驱动学习整理

目录

[热插拔 3](#_Toc462932484)

[参考资料 6](#_Toc462932485)

[Sysfs以及设备模型 7](#_Toc462932486)

[内核空间与用户空间的映射关系 7](#_Toc462932487)

[设备模型底层容器 7](#_Toc462932488)

[struct kobj 7](#_Toc462932489)

[struct kobj\_type 8](#_Toc462932490)

[struct kset 8](#_Toc462932491)

[struct kobj\_attribute 8](#_Toc462932492)

[设备模型上层容器 9](#_Toc462932493)

[总线类型bus\_type 9](#_Toc462932494)

[Struct device 11](#_Toc462932495)

[Struct device\_driver 14](#_Toc462932496)

[从面向对象的角度 16](#_Toc462932497)

[Kobject，kset，ktype结构图 17](#_Toc462932498)

[Kobj type 19](#_Toc462932499)

[参考资料 20](#_Toc462932500)

[最小子系统+ramdisk+initrd+系统启动过程 20](#_Toc462932501)

[ramdisk+initrd 20](#_Toc462932502)

[BIOS(Basic Input Output System) 21](#_Toc462932503)

[系统引导 22](#_Toc462932504)

[MBR 22](#_Toc462932505)

[参考资料 26](#_Toc462932506)

[Paltform 27](#_Toc462932507)

[Platform设备驱动 27](#_Toc462932508)

[Platform模块的软件架构 27](#_Toc462932509)

[Platform模块向其它模块提供的API汇整 28](#_Toc462932510)

[总结大多数驱动框架 28](#_Toc462932511)

[Platform led驱动 28](#_Toc462932512)

[Platform input驱动 28](#_Toc462932513)

[Platform i2c驱动 29](#_Toc462932514)

[Platform spi驱动 29](#_Toc462932515)

[参考资料 30](#_Toc462932516)

[中断子系统 31](#_Toc462932517)

[中断 31](#_Toc462932518)

[IRQ编号 32](#_Toc462932519)

[中断处理程序 32](#_Toc462932520)

[上半部与下半部 33](#_Toc462932521)

[在驱动程序中申请中断 33](#_Toc462932522)

[flags:与中断相关的标志 35](#_Toc462932523)

[request\_threaded\_irq的工作流程 36](#_Toc462932524)

[下半部的几种机制 38](#_Toc462932525)

[软中断(softirqs) 38](#_Toc462932526)

[Tasklet 39](#_Toc462932527)

[work queue 39](#_Toc462932528)

[内核定时器 39](#_Toc462932529)

[参考资料 40](#_Toc462932530)

[Input子系统与多点触摸技术 41](#_Toc462932531)

[简介 41](#_Toc462932532)

[设备驱动层 42](#_Toc462932533)

[换个角度看input子系统 42](#_Toc462932534)

[软件设计流程 44](#_Toc462932535)

[与软件设计有关的API函数 44](#_Toc462932536)

[多点触摸技术A/B(Slot)协议 45](#_Toc462932537)

[A/B协议究竟是如何划分 45](#_Toc462932538)

[A协议实现方式 46](#_Toc462932539)

[B协议实现方式 47](#_Toc462932540)

[总结 48](#_Toc462932541)

[参考资料 48](#_Toc462932542)

[Android调试方法 50](#_Toc462932543)

[getevent，sendevent，input三命令 50](#_Toc462932544)

[Getevent 50](#_Toc462932545)

[Sendevent 53](#_Toc462932546)

[Input 53](#_Toc462932547)

[参考资料 55](#_Toc462932548)

[ARM概念梳理：Architecture, Core, CPU，SOC 56](#_Toc462932549)

[概念梳理 56](#_Toc462932550)

[ARM 64bit 56](#_Toc462932551)

[参考资料 57](#_Toc462932552)

[pmic与pmu的区别 57](#_Toc462932553)

[Linux中SD/MMC设备驱动流程 57](#_Toc462932554)

[SD（Secure Digital）与 MMC（Multimedia Card） 57](#_Toc462932555)

[SDIO（Secure Digital I/O） 58](#_Toc462932556)

[SD/SDIO/MMC/SPI之间的关系 58](#_Toc462932557)

[同一个控制器，支持SD/MMC/SDIO的所以卡，如何区分出现在卡槽里的是个什么卡呢？ 58](#_Toc462932558)

[SD/SDIO 的传输模式 59](#_Toc462932559)

[参考资料 60](#_Toc462932560)

# 热插拔

一个完整的热插拔系统包括热插拔系统的硬件，支持热插拔的软件和操作系统，支持热插拔的设备驱动程序和支持热插拔的用户接口。

**硬件支持：**插入->先数据，后电源；拔出->先电源，后数据(具体硬件机理不详)

**软件支持：**uevent, user space event. 内核与用户空间的一种通信机制

**Uevent**是**Kobject**的一部分，用于在Kobject状态发生改变时，例如增加、移除等，通知用户空间程序。用户空间程序收到这样的事件后，会做相应的处理。

该机制通常是用来支持热拔插设备的，例如U盘插入后，USB相关的驱动软件会动态创建用于表示该U盘的device结构（相应的也包括其中的kobject），并告知用户空间程序，为该U盘动态的创建/dev/目录下的设备节点，更进一步，可以通知其它的应用程序，将该U盘设备mount到系统中，从而动态的支持该设备。

**Uevent的机制：**设备模型中任何设备有事件需要上报时，会触发Uevent提供的接口。Uevent模块准备好上报事件的格式后，可以通过两个途径把事件上报到用户空间：一种是通过kmod模块，直接调用用户空间的可执行文件；另一种是通过netlink通信机制，将事件从内核空间传递给用户空间。

如图来自蜗居科技：http://www.wowotech.net/linux\_kenrel/uevent.html





上面所说的过程可以去除driver部分，简化如下：  
① 外设插入；  
② 总线发现（中断？ usb有中断hub\_irq，pci的中断没找到）新设备，调用device\_add，添加新设备到设备管理系统；  
③ device\_add中调用kobject\_uevent(, KOBJ\_ADD)，向userspace广播新设备加入event通知；这里发出通知的方式，就是netlink；  
④ 用户空间运行的daemon(udevd)收到event事件广播；udevd (udev 后台程序)  
⑤ udevd根据消息和环境变量，查询/sys的变化，按照规则(/etc/udev/rules.d/\*)，在/dev目录下自动创建设备节点；

## 参考资料

http://www.wowotech.net/linux\_kenrel/uevent.html 主要讲述的是头文件热插拔事件从内核空间到用户空间

http://blog.csdn.net/bingqingsuimeng/article/details/7924300 有源代码的讲解，比较清晰

http://www.cnblogs.com/image-eye/archive/2011/08/19/2145858.html 内核启动过程的驱动加载

http://blog.chinaunix.net/uid-25721104-id-3023525.html 讨论的关于热插拔

http://blog.csdn.net/zirconsdu/article/details/8792184 图解很清晰

http://blog.chinaunix.net/uid-27666459-id-5747129.html讲解call\_usermodehelper

# Sysfs以及设备模型

Sysfs被加载在 /sys/目录下,它的子目录包括

1）Block:在系统中发现的每个块设备在该目录下对应一个子目录。每个子目录中又包含一些属性文件,它们描述了这个块设备的各方面属性,如:设备大小。(loop块设备是使用文件来模拟的)

2）Bus:在内核中注册的每条总线在该目录下对应一个子目录,如: ide pci scsi usbpcmcia 其中每个总线目录内又包含两个子目录:devices和drivers ,devices目录包含了在整个系统中发现的属于该总线类型的设备,drivers目录包含了注册到该总线的所有驱动。

3）Class:将设备按照功能进行的分类,如/sys/class/net目录下包含了所有网络接口。

4）Devices:包含系统所有的设备。

5）Kernel:内核中的配置参数

6）Module:系统中所有模块的信息

7）Firmware:系统中的固件

8）Fs:描述系统中的文件系统

9）Power:系统中电源选项

## 内核空间与用户空间的映射关系

内核空间(internel) ——->用户空间(externel)  
内核对象(kernel objects) ——->目录(directories)  
对象属性(object attributes) ——->普通文件(regular files)  
对象关系(object relationships) ——->符号链接(symbolic links)

## 设备模型底层容器

### struct kobj

struct kobject {

const char \*name; //kobject的名称

struct list\_head entry; //kobject结构链表

struct kobject \*parent; //父kobject结构体

struct kset \*kset; //kset集合

struct kobj\_type \*ktype; //kobject的类型描述符

struct sysfs\_dirent \*sd; //sysfs文件目录

struct kref kref; //kobject引用计数

#ifdef CONFIG\_DEBUG\_KOBJECT\_RELEASE

struct [delayed\_work](http://lxr.free-electrons.com/ident?i=delayed_work) [release](http://lxr.free-electrons.com/ident?i=release);

#endif

unsigned int state\_initialized:1; //kobject是否初始化

unsigned int state\_in\_sysfs:1; //是否已经加入sysfs

unsigned int state\_add\_uevent\_sent:1;

unsigned int state\_remove\_uevent\_sent:1;

unsigned int uevent\_suppress:1;

};

### struct kobj\_type

struct kobj\_type {

void (\*release)(struct kobject \*kobj); //释放函数（驱动编写时提供），此函数会被kobject\_put函数调用

struct sysfs\_ops \*sysfs\_ops; //属性文件的操作函数（只有读和写操作）

struct attribute \*\*default\_attrs; //属性数组

const struct kobj\_ns\_type\_operations \*(\*child\_ns\_type)(struct kobject \*kobj);

const void \*(\*namespace)(struct kobject \*kobj);

};

### struct kset

struct kset {

struct list\_head list; //这个链表存放这个kset关联的所有kobject

spinlock\_t list\_lock; //维护此链表的锁

struct kobject kobj; //内嵌的kobject。这样kset本身也是一个kobject也被表现为一个目录

struct kset\_uevent\_ops \*uevent\_ops; //支持热插拔事件的函数集

};

### struct kobj\_attribute

struct kobj\_attribute {

struct attribute attr;

ssize\_t (\*show)(struct kobject \*kobj, struct kobj\_attribute \*attr,

char \*buf);

ssize\_t (\*store)(struct kobject \*kobj, struct kobj\_attribute \*attr,

const char \*buf, size\_t count);

};

## 设备模型上层容器

### 总线类型bus\_type

struct bus\_type {

const char \*name; /\* 总线类型名 \*/

struct bus\_attribute \*bus\_attrs; /\* 总线的属性 \*/

struct device\_attribute \*dev\_attrs; /\* 设备属性,为每个加入总线的设备建立属性链表 \*/

struct driver\_attribute \*drv\_attrs; /\* 驱动属性,为每个加入总线的驱动建立属性链表 \*/

/\* 驱动与设备匹配函数:当一个新设备或者驱动被添加到这个总线时，这个方法会被调用一次或多次，若指定的驱动程序能够处理指定的设备，则返回非零值。必须在总线层使用这个函数, 因为那里存在正确的逻辑，核心内核不知道如何为每个总线类型匹配设备和驱动程序 \*/

int (\*match)(struct device \*dev, struct device\_driver \*drv);

/\*在为用户空间产生热插拔事件之前，这个方法允许总线添加环境变量（参数和 kset 的uevent方法相同）\*/

int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

int (\*probe)(struct device \*dev); /\* \*/

int (\*remove)(struct device \*dev); /\* 设备移除调用操作 \*/

void (\*shutdown)(struct device \*dev);

int (\*suspend)(struct device \*dev, pm\_message\_t state);

int (\*resume)(struct device \*dev);

const struct dev\_pm\_ops \*pm;

struct subsys\_private \*p; /\* 一个很重要的域，包含了device链表和drivers链表 \*/

};

/\*\*

63 \* struct bus\_type - The bus type of the device

64 \*

65 \* @name: The name of the bus.

