

Android Kernel 开发系列培训
主讲人：吴庆棋

E-Mail:wqq@rockchip.com

Linux 开发环境篇

Linux 驱动开发篇

Linux 内核篇

一。Android Kernel 开发环境搭建

kernel version :linux 2.6.25

GCC 编译器 : toolchain/arm-eabi-4.2.1

linux 主机环境:ubuntu-8.10

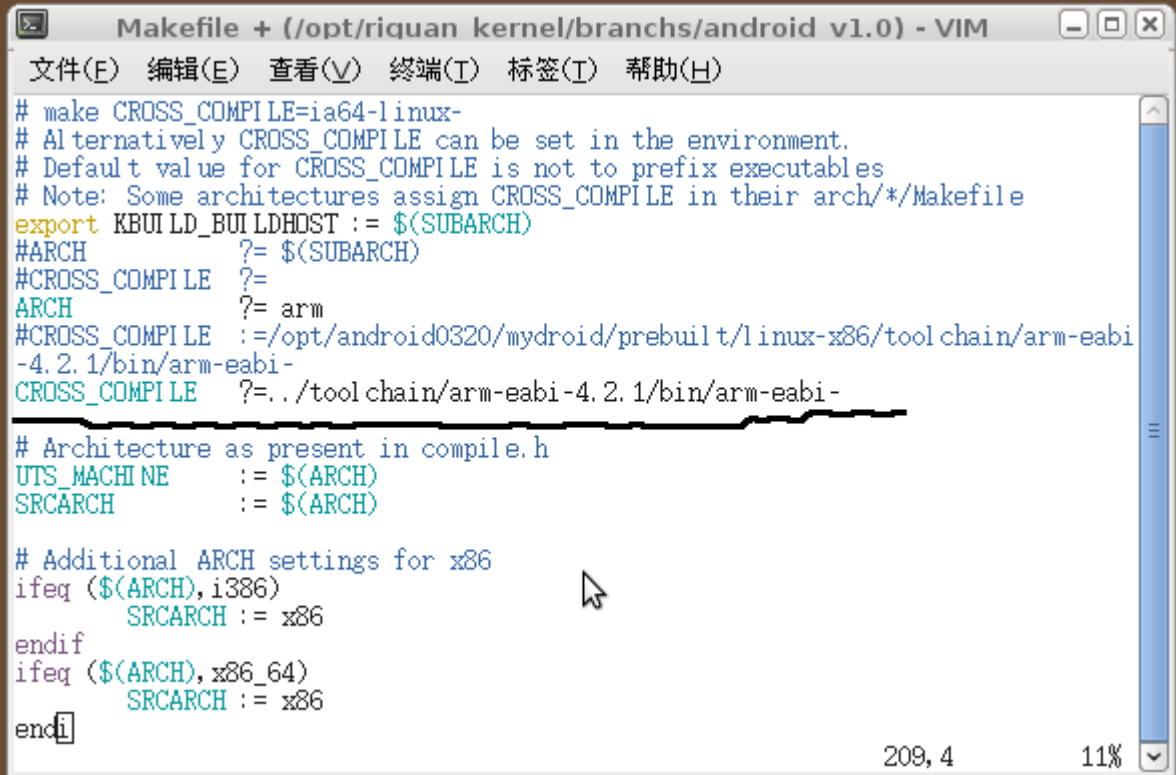
1.编译器安装

kernel 开发源码包中有 toolchain 目录，此目录为 arm gcc 交叉编译器

2.配置及编译 kernel

<1>编辑 Makefile 指定编译器路径

\$ sudo vim Makefile



```
Makefile + (/opt/riquan kernel/branches/android v1.0) - VIM
文件(F) 编辑(E) 查看(V) 终端(T) 标签(I) 帮助(H)
# make CROSS_COMPILE=ia64-linux-
# Alternatively CROSS_COMPILE can be set in the environment.
# Default value for CROSS_COMPILE is not to prefix executables
# Note: Some architectures assign CROSS_COMPILE in their arch/*/Makefile
export KBUILD_BUILDHOST := $(SUBARCH)
#ARCH ?= $(SUBARCH)
#CROSS_COMPILE ?=
ARCH ?= arm
#CROSS_COMPILE :=/opt/android0320/mydroid/prebuilt/linux-x86/toolchain/arm-eabi-4.2.1/bin/arm-eabi-
CROSS_COMPILE ?=../toolchain/arm-eabi-4.2.1/bin/arm-eabi-

# Architecture as present in compile.h
UTS_MACHINE := $(ARCH)
SRCARCH := $(ARCH)

# Additional ARCH settings for x86
ifeq ($(ARCH),i386)
    SRCARCH := x86
endif
ifeq ($(ARCH),x86_64)
    SRCARCH := x86
endif
end
```

<2>使用如下命令来配置内核

\$sudo make menuconfig

注意：

运行过程中若出现如下错误：

```
HOSTCC scripts/basic/fixdep
HOSTCC scripts/basic/docproc
HOSTCC scripts/basic/hash
HOSTCC scripts/kconfig/conf.o
scripts/kconfig/conf.c: 在函数‘conf_askvalue’中:
scripts/kconfig/conf.c:104: 警告：忽略声明有 warn_unused_result 属性的‘fgets’的返回值
scripts/kconfig/conf.c: 在函数‘conf_choice’中:
scripts/kconfig/conf.c:306: 警告：忽略声明有 warn_unused_result 属性的‘fgets’的返回值
HOSTCC scripts/kconfig/kxgettext.o
*** Unable to find the ncurses libraries or the
*** required header files.
*** 'make menuconfig' requires the ncurses libraries.
***
*** Install ncurses (ncurses-devel) and try again.
***
make[1]: *** [scripts/kconfig/dochecklxdialog] 错误 1
make: *** [menuconfig] 错误 2
```

请执行如下命令安装 ncurses

```
$ sudo apt-get install libncurses5-dev
```

安装完成后就可以使用如下命令了：

```
$ make menuconfig /*配置内核*/
$ make /*编译内核*/
$ make clean /*清除已编译的文件*/
```

二.Android Kernel 源代码阅读工具介绍

方案一：在 window 环境下阅读源代码

- linux 下建立 SAMBA 文件服务器 和 SSH 服务器
- window 下安装 source insight 代码阅读工具
- window 下安装 putty 远程终端登入工具

方案二：在 linux 环境下阅读源代码

- vi 文本编辑器
- cscope + ctags

1.搭建 window 下的阅读代码环境

<1>Ubuntu linux 下建立 SAMBA 文件服务器

建立不需用户名、密码的 Samba 共享文件夹

(注：若在 window 下无法创建和修改文件，在 linux 下用 `chmod 777 XXX -R` 改变权限)

- 安装 Samba 服务器:

```
#apt-get install samba
```

- 创建要共享的文件夹：

```
#mkdir /root/share
```

```
#chmod 777 /root/share -R /*改变文档属性为所有用户可读写*/
```

- 备份并编辑 smb.conf

```
#cp /etc/samba/smb.conf /etc/samba/smb.conf.bak
```

```
#vim /etc/samba/smb.conf
```

修改[global]的内容如下：

修改 workgroup：

```
Workgroup = SAMBA
```

增加对中文的支持：

```
display charset = UTF-8
```

```
dos charset = cp936
```

```
unix charset = UTF-8
```

修改 security：

```
Security = user
```

修改 encrypt passwords：

```
encrypt passwords = yes
```

增加共享的文件夹：

```
[Share]
```

```
comment = My Share Directory
```

```
path=/root/share
```

```
available = yes
```

```
browseable = yes
```

```
public = yes
```

```
writable = yes
```

```
create mode = 0664
```

```
directory mode = 0775
```

- 测试参数是否正确：

```
#testparm
```

- 重启 samba 服务：

```
#/etc/init.d/samba restart
```

- 在 window 下测试 samba 服务器

程序 - > 运行 输入 linux 的 IP 地址：\\xxx.xxx.xxx.xxx

若可以看到 share 的文件夹，说明已成功了。

- 将内核源代码拷贝到这个目录，现在可以用 source insight 看代码了!!!

<2>Ubuntu linux 安装 SSH 终端登入服务器

- 请在终端使用命令测试：

```
ssh localhost
```

若出现以下错误，则是因为还没有安装 ssh-server：

ssh: connect to host localhost port 22: Connection refused

- 安装 SSH-server :

```
$ sudo apt-get install openssh-server
```

- 启动 SSH-Server

```
sudo /etc/init.d/ssh start
```

现在可以在 window 下用终端工具登入 linux 机器进行 make 编译。

(建议使用 : Putty、secureCRT)

2.搭建 linux 下的阅读代码环境

<1>安装文本编辑器 : `$ sudo apt-get install vim`

<2>制作 cscope 索引文件 : `$sudo apt-get install cscope`

<3>制作 ctags 文件 : `$sudo apt-get install ctags`

使用方法:

- a) 首先进入 kernel 的主目录(也就是内核源代码根目录)
- b) 输入: `make cscope`
- c) 输入: `ctags -R`

然后就可以用 vi 来阅读源码了.

注意: 要记住,不要再改变你的当前工作目录了.

比如你要查看 `init/main.c`,你要用: `vi init/main.c`

而不要 `cd init; vi main.c`

跟踪函数使用: `Ctrl+] (同时按下 Ctrl 键和"]"键)`

如果此函数有多个实例,会有个列表供你选择.

返回上一级函数使用: `Ctrl+t (同时按下 Ctrl 键和"t"键)`

在 vim 中执行":`help tags`"命令查询它的用法

二.Android linux 驱动开发

1.Android linux 驱动开发术语

- 内核模块与应用程序

一个应用程序从头到尾完成一个任务,而模块则是为以后处理某些请求而注册自己,完成这个任务后,它的“主”函数就立即中止了。

- 内核空间和用户空间

当谈到软件时,我们通常称执行态为“内核空间”和“用户空间”,在 Linux 系统中,内核在**最高级**执行,也称为“管理员态”,在这一级任何操作都可以执行。而应用程序则执行在**最低级**,所谓的“用户态”,在这一级处理器禁止对硬件的直接访问和对内存的未授权访问。

模块是在所谓的“内核空间”中运行的,而应用程序则是在“用户空间”中运行的。它们分别引用不同的内存映射,也就是程序代码使用不同的“地址空间”。

Linux 通过系统调用和硬件中断完成从用户空间到内核空间的控制转移。执行系统调用的内核代码在进程的上下文中执行,它执行调用进程的操作而且可以访问进程地址空间的数据。但处理中断与此不同,处理中断的代码相对进程而言是异步的,

而且与任何一个进程都无关。模块的作用就是扩展内核的功能，是运行在内核空间的模块化的代码。模块的某些函数作为系统调用执行，而某些函数则负责处理中断。

2.Android linux 设备驱动开发的基本流程

由于嵌入式设备由于硬件种类非常丰富，在默认的内核发布版中不一定包括所有驱动程序。所以进行嵌入式 Linux 系统的开发，很大的工作量是为各种设备编写驱动程序。除非系统不使用操作系统，程序直接操纵硬件。Android Linux 系统驱动程序开发与普通 Linux 开发没有区别。可以在硬件生产厂家或者 Internet 上寻找驱动程序，也可以根据相近的硬件驱动程序来改写，这样可以加快开发速度。实现一个嵌入式 Linux 设备驱动的大致流程如下。

(1) 查看原理图，理解设备的工作原理。一般嵌入式处理器的生产商提供参考电路，也可以根据需要自行设计。

(2) 实现初始化函数。在驱动程序中实现驱动的注册和卸载。

(3) 实现中断服务，并用 request_irq 向内核注册，中断并不是每个设备驱动所必需的。

(4) 实现定时器扫描服务，扫描方式也不是每个设备驱动所必需的。

(5) 编译该驱动程序到内核中，或者用 insmod 命令加载模块。

(6) 测试该设备，编写应用程序，对驱动程序进行测试。

3.Android linux 设备驱动开发关键函数介绍

(1) 设备驱动初始化与注册函数

从 Linux 2.6 起引入了一套新的驱动管理和注册机制:Platform_device 和 Platform_driver。Linux 中大部分的设备驱动，都可以使用这套机制，设备用 Platform_device 表示，驱动用 Platform_driver 进行注册。

