

实验 25 中断之独立按键

25.1 本章导读

在学习了前面中断的知识，就可以尝试来写一个独立按键的驱动。本实验除了介绍 Datasheet 部分、中断函数以及中断的例子，还介绍了几个常用的调试方法，这些调试方法目前就现学现用，到了后面再给大家做一个总结。

25.1.1 工具

25.1.1.1 硬件工具

- 1) iTOP4412 开发板
- 2) U 盘或者 TF 卡
- 3) PC 机
- 4) 串口

25.1.1.2 软件工具

- 1) 虚拟机 Vmware
- 2) Ubuntu12.04.2
- 3) 超级终端 (串口助手)
- 4) 源码文件夹 "itop4412_irq"

25.1.2 预备课程

24_中断的基础知识

25.1.3 视频资源

本节配套视频为 “视频 25_中断之独立按键”

25.2 学习目标

本章需要学习以下内容：

中断的硬件知识

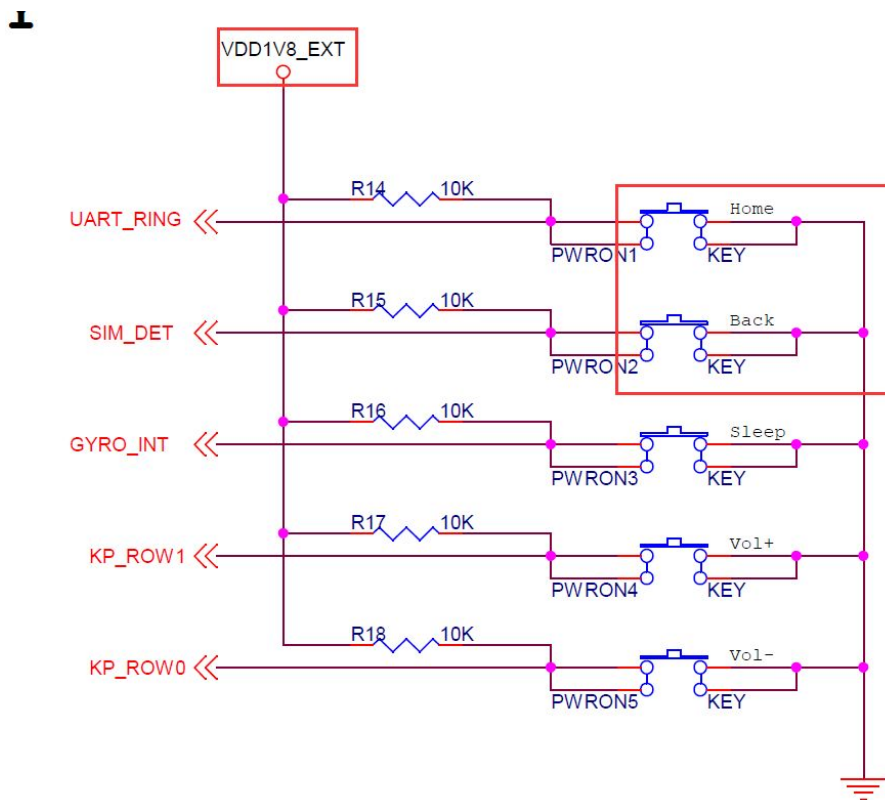
外部中断 datasheet 阅读

申请中断函数和中断处理函数

驱动程序的几个调试方法

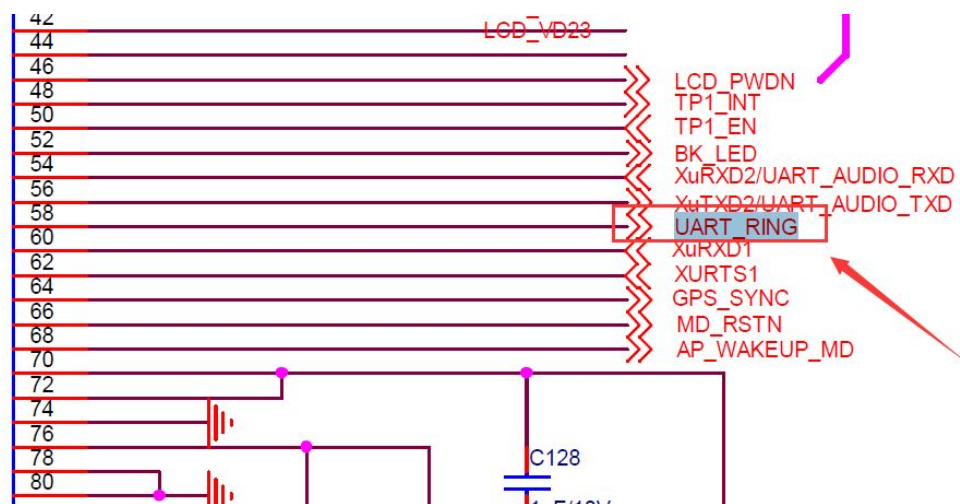
25.3 中断的硬件知识和外部中断 datasheet 阅读

底板上有 5 个在一起的独立按键，以 HOME 和 BACK 为例，先来看一下原理图，如下图所示。

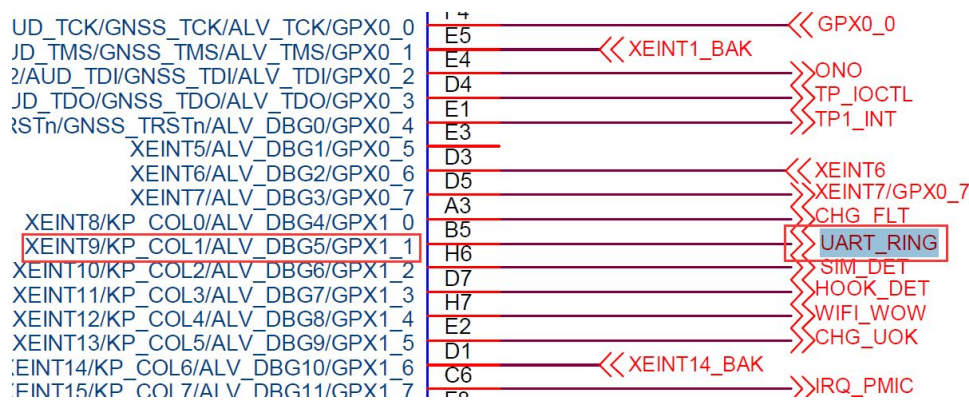


如上图所示，通过硬件连接可知，默认状态给中断的 IO 是高电平。所以可以判断后面对中断触发方式要为下降沿触发。

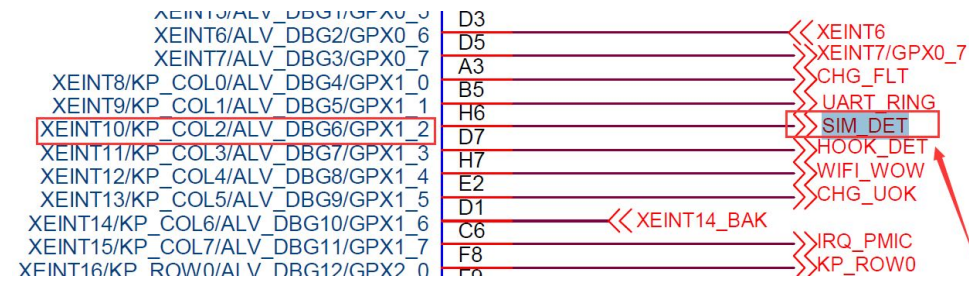
如下图所示，查找一下这个两个网络 UART_RING 和 SIM_DET。



然后打开核心板原理图文档，查找“UART_RING”网络，如下图所示，对应的 GPIO 为“EXYNOS4_GPX1(1)”，中断号为“XEINT9”。



另外一个网络,如下图所示,对应的 GPIO 为“EXYNOS4_GPX1(2)” 中断号为“XEINT10”。



然后打开 4412 的 Datasheet ,搜索“GPX1CON” ,如下图所示 ,下图以 SCP 的 Datasheet 为例，POP 的也是类似。

6.2.3.198 GPX1CON

- Base Address: 0x1100_0000
- Address = Base Address + 0x0C20, Reset Value = 0x0000_0000

Name	Bit	Type	Description	Reset Value
GPX1CON[7]	[31:28]	RW	0x0 = Input 0x1 = Output 0x2 = WAKEUP_INT1[7] 0x3 = KP_COL[7] 0x4 = Reserved 0x5 = ALV_DBG[11] 0x6 to 0xE = Reserved 0xF = EXT_INT41[7]	0x00
			0x0 = Input 0x1 = Output	

如下图所示，找到其中对应的“GPX1CON[1]”和“GPX1CON[2]”，如下图所示。

GPX1CON[2]	[11:8]	RW	0x0 = Input, 0x1 = Output, 0x2 = WAKEUP_INT1[2] 0x3 = KP_COL[2]	0x00
------------	--------	----	--	------

SAMSUNG ELECTRONICS

6-257



Preliminary product information describes products that are in development, for which full characterization data and associated errata are not yet available. Specifications and information herein are subject to change without notice.