66 \* @dev\_name: Used for subsystems to enumerate devices like ("foo%u", dev->id).

67 \* @dev\_root: Default device to use as the parent.

68 \* @dev\_attrs: Default attributes of the devices on the bus.

69 \* @bus\_groups: Default attributes of the bus.

70 \* @dev\_groups: Default attributes of the devices on the bus.

71 \* @drv\_groups: Default attributes of the device drivers on the bus.

72 \* @match: Called, perhaps multiple times, whenever a new device or driver

73 \* is added for this bus. It should return a positive value if the

74 \* given device can be handled by the given driver and zero

75 \* otherwise. It may also return error code if determining that

76 \* the driver supports the device is not possible. In case of

77 \* -EPROBE\_DEFER it will queue the device for deferred probing.

78 \* @uevent: Called when a device is added, removed, or a few other things

79 \* that generate uevents to add the environment variables.

80 \* @probe: Called when a new device or driver add to this bus, and callback

81 \* the specific driver's probe to initial the matched device.

82 \* @remove: Called when a device removed from this bus.

83 \* @shutdown: Called at shut-down time to quiesce the device.

84 \*

85 \* @online: Called to put the device back online (after offlining it).

86 \* @offline: Called to put the device offline for hot-removal. May fail.

87 \*

88 \* @suspend: Called when a device on this bus wants to go to sleep mode.

89 \* @resume: Called to bring a device on this bus out of sleep mode.

90 \* @pm: Power management operations of this bus, callback the specific

91 \* device driver's pm-ops.

92 \* @iommu\_ops: IOMMU specific operations for this bus, used to attach IOMMU

93 \* driver implementations to a bus and allow the driver to do

94 \* bus-specific setup

95 \* @p: The private data of the driver core, only the driver core can

96 \* touch this.

97 \* @lock\_key: Lock class key for use by the lock validator

98 \*

99 \* A bus is a channel between the processor and one or more devices. For the

100 \* purposes of the device model, all devices are connected via a bus, even if

101 \* it is an internal, virtual, "platform" bus. Buses can plug into each other.

102 \* A USB controller is usually a PCI device, for example. The device model

103 \* represents the actual connections between buses and the devices they control.

104 \* A bus is represented by the bus\_type structure. It contains the name, the

105 \* default attributes, the bus' methods, PM operations, and the driver core's

106 \* private data.

107 \*/

108 struct bus\_type {

109 const char \*name;

110 const char \*dev\_name;

111 struct device \*dev\_root;

112 struct device\_attribute \*dev\_attrs; /\* use dev\_groups instead \*/

113 const struct attribute\_group \*\*bus\_groups;

114 const struct attribute\_group \*\*dev\_groups;

115 const struct attribute\_group \*\*drv\_groups;

116

117 int (\*match)(struct device \*dev, struct device\_driver \*drv);

118 int (\*uevent)(struct device \*dev, struct kobj\_uevent\_env \*env);

119 int (\*probe)(struct device \*dev);

120 int (\*remove)(struct device \*dev);

121 void (\*shutdown)(struct device \*dev);

122

123 int (\*online)(struct device \*dev);

124 int (\*offline)(struct device \*dev);

125

126 int (\*suspend)(struct device \*dev, pm\_message\_t state);

127 int (\*resume)(struct device \*dev);

128

129 const struct dev\_pm\_ops \*pm;

130

131 const struct iommu\_ops \*iommu\_ops;

132

133 struct subsys\_private \*p;

134 struct lock\_class\_key lock\_key;

135 };

### Struct device

709 /\*\*

710 \* struct device - The basic device structure

711 \* @parent: The device's "parent" device, the device to which it is attached.

712 \* In most cases, a parent device is some sort of bus or host

713 \* controller. If parent is NULL, the device, is a top-level device,

714 \* which is not usually what you want.

715 \* @p: Holds the private data of the driver core portions of the device.

716 \* See the comment of the struct device\_private for detail.

717 \* @kobj: A top-level, abstract class from which other classes are derived.

718 \* @init\_name: Initial name of the device.

719 \* @type: The type of device.

720 \* This identifies the device type and carries type-specific

721 \* information.

722 \* @mutex: Mutex to synchronize calls to its driver.

723 \* @bus: Type of bus device is on.

724 \* @driver: Which driver has allocated this

725 \* @platform\_data: Platform data specific to the device.

726 \* Example: For devices on custom boards, as typical of embedded

727 \* and SOC based hardware, Linux often uses platform\_data to point

728 \* to board-specific structures describing devices and how they

729 \* are wired. That can include what ports are available, chip

730 \* variants, which GPIO pins act in what additional roles, and so

731 \* on. This shrinks the "Board Support Packages" (BSPs) and

732 \* minimizes board-specific #ifdefs in drivers.

733 \* @driver\_data: Private pointer for driver specific info.

734 \* @power: For device power management.

735 \* See Documentation/power/devices.txt for details.

736 \* @pm\_domain: Provide callbacks that are executed during system suspend,

737 \* hibernation, system resume and during runtime PM transitions

738 \* along with subsystem-level and driver-level callbacks.

739 \* @pins: For device pin management.

740 \* See Documentation/pinctrl.txt for details.

741 \* @msi\_list: Hosts MSI descriptors

742 \* @msi\_domain: The generic MSI domain this device is using.

743 \* @numa\_node: NUMA node this device is close to.

744 \* @dma\_mask: Dma mask (if dma'ble device).

745 \* @coherent\_dma\_mask: Like dma\_mask, but for alloc\_coherent mapping as not all

746 \* hardware supports 64-bit addresses for consistent allocations

747 \* such descriptors.

748 \* @dma\_pfn\_offset: offset of DMA memory range relatively of RAM

749 \* @dma\_parms: A low level driver may set these to teach IOMMU code about

750 \* segment limitations.

751 \* @dma\_pools: Dma pools (if dma'ble device).

752 \* @dma\_mem: Internal for coherent mem override.

753 \* @cma\_area: Contiguous memory area for dma allocations

754 \* @archdata: For arch-specific additions.

755 \* @of\_node: Associated device tree node.

756 \* @fwnode: Associated device node supplied by platform firmware.

757 \* @devt: For creating the sysfs "dev".

758 \* @id: device instance

759 \* @devres\_lock: Spinlock to protect the resource of the device.

760 \* @devres\_head: The resources list of the device.

761 \* @knode\_class: The node used to add the device to the class list.

762 \* @class: The class of the device.

763 \* @groups: Optional attribute groups.

764 \* @release: Callback to free the device after all references have

765 \* gone away. This should be set by the allocator of the

766 \* device (i.e. the bus driver that discovered the device).

767 \* @iommu\_group: IOMMU group the device belongs to.

768 \*

769 \* @offline\_disabled: If set, the device is permanently online.

770 \* @offline: Set after successful invocation of bus type's .offline().

771 \*

772 \* At the lowest level, every device in a Linux system is represented by an

773 \* instance of struct device. The device structure contains the information

774 \* that the device model core needs to model the system. Most subsystems,

775 \* however, track additional information about the devices they host. As a

776 \* result, it is rare for devices to be represented by bare device structures;

777 \* instead, that structure, like kobject structures, is usually embedded within

778 \* a higher-level representation of the device.

779 \*/

780 struct device {

781 struct device \*parent;

782

783 struct device\_private \*p;

784

785 struct kobject kobj;

786 const char \*init\_name; /\* initial name of the device \*/

787 const struct device\_type \*type;

788

789 struct mutex mutex; /\* mutex to synchronize calls to

790 \* its driver.

791 \*/

792

793 struct bus\_type \*bus; /\* type of bus device is on \*/

794 struct device\_driver \*driver; /\* which driver has allocated this

795 device \*/

796 void \*platform\_data; /\* Platform specific data, device

797 core doesn't touch it \*/

798 void \*driver\_data; /\* Driver data, set and get with

799 dev\_set/get\_drvdata \*/

800 struct dev\_pm\_info power;

801 struct dev\_pm\_domain \*pm\_domain;

802

803 #ifdef CONFIG\_GENERIC\_MSI\_IRQ\_DOMAIN

804 struct irq\_domain \*msi\_domain;

805 #endif

806 #ifdef CONFIG\_PINCTRL

807 struct dev\_pin\_info \*pins;

808 #endif

809 #ifdef CONFIG\_GENERIC\_MSI\_IRQ

810 struct list\_head msi\_list;

811 #endif

812

813 #ifdef CONFIG\_NUMA

814 int numa\_node; /\* NUMA node this device is close to \*/

815 #endif

816 u64 \*dma\_mask; /\* dma mask (if dma'able device) \*/

817 u64 coherent\_dma\_mask;/\* Like dma\_mask, but for

818 alloc\_coherent mappings as

819 not all hardware supports

820 64 bit addresses for consistent

821 allocations such descriptors. \*/

822 unsigned long dma\_pfn\_offset;

823

824 struct device\_dma\_parameters \*dma\_parms;

825

826 struct list\_head dma\_pools; /\* dma pools (if dma'ble) \*/

827

828 struct dma\_coherent\_mem \*dma\_mem; /\* internal for coherent mem

829 override \*/

830 #ifdef CONFIG\_DMA\_CMA

831 struct cma \*cma\_area; /\* contiguous memory area for dma

832 allocations \*/

833 #endif

834 /\* arch specific additions \*/

835 struct dev\_archdata archdata;

836

837 struct device\_node \*of\_node; /\* associated device tree node \*/

838 struct fwnode\_handle \*fwnode; /\* firmware device node \*/

839

840 dev\_t devt; /\* dev\_t, creates the sysfs "dev" \*/

841 u32 id; /\* device instance \*/

842

843 spinlock\_t devres\_lock;

844 struct list\_head devres\_head;

845

846 struct klist\_node knode\_class;

847 struct class \*class;

848 const struct attribute\_group \*\*groups; /\* optional groups \*/

849

850 void (\*release)(struct device \*dev);

851 struct iommu\_group \*iommu\_group;

852

853 bool offline\_disabled:1;

854 bool offline:1;

855 };

### Struct device\_driver

/\*\*

231 \* struct device\_driver - The basic device driver structure

232 \* @name: Name of the device driver.

233 \* @bus: The bus which the device of this driver belongs to.

234 \* @owner: The module owner.

235 \* @mod\_name: Used for built-in modules.

236 \* @suppress\_bind\_attrs: Disables bind/unbind via sysfs.

237 \* @probe\_type: Type of the probe (synchronous or asynchronous) to use.