Linux platform driver 机制和传统的 device driver 机制(通过 driver_register 函数进行注册)相比，一个十分明显的优势在于 platform 机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序中使用这些资源时通过 platform device 提供的标准接口进行申请并使用。这样提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性(这些标准接口是安全的)。

```
int platform_device_register(struct platform_device *pdev)
```

```
void platform_device_unregister(struct platform_device *pdev)
```

```
int platform_driver_register(struct platform_driver *drv)
```

```
void platform_driver_unregister(struct platform_driver *drv)
```

(2) 加载和卸载驱动程序

■ 驱动入口函数

在编写模块程序时，必须提供两个函数，

- 一个是 int init_module()，

在加载此模块的时候自动调用，负责进行设备驱动程序的初始化工作。

init_module()返回 0，表示初始化成功，返回负数表示失败，它在内核中注册一定的功能函数。在注册之后，如果有程序访问内核模块的某个功能，内核将查表获得该功能的位置，然后调用功能函数。

init_module()的任务就是为以后调用模块的函数做准备。

• 另一个函数是 `void cleanup_module()` ,
该函数在模块被卸载时调用, 负责进行 设备驱动程序的清除工作。
这个函数的功能是取消 `init_module()` 所做的事情, 把 `init_module()` 函数在内核中注册的功能函数完全卸载, 如果没有完全卸载, 在此模块下次调用时, 将会因为有重名的函数而导致调入失败。

函数原型:

```
#define module_init(initfn) \
    static inline initcall_t __inittest(void) \
    { return initfn; } \
    int init_module(void) __attribute__((alias(#initfn)));
/* This is only required if you want to be unloadable. */
#define module_exit(exitfn) \
    static inline exitcall_t __exittest(void) \
    { return exitfn; } \
    void cleanup_module(void) __attribute__((alias(#exitfn)));
```

■ 模块加载与卸载

虽然模块作为内核的一部分, 但并未被编译到内核中, 它们被分别编译和链接 成目标文件。

Linux 中模块可以用 C 语言编写, 用 `gcc` 命令编译成模块*.o ,
在命令行里加上 `-c` 的参数和 `“-D__KERNEL__ -DMODULE ”` 参数。

然后用 `depmod -a` 使此模块成为可加载模块。

模块用 `insmod` 命令加载, 用 `rmmod` 命令来卸载, 这两个命令分别调用 `init_module()` 和 `cleanup_module()` 函数, 还可以用 `lsmod` 命令来查看所有已加载的模块的状态。

`insmod` 命令可将编译好的模块调入内存。内核模块与系统中其他程序一样是已链接的目标文件, 但不同的是它们被链接成可重定位映像。`insmod` 将执行一个 特权级系统调用 `get_kernel_sysms()` 函数以找到内核的输出内容, `insmod` 修改模块对内核符号的引用后, 将再次使用特权级系统调用 `create_module()` 函数来申请足够的物理内存空间, 以保存新的模块。内核将为其分配一个新的 `module` 结构, 以及足够的内核内存, 并将新模块添加在内核模块链表的尾部, 然后将新模块标记为 `uninitialized`。

利用 `rmmod` 命令可以卸载模块。如果内核中还在使用此模块, 这个模块就不能被卸载。原因是如果设备文件正被一个进程打开就卸载还在使用的内核模块, 并导致对内核模块的读/写函数所在内存区域的调用。如果幸运, 没有其他代码被加载到那个内存区域, 将得到一个错误提示; 否则, 另一个内核模块被加载到同一区域, 这就意味着程序跳到内核中另一个函数的中间, 结果是不可预见的。

(3)内存操作函数

作为系统核心的一部分, 设备驱动程序在申请和释放内存时不是调用 `malloc` 和 `free`, 而代之以调用 `kmalloc` 和 `kfree`, 它们在 `linux/kernel.h` 中被定义为:

```
void * kmalloc(unsigned int len, int priority);
```

```
void kfree(void * obj);
```

参数 `len` 为希望申请的字节数, `obj` 为要释放的内存指针。 `priority` 为分配内存操作的优先级, 即在没有足够空闲内存时如何操作, 一般由取值 `GFP_KERNEL` 解决即可。

(4)时钟函数

在设备驱动程序中，一般都需要用到计时机制。在Linux系统中，时钟是由系统接管的，设备驱动程序可以向系统申请时钟。与时钟有关的系统调用有：

```
#include <asm/param.h>
#include <linux/timer.h>
void add_timer(struct timer_list * timer);
int del_timer(struct timer_list * timer);
inline void init_timer(struct timer_list * timer);
struct timer_list 的定义为：
struct timer_list {
    struct timer_list *next;
    struct timer_list *prev;
    unsigned long expires;
    unsigned long data;
    void (*function)(unsigned long d);
};
```

其中，expires 是要执行 function 的时间。系统核心有一个全局变量 jiffies 表示当前时间，一般在调用 add_timer 时 $jiffies = JIFFIES + num$ ，表示在 num 个系统最小时间间隔后执行 function 函数。系统最小时间间隔与所用的硬件平台有关，在核心里定义了常数 HZ 表示一秒内最小时间间隔的数目，则 $num * HZ$ 表示 num 秒。系统计时到预定时间就调用 function，并把此子程序从定时队列里删除，可见，如果想要每隔一定时间间隔执行一次的话，就必须在 function 里再一次调用 add_timer。Function 的参数 d 即为 timer 里面的 data 项。

在 linux 2.6 版本以上的内核中新增了高精度定时器的支持。有关的系统调用有：

```
#include <linux/interrupt.h>
#include <linux/timer.h>
#include <linux/jiffies.h>
void hrtimer_init(struct hrtimer *timer, clockid_t which_clock, enum hrtimer_mode mode);

/* Basic timer operations: */
int hrtimer_start(struct hrtimer *timer, ktime_t tim, const enum hrtimer_mode mode);
int hrtimer_cancel(struct hrtimer *timer);
static inline ktime_t ktime_add(const ktime_t add1, const ktime_t add2)
static inline int hrtimer_restart(struct hrtimer *timer)
struct hrtimer { 定义为
    struct rb_node          node;
    ktime_t                 _expires;
    ktime_t                 _softexpires;
    enum hrtimer_restart    (*function)(struct hrtimer *);
    struct hrtimer_clock_base *base;
    unsigned long           state;
    struct list_head        cb_entry;
#ifdef CONFIG_TIMER_STATS
```

```

        int                                start_pid;
        void                                *start_site;
        char                                start_comm[16];
#endif
};

```

其中，`ktime_t tim` 是要执行 function 的时间。系统计时到预定时间就调用 function，并把此子程序从定时队列里删除，可见，如果想要每隔一定时间间隔执行一次的话，就必须在 function 里再一次调用 `hrtimer_restart` 或是返回宏 `HRTIMER_RESTART`。

(5)中断管理

设备驱动程序通过调用 `request_irq` 函数来申请中断，通过 `free_irq` 来释放中断。

```

int request_irq(
    unsigned int irq,
    void (*handler)(int irq,void dev_id,struct pt_regs *regs),
    unsigned long flags,
    const char *device,
    void *dev_id
);
void free_irq(unsigned int irq, void *dev_id);

```

通常从 `request_irq` 函数返回的值为 0 时，表示申请成功；负值表示出现错误。

- `irq` 表示所要申请的硬件中断号。
- `handler` 为向系统登记的中断处理子程序，中断产生时由系统来调用，调用时所带参数 `irq` 为中断号，`dev_id` 为申请时告诉系统的设备标识，`regs` 为中断发生时寄存器内容。
- `device` 为设备名，将会出现在 `/proc/interrupts` 文件里。
- `flag` 是申请时的选项，它决定中断处理程序的一些特性，其中最重要的是决定中断处理程序是快速处理程序（`flag` 里设置了 `SA_INTERRUPT`）还是慢速处理程序（不设置 `SA_INTERRUPT`）。

下面的代码将在 SBC-2410X 的 Linux 中注册外部中断 2。

```

eint_irq = IRQ_EINT2;
set_external_irq (eint_irq, EXT_FALLING_EDGE,GPIO_PULLUP_DIS);
ret_val = request_irq(eint_irq,eint2_handler, "S3C2410X eint2",0);
if(ret_val < 0)
return ret_val;
}

```

4.如何向内核增加一个新的驱动

对于一个开发者来说，将自己开发的内核代码添加到 Linux 内核中，需要有三个步骤。

- (1) 确定把自己的开发代码放入到内核的位置；
- (2) 把自己开发的功能增加到 Linux 内核的配置选项中，使用户能够选择此功能
- (3) 构建子目录 Makefile，根据用户的选择，将相应的代码编译 Linux 内核中。

下面，我们就通过一个简单的例子，结合前面学到的知识，来说明如何将前面设计的驱动加入到 Linux 内核。

- 将在 `driver/input/keyboard` 目录中添加一个按键驱动 `testkbd.c`
- 修改 `driver/input/keyboard/Makefile` 文件


```
root@localhost:/opt/jane090304/mydroid/kernel/drivers/input/keyboard
#
# Makefile for the input core drivers.
#
# Each configuration option enables a list of files.

obj-$(CONFIG_KEYBOARD_ATKBD) += atkbd.o
obj-$(CONFIG_KEYBOARD_TEST) += testkbd.o
obj-$(CONFIG_KEYBOARD_SUNKBD) += sunkbd.o
obj-$(CONFIG_KEYBOARD_LKKBD) += lkkbd.o
obj-$(CONFIG_KEYBOARD_XTKBD) += xtkbd.o
obj-$(CONFIG_KEYBOARD_AMIGA) += amikbd.o
obj-$(CONFIG_KEYBOARD_ATARI) += atakbd.o
obj-$(CONFIG_KEYBOARD_LOCOMO) += locomokbd.o
obj-$(CONFIG_KEYBOARD_NEWTON) += newtonkbd.o
obj-$(CONFIG_KEYBOARD_STOWAWAY) += stowaway.o
obj-$(CONFIG_KEYBOARD_CORGI) += corgikbd.o
obj-$(CONFIG_KEYBOARD_SPITZ) += spitzkbd.o
obj-$(CONFIG_KEYBOARD_TOSA) += tosakbd.o
obj-$(CONFIG_KEYBOARD_HIL) += hil_kbd.o
obj-$(CONFIG_KEYBOARD_HIL_OLD) += hilkbd.o
obj-$(CONFIG_KEYBOARD_OMAP) += omap-keypad.o
obj-$(CONFIG_KEYBOARD_PXA27x) += pxa27x_keypad.o
"Makefile" 30L, 1196C 8,1 顶端
```