Samsung Confidential

Exynos 4412_UM

6 General Purpose Input/Output (GPIO) Control

Name	Bit	Type	Description	Reset Value
			0x4 = Reserved 0x5 = ALV_DBG[6] 0x6 to 0xE = Reserved 0xF = EXT_INT41[2]	
GPX1CON[1]	[7:4]	RW	0x0 = Input 0x1 = Output 0x2 = WAKEUP_INT1[1], 0x3 = KP_COL[1], 0x4 = Reserved, 0x5 = ALV_DBG[5], 0x6 to 0xE = Reserved, 0xF = EXT_INT41[1]	0x00

如上图所示，可以看到将管脚配置为外部中断模式需要设置为 0xF。

最后再带大家看一个部分，在 Datasheet 的 56 章，找到 Debouncing Filter，这是 4412 对于按键自动防抖的部分，如下图所示，可以看到有自带的防抖。

56.2 Debouncing Filter

Supports debouncing filter for keypad interrupt of any key input. The filtering width is approximately 62.5 μs ("FCLK" two-clock, when the FCLK is 32 kHz). The keypad interrupt (key pressed or key released) to the CPU in software scan mode is an ANDed signal of the all row input lines after filtering.

Figure 56-2 illustrates the internal debouncing filter operation.

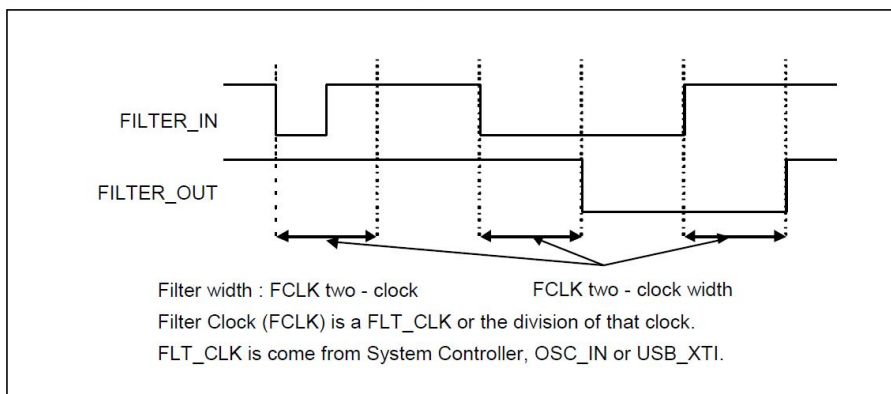


Figure 56-2 Internal Debouncing Filter Operation

25.4 中断相关函数简介

Linux 中的中断在使用前，都需要申请。中断申请函数是 “request_irq”，在头文件 “include/linux/interrupt.h” 中，如下图所示。

```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
} ____cacheline_internodealigned_in_smp;

extern irqreturn_t no_action(int cpl, void *dev_id);

#ifdef CONFIG_GENERIC_HARDIRQS
extern int __must_check
request_threaded_irq(unsigned int irq, irq_handler_t handler,
                    irq_handler_t thread_fn,
                    unsigned long flags, const char *name, void *dev);

static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}

extern int __must_check
request_any_context_irq(unsigned int irq, irq_handler_t handler,
                       unsigned long flags, const char *name, void *dev_id);

extern void exit_irq_thread(void);
#else
```