238 \* @of\_match\_table: The open firmware table.

239 \* @acpi\_match\_table: The ACPI match table.

240 \* @probe: Called to query the existence of a specific device,

241 \* whether this driver can work with it, and bind the driver

242 \* to a specific device.

243 \* @remove: Called when the device is removed from the system to

244 \* unbind a device from this driver.

245 \* @shutdown: Called at shut-down time to quiesce the device.

246 \* @suspend: Called to put the device to sleep mode. Usually to a

247 \* low power state.

248 \* @resume: Called to bring a device from sleep mode.

249 \* @groups: Default attributes that get created by the driver core

250 \* automatically.

251 \* @pm: Power management operations of the device which matched

252 \* this driver.

253 \* @p: Driver core's private data, no one other than the driver

254 \* core can touch this.

255 \*

256 \* The device driver-model tracks all of the drivers known to the system.

257 \* The main reason for this tracking is to enable the driver core to match

258 \* up drivers with new devices. Once drivers are known objects within the

259 \* system, however, a number of other things become possible. Device drivers

260 \* can export information and configuration variables that are independent

261 \* of any specific device.

262 \*/

263 struct device\_driver {

264 const char \*name;

265 struct bus\_type \*bus;

266

267 struct module \*owner;

268 const char \*mod\_name; /\* used for built-in modules \*/

269

270 bool suppress\_bind\_attrs; /\* disables bind/unbind via sysfs \*/

271 enum probe\_type probe\_type;

272

273 const struct of\_device\_id \*of\_match\_table;

274 const struct acpi\_device\_id \*acpi\_match\_table;

275

276 int (\*probe) (struct device \*dev);

277 int (\*remove) (struct device \*dev);

278 void (\*shutdown) (struct device \*dev);

279 int (\*suspend) (struct device \*dev, pm\_message\_t state);

280 int (\*resume) (struct device \*dev);

281 const struct attribute\_group \*\*groups;

282

283 const struct dev\_pm\_ops \*pm;

284

285 struct driver\_private \*p;

286 };

## 从面向对象的角度

- struct kobj（及其相关结构如kset, ktype等）属于最抽象的基类，代码最简洁，最不具体； 通常我们并不关注 kobject 本身，而应该关注那些嵌入了 kobject 的那些结构体。；ktype 是嵌入了 kobject 的对象的类型。每个嵌入了 kobject 的对象都需要一个相应的 ktype。ktype 用来控制当 kobject 创建和销毁时所发生的操作。 这些 kobject 可以是同样的 ktype，也可以分别属于不同的 ktype。kset 是 kobject 集合的基本容器类型。kset 是 kobject 的一组集合。kset 也包含它们自己的 kobject，但是你可以放心的忽略这些 kobjects，因为 kset 的核心代码会自动处理这些 kobject。

- struct device（及其相关结构如device\_driver，device\_type等）是对kobj的封装，是第一层派生类；

- 再上层的结构（如platform\_device等），是在struct device的基础上再封装一次，是第二层派生类。

因此，例如我们创建了一个struct platform\_device的实例，使用完毕后要释放它。那么这个过程按道理应该是：

- 系统内部先调用platform\_device的remove函数，它只处理自己层特有的变量；

- 完毕后，系统调用第一层派生类struct device的release函数，处理了自己这一层的特有变量；

- 最后，kobject的release函数，将整个空间释放掉。

整个过程应该会跟C++析构过程比较类似，上述的“系统内部”也应该类似于C++编译器自动生成的代码，因为C++中析构函数的逆向调用是自动进行的，并没有在派生类的析构函数中显示调用。类似地，在此处上层的release中也不会显式调用下层release，都是由系统内部完成的。

## Kobject，kset，ktype结构图

一个kobject结构如下图的kobject 类型部分，而一个kset结构如下图的kset 类型部分，一个kobject加入一个kset，主要是kobject结构体中的相关字段记录了对应的kset信息，①记录了kobject所对应 kset，其所指向的是kset所包含的kobject的地址，②记录了kobject所对应的kset的kset指针，③记录了kobject的类 型，④记录了kset所有的kobject的链子，这个链子是一个双向链表，每当有一个kobject加入到当前的kset,就会调用 list\_add\_tail()函数，把要加入kset的kobject连入链表的结尾，最终形成一个链表。



当有另外一个kobject要加入当前的kset，其中的①②③步跟第一个加入当前kset的kobject是一样的，即把要加入 的kobject的成员设置，使之指向当前的kset对应数据，而④需要把kobject添加到kset的list的尾部，下图表示了kobject b加入到kset A的图示：



当有一个kset，需要加入到当前的kset,其方法也跟一个kobject要加入到当前kset一样，即把要加入的kset中所 包含的kobject的成员设置，使这些成员指向对应的kset的对应数据。而当前kset要加入另一个kset，其方式也是跟一个kset加入到当前 kset一样，都是设备kset中的kobject，使kobject的成员指向要加入的kset的对应数据即可，下图显示了一个kset B加入到kset A中的图示。



由于一条总线要管理总线上的所有驱动，同时要管理总线上的有所设备，则需要再把所有设备和所有驱动都分开，分别设立一个设 备kset和一个设备驱动kset，用于管理所有的设备和设备驱动，如此，则总线kset实际上包含了两个kset（设备kset，设备驱动kset）， 设备kset又包含了所有的当前总线的设备的kobject，设备驱动kset包含了所有的当前总线的设备驱动的kobject；而所有的总线，又形成了 bus kset，归结起来就形成下图的层次关系：



经过上述的设备插入，或者驱动安装，系统就会出现只有设备，而没有设备驱动程序的情况，也会出现，只有设备驱动程序，没有对应的设备的情况，此时，设备或者设备驱动程序，就会暂时在各自的队列里等待，一旦有驱动程序安装，或新的设备插入，就都会自动的去扫描对应的链表，来检测是否有配对的可能。

综合上述三者的关系，如图：



## **Kobj type**

数据结构包含三个域：一个release方法用于释放kobject占用的资源；一个sysfs ops指针指向sysfs操作表和一个sysfs文件系统缺省属性列表。Sysfs操作表包括两个函数store()和show()。当用户态读取属性时，show()函数被调用，该函数编码指定属性值存入buffer中返回给用户态；而store()函数用于存储用户态传入的属性值。

## 参考资料

http://eeepage.info/sysfs/ 对新的sysfs的讲解非常到位

http://www.linuxidc.com/Linux/2012-05/60757.htm 同样讲解sysfs文件系统

http://blog.chinaunix.net/uid-24227137-id-3266449.html 讲解kobject,kset,sysfs

http://blog.csdn.net/xiahouzuoxin/article/details/8943863

# 最小子系统+ramdisk+initrd+系统启动过程

## ramdisk+initrd

ramdisk是一种基于内存的虚拟文件系统，通常用于放置内核的中间数据。

而initrd全称为"boot loader initialized RAM disk"，也就是由启动加载器所初始化的RamDisk设备，它的作用是完善内核的模块机制，让内核的初始化流程更具弹性；内核以及initrd，都由bootloader在机子启动后被加载至内存的指定位置，主要功能为按需加载模块以及按需改变根文件系统。initrd 大体上就是指包含根文件系统的ramdisk。initrd即initial ramdisk,它是在系统引导过程中挂载的一个临时根文件系统.激活系统所须加载的文件系统.



### BIOS(Basic Input Output System)

可以视为是一个永久地记录在ROM中的一个软件，是操作系统输入输出管理系统的一部分

包括：

**自检及初始化**

POST上电自检（如果内存没有插好，开机会滴滴的响，这就是上电自检查处了问题）

**初始化**

程序服务处理

硬件中断处理

枚举本地设备并对其初始化;’

BIOS在内存中建立中断向量表和中断服务程序

系统设置程序（开机按f2或者f12跳出来的配置界面就是配置这个参数）：准确的说法应是通过BIOS设置程序对CMOS参数进行设置

以硬盘启动为例，BIOS此时去读取硬盘驱动器的第一个扇区(MBR，512字节)，然后执行里面的代码；**将控制权转交给MBR**

### 系统引导

MBR，它是Master Boot Record的缩写。硬盘的0柱面、0磁头、1扇区称为主引导扇区。它由三个部分组成，**主引导程序(Bootloader)**、 **硬盘分区表DPT（Disk Partition table）**和**硬盘有效标志（55AA）**，其结构图如下所示：



        磁盘分区表包含以下三部分：

        1）Partition ID  （5：延伸  82：Swap   83：Linux   8e：LVM     fd：RAID）

        2）Partition起始磁柱

        3）Partition的磁柱数量

       通常情况下，诸如lilo、grub这些常见的引导程序都直接安装在MBR中。我们以grub为例来分析这个引导过程。

       grub引导也分为两个阶段stage1阶段和stage2阶段(有些较新的grub又定义了stage1.5阶段)。

        1)stage1：stage1是直接被写入到MBR中去的，也就是上图所看到的前446个字节空间中存放的是stage1的代码。BIOS将stage1载入内存中0x7c00处并跳转执行。stage1（/stage1/start.S）仅仅是将硬盘0头0道2扇区读入内存。而0头0道2扇区内容是源代码中的/stage2/start.S，编译后512字节，它是stage2或者stage1\_5的入口。

         2)stage2：严格来说这里还应该再区分个stage1.5的，就一并把stage1.5放在这里一起介绍了，免得大家看得心里乱哄哄的。好的，我们继续说0头0到2扇区的/stage2/start.S文件，当它的内容被读入到内存之后，它的主要作用就是负责将stage2或stage1.5从硬盘读到内存中。如果是stage2，它将被载入到0x820处；如果是stage1.5，它将被载入到0x2200处。这里的stage2或者stage1\_5不是/boot分区/boot/grub目录下的文件，因为这个时候grub还没有能力识别任何文件系统。

        如果start.S加载stage1.5：stage1.5它存放在硬盘0头0道3扇区向后的位置，stage1\_5作为stage1和stage2中间的桥梁，stage1\_5有识别文件系统的能力，此后grub才有能力去访问/boot分区/boot/grub目录下的 stage2文件，将stage2载入内存并执行。

        如果start.S加载stage2：同样，这个stage2也不是/boot分区/boot/grub目录下的stage2，这个时候start.S读取的是存放在/boot分区Boot Sector的stage2。这种情况下就有一个限制：因为start.S通过BIOS中断方式直接对硬盘寻址（而非通过访问具体的文件系统），其寻址范围有限，限制在8GB以内。因此这种情况需要将/boot分区分在硬盘8GB寻址空间之前。

        假如是情形2，我们将/boot/grub目录下的内容清空，依然能成功启动grub；假如是情形1，将/boot/grub目录下stage2删除后，则系统启动过程中grub会启动失败。



而最终 boot loader 的功能就是加载 kernel（内核）文件

 Linux 内核是可以通过动态加载内核模块的 (就请想成驱动程序即可)，这些内核模块就放置在 /lib/modules/ 目录内。 由于模块放置到磁盘根目录内 (要记得 /lib 不可以与 / 分别放在不同的分区)，因此在启动的过程中内核必须要挂载根目录，这样才能够读取内核模块提供加载驱动程序的功能。而且为了避免影响到磁盘内的文件系统，因此启动过程中根目录是以只读的方式来挂载。

