        button->gpio, error);
        goto fail2;
    }
    //设置 GPIO 为输入方向
    error = gpio_direction_input(button->gpio);
    if (error < 0)
    {
        dev_err(dev, "failed to configure"
        " direction for GPIO %d, error %d\n",button->gpio, error);
        goto fail3;
    }
    //设置按键的去抖动时间
    if (button->debounce_interval)
    {
        error = gpio_set_debounce(button->gpio,
        button->debounce_interval * 1000);
        /* use timer if gpiolib doesn't provide debounce */
        if (error < 0)
            bdata->timer_debounce = button->debounce_interval;
    }
    //设置按键的 GPIO 端口为 IRQ 中断
    irq = gpio_to_irq(button->gpio);
    if (irq < 0)
```

```
    {
        error = irq;
        dev_err(dev, "Unable to get irq number for GPIO %d, error %d\n",
        button->gpio, error);
        goto fail3;
    }
    //按键的中断的触发电平标志
    irqflags = IRQF_TRIGGER_RISING | IRQF_TRIGGER_FALLING;

    //设置按键中断标志为多个中断处理程序共享形式
    if (!button->can_disable)
        irqflags |= IRQF_SHARED;
    //申请中断资源，中断处理函数为 gpio_keys_isr，如果失败进入错误处理，关于//request_irq 函数参数
    //的说明请参考 1.2 中红外申请中断资源的说明
    error = request_irq(irq, gpio_keys_isr, irqflags, desc, bdata);
    if (error)
    {
        dev_err(dev, "Unable to claim irq %d; error %d\n",
        irq, error);
        goto fail3;
    }
    return 0;
    fail3:
        gpio_free(button->gpio);
    fail2:
        return error;
}
```

至此，在 1.4 章节中说明的注册输入设备驱动的几个基本条件的 1，2（下图），也还没有真正得到完成，因为上面的函数 gpio_keys_setup_key 中,在注册中断时又有了一个中断处理函数 gpio_keys_isr，一环套一环，还是得进一步完成中断处理函数。


通过这个简单的例子，我们可以概括的说明注册一个输入设备的步骤如下：

1. 驱动中要有一个由输入设备触发的处理函数
   这个处理过程主要是向子系统汇报事件，向上层交代处理结果
2. 一个初始化函数
   初始化包括注册触发时的处理接口，创建新设备，设置事件类型，设置事件码。
3. 一个注销函数
   注销输入设备，如果有中断注册，那也需要注销中断。

gpio_keys_isr 函数代码如下：

```
static irqreturn_t gpio_keys_isr(int irq, void *dev_id)
{
    //dev_id 是在执行 request_irq(irq, gpio_keys_isr, irqflags, desc, bdata);
```

```
//时传递的第 5 个参量
struct gpio_button_data *bdata = dev_id;
struct gpio_keys_button *button = bdata->button;
```

```
struct gpio_button_data {
        struct gpio_keys_button *button;
        struct input_dev *input;
        struct timer_list timer;
        struct work_struct work;
        int timer_debounce;     /* in msecs */
        bool disabled;
};

struct gpio_keys_drvdata {
        struct input_dev *input;
        struct mutex disable_lock;
        unsigned int n_buttons;
        int (*enable)(struct device *dev);
        void (*disable)(struct device *dev);
        struct gpio_button_data data[0];
};
```

新设备

相关联的button

```
#ifdef CONFIG_TQ6410_DEBUG_BUTTONS
    printk(" %s \n",__func__);
#endif

BUG_ON(irq != gpio_to_irq(button->gpio));

//是否到达去抖动的时间，如果是修改定时器定时时间
if (bdata->timer_debounce)
    mod_timer(&bdata->timer,jiffies + msecs_to_jiffies(bdata->timer_debounce));
else
//否则加入工作队列中。
schedule_work(&bdata->work);

    return IRQ_HANDLED;
}
```

在 gpio_keys_setup_key 函数中用 setup_timer(&bdata->timer, gpio_keys_timer, (unsigned long)bdata)函数初始化 timer ， gpio_keys_timer 在本文件中定义主要是将工作添加到 keventd_wq 队列中。其中的 gpio_keys_timer 函数在本文件定义如下：

```
static void gpio_keys_timer(unsigned long _data)
{
    struct gpio_button_data *data = (struct gpio_button_data *)_data;
    //将工作添加到 keventd_wq 队列中
    schedule_work(&data->work);
}
```

利用 INIT_WORK(&bdata->work, gpio_keys_work_func)函数动态创建一个按键报告工作，调用函数为 gpio_keys_work_func，此函数的定义在本文件中，代码如下：

```
static void gpio_keys_work_func(struct work_struct *work)
{
    //获取指向 gpio_button_data 数据结构的指针
    struct gpio_button_data *bdata =
    container_of(work, struct gpio_button_data, work);
    //报告事件
    gpio_keys_report_event(bdata);
}
```

gpio_keys_work_func 函数主要调用 gpio_keys_report_event 函数向 input 子系统报告事件，他在本文件中定义，下面来看看 gpio_keys_report_event 代码的实现。

```
static void gpio_keys_report_event(struct gpio_button_data *bdata)
{
    //获取平台设备的 gpio_keys_button 数据结构信息
    struct gpio_keys_button *button = bdata->button;
    //获取输入设备信息的数据结构
    struct input_dev *input = bdata->input;
    //获取按键的电平
    unsigned int type = button->type ?: EV_KEY;
    //获取端口电平是否要反转的标志
    int state = (gpio_get_value(button->gpio) ? 1 : 0) ^ button->active_low;
    //根据 code 的值设置 led 的状态，这个是 gpio_keys_button 的结构体，它的信息
    //在 arch/arm/mach-s3c64xx/dev-keypad.c 定义
    toggle_led(button->code, state);
    #ifdef CONFIG_TQ6410_DEBUG_BUTTONS
        printk(" %s :key status is=> %d   key value is=> %d ",button->desc,state,button->code);
    #endif
    //发送报告信息给 input 子系统，进行相关的处理
    input_event(input, type, button->code, !!state);
    //等待发送报告信息完成
    input_sync(input);
}
```

当注销驱动时调用 gpio_keys_remove 函数，其代码解析如下所示：

```
static int __devexit gpio_keys_remove(struct platform_device *pdev)
{
    struct gpio_keys_platform_data *pdata = pdev->dev.platform_data;
    struct gpio_keys_drvdata *ddata = platform_get_drvdata(pdev);
    struct input_dev *input = ddata->input;
    int i;

    sysfs_remove_group(&pdev->dev.kobj, &gpio_keys_attr_group);
    //初始化设备，不支持电源管理的，device_init_wakeup 定义在 pm_wakeup.h 中
    device_init_wakeup(&pdev->dev, 0);
    //释放申请的中断和内存资源等
    for (i = 0; i < pdata->nbuttons; i++)
```

```
    {
        int irq = gpio_to_irq(pdata->buttons[i].gpio);
        //释放中断资源
        free_irq(irq, &ddata->data[i]);
            if (ddata->data[i].timer_debounce)
                del_timer_sync(&ddata->data[i].timer);
                cancel_work_sync(&ddata->data[i].work);
                //释放内存资源
                gpio_free(pdata->buttons[i].gpio);
    }
        //注销输入设备
        input_unregister_device(input);
        return 0;
}
```

至此整个按键驱动的简析完成

2.5 字符设备驱动注册的简介

Linux 内核中针对于字符设备的注册细节过程，这里不做介绍，仅仅需要弄清楚，注册一个字符设备的干要，也就是其驱动框架。注册字符设备时主要利用的注册函数是