如上图所示，中断申请函数 `request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev)` 有下面几个参数。

参数 `unsigned int irq` : `irq` 是中断号

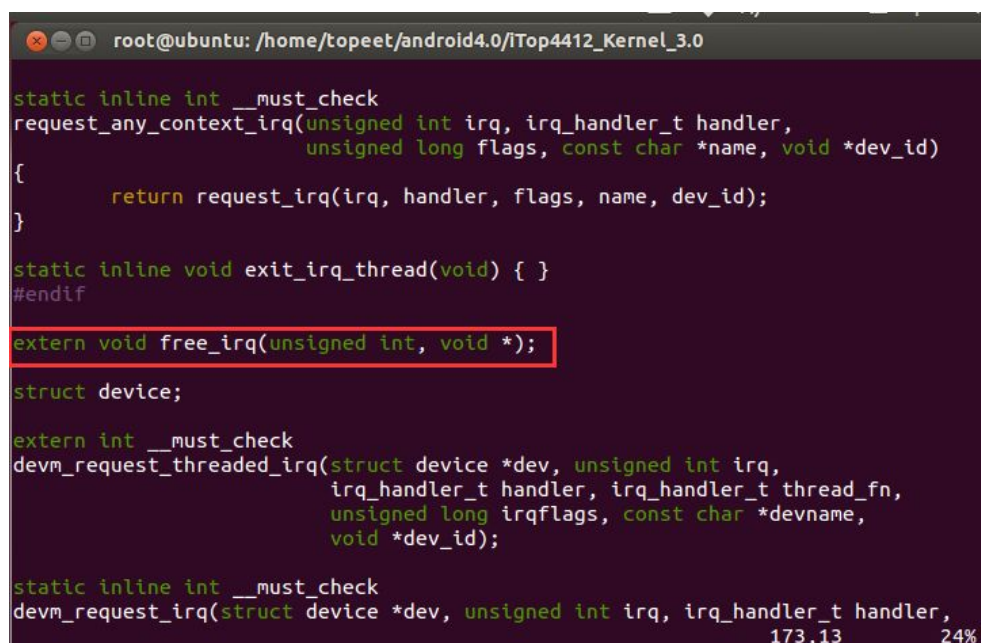
参数 `irq_handler_t handler` : `handler` 是向系统登记的处理函数

参数 `unsigned long flags` : `irqflags` 是触发标志位

参数 `const char *name` : `devname` 是中断名称，可以通过注册之后可以通过 “`cat /proc/interrupts`” 查看

参数 `void *dev` : `dev_id` 是设备

和上面中断申请函数对应的就是中断释放函数 `free_irq`，卸载驱动的时候需要调用，如下图所示，也是在头文件 “`include/linux/interrupt.h`” 中。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0

static inline int __must_check
request_any_context_irq(unsigned int irq, irq_handler_t handler,
                        unsigned long flags, const char *name, void *dev_id)
{
    return request_irq(irq, handler, flags, name, dev_id);
}

static inline void exit_irq_thread(void) { }
#endif

extern void free_irq(unsigned int, void *);

struct device;

extern int __must_check
devm_request_threaded_irq(struct device *dev, unsigned int irq,
                        irq_handler_t handler, irq_handler_t thread_fn,
                        unsigned long irqflags, const char *devname,
                        void *dev_id);

static inline int __must_check
devm_request_irq(struct device *dev, unsigned int irq, irq_handler_t handler,
```

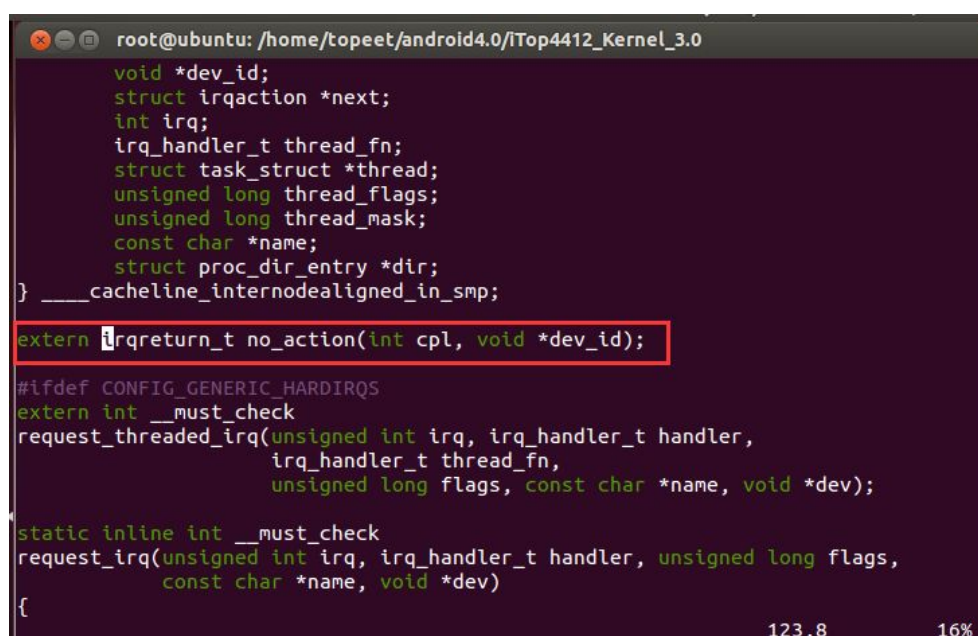
如上图所示中断释放函数 `extern void free_irq(unsigned int, void *);` 的参数如下。

参数 1 : irq 是中断号

参数 2 : dev_id 是设备

产生中断之后，会调用中断处理函数 irqreturn_t，这个函数也是在头文件

“include/linux/interrupt.h” 中如下图所示。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
void *dev_id;
struct irqaction *next;
int irq;
irq_handler_t thread_fn;
struct task_struct *thread;
unsigned long thread_flags;
unsigned long thread_mask;
const char *name;
struct proc_dir_entry *dir;
} ____cacheline_internodealigned_in_smp;

extern irqreturn_t no_action(int cpl, void *dev_id);

#ifdef CONFIG_GENERIC_HARDIRQS
extern int __must_check
request_threaded_irq(unsigned int irq, irq_handler_t handler,
                    irq_handler_t thread_fn,
                    unsigned long flags, const char *name, void *dev);

static inline int __must_check
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
            const char *name, void *dev)
{
    123,8      16%
```