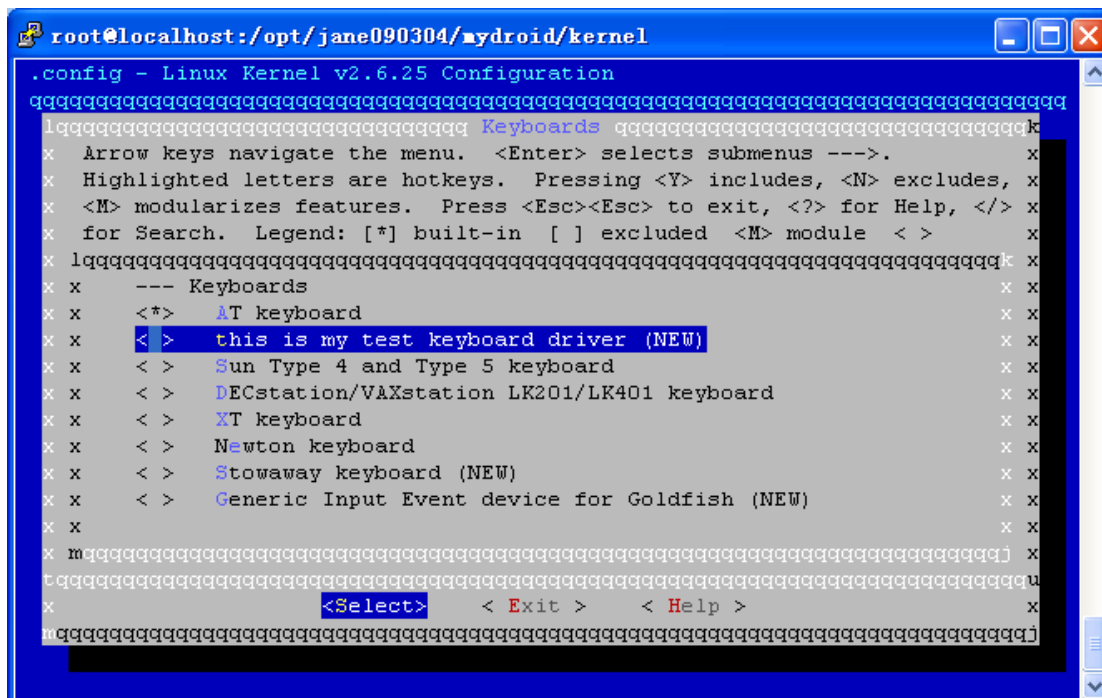
- 修改 driver/input/keyboard/Kconfig 文件

```
root@localhost:/opt/jane090304/mydroid/kernel/drivers/input/keyboard
To compile this driver as a module, choose I here: the
module will be called atkbd.
config KEYBOARD_TEST
    tristate "this is my test keyboard driver" if EMBEDDED || !X86_PC
    default n
    select SERIO
    select SERIO_LIBPS2
    select SERIO_I8042 if X86_PC
    select SERIO_GSCPS2 if GSC
    help
        Say Y here if you want to use a standard AT or PS/2 keyboard. Usually
        you'll need this, unless you have a different type keyboard (USB, I/O
        or other). This also works for AT and PS/2 keyboards connected over a
        PS/2 to serial converter.

        If unsure, say Y.

To compile this driver as a module, choose I here: the
module will be called atkbd.
config KEYBOARD_ATKBD_HP_KEYCODES
    bool "Use HP keyboard scancodes"
    depends on PARISC && KEYBOARD_ATKBD
50,4-11 8%
```

- 修改完成后,用 make menuconfig 查看键盘菜单中是否有新增了一项驱动 (如下图)



5.AD 按键驱动实例 (T28 MID 按键)

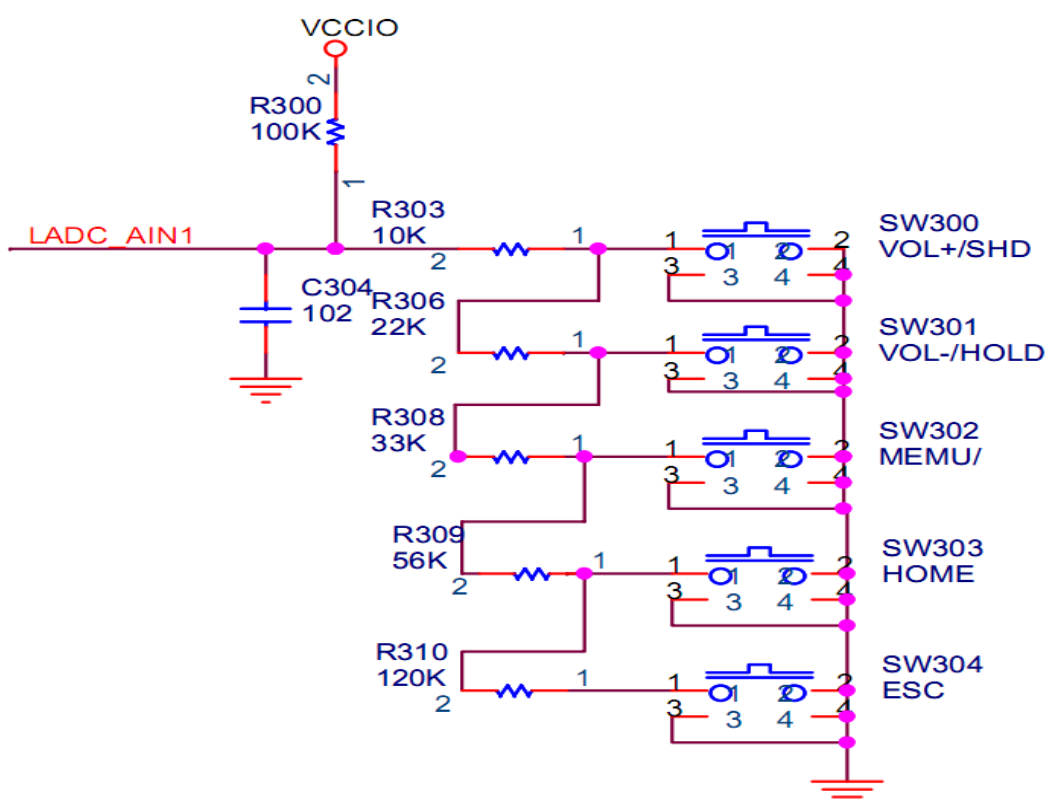


图 2-4-1 Ad 按键电路图

(1) Ad 按键接口设计

RK2806 提供了 4 个输入通道的 AD 控制器，可以方便地输入各种信号。T28 目标板选用 RK2806 微处理器，带有 5 个接到 AD 通道 1 的按键，硬件原理图如图 2-4-1 所示。按键控制采用 AD 采样方式，利用采样出来的 AD 值的不同分别区别不同的按键按下。与按键相连的通用 AD 控制器由表 6.3 所示的控制寄存器配置

Name	Offset	Size	Reset Value	Description
ADC_DATA	0x0000	W	0x00000000	ADC data registers
ADC_STAS	0x0004	W	0x00000000	ADC status register
ADC_CTRL	0x0008	W	0x00000000	ADC controller register

图 2-4-2 AD 控制器寄存器列表

ADC_CTRL

Address: Operational Base + offset(0x08)

The control register of A/D Converter.

bit	Attr	Reset Value	Description
31:7	-	-	Reserved.
6	RW	0	Interrupt status. This bit will be set to 1 when end-of-conversion. Set 0 to clear the interrupt.
5	RW	0	Interrupt enable. 0: Disable 1: Enable
4	RW	0	Start of Conversion(SOC) Set this bit to 1 to start an ADC conversion. This bit will reset to 0 by hardware when ADC conversion has started.
3	RW	0	ADC power down control bit 0: ADC power down 1: ADC power up and reset
2:0	RW	0	ADC input source selection. 000 : Input source 0 (ADC_CH[0]) 001 : Input source 1 (ADC_CH[1]) 010 : Input source 2 (ADC_CH[2]) 011 : Input source 3 (ADC_CH[3])

Notes: Attr: **RW** – Read/writable, **R** – read only, **W** – write only

图 2-4-3 AD 控制寄存器

(2) 驱动源代码解析

- Ad 控制器驱动接口函数

```
#define IN_API_DRIVER_ADC
#include <asm/arch/hw_define.h>
#include <asm/arch/typedef.h>
#include <asm/arch/hardware.h>
#include <asm/arch/api_intc.h>
#include <asm/arch/iomux.h>
```

```

#include <asm/arch/adc.h>
#include <asm/arch/rk28_scu.h>
#include <asm/arch/pll.h>
#include <asm/delay.h>

#include <linux/kernel.h>
#include <asm/io.h>

/*-----
Name      : ADCReadData(void)
Desc      :读 ADC 的值
Params    : 无
Return    : 还回 ADC 的值
-----*/
int32 ADCReadData(void)
{
    pADC_REG pheadAdc;
    int32  adcReturn;
    pheadAdc = (pADC_REG) ADC_BASE_ADDR_VA;
    if (pheadAdc->ADC_STAS == ADC_STOP)
    {
        adcReturn = (pheadAdc->ADC_DATA & 0x3ff);
    }
    else
    {
        adcReturn= -1;
    }
    return (adcReturn);
}

/*-----
Name      : ADCStart(uint8 ch)
Desc      :启动 ADC 的转换  ch (0---3)
Params    : 无
Return    :启动成功还回为 0 , 否则为 -1
-----*/
int32 ADCStart(uint8 ch)
{
    pADC_REG pheadAdc;
    pheadAdc = (pADC_REG) ADC_BASE_ADDR_VA;
    //rockchip_scu_reset_unit( 7 );

    if(ch>3)
    {
        return (-1);
    }
}

```

```

        pheadAdc->ADC_CTRL = (pheadAdc->ADC_CTRL & 0xe0) | ADC_POWER_ON |
ADC_START |ch;
        return (0);
    }

```

```

/*-----

```

```

Name      : ADCIntEnabled()
Desc      :打开 ADC 中断使能
Params    : 无
Return    :

```

```

-----*/

```

```

void ADCIntEnabled(void)
{
    pADC_REG pheadAdc;
    pheadAdc = (pADC_REG) ADC_BASE_ADDR_VA;
    pheadAdc->ADC_CTRL = (pheadAdc->ADC_CTRL & 0xdf) | ADC_ENABLED_INT;
}

```

```

/*-----

```

```

Name      : ADCIntDisabled(void)
Desc      :关闭 ADC 中断使能
Params    : 无
Return    :

```

```

-----*/

```

```

void ADCIntDisabled(void)
{
    pADC_REG pheadAdc;
    pheadAdc = (pADC_REG) ADC_BASE_ADDR_VA;
    pheadAdc->ADC_CTRL = (pheadAdc->ADC_CTRL & 0xdf) | ADC_DISABLED_INT;
}

```

```

/*-----

```

```

Name      : ADCIntHandler(void)
Desc      :ADC 中断服务程序
Params    : 无
Return    :

```

```

-----*/

```

```

void ADCIntHandler(void)
{
    int32 adcData;
    ADCIntDisabled();
    adcData = ADCReadData();
}

```

```

/*-----
Name      :ADCInit(void)
Desc      :ADC 初值化
Params    : 无
Return    :初始化成功还回为 0 , 否则为 -1
-----*/

int32 ADCInit(void)
{
    pADC_REG pheadAdc;

    pheadAdc = (pADC_REG) ADC_BASE_ADDR_VA;

    rockchip_scu_register( SCU_IPID_LSADC , SCU_MODE_FREQ , 1 , NULL );    /* max 1M
CLK*/

    pheadAdc->ADC_CTRL = ADC_POWER_ON;
    if(ADCStart(Adc_channel0) !=0)
    {
        return (-1);
    }

    return (0);
}

/*-----
Name      : ADCDeinit(void)
Desc      :ADC 反初值化
Params    : 无
Return    :
-----*/

void ADCDeinit(void)
{
    pADC_REG pheadAdc;
    pheadAdc = (pADC_REG) ADC_BASE_ADDR_VA;

    pheadAdc->ADC_CTRL = ADC_POWER_OFF;

    rockchip_scu_disableclk( SCU_IPID_LSADC );
}

/*-----
Name      : RockAdcScanning(void)
Desc      :系统 adc 扫描 , 对四个通道的 adc 定时扫描

```

Params : 无

Return :

-----*/

void RockAdcScanning(void)

{

int32 adcTemp;

adcTemp = ADCReadData();

if (adcTemp != -1)

{

g_adcValue[g_adcch] = (uint16)adcTemp;

}

else

{

printf ("\nRockAdcScanning: not stop");

return;

}

g_adcch++;

if (g_adcch>=Adc_channel_max)

g_adcch = Adc_channel0;

ADCStart(g_adcch);

}

/*BATTERY ADC*/

u16 get_rock_adc0(void)

{

return g_adcValue[0];