```
int misc_register(struct miscdevice * misc);
```

卸载字符设备的函数为

```
int misc_deregister(struct miscdevice *misc);
```

它们均需要一个 miscdevice 结构体实例，所以在驱动中，必须定义一个 miscdevice 实例来完成注册，miscdevice 结构体定义于 include/linux/miscdevice.h 中：

```
struct miscdevice   {
    int minor;   //设备的次设备号
    const char *name;//设备名称
    const struct file_operations *fops;//设备的操作接口集
    struct list_head list; //misc_list 的成员
    struct device *parent;//父设备
    struct device *this_device;//指向当前设备节点
    const char *nodename;
    mode_t mode;
};
```

struct miscdevice 结构中， 主要关心的是 minor（次设备号），name（设备名称），fops（操作设备的接口集,read，write 等），其它的极少运用得到。综合以上，注册一个字符驱动要完成的任务是：

1、定义一个 miscdevice 结构体实例，让注册函数 int misc_register(struct miscdevice * misc) 调用来完成注册或者卸载设备函数 int misc_deregister(struct miscdevice *misc) 调用来完成设备卸载工作

2、通过 module_init 来指定驱动的初始化第一访问接口。通过 module_exit 来指定卸载设备的最终处理接口，所以只要完成以上工作，就可以完成一个字符设备的驱动。

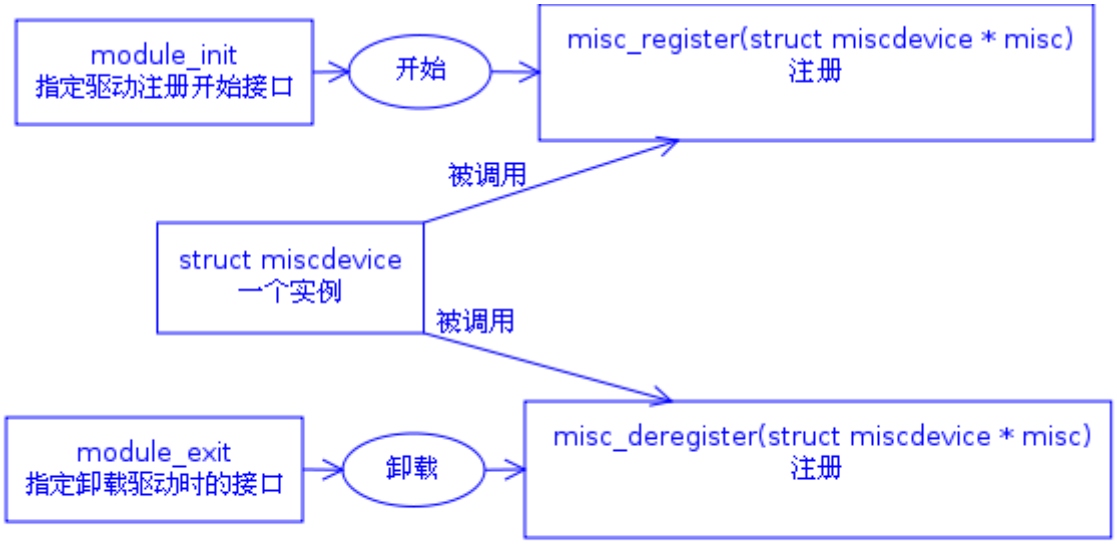


图 2-25

2.6 ADC 字符驱动

2.6.1 简析内核中的针对 ARM 芯片的 ADC 驱动接口

在 3.0 版本的内核中，针对于 ARM 芯片的一些功能代码已经进一步整理，划分。其中 ADC 的驱动也集中到了一起，弄成了一个通用的接口，只要去了解怎么去运用这些接口就可以。不过由于主要目的是了解，学习 ADC 驱动，那么就分析一下内核中已经提供了的 ADC 通用驱动，因为只有里面才是真正涉及了关于芯片中的 ADC 各个寄存器的配置。

内核中的 ADC 驱动是文件 arch/arm/plat-samsung/adc.c 。在解析过程中，又涉及到了 platform device 以及 platform driver 之间的知识要点，这样也有利于读者巩固 platform driver 注册的过程。先看看关于 ADC 驱动的 platform device 的定义，它位于 arch/arm/plat-samsung/dev-adc.c 中。截图如下：

```
static struct resource s3c_adc_resource[] = {
        [0] = {
                .start = SAMSUNG_PA_ADC,
                .end   = SAMSUNG_PA_ADC + SZ_256 - 1,
                .flags = IORESOURCE_MEM,
        },
        [1] = {
                .start = IRQ_TC,
                .end   = IRQ_TC,
                .flags = IORESOURCE_IRQ,
        },
        [2] = {
                .start = IRQ_ADC,
                .end   = IRQ_ADC,
                .flags = IORESOURCE_IRQ,
        },
};

struct platform_device s3c_device_adc = {
        .name           = "samsung-adc",
        .id             = -1,
        .num_resources  = ARRAY_SIZE(s3c_adc_resource),
        .resource       = s3c_adc_resource,
};
```

内存资源设置，驱动中可以通过 platform_get_resource 获取

中断源设置，驱动中可以通过 platform_get_irq 获取

名称要和driver中的一样

资源数组

图 2-26

设备定义好后，将其（s3c_device_adc）增加到 arch/arm/mach-s3c64xx/mach-tq6410.c 中的 tq6410_devices[]数组中，让系统为其注册，这些步骤在前面的章节中都有提到。接下来看它的驱动 arch/arm/plat-samsung/adc.c，platform driver 的注册要点，在第一章以及本章的前面两节的驱动解析中都重复的提到，先创建一个驱动（platform driver）实例，下面就是其实例：

```
static struct platform_device_id s3c_adc_driver_ids[] = {
        {
                .name           = "s3c24xx-adc",//用来检索device的.name
                .driver_data    = TYPE_S3C24XX,
        }, {
                .name           = "s3c64xx-adc",//用来检索device的.name
                .driver_data    = TYPE_S3C64XX,
        },
        { }
};
MODULE_DEVICE_TABLE(platform, s3c_adc_driver_ids);