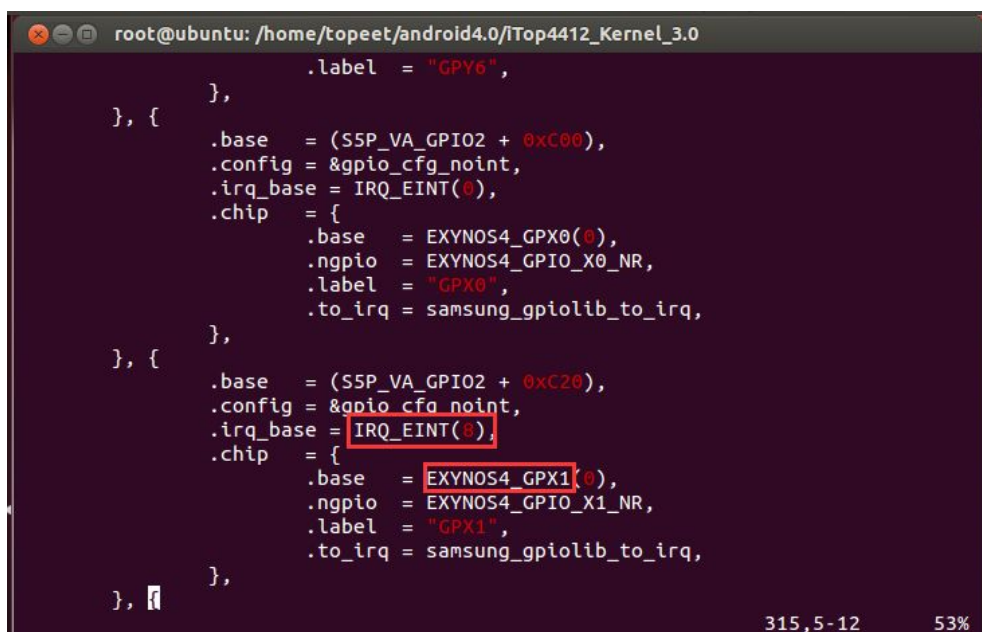
如上图所示，该函数为 extern irqreturn_t no_action(int cpl, void *dev_id);