一般来说，非必要的功能且可以编译成为模块的内核功能，目前的 Linux distributions 都会将它编译成为模块。因此 U盘, SATA, SCSI... 等磁盘设备的驱动程序通常都是以模块的方式来存在的。现在来思考一种情况，假设你的 linux 是安装在 SATA 磁盘上面的，你可以通过 BIOS 的 INT 13 取得 boot loader 与 kernel 文件来启动，然后 kernel 会开始接管系统并且检测硬件及尝试挂载根目录来取得额外的驱动程序。

问题是，内核根本不认识 SATA 磁盘，所以需要加载 SATA 磁盘的驱动程序，否则根本就无法挂载根目录。但是 SATA 的驱动程序在 /lib/modules 内，你根本无法挂载根目录，又怎么读取到 /lib/modules/ 内的驱动程序？是吧！是不是有点进退两难？在这个情况之下，你的 Linux 是无法顺利启动的！ 那怎办？没关系，我们可以透过虚拟文件系统来处理这个问题。

虚拟文件系统 (Initial RAM Disk) 一般使用的文件名为 /boot/initrd ，这个文件的特色是，它也能够通过boot loader 来加载到内存中， 然后这个文件会被解压缩并且在内存当中仿真成一个根目录，且此仿真在内存当中的文件系统能够提供一个可执行的程序，通过该程序来加载启动过程中所最需要的内核模块，通常这些模块就是 U盘, RAID, LVM, SCSI 等文件系统与磁盘接口的驱动程序。等载入完成后，会帮助内核重新调用 /sbin/init 来开始后续的正常启动流程，如图



一张图片关于bios以后的bootloader 这张图片 讲解的非常详细



## 参考资料

<http://blog.chinaunix.net/uid-23069658-id-3142047.html>

<http://blog.sina.com.cn/s/blog_c70e10380102w9b5.html>

<http://blog.csdn.net/miss_acha/article/details/50004717>

<http://www.2cto.com/os/201603/494064.html>

http://blog.itpub.net/8111049/viewspace-680043

<http://www.ibm.com/developerworks/cn/linux/l-initrd.html>

<http://bbs.chinaunix.net/thread-2046548-1-1.html>

http://bbs.chinaunix.net/thread-2046548-1-1.html

小知识：

\*.o 中间文件

\*.so 文件是动态链接库文件，相当于 win下的 .dll 文件。

\*.a 文件是静态库文件。

\*.ko 是内核模块文件，是内核加载的某个模块，一般是驱动程序。

具体的编译命令请百度

# Paltform

## Platform设备驱动

可以通过CPU bus直接寻址（例如在嵌入式系统常见的“寄存器”）。因此，由于这个共性，内核在设备模型的基础上（device和device\_driver），对这些设备进行了更进一步的封装，抽象出paltform bus、platform device和platform driver，以便驱动开发人员可以方便的开发这类设备的驱动。

Platform设备驱动包含三部分：**Platform总线**，**Platform设备**，**Platform设备驱动**

然而这三者是基于设备模型的概念：总线，设备与驱动。

## Platform模块的软件架构

内核中Platform设备有关的实现位include/linux/platform\_device.h和drivers/base/platform.c两个文件中，它的软件架构如下：



结合之前的知识讲解一下这幅图：每一个单独的结构体都是一个kobject（上图中的bus,device,device\_driver,platformbus,platform\_device,platform\_driver）；内核对这些东西又有一个整理，所有的bus是一个kset；所有的device是一个kset；所有的driver是一个kset；而ktype则是对这些kobject的附属操作。

Platform Bus，基于底层bus模块，抽象出一个虚拟的Platform bus，用于挂载Platform设备；

Platform Device，基于底层device模块，抽象出Platform Device，用于表示Platform设备；

Platform Driver，基于底层device\_driver模块，抽象出Platform Driver，用于驱动Platform设备。

其中Platform Device和Platform Driver会为其它Driver提供封装好的API

## Platform模块向其它模块提供的API汇整

Platform提供的接口包括：Platform Device和Platform Driver两个数据结构，以及它们的操作函数

用于抽象Platform设备的数据结构----“struct platform\_device”：

用于抽象Platform设备驱动的数据结构----“struct platform\_driver”：

具体的各类api请参考：<http://www.wowotech.net/device_model/platform_device.html>

## 总结大多数驱动框架

## Platform led驱动

最简单的了解platform平台的例子，可以理解为3部分，由驱动层，系统核心层，设备驱动三部分组成：

驱动层：硬件设备注册部分。

系统核心层：无

设备驱动层：设备端的实现，如led闪烁等

实际上之所以这里分成3部分，是为了与后面的设备驱动程序对应起来。

使用步骤示例：

(1)platform\_device\_register()：注册平台led设备。

(2)platform\_driver\_register()：注册平台led驱动。

## Platform input驱动

Linux系统提供了input子系统，按键、触摸屏、键盘、鼠标等输入都可以利用input接口函数来实现设备驱动。

在linux主要由驱动层，系统核心层(Input Core)和事件处理层（Event Handler）三部份组成。

驱动层：硬件设备注册部分，只是把输入设备注册到input子系统中，在驱动层的代码本身并不创建结点。对应文件如gpio\_key.c

Input core：向系统报告按键、触摸屏、键盘、鼠标等输入事件(event,通过input\_event结构体描述)，使得驱动层不需要关心文件操作接口。对应文件如Input.c

Event Handler：提供input设备接口。 对应文件如evdev.c,mousedev.c等。

一般来说，如果要使用input子系统，只需要更改驱动层部分就可以了。

## Platform i2c驱动

Linux系统中,i2c驱动由3部分组成，即i2c总线驱动、i2c core、i2c设备驱动。

I2c总线驱动：对i2c硬件体系结构中适配器端的实现，适配器可由CPU控制，或集成在CPU内部。对应文件如:i2c-at91.c

I2c core：提供了i2c总线驱动和设备驱动的注册、注销方法，i2c algorithm。与具体适配器无关的代码以及探测设备、检测设备地址的上层代码。对应文件如:i2c-core.c

I2c设备驱动：i2c体系硬件结构中设备端的实现，设备一般挂在受CPU控制的i2c适配器上，通过i2c适配器与CPU交换数据。对应文件如:at24.c,i2c-dev.c等。

对于常见的开发板来说，主芯片已经带了i2c总线，i2c总线驱动基本上提供了，不用怎么动。即使不带i2c总线，基本上也会提供io模拟的i2c，也就是说i2c总线驱动部分一般情况下不需要自己写或者更改。I2c core部分就更不用动了，呵呵。因此，写一个i2c设备的驱动，只需要写i2c设备驱动（这里对应于上面说的i2c驱动的3部分之一）就可以了。

大多数i2c设备驱动，内核已经提供了。而且简单的应用还可以利用i2c-dev.c来实现。

## Platform spi驱动

   Linux系统中，spi驱动由3部分组成，即spi总线驱动、spi core、spi设备驱动。

   Spi总线驱动:硬件spi驱动的实现，spi可为主芯片内部集成，也可以io口模拟。对应文件如：atmel\_spi.c

   Spi core:提供了spi总线驱动和设备驱动的注册、注销方法。

   Spi 设备驱动：spi体系结构中,spi设备端的实现。

综合上述几个比较简单的驱动可以看出一个共性：

这几个驱动基本都是由3部分组成：

（1） 总线驱动：与所选用的主芯片相关联，一般都有提供。

（2） 总线core：与具体的硬件无关，内核已经提供。

（3） 总线设备驱动：所操作的具体设备。根据实际应用需要，使用或更改内核已经提供的驱动，或者自己重新写一个驱动。

实际上，写一个设备驱动，我们所要做的工作基本上集中在第3部分，而这部分，内核也提供了大多数设备的驱动，即使没有提供，我们也可以根据已有的设备自己更改。

## 参考资料

<http://blog.chinaunix.net/uid-27041925-id-3884955.html>

<http://www.wowotech.net/device_model/platform_device.html>

http://blog.chinaunix.net/uid-25014876-id-111745.html

# 中断子系统

## 中断

Cpu在执行程序的过程中，出现某些突发时间急待处理，cpu必须暂时停止当前程序的执行，转去处理突发事件，处理完毕后又返回原程序被中断的位置继续执行。

外设产生中断是异步发生的，硬件设备生成中断的时候并不考虑与处理器的时钟同步——也就是说中断随时可以产生。内核随时可能因为新到来的中断而被打断。从物理学的角度看，中断是一种电信号，由硬件设备生成，并直接送入中断控制器的输入引脚。然后由中断控制器向处理器发送相应的信号。

中断是处理器和外设之间的一种通信机制，也是操作系统内核对外设进行管理的一种机制。外设通过发出特殊的电信号通知处理器发生了一次中断，处理器收到信号后就通知操作系统，然后操作系统负责做出相应的处理。

一个完整的设备中，与中断相关的硬件可以划分为3类，它们分别是：设备、中断控制器和CPU本身，下图展示了一个smp系统中的中断硬件的组成结构：



                          图  中断系统的硬件组成

**设备**  设备是发起中断的源，当设备需要请求某种服务的时候，它会发起一个硬件中断信号，通常，该信号会连接至中断控制器，由中断控制器做进一步的处理。在现代的移动设备中，发起中断的设备可以位于soc（system-on-chip）芯片的外部，也可以位于soc的内部，因为目前大多数soc都集成了大量的硬件IP，例如I2C、SPI、Display Controller等等。

Ps：中断优先寄存器，其英文缩写IP，实为“Interrupt Priority”的简写。

**中断控制器**  中断控制器负责收集所有中断源发起的中断，现有的中断控制器几乎都是可编程的，通过对中断控制器的编程，我们可以控制每个中断源的优先级、中断的电器类型，还可以打开和关闭某一个中断源，在smp系统中，甚至可以控制某个中断源发往哪一个CPU进行处理。对于ARM架构的soc，使用较多的中断控制器是VIC（Vector Interrupt Controller），进入多核时代以后，GIC（General Interrupt Controller）的应用也开始逐渐变多。

**CPU**  cpu是最终响应中断的部件，它通过对可编程中断控制器的编程操作，控制和管理者系统中的每个中断，当中断控制器最终判定一个中断可以被处理时，他会根据事先的设定，通知其中一个或者是某几个cpu对该中断进行处理，虽然中断控制器可以同时通知数个cpu对某一个中断进行处理，实际上，最后只会有一个cpu相应这个中断请求，但具体是哪个cpu进行响应是可能是随机的，中断控制器在硬件上对这一特性进行了保证，不过这也依赖于操作系统对中断系统的软件实现。在smp系统中，cpu之间也通过IPI（inter processor interrupt）中断进行通信

## IRQ编号

系统中每一个注册的中断源，都会分配一个唯一的编号用于识别该中断，我们称之为IRQ编号。IRQ编号贯穿在整个Linux的通用中断子系统中。在移动设备中，每个中断源的IRQ编号都会在arch相关的一些头文件中，例如arch/xxx/mach-xxx/include/irqs.h。驱动程序在请求中断服务时，它会使用IRQ编号注册该中断，中断发生时，cpu通常会从中断控制器中获取相关信息，然后计算出相应的IRQ编号，然后把该IRQ编号传递到相应的驱动程序中。