static struct platform_driver s3c_adc_driver = {
        //id_table不为空，将来在其中的每组 .name 来检索已经注册的device
        .id_table       = s3c_adc_driver_ids,
        .driver         = {
                .name   = "s3c-adc",//因为提供了id_table,这里的.name不在做为检索device的判断依据
                .owner  = THIS_MODULE,
        },
        //设置的一些接口，这些接口通通需要逐个定义
        .probe          = s3c_adc_probe,
        .remove         = __devexit_p(s3c_adc_remove),
        .suspend        = s3c_adc_suspend,
        .resume         = s3c_adc_resume,
};
```

从上图的 s3c_adc_drivers_ids 数组中，它们的.name (s3c24xx-adc 和 s3c6xx-adc)没有一个和前面一张截图的.name(samsung-adc)相同的（也就是和 device 的 name 不同），这里需要处理一下。所以在文件中的

arch/arm/mach-s3c64xx/s3c6410.c 中的 void __init s3c6410_map_io(void) 函数中重新设置一下 device 的名称，仅仅需要加下面一句代码就可以：

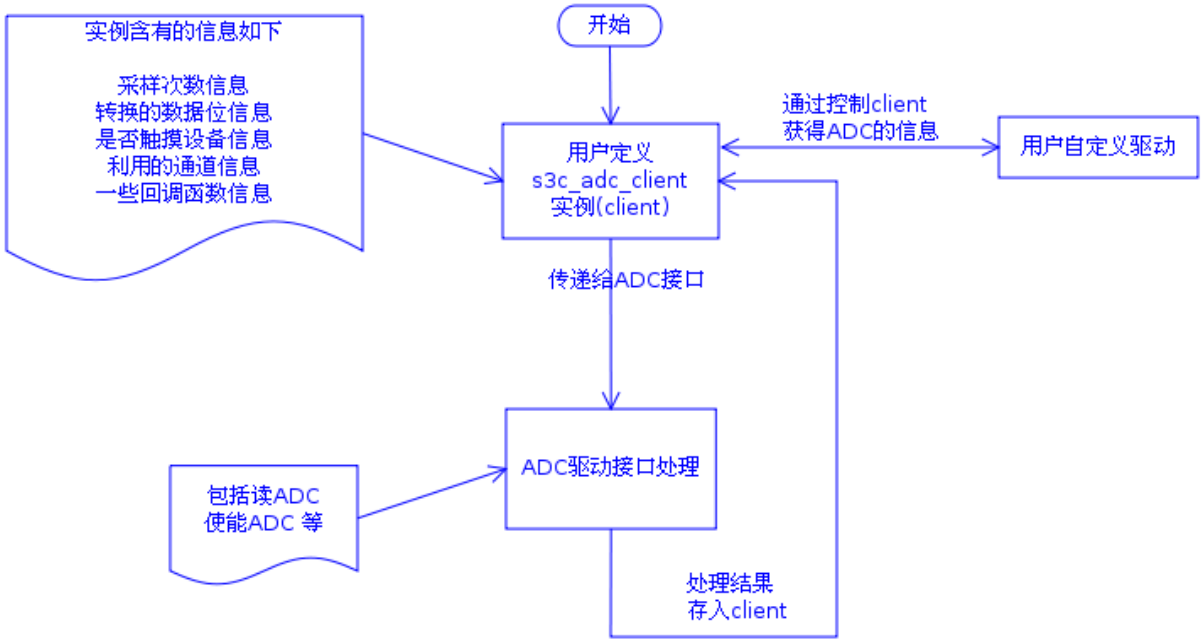
```
s3c_adc_setname("s3c64xx-adc");
```

平台设备驱动的实例创建完成，arch/arm/mach-s3c64xx/s3c6410.c 中利用了利用 arch_initcall(adc_init); 来告诉内核，该设备驱动的初始化第一入口函数为 adc_init，在之前的 platform driver 例子中，都是通过 module_init 函数来告诉内核该驱动的初始化最初接口函数，而这里利用了 arch_initcall 来实现，这是因为这里的 ADC 驱动是最为一个通用的驱动接口，所以它必须优先完成注册，所以就要利用优先级高于 module_init 的 arch_initcall 函数来处理。

至此针对 platfomr device 以及 platform driver 的搭建框架已经完成，对于 platform driver 中的接口函数去逐个定义就可以。为了避免冗余，而且主要目的是了解和利用该 ADC 驱动的接口，所以下面主要了解一下该驱动对外提供的接口函数的作用，以及说明一下 ADC（针对 TQ6410）一些寄存器的配置。

1)、通用 ADC 接口函数

内核中提供的 ADC 驱动，它的访问方式是，用户定义一个外来访问对象（结构体 s3c_adc_client 的实例），将其为主要对象，传递给 ADC 中提供的一些接口来处理，用户需要的信息，包括数据，数据位，端口号等都可以从 s3c_adc_client 实例中获得，下面会说明 s3c_adc_client 结构体，这里以图示意内核中的 ADC 驱动和自定义的 ADC 字符驱动的工作流程图：



下面来了解一下 ADC 驱动的几个重要接口函数的功能。

1. s3c_adc_register 函数

该函数可以注册一个 ADC 的 client，ADC 驱动既然作为一个通用的接口，所以它采取 client 的方式来处理外来对象的任务。也就是要想调用它提供的接口（例如 s3c_adc_read），那么必须生成一个 client 对象传递给它，才可以处理。s3c_adc_register 函数需要传递的参数有 3 个：

i) struct platform_device *pdev, //平台设备对象

ii) /选择通道时的调用函数接口
void (*select)(struct s3c_adc_client *client, unsigned int selected),
iii) //转换完成时回调的函数接口
void (*conv)(struct s3c_adc_client *client, unsigned d0, unsigned d1,unsigned *samples_left),
iv) //是否是触摸设备，因为 ADC 驱动也做了触摸的处理，
//所以需要它来判断针对触摸屏做一些特殊处理
unsigned int is_ts

所以要利用 ADC 驱动接口，必须定义个 client 来配合访问它们。至于 client 是自己定义还是通过 s3c_adc_register 来完成，随编写者喜欢。下面是 s3c_adc_client 结构体的截图说明：

```
struct s3c_adc_client {
        struct platform_device  *pdev;//生成adc client的平台设备
        struct list_head         pend;//用于管理ADC驱动中的多个client
        wait_queue_head_t       *wait;//多个client时的队列等待