中断函数类型为 irqreturn_t

参数 int cpl : 中断号

参数 void *dev_id : 设备

在初始化文件 “drivers/gpio/gpio-exynos4.c” 中，如下图所示，



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0

        .label = "GPX6",
    }, {
        .base = (SSP_VA_GPIO2 + 0xC00),
        .config = &gpio_cfg_noint,
        .irq_base = IRQ_EINT(8),
        .chip = {
            .base = EXYNOS4_GPX0(0),
            .ngpio = EXYNOS4_GPIO_X0_NR,
            .label = "GPX0",
            .to_irq = samsung_gpiolib_to_irq,
        },
    }, {
        .base = (SSP_VA_GPIO2 + 0xC20),
        .config = &gpio_cfg_noint,
        .irq_base = IRQ_EINT(8),
        .chip = {
            .base = EXYNOS4_GPX1(0),
            .ngpio = EXYNOS4_GPIO_X1_NR,
            .label = "GPX1",
            .to_irq = samsung_gpiolib_to_irq,
        },
    },
}, {
```

选取的两个中断管脚都属于“GPX1”，所以如上图所示，中断编号的基础数值是IRQ_EINT(8)，那么GPX1CON[0]和GPX1CON[1]则对应着中断号IRQ_EINT(9)和IRQ_EINT(10)。

如果想调用其他外部中断也可以通过这种方式来查阅中断号，不过前提是这个中断没有被占用。

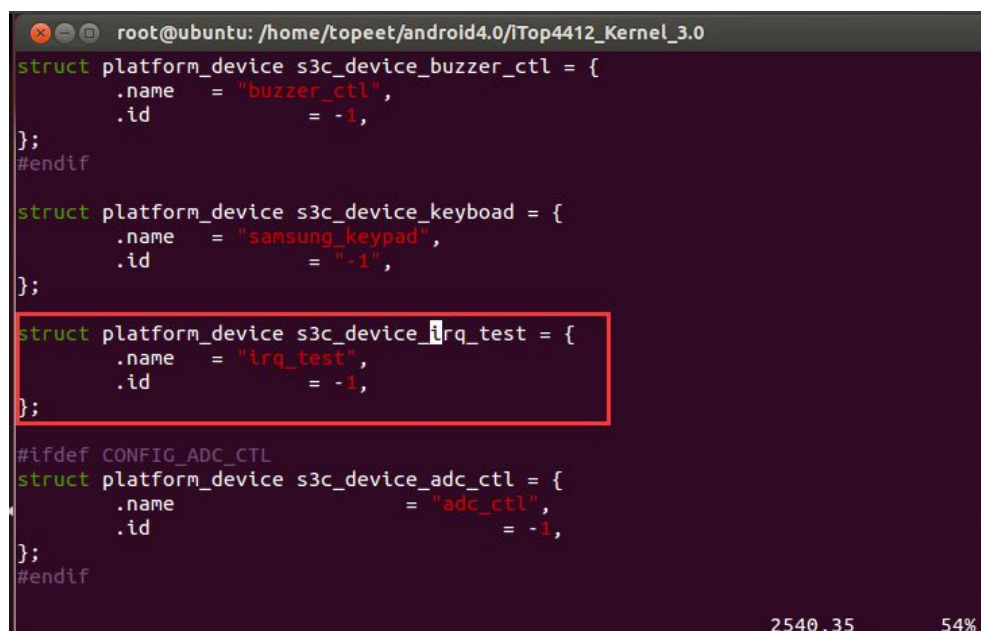
25.5 实验操作

外部中断的实际应用一般是集成在一些类似声卡、显卡、其他总线设备中，实际的应用也就是调用前面提到的头文件和函数，在驱动初始化的时候申请中断，然后针对具体的驱动写中断函数处理函数。例如声卡中调音量的按键，就是将按键部分的代码集成到声卡中，检测到按键就可以对应的调低调高音量。

这里注册一个简单的字符驱动，这样的话会便于大家的理解。要完成的功能就是按键中断产生之后打印数据。

如下图所示，首先注册设备，如下图所示，在平台文件

“arch/arm/mach-exynos/mach-itop4412.c” 中添加注册设备的代码。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
struct platform_device s3c_device_buzzer_ctl = {
    .name = "buzzer_ctl",
    .id = -1,
};
#endif

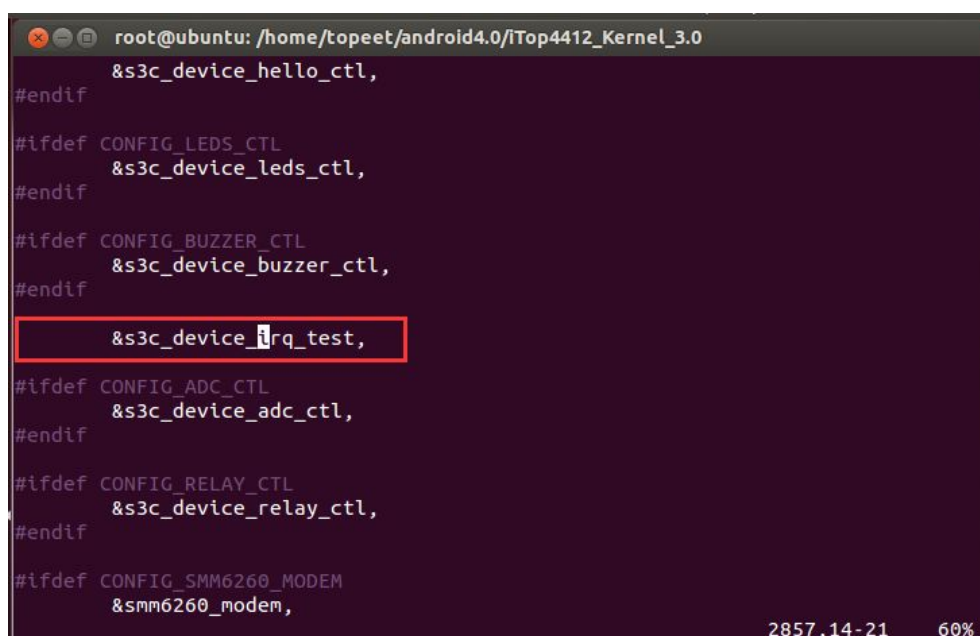
struct platform_device s3c_device_keyboard = {
    .name = "samsung_keypad",
    .id = -1,
};

struct platform_device s3c_device_irq_test = {
    .name = "irq_test",
    .id = -1,
};

#ifdef CONFIG_ADC_CTL
struct platform_device s3c_device_adc_ctl = {
    .name = "adc_ctl",
    .id = -1,
};
#endif
```

如上图所示，为了简单，可以也可以不定义宏变量，强制注册设备。

如下图所示，添加设备调用的代码。



```
root@ubuntu: /home/topeet/android4.0/iTop4412_Kernel_3.0
&s3c_device_hello_ctl,
#endif

#ifdef CONFIG_LEDS_CTL
&s3c_device_leds_ctl,
#endif

#ifdef CONFIG_BUZZER_CTL
&s3c_device_buzzer_ctl,
#endif

&s3c_device_irq_test,

#ifdef CONFIG_ADC_CTL
&s3c_device_adc_ctl,
#endif

#ifdef CONFIG_RELAY_CTL
&s3c_device_relay_ctl,
#endif

#ifdef CONFIG_SMM6260_MODEM
&smm6260_modem,

2857,14-21 60%
```