## 中断处理程序

      在响应一个特定中断的时候，内核会执行一个函数，该函数叫做中断处理程序或中断服务例程（interrupt service routine,ISR）。产生中断的每一个设备都有一个（中断处理程序通常不是和特定设备相关联，而是和特定中断相关联，也就是说，若一个设备可以产生 多种不同的中断，那么该设备就可以对应多个中断处理程序，相应的，该设备的驱动也就需要准备多个这样的函数。）相应的中断处理程序。一个设备的中断程序是它设备驱动程序的一部分——设备驱动程序是用于对设备进行管理的内核代码。

       在Linux中，中断处理程序看起来就是普普通通的C函数。只不过这些函数必须按照特定的类型声明，以便内核能够以标准的方式传递处理程序的信息。中断处理程序与其它内核函数的真正区别在于：中断处理程序是被内核调用来响应中断的，而它们运行于我们称之为中断上下文的特殊上下文中。

**执行在中断上下文中的代码需要注意的一些事项：**

**中断上下文中的代码不能进入休眠。**

**不能使用mutex，只能使用自旋锁，且仅当必须时。**

**中断处理函数不能直接与用户空间进行数据交换。**

**中断处理程序应该尽快结束。**

**中断处理程序不需要是可重入的，因为相同的中断处理函数不能同时在多个处理器上运行。**

**中断处理程序可能被一个优先级更高的中断处理程序所中断。为了避免这种情况，可以要求内核将中断处理程序标记为一个快速中断处理程序（将本地CPU上的所有中断禁用），不过在采取这个动作前要慎重考虑对系统的影响**

## 上半部与下半部

ISR的执行必须足够快速，这是由多种原因决定的。由于中断可以随时发生，因此ISR可以随时被执行，同时ISR打断了系统正常的流程，另外产生中断的外设也需要系统快速响应，最后由于linux不允许相同ISR中断嵌套，执行一个ISR时，当前中断线会在所有CPU上被屏蔽，这段屏蔽时间最好越短越好（否则会降低系统反应速度）。以上这些限制条件都要求ISR执行足够快速。由于速度的要求，ISR中不可能做太多的工作，而只能做一些必不可少的、跟硬件相关的工作，其他的工作就放到所谓的下半部里完成。

把中断处理划分为上半部和下半部是出于现实的考虑。上半部就是ISR，一接收到中断就立即执行，执行时禁止相同中断（必要时禁止所有中断），只做有严格时限且与硬件相关的工作，例如对接收的中断进行应答或复位硬件；而可以稍后执行或者与硬件无关的动作都放到下半部，在合适的时机下半部才开始执行。上半部和下半部的关键区别是，上半部简单快速，执行时禁止一部分或者全部中断；下半部稍后执行，而且执行期间可以响应所有中断。

举个网卡的例子，当网卡收到来自网络的数据包时，会立刻产生中断来通知内核，而内核收到中断后也会立刻调用网卡已注册的ISR，ISR会通知网卡“你的中断已经被组织收到”，并拷贝网卡收到的数据到内存。这些都是紧迫并与硬件相关的动作，如果内核不及时拷贝网卡缓存中的数据，网卡缓存很可能溢出，造成丢包现象。当数据都被拷贝到内存之后，ISR的任务算完成了，此时它将控制权转交给被它打断的程序。而处理内存中的数据的任务则落在下半部里，在内核觉得合适的时机（不太繁忙），下半部将会执行。

上半部和下半部的划分可以参考以下原则：

1) 如果一个任务对时间非常敏感，把它放在上半部（ISR）里；

2) 如果一个任务跟硬件相关，把它放在上半部里；

3) 如果一个任务要保证不被其他中断（特别是相同的中断）打断，把它放在上半部里；

4) 其他所有的任务，都放在下半部。

下半部负责推后完成的工作，但是并不需要指明一个具体的时间，只是将任务稍微推迟，待到系统不是那么繁忙并且中断恢复之后执行就可以了（一般情况下，ISR一返回，下半部就会执行）。不同于ISR的最关键之处是，下半部执行的时候允许响应所有的中断。内核的策略是，当中断不是特别多的时候，及时处理中断，所以do\_irq会调用do\_softirq。

当系统中断过多时，do\_softirq才会被推迟到内核的ksoftirq内核线程中去。如何判断中断过多呢，linux的认为发生中断嵌套了，就是中断过多。do\_irq在调用do\_softirq时会以此为判断条件。

## 在驱动程序中申请中断

Linux中断子系统向驱动程序提供了一系列的API，其中的一个用于向系统申请中断：



request\_threaded\_irq()是Linux kernel 2.6.30 之后新加的irq handler API，如何确定可以用到 request\_threaded\_irq() ?

Linux kernel config 需要定义CONFIG\_GENERIC\_HARDIQS

kernel config 才有支援threaded irq ；Moving interrupts to threads 介绍request\_threaded\_irq() 的由来

http://lwn.net/Articles/302043/

从realtime tree 移植而来，为了减少kernel 因为要等待每一个硬件中断处理的时间 ，就另外交给kernel thread 处理中断后续工作。

 优点：

1 减少 kernel 延迟时间

2 避免处理中断时要分辨是在硬体中断或软体中断？

3 更容易为kernel 中断处理除错，可能可完全取代tasklet

原本的中断处理分上半部(硬体中断处理，必须关闭中断无法处理新的中断)跟下半部( 软体中断处理)，因此上半部的硬体中断处理必须尽可能简短，让系统反应速度更快。  request\_threaded\_irq 是在将上半部的硬件中断处理缩短为只确定硬体中断来 自我们要处理的装置，唤醒kernel thread 执行后续中断任务。

  缺点：

对于非irq 中断的kernel threads ，需要在原本task\_struct 新增struct irqaction 多占 4/8 bytes 记忆体空间，linux kernel 2.6.29 之后(2.6.30)加入request\_threaded\_irq 跟传统top/bottom havles 的差异是threaded\_irq 受Linux kernel system 的 process scheduling 控制，不会因为写错的bottom half 代码造成整个系统 延迟的问题。

也可以透过RT/non RT 跟nice 等工具调整各个thread 优先权，丢给使用率较低的  cpu 以及受惠于kernel 原本可以对threads 做的各种控制，包括但不限于sleep, lock, allocate 新的记忆体区块。

受惠最大的是shared irq line 的多个中断处理。除了可以加速共享中断造成的延迟threaded\_irq 也可以降低在同一段程式码处理多个装置中断的复杂度。 threaded irq 在使用性上也比tasklet(接着top half 直接执行，无法sleep) /workqueue(kernel context?) 等需要在top half 增加跟bottom half 连结与沟通  的麻烦。

**Ps:最简单直观的理解，handler是顶半部，thread\_fn是底半部（个人猜测，不确定正确）**

**现在看来上面那句话确实不准确。感觉这个函数把中断重新定义了。不能说什么顶半部和底半部。**

理解自宋宝华：申请一个线程化的IRQ，kernel会为中断的底版本创建一个名字为irq/%d-%s的线程，%d对应着中断号。其中顶半部（硬中断）**handler**在做完必要的处理工作之后，会返回IRQ\_WAKE\_THREAD，之后kernel会唤醒irq/%d-%s线程，而该kernel线程会调用**thread\_fn**函数，因此，该线程成为底半部。

该机制目前在kernel中使用已经十分广泛，可以认为是继softirq（含tasklet）和workqueue之后的又一大中断底半部方式。

**irq**  需要申请的irq编号，对于ARM体系，irq编号通常在平台级的代码中事先定义好，有时候也可以动态申请。

**handler**  中断服务回调函数，该回调运行在中断上下文中，并且cpu的本地中断处于关闭状态，所以该回调函数应该只是执行需要快速响应的操作，执行时间应该尽可能短小，耗时的工作最好留给下面的thread\_fn回调处理。

**thread\_fn**  如果该参数不为NULL，内核会为该irq创建一个内核线程，当中断发生时，如果handler回调返回值是IRQ\_WAKE\_THREAD，内核将会激活中断线程，在中断线程中，该回调函数将被调用，所以，该回调函数运行在进程上下文中，允许进行阻塞操作。

**flags**  控制中断行为的位标志，IRQF\_XXXX，例如：IRQF\_TRIGGER\_RISING，IRQF\_TRIGGER\_LOW，IRQF\_SHARED等，在include/linux/interrupt.h中定义。

**name**  申请本中断服务的设备名称，同时也作为中断线程的名称，该名称可以在/proc/interrupts文件中显示。

**dev**  当多个设备的中断线共享同一个irq时，它会作为handler的参数，用于区分不同的设备。

## flags:与中断相关的标志

IRQF\_TRIGGER\_RISING：上升沿触发

IRQF\_TRIGGER\_FALLING：下降沿触发

IRQF\_TRIGGER\_HIGH：高电平触发

IRQF\_TRIGGER\_LOW：低电平触发

IRQF\_SAMPLE\_RANDOM：为系统随机发生器提供支持

IRQF\_SHARED：中断可在设备间共享

IRQF\_DISABLED：是否快速中断

IRQF\_ONESHOT:选项说明该中断已经被线程化了（而且是特殊的one shot类型的）

（虽然kernel的注释上是这样说，但是request\_threaded\_irq时还是必须要设定IRQF\_ONESHOT，否者它就会被错误的强制线程化了。

其实我觉得这个ONESHOT的命名也是很不好，它不仅仅表示one shot，而且还是thread irq的标志。）

## **request\_threaded\_irq的工作流程**

函数先是根据irq编号取出对应的irq\_desc实例的指针，然后分配了一个irqaction结构，用参数handler，thread\_fn，irqflags，devname，dev\_id初始化irqaction结构的各字段，同时做了一些必要的条件判断：该irq是否禁止申请？handler和thread\_fn不允许同时为NULL，最后把大部分工作委托给\_\_setup\_irq函数：



图 request\_threaded\_irq（部分）

进入\_\_setup\_irq函数，如果参数flag中设置了IRQF\_SAMPLE\_RANDOM标志，它会调用rand\_initialize\_irq，以便对随机数的生成产生影响。如果申请的不是一个线程嵌套中断（关于线程嵌套中断，请参阅[Linux中断（interrupt）子系统之三：中断流控处理层](http://blog.csdn.net/droidphone/article/details/7489756)中的handle\_nested\_irq一节），而且提供了thread\_fn参数，它将创建一个内核线程;这个函数会判断handler是否可以线程化，如果可以，就会将它线程化：



图 \_\_setup\_irq（部分—与最新内核有区别）

**最新内核:**

**\_setup\_irq->setup\_irq\_thread->kthread\_create->**[**kthread\_create\_on\_node**](http://lxr.free-electrons.com/ident?i=kthread_create_on_node)

如果irq\_desc结构中断action链表不为空，说明这个irq已经被其它设备申请过，也就是说，这是一个共享中断，所以接下来会判断这个新申请的中断与已经申请的旧中断的以下几个标志是否一致：