        unsigned int             nr_samples;//采样次数
        int                      result;//转换完成的结果
#ifdef CONFIG_MACH_TQ6410
        int                     data_bit; //用于转换的数据位设置
#endif
        unsigned char            is_ts;//指定client是否触摸设备
        unsigned char            channel;//client所利用的通道
        //通道选择的回调函数
        void    (*select_cb)(struct s3c_adc_client *c, unsigned selected);
        //转换结束的回调函数
        void    (*convert_cb)(struct s3c_adc_client *c,
                              unsigned val1, unsigned val2,
                              unsigned *samples_left);
};
```

2. s3c_adc_release(struct s3c_adc_client *client)函数
关联第一个函数，这个函数是用来释放 client 的，这里，需要传递的就是 client 对象。
3. s3c_adc_start(struct s3c_adc_client *client, unsigned int channel, unsigned int nr_samples)函数
开始执行 ADC 转换，参数中意义，第一个参数 client 是必须的，这个可以从图 2-20 中有了解到，第二个是通道，第三个是指定采样次数。
4. s3c_adc_read(struct s3c_adc_client *client, unsigned int ch)函数
读取转换的结果，第二个参数是读取的通道，该函数先调用了 s3c_adc_start 后，再去读取数据寄存

2)、ADC 相关寄存器的设置

对于一般的 ADC 转换（没有涉及触摸屏处理），最主要是 ADCCON 寄存器的设置。针对于 ADCCON 的设置大致过程如下：

1. 设置分频值，也就是 ADCCON 寄存器的 6—13 位的数值：

PRSCVL	[13:6]	A/D converter prescaler value Data value: 5 ~ 255 **NOTE:** Note that division factor is (N+1) when the prescaler value is N. ADC frequency should be set less than PCLK by 5 times. (Ex. If PCLK=10MHz, ADC Frequency<2MHz) This A/D converter is designed to operate at maximum 5MHz clock

这里涉及到转换的频率，转换频率的换算方法如下：

39.4.1 A/D CONVERSION TIME

When the PCLK frequency is 50MHz and the prescaler value is 49, total 10-bit or 12-bit conversion time is as follows.

A/D converter freq. = 50MHz/(49+1) = 1MHz

Conversion time = 1/(1MHz / (5cycles)) = 1/200KHz =5 us

Note:
This A/D converter was designed to operate at maximum 5MHz clock, so the conversion rate can go up to 1MSPS.

也就是转换频率是 PCLK/（6—13 位寄存器的数值+1）的结果，而且这个结果最大是 5MHz，TQ6410 中的 Linux 系统的 PCLK 是 66MHz，所以这里的分频数值可以设置为 49.

设置好分频数值后，可以时能分频计数器了，也就是设置 ADCCON 的位 14 为 1.如下图：

PRSCEN	[14]	A/D converter prescaler enable 0 = Disable 1 = Enable

2. 设置数据位

RESSEL	[16]	A/D converter resolution selection 0 = 10-bit A/D conversion 1 = 12-bit A/D conversion

TQ6410 的数据位有 12 位和 10 位,ADCCON 的位 16 为 1 时是表示 12 位数据转换，否则是 10 位数据转换。

3. 设置通道

SEL_MUX	[5:3]	Analog input channel select 000 = AIN 0 001 = AIN 1 010 = AIN 2 011 = AIN 3 100 = YM 101 = YP 110 = XM 111 = XP

通道的设置由 ADCCON 的位 3—5 的数值确定，由上图可以看出通道设置的选择以及数值的设置对应关系。

4. 设置为正常的工作模式

STDBM	[2]	Standby mode select 0 = Normal operation mode 1 = Standby mode

在一般情况下，该位为 1，如果在进入休眠的时候，该为必须设置为 1，表示等待状态。

5. 使能中断位，开始转换

READ_ START	[1]	A/D conversion start by read 0 = Disable start by read operation 1 = Enable start by read operation

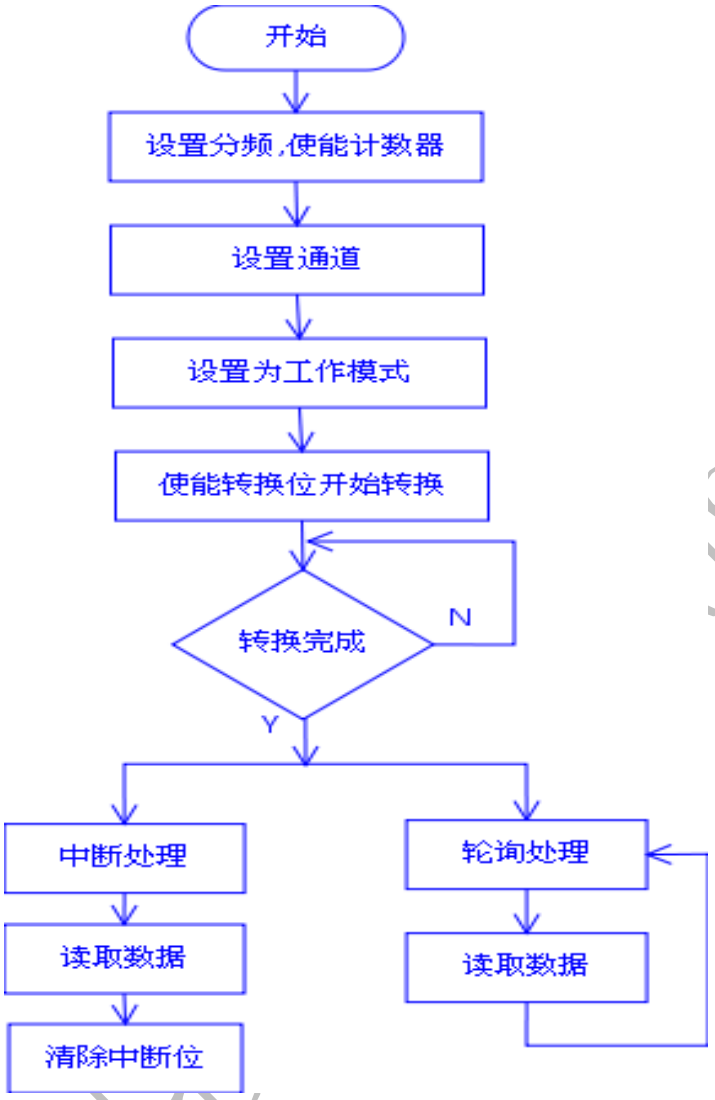
设置好基本的功能以后，将 ADCCON 的位 1 设置为 1，使能转换，那么 ADC 工作开始。

6. 读取数据

ADC 有自己的数据寄存器，当转换完成以后，直接去读取它的数据寄存器就可以获得了数据。这里涉及了它转换完成的判断，一种是以中断的方式，当转换完成，进入中断程序在读取数据。另一种是以轮询的方式，不停的去检测 ADCCON 的位 15 来确定是否已经完成了转换

ECFLG	[15]	End of conversion flag(Read only) 0 = A/D conversion in process 1 = End of A/D conversion

整个的流程如下图所示：



3)、小结

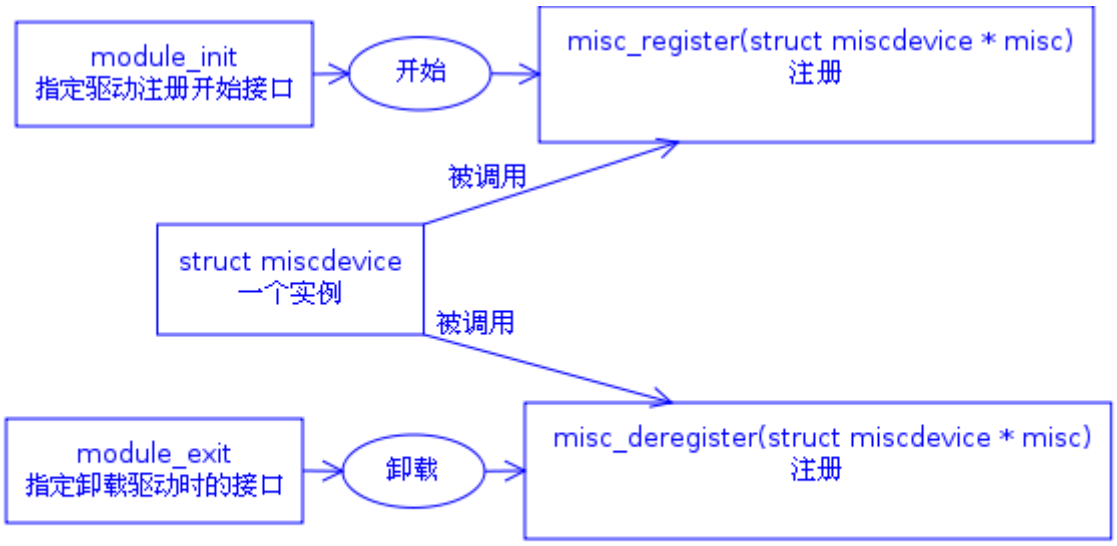
本节主要介绍 Linux 内核中的 ADC 驱动接口函数的功能，以及 ADC 寄存器的功能设置，在此过程中也重复的讲述一下 platform driver 以及 platform device 之间的关系，进一步弄清 platform driver 的注册流程。

2.6.2 简析注册 ADC 字符驱动

1)、ADC 字符驱动简析

在 2.4 章节中，了解了字符驱动注册的流程，2.5.1 章节中了解了内核中提供的关于 ADC 的几个接口函数的功能，所以在这一节中，要注册一个 ADC 的字符设备，目的就是利用前面两节的知识点来完成任务。TQ6410 的 ADC 字符驱动代码位于文件 drivers/char/tq6410_adc.c 中。 下面根据注册字符设备的框架

来逐个完成代码：



从框架图中，先定义一个 miscdevice 的实例（在 2.4 节中，有说明了关于结构体 miscdevice），其实例如下截图：