}

int get_rock_adc1(void)

{

return g_adcValue[1];

}

int get_rock_adc2(void)

{

return g_adcValue[2];

```
}
```

```
/*BATTERY VREF_ADC*/
```

```
u16 get_rock_adc3(void)
```

```
{
```

```
    return g_adcValue[3];
```

```
}
```

● 按键驱动

```
/*设备资源*/
```

```
static struct resource key_resources[] = {
```

```
    [0] = {
```

```
        .start= ADC_BASE_ADDR,
```

```
        .end = ADC_BASE_ADDR + SZ_4K - 1,
```

```
        .flags    = IORESOURCE_MEM,
```

```
    },
```

```
    [1] = {
```

```
        .start= RK28_ID_ADC,
```

```
        .end = RK28_ID_ADC,
```

```
        .flags    = IORESOURCE_IRQ,
```

```
    },
```

```
};
```

```
static struct platform_device rk28_key_device = {
```

```
    .name          = "rk28_AD_button",
```

```
    .id   = -1,
```

```
    .resource = key_resources,
```

```
    .num_resources = ARRAY_SIZE(key_resources),
```

```
};
```

```
void rock28_add_device_key(void)
```

```
{
```

```
    platform_device_register(&rk28_key_device); /*设备资源注册*/
```

```
}
```

1 . 系统资源和宏定义

```
#include <linux/kernel.h>
```

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

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

```
#include <linux/interrupt.h>
```

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

```
#include <linux/device.h>
```

```
#include <linux/platform_device.h>
```

```
#include <linux/clk.h>
```

```
#include <linux/err.h>
```

```
#include <linux/delay.h>
```



```

#include <asm/mach-types.h>
#include <asm/mach/arch.h>
#include <asm/mach/map.h>
#include <asm/arch/typedef.h>
#include <asm/arch/gpio.h>
#include <asm/arch/hardware.h>
/*
 * Keypad Controller registers
 */
// #define RK28_PRINT
#include <asm/arch/rk28_debug.h> 0
#include <asm/arch/adc.h>
/* Debug */
#if 0
#define DBG(x...) printk(KERN_INFO x)
#else
#define DBG(x...)
#endif
// ROCKCHIP AD KEY CODE ,for demo board
//      key      ---> EV
#if 1
#define AD1KEY1      103//DPAD_UP      -----UP
#define AD1KEY2      106//DPAD_RIGHT-----FFD
#define AD1KEY3      105//DPAD_LEFT-----FFW
#define AD1KEY4      108//DPAD_DOWN-----DOWN
#define AD1KEY5      62  //ENDCALL          WAKE_DROPPED
#define AD1KEY6      28  //ENTER

#define AD2KEY1      59  //MENU              //115  //VOLUME_UP
#define AD2KEY2      102  //HOME              //114  //VOLUME_DOWN
#define AD2KEY3      158  //BACK----ESC      //59  //MENU
#define AD2KEY4      114  //VOLUME_DOWN    //62  //ENDCALL
#define AD2KEY5      115  //VOLUME_UP    //158//BACK-----ESC
#define AD2KEY6      116  //POWER

#else
#define AD1KEY1      103//DPAD_UP      -----UP
#define AD1KEY2      106//DPAD_RIGHT-----FFD
#define AD1KEY3      105//DPAD_LEFT-----FFW
#define AD1KEY4      108//DPAD_DOWN-----DOWN
#define AD1KEY5      28  //ENTER
#define AD1KEY6      28  //ENTER

#define AD2KEY1      103//DPAD_UP      -----UP
#define AD2KEY2      106//DPAD_RIGHT-----FFD

```

```

#define AD2KEY3      108//DPAD_DOWN-----DOWN
#define AD2KEY4      59  //MENU
#define AD2KEY5      158 //BACK-----ESC
#define AD2KEY6      116 //POWER

```

```

#endif

```

```

#define Valuedrift      70
#define EmptyADValue    950  //1000
#define InvalidADValue  10
#define ADKEYNum        12

```

```

/*power event button*/

```

```

#define POWER          116
#define ENDCALL         62
#define ONESEC_TIMES   100
#define SEC_NUM         1
#define SLEEP_TIME     2   /*per 40ms*/

```

```

static unsigned int pwrscantimes = 0;

```

```

static unsigned int valuecount = 0;

```

```

static unsigned int g_code = 0;

```

```

static unsigned int g_wake = 0;

```

```

extern int rk28_pm_status ;

```

```

#define KEY_PHYS_NAME  "rk28_AD_button/input0"

```

```

//ADC Registers

```

```

typedef struct tagADC_keyst

```

```

{
    unsigned int adc_value;
    unsigned int adc_keycode;
}ADC_keyst,*pADC_keyst;

```

```

//  adc  ---> key

```

```

static ADC_keyst ad1valuetab[] = {
    { 95,AD1KEY1},
    {247,AD1KEY2},
    {403,AD1KEY3},
    {561,AD1KEY4},
    {723,AD1KEY5},
    {899,AD1KEY6},
    {EmptyADValue,0}
};

```

```

static ADC_keyst ad2valuetab[] = {
    {95, AD2KEY1},
    {249, AD2KEY2},
    {406, AD2KEY3},

```

```

        {561, AD2KEY4},
        {726, AD2KEY5},
        {899, AD2KEY6},
        {EmptyADValue, 0}

};

//key code tab
static unsigned char initkey_code[ ] =
{
    AD1KEY1, AD1KEY2, AD1KEY3, AD1KEY4, AD1KEY5, AD1KEY6,
    AD2KEY1,
    AD2KEY2, AD2KEY3, AD2KEY4, AD2KEY5, AD2KEY6, ENDCALL, KEY_WAKEUP
};

struct rk28_AD_button_platform_data {
    int x;

};

struct rk28_AD_button {
    struct rk28_AD_button_platform_data *pdata;
    struct timer_list timer;

    struct clk *clk;
    struct input_dev *input_dev;
    void __iomem *mmio_base;
    /* matrix key code map */
    unsigned char keycodes[13];
    /* state row bits of each column scan */
    uint32_t direct_key_state;
    unsigned int direct_key_mask;
    int rotary_rel_code[2];
    int rotary_up_key[2];
    int rotary_down_key[2];
};

struct rk28_AD_button *prockAD_button;
extern void RockAdcScanning(void);
extern void printADCValue(void);
extern int32 ADCInit(void);
extern int get_rock_adc1(void);
extern int get_rock_adc2(void);

unsigned int find_rock_adkeycode(unsigned int advalue, pADC_keyst ptab)

```

```

{
    while(ptab->adc_value!=EmptyADValue)
    {
        if((advalue>ptab->adc_value-Valuedrift)&&(advalue<ptab->adc_value+Valuedrift))
            return ptab->adc_keycode;
        ptab++;
    }

    return 0;
}

```

```

static int rk28_AD_button_open(struct input_dev *dev)
{
    // struct rk28_AD_button *AD_button = input_get_drvdata(dev);

    return 0;
}

```

```

static void rk28_AD_button_close(struct input_dev *dev)
{
    // struct rk28_AD_button *AD_button = input_get_drvdata(dev);

}

```

```

#define res_size(res)((res)->end - (res)->start + 1)

```

```

int keydata=58;
static int ADSampleTimes = 0;
/*定时器扫描函数*/
static void rk28_adkeyscan_timer(unsigned long data)
{
    unsigned int ADKEY1,code = 0;
    /*Enable AD controller to sample */
    prockAD_button->timer.expires = jiffies+msecs_to_jiffies(10);
    add_timer(&prockAD_button->timer);
    RockAdcScanning();
    if (ADSampleTimes < 4)
    {
        ADSampleTimes ++;
        goto scan_io_key; /* scan gpio button event*/
    }
    ADSampleTimes = 0;
}

```

```

/*Get button value*/
ADKEY1=get_rock_adc2();
if((ADKEY1>EmptyADValue)&&(ADKEY1<=InvalidADValue))
    goto scan_io_key1;
valuecount++;
if(valuecount < 2)
    goto scan_code;
code=find_rock_adkeycode(ADKEY1,ad2valuetab);
valuecount = 2;
goto scan_code;  ///scan_code;
scan_io_key1:
    valuecount = 0;

scan_code:
    if((g_code == 0) && (code == 0)){
        goto scan_io_key;
    }
    DBG("\n key button PE2 == %d  \n",GPIOGetPinLevel(GPIOPortE_Pin2));
    if(code != 0){
        if(valuecount<2)
            goto scan_io_key;
        if(g_code == 0){
            g_code = code;
            DBG("\n  %s::%d  rock  adc1  key  scan  ,find  press  down  a  key=%d
\n",__func__,__LINE__,g_code);
            input_report_key(prockAD_button->input_dev,g_code,1);
            input_sync(prockAD_button->input_dev);
            goto scan_io_key;
        }else{
            if(g_code != code){
                DBG("\n  %s::%d  rock  adc1  key  scan  ,find  press  up  a  key=%d
\n",__func__,__LINE__,g_code);
                input_report_key(prockAD_button->input_dev,g_code,0);
                input_sync(prockAD_button->input_dev);
                DBG("\n  %s::%d  rock  adc1  key  scan  ,find  press  down  a  key=%d
\n",__func__,__LINE__,code);
                input_report_key(prockAD_button->input_dev,code,1);
                input_sync(prockAD_button->input_dev);
                g_code = code;
                goto scan_io_key;
            }
        }
    }
}

```

```

        if((g_code != 0)&&(code == 0)&&(ADSampleTimes == 0)){
            DBG("\n  %s::%d  rock  adc1  key  scan  ,find  press  up  a  key=%d\n",__func__,__LINE__,g_code);
            input_report_key(prockAD_button->input_dev,g_code,0);
            input_sync(prockAD_button->input_dev);
            valuecount = 0;
            g_code = 0;
            goto scan_io_key;
        }
scan_io_key :
    if(!GPIOGetPinLevel(GPIOPortE_Pin2))
    {
        pwrscantimes += 1;
        if(pwrscantimes == (SEC_NUM * ONESEC_TIMES))
        {
            input_report_key(prockAD_button->input_dev,ENDCALL,1);
            input_sync(prockAD_button->input_dev);
            printk("the kernel come to power down!!!\n");
        }
        if(pwrscantimes == (SEC_NUM + 1)* ONESEC_TIMES))
        {
            pwrscantimes = 0;
            input_report_key(prockAD_button->input_dev,ENDCALL,0);
            input_sync(prockAD_button->input_dev);
            printk("the kernel come to power up!!!\n");
        }
        return ;
    }
    if( pwrscantimes > SLEEP_TIME)
    {
        pwrscantimes = 0;
        if(rk28_pm_status == 0)
        {
            if(g_wake == 1)    /*already wake up*/
            {
                g_wake = 0;
                return;
            }
            input_report_key(prockAD_button->input_dev,AD1KEY5,1);
            input_sync(prockAD_button->input_dev);
            input_report_key(prockAD_button->input_dev,AD1KEY5,0);
            input_sync(prockAD_button->input_dev);
        }
    }

```

```

    }
    rk28printf("\n%s^^^^Wake Up ^^^^^!!\n",__FUNCTION__);
}
}
void rk28_send_wakeup_key( void )
{
    input_report_key(prockAD_button->input_dev,KEY_WAKEUP,1);
    input_sync(prockAD_button->input_dev);
    input_report_key(prockAD_button->input_dev,KEY_WAKEUP,0);
    input_sync(prockAD_button->input_dev);
}
/*中断产生调用此函数*/
static irqreturn_t rk28_AD_irq_handler(s32 irq, void *dev_id)
{
    if( rk28_pm_status == 1)
    {
        /*用于给上层上报一个按键动作*/
        input_report_key(prockAD_button->input_dev,AD1KEY5,1);
        /*用来告诉上层，本次的事件已经完成了*/
        input_sync(prockAD_button->input_dev);
        input_report_key(prockAD_button->input_dev,AD1KEY5,0);
        input_sync(prockAD_button->input_dev);
        g_wake =1;
    }
    rk28printf("\n%s^^^^Wake Up ^^^^^!!\n",__FUNCTION__);
    return IRQ_HANDLED;
}

static int __devinit rk28_AD_button_probe(struct platform_device *pdev)
{
    struct rk28_AD_button *AD_button;
    struct input_dev *input_dev;
    int error,i;
    /*内存申请*/
    AD_button = kzalloc(sizeof(struct rk28_AD_button), GFP_KERNEL);
    /* Create and register the input driver. */
    input_dev = input_allocate_device();
    if (!input_dev || !AD_button) {
        dev_err(&pdev->dev, "failed to allocate input device\n");
        error = -ENOMEM;
        goto failed1;
    }
    int ret = request_irq(IRQ_NR_ADC, rk28_AD_irq_handler, 0, "ADC", NULL);