一定要设置了IRQF\_SHARED标志

电气触发方式要完全一样（IRQF\_TRIGGER\_XXXX）

IRQF\_PERCPU要一致

IRQF\_ONESHOT要一致

检查这些条件都是因为多个设备试图共享一根中断线，试想一下，如果一个设备要求上升沿中断，一个设备要求电平中断，当中断到达时，内核将不知如何选择合适的流控操作。完成检查后，函数找出action链表中最后一个irqaction实例的指针。

如果这不是一个共享中断，或者是共享中断的第一次申请，函数将初始化irq\_desc结构中断线程等待结构：wait\_for\_threads，disable\_irq函数会使用该字段等待所有irq线程的结束。接下来设置中断控制器的电气触发类型，然后处理一些必要的IRQF\_XXXX标志位。如果没有设置IRQF\_NOAUTOEN标志，则调用irq\_startup()打开该irq，在irq\_startup()函数中irq\_desc中的enable\_irq/disable\_irq嵌套深度字段depth设置为0，代表该irq已经打开，如果在没有任何disable\_irq被调用的情况下，enable\_irq将会打印一个警告信息。

接着，设置cpu和irq的亲缘关系

然后，把新的irqaction实例链接到action链表的最后

最后，唤醒中断线程，注册相关的/proc文件节点

至此，irq的申请宣告完毕，当中断发生时，处理的路径将会沿着：irq\_desc.handle\_irq，irqaction.handler，irqaction.thread\_fn（irqaction.handler的返回值是IRQ\_WAKE\_THREAD）这个过程进行处理。下图表明了某个irq被申请后，各个数据结构之间的关系：



## 下半部的几种机制

## 软中断(softirqs)

软中断是一组在编译期间静态分配的下半部接口，一共有个32个。一个软中断不会抢占另一个软中断，唯一能够抢占软中断的是ISR。不同的软中断可以同时在多个CPU上执行，甚至同一个软中断也可以在所有CPU上同时执行，所以软中断必须设计成可重入函数，同时往往需要使用锁机制来保护共享数据

软中断保留给系统中对时间要求最严格以及最重要的下半部使用。在2.6版本中，只有两个子系统（网络和SCSI）直接使用了软中断。此外内核定时器和tasklet都是建立在软中断基础上的。

软中断运行时可以响应中断，但是不能休眠（因为此时仍然处于中断上下文）。当一个CPU正在执行某个软中断时，这个CPU会禁止软中断，但是别的CPU仍然可以执行新触发的软中断（相同的或不同的都可以），这也是软中断的优势——可以在多CPU上并行执行；但相应的tradeoff是，你要小心的保护好共享数据以避免竞争。在实践中，大部分软中断处理程序都采用per-CPU变量来避免显示的加锁，从而提高性能。

## Tasklet

tasklet是一种基于软中断的延时处理机制，是中断底半部的一种处理方式。基本上要使用就是申请中断，通过宏创建tasklet与处理函数的关联，在顶半部调用tasklet\_schedule使系统在适当的时候进行调（就是把tasklet\_struct结构体挂到tasklet\_vec链表或者挂接到tasklet\_hi\_vec链表上，并调度软中断TASKLET\_SOFTIRQ或者HI\_SOFTIRQ；Tasklet\_action在软中断TASKLET\_SOFTIRQ被调度到后会被执行，它从tasklet\_vec链表中把tasklet\_struct结构体都取下来，然后逐个执行。如果t->count的值等于0，说明这个tasklet在调度之后，被disable掉了，所以会将tasklet结构体重新放回到tasklet\_vec链表，并重新调度TASKLET\_SOFTIRQ软中断，在之后enable这个tasklet之后重新再执行它。）；

## work queue

前面提到的软中断和tasklet都运行于中断上下文，因此不可以睡眠；如果需要在下半部中睡眠，那就只能使用工作队列了。工作队列可以把下半部的操作交给一个内核线程来执行——也就是说运行在进程上下文中，因而是可以睡眠的。

Ps：参考request\_threaded\_irq

## 内核定时器

前面提到的下半部机制都是将操作推迟到除了现在之外的其他时间，而内核定时器可以确保将操作推迟到某个确定的时间来执行。如果必须确保在某一个确定的时间间隔以后再运行下半部操作，那么需要使用内核定时器。内核定时器其实也是在软中断基础上实现的。

## 参考资料

http://blog.csdn.net/21cnbao/article/details/8090398 request\_threaded\_irq

http://www.kuqin.com/shuoit/20140104/337421.html 讲述了什么是tasklet

http://blog.csdn.net/lizuobin2/article/details/51793911 这篇讲的比较好，

把中断对应机制分的很详细；但是只是做了一个详细的划分，机制原理讲述不清

http://blog.csdn.net/songjinshi/article/details/23262923 讲述调用调度程序的时机

http://blog.sina.com.cn/s/blog\_70a9dd840100uqfh.html 讲述了什么是isr，

也涉及了中断上下文以及中断上半部和下半部

http://blog.sina.com.cn/s/blog\_65373f1401018w15.html

http://blog.sina.com.cn/s/blog\_510ac74901015fgz.html

http://blog.csdn.net/DroidPhone/article/details/7445825 中断子系统的一系列文章

http://blog.csdn.net/lickylin/article/details/12657373

http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=21977330&id=3755609

# Input子系统与多点触摸技术

## 简介

Linux 的输入子系统不仅支持鼠标、键盘等常规输入设备,而且还支持蜂鸣器、触摸屏等设备。输入子系统又叫 input 子系统。其构建非常灵活,只需要调用一些简单的函数,就可以将一个输入设备的功能呈现给应用程序。

输入设备(如按键,键盘，触摸屏，鼠标等)是典型的字符设备，其一般的工作机制是低层在按键，触摸等动作发生时产生一个中断(或驱动通过timer定时查询)，然后cpu通过SPI，I2C或者外部存储器总线读取键值，坐标等数据，放一个缓冲区，字符设备驱动管理该缓冲区，而驱动的read()接口让用户可以读取键值，坐标等数据。

在Linux中，输入子系统是由输入子系统设备驱动层、输入子系统核心层(Input Core)和输入子系统事件处理层(Event Handler)组成。

其中**设备驱动层**提供对硬件各寄存器的读写访问和将底层硬件对用户输入访问的响应转换为标准的输入事件，再通过核心层提交给事件处理层；

而**核心层**对下提供了设备驱动层的编程接口，对上又提供了事件处理层的编程接口；

而**事件处理层**就为我们用户空间的应用程序提供了统一访问设备的接口和驱动层提交来的事件处理。

所以这使得我们输入设备的驱动部分不在用关心对设备文件的操作，而是要关心对各硬件寄存器的操作和提交的输入事件。

下面是输入子系统结构图：



## ****设备驱动层****

**输入子系统设备驱动层实现原理**

在Linux中，Input设备用input\_dev结构体描述，定义在input.h中。设备的驱动只需按照如下步骤就可实现了。

1).在驱动模块加载函数中设置Input设备支持input子系统的哪些事件；

2).将Input设备注册到input子系统中；

3).在Input设备发生输入操作时(如：键盘被按下/抬起、触摸屏被触摸/抬起/移动、鼠标被移动/单击/抬起时等)，提交所发生的事件及对应的键值/坐标等状态。

linux中输入设备驱动的分层：



## 换个角度看input子系统

 整体来看，Input子系统有一个主线，称为三方关系，input\_dev对应于实际的device端，input\_handler从名字也可以猜出来是对device的处理。“处理”这个词语不单单指的是对device数据的处理，比如report等；它其实可以包括系统在该device事件发生时想做的任何动作。至于input\_handle，它是连接input\_dev与input\_handler。

struct input\_handle {

    ...

struct input\_dev \*dev;//对应dev结构体

struct input\_handler \*handler;//对应handler结构体

    struct list\_head    d\_node;//链入input\_dev的h\_list代表的链表

    struct list\_head    h\_node;//链入input\_handler的h\_list代表的链表

};

至此，三方关系形成完毕。我们实现了最终的目的，通过input\_dev，可以遍历所有与它有关的input\_handler；通过input\_handler，也可以遍历所有与它有关的input\_dev。

图解如下：图中单向箭头表示指针，双向箭头表示list\_head。可以看出，从任何一个双向箭头出发，通过handle的过度，完全实现了我们的最终目标。掌握了这点，再看input\_report那些流程的时候就非常容易了，dev想要report数据的时候无非是调用了handler的event函数指针指向的函数，我们可以在这个函数里定义任何想让系统去做的任务，比如cpu调频等，而不仅限于数据上报。熟悉面向对象编程的人可能想到了，其实这个设计运用了面向对象的observer设计模式。



## 软件设计流程

[](http://blog.chinaunix.net/link.php?url=http://blog.chinaunix.net/link.php?url%3Dhttp://blog.chinaunix.net/attachment/201108/1/23770712_1312213167mtdZ.gif)

图 3 input子系统软件设计流程

## 与软件设计有关的API函数

**分配一个输入设备**

Struct input\_dev \*input\_allocate\_device\*(void);

**注册一个输入设备**

Int input\_register\_device (struct input\_dev \*dev);

**驱动实现-事件支持**

Set\_bit告诉in/out子系统它支持哪些事件

Set\_bit (EV\_KEY, button\_dev.evbit)

Struct input\_dev中有两个成员，一个是evbit;一个是keybit.分别用来表示设备所支持的事件类型和按键类型。

**事件类型**

Linux中输入设备的事件类型有(这里只列出了常用的一些，更多请看linux/input.h中)：

EV\_SYN 0x00 同步事件

EV\_KEY 0x01 按键事件

EV\_REL 0x02 相对坐标

EV\_ABS 0x03 绝对坐标

EV\_MSC 0x04 其它

EV\_LED 0x11 LED

EV\_SND 0x12 声音

EV\_REP 0x14 Repeat

EV\_FF 0x15 力反馈

EV\_PWR     0x16   电源管理事件

驱动实现-报告事件

Void input\_event(struct input\_dev \* dev, unsigned int type, unsigned int code, int value);//报告指定type ,code的输入事件

Void input\_report\_key(struct input\_dev \*dev, unsigned int code, int value);//报告键值

Void input\_report\_rel(struct input\_dev \*dev, unsigned int code, int value);//报告相对坐标

Void input\_report\_abs(struct input\_dev \*dev, unsigned int code, int value);//报告绝对坐标

Void input\_sync(struct input\_dev \*dev);//报告同步事件

在触摸屏驱动设计中，一次坐标及按下状态的整个报告过程如下：

Input\_report\_abs(input\_dev,ABS\_X,x);//X坐标

Input\_report\_abs(input\_dev,ABS\_Y,y);//Y坐标

Input\_report\_abs(input\_dev,ABS\_PRESSURE,pres);//压力

input\_sync(struct input\_dev \*dev);//同步

1.6.5释放与注销设备

Void input\_free\_device(struct input\_dev \*dev);

Void input\_unregister\_device(struct input\_dev \*);