```
static struct file_operations s3c_adc_fops =
{
        //操作接口所属关系
        .owner          = THIS_MODULE,
        .read           = tq6410_adc_read,//读接口
        .open           = tq6410_adc_open,//打开接口
        .release        = tq6410_adc_close,//关闭接口
        //ioctl接口
        .unlocked_ioctl = tq6410_adc_ioctl,
};
//miscdevice 实例
static struct miscdevice s3c_adc_miscdev =
{
        //次设备号，这里表示由系统分配
        .minor          = MISC_DYNAMIC_MINOR,
        //设备名，DEVICE_NAME是一个宏，它等于"tq6410-adc"
        .name           = DEVICE_NAME,
        //设备的操作接口(read,write,open等)
        .fops           = &s3c_adc_fops,
};
```

操作接口集

实例定义好了，接着为 module_init() 定义一个入口函数 tq6410_adc_init ；为 module_exit() 定义一个卸载入口函数 tq6410_adc_exit. 这样框架就搭建完成，接下来的任务就是逐个定义这些接口函数。下面就是 tq6410_adc_init 函数的定义：

```
int __init tq6410_adc_init(void)
{

    int ret=0;
```

//设置关于 ADC 寄存器的 IO 地址，将其映射到内存初始化
//_adc 中的 base_addr
//这里的 0x20 是由具体的寄存器跨度多决定的
//_adc 是结构体的实例，为了直观，下面将插入图片说明
//_adc 以及 0x20
_adc.base_addr = ioremap(S3C64XX_PA_ADC, 0x20);
关于_adc 插图

```
struct tq6410adc {
        //需要一个client，因为我们需要内核中的ADC
        //驱动接口
        struct s3c_adc_client *client;
        //关于ADC的时钟
        struct clk      *adc_clock;
        //IO映射的基址
        void __iomem    *base_addr;
        //通道号
        int  adc_port;
        //读取到的数据
        int  adc_data;
};

static struct tq6410adc _adc;
```

关于 0x20 的获取的依据截图（0x7E00B000-0x7E00B020）：

Register	Address	R/W
ADCCON	0x7E00_B000	R/W
ADCTSC	0x7E00_B004	R/W
ADCDLY	0x7E00_B008	R/W
ADCDAT0	0x7E00_B00C	R
ADCDAT1	0x7E00_B010	R
ADCUPDN	0x7E00_B014	R/W
ADCCLRINT	0x7E00_B018	W
Reserved	0x7E00_B01C	-
ADCCLRINTPNDNUP	0x7E00_B020	W

00 到 20

```
if(_adc.base_addr ==    NULL){
        ret = -ENOENT;
        goto err_map;
}
```

//从时钟队列中获取 ADC 的时钟,初始化_adc 中的 adc_clock

```
    _adc.adc_clock = clk_get(NULL, "adc");
    if(IS_ERR(_adc.adc_clock)){
        printk("failed to fine ADC clock source: %s \n",KERN_ERR);
        goto err_clk;
    }
    //使能 ADC 的时钟
    clk_enable(_adc.adc_clock);
    //利用 misc_register 注册设备，这个在之前的步骤图中有说明
    ret = misc_register(&s3c_adc_miscdev);
    if (ret) {
        printk(DEVICE_NAME "can't register major number\n");
        goto err_clk;
    }
    //要为 client 申请空间，因为需要初始化 client 实例
    _adc.client = kzalloc(sizeof(struct s3c_adc_client), GFP_KERNEL);
    if (!_adc.client) {
        printk("no memory for adc client\n");
        goto err_clk;
    }
    //下面是 client 的初始化工作，is_ts=0 表示 client 不是触摸屏
    //所以在访问内核中 ADC 驱动的接口时，它不会以触摸屏
    //的对象处理，对于 s3c_adc_client 的简介在 2.5.1.1 章节中有
    //解图说明
    _adc.client->is_ts = 0;
    //client 访问 ADC 接口时的回调接口设置
    _adc.client->select_cb = tq6410_adc_select;
    _adc.client->convert_cb = tq6410_adc_conversion;
    //默认情况下，利用通道 0
    _adc.adc_port=0;
    /默认情况下，设置为 12 位的转换数据
    _adc.client->data_bit=12;
    printk(KERN_INFO "TQ6410 ADC driver successfully probed\n");
        return 0;
    //下面是注册过程的出错处理
    err_clk:
        clk_disable(_adc.adc_clock);
        clk_put(_adc.adc_clock);
    err_map:
        iounmap(_adc.base_addr);
        return ret;
    }
```

针对 module_init 需要的接口 tq6410_adc_init(void) 简析完成，下面看 module_exit 需要的接口 tq6410_adc_exit(void)的定义，其实卸载过程往往是注册，初始化的反过程，其代码如下:

```
static void __exit tq6410_adc_exit(void)
{
    //释放在注册中利用 iomap 申请的内存映射空间
    iounmap(_adc.base_addr);
    if (_adc.adc_clock)
    {
        //禁止了 ADC 的时钟
        clk_disable(_adc.adc_clock);
        clk_put(_adc.adc_clock);
        _adc.adc_clock = NULL;
    }
    //卸载之前利用 misc_register 注册的字符设备
    misc_deregister(&s3c_adc_miscdev);
}
```

从 miscdevice 定义开始，到 module_init 和 module_exit 的接口定义，整个框架已经完成，也做了简析。不过在定义 miscdevice 的时候，有定义了关于设备的操作接口集合，也就是常常用到的 open，read，write，ioctrl，close。所以还是要去完成这些接口函数的定义。要了解这些接口函数的返回值，需要的参数，可以从 include/linux/fs.h 中的 file_operations 结构体获得。由于 ADC 字符驱动中，open，close 函数没有做什么处理，所以这里不再去解析它们，看看 read 接口 tq6410_adc_read：

```
static ssize_t   tq6410_adc_read(struct file *file, char __user * buffer, size_t size, loff_t * pos)
{
    //这里直接访问内核中的 ADC 驱动接口 s3c_adc_read 来直接去读取 ADC
    //在这看到了定义 client 以及 adc_port 的作用了。
    _adc.adc_data=s3c_adc_read(_adc.client, _adc.adc_port);
    //将读取到的数据，拷贝到用户空间
    //这里的 buffer 是用户在调用 read 函数时，传递进来变量地址，数据就会拷贝
    //到该变量中去，用户就可以在应用层运用
    if(copy_to_user(buffer, (char *)&_adc.adc_data, sizeof(_adc.adc_data)))
    {
        printk("copy data to user space failed !\n");
        return -EFAULT;
    }
        //拷贝成功就返回拷贝的字节数
        return sizeof(_adc.adc_data);
}
```

再看看另一个重要的接口函数 unlocked_ioctl 接口 tq6410_adc_ioctl，它的主要功能是用来设置 ADC 通道和数据位的，下面是其具体代码 ：