然后打开 menuconfig 配置文件，将使用这两个中断的驱动卸载掉。

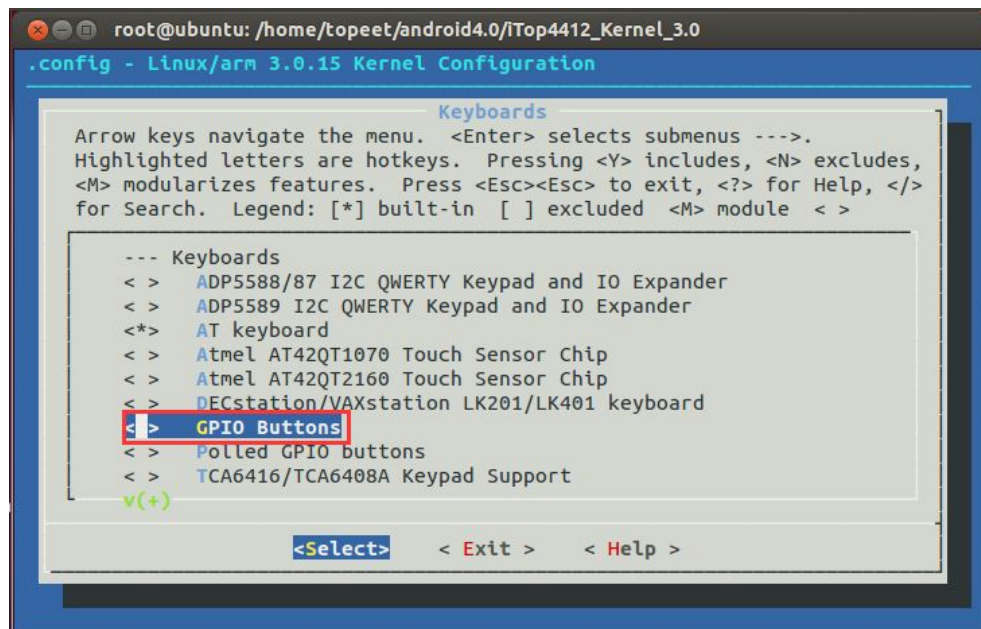
Device Drivers --->

Input device support --->

Keyboards --->

取消 GPIO Buttons --->

如下图所示。



重新设置之后，将内核重新编译，并将生成的二进制 zImage 文件烧写到开发板中替换原来的内核。

然后将驱动的基本框架完成。

如下图所示，头文件部分，以后写 4412 的驱动可以将这些头文件一股脑的添加到代码前面。

```
/*以后写驱动可以讲头文件一股脑的加载代码前面*/
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <mach/gpio.h>
#include <plat/gpio-cfg.h>
#include <linux/miscdevice.h>
#include <linux/platform_device.h>
#include <mach/regs-gpio.h>
#include <asm/io.h>
#include <linux/regulator/consumer.h>
#include <linux/delay.h>

/*中断函数头文件*/
#include <linux/interrupt.h>
#include <linux/irq.h>
```

然后是驱动模块的入口函数和出口函数，如下图所示。

```
static struct platform_driver irq_driver = {
    .probe = irq_probe,
    .remove = irq_remove,
    .suspend = irq_suspend,
    .resume = irq_resume,
    .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
    },
};

static void __exit irq_test_exit(void)
{
    platform_driver_unregister(&irq_driver);
}

static int __init irq_test_init(void)
{
    return platform_driver_register(&irq_driver);
}

module_init(irq_test_init);
module_exit(irq_test_exit);

MODULE_LICENSE("Dual BSD/GPL");
```

如下图所示，是 platform_driver 的 irq_resume、irq_suspend、irq_remove 函数，在 irq_remove 函数中使用中断释放函数。

```
static int irq_remove (struct platform_device *pdev)
{
    free_irq(IRQ_EINT(9), pdev);
    free_irq(IRQ_EINT(10), pdev);

    return 0;
}

static int irq_suspend (struct platform_device *pdev, pm_message_t state)
{
    DPRINTK("irq suspend:power off!\n");
    return 0;
}

static int irq_resume (struct platform_device *pdev)
{
    DPRINTK("irq resume:power on!\n");
    return 0;
}
```

接着重点介绍一下初始化函数 irq_probe，如下图所示。首先对中断 IO 进行检测，是否被占用了，处理方式一般就是申请 IO，看是否成功，申请成功之后就将 GPIO 配置为上拉模式，然后调用 gpio_free 将其释放。

```
static int irq_probe(struct platform_device *pdev)
{
    int ret;
    char *banner = "irq_test Initialize\n";

    printk(banner);

    ret = gpio_request(EXYNOS4_GPX1(1), "EINT9");
    if (ret) {
        printk("%s: request GPIO %d for EINT9 failed, ret = %d\n", DRIVER_NAME,
            EXYNOS4_GPX1(1), ret);
        return ret;
    }

    s3c_gpio_cfgpin(EXYNOS4_GPX1(1), S3C_GPIO_SFN(0xF));
    s3c_gpio_setpull(EXYNOS4_GPX1(1), S3C_GPIO_PULL_UP);
    gpio_free(EXYNOS4_GPX1(1));
}
```

然后对另一个中断 IO 进行初始化，如下图所示。


```
s3c_gpio_cfgpin(EXYNOS4_GPX1(2), S3C_GPIO_SFN(0xF));
s3c_gpio_setpull(EXYNOS4_GPX1(2), S3C_GPIO_PULL_UP);
gpio_free(EXYNOS4_GPX1(2));

ret = request_irq(IRQ_EINT(9), eint9_interrupt,
                 IRQ_TYPE_EDGE_FALLING /*IRQF_TRIGGER_FALLING*/, "eint9", pdev);
if (ret < 0) {
    printk("Request IRQ %d failed, %d\n", IRQ_EINT(9), ret);
    goto exit;
}

ret = request_irq(IRQ_EINT(10), eint10_interrupt,
                 IRQ_TYPE_EDGE_FALLING /*IRQF_TRIGGER_FALLING*/, "eint10", pdev);
if (ret < 0) {
    printk("Request IRQ %d failed, %d\n", IRQ_EINT(10), ret);
    goto exit;
}

return 0;

exit:
return ret;
}
```

如上图所示，对 IO 初始化之后，调用中断申请函数 `request_irq`，申请这两个中断。其中用到了参数 “`IRQ_TYPE_EDGE_FALLING`”，这个代表下降沿触发，这个宏定义在头文件 “`include/linux/irq.h`” 中，如下图所示。

```
* request_irq()
* IRQ_NOTHREAD - Interrupt cannot be threaded
* IRQ_NOAUTOEN - Interrupt is not automatically enabled in
*               request/setup_irq()
* IRQ_NO_BALANCING - Interrupt cannot be balanced (affinity set)
* IRQ_MOVE_PCNTXT - Interrupt can be migrated from process context
* IRQ_NESTED_THEAD - Interrupt nests into another thread
*/
enum {
    IRQ_TYPE_NONE = 0x00000000,
    IRQ_TYPE_EDGE_RISING = 0x00000001,
    IRQ_TYPE_EDGE_FALLING = 0x00000002,
    IRQ_TYPE_EDGE_BOTH = (IRQ_TYPE_EDGE_FALLING | IRQ_TYPE_EDGE_RISING),

    IRQ_TYPE_LEVEL_HIGH = 0x00000004,
    IRQ_TYPE_LEVEL_LOW = 0x00000008,
    IRQ_TYPE_LEVEL_MASK = (IRQ_TYPE_LEVEL_LOW | IRQ_TYPE_LEVEL_HIGH),
    IRQ_TYPE_SENSE_MASK = 0x0000000f,

    IRQ_TYPE_PROBE = 0x00000010,

    IRQ_LEVEL = (1 << 8),
    IRQ_PER_CPU = (1 << 9),
}
```