```

```

if (ret < 0) {
    printk(KERN_CRIT "Can't register IRQ for ADC\n");
    return ret;
}
memcpy(AD_button->keycodes, initkey_code, sizeof(AD_button->keycodes));
input_dev->name = pdev->name;
input_dev->open = rk28_AD_button_open;
input_dev->close = rk28_AD_button_close;
input_dev->dev.parent = &pdev->dev;
input_dev->phys = KEY_PHYS_NAME;
input_dev->id.vendor = 0x0001;
input_dev->id.product = 0x0001;
input_dev->id.version = 0x0100;
input_dev->keycode = AD_button->keycodes;
input_dev->keycodesize = sizeof(unsigned char);
input_dev->keycodemax = ARRAY_SIZE(initkey_code);
for (i = 0; i < ARRAY_SIZE(initkey_code); i++)
    set_bit(initkey_code[i], input_dev->keybit);
/*
 *set_bit(EV_KEY, button_dev.evbit);
 *set_bit(BTN_0, button_dev.keybit)
 *分别用来设置设备所产生的事件以及上报的按键值。
 *Struct input_dev 中有两个成员，一个是 evbit.一个是 keybit.
 *分别用表示设备所支持的动作和按键类型。
 */
clear_bit(0, input_dev->keybit);
AD_button->input_dev = input_dev;
input_set_drvdata(input_dev, AD_button);
input_dev->evbit[0] = BIT_MASK(EV_KEY);
platform_set_drvdata(pdev, AD_button);
ADCInit();
prockAD_button=AD_button;
/* Register the input device 用来注册一个 input device.*/
error = input_register_device(input_dev);
if (error) {
    dev_err(&pdev->dev, "failed to register input device\n");
    goto failed2;
}
/*定时器初始化*/
setup_timer(&AD_button->timer, rk28_adkeyscan_timer, (unsigned long)AD_button);
AD_button->timer.expires = jiffies + 3;
/*启用定时器*/
add_timer(&AD_button->timer);
/*注册中断回调函数*/

```



```

error = request_gpio_irq(GPIOPortE_Pin2,rk28_AD_irq_handler,GPIODgeFalling,NULL);
if(error)
{
    printk("unable to request recover key IRQ\n");
    goto failed2;
}
return 0;
failed2:
    input_unregister_device(AD_button->input_dev);
    platform_set_drvdata(pdev, NULL);
failed1:
    input_free_device(input_dev);
    kfree(AD_button);
    return error;
}

static int __devexit rk28_AD_button_remove(struct platform_device *pdev)
{
    struct rk28_AD_button *AD_button = platform_get_drvdata(pdev);

    input_unregister_device(AD_button->input_dev);
    input_free_device(AD_button->input_dev);
    kfree(AD_button);
    platform_set_drvdata(pdev, NULL);
    return 0;
}

static struct platform_driver rk28_AD_button_driver = {
    .probe      = rk28_AD_button_probe,
    .remove     = __devexit_p(rk28_AD_button_remove),
    .driver     = {
        .name     = "rk28_AD_button",
        .owner    = THIS_MODULE,
    },
};

int __init rk28_AD_button_init(void)
{
    /*设备驱动注册*/
    return platform_driver_register(&rk28_AD_button_driver);
}

static void __exit rk28_AD_button_exit(void)

```

```

{
    platform_driver_unregister(&rk28_AD_button_driver);
}
/*模块化*/
module_init(rk28_AD_button_init);
module_exit(rk28_AD_button_exit);

MODULE_DESCRIPTION("rk28 AD button Controller Driver");
MODULE_LICENSE("GPL");

```

三.Android Kernel 源代码分析

1.Platform Device and Driver

platform是一个虚拟总线，相比**PCI**，**USB**，它主要用于描述**SOC**上的资源。**platform**所描述的资源有一个共同点，就是在**CPU**总线直接取址。**platform**机制将设备本身的资源注册进内核，由内核统一管理，在驱动程序中使用这些资源时通过**platform device**提供的标准接口进行申请并使用。这样提高了驱动和资源管理的独立性，并且拥有较好的可移植性和安全性(这些标准接口是安全的)。

Platform机制的本身使用并不复杂，由两部分组成：**platform_device**和**platform_driver**。

通过**Platform**机制开发发底层驱动的大致流程为：

定义 **platform_device_register** 注册 **platform_device**

定义 **platform_driver_register** 注册 **platform_driver**。

- 确认的就是 2.6 内核定义的**platform**总线类型

```

struct bus_type platform_bus_type = { /*drivers/base/platform.c*/
    .name = "platform",
    .dev_attrs = platform_dev_attrs,
    .match = platform_match,
    .uevent = platform_uevent,
    .pm = PLATFORM_PM_OPS_PTR,
};

```

- 在 2.6 内核中 **platform** 设备用结构体 **platform_device** 来描述设备的资源信息，例如设备的地址，中断号等。

```

struct platform_device { /*include/linux/platform_device.h*/
    const char * name;
    int id;
    struct device dev;
    u32 num_resources;
    struct resource * resource;
};

```

- 该结构一个重要的元素是 **resource**，该元素存入了最为重要的设备资源信息

```

struct resource { /*include/linux/ioport.h*/
    resource_size_t start;
    /*表示资源的起始物理地址和终止物理地址。
    *它们确定了资源的范围，也即是一个闭区间[start,end]。
    */
};

```

```

resource_size_t end;
const char name;      /*指向此资源的名称。*/
                        /*描述此资源属性的标志,属性
                        *flags 是一个 unsigned long 类型的 32 位标志值,用以描述资源的属性。
                        *比如：资源的类型、是否只读、是否可缓存，以及是否已被占用等。
                        */
unsigned long flags;
/*指针 parent、sibling 和 child：分别为指向父亲、兄弟和子资源的指针*/
struct resource parent, sibling, child;

};

```

- 内核初始化过程

do_basic_setup()/*init/main.c 调用/driver/base 下各子系统初始化函数*/

->driver_init() /*drivers/base/init.c 初始化驱动模型*/

->platform_bus_init()/*drivers/base/platform.c

初始化/sys/bus/platform_bus 目录/

->...初始化 platform_bus(虚拟总线)

(1)设备向内核注册的时候

platform_device_register()

->platform_device_add()

->...内核把设备挂在虚拟的 platform_bus 下

(2)驱动注册的时候

platform_driver_register()

->driver_register()

->bus_add_driver()

->driver_attach()

->bus_for_each_dev()对每个挂在虚拟的 platform_bus 的设备
作

__driver_attach()

->driver_probe_device()

->drv->bus->match()==platform_match();

比较 strcmp(pdev->name,drv->name,BUS_ID_SIZE) ,

如果相符就调用 platform_drv_probe()->driver->probe() ,

如果 probe 成功则绑定该设备到该驱动.

init 执行/etc/rc.d 和启动内核外挂模块

2.linux input subsystem 架构分析

- 主要数据结构

Input Subsystem main data structure

数据结构	用途	定义位置	具体数据结构的分配和初始化
Struct input_dev	驱动层物理 Input 设备的基本数据结构	Input.h	通常在具体的设备驱动中分配和填充具体的设备结构
Struct Evdev Struct Mousedev Struct Keybdev...	Event Handler 层逻辑 Input 设备的数据结构	Evdev.c Mousedev.c Keybdev.c	Evdev.c/Mousedev.c ... 中分配
Struct Input_handler	Event Handler 的结构	Input.h	Event Handler 层，定义一个具体的 Event Handler。
Struct Input_handle	用来创建驱动层 Dev 和 Handler 链表的链表项结构	Input.h	Event Handler 层中分配，包含在 Evdev/Mousedev... 中。

```
struct input_dev { /*driver/input/input.h*/  
  
    void *private;  
  
    const char *name;  
    const char *phys;  
    const char *uniq;  
    struct input_id id;  
  
    unsigned long evbit[NBITS(EV_MAX)];  
    unsigned long keybit[NBITS(KEY_MAX)];  
    unsigned long relbit[NBITS(REL_MAX)];  
    unsigned long absbit[NBITS(ABS_MAX)];  
    unsigned long mscbit[NBITS(MSC_MAX)];  
    unsigned long ledbit[NBITS(LED_MAX)];  
    unsigned long sndbit[NBITS(SND_MAX)];  
    unsigned long ffbbit[NBITS(FF_MAX)];  
    unsigned long swbit[NBITS(SW_MAX)];  
  
    unsigned int keycodemax;  
    unsigned int keycodesize;  
    void *keycode;  
    int (*setkeycode)(struct input_dev *dev, int scancode, int keycode);
```

```

int (*getkeycode)(struct input_dev *dev, int scancode, int *keycode);

struct ff_device *ff;

unsigned int repeat_key;
struct timer_list timer;

0 int state;

int sync;

int abs[ABS_MAX + 1];
int rep[REP_MAX + 1];

unsigned long key[NBITS(KEY_MAX)];
unsigned long led[NBITS(LED_MAX)];
unsigned long snd[NBITS(SND_MAX)];
unsigned long sw[NBITS(SW_MAX)];

int absmax[ABS_MAX + 1];
int absmin[ABS_MAX + 1];
int absfuzz[ABS_MAX + 1];
int absflat[ABS_MAX + 1];

int (*open)(struct input_dev *dev);
void (*close)(struct input_dev *dev);
int (*flush)(struct input_dev *dev, struct file *file);
int (*event)(struct input_dev *dev, unsigned int type, unsigned int code,
int value);

struct input_handle *grab;

struct mutex mutex;
unsigned int users;

struct device dev;
union {
    struct device *dev;
} cdev;

struct list_head h_list; /* 是 input_handle 链表的 list 节点*/
struct list_head node;
};

struct input_handle {
    void *private;

    int open;
    const char *name;

    struct input_dev *dev;
    struct input_handler *handler;

```

```

    struct list_head d_node;
    struct list_head h_node;
};

struct input_handler {
    void *private;

    void (*event)(struct input_handle *handle, unsigned int type,
                  unsigned int code, int value);
    int (*connect)(struct input_handler *handler, struct input_dev *dev,
                  const struct input_device_id *id);
    void (*disconnect)(struct input_handle *handle);
    void (*start)(struct input_handle *handle);

    const struct file_operations *fops;
    int minor;
    const char *name;

    const struct input_device_id *id_table;
    const struct input_device_id *blacklist;

    struct list_head    h_list;
    struct list_head    node;
};