```
static long tq6410_adc_ioctl(struct file *file,unsigned int cmd, unsigned long arg)
{
    unsigned int temp= (unsigned int)arg; //获得传输进来的参数值
    switch (cmd)//检索用户的命令
    {
        //如果是选择通道命令, PORT_SELECTED 是定义的常量，数值为 1
        case PORT_SELECTED:
```

```
                if (temp >= 4)//只允许用户利用通道 0 到通道 4
                {
                        printk(" %d is already reserved for TouchScreen\n", _adc.adc_port);
                }else{//设置通道，在 read 函数中，是直接利用_adc.adc_port 的
                        _adc.adc_port = temp;
                }
                return 0;
        //如果是数据位设置命令，BIT_SELECTED 是定义的常量，其数值为 3
        case BIT_SELECTED:
                if(temp ==12 || temp ==10)//TQ6410 的 ADC 仅仅支持 10 位和 12 位的数据
                {
                        _adc.client->data_bit=temp;//设置好数据位
                }else{
                        printk("nice guy,data bits should be 12 or 10 !\n");
                }
                return 0;
        default://其他命令当作出错处理
                printk("unknowed cmd :%d!\n", cmd);
                return -ENOIOCTLCMD;
    }
}
```

2)、小结

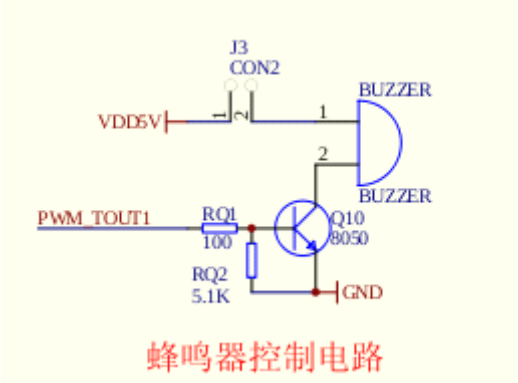
通过剖析 ADC 的字符驱动，学习实际利用 misc_register 来注册一个字符设备的过程，同时了解了 ADC 相关的一些寄存器的设置，了解内核中 ADC 驱动所提供的可用接口以及如何定义一个 sc_adc_client 实例来访问 ADC 驱动的接口。

2.7 蜂鸣器驱动

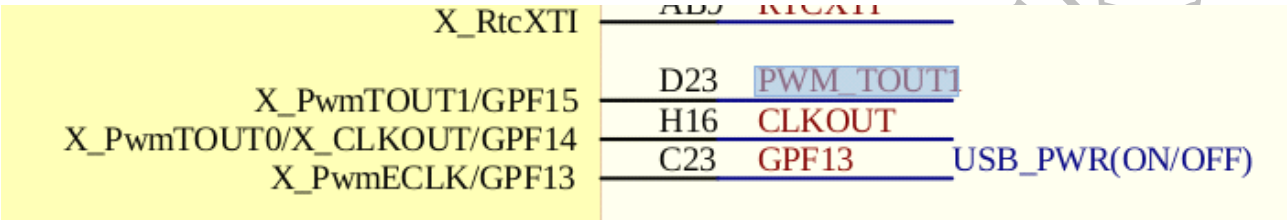
对于蜂鸣器的驱动，主要学习的知识点是继续温习字符设备的注册流程以及 TQ6410（CPU 为 S3C6410，以下皆表示同一意思）的定时器相关寄存器的设置。针对于字符设备的注册流程和基本框架，在 2.4 章节中有了说明，并且在 2.5 章节中也实际的操作了一次，这里不在赘述字符设备的注册流程。将重点转向 TQ6410 中的定时器 0 的相关寄存器的设置解析。

2.7.1 定时器 1 的寄存器简介

TQ6410 中的蜂鸣器利用了定时器 1，其电路路如下：



通过网络名称 PWM_TOUT1 可以从内核的原理图中找到是利用 GPIO 的 GPF15 引脚：



所以重点说明一下关于定时器 1 的 PWM 功能配置步骤以及相关的寄存器数值的设置。对于设置定时器 1 的相关寄存器如下：

Register	Offset	R/W	Description
TCFG0	0x7F006000	R/W	Timer Configuration Register 0 that configures the two 8-bit Prescaler and DeadZone Length
TCFG1	0x7F006004	R/W	Timer Configuration Register 1 that controls 5 MUX and DMA Mode Select Bit
TCON	0x7F006008	R/W	Timer Control Register
TCNTB1	0x7F006018	R/W	Timer 1 Count Buffer Register
TCMPB1	0x7F00601c	R/W	Timer 1 Compare Buffer Register

其实和定时器 1 相关的寄存器还有中断控制的寄存器 TINT_CSTAT，不过蜂鸣器配置的 PWM 工作模式没有涉及中断处理，所以这里不对该寄存器做说明。下面按照定时器 1 设置的先后来对以上截图中的寄存器逐个说明。PWM 的输出，有一个频率值，这个频率的计算方法如下：