接着就可以定义中断处理函数，如下图所示，对这两个中断分别定义 `irqreturn_t` 函数。

```
static irqreturn_t eint9_interrupt(int irq, void *dev_id) {  
    printk("%s(%d)\n", __FUNCTION__, __LINE__);  
    return IRQ_HANDLED;  
}  
  
static irqreturn_t eint10_interrupt(int irq, void *dev_id) {  
    printk("%s(%d)\n", __FUNCTION__, __LINE__);  
    return IRQ_HANDLED;  
}
```

如上图所示，上面使用了一句比较特殊的代码 “`printk("%s(%d)\n", __FUNCTION__, __LINE__);`” 这句代码在调试的过程中非常有用，就是打印当前所在的函数以及对应的行，在后面测试的时候就可以看到其效果。

另外将打印函数做一个简单的处理，在调试过程中少不了用到打印函数，在调试完了之后，要一个一个的去删除打印信息会很麻烦，这里将打印函数做一个宏定义。如果定义了宏变量 “`IRQ_DEBUG`”，那么运行驱动的时候就会打印 `DPRINTK` 就会生效。当然还有一些很重要的信息，例如在初始化中，有些信息可能需要每次运行的时候都打印，如下图所示。

```
/*中断函数头文件*/
#include <linux/interrupt.h>
#include <linux/irq.h>

#define IRQ_DEBUG
#ifdef IRQ_DEBUG
#define DPRINTK(x...) printk("IRQ_CTL DEBUG:" x)
#else
#define DPRINTK(x...)
#endif

#define DRIVER_NAME "irq_test"
```

如下图所示，简单修改一下编译文件 Makefile，如下图所示。

```
#!/bin/bash
#通知编译器我们要编译模块的哪些源码
#这里是编译itop4412_irq.c这个文件编译成中间文件itop4412_irq.o
obj-m += itop4412_irq.o

#源码目录变量，这里用户需要根据实际情况选择路径
#作者是将Linux的源码拷贝到目录/home/topeet/android4.0下并解压的
KDIR := /home/topeet/android4.0/iTop4412_Kernel_3.0

#当前目录变量
PWD ?= $(shell pwd)

#make命名默认寻找第一个目标
#make -C就是指调用执行的路径
#$(KDIR) Linux源码目录，作者这里指的是/home/topeet/android4.0/iTop4412_Kernel_3.0
#$(PWD) 当前目录变量
#modules要执行的操作
all:
    make -C $(KDIR) M=$(PWD) modules

#make clean执行的操作是删除后缀为o的文件
clean:
    rm -rf *.mod.c *.o *.order *.ko *.mod.o *.symvers
```

修改完成之后，在 Ubuntu 系统下使用命令 “mkdir irq_test”，新建文件夹 “irq_test”，然后将修改好的驱动文件 “itop4412_irq.c”、Makefile 文件拷贝到文件夹 “irq_test” 中，如下图所示。

```
root@ubuntu: /home/topeet/irq_test
root@ubuntu:/home/topeet# mkdir irq_test
root@ubuntu:/home/topeet# cd irq_test/
root@ubuntu:/home/topeet/irq_test# ls
itop4412_irq.c  Makefile
root@ubuntu:/home/topeet/irq_test#
```

使用编译命令“make”编译驱动，如下图所示。

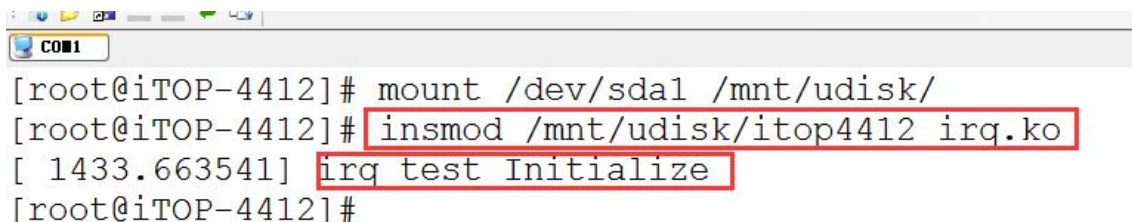
```
root@ubuntu: /home/topeet/irq_test
root@ubuntu:/home/topeet# mkdir irq_test
root@ubuntu:/home/topeet# cd irq_test/
root@ubuntu:/home/topeet/irq_test# ls
itop4412_irq.c  Makefile
root@ubuntu:/home/topeet/irq_test# make
make -C /home/topeet/android4.0/iTop4412_Kernel_3.0 M=/home/topeet/irq_test modules
make[1]: Entering directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
  CC [M] /home/topeet/irq_test/itop4412_irq.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/topeet/irq_test/itop4412_irq.mod.o
  LD [M] /home/topeet/irq_test/itop4412_irq.ko
make[1]: Leaving directory `/home/topeet/android4.0/iTop4412_Kernel_3.0'
root@ubuntu:/home/topeet/irq_test# ls
itop4412_irq.c  itop4412_irq.mod.c  itop4412_irq.o  modules.order
itop4412_irq.ko  itop4412_irq.mod.o  Makefile       Module.synvers
root@ubuntu:/home/topeet/irq_test#
```

将生成的驱动模块“itop4412_irq.ko”拷贝到 U 盘。

启动开发板，将 U 盘插入开发板，使用命令“mount /dev/sda1 /mnt/udisk/”加载 U 盘，如下图所示。


```
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]#
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
```

使用命令 “insmod /mnt/udisk/itop4412_irq.ko” 加载驱动模块，如下图所示。



```
COM1
[root@iTOP-4412]# mount /dev/sda1 /mnt/udisk/
[root@iTOP-4412]# insmod /mnt/udisk/itop4412_irq.ko
[ 1433.663541] irq test Initialize
[root@iTOP-4412]#
```

如上图所示，打印了初始化代码，没有打印任何错误。

接着使用命令 “cat /proc/interrupts” 查看申请的中断。

```
[root@iTOP-4412]# cat /proc/interrupts
```