```

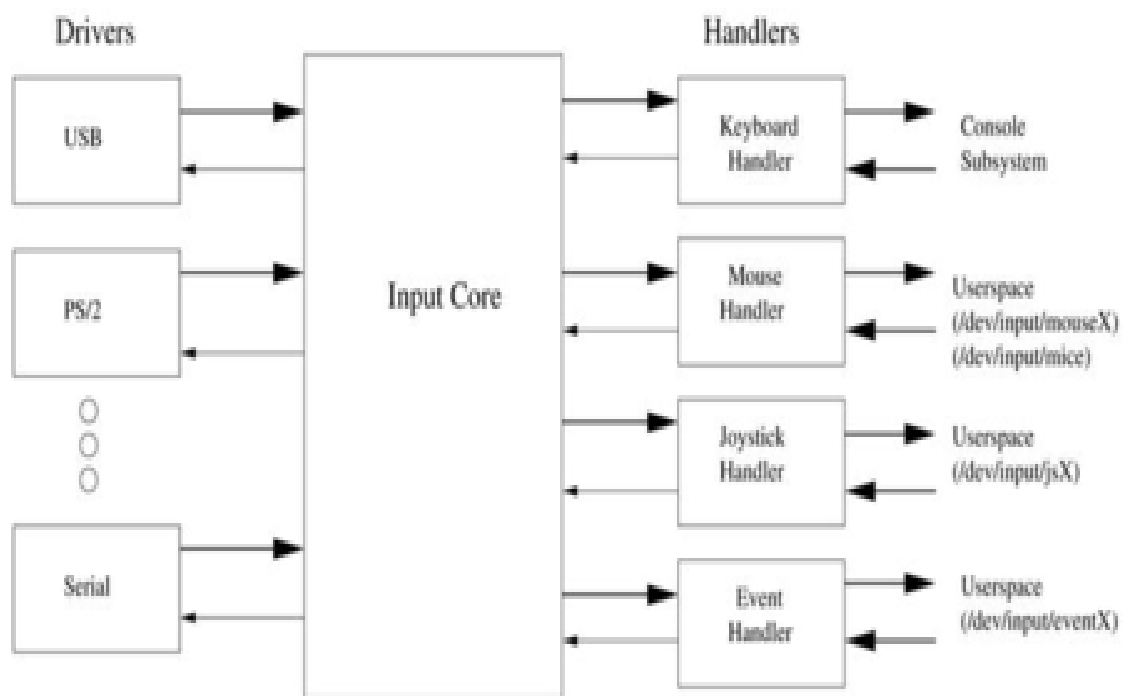
- 架构模型

input subsystem 用来统一处理数据输入设备，例如键盘，鼠标，游戏杆，触摸屏等等。

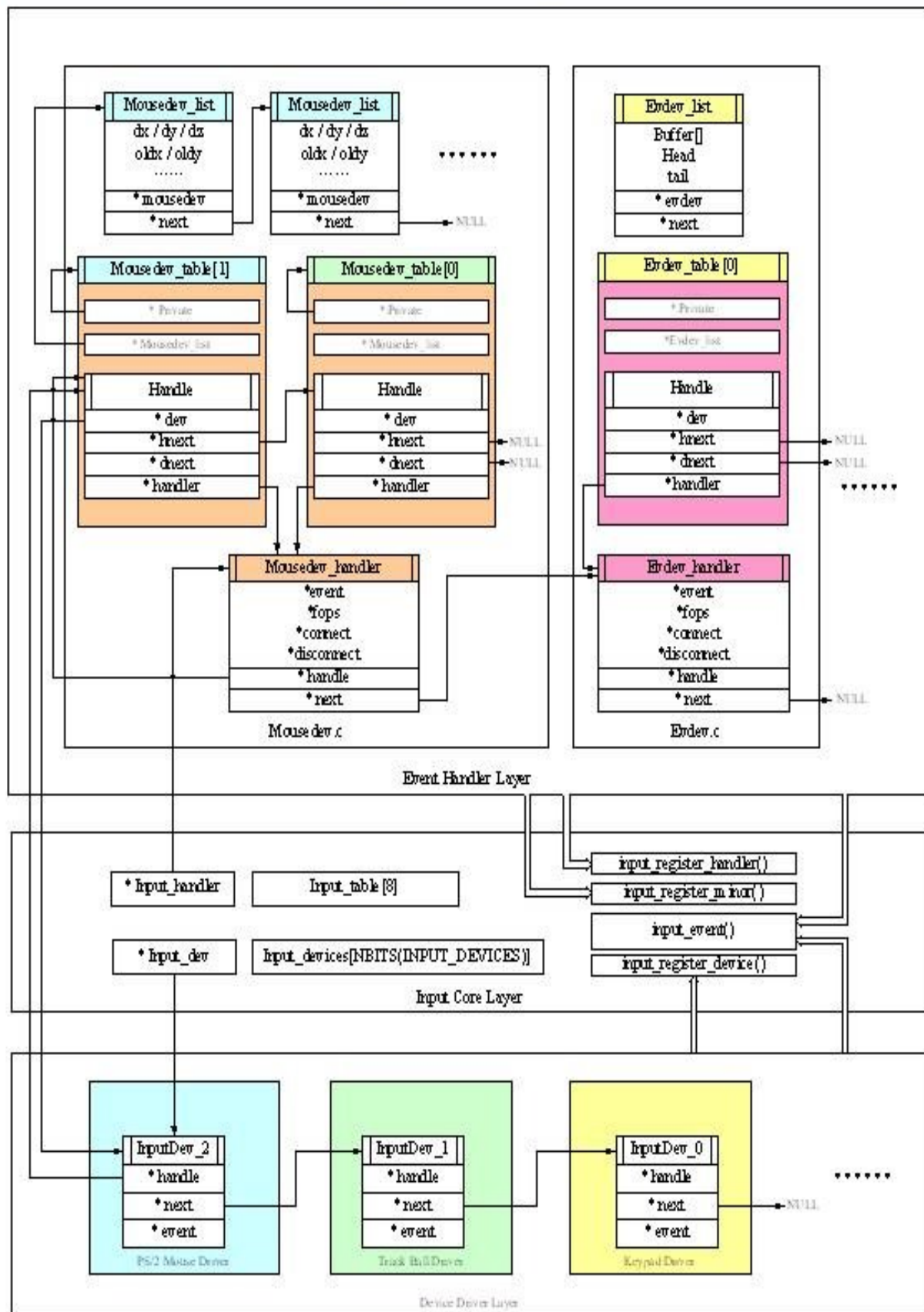
输入子系统由输入子系统核心层（ Input Core ），驱动层和事件处理层（ Event Handler ）三部份组成。从下图我们可以看到，input core 用来协调硬件的 input 事件 和 用户层应用之间的通讯。

一个输入事件，如鼠标移动，键盘按键按下，joystick 的移动等等通过 Driver -> InputCore -> Eventhandler -> userspace 的顺序到达用户空间传给应用程序。

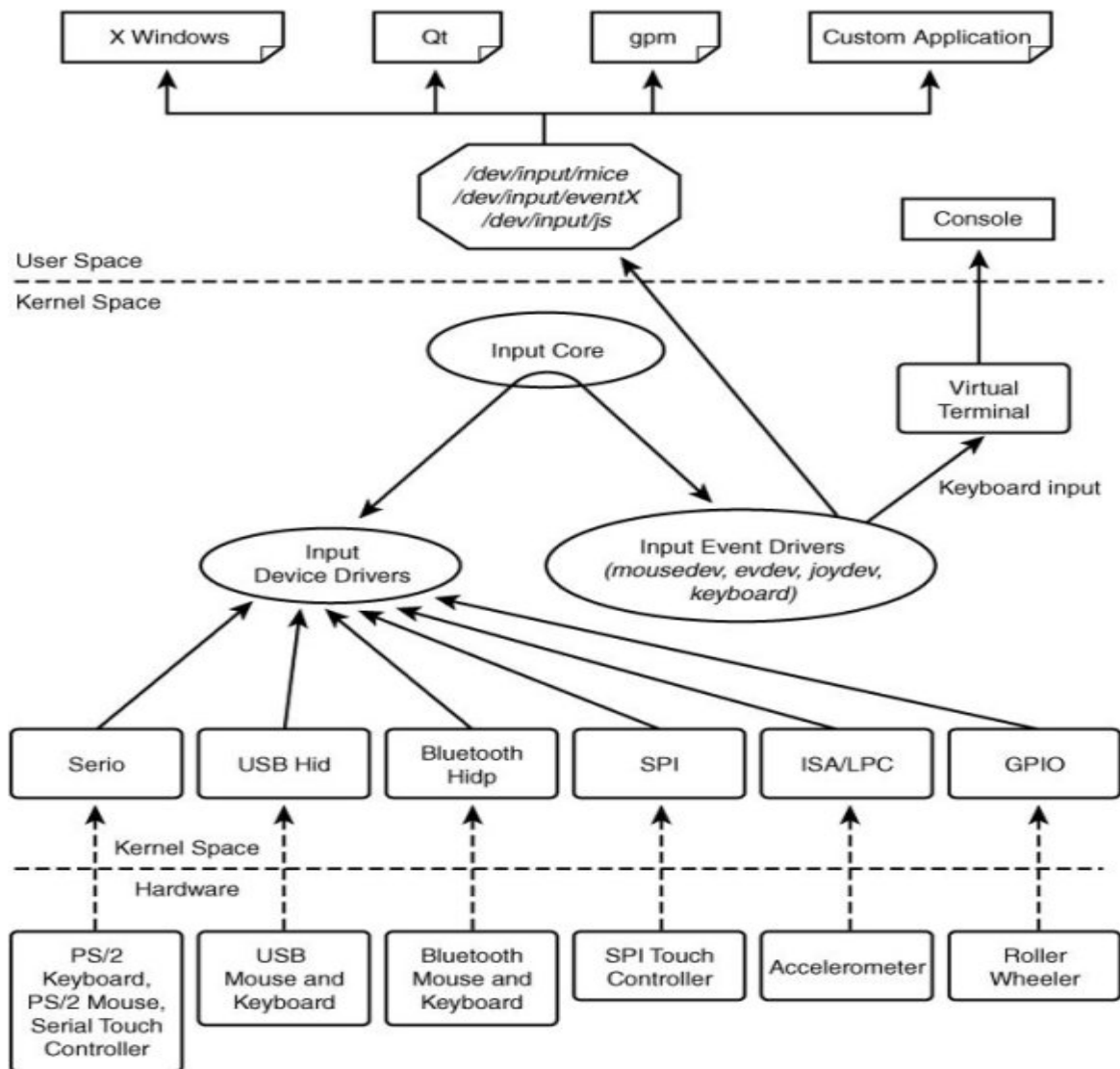
（其中 Input Core 即 Input Layer 由 driver/input/input.c 及相关头文件实现。对下提供了设备驱动的接口，对上提供了 Event Handler 层的编程接口。）



- 输入子系统架构示意图

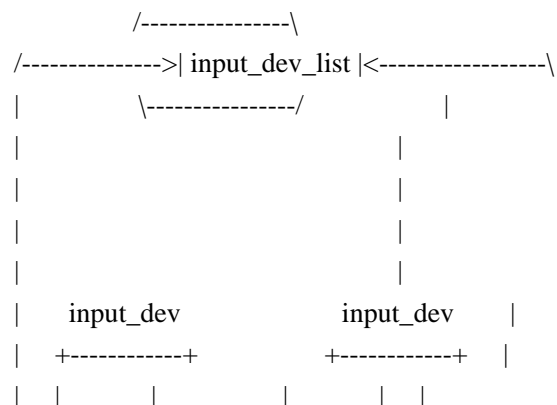


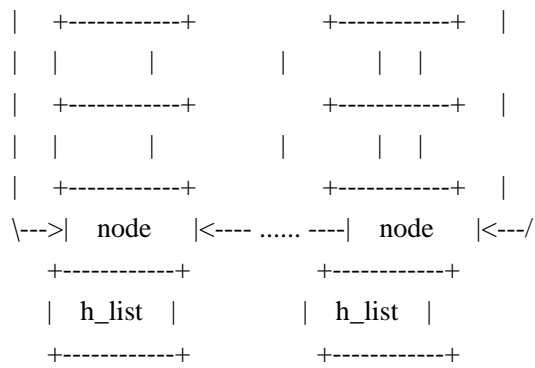
- 内部结构框架



(1) 在内核中，input_dev 表示一个 input 设备；

所有的 input_dev 用双向链表 input_dev_list 连起来，如图所示：

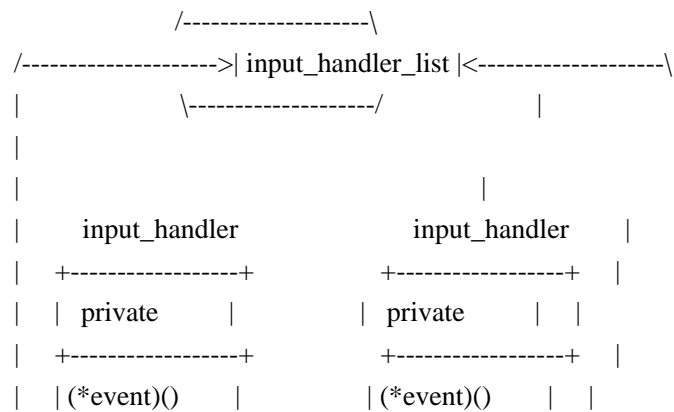


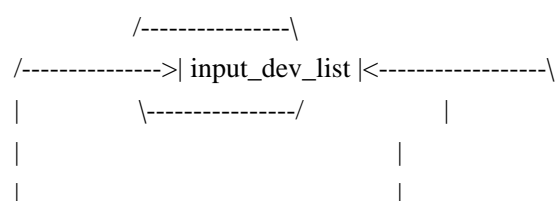


在调用 `int input_register_device(struct input_dev *dev)` 的时候，会将新的 `input_dev` 加入到这个链表中。