## 多点触摸技术A/B(Slot)协议

### A/B协议究竟是如何划分

B协议又称为slot协议，slot直译为位置、槽，有两层含义，一层是位置，另一层是容器。在Input子系统中，它扮演的就是这两个角色。它产生于这样一个背景：

如果从Device获取的当前数据与上一个数据相同，我们有必要再上报当前数据吗？如果我们不管两次数据是否一致都上报，那就是A协议；如果我们选择不上报，那么既然需要比较，总需要把上一次数据存起来吧，slot就是做这个事情的，显然这就是Slot(B)协议。

### A协议实现方式

A协议不会使用slot，多指处理中，它的报点序列如下（每一个序列都以input\_report\_\*\*\*函数实现）：

手指按下的动作：

**ABS\_MT\_POSITION\_X x[0]**

**ABS\_MT\_POSITION\_Y y[0]**

**SYN\_MT\_REPORT**

**ABS\_MT\_POSITION\_X x[1]**

**ABS\_MT\_POSITION\_Y y[1]**

**SYN\_MT\_REPORT**

**…**

**SYN\_REPORT**

上面的序列中需要说明的是系统以SYN\_MT\_REPORT为一个点的信息的结尾，以SYN\_REPORT为一次事件的结尾。也就是说多指触摸的时候，android的中间件部分每收到一次SYN\_MT\_REPORT就形成一个点信息，收到一个点之后并不会立即处理，而是一个事件完成之后才会处理，SYN\_REPORT就是这个事件的标志。A协议比较简单，我们也可以发现在上面的序列中根本就没有轨迹跟踪的信息，有的只是点坐标等信息

#### 系统如果去判断当前的多个点各属于哪一条线

我们假设前一次事件共有5个点，本次触摸也有5个点，系统会分别计算前一次5个点与本次5个点的距离，distance[prev\_i, curr\_j] (i=0,1,...,4; j=0,1,...4)，这样会产生总共5\*5=25个数字。然后对这25个数字进行排序，android用的是堆排序。（我们在系统上如果用多指，一般最多也是双值，也就是4个数据，这里采用了堆排序，不知是出于什么情况考虑，感觉换个方法可能更实用些。）下面的任务就是判断哪些当前点与前一次的点最近，那么赋予它们相同的id，应用收到这个信息后，就可以知道当前点属于哪条线了。

#### 手抬起来的时候又用什么样的序列来通知系统

SYN\_MT\_REPORT

SYN\_REPORT

只有SYNC，没有其它任何信息，系统就会认为此次事件为UP。

### B协议实现方式

B协议使用了slot，还有一个新面孔TRACKING\_ID.

手指按下的动作：

ABS\_MT\_SLOT 0

ABS\_MT\_TRACKING\_ID \*\*

ABS\_MT\_POSITION\_X x[0]

ABS\_MT\_POSITION\_Y y[0]

ABS\_MT\_SLOT 1

ABS\_MT\_TRACKING\_ID \*\*

ABS\_MT\_POSITION\_X x[1]

ABS\_MT\_POSITION\_Y y[1]

SYN\_REPORT

没有SYN\_MT\_REPORT，那么它用什么来跟踪当前点属于哪一条线呢，用的就是ABS\_MT\_TRACKING\_ID，当前序列中某点的ID值，如果与前一次序列中某点的ID值相等，那么他们就属于同一条线。既然如此，那么android系统中还需要做排序等运算吗？当然不需要。

#### 手指全部抬起的时候序列又是怎样的

ABS\_MT\_SLOT 0

ABS\_MT\_TRACKING\_ID -1

SYN\_REPORT

ABS\_MT\_SLOT 1

ABS\_MT\_TRACKING\_ID -1

SYN\_REPORT

这里上报的ABS\_MT\_TRACKING\_ID为-1，也只有这里该值才可以小于零，收到该值，系统就会清除对应的ID。看似简单的两个协议内容到这里就分析完毕了。

#### 附上android 4.0上报方式（linux必须2.6.38以上）

1. #include <linux/input/mt.h>

1. //down
2. input\_mt\_slot(ts->input\_dev, id);
3. //input\_report\_abs(ts->input\_dev, ABS\_MT\_TRACKING\_ID, id);
4. input\_mt\_report\_slot\_state(data->input\_dev, MT\_TOOL\_FINGER, true);
5. input\_report\_abs(ts->input\_dev, ABS\_MT\_TOUCH\_MAJOR, w);
6. input\_report\_abs(ts->input\_dev, ABS\_MT\_POSITION\_X, x);
7. input\_report\_abs(ts->input\_dev, ABS\_MT\_POSITION\_Y, y);

ps：个人感觉注释的那一行是在初次手指摁下才会执行，移动的时候就不执行这句话了  
//up

1. input\_mt\_slot(ts->input\_dev, id);
2. //input\_report\_abs(ts->input\_dev, ABS\_MT\_TRACKING\_ID, -1);
3. input\_mt\_report\_slot\_state(ts->input\_dev, MT\_TOOL\_FINGER, false);

ps：同理，注释的这句应该是在slot为0的时候不执行

1. //init
2. //\_\_set\_bit(INPUT\_PROP\_DIRECT, ts->input\_dev->propbit);
3. input\_mt\_init\_slots(ts->input\_dev, 255);

**上面两条ps是在本人看了内核文档介绍多触点以后的感觉，不一定对**

### 总结

看了上面的分析，明显可以看出B协议要由于A协议，但事实上并不如此简单。B协议需要硬件上的支持，ID值并不是随便赋值的，而是硬件上跟踪了点的轨迹；如果硬件上满足不了这个条件，那么采用B协议只能闹成笑话。另外，B协议的复杂性如果掌握不好往往会带来一些莫名其妙的问题，比如如果因为某些因素（同步等），在UP的时候少清除了一个slot的信息，那么下次单击的时候你也会惊奇地发现竟然有两个点（采用了B协议，slot已经保存了点信息，除非明确清除）。

## 参考资料

http://blog.csdn.net/droidphone/article/details/8434768 多点触控的input TP

http://blog.csdn.net/yaozhenguo2006/article/details/6775751 input\_event 函数

http://blog.csdn.net/loongembedded/article/details/51166888

http://blog.csdn.net/loongembedded/article/details/51167111 input事件类型

http://blog.csdn.net/tianruxishui/article/details/7173045 写得不错，

https://www.kernel.org/doc/Documentation/input/multi-touch-protocol.txt 源码的文档

<http://www.arm9home.net/read.php?tid=24754> 对源码文档的翻译

http://blog.chinaunix.net/uid-20776117-id-3212095.html 内部函数剖析

<http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=29151914&id=3921536>

<http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=29151914&id=3887032>

ps：其实源码的说明文档介绍的很清楚，但是不细节，所以需要在了解一定的基础上去看看说明文档

# Android调试方法

## getevent，sendevent，input三命令

### Getevent

**Usage**: getevent [-t] [-n] [-sswitchmask] [-**S**] [-v [mask]] [-d] [-p] [-i] [-l] [-q] [-c count] [-r] [device]

-t: show time stamps

-n: don't print newlines

-s: print **switch** states **for** given bits

-**S**: print all **switch** states

-v: verbosity mask (errs=1， dev=2， name=4， info=8， vers=16， pos. events=32，props=64)

-d: show **HID** descriptor， **if** available

-p: show possible events (errs， dev， name， pos. events)

-i: show all device info and possible events

-l: label event types and names **in** plain text

-q: quiet (clear verbosity mask)

-c: print given number of events then exit

-r: print rate events are received

键入getevent后，我们能够看到设备中的一些列输入硬件驱动信息，同样下面会出现很多输入指令信号，通常情况下，这些信号量都在刷屏

#### 格式说明

root@android:/ # getevent

/dev/input/event5: 0005 0002 00000001

device的名字：事件类型 键码类别 具体的数值

/dev/input/event5: 0000 0000 00000000

表示一次输入结束

其实这些Lable已经在其input.h头文件中定义好，其中type的定义如下：

/\*

\* 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)**

一般来说，常用的是EV\_KEY、EV\_REL、EV\_ABS、EV\_SYN，分别对应键盘按键、相对坐标、绝对坐标、同步事件。EV\_SYN则表示一组完整事件已经完成，需要处理，EV\_SYN的code定义事件分发的类型。

在触摸事件上的几个常见的Label说明如下表所示：

| **标签名** | **说明** |
| --- | --- |
| ABS\_X | 对应触摸屏的X坐标 |
| ABS\_Y | 对应触摸屏的Y坐标 |
| ABS\_PRESSURE | 压力值，一般触摸屏也只是区分是否有按下去，按下去的话值会大于多少，没有按的话值小于多少。 |
| ABS\_TOOL\_WIDTH | 触摸工具的宽度 |
| ABS\_MT\_POSITION\_X | 接触面的形心的X坐标值 |
| ABS\_MT\_POSITION\_Y | 接触面的形心的Y坐标值 |
| ABS\_MT\_TOUCH\_MAJOR | 触摸手指大小 |
| ABS\_MT\_WIDTH\_MAJOR | 触摸面积大小 |

了解了这些Label的含义我们再看看信号量就简单多了，如我们列举几个常见的事件与信号，如下表所示：

| **操作** | **输出信号** |
| --- | --- |
| 按下电源键 | /dev/input/event0: EV\_KEY KEY\_POWER DOWN /dev/input/event0: EV\_SYN SYN\_REPORT 000000 /dev/input/event0: EV\_KEY KEY\_POWER UP /dev/input/event0: EV\_SYN SYN\_REPORT 000000 |
| 音量键下 | /dev/input/event8: EV\_KEY KEY\_VOLUMEDOWN DOWN /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 /dev/input/event8: EV\_KEY KEY\_VOLUMEDOWN UP /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 |
| 音量键上 | /dev/input/event8: EV\_KEY KEY\_VOLUMEUP DOWN /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 /dev/input/event8: EV\_KEY KEY\_VOLUMEUP UP /dev/input/event8: EV\_SYN SYN\_REPORT 00000000 |
| 按下物理按键“1” | /dev/input/event0: EV\_KEY KEY\_1 DOWN /dev/input/event0: EV\_KEY KEY\_1 UP |
| 按下物理按键“q” | /dev/input/event0: EV\_KEY KEY\_Q DOWN /dev/input/event0: EV\_KEY KEY\_Q UP |
| 按下软键盘上的“q”字母 | /dev/input/event0: EV\_ABS ABS\_X 0000001b /dev/input/event0: EV\_ABS ABS\_Y 000001d5 /dev/input/event0: EV\_KEY BTN\_TOUCH DOWN /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 /dev/input/event0: EV\_KEY BTN\_TOUCH UP /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 |
| 按下软件键盘上的的“1”按键 | /dev/input/event0: EV\_ABS ABS\_X 00000019 /dev/input/event0: EV\_ABS ABS\_Y 000001d7 /dev/input/event0: EV\_KEY BTN\_TOUCH DOWN /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 /dev/input/event0: EV\_KEY BTN\_TOUCH UP /dev/input/event0: EV\_SYN SYN\_REPORT 00000000 |

从上表中，我们发现要是按下的是物理按键，其输入出来的信息我们很容易读懂，如果按下的是软键盘中的按键，给出的信号信息就是一些位置坐标信息。我们无法直接读懂，当然，我们可以根据这些位置坐标信息，再拿到Android设备的屏幕尺寸，计算比例也能够直接获得按键的具体内容。