```
Timer input clock Frequency = PCLK / ( {prescaler value + 1} ) / {divider value}
{prescaler value} = 1~255
{divider value} = 1, 2, 4, 8, 16, TCLK
```

从以上的计算公式中，需要设置好 prescaler value 的数值以及 divider value 的数值，而这两个数值分别在寄存器 TCFG0 和 TCFG1 中。所以需要设置这两个寄存器。

设置 prescaler value

TCFG0	Bit	R/W	Description
Reserved	[31:24]	R	Reserved Bits
Dead zone length	[23:16]	R/W	Dead zone length
Prescaler 1	[15:8]	R/W	Prescaler 1 value for Timer 2, 3 and 4
Prescaler 0	[7:0]	R/W	Prescaler 0 value for timer 0 & 1

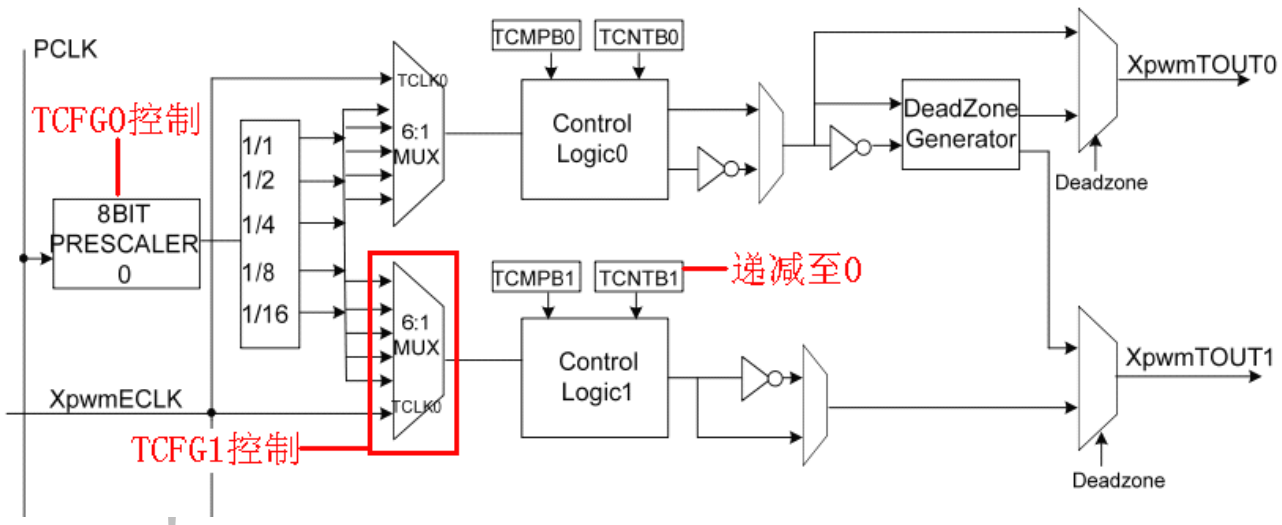
从寄存器的说明中，只要设置 TCFG0 的 位 0—7 的数值那就是设置了 prescaler value。而它的数值是被定时器 0 以及定时器 1 共用的。

设置 divider value

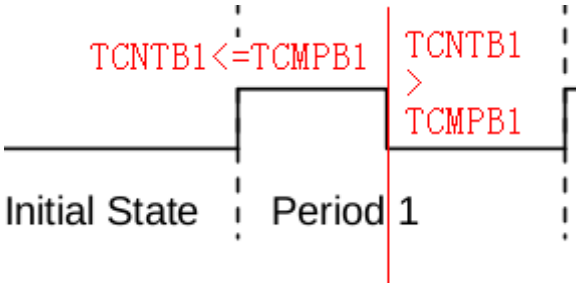
TCFG1	Bit	R/W	Description
Divider MUX1	[7:4]	R/W	Select Mux input for PWM Timer 1
			0000:1/1 0001:1/2 0010:1/4 0011:1/8 0100: 1/16 0101: External TCLK0 0110: External TCLK0 0111: External TCLK0

从芯片手册的截图，需要设置 TCFG1 的位 4—7 位的数值就是设置了 divider value。而且里面不同的数值对应着相应的数值一目了然。这就是 TCFG0 和 TCFG1 的作用。它们决定了计数频率的大小。

寄存器 TCNTB1 和 TCMPB1 的作用，笼统的说 TCNTB1 决定了 PWM 的输出频率，TCMPB1 决定着占空比的大小（整个周期的输出，有高电平阶段和低电平阶段，占空比就是高/低电平的比例）。至于高电平，低电平的输出，TCON 控制寄存器可以控制的。下面是手册中定时器工作的整体图：



从整体图中结合之前说明的寄存器，概括性的说明一下它们之间的作用，由 TCFG0 设置 prescaler 的数值，由 TCFG1 设置了 divider 的数值，设置好后产生了一个输入的 frequency（工作频率），TCNTB1 的数值被加载来递减（也就在 frequency 下倒计时），当 TCNTB1 中的数值倒计到和 TCMPB1 中的数值相等以后，它输出的电平将出现反相（高或者低）。如下图示例：



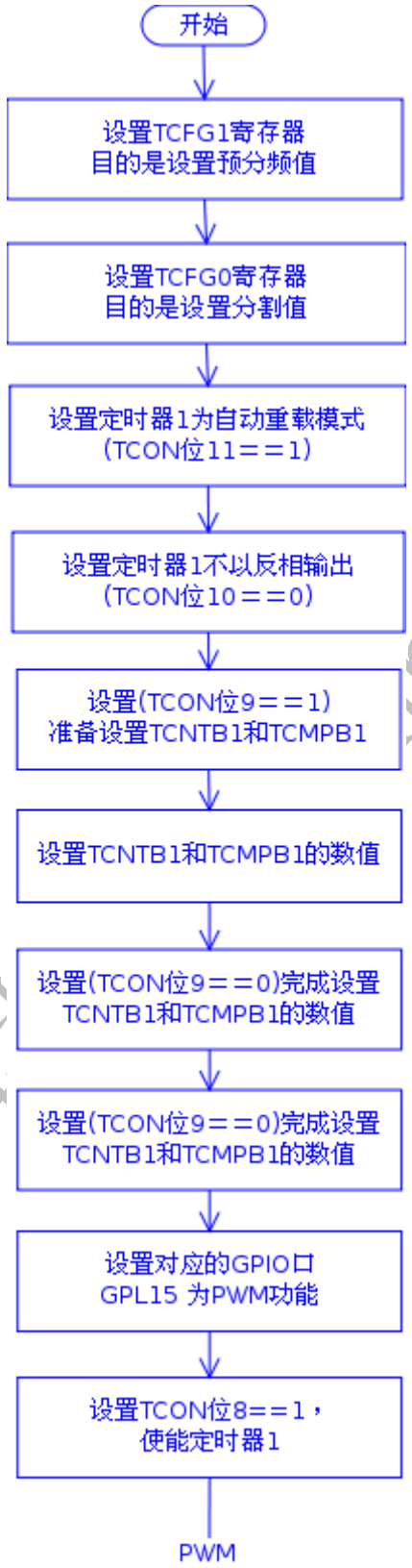
了解了 TCFG0，TCFG1,TCNTB1 以及 TCMPB1 的功能，在来了解另一个重要的寄存器 TCON 的作用。其实 TCON 是整体的一个控制器，它可以对定时器 0—4 进行几种功能的控制，设置。针对于定时器 1 的截图如下：

TCON	Bit	R/W	Description
Timer 1 Auto Reload on/off	[11]	R/W	0: One-Shot 1: Interval Mode(Auto-Reload)
Timer 1 Output Inverter on/off	[10]	R/W	0: Inverter Off 1: TOUT1 Inverter-On
Timer 1 Manual Update	[9]	R/W	0: No Operation 1: Update TCNTB1,TCMPB1
Timer 1 Start/Stop	[8]	R/W	0: Stop 1: Start Timer 1

从截图中可以看出 TCON 寄存器的功能是：

1. 设置定时器 1 在 TCNTB1 递减到 0 时，是否需要重头再来（也就是不停的重复）。位 11 为 1 时就是不停的重复
2. 设置定时器 1 的输出是否反相（高变成低，低变成高）,位 10 为 1 时，输出反相。
3. 设置是否更新 TCNTB1,TCMPB1。位 9 为 1 时，就是要更新。当要设置 TCNTB1，TCMPB1 的数值时，必须设置它为 1，等设置完成在设置其位为 0。
4. 设置是否开启定时器 1。位 8 为 1 时，就是开启。当设置好一切数值以后，需要开启定时器，那就将改位置 1 来使能定时器 1 输出 PWM。

综合所有寄存器的功能了解，将定时器的 PWM 工作模式的流程画图如下：



2.7.2 蜂鸣器驱动简析

结合 2.6.1 章节的说明，来简单说明一下蜂鸣器驱动的编写过程。由于蜂鸣器驱动也是字符驱动，所以它的框架也是以字符设备驱动来定制的。字符驱动的框架在 2.5 章节中有了详细的说明。先来看看它的框架代码：