(2) input_handler 来表示 input 设备的 interface。

所有的 `input_handler` 用双向链表 `input_handler_list` 连起来，如图所示：






```

| |(*disconnect)() | |
| +-----+ |
| |(*start)() | |
| +-----+ |
| | fops | |
| +-----+ |
| | minor | |
| +-----+ |
| | name | |
| +-----+ |
| | id_table | |
| +-----+ |
| | blacklist | |
| +-----+ |
|-->| hlist |<---/
| +-----+
| | node |
| +-----+

```

这里需要额外说明一下的是：input_dev 中的 h_node 是 input_handle 链表的 list 节点，也就是说，一个 input_dev，可以对应多个 input_handle。

当设备产生 input event 的时候，例如按下了一个键，驱动就会调用 input_handler 中的 event 函数，同时，如果 input_dev 支持的话，也会调用 input_dev 的 event 函数。这样，设备产生的事件就会被驱动记录下来。

当用户层的程序需要获知这些事件的时候，会调用 input_handler 中的 struct file_operations *fops 中的相应函数，例如 read 等等。

可以看出，整个 input 框架的构思还是比较简洁方便的。

● 输入子系统源代码解析

输入链路的创建过程：由于 input 子系统通过分层将一个输入设备的输入过程分隔为独立的两部份：驱动到 Input Core，Input Core 到 Event Handler。所以整个链路的这两部分的接口的创建是独立的。

(1) 硬件设备的注册

驱动层任务：负责和底层的硬件设备打交道，将底层硬件对用户输入的响应转换为标准的输入事件以后再向上发送给 Input Core。

驱动层通过调用 Input_register_device 函数和 Input_unregister_device 函数来向输入子系统中注册和注销输入设备。

这两个函数调用的参数是一个 Input_dev 结构，这个结构在 driver/input/input.h 中定义。驱动

层在调用 Input_register_device 之前需要填充该结构中的部分字段.

```
set_bit(EV_KEY, input_dev.evbit);
set_bit(KEY_B, input_dev.keybit);
set_bit(KEY_A, input_dev.keybit);
```

*分别用来设置设备所产生的事件以及上报的按键值。

*Struct input_dev 中有两个成员，一个是 evbit.一个是 keybit.

*分别用表示设备所支持的动作和按键类型。

evbit 字段用来定义该输入设备可以支持的（产生和响应）的事件的类型。

包括：

```
Ø EV_RST    0x00  Reset
Ø EV_KEY    0x01  按键
Ø EV_REL    0x02  相对坐标
Ø EV_ABS    0x03  绝对坐标
Ø EV_MSC    0x04  其它
Ø EV_LED    0x11  LED
Ø EV_SND    0x12  声音
Ø EV_REP    0x14  Repeat
Ø EV_FF     0x15  力反馈
```

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，比如 EV_KEY 事件，支持哪些按键等。

(2)Event Handler 层

● 注册 Input Handler

驱动层只是把输入设备注册到输入子系统中，在驱动层的代码中本身并不创建设备结点。

应用程序用来与设备打交道的设备结点的创建由 Event Handler 层调用 Input core 中的函数来实现。而在创建具体的设备节点之前，Event Handler 层需要先注册一类设备的输入事件处理函数及相关接口

以 MouseDev Handler 为例：/*drivers/input/mousedev.c*/

```
static struct input_handler mousedev_handler = {

    .event =          mousedev_event,

    .connect =        mousedev_connect,

    .disconnect =     mousedev_disconnect,

    .fops =           &mousedev_fops,

    .minor =          MOUSEDEV_MINOR_BASE,

    .name =           "mousedev",

    .id_table =        mousedev_ids,
```

```

};

static int __init mousedev_init(void)
{
    /*调用 input.c 中定义的 input_register_handler 来注册一个鼠标类型的 Handler. 这里的 Handler 不是具体的用户可以操作的设备, 而是鼠标类设备的统一的处理函数接口. */

    int error;

    mousedev_mix = mousedev_create(NULL, &mousedev_handler, MOUSEDEV_MIX);

    if (IS_ERR(mousedev_mix))

        return PTR_ERR(mousedev_mix);

    error = input_register_handler(&mousedev_handler);

    if (error) {

        mousedev_destroy(mousedev_mix);

        return error;

    }

#ifdef CONFIG_INPUT_MOUSEDEV_PSAUX

    error = misc_register(&psaux_mouse);

    if (error)

        printk(KERN_WARNING "mice: could not register psaux device, "
            "error: %d\n", error);

    else

        psaux_registered = 1;

#endif

    printk(KERN_INFO "mice: PS/2 mouse device common for all mice\n");

    return 0;

}

```

- 注册 input device

注册 input device 的过程就是为 input device 设置默认值, 并将其挂以 input_dev_list. 与挂载在 input_handler_list 中的 handler 相匹配。如果匹配成功, 就会调用 handler 的 connect 函数. connect 函数中, 就会创建 input_handle, 而 input_handle 就是负责将 input_dev 和 input_handler 联系在一起。

```

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;

```

在 前面的分析中曾分析过。Input_device 的 evbit 表示该设备所支持的事件。在这里将其 EV_SYN 置位，即所有设备都支持这个事件。如果 dev->rep[REP_DELAY] 和 dev->rep[REP_PERIOD]没有设值，则将其赋默认值。这主要是处理重复按键的。

```

__set_bit(EV_SYN, dev->evbit);
/*
 * 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;
}

```

如 果 input device 没有定义 getkeycode 和 setkeycode,则将其赋默认值。还记得在键盘驱动中的分析吗?这两个操作函数就可以用来取键的扫描码 和设置键的扫描码。然后调用 device_add()将 input_dev 中封装的 device 注册到 sysfs

```

if (!dev->getkeycode)
    dev->getkeycode = input_default_getkeycode;
if (!dev->setkeycode)
    dev->setkeycode = input_default_setkeycode;
snprintf(dev->dev.bus_id, sizeof(dev->dev.bus_id),
         "input%d", (unsigned long) atomic_inc_return(&input_no) - 1);
error = device_add(&dev->dev);
if (error)
    return error;
path = kobject_get_path(&dev->dev.kobj, GFP_KERNEL);
printk(KERN_INFO "input: %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;
}

```

这 里就是重点了。将 input device 挂到 input_dev_list 链表上,然后，对每一个挂在 input_handler_list 的 handler 调用 input_attach_handler()在这里的情况有好比设备模型中的 device 和 driver 的匹配。所有的 input device 都挂在 input_dev_list 链上。所有的 handle 都挂在 input_handler_list 上。


```

list_add_tail(&dev->node, &input_dev_list);
list_for_each_entry(handler, &input_handler_list, node)
input_attach_handler(dev, handler);
input_wakeup_procfs_readers();
mutex_unlock(&input_mutex);
return 0;
}

```

```

static int input_attach_handler(struct input_dev *dev,
                                struct input_handler *handler)
{

```

```

    const struct input_device_id *id;
    int error;

```

如果 handle 的 blacklist 被赋值。要先匹配 blacklist 中的数据跟 dev->id 的数据是否匹配。匹配成功后再来匹配 handler->id 和 dev->id 中的数据。如果匹配成功，则调用 handler->connect()

```

    if (handler->blacklist && input_match_device(handler->blacklist, dev))
        return -ENODEV;
    id = input_match_device(handler->id_table, dev);
    if (!id)
        return -ENODEV;
    error = handler->connect(handler, dev, id);
    if (error && error != -ENODEV)
        printk(KERN_ERR "input: failed to attach handler %s to device %s, "
                "error: %d\n", handler->name, kobject_name(&dev->dev.kobj), error);
    return error;
}

```

```

static const struct input_device_id *input_match_device(const struct
                                                         input_device_id *id, struct input_dev *dev)
{

```

如果 id->flags 定义的类型匹配成功。或者是 id->flags 没有定义，就会进入到 MATCH_BIT 的匹配项了从 MATCH_BIT 宏的定义可以看出。只有当 input device 和 input handler 的 id 成员在 evbit, keybit,... swbit 项相同才会匹配成功。而且匹配的顺序是从 evbit, keybit 到 swbit。只要有一项不同，就会循环到 id 中的下一项进行比较。

```

int i;
for (; id->flags || id->driver_info; id++) {

```

在 id->flags 中定义了要匹配的项。定义 INPUT_DEVICE_ID_MATCH_BUS。则是要比较 input device 和 input handler 的总线类型。

```

    if (id->flags & INPUT_DEVICE_ID_MATCH_BUS)
        if (id->bustype != dev->id.bustype)
            continue;

```

INPUT_DEVICE_ID_MATCH_VENDOR , INPUT_DEVICE_ID_MATCH_PRODUCT , INPUT_DEVICE_ID_MATCH_VERSION 分别要求设备厂商。设备号和设备版本

```

        if (id->flags & INPUT_DEVICE_ID_MATCH_VENDOR)
            if (id->vendor != dev->id.vendor)
                continue;
        if (id->flags & INPUT_DEVICE_ID_MATCH_PRODUCT)
            if (id->product != dev->id.product)
                continue;
        if (id->flags & INPUT_DEVICE_ID_MATCH_VERSION)
            if (id->version != dev->id.version)
                continue;
        MATCH_BIT(evbit, EV_MAX);
        MATCH_BIT(, 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);
        return id;
    }
    return NULL;
}

```

MATCH_BIT 宏的定义如下：

```

#define MATCH_BIT(bit, max) \
    for (i = 0; i < BITS_TO_LONGS(max); i++) \
        if ((id->bit[i] & dev->bit[i]) != id->bit[i]) \
            break; \
    if (i != BITS_TO_LONGS(max)) \
        continue;

```

此外如果已经注册了一些硬件设备，此后再注册一类新的 Input Handler，则同样会对所有已注册的 Device 调用新的 Input Handler 的 Connect 函数**确定是否需要创建新的设备节点**：

```

int input_register_handler(struct input_handler *handler)
{
    struct input_dev *dev;
    int retval;
    retval = mutex_lock_interruptible(&input_mutex);
    if (retval)
        return retval;
    INIT_LIST_HEAD(&handler->h_list);
    if (handler->fops != NULL) {

```

handler->minor 表示对应 input 设备节点的次设备号.以 handler->minor 右移五位做为索引值插入到 input_table[]中..之后再来分析 input_table[]的作用.

```

        if (input_table[handler->minor >> 5]) {