当然，输出条件不会是想我们表格中的这么规范，中间会夹杂则各式各样的信息，有些可能是你不关心的。这里我们把一些无关的信号量过滤去掉了。实际查看上对应信息条件比较多，大家可以将Android设备连接如自己的电脑进行调试，这里我们就不做一一的解释了。

Ps:以上结果仅供参考。实际消息会有一些偏差

### Sendevent

### Input

**Usage: input [<source>] <command> [<arg>...]**

**The sources are:**

**mouse**

**keyboard**

**joystick**

**touchnavigation**

**touchpad**

**trackball**

**stylus**

**dpad**

**touchscreen**

**gamepad**

**The commands and default sources are:**

**text <string> (Default: touchscreen)**

**keyevent [--longpress] <key code number or name> ... (Default: keyboard)**

**tap <x> <y> (Default: touchscreen)**

**swipe <x1> <y1> <x2> <y2> [duration(ms)] (Default: touchscreen)**

**press (Default: trackball)**

**roll <dx> <dy> (Default: trackball)**

#### 基本用法

input keyevent [key code or name]//用来对应按键的触发

Keycode需要参考:$ROOT\_DIR/frameworks/base/core/java/android/view/KEYEVENT.java

Input tap x y //对应触摸点

Input swipe x1 y1 x2 y2 //对应滑动触摸

Input text “text” //输入text内容

ps 三个有用的调试函数

﻿﻿

#define hardirq\_count() (preempt\_count() & HARDIRQ\_MASK)  
#define softirq\_count() (preempt\_count() & SOFTIRQ\_MASK)  
#define irq\_count() (preempt\_count() & (HARDIRQ\_MASK | SOFTIRQ\_MASK \  
     | NMI\_MASK))

/\*  
 \* Are we doing bottom half or hardware interrupt processing?  
 \* Are we in a softirq context? Interrupt context?  
 \*/  
#define in\_irq()   (hardirq\_count())      //判断当前是否在硬件中断上下文  
#define in\_softirq()  (softirq\_count())  //判断当前是否在软件中断上下文  
#define in\_interrupt()  (irq\_count())    //判断当前是否在硬件、软件、底半部中断上下文

## 参考资料

<http://blog.csdn.net/shift_wwx/article/details/49760735> **input**

<http://blog.csdn.net/hudan2714/article/details/8003585> **input**

http://blog.csdn.net/shift\_wwx/article/details/11481797 **input**

<http://myeyeofjava.iteye.com/blog/1999615> **getevent**

<http://www.codeceo.com/article/android-keyboard-monitor.html> **getevent**

# ARM概念梳理：Architecture, Core, CPU，SOC

## 概念梳理

1）ARM architecture

ARM architecture，是指ARM公司开发的、基于精简指令集架构（RISC, Reduced Instruction Set Computing architecture）的指令集架构（Instruction set architecture）。我们常说的ARMv7、ARMv8、ARMv8-A，就是指ARM architecture。类似的基于RISC的architecture也有很多，例如MIPS、AVR、Blackfin等等，都是这个概念。

2）ARM core

ARM core是基于ARM architecture开发出来的IP core，它是介于architecture和最终的CPU(MCU)之间的中间产品，这也是ARM商业模式的独特之处。

有两种类型的ARM core：一种是ARM公司自己发布的，如我们耳熟能详的ARM7、ARM9、ARM Cortex M3、ARM Cortex A57等等；另一种是ARM授权其它公司开发的Core，如苹果的A6/A6X等等。下面链接是维基百科上的ARM core的列表，共大家参考：

<http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures>

3）ARM CPU（MCU）

其它的芯片厂商，如Phillips、ST、TI等，会基于ARM公司发布的Core，开发自己的ARM处理器，这称作ARM CPU（也可称为MCU）。这些是我们工作过程中接触最多的，如LPCxxxx、STM32xxx、OMAPxxxx、S3Cxxxx等等。

4）ARM Soc

对于一些比较专业的应用场景，如视频、音频等，为了追求更小的size、更低的功耗，厂商会在芯片上，集成除处理器之外的东西，如视频编解码器、DSP等。这些集成了其它功能的芯片，称作片上系统（SOC），如TI的DM37x Video SOC。

注1：其实ARM的技术和商业模式，正体现了软件工程中抽象和封装的思想。

## ARM 64bit

我们以一款64bit ARM CPU为例，反向阐述一下ARM处理的诞生过程，同时罗列一些学习、研究方向。

1）我们熟悉一个CPU（假设它的型号是WW9000）的第一手资料，是芯片厂家发布的Datasheet，例如WW9000\_SPEC.pdf。

2）WW9000是基于ARM Cortex-A57 Core封装而来的，该ARM core的资料可以从下面链接下载

<http://infocenter.arm.com/help/topic/com.arm.doc.ddi0488g/DDI0488G_cortex_a57_mpcore_trm.pdf>

3）ARM Cortex-A57 Core又是基于ARMv8-A architecture，该结构的资料可以通过如下方式获取：

Go to [ARM Infocenter](http://infocenter.arm.com/) and navigate through ARM architecture / Reference Manuals

注2：[ARM Infocenter](http://infocenter.arm.com/)中资料是非常全面的，没事时可以多逛逛。

## 参考资料

<http://www.wowotech.net/armv8a_arch/arm_concept.html>

## pmic与pmu的区别

pmu是cpu这边的，pmic是外设；pmic是供电给主芯片的，pmu负责把这些电分配给cpu的各个单元，也负责问pmic要合适的电；pmu是cpu内部负责电源管理的单元

ps:个人猜测pmic对应很多单元

# Linux中SD/MMC设备驱动流程

SD（Secure Digital）与 MMC（Multimedia Card）

SD 是一种 flash memory card 的标准，也就是一般常见的 SD 记忆卡，而 MMC 则是较早的一种记忆卡标准，目前已经被 SD 标准所取代。在维基百科上有相当详细的 SD/MMC 规格说明：

## SDIO（Secure Digital I/O）

SDIO 是目前我们比较关心的技术，SDIO 故名思义，就是 SD 的 I/O 接口（interface）的意思，不过这样解释可能还有点抽像。更具体的说明，SD 本来是记忆卡的标准，但是现在也可以把 SD 拿来插上一些外围接口使用，这样的技术便是 SDIO。

所以 SDIO 本身是一种相当单纯的技术，透过 SD 的 I/O 接脚来连接外部外围，并且透过 SD 上的 I/O 数据接位与这些外围传输数据，而且 SD 协会会员也推出很完整的 SDIO stack 驱动程序，使得 SDIO 外围（我们称为 SDIO 卡）的开发与应用变得相当热门。 现在已经有非常多的手机或是手持装置都支持 SDIO 的功能（SD 标准原本就是针对 mobile device 而制定），而且许多 SDIO 外围也都被开发出来，让手机外接外围更加容易，并且开发上更有弹性（不需要内建外围）。目前常见的 SDIO 外围（SDIO 卡）有：

 Wi-Fi card（无线网络卡）

 CMOS sensor card（照相模块）

   GPS card

 GSM/GPRS modem card

 Bluetooth card

 Radio/TV card（很好玩）

SDIO 的应用将是未来嵌入式系统最重要的接口技术之一，并且也会取代目前 GPIO 式的 SPI 接口。

## **SD/SDIO/MMC/SPI之间的关系**

SD/SDIO/MMC是三种不同的卡，这3种卡的管脚很类似，卡的样子也类似。最开始是先由MMC 卡，然后在MMC的基础上做了修改有了SD卡，SD卡的基础上有了SDIO卡。所以SD/SDIO/MMC的协 议大同小异，只有少量差异。 SPI在SD的大军里，只是一种传输模式。若是慢速的情况下，可选择SPI传输模式，这样SD接口 就变成SPI协议了。Host也要根据SPI的协议对卡进行操作。 具体的区别需参考各协议。

## **同一个控制器，支持SD/MMC/SDIO的所以卡，如何区分出现在卡槽里的是个什么卡**

这个需要软件来区分。软件不停的循环访问卡，对卡按照一定的顺序发出对应格式的命令，那么不同的卡会做出不同的反应。几个CMD发完，根据不用的反应就可以分出是什么卡了。比如已经判断出是SD卡了，但是不知道是什么类型的SD卡。那么我们就发一个CMD8，如果 没有response，那么这个卡就是SD1.X的卡；如果有response，那么就是SD2.0以上的卡。

## SD/SDIO 的传输模式

SD 传输模式有以下 3 种：

 SPI mode（required）

 1-bit mode

 4-bit mode

SDIO 同样也支持以上 3 种传输模式。依据 SD 标准，所有的 SD（记忆卡）与 SDIO（外围）都必须支持 SPI mode，因此 SPI mode 是「required」。此外，早期的 MMC 卡（使用 SPI 传输）也能接到 SD 插糟（SD slot），并且使用 SPI mode 或 1-bit mode 来读取。 SD 的 MMC Mode SD 也能读取 MMC 内存，虽然 MMC 标准上提到，MMC 内存不见得要支持 SPI mode（但是一定要支持 1-bit mode），但是市面上能看到的 MMC 卡其实都有支持 SPI mode。因此，我们可以把 SD 设定成 SPI mode 的传输方式来读取 MMC 记忆卡。 SD 的 MMC Mode 就是用来读取 MMC 卡的一种传输模式。不过，SD 的 MMC Mode 虽然也是使用 SPI mode，但其物理特性仍是有差异的：  MMC 的 SPI mode 最大传输速率为 20 Mbit/s；  SD 的 SPI mode 最大传输速率为 25 Mbit/s。 为避免混淆，有时也用 SPI/MMC mode 与 SPI/SD mode 的写法来做清楚区别



## 参考资料

http://blog.csdn.net/evilcode/article/details/7408921

http://blog.csdn.net/evilcode/article/details/7418323

<http://wenku.baidu.com/link?url=lPg26KHW_IUD6de7wxooapFMfzJNXAhuyRabcnj0WpmlT23LpGG5KRwEXIrUFqe8UfIqL1_i3CMUIZYxrNNsOmz1Dsvv_7-foXWH0ZwZIAa>

# 内核通知链

大多数内核子系统都是相互独立的，因此某个子系统可能对其它子系统产生的事件感兴趣。为了满足这个需求，也即是让某个子系统在发生某个事件时通知其它的子系统，Linux内核提供了通知链的机制。通知链表只能够在内核的子系统之间使用，而不能够在内核与用户空间之间进行事件的通知。通知链表是一个函数链表，链表上的每一个节点都注册了一个函数。当某个事情发生时，链表上所有节点对应的函数就会被执行。所以对于通知链表来说有一个通知方与一个接收方。在通知这个事件时所运行的函数由被通知方决定，实际上也即是被通知方注册了某个函数，在发生某个事件时这些函数就得到执行。通知链技术可以概括为：事件的接收者将事件发生时应该执行的操作通过函数指针方式保存在链表中，然后当事件发生时通知者依次执行链表中每一个元素的回调函数。

## 参考资料

<http://blog.csdn.net/g_salamander/article/details/8081724>

http://blog.csdn.net/wuhzossibility/article/details/8079025