	CPU0	CPU1	CPU2	CPU3		
24:	58	0	0	0	s3c-uart	s5pv210-uart
26:	119	0	0	0	s3c-uart	s5pv210-uart
98:	0	0	0	0	GIC	s3c-pl330.0
99:	0	0	0	0	GIC	s3c-pl330.1
100:	0	0	0	0	GIC	s3c-pl330.2
107:	0	0	0	0	GIC	s3c2410-wdt
108:	0	0	0	0	GIC	s3c2410-rtc alarm
121:	9	0	0	0	GIC	mct_comp_irq
123:	42509	0	0	0	GIC	s3c2440-i2c.1
125:	1	0	0	0	GIC	s3c2440-i2c.3
126:	60	0	0	0	GIC	s3c2440-i2c.4
127:	0	0	0	0	GIC	s3c2440-i2c.5
129:	6	0	0	0	GIC	s3c2440-i2c.7
134:	49064	0	0	0	GIC	ehci_hcd:usb1
135:	41	0	0	0	GIC	s3c-udc
139:	0	0	0	0	GIC	mmc1
140:	52	0	0	0	GIC	mmc2
141:	3687	0	0	0	GIC	mmc0
160:	0	0	0	0	GIC	samsung-rp
281:	0	0	0	0	COMBINER	s3cfb
352:	1	0	0	0	exynos-eint	
359:	0	0	0	0	exynos-eint	s3c-sdhci.2
361:	0	0	0	0	exynos-eint	eint9
362:	0	0	0	0	exynos-eint	eint10
367:	1	0	0	0	exynos-eint	s5m87xx-irq
370:	0	0	0	0	exynos-eint	switch-gpio
428:	0	0	0	0	s5m8767	rtc-alarm0
TOTL:	0	0	0	0		Timer broadcast interrupts

使用命令 “rmmod itop4412_irq” 卸载中断的驱动，如下图所示。

```
IPI5:      0      0      0      0      0      CPU backtrace
LOC:      35800      54125      37631      40424      Local timer interrupts
Err:      0
[root@iTOP-4412]# lsmod
itop4412_irq 2142 0 - Live 0xbf000000
[root@iTOP-4412]# rmmod itop4412_irq
[root@iTOP-4412]#
```

接着再次使用命令 “cat /proc/interrupts” 查看申请的中断，如下图所示已经没有了。

```
141:      3932      0      0      0      0      GIC      mmc0
160:      0      0      0      0      0      GIC      samsung-rp
281:      0      0      0      0      0      COMBINER s3cfb
352:      1      0      0      0      0      exynos-eint
359:      0      0      0      0      0      exynos-eint s3c-sdhci.2
367:      1      0      0      0      0      exynos-eint s5m87xx-irq
370:      0      0      0      0      0      exynos-eint switch-gpio
428:      0      0      0      0      0      s5m8767 rtc-alarm0
IPI0:      0      0      0      0      0      Timer broadcast interrupts
IPI1:      9237      12294      9163      8835      Rescheduling interrupts
IPI2:      97      94      102      105      Function call interrupts
IPI3:      60      40      78      53      Single function call interrupts
IPI4:      0      0      0      0      0      CPU stop interrupts
IPI5:      0      0      0      0      0      CPU backtrace
LOC:      35800      54125      37631      40424      Local timer interrupts
```

接着使用命令 “insmod /mnt/udisk/itop4412_irq.ko” 加载驱动模块，测试功能。然后

按几下 HOME 和 BACK 按键，会出现类似下面的打印信息。

```
[root@iTOP-4412]# insmod /mnt/udisk/itop4412_irq.ko
[ 1732.895078] irq_test Initialize
[root@iTOP-4412]# [ 1736.489298] eint9_interrupt(32)
[ 1736.632151] eint9_interrupt(32)
[ 1737.393576] eint10_interrupt(39)
[ 1738.210704] eint10_interrupt(39)
[ 1739.196217] eint10_interrupt(39)
```

再回过头去看一下源码，如下图所示，32 行和 39 行正是对应的那两处打印的代码，分别在函数 eint9_interrupt 和 eint10_interrupt 中。


```

30 static irqreturn_t eint9_interrupt(int irq, void *dev_id) {
31
32     printk("%s(%d)\n", __FUNCTION__, __LINE__);
33
34     return IRQ_HANDLED;
35 }
36
37 static irqreturn_t eint10_interrupt(int irq, void *dev_id) {
38
39     printk("%s(%d)\n", __FUNCTION__, __LINE__);
40
41     return IRQ_HANDLED;
42 }
43

```

最后再使用命令 “cat /proc/interrupts” 查看申请的中断，如下图所示，已经检测到这两个中断分别触发了几次。

126:	60	0	0	0	GIC	s3c2440-i2c.4
127:	0	0	0	0	GIC	s3c2440-i2c.5
129:	6	0	0	0	GIC	s3c2440-i2c.7
134:	60220	0	0	0	GIC	ehci_hcd:usb1
135:	41	0	0	0	GIC	s3c-udc
139:	0	0	0	0	GIC	mmc1
140:	52	0	0	0	GIC	mmc2
141:	4443	0	0	0	GIC	mmc0
160:	0	0	0	0	GIC	samsung-rp
281:	0	0	0	0	COMBINER	s3cfb
352:	1	0	0	0	exynos-eint	
359:	0	0	0	0	exynos-eint	s3c-sdhci.2
361:	2	0	0	0	exynos-eint	eint9
362:	3	0	0	0	exynos-eint	eint10
367:	1	0	0	0	exynos-eint	s5m87xx-irq
370:	0	0	0	0	exynos-eint	switch-gpio
428:	0	0	0	0	s5m8767	rtc-alarm0
IPI0:	0	0	0	0	Timer	broadcast interrupts
IPI1:	10236	13922	10705	9993	Rescheduling	interrupts
IPI2:	111	108	115	118	Function call	interrupts
IPI3:	83	44	95	66	Single function call	interrupts
TPT4:	0	0	0	0	CPU stop	interrupts