```

```

        retval = -EBUSY;
        goto out;
    }
    input_table[handler->minor >> 5] = handler;
}

将 handler 挂到 input_handler_list 中
list_add_tail(&handler->node, &input_handler_list);
将其与挂在 input_dev_list 中的 input device 匹配.这个过程和 input device 的注册有相似的地方.都是注册到各自的链表,.然后与另外一条链表的对象相匹配.
list_for_each_entry(dev, &input_dev_list, node)
                                input_attach_handler(dev, handler);

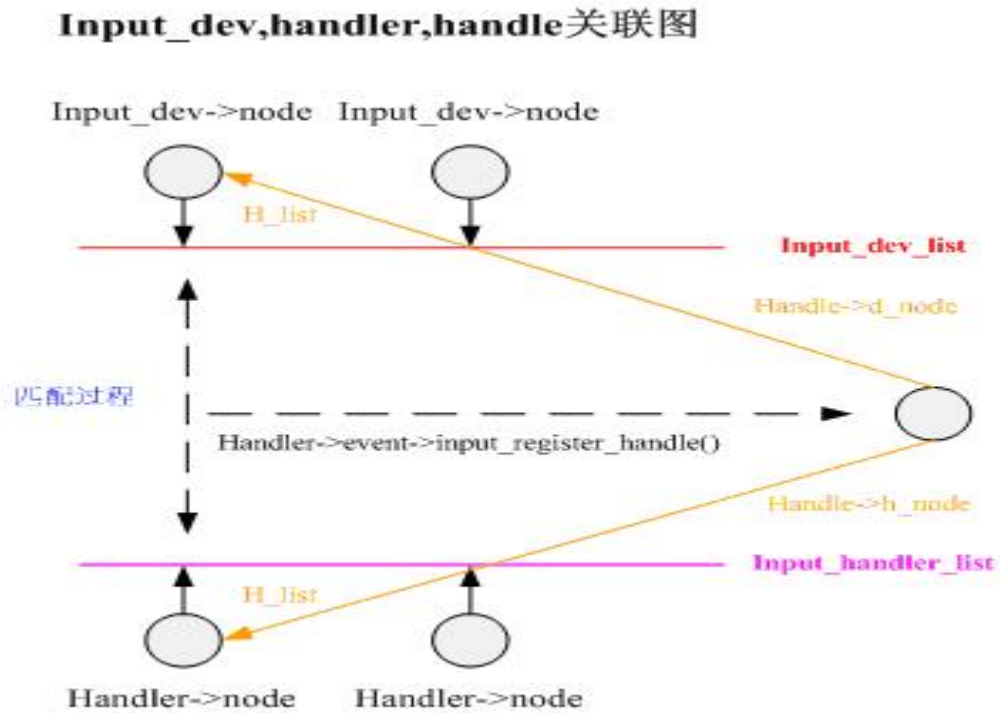
input_wakeup_procfs_readers();
out:
mutex_unlock(&input_mutex);
return retval;
}

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;
    list_add_tail_rcu(&handle->d_node, &dev->h_list);
    mutex_unlock(&dev->mutex);
    synchronize_rcu();
    /*
     * 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 handle 挂到所对应 input device 的 h_list 链表上.还将 handle 挂到对应的 handler 的 hlist 链表上.如果 handler 定义了 start 函数,将调用之.
    list_add_tail(&handle->h_node, &handler->h_list);
    if (handler->start)
        handler->start(handle);
    return 0;
}

```

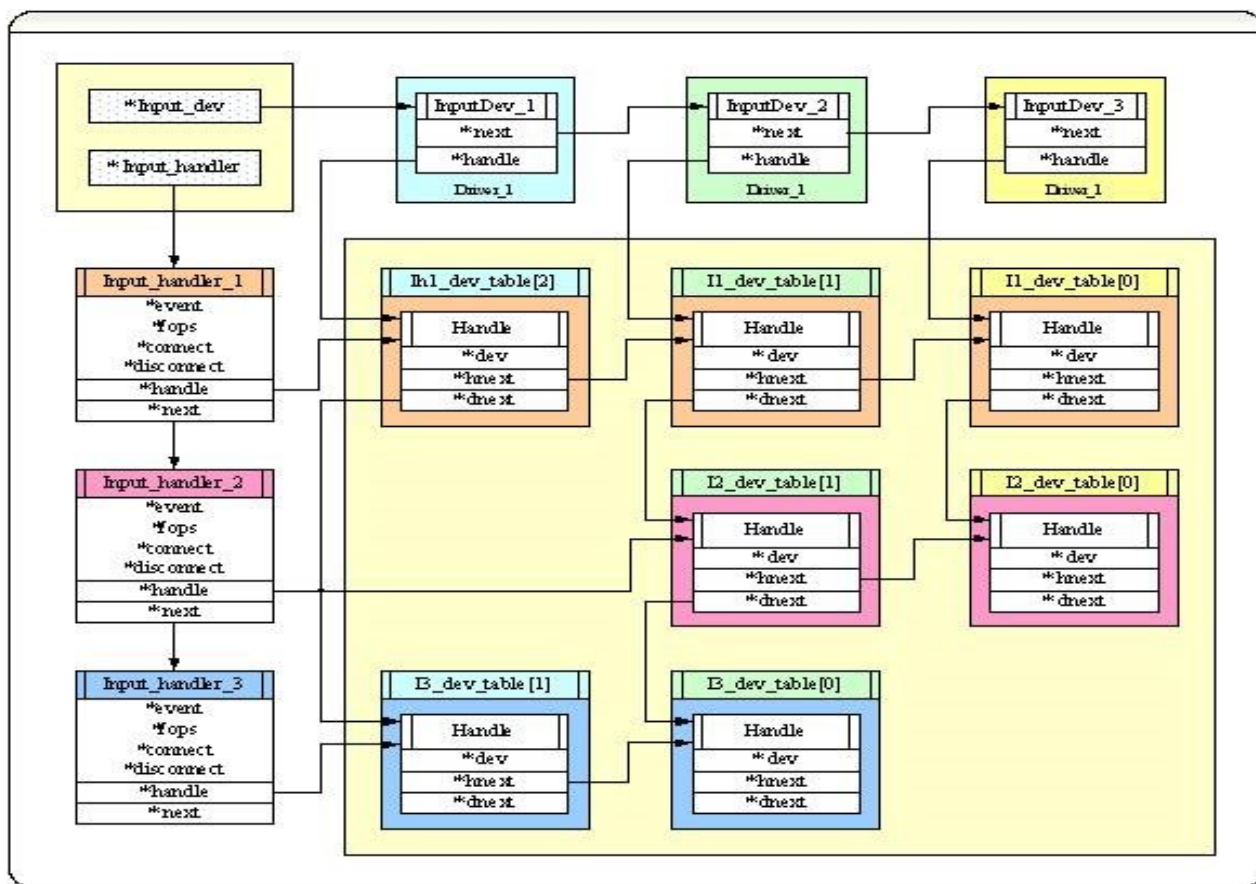
}

我们已经看到了 input device, handler 和 handle 是怎么关联起来的了.以图的方式总结如下：



从上面的分析中可以看到一类 Input Handler 可以和多个硬件设备相关联，创建多个设备节点。而一个设备也可能与多个 Input Handler 相关联，创建多个设备节点。

直观起见，物理设备，Input Handler，逻辑设备之间的多对多关系可见下图：



物理设备，Input Handler，逻辑设备关系图

设备的打开和读写

用户程序通过 Input Handler 层创建的设备节点的 Open，read，write 等函数打开和读写输入设备。

• Open

设备节点的 Open 函数，首先会调用一类具体的 Input Handler 的 Open 函数，处理一些和该类型设备相关的通用事务，比如初始化事件缓冲区等。然后通过 Input.c 中的 input_open_device 函数调用驱动层中具体硬件设备的 Open 函数。

• Read

大多数 Input Handler 的 Read 函数等待在 Event Layer 层逻辑设备的 wait 队列上。当设备驱动程序通过调用 Input_event 函数将输入以事件的形式通知给输入子系统的时候，相关的 Input Handler 的 event 函数被调用，该 event 函数填充事件缓冲区后将等待队列唤醒。

在驱动层中，读取设备输入的一种可能的实现机制是扫描输入的函数睡眠在驱动设备的等待队列上，在设备驱动的中断函数中唤醒等待队列，而后扫描输入函数将设备输入包装成事件的形式通知给输入子系统。

• Write

2.6 内核的代码中，通过调用 `Input_event` 将写入的数据以事件的形式再次通知给输入子系统，而后在 `Input.c` 中根据事件的类型，将需要反馈给物理设备的事件通过调用物理设备的 `Event` 函数传给设备驱动处理，如 `EV_LED` 事件：

```
void input_event(struct input_dev *dev,
                 unsigned int type, unsigned int code, int value)
{
    unsigned long flags;
    //判断设备是否支持这类事件
    if (is_event_supported(type, dev->evbit, EV_MAX)) {
        spin_lock_irqsave(&dev->event_lock, flags);
        //利用键盘输入来调整随机数产生器
        add_input_randomness(type, code, value);
        input_handle_event(dev, type, code, value);
        spin_unlock_irqrestore(&dev->event_lock, flags);
    }
}
```

首先,先判断设备产生的这个事件是否合法.如果合法,流程转入到 `input_handle_event()` 中.代码如下:

```
static void input_handle_event(struct input_dev *dev,
                              unsigned int type, unsigned int code, int value)
{
    int disposition = INPUT_IGNORE_EVENT;
    switch (type) {
        case EV_SYN:
            switch (code) {
                case SYN_CONFIG:
                    disposition = INPUT_PASS_TO_ALL;
                    break;
                case SYN_REPORT:
                    if (!dev->sync) {
                        dev->sync = 1;
                        disposition = INPUT_PASS_TO_HANDLERS;
                    }
                    break;
            }
            break;
        case EV_KEY:
            //判断按键值是否被支持
            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);
    }
    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)) {
        value = input_defuzz_abs_event(value,
            dev->abs[code], dev->absfuzz[code]);
        if (dev->abs[code] != value) {
            dev->abs[code] = value;
            disposition = INPUT_PASS_TO_HANDLERS;
        }
    }
    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 (type != EV_SYN)
        dev->sync = 0;
    if ((disposition & INPUT_PASS_TO_DEVICE) && dev->event)
        dev->event(dev, type, code, value);
    if (disposition & INPUT_PASS_TO_HANDLERS)
        input_pass_event (dev, type, code, value);
}

```

在 这里,我们忽略掉具体事件的处理.到最后,如果该事件需要 input device 来完成的,就会将 disposition 设置成 INPUT_PASS_TO_DEVICE.如果需要 handler 来完成的,就将 disposition 设为 INPUT_PASS_TO_DEVICE. 如 果 需 要 两 者 都 参 与 , 将 disposition 设置 为 INPUT_PASS_TO_ALL.

需要输入设备参与的,回调设备的 event 函数.如果需要 handler 参与的.调用 input_pass_event(). 代码如下:

```

static void input_pass_event(struct input_dev *dev,
                           unsigned int type, unsigned int code, int value)
{
    struct input_handle *handle;
    rcu_read_lock();
    handle = rcu_dereference(dev->grab);
    if (handle)
        handle->handler->event(handle, type, code, value);
    else
        list_for_each_entry_rcu(handle, &dev->h_list, d_node)
            if (handle->open)
                handle->handler->event(handle, type, code, value);
    rcu_read_unlock();
}

```

如果 input device 被强制指定了 handler,则调用该 handler 的 event 函数.

结合 handle 注册的分析.我们知道.会将 handle 挂到 input device 的 h_list 链表上.

如果没有为 input device 强制指定 handler.就会遍历 input device->h_list 上的 handle 成员.如果

该 handle 被打开,则调用与输入设备对应的 handler 的 event()函数.注意,只有在 handle 被打开的情况下才会接收到事件.

另外,输入设备的 handler 强制设置一般是用带 EVIOCGRAB 标志的 ioctl 来完成的.如下是发图的方示总结 evnet 的处理过程：