```
//字符设备节点操作的端口定义
static struct file_operations s3c6410_beep_fops=
{
        .owner          = THIS_MODULE,
        .unlocked_ioctl     = tq6410_beep_ioctl,//ioctl 接口函数
        .open           = tq6410_beep_open,//open 接口函数
        .release    = tq6410_adc_close,//close 接口函数
};
//定义一个 miscdevice  实例
static struct miscdevice tq6410_beep_miscdev =
{
    .minor          = MISC_DYNAMIC_MINOR,
    .name           = DEVICE_NAME,
    .fops       = &s3c6410_beep_fops,
};
//定义 module_init  使用的入口
static int tq6410_beep_init(void)
{
    int ret;
    //利用 misc_register 来注册设备
    ret = misc_register(&tq6410_beep_miscdev);
    if(ret<0)
    {
        printk(DEVICE_NAME "can't register major number\n");
        return ret;
    }
    //初始化一下定时器 1 的一些功能
    timer1_init();//截图说明该函数如下
```

```
static void timer1_init(void)
{
        unsigned long tcon;
        unsigned long tcfg1;
        unsigned long tcfg0;

        tcon = __raw_readl(S3C2410_TCON);//读取TCON寄存器
        tcfg1 =__raw_readl(S3C2410_TCFG1);//读取TCFG1寄存器
        tcfg0 =__raw_readl(S3C2410_TCFG0);//读取TCFG0寄存器

        //设置GPG15 为 PWM 功能,配置某个GPIO 可以用 s3c_gpio_cfgpin函数
        s3c_gpio_cfgpin(S3C64XX_GPF(15),S3C64XX_GPF15_PWM_TOUT1);

        //设置divider，先清0 TCFG1的位4－－7
        //S3C2410_TCFG1_MUX1_MASK 定义在 arch/arm/plat-samsung/plat/regs-timer.h
        tcfg1 &= ~S3C2410_TCFG1_MUX1_MASK;
        //设置它的数值为2，
        //S3C2410_TCFG1_MUX1_DIV2 arch/arm/plat-samsung/plat/regs-timer.h
        tcfg1 |= S3C2410_TCFG1_MUX1_DIV2;

        //设置prescaler,清0 TCFG0 位 0--7
        tcfg0 &= ~S3C2410_TCFG_PRESCALER0_MASK;
        //设置prescaler为49，PRESCALER==49,
        //S3C6410_TCFG_PRESCALER0_SHIFT定义在 arch/arm/plat-samsung/plat/regs-timer.h
        tcfg0 |= (PRESCALER) << S3C6410_TCFG_PRESCALER0_SHIFT;
        //将设置结果写回TCFG0，TCFG1
        __raw_writel(tcfg1, S3C2410_TCFG1);
        __raw_writel(tcfg0, S3C2410_TCFG0);
        //设置定时器1为auto-reload模式
        tcon &= ~(7<<8);//先清0 TCON 位8－－10，我们需要重新设置
        tcon |= S3C2410_TCON_T1RELOAD;//设置为auto-reload模式
        __raw_writel(tcon, S3C2410_TCON);//将设置数值写进寄存器
}
```

```
    printk(KERN_INFO "TQ6410 Beep driver successfully probed\n");
    return 0;
}
//定义 module_exit 使用的入口函数
static void tq6410_beep_exit(void)
{
    //利用 misc_deregister 来卸载设备
    misc_deregister(&tq6410_beep_miscdev);
    printk("Goodbye EmbedSky-beep module !\n");
}
module_init(tq6410_beep_init);//设置入口函数
module_exit(tq6410_beep_exit);//设置入口函数
```

这就是说的字符设备的框架搭建，接下来就逐个完成了其操作接口的各个函数的定义就可以。由于 close，open 函数没有做任何操作，所以这里不再解说它们，重点说明一下 ioctl 函数：

```
static long tq6410_beep_ioctl(struct file *file,unsigned int CMD_ON_OFF, unsigned long Val)
{
    if(CMD_ON_OFF<=0)//如果不是打开蜂鸣器，则关闭
    {
        s3c6410_beep_off();//截图该函数说明如下：
```

```
static void s3c6410_beep_off(void)
{
        unsigned long tcon;
        //设置GPF15引脚为输入
        s3c_gpio_cfgpin(S3C64XX_GPF(15),S3C64XX_GPF15_INPUT);
        //读取TCON寄存器
        tcon = __raw_readl(S3C2410_TCON);
        //设置TCON位8==0，也就是停止定时器1
        tcon &= ~S3C2410_TCON_T1START;
        //设置TCON位9==0,也就是不对TCNTB1，TCMPB1作任何操作
        tcon &= ~S3C2410_TCON_T1MANUALUPD;
        //将设置数值写如寄存器生效
        __raw_writel(tcon, S3C2410_TCON);
}
```

}else{

//设置蜂鸣器的 TCMPB1，TCNTB1

s3c6410_set_timer1(Val);//截图说明该函数

```
static void s3c6410_set_timer1(unsigned long Val)
{
        unsigned long tcon;
        unsigned long tcnt;
        unsigned long tcmp;
        tcnt = 0xffffffff;
        //读取TCON寄存器
        tcon = __raw_readl(S3C2410_TCON);
        //设置TCON的位9==1,准备更新TCNTB1以及TCMPB1
        tcon |= S3C2410_TCON_T1MANUALUPD;
        __raw_writel(tcon, S3C2410_TCON);
        //设置TCNTB1的数值，它的数值一直没有变化
        //也就是定时器1的频率没有变化，我们只改变其脉宽
        tcnt = 101;
         __raw_writel(tcnt, S3C2410_TCNTB(1));//写入TCNB1寄存器
        //设置TCMPB1的数值
        tcmp = Val;
        __raw_writel(tcmp, S3C2410_TCMPB(1));//写入寄存器
}
```

//使能蜂鸣器

s3c6410_beep_start();//截图说明该函数如下

```
static void s3c6410_beep_start(void)
{
        unsigned long tcon;
        //读取TCON寄存器
        tcon = __raw_readl(S3C2410_TCON);
        //设置TCON位9==0，表示不在操作TCNTB1，TCMPB1
        tcon &= ~S3C2410_TCON_T1MANUALUPD;
        //设置TCON位8==1，表示使能定时器1，开始工作
        tcon |= S3C2410_TCON_T1START;
        __raw_writel(tcon, S3C2410_TCON);//将结果写入TCON,使其生效
        //设置GPG15 为PWM功能，蜂鸣器会发出声响
        s3c_gpio_cfgpin(S3C64XX_GPF(15),S3C64XX_GPF15_PWM_TOUT1);

}


    }
return 0;
}
```

2.7.3 小结

本节主要通过说明定时器 1 的相关寄存器的功能以及设置，同时说明了定时器的 PWM 动作模式的设置流程。结合了蜂鸣器驱动的实例。进一步体会字符设备注册的步骤以及加深理解定时器 PWM 工作模式的设置流程。

2.8 小结

这一章的内容，主要是结合第一章的子系统介绍知识点，针对 TQ6410 的按键驱动以及红外摄像头驱动，重点说明了 Linux 系统中的输入子系统的运用以及输入设备的注册流程，要点。在这一分析的过程中，同时多次提及 platform driver 和 platform deive 之间的关系，说明了平台驱动注册的步骤。

另外增加了字符设备驱动注册的框架要点，进而举例 ADC 和蜂鸣器驱动，说明了字符设备驱动的编写要点。从中也说明了内核总 ADC 驱动的一些接口的运用以及定时器的相关寄存器的功能